610 Exercises on Data Structures and Algorithms

Version 1.0 (September 1, 2026)

Alexandros V. Gerbessiotis

CS Department
NJIT
Newark, NJ 07102.

Email: alexg@njit.edu

Printed on September 1, 2026

DRAFT. Copyright (c) 2021-2026. Alex. Gerbessiotis.
All rights reserved. Not to be posted online
or on the web or to be made available outside of
copyright holder's web-page

Preface

This e-book contains a collection of exercises with indicative solutions to a variety of topics on elementary algorithms and their data structures. It is intended primarily for mid-level to upper-level undergraduate students and entry level graduate students at the master's level. Some topics are related to electives courses at the upper undergraduate level.

This material is neither final nor thoroughly proofread. It constitutes work in progress and might contain errors. It should be used in conjunction with other references if consulted for factual checking. Report discrepancies with other sources, or factual errors, or typos to the author. Distribution of this material in any form, without the expressly written consent of the author is prohibited.

©2021-2026. Alexandros Gerbessiotis. All rights reserved.

Contents

I Programming and mathematics background **1**

1 Math, iteration, recursion **3**
1.1 Bit for numbers . . . 3
1.2 Limit math review . . . 5
1.3 Sums of sequences . . . 7
1.4 Equalities and inequalities . . . 25
1.5 Entropy and probability . . . 33
1.6 Recursion . . . 37
1.7 Trace the code . . . 40
1.8 Linear and binary search review . . . 54
1.9 Tree traversals . . . 57
1.10 Graph theory review . . . 62
1.11 Elementary school arithmetic and merging . . . 66
1.12 Sorting, merging, or binary search review . . . 72

2 Asymptotic notation **83**
2.1 Compare vs asymptotically compare . . . 83
2.2 Little, big, or Θ . . . 86
2.3 Asymptotically rank functions . . . 93
2.4 The math of o, ω, and Θ . . . 107

3 Recurrrence relations **123**
3.1 Recurrences and the master method . . . 123
3.2 Recurrences and the iteration method . . . 131
3.3 If all else fails: subsitution method . . . 150
3.4 Recurrences and algorithm analysis: warming up . . . 167

II Searching, sorting and selection **169**

4 Elementary data structures and graph representation **171**
4.1 Explain the code . . . 171
4.2 Array and linked list structures: performance and more . . . 174
4.3 Properties of elementary sorting algorithms . . . 182
4.4 Arithmetic, Lionardo Bonacci, and filius Bonacci . . . 190
4.5 Combine two algorithm design techniques or methods . . . 195
4.6 More on other elementary topics . . . 198
4.7 Elementary arithmetic time complexity . . . 210
4.8 Graph representation and their complexity . . . 211
4.9 Challenging problems . . . 216

5 More sorting, heaps, and applications **227**
5.1 Heap math . . . 227
5.2 Heap properties . . . 228
5.3 Heap operations . . . 233
5.4 HeapSort vs Heap operations . . . 238
5.5 Extend a heap data structure . . . 239
5.6 Past heap . . . 244
5.7 An application of heaps: Huffman's data compression . . . 248
5.8 Distribution-based sorting: CountSort and RadixSort and friends . . . 251
5.9 Fake coins . . . 257
5.10 More on sorting . . . 259
5.11 Some challenging problems . . . 260
5.12 Decision trees . . . 310

6 Hashing **325**
6.1 Hash tables and their properties . . . 325
6.2 Hashing examples . . . 332
6.3 Some challenging problems . . . 335

7 Selection and complexity of sorting **341**
7.1 Selection algorithm properties . . . 341
7.2 Selection algorithm applications . . . 343
7.3 Some challenging problems . . . 356
7.4 Review search, sort selection . . . 365

8 More on graph searches **375**
8.1 Graphs and their representations . . . 375
8.2 Elementary graph algorithm properties . . . 377
8.3 Graph algorithm applications . . . 379
8.4 Dijkstra's algorithm . . . 392
8.5 Floyd-Warshall's algorithm . . . 395
8.6 Spanning tree and extremes of . . . 398
8.7 Triangles . . . 406
8.8 Miscellanea review . . . 409

III Numbers, polynomials, and matrices **417**

9 Arithmetic problems **419**
9.1 Integers . . . 419
9.2 Polynomials . . . 425
9.3 Matrices . . . 427
9.4 Miscellanea . . . 431

IV Trees of bounded height and depth **433**

10 Binary search trees: bounded height or not **435**
10.1 Binary search trees: operations and properties . . . 435
10.2 Balanced binary search trees: operations and properties . . . 441
10.3 Miscellanea . . . 448

11 Non-binary search trees: bounded depth **465**
11.1 2-3-4 trees aka B-trees with $B = 4$ . . . 465

V Strings 469

12 String matching **471**
12.1 Naive, Boyer-Moore, KMP . . . 471
12.2 Rabin-Karp's fingerprinting method . . . 482
12.3 Miscellanea . . . 485

VI Advanced topics 489

13 More arithmetic problems **491**
13.1 Integers or not . . . 491
13.2 Polynomial . . . 510
13.3 Matrices . . . 527
13.4 Some challenging problems . . . 538

14 Parallel algorithms **547**
14.1 Fundamentals . . . 547
14.2 PRAMs . . . 554
14.3 Network algorithms and more . . . 581
14.4 Uni vs Multi . . . 594
14.5 Slack greater than one . . . 596
14.6 Challenging problems . . . 599
14.7 BSP . . . 624

15 Sorting networks **627**
15.1 Properties . . . 627
15.2 Sorting networks . . . 629
15.3 Bitonic sorting . . . 636

VII Probability and randomness 645

16 Probability review problems **647**
16.1 Balls and bins . . . 647
16.2 Binomial properties . . . 659
16.3 Binomial properties and Chernoff/Hoeffding bounds . . . 665
16.4 Miscellanea . . . 667
16.5 Randomness . . . 680

VIII Brief math review 691

A Notation **693**
A.1 Notation . . . 693
A.2 Boolean function . . . 696
A.3 Axioms, Propositions, Theorems . . . 697

B Functions **699**
B.1 Exponentials and logarithms . . . 699
B.2 Logarithms base two (and e, and 10) . . . 700
B.2.1 Other functions . . . 701
B.2.2 Inequalities . . . 702
B.3 The factorial and Stirling's approximation formula . . . 703
B.4 Limits, Derivatives and Integrals . . . 704
B.5 Minimum and Maximum of a function: f(n) . . . 705

C Collections : sets and sequences **707**
C.1 Collections . . . 707
C.1.1 Sets . . . 707
C.1.2 Sequences . . . 709
C.1.3 Sums of sequences: finite and infinite . . . 710

D Comparison of functions **717**
D.1 Comparison of (discrete or continuous) functions . . . 717

E Mathematical induction **721**
E.1 Mathematical induction . . . 721

F Asymptotic notation **727**
F.1 Asymptotic Comparison: informal and error-prone approach . . . 727
F.2 Asymptotic Comparison: formal definitions . . . 728
F.3 Asymptotic notation . . . 729

G Divisibility **733**
G.1 Divisibility and Division . . . 733
G.2 Greatest Common Divisor . . . 737
G.3 Least Common Multiplier . . . 743

H String Matching **745**
H.1 String Matching . . . 745
H.1.1 Naive approach . . . 745
H.1.2 Rabin-Karp . . . 745
H.1.3 Boyer-Moore . . . 748
H.1.4 Knuth-Morris-Pratt . . . 749
H.1.5 Approximate matching . . . 751

IX Brief probability theory review **753**

I Probability theory **755**
I.1 Experiments and random processes . . . 755
I.1.1 Event spaces and measurable spaces . . . 756
I.1.2 Probability spaces . . . 757
I.1.3 Probability of events . . . 758
I.1.4 Conditional probabilities . . . 759
I.1.5 Random variables . . . 761
I.1.6 Expectation of a random variable . . . 764
I.1.7 Independent random variables . . . 765
I.1.8 Indicator random variable . . . 766

I.1.9 More on expectation . . . 767
I.1.10 Variance of random variables . . . 767
I.1.11 Standard distributions . . . 768
I.1.12 Moment generating function . . . 770
I.1.13 Probability inequalities . . . 771
I.1.14 Example . . . 773
I.2 Chernoff's method . . . 774
I.2.1 Derived right tails . . . 775
I.2.2 Derived left tails . . . 777
I.2.3 Derived concentration bounds . . . 778
I.2.4 Hoeffding's method . . . 778
I.2.5 Derived right tails . . . 779
I.2.6 Derived left tails . . . 781
I.2.7 Derived concentration bounds . . . 782
I.3 Identities and Inequalities . . . 783
I.3.1 Series . . . 784
I.3.2 Exponential inequalities . . . 785
I.3.3 Logarithmic and other inequalities . . . 786
I.3.4 Combinatorial equalities . . . 789

Part I

Programming and mathematics background

Chapter 1

Math, iteration, recursion

1.1 Bit for numbers

Exercise 1.
(a) How many non-negative (consecutive) integers can we represent with 4 bits? How many with n bits? How many with $\lg n$ bits? ($\lg n$ is the base two logarithm of n.)
(b) How many bits do we need to represent 7? How about 8? How about 15?
(c) How many bits do we need to represent $2^n - 1$? How many for 2^n?
(d) Assume n is a power of two for this question. How many bits do we need to represent all unsigned integers between 0 and $n-1$ in fixed width ? between 0 and $n^5 - 1$? (Such integers that will have fewer bit than the maximum $n^5 - 1$ will get leading zeroes.) Between 0 and $2^n - 1$? Between 0 and $n^n - 1$?

Proof.
(a) With 4bit we can represent $2^4 = 16$ integers. With n bit, 2^n. With $\lg n$, we can $2^{\lg n} = n$.
(b) 7=0b111 i.e. 3 bits. 8=0b1000 i.e. 4 bits. 15=0b1111 i.e. 4 bits. The prefix $0b$ informs that the rest of the sequence is a 0-1 sequence representing a binary (integer, usually unsigned) number.
(c) $2^n - 1$ can be represented by n bits all of which are 1. 2^n can be represented by $n+1$ bits, an 1 followed by n 0s.
(d) All integers between 0 and $n-1$ can be represented minimally with $\lg(n-1+1) = \lg n$ bits by part (c) above. All integers between 0 and $n^5 - 1$ can be represented with $\lg(n^5 - 1 + 1) = \lg n^5 = 5 \lg n$ bits. All integers between 0 and $2^n - 1$ can be represented with $\lg(2^n - 1 + 1) = \lg 2^n = n$ bits. All integers between 0 and $n^n - 1$ can be represented with $\lg(n^n - 1 + 1) = \lg n^n = n \lg n$ bits. ■

Exercise 2.

(a) How many bit do we need to minimally represent $A = 2^{m-1}$? How many for $B = 2^m - 1$? Minimally means without leading zeroes. How many bit do we need to represent all unsigned integers between 2^{m-1} and $2^m - 1$? Express the answer as a function of m.

(b) Use the result from (a) to show that every positive integer L can be represented using $\lceil \lg(L+1) \rceil$ bit.

Proof.
(a) $A = 2^{m-1}$ is an 1 followed by $m-1$ zeroes. $B = 2^m - 1$ is a sequence of m 1's.
So both integers need m bit to be represented. If the extremes $A = 2^{m-1}$ and $B = 2^m - 1$ need m bit, so do all integers L between A and B such that $A \leq L \leq B$.

(b) Every positive integer L can be sandwitched between an A, and a B of part (a), for some integer value m. This means m bit are needed to represent (minimally) L. Thus

$$\begin{aligned} 2^{m-1} &\leq L \leq 2^m - 1 \\ 2^{m-1} + 1 &\leq L + 1 \leq 2^m \\ 2^{m-1} &< L + 1 \leq 2^m \\ \lg(2^{m-1}) &< \lg(L+1) \leq \lg(2^m) \\ m - 1 &< \lg(L+1) \leq m \end{aligned}$$

Bear in mind that $\lg(L+1) \leq \lceil \lg(L+1) \rceil$. The term $\lg(L+1)$ is strictly greater than $m-1$ and at most m. Its ceiling $\lceil \lg(L+1) \rceil$, which is an integer, can not thus be $m-1$, because the smaller $\lg(L+1)$ is greater than $m-1$, and therefore $\lceil \lg(L+1) \rceil$ must be the next greater integer, i.e. m. Therefore, $m = \lceil \lg(L+1) \rceil$. One can use the same method, symmetrically on the right inequality, to also show that $m = \lfloor \lg L \rfloor + 1$.

$$\begin{aligned} 2^{m-1} &\leq L \leq 2^m - 1 \\ 2^{m-1} &\leq L < 2^m \\ m - 1 &\leq \lg L < m \\ m - 1 &\leq \lfloor \lg L \rfloor \leq \lg L < m \end{aligned}$$

Therefore, $\lfloor \lg L \rfloor = m - 1$ (as it cannot be m), and solving for m we find that $m = \lfloor \lg L \rfloor + 1 = \lceil \lg(L+1) \rceil$. ■

1.2 Limit math review

Exercise 3.

Find the following limits.

(i)

$$\lim_{n\to\infty} n^2/2^n,$$

(ii)

$$\lim_{n\to\infty} 2^{\lg n}/n,$$

(iii)

$$\lim_{n\to\infty} n!/2^n.$$

Note $n! = 1 \cdot 2 \cdot \ldots \cdot n$.

Proof.
$\lim_{n\to\infty} \frac{n^2}{2^n} = 0$, after two applications of L'Hopital's theorem.
$\lim_{n\to\infty} 2^{\lg n}/n = \lim_{n\to\infty} n/n = 1$.
Note that from Stirling's approximation formula, $n! \approx (n/e)^n$.
$\lim_{n\to\infty} n!/2^n = \lim_{n\to\infty} (n/2)\cdots(n-1)/2\ldots(3/2)\cdot(2/2)\cdot(1/2) = \infty$, or
$\lim_{n\to\infty} n!/2^n = \lim_{n\to\infty} (n/e)^n/2^n = \lim_{n\to\infty} (n/(2e))^n = \infty$ ■

Exercise 4.
Find the following limit.

$$\lim_{n\to\infty} \lg(n!)/n\lg n,$$

Proof.
Note that from Stirling's approximation formula, $n! \approx (n/e)^n$, we have

$$\lim_{n\to\infty} \frac{\lg(n!)}{n\lg n} = \lim_{n\to\infty} \frac{\lg(n/e)^n}{n\lg n} = \lim_{n\to\infty} \frac{n\lg n - n\lg e}{n\lg n} = 1.$$

■

1.3 Sums of sequences

Exercise 5.

Show that a closed form write-up of the sum

$$\frac{1}{2}+\frac{1}{2^2}+\ldots+\frac{1}{2^{\lg n-1}}$$

is $1-2/n$. This explains what **closed form** is.

Proof. Let $X=\frac{1}{2}+\frac{1}{2^2}+\ldots+\frac{1}{2^{\lg n-1}}$. We use the fact that $1+x+\ldots+x^{m-1}=(x^m-1)/(x-1)$ for $x=1/2$ and $m=\lg n$.

$$\begin{aligned} 1+X &= 1+\frac{1}{2}+\frac{1}{2^2}+\ldots+\frac{1}{2^{\lg n-1}} \\ &= \frac{1/2^{\lg n}-1}{1/2-1} \\ &= \frac{1/n-1}{-1/2} \\ &= 2-2/n \end{aligned}$$

Since $1+X=2-2/n$ then $X=1-2/n$ and the result has been proved. ■

Exercise 6.

Calculate the following sum for any $x \neq 1$

$$x + 2x^2 + 3x^3 + \ldots + nx^n = \sum_{i=1}^{n} ix^i.$$

and show it is equal to

$$\frac{nx^{n+2} - (n+1)x^{n+1} + x}{(x-1)^2}.$$

Proof. By taking the first derivative with respect to x of $1 + x + x^2 + \ldots + x^n$ we obtain the derivations shown below.

$$\begin{aligned}
1 + x + x^2 + x^3 + \ldots + x^n &= \frac{x^{n+1} - 1}{x - 1} \\
&\quad (\textit{take first derivative of both sides}) \\
0 + 1 \cdot x^0 + 2 \cdot x^1 + 3 \cdot x^2 + \ldots + n \cdot x^{n-1} &= \frac{(n+1)x^n(x-1) - x^{n+1} + 1}{(x-1)^2} \\
0 + 1 \cdot x^0 + 2 \cdot x^1 + 3 \cdot x^2 + \ldots + n \cdot x^{n-1} &= \frac{nx^{n+1} - (n+1)x^n + 1}{(x-1)^2} \\
&\quad (\textit{mulitiply by x both sides}) \\
0 + 1 \cdot x^1 + 2 \cdot x^2 + 3 \cdot x^3 + \ldots + n \cdot x^n &= \frac{nx^{n+2} - (n+1)x^{n+1} + x}{(x-1)^2} \\
x + 2x^2 + 3x^3 + \ldots + nx^n &= \frac{nx^{n+2} - (n+1)x^{n+1} + x}{(x-1)^2} \\
\sum_{i=1}^{n} ix^i &= \frac{nx^{n+2} - (n+1)x^{n+1} + x}{(x-1)^2}
\end{aligned}$$

Note 1. For $x < 1$ and $n \to \infty$ we have

$$x + 2x^2 + 3x^3 + \ldots + ix^i + \ldots = \sum_{i=0}^{\infty} ix^i = \frac{x}{(1-x)^2}.$$

Note 2. For $x = 2$ and m replaced by $\lg n$ we derive the following.

$$\begin{aligned}
\sum_{i=1}^{m} ix^i &= \frac{mx^{m+2} - (m+1)x^{m+1} + x}{(x-1)^2} \\
&= \frac{\lg n x^{\lg n + 2} - (\lg n + 1)x^{\lg n + 1} + x}{(x-1)^2} \\
&= \frac{\lg n 2^{\lg n + 2} - (\lg n + 1)2^{\lg n + 1} + 2}{(2-1)^2} \\
&= 4n \lg n - 2n(\lg n + 1) + 2 \\
&= 2n \lg n - 2n + 2.
\end{aligned}$$

■

Exercise 7.
Give a closed-form solution for the sum (i.e. compute the sum).

$$\sum_{i=0}^{n} \left(i^2 + i^3\right).$$

Proof. The sum

$$\sum_{i=0}^{n} \left(i^2 + i^3\right),$$

can be rewritten as

$$\sum_{i=0}^{n} \left(i^2 + i^3\right) = \sum_{i=0}^{n} i^2 + \sum_{i=0}^{n} i^3.$$

The first sum is given in page 711 for Fact C.3, and the second in page 711 for Fact C.3. $\sum_{i=0}^{n} i^3 = n^2(n+1)^2/4$. The two sums can be computed using the technique outlined in the proof of page 711. Let the first sum be denoted by S_2.
One can appreciate asymptotic analysis of the performance of an algorithm or asymptotic notation if he or she tries to solve the following question or problem. What is the smallest value of n such that an algorithm whose running time is $128n \lg n$ runs faster than a algorithm whose running time is n^2. What is the answer? Faster means smaller running time! For $n = 1024$ the former algorithm is slower than the latter. For $n = 2048$, the former is faster. If we spend a little bit more time trying values between 1024 and 2048, using binary search in the interval $[1024, 2048]$, we conclude that $n = 1329$ is the cutoff point!

$$S_2 = \sum_{i=0}^{n} i^2$$

It is known that $S_1 = \sum_{i=0}^{n} i = n(n+1)/2$. and also

$$(i+1)^3 = i^3 + 3i^2 + 3i + 1.$$

We substitute $1, 2, 3, \ldots, n$ for i in the identity and we get a set of n equations

$$\begin{aligned}
(1+1)^3 &= 1^3 + 3 \cdot 1^2 + 3 \cdot 1 + 1 \\
(2+1)^3 &= 2^3 + 3 \cdot 2^2 + 3 \cdot 2 + 1 \\
(3+1)^3 &= 3^3 + 3 \cdot 3^2 + 3 \cdot 3 + 1 \\
\ldots &= \ldots \\
(n+1)^3 &= n^3 + 3 \cdot n^2 + 3 \cdot n + 1
\end{aligned}$$

By summing up these n equalities we obtain the following.

(Note that the left hand side of the i-th equality cancels out in the grand sum with the first term of the right hand side of the $(i+1)$-st equality, i.e. $(2+1)^3$ of the second equality cancels out in the sum with 3^3 of third equality).

$$\begin{aligned}
(n+1)^3 &= 1^3 + 3(\sum_{i=1}^{n} i^2) + 3(\sum_{i=1}^{n} i) + n \\
(n+1)^3 &= 1^3 + 3(\sum_{i=0}^{n} i) + 3(\sum_{i=0}^{n} i) + n \\
(n+1)^3 &= 1^3 + 3S_2 + 3S_1 + n \\
(n+1)^3 - 1^3 - n - 3S_1 &= 3S_2 \\
(n+1)^3 - (n+1) - 3n(n+1)/2 &= 3S_2 \\
(n+1)(n^2 + 2n + 1 - 1 - 3n/2) &= 3S_2 \\
(n+1)n(2n+1)/6 &= S_2
\end{aligned}$$

Therefore,

$$S_2 = \sum_{i=0}^{n} i^2 = n(n+1)(2n+1)/6.$$

Then by adding up the expressions for S_2 and S_3 we get that

$$\sum_{i=0}^{n} \left(i^2 + i^3\right) = S_2 + S_3 = n(n+1)(2n+1)/6 + n^2(n+1)^2/4 = n(n+1)(3n^2+7n+2)/12.$$

■

Exercise 8.

Use mathematical induction to show that when n is an exact power of two, the solution of the recurrence

$$T(n) = \begin{cases} 8 & \text{if } n = 2 \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n\lg n + 3n$.

Proof. We show the claim by induction. We first form proposition $P(n)$ that is to be proved.
Proposition $P(n)$: $T(n) = 3n + n\lg n$.
Since we only need to show $P(n)$ not for all n but only for n being a power of two, we can transform $P(n)$ into a $Q(k)$ such that $Q(k) = P(2^k)$ using $n = 2^k$ and $k \geq 1$. $Q(k)$ is thus given by the following expression.

$$Q(k) \quad : \quad T(2^k) = 3 \cdot 2^k + k \cdot 2^k = (k+3) \cdot 2^k.$$

The recurrence $T(n) = 2T(n/2) + n$ can also be written in terms of k since $n = 2^k$ as

$$T(2^k) = 2T(2^{k-1}) + 2^k.$$

We show $Q(k)$ by induction on k.
Base case ($Q(1)$). For $k = 1$ and subsituting for $k = 1$ in $Q(k)$ we get that $T(2^1) = (1+3)2^1 = 8$. This is true by the way of the base case $T(2) = 8$ that is given.
Inductive step $Q(k-1) \Longrightarrow Q(k)$. $Q(k-1)$ is equivalent to $T(2^{k-1})$ i.e. $T(2^{k-1}) = (k+2)2^{k-1}$. We expect to establish the implication involving $Q(k)$ i.e. $T(2^k) = (k+3) \cdot 2^k$, from $Q(k-1)$.
We use the recurrence $T(2^k) = 2T(2^{k-1}) + 2^k$ and in it we substitute for the $T(2^{k-1})$ as obtained from $Q(k-1)$.

$$\begin{aligned} T(n) = T(2^k) &= 2 \cdot T(2^{k-1}) + 2^k \\ &\overset{Q(k-1)}{=} 2 \cdot ((k+2) \cdot 2^{k-1}) + 2^k \\ &= k \cdot 2^k + 2 \cdot 2^k + 2^k \\ &= k \cdot 2^k + 3 \cdot 2^k = (k+3) \cdot 2^k \end{aligned}$$

We have thus proved that $T(2^k) = (k+3) \cdot 2^k$ i.e. $Q(k)$ is true. This completes the induction and shows also $P(n)$. ■

Exercise 9.

How much is the sum $\sum_{i=0}^{\infty} i(i-1)x^i$, $0 < x < 1$? Explain.

Proof. We use the derivation available in the appendix on page 714 as it relates to Fact C.8, and Corollary C.1 for $0 < x < 1$.

$$\sum_{i=0}^{\infty} x^i = \frac{1}{(1-x)}$$

Taking the first derivative of both sides the following is derived.

$$\sum_{i=0}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2}$$

Multiplying with x both sides we get the following.

$$\sum_{i=0}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2}.$$

From the second derivation above, i.e. $\sum_{i=0}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2}$, we take the first derivative once more of both sides. We get

$$\sum_{i=0}^{\infty} i(i-1) \cdot x^{i-2} = \frac{2}{(1-x)^3}$$

We then multiply both sides with x^2. We get

$$\sum_{i=0}^{\infty} i(i-1) \cdot x^i = \frac{2x^2}{(1-x)^3}.$$

All of the steps require $0 < x < 1$. Result proven.

■

Exercise 10.

Use mathematical induction to show that when n is an exact power of two, the solution of the recurrence

$$T(n) = \begin{cases} 4 & \text{if } n = 1 \\ 2T(n/2) + 4n & \text{if } n = 2^k, \text{ for } k \geq 1 \end{cases}$$

is $T(n) = 4n \lg n + 4n$.

Proof.
We show the claim by induction. $P(n)$ is $T(n) = 4n + 4n \lg n$. Since we only need to show $P(n)$ not for all n but only for n being a power of two, we can transform $P(n)$ into $Q(k) = P(2^k)$ using $n = 2^k$. $Q(k)$ is thus $T(2^k) = 2^{k+2} + k2^{k+2} = (k+1)2^{k+2}$. The recurrence $T(n) = 2T(n/2) + 2n$ can also be written in terms of k as $T(2^k) = 2T(2^{k-1}) + 2^{k+2}$. We show $Q(k)$ by induction on k.
Base case. For $k = 0$, we have $n = 2^0 = 1$. We substitute 0 for k in $T(2^k)$. Thus $T(2^k) = T(1)$ which is 4 by the base case of the recurrence that was given to us. Starting from the right side of $T(2^k)$ which is $(k+1)2^{k+2}$ for $k = 0$ we get $(0+1)2^{0+2} = 4$ as well. Therefore we have just shown that $Q(0)$ is true because both sides of $T(2^k) = (k+1)2^{k+2}$ give 4 for $k = 0$.
Inductive Hypothesis. We show that if $Q(k-1)$ is true, then $Q(k)$ will also be true.
Inductive Step. If $Q(k-1)$ is true, then $T(2^{k-1}) = k2^{k+1}$. We expect to establish that $Q(k)$ is true, i.e. $T(2^k) = (k+1) \cdot 2^{k+2}$. We use the recurrence $T(2^k) = 2T(2^{k-1}) + 2^{k+2}$ to show that $Q(k)$ is true then. We start from the recurrence above and we substitute there for $T(2^{k-1})$ the expression obtained by the inductive hypothesis for $Q(k-1)$.

$$\begin{aligned} T(n) = T(2^k) &= 2T(2^{k-1}) + 2^{k+2} \\ &= 2(k2^{k+1}) + 2^{k+2} \\ &= k2^{k+2} + 2^{k+2} \\ &= (k+1) \cdot 2^{k+2} \end{aligned}$$

We have thus proved that $T(2^k) = (k+1)2^{k+2}$. We express the answer as a function of n now. Since $n = 2^k$ we have that

$$\begin{aligned} T(2^k) &= (k+1) \cdot 2^{k+2} \\ T(n) &= (\lg n + 1) \cdot n \cdot 4 \\ T(n) &= 4n \lg n + 4n \end{aligned}$$

We have thus established that $T(n) = 4n \lg n + 4n$ for all $n = 2^k$, k non-negative integer. ■

Exercise 11.

Use mathematical induction to show that when n is an exact power of two, the solution of the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n + n$.

Proof.
We show the claim by induction. The $P(n)$ of induction is $T(n) = n + n \lg n$. Since we only need to show $P(n)$ not for all n but only for n being a power of two, we can transform $P(n)$ into $Q(k) = P(2^k)$ using $n = 2^k$ and then the expression for $T(n)$ to be proved becomes $T(2^k) = 2^k + k2^k = (k+1)2^k$.
Base case. For $k = 0$, we have $n = 2^0 = n = 1$ we have that $T(1) = 1 \lg 1 + 1 = 1$. We also have by the boundary case that $T(1) = 1$. Therefore the expression for $T(n)$ is satisfied by the solution $T(n) = n + n \lg n$ for $k = 0$ (equivalently, $n = 1$).
Inductive step. Let $n = 2^k$. We assume that if $Q(k-1)$ is true then we will show that $Q(k)$ is also true. If $Q(k-1)$ is true $T(2^{k-1}) = 2^{k-1} + (k-1)2^{k-1}$.
We also have by the recurrence that $T(n) = 2T(n/2) + n$ i.e. $T(2^k) = 2T(2^{k-1}) + 2^k$. We start from the recurrence above and we substitute there for $T(2^{k-1})$ the expression obtained by the inductive hypothesis for $Q(k-1)$.

$$\begin{aligned} T(n) = T(2^k) &= 2T(2^{k-1}) + 2^k \\ &= 2(2^{k-1} + (k-1)2^{k-1}) + 2^k \\ &= 2^k + (k-1)2^k + 2^k \\ &= 2^k + k2^k \\ &= n + n \lg n \end{aligned}$$

This completes the induction as we have shown $P(k)$ (or equivalently $P(n)$) assuming that $P(k-1)$ was true (or equivalently $P(n/2)$). ■

Exercise 12.

One can appreciate asymptotic analysis of the performance of an algorithm or asymptotic notation if he or she tries to solve the following question or problem. What is the smallest value of n such that an algorithm whose running time is $128n\lg n$ runs faster than a algorithm whose running time is n^2. What is the answer?

Proof. Faster means smaller running time! For $n = 1024$ the former algorithm is slower than the latter. For $n = 2048$, the former is faster. If we spend a little bit more time trying values between 1024 and 2048, using binary search in the interval $[1024, 2048]$, we conclude that $n = 1329$ is the cutoff point! ■

Exercise 13.
Show that for any $n \geq 0$

$$\sum_{i=0}^{i=n} i^2 = n(n+1)(2n+1)/6$$

Proof. Use induction. ■

Exercise 14.
Show that for any $n \geq 2$, $\sum_{i=1}^{n} i \leq 3n^2/4$.

Proof. Use induction or may be not. ■

Exercise 15.

Show by induction that

$$\sum_{i=0}^{n} 36i^2 = 6n(n+1)(2n+1)$$

Proof.
Let us call $P(n)$ the proposition $\sum_{i=0}^{n} 36i^2 = 6n(n+1)(2n+1)$. The inductive proof follows.
Base case. We show that $P(0)$ is true. Indeed for $P(0)$ its left hand side works as $\sum_{i=0}^{0} 16i^2 = 0$ and its right hand side works as $6 \cdot 0 \cdot (0+1) \cdot (2 \cdot 0+1) = 0$. Both sides are equal thus $P(0)$ is true.
Inductive Hypothesis. We show that if $P(n)$ is true then $P(n+1)$ is true. More explicitly, we show that if $\sum_{i=0}^{n} 36i^2 = 6(n)(n+1)(2n+1)$ is true, then $\sum_{i=0}^{n+1} 36i^2 = 6(n+1)(n+2)(2n+3)$ is also true.
Inductive Step. We start with stating the left hand side of $P(n+1)$ decomposing it into the left hand side of $P(n)$ plus other terms and then substituting for the right hand side of $P(n)$ to derive eventually the right hand size of $P(n+1)$.

$$\begin{aligned} P(n) &: \sum_{i=0}^{n} 36i^2 = 6n(n+1)(2n+1) \\ P(n+1) &: \sum_{i=0}^{n+1} 36i^2 = 6(n+1)(n+2)(2n+3) \end{aligned}$$

$$\begin{aligned} \sum_{i=0}^{n+1} 36i^2 &= \sum_{i=0}^{n} 36i^2 + 36(n+1)^2 \\ &= 6n(n+1)(2n+1) + 36(n+1)(n+1) \\ &= 6(n+1)(n(2n+1) + 6(n+1)) \\ &= 6(n+1)(2n^2 + n + 6n + 6) \\ &= 6(n+1)(2n^2 + 7n + 6) \\ &= 6(n+1)(n+2)(2n+3) \end{aligned}$$

We have just proven that if $P(n)$ is true, then $P(n+1)$ will be true. This completes the induction. Therefore $P(n)$ is true for all $n \geq 0$. ■

Exercise 16.
Show by induction that

$$\sum_{i=0}^{n} i^3 = n^2(n+1)^2/4$$

Proof.
Let us call $P(n)$ the proposition $\sum_{i=0}^{n} i^3 = n^2(n+1)^2/4$.
Base case. We show that $P(0)$ is true. Indeed for $P(0)$ its left hand side works as $\sum_{i=0}^{0} i^3 = 0$ and its right hand side works as $0^2(0+1)^2/4 = 0$. Both sides are equal and equal to zero thus $P(0)$ is true.
Inductive Step. Suppose than $P(n-1)$ is true. We will then show than $P(n)$ is true, i.e. we are going to prove that if $\sum_{i=0}^{n-1} i^3 = (n-1)^2(n)^2/4$ is true, then $\sum_{i=0}^{n} i^3 = n^2(n+1)^2/4$ is also true. We start by stating the inductive hypothesis $P(n-1)$ and then by deriving $P(n)$ by manipulating the terms of the sum as follows.

$$\begin{aligned}
\sum_{i=0}^{n-1} i^3 &= (n-1)^2(n)^2/4 \\
\sum_{i=0}^{n-1} i^3 + n^3 &= (n-1)^2(n)^2/4 + n^3 \\
\sum_{i=0}^{n} i^3 &= (n-1)^2(n)^2/4 + n^3 \\
\sum_{i=0}^{n} i^3 &= (n^4 - 2n^3 + n^2)/4 + 4n^3/4 \\
\sum_{i=0}^{n} i^3 &= (n^4 + 2n^3 + n^2)/4 \\
\sum_{i=0}^{n} i^3 &= n^2(n+1)^2/4
\end{aligned}$$

Therefore starting from $P(n-1)$ and adding to both sides of the equation n^3 we have shown the validity of $P(n-1)$. This completes the proof of the inductive step and the induction.
Therefore $P(n)$ is true for all $n \geq 0$. ■

Exercise 17.
Show that for any $x \geq 3$, $\sum_{i=0}^{n-1} x^i \leq x^n/2$.

Proof. Use induction or may be not. ■

Exercise 18.
Show that for any $n \geq 1$, $\sum_{i=0}^{n} i^2 \leq (n^3 + 2n^2)/3$.

Proof. Use induction or may be not. ■

Exercise 19.

What is wrong with the proof of the theorem below? Explain.
Horsy theorem. All horses of the world are of the same color.

Proof. The proof is (supposed to be) by induction on the number of horses n. **P(n): In any set of $n \geq 1$ horses, all the horses of the set are of the same color**. **1. Base Case: Show $P(1)$ is true.** $P(1)$ is always true as in a set consisting of a single horse, all the horses (there is only one) of the set have the same color. **2. Inductive step : $\forall n \in N$, i.e. $n \geq 1$, $P(n) \Rightarrow P(n+1)$.** Let us assume (induction hypothesis) that for any $n \geq 1$, $P(n)$ is true. Since we assume $P(n)$ to be true, every set of n horses have the same color. Then we will prove that $P(n+1)$ is also true (inductive step), i.e. we will show that in every set of $n+1$ horses, all of them are of the same color. To show the inductive step, i.e. that $P(n+1)$ is true let us consider ANY set of $n+1$ horses $H_1, H_2, \ldots H_n, H_{n+1}$. The set of horses $H_1, H_2, \ldots, H_n$, consists of n horses, and by the induction hypothesis any set of n horses are of the same color. Therefore color(H_1) =color$(H_2) = \ldots$ =color(H_n). The set of horses $H_2, H_3, \ldots, H_{n+1}$, consists of n horses, and by the induction hypothesis any set of n horses are of the same color. Therefore color(H_2) =color$(H_3) = \ldots$ =color(H_{n+1}). Since from the first set of horses color(H_2) =color(H_n), and from the second set color(H_2) =color(H_{n+1}), we conclude that the color of horse H_{n+1} is that of horse H_2, and since all horses $H_1, H_2, \ldots, H_n$ are of the same color, then all horses $H_1, H_2, \ldots, H_n, H_{n+1}$ have the same color. This proves the inductive step. The induction is complete and we have thus proved that for any n, in any set of n horses all horses (in that set) are of the same color. ■

(Hint: The key to this proof is the existence of horse H_2.)

Of the erratic proof. Horses revisited i.e. why Theorem 1.3 is false. On the previous page we proved that all horses have the same color, an otherwise nonsense statement (or false proposition). What's wrong with the inductive proof?

Many may argue that the error is in the logic of the inductive step.

The logic is fine, the quantification "for all n" is not, since the assumption of always having horse H_2 might not be true. The crux of the inductive step is the existence of three 'different' horses H_1, H_2, H_{n+1}. We first form a set of n horses $H_1, H_2, \ldots, H_n$ and apply the induction hypothesis and then form another set of n horses, $H_2, \ldots H_n, H_{n+1}$, and apply the induction hypothesis again. Crucial to the proof is that $c(H_1) = c(H_2)$ from the first application of the induction hypothesis, and $c(H_2) = c(H_{n+1})$ from the second thus concluding that $c(H_1) = c(H_2) = \ldots = c(H_n)$.

Let's see what happens for $n = 1$, i.e. let's try to show that $P(1) \Rightarrow P(2)$, i.e. show the inductive step for a certain value of n equal to 1. If we try to form $H_1, \ldots H_n$ this set contains only one 'horse' element for $n = 1$: H_1 ! If we try to form $H_2, \ldots, H_{n+1}$ this set contains only one 'horse' element H_2. There is no common third 'horse' H_k in the set containing H_1 nor in the set containing H_2. This is because the argument in the previous paragraph works only for $n \geq 2$. In that case one set is formed from H_1, H_2 and the other set from H_2, H_3. However even if we can prove the inductive step nicely in that case there is no way to prove the base case set for $n = 2$ i.e. $P(2)$! Therefore the inductive step that "we proved" before $P(n) \Rightarrow P(n+1)$ is not true for all $n \geq 1$ but only for $n \geq 2$. This however can not establish the trueness of $P(n)$ for all $n \geq 1$ because $P(2)$ may or may not be true.

What is $P(2)$?

$P(2)$ **is "in any set of two horses, both horses are of the same color".**
In conclusion, the whole "horsy argument" breaks down because

A2. $P(n) \Rightarrow P(n+1)$ for all $n \geq 1$,
was not shown, for all $n \geq 1$; it was only proved for all $n \geq 2$, i.e.

A2. $P(n) \Rightarrow P(n+1)$ for all $n \geq 2$,

and thus the base case "$P(1)$ is true" can not be used with the latter version of the inductive step; for the latter we need $P(2)$ to be true WHICH IS NOT!

■

Exercise 20.

For the Fibonacci sequence we have shown that $F_n \leq 2^n$ for all $n \geq 0$. We can do a little better with the upper bound on F_n. The formal definition of the Fibonacci sequence is $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$. The method of solving recurrences through induction and in particular strong induction is known as **the substitution or the guess-and-check method**. In order to use it we need to know what to show, i.e. the predicate $P(n)$ must be given to us in advance, eg $F_n \leq 2^n$.

(a) Show that for any $n \geq 0$

$$F_n \leq 2^{n-1}.$$

(b) Show that for any $n \geq 1$

$$F_n \geq 2^{(n-1)/2}.$$

Proof. Omitted. ■

1.4 Equalities and inequalities

Exercise 21.

For any integer $n >$ show that we have the following.

$$n! \leq n^n, \quad \text{and} \quad n! \geq (n/2)^{n/2}.$$

We also have the following.

$$n! \leq \left(\frac{n}{2}\right)^{\frac{n}{2}} n^{n/2} \quad \text{and} \quad n! \geq n^{\frac{n}{2}-\frac{\sqrt{n}}{2}}.$$

Proof.
Starting with $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$, we have that

$$\begin{aligned} n! &= 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n \\ &\leq n \cdot n \cdot \ldots \cdot n \cdot n \\ &\leq n^n. \end{aligned}$$

Likewise $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$ leads to

$$\begin{aligned} n! &= 1 \cdot 2 \cdot \ldots \cdot n/2 \cdot \ldots (n-1) \cdot n \\ &\geq 1 \cdot 1 \cdot \ldots \cdot n/2 \cdot \ldots n/2 \cdot n/2 \\ &\geq (n/2)^{(n/2)}. \end{aligned}$$

Ignoring $\sqrt{n}$ terms we have the following.

$$\begin{aligned} n! &= 1 \cdot 2 \cdot \ldots \cdot \sqrt{n} \cdot \ldots (n-1) \cdot n \\ &\geq 1 \cdot 1 \cdot \ldots \cdot \sqrt{n} \cdot \ldots (n-1) \cdot n \\ &\geq \left(\sqrt{n}\right)^{n-\sqrt{n}} \\ &\geq n^{\frac{n}{2}-\frac{\sqrt{n}}{2}}. \end{aligned}$$

The corresponding upper bound can be proven similarly. ■

Exercise 22.
(a) For any integer $n > 0$ show the following.

$$(n/e)^n \leq n! \leq en(n/e)^n.$$

(b) For any integer $n > 0$ and $1 \leq k \leq n$ show the following.

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{ne}{k}\right)^k.$$

Proof.

$$\begin{aligned}
\binom{n}{k} &= \frac{n!}{k!(n-k)!} \\
&= \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k!} \\
&= \frac{n^k}{k!} \quad \text{Stirling's approximation} \\
&\leq \frac{n^k}{(k/e)^k} \\
&= \left(\frac{en}{k}\right)^k
\end{aligned}$$

Instead of using Stirling's approximation for the factorial we could have used the following Taylor expansion for e^k.

$$e^k = \sum_{i=0}^{\infty} \frac{k^i}{i!} \geq \sum_{i=k}^{i=k} \frac{k^i}{i!} \geq \frac{k^k}{k!}$$

The last one implies $e^k/k^k \geq 1/k!$.
For the lower bound we have the following.

$$\begin{aligned}
\binom{n}{k} &= \frac{n!}{k!(n-k)!} \\
&= \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k \cdot (k-1) \cdot \ldots \cdot 1} = \prod_{t=0}^{k-1} \frac{n-t}{k-t} \geq \prod_{t=0}^{k-1} \frac{n}{k} \geq \left(\frac{n}{k}\right)^k
\end{aligned}$$

For $t = 0$, $(n-t)/(k-t) = n/k \geq n/k$. For $t = 1$ through $t = k-1$, we have that $(n-t)/(k-t) \geq n/k$, as long as $k \leq n$. ∎

Exercise 23.
Show the following.

$$\binom{2n}{n} = \frac{(2n)!}{n! \cdot n!} \geq 2^n.$$

Proof.

$$\begin{aligned}
\frac{(2n)!}{n! \cdot n!} &= \frac{(2n) \cdot (2n-1) \cdot \ldots \cdot (n+1)}{n \cdot (n-1) \cdot \ldots \cdot 1} \\
&= \frac{(2n) \cdot (2n-1) \cdot \ldots \cdot (2n-i) \cdot \ldots \cdot (2n-(n-1))}{n \cdot (n-1) \cdot \ldots \cdot (n-i) \cdot \ldots \cdot (n-(n-1))} \\
&= \frac{(2n)}{n} \cdot \frac{(2n-1)}{n-1} \cdot \ldots \cdot \frac{(2n-i)}{n-i} \cdot \ldots \cdot \frac{2n-(n-1)}{n-(n-1)} \\
&\geq 2 \cdot \frac{n}{n} \cdot 2 \cdot \frac{(n-1/2)}{n-1} \cdot \ldots \cdot 2 \cdot \frac{(n-i/2)}{n-i} \cdot \ldots \cdot 2 \cdot \frac{n-(n-1)/2}{n-(n-1)} \\
&\geq 2^n
\end{aligned}$$

The latter is due to the fact that

$$\frac{n-i/2}{n-i} \geq 1 \Leftrightarrow n - i/2 \geq n - i \Leftrightarrow i/2 \leq i.$$

■

Exercise 24.
Show that $(1-1/N)^n \geq (1-n/N)$ for any integer $n \geq 1$ and $N > 0$.

Proof.
Proof by induction on n. For $n = 1$ obviously $(1-1/N)^1 \geq (1-1/N)$.
For $n = k$ let $(1-1/N)^k \geq 1-k/N$. Then for $n = k+1$ we have

$$\begin{aligned}(1-1/N)^{k+1} &= (1-1/N)^k \cdot (1-1/N) \\ &\geq (1-k/N) \cdot (1-1/N) \\ &= 1-(k+1)/N+k/N^2 \\ &\geq 1-(k+1)/N.\end{aligned}$$

■

Exercise 25.

(a) For integer $n > 0$ show that the following holds.

$$\frac{1}{n+1} \leq \ln(1+\frac{1}{n}) \leq \frac{1}{n}$$

(b) For integer $n > 1$ show that the following holds.

$$\frac{1}{n} \leq -\ln(1-\frac{1}{n}) \leq \frac{1}{n-1}$$

Proof.
Both are also a consequence of Inequality I.3.20. (a) Graph $y = 1/x$ on the Euclidean plane. Consider $x = 1$ and $x = 1+1/n$. The curve $y = 1/x$ between the two x points is bounded below by a rectangle of x side $1/n$ and y side $1/(1+1/n)$. It is also bounded above by a rectangle of x side $1/n$ and y side 1. Then we have the following.

$$\begin{aligned}
\frac{1}{n} \cdot \frac{1}{1+1/n} &\leq \int_1^{1+1/n} \frac{1}{x} dx \leq \frac{1}{n} \cdot 1 \iff \\
\frac{1}{n+1} &\leq \ln(1+\frac{1}{n}) \leq \frac{1}{n} \iff \\
\frac{n}{n+1} &\leq n \cdot \ln(1+\frac{1}{n}) \leq 1 \iff \\
\frac{n}{n+1} &\leq \ln(1+\frac{1}{n})^n \leq 1 \iff
\end{aligned}$$

The second part above proves (a). Furthermore, We take the limit $\lim_{n\to\infty} \ln(1+\frac{1}{n})^n$. We have

$$\lim_{n\to\infty} \ln(1+\frac{1}{n})^n \geq \lim_{n\to\infty} \frac{n}{n+1} = 1$$

and

$$\lim_{n\to\infty} \ln(1+\frac{1}{n})^n \leq \lim_{n\to\infty} 1 = 1$$

Therefore $\lim_{n\to\infty} \ln(1+\frac{1}{n})^n = 1$ or equivalently $\lim_{n\to\infty}(1+\frac{1}{n})^n = e$.
(b) Graph $y = 1/x$ on the Euclidean plane. Consider $x = 1$ and $x = 1-1/n$. The curve $y = 1/x$ between the two x points is bounded above by a rectangle of x side $1/n$ and y side $1/(1-1/n)$. It is also bounded below by a rectangle of x side $1/n$ and y side 1. Then we have the following.

$$\begin{aligned}
1 \cdot \frac{1}{n} &\leq \int_{1-1/n}^{1} \frac{1}{x} dx \leq \frac{1}{1-1/n} \cdot \frac{1}{n} \iff \\
\frac{1}{n} &\leq -\ln(1-\frac{1}{n}) \leq \frac{1}{n-1} \iff \\
1 &\leq -n \cdot \ln(1-\frac{1}{n}) \leq \frac{n}{n-1} \iff
\end{aligned}$$

The second part above proves (b). Furthermore, We take the limit $\lim_{n\to\infty} -\ln(1-\frac{1}{n})^n$. We have

$$\lim_{n\to\infty} -\ln(1-\frac{1}{n})^n \geq \lim_{n\to\infty} 1 = 1$$

and

$$\lim_{n\to\infty} -\ln(1-\frac{1}{n})^n \leq \lim_{n\to\infty} \frac{n}{n-1} = 1$$

Therefore $\lim_{n\to\infty} -\ln(1-\frac{1}{n})^n = 1$ or equivalently $\lim_{n\to\infty}(1-\frac{1}{n})^n = 1/e$. ■

Exercise 26.
For any n,k show that we have the following.

$$\binom{n}{k} = \binom{n}{n-k}$$

Proof. The number of ways to select k objects out of n is the number of ways to UNselect $n-k$ objects out of n thus selecting the remaining k objects. ■

Exercise 27.
For any n,k show that we have the following.

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Proof. The number of ways to select k objects out of n is equal to the number of selections of k objects that contain 1 plus the number of selections of k objects that DO NOT contain 1. If the selection of k objects contains 1 the number of such selections is to select out of the (out of 1) remaining $n-1$ objects $k-1$ of them and add to the mix 1. This is $\binom{n-1}{k-1}$. If the selection of k objects does not contain 1 the number of such selections is to select out of the (out of 1) remaining $n-1$ objects k of them. This is $\binom{n-1}{k}$. ■

Exercise 28.

Use the binomial identity below For any $x, y \in R$, and n we have the following.

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^n y^{n-k},$$

to prove the following.

The following holds. (a)

$$(1+1)^n = 2^n = \sum_{k=0}^{n} \binom{n}{k}.$$

(b)

$$(1-1)^n = 0 = \sum_{k=0}^{n} \binom{n}{k} (-1)^k.$$

(c) The number of subsets of $\{1,2,\ldots,n\}$ of even cardinality is equal to the number of subsets of $\{1,2,\ldots,n\}$ of odd cardinality and thus we have the following.

$$(1-1)^n = 0 = \sum_{k=0}^{n} \binom{n}{k} (-1)^k.$$

Proof.

$$(1-1)^n = 0 = \sum_{k=0}^{n} \binom{n}{k} (-1)^k.$$

implies

$$\binom{n}{0} + \binom{n}{2} + \ldots = \binom{n}{1} + \binom{n}{3} + \ldots$$

■

1.5 Entropy and probability

Exercise 29.

(a) Find the extrema of $F(x) = x\lg(x) + (n-x)\lg(n-x)$, $0 < x < n$.
(b) Find the minimum of the entropy function $f(x) = x\lg x + (1-x)\lg(1-x)$. $0 < x < 1$ and relate it to part (a).

Proof. For $0 < x < n$ we use $y = x/n$ i.e. $x = yn$, and then $0 < y < 1$. Then

$$\begin{aligned} F(x) &= x\lg(x) + (n-x)\lg(n-x) \\ F(yn) &= yn\lg(yn) + (n-yn)\lg(n-yn) = yn\lg y + yn\lg n + n(1-y)\lg(1-y) + n(1-y)\lg n \\ &= yn\lg y + n(1-y)\lg(1-y) + n\lg n \\ &= nf(y) + n\lg n \end{aligned}$$

In the remainder we study $f(y)$ with $0 < y < 1$.

$$f(y) = y\lg(y) + (1-y)\lg(1-y).$$

We first find $f'(y)$.

$$\begin{aligned} f'(y) &= \frac{\ln y + y \cdot 1/y - \ln(1-y) + (1-y)(-1)/(1+y)}{\ln 2} \\ &= \frac{\ln y - \ln(1-y)}{\ln 2}. \end{aligned}$$

$f'(y) = 0$ for $y = 1-y$ i.e. $y = 1/2$. Then we find the second derivative of f.

$$f''(y) = \frac{1}{\ln 2}\left(\frac{1}{y} + \frac{1}{1-y}\right).$$

Then we conclude $f''(1/2) = 4/\ln 2 > 0$. Thus f has a minimum for $y = 1/2$ and so does F at $x = yn = n/2$. ■

Exercise 30.

You are given a random vector $a = (a_1, \ldots a_n)$, where a_i is equally likely and independently to be 0 or 1, i.e. $Pr(a_i = 1) = Pr(a_i = 0) = 1/2$. Answer the following questions.
(a) **(Warmup)** What's the probability that a is the all zero vector ?
(b) Suppose that a, b are two 0-1 vectors of length n whose components were chosen uniformly at random as discussed previously. What is the expected value of the inner product $a \cdot b = \sum_{i=1}^{n} a_i b_i$? Explain.
(c) Let d be a vector of integers mod p (i.e. elements of d are $0, \ldots p-1$), where p is a prime. Let a be a random vector of 0-1's chosen as before. What is an upper bound on the probability that $\sum d_i a_i \equiv 0 \bmod p$? Explain.

Proof.
(a) $1/2^n$.
(b) $n/4$.

$$E[c] = E[ab] = E[\sum_i a_i b_i] = \sum_i E[a_i b_i]$$

$a_i b_i$ is 0 with probability $3/4$ and 1 with probability $1/4$ (when both $a_i = b_i = 1$).

$$E[a_i b_i] = 0(3/4) + 1(1/4) = 1/4$$

Therefore

$$E[c] = E[ab] = E[\sum_i a_i b_i] = \sum_i E[a_i b_i] = n(1/4)$$

(c) The vector d is given (and is not necessarily random). Assume $d \neq 0$, i.e. at least one component of the vector is non-zero since otherwise the problem is trivial. $\sum d_i a_i \equiv 0 \bmod p$.

$$\begin{aligned} \sum_i d_i a_i &\equiv 0 \bmod p \\ d_1 a_1 + d_2 a_2 + \ldots + d_n a_n &\equiv 0 \bmod p \\ d_1 a_1 &\equiv (-d_2 a_2 - \ldots - d_n a_n) \bmod p \\ d_1 a_1 &\equiv Z \bmod p \end{aligned}$$

where $Z \equiv (-d_2 a_2 - \ldots - d_n a_n) \bmod p$. Then,

$$\begin{aligned} \sum_i d_i a_i &\equiv 0 \bmod p \\ d_1 a_1 &\equiv Z \bmod p \\ a_1 &\equiv (d_1)^{-1} Z \bmod p \\ a_1 &\equiv B \bmod p \end{aligned}$$

The inverse of d_1 exists since $d_1 x \equiv 1 \bmod p$ has a single solution for x by the fact that $d_1 < p$ is such that $(d_1, p) = 1$ and p is prime. $B = (d_1)^{-1} Z \bmod p$ is an integer in $0, \ldots, p-1$. a_1 is a (uniformly at) random (chosen) 0,1. What is the probability that the random a_1 is B? Naturally this probability is at most $1/2$, as after we fix B, a_1 can agree with this fixed value of B half of the time only. If the a_i's were not binary but ternary, then the probability bound would be $1/3$ instead. ■

Exercise 31.

(Some parts are from a previous problem.)
(a) Find the extrema of $F(x) = x\lg(x) + (n-x)\lg(n-x)$, $0 < x < n$.
Is $F(n/4) \geq n\lg n - n$? Is $F(n/2) \geq n\lg n - n$?
(b) Analyze then entropy function $f(x) = -x\lg x - (1-x)\lg(1-x)$, with $0 < x < 1$, by analysing $g(x) = x\lg x + (1-x)\lg(1-x)$.

Proof. This is primarily Calculus 1. This problem reviews topics related to finding the extrema (min or max) of functions. One needs it to prove the best case running time of QuickSort. Consider.

$$f(x) = -x\lg x - (1-x)\lg(1-x)$$

and $g(x) = -f(x)$ i.e.

$$g(x) = x\lg x + (1-x)\lg(1-x).$$

If $g(x)$ has a minimum or maximum then its negative i.e. $f(x)$ will have a maximum or a minimum respectively.
STEP 1. (Symmetry over $x = 1/2$). The first observation is that $f(x), g(x)$ are symmetric around $x = 1/2$. Thus $f(0) = f(1)$ if they were defined and in general $f(x) = f(1-x)$.
Step 2. (Behavior at $x = 0$. What is the limit $\lim_{x\to 0} g(x)$?

$$\lim_{x\to 0} g(x) = \lim_{x\to 0}(x\lg x + (1-x)\lg(1-x)) = \lim_{x\to 0}(x\lg x + (1-0)\lg(1-0)) = \lim_{x\to 0} x\lg x.$$

Step 3. (Change variables $y = 1/x$.) At this point we change variables $y = 1/x$ or $x = 1/y$. For $x \to 0$, or $1/y \to 0$, we have $y \to \infty$. Therefore

$$\lim_{x\to 0} g(x) = \lim_{x\to 0} x\lg x = \lim_{y\to\infty}(1/y)\lg(1/y) = \lim_{y\to\infty} \frac{-\lg y}{y} \overset{L'Hopital}{=} \lim_{y\to\infty} \frac{-(\lg y)'}{(y)'} = \lim_{y\to\infty} \frac{-1/y}{1} = 0^-$$

Thus the limit $\lim_{x\to 0} g(x) = 0$ from the left side (from the negative) and we represent this by having a minus as a superscript to the 0. It also suggests that $\lim_{x\to 1} g(x) = 0^-$ as well by STEP 1. Therefore for $f(x) = -g(x)$ we have $\lim_{x\to 1} f(x) = 0^+ = \lim_{x\to 0} f(x)$.

STEP 4. (First Derivative of g) What is $g'(x)$?
Simple calculations reveal that

$$g'(x) = \lg x - \lg(1-x)$$

The first derivative is 0 for $\lg x = \lg(1-x)$ i.e. for $x = 1/2$. Therefore at $x = 1/2$ we have a minimum or maximum (most likely).
STEP 5. (Second Derivative of g) To determine the type of the extreme (minimum or maximum), we examine the second derivative for $x = 1/2$ and then we obtain that

$$g''(x) = 1/x + 1/(1-x), \text{ and thus } g''(1/2) = 4 > 0.$$

Therefore $g(x)$ has a minimum at $x = 1/2$ and symmetrically $f(x)$ has a maximum. It is easy to find that $g(1/2) = -1$, and thus $f(1/2) = 1$.
STEP 6 ($F(x)$). Use $y = x/n$ in $F(x)$, $0 < x < n$. Then, $x = yn$ with $0 < y < 1$. Then

$$\begin{aligned} F(x) &= x\lg(x) + (n-x)\lg(n-x) \\ F(yn) &= yn\lg(yn) + (n-yn)\lg(n-yn) = yn\lg y + yn\lg n + n(1-y)\lg(1-y) + n(1-y)\lg n \\ &= yn\lg y + n(1-y)\lg(1-y) + n\lg n \\ H(y) &= n(y\lg y + (1-y)\lg(1-y)) + n\lg n = ng(y) + n\lg n \end{aligned}$$

Function $H(y)$ has a constant term (not dependent on y) $n\lg n$ plus $ng(y)$. Therefore it retains a minimum at $y = 1/2$ i.e. $x = yn = n/2$ and $g(1/2) = -1$ by Step 5.

STEP 7 (Conclusion). As far as $F(n/4)$ is concerned, given that $g(x)$ has a minimum for $g(1/2)$ we have that $g(1/4) \geq g(1/2)$ and thus for $x = yn$, i.e. $y = 1/4$ we have that

$$F(n/4) = n\,(g(1/4)) + n\lg n \geq ng(1/2) + n\lg n \geq n\lg n - n.$$

■

1.6 Recursion

Exercise 32.

You are given the following function definition in pseudo C++/C. How much is $f(91)$? How much is $f(92)$? How much is $f(99)$? How much is $f(100)$? Explain.

```
int f(int x) // x is integer
  return( (x>100) ? (x-10) : f(f(x+11)) );
```

Proof. We start establishing $f(91)$ by using the two conditions A: $(x-10)$ if $x > 100$ and B: $f(x) = f(f(x+11))$ otherwise.

$$\begin{array}{rll} f(91) & =_{ConditionB} & f(f(91+11)) \\ & = & f(f(102)) \\ & =_{ConditionA} & f(102-10) \\ & = & f(92) \end{array}$$

We have just proved that $f(91) = f(92)$. We next find $f(92)$.

$$\begin{array}{rll} f(92) & =_{ConditionB} & f(f(92+11)) \\ & = & f(f(103)) \\ & =_{ConditionA} & f(103-10) \\ & = & f(93) \end{array}$$

We have just proved that $f(91) = f(92) = f(93)$. We next find that $f(91) = f(92) = f(93) = f(94)$, and proceeding similarly that $f(91) = f(92) = \ldots = f(100)$. We thus find now $f(100)$.

$$\begin{array}{rll} f(100) & =_{ConditionB} & f(f(100+11)) \\ & = & f(f(111)) \\ & =_{ConditionA} & f(111-10) \\ & = & f(101) \\ & =_{ConditionA} & 101-10 \\ & = & 91 \end{array}$$

Therefore $f(91) = f(92) = \ldots = f(99) = f(100) = 91$. ■

Exercise 33.

```
int f(int x) // x is integer
  if (x>=100) {
    return(x-10);
  } else {
    return( f(f(x+14)) );
    }
```

How much is f(91)? f(92)? f(93)? f(94)?

Proof. Let's start with $f(91)$.
It is obvious that $f(91) = f(f(105)) = f(95) = f(f(109)) = f(99) = f(f(113)) = f(103) = 93$. Thus $f(91) = f(99) = 93$. Similarly, $f(92) = f(f(106)) = f(96) = f(f(110)) = f(100) = 90$, and thus $f(92) = f(100) = 90$.
Let's go on with $f(93)$.
It is obvious that $f(93) = f(f(107)) = f(97) = f(f(111)) = f(101) = 91$. Thus $f(93) = f(101) = 91$. Similarly, $f(94) = f(f(108)) = f(98) = f(f(112)) = f(102) = 92$, and thus $f(94) = f(98) = 92$. ■

Exercise 34.

What does the following program do? Explain.

```
int  Findme ( int x , int y) {
 if ( y == 0)  return (0);
   else
    return( Findme( 2*x, floor(y/2)) + x * (y % 2 ) );
}
```

Proof.
Suppose we want to divide an integer y by 2. y is the dividend and 2 is the divisor. By the elementary school division we have that

$$y = 2Q + R$$

where Q is the quotient and R is the remainder. R is 0 or 1. The quotient is $Q = floor(y/2)$, and the remainder is $R = y\%2$. Therefore,

$$y = 2Q + R = 2floor(y/2) + (y\%2)$$

If we multiply by x both sides of the equality we get that

$$x * y = x * (2floor(y/2)) + x * (y\%2) = (2 * x) * (floor(y/2)) + x * (y\%2).$$

Therefore the product $x * y$ is equal to the sum of the product of $2x$ times $floor(y/2)$ and $x * (y\%2)$. Note that the product $x * (y\%2)$ is easy to find. If the $y\%2$ is 0, the product is 0, if $y\%2$ is 1 the product is x. Therefore Findme finds the product $x * y$ recursively. ■

1.7 Trace the code

Exercise 35.

```
// CODE A
1. int  A (int n)
2. int  i,j;
3. int  sum=0;
4. for (i=1; i<= 4; i++) {
5.   for (j=1; j<= n; j++) {
6.      sum = sum + i;
7.   }
8. }
9. return(sum);
```

```
// CODE B
1. int  B (int n)
2. int  i,j;
3. int  sum=0;
4. for (i=1; i<= n; i++) {
5.   for (j=1; j<= 4; j++) {
6.      sum = sum + j;
7.   }
8. }
9. return(sum);
```

What is the value returned by sum in Code A and in Code B?

Proof.
For the first-case we have $\sum_{i=1}^{4}\sum_{j=1}^{n} i = \sum_{i=1}^{4} n*i = n*\sum_{i=1}^{4} i = n*(1+2+3+4) = 10n$ For the second-case we have $\sum_{i=1}^{n}\sum_{j=1}^{4} j = \sum_{i=1}^{n} 10 = 10n$. ■

Exercise 36.

```
1. int Loop (int n) // Search for key in A.
2. int i,j;
3. j=0;
4. for (i=0;i<n;i++)
5.   j=j+i;
6. return(j);
```

(a) How many times is line 4 executed? How many times is line 5 executed? Express your answer as a function of n. (Be very very careful with the first question!)
(b) What is the value returned by j as a function of n? Express the answer in closed form.

Proof.
(a) Line 4 is ececuted once more than line 5. This is because i is first compared to n, and if $i < n$, then line 5 is executed and then i is incremented; i is incremented a total of n times for $i = 0, \ldots, n-1$ to reach values $i = 1, \ldots, n$, after it tests successfully $i < n$ and Line 5 is executed. Thus Line 5 is executed n times. The last n-th time i is incremented to n, having previously executed for the n-th time line 5. It is then compared through the test of Line 4 $i < n$ to n; the comparison is false and Line 5 is not executed. Line 4 has been executed $n+1$ times. Among them the comparison $i < n$ was TRUE n times (for $i = 0, \ldots, n-1$) and FALSE once (for $i = n$).
(b) The value returned by j is $j = \sum_{i=0}^{n-1} i = (n-1)n/2$. ■

Exercise 37.

```
1. int CodeA (int n)
2. int i,j,t;
3. t=0;
4. for(i=0;i<n;i++)
5.   for(j=0;j<n;j++)
6.     t= t + i;
7. return(t);
```

```
1. int CodeB( int n)
2. int i,t;
3. t=0;
4. for(i=1;i<n;i*= 2)
5.   t=t+i;
6. return((t+1));
```

Let n be a power of two, i.e. $n = 2^k$ for some integer k. What is the value returned in CodeA and CodeB as a function of n? Explain.

Proof.
Let us consider first program CodeA. Note that line 6 depends only on j and not on i. Since the body of the i-forloop is entered n times for $i = 0$ through $i = n-1$, this means that lines $5-6$ are executed n times. The value t is thus

$$t = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} i = n \sum_{i=0}^{n-1} i = n(n(n-1)/2) = n^2(n-1)/2.$$

This is the value returned by CodeA. Because of Line 4, i is doubled every time line 5 is executed, i.e. i is initially set $i = 1 = 2^0$, after the first execution of line 5, i becomes $i = 2 = 2^1$, after the second execution of line 5, i becomes $i = 2 = 2^2$, and so one after the m-th execution of line 5, i becomes $i = 2 = 2^m$. This terminates when the test $i < n$ fails i.e. when $2^m \geq n$. Since $n = 2^k$, $2^m \geq 2^k$ gives that $m \geq k$. Therefore at the end of the k-th execution of line 5, the test of the forloop fails and control is transferred to line 6. During the first execution of line 5, $i = 2^0$, and this values is added to t. During the second execution of line 5, $i = 2^1$ and this value is added to t. During the k execution of line 5, $i = 2^{k-1}$ and this value is added to t. Therefore, to the original values of t which is zero, the term $2^0 + 2^1 + 2^2 + \ldots 2^{k-1}$ is added. Therefore t becomes $t = 2^0 + 2^1 + 2^2 + \ldots 2^{k-1} = 2^k - 1 = n - 1$, since the sum is a geometric sequence. The value $t + 1 = n$ is thus returned by CodeB. ■

Exercise 38.

```
1. int A (int n)
2. int i=0,j=1;
3. for (i=0;i<n;i++)
4.   j=j+2*i;
5. return(j);
```

What is the value of j returned by code A?

Proof. Line 3 is executed $n+1$ times. Line 4 is executed n times. The value returned by j is

$$1+\sum_{i=0}^{n-1} 2i = 1+2\sum_{i=0}^{n-1} i = 1+2n(n-1)/2 = 1+n(n-1) = n^2-n+1.$$

■

Exercise 39.

```
1. int  CodeA (int n)
2. int  i,j;
3. int  sum=0;
4. for (i=1; i<= n; i++) {
5.   for (j=1; j<= n; j++) {
6.      sum = sum +  2*i  ;
7.   }
8. }
9. return(sum+1);
```

```
 int CodeB (int n)
1. int i;
2. int sum=0;
3. for(i=1;i<=n;i=i*2) {
4.   sum = sum + 1;
5. }
6. return(sum)
```

What is the value returned by `CodeA` and `CodeB`? Explain.

Proof.
The value returned by `CodeA` is 1 plus the value of the sum. The value of `sum` is

$$sum = \sum_{i=1}^{n}\sum_{j=1}^{n}(2i) = \sum_{i=1}^{n}(2ni) = 2n\sum_{i=1}^{n} i = 2nn(n+1)/2 = n^2(n+1)$$

and thus the value returned is $1+n^2(n+1)$. For the type B exam the value of `sum` is twice as much i.e. $2n^2(n+1)$ since the additive term is $4*i$ not $2*i$, and thus the value returned is $2n^2(n+1)+1$.
The value returned by `CodeB` is given below. Note that if i runs through the loop of line 3 as $i = 2^0, 2^1, 2^2, 2^{\lfloor \lg n \rfloor}$, by using a new index $j = \lg i$, we have that j runs through the same loop as $j = 0, \ldots, \lfloor \lg n \rfloor$. No matter whether one uses i or j the value added is the same constant 1. Therefore the sum

$$sum = \sum_{j=0}^{j=\lfloor \lg n \rfloor} 1 = \lfloor \lg n \rfloor + 1.$$

■

Exercise 40.

```
1. int CodeA (int n)
2. int i,j;
3. j=1;
4. for(i=0;i<n;i++)
5.   j = j*2;
6. return(j);
```

```
1. int CodeB( int n)
2. int i,j;
3. j=1;
4. for(i=1;i<n;i*= 2)
5.   j=j+1;
6. return(j);
```

What is the value returned by CodeA and CodeB as a function of n, if n is a power of two i.e. $n = 2^k$ for some positive integer k? Explain.

Proof.
(a) Let us consider first program CodeA. i increases $0,1,2,3,\ldots,n-1,n$. For $i = 0,\ldots,n-1$ line 5 is executed; for $i = n$ the loop exits and line 5 is NOT executed. Every time line 5 is executed the value of j is doubled. The first time j becomes 2 and thus j takes values $2,4,8,16,\ldots,2^n$. It is easy to show by induction that when i takes value K, j receives value 2^{K+1}. Since the last value i takes in the loop of line 5 is $n-1$ the largest value j assumes is thus 2^n.
(b) Let $n = 2^k$. In CodeB i is doubled every time and thus i increases $i = 1,2,4,\ldots,n/2 = 2^{k-1}$. When $i = n/2 = 2^{k-1}$ and i is subsequently doubled, i becomes n and the test $i < n$ thus fails and the loop stops. Therefore whenever $i = 2^0, 2^1, 2^2, \ldots, 2^{k-1}$ variable j is incremented by one thus taking values $j = 2,3,4,\ldots(k-1)+2 = k+1$, i.e. j is set at the i-th iteration equal to $\lg i + 2$. When the loop is executed for the last time, $i = 2^{k-1}$, and thus $j = \lg 2^{k-1} + 2 = k+1$. Since $n = 2^k$, we have $k = \lg n$ and thus, j returns $\lg n + 1$. ■

Exercise 41.

```
1. int  A (int n)
2. int  i,j;
3. int  sum=0;
4. for (i=1; i<= n; i++) {
5.   for (j=1; j<= n; j++) {
6.      sum = sum + 2*i +4*j;
7.   }
8. }
9. return(sum);
```

What is the value of sum returned?

Proof. For the value of sum returned, $sum = \sum_{i=1}^{n}\sum_{j=1}^{n}(2i+4j)$.

$$\sum_{i=1}^{n}\sum_{j=1}^{n}(2i+4j) = \sum_{i=1}^{n}((2in)+\sum_{j=1}^{n}4j) = \sum_{i=1}^{n}(2in+4n(n+1)/2) = \sum_{i=1}^{n}(2in+2n(n+1)) =$$

concluding

$$sum = (2nn(n+1)/2+2n^2(n+1)) = n^2(n+1)+2n^2(n+1) = 3n^2(n+1)$$

■

Exercise 42.

```
1. int  A (int n)
2. int  i,j;
3. int  sum=0;
4. for (i=1; i<= n; i++) {
5.   for (j=1; j<= 6; j++) {
6.      sum = sum +  i    ;
7.   }
8. }
9. return(sum);
```

```
1. int B (int n)
2. int sum=1;
3. int x=1;
4. while (x <= 32) {
5.    x = 2*x;
6.   sum = sum+1;
7. }
8. return(sum);
```

What is the value of sum returned?

Proof.
The value of `sum` returned by Code `A` is

$$\text{sum} = \sum_{i=1}^{n}\sum_{j=1}^{6} i = \sum_{i=1}^{n}(6i) = 6\sum_{i=1}^{n} i = 6n(n+1)/2 = 3n(n+1).$$

For `Code B` the while loop is executed 7 times with the body of the loop being executed 6 times. Each time the body is executed variable `sum` is incremented by 1. Therefore `sum=7` at the end and this value is returned. ■

Exercise 43.

```
1. int  CodeA (int n)
2. int  i,j;
3. int  sum=0;
4. for (i=1; i<= n; i++) {
5.   for (j=1; j<= n; j++) {
6.      sum = sum +  i+j  ;
7.   }
8. }
9. return(sum);
```

```
 int CodeB (int n)
1. int i;
2. int sum=0;
3. for(i=1;i<=1024;i=i*2) {
4.   sum = sum + 10;
5. }
6. return(sum)
```

What is the value of sum returned by CodeA and CodeB respectively. Justify your answer.

Proof.
(a) For CodeA the two for loops give a running time (worst case) of $\Theta(n^2)$, a tight bound; $O(n^2)$ would be accepted for partial credit even if it is not tight. For CodeB running time is fixed i.e. $\Theta(1)$, constant.
(b) For CodeA

$$\begin{aligned} sum &= \sum_{i=1}^{n}\sum_{j=1}^{n}(i+j) = \sum_{i=1}^{n}\left(ni+\sum_{j=1}^{n}(j)\right) \\ &= \sum_{i=1}^{n}(ni+n(n+1)/2) = \left(\sum_{i=1}^{n}ni\right)+n^2(n+1)/2 \\ &= n^2(n+1)/2+n^2(n+1)/2 = n^2(n+1) \end{aligned}$$

For CodeB, consider variable $j = \lg i$. While i runs $i = 1,2,4,8,\ldots,1024$, j runs $j = 0,1,2,3,\ldots 10$. Therefore

$$sum = \sum_{j=0}^{10} 10 = 11 \cdot 10 = 110.$$

■

Exercise 44.

```
1. int CodeA (int n)
2. int i,j,sum=0;
3. j=1;
4. for(i=1;i<n;i*=2)
5.   sum = sum+i;//this is i not one
6. return(sum+1);
```

```
1. int CodeB( int n)
2. int i,j,k;
3. j=1;k=0
4. for(i=1;i<=n;i++)
5.   for(j=1;j<=n;j++)
6.    k = k+2;
7. return(k);
```

What is the value returned by `CodeA` and `CodeB`? Express your answer in asymptotic notation. (For example, instead of saying $n+1$, you should say $\Theta(n)$.)

Proof.
Variable i is doubled every time (i.e. $i = 1 = 2^0$, then $i = 2 = 2^1$, $i = 4 = 2^2$, and so on). So we can trace i if we trace another "notional" variable k such that $i = 2^k$ or $k = \lg i$. When i goes through $1, 2, 4, \ldots$, k goes through $0, 1, 2$. If k is such that $2^k \geq n$ line 5 is NOT EXECUTED. Solving for k we have that for $k \geq \lceil \lg n \rceil$ line 5 is not executed, i.e. it is executed for $k = 0 \ldots \lceil \lg n \rceil - 1$. These are a total of $\lceil \lg n \rceil$ values (since k starts from 0) for k, they correspond to $i = 1, 2, 4, \ldots$, and give the number of times line 5 is executed.
Similarly for `CodeB` lines 4 and 5 are executed $n+1$ times since i or j start from 1 and increment by one every time. Since the two loops are nested, line 6 is executed a number of times equal to that product. For `CodeA`, the value of `sum+1` is,

$$1 + \left(2^0 + 2^1 + 2^2 + \ldots + 2^{\lceil \lg n \rceil - 1}\right) = 1 + 2^{\lceil \lg n \rceil} - 1 = 2^{\lceil \lg n \rceil}.$$

The first one counts the one of `sum+1`. The parenthesized expression is the value of `sum` overall iterations and individual terms give the value of i per iteration. How much is $2^{\lceil \lg n \rceil}$? Since $x \leq \lceil x \rceil \leq x+1$ we have that $2^x \leq 2^{\lceil x \rceil} \leq 2^{x+1} = 2 \cdot 2^x$. Substituting $\lg n$ for x we get a value between n and $2n$, i.e. the value returned is $\Theta(n)$.
For `CodeB` the case is easier to handle.

$$k = \sum_{i=1}^{n} \sum_{j=1}^{n} 2 = n \cdot n \cdot 2 = 2n^2$$

i.e. the value returned is $\Theta(n^2)$. ■

Exercise 45.

```
1. int Loop (int n) // Search for key in A.
2. int i,j;
3. j=0;
4. for (i=0;i<n;i++)
5.   j=j+i;
6. return(j);
```

(a) How many times is line 4 executed? How many times is line 5 executed? Express your answer as a function of n. (Be very very careful with the first question!)
(b) What is the value returned by j as a function of n? Express the answer in closed form.

Proof.
Line 4 is ececuted once more than line 5. This is because i is first compared to n, and if $i < n$, then line 5 is executed and then i is incremented; i is incremented a total of n times for $i = 0, \ldots, n-1$ to reach values $i = 1, \ldots, n$, after it tests successfully $i < n$ and Line 5 is executed. Thus Line 5 is executed n times. The last n-th time i is incremented to n, having previously executed for the n-th time line 5. It is then compared through the test of Line 4 $i < n$ to n; the comparison is false and Line 5 is not executed. Line 4 has been executed $n+1$ times. Among them the comparison $i < n$ was TRUE n times (for $i = 0, \ldots, n-1$) and FALSE once (for $i = n$). The value returned by j is $j = \sum_{i=0}^{n-1} i = (n-1)n/2$. ■

Exercise 46.

```
1. int  A (int n)                    1. int B (int n)
2. int  i,j;                         2. int sum=1;
3. int  sum=0;                       3. int x=1;
4. for (i=1; i<= n; i++) {           4. while (x <= 1024) {
5.   for (j=1; j<= n; j++) {         5.    x = 2**x;
6.      sum = sum +  i    ;         6.    sum = sum+1;
7.   }                               7. }
8. }                                 8. return(sum);
9. return(sum);        //Note: 2**x is 2 raised to the x-th power.
```

What is the value returned by sum?

Proof.
Code A has two for loops in lines 4 and 5. The bodies of the for loops are repeated $n \cdot n = n^2$ times. Code B has a single while loop. It runs a constant number of times. Lines 5 and 6 are executed 4 times i.e a constant number.

The value of sum returned by Code A is

$$\text{sum} = \sum_{i=1}^{n} \sum_{j=1}^{n} i = \sum_{i=1}^{n} (ni) = n \sum_{i=1}^{n} i = n^2(n+1)/2.$$

The value of sum returned by Code B is $\text{sum} = 5$, since if we trace lines 5 and 6 we get:

$$\begin{aligned}
&\text{sum} = 2 \quad , \quad \text{x} = 2 \\
&\text{sum} = 3 \quad , \quad \text{x} = 4 \\
&\text{sum} = 4 \quad , \quad \text{x} = 16 \\
&\text{sum} = 5 \quad , \quad \text{x} = 65536 \quad \text{and exit from loop}
\end{aligned}$$

■

Exercise 47.

```
1. int  X (int n)
2. int  i,j;
3. int  sum=0;
4. for (i=1; i<= n; i++) {
5.   for (j=n; j<= n+11; j++) {
6.      sum = sum + i;
7.   }
8. }
9. return(sum);
```

What is the value of sum returned by X?

Proof.
The loop of line 4 iterates n times from $i = 1$ to $i = n$ inclusive. The loop of line 5 iterates 12 times from $j = n = n+0$ to $j = n+11$. Thus line 6 is executed a total of $12n$ times.

The value returned by `sum` is

$$\texttt{sum} = \sum_{i=1}^{i=n} \sum_{j=n}^{j=n+11} i = \sum_{i=1}^{i=n} 12i = 12 \sum_{i=1}^{i=n} i = 12n(n+1)/2 = 6n(n+1).$$

Thus the value returned by `sum` is $6n(n+1)$. ■

Exercise 48.

```
1. void M (X,Y,Z,n)
2. int  X[n][n] , Y[n][n], Z[n][n];
3. int  i,j,k;
4. int  sum=0;
5.      for (i=0;i<n;i++)
6.         for (j=0;j<n;j++) {
7.             sum=0;
8.             for(k=0;k<n;k++) {
9.                sum = sum+ x[i][k] * y[k][j];
10.            }
11.            Z[i][j]=sum;
12.        }
```

You are given the piece of C/C++ pseudocode shown above. X, Y and Z are three $n \times n$ two-dimensional arrays.
(a) How many times is the assignment of line 7 executed? Express the answer as a function of n. Explain.
(b) How many times is variable sum updated in line 9 of Program M ? Express the answer as a function of n. Explain.
(c) What is the total number of multiplications performed in line 9 of Program M? Express the answer as a function of n. Explain.

Proof. Function M is an implementation of matrix multiplication of two matrices.
(a) The assignment in line 7 is executed

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (1)$$

number of times because of the two **for loop**s of lines 5 and 6. However,

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (1) = n^2$$

(b) Similarly, line 9 is updated

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} (1) = n^3$$

times.
(c) The total number of multiplications performed in line 9 is equal to the number of times sum is updated in line 9, i.e. n^3. ■

1.8 Linear and binary search review

Exercise 49.

You are given the following piece of code. The top level call is Search(A, key, compare), where A is a sequence of arbitrarily ordered keys, `key` is the key to be found and `compare` is the function that compares two keys and determines whether first argument is less, equal, or greater than the second and returns a negative or -1, 0, a positive or $+1$ value recursively. Arbitrarily means you can not assume that keys have a given order. You may assume however that they are distinct and each key value appears once. (One can withdraw this assumption.)
(a) If `key` is not in A, how many times does the $==$ of line 3 is tested? Express your answer as a function of n.
(b) If `key` is in A, what is the maximum number of times the $==$ of line 3 is tested? Express your answer as a function of n.
(c) What is the maximum number of times line 2 is executed? (This is a tricky question). Express your answer as a function of n.
(d) Suppose that `key` is in A and is equally likely to be in position $A[0]$, or $A[1]$, or ..., or $A[n-1]$. What is the average number of comparisons performed to find `key`? Explain.

```
//
    Search(A,key,compare,n)        // C/C++/Java indexing
     LinS (A,key,compare,0,A.length()-1); // A[0..n-1]
//  Implicit assumption A.length() is available or otherwise use
//   LinS (A,key,compare,n);   // A[0..n-1]
//
 LinS(A,key,compare,n) {
1. int i;
2. for(i=0;i<n;i++)
3.   if (A[i] == key ) return(i);
4. return(-1);
```

Proof.
(a) If `key` is not in A the test in line 3 is executed n times once per each iteration of i.
(b) If `key` is in A the test in line 4 is executed $j+1$ times where j is the first location of A such that $A[j] = key$. Since j is at most $n-1$, thus at most n times is line 3 executed as well.
(c) Line 2 is executed once more than line 3. The extra case includes the case $i = n$ when the test $i < n$ fails and the loop exits. Thus $n+1$ times is line 2 executed.
(d) `key` is in position i with probability $1/n$ where i can be $0,\ldots,n-1$. If `key` is in position i, then $i+1$ comparisons are performed on the elements in locations $0\ldots i$ to find `key`. Therefore the average number of comparisons performed is $1/n * \sum_{i=0}^{n-1}(i+1) = 1/n * \sum_{i=1}^{n} i = 1/n * n(n+1)/2 = (n+1)/2$. ■

Exercise 50.

Give (pseudo) code for a recursive binary search. The top level call is BinSearch(A, key, compare), where A is a sorted sequence, key is the key to be found and compare is the function that compares two keys and determines whether first argument is less, equal, or greater than the second and returns a negative or -1, 0, a positive or $+1$ value recursively.
What is the worst-case number of comparisons performed? The output should be -1 for not found or index i such that $A[i] == key$.

Proof.

```
// A Recursive Binary Search.
//
 BinSearch(A,key,compare)   // Pascal indexing
   BinS (A,key,compare,1,n); // A[1..n] is sorted
//
//
 BinS(A,key,compare,l,r) {
    // A[l..r] is searched for key in this call to BinS
1.  if (l > r) return(-1);  // not found
2.    else {
3.       k = compare(key, A[floor(l+(r-l)/2)]);
4.       if ( k == 0) {  return(l+(r-l)/2); }
5.       else {
6.         if ( k < 0 )
7.           return BinS(A,key,compare,l,floor(l+(r-l)/2)-1);
8.         else
9.           return BinS(A,key,compare,floor(l+(r-l)/2)+1,r);
10.      }
11.   }
12.
```

To simplify exposition of the worst case, let $n = 2^k - 1$, key=A[1] (the first element) and all $A[i]$ are distinct (and of course, they appear in increasing a.k.a. sorted order). In the first call to BinS key is to search the whole array $A[1..n = 2^k - 1]$ i.e. $l = 1$ and $r = 2^k - 1$, and it is first compared and found to be smaller than $A[(1+n)/2] = A[2^{k-1}]$. The recursive call in step 7 is then issued which means that subarray $A[1..2^{k-1} - 1]$ will be searched for key. In the following call key is compared and found to be smaller than $A[(1+2^{k-1}-1)/2] = A[2^{k-2}]$, and thus step 7 is executed. The sequence of elements of A key is compared with is thus $A[2^{k-1}]$, $A[2^{k-2}]\ldots$, $A[2^i]\ldots$, $A[2^0]$ ie a total of k keys. Eventually in the last comparison key is found to be equal to $A[2^0 = 1]$, line 4 is then true and procedure Bin-Search returns successfully. The number of comparisons performed in line 6 of all recursive calls is equal to k, and since $k = \lg(n+1)$, the total number of comparisons is $\Theta(\lg(n+1)) = \Theta(\lg n)$. This is one scenario for the worst-case. The implication also is that the running time or the number of comparisons is $O(\lg n)$. (We might get lucky and the first time line 4's conditional evaluates to true.) ■

Exercise 51.

Give (pseudo) code for an iterative binary search. The top level call is BinSearch(A, key, compare),

Proof.

```
 BinSearch(A,key,compare)   // Pascal indexing
   BinSI(A,key,compare,1,n); // A[1..n] is sorted
//
//
 BinSI(A,key,compare,l,r) {
    // A[l..r] is searched for key in this call to BinS
1.  while (l <= r) {
2.    m = l + (r-l)/2 ;
3.    k = compare(key, A[m]);
4.    if ( k == 0) return(l+(r-l)/2);
5.    else
6.      if ( k < 0 )
7.          r = m - 1 ;
8.        else
9.          l = m + 1 ;
10. }
```

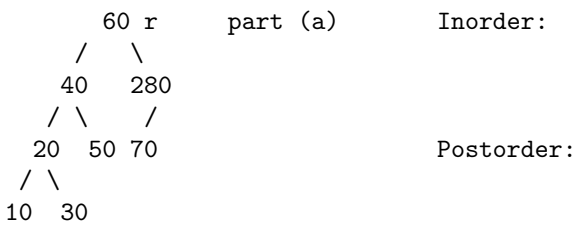
See the recursive version for more information. ■

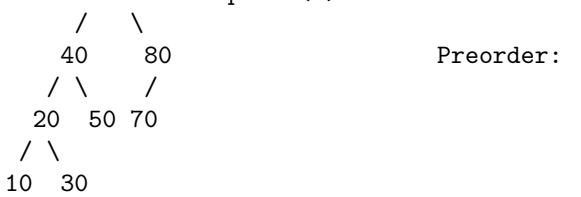
1.9 Tree traversals

Exercise 52.

(a) On the following tree starting with its root which is located at the top level perform the indicated traversals: Inorder, Postorder.
Show the sequence of nodes visited. (A reference r is next to the root's key value.)
(b) On the following tree starting with its root which is located at the top level perform the indicated traversals: Inorder, Preorder.
Show the sequence of nodes visited. (A reference r is next to the root's key value.)

Proof.





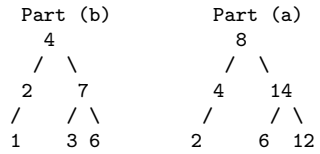
Inorder: 10,20,30,40,50,60,70,280
Postorder: 10,30,20,50,40,70,280,60

Inorder: 10,20,30,40,50,360,70,80
Preorder: 360,40,20,10,30,50,80,70 ■

Exercise 53.

(a) We perform an inorder and postorder traversal on an arbitrary binary tree For each one of the traversals we get as output:
Inorder : $2,4,8,6,14,12.$
Postorder : $2,4,6,12,14,8.$
Show the binary tree that is consistent with these two traversals.
(b) We perform an inorder and preorder traversal on an arbitrary binary tree For each one of the traversals we get as output:
Inorder : $1,2,4,3,7,6.$
Preorder : $4,2,1,7,3,6.$

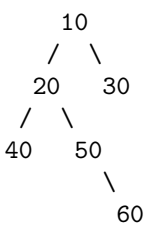
Proof.



■

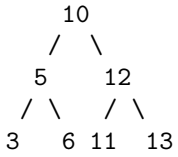
Exercise 54.
What is the binary tree T that is consistent with the following two traversals of it?
Inorder traversal of T : $40, 20, 50, 60, 10, 30$.
Postorder traversal of T: $40, 60, 50, 20, 30, 10$.

Proof.



■

Exercise 55.
In the tree shown perform



(a) inorder, (b) preorder, (c) postorder traversals. Show your results.

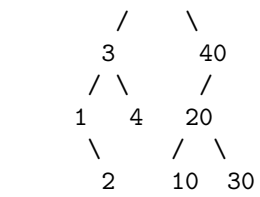
Proof. (a) An inorder traversal yields the sequence: 3 5 6 10 11 12 13.
(b) A preorder traversal yields the sequence: 10 5 3 6 12 11 13.
(c) A postorder traversal yields the sequence: 3 6 5 11 13 12 10.

■

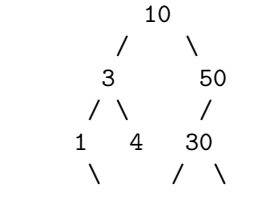
Exercise 56.

What is the statement of the problem that has the following solution? No kidding!

Proof.



Preorder Traversal: 5 3 1 2 4 40 20 10 30



Postorder Traversal: 2 1 4 3 20 40 30 50 10

■

1.10 Graph theory review

Exercise 57.

Definition 1.1

A connected undirected graph $G = (V, E)$ is a tree T if it is acyclic (no cycles). If it is not connected, then it is called a forest. A spanning tree of G is a tree that touches all of its vertices.

Let $G = (V, E)$. For a subgraph T of G, we have $T = (V, E')$, where $E' \subset E$. We might also write $V(T)$, or $E(T)$ for V and E') respectively of $E(G)$ and $V(G)$.
(i) A tree has exactly one path between any pair of vertices.
(ii) A tree with at least two vertices has at least two leaves (vertex with degree one).
(iii) A tree (undirected connected graph) with n vertices has $n-1$ edges.
(iv) Every connected graph has a spanning tree.
(v) If G is connected it has at least $n-1$ edges.
(vi) A spanning tree of a connected undirected graph with n vertices has $n-1$ edges, and it always exists.
(vii) Let G be a connected undirected graph with n vertices and n edges. G has a cycle.
(viii) T is a tree but $T - e$ is disconnected for every edge $e \in E(T)$.
(ix) If G has a spanning tree T it is connected.

Proof.
(i) Let u and v be two distinct vertices. Since T is connected there is a path from u to v. Suppose there are two paths from u to v. Removing common edges (and one vertex per removed edge)) leads to a cycle. But T is acyclic; thus no two paths exist.
(ii) Pick a vertex a. Since graph has at least two vertices and is connected a has a an edge. Let it lead to b. Either we reached a leaf or b points to c, another vertex through its second edge. Eventually we reach say z. Vertex z has degree at least one. If z is not a leaf either we can go on, or hit the path again leading to a cycle that is not supposed to exist in T. Thus z is a leaf. Note that if a is not a leaf we can extend the path from a to z, by expanding it through vertex a's other (second) edge. Eventually we are going to hit a leaf as well by the same argument.
(iii) By induction. (We could start induction for $n = 1$ vertex and no edges.) A tree with two vertices but include the edge connecting them to be tree. Thus number of edges is one less than number of vertices. Let it be true for trees with $n-1$ vertices or less. Consider a tree T with n vertices. It must have two leaves. Let u be such a leaf, and v its neighbor. Let T' is created by removing u and edge (u, v). T' has $n-1$ vertices and is acyclic AND connected. This is because T it came from was acyclic and the removal of u retained the acyclicity and connectedness for T'. It is a tree with $n-1$ vertices. By the induction hypothesis it has $n-2$ edges. Thus T has $(n-2)+1 = n-1$ edges for its $(n-1)+1 = n$ vertices.
(iv) By induction on edges. A graph G that is connected and has no edges it is a single vertex and it is also a tree T. Let G have $m \geq 1$ edges. If G it is a tree it is its own spanning tree. Otherwise it contains a cycle. Remove edge $e = (u, v)$ from the cycle. The remainder is still connected with same number of vertices and fewer (by one edges). So by induction it has a spanning tree which is also a spanning tree for G that has one more edge!
(v) If G is connected it has a spanning tree i.e. at least $n-1$ edges which are of G. If G has exactly $n-1$ edges, then G is a tree and the edges are those of its spanning tree.
(vi) From (iii).
(vii) G is connected so there is a spanning tree T with n vertices and $n-1$ edges. Let edge $e = (u, v)$ be the one missing from T to bring the edge sum to n edges. In T there is a path p from u to v; the addition to it of e causes a cycle. Moreover the cycle is unique. If there was another cycle call it c, and c does not contain e then c is part of T contradicting to T having no cycles (it is a tree). Otherwise c contains e and another path from u to v. If there are two paths from u to v in T this contradicts to the treeness of T. Thus c is a double count, not a second cycle.
(Another argument: Let start with something easier. An undirected and connected graph with n vertices and n

edges has a cycle. Start deleting vertices of degree one. Each vertex is deleted along with one edge. At some point we reach graph G_1. G_1 cannot have one vertex and one edge, since an edge implies at least two vertices.)
(viii) If $e = (u,v)$ is an edge of T, it creates a path from u to v. By (i) it is unique. Thus its removal from T causes no path in $T-e$ from u to v. Thus $T-e$ is disconnected.
(ix) G has a spanning tree T. Let u and v be in G. Since T is a spanning tree and connected u and v have a path in T and also in G. Repeat for every u and v in G. ■

Exercise 58.
If every vertex in a undirected graph has degree at least $(n-1)/2$ then the graph is connected.

Proof. Pick x,y with degree $(n-1)/2$ or more. Then x,y either they are adjacent and we are done. Or they have a common neighbor since their degrees add up to $((n-1)/2+(n-1)/2) = n-1$ and there are only $n-2$ other vertices. One of them must be common to x,y If there is a common neighbor, then x,y are connected through it. ■

Exercise 59.

A wolf, a goat, and a cabbage are on one bank of a river. A boatman will take them across, but can take only one at a time. The wolf and goat cannot be left together at either bank of the river without the boatman, nor can the goat and the cabbage. Model this problem with a graph, and then based on this modeling determine the shortest time required for the boatman to achieve his task, without any loss of life for its passengers. **Note: Your solution is an acceptable one if it uses graph modeling, and is based on this modeling.**

Proof. Draw a graph, solve the problem. ■

1.11 Elementary school arithmetic and merging

Exercise 60.

You are given four unsigned integers X, Y, Z, W with $2n$, $3n$, n, and $4n$ bit respectively. We want to multiply them into one integer R, where $R = X * Y * Z * W$. The cost of integer multiplication is the elementary school one: in order to multiply a 3-bit integer and a 5-bit integer the cost is assumed to be the product of the number of their digit i.e. $3 \cdot 5 = 15$ bit operations.
(1) What is a tight upper bound on the number of bit operations needed to calculater R if the schedule of multiplications is $A = X * Y$, $B = A * Z$, and $R = B * W$? Give the total number of bit operations performed in the calculation involving A, B and R.
(2) Can you beat this bit operation count by using another schedule? Justify your answer.

Proof.
(1) See below.

```
A=XmY allocates  5n for A (X is 2n, Y is 3n: Mult cost:  6n**2
B=AmZ allocates  6n for B (A is 5n, Z is  n) Mult cost:  5n**2
C=BmW allocates 10n for C (B is 6n, W is 4n) Mult cost: 24n**2
Total Mult cost is 35n**2.
```

(2) No it cannot be improved. Best schedule is any schedule. No best schedule all schedules are of the same total cost $35n^2$. Proof is by induction. Here we only have four numbers. Let the bit count be x,y,z,w for four numbers X,Y,Z,W Show by induction (on the number of input numbers) that the total cost is xy + yz+ xz + xw + yw + zw. Details omitted: do the induction part from 2 to 3 to 4 numbers. All six pair-wise bit products appear in the cost of any schedule for X,Y,Z,W!

■

Exercise 61.

You are now given four sorted sequences X, Y, Z, W with $2n$, $3n$, n, and $4n$ keys respectively. We want to merge them into one sequence containing all the keys.
(1) What is a tight upper bound on the amount of space allocated for the auxiliary arrays A, B, C if the merging schedule is $A = XmY$, $B = AmZ$, and $C = BmW$? Give the total.

(We assume there is no de-allocator nor garbage collection to recycle space once allocated. Operator m stands for merging.)

(2) Can you beat the amount of space used by the schedule in (1)? Give the schedule and give also the amount of space allocated by your proposed schedule to achieve this ?

(We assume there is no de-allocator nor garbage collection to recycle space once allocated.)

Proof.
(1) The first schedule is not worst-case optimal.

```
A=XmY allocates  5n    for A (X is 2n, Y is 3n)
B=AmZ allocates  6n    for B (A is 5n, Z is  n) Total is 21n
C=BmW allocates 10n    for C (B is 6n, W is 4n)
```

(2) There is a better schedule shown next.

```
Best schedule
    A=XmZ allocates  3n     for A (X is 2n, Z is  n)
    B=AmY allocates  6n     for B (A is 3n, Y is 3n) Total is 19n
    C=BmW allocates 10n     for C (B is 6n, W is 4n)
```

■

Exercise 62.

You are given four unsigned integers X, Y, Z, W with $2n$, $3n$, n, and $4n$ bit respectively. We want to multiply them into one integer R, where $R = X * Y * Z * W$. The cost of integer multiplication is the amount of space required to allocate for the result: that is, in order to multiply a 3-bit integer and a 5-bit integer the cost is assumed to be the sum of the number of their digit i.e. $3+5=8$, since the result can be an 8-bit integer and we must allocate space for it. (Space is not reused.)
(1) What is the cost of multiplication of X, Y, Z, W to calculate R if the schedule of multiplications is $A = X * Y$, $B = A * Z$, and $R = B * W$? Give the total cost of all operations performed in the calculation involving A, B and R.
(2) Can you beat this cost (of (1) above) by using another schedule? Justify your answer.

Proof.
(1)

```
A=X*Y allocates  5n for A (cost:5n)
B=A*Z allocates  6n for B (cost:6n)
C=B*W allocates 10n for C (cost:10n)
Total cost is 5n+6n+10n= 21n
```

(2) There is a better schedule.

```
P=X*Z allocates  3n for P (cost:3n)
Q=P*Y allocates  6n for Q (cost:6n)
R=Q*W allocates 10n for R (cost:10n)
Total cost is 3n+6n+10n= 19n better.
```

■

Exercise 63.

You are given a 15-bit, 35-bit, 40-bit, 45-bit, 55-bit numbers.
Call them A, B, C, D, E respectively. We want to multiply them together to get the answer $Z = A \cdot B \cdot C \cdot D \cdot E$. What is the best schedule to multiply them together and the associated total cost of that schedule, if by cost we mean 'space allocated during the computation (multiplication) that eventually results to computing Z', and total cost accounts for all space allocated in all multiplications involved.
For example, if you plan to multiply the 35-bit with the 55-bit integers, the space allocated for this multiplication would be $35 + 55 = 90$, and that 90 would be included into the total cost.

Proof.

```
              SpaceCost
Z1 =  A *  B     50
Z2 =  C *  D     85
Z3 = Z1 *  E    105
Z  = Z3 * Z2    190
         Total 430
```

■

Exercise 64.

You are given four sorted sequence X1,X2,X3,X4 of 2, 2n, n and n keys respectively. We would like to merge them into a sequence X of 4n+2 keys by performing the fewest number of comparisons in the worst case. Given an algorithm to do so and specify the upper bound on number of comparisons in the form an+o(n) where a is a constant, and the rest is as shown (and it uses a little oh).

Proof.

The sequence with 2 keys can be merged either first or anywhere else using binary search. To keep this simple one schedule is as follows. Merge the two sequences with n elements first X3 and X4. The output X5 contains 2n keys; this requires no more than $n+n-1=2n-1$ comparisons. Then merge the result X5 with the 2n-key sequence X2 in at most $2n+2n-1=4n-1$ additional comparisons into X6. Total comparison count is at most $(2n-1)+(4n-1)=6n-2$. For X1 do a binary search into the sequence X6 of 4n keys. This takes $O(\lg n)$ comparisons and this is o(n). (There are 2 keys in X1 and each one requires that many comparisons.) Thus the total comparison count is $6n-2+O(\lg n)=6n+o(n)$. Thus a $=$ 6. Note that in general the smaller-size sequences must be merged first: a 'greedy' principle.

■

Exercise 65.

There are given two sorted sequences of integers in arbitrary ranges $A[0..n-1]$ and $B[0..n-1]$. An integer C is also given. Give an algorithm that says YES if there exists $a \in A, b \in B$ such that $a+b=C$ and NO otherwise

Proof.

```
       Find(A,B,n)
1. if A[0] + B[n-1] == C return(YES);
2. i=0; j=n-1;
3. found = FALSE;
4. while ( (i <= n-1) && ( j >= 0)) {
5.   if ( ( A[i]+B[j]) == C) {
6.     found = YES;
7.     break;
8.   }
9.   else { if ( ( A[i]+B[j]) < C) i++;
10.         else                    j--;
11.  }
12.  if found == YES  print(YES);
13.  else             print(NO);
```

■

1.12 Sorting, merging, or binary search review

Exercise 66.

You are given n arbitrary keys. We would like to identify which key appears most often (i.e. the highest-frequency key). Give an efficient algorithm that finds the highest-frequency key of an input of n keys. If two or more keys have the highest-frequency, then you are allowed to output any single one of these keys attaining the highest-frequency.

(Example. If $\langle 10,9,4,9,4,9,10,2\rangle$ is an input key sequence, then the answer will be key 9. If $\langle Jim, Jerry, Jerry, Jim\rangle$ is the input, your answer can be `Jim`; an equally correct answer is `Jerry`).

Proof.

The idea is a variant of the duplicate key problem. Sort the keys with merge-sort. This takes $O(n\lg n)$ worst-case running time. Duplicate keys appear next to each other in the sorted sequence. The most frequent of them is the desired key, i.e. we need to count the occurrences of a duplicate key. To do so we do a linear scan left to right of the output of merge-sort (i.e. $O(n)$ additional running time for the scan) on the sorted array to determine the longest repetition of a given key. We use counters; every time we move from key x to a new key y, $x \neq y$ we reset the counters and check whether x is more frequent than the current most frequent key stored in a maxkey variable and update maxkey as needed; then we proceed to examining the frequency of y. If we go from x to $y = x$ we update the counters (increment a frequency counter by 1). Total time is $O(n\lg n)+O(n)$ i.e. $O(n\lg n)$. If pseudocode is desired, the following is a good approximation. We keep track of the frequency of the current key in count, the maximum frequency in maxfreq and the index of the key attaining it (in the sorted sequence) in maxkeyindex (i.e. the max frequency key has value $A[maxkeyindex]$). The key at $A[0]$ is used initially as the current best (line 2) and its index (0) stored in maxkeyindex along with its current frequency (1) in maxfreq and count.

In line 3 we go through the remaining $n-1$ keys (C++/C notation for arrays). If the key value for $A[i]$ is the same as before (start) we increment the counter count by one. Otherwise we have a new key with possibly higher frequency. If this is so (test of line 6) we update our relevant info to reflect this (lines 7-9) otherwise we restart with this new key (count, start are updated as in line 13). The code of lines 1-16 works fine unless the highest-frequency key is the last one, i.e $A[n-1]$. For this we need lines 17-19, to detect and correctly record that key.

```
  HighFrequency(A[0..n-1],n) //A C++ version
1. A= Sort(A,n)
2. count=1; maxkeyindex = 0;maxfreq=1; start=A[0];
3. for(i=1;i< n;i++} {
4.   if (A[i] == start) count++;
5.     else {
6.         if (count > maxfreq ) {
7.                  maxfreq=count;
8.                  maxkeyindex=i-1;
9.                  count=1;
10.                 start=A[i];
11.        }
12.        else {
13.           count=1; start=A[i];
14.        }
15     }
16. }
17. if (count >maxfreq  ) {
18. maxkeyindex=n-1; maxfreq=count;
19. }
20. print(A[maxkeyindex] has frequency maxfreq);
```

The pseudocode is also easy to analyze. Line 1 is $O(n\lg n)$. The loop of line 3 runs for $n-1$ times each time executing a constant number of operations (lines 4-15), i.e. total cost for these lines is $O(n)$. Lines 17-20 cost $O(1)$. Total cost is $O(n\lg n)+O(n)+O(1)=O(n\lg n)$. ■

Exercise 67.

(Some parts are from a previous problem.)
(a) Find the extrema of $F(x) = x\lg(x) + (n-x)\lg(n-x)$, $0 < x < n$.
Is $F(n/4) \geq n\lg n - n$? Is $F(n/2) \geq n\lg n - n$?
(b) Analyze then entropy function $f(x) = -x\lg x - (1-x)\lg(1-x)$, with $0 < x < 1$, by analysing $g(x) = x\lg x + (1-x)\lg(1-x)$.

Proof. This is primarily Calculus 1. This problem reviews topics related to finding the extrema (min or max) of functions. One needs it to prove the best case running time of QuickSort. Consider.

$$f(x) = -x\lg x - (1-x)\lg(1-x)$$

and $g(x) = -f(x)$ i.e.

$$g(x) = x\lg x + (1-x)\lg(1-x).$$

If $g(x)$ has a minimum or maximum then its negative i.e. $f(x)$ will have a maximum or a minimum respectively.
STEP 1. (Symmetry over $x = 1/2$). The first observation is that $f(x), g(x)$ are symmetric around $x = 1/2$. Thus $f(0) = f(1)$ if they were defined and in general $f(x) = f(1-x)$.
Step 2. (Behavior at $x = 0$. What is the limit $\lim_{x\to 0} g(x)$?

$$\lim_{x\to 0} g(x) = \lim_{x\to 0}(x\lg x + (1-x)\lg(1-x)) = \lim_{x\to 0}(x\lg x + (1-0)\lg(1-0)) = \lim_{x\to 0} x\lg x.$$

Step 3. (Change variables $y = 1/x$.) At this point we change variables $y = 1/x$ or $x = 1/y$. For $x \to 0$, or $1/y \to 0$, we have $y \to \infty$. Therefore

$$\lim_{x\to 0} g(x) = \lim_{x\to 0} x\lg x = \lim_{y\to\infty}(1/y)\lg(1/y) = \lim_{y\to\infty} \frac{-\lg y}{y} \overset{L'Hopital}{=} \lim_{y\to\infty} \frac{-(\lg y)'}{(y)'} = \lim_{y\to\infty} \frac{-1/y}{1} = 0^-$$

Thus the limit $\lim_{x\to 0} g(x) = 0$ from the left side (from the negative) and we represent this by having a minus as a superscript to the 0. It also suggests that $\lim_{x\to 1} g(x) = 0^-$ as well by STEP 1. Therefore for $f(x) = -g(x)$ we have $\lim_{x\to 1} f(x) = 0^+ = \lim_{x\to 0} f(x)$.

STEP 4. (First Derivative of g) What is $g'(x)$?
Simple calculations reveal that

$$g'(x) = \lg x - \lg(1-x)$$

The first derivative is 0 for $\lg x = \lg(1-x)$ i.e. for $x = 1/2$. Therefore at $x = 1/2$ we have a minimum or maximum (most likely).
STEP 5. (Second Derivative of g) To determine the type of the extreme (minimum or maximum), we examine the second derivative for $x = 1/2$ and then we obtain that

$$g''(x) = 1/x + 1/(1-x), \text{ and thus } g''(1/2) = 4 > 0.$$

Therefore $g(x)$ has a minimum at $x = 1/2$ and symmetrically $f(x)$ has a maximum. It is easy to find that $g(1/2) = -1$, and thus $f(1/2) = 1$.
STEP 6 ($F(x)$). Use $y = x/n$ in $F(x)$, $0 < x < n$. Then, $x = yn$ with $0 < y < 1$. Then

$$\begin{aligned} F(x) &= x\lg(x) + (n-x)\lg(n-x) \\ F(yn) &= yn\lg(yn) + (n-yn)\lg(n-yn) = yn\lg y + yn\lg n + n(1-y)\lg(1-y) + n(1-y)\lg n \\ &= yn\lg y + n(1-y)\lg(1-y) + n\lg n \\ H(y) &= n(y\lg y + (1-y)\lg(1-y)) + n\lg n = ng(y) + n\lg n \end{aligned}$$

Function $H(y)$ has a constant term (not dependent on y) $n\lg n$ plus $ng(y)$. Therefore it retains a minimum at $y = 1/2$ i.e. $x = yn = n/2$ and $g(1/2) = -1$ by Step 5.

STEP 7 (Conclusion). As far as $F(n/4)$ is concerned, given that $g(x)$ has a minimum for $g(1/2)$ we have that $g(1/4) \geq g(1/2)$ and thus for $x = yn$, i.e. $y = 1/4$ we have that

$$F(n/4) = n\,(g(1/4)) + n\lg n \geq ng(1/2) + n\lg n \geq n\lg n - n.$$

■

Exercise 68.

Write pseudocde for merging two sorted sequence A, B of n keys each, into sorted sequence C. Each one of the indices indA and indB gives the index of a key in A and a key in B that are compared. Index indA (resp. indB) run through A (resp. B) left to right, smallest to largest of A (resp. B). The smallest is inserted into C at position indC (step 7 or 9), and indices indA or indB (step 7 or 9), and indC (step 10) are updated accordingly. If one of the two arrays has already been copied into C, we just copy the other array into C (steps 12 and 13).

Proof. Do it! Or fill in the details!

```
  Merge(A,B,C)
1. integer indA,indB,indC;
   /* indA, indB, indC run through A,B,C */
2. indA=indB=indC=0;
3. while (indA <= n) and (indB <=n)
4.    tempA=       ;
5.    tempB=       ;
6.    if
7.       C[indC]=tempA; indA++;
8.     else
9.       C[indC]=tempB; indB++;
10.   indC++;
11.endwhile
12.   if (indA < n)
13.   if (indB < n)
```

Question: Say the architecture does not support a $\leq$ operation but only supports $>$. What can you do? ■

Exercise 69.

Rewrite InsertionSort on $A[1..n]$ to sort keys in nonincreasing order.

Proof.
Consider the code of InsertionSort. The change outlined below suffices for InsertionSort to sort keys in nonincreasing order.

```
5.  while i > 0 and A[i] < key
```

■

Exercise 70.

Replace the while loop of InsertionSort with a for loop for $A[1..n]$.

Proof.

```
for i=2 to n //Language agnostic
 key=A[i];
 for j=i-1 to 1
  if key < A[j]  A[j+1]=A[j];
  else           A[j+1]=key, break;
 if j=0 A[1]=key
 end
```

■

Exercise 71.
You are given an array A. The first $n - \lg n$ keys are sorted. The remaining keys are not. What would the running time of InsertionSort be on A?

Proof.
$O(n \lg n)$. THe first $n - \lg n$ or so iterations will take $O(1)$ each for an aggregate $O(n)$. For the last $\lg n$ iterations, InsertionSort can take $O(n)$ per key or a total of $O(n \lg n)$. Combining the two contributions an $O(n \lg n)$ bound is derived!

■

Exercise 72.

Give pseudocode for a Recursive Insertion Sort. The top level call of the wrapper function is or will be

```
InsertionSortR (A,n);
```

Proof.

```
  InsertionSortR (A,N)
  /* N is such that A[1..N] is sorted in this call */
0.     if  (N == 1) return;
1.     InsertionSortR(A,N-1);
2.     key= A[N];
       // Insert key in the sorted sequence A[1..N-1]
3.     i=N-1;
4.     while (i>0) and (A[i] > key) {
5.       A[i+1]= A[i]
6.       i=i-1
       }
8.     A[i+1]=key;
```

Proof of correctness by induction on N. Base case $N = 1$. Line 0 handles this case. Inductive step: The first $N-1$ keys have been sorted. Then the first N keys will be sorted by way of lines 2-8. Line 1 deals with the induction hypothesis (sorting of the first $N-1$ keys). Then the N-th key is inserted into $A[1..N-1]$ to form a sorted sequence $A[1..N]$. This is done through lines 2-8, the lines of Regular InsertionSort. Done. ■

Exercise 73.

You may assume that n is a power of two. You have an input that is a sorted sequence. We apply MergeSort on it. How many comparisons does MergeSort require? Justify your answer. The better the bound you provide the more points you received. A $O(n\lg(n))$ is not good.

How about on a reverse sorted sequence?

Proof.
$C(n) = n\lg n/2$. When x sorted keys get merged with y sorted keys the number of comparisons is at most $x+y-1$. When x sorted keys get merged with y sorted keys but we know all x keys are smaller than the y keys, number of comparisons is at most x. (Think of 1,2,3,4 gets merged with 5,6,7,8) At the last iteration $1,2,3,...,n/2$ and $n/2+1,...,n$ get merged. We need only n/2 comparisons that move to the output $1,...,n/2$ and then we copy the $n/2+1...n$ keys to the output. Likewise for previous iterations. Thus starting from the bottom of the divide phase Level (bottom is lgn, top is 1)

```
lgn   n/2 groups x each group:  2 sequences of 1 key  each: 1 comp
        Total  = n/2 x 1   =   n/2 comp
lgn-1 n/4 groups x each group:  2 sequences of 2 keys each: 2 comp
        Total  = n/4 x 2   =   n/2 comp
 ...      ....
 1    1   group  x each group:  2 sequences of n/2 keys each: n/2 comp
        Total  = 1   x n/2 =   n/2 comp
```

GrandTotal is $n/2 * lgn = n * lg(n)/2$. For a reverse sorted sequence: left as an exercise. ■

Exercise 74.

Show (give algorithm, explain it and analyze its running time) how to merge t sorted sequences each one of size n/t into a sorted sequence of size n in time proportional to $n\lg t$. Ignore constants. Assume n is a power of t if needed or if it is convenient.

Proof.

Merging two sorted sequences of size a and b respectively requires at most $a+b-1$ comparisons, or simply put $< a+b$ comparisons In the base case we have t sets of size n/t each. We pair the sets in $t/2$ pairs and we merge the pairs in $< n/t+n/t$ comparisons per pair. Total number of comparisons is $< (n/t+n/t)*(t/2) = n$. The number of merged sequences is $t/2$. We now pair the $t/2$ sequences of size $2n/t$ into $t/4$ pairs and merge them into sequences of size $4n/t$. The total comparison count is $< (2n/t+2n/t)*(t/4) = n$ as well. We repeat the same method until we get a sorted sequence of n keys by merging the one remaining pair of $n/2$ keys in $< n/2+n/2 = n$ comparisons. The number of merging iterations is $t/2 \to t/4 \to \ldots \to 1 = t/2^{\lg t}$ i.e. $\lg t$ iterations. Since $< n$ comparisons are performed per iteration the total comparison count is $< n\lg t$.

■

Chapter 2

Asymptotic notation

2.1 Compare vs asymptotically compare

Exercise 75.
Which of the two is asympotically larger? $f(n) = 2^{n/3}$ or $g(n) = 2^{(n/2)}$? Explain.

Proof. Take the limit $\lim_{n\to\infty} f(n)/g(n)$.

$$\lim_{n\to\infty} f(n)/g(n) = 2^{n/3}/2^{n/2} = 0$$

Since it is zero $f(n)$ is asymptotically smaller than $g(n)$ and therefore $g(n)$ is asymptotically larger than $f(n)$. ■

Exercise 76.
What's the order of growth of the sum $f(n) = \sum_{i=0}^{n} x^i$ for $x > 1$. In this case the running variable is n and x is a fixed parameter.

Proof. If $x > 1$, the sum or equivalently $f(n)$ is

$$f(n) = \sum_{i=0}^{n} x^i = (x^{n+1} - 1)/(x-1) = \Theta(x^n).$$

■

Exercise 77.

Definition 2.1 **Quick o, ω, Θ**

In order to asymptotically compare two non-negative functions $f(n)$ and $g(n)$ with the three symbols o, ω, Θ, we first find the limit $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ and then determine which case is applicable as follows. (c is a positive constant).

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \Rightarrow f(n) = o(g(n)) \\ \infty & \Rightarrow f(n) = \omega(g(n)) \\ c \neq 0 & \Rightarrow f(n) = \Theta(g(n)) \end{cases}$$

Replace ? with each one of $\omega, o, \Theta, O, \Omega$ that makes the relationship between the two sides true. If none is applicable write "none". If necessary, use the approximation $n! \approx (n/e)^n$, where $e = 2.71\ldots$. Assume that $c > 1$ is a constant. Justify your answers.

1. $\lg(n!) = ?\ (n^5)$
2. $2^n = ?\ (8^{n/3})$
3. $n^{\lg n} = ?\ (4^{\lg n * \lg n})$
4. $1 = ?\ (100)$
5. $n^4 - n^3 = ?\ (n^4)$
6. $4^n = ?\ (3^{2n+c})$

Proof.

1. $\lg n! = o(n^5)$ since by Sterling's approximation to the factorial $\lg(n!) = \lg(n/e)^n = n\lg n - n\lg e$, and thus

$$\lim \frac{\lg n!}{n^5} = \lim \frac{n\lg n}{n^5} = \lim \frac{\lg n}{n^4} = 0$$

if in the latter, we apply L'Hopital's rule once at the end. If it is o, then it will also be O.

2. This is Θ and thus also O, Ω. Why? $8^{n/3} = (8^{1/3})^n = 2^n$.

3. $n^{\lg n} = 2^{\lg n * \lg n}$. This is because $2^{\lg n * \lg n} = \left(2^{\lg n}\right)^{\lg n} = n^{\lg n}$. Also $4^{\lg n * \lg n} = 2^{2\lg n \lg n} = 2^{\lg n \lg n} \cdot 2^{\lg n \lg n} = n^{\lg n} \cdot n^{\lg n}$. Therefore $\lim n^{\lg n}/4^{\lg n * \lg n} = \lim 1/n^{\lg n} = 0$. It is o and also O.

4. $1 = \Theta(100)$ and thus O, Ω as well.

5. Same arguments and answers as 4.

6. $3^{2n+c} = 9^n \cdot 3^c$. Taking limits $\lim(4^n/9^n) \cdot 1/3^c = 0$ since $(4/9)^n \to 0$, and $3^c \to 3^c$ since c is a constant, as $n \to \infty$. ∎

2.2 Little, big, or Θ

Exercise 78.

TRUE or FALSE. Any other response is wrong.

(1) $2^{\lg n} = \Theta(n^2)$.
(2) $n/\lg n = \Theta(\lg(n!))$.
(3) $3^n = o(4^{n/2})$.
(4) $n! = \Theta((n+1)!)$.
(5) $2^{\lg n} = o(n^2)$.
(6) $n \lg n = \Theta(\lg(n!))$.
(7) $2^n = o(3^n)$.
(8) $n! = \Theta((n-1)!)$.

Proof.

(1) $2^{\lg n} = \Theta(n^2)$. **FALSE**
(2) $n/\lg n = \Theta(\lg(n!))$. **FALSE**
(3) $3^n = o(4^{n/2})$. **FALSE**
(4) $n! = \Theta((n+1)!)$. **FALSE**
(5) $2^{\lg n} = o(n^2)$. **TRUE**
(6) $n \lg n = \Theta(\lg(n!))$. **TRUE**
(7) $2^n = o(3^n)$. **TRUE**
(8) $n! = \Theta((n-1)!)$. **FALSE**

■

Exercise 79.
Replace ? with the tightest of o, ω, Θ that best describes the relationship between the two sides. If necessary, use the approximation $n! \approx (n/e)^n$, where $e = 2.71\ldots$.

(1) $\lg(n!) = ?\ (n \lg n)$.
(2) $\lg(n!) = ?\ (n^2)$.
(3) $n^n = ?\ (n!)$.
(4) $2^n = ?\ (n!)$.
(5) $2^{\lg n} = ?\ (n)$.
(6) $n/\lg n = ?\ (n)$.
(7) $2^{3\lg n} = ?\ (n^3)$.
(8) $2^n = ?\ (3^n)$.
(9) $(n!) = ?\ ((n+1)!)$.
(10) $2^{\lg n} = ?\ (n^3)$.
(11) $n \lg n = ?\ (\lg(n!))$.
(12) $4^{2n} = ?\ (16^{n/2})$.
(13) $2^{\lg n} = ?\ (n^2)$.
(14) $2^{610n} = ?\ (610^{2n})$.
(15) $2^{n+n} = ?\ (2^{610n})$.
(16) $(n!) = ?\ ((n+610)^n)$.
(17) $3^{2n} = ?\ (2^{3n})$.

Proof. From Stirling's approximation formula for the factorial $n! \approx (n/e)^n$. This answers the first two questions. Properties of the factorial answer the next two. The definition of $\lg n$ leads to the last answer.

(1) $\lg(n!) = \Theta(n \lg n)$.
(2) $\lg(n!) = o(n^2)$.
(3) $n^n = \omega(n!)$.
(4) $2^n = o(n!)$.
(5) $2^{\lg n} = \Theta(n)$.
(6) $n/\lg n = o(n)$.
(7) $2^{3\lg n} = \Theta(n^3)$.
(8) $2^n = o(3^n)$.
(9) $(n!) = o((n+1)!)$.
(10) $2^{\lg n} = o(n^3)$.
(11) $n \lg n = \Theta(\lg(n!))$.
(12) $4^{2n} = \omega(16^{n/2})$.
(13) $2^{\lg n} = o(n^2)$.
(14) $2^{610n} = \omega(610^{2n})$.
(15) $2^{n+n} = o(2^{610n})$.
(16) $(n!) = o((n+610)^n)$.
(17) $3^{2n} = \omega(2^{3n})$.

■

Exercise 80.
Replace ? with the tightest of o, ω, Θ that best describes the relationship between the two sides. If necessary, use the approximation $n! \approx (n/e)^n$, where $e = 2.71\ldots$.

(1) $610 = ?\ (102)$.
(2) $2^n = ?\ (9^{n/2})$.
(3) $2^{3\lg n} = ?\ (n^3)$.
(4) $2^{n+3} = ?\ (2^n)$.
(5) $\lg(n!) = ?\ (n)$.
(6) $2^n = ?\ ((n-1)!)$.
(7) $n^{(1/\lg n)} = ?\ (\lg n)$.
(8) $102 = ?\ (610)$.
(9) $2^{2n} = ?\ (4^{n/2})$.
(10) $2^{2\lg n} = ?\ (n^4)$.
(11) $2^{n+2} = ?\ (2^n)$.
(12) $\lg(n!) = ?\ (n)$.
(13) $2^n = ?\ (n!)$.
(14) $n^{(1/\lg n)} = ?\ (610)$.
(15) $2^n = ?\ (4^{n/2})$.
(16) $2^{2\lg n} = ?\ (n^2)$.
(17) $2^{n+1} = ?\ (2^n)$.
(18) $n \lg n = ?\ (\lg(n!))$.
(19) $\lg(n!) = ?\ (n)$.
(20) $2^n = ?\ (9^{n/2})$.
(21) $n^{(1/\lg n)} = ?\ (\lg n)$.

Proof.

(1) $610 = \Theta(102)$.
(2) $2^n = o(9^{n/2})$.
(3) $2^{3\lg n} = \Theta(n^3)$.
(4) $2^{n+3} = \Theta(2^n)$.
(5) $\lg(n!) = \omega(n)$.
(6) $2^n = o((n-1)!)$.
(7) $n^{(1/\lg n)} = o(\lg n)$.
(8) $102 = \Theta(610)$.
(9) $2^{2n} = \omega(4^{n/2})$.
(10) $2^{2\lg n} = o(n^4)$.
(11) $2^{n+2} = \Theta(2^n)$.
(12) $\lg(n!) = \omega(n)$.
(13) $2^n = o(n!)$.
(14) $n^{(1/\lg n)} = \Theta(610)$.
(15) $2^n = \Theta(4^{n/2})$.
(16) $2^{2\lg n} = \Theta(n^2)$.
(17) $2^{n+1} = \Theta(2^n)$.
(18) $n\lg n = \Theta(\lg(n!))$.
(19) $\lg(n!) = \omega(n)$.
(20) $2^n = \mathbf{o}(9^{n/2})$.
(21) $n^{(1/\lg n)} = \mathbf{o}(\lg n)$.

■

Exercise 81.
Do the same. Replace ? with the tightest of o, ω, Θ that best describes the relationship between the two sides.

(1) $n/\lg n = ?\ (n)$.
(2) $2^{2n} = ?\ ((\frac{9}{2})^n)$.
(3) $9^n = ?\ (n!)$.
(4) $2^{\lg n} = ?\ (n^2/\lg n)$.
(5) $4^{n/2} = ?\ ((8^{n/3})$.
(6) $n! = ?\ (n\lg n)$.
(7) $4^n = ?\ (3^n)$.
(8) $(n+2)! = ?\ ((n+1)!)$.
(9) $2^{n\lg n} = ?\ (n^n)$.
(10) $n^{1/\lg n} = ?\ (n)$.
(11) $\lg(n!) = ?\ (n^2)$.
(12) $2^{\lg n} = ?\ (\lg n)$.
(13) $n\lg n = ?\ (n^2)$.
(14) $n^{1/\lg n} = ?\ (1)$.
(15) $\lg(n!) = ?\ (n\lg n)$.
(16) $2^{\lg n} = ?\ (n)$.
(17) $n\lg n = ?\ (1)$.
(18) $\lg(n!) = ?\ (n\lg n)$.
(19) $3^n = ?\ (n^k)$.
(20) $n^{\lg n} = ?\ (2^{\lg n * \lg n})$.
(21) $n/\lg n = ?\ (n^3)$.
(22) $n^3 = ?\ (n^3)$.
(23) $3^{n+c} = ?\ (4^n)$.

Proof.

(1) $n/\lg n = o(n)$.
(2) $2^{2n} = o((\frac{9}{2})^n)$.
(3) $9^n = o(n!)$.
(4) $2^{\lg n} = o(n^2/\lg n)$.
(5) $4^{n/2} = \Theta((8^{n/3})$.
(6) $n! = \omega(n\lg n)$.
(7) $4^n = \omega(3^n)$.
(8) $(n+2)! = \omega((n+1)!)$.
(9) $2^{n\lg n} = \Theta(n^n)$.
(10) $n^{1/\lg n} = o(n)$.
(11) $\lg(n!) = o(n^2)$.
(12) $2^{\lg n} = \omega(\lg n)$.
(13) $n\lg n = o(n^2)$.
(14) $n^{1/\lg n} = \Theta(1)$.
(15) $\lg(n!) = \Theta(n\lg n)$.
(16) $2^{\lg n} = \Theta(n)$.
(17) $n\lg n = \omega(1)$.
(18) $\lg(n!) = \Theta(n\lg n)$.
(19) $3^n = \omega(n^k)$.
(20) $n^{\lg n} = \Theta(2^{\lg n * \lg n})$.
(21) $n/\lg n = o(n^3)$.
(22) $n^3 = \Theta(n^3)$.
(23) $3^{n+c} = o(4^n)$.

■

2.3 Asymptotically rank functions

Exercise 82.

Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $\quad n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$n,\ (n-2)!,\ 5^n,\ (1/3)^n,\ (\lg n)^{\lg n},\ 4^{\lg n},\ n\lg n,\ \lg(n!),\ 2^{\lg n},\ n^2,\ 2^{4\lg n},\ 5,\ n^{\lg\lg n},\ n!,\ 3^n,\ n^n$$

Proof. Use the results of prior problems to conclude the following.

$$n^n,\ n!,\ (n-2)!,\ 5^n,\ 3^n,\ \left\{(\lg n)^{\lg n}, n^{\lg n}\right\},\ 2^{4\lg n},\ \left\{4^{\lg n}, n^2\right\},\ \{n\lg n, \lg(n!)\},\ \left\{2^{\lg n}, n\right\},\ 5,\ (1/3)^n.$$

■

Exercise 83.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$e^n, (n+1)!, 2^{2^n}, 2^{2^{n+1}}, n!, n2^n, 2^n, (3/2)^n, n^{\lg\lg n}, (\lg n)^{\lg n}), \sqrt{2})^{\lg n}, (\lg n)!, n^2, n^3, 4^{\lg n}, n\lg n, n, \lg(n!), 2^{\lg n}.$$

Proof. Use the results of prior problems to conclude the following.

$$2^{2^{n+1}}, 2^{2^n}, (n+1)!, n!, e^n, n2^n, 2^n, (3/2)^n, (n^{\lg\lg n}, (\lg n)^{\lg n}), (\lg n)!, n^3, (n^2, 4^{\lg n}), (n\lg n, \lg(n!)), n = 2^{\lg n}, (\sqrt{2})^{\lg n},$$

■

Exercise 84.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $\quad n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$n, \quad (n-1)!, \quad (\lg n)^{\lg n}, \quad n \lg n, \quad \lg(n!), \quad 2^{\lg n}, \quad n^{\lg \lg n}, \quad n!.$$

Proof. Use the results of prior problems to conclude the following.

$$n!, \quad (n-1)!, \quad \left\{n^{\lg \lg n}, \quad (\lg n)^{\lg n}\right\}, \quad \{n \lg n, \quad \lg(n!)\}, \quad \left\{n, \quad 2^{\lg n}\right\}.$$

■

Exercise 85.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $\quad n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$2^n, \quad n^2 \lg n, \quad 3^{\lg n}, \quad n!, \quad 2^{3\lg n}, \quad \lg(n!)$$

Proof. We have that $n! = \omega(2^n)$. Also $2^n = \omega(n^k)$ for any constant $k > 0$. Since $2^{3\lg n} = (2^{\lg n})^3 = n^3$, and $n^3 = \omega(n^2 \lg n)$, and $n^2 \lg n = 2^{2\lg n} \lg n = 4^{\lg n} \lg n = \omega(3^{\lg n})$ the result follows also taking into consideration that $3^{\lg n} = \omega(n \lg n)$, and $\lg(n!) = \Theta(n \lg n)$.

$$n!, \quad 2^n, \quad 2^{3\lg n}, \quad n^2 \lg n, \quad 3^{\lg n}, \quad \lg(n!)$$

■

Exercise 86.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $\quad n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$n^3, \quad n!, \quad \lg(n!)(n-1)!, \quad 8^{\lg n} \quad n \lg n.$$

Proof. We have $n! = \omega((n-1)!)$, and $(n-1)! = \omega(n^3)$, and $\lg n! = \Theta(n \lg n)$ and we know $8^{\lg n} = 2^{3 \lg n} = n^3$, and thus $n^3 = \omega(n \lg n)$ and the ranking is as follows if we take into consideration the obvious $\omega \to \Omega$.

$$n!, \quad (n-1)!, \quad \left\{ \quad n^3, \quad 8^{\lg n} \quad \right\}, \quad \{ \quad n \lg n, \quad \lg(n!) \quad \}$$

■

Exercise 87.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$n!, \quad (n+1)!, \qquad n^2, \quad 4^{\lg n}, \quad \lg(n!), \quad n/\lg n,$$

Proof.

$$(n+1)!, \quad n!, \quad \left\{ \quad n^2, \quad 4^{\lg n} \right\} \quad , \lg(n!), \quad n/\lg n,$$

■

Exercise 88.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$4^n, \quad n!, \quad n\lg n, \quad n^4, \quad 2^{\lg n}, \quad n/\lg n.$$

Proof.

$$n!, \quad 4^n, \quad n^4, \quad n\lg n, \quad 2^{\lg n}, \quad n/\lg n.$$

■

Exercise 89.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$1/n, \quad 2^{3\lg n}, \quad (n+1)!, \quad n^3, \lg(n!), \quad n,$$

Proof.

$$(n+1)!, \quad \left\{n^3, 2^{3\lg n}\right\}, \quad \lg(n!), \quad n, \quad 1/n$$

■

Exercise 90.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$(n+1)!, \quad n!, \quad n^4, \quad \lg(n!), \quad 2^{\lg n}, \quad n/\lg n,$$

Proof. We have $(n+1)! = \omega(n!)$, $n! = \omega(n^4)$, $\lg n! = \Theta(n \lg n)$ and we know $n^4 = \omega(n \lg n)$, $2^{\lg n} = n$ by definition, and thus $n \lg n = \omega(n)$, and $2^{\lg n} = \omega(n/\lg n)$. The ranking is as follows if we take into consideration the obvious $\omega \to \Omega$.

$$(n+1)!, \quad n!, \quad n^4, \quad \lg(n!), \quad 2^{\lg n}, \quad n/\lg n,$$

■

Exercise 91.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $\quad n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$3^n, \quad 2^n, \quad 2^{3\lg n}, \quad n!, \quad n^3, \quad n\lg n$$

Proof.

$$n!, \quad 3^n, \quad 2^n, \quad \left\{n^3, \quad 2^{3\lg n} \quad \right\}, \quad n\lg n$$

■

Exercise 92.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $\quad n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$n!, 3^{\lg n}, 2^{3\lg n}, 2^n, \lg(n!), n^2 \lg n,$$

Proof.

$$n!, \quad 2^n, \quad 2^{3\lg n}, \quad n^2 \lg n, \quad 3^{\lg n}, \quad \lg(n!)$$

■

Exercise 93.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$n/\lg n, \quad n^2, \quad (n+1)!, \quad 4^{\lg n} \quad , n!, \quad \lg(n!),$$

Proof.

$$(n+1)!, \quad n!, \quad \left\{ \quad n^2, \quad 4^{\lg n} \right\} \quad , \lg(n!), \quad n/\lg n,$$

■

Exercise 94.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $\quad n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$n \lg n, \quad n^3, \quad n!, \quad (n-1)!, \quad 8^{\lg n} \quad \lg(n!)$$

Proof.

$$n!, \quad (n-1)!, \quad \left\{ \quad n^3, \quad 8^{\lg n} \quad \right\}, \quad \{ \quad n \lg n, \quad \lg(n!) \quad \}$$

■

Exercise 95.
Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, g_3, \ldots$ of the functions satisfying $g_1 = \Omega(g_2), \quad g_2 = \Omega(g_3), \quad g_3 = \Omega(g_4), \quad \ldots$. Partition your list in equivalence classes such that $g(n)$ and $h(n)$ are in the same class if and only if $h(n) = \Theta(g(n))$.
For example for functions $\lg n, n, n^2$, and $2^{\lg n}$ you could write: $n^2, \quad \{n, 2^{\lg n}\}, \quad \lg n$

$$n^n, n!, 2^n, n^{\lg\lg n}, (\lg n)^{\lg n}, 4^{\lg n}, n^2, 2^{2\lg n}, n\lg n, \lg(n!), n, 2^{\lg n}, 5(1/3)^n$$

Proof. Note that $(\lg n)^{\lg n} = n^{\lg\lg n} = 2^{\lg\lg n \cdot \lg n}$, $2^{2\lg n} = 4^{\lg n} = n^2$, $\lg(n!) = \Theta(n\lg n)$, and $2^{\lg n} = n$.

$$n^n,\ n!,\ 2^n,\ \{(\lg n)^{\lg n},\ n^{\lg\lg n}\},\ \{2^{2\lg n},\ 4^{\lg n}, n^2\}\ \{\lg(n!),\ n\lg n\},\ \{2^{\lg n},\ n\},\ 5, (1/3)^n.$$

■

2.4 The math of o, ω, and Θ

Exercise 96.

Let $a_i > 0$ be constant for all i. Then

$$p(n) = a_0 + a_1\, n + \ldots + a_d\, n^d$$

Show that $p(n) \leq 2Cn^d$, for $n > 1$, where $C = \max\{a_0, a_1, \ldots a_d\}$. Show then that $p(n) = O(n^d)$.

Proof. We have that $p(n) = a_0 + a_1\, n + \ldots + a_d\, n^d$. Let $C = \max\{a_0, a_1, \ldots a_d\}$. As at least $a_d > 0$ is positive and all a_i are constant, we conclude that $C > 0$ and constant.

For $n \geq 2$ and using the fact that $1 + n + \ldots + n^{d-1} = (n^d - 1)/(n-1)$, $(n \neq 1)$ we have that

$$\begin{aligned}
p(n) &= a_0 + a_1\, n + \ldots + a_{d-1}\, n^{d-1} + a_d\, n^d \\
&\leq C + Cn + \ldots + Cn^{d-1} + C\, n^d \\
&= C(1 + n + \ldots + n^{d-1}) + C\, n^d \\
&= C\frac{n^d - 1}{n-1} + C\, n^d \\
&\leq C\frac{n^d - 1}{1} + C\, n^d \\
&\leq C(n^d - 1) + C\, n^d \\
&\leq Cn^d + C\, n^d \\
&= 2Cn^d
\end{aligned}$$

Thus for $c_2 = 2C$ and $n_0 = 2$ we have $p(n) = O(n^d)$. ■

Exercise 97.

(a) Show that $(n-3)^2 = \Theta(n^2)$ by providing the n_0, c_1, c_2 in the definition of the Θ notation.
(b) Show that $(n+3)^2 = \Theta(n^2)$ by providing the n_0, c_1, c_2 in the definition of the Θ notation.
(c) Show that $n^5 - 18n^3 = \Theta(n^5)$ by providing the n_0, c_1, c_2 in the definition of the Θ notation.

Proof. **Part (a):** We show first that $(n-3)^2 = O(n^2)$ and then that $(n-3)^2 = \Omega(n^2)$.
Case 1. Show that $(n-3)^2 = O(n^2)$
For all $n \geq 1$ we have that

$$\begin{aligned}(n-3)^2 &= n^2 - 6n + 9 \\ &\leq n^2 + 0 \cdot n^2 + 9 \\ &\leq n^2 + 0 \cdot n^2 + 9 \cdot n^2 \\ &\leq 10n^2\end{aligned}$$

Therefore there exist constant $n_2 = 1$ and $c_2 = 10$ such that $(n-3)^2 \leq c_2 n^2$ for all $n \geq n_2$. This proves Claim 1.
Technique 1. We upper bound $n^2 \pm An \pm B$, for positive A, B as follows: $n^2 \pm An \pm B \leq (1 + A + B)n^2$.

Case 2. Show that $(n-3)^2 = \Omega(n^2)$
Technique 1 can not be used in this case. However, since $9 \geq 0$ we can eliminate 9 easily.

$$\begin{aligned}(n-3)^2 &= n^2 - 6n + 9 \\ &\geq n^2 - 6n\end{aligned}$$

The next step is non-trivial. We bound $n^2 - 6n$ from below by $n^2/2$. This is so as long as

$$\begin{aligned}n^2 - 6n &\geq n^2/2 \Leftrightarrow \\ n^2/2 &\geq 6n \Leftrightarrow \\ n/2 &\geq 6 \Leftrightarrow \\ n &\geq 12\end{aligned}$$

We can do this as long as n is not zero; this is true since for all cases we assume that at least $n \geq 1$. Therefore there exist constant $n_1 = 12$ and $c_1 = 1/2$ such that $(n-3)^2 \geq n^2/2$ for all $n \geq n_1 = 12$. This proves Claim 2. In order to show that $(n-3)^2 = \Theta(n^2)$ we need to establish c_1, c_2 and n_0 by combining the results of Claim 1 and Claim 2. Pick $c_1 = 1/2$ and $c_2 = 10$ and $n_0 = 12$.

(b) (O part first) To establish an upper bound we observe that,

$$(n+3)^2 = n^2 + 6n + 9 \leq n^2 + 6n^2 + 9n^2 = 16 \cdot n^2.$$

(Ω part next) To establish a lower bound we observe that since n is non-negative,

$$(n+3)^2 = n^2 + 6n + 9 \geq 1 \cdot n^2$$

Therefore for $c_1 = 1, c_2 = 16$ and for all $n \geq 1 = n_0$ we have that $(n+3)^2 = \Theta(n^2)$.
(c) Left as an exercise. ■

Exercise 98.

Use the definition of Θ to show that $(n-3)^2 = \Theta(n^2)$.

Proof.
We show first that $(n-3)^2 = O(n^2)$ and then that $(n-3)^2 = \Omega(n^2)$.
Case 1. Show that $(n-3)^2 = O(n^2)$ For all $n \geq 1$ we have that

$$\begin{aligned}(n-3)^2 &= n^2 - 6n + 9 \\ &\leq n^2 + 0 \cdot n^2 + 9 \\ &\leq n^2 + 0 \cdot n^2 + 9 \cdot n^2 \\ &\leq 10n^2\end{aligned}$$

Therefore there exist constant $n_2 = 1$ and $c_2 = 10$ such that $(n-3)^2 \leq c_2 n^2$ for all $n \geq n_2$. This proves the claim.
Technique 1. What we use in this proof is the fact that $n^2 \pm An \pm B$ is bounded above, for positive A, B, by $n^2 \pm An \pm B \leq (1 + A + B)n^2$.
Case 2. Show that $(n-3)^2 = \Omega(n^2)$ Technique 1 can not be used in this case. However, since $4 \geq 0$ we can eliminate 4 easily.

$$\begin{aligned}(n-3)^2 &= n^2 - 6n + 9 \\ &\geq n^2 - 6n\end{aligned}$$

The next step is non-trivial. We bound $n^2 - 6n$ from below by $n^2/2$. This is so as long as

$$\begin{aligned}n^2 - 6n &\geq n^2/2 \Leftrightarrow \\ n^2/2 &\geq 6n \Leftrightarrow \\ n/2 &\geq 6 \Leftrightarrow \\ n &\geq 12\end{aligned}$$

We can do this as long as n is not zero; this is true since for all cases we assume that at least $n \geq 1$. Therefore there exist constant $n_1 = 12$ and $c_1 = 1/2$ such that $(n-3)^2 \geq n^2/2$ for all $n \geq n_1 = 12$. This proves the claim.
In order to show that $(n-3)^2 = \Theta(n^2)$ we need to establish c_1, c_2 and n_0. c_1 and c_2 are $1/2$ and 10 respectively. $n_0 = \max(n_1, n_2) = 12$. For these values the problem is thus shown. ■

Exercise 99.

Use the definition of Θ to show that $n^5 - 25n = \Theta(n^5)$.

Proof.
We show first that $n^5 - 25 = O(n^5)$ and then that $n^5 - 25 = \Omega(n^5)$.
Case 1. Show that $n^5 - 25 = O(n^5)$.
For all $n \geq 1$ we have that

$$n^5 - 25 \quad \leq \quad n^5$$

Therefore there exist constant $n_2 = 1$ and $c_2 = 1$ such that $n^5 - 25 \leq c_2 n^5$ for all $n \geq n_2$. This proves the claim.
Technique 1. What we use in this proof is the fact that $n^2 \pm An \pm B$ is bounded above, for positive A, B, by $n^2 + An^2 + Bn^2 \leq (1 + A + B)n^2$.
Case 2. Show that $n^5 - 25 = \Omega(n^5)$
Technique 1 can not be used in this case. The next step is non-trivial. We bound $n^5 - 25$ from below by $n^5/2$. This is so as long as

$$\begin{aligned} n^5 - 25 &\geq n^5/2 \Leftrightarrow \\ n^5/2 &\geq 25 \Leftrightarrow \\ n^5 &\geq 50 \Leftrightarrow \\ n &\geq 3 \end{aligned}$$

We can do this as long as n is not zero; this is true since for all cases we assume that at least $n \geq 1$. Therefore there exist constant $n_1 = 3$ and $c_1 = 1/2$ such that $n^5 - 25 \geq n^5/2$ for all $n \geq n_1 = 3$. This proves the claim. Note that 3 is not the best possible constant. We don't need to find the best possible constant but THE EASIEST POSSIBLE! In order to show that $n^5 - 25 = \Theta(n^5)$ we need to establish c_1, c_2 and n_0. c_1 and c_2 are $1/2$ and 1 respectively. $n_0 = \max(n_1, n_2) = 3$. For these values the problem is thus shown. ■

Exercise 100.
(a) Show using the definition of Ω that $n^4 - 2n^4 = \Omega(n^4)$ by providing the constants c_1, n_0.
(b) Show using the definition of Ω that $n^3 - 5n^2 = \Omega(n^3)$.
(c) Show using the definition of Ω that $n^5 - 2n^2 = \Omega(n^5)$.
(d) Show using the definition of Ω that $n^2 - 5n = \Omega(n^2)$.

Proof.
(a) We have that $n \geq 2 \Rightarrow n^4/4 \geq n^2 \Rightarrow -2n^2 \geq -2n^4/4$. Therefore for $n \geq 2$,

$$n^4 - 2n^2 \geq n^4 - 2n^4/4 = n^4 - n^4/2 = n^4/2.$$

Thus $n^4 - 2n^2 = \Omega(n^4)$ for $c_1 = 1/2$ and $n_0 = 2$.
(b) Since for all $n \geq 10$, we have $5n^2 \leq n^3/2$ i.e $-5n^2 \geq -n^3/2$, we then have that

$$n^3 - 5n^2 \geq n^3 - (1/2)n^3 \geq n^3/2$$

Thus $c_1 = 1/2$ and $n_0 = 10$. Then $n^3 - 5n^2 \geq n^3/2$ for all $n \geq n_0$.
(c) Since for all $n \geq 2$, we have $2n^2 \leq n^5/2$ i.e $-2n^2 \geq -n^5/2$, we then have that

$$n^5 - 2n^2 \geq n^5 - (1/2)n^5 \geq n^5/2$$

Thus $c_1 = 1/2$ and $n_0 = 2$. Then $n^5 - 2n^2 \geq n^5/2$ for all $n \geq n_0$.
(d) We have that $-5n \geq -5n^2/10$ for $n \geq 10$. Therefore for $n \geq 10$,

$$n^2 - 5n \geq n^2 - 5n^2/10 = n^2 - n^2/2 = n^2/2.$$

Thus $n^2 - 5n = \Omega(n^2)$ for $c_1 = 1/2$ and $n_0 = 10$. ■

Exercise 101.
What is the order of growth of

$$1+x+x^2+\ldots+x^n$$

for constant x, and integer n ($n \geq 1$) for the following three cases? Asymptotics are for n going to ∞. Justify your answer(s) and provide simplest possible answer(s).

1. x is a constant greater than 1,
2. $x = 1$,
3. $0 < x < 1$.

Proof. $\Theta(n)$, $\Theta(1/1-x)) = \Theta(1)$, $\Theta(x^n)$, in no particular order. One has already been proven elsewhere. Match questions with answers. ■

Exercise 102.

Show using the definition that $n^2 - 3n + 2 = \Theta(n^2)$.

Proof.
Big-Oh part. Since $n^2 - 3n + 2 \leq n^2 + 2 \leq n^2 + 2n^2 = 3n^2$ for all $n \geq 1$ we conclude that there exist constant $c_2 = 3$ and $n_0 = 1$ such that $n^2 - 3n + 2 \leq 3n^2$ for all $n \geq n_0 = 1$.
Big-Omega part. Since $n^2 - 3n + 2 \geq n^2 - 3n \geq (1/2)n^2$ for $n \geq 6$ (you can verify $n^2 - 3n \geq n^2/2$ for $n \geq 6$, by solving it for n) we have that for $c_1 = 1/2$ and $n_0 = 6$ we have $n^2 - 3n + 2 \geq 1/2n^2$ for all $n \geq n_0 = 6$.
Theta part. Combining the two parts above we obtain $c_1 = 1/2$, $c_2 = 3$ and n_0 the larger of the two i.e. $n_0 = 6$. Then $1/2n^2 \leq n^2 - 3n + 2 \leq 3n^2$ for all $n \geq 6$ and thus $n^2 - 3n + 2 = \Theta(n^2)$. ■

Exercise 103.
Show using the definition that $n^5 - n^2 - 8 = \Theta(n^5)$.

Proof.
Obviously, for all $n \geq 1$, we have

$$n^5 - n^2 - 8 \leq n^5.$$

This establishes $n^5 - n^2 - 8 = O(n^5)$ for $c_2 = 1$ and $n_0 = 1$.
For the lower bound, observe that for $n \geq 2$, we have $n^3 \geq 8 \geq 4$, i.e. $-n^3 \leq -8$, $-n^5/4 \leq -n^2$. Also if $n \geq 2$, $n^2 \geq 4$ or $n^5/4 \geq n^3$.

$$0.5n^5 \leq n^5 - n^5/4 - n^5/4 \leq n^5 - n^5/4 - n^3 \leq n^5 - n^2 - 8$$

This establishes $n^5 - n^2 - 8 = \Omega(n^5)$ for $c_1 = 1/2$ and $n_0 = 2$.
Conclusion: $n^5 - n^2 - 8 = \Theta(n^5)$ for $c_1 = 1/2$, $c_2 = 1$ and $n_0 = 2$. ■

Exercise 104.
Show using the definition that $n^3 - 2n^2 + 1 = \Theta(n^3)$.

Proof.
(a) O first. $n^3 - 2n^2 + 1 \leq n^3 + 1 \leq 2n^3$. Therefore for $c_2 = 2$ and $n_0 = 1$ we have $n^3 - 2n^2 + 1 = O(n^3)$.
(b) Ω then. $n^3/2 \leq n^3 - 2n^2 \leq n^3 - 2n^2 + 1$ as long as $n \geq 4$. Thefore for $c_1 = 1/2$ and $n_0 = 4$ we have that $n^3 - 2n^2 + 1 = \Omega(n^3)$.
Combining parts (a) and (b) we have that $c_1 = 1/2$ and $c_2 = 2$ and $n_0 = 4$ are the constants to claim that $n^3 - 2n^2 + 1 = \Theta(n^3)$. ■

Exercise 105.
Show using the definition that $n^4 - n = \Theta(n^4)$

Proof.
(a) We prove the O first. Since $n^4 - n \leq 1 \cdot n^4$, we have for $c_2 = 1$ and $n_0 = 1$ that $n^4 - n = O(n^4)$.
(b) For the Ω part we observe that $0.5n^4 \leq n^4 - n$ as long as $n \geq 2$. Therefore for $c_1 = 1/2$ and $n_0 = 2$ we have $n^4 - n = \Omega(n^4)$.
The result follows for $c_1 = 1/2$, $c_2 = 1$ and $n_0 = 2$. We show that

$$(1/2)n^2 \leq^? n^2 - 3n$$

for all $n \geq 6$. This is trivially true since $n^2 - 3n \geq 0.5n^2$ is true as long as $n^2 - 0.5n^2 \geq 3n$ is true, which is true as long as $0.5n^2 \geq 3n$, which is true for all $n \geq 6$. Thus for $c_1 = 1/2$ and $n_0 = 6$ we have $(1/2)n^2 \leq n^2 - 3n \quad \forall n \geq 6 = n_0$. ■

Exercise 106.
(a) Is $3^{n+2} = O(3^n)$? Is $3^{2n} = O(3^n)$? Is $2^{n+1} = \Omega(2^n)$? Is $2^{2n} = o(5^n)$? Justify your answer.
(b) Show that $n^3 - n^2 + 2n - 2 = \Theta(n^3)$ by providing the n_0, c_1, c_2 in the definition of Θ.

Proof.
(a) $3^{n+2} \leq 9 \cdot 3^n$ and therefore $3^{n+2} = O(3^n)$ as claimed. $\lim 3^{2n}/3^n = \infty$ and thus $3^{2n} = \omega(3^n)$. $2^{n+1} \geq 2 \cdot 2^n$ and thus $2^{n+1} = \Omega(2^n)$. Also, $\lim 2^{2n}/5^n = 0$ and thus $2^{2n} = o(5^n)$.
For the O part we will show that $n^3 - n^2 + 2n - 2 = O(n^3)$ using the definitions.

$$\begin{array}{rcl} n^3 & \leq & n^3 \\ -n^2 & \leq & 0 \\ 2n & \leq & 2n^3 \\ -2 & \leq & 0 \\ --- & \leq & ---- \\ n^3 - n^2 + 2n - 2 & \leq & 3n^3 \end{array}$$

for all $n \geq 1$. Therfore for $n_1 = 1$ and $c_2 = 3$ the upper bound is derived. Note that the upper bound yields its own n_0 that we call n_1.

For the Ω part, we have that

$$\begin{array}{rcl} n^3 & \geq & n^3 \\ -n^2 & \geq & -(1/4)n^3 \\ 2n & \geq & 0 \\ -2 & \geq & -(1/4)n^3 \\ --- & \geq & ---- \\ n^3 - n^2 + 2n - 2 & \geq & n^3/2 \end{array}$$

The first inequality is immediately true. The second inequality is true provided that $4 \leq n$. The third inequality is trivially true. The fourth inequality is true for $2 \leq n$. Combining the four inequalities by adding them up together we obtain that $n^3 - n^2 + 2n - 2 \geq n^3/2$ for $n \geq \max\{2,4\}$. Therfore $c_1 = 1/2$ and $n_2 = 4$. Note that the lower bound yields its own n_0 different from that of the upper bound that we call n_2.

Therefore the constant of the definition are $c_1 = 1/2, c_2 = 3$ and $n_0 = \max\{n_1, n_2\} = 4$ (the n_0 of the definition is the maximum of the two n_1, n_2 found while deriving the upper and lower bounds). ■

Exercise 107.
Show that $n^5 - 18 \cdot n^3 = \Theta(n^5)$ by providing the n_0, c_1, c_2 in the definition of Θ.

Proof.
We first show that $n^5 - 18n^3 = O(n^5)$, i.e. find constant c_2 and n_2 such that $n^5 - 18n^3 \leq c_2 n^5$ for all $n \geq n_2$. We observe that

$$n^5 - 18n^3 \leq 1 \cdot n^5$$

i.e. for $c_2 = 1$ and $n_2 = 1$, $n^5 - 18n^3 \leq c_2 n^5$ for all $n \geq n_2$.Similarly, we find constants c_1 and n_1 such that $c_1 \cdot n^5 \leq n^5 - 18n^3$ for all $n \geq n_1$. We observe that

$$n^5/2 \leq n^5 - 18n^3$$

provided that

$$\begin{aligned} 18n^3 &\leq n^5/2 \\ 36 &\leq n^2 \\ 6 &\leq n \end{aligned}$$

Therefore for $n_1 = 6$ and $c_1 = 1/2$ we have shown that $n^5 - 18n^3 \geq n^5/2$, i.e. $n^5 - 18n^3 = \Omega(n^5)$. By taking $c_1 = 1/2$, $c_2 = 1$ and $n_0 = \max\{n_1, n_2\} = \max\{6, 1\} = 6$, we obtain that $n^5 - 18n^3 = \Theta(n^5)$. ■

Exercise 108.
TRUE or FALSE? Do not provide any justification.

(1) $2^{\lg n} = o(n^2)$.
(2) $2^n = o(3^n)$.
(3) $2^n = \omega(3^{n/2})$.
(4) $n = \Theta(n \lg^* n)$.
(5) $\lg^* \lg n = \Theta(\lg^* n)$.
(6) $2^{3 \lg n} = o(n^4)$.
(7) $3^n = \omega(2^n)$.
(8) $2^n = \omega(2^{n/2})$.
(9) $n / \lg^* n = \Theta(n)$.
(10) $2^{\lg n} = \Theta(n^2)$.
(11) $n / \lg n = \Theta(\lg(n!))$.
(12) $3^n = o(4^{n/2})$.
(13) $n! = \Theta((n+1)!)$.
(14) $2^{\lg n} = o(n^2)$.
(15) $n \lg n = \Theta(\lg(n!))$.
(16) $2^n = o(3^n)$.
(17) $n! = \Theta((n-1)!)$.

Proof. Practice makes perfect. ■

Exercise 109.

The Fast vs the (furiously) Good. For some problem P we have implemented two algorithms that solve P. An algorithm `Good` runs in time $g(n)$ and an algorithm `Fast` runs in time $f(n)$, where n is problem size of P. Answer the following question Q for each one of the following three cases C1, C2 and C3. Justify your answer.

Q. Is *Fast* faster than *Good* for all $n \geq n_0$, for some constant n_0?

C1. $g(n) = \Omega(f(n) \lg n)$.

C2. $g(n) = \Theta(f(n) \lg n)$.

C3. $g(n) = O(f(n) \lg n)$.

Proof. Asymptotic notation roundup: Think!

■

Exercise 110.
True or False? Explain.
f, g are asymptotically positive functions.
(a) If $f(n) = O(g(n)$ then $g(n) = O(f(n)$.
(b) If $f(n) + g(n) = O(\min(f(n), g(n)$.
(c) If $f(n) = O(g(n)$ then $2^{f(n)} = O(2^{g(n)})$.
(d) $f(n) = O(f(n)^2)$.
(e) $f(n) = \Theta(f(n/2))$.

Proof.
Use $f(n) = 1$ and $g(n) = n$ for the first two. Use $f(n) = 2\lg n$ and $g(n) = \lg n$ for the third. Use $f(n) = 1/n$ for the fourth. Use $f(n) = 2^n$ for the fifth.

■

Exercise 111.
Let $a > 0, b > 0$ be constant. Show that $(n+a)^b = \Theta(n^b)$. In other words find c_1, c_2, n_0 of the definition of Θ.

Proof.
(a) $n+a \leq n+n \leq 2n$ and therefore $(n+a)^b \leq 2^b n^b$. The former is true as long as $n \geq a$. From the latter we isolate $c_2 = 2^b$, that is constant since b is constant.
(b) $(n+a) \geq n/2$ since $a > 0$ and $n > 0$. Thus $(n+a)^b \geq (1/2)^b n^b$. From this $c_1 = (1/2)^b$, is a positive constant. Resul follows from (a), (b) with $n \geq n_0 = 1$.

■

Chapter 3

Recurrrence relations

3.1 Recurrences and the master method

Exercise 112.

Give a **TRUE** or **FALSE** for each one of the statements below. Any answer other than a full TRUE or FALSE will be considered wrong.
(a) The asymptotic solution of $T(n) = 16T(n/2) + n^3$ is $T(n) = \Theta(n^3)$.
(b) The asymptotic solution of $T(n) = 8T(n/2) + n^3 \lg n$ is $T(n) = \Theta(n^3 \lg n)$.
(c) The asymptotic solution of $T(n) = 2T(n/2) + \lg n$ is $T(n) = \Theta(n \lg n)$
(d) Let $f(n) > 0$ be a positive function for all $n \geq 0$. The following is true for every f: $f(n) = \Theta(f(n/2))$.
(e) The solution of $T(n) = T(n-1) + 1$ with $T(1)$ constant is $T(n) = \Theta(n^2)$.
(f) The solution of $T(n) = T(n-1) + n$ with $T(1)$ constant is $T(n) = \Theta(n^2)$.
(g) The asymptotic solution of $T(n) = T(n/2) + \lg n$ is $T(n) = \Theta(n)$.
(h) The asymptotic solution of $T(n) = 3T(n/2) + T(2n/4) + n^2$ is $T(n) = \Theta(n^2)$.
(i) The asymptotic solution of $T(n) = 32T(n/2) + n^4$, is $T(n) = \Theta(n^4)$.
(j) The asymptotic solution of $T(n) = T(n/2) + n$, is $T(n) = \Theta(n)$.
(k) The asymptotic solution of $T(n) = 4T(n/2) + n \lg n$ is $T(n) = \Theta(n(\lg n)^2)$.
(l) The asymptotic solution of $T(n) = 8T(n/8) + n$, is $T(n) = \Theta(n)$.
(m) The asymptotic solution of $T(n) = T(n/8) + n$, is $T(n) = \Theta(n)$.
(n) The asymptotic solution of $T(n) = 8T(n/2) + n^2$, is $T(n) = \Theta(n^2)$.
(o) The asymptotic solution of $T(n) = T(n/2) + n$, is $T(n) = \Theta(n)$.
(p) The asymptotic solution of $T(n) = 4T(n/2) + n^2$, is $T(n) = \Theta(n^2)$.
(q) The asymptotic solution of $T(n) = 16T(n/2) + n^5$, is $T(n) = \Theta(n^4)$.
(r) The asymptotic solution of $T(n) = 2T(n/4) + \Theta((n-1)!)$ is $T(n) = \Theta(n!)$. (factorial involved).
(s) The asymptotic solution of $T(n) = T(n/5) + T(7n/10) + n$ is $T(n) = \Theta(n)$.
(t) The asymptotic solution of $T(n) = 32T(n/2) + n^4$, is $T(n) = \Theta(n^4)$.
(u) The asymptotic solution of $T(n) = 16T(n/4) + n$ is $T(n) = \Theta(n^2)$.

Proof. FALSE,FALSE,FALSE, FALSE
FALSE it is $\Theta(n)$, TRUE since $T(n) = T(n-1) + n = T(n-2) + (n-1) + n = \ldots = T(1) + 2 + 3 + \ldots + (n-1) + n$, FALSE; Case 2 of the master method shows that it is $T(n) = \Theta(\lg^2 n)$,
FALSE; Case 2 of the master method show that it is $T(n) = \Theta(n^2 \lg n)$. Note also that $3T(n/2) + T(2n/4) = 4T(n/2)$, FALSE; Case 1 of the master method case shows that it is $T(n) = \Theta(n^5)$, TRUE; Case 3 of the master method case,
FALSE; Case 1 implies $T(n) = \Theta(n^2)$, FALSE; Case 2 of the master method gives $T(n) = \Theta(n \lg n)$, TRUE; Case 3 of the master method gives $T(n) = \Theta(n)$, FALSE; Case 1 of the master method gives $T(n) = \Theta(n^3)$,

TRUE; Case 3 of the master method, FALSE; Case 2 of the master method gives $T(n) = \Theta(n^2 \lg n)$, FALSE; Case 3 of the master method gives $T(n) = \Theta(n^5)$, Think and do it, TRUE, FALSE, TRUE.

■

Exercise 113.

Solve the following recurrences. You may assume $T(1) = \Theta(1)$, where necessary. Make your bounds as tight as possible. Use asymptotic notation to express your answers. Justify your answers.

a. $T(n) = 4T(n/2) + 3n^2$
b. $T(n) = 32T(n/4) + 3n^2$
c. $T(n) = 4T(n/2) + n^3$
d. $T(n) = 9T(n/3) + n.$
e. $T(n) = 25T(n/5) + n^2.$

Proof. One can use the master method for all three of them.

a. $T(n) = 4T(n/2) + 3n^2$. It has $a = 4 \geq 1$, $b = 2 > 1$, $f(n) = 3n^2$. $n^{\log_b a} = n^{\lg 4} = n^2$. Obviously $n^2 = f(n) = \Theta(n^{\log_b a})$. Case 2 of the Master method applies and thus $T(n) = \Theta(n^2 \lg n)$.

b. $T(n) = 32T(n/4) + 3n^2$. Note that $32 \cdot 32 = 1024 = 2^{10}$, $4^5 = 1024$ i.e. $4^{5/2} = 1024^{1/2} = 32$. It has $a = 32 \geq 1$, $b = 4 > 1$, $f(n) = 3n^2$. $n^{\log_b a} = n^{\lg_4 32} = n^{5/2}$. Obviously $n^2 = f(n) = O(n^{\log_b a - \varepsilon}) = O(n^2)$, by choosing $\varepsilon = 1/2 > 0$. Case 1 of the Master method applies and thus $T(n) = \Theta(n^{5/2})$.

c. $T(n) = 4T(n/2) + n^3$. It has $a = 4 \geq 1$, $b = 2 > 1$, $f(n) = n^3$. $n^{\log_b a} = n^{\lg 4} = n^2$. Obviously $n^3 = f(n) = \Omega(n^{\log_b a + \varepsilon}) = O(n^3)$, by choosing $\varepsilon = 1 > 0$. Case 3 of the Master method applies since also $af(n/b) = 4(n/2)^3 = n^3/2 \leq cn^3 = (1/2)n^3$, by choosing $c = 1/2 < 1$, and thus $T(n) = \Theta(n^3)$.

d. We use the master method. $a = 9 \geq 1$, $b = 3 > 1$, $f(n) = n$, and $\log_b a = 2$. Then $n^{\lg_b a} = n^2$. As $f(n) = n = O(n^{\lg_b a - 1}) = O(n)$, case one of the master method applies, and thus $T(n) = \Theta(n^2)$.

e. We use the master method. $a = 25 \geq 1$, $b = 5 > 1$, $f(n) = n^2$, and $\log_b a = 2$. Then $n^{\lg_b a} = n^2$. As $f(n) = n^2 = \Theta(n^{\lg_b a}) = \Theta(n^2)$, case two of the master method applies, and thus $T(n) = \Theta(n^2 \lg n)$. ∎

Exercise 114.

Solve the following recurrence relations by providing asymptotically tight bounds. You only need to provide the bound, intermediate derivations are not required. If no boundary case is given, the choice is yours. You may assume that $T(n)$ is positive and monotonically increasing, if you need to do so.

$$\begin{aligned} T(n) &= 8T(n/2)+n^3. \\ T(n) &= 8T(n/2)+n^8. \\ T(n) &= 4T(n/2)+n^2. \\ T(n) &= 4T(n/2)+n^3. \end{aligned}$$

Proof.
(1) We have $a = 8 \geq 1$, $b = 2 > 1$ and $n^{\lg_b a} = n^3$. Since $f(n) = n^3$, MasterMethod.case2 gives $T(n) = \Theta(n^3 \lg n)$.
(2) The a, b are as before and $f(n) = n^8$. Master Method.case.3 applies since $n^8 = f(n) = \Omega(n^{3+1})$ if we choose $\varepsilon = 1 > 0$. Note that $af(n/b) = 8f(n/2) = 8n^8/2^8 = n^8/32 \leq 0.5n^8 = 0.5f(n) = cf(n)$. Thus Case 3 applies and $T(n) = \Theta(n^8)$.
(3) We have $a = 4 \geq 1$, $b = 2 > 1$ and $n^{\lg_b a} = n^2$. Since $f(n) = n^2$, MasterMethod.Case.2 gives $T(n) = \Theta(n^2 \lg n)$.
(4) The a, b are as before but $f(n) = n^3$. Case 3 of the Master Method applies since $n^3 = f(n) = \Omega(n^{2+1})$ if we choose $\varepsilon = 1 > 0$. Note that $af(n/b) = 4f(n/2) = 4n^3/2^3 = n^3/2 \leq 0.5n^3 = 0.5f(n) = cf(n)$. Thus Case 3 applies and $T(n) = \Theta(n^3)$. ■

Exercise 115.

Solve the following recurrence relations by providing asymptotically tight bounds. You only need to provide the bound, intermediate derivations are not required. If no boundary case is given, the choice is yours. You may assume that $T(n)$ is positive and monotonically increasing, if you need to do so.

$$
\begin{array}{lrcl}
(1) & T(n) & = & 8T(n/2)+n^4. \\
(2) & T(n) & = & 8T(n/2)+n/\lg n. \\
(3) & T(n) & = & 8T(n/2)+n^3. \\
(4) & T(n) & = & 16T(n/4)+n^3. \\
(5) & T(n) & = & 4T(n/4)+n\lg n. \\
(6) & T(n) & = & 4T(n/4)+\lg n. \\
(7) & T(n) & = & 16T(n/4)+n^2\lg n. \\
(8) & T(n) & = & 16T(n/4)+n\lg n. \\
(9) & T(n) & = & 4T(n/4)+n^2\lg n. \\
(10) & T(n) & = & 9T(n/3)+\sqrt{n}. \\
(11) & T(n) & = & T(n/3)+n^3\lg n. \\
(12) & T(n) & = & 9T(n/3)+n^4. \\
(13) & T(n) & = & 8T(n/2)+n^4. \\
(14) & T(n) & = & 4T(n/2)+n^2. \\
(15) & T(n) & = & T(n-1)+1. \\
(16) & T(n) & = & 8T(n/2)+n^3. \\
(17) & T(n) & = & 16T(n/2)+n^5. \\
(18) & T(n) & = & T(n-1)+n.
\end{array}
$$

Proof. Practice makes perfect! You practice!
(13) $T(n)=8T(n/2)+n^4$. By case 3 of MM, $T(n)=\Theta(n^4)$.
(14) $T(n)=4T(n/2)+n^2$. By case 2 of MM, $T(n)=\Theta(n^2\lg n)$.
(15) $T(n)=T(n-1)+1$. Iteratively, $T(n)=1+T(n-1)=1+1+T(n-2)=2+T(n-2)=\ldots=i+T(n-i)=\ldots=n-1+T(1)$. We choose conveniently $T(1)=1$ to get $T(n)=n$. It's tighter than $T(n)=\Theta(n)$.
(16) $T(n)=8T(n/2)+n^3$. By case 2 of MM, $T(n)=\Theta(n^3\lg n)$.
(17) $T(n)=16T(n/2)+n^5$. By case 3 of MM, $T(n)=\Theta(n^5)$.
(18) $T(n)=T(n-1)+n$. Iteratively, $T(n)=n+T(n-1)=n+(n-1)+T(n-2)=\ldots=n+(n-1)+(n-2)+\ldots+2+T(1)=n(n+1)/2$. We chose conveniently $T(1)=1$ to get $T(n)=n(n+1)/2$ in the last inequality. It's tighter than $T(n)=\Theta(n^2)$. ■

Exercise 116.

Solve the following recurrence relations by providing asymptotically tight bounds. You only need to provide the bound, intermediate derivations are not required. If no boundary case is given, the choice is yours. You may assume that $T(n)$ is positive and monotonically increasing, if you need to do so.

(1) $T(n) = 8T(n/2)+n^4 \qquad T(n)=\Theta(n^4)$
(2) $T(n) = 8T(n/2)+n/\lg n \qquad T(n)=\Theta(n^3)$
(3) $T(n) = 8T(n/2)+n^3 \qquad T(n)=\Theta(n^3\lg n)$
(4) $T(n) = 16T(n/4)+n/\lg n. \qquad T(n)=\Theta(n^2)$
(5) $T(n) = 16T(n/4)+n^4. \qquad T(n)=\Theta(n^4)$
(6) $T(n) = T(n/2)+T(2n/4)+n. \qquad T(n)=\Theta(n\lg n)$
(7) $T(n) = 9T(n/3)+\sqrt{n}. \quad T(n)=\Theta(n^2).$
(8) $T(n) = T(n/3)+n^3\lg n. \quad T(n)=\Theta(n^3\lg n).$
(9) $T(n) = 9T(n/3)+n^4. \quad T(n)=\Theta(n^4).$
(10) $T(n) = 8T(n/2)+n^4 \quad T(n)=\Theta(n^4)$
(11) $T(n) = 8T(n/2)+n/\lg n \quad T(n)=\Theta(n^3)$
(12) $T(n) = 8T(n/2)+n^3 \quad T(n)=\Theta(n^3\lg n)$
(13) $T(n) = 9T(n/3)+n^3.$
(14) $T(n) = 9T(n/3)+n^2\lg n.$
(15) $T(n) = 9T(n/3)+n.$
(16) $T(n) = 25T(n/5)+n.$
(17) $T(n) = 25T(n/5)+n^2\lg n.$
(18) $T(n) = 25T(n/5)+n^3.$
(19) $T(n) = 8T(n/2)+n^4.$
(20) $T(n) = T(n-1)+1.$
(21) $T(n) = 4T(n/2)+n^2\lg n.$
(22) $T(n) = 8T(n/2)+n\lg n.$
(23) $T(n) = 2T(n/2)+n.$
(24) $T(n) = 4T(n/2)+n^3.$

Proof. Practice makes perfect! You practice!

(13) $T(n) = 9T(n/3)+n^3 \to T(n)=\Theta(n^3).$
(14) $T(n) = 9T(n/3)+n^2\lg n \to T(n)=\Theta(n^2(\lg n)^2).$
(15) $T(n) = 9T(n/3)+n. \to T(n)=\Theta(n^2).$
(16) $T(n) = 25T(n/5)+n. \to T(n)=\Theta(n^2).$
(17) $T(n) = 25T(n/5)+n^2\lg n. \to T(n)=\Theta(n^2(\lg n)^2).$
(18) $T(n) = 25T(n/5)+n^3. \to T(n)=\Theta(n^3).$
(19) $T(n) = 8T(n/2)+n^4. \to T(n)=\Theta(n^4).$
(20) $T(n) = T(n-1)+1. \to T(n)=\Theta(n).$
(21) $T(n) = 4T(n/2)+n^2\lg n. \to T(n)=\Theta(n^2(\lg n)^2).$

(22) $T(n) = 8T(n/2)+n\lg n.$ **Case 1 of MM, gives** $T(n) = \Theta(n^3)$
(23) $T(n) = 2T(n/2)+n.$ **Case 2 of MM, gives** $T(n) = \Theta(n\lg n)$
(24) $T(n) = 4T(n/2)+n^3.$ **Case 3 of MM, gives** $T(n) = \Theta(n^3)$

■

Exercise 117.
Solve the following recurrence relations by providing asymptotically tight bounds. You only need to provide the bound, intermediate derivations are not required. If no boundary case is given, the choice is yours. You may assume that $T(n)$ is positive and monotonically increasing, if you need to do so.

$$\begin{aligned}
T(n) &= T(n/4)+n.\\
T(n) &= 8T(n/4)+n!.\\
T(n) &= 2T(n/2)+\lg n.\\
T(n) &= 2T(n/2)+T(n/4)+n^2.\\
T(n) &= T(n/2)+n.\\
T(n) &= T(n/2)+\lg n.\\
T(n) &= 2T(n/2)+T(n/4)+n^3.
\end{aligned}$$

Proof. Practice makes perfect! You practice! ■

Exercise 118.

Solve the following recurrence relations by providing asymptotically tight bounds. If no boundary case is given, the choice (of the constants) is yours or you can use $T(1) = 1$. You may assume that $T(n)$ is positive and monotonically increasing, if you need to do so.

$$\begin{aligned}
T(n) &= 4T(n/2)+n^4.\\
T(n) &= T(n/2)+\sqrt{n}.\\
T(n) &= 2T(n/2)+8n\\
T(n) &= 8T(n/2)+n^4.\\
T(n) &= 4T(n/2)+n^2.\\
T(n) &= T(n-1)+n\\
T(n) &= 8T(n/2)+n^3.\\
T(n) &= T(n-2)+n.\\
T(n) &= 16T(n/2)+n^5.
\end{aligned}$$

Proof.
(a) Case 3 of the master method gives $T(n) = \Theta(n^4)$. $a = 4$, $b = 2$, $n^{\lg_b a} = n^2$, $n^4 = \Omega(n^{2+1})$ and $af(n/b) = 4f(n/2) = 4n^4/16 \leq 1/2n^4$ for $c = 1/2$.
(b) Case 3 of the master method gives $T(n) = \Theta(\sqrt{n})$. $a = 1$, $b = 2$, $n^{\lg_b a} = n^0 = 1$, $\sqrt{n} = \Omega(n^{0+1/2})$ and $af(n/b) = 1f(n/2) = \sqrt{n}/\sqrt{2} \leq 0.9\sqrt{n}$, $c = 0.9$.
(c) Case 2 of the master method gives $T(n) = \Theta(n\lg n)$. $a = b = 2$ and $f(n) = n^{\lg_b a} = n = \Theta(n^{\lg_b a}) = \Theta(n)$.
(d) For $T(n) = 8T(n/2)+n^4$, master method case (3) gives $T(n) = \Theta(n^4)$.
(e) For $T(n) = 4T(n/2)+n^2$, master method case (2) gives $T(n) = \Theta(n^2 \lg n)$.
(f) For $T(n) = T(n-1)+n$, the iteration method gives $T(n) = T(n-1)+n = T(n-2)+(n-1)+n = \ldots = T(1)+2+\ldots+(n-1)+n$. Choose $T(1) = 1$ (our choice), and $T(n) = n(n+1)/2 = \Theta(n^2)$.
(g) For $T(n) = 8T(n/2)+n^3$, master method case (2) gives $T(n) = \Theta(n^3 \lg n)$.
(h) For $T(n) = 16T(n/2)+n^5$, master method case (3) gives $T(n) = \Theta(n^5)$.
(i) For $T(n) = T(n-2)+n$, the iteration method gives $T(n) = T(n-2)+n = T(n-4)+(n-2)+n = \ldots = T(1)\text{or}T(2)+\ldots+(n-4)+(n-2)+n$. Choose $T(1) = 1, T(2) = 2$ (our choice), and $T(n) = \Theta(n^2)$. Alternatively, use the substitution method to guess $T(n) = n^2/4+n/2$. ■

3.2 Recurrences and the iteration method

It is also known as the repeated substitution method.

Exercise 119.

Let n be a power of two. Solve exactly the following recurrence.
$T(n) = T(n/2) + n$, where $T(1) = 2$.

Proof. We could use the master method (case three) to derive that $T(n) = \Theta(n)$ (using $\varepsilon = 1/2$ and $c = 1/2$), if boundary condition $T(1) = 2$ was not explicitly stated. Since it is, we use the recursion tree/iteration method, in fact the iteration method.

$$\begin{aligned}
T(n) &= T(n/2) + n \\
&= T(n/4) + n/2 + n \\
&= T(n/8) + n/4 + n/2 + n \\
&= T(n/2^i) + n/2^{i-1} + \ldots + n/4 + n/2 + n \\
&= T(n/2^{\lg n}) + n/(n/2) + \ldots + n/2 + n \\
&= T(1) + 2 + 4 + 8 + \ldots + n/2 + n \\
&= 2 + 2 + 4 + 8 + \ldots + n/2 + n \\
&= 1 + (1 + 2 + 4 + 8 + \ldots + n/2 + n) \\
&= 1 + (2n - 1) = 2n
\end{aligned}$$

We used the fact that $1 + 2 + 4 + \ldots + n = 2n - 1$.

■

Exercise 120.
Solve the recurrence

$$T(n) = T(n-1) + 2n, \quad \text{where} \quad T(1) = 2.$$

Proof.
We use the iteration method.

$$\begin{aligned}
T(n) &= T(n-1) + 2n = (T(n-2) + 2(n-1)) + 2n \\
&= T(n-2) + 2(n-1) + 2n = (T(n-3) + 2(n-2)) + 2(n-1) + 2n \\
&= T(n-3) + 2(n-2) + 2(n-1) + 2n \\
&= \dots \\
&= T(i) + 2(i+1) + \dots + 2(n-1) + 2n \\
&= T(1) + 2(1+1) + \dots + 2(n-1) + 2n \\
&= 2 \cdot 1 + 2(1+1) + \dots + 2(n-1) + 2n \\
&= 2(1 + 2 + \dots + n) \\
&= 2n(n+1)/2 = n(n+1)
\end{aligned}$$

■

Exercise 121.

Solve the following recurrence exactly. You may assume n is a power of two.

$$T(n) = 2T(n/2) + 6n,$$

where $T(4) = 12$.

Proof.

$$\begin{aligned} T(n) &= 2T(n/2) + 6n \\ &= 2(2T(n/2^2) + 6n/2) + 6n = 2^2 T(n/2^2) + 2(6n) \\ &= 2^3 T(n/2^3) + 3(6n) \\ &\cdots \\ &= 2^i T(n/2^i) + i(6n) \end{aligned}$$

Substituting $n/2^i = 4$ and solving for i we get $i = \lg n - 2$. Then $T(n/2^i) = T(4)$ and $T(4) = 12$. Substituting $\lg n - 2$ for i in.

$$\begin{aligned} T(n) &= 2^i T(n/2^i) + i(6n) \\ &= (n/4)T(4) + (\lg n - 2)6n \\ &= 12n/4 + 6n\lg n - 12n \\ &= 6n\lg n - 9n \end{aligned}$$

The result follows. ■

Exercise 122.
Solve exactly the following recurrence. You may assume that n is a power of two.

$$T(n) = 2T(n/2)+18n, \qquad T(8)=6.$$

Do the same for

$$T(n) = 2T(n/2)+6n, \qquad T(8)=18.$$

Proof.

$$\begin{aligned} T(n) &= 2T(n/2)+18n, \qquad T(8)=6. \\ &= 2^2T(n/2^2)+2\cdot 18n \\ &= 2^iT(n/2^i)+i\cdot 18n \\ &= 2^{\lg n-3}T(n/2^{\lg n-3})+(\lg n-3)\cdot 18n \\ &= (n/8)T(8)+18n\lg n-54n \\ &= (n/8)6+18n\lg n-54n \\ &= 18n\lg n-53.25n \end{aligned}$$

We used in the fourth equality the facts that $T(8)=6$ and $2^{\lg n-3}=n/8$.

$$\begin{aligned} T(n) &= 2T(n/2)+6n, \qquad T(8)=18. \\ &= 2^2T(n/2^2)+2\cdot 6n \\ &= 2^iT(n/2^i)+i\cdot 6n \\ &= 2^{\lg n-3}T(n/2^{\lg n-3})+(\lg n-3)\cdot 6n \\ &= (n/8)T(8)+6n\lg n-18n \\ &= (n/8)18+6n\lg n-18n \\ &= 6n\lg n-15.75n \end{aligned}$$

We used in the fourth equality the facts that $T(8)=18$ and $2^{\lg n-3}=n/8$. ■

Exercise 123.
Solve exactly the following recurrence. You may assume that n is a power of two.

$$T(n) = 2T(n/2)+4n, \qquad T(8)=8.$$

Do the same for

$$T(n) = 2T(n/2)+6n, \qquad T(2)=8.$$

Proof.

$$\begin{aligned}
T(n) &= 2T(n/2)+4n, \qquad T(8)=8. \\
&= 2^2T(n/2^2)+2\cdot 4n = \dots \\
&= 2^iT(n/2^i)+i\cdot 4n \\
&= 2^{\lg n-3}T(n/2^{\lg n-3})+(\lg n-3)\cdot 4n \\
&= (n/8)T(8)+4n\lg n-12n \\
&= 4n\lg n-11n
\end{aligned}$$

$$\begin{aligned}
T(n) &= 2T(n/2)+6n, \qquad T(2)=8. \\
&= 2^2T(n/2^2)+2\cdot 6n = \dots \\
&= 2^iT(n/2^i)+i\cdot 6n \\
&= 2^{\lg n-1}T(n/2^{\lg n-1})+(\lg n-1)\cdot 6n \\
&= (n/2)T(2)+6n\lg n-6n \\
&= 6n\lg n-2n
\end{aligned}$$

■

Exercise 124.
Solve exactly the following recurrence. You may assume that n is a power of two.

$$T(n) = 2T(n/2)+12n, \qquad T(4)=6.$$

Do the same for

$$T(n) = 2T(n/2)+4n, \qquad T(8)=16.$$

Proof.

$$\begin{aligned} T(n) &= 2T(n/2)+12n, \qquad T(4)=6. \\ &= 2^2T(n/2^2)+2\cdot 12n \\ &= 2^iT(n/2^i)+i\cdot 12n \\ &= 2^{\lg n-2}T(n/2^{\lg n-2})+(\lg n-2)\cdot 12n \\ &= (n/4)T(4)+12n\lg n-24n \\ &= 12n\lg n-22.5n \end{aligned}$$

$$\begin{aligned} T(n) &= 2T(n/2)+4n, \qquad T(8)=16. \\ &= 2T(n/2)+4n = 2^2T(n/2^2)+2\cdot 4n \\ &= 2^iT(n/2^i)+i\cdot 4n = 2^{\lg n-3}T(n/2^{\lg n-3})+(\lg n-3)\cdot 4n \\ &= (n/8)T(8)+4n\lg n-12n = \\ &= 4n\lg n-10n \end{aligned}$$

■

Exercise 125.

Solve exactly using the iteration method the following recurrence. You may assume that n is a power of 3.

$$T(n) = 3T(n/3) + \lg_3 n, \text{ where } T(1) = 1.$$

Proof. We have that

$$\sum_{k=0}^{m-1} kx^k = \frac{mx^m(x-1) - x^{m+1} + x}{(x-1)^2} \; and \; for \; x = 3 \; \rightarrow \sum_{k=0}^{m-1} k3^k = (2m3^m - 3^{m+1} + 3)/4$$

Then,

$$\begin{aligned} T(n) &= 3T(n/3) + \lg_3 n \\ &= 3^{\lg_3 n} T(1) + 3^{\lg_3 n - 1} \lg_3 \frac{n}{3^{\lg_3 n - 1}} + \ldots + 3^i \lg_3 \frac{n}{3^i} + \ldots + 3^1 \lg_3 \frac{n}{3^1} + 3^0 \lg_3 \frac{n}{3^0}. \end{aligned}$$

Therefore, by substituting for $T(1) = 1$ we get

$$T(n) = 3^{\lg_3 n} + \lg_3 n(1 + 3 + \ldots + 3^{\lg_3 n - 1}) - \sum_{k=0}^{\lg_3 n - 1} k3^k = n + \lg_3 n(n-1)/2 - \lg_3 n \, n/2 + 3n/4 - 3/4 = 7n/4 - \lg_3 n/2 - 3/4.$$

■

Exercise 126.

Solve exactly using the iteration method the following recurrence. You may assume that n is a power of 4, i.e. $n = 4^k$.

$$T(n) = 4T(n/4) + n^2, \text{ where } T(1) = 4.$$

Proof. $T(n) = 4T(n/4) + n^2$. We use the iteration method. **This first example will be done in fine detail showing all calculations**.

$$\begin{aligned}
T(n) &= 4T(n/4) + n^2 \\
&= 4\left(4T\left(\frac{n}{4^2}\right) + \left(\frac{n}{4}\right)^2\right) + n^2 \\
&= 4^2 T\left(\frac{n}{4^2}\right) + \frac{n^2}{4^1} + \frac{n^2}{4^0} \\
&= 4^2\left(4T\left(\frac{n}{4^3}\right) + \left(\frac{n}{4^2}\right)^2\right) + \frac{n^2}{4^1} + \frac{n^2}{4^0} \\
&= 4^3 T\left(\frac{n}{4^3}\right) + \frac{n^2}{4^2} + \frac{n^2}{4^1} + \frac{n^2}{4^0} \\
&= \dots \\
&= 4^i T\left(\frac{n}{4^i}\right) + \frac{n^2}{4^{i-1}} + \dots + \frac{n^2}{4^1} + \frac{n^2}{4^0} \\
&= 4^i T\left(\frac{n}{4^i}\right) + n^2\left(\frac{1}{4^{i-1}} + \dots + \frac{1}{4^1} + \frac{1}{4^0}\right) \\
&= 4^i T\left(\frac{n}{4^i}\right) + n^2\left(\frac{\frac{1}{4^i} - 1}{\frac{1}{4} - 1}\right) \\
&= 4^i T\left(\frac{n}{4^i}\right) + n^2\left(\frac{1 - \frac{1}{4^i}}{1 - \frac{1}{4}}\right) \\
&= 4^i T\left(\frac{n}{4^i}\right) + \frac{4n^2}{3}\left(1 - \frac{1}{4^i}\right)
\end{aligned}$$

The base case is $T(1) = 4$. We set $n/4^i = 1$, i.e. $n = 4^i$. If we solve this for i, we get $i = \log_4 n = \lg n/2$. Then $T(n/4^i) = T(1) = 4$. In addition $4^i = n$. Therfore the formula for $T(n)$ becomes.

$$\begin{aligned}
T(n) &= 4^i T\left(\frac{n}{4^i}\right) + \frac{4n^2}{3}\left(1 - \frac{1}{4^i}\right) \\
&= nT(1) + \frac{4n^2}{3}(1 - 1/n) \\
&= 4n + \frac{4n^2}{3}(1 - 1/n) \\
&= \frac{4n^2}{3} + \frac{8n}{3}
\end{aligned}$$

Note. It is easy to verify for example that $T(1) = \frac{4\cdot 1^2}{3} + \frac{8\cdot 1}{3} = 4$. This can serve as a checking mechanism that the solution that you have established satisfies at least the base case.

■

Exercise 127.

Solve exactly using the recursion tree method or the iteration method the following recurrence. You may assume that n is a power of 2.

$$T(n) = 4T(n/2) + n$$

with $T(4) = 6$.

Proof.

$$\begin{aligned}
T(n) &= 4T(n/2)+n \\
&= 4\left(4T\left(\frac{n}{2^2}\right)+\left(\frac{n}{2}\right)\right)+n \\
&= 4^2T\left(\frac{n}{2^2}\right)+\frac{4^1n}{2^1}+n \\
&= 4^2\left(4T\left(\frac{n}{2^3}\right)+\left(\frac{n}{2^2}\right)\right)+2^1n+2^0n \\
&= 4^3T\left(\frac{n}{2^3}\right)+2^2n+2^1n+2^0n \\
&= \dots \\
&= 4^iT\left(\frac{n}{2^i}\right)+2^{i-1}n+\ldots+2^1n+2^0n \\
&= 4^iT\left(\frac{n}{2^i}\right)+n\left(2^i-1\right)
\end{aligned}$$

The base case is $T(4) = 6$. We set $n/2^i = 4$, i.e. $n = 2^{i+2}$. If we solve this for i, we get $i = \lg n - 2$. Then $T(n/2^i) = T(4) = 6$. In addition, $2^i = n/4$, $4^i = 2^i \cdot 2^i = n/4 \cdot n/4 = n^2/16$. Therfore the formula for $T(n)$ becomes.

$$\begin{aligned}
T(n) &= 4^iT\left(\frac{n}{2^i}\right)+n\left(2^i-1\right) \\
&= (n^2/16)T(4)+n(n/4-1) \\
&= 3n^2/8+n^2/4-n \\
&= 5n^2/8-n
\end{aligned}$$

Note. It is easy to verify for example that $T(4) = 5 \cdot 4^2/8 - 4 = 6$. This can serve as a checking mechanism that the solution that you have established satisfies at least the base case. ■

Exercise 128.

Solve exactly using the recursion tree method or the iteration method the following recurrence. You may assume that n is a power of 2.

$$T(n) = 8T(n/4) + n$$

with $T(1) = 1$.

Proof.

$$\begin{aligned} T(n) &= 8T(n/4) + n \\ &= 8(8T(n/4^2) + n/4) + n \\ &= 8^2 T(n/4^2) + 2^1 n + 2^0 n \\ &= 8^3 T(n/4^3) + 2^2 n + 2^1 n + 2^0 n \\ &= 8^i T(n/4^i) + 2^{i-1} n + \ldots + 2^1 n + 2^0 n \\ &= 8^i T(n/4^i) + (2^i - 1) \cdot n \end{aligned}$$

For $n/4^i = 1$ we have $i = \lg n/2$ and thus $T(n/4^i) = T(1) = 1$. Then the recurrence gives.

$$\begin{aligned} T(n) &= 8^i T(n/4^i) + (2^i - 1) \cdot n \\ &= 8^{\lg n/2} \cdot 1 + (2^{\lg n/2} - 1)n \\ &= 8^{\lg n/2} \cdot 1 + (n^{1/2} - 1)n \\ &= n^{3/2} + n^{3/2} - n \\ &= 2n^{3/2} - n \end{aligned}$$

Therefore, $T(n) = 2n^{3/2} - n$. ■

Exercise 129.

Solve exactly using the recursion tree method or the iteration method the following recurrence. You may assume that n is a power of 2.

$$T(n) = 8T(n/2) + n^2, \text{ where } T(1) = 1.$$

Proof.

$$\begin{aligned} T(n) &= 8T(n/2) + n^2 \\ &= 8(8T(n/2^2) + n^2/2^2) + n^2 \\ &= 8^2 T(n/2^2) + 2^1 n^2 + n^2 \\ &= 8^3 T(n/2^3) + 2^2 n^2 + 2^1 n^2 + n^2 \\ &= 8^i T(n/2^i) + 2^{i-1} n^2 + \ldots + 2^1 n^2 + n^2 \\ &= 8^i T(n/2^i) + (2^i - 1) \cdot n^2 \end{aligned}$$

For $n/2^i = 1$ we have $i = \lg n$ and thus $T(n/2^i) = T(1) = 1$. Then the recurrence gives.

$$\begin{aligned} T(n) &= 8^i T(n/2^i) + (2^i - 1) \cdot n^2 \\ &= 8^{\lg n} \cdot 1 + (n-1)n^2 \\ &= n^3 + (n-1)n^2 \\ &= 2n^3 - n^2 \end{aligned}$$

Therefore, $T(n) = 2n^3 - n^2$. ■

Exercise 130.

Solve exactly using the recursion tree method the following recurrence. You may assume that n is a power of 3.

$$T(n) = 9T(n/3) + n^2, \text{ where } T(1) = 1.$$

Proof.

$$\begin{aligned} T(n) &= 9T(n/3) + n^2 \\ &= 9(9T(n/3^2) + n^2/9) + n^2 \\ &= 9^2T(n/3^2) + n^2 + n^2 \\ &= 9^iT(n/3^i) + n^2 + \ldots + n^2 + n^2 \\ &= 9^iT(n/3^i) + i \cdot n^2 \end{aligned}$$

A first observation is that the number of n^2's we have is equal to the power of 9 (or equivalently, of 3) in the coefficient of $T(.)$ (or the denominator of $T(n/.)$ respectively). In the base case $T(1) = 1$, therefore we choose $n/3^i = 1$, which gives $i = \lg n/\lg 3$. Then, for this i, $T(n/3^i) = T(1) = 1$. We note that $9 = 3^2 = (2^{\lg 3})^2 = 2^{2\lg 3}$. We therefore get that

$$\begin{aligned} T(n) &= 9^iT(n/3^i) + in^2 \\ &= 9^{\lg n/\lg 3}T(1) + n^2 \lg n/\lg 3 \\ &= 3^{2\lg n/\lg 3} + n^2 \lg n/\lg 3 \\ &= 2^{2\lg 3 \lg n/\lg 3} + n^2 \lg n/\lg 3 \\ &= 2^{2\lg n} + n^2 \lg n/\lg 3 \\ &= n^2 + n^2 \lg n/\lg 3 \end{aligned}$$

It is easy to verify that this solution satisfies both the recurrence and the boundary condition.

■

Exercise 131.

Solve the following recurrences. Make your bounds as tight as possible. You may assume that n is a power of two.

$$T(n) = T(n/2) + \lg n, \qquad \text{where} \quad T(2) = 1.$$

Proof. $T(n) = T(n/2) + \lg n$. We use the iteration method, noting that $T(2) = 1$.

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + \lg n \\ &= \left(T\left(\frac{n}{2^2}\right) + \left(\lg \frac{n}{2}\right)\right) + \lg n \\ &= T\left(\frac{n}{2^2}\right) + \lg \frac{n}{2^1} + \lg \frac{n}{2^0} \\ &= \dots \\ &= T\left(\frac{n}{2^i}\right) + \lg \frac{n}{2^{i-1}} + \ldots + \lg \frac{n}{2^1} + \lg \frac{n}{2^0} \end{aligned}$$

The base case is $T(2) = 1$. We set $n/2^i = 2$, i.e. $n = 2^{i+1}$. If we solve this for i, we get $i = \lg n - 1$. Then $T(n/2^i) = T(2) = 1$. In addition $2^i = n/2$. Therfore the formula for $T(n)$ becomes.

$$\begin{aligned} T(n) &= T\left(\frac{n}{2^i}\right) + \lg \frac{n}{2^{i-1}} + \ldots + \lg \frac{n}{2^1} + \lg \frac{n}{2^0} \\ &= T\left(\frac{n}{2^{\lg n - 1}}\right) + \lg \frac{n}{2^{\lg n - 1 - 1}} + \ldots + \lg \frac{n}{2^1} + \lg \frac{n}{2^0} \\ &= T(2) + \left(\lg n - (\lg 2^{\lg n - 1 - 1})\right) + \left(\lg n - (\lg 2^{\lg n - 2 - 1})\right) + \ldots + \left(\lg n - \lg 2^1\right) + \left(\lg n - \lg 2^0\right) \\ &= 1 + (\lg n - (\lg n - 2)) + (\lg n - (\lg n - 3)) + \ldots + (\lg n - 1) + (\lg n - 0) \\ &= 1 + 2 + 3 + \ldots + (\lg n - 1) + (\lg n) \\ &= \sum_{j=1}^{\lg n} j \\ &= \frac{\lg n (\lg n + 1)}{2} \end{aligned}$$

Therefore, $T(n) = \lg n(\lg n + 1)/2$. It is easy to verify that $T(2)$ is indeed 1. ∎

Exercise 132.
Solve exactly, assuming n is a power of two, $T(n) = 2T(n/2) + \lg n$ where $T(1) = 1$.

Proof.

$$T(n) = 2T(n/2) + \lg n = \ldots = 2^{\lg n} T(1) + 2^{\lg n - 1} \lg \frac{n}{2^{\lg n - 1}} + \ldots + 2^i \lg \frac{n}{2^i} + \ldots + 2^1 \lg \frac{n}{2^1} + 2^0 \lg \frac{n}{2^0}.$$

Therefore, by substituting for $T(1) = 1$ we get

$$T(n) = 2^{\lg n} + \lg n(1 + 2 + \ldots + 2^{\lg n - 1}) - \sum_{k=0}^{\lg n - 1} k 2^k = n + (n-1) \cdot \lg n - n \cdot \lg nn + 2n - 2 = 3n - \lg n - 2.$$

■

Exercise 133.

The recurrence to solve is the following one for n a power of two.

$$T(n) = 2T(n/2) + (n \lg n), \text{ where } T(1) = 1.$$

Proof. We solve the recurrence using the iteration method. Below, we set $n/2^i = 1$ to satisfy the boundary condition $T(1)$. Then, $i = \lg n$.

$$\begin{aligned}
T(n) = T(2^k) &= 2T(n/2) + (n \lg n) \\
&= 2^2 T(n/4) + 2(n/2)\lg(n/2) + n \lg n \\
&= 2^2 T(n/4) + n \lg(n/2) + n \lg n \\
&= 2^i T(n/2^i) + n \lg(n/2^{i-1}) + \ldots + n \lg(n/2^0) \\
&= 2^i T(n/2^i) + n \sum_{k=0}^{i-1} \lg(n/2^k) = 2^{\lg n} T(1) + n \sum_{k=0}^{\lg n - 1} (\lg(n) - k \lg 2) \\
&= n + n \sum_{k=0}^{\lg n - 1} \lg n - n \sum_{k=0}^{\lg n - 1} k \\
&= n + n \lg^2 n - n(\lg n - 1)(\lg n)/2 \\
&= n + n \lg n (\lg n + 1)/2
\end{aligned}$$

Therefore, $T(n) = \Theta(n \lg^2 n)$.

■

Exercise 134.

Solve exactly using the iteration method the following recurrence. You may assume that n is a power of 3, ie $n = 3^k$.

$$T(n) = 3T(n/3) + 9n, \text{ where } T(1) = 1.$$

Proof.
Let $n = 3^k$ and thus $k = \log_3 n$. As $T(n) = 3T(n/3) + 9n$, then $T(n/3) = 3T(n/3) + 9n/3$ and similarly for $T(n/3^2) = \ldots$ and so on.

$$\begin{aligned}
T(n) &= 3\,T(n/3) + 9n \\
T(3^k) &= 3\,T(3^{k-1}) + 9\,3^k \\
&= 3\,(3\,T(3^{k-2}) + 9\,3^{k-1}) + 9\,3^k \\
&= 3^2\,T(3^{k-2}) + 9\,3^k + 9\,3^k \\
&= 3^2\,(3\,T(3^{k-3}) + 9\,3^{k-2}) + 9\,3^k + 9\,3^k \\
&= 3^3\,T(3^{k-3}) + 9\,3^k + 9\,3^{k-1} + 9\,3^k \\
&\ldots \\
&= 3^k\,T(3^{k-k}) + 9\,3^k + \ldots + 9\,3^k + 9\,3^k \\
&= 3^k\,T(1) + 9(3^k + \ldots + 3^{k-1} + 3^k) \\
&= 3^k\,1 + 9\,3^k\,k \\
&= n + 9n\log_3 n
\end{aligned}$$

■

Exercise 135.
Solve exactly using the iteration method the following recurrence. You may assume that n is a power of 5.

$$T(n) = 5T(n/5) + n \lg_5 n, \text{ where } T(1) = 1.$$

Proof.
Let $n = 5^k$ and thus $k = \log_5 n$. As $T(n) = 5T(n/5) + n\lg_5 n$, then $T(5^k) = 5T(5^{k-1}) + k5^k$.

$$\begin{aligned}
T(n) &= 5\,T(n/5) + n\lg_5 n \\
T(5^k) &= 5\,T(5^{k-1}) + k5^k \\
&= 5\,(5\,T(5^{k-2}) + (k-1)5^{k-1}) + k5^k \\
&= 5^2\,T(5^{k-2}) + (k-1)5^k + k5^k \\
&= 5^2\,(5\,T(5^{k-3}) + (k-2)5^{k-2}) + (k-1)5^k + k5^k \\
&= 5^3\,T(5^{k-3}) + (k-2)5^k + (k-1)5^k + k5^k \\
&\dots \\
&= 5^k\,T(5^{k-k}) + 1\cdot 5^k + \dots + (k-1)5^k + k5^k \\
&= 5^k\,T(1) + 5^k(1+2+\dots+k) \\
&= 5^k\,1 + 5^k\,k(k+1)/2 \\
&= n + n\log_5(n)(\log_5(n)+1)/2
\end{aligned}$$

■

Exercise 136.

Solve exactly using the iteration method the following recurrence. You may assume that n is a power of 2.

$$T(n) = 8T(n/2) + n, \text{ where } T(1) = 1.$$

Proof.
We are reminded of the geometric sequence $1 + x^1 + x^2 + \ldots + x^{i-1} = (x^i - 1)/(x-1)$. If $x = 4$ this gives $1 + 4^1 + 4^2 + \ldots + 4^{i-1} = (4^i - 1)/(4-1)$.

$$\begin{aligned}
T(n) &= 8T(n/2) + n \\
&= 8(8T(n/2^2) + n/2) + n \\
&= 8^2 T(n/2^2) + 4^1 n + 4^0 n \\
&= 8^2(8T(n/2^3) + n/2^2) + 4^1 n + 4^0 n \\
&= 8^3 T(n/2^3) + 4^2 n + 4^1 n + 4^0 n \\
&= 8^i T(n/2^i) + 4^{i-1} n + \ldots + 4^1 n + 4^0 n \\
&= 8^i T(n/2^i) + ((4^i - 1)/(4-1)) \cdot n
\end{aligned}$$

For $n/2^i = 1$ we have $i = \lg n$ and thus $T(n/2^i) = T(1) = 1$. We are also reminded that $8^{\lg n} = (2^3)^{\lg n} = 2^{3\lg n} = (2^{\lg n})^3 = n^3$. For the same reason, $4^{\lg n} = n^2$. Then the recurrence gives.

$$\begin{aligned}
T(n) &= 8^i T(n/2^i) + (4^i - 1)n/3 \\
&= 8^{\lg n} + (4^{\lg n} - 1)n/3 \\
&= 8^{\lg n} + (n^2 - 1)n/3 \\
&= n^3 + (n^2 - n)/3 \\
&= 4n^3/3 - n/3
\end{aligned}$$

Therefore, $T(n) = 4n^3/3 - n/3$. ■

Exercise 137.

Solve exactly using the iteration method the following recurrence. You may assume that n is a power of 2.

$$T(n) = 4T(n/4) + n^2, \text{ where } T(1) = 4.$$

Proof.
We use the iteration method.

$$\begin{aligned}
T(n) &= 4T(n/4) + n^2 \\
&= 4\left(4T\left(\frac{n}{4^2}\right) + \left(\frac{n}{4}\right)^2\right) + n^2 \\
&= 4^2 T\left(\frac{n}{4^2}\right) + \frac{n^2}{4^1} + \frac{n^2}{4^0} \\
&= 4^2\left(4T\left(\frac{n}{4^3}\right) + \left(\frac{n}{4^2}\right)^2\right) + \frac{n^2}{4^1} + \frac{n^2}{4^0} \\
&= 4^3 T\left(\frac{n}{4^3}\right) + \frac{n^2}{4^2} + \frac{n^2}{4^1} + \frac{n^2}{4^0} \\
&= \dots \\
&= 4^i T\left(\frac{n}{4^i}\right) + \frac{n^2}{4^{i-1}} + \dots + \frac{n^2}{4^1} + \frac{n^2}{4^0} \\
&= 4^i T\left(\frac{n}{4^i}\right) + n^2\left(\frac{1}{4^{i-1}} + \dots + \frac{1}{4^1} + \frac{1}{4^0}\right) \\
&= 4^i T\left(\frac{n}{4^i}\right) + n^2\left(\frac{\frac{1}{4^i} - 1}{\frac{1}{4} - 1}\right) \\
&= 4^i T\left(\frac{n}{4^i}\right) + n^2\left(\frac{1 - \frac{1}{4^i}}{1 - \frac{1}{4}}\right) \\
&= 4^i T\left(\frac{n}{4^i}\right) + \frac{4n^2}{3}\left(1 - \frac{1}{4^i}\right)
\end{aligned}$$

The base case is $T(1) = 4$. We set $n/4^i = 1$, i.e. $n = 4^i$. If we solve this for i, we get $i = \log_4 n = \lg n/2$. Then $T(n/4^i) = T(1) = 4$. In addition $4^i = n$. Therfore the formula for $T(n)$ becomes.

$$\begin{aligned}
T(n) &= 4^i T\left(\frac{n}{4^i}\right) + \frac{4n^2}{3}\left(1 - \frac{1}{4^i}\right) \\
&= nT(1) + \frac{4n^2}{3}(1 - 1/n) \\
&= 4n + \frac{4n^2}{3}(1 - 1/n) \\
&= \frac{4n^2}{3} + \frac{8n}{3}
\end{aligned}$$

Note. It is easy to verify for example that $T(1) = \frac{4\cdot 1^2}{3} + \frac{8\cdot 1}{3} = 4$. This can serve as a checking mechanism that the solution that you have established satisfies at least the base case. ■

3.3 If all else fails: subsitution method

I prefer to call it guess and check, and if you become desperate or running out of time you have a tendency to call it guess and pray!

Exercise 138.
Solve the following recurrences; make your bounds as tight as possible. Use strong induction if necessary.

$$T(n) \quad = \quad T(n/2)+T(n/3)+32n, \quad \text{where} \quad T(1)=100.$$

Proof. Since $T(n)=T(n/2)+T(n/3)+32n$ we have for $n=1$ from the base case that $T(1)=100$. We show that $T(n)=\Omega(n)$. By the recurrence we have that $T(n)\geq 32n$ for all $n>1$ since $T(n/2), T(n/3)$ are non-negative and the other term is $32n$. In addition, $T(1)=100\geq 32\cdot 1=32$. Therefore $T(n)\geq 32n$ for $n=1$ as well. Therefore $T(n)=\Omega(n)$.
We now prove that $T(n)=O(n)$ or equivalently that there exist positive constants n_2, c_2 such that $T(n)\leq c_2 n$ for all $n\geq n_2$. Proof is by induction on n.
Base Case. $T(1)=100\leq c_2\cdot 1$ is true provided that $c_2\geq 100$. This completes the base case.
Inductive Step. Let $T(i)\leq c_2 i$ be true for all $1\leq i<n$. We want to show that $T(n)\leq c_2 n$. Since $T(i)\leq c_2 i$ for all $i<n$, it is also true for $i=n/2$ and $i=n/3$ as well, and thus by the inductive hypothesis we obtain $T(n/2)\leq c_2 n/2$ and $T(n/3)\leq c_2 n/3$. Then we use these two inequalities in the recurrence to obtain.

$$\begin{aligned} T(n) &= T(n/2)+T(n/3)+32n \\ &\leq c_2 n/2 + c_2 n/3 + 32n \\ &= 5c_2 n/6 + 32n \end{aligned}$$

In order to complete the inductive step we need to prove that $T(n)\leq c_2 n$ or equivalently $5c_2 n/6+32n\leq c_2 n$. The latter is equivalent to $32n\leq c_2 n/6$ or $c_2\geq 192$. Therefore the inductive step is completed and $T(n)\leq c_2 n$ for any $c_2\geq 192$. In conclusion, combining the inductive step and the base case we obtain that $T(n)\leq c_2 n$ for all $n\geq 1$ as long as $c_2\geq 192$. We can fix $c_2=192$ and thus show that $T(n)=O(n)$.
Therefore since $T(n)=O(n)$ and $T(n)=\Omega(n)$ we obtain that $T(n)=\Theta(n)$. The constants in the definition are $1, 192$ and for n_0 we can still use 1.

■

Exercise 139.

Solve the recurrence $T(n) = T(n/2) + T(2n/3) + n$, where $T(1) = \Theta(1)$.

Proof. **Attempt 1.** We try $T(n) = O(n^2)$, i.e. $T(n) \leq cn^2$ for for some constant $c, n_0 > 0$ and for all $n \geq n_0$. Using by the inductive hypothesis $T(n/2) \leq c(n/2)^2$ and $T(2n/3) \leq c(2n/3)^2$ and

$$T(n) \leq cn^2/4 + 4cn^2/9 + n = c(25/36)n^2 + n \leq cn^2$$

for large enough n and $c > 0$. The bound, however, is not the best possible one.

Attempt 2. We try $T(n) \leq cn^\alpha$ where α is a constant such that $1 < \alpha < 2$. Similarly as before,

$$T(n) \leq cn^\alpha((1/2)^\alpha + (2/3)^\alpha) + n \leq cn^\alpha$$

The latter inequality holds asymptotically for large n provided that $(1/2)^\alpha + (2/3)^\alpha < 1$. If this value of $(1/2)^\alpha + (2/3)^\alpha$ is 0.99 then $0.99cn^\alpha + n$ is indeed smaller than cn^α asymptotically for large n.

Attempt 3. We try $T(n) \leq cn^\alpha - 6n$ where α are constant such that $1 < \alpha < 2$ and $\beta > 1$. (There is no magic in choosing a 6: it relates to $1/2$ and $2/3$ after originally choosing a β instead.) Similarly as before,

$$T(n) \leq cn^\alpha((1/2)^\alpha - 6n/2 + (2/3)^\alpha) - 2n/3 + n \leq cn^\alpha - 6n$$

The latter inequality holds asymptotically for large n provided that $(1/2)^\alpha + (2/3)^\alpha < 1$. If this value of $(1/2)^\alpha + (2/3)^\alpha$ is 0.99 then $0.99cn^\alpha - 6n$ is indeed smaller than $cn^\alpha - 6n$ asymptotically for large n. A numerical computation of an α yields one in the range $[1.293, 1.294]$. The exact value is of no interest; only the fact that it is between 1 and 2!

■

Exercise 140.

There are more examples with the substitution method in the pages to follow.
Solve the following recurrences. You may assume $T(1) = \Theta(1)$, where necessary. Make your bounds as tight as possible. Use asymptotic notation to express your answers. Justify your answers.
$T(n) = T(n/4) + T(3n/4) + n\lg n$.

Proof. We use the substitution method to solve this recurrence. We guess as a solution $T(n) = O(n\lg^2 n)$ ie $T(n) \leq cn\lg^2 n$, for some positive constant $c > 0$ and for all $n \geq n_0$. We use induction to prove that this is indeed a solution to the recurrence.

$$\begin{aligned} T(n) &= T(n/4) + T(3n/4) + n\lg n \\ &\leq (cn/4)\lg^2(n/4) + (3cn/4)\lg^2(3n/4) + n\lg n \\ &= c(n/4)(\lg n - 2)^2 + c(3n/4)(\lg n + \lg 3 - 2)^2 + n\lg n \\ &= c(n/4)(\lg n - 2)^2 + c(3n/4)(\lg n)^2 + n\lg n \\ &\leq cn\lg^2 ncn + n - cn\lg n \\ &\leq cn\lg^2 n \end{aligned}$$

for any $n \geq 4$ and $c \geq 1$. We choose for boundary condition $T(2) = 2$ so that $T(n) \leq cn\lg^2 n$ would result in $2 \leq c2\lg^2 2 = 2c$, ie $c \geq 1$. Therefore for any $n \geq 4$ and any $c \geq 1$, we have $T(n) \leq cn\lg^2 n$.

■

Exercise 141.

Solve the following recurrences associated with three problems introduced in class. First name the problem associated with a recurrence and then solve the recurrence as best as you can (i.e. tightest possible bound) asymptotic or not! You may assume n is a power of 2. Formulate appropriate base case as well.

(a) $T(n) = T(n/2) + 1$.
(b) $T(n) = 2T(n/2) + (n-1)$.
(c) $T(n) = T(n-1) + (n-1)$.

Proof. (a) $T(1) = 1$, Binary search and $T(n) = \lg n + 1$ for (upper bound on) number of comparison in worst-case thus determining running time also but using a rougher analysis ($n/2$ should have been $(n-1)/2$).
(b) $T(2) = 1$, Merge sort and $T(n) = n\lg n - (n-1)$ for number of comparisons in worst-case thus determining running time also.
(c) $T(1) = 0$, Insertion sort (recursive formulation) $T(n) = n(n-1)/2$ for number of comparisons in the worst-case determining running time also.

■

Exercise 142.

Solve the recurrennce $T(n) = T(an) + T(bn) + n$ where $a + b < 1$ and $a > 0$ and $b > 0$ are constant. In the following problem, we address the case $a + b = 1$.

Proof. We could use the substitution method to solve this recurrence.

(1.) Ω part: $T(n) = \Omega(n)$ is straightforward as $T(n) = T(an) + T(bn) + n \geq n$.

(2.) O part: We show $T(n) = O(n)$ i.e. we show that there exist c, n_0 positive and constant such that $T(n) \leq cn$ for $\forall n \geq n_0$. Proof by induction. Since $an < n$ and $bn < n$ the induction hypothesis for $T(an)$ and $T(bn)$ gives

$$\begin{aligned} T(n) &= T(an) + T(bn) + n \\ &\leq c(an) + c(bn) + n \\ &= cn(a+b) + n \end{aligned}$$

For the last expression to be at most cn we need to have

$$\begin{aligned} cn(a+b) + n &\leq cn \\ n &\leq c(1-a-b)n \\ 1 &\leq c(1-a-b) \end{aligned}$$

Since $a + b < 1$ we have $1 - a - b > 0$. Therefore if we choose c such that $c > 1/(1-a-b)$ the last inequality is true and the inductive step is proven.

■

Exercise 143.

Solve the following recurrence. Use asymptotic notation to express your solution. Give a solution as asymptotically tight as possible.

$$T(n) = T(n/2) + T(3n/4) + n$$

Proof.

1st Approximation By using the substitution method we guess as a first solution $T(n) = O(n^2)$. It is easy to see that it satisfies the recurrence. However it is not the tightest possible solution.

2nd Approximation

Subsequently, we guess $T(n) = O(n^a)$ for some constant $1 < a < 2$. Then by induction

$$\begin{aligned} T(n) &= T(n/2) + T(3n/4) + n \\ &\leq c(n/2)^a + c(3n/4)^a + n \end{aligned}$$

The last expression $c(n/2)^a + c(3n/4)^a + n$ is at most cn^a provided that

$$c(n/2)^a + c(3n/4)^a + n \leq cn^a$$

which is equivalent to

$$(1/2)^a + (3/4)^a + (1/c)n^{1-a} \leq 1$$

Note that the third term of the left hand side is the only non-constant term and since $0 < a < 1$ and a is constant, it grows asymptotically to zero. We then observe by numerical methods that $(1/2)^a + (3/4)^a - 1$ for $a = 1.507$ is positive and $a = 1.508$ is negative ie the solution to the $(1/2)^a + (3/4)^a = 1$ is an a such that $1.507 < a < 1.508$. We call this solution b. For $a = 1.508$, we have that $(1/2)^a + (3/4)^a - 1$ is -0.003, therefore the expression

$$(1/2)^a + (3/4)^a + (1/c)n^{1-a} \leq 1$$

becomes

$$1 - 0.003 + (1/c)n^{1-a} \leq 1$$

and

$$(1/c)n^{1-a} \leq 0.003$$

for $c = 1$ and any $n > n_0 = 10^9$. This solution, however **is not tight**. The reason is the existence of the additive term n, which, however, is asymptotically smaller than n^a as $a > 1$.

3rd Approximation

We thus guess as a solution a variant of the 2nd approximation ie $T(n) = cn^a - bn$ for some constants $1 < a < 2$, $c, b > 0$. Then by induction

$$\begin{aligned} T(n) &= T(n/2) + T(3n/4) + n \\ &\leq c(n/2)^a - bn/2 + c(3n/4)^a - b3n/4 + n \end{aligned}$$

We examine the last expression $c(n/2)^a + c(3n/4)^a + n - b5n/4$ which is at most $cn^a - bn$ provided that (a) we first eliminate the linear terms, ie set $n - 5bn/4 = -bn$ which implies that we need to set $b = 5$. (b) Then, it simplifies into

$$c(n/2)^a + c(3n/4)^a \leq cn^a$$

which is equivalent to

$$(1/2)^a + (3/4)^a \leq 1.$$

Thus the solution for a of $(1/2)^a + (3/4)^a = 1$ which lies somewhere between 1.507 and $a = 1.508$ gives the value of the exponent a in $T(n) = \Theta(n^a)$.

Thus, a tight bound for the recurrence is $T(n) = cn^a - 5n$, where c can be any constant $c > 0$ and a is the solution of $(1/2)^a + (3/4)^a = 1$ which is a real number between 1 and 2 (or more accurately between 1.507 and 1.508 if one relies to numerical methods). **Note.** The best possible bound is the one derived from the third approximation. The other two steps are not necessary; they just show how one can successively refine a guess in the substitution method. Note also, that the numerical computations we used to find that a is between 1.507 and 1.508 are not necessary, as the exponent a can just be expressed as a solution to an equation (ie $(1/2)^a + (3/4)^a = 1$). ■

Exercise 144.

Solve the following recurrences. You may assume $T(1) = \Theta(1)$, where necessary. Make your bounds as tight as possible. Use asymptotic notation to express your answers. Justify your answers.

$$T(n) = T(n/4) + T(3n/4) + n\lg n.$$

Proof.
We use the substitution method to solve this recurrence. We guess as a solution $T(n) = O(n\lg^2 n)$ ie $T(n) \leq cn\lg^2 n$, for some positive constant $c > 0$ and for all $n \geq n_0$. We use induction to prove that this is indeed a solution to the recurrence.

$$\begin{aligned}
T(n) &= T(n/4) + T(3n/4) + n\lg n \\
&\leq (cn/4)\lg^2(n/2) + (3cn/4)\lg^2(3n/2) \\
&= cn\lg^2 n + cn(4 + 3\lg^2 3 - 3\lg 3) + cn\lg n(-4 + 3\lg 3/2) \\
&\leq cn\lg^2 n + 1.13cn - 1.6225cn\lg n \\
&\leq cn\lg^2 n + 2cn - 2cn\lg n \\
&\leq cn\lg^2 n
\end{aligned}$$

for any $n \geq 2$ and $c > 0$. We choose for boundary condition $T(2) = 2$ so that $T(n) \leq cn\lg^2 n$ would result in $2 \leq c2\lg^2 2 = 2c$, ie $c \geq 1$. Therefore for any $n \geq 2$ and any $c \geq 1$, we have $T(n) \leq cn\lg^2 n$. ■

Exercise 145.

Solve the following recurrence. Use asymptotic notation to express your solution.

$$T(n) = 5T(n^{1/5}) + \log_5 n,$$

where $T(5) = 1$ and $n = 5^{5^k}$ for some k.

Proof.
As $n = 5^{5^k}$, $\log_5 n$ is well defined as it is an integer. We set $n = 5^l$ ie $l = \log_5 n$. Then, $T(n) = T(5^l)$, and we rename $A(l) = T(5^l) = T(n)$. Note that $A(1) = T(5^1) = T(5) = 1$. Then the original recurrence becomes (note again that $l/5$ is an integer, as l itself is also a power of 5).

$$\begin{aligned} A(l) = T(n) &= 5T(n^{1/5}) + \log_5 n \\ &= 5T(5^{l/5}) + l \\ &= 5A(l/5) + l. \end{aligned}$$

The simplified recurrence has solution $A(l) = \Theta(l \lg l)$ (master method). As $n = 5^l$, then $l = \log_5 n$ and $\lg l = \Theta(\log_5 l) = \Theta(\lg \lg n)$. Therefore $T(n) = A(l) = \Theta(\lg n \lg \lg n)$. ■

Exercise 146.

Solve the following recurrence. Use asymptotic notation to express your solution.

$$T(n) = 2T(n/2) + T(n/4) + n^2$$

Proof.
We first note that $T(n) = 2T(n/2) + T(n/4) + n^2 \geq n^2$, and thus $T(n) = \Omega(n^2)$.
Moreover, since

$$T(n) = 2T(n/2) + T(n/4) + n^2 \leq 3T(n/2) + n^2$$

either through master method (case three) or by using the substitution method we deduce that $T(n) = O(n^2)$. Therefore $T(n) = \Theta(n^2)$. Let's show that $T(n) \leq 3T(n/2) + n^2$ has $T(n) = O(n^2)$ as a solution i.e. $T(n) \leq cn^2$ for all $n \geq n_0$. We choose the base case ourselves $T(1) = 1$ which implies $c \geq 1$. By the inductive hypothesis $T(i) \leq ci^2$ we have for $i = n/2$ that $T(n) \leq 3T(n/2) + n^2 \leq 3c(n/2)^2 + n^2 = 3cn^2/4 + n^2$. The latter is at most cn^2 for $c \geq 4$. Therefore for $c = 4$ and $n_0 = 1$ we have shown $T(n) = O(n^2)$. If we combine this with the $T(n) = \Omega(n^2)$ the result follows. ■

Exercise 147.

Solve the following recurrences. You may assume $T(1) = \Theta(1)$, where necessary. Use asymptotic notation. Justify your answers.

a. $T(n) = 16T(n/2) + n^2$.
b. $T(n) = 9T(n/3) + n^4$.
c. $T(n) = 36T(n/6) + n^2$.
d. $T(n) = T(n/2) + n$.
e. $T(n) = 8T(n/8) + n^2$.
f. $T(n) = 16T(n/2) + n$.
g. $T(n) = 9T(n/3) + n^2$.
h. $T(n) = 4T(n/2) + n^4$.
i. $T(n) = 4T(n/2) + n$.
j. $T(n) = T(n/6) + T(2n/3) + 2n$
k. $T(n) = 4T(n/2) + 3n^2$
l. $T(n) = 32T(n/4) + 3n^2$

Proof.
a. We use the master method. $a = 16 \geq 1$, $b = 2 > 1$, $f(n) = n^2$, and $\log_b a = 4$. Then $n^{\lg_b a} = n^4$. As $f(n) = n^2 = O(n^{\lg_b a - 1}) = O(n^3)$, case one of the master method applies, and thus $T(n) = \Theta(n^4)$.
b. We use the master method. $a = 9 \geq 1$, $b = 3 > 1$, $f(n) = n^4$, and $\log_b a = 2$. Then $n^{\lg_b a} = n^2$. As $f(n) = n^4 = \Omega(n^{2+2}) = O(n^{\lg_b a + 2})$, for $\varepsilon = 2$, case three of the master method applies, and thus $T(n) = \Theta(n^4)$, as in addition $af(n/b) = 9f(n/3) = 9n^4/81 \leq cn^4 = cf(n)$ for say $c = 1/2$.
c. We use the master method. $a = 36 \geq 1$, $b = 6 > 1$, $f(n) = n^2$, and $\log_b a = 2$. Then $n^{\lg_b a} = n^2$. As $f(n) = n^2 = \Theta(n^{\lg_b a}) = \Theta(n^2)$, case two of the master method applies, and thus $T(n) = \Theta(n^2 \lg n)$.
d. We use the master method. $a = 1 \geq 1$, $b = 2 > 1$, $f(n) = n$, and $\log_b a = 0$. Then $n^{\lg_b a} = n^0 = 1$. As $f(n) = n = \Omega(n^{0+1}) = O(n^{\lg_b a + 1})$, for $\varepsilon = 1$, case three of the master method applies, and thus $T(n) = \Theta(n)$, as in addition $af(n/b) = 1f(n/2) = n/2 \leq cn = cf(n)$ for say $c = 1/2$.
(e) We use the master method. $a = 8 \geq 1$, $b = 8 > 1$, $f(n) = n^2$, and $\log_b a = 1$. Then $n^{\lg_b a} = n$. As $f(n) = n^2 = \Omega(n^{1+1})$ case three of the master method applies, since also $8f(n/8) = 8n^2/64 \leq 1/2n^2$ for $c = 1/2 < 1$, and thus $T(n) = \Theta(n^2)$.
(f). We use the master method. $a = 16 \geq 1$, $b = 2 > 1$, $f(n) = n$, and $\log_b a = 4$. Then $n^{\lg_b a} = n^4$. As $f(n) = n = O(n^{4-2})$, case one of the master method applies, and thus $T(n) = \Theta(n^4)$.
(g). We use the master method. $a = 9 \geq 1$, $b = 3 > 1$, $f(n) = n^2$, and $\log_b a = 2$. Then $n^{\lg_b a} = n^2$. As $f(n) = n^2 = \Theta(n^2)$, case two of the master method applies, and thus $T(n) = \Theta(n^2 \lg n)$.
It has $a = 4 \geq 1$, $b = 2 > 1$, $f(n) = n^4$. $n^{\log_b a} = n^{\lg 4} = n^2$. Obviously $n^4 = f(n) = \Omega(n^{\log_b a + \varepsilon}) = \Omega(n^3)$, by choosing $\varepsilon = 1 > 0$. Case 3 of the Master method applies since also $af(n/b) = 4(n/2)^4 = n^4/4 \leq cn^4 = (1/2)n^3$, by choosing $c = 1/2 < 1$, and thus $T(n) = \Theta(n^4)$.
(i) It has $a = 4 \geq 1$, $b = 2 > 1$, $f(n) = n$. $n^{\log_b a} = n^{\lg 4} = n^2$. Obviously $n = f(n) = O(n^{\log_b a - \varepsilon}) = \Omega(n)$, by choosing $\varepsilon = 1 > 0$. Case 1 of the Master method applies and thus $T(n) = \Theta(n^2)$.
(j) We use guess and check (aka the substitution method). Let $T(n) \leq cn$. Then

$$T(n) \leq cn/6 + 2cn/3 + 2n = 5cn/6 + 2n = (5c/6 + 2)n.$$

In order to get $5c/6 + 2$ less than or equal to c it suffices to choose any c such that $c \geq 12$. Then $T(n) = \Theta(n)$ (as $T(n) \geq 2n$ as well).
(k) We use the master method. $a = 4 \geq 1$, $b = 2 > 1$, and $n^{\log_b a} = n^2$. $f(n) = 3n^2$ and positive. Therefore, $f(n) = \Theta(n^{\log_b a})$ and therefore $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^2 \lg n)$.
(l) We use the master method. $b = 4 > 1$, $a = 32 = 2^5 = 4^{5/2} = 4^{2.5} \geq 1$, and $n^{\log_b a} = n^{2.5}$. $f(n) = 3n^2$ and positive. Therefore, $f(n) = O(n^{\log_b a - \varepsilon})$ and therefore $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{2.5})$.

■

Exercise 148.

Solve the following recurrences. You may assume $T(1) = \Theta(1)$, where necessary. Use asymptotic notation. Justify your answers.

a. $T(n) = 4T(n^{1/4}) + \log_4 n$, where $T(4) = 1$ and $n = 4^{4^k}$ for some k.
b. $T(n) = 8n + \sum_{i=1}^{\lg n} T(n/2^i)$.
c. $T(n) = T(n/2) + T(2n/3) + n$.
d. $T(n) = 4T(n/2) + n^2/\lg n$, n a power of 2.
e. $T(n) = 4T(n/3) + n^{1/3}$.
f. $T(n) = n^{3/4}T(n^{1/4}) + n$, where $T(n) \leq c$, for all $n \leq n_0$, for some positive constant n_0, c.
g. $T(n) = 9T(n/3) + n\lg^2 n$
h. $T(n) = 3T(n/9) + n^2$
i. $T(n) = 2T(n/4) + \sqrt{n}$
j. $T(n) = 4T(n/2) + n^2$.
k. $T(n) = 16T(n/2) + n$.
l. $T(n) = 3T(n/9) + n$.

Proof.
(a) As $n = 4^{4^k}$, $\log_4 n$ is well defined as it is an integer. We set $n = 4^l$ ie $l = \log_4 n$. Then, $T(n) = T(4^l)$, and we rename $A(l) = T(4^l) = T(n)$. Note that $A(1) = T(4^1) = T(4) = 1$. Then the original recurrence becomes (note again that $l/4$ is an integer, as l itself is also a power of 4).

$$\begin{aligned} A(l) = T(n) &= 4T(n^{1/4}) + \log_4 n \\ &= 4T(4^{l/4}) + l \\ &= 4A(l/4) + l. \end{aligned}$$

The simplified recurrence has solution $A(l) = \Theta(l \lg l)$. As $n = 4^l$, $l = \log_4 n$ and $\lg l = \Theta(\log_4 l) = \Theta(\lg\lg n)$. Therefore $T(n) = A(l) = \Theta(\lg n \lg\lg n)$.
(b) We write the recurrence as follows

$$T(n) = 8n + T(n/2) + T(n/4) + \ldots + T(n/n) = (4n + T(n/2)) + (4n + T(n/4) + \ldots + T(1))$$

We then substitute $n/2$ for n in the original recurrence to get.

$$T(n/2) = 8n/2 + \sum_{i=1}^{\lg(n/2)} T((n/2)/2^i) = 4n + T((n/2)/2) + T((n/2)/4) + \ldots + T(1) = 4n + T(n/4) + T(n/8) + \ldots + T(1).$$

In the derived expression for $T(n)$ we recognize the second parenthesized term to be $T(n/2)$. Therefore,

$$T(n) = (4n + T(n/2)) + (4n + T(n/4) + \ldots + T(1)) = (4n + T(n/2)) + T(n/2) = 2T(n/2) + 4n$$

This is the well-known "mergesort" recurrence ie $T(n) = \Theta(n \lg n)$.
(c) We first observe that $T(n)$ is such that $T(n) \geq n$. We note that the guess and check (substitution method) does not work for $T(n) \leq cn$. A second choice, $T(n) \leq cn^2$ would work, however, as by the inductive hypothesis $T(n/2) \leq c(n/2)^2$ and $T(2n/3) \leq c(2n/3)^2$ and

$$T(n) \leq cn^2/4 + 4cn^2/9 + n = c(25/36)n^2 + n \leq cn^2$$

for large enough n and $c > 0$. It is not, however, the best possible one. **Partial credit will be given to those who suggested such a solution**. This leads us to try a solution $T(n) \leq cn^\alpha$ where α is a constant such that $1 < \alpha < 2$. Similarly as before,

$$T(n) \leq cn^\alpha((1/2)^\alpha + (2/3)^\alpha) + n \leq cn^\alpha$$

The latter inequality holds asymptotically for large n provided that $(1/2)^\alpha + (2/3)^\alpha < 1$. If this value is for example 0.99, then $0.99cn^\alpha + n \leq cn^\alpha$ asymptotically for large n.

A more accurate solution is to try $T(n) \leq cn^\alpha - \beta n$ (β a constant). A similar lower bound can also be proved. For the choice $T(n) \leq cn^\alpha - 6n$, as long as $(1/2)^\alpha + (2/3)^\alpha) \leq 1$, the recurrence can be verified as by the inductive hypothesis $T(n/2) \leq c(n/2)^\alpha - 3n$ and $T(2n/3) \leq c(2n/3)^\alpha - 4n$ and

$$T(n) \leq cn^\alpha((1/2)^\alpha + (2/3)^\alpha) - 3n - 4n + n \leq cn^\alpha - 6n$$

An α such that $(1/2)^\alpha + (2/3)^\alpha = 1$ is one such that $1.293 < \alpha < 1.294$ (α can be found numerically, but the exact value is of no particular interest; **what matters, is that it is a constant between 1 and 2**). Therefore $T(n) = \Theta(n^{O(1)})$, where the exponent of n is some constant between 1 and 2 that can be found as shown above.
(d)

$$\begin{aligned}
T(n) &= 4T(n/2) + n^2/\lg n \\
&= 4(4T(n/4) + (n^2/4)\lg(n/2)) + n^2/\lg n = 4^2T(n/2^2) + n^2/\lg(n/2) + n^2/\lg n \\
&= 4^iT(n/2^i) + n^2/\lg(n/2^{i-1}) + \ldots + n^2/\lg n = 4^iT(n/2^i) + \sum_{j=0}^{i-1} n^2/\lg(n/2^j) = 4^iT(n/2^i) + \sum_{j=0}^{i-1} n^2/(\lg n - j) \\
&= 4^iT(n/2^i) + n^2\sum_{j=0}^{i-1} 1/(\lg n - j)
\end{aligned}$$

We then set $n/2^i = 1$, ie $i = \lg n$. We then get.

$$T(n) = 4^iT(1) + n^2\sum_{j=0}^{i-1} 1/(\lg n - j) = n^2 + n^2\sum_{j=0}^{\lg n - 1} 1/(\lg n - j) = n^2 + n^2\sum_{k=1}^{\lg n} 1/k$$

The sum, from Chapter 3 (equation (3.5)) is equal to $\Theta(\lg\lg n) + \Theta(1)$. Therefore, $T(n) = \Theta(n\lg\lg n)$.
(e) We use the master method. $a = 4 \geq 1$. $b = 3 > 1$. $f(n) = n^{1/3}$ and $n^{\log_b a} = n^{\log_3 4} \approx n^{1.26}$. From case 1 of the master theorem, $f(n) = O(n^{\log_3 4 - \varepsilon})$, by choosing any $0 < \varepsilon < \log_3 4 - 1/3 \approx 0.9$. Therefore $T(n) = \Theta(n^{\log_3 4})$.
(f) We use the iteration method.

$$\begin{aligned}
T(n) &= n + n^{3/4}T(n^{1/4}) \\
&= n + n^{3/4}(n^{1/4} + (n^{1/4})^{3/4}T((n^{1/4})^{1/4})) = n + n^{3/4}(n^{1/4} + n^{3/16}T(n^{1/16})) = n + n + n^{15/16}T(n^{1/16}) \\
&= n + n + n^{1-1/16}T(n^{1/16}) \\
&= 2n + n^{15/16}(n^{1/16} + (n^{1/16})^{3/4}T((n^{1/16})^{1/4})) = 2n + n + n^{63/64}T(n^{1/64}) = 3n + n^{1-(1/4)^3}T(n^{1/4^3}) \\
&= kn + n^{1-(1/4)^k}T(n^{1/4^k})
\end{aligned}$$

We then set $k = \log_4\log_4 n$. Then, $n^{1/4^k} = n^{1/\log_4 n} = 4$. In addition, $n^{1-1/4^k} < n$.

$$T(n) = n\log_4\log_4 n + n^{1-(1/4)^k}T(4) = n\log_4\log_4 n + O(n)T(4) = n\log_4\log_4 n + O(n)$$

Therefore $T(n) = \Theta(n\log_4\log_4 n)$.
g) We use the master method. $a = 9 \geq 1$, $b = 3 > 1$, and $n^{\log_b a} = n^2$. $f(n) = n\lg^2 n$ and positive for $n > 1$. Therefore, $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon = 1/2$. Then, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.
h) We use the master method. $a = 3 \geq 1$, $b = 9 > 1$, and $n^{\log_b a} = n^{1/2}$. $f(n) = n^2$ and positive. Therefore, $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for $\varepsilon = 1/2$. In addition $af(n/b) \leq cf(n)$ for $c = 3/81 < 1$. Then, $T(n) = \Theta(f(n)) = \Theta(n^2)$.
i) We use the master method. $b = 4 > 1$, $a = 2 \geq 1$, and $n^{\log_b a} = n^{1/2}$. $f(n) = n^{1/2}$ and positive. Then, $f(n) = \Theta(n^{\log_b a})$ for all n. Therefore $T(n) = \Theta(n^{\log_b a}\lg n) = \Theta(\sqrt{n}\lg n)$.

(j) We use the master method. $a = 4 \geq 1$, $b = 2 > 1$, $f(n) = n^2$, and $\log_b a = 2$. Then $n^{\lg_b a} = n^2$. As $f(n) = n^2 = \Theta(n^{\lg_b a}) = \Theta(n^2)$, case two of the master method applies, and thus $T(n) = \Theta(n^2 \lg n)$.
(k) We use the master method. $a = 16 \geq 1$, $b = 2 > 1$, $f(n) = n$, and $\log_b a = 4$. Then $n^{\lg_b a} = n^4$. As $f(n) = n = O(n^{4-1}) = O(n^{\lg_b a - 1})$, for $\varepsilon = 1$, case one of the master method applies, and thus $T(n) = \Theta(n^4)$.
(l) We use the master method. $a = 3 \geq 1$, $b = 9 > 1$, $f(n) = n$, and $\log_b a = 1/2$. Then $n^{\lg_b a} = n^{1/2}$. As $f(n) = n = \Omega(n^{1/2+1/2}) = O(n^{\lg_b a + 1/2})$, for $\varepsilon = 1/2$, case three of the master method applies, and thus $T(n) = \Theta(n)$, as in addition $af(n/b) = 3f(n/9) = 3n/9 < cn = cf(n)$ for any choice of c such that $1/3 < c < 1$ (e.g. $c = 1/2$). ■

Exercise 149.

Solve the following recurrence. Make your bounds as tight as possible. Use asymptotic notation to express your solution.

$T(n) = T(n/5) + T(3n/4) + 8n$, where $T(1) = 100$.

Proof.

Since $T(n) = T(n/5) + T(3n/4) + 8n$ we have for $n = 1$ from the base case that $T(1) = 100$. We show that $T(n) = \Omega(n)$. By the recurrence we have that $T(n) \geq 8n$ for all $n > 1$ since $T(n/5), T(3n/4)$ are non-negative and the other term is $8n$. In addition, $T(1) = 100 \geq 8 \cdot 1 = 8$. Therefore $T(n) \geq 8n$ for $n = 1$ as well. Therefore $T(n) = \Omega(n)$.

We now prove that $T(n) = O(n)$ or equivalently that there exist positive constants n_2, c_2 such that $T(n) \leq c_2 n$ for all $n \geq n_2$. Proof is by induction on n. **Base Case.** $T(1) = 100 \leq c_2 \cdot 1$ provided that $c_2 \geq 100$. This completes the base case.

Inductive Step. Let $T(i) \leq c_2 i$ be true for all $1 \leq i < n$. We want to show that $T(n) \leq c_2 n$. Since $T(i) \leq c_2 i$ for all $i < n$, it is also true for $i = n/5$ and $i = 3n/4$ as well, and thus by the inductive hypothesis we obtain $T(n/5) \leq c_2 n/5$ and $T(3n/4) \leq 3c_2 n/4$. Then we use these two inequalities in the recurrence to obtain.

$$\begin{aligned} T(n) &= T(n/5) + T(3n/4) + 8n \\ &\leq c_2 n/5 + c_2 3n/4 + 8n \\ &= 19c_2 n/20 + 8n \end{aligned}$$

In order to complete the inductive step we need to prove that $T(n) \leq c_2 n$ or equivalently $19c_2 n/20 + 8n \leq c_2 n$. The latter is equivalent to $8n \leq c_2 n/20$ or $c_2 \geq 160$. Therefore the inductive step is completed and $T(n) \leq c_2 n$ for any $c_2 \geq 160$.

In conclusion, combining the inductive step and the base case we obtain that $T(n) \leq c_2 n$ for all $n \geq 1$ as long as $c_2 \geq 160$. We can fix $c_2 = 160$ and thus show that $T(n) = O(n)$. Therefore since $T(n) = O(n)$ and $T(n) = \Omega(n)$ we obtain that $T(n) = \Theta(n)$. ■

Exercise 150.

Solve the following recurrence. Make your bounds as tight as possible. Use asymptotic notation to express your solution for each one of them.

$T(n) = T(n/4) + T(2n/3) + 8n$, where $T(1) = 40$.

Proof.

Since $T(n) = T(n/4) + T(2n/3) + 8n$ we have for $n = 1$ from the base case that $T(1) = 40$. We show that $T(n) = \Omega(n)$. By the recurrence we have that $T(n) \geq 8n$ for all $n > 1$ since $T(n/4), T(2n/3)$ are non-negative and the other term is $8n$. In addition, $T(1) = 40 \geq 8 \cdot 1 = 8$. Therefore $T(n) \geq 8n$ for $n = 1$ as well. Therefore $T(n) = \Omega(n)$.

(b) We now prove that $T(n) = O(n)$ or equivalently that there exist positive constants n_2, c_2 such that $T(n) \geq c_2 n$ for all $n \geq n_2$. Proof is by induction on n.

Base Case. $T(1) = 40 \leq c_2 \cdot 1$ provided that $c_2 \geq 40$. This completes the base case.

Inductive Step. Let $T(i) \leq c_2 i$ be true for all $1 \leq i < n$. We want to show that $T(n) \leq c_2 n$. Since $T(i) \leq c_2 i$ for all $i < n$, it is also true for $i = n/4$ and $i = 2n/3$ as well, and thus by the inductive hypothesis we obtain $T(n/4) \leq c_2 n/4$ and $T(2n/3) \leq 2c_2 n/3$. Then we use these two inequalities in the recurrence to obtain.

$$\begin{aligned} T(n) &= T(n/4) + T(2n/3) + 8n \\ &\leq c_2 n/4 + c_2 2n/3 + 8n \\ &= 11c_2 n/12 + 8n \end{aligned}$$

In order to complete the inductive step we need to prove that $T(n) \leq c_2 n$ or equivalently $11c_2 n/12 + 8n \leq c_2 n$. The latter is equivalent to $8n \leq c_2 n/12$ or $c_2 \geq 96$. Therfore the inductive step is completed and $T(n) \leq c_2 n$ for any $c_2 \geq 96$. In conclusion, combining the inductive step and the base case we obtain that $T(n) \leq c_2 n$ for all $n \geq 1$ as long as $c_2 \geq 96$. We can fix $c_2 = 96$ and thus show that $T(n) = O(n)$. Therefore since $T(n) = O(n)$ and $T(n) = \Omega(n)$ we obtain that $T(n) = \Theta(n)$. ■

Exercise 151.

Solve the following recurrence. Make your bounds as tight as possible. Use asymptotic notation to express your solution for each one of them.

$T(n) = T(n/8) + T(7n/8) + n.$

Proof.

If we try $T(n) \leq cn$, we realize this guessed solution does not work. One reason is that $n/8 + 7n/8 = n$; if the sum were dn, where d is a positive constant STRICTLY less than 1, our guess would have worked. We then try a solution $T(n) = O(n \lg n)$, i.e. we assume that there exist positive constants c, n_0 such that $T(n) \leq cn \lg n$ for all $n \geq n_0$. We choose $T(1) = 0$. Then $0 = T(1) \leq c \cdot 1 \cdot 0$, i.e. basis of induction is true. We assume that $T(i) \leq ci \lg i$ for all $i < n$ (inductive hypothesis). We use the inductive hypothesis to bound $T(n/8)$ and $T(7n/8)$, and by way of the recurrence we obtain.

$$\begin{aligned} T(n) &= T(n/8) + T(7n/8) + n \\ &\leq c(n/8)\lg(n/8) + c(7n/8)\lg(7n/8) + n \\ &= cn\lg n + (cn/8)(7\lg(7/8) + \lg(1/8)) + n \\ &\leq cn\lg n + (cn/8)(7\lg 1 + (-3)) + n \\ &\leq cn\lg n - 3cn/8 + n \\ &\leq cn\lg n \end{aligned}$$

provided that $c \geq 8/3$. Therefore $T(n) = O(n \lg n)$. We then prove that $T(n) = \Omega(n \lg n)$ in a similar way. He need to use the following result. The entropy function $H(p)$, is defined for any $0 < p < 1$ as follows

$$H(p) = p\lg p + (1-p)\lg(1-p)$$

It attains its minimum for $p = 1/2$, i.e. $H(p) \geq -1$. We show that for some positive constant c, $T(n) \geq cn \lg n$ for asymptotically large n.

$$\begin{aligned} T(n) &= T(n/8) + T(7n/8) + n \\ &\geq c(n/8)\lg(n/8) + c(7n/8)\lg(7n/8) + n \\ &= cn\lg n + (cn)((7/8)\lg(7/8) + (1/8)\lg(1/8)) + n \\ &\geq cn\lg n + (cn)(-1) + n \\ &\geq cn\lg n - cn + n \\ &\geq cn\lg n \end{aligned}$$

provided that $c \leq 1$ (e.g. $c = 1/2$ is such a choice). Therefore $T(n) = \Omega(n \lg n)$. As a conclusion we proved that $T(n) = \Theta(n \lg n)$, where $c_1 = 1/2$, $c_2 = 8/3$ and $n_0 = 1$. ■

3.4 Recurrences and algorithm analysis: warming up

Exercise 152.

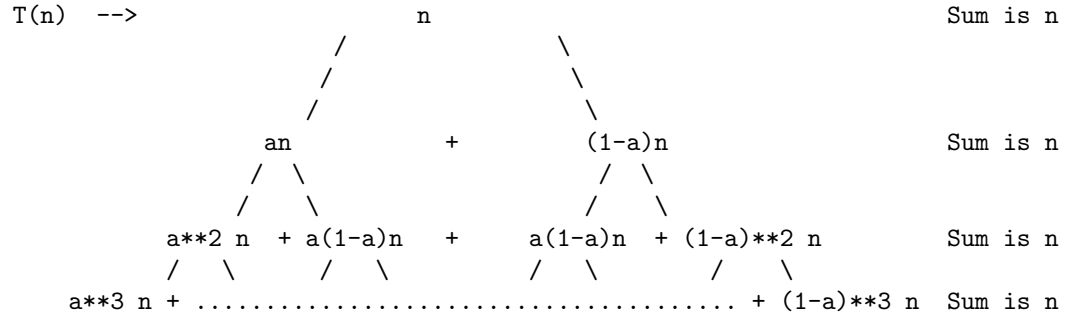
Consider the following variation of MergeSort: Instead of an even split, we split 10-percent on one-size and 90-percent on the other. Then the recurrence for the running time becomes

$$T(n) = T(0.1n) + T(0.9n) + \Theta(n)$$

In general we might have an an split and (1-a)n$splitwhere 0 < a \leq 1/2$. The recurrence then becomes after simplifying the last term (one may use cn instead for some constant $c > 0$).

$$T(n) = T(an) + T((1-a)n) + n$$

Proof. **A solution using the Recursion tree method.** Before you go back to Problem 12 of this Problem Set, note the catch there with $a+b<1$ but in our case $a+(1-a)=1$! This recurrence is easier solved by the recursion tree approach (used in MergeSort and Fibonacci sequence in class). A very flat tree is shown in textual/graphic form below. Isolate the leftmost and the rightmost paths of this tree.
The leftmost path is $n, an, a^2n, a^3n, \ldots$ and the rightmost path is $n, (1-a)n, (1-a)^2n, (1-a)^3n, \ldots$.
How long is the leftmost path? It is L long if $a^L n = 1$. Taking logarithms base two $L\lg a + \lg n = 0$, i.e. $L = -\lg n / \lg a$. Substitute 0.1 for a and we get the desired answer.
How long is the rightmost path? It is R long if $(1-a)^R n = 1$. Taking logarithms base two $R\lg(1-a) + \lg n = 0$, i.e. $R = -\lg n / \lg(1-a)$. Subsitute 0.1 for a or 0.9 for $1-a$ and we get the desired answer.



Remark. As a byproduct of this problem, we conclude that in the context also of the QuickSort algorithm of Subject 6, if there is a PARTITION function (we shall call the implementation of the designated textbook GTP) that always causes a 10-percent and 90-percent split or in the more general case an an-$(1-a)n$ split, then the running time of quicksort that uses this partition is going to be $O(n\lg n)$. This is because we just proved that the depth of the recursion is $O(\lg n)$ and at each level, partition spends total time $O(n)$.

A solution using the Substitution method. We could use the substitution method to solve this recurrence.
(1.) Ω part: $T(n) = \Omega(n\lg n)$ uses a line of argument similar to the O part to follow.
(2.) O part: We show $T(n) = O(n\lg n)$ i.e. there exist positive constant c, n_0 such that $T(n) \leq cn\lg n$, for all $n \geq n_0$ by utilizing strong induction. Since $an < n$ and $(1-a)n < n$ the induction hypothesis applies to $T(an)$ and $T((1-a)n$.

$$\begin{aligned} T(n) &= T(an) + T((1-a)n) + n \\ &\leq can\lg(an) + c(1-a)n\lg((1-a)n) + n \\ &= cn\lg n + cn(a\lg a + (1-a)\lg(1-a)) + n \leq cn\lg n - cnA + n \\ &\leq cn\lg n + (1-c_2A)n \leq cn\lg n \qquad \text{for} \quad c > 1/A. \end{aligned}$$

where we used that the entropy function i.e. the expression $a\lg a + (1-a)\lg(1-a)$ is negative for $0 < a < 1$. Let that negative value be called $-A = a\lg a + (1-a)\lg(1-a)$, where $A > 0$. We obtain the final answer (i.e.

inductive step) by making sure that the term $(1-cA)n$ is negative which is so for $1-cA<0$ i.e. $c>1/A$. Note that A is constant because a is constant. **If a were not a constant, then A would not have been a constant either, and c would not have been bounded by a constant either.**

■

Part II

Searching, sorting and selection

Chapter 4

Elementary data structures and graph representation

4.1 Explain the code

Exercise 153.
What does the following code do? Explain. A, B are bit arrays.

```
  Who(A,B,C,n)
1.  integer c;
2.  c=0;
3.  for i=1 to n do
4.    C[i] = ( A[i] + B[i] ) + c;
      // majority(a,b,c) =1 if at least two of a,b,c are 1; 0 otherwise
5.    c = majority (A[i] , B[i] , c);
6.  endfor
7.  C[n+1] = c;
```

Proof. Self-explanatory.

```
  Sum(A,B,C,n)
1.  integer c; /* the carry of the addition */
2.  c=0;
3.  for i=1 to n do
4.    C[i] = ( A[i] + B[i] ) + c;
      // majority(a,b,c) =1 if at least two of a,b,c are 1; 0 otherwise
5.    c = majority (A[i] , B[i] , c);
6.  endfor
7.  C[n+1] = c;
```

■

Exercise 154.

The following recursive algorithm GOOD (lines $0-6$) is invoked on an array of n keys according to the invocation also shown below `GOOD(A[1..n],1,n`. What is the operation performed by GOOD ? Reason about it. What is the running time of GOOD on an unordered array of n keys? Provide a tight asymptotic bound. Justify your answer.

```
   GOOD(keys A, int p, int r) {
0.  int q;
1.  if (p==r)
2.    return(TRUE);
3.  if (A[p] != A[r])
4.    return(FALSE);
5.  q= p + (r-p)/2;
6.  return (    GOOD(A,p,q) &&    GOOD(A,q+1,r))
    }

1.   GOOD(A[0..n-1],0,n-1); // Invocation of GOOD !
```

What does the following program GOOD do if called on an array of n keys (very last line of the code)? Explain. Analyze its worst-case running time and provide a tight asymptotic bound for it.

Why line 5 uses a $p+(r-p)/2$ instead of $(p+r)/2$?

Proof. The algorithm determines by divide-and-conquer whether all the keys of array A are the same or not.

Split A into two halves. If $A[0]$ and $A[n-1]$ are not equal output FALSE since we know that all the keys are not equal to the same value since $A[0]$ and $A[n-1]$ differ.

Otherwise recur on the left and the right half. Boundary case is the case where a subarray has only one key; return TRUE by default.

The running time recurrence for the running time $T(n)$ is thus $T(n) = 2T(n/2)+1$. Case 1 of the master method gives $T(n) = \Theta(n)$. ■

Exercise 155.

(You might benefit reading some definitions from the next problem, but this one stands independently.)
Before you solve this problem undertand carefully the semantics of the operations. In particular in Delete(S,z) z is a pointer to an element NOT a KEY VALUE. Let n be the size of the data structure before the operation is performed.
(a) Insertion of a key x in a data structure S, i.e. operation Insert(S,x).
(b) Deletion of a record of S pointed by pointer z, i.e. operation Delete(S,z),
(c) Search of a key x, i.e. operation Search(S,x),
(d) FindMin of S i.e. operation FindMin(S),
(e) ExtractMax of S i.e. operation ExtractMax(S),
for the following data structures S.
(1) Unordered array of n keys.
(2) Sorted array of n keys.
(3) Ordered singly linked list.
(4) Ordered doubly linked list.
(5) Ordered doubly linked list with tail pointer.
Note: All keys in S are distinct. If S is sorted, it is sorted in increasing order. If a linked list (singly or doubly linked) is used, assume the existence of a head pointer; the existence or not of a tail pointer depends on the description of the data structure. **Express the running time in terms of the number of elements n in the data structure just before the application of the operation**.

Proof. **DISCLAIMER: CHECK FOR CORRECTNESS i.e. PROOFCHECK...**
For an unsorted singly linked list answers are $O(n)$, $O(1)$ (into the head of the list), $O(n)$ (fix pointer of the previous list element), $O(n)$, $O(n)$, $O(n)$. For a sorted singly linked list answers are $O(n)$, $O(1)$ (into the head of the list), $O(n)$ (fix pointer of the previous list element), $O(1)$, $O(n)$, $O(1)$, $O(n)$ (if a pointer to the tail is available $O(1)$). For an unsorted doubly linked list answers are $O(n)$, $O(1)$ (into the head of the list), $O(1)$, $O(n)$, $O(n)$, $O(n)$, $O(n)$. For an sorted doubly linked list answers are $O(n)$, $O(1)$ (into the head of the list), $O(1)$, $O(1)$, $O(1)$, $O(1)$, $O(1)$. Give the running time for each one of the following five operations. (1) Unordered arrays. Insert at the end of array in $O(1)$ time. Delete in $O(n)$ total time; free entry in $O(1)$ and shift all records on the right of deleted key one position to the left in $O(n)$ additional time for a total of $O(n)$. Search is $O(n)$ worst case. FindMin and ExtractMax are similar to Search and Search followed by a Delete respectively, i.e. $O(n)$ total each. (2) Sorted array. Binary Insert $O(n)$ (requires shift right for records on the right of inserted key even though searching for inserted position requires only $O(\lg n)$). Deletion $O(n)$ as in case 1. Search is $O(\lg n)$ by using binary search. FindMin $O(1)$ (first record, sorted means keys are in increasing order). ExtractMax $O(1)$ as deletion of last key involves no shifts.
(3) Ordered singly linked list. $O(n)$ for Insert, Search, Delete. FindMin $O(1)$ (first record). ExtractMax $O(n)$ (no tail pointer).
(4) Ordered doubly linked list. $O(n)$ for insert, search. $O(1)$ for Delete, and FindMin, and $O(n)$ for ExtractMax (deletion of maximum would take $O(1)$ because it would be pointed through z pointer, since no tail pointer is available however, we need to traverse the list).
(5) Same as in (4) for operations other than ExtractMax. For ExtractMax use tail pointer to report and delete in $O(1)$ total time. ■

4.2 Array and linked list structures: performance and more

Exercise 156.

The following data structures are introduced below. Each linked list structure (SSL, SDL, RSL, RDL below) has a head and tail pointer pointing to the first and last element of the list. Array length can be readily available in constant time $\Theta(1)$. Thus for array A, $A.length$ is readily available and indicates the length of A. Furthermore A.size is readily available and indicates the number of elements stored in A: A.size $\leq$ A.length. For a MinH or MaxH (MinHeap and MaxHeap), MinH.length and MinH.size are likewise defined and similarly for MaxH.
The following data structures are defined.

UA	An unordered array with elements in increasing order
SA	A sorted array with elements in increasing order
RA	A reverse-sorted array with elements in decreasing order
SSL	A sorted singly linked list with elements in increasing order
SDL	A sorted doubly linked list with elements in increasing order
RSL	A reverse-sorted singly linked list
RDL	A reverse-sorted doubly linked list
MinH	A minHeap array whose elements satisfy the MinHeap property
MaxH	A reverse-sorted doubly linked list

We are interested in the performance of the following operations. We assume the above data structures contain comparable keys (keys for which a total order has been defined.) Index or Reference or Pointer z already points to the element that will be deleted or whose predecessor and successor would be found. After a removal or insertion the property of the elements is maintained (e.g. sorted) and the elements are packed without gaps.

Ins(k)	Insert a key k
Sea(k)	Search for a key k
Del(z)	Delete the (record of the)key from location indexed/pointed/ referenced by z
FindMin	Determine (and thus find) and return Minimum of keys
FindMax	Determine (and thus find) and return Maximum of keys
DelMin	Find and delete minimum of keys
DelMax	Find and delete maximum of keys
Succ(z)	Find and return successor of z (no removal)
Pred(z)	Find and return predecessor of z (no removal)

Complete the following table for each data structure indicated and the corresponding operation. An 1 can be $O(1)$ or $\Theta(1)$. Likewise for n, $\lg n$, etc. Use the tightest bound if more than one applicable.

	SA	RA	UA
Ins(k)			
Sea(k)			
Del(z)			
FindMin	1	1	
FindMax	1	1	
DelMin			
DelMax			
Succ(z)			
Pred(z)			

Proof.

	SA	RA	UA
Ins(k)	n	n	1
Sea(k)	$\lg n$	$\lg n$	n
Del(z)	n	n	n
FindMin	1	1	n
FindMax	1	1	n
DelMin	n	1	n
DelMax	1	n	n
Succ(z)	1	1	n
Pred(z)	1	1	n

■

Exercise 157.

Complete the following table for each data structure indicated and the corresponding operation. An 1 can be $O(1)$ or $\Theta(1)$. Likewise for n, $\lg n$, etc. Use the tightest bound if more than one applicable.

The following data structures are defined. Column MinH is related to the next chapter's questions.

UA	An unordered array with elements in increasing order
SA	A sorted array with elements in increasing order
RA	A reverse-sorted array with elements in decreasing order
SSL	A sorted singly linked list with elements in increasing order
SDL	A sorted doubly linked list with elements in increasing order
RSL	A reverse-sorted singly linked list
RDL	A reverse-sorted doubly linked list
MinH	A minHeap array whose elements satisfy the MinHeap property
MaxH	A reverse-sorted doubly linked list

	SSL	SDL	MinH
Ins(k)			
Sea(k)			
Del(z)			
FindMin	1	1	
FindMax	1	1	
DelMin			
DelMax			
Succ(z)			
Pred(z)			

Proof.

	SSL	SDL	MinH
Ins(k)	n	n	$\lg n$
Sea(k)	n	n	n
Del(z)	n	1	$\lg n$
FindMin	1	1	1
FindMax	1	1	n
DelMin	1	1	$\lg n$
DelMax	1	1	n
Succ(z)	1	1	n
Pred(z)	n	1	n

■

Exercise 158.

Reverse a (singly) linked list L. L has a head and tail pointer.

Proof.

```
   Reverse(L.head)  // Call reverse

   Reverse(head) // Reverse list pointed by head
1.   if head == NULL return;      //empty
2.   if head.next == NULL return; //one-element; already reversed
3.   cur =head;
4.   prev=NULL;
     // changes next pointer of cur to point to the previous element prev
     // instead of cur.next
5.   while (cur.next != NULL) {
6.     next = cur.next;
7.     cur.next =prev;
8.     prev=cur;
9.     cur=next;
10.  }
11.  head= prev; //at the end prev is last element; cur is NULL.
12.  return(head)
```

■

Exercise 159.

Show how to implement a queue using two stacks. Analyze the running time of the queue operations. Show how to implement a stack using two queues. Analyze the running time of the stack operations.

```
Enqueue(x)
{
 push(A,x);
}
```

```
Dequeue()
{  item x;
   if isempty(A) and isempty(B)
     return("UNDERFLOW::queue is empty");
   while (isempty(A)==FALSE) {
     push(B,pop(A));
   }
   x=pop(B);
   while (isempty(B)==FALSE) { //ie B has a key
      push(A,pop(B));
   }
   return(x);
}
```

```
Push(x)
{

 Enqueue(A,x);
}
```

```
Pop()
{
  item x;
  if (IsEmpty(A)==TRUE)
    return("Underflow: Empty(A)");

  while (IsEmpty(A)==FALSE) {
    x=Dequeue(A);
    if (IsEmpty(A)==FALSE)
      enqueue(B,x);
  }
```

Proof. We are going to use two stacks, A and B to implement the queue operations. A key x is always inserted by pushing it into stack A. The least recently inserted item, i.e. the next to be dequeued is always in variable x popped off the top of stack B. Therefore Enqueue pushes an element on top of stack A. For a DeQueue operation A is emptied into B, the top of B is popped off and emptied into x, and then B is sent back to A. The running time of an enqueue is $O(1)$ and of a dequeue $O(n)$. A push is implemented by an Enqueue operation. For a pop, we empty all of elements of A (using Dequeue operations) into B except for the last inserted element at the tail of the A queue that is stored in x but not in B. We achieve this by checking every time we Dequeue an element from A into x whether the remaining A is empty or not. If it is not, THEN and ONLY THEN do we enqueue x into B. If the remaining A is empty, x was the last element of A and we eventually return it, after we copy back queue B into A. The running time of a push is $O(1)$ and of a pop $O(n)$.

■

Exercise 160.
How can you implement a stack or a queue if you only have access to a TimeFunction and a PriorityQueue. That is implement Push(S,x) and Pop(S) for inserting x into a stack, and Enqueue(Q,x) and Dequeue(Q) for inserting x into a queue by using a PriorityQueue P and operations Insert(P,x,pr(x)) and ExtractHigh(P,op) where the former inserts into P an element x with priority `pr(x)` and the latter extracts from P the element with the highest priority: if op is `min` it is the lowest `pr(x)` valued element, if op is `max` is the highest `pr(x)` valued element.

Proof. A FIFO (aka queue) means first-in first-out i.e. if you timestamp insertions the lowest time-stamped key is to be extracted first. A LIFO (aka stack) means last-in first-out i.e. the highest time-stamped key is to be extracted first. That Push(S,x) and Enqueue(Q,x) are implemented as Insert(S,x,TimeFunction) and Insert(Q,x,TimeFunction) identically. The only difference is extraction! Pop(S) and Dequeue(Q) are implemented as ExtractHigh(S,max) and ExtractHigh(Q,min) respectively. So stacks and queues are instances of a priority queue or can be implemented as an extension of a priority queue! Operations then are going to be slower than the time operations Push, Pop, Enqueue, Dequeue, require on traditional data structure implementations of a Stack or a (FIFO) queue. ■

Exercise 161.

Explain how to implement two stacks in one array A[1..n] in such as way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations must still run in $O(1)$ time.

Proof. The solution is quite straightforward: the first stack STACK1 grows from the leftmost boundary of the array (that becomes the bottom of STACK1) to the right, whereas STACK2 grows from the rightmost boundary of the array to the left. We may keep track of the sizes of the two stacks (this is one way to solve the problem) in Length1 and Length2 so that Length1+Length2 $\leq n$, where n is the size of array S storing STACK1 and STACK2. As Top1, Top2 pointers are available, we can use them to maintain the invariant Length1+Length2 $\leq n$ as follows.

```
 STACK1-EMPTY (S)        STACK2-EMPTY(S)
1. if Top1[S]=0          if Top2[S]=n
2.       return TRUE           return TRUE
3.  else return FALSE     else return FALSE
```

```
 PUSH1(S,x)                      PUSH2(S,x)
1. if top1[S]+1 >= top2[S]      if top2[S]-1 <= top1[S]
            return overflow       return overflow
2. top1[S]=top1[S]+1;           top2[S]=top2[S]-1;
3. S[top1[S]]=x;                S[top2[S]]=x;
```

POP1, POP2 are similarly defined (pseudocode omitted in this outline). ■

4.3 Properties of elementary sorting algorithms

Exercise 162.

Give a **TRUE** or **FALSE** for each one of the statements below. Any answer other than a full TRUE or FALSE will be considered wrong. All algorithms are the book/notes implementations.

(1) InsertionSort is stable.
(2) InsertionSort is not stable.
(3) InsertionSort sorts in-place.
(4) MergeSort is stable.
(5) MergeSort is not stable.
(6) MergeSort sorts in-place.
(7) SelectionSort sorts in-place.
(8) SelectionSort is stable.
(9) SelectionSort without SlideInsert is stable.
(10) Odd-Even Transposition Sort is stable.
(11) Odd-Even Transposition Sort is not stable.
(12) Odd-Even Transposition Sort sorts in-place.
(13) InsertionSort is an incremental algorithm.
(14) SelectionSort is an incremental algorithm.
(15) MergeSort is an incremental algorithm.
(16) MergeSort is a divide-and-conquer algorithm.

Proof.
TRUE, FALSE, TRUE,
TRUE, FALSE, FALSE,
TRUE, TRUE, FALSE,
TRUE, FALSE, TRUE,
TRUE, TRUE, FALSE, TRUE.

■

Exercise 163.
Give a **TRUE** or **FALSE** for each one of the statements below. Any answer other than a full TRUE or FALSE will be considered wrong. All algorithms are the book/notes implementations.

(1) The best case running time of InsertionSort is $O(n^2)$.

(2) The best case running time of InsertionSort is $O(n \lg n)$.

(3) What algorithmic principle do we use in Binary search?

(4) What algorithmic principle do we use in SelectionSort?

(5) Name an incremental and in-place sorting algorithm introduced in class.

(6) Name 3 incremental algorithms introduced in class.

Proof. TRUE and TRUE ($\Theta(n)$ is $O(n^2)$ and $O(n \lg n)$ respectively),
divide-and-conquer (binary search), incremental approach (SelectionSort), SelectionSort or a safer choice is InsertionSort, one can start with Fibonacci, InsertionSort, and SelectionSort. ■

Exercise 164.

Give a **TRUE** or **FALSE** for each one of the statements below. Any answer other than a full TRUE or FALSE will be considered wrong. All algorithms are the book/notes implementations.

(1) On the increasing input sequence $\langle 1,2,\ldots,n\rangle$, InsertionSort has running time that is $O(n^2)$.

(2) On the increasing input sequence $\langle 1,2,\ldots,n\rangle$, MergeSort has running time that is $\Omega(n)$.

(3) On the increasing input sequence $\langle 1,2,\ldots,n\rangle$, InsertionSort is asymptotically faster than Selectionsort.

(4) On the increasing input sequence $\langle 1,2,\ldots,n\rangle$, SelectionSort is asymptotically faster than MergeSort.

(5) On the decreasing input sequence $\langle n,n-1,\ldots,1\rangle$, InsertionSort has running time that is $\Theta(n^2)$.

(6) On the decreasing input sequence $\langle n,n-1,\ldots,1\rangle$, InsertionSort has running time that is $O(n^3)$.

(7) On the decreasing input sequence $\langle n,n-1,\ldots,1\rangle$, MergeSort has running time that is $O(n^3)$.

(8) On the decreasing input sequence $\langle n,n-1,\ldots,1\rangle$, InsertionSort is asymptotically faster than MergeSort.

(9) The best case running time of InsertionSort is $O(n\lg n)$. TRUE or FALSE?

(10) The best case running time of InsertionSort is $\Theta(n)$. TRUE or FALSE?

(11) The worst case running time of InsertionSort is $O(n^3)$. TRUE or FALSE?

(12) The worst case running time of InsertionSort is $\Theta(n^2)$. TRUE or FALSE?

(13) The best case running time of InsertionSort is $\Omega(n)$. TRUE or FALSE?

Proof.
TRUE (since $\Theta(n)$ is $O(n^2)$), TRUE (since $\Theta(n\lg n)$ is $\Omega(n)$), TRUE, FALSE,
TRUE (indeed it is $\Theta(n^2)$), TRUE (since $\Theta(n^2)$ is $O(n^3)$), TRUE (since $\Theta(n\lg n)$ is $O(n^3)$), FALSE,
TRUE, TRUE, TRUE, TRUE, TRUE.
There is a difference between "running time of X is" and "X has running time that is (of order) that". E.g. best case running time of InsertionSort is $\Theta(n)$. But running time of InsertionSort on a best case input is $O(n^4)$. Why ? Because $\Theta(n)$ is $O(n^4)$. Any linear function such as n and $5n+100$ are: $n=O(n^4)$ and $5n+100=O(n^4)$. ■

Exercise 165.

Provide a short answer to the following questions. The universe of algorithms is what we have done in class so far. Use appropriate asymptotic notation as introduced in class. (And make it as tight as possible.) Make sure you name functions properly and problem size correctly.

(1) Give the best case running time of InsertionSort.
(2) Give the worst case running time of InsertionSort.
(3) What is the running time of InsertionSort?
(4) Give the best case running time of SelectionSort.
(5) Give the worst case running time of SelectionSort.
(6) What is the running time of SelectionSort?
(7) Give the best case running time of MergeSort.
(8) Give the worst case running time of MergeSort.
(9) What is the running time of MergeSort?
(10) What is the best case of InsertionSort?
(11) What is the worst-case of Insertion-sort?
(12) What is the best case running time of Insertion-sort?
(13) Does InsertionSort sort in place?
(14) Is Merge-sort stable?
(15) What is the running time of MergeSort?
(16) Is InsertionSort stable?
(17) Does MergeSort sort in place?

Proof. $T(n) = \Theta(n)$, $T(n) = \Theta(n^2)$, $T(n) = O(n^2)$, $T(n) = \Theta(n^2)$, $T(n) = \Theta(n^2)$, $T(n) = \Theta(n^2)$, $T(n) = \Theta(n \lg n)$, $T(n) = \Theta(n \lg n)$, $T(n) = \Theta(n \lg n)$, SortedSequence, ReverseSortedSequence, $T(n) = \Theta(n)$, YES, YES for the version described, $T(n) = \Theta(n \lg n)$, YES, NO for the version described. ■

Exercise 166.

Provide a short answer to the following questions. The universe of algorithms is what we have done in class so far. Use appropriate asymptotic notation as introduced in class. (And make it as tight as possible.) Make sure you name functions properly and problem size correctly.

1. Give the running time of InsertionSort for sorting $\sqrt{n}$ keys. ($\sqrt{}$ is the square root function.)
2. Give the running time of InsertionSort for sorting $n^2 \cdot \sqrt{n}$ keys.
3. Give running time of InsertionSort of n^2 keys.
4. Give the running time of MergeSort for sorting $n/\lg n$ keys.
5. Give the running time of MergeSort of n^3 keys.
6. Give the running time of BinarySearch on a sorted sequence of $2^m - 1$ keys.
7. Give the running time of BinarySearch on a sorted sequence of $2^{m^2} - 1$ keys.
8. On the increasing input sequence $\langle 1, 2, \ldots, n \rangle$, how many comparisons $C(n)$ does InsertionSort perform?
9. On the decreasing input sequence $\langle n, n-1, \ldots, 1 \rangle$, how many comparisons $C(n)$ does InsertionSort perform?
10. What is the running time of InsertionSort on the n-key sequence $\langle n-1, n, n-2, n-3, \ldots, 2, 1 \rangle$? (All but the first two keys are in decreasing order).
11. What is the running time of InsertionSort on the n-key sequence $\langle 2, 1, 3, 4, \ldots, n-1, n \rangle$? (All but the first two keys are in increasing order).

Proof. $T(n) = O(n)$, $T(n) = O(n^5)$, $T(n) = O(n^4)$,
$T(n) = \Theta(n)$, $T(n) = \Theta(n^3 \lg n)$,
$T(m) = O(m)$, $T(m) = O(m^2)$,
$C(n) = (n-1)$, $C(n) = n(n-1)/2$,
$\Theta(n^2)$; the sorted first two keys have no effect on the number of comparisons over the worst-case input, $\Theta(n)$. ■

Exercise 167.

(1) What is the number of comparisons of InsertionSort on the n-key sequence $\langle 2,3,\ldots,n-1,n,1\rangle$?

(2) What is the number of key comparisons of InsertionSort on the $(n-1)$-key sequence $\langle n-1,n-2,\ldots,2,1\rangle$?

(3) What is the running time of InsertionSort on the n-key sequence $\langle n-1,n,n-2,n-3,\ldots,2,1\rangle$? (All but the first two keys are in decreasing order). do not swap and then we have a worst-case run)

(4) What is the number of key comparisons of InsertionSort on the n-key sequence $\langle 10,10,\ldots,10,10,9\rangle$, which are all 10 but the last one which is a 9? Justify your answer.

(5) What is the running time (and also number of comparisons in the worst case) for an already sorted input when MergeSort is being used? First use an asymptotic bound (and then a tighter/exact bound).

Proof.
(1) The first $n-1$ keys are sorted, thus $n-2$ comparisons are needed, the last key requires $n-1$ comparisons to insert it into the left-most position. Thus $n-2+n-1=2n-3$ comparisons,
(2) The reverse sorted sequence is of length $n-1$. Thus $(n-2)(n-1)/2$ comparisons,
(3) $n(n-1)/2$ comparisons i.e. $\Theta(n^2)$ running time; $n-1,n$ do not swap and then we have a worst-case run,
(4) The $n-1$ keys with value 10 require $n-2$ comparisons to get 'sorted-out' in the first n-2 iterations of InsertionSort (the first iteration is skipped!)! The last key with value 9 requires by itself $n-1$ extra comparisons. The total is $2n-3$.
(5) Hint: Show it is not a ... linear bound! ■

Exercise 168.

In the word model of computation the following operations are allowed and each one is computed in 1operation (1op).

ADD(x,y)	returns $x+y$ in 1op i.e. 1A (A for addition operation)
SUB(x,y)	returns $x-y$ in 1op i.e. 1S (S for subtraction operation)
	From now on 1A/S indicate cost of A or S is the same
MUL(x,y)	returns $x*y$ in 1op i.e. 1M (M for multiplication operation)
SHL(x,k)	returns $x*2^k$ in 1op
SHR(x,k)	returns $x/2^k$ in 1op

Find the n-th Fibonacci number using a Divide and Conquer algorithm in the word model. How inefficient (space or time or both) is it?

Proof.

```
int  fib(n) {
1. if (n==0) return(1);
2. if (n==1) return(1);
3. return(fib(n-1)+fib(n-2));
}
```

The algorithm above is a clear implementation of the fibonacci sequence problem. Function `fib(n)` computes F_n also known as $F[n]$. In the WORD model, we identify the most important operation as the addition performed in Line 3. Thus time becomes equivalent or proportional to the number of additions performed. Let $T(n)$ be the time, counting additions only, to compute $F(n)$ and similarly $T(n-1)$ and $T(n-2)$ the time to compute $F(n-1)$ and $F(n-2)$. Combining all the information derived so far, line 3 of the code yields $T(n)=T(n-1)+T(n-2)+1$. $T(n-1)$ and $T(n-2)$ accounts for the two recursive calls of line 3 and the 1 for the addition of line 3. Note however that $T(0)=T(1)=0$ since no addition is required to find $F[0]$ and $F[1]$ which are directly given as 0 and 1 respectively. These two are known as the boundary cases or initial conditions. In induction-language we called them base case(s). So we need to solve the following recurrence.

$$T(n)=T(n-1)+T(n-2)+1, \text{ **where** } T(0)=T(1)=0.$$

Bear in mind that the n in the recurrence is $n>1$. (A simple inductive argument can show $T(n)=\omega(n)$. A more detailed and tighter bound is derived next.)

Lemma 4.1

The solution of $T(n)=T(n-1)+T(n-2)+1$ with $T(0)=T(1)=0$ is $T(n)=F(n+2)-1$, for $n>1$.

One could rewrite this recurrence $T(n)=T(n-1)+T(n-2)+1$ by adding one to both sides and then using $G(n)=T(n)+1$.

$$\begin{aligned} T(n) &= T(n-1)+T(n-2)+1 \quad \Rightarrow \\ T(n)+1 &= T(n-1)+1+T(n-2)+1 \quad \Rightarrow \\ G(n) &= G(n-1)+G(n-2). \end{aligned}$$

Let's see what happens to $G(0)$ and $G(1)$. We have $G(0)=T(0)+1=1$, and $G(1)=T(1)+1=1$. Thus

$$G(n)=G(n-1)+G(n-2), \quad \text{where} \quad G(0)=1, \quad G(1)=1$$

The Fibonacci recurrence we used is

$$F(n) = F(n-1) + F(n-2), \quad \text{where} \quad F(0) = 0, \quad F(1) = 1, \quad F(2) = 1, \quad F(3) = 2$$

or equivalently

$$F(n) = F(n-1) + F(n-2), \quad F(2) = 1, \quad F(3) = 2, \text{ for } n > 3.$$

It is quite clear that $G(n)$ is $F(n)$ shifted by two i.e. $G(n) = F(n+1)$. Given that $G(n) = T(n) + 1$ this implies that $T(n) = F(n+1) - 1$.

$$T(n) = F(n+1) - 1.$$

If you feel disoriented with the changes of variables, we we can use asymptotic notation that would lead to claim that $T(n) = \Theta(F(n))$ though a $T(n) = \Theta(F(n+2))$ would still be possible. Thus $T(n) \approx ((1+\phi)/2)^n$, or in asymptotic notation Thus $T(n) = \Theta(((1+\phi)/2)^n)$, Since $2^{1/2} \leq \frac{1+\phi}{2} \leq 2$, we also have that $T(n) = \Omega(2^{n/2})$, and $T(n) = O(2^n)$. ■

4.4 Arithmetic, Lionardo Bonacci, and filius Bonacci

Exercise 169.

Checking Legal Parenthesizations.. You are given a string of $n > 0$ left and right parentheses. Give an algorithm (eg. pseudocode) that checks in $O(n)$ worst case running time whether the parenthesization implied by the string is a legal one. Justify your answer. For example $(())$ is NOT a legal parenthesization as we have more right than left parentheses. $))(($ is not legal either as it does not make sense. $(())$, $((())()())$, $()()()$, are legal ones. (6 points)

Proof. Parenthesis checking is a Stack problem. Every left parenthesis causes a push stack operation. Every right parenthesis causes a pop operation provided the stack is not empty. If the stack is empty it means we have encountered fewer left than right parentheses, a problem. A non-empty stack at the end, is also a flag to an error. The running time of the algorithm is $O(n)$, as the input is scanned once for every left or right parenthesis.

```
int IsLegal (A,n) { //A contains n  "(" and ")" parentheses.
  stack S;
  flag  status=OK;
  i=1;

  while ((status == OK) and (i<=n)) {
    if (A[i] ==  "("  //push a left paren
      push(S,"(");
    else
      if (A[i] == ")") { // Attempt to pop a left paren
        if (IsEmpty(S) == TRUE){ //I can't pop: #of left < # of right paren.
          status=PROBLEM;
          return(status);
        }
        else
          pop(S); // Successfully popped a left paren.
      }
    i++; // Get next input.
  }
  if (IsEmpty(S)==FALSE) status=PROBLEM;//If stack nonempty, something's wrong
  return(status);
}
```

■

Exercise 170.

Find the n-th Fibonacci number in $\Theta(n)$ time and $\Theta(1)$ space. Provide an algorithm (pseudocode) and analyze its worst-case running time and space requirements.

Proof.

```
   fib(n)     // F[n] = F[n-1]+ F[n-2] n>1  F[0]=0, F[1]=1
1. a=0;       // a holds  F[0]
2. b=1; c=0;  // b holds  F[1]
3. for(i=2;i<=n;i++){
4.  c=a+b;    // c computes the next fibonacci
5.  a=b; b=c;
6. }
7. print(c);
```

Details missing. Space used is $\Theta(1)$. Time required is $\Theta(n)$. Note that the time bound can be decreased further to $O(\lg n)$ ■

Exercise 171.

Improve the recursive version of the fibonacci sequence problem.

Proof.

```
int  fibmem(n) {
1. if (n==0) return(1);
2. if (n==1) return(1);
3. return(fib(n-1)+fib(n-2));
}
```

We use the principle (of algorithm design) of dynamic programming know also as memoization in this setting. The recursive algorithm invokes $fib(i)$ several times. One can prove by induction that $fib(i)$ is invoked $F[n+1-i]$ times (thus $fib(n), fib(n-1)$ are invoked once, $fib(n-2)$ twice, and so on). Therefore we compute $fib(i)$ once, store the result in an array and subsequence invocations of $fib(i)$ are resolved by retrieval of the value stored.

```
0.      F(n)    // Memoization
1.      table int T[0..n];
2.      T[0]=0; T[1]=1; T[ 2<=i <=n] = UNDEFINED;
3.      return(TableRecursive_Fn(n,T));

0.     int TableRecursive_Fn(n,T);
1.      if (n==0) return (0);//F[0]
2.      if (n==1) rerurn (1);//F[1]
3.      if T[n-1] != UNDEFINED left= T[n-1];
4.         else                left= T[n-1]= TableRecursive_Fn(n-1,T);
5.      if T[n-2] != UNDEFINED right= T[n-2];
6.         else                right=T[n-2]= TableRecursive_Fn(n-2,T);
7.      return(left + right);
```

■

Exercise 172.

Find the n-th Fibonacci number in $\Theta(\lg n)$ time and $\Theta(1)$ space. Provide an algorithm (pseudocode) and analyze its worst-case running time and space requirements.

Proof.

```
Let  t[n] be the vector t[n] = [F[n] F[n-1] ]'.
Then
   t[n] =  [ 1 1 ] t[n-1] or equiv [ F[n] ] = [ 1  1 ]  [F[n-1]]
           [ 1 0 ]                 [F[n-1]]   [ 1  0 ]  [F[n-2]]

Then t[n]  = A . t[n-1]
    t[n-1] = A . t[n-2]
     ...   =   ...
    t[2]   = A . t[1] i.e.   t[n] = A**(n-1) . t[1]
But t[1] = [ F[1] F[0]]' = [1 0 ]'
```

By way of $t_n = A^{n-1}t_1$, we can compute t_n in $O(\lg n)$ with repeated doubling. Note that the x of repeated doubling is A. Thus 1M for say $x*x$ becomes in the context of matrix multiplication of 2×2 matrices, 8M and 4A respectively. Asymptotically speaking $O(\lg n)$ is still the time to compute A^{n-1}. ■

Exercise 173.

Show how to compute x^n in at most $2\lg n$ Ms in the word model. The input is x and integer $n > 1$.

Proof.
Let $k = \lfloor \lg n \rfloor$.
Step 1. We first compute k powers $y_i = x^{2^i}$, $i > 0$, with $y_0 = x$. (`Count1`). Note that $y_k \leq x^n \leq y_k^2$ implies $k \leq \lg n$.

$$y_0 = x = x^1, y_1 = x^2, y_2 = y_1^2 = x^4, y_3 = y_2^2 = x^8, \ldots y_k = y_{k-1}^2 = x^{2^k}$$

Step 2. Then we get the binary representation of n. ($|n| = k+1$.) Let it be $n = b_k 2^k + \ldots + b_1 2^1 + b_0 2^0$, where $b_i = 0, 1$.

$$x^n = x^{b_0 + b_1 2^1 + \ldots + b_k 2^k} = x^{b_0} x^{b_1 2^1} \ldots x^{b_k 2^k}$$

Let m of the b_i's that are one be $b_{l_1}, b_{l_2}, \ldots, b_{l_m}$, where $m \leq k+1$.
Step 3. In the next step we extract the y_j whose corresponding b_j is one. Those y_j's are all multiplied together to yield x^n. This is because $x^n = x^{2^{l_1}} x^{2^{l_2}} \ldots x^{2^{l_m}} = y_{l_1} y_{l_2} \ldots y_{l_m}$, i.e. x^n requires $m - 1 \leq k + 1 - 1 = k$ multiplications (`Count2`) in Step 3. The total multiplication Count is 2k i.e. $2\lfloor \lg n \rfloor \leq 2\lg n$. In the word model, x^n can be evaluated in at most $2\lg n$ (scalar) multiplications. (Other operations are not counted.)

```
   Power( x, n)
1.  k = floor(lgn); X[0]=x;
2.  for (i=1;i<=k;i++)
3.   X[i] = X[i-1]*X[i-1] // X[i]=x**(2^i)
4.  n= B[k]B[k-1]...B[0] //bin(n)
5.  if (B[0] == 1)  P=X;
6.   else           P=1;
7.  for(i=1;i<=k;i++);
8.   if (B[i]==1)  P= P* X[i];
9.  return(P);
```

```
  Power(x,n)
1. if (n==0) return(1);
2. if (n==1) return(x);
3. if mod(n,2) ==1
4. return(x*Power(x*x,(n-1)/2));
5. else
6. return(  Power(x*x,n/2));
```

■

4.5 Combine two algorithm design techniques or methods

Exercise 174.

Incremental and Divide-and Conquer.

```
    Something (A,nsize)
0.     if  (nsize == 1) return;
1.     Something(A,nsize-1);
2.     key= A[nsize];
3.     i=nsize-1;
4.     while ( (i>0) and (A[i] > key) )
5.       A[i+1]= A[i]
6.       i=i-1
       }
8.     A[i+1]=key;
```

(a) We invoke function `Something` with `Something(A,n)` where $A[1..n]$ is an array of n keys on which a total order is defined (aka key s are comparable). Explain what `Something` does succinctly and with sound arguments.

(b) Let $T(n)$ be the running time of `Something` on an array A of n keys. Provide a recurrence relation for the running time $T(n)$. Justify your arguments. Use asymptotic notation as needed carefully and correctly.

(c) What is the solution of $T(n)$ from part (b)? Use as tight asymptotic notation as needed.

Proof. It is a recursive version of InsertionSort. We shall prove that this is so by induction. Base case is a problem of size 1 and it is so sorted correctly by line 0 of `Something`. Line 1 operates on a problem size of one less than the original i.e. `nsize-1` vs `nsize`. By the induction hypothesis Line 1 sorts the first `nsize-1` keys. Lines 2-8 is the lines of InsertionSort that insert the `nsize`-th input key into the sorted sequence of the first `nsize-1` keys. As this is the major loop of InsertionSort (iteration version) no proof is needed thus for Lines 2-8. Inductive step completed and so is the induction. Obviously $T(n) = T(n-1) + O(n)$ and this gives $T(n) = O(n^2)$. Further details omitted.

■

Exercise 175.

You are give a sequence A of n keys. Determine whether there exist duplicate keys in the sequence by an algorithm whose worst case running time is $o(n^2)$. (o is a little oh). Your algorithm should answer a YES or NO depending on whether duplicate keys exist or not. **Example.** Sequence $A = \langle 3,5,6,8,0\rangle$ has no duplicate keys but $A = \langle Tony, Jim, Adam, Jim\rangle$, has duplicate keys (e.g. Jim appears twice).

Proof. Sort the keys; then compare adjacent keys in the sorted sequence. If two of them are equal say YES, else if the end is reached, say NO. Running time is that of the sorting operation; lines 2-5 are $\Theta(n)$. Running time is $o(n^2)$ if MergeSort or similar algorithms are used.

```
  DUP(A,n) // A[0..n-1]
1. MergeSort(A,n);
2. for(i=0; i < n-1 ; i++)
3.  if A[i] == A[i+1]
4.   return(YES)
5  return(NO);
```

■

Exercise 176.

You are give a sequence A of n keys. Determine whether there exist triplicate keys in the sequence by proposing an algorithm whose worst case running time is $o(n^2)$. (o is a little oh). Your algorithm should answer a YES or NO depending on whether triplicate keys exist or not. **Example.** Sequence $A = \langle 3,5,6,8,0\rangle$ has no triplicate keys but $A = \langle Tony, Jim, Jim, Adam, Jim\rangle$, has triplicate keys (e.g. Jim appears thrice).

Proof. Sort the keys; then compare adjacent keys in the sorted sequence. If three consecutive ones are equal say YES, else if the end is reached, say NO. Running time is that of the sorting operation; lines 2-5 are $\Theta(n)$. Running time is $o(n^2)$ if MergeSort or similar algorithms are used.

```
   Triplicate(X[1..n],n)
1. MergeSort(X,n);
2. for(i=1;i<=n-2;i++) {
3.  if (X[i] == X[i+1]) && (X[i+1] == X[i+2]))
4.    return(YES);
5. }
6. return(NO);
```

■

4.6 More on other elementary topics

Exercise 177.

What is the worst possible number of comparisons for merging two sorted sequences of n keys? Argue about the number and show two such sequences.
What if we have two sorted sequences of 1 and $n-1$ keys?

Proof. Let the two sequences to be merged be $A = \langle a_1, \ldots, a_n \rangle$ and $B = \langle b_1, \ldots, b_n \rangle$. The worst possible case is when $a_1 < b_1 < a_2 < b_2 < \ldots < a_n < b_n$. The position of b_i can only be determined after comparing b_i to a_{i-1} and a_i, except of course for b_n (we compare it only to a_n). Total number of comparisons performed is at least $2n-1$. If b_i is NOT compared to a_{i-1} is not performed $a_{i-1}\, b_i$ is possible as an outcome and so is $b_i\, a_{i-1}$, as there has been no comparison that excludes the latter arrangement. Of course the latter arrangement is impossible (ie it results in non-sorted output). Similar conclusions are derived if the second comparison of b_i is not performed.
For the other case, do a binary search of the smaller to the larger, and in at most about $\lg n$ (ignore ceilings or floors) you can merge the asymmetrical length sequences. Note that merging would still need $\Theta(n)$ copy operations but this is beyond the scope of the question. ■

Exercise 178.

You are given n keys each one of which has a key value A,B,C, or D (say A means freshperson, B sophomore and so on). The keys are stored in array X. Sort the keys in worst-case linear time $\Theta(n)$ so that in the output sequence the As are before (on the left of) the Bs, which are before the Cs, which are before the Ds. **Variation.** Every key $X[i]$ has a key value $X[i].key$ and key info $X[i].info$. Do the same. You are not allowed to separate key values from key info. Is it in-place? This becomes an interview question!

Proof. We count the As, Bs, Cs and Ds. We go through the keys once ($O(n)$ time) and count the 1's etc in lines 2-7. Then we print enough As etc in linear time $O(n)$ (lines 8-16). The solution is very simple, very inelegant but it works! More on this problem called count-sort later in class. The version shown is not stable; think about generating a stable version of such an algorithm.

```
  Count1234 (X,n)
1. c1=c2=c3=c4=0;
2. for(i=1;i<=n;i++) { //Very naive but it works!
3.  if (X[i] == A) c1++;// We don't care here about syntactice correctness cf A vs 'A'
4.  if (X[i] == B) c2++;// How can you optimize these rather sloppy lines 3-6?
5.  if (X[i] == C) c3++;
6.  if (X[i] == D) c4++;
7. }
8. c=1;                  //  What can you do to optimize lines 9-17?
9. for(i=1;i<=c1;i++) // print as many  As as c1
10.  X[c++]=A;
11.for(i=1;i<=c2;i++) // print the  Bs
12.  X[c++]=B;
13.for(i=1;i<=c3;i++) // and the  Cs
14.  X[c++]=C;
15.for(i=1;i<=c4;i++) // and the  Ds
16.  X[c++]=D;
17. return(X)
```

■

Exercise 179.

Can you sort 4 keys such as a,b,c,d with at most 5 comparisons in the worst case? If yes, show your algorithm. Assume all keys are distinct, if it helps.

Proof. The problem is tricky. One cannot use asymptotically optimal sorting algorithms. A non-optimal algorithm such as InsertionSort might be more suitable because of the small number of keys. For example, the worst-case comparison bound of InsertionSort is $n(n-1)/2$ which means for four keys only $4(4-1)/2 = 6$ and for three keys only $3(3-1)/2 = 3$ comparisons are needed. The idea to achieve the (comparison-based lower bound of 5 comparisons is to do an InsertionSort for the first three keys and what is known as BinarySearch Insertion for the fourth key. Roughly instead of doing linear search from right to left, as we do in InsertionSort to find the finalized position of the i-th input key, we do a BinarySearch instead involving the fourth keys on the sorted first three keys. Note that we are dealing with comparisons not swap operations.
Let's say our keys are a,b,c,d and are all distinct. We first sort with InsertionSort a,b,c and and let the keys in sorted order be y_1,y_2,y_3. This requires three comparisons in the worst case. For key d we do a binary search insertions on $\langle y_1,y_2,y_3\rangle$: it involved a fourth comparison between $d : y_2$ and then the fifth comparisons is beween $d : y_1$ or $d : y_3$ depending on the outcome of the fourth comparison. We can remove this assumption (distinctness) but let's avoid headaches for the time being. A more detailed description follows.
Comparison 1. We first compare a,b using our first comparison. The output (left-to-right smallest-to-largest) is either a,b or b,a depending on this outcome. Let us call this output generically x_1,x_2.
Comparisons 2,3. Now that we have settled a versus b, we deal with c. For c we insert c in the sorted sequence of a,b denoted x_1,x_2. That is, we first compare c to x_2 and then if necessary c to x_1. This requires one or two comparisons, i.e. two in the worst-case. Why? If c is greater than x_2 we are done: the output is x_1,x_2,c. Otherwise we need to determine whether c goes to left or right of x_1 by utilizing comparison 3. Let us again call the three sorted keys y_1,y_2,y_3, a permutation of a,b,c.

Comparison 4,5. In the final step we binary search/insert d into the sorted sequence y_1,y_2,y_3. We can use again 2 comparisons, as many as previously, even if the keys are three and not two. How? We first compare d to y_2 (comparison 4). If d is greater than y_2, we compare d to y_3 (comparison 5) and thus d is inserted either on the left of y_3 or on the right depending on whether d is smaller or larger than y_3. If however the result of comparison 4 says d is less than y_2, then in comparison 5 we compare d to y_1 and decide whether to insert on the left or right of y_1.
In 5 comparisons we managed to sort 4 keys in the worst-case. This is the best we can. The lower bound for sorting dictates that we need in the worst-case a number of comparisons that is the smallest integer greater than $\lg(4!)$.

■

Exercise 180.
Can you sort 5 keys with 7 comparisons in the worst case ? Explain. Is it optimal? Explain.

Proof. No comment! Think! We solved earlier the 4 key case. Merging is the theme for this case. A solution will wait until some time later.

■

Exercise 181.

This is a decision problem expecting a YES or NO answer. The input consists of two sorted sequences of n integer keys each, $A[0..n-1]$ and $B[0..n-1]$ and integer t. The output is a YES if there exist $a \in A$ and $b \in B$ such that $t = a + b$ and NO otherwise. You may assume all the keys are distinct. Give a linear time algorithm that solves this problem.

Proof. We scan A and B left-to-right and right-to-left respectively.

Consider $A[0]$ and $B[n-1]$. Take the sum $x = A[0] + B[n-1]$. If $x = t$ we are done and report a YES. If $x < t$ then consider $y = A[1] + B[n-1]$, where $y > x$. If $x > t$ then consider $z = A[0] + B[n-2]$, where $z < x$.

```
   Find(A,n,B,n,t)
1.  i=0; j=n-1; setflag = FALSE;
2.  while (i < n && j >= 0 ) {
3.    if ((A[i] + B[j]) == t) {
4.          setflag = TRUE;
5.          break;
6.    }
7.    else if ((A[i]+B[j]) < t) i++;
8.    else                      j--;
9.  }
10. if (setflag == TRUE)  print(A[i],B[j],t);
```

■

Exercise 182.

Divide-and-Conquer MIN-MAX. Give a divide-and-conquer algorithm that finds both the MINIMUM and the MAXIMUM of n keys $x_1, x_2, \ldots, x_n$ by performing at most $3n/2-1$ comparisons. You may assume that n is a power of two.

Proof.

```
 MinMax(A,1,n)

  // MinMax(A,l,r) returns (min,max) ,
  // min=min(A[l..r]), max=max(A[l..r]),
  // for the subequence of A between A[l] and A[r]
  // (inclusive) of size r-l+1
  // A a sequence of size n,  n is a power of two ie n=2**k.
  MinMax(A,l,r)
1. if lengthof(A[l..r]) is 2
2.   if (A[l] <= A[r]) return (A[l],A[r])
                else   return (A[r],A[l]);
   }
   // Split A into two halves as in mergesort
                // Call this half A[l...(l+r)/2] A1
3. (min1,max1) = MinMax(A,l,(l+r)/2);
                // and this one                 A2
4. (min2,max2) = MinMax(A,(l+r)/2+1,r);
   // and recursively find min,max of two halves.
   // Compare minima of two halves to find minimum
5. if min1 <= min2  then min= min1;
                    else min= min2;
   // Compare maxima of two halves to find maximum
6. if max1 <= max2  then max= max2;
                    else max= m
```

Split the keys into two equally-size halves. Find the MIN-MAX of each half recursively. The in two comparisons you find the minimum of the two minima and the maximum of the two maxima. They give the global minimum and maximum respectively. The top-level recursive call is MinMax(A,1,n).

The number of **key** comparisons $T(n)$ of `MinMax(A,1,n)` is derived as follows. Step 2 requires one comparison; A is split as in lines 2-4 of MergeSort (page 13). Step 3 requires $T(n/2)$ comparisons and so does Step 4. We call the half of A in step 3, $A1$ and that of step 4 $A2$. In steps 5 and 6 we perform a total of two (2) comparisons. Note that the minimum of A is the minimum of the minima of A1 and A2. and the minimum of A is the maximum of the maxima of A1 and A2. We don't need to compare the minima to the maxima. Therefore, the recurrence for the number of comparisons becomes:

$$T(n) = 2T(n/2)+2 = 2(2T(n/4)+2)+2 = 2^2T(n/2^2)+2^2+2 = 2^{\lg n-1}T(2)+2^{\lg n-1}+\ldots+2 = 3n/2-2.$$

with $T(2)=1$ by way of lines 1-2. By the iteration/recursion tree method we obtain that $T(n)=3n/2-2$, as required. [Note that the solution of the recurrence was compressed into one line.]

■

Exercise 183.

MIN-MAX revisited. Previously you used a divide and conquer approach to solve this problem. Now you are asked NOT to use such an approach and moreover not to assume that n is a power of two. Solve this problem **again** but for a $\lceil 3n/2 \rceil - 2$ bound.

Proof. Distinguish two cases : odd n and even n respectively, i.e. $n = 2k$ and $n = 2k+1$. I'll do the easy one. I am skipping details: pair the first two keys , then the next two and so on. As $n = 2k$ we have k pairs. With one comparison per pair we know the min and the max per pair. Group the k mins in A and the k max in B. Find the minimum of A in $k-1$ comparisons and return it and the max of B in $k-1$ comparisons and return it. Done (the returned values are the global min and max respectively). Total number of comparisons is $k+(k-1)+(k-1) = 3k-2$. Since $n = 2k$, this is also $3k-2 = 3n/2-2 = \lceil 3n/2 \rceil - 2$ as $3n/2$ is an integer! You do the odd case!

```
  FindMinMax(A,n)  // A[0..n-1] indices below might be 'problematic'
1. for(i=0 ; i < n/2; i++) {
2.  if (A[2*i] <= A[2*i+1]) {              // floor(n/2) comps
3.    min(i)= A[2*i]   ; max(i)= A[2*i+1]
4.  }
5.  else {
6.    min(i)= A[2*i+1] ; max(i)= A[2*i]
7.  }
8.  m= FinMin(min[],n/2);                 // floor(n/2)-1 comps
9.  M= FinMax(max[],n/2);                 // floor(n/2)-1 comps
10. if odd(n) {
11.  if ( A[n-1] < m )  m= A[n-1];
12.  if ( A[n-1] > M )  M= A[n-1];        // 2 more
13. }
```

■

Exercise 184.

Consider a modification of merge-sort in which out of n keys n/k subsequences of length k are formed, and each one is sorted by insertion sort and then all subsequences are merged. The optimal k will be determined after analyzing the worst case performance of the algorithm. In other words, this new algorithm works as follows.
(i) Form n/k subsequences each of length k. Sort each one with insertion sort. What is the overall running time for sorting all n/k subsequences using insertion sort in asymptotic notation? Explain.
(ii) The n/k sorted subsequences of length k are to be merged in $\Theta(n\lg(n/k))$ time. Show how.
(iii) Show that the running time of the whole algorithm (steps (i) and (ii)) is $O(nk+n\lg(n/k))$.
(iv) What is the (asymptotically) largest value of k for which this algorithm is as good as merge-sort? Explain.

Proof.

```
[ Figure 1.]                                              Level Size/Seq*#ofSeq.
                          (n)                                 0  n/2**0*2**0 = n
            (n/2)                                 (n/2)       1  n/2**1*2**1 = n
    (n/4)          (n/4)              (n/4)         (n/4)      2  n/2**2*2**2 = n
(n/8) (n/8)    (n/8)  (n/8)       (n/8) (n/8)   (n/8)  (n/8)  3  n/2**3*2**3 = n


                  ....... intermediate levels

          (4k)                                                ...  4k  * n/(4k)= n
    (2k)          (2k)                                  lgn-lgk+1  2k  * n/(2k)= n
 (k)   (k)     (k)   (k)                                  lgn-lgk     k  * n/k   = n
```

(i) Each sequence of length k is sorted by InsertionSort in $O(k^2)$ time. We have n/k sequences, each one requires $O(k^2)$ time, therefore the overall running time for is as follows.

$$(n/k)\cdot O(k^2) = O(nk).$$

(ii) The merging operation resembles the merging recursion tree of MergeSort: it has n/k leaves rather than the n leaves of MergeSort. In other words, $\lg k$ levels of the MergeSort tree are ignored, as the merging starts after step (i) involving sorted sequences of length k. Thus thre tree in this merging operation has $\lg(n/k) = \lg n - \lg k$ levels only. With reference to Figure 1., at level i of the recursion we have 2^i sequences each one consisting of $n/2^i$ keys.
The base case is for $i = \lg(n/k) = \lg n - \lg k$, where n/k subsequences have k keys each, that have been sorted by InsertionSort in step (i).
The 2^i subsequences of level i are matched in pairs and then the pairs of subsequences merged. Each pair is merged in at most $2\cdot(n/2^i)-1)$ comparisons (forget about the -1) and since the number of pairs (half the number of sequences) is $(1/2)2^i$ the total cost of merging at level i is at most $n-2^{i-1}$ comparisons i.e. at most n comparisons. There are $\lg(n/k)$ levels thus the total cost of merging is $O(n\lg(n/k))$. (The upper bound on the number of comparisons is $n\lg(n/k)-2n/k+1$.)
iii. The running time of parts (i) and (ii) give a total running time of $O(nk+n\lg(n/k))$.
iv. We choose k so that $nk = n\lg(n/k)$ i.e. $k = \lg(n/k)$, which gives that $k = \Theta(\lg n)$. Then the performance of the algorithm is $O(n\lg n)$.

■

Exercise 185.

Professor I. M. Nuts invented the following 'sorting' algorithm. He calls it NutSort The pseudocode for sorting n keys is given below. You may assume n is a multiple of 3. Prove that it sorts; analyze its running time.

```
   NutSort(A[0..n-1],n)
1.  NutSrt(A,0,n-1);

   NutSrt(A,l,r);
1. if A[l]>A[r] swap(A[l],A[r]);
2. if (l+1) >= r return;
3. k= floor( (r-l+1)/3);
4. NutSrt(A,  l, r-k ) ;  // First  2/3 of the keys
5. NutSrt(A,l+k, r  )  ;  // Second 2/3 of the keys
6. NutSrt(A,  l, r-k ) ;  // First  2/3 of the keys
```

Proof. Let us call A_1, A_2, A_3 the subsequences $A_1 = A[0..n/3-1]$, $A_2 = A[n/3..2n/3-1]$, $A_3 = A[2n/3..n-1]$. Let $B = A[0..2n/3-1]$ and $C = [n/3..n-1]$. In other words A_1, A_2, A_3 are the first, second and third one-third of A, and B the first (leftmost) two-third of A and C the last (rightmost) two-third of A.

When line 4 is completed the largest elements of B are in A_2. The elements in A_2 along with the inactive (during line 4) elements of A_3 are then nut-sorted in line 5. After line 5, the largest elements of A are in A_3 as they should be. The remaining keys are in area B: they get sorted out in line 6. Lines 1-2 deals with some base case(s). The recurrence for the running time is $T(n) = 3T(2n/3) + \Theta(1)$.

■

Exercise 186.

We have an $n \times n$ matrix A. Its rows are sorted left-to-right smallest to largest and so are its columns top-to-bottom. You may assume keys are distinct. A key k is given. Determine whether key k is in A using a time efficient method.

Proof.
(a) Since each row is sorted, search for k in each row by binary search. Symmetrically, you can also do a binary search in each column. The running time of a binary search is $O(\lg n)$, so the entire algorithm (over all rows) takes $O(n \lg n)$ time.
(b) We can do it in $O(n)$ time. We begin with the top right element M of A. If k is equal to M, we are done. If k is less than M, because every element in the last column is at least M, we denitely can not find k among them. Therefore, we eliminate the last column. Symmetrically, if k is greater than M, the left part of the row is eliminated. Repeat this process until k is found or all rows and columns are eliminated. Since each time we eliminate one row or one column by one comparison, and there are $2n$ rows and columns altogether, the running time of this algorithm is $O(n)$. ■

Exercise 187.
You are given two sorted sequences of comparable keys A and B. The length of A is $\lg n$ and the length of B is n. Find the intersection T of A and B, that is $T = A \cap B$. Provide a time efficient algorithm.

Proof. Obviously T cannot be larger than A, the smallest of the two sequences. In addition T should be a subset of A. Therefore we try in turn to determine whether every element in A is in B. If this is the case we add that element of A in T. Otherwise we do not add it to T. To determine whether an element of A is in B we use binary search. This is possible since A and B are already sorted. Binary search of a single element of A into B takes time $O(\lg n)$ since B has n elements. Since A has $\lg n$ elements the total time for binary search of A (every element of) into B is $|A| \times \lg |B|$ which is $\lg n \times O(\lg n) = O(\lg^2 n)$. Note that if we pick the elements of A left-to-right, set T that is being formed from A is also sorted by default. Note what happens if we do the binary search the other way B into A. It is more time consuming. An alternative approach would be to merge A and B. The duplicate elements of the merged sequence denote an element of T (both A and B contributes one into the merged sequence). This approach would have cost $\Theta(n)$ though, too much relative to the binary search approach. ■

Exercise 188.
You are given three sorted sequences of comparable keys A, B, and C. The length of A is $\lg n$ and the length of B is n and the length of C is n^2. Find the intersection T of A, B, and C, that is $T = A \cap B \cap C$. Provide a time efficient algorithm.

Proof. Use the previous method to derive an $O(\lg^2 n)$ time bound algorithm. ■

4.7 Elementary arithmetic time complexity

Exercise 189.

You are given two or more (unsigned i.e. non-negative) integers A, B such that $A,B < N$. All questions are for the bit model of computation.

Note: Do not think of A, B as being `int A, B;` of C or C++ or Java. Think of them as being `LongInteger A,B;` or `BigInteger A,B;`

Give the 'running time' of the following arithmetic operations. You do not need to use any 'special' notation. You may say constant, linear, log-linear, quadratic or logarithmic to a parameter, which is usually the problem size.

Constant means 1, 10, 1000, 10000000000000000000000000000, 100000000.000001. Thus linear to k might mean k, 5k, 100k+120 etc. Quadratic to p might mean p**2, 2 p**2 , 1000 p**2 -10p+ 50, etc. But note that linear to lgn might mean lgn, 5lgn, 100lgn+120 etc, and

Your universe of algorithms is your elementary school, high school, may be college repertoire (collection).

As noted, integer below means unsigned integer.

Problem 1 (addition).
Input: Two integers A,B such that $A > B$.
Output: Integer S such that $S = A + B$.
Problem 2 (multiplication).
Input: Two integers A,B such that $A > B$.
Output: Integer M such that $M = A * B$.
Problem 3 (verification of product).
Input: Three integers A,B,P such that $P > A > B$.
Output: TRUE if $A * B == P$ and FALSE if $A * B! = P$.
Problem 4 (two factors).
Input: Integer N that has two factors A,B other than 1 and N. One can assume that they are prime numbers.
Output: The factors A and B of N such that $N = A * B$.

Proof.
Problem 1. Elementary school addition solves this problem in $O(\lg A + \lg B) = O(\lg A)$ given that $A > B$. Let $|.| = bit(A)$ indicate not the absolute value but the number of bit of A. If $n = |A|$ and since $A > B$ we have $n = |A| > |B|$. Then n, the number of bit of A, or an upper bound on the number of bit of B is $n = \lceil \lg(A+1) \rceil = \lfloor \lg A \rfloor + 1 = \Theta(\lg A)$. We may also say that elementary school addition is $O(n)$. In general the two bounds are tight and thus one can also say that elementary school addition is $\Theta(\lg A)$ or $\Theta(n)$, where $n = |A| = \lceil \lg(A+1) \rceil = \lfloor \lg A \rfloor + 1$.
Problem 2. Elementary school multiplication solves this problem in $O(\lg A * \lg B) = O(\lg^2 A)$ given that $A > B$. Using the notation of Problem 1, this is $O(n^2)$. The bounds are tight and thus $\Theta(\lg^2 A)$ or $\Theta(n^2)$ can be used.
Problem 3. Elementary school multiplication and subtraction (which is addition) solves this problem. In order to verify $A * B == P$ it suffices to check whether $A * B - P == 0$ instead. This requires one multiplication and one subtraction (i.e. addition). Combining Problem 1 and Problem 2, we have $\Theta(\lg^2 A)$ or $\Theta(n^2)$ for the running time of this approach.
Problem 4. If we have $Z * Z = Y$ i.e. Y is the square of Z, or $Z = \sqrt{Y}$, then $\lg Y = 2 \lg Z$ and this also indicates that the number of bit of Z are double those of Y. Equivalently the number of bit of Z are half of those of Y. We do not get into details of floor, ceilings or plus one. We can add one to our estimate if it is a lower bound and turn into the answer or an upper bound.
Let integer n be such that $n = \lceil \lg(N+1) \rceil = \lfloor \lg N \rfloor + 1$, that is n is the number of bit for N. One of A,B is at most $\sqrt{N}$ since otherwise if both are greater than $\sqrt{N}$, $A * B > N$ and $A * B = \sqrt{N}\sqrt{N} = N$, a contradiction. First find an approximation to $S = \sqrt{N}$ by shifting N right $n/2$ positions, that takes linear to n time i.e. $\Theta(n)$. Then for each choise A from 2 to S do a binary search on the interval of integers $[2,N]$ to find a B such that $A * B = N$. There are $\Theta(\sqrt{N})$ choices of A, binary search for B also takes $O(\lg \sqrt{N}) = O(\lg N)$, and the multiplication of A and B takes $O(\lg N * \lg N)$. Total is $O(\lg^3 N * \sqrt{N})$. A naive method that works ok. ■

4.8 Graph representation and their complexity

Exercise 190.

You are given a directed graph $G = (V, E)$ represented through its adjacency matrix. Thus you only have n and the adjacency matrix A which is $n \times n$. You are given m. Provide running time (tight asymptotic bounds) for each one of the following problems/questions.

For a vertex u, $\mathrm{outd(u)}$ or $\mathrm{Outd(u)}$ denotes the out-degree of vertex u. The $\mathrm{outd(u)}$ is the number of edges incident FROM vertex u. For a vertex u, $\mathrm{ind(u)}$ or $\mathrm{Ind(u)}$ denotes the in-degree of vertex u. The $\mathrm{ind(u)}$ is the number of edges incident TO vertex u. You might also see $i(u)$ and $o(u)$. The $\deg(u)$ or $\mathrm{Deg}(u)$ or $\mathrm{d}(u)$ or $\mathrm{D}(u)$ is the degree of u, the number of edges incident from u or to u and is thus the sum of the out-degree and in-degree of u.

For an undirected graph, $\deg(u)$ or $\mathrm{Deg}(u)$ or $\mathrm{d}(u)$ or $\mathrm{D}(u)$ is the degree of u, the number of edges incident ON vertex u.

1. Does edge (u, v) exist?
2. Outd(u)?
3. Ind(u)?
4. Deg(u)?
5. Find number of edges of G. (Saying it is m is not enough. You have got to compute m itself!)

Proof. 1. $\Theta(1)$. 2. $\Theta(n)$, 3. $\Theta(n)$. 4. $\Theta(n)$ 5. $\Theta(n^2)$.

■

Exercise 191.

You are given a directed graph $G = (V, E)$ represented through its adjacency list. Thus you only have n and the adjacency array A which is an array/vector of length n. In $A[i]$ we can retrieve the heads of linked list $A(i)$ that points to the adjacency list of vertex i. Provide running time (tight asymptotic bounds) for each one of the following problems/questions.

1. Does edge (u, v) exist?
2. Outd(u)?
3. Find number of edges of G. (Saying it is m is not enough. You have got to compute m.)

Proof. 1. $O(outd(u))$. $outd(u)$ is the length of linked list $A(u)$.
2. $\Theta(outd(u))$. Cute isn't it? To avoid this some optimized implementation store in A[i] the length of the linked list.
3. $\Theta(n+m)$. Cuter isn't it? The time to find m is proportional to m and n. Traverse all linked lists $A(i)$ for all i. Do not forget $\sum_{u \in V} outd(u) = \Theta(m)$.

■

Exercise 192.

You are given an undirected graph G represented through its adjacency list. Based on Subject 3, How long does it take to answer the following questions? How?

1. Find the number c of connected components of G.
2. Is G connected? (i.e. is $c = 1$?
3. For input u, v in addition to G does there exist a path from u to v?
4. Modify existing code to collect this info. (Eg 3.40 of Subject 3.)

Proof. 1. Union-Find $O(m + n \lg n)$.
2. If from 1, $c == 1$ done in $\Theta(1)$ additional time.
3. If 1 is done already, $\Theta(1)$ additional time.
4. Practice makes perfect!

■

Exercise 193.
Give the running time of Union-Find that involves n elements, $n-1$ Union operations, n MakeSet operations and n^2 Find operations.

Union-Find refers to the linked list implementation.

Proof. $T(n) = \Theta(n^2)$. ■

Exercise 194.

You are given a set A of n integers and an integer c. We would like to establish whether there exist integers x and y in A such that $x+y=c$.

(i) Give an efficient algorithm for this problem and explain why it works so.

(ii) If in addition we require that the proposed algorithm should use $O(1)$ (constant) space besides the space used for A, would your solution for (i) change? Explain.

Proof.

We need to determine x,y in A such that $x+y=c$. Consider an element X of A; if $Y=c-X$ is also an element of A, then we have found such a pair, $x=X$ and $y=Y$ such that $x+y=X+Y=X+(c-X)=c$. For every element X of A we subtract X from c, and then check whether the result $c-X$ appears in A. The latter search procedure is facilitated by sorting A in advance (line 1); then the search of $c-X$ into A can be done by binary search (line 4) and takes $O(\lg n)$ per element $c-X$ searched. Since there are n elements X in A, and we go through them through the loop of lines 2-7, the total search time is n times $O(\lg n)$, i.e. $O(n\lg n)$ (loop of lines 2-7). In addition, sorting the n elements of A also requires $O(n\lg n)$ if we use mergesort (line 1). Therefore total time is $O(n\lg n)$. For constant space, use HeapSort below.

```
  Searchfor(c,A[1..n]) {
1.  A=MergeSort(A); //After this call A is sorted
2.  for(i=1;i<=n;i++) {
3.     X=A[i];
4.     index= BinarySearch(c-X,A[1..n],n) ;
5.     if (index != -1) && (index !=i) //-1 is not found,
6       return(YES);
7.  }
8.  return(NO);
```

■

4.9 Challenging problems

Exercise 195.

Average case of InsertionSort.

Proof.
Consider the j-th key of $A[1..n]$ for $j = 2$ to n. The j-th key can be inserted into any of the x positions (left of the key at offset 1, ..., left of the key at offset $j-1$ or retain its position. The probability that the j-th key falls into any of the j positions is equal to $1/j$ since there are j x possible insertion positions. For the rightmost x one comparison is needed, etc and for the x between offset 1 and offset 2, $j-1$ comparisons are needed. So also we need $j-1$ comparisons for the leftmost x.

Thus let X_j be the average number of comparisons performed for the key at offset j. We have

$$X_j = \frac{1}{j}\left(\sum_{k=1}^{j-1} k + (j-1)\right)$$

The sum is the $j-1$ rightmost x's, and the $+(j-1)$ accounts for the leftmost x. Then, we have the following.

$$X_j = \frac{1}{j}\left(\sum_{k=1}^{j-1} k + (j-1)\right) = (j-1)/2 + (j-1)/j = (j+1)/2 - 1/j.$$

Therefore

$$\begin{aligned} A(n) &= \sum_{j=2}^{n} X_j \\ &= \sum_{j=2}^{n} (j+1)/2 - \sum_{j=2}^{n} 1/j \\ &= 1/2(n(n+1)/2 - 1) + 1/2(n-1) - H_n + 1 \\ &= \frac{n^2+3n}{4} - H_n \end{aligned}$$

For $n = 3$ we have $A(n) = (3^2+9)/4 - (1+1/2+1/3) = 4.5 - 1.5 - 1/3 = 2.66$. For the decision tree of InsertionSort the average number of comparisons is $(2+2+3+3+3+3)/6 = 2.66$! ■

Exercise 196.

A sequence is up-down if the keys are

$$\leq \geq \leq \geq \ldots \leq \geq .$$

For a sequence $A[0..2n]$ of $2n+1$ keys that are comparable (i.e. a total order is defined on them) convert A into an up-down sequence in $o(n^2)$ time.

Proof.
Can be done faster than this algorithm.

```
  UpDown  (A,n)
  // n is the problem size but the input consists  of
  // 2n+1 keys i.e. input size is 2n+1
  // One could use HeapSort (space efficient)
1.  MergeSort(A,2n+1);  // Line  1   : O(nlgn) time
2.  for(i=1;i<=n;i+=2)  // Lines 2-3 : O(n) time
3.    swap(i,2n-i+1);   // swap A[i] and A[2n-i+1]
```

After you sort the first $n+1$ keys, the n gaps formed by consecutive such keys will be filled by the remaining n keys. ■

Exercise 197.

For an array $A[0..n-1]$ of n numbers find in $\Theta(n)$ time, two keys a,b of A such that $a-b < (M-m)/(n-1)$.

Keys m,M is the minimum and maximum respectively of A.

Proof.

Find m and M in $2n-3$ comparisons. Asymptotics only matter. This step is $\Theta(n)$. Then split interval $[m,M]$ into $n-1$ sub-intervals of width $(M-m)/(n-1)$ as needed. There are n keys and $n-1$ subintervals. By the pigeon-hole principle, there would be at least two keys in a subinterval. Let them be a,b. It is clear that $a-b$ or $b-a$ is at most $(M-m)/(n-1)$. Create $n-1$ intervals $I[0..n-2]$ and initialize to zero. Determining the interval of $A[i]$ is constant time; increment $I[interval(A[i])]++$. Total time is $\Theta(n)$. In a second scan find an interval whose count is 2 or more. Go through A to find the keys that map to that interval. (One could build a linked list per intervala ala BucketSort.)

■

Exercise 198.

An array $A[0..N-1]$ contains all the integers from 0 to N except one. Let $N = 2^n - 1$ for simplicity. Find the missing integer in $\Theta(n)$ time. You cannot use functions that operate on all bit of an element (e.g. add, subtract, OR, AND) but only one bit at a time (see two functions below). Thus adding all the elements and subtracting from $n(n+1)/2$ is not a solution. Neither taking the XOR of all the integers of A. We utilize function rightmost bit of m i.e. Rmost(m) which is equivalent to $m\&1$. Also Rest(m) which is equivalent to $m >> 1$. Moreover Append(X,q) does the obvious, as needed.

Proof.
Since N is odd the integers $0 \ldots n$ have a total that is an even number. Same number of integers ending in 0 and integers ending in 1. One integer is missing, means that if num(1) ¡ num(0) implies an odd number is missing, otherwise an even number is missing. (num(1) is the number of integers ending in one.) Eliminate the non-missing group and recur on the remaining bit. You have also established that the rightmost bit is the lower set cardinality's parity. The recurrence for the running time is $T(n) = T(n/2) + \Theta(n)$ leading to $T(n) = \Theta(n)$.

```
    Find(A,N,n) {  // N= 2**n -1 ; A[0..N-1]
0   X0 = o ; X1 = o;          // o empty set
1.  Mint[n] = o;              // Mint array of n bit
1.  if ( n == 1)
2.     if A[0] == 1 return(0);
3.     else         return(1);
4.  zeroes = 0 ; ones = 0;
5.  for( bit = 0 ; bit < n ; bit++) {
5.     for( i=0 ; i < n ; i++) {
6.       if (Rmost(A[i]) == 0) {         // Rmost(x)=x&1
8.         X0[zeroes++] = Rest(A[i]);    // Rest(x) =x>>1
7.       }
8.       else {
9          X1[ones++ ]  = Rest(A[i]);
10.      }
11.      if (zeroes <  ones) {
12.         A= X0;
13.         Mint[bit]=0;
13       }
14.      else {
15.         A= X1;
13.         Mint[bit]=1;
14.      zeroes=ones=0;
15     }         // Little endian output
15  }            // rightmost leftmost
17. return(Mint);// Mint[0]...Mint[n-1] is int   Mint[n-1]...Mint[0]
```

■

Exercise 199.

You are given a sorted array $A[1..n]$ of n **distinct** integers (negative or positive). Give an efficient algorithm to find an index i, $1 \leq i \leq n$ such that $A[i] = i$ provided such an index exists. If there are many such indices, your algorithm can return any one of them (and exit). For example if the input is $A = \langle -10, \quad 2, \quad 20, \quad 40 \rangle$ your algorithm should return index 2 since $A[2] = 2$. We ask for an efficient algorithm that takes $o(n)$ time in the worst-case because the simple linear algorithm that scans the array and checks whether $A[i]$ is equal to i in turn, is one solution to the problem; however it takes too much time.

Proof.
Use binary search. Idea is as follows. If $A[j] > j$ and $A[i] = i$ then $i < j$. If $A[j] < j$ and $A[i] = i$ then $i > j$.

```
   FindMissing(A,l,r)
1. if (l >  r) return(-1) ; //for not found
2. m = l+(r-l)/2;
3. res = compare(A[m],m);
3. if ( A[m] == m ) return(m);                  // res == 0
4. if ( A[m] >  m ) return(FindMissing(A,l,m-1) // res  > 0
5.  esle            return(FindMissing(A,m+1,r) // res  < 0
```

■

Exercise 200.

An array $A[0..n-2]$ is given that contains all the integers from 1 throught n except one integer that might be missing. Give a time efficient algorithm for finding the missing integer. Linear running time.

Proof.

```
Missing(A,n-1)
count=0;
for(i=0;i<n-1;i++)
  count += A[i];
return( n*(n+1)/2 - count);
```

■

Exercise 201.

Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A.
(a) List the inversions of array $\langle 5,7,2,1,4\rangle$.
(b) What array with elements from the set $\{1,2,\ldots,n\}$ has the most inversions? How many does it have? Express the answer as a function of n.
(c) What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.
(d) Find the number of inversions of $A[0..n-1]$ in $O(n \lg n)$. (Rather than in $O(n^2)$ or rather of $A[1..n]$.)

Proof.
(a) We have seven inversions $(1,3)$, $(1,4)$, $(1,5)$, $(2,3)$, $(2,4)$, $(2,5)$, $(3,4)$.
Do (b) and (c) yourselves. (b). $\langle n, \quad n-1, \quad \ldots \quad ,1\rangle$, a reverse-sorted sequence has the most inversions. These are as follows.

```
(1, 2 )    (1,  3 ) (1  ,  4)  ...  (1,n-1)  (1,  n)    ==  n-1 inversions
           (2,  3 ) (2  ,  4)  ...  (2,n-1)  (2,  n)    ==  n-2 inversions
                    (3  ,  4)  ...  (3,n-1)  (3,  n)    ==  n-3 inversions
                    ...        ...                               ...
                                  (n-2,n-1)  (n-2,n)    ==   2  inversions
                                             (n-1,n)    ==   1  inversion
```

Therefore the total number of inversions is $1+2+\ldots+(n-1) = n(n-1)/2$.
(c) The inversions correspond to the swap-operations (or half-swaps traditionally performed in InsertionSort. Each inversion corresponds to a misplaced key that needs to be fixed (inserted). That is, the number of inversions corresponds to the running time of InsertionSort for the given input.
(d) Use MergeSort. Find the inversions on the left side of a divide-and-conquer MergeSort, those of the right side, recursively and for the Merging step count the number of inversions that involve both sides during the merging.

```
    CountI(A, n)  // A[0..n-1]
1.    CountInv(A,0,n-1);

    CountInv(A, l, r) {
1.  if (l<r) {
2.    m = l + (r-l)/2;
3.    cnt = cnt + CountInv(A,l,m);
4.    cnt = cnt + CountInv(A, m+1, r);
5.    cnt = cnt + MergeInv(A,l,m,r);
6.    return(cnt);
}

    MergeInv(A,l,m,r)
0.  nL= (m-l+1);  nR=(r-m);
1.  L[0:nL-1] = A[l:m]; R[0:nR-1] = A[m+1:r];
2.  cnt = 0;
3.  i=0; j=0; k=l;
4.  while ( i < nL  && j < nR) {
5.     if ( L[i] <= R[j] ) {
6.        A[k++] = L[i]; i++;
7.     }
8.     else {
9.        A[k++] = R[j]; j++;
10.       cnt += (nL - i);
11.    }
12. A[k: ] = L[i:]; k += (nL-i);
13. A[k: ] = R[j:];
14. return(cnt);
```

■

Exercise 202.

The insertion sort procedure usually outlined uses a linear search to scan (backward/from the right) through the sorted subarray $A[1 \dots j-1]$.
a) Let us assume that $C(n)$ gives the number of **comparisons** required to sort n keys. Can we use a binary search procedure instead of a linear search to improve the overall worst-case performance of insertion sort so that $C(n) = O(n \lg n)$? Justify your answer.
b) Let us call the algorithm of part (a) that uses binary search BIN_INSERTION_SORT. Let the running time $T(n)$ of BIN_INSERTION_SORT includes both comparison and assignment operations. Let the cost of a comparison operation be A and the cost of an assignment operation be B. Express the worst-case time performance of insertion sort in terms of n, A and B. Use asymptotic notation to express your answer. (*Note that A and B may not be constant*).

Proof. Binary search on an array of size n takes $O(\lg n)$ steps i.e. time i.e. comparisons. Therefore, binary search on an array of size j take $O(\lg j)$ steps.
a) We can use binary search to insert key into the array $A[1 \dots j-1]$. This would require $O(\lg j)$ comparisons. Therefore the total number of comparisons $C(n)$ required in the worst case for all keys is

$$C(n) = \sum_{j=1}^{n-1} O(\lg j) = O(\sum_{j=1}^{n-1} \lg j) = O(\lg 1 + \lg 2 + \ldots + \lg n) = O(\lg (1 \cdot 2 \cdot \ldots \cdot n)) = O(\lg n!) = O(n \lg n),$$

where in the last equality we used Stirling's approximation formula for the factorial.
b) Even if the number of comparisons is decreased from $\Theta(n^2)$ to $O(n \lg n)$, the number of assignments remains the same as by say inserting key x into position l all keys occupying positions $A[l \dots j-1]$ must still be shifted one position to the right. Therefore the worst-case time will contain a $\Theta(n^2)$ term to include the number of assignment operations performed. By multiplying each comparison by A and each assignment by B, the worst-case running time is given by $\Theta(An \lg n + Bn^2)$. (In practical cases the cost of a comparison A can be quite high if it involves a function call compared to the cost of an assignment B which can be considered very small (a constant) if it involves memory transfers only (no function calls).

■

Exercise 203.

(a) What does the following recursive algorithm do? Assume integer numbers as input.

```
    FindMeWrapper(A[0..n-1],n);
      FindMe(A[0..n-1],0,n-1);

    FindMe(A,l,r)
1.  if (l==r) return A[l];
2.  a = FindMe(A,l,           l+ (r-l)/2 ); //
3.  b = FindMe(A,l+(r-l)/2+1, r);           //
4.  if (a >= b) return(a);
5.  else        return(b);
```

(b) How many times does FindMe (all calls) inspect element $A[i]$? Explain and give a precise bound, not an upper bound.

Proof.
(a) It finds the Max of A recursively.
(b) Observe the following. The left and right side extend

$$\text{Left side from} \quad l \text{to} \quad l+\lfloor\frac{r-l}{2}\rfloor, \quad \text{Right side from} \quad l+\lfloor\frac{r-l}{2}\rfloor+1 \text{to} \quad r$$

Furthermore, the number of elements on the left side is

$$\text{Left side size} \quad \lfloor\frac{r-l}{2}\rfloor+1, \quad \text{Right side size} \quad r-l-\lfloor\frac{r-l}{2}\rfloor.$$

Consider the number of elements from l through r. It is $N=r-l+1$. Consider two cases (a) $N=2k=r-l+1$ and thus $r-l=2k-1$, and (b) $N=2k+1=r-l+1$ and thus $r-l=2k$.
For case (a) we have the following.

$$\text{(a) Left side size} \quad \lfloor\frac{r-l}{2}\rfloor+1=k-1+1=k, \quad \text{Right side size} \quad r-l-\lfloor\frac{r-l}{2}\rfloor=2k-1-(k-1)=k,$$

that is each side gets $N/2$ elements.
For case (b) we have the following.

$$\text{(b) Left side size} \quad \lfloor\frac{r-l}{2}\rfloor+1=k+1, \quad \text{Right side size} \quad r-l-\lfloor\frac{r-l}{2}\rfloor=k,$$

that is the left side gets $\lceil(N/2)\rceil$, and the right side $\lfloor(N/2)\rfloor$ elements. Combining the two, the left side size is $\lceil(N/2)\rceil$, and the right side size is $\lfloor(N/2)\rfloor$.
The depth of the recursion to enable line 1 is $\lceil\lg n\rceil$. This follows from

$$\lceil\frac{\lceil\frac{n}{2}\rceil}{2}\rceil=\lceil\frac{n}{2^2}\rceil$$

and generalizing it by induction into.

$$\lceil\frac{\lceil\frac{n}{2^t}\rceil}{2}\rceil=\lceil\frac{n}{2^{t+1}}\rceil$$

Every time recursion returns one comparison takes place in line 4. Thus the total per key is at most $\lceil\lg n\rceil$. Sometimes we call this the tournament method to find the max.

■

Exercise 204.

Find the largest and second largest of n keys in at most $n + \lceil \lg n \rceil - 2$ comparisons.

Proof.
An elegant solution is as follows utilizing binomial trees. Details are omitted. Example is provided. The MAX is found in $n-1$ comparisons. The $\lg n$ children of the root in the binomial tree contain the second largest key. In the array representation the second largest is at index positions $k-2^0, k-2^1, k-2^2, \ldots$

```
70 20 40 80 60 10 50 30        A: 70 20 40 80 60 10 50 30

70    80    60    50           A: 20 70 40 80 10 60 30 50
|      |     |     |                  *     *     *     *
20    40    10    30

70    80    60    50           A: 20 70 40 80 30 50 10 60
                                           *           *
---80       ---60
|   |       |   |
70 40       50 10
|           |
20          30

     ------80                  A: 30 50 10 60 20 70 40 80
     |  |   |                              ^     ^  ^  *
  ---60 70 40                              |     |  |  |
  |  |  |                                  |     |  |  k (=7)
  50 10 20                                 |     |  k-2**0
  |                                        |     k-2**1
  30                                       k-2**2
```

■

Chapter 5

More sorting, heaps, and applications

5.1 Heap math

Exercise 205.
(a) How much is the sum

$$\sum_{i=0}^{\infty} i \cdot x^i \quad = \quad 1 \cdot x^1 + 2 \cdot x^2 + \ldots + i \cdot x^i + \ldots = ? \qquad 0 < x < 1$$

(b) How much is the sum $\sum_{i=0}^{\infty} i(i-1)x^i$, $0 < x < 1$? Explain.

Proof.
(a) The derivation of interest is the

$$\sum_{i=0}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2},$$

that will be established in part (b) below. (Note it has also been derived earlier.)

(b) Start with the infinite geometric sequence.

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}.$$

Take the first derivative of both sides. We get

$$\sum_{i=0}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2}.$$

If at this point you multiply both sides with x part (a) is resolved. We continue with part (b). Take the first derivative of both resulting sides. We get

$$\sum_{i=0}^{\infty} i(i-1) \cdot x^{i-2} = \frac{2}{(1-x)^3}.$$

We then multiply both sides with x^2. We get

$$\sum_{i=0}^{\infty} i(i-1) \cdot x^i = \frac{2x^2}{(1-x)^3}.$$

All of the steps are ok since $0 < x < 1$. Result is shown above.

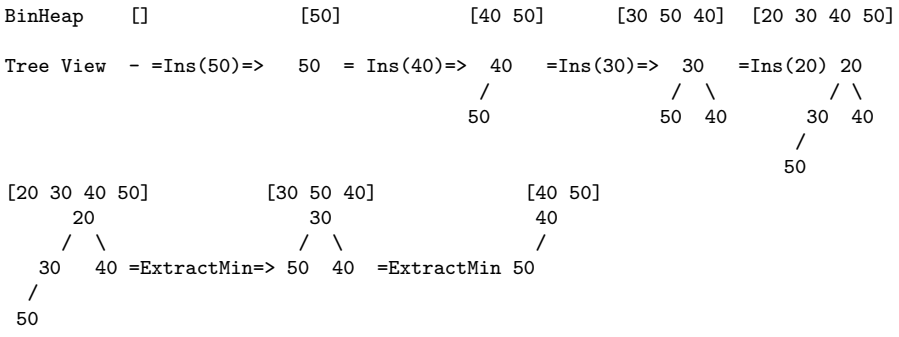
■

5.2 Heap properties

Exercise 206.

In an initially empty MinHeap insert 50,40,30,20 and then ExtractMin and ExtractMin. Show the state of the binary Heap after each one of the 6 operations above is completed.

Proof.



■

Exercise 207.
(a) Is the following sequence a MaxHeap? Explain. $\langle 10,\ 9,\ 10,\ 3,\ 5,\ 7\ \rangle$
(b) What happens if I perform an ExtractMax (RemoveMax, DeleteMax) operation on the heap of question (a)? Show the heap adjustment steps.
(c) In the resulting heap of part (b) I perform an Insertion of key 10. Show the steps.
(d) Where in a MinHeap might the largest element reside, assuming that all elements are distinct?

Proof.
(a) Yes it is. Proof by inspecting the MAX-HEAP property at every node.
(b) After ExtractMax is realized the structure becomes $\langle 7,9,10,3,5\rangle$ which is not a heap. Apply DownHeap/Heapify to the root node we get $\langle 10,9,7,3,5\rangle$ and this is a heap.
(c) Inserting 10 in the resulting heap we immediately get, $\langle 10,9,7,3,5,10\rangle$ and continuing the insertion operation we obtain $\langle 10,9,10,3,5,7\rangle$.
Let the node indexed u store the largest element of the MinHeap. The children $2u+1$ and $2u+2$ (for Pascal indexing $2u$ and $2u+1$) of u must be empty (i.e. no heap elements are stored there) otherwise the heap property would be violated (since then, the value stored at u, which is the largest key of the whole heap, would be larger NOT SMALLER than the values of its children). Therefore u must be a leaf, i.e. the largest element of a heap is stored in a leaf. ■

Exercise 208.
(a) What are the minimum and maximum numbers of elements in a heap (max- or min- heap) of height h?
(b) What is the height of a heap with n elements?
(c) Show that in any subtree of a MAX-HEAP, the root of the subtree contains the largest value occurring anywhere in that subtree.

Proof. (a) Since the heap has size h, all levels but the last form a complete binary tree of height $h-1$. Such a tree has $1+2+\ldots+2^{h-1}=2^h-1$ nodes. A heap of height h must have at least one node (leftmost) at the next level h (that may be incomplete) because otherwise it won't be of height h but $h-1$. Moreover the maximum number of nodes (leaves) at level h is 2^h Therefore minimum number of nodes in a heap of height h is $2^h-1+1=2^h$; maximum is $2^h-1+2^h=2^{h+1}-1$.

(b) An n element heap has height h such that, by way of part (a), $2^h \leq n \leq 2^{h+1}-1 < 2^{h+1}$. Therefore by taking logarithms base 2 of both sides $h \leq \lg n < h+1$. Height h is always an integer. By the definition of the floor function we derive that $h=\lfloor \lg n \rfloor$.

(c)Let $S(u)$ be the subtree of a heap rooted at some node u. Suppose by way of contradiction that the maximum value M stored in $S(u)$ is not at u. Let the left and right subtrees of u be $LS(u)$ and $RS(u)$. Then the maximum of $S(u)$ must be in either $LS(u)$ or $RS(u)$. Let the index of the heap node of $S(u)$ with the maximum M be m (note that if there are more than one nodes storing M we take the node with the minimum index value m). As the node indexed m cannot be node u it must be an internal node of $S(u)$ and therefore it must have a parent p. The index of the parent will be floor$(m/2)$ i.e. smaller than m. The value stored in p must also be less than M as m is the lowest indexed node storing the maximum value M and therefore the value of the parent whose index is less than m must be smaller. Therefore the MAX heap-property fails for p and m, a contradiction to the assumption that $S(u)$ is a subtree of a heap.

■

Exercise 209.

Provide short answers to the following questions.
(a) Give the running time of BuildMaxHeap of n^2 keys.

(b) Give the running time of HeapSort of $m \lg m$ keys.

(c) Give the running time of InsertMaxHeap(key) of `key` into a MaxHeap containing 2^n keys.

(d) For n keys that are comparable, is it possible to have a heap operation ExtractMax and BuildMaxHeap that operate in time $\Theta(1)$ and $\Theta(n)$ respectively?

(e) What is the minimum and maximum number of elements in a heap of height h?
(f) Show that an n element heap has height $\lfloor \lg n \rfloor$.
(g) Show that in any subtree of a MinHeap, the root of the subtree contains the smallest value occurring anywhere in that subtree.
(h) Is a sorted (non-decreasing order) array a MinHeap?

Proof.
(a) The running time is $T(n) = \Theta(n^2)$. Note that in this case n is problem size but input size is n^2.
(b) The running time of HeapSort is $T(m) = O(m \lg m \cdot \lg(m \lg m))$. We refine it as follows.

$$\begin{aligned} T(m) &= O(m \lg m \cdot \lg(m \lg m)) \\ &= O(m \lg m \cdot (\lg m + \lg \lg m)) \\ &= O(m \lg m \cdot \lg m) \\ &= O\left(m \lg^2 m\right). \end{aligned}$$

(c) $T(n) = O(\lg(2^n)) = O(n)$.
(d) NO. Then a version of HeapSort would take $T(n) = \Theta(n)$ time, which violates a comparison-based lower bound on sorting.
(e) A heap of height h has its first $h-1$ levels completely filled and in the last level there exists at least one element and at most 2^h. Therefore the number of elements is $1+2+\ldots+2^{h-1}+m = 2^h - 1 + m$, where $1 \leq m \leq 2^h$. Therefore the minimum number of elements is 2^h and the maximum number of elements is $2^{h+1}-1$.
(f) An n element heap has height h such that, by way of part (a), $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$. Therefore by taking logarithms base 2 of both sides $h \leq \lg n < h+1$. Height h is always an integer. By the definition of the floor function we derive that $h = \lfloor \lg n \rfloor$.
(g) Let $S(u)$ be the subtree of a heap rooted at some node u. Suppose by way of contradiction that the minimum value M stored in $S(u)$ is not at u. Let the left and right subtrees of u be $LS(u)$ and $RS(u)$. Then the minimum of $S(u)$ must be in either $LS(u)$ or $RS(u)$. Let the index of the heap node of $S(u)$ with the minimum M be m (note that if there are more than one nodes storing M we take the node with the minimum index value m). As the node indexed m cannot be node u it must be an internal node of $S(u)$ and therefore it must have a parent p. The index of the parent will be floor$(m/2)$ i.e. smaller than m. The value stored in p must also be greater than M as m is the lowest indexed node storing the minimum value M and therefore the value of the parent whose index is less than m must be greater. Therefore the MIN heap-property fails for p and m, a contradiction to the assumption that $S(u)$ is a subtree of a heap.
(h) Yes it is trivially as the min-heap property applies trivially to every node. ■

Exercise 210.
TRUE or FALSE?

1. Heap-sort is stable.
2. For keys that are comparable, it is possible to have heap operations ExtractMin and BuildMinHeap that operate in time $\Theta(1)$ and $\Theta(n)$ respectively.
3. Heap-sort sorts in-place.
4. The running time of BuildMinHeap is $\Theta(n)$.
5. HeapSort is in fact SelectionSort; we first build a Heap of n keys in linear time, and then select the largest, second largest, etc in additional $O(n \lg n)$ time.

Proof. 1. FALSE, 2. FALSE, 3. TRUE, 4. TRUE, 5. TRUE. ■

5.3 Heap operations

Exercise 211.

Give pseudocode of MaxHeapify(A,H,i) (or MinHeapify(A,H,i)) that is non recursive.

Proof.

```
  Max-Heapify(A,i) // Pascal indexing implied
1.  down =i; // store i in down and work with down
    // if down > heapsize(A)/2 STOP as i has no children; otherwise,
2.  while (down <=  (heapsize(A) /2 )  {
3.   if (2*down == heapsize(A) ) { //down has only one child
4.     if (A[down] < A[2*down]) {
5.      exchange(A[down],A[2*down]);
6.      down=heapsize(A); //force an exit from the while loop
7.     }
8.   }
9.   else { // down has two children
10.   largest = index of maximum among A[down],A[left(down)],A[right(down)]
11.   if largest != down  {
12.    exchange(A[down],A[largest]);
13.    down =largest; //down moves down; ready for new iteration
14.   }
15.  } //end of else
16  } //end of while
```

■

Exercise 212.
Write code for MeanHeapify.

Proof.

```
 MinHeapify(A,i)
1. l = left(i);
2. r = right(i);
3. if l <= heap-size(A) and A[l] < A[i]
4.  then smallest =l;
5.  else smallest =i;
6. if r <= heap-size(A) and A[r] < A[i]
7.  then smallest =r;
8. if (smallest!= i)
9.  then swap     A[i] <-> A[smallest]
10.      MinHeapify(A,smallest);
```

■

Exercise 213.

In a BinaryHeap A that is a MaxHeap we would like to implement operation UpdatePriority(A; i; v) that updates the priority of A[i] into the new value v i.e. the end result should be that A[i] is set to v and still maintains A as a MaxHeap adjusting A as needed afterwards. Show how UpdatePriority can be implemented in time that is O(lg n), where n the number of heap elements in A. Justify your answer and analyze its worst-case running time performance.

Proof.

Case 1: A[i] ¡ v. If the updated value v is larger than A[i] (prior to its update), the MaxHeap property might get violated somewhere along the path from index i to the root of the heap. An operation similar to MaxHeapInsert needs to be performed.

Case 2: A[i] ¿ v. If the updated value v is smaller than A[i], the MaxHeap property might get violated at position i. DownHeap at i needs to be initiated. Case 3: A[i] == v. Do nothing but return. Whether Case 1 or Case 2 is pursued, and the update needs to be propagated towards the root (i.e. key at index 0) or towards the leaves (i.e. keys at indexes heapsize(A)=2) the running time would be O(lg n) absorbing also in it the running time of Case 3.

```
     UpdatePri(A,i,v) // A[1..heapsize(A)]
1. if A[i] > v {
2.    A[i] = v;
3.    MaxHeapify(A,i);
4. }
5. elseif A[i] < v {
6.    A[i] = v;
7.    while (i>1 ) && A[parent(i)] < A[i] {
8.       swap A[i] and A[parent(i)];
9.       i=parent(i);
10.   }
11. }//if A[i]== v nothing needs to be done; it retains its value
```

■

Exercise 214.

(Pascal indexing.)
A d-ary heap is to be represented by an one dimensional array A. $A[1]$ stores the root and $A[2],\ldots A[d+1]$ store the d children of the root. The j-th child of node indexed i is $d(i-1)+j+1$ where $1 \leq j \leq d$. The parent of node i is floor$((i-2)/d+1)$. A binary heap is a special case for $d=2$.
(a) What is the (asymptotic) height of a d-ary heap with n elements? Express it in terms of n and d.
(b) How does ExtractMax work in Max-d-Heap? What is its asymptotic running time ?
(c) How does InsertMax work in Max-d-Heap? What is its asymptotic running time ?
(d) How does HeapMaxIncreaseKey work in Max-d-Heap? What is its asymptotic running time ?

Proof.
(a) A heap of height h has $n = 1+d+d^2+\ldots+d^{h-1}+m = (d^h-1)/(d-1)+m$ elements, where $1 \leq m \leq d^h$. Therefore its height is $O(\lg n/\lg d)$.
(b) ExtractMax works the same way as HeapExtractMax for binary heaps except that DownHeap/Heapify(A,i) on d-ary heaps works a little bit differently. It assumes that the d subtrees of node indexed i have the heap property and the maximum is taken over the value stored at i and the values stored in all d children of i. The maximum operation takes time $O(d)$ now per node, and ExtractMax may call Heapify $O(h)$ times. By way of part (a) $O(h) = O(\lg n/\lg d)$ and therefore the time for Heapify will be $O(d \lg n/\lg d)$.
(c) Insert for binary heaps works fine for d-ary heaps as well (remember to adjust the new definition of a parent though). An upward movement is independent of the number of children of a parent. Therefore its running time is $O(h) = O(\lg n/\lg d)$.
(d) This is similar to insertion. If the value k is at most the value of $A[i]$ nothing changes, otherwise the value of $A[i]$ is changed and propagated upwards just as we did in Heap-Insert. The observation is that since the value k is propagated upwards, we only need to change the value of the final destination only, just as we did in Heap-Insert. The following program demonstrates this. Note that parent(i) follows the definition of part (b.a) not that of a binary heap.

```
 HeapMaxIncreaseKey (A,i,k)
1. If A[i] >= k the return ;
2. while i>1 and A[parent(i)] < k
3.     do A[i] = A[parent(i)];
4.        i=parent(i);
5. A[i]=k;
```

■

Exercise 215.

Implement a HeapIncrease(A,i,k) for a MaxHeap that increases the priority of $old = A[i]$ from the current $A[i]$ to a new $A[i] = k$ where k is larger than the previous $A[i]$ value (i.e. the old value).

Proof. Idea: Change the value at node i as appropriately in step 2. If the value is indeed changed, propagate this value to the root of the tree as in Heap-Insert to maintain MAX-heap property for nodes located higher in the tree (as new value can only be larger than old one, heap-property is maintained for subtree-view rooted at i).

```
 Heap-Increase-Key(A,i,k)
1. old=A[i]; j=i;
2. A[i]= max(A[i],k);
3. if old not equal A[i]  //i.e. A[i] has been changed
4. do key=A[i];
5.    while j>1 and A[parent(j)] < key
6.     do A[j] = A[parent(j)];
7.        j = parent(j);
8.    A[j]=key;
```

■

5.4 HeapSort vs Heap operations

Exercise 216.

The following two methods are proposed to find M_k, the k-th smallest of n keys stored in array $A[j..n]$.
a) Sort A and then output the k-th element of the sorted A. This is M_k.
b) Build a MIN-heap out of the n elements of A, and then perform k EXTRACT-MIN operations. The last (i.e. k-th) EXTRACT-MIN operation returns M_k.
Which of the two algorithms is asymptotically faster? Explain, i.e. provide details of efficient implementation of the various steps, and analyze their worst-case running time using asymptotic notation. Use case analysis, if necessary.

Proof.
The two algorithms are

```
  // Pascal indexing
a. FindMv1 (A[1..n],n)
1. HeapSort(A);
2. return(A[k]);
```

```
b. FindMv2 (A[1..n],n)
1. BuildMINheap(A,n);
2. for(i=1;i<k;i++){
3.  z=EXTRACTMIN(A);
4. }
5. return(z);
```

The sorting-based one has obviously running time $T(n) = O(n \lg n)$ (Θ can also be used). The other algorithm requires $T(n) = \Theta(n)$ time for building a MinHeap, and then $k \cdot O(\lg n)$, for k extraction operations for a total of $O(n + k \cdot \lg n)$. We now compare the asymptotic performance of both algorithms.
(i) If $k = o(n)$ then $O(n + k \cdot \lg n) = o(n \lg n)$ thus method (b) is asymptotically better than method (a).
(ii) if $k = \Omega(n)$ and of course $k \leq n$, then both methods are $O(n \lg n)$ and have the same asymptotic performance. (O can be Θ .) ■

5.5 Extend a heap data structure

Exercise 217.

Design a data structure H using arrays (i.e. binary heaps) that implements efficiently the following three operations:
(i) `Insert(H,x)` for inserting key x in it,
(ii) `ExtractMin(H)` fro extracting the minimum and
(iii) `ExtractMax(H)` for extracting the maximum respectively.
Show the organization of the data in H and describe the three operations involved. Make sure that all operations are in the worst-case $o(n)$ (and this is a little-oh.) Explain.

Proof.
Maintain both a MinHeap and MaxHeap at the same time. An element is inserted into both and through pointers/indices the location to the other heap is known. For ExtractMax the Max is extracted from the MaxHeap first; then through the maintained pointer/index it is also Deleted from the MinHeap. A symmetric operation resolves ExtractMin.

Let N[0..] be the array for the MinHeap, and X[0..] for the MaxHeap. N[i].key will hold the priority of the key and N[i].other will hold the index of N[i].key in the MaxHeap. Similarly X[i].other will hold the index of X[i] in the MinHeap.

Thus, for a MinHeap element stored in position i, we know where it is stored in the MaxHeap: it is in position N[i].other. The same is true for MaxHeap elements.

Our H is thus the pair of heaps `(N,X)` augmented with the information maintained in the `other` fields.

Insert(H,x) inserts key with priority x in both N, X. This takes $O(\lg n)$ time if n is the number of elements in N, X. During the insertion of x into N, it is possible that the $O(\lg n)$ keys in the path from a leaf to the Insert position of x get swapped with x. If this is the case the info in X needs to be updates. This is still $O(\lg n)$. When one inserts into N, one remembers to do likewise. This is still $O(\lg n)$. Likewise for an ExtractMax(H) or ExtractMin(H) operation. For an ExtractMax(H) operation, we perform ExtractMax(X) first. This delete $t = X[0]$. During the extraction, a DownHeap/Heapify operation might change the location of keys in X and thus the corresponding N .other entries needs to be updated. When ExtractMax(X) completes its execution we perform a Delete operation on $N[t.other]$. Then first $N[t.other].key = N[heapsize(N) - 1].key$ followed by a $heapsize - -$.

If the new value $N[heapsize(N) - 1].key < t = X[0]$, we work similar to Insertion upwards. Otherwise similar to DownHeap/Heapify downwards. All swapped nodes have their $X[].other$ updated accordingly. Like-wise for the other operation. All operations are $O(\lg n)$. ■

Exercise 218.
In a standard implementation of a heap, heap elements have arbitrary key values. Let us deal with a specific case where key values belong to a restricted range $0,\ldots,d-1$, where d is a constant, i.e. $d = O(1)$ (for example, think of d being 10 if it helps). We want to design a MinHeap whose Insert(x) and ExtractMin() operations take constant worst-case running time $O(1)$ on a heap that contains n keys. Design such a heap, and show how Insert and ExtractMin can be implemented to achieve the required worst-case time bound of $O(1)$.
Note: x is an element (object) that has attributes (data members) $x.info$ and $x.k$. The former contain information about x. The priority of the element is $x.k$. Thus it is possible for two $x \neq y$ that have $x.k$ equal to $y.k$.

Proof. The key is that d is constant. A naive approach is to have a heap of d priorities and in each one there is an attached linked list of elements with the same priority. If you try to insert a key you might have problems or make the code unreasonably complicated. The easier approach is to create a array A of linked lists. (Think of adjacency list.) In linked list $A[i]$ all elements with priority i are inserted (at the end of the list). Insert is quite easy. ExtractMin starts checking $A[0]\ldots A[d-1]$. If a list is not empty the first element is removed from the linked list and returned. Because the length of A is d and d is constant all ops are $\Theta(1)$ time! Fill in the details! ■

Exercise 219.

How can you realize a data structure that supports ExtractMax, ExtractMin and Insert all three of them in $O(\lg n)$ time using structures already defined and used in class? Use as little space as possible.

Proof. An outline by skipping details. Use a Binary MinHeap and MaxHeap with two auxiliary arrays ToMax and ToMin. For a MinHeap[i] element, ToMax[i] shows its position (index) into the MaxHeap. Likewise for MaxHeap[i] element.
When an element is inserted it is inserted into the MinHeap first; in DownHeap when two elements are swapped say MinHeap[i] and MinHeap[j] to maintain the heap property, then make sure that the corresponding ToMin entries get swapped i.e. ToMin[ToMax[i]] and ToMin[ToMax[j]]. Then it is inserted into the MaxHeap similarly.
When Extraction is to be performed, say ExtractMax, the key is HeapExtracted from the MaxHeap first. Any DownHeap operations should update ToMax as well. When the key is erased from the MaxHeap, its ToMin[0] entry of the MinHeap must be 'deleted' and 'not extracted'. Naive 'deletion' would create a gap: replace the gap with the last element as in an Extract operation. From that point on it works either as a DownHeap if the replaced value is smaller than the value replacing it, or upwards as an Insert operation otherwise. See example below and try to "delete" the 10 or the 25. The ExtractMin operation is analogous. ■

Exercise 220.

A priority queue is an abstract data type (ADT) consisting of a set of elements, each with a PRIORITY key value. INSERT (to add a new element), EXTRACTMAX (to delete an element of highest priority). In this problem, we consider a special case of a priority queue with **integer keys** in the range 1 to 10. For this special case, describe an appropriate data structure to use to implement the priority queue, describe the implementation of INSERT and DELETEMIN operations and their time complexity. Make sure that you design the data structure in such a way that both operations require time $o(\lg n)$, i.e. they are faster than traditional heaps.

Proof.

We use an array of 10 linked-lists (singly or doubly) to implement this data structure. We assume $A[1..10]$ is an array of 10 linked lists in the pseudocode below. To insert key k we just add it to the list $A[k]$. We can insert at the start or the end of the list. For a deletion (i.e. ExtractMax operation) we scan the lists backwards starting from $A[10]$ and going all the way down to $A[1]$. If $A[10]$ is non empty, then there exists at least one key with value 10 and one such key is extracted (deleted) through line 3 and returned by ExtractMaxHeap. If no such keys exists, we check $A[9]$, $A[8]$, until we find a non-empty list $A[i]$ and we extract and return a key from that list. If no $A[i]$ is nonempty this means that all are empty and an underflow condition is thrown.

```
 Insert(Q,k)
1. if firsttime {
   for(i=1;i<=10;i++ A[i]=NULL
2.  firsttime=0;
3.  InsertLinkL(A[k])

ExHigh(Q)
1. for(i=10;i>=1;i--)
2.  If A[i] !=NULL ret (DeletLinkL(A[i]));
3. return(-1);

   InsertMaxHeap(Heap A[1..10], key k)
1. if  ((k < 1 ) || (k > 10))
     ThrowException("Illegal Key Value");
2. InsertLinkedList(A[k],k);
   //Add k to linked list holding all keys with value k, i.e A[k]

   ExtractMaxHeap(Heap A[])
1. for(i=10;i>=1 ;i--) {
2.  if ( A[i] != NULL )  {
      //If A[i]==NULL there is no key with value i=10,9,... in heap
3.    return(DeleteLinkedList(A[i]));
      // and skip backward (i--); the first non NULL list
4.  }
// A[i] will provide the key with the highest value;
// Deletion is from the head
// of the list to take constant time.
5.  ThrowException("Underflow");
//If all lists are NULL, there are no keys in Heap.
```

■

Exercise 221.

Professor I. M Nuts claims that he has a function PrintMaxHeap(H), that can print the elements of a heap that is a maxheap in sorted (increasing, a.k.a. non-decreasing) order in $\Theta(n)$ time, where n is the number of keys in the Heap. Moreover his algorithms is comparison-based. Elements in the heap are keys with comparable priorities.

Prove to him that he is NUTS.

Proof.
If this is possible we can sort in $o(n \lg n)$ time by first building a MaxHeap in $\Theta(n)$ (for n keys), and then invoking his PrintMaxHeap for a further $\Theta(n)$. Thus sorting would require $\Theta(n)$ violating the lower bound. ■

5.6 Past heap

Exercise 222.

Give an $O(n \lg k)$-time algorithm to merge k sorted sequences into one sorted sequence, where n is the total number of keys in all the input sequences.

Proof. The idea is simple. Take the largest key of each sequence and insert it into a MaxHeap with k keys. Such a Heap never grows beyong k in size, and has one key representative of each one of the k sequences (the largest key still there and not moved to the output) unless a sequence become empty because all of its keys moved to the output. For each key in the heap we record also the (index of the) sequence from which this key was extracted. For example if 25 was extracted from sequence 4, we add (25,4) into the heap with 25 being the key value and 4 the additional information maintained.

To fill the output sequence we perform n ExtractMax operations from the Heap, each ExtractMax followed by an Insert of the maximum key of the remaining sequence to which the Extracted key belonged. This way the size of the heap is maintained at k; it can only grow below k if one of the sequences becomes empty (i.e. all its key moved to the output already). In our example if 25 of sequence 4 was the key that ExtractMax returns, the next largest key of sequence 4 will be added to the k-key heap from the input sequence 4. If that sequence has no more keys, then we leave the heap deficient of one key. The TOTAL number of ExtractMax operations is going to be n, the number of keys in the output, and each such operation involves a heap of no more than k keys i.e. requires $O(\lg k)$ for an Extract. The subsequent Insert to an Extract also takes that much time. Finding the next largest key of a sequence is an $O(1)$ time operation given that the sequences are sorted. Details follow.

Step 1. Let $j = n$ be the total length of the sorted sequences. Let the tail (largest key) of sorted sequence i be denoted by t_i, $i = 1, \ldots, k$. Take the current k tails t_i, remove each one from corresponding sequence, form a heap based on the key values of the k tails in $O(k)$ time, and then update the t_i to point to the second largest key of the sorted sequences (as largest keys are in the heap and not in the sequence any more).

Step 2. ExtractMax the largest key M of the heap in $O(\lg k)$ and place it in the j-th position of the output sequence. Decrement j by 1. If M came off the l-th sorted sequence (i.e. it was before step 1 $M = t_l$) then remove the current tail of the l-th sequence and insert it into the $k-1$ key heap in $O(\lg k)$, and update t_l appropriately.

Step 3. Repeat Step 2 until $j = 0$.

Step 1 takes $O(k)$ time. In step 2, every time M is removed, it is larger than any of the remaining heap or sorted sequence keys. Time for step 2 is $O(\lg k)$ for ExtractMax M or for HeapInsert of the next largest key of M's sequence. As there are a maximum of n keys that are HeapInsert-ed or ExtracMax-ed in/from the heap the total running is $O(n \lg k)$. ■

Exercise 223.

d-HEAPS. Analysis of d-ary heaps. (Assume MAXheaps in the remainder).

A d-ary heap is like a binary heap but non-leaf nodes have d children rather than 2.

a. How would you represent a d-ary heap in an array?

b. What is the height of a d-ary heap of n elements in terms of n and d?

c. Give an efficient algorithm that extracts the maximum element from a d-ary max-heap. Analyze its running time in terms of n and d.

d. Implement `Insert` for d-ary heaps and analyze its running time.

e. Given an efficient implementation of an operation `Increase(A,i,value)` which first sets $A[i] = \max\{A[i], value\}$ and then properly updates the d-ary max-heap. Analyze the worst-case running time of `Increase`.

Proof. (a) A d-ary heap is to be represented by an one dimensional array A. $A[1]$ stores the root and $A[2], \ldots A[d+1]$ store the d children of the root. The j-th child of node indexed i is $d(i-1)+j+1$ where $1 \leq j \leq d$. Therefore the children of i can have potential indexes $d(i-1)+2$ to $di+1$.
If the indexing starts from 0 i.e. the root is at $A[0]$, then the children of i have indices starting with $di+1$ through $di+d$.
The parent of node i is floor$((i-2)/d+1)$, in other words floor$((i+d-2)/d)$ if the indexing starts from 1. If the indexing starts from zero, it is floor$((i-1)/d)$. A binary heap is a special case for $d=2$.
(b) A heap of height h has $n = 1+d+d^2+\ldots+d^{h-1}+m = (d^h-1)/(d-1)+m$ elements, where $1 \leq m \leq d^h$. Therefore its height is $O(\lg n/\lg d)$.
(c) Extract-Max works the same way as Heap-Extract-Max for binary heaps except that DownHeap(A,i) on d-ary heaps works a little bit differently. It assumes that the d subtrees of node indexed i have the heap property and the maximum is taken over the value stored at i and the values stored in all d children of i. The maximum operation takes time $O(d)$ now, and Extract-Max may call DownHeap $O(h)$ times. By way of part (b.b) $O(h) = O(\lg n/\lg d)$ and therefore the time for DownHeap will be $O(d \lg n/\lg d)$.
(d) Insert for binary heaps works fine for d-ary heaps as well (remember to use the new definition of a parent though). An upward movement is independent of the number of children of a parent. Therefore its running time is $O(h) = O(\lg n/\lg d)$.

(e) This is similar to insertion. If the value k is at most the value of $A[i]$ nothing changes, otherwise the value of $A[i]$ is changed and propagated upwards just as we did in Heap-Insert. The observation is that since the value k is propagated upwards, we only need to change the value of the final destination only, just as we did in Heap-Insert. The following program demonstrates this. Note that parent(i) follows the definition of part (b.a) not that of a binary heap.

```
Heap-Increase-Key (A,i,k)
1. If A[i] >= k then return ;
2. while i>1 and A[parent(i)] < k
3.     do A[i] = A[parent(i)];
4.        i=parent(i);
5. A[i]=k;
```

■

Exercise 224.

The following two methods are proposed to find M_k, the k-th smallest of n keys stored in array $A[0..n-1]$.
a) Sort A and then output the k-th element of the sorted A. This is M_k.
b) Build a MIN-heap out of the n elements of A, and then perform k EXTRACT-MIN operations. The last (i.e. k-th) EXTRACT-MIN operation returns M_k.
Which of the two algorithms is asymptotically faster? Explain, i.e. provide details of efficient implementation of the various steps, and analyze their worst-case running time using asymptotic notation. Use case analysis, if necessary.

Proof. The two implementations are shown in the figure.

```
a. FindMkOne (A[1..n],n)
1. HeapSort(A);
2. return(A[k]);
```

```
b. FindMkTwo (A[1..n],n)
1. BuildMINheap(A,n);
2. for(i=1;i<k;i++){
3.  z=EXTRACTMIN(A);
4. }
5. return(z);
```

(a) FindMkOne requires $O(n \lg n)$ worst-case running time for step 1 (HeapSort can be replaced by MergeSort) and step 2 requires constant time. The total is $O(n \lg n)$
(b) FindMkTwo requires $O(n)$ worst-case running time for the build-heap operation of step 1, and $O(1)$ time for step 5. The loop involves k EXTRACT-MIN operations. Each one takes $O(\lg n)$ time for a total of $O(k \lg n)$. Note that the z returned in step 5, is the result of the k-th EXTRACT-MIN operation, the k-th smallest of the n keys of the input. Total running time is $O(n + k \lg n)$.
Let us asymptotically compare the performance of the two algorithms for various values of k. To make thing less confusing consider algorithm (a) has performance $n \lg n$ and (b) $n + k \lg n$.
Test Case 1. Let us start with small values of k. Let $k = \Theta(1)$, i.e. k is constant. The running time of (a) is $n \lg n$ and (b) is $n + 1 \lg n = \Theta(n)$. The latter algorithm is asymptotically faster.
How much can we increase k beyond a constant and still get the same performance? For what values of k is the term $k \lg n$ comparable to n? This is for $k \lg n = O(n)$, i.e. $k = O(n/\lg n)$. For such values (b) is $\Theta(n)$ and (a) is $n \lg n$ and (b) is asymptotically faster.
Test Case 2. Suppose that $k = \omega(n/\lg n)$. Algorithm (b) has performance $n + k \lg n = n + \omega(n/\lg n) \lg n$ which is $\omega(n + (n/\lg n) \lg n)$ i.e. it is $\omega(n)$.

Test Case 3. However as long as $k = o(n)$, the term $n + k \lg n$ is still $n + o(n \lg n) = o(n \lg n)$, i.e. algorithm (b) is still asymptotically faster that algorithm (a).

Summary Case. A.
Cases 1, 2, and 3 can unified into: if $k = o(n)$ then (b) is asymptotically faster than (a).
Summary Case B. If k is not $k = o(n)$, then $k = \Theta(n)$, and algorithm (b) has performance $n + k \lg n = O(n \lg n)$ and both algorithms have the same asymptotic performance.
Conclusion: For large k such as $k = \Theta(n)$ both have the same asymptotic performance. For any other value of k i.e. $k = o(n)$ (b) is asymptotically faster than (a). ■

Exercise 225.
You are given n arbritrary (but distinct) keys. We are interested in finding the $\sqrt{n}$ keys that are the smallest among the n-keys. Given an asymptotically optimal algorithm for returning/printing those keys in sorted (i.e. increasing) order.

Proof. Generalization of a prior Problem. Think! ■

5.7 An application of heaps: Huffman's data compression

Exercise 226.

Compress the text below (alphabetic characters only) using Huffman coding. What is the total number of bits used for the compressed text (ignore table storage requirements)?

ABRACADABRAZABRAZEBRA

Proof.

```
ABRACADABRAZABRAZEBRA
-x.-,-'-x.-/-x.-/;x.-
A: 8, B: 4, C: 1  D: 1, E: 1, R:4  Z: 2
```

```
                    Frequency   PrefixCode   #of bits   Total # bit
                                             in code    in 'compressed' text
         21             8     A:  1            1          8
        / \             4     B:  011          3         12
      13   \            4     R:  010          3         12
      /\    \           2     Z:  0011         4          8
     /  \    \          1     E:  0010         4          4
    5     \   \         1     D:  0001         4          4
    /\     \   \        1     C:  0000         4          4
   /  \     \   \                                        ----
  2    3     8   \                                        52 <<<<<Answer
 /\    /\    /\   \     Original Text (ASCII) Size : 21B            or '168bit'
C  D  E  Z  R  B  A     Compressed Text Size       : ceil(52/8)=7B  or ' 52bit'
1  1  1  2  4  4  8     compression factor           21/7=3             168/52=3.x
```

■

Exercise 227.

Compress the string (ignore spaces) of A,C,G,T,W using Huffman coding. How many bit and how many byte do we need?

```
GGACW WWGGG TAACW WGGAA TGACA
```

Proof.

```
Problem 9
  GGACW WWGGG TAACW WGGAA TGACA ASCII Bits=25*8=200bits in 25B.
           C=3 , A = 7 G = 8, T=2, W =5
                       25
                      /   \     Huffman bits
                    10     \    = 2*3+3*3+5*2+7*2+8*2 = 55 bits in 7B
                   /  \     \
                  5    \     15
                 /  \   \    /  \
                T:2 C:3 W:5 A:7  G:8
                000 001  01 10   11
```

■

Exercise 228.
Using Huffman coding, how many bits do you need to compress this text of characters (count only bits of character representation)?

```
ILOVEALGORITHMS
```

Proof. Be careful! Love (of algorithms) hurts and takes time and practice! I just couldn't resist! ■

5.8 Distribution-based sorting: CountSort and RadixSort and friends

Exercise 229.

Name the most time-efficient algorithm and specify its worst-case running time among `HeapSort`, `CountSort` and/or `RadixSort` that solves each one of the problems below. If `RadixSort` is the answer, mention how many iterations of `CountSort` are involved in it. In case of a tie, a stable beats a non-stable algorithm. Indicate running time as needed.

1. Sorting n keys in the range $0,\ldots,n-1$.
2. Sorting n keys in the range $0,\ldots,2^n-1$.
3. Sorting n keys in the range $0,\ldots,n^{\lg n}-1$.
4. Sorting n keys in the range $0,\ldots,n^2-1$.

Proof.
Few things to note.
CountSort-related. $\Theta(n+k)$ of CountSort: n is the number of keys k is the range of consecutive values. In the definition of the problem the range of values is $0..k-1$. What if they are $1..k$? Subtract one at the start from every input key, sort, and then add one. Or assume the range that subsumes $1..k$ is the $0..k$. In that case in the running time expression one nominally adds one i.e. $\Theta(n+k+1)$. The latter is of course $\Theta(n+k)$: nothing changes asymptotically. After the sorting add one to the output values.
RadixSort-related. Since RadixSort uses CountSort make sure for the per-round running time (i.e. CountSort running time) the $\Theta(n+k)$ is as close to $\Theta(n)$ as possible. This means choose $k=\Theta(n)$. Thus if the key values are $0..n^3-1$, split the key value into roughly three 'digits' each one take n values. Thus relate that for $0..999$ we do not do one CountSort with $k=1000$, but instead three rounds of CountSort, which is RadixSort on $d=3$ digits, and each digit has $k=10$: $10^3=1000$. If for $0..n^3-1$ we use 'three' digit, each digit using n values $(0..n-1)$, a round of CountSort, will take $\Theta(n+k)$, where the k is that of a single digit i.e. $k=n$. Moreover we need three rounds of CountSort for a total running time $3\cdot\Theta(n+n)=\Theta(n)$. It is thus also important that the number of rounds (three in this example) is also constant so that the total running time is still $\Theta(n)$. How do we find the number of digits for a chose radix, say $k=n$? Simple: divide the logarithm of the number of values of the range with the logarithm of the radix k. Thus $\lg(n^3)/\lg k=\lg(n^3)/\lg n=3$. What if the range is $0..n^{\lg n}-1$? Doing what we proposed earlier the number of rounds of CountSort in RadixSort is the number of digits the range $0..n^{\lg n}-1$ will be split into. $\lg(n^{\lg n}/\lg n=\lg n$, thus we have $\lg n$ digits or equivalently $\lg n$ rounds of CountSort. Then $\lg n\cdot\Theta(n+n)=\Theta(n\lg n)$! Finally why do we start RadixSort from the rightmost digit? Be reminded on the inductive proof of correctness. If the integers are sorted based on the $i-1$ rightmost digit, when we examine the i rightmost digit (i digit from the right) of two integers a,b and let them be a_i and b_i, then if $a_i<b_i$ or $a_i>b_i$ we will correctly sort a,b in iteration i depending on whether $a_i<b_i$ (i.e. a before b) or $a_i>b_i$ (i.e. b before a). Moreover if $a_i=b_i$ we do nothing since the relative order of a,b cannot be resolved in iteration i further. We use the correct work (induction assumption) done in the previous $i-1$ iterations: we use the relative order of a,b as decided at the end of iteration $i-1$ and iteration i does not disturb it.
(1) CountSort $\Theta(n)$, (2) HeapSort $O(n\lg n)$, (3) RadixSort, $\lg n$ rounds, $O(n\lg n)$ beats in a tie-breaker HeapSort, (4) RadixSort, 2 rounds, $\Theta(n)$. ■

Exercise 230.
Answer the following questions briefly by justifying your answers. If needed provide asymptotically tight bounds as answers. ('BucketSort' is the probabilistic version of BucketSort; not the linked list version of CountSort).
(a) What is the worst-case running time of 'BucketSort' ?
(b) What is the running time of 'BucketSort' ?
(c) What is the expected running time of 'BucketSort' ?

Proof. (a) $T(n) = \Theta(n^2)$.
(b) $T(n) = O(n^2)$.
(c) $T(n) = \Theta(n)$.

■

Exercise 231.

Give an efficient algorithm that works in-place which reorders an array A of n integer keys so that all negative keys appear before the positive keys in the output sequence. Justify your answer. Your algorithm must be **efficient** (i.e. one can not give an algorithm whose asymptotic worst-case performance can be improved) and **should work in place** (i.e. you can not use an extra array, and in summary the extra space you use besides the one needed for storing A is constant).

Proof. Think! ■

Exercise 232.

In an earlier problem set you were asked to sort an array of 0 and 1. The following solution was proposed.

```
 Sort(Y[1..n],n)
0. count=0;
1. for (i=1;i<=n;i++)
2.  if (Y[i]== 0) count++;
3. for (i=1;i<=count;i++)
4.   Y[i]=0;
5. for (i=count+1;i<=n;i++)
6.   Y[i]=1;
```

Now array Y contains not only $Y[i].k$ the key value that is 0 or 1, but also contains $Y[i].info$. Think about k being an indicator of PhD student or MS student, and $Y[i].info$ being the last name of a student (no students of the same matriculation level have the same last name!). Give a time and space efficient algorithm that sorts by last name but PhD students appear last and MS students first!

Proof. The count works up to a point. Output needs to filled as in CountSort. ■

Exercise 233.

Which algorithm among heap-sort, radix-sort, count-sort is asymptotically faster for each one of the following sorting problems. Justify your answer. In case of a tie, the tie-breaker is an in-place algorithm.
(a) Sorting of n keys, where each key takes values in the range $0 \ldots n^4 - 1$.
(b) Sorting of n keys, where each key takes values in the range $0 \ldots n^{2\lg n} - 1$.
(c) Sorting of n keys, where each key takes values in the range $0 \ldots n^n - 1$.

Proof. (a) We use 4-digit radix-n sorting of the keys in the range from 0 to $n^4 - 1$. We use 4 rounds of count sort (i.e. radix-sort) for a total time of $O(4n) = O(n)$. Radix-Sort beats Count-Sort that would require $O(n^4)$ or heap-sort.
(b) If we use $2\lg n$-digit radix-n sorting of the keys we need time $\Theta(\lg n)$ rounds of count-sort for sorting keys in the radix-n range i.e. $O(n)$ time per round for the Radix-Sort algorithm. Total is $\Theta(n \lg n)$. (One-round) Count-Sort requires $O(n^{2\lg n})$ i.e. too much. Heap-Sort on the other hand also takes $O(n \lg n)$.
There is a tie between HeapSort and Radix-Sort. The tie-breaker is the in-place property of Heap-Sort. Heap-Sort wins.
(c) If we use Radix-Sort for the keys in the range from 0 to $n^n - 1$, we need n rounds of Count-Sort i.e. $\Theta(n^2)$ time since we will view the keys as n-digit radix-n integers. Count-Sort itselft requires too much time for initialization alone $\Theta(n^n)$. Heap-Sort however uses only $O(n \lg n)$. The winner is Heap-Sort by a margin.

■

Exercise 234.

RadixSort with HeapSort? RadixSort of n keys in the range $0 \dots k^d - 1$ done in class involved d rounds of CountSort from least significant to most significant digit in time $\Theta(d(n+k))$. We decide instead of using CountSort to use an in-place sorting algorithm and in particual HeapSort.

(a) What would the running time of RadixSort be if HeapSort replaces CountSort. Justify your answer.

(b) Would RadixSort (with HeapSort in place of CountSort) work correctly then? That is would it sort arbitrary integers keys in the range $0 \dots k^d - 1$? Explain.

Proof. For (a) you are allowed to change the structure of the heap to use the CountSort available information. ■

5.9 Fake coins

Exercise 235.

We have 4 coins all of the same weight except one that is fake and may weigh LESS or MORE than the other coins (we don't know what the case is). We also have a balance scale; any number of coins can be put on one or the other side of the scale at any one time and the scale will tell us whether the two sides weigh the same or which side weighs more (or less). If two coins are weighed (one on each side of the scale) and they do not weigh the same, we can't decide by this weighing alone which of the two is the fake.

Can you find the fake coin with only 2 weighings? Explain (and justify your answer).

Proof. We indicate a weight of A and B by showing A:B. As stated, the problem has 4 states (i,F), where i is the coin label and F its status as fake. (Implied is that the other coins are genuine.) With two weighing (Equal or NotEqual) we are done. The decision tree is binary and of height logarithmic to the leaves/states. Here is how to find the fake coin.

```
A:B  --- == ---> (C or D is F)--- A:C --- ==     D is F; A,B,C genuine
     |                                    |
     |                                    -- !=  C is F; A,B,D genuine
     --- != ---> (A or B is F)--- A:C --- ==     B is F; A,C,D genuine
                                          |
                                          -- !=  A is F; B,C,D genuine
```

We now rephrase the problem by also asking to find whether the fake coin is lighter or not and allowing for the possibility that all are genuine. The problem has 9 states: eight of the form (i,S) where i is the coin label of a fake coin and S an L for lighter or H for heavier with the ninth state being all-g i.e. all are genuine. As $\log_3(9)$ is still a two we can use a ternary decision tree and try with two weighings with it. A lower bound does not mean it is an upper bound as well. Try three if you can't do it with two weighings! ■

Exercise 236.
We have 27 coins all of the same weight except one that is fake and weighs LESS than the other coins. We also have a balance scale; any number of coins can be put on one or the other side of the scale at any one time and the scale will tell us whether the two sides weigh the same or which side weighs more (or less). Can you find the fake coin with only 3 weighings? Explain (and justify your answer).

Proof. Split into 3 groups. ■

5.10 More on sorting

Exercise 237.

How to turn any sorting algorithm that is not stable into a stable sorting algorithm by paying a space penalty and also a comparison penalty.

Proof. If input sequence consists of keys $x_1, \ldots, x_n$ extend each key by a pair (x_i, i), where second element gives position in input sequence. This makes all keys distinct. Sort the extended keys (pairs) where a key is a pair, not just x_i. This scheme may at most double the running time and uses $O(n)$ additional memory. Two keys (x_i, i) and (x_j, j) are compared as follows. If $x_i \neq x_j$ report the result of the comparison of x_i and x_j; if $x_i = x_j$ report the result of the comparison to be that of i and j. We increase the input by n integer index values or n integer pointer values. The comparison count also potentially doubles; not however when we compared indexes we do a subtraction not call the compare function a second time! Suppose the input sequence $x_1, \ldots, x_n$ is to be sorted in non-decreasing order. As some x_i's may be equal we create for key x_i a new key which is the pair (x_i, i). The first element of the pair is x_i and the latter is the index/position of x_i in the input sequence. The input to the sorting algorithm is going to be not $x_1, \ldots, x_n$ but $(x_1, 1), \ldots, (x_n, n)$. Note that the keys of the newly formed input are always distinct; even if $x_i = x_j$ for some $i \neq j$, the second elements of the two pairs are distinct as $i \neq j$. As any of the sorting algorithms works fine with distinct keys (ie stability is no issue then), it will sort the n augmented keys. If key x_i was before x_j in the input sequence, it means $i < j$ and therefore $(x_i, i) < (x_j, j)$, and therefore in the output sequence (x_i, i) will appear before (x_j, j). When the output sequence is obtained, we reobtain the original keys from the pairs (by reading off the first element of the pair) to get the output sequence for the original problem. ■

5.11 Some challenging problems

Exercise 238.

Is it possible to print the n keys of a MaxHeap in sorted order by performing $\Theta(n)$ comparisons? Explain and Justify your answer.

Proof. Suppose that it was possible to print the keys of a heap in sorted order in $O(n)$ worst-case time. As MakeHeap running time is $O(n)$, and printing a heap $O(n)$ more, sorting n keys would thus take $O(n)$ time in the worst case, a contradiction to the comparison-tree derived lower bound of $\Omega(n \lg n)$. ■

Exercise 239.

Expected number of assignments in line 4 of MIN or (in the example below) MAX. (Assume keys are distinct and 'random').

```
 MAX(A[1..n])
1.  M =A[1];
2. for (i=2; i <= n; i++)
3.  if A[i] > M
4.    M =A[i];
5  return(M);
```

Proof. Among the n distinct numbers only one number can be the maximum. The probability that M randomly chosen is this maximum is $1/n$.
When line 4 is executed M is set to $A[i]$, i.e. $A[i]$ is the maximum of all keys $A[j]$, where $1 \leq j \leq i$, and therefore $A[i] \geq A[j]$. Fix a value for i. The probability that line 4 is executed is the probability that $A[i]$ is the maximum of all i keys $A[j]$, $j \leq i$. This is $1/i$: the probability that $A[i]$ is the max of $A[1] \ldots A[i]$.
Line 4 is either executed or not, i.e. $M_i = 1$ or $M_i = 0$. The probability that $M_i = 1$ is given by $1/i$. The probability that $M_i = 0$ is 1 minus the probability that $M_i = 1$.

$$E[M_i] = 1 \cdot Prob(M_i = 1) + 0 \cdot Prob(M_i = 0) = 1 \cdot (1/i) + 0 = 1/i.$$

Let $M = M_1 + M_2 + \ldots + M_n$. M indicates the number of times line 4 was executed. Then by the linearity of expectation we have that

$$E[M] = E[M_1 + M_2 + \ldots + M_n] = E[M_1] + E[M_2] + \ldots + E[M_n] = 1 + 1/2 + 1/3 + \ldots + 1/n = \ln n + O(1)$$

where the last term is by Euler's gamma function. ■

Exercise 240.

You are given a set A of n numbers (real, integer, positive, or negative) out of which only m are distinct where $m \leq \sqrt{n}$. Find the m distinct values and print them in sorted order. For example if A looks like
$\{\pi, e, 5, 10^7, \pi, 10^7, e, 5, 10^7, e, \pi, 5, 10^7, e, e, 10^7\}$
we have $n = 16$ but $m = 4$. Your algorithm should be faster than an $O(n \lg n)$ generic algorithm for a variety of choices of m. For example, if $m = O(1)$ your algorithms should be $O(n)$ which is faster than $O(n \lg n)$. Similarly if $m = O(\lg n)$. **Additional question.** How would you sort the n keys in the same asymptotic time?

Hint. Give an $O(n \lg m)$ algorithm.

Proof. No solution is provided. Great as an interview question though. There are several ways to deliberate about this problem.
(a) Use arrays and ... Countsort-type methods. But beware: CountSort, BucketSort or RadixSort can not be used. Keys might not be integers. They might be arbitrary key values. Moreover noone says that the keys are from a given range of k or m consecutive values $0 \ldots m-1$. If this is a distraction
(b) a solution that is not space efficient or space optimal is available with a follow-up Problem Set. However the problem is completely rephrased! So you have got to figure out what problem it is!

■

Exercise 241.

Facepeek. You are an early adopter (or is it adoptee) of a social network called Facepuke. You have lots of friends, in fact n of them; each friend is represented by a 64-bit integer. You want to sort ... out ... your friends. What sorting algorithm would you use and what would its running time be? (Give name and running time only, using as tight asymptotic notation as possible.)

Proof. Radix-sort, $\Theta(n)$, around 8 or 4 rounds. ■

Exercise 242.

Book-keeping. The NJIT library has N books and every book belongs to one of M publishing houses. Array $A[1..N]$ maintains the publisher of a book, i.e. if $A[i] = j$, then the publisher of book i is j, where $j = 1, \ldots, M$. Give an algorithm whose worst-case running time is $O(N+M)$ that calculates the number of books belonging to publisher j.

Proof. CountSort type. ■

Exercise 243.

You are given two sorted arrays $A[1..k]$ and $B[1..m]$ of $k = n^2$ and $m = \lg n$ keys respectively. All keys in A are distinct from each other; all B keys are also distinct from each other. We would like to find the intersection C of A and B i.e. $C = A \cap B$, where C contains the keys that are common to A and B. Intersection C does not need to be sorted. Provide a time and space efficient algorithm and justify your answer.

Anti-Hint. There is a sublinear algorithm.

Proof. Do a binary search of each element of B into A. Since A is sorted binary search on A will take $O(\lg n^2) = O(\lg n)$ for each $b \in B$. Thus for all $b \in B$ the cost would be $m \times O(\lg n)$. Since $m = \lg n$ this gives $O(\lg n \cdot \lg n) = O(\lg^2 n)$. (The sublinear hint.) The algorithm should be time efficient (i.e. asymptotically time efficient) and also space efficient (asymptotically speaking as well). The intersection $A \cap B$ contains no more elements than the smaller of the two sets which is B. But space for C needs to be tight: we run the algorithm once to determine the cardinality of C i.e. $|C| = |A \cap B|$.

```
  Intersect(A,B,k,m,n)
1.  c = 0 ;                // c will store |C|, C=intersect(A,B)
2.  for(i=0; i< m ; i++) { // m= |B|
3.    indx = BinSearch(B[i],A);
4.    if indx != -1 c++;
5.  }
6.  C = allocate(c);       // allocate space for C
7.  c = 0 ;
7.  for(i=0; i< m ; i++) { // m= |B|
8.    indx = BinSearch(B[i],A);
9.    if indx != -1 {
10.      C[c]=B[i] ;  c++ ;
11.   }
12. }
```

■

Exercise 244.

Geegelo search engine. I.M.NJIT is interested in finding all the documents of the web that contain both keywords DATA and STRUCTURES. I. M. NJIT thus searches for DATA and STRUCTURES by typing in Geegelo search engine's textbox first the index term (a.k.a. keyword) `DATA` and then `STRUCTURES`. The engine though returns for the first query a sequence D of document numbers (document IDs i.e. docID's) associated with all documents containing the word `DATA`, and a sequence S of dociDs for those documents containing the word `STRUCTURES`. The length (number of documents) of D is n^2 and that of S is $\sqrt{n}$. You need to generate the output sequence A that contains all the docIDs whose documents contain both DATA and STRUCTURES; no duplicates should be output, and sequence A does not need to be sorted (on docIDs). How much space and time does I.M.NJIT need to determine the answer A using a space and time efficient algorithm that would determine all the docIDs of the documents containing both index terms DATA, STRUCTURES? Explain.

Proof. This is similar to the previous problem. The difference is that neight D nor S is sorted. Sorting the former D with HeapSort (space and time efficient) takes $O(n^2 \lg n)$, but sorting the latter S takes $O(\lg n \lg\lg n)$. In the former case binary search is still $O(\lg^2 n)$, but in the latter case $O(n^2 \lg\lg n)$. Thus the overall time for the first case is $O(n^2 \lg n) + O(\lg^2 n) = O(n^2 \lg n)$, and the latter case is $O(\lg n \lg\lg n) + O(n^2 \lg\lg n)$. The latter case is preferable and thus $O(n^2 \lg\lg n)$ is the running time.

```
  Intersect(A,B,k,m,n)
0.  HeapSort(B,m);
1.  c = 0 ;                // c will store |C|, C=intersect(A,B)
2.  for(i=0; i< k ; i++) { // k= |A|
3.    indx = BinSearch(A[i],B);
4.    if indx != -1 c++;
5.  }
6.  C = allocate(c);       // allocate space for C
7.  c = 0 ;
7.  for(i=0; i< k ; i++) {
8.    indx = BinSearch(A[i],B);
9.    if indx != -1 {
10.      C[c]=A[i] ;  c++ ;
11.   }
12. }
```

■

Exercise 245.

You are given two sorted arrays $A[1..k]$ and $B[1..m]$ of $k = m = n$ keys respectively. All keys in A are distinct from each other; all B keys are also distinct from each other. We would like to find the union C of A and B i.e. $C = A \cup B$, where C contains the keys that are in A or B and each key in C is listed only once. Union C does not need to be sorted. Provide a time efficient algorithm and justify your answer.

Proof. Strategically, the space efficiency requirement has been dropped. We copy all keys of A and B into C and HeapSort C then. Time is $O(n \lg n)$ for sorting. Everything else is linear to n.

```
  Intersect(A,B,k,m,n)
1.  for(i=0; i<n ; i++)
2.    C[i]   = A[i];
3.  for(i=0; i<n ; i++)
4.    C[n+i] = B[i];
5.  HeapSort(C,2*n);
6.  c = 0 ; /C[0] is alway in the union
7.  for(i=1 ; i < 2*n ; i++)
8.   if C[i] != C[c] {
9.      c++ ;
10.     swap(C[i],C[c]); //it can be C[c]=C[i] instead
11.  }
12.  else {
13.    ;  // do nothing
14.  }
15.  return(< C , c+1>);// C[0...c] or C[0..c-1] is the intersection
                       // C[c+1..] or C[c..   ] is the union
                       // Exercise: decide on c vs c-1 !
```

■

Exercise 246.

You are given three sorted arrays $A[1..k]$, $B[1..m]$, $C[1..n]$ of $k = n^2$, $m = \lg n$, and n keys respectively. All keys in A are distinct from each other; all B keys are also distinct from each other and so are those of C. We would like to find the intersection D of A, B, and C, i.e. $D = A \cap B \cap C$, where D contains the keys that are common to A and B and C. Intersection D does not need to be sorted. Provide a time efficient algorithm and justify your answer.

Anti-Hint. There is a sublinear algorithm.

Proof. Since A, B, C have sizes n^2, $\lg n$, n, it means that D can have size at most $\lg n$, the size of the smallest of the three sets. In other words D must be a subset of B. Perform a binary search of B into A and C. This takes $\lg n$ (the size of B) times log the size of A (n^2) plus log the size of C (n). The overall running time is thus $\lg n \cdot O(\big(\lg(n+1) + \lg(n^2+1)\big)) = O(\lg^2 n)$.

```
  Intersect(A,B,C,k,m,n) {
1.  k = 0; D = new(lgn);
2.  for(i=0 ; i < D.length ; i++) {
3.    D[i] = empty;
4.  }
5.  for(i=0 ; i < B.length ; i++) {
6.   a = BinSearch(B[i],A);
7.   c = BinSearch(B[i],C);
8.   if (a != -1 ) && (c != -1) {
9.     D[k] = B[i]; k++;
10.  }
11. }
12. return(<D, k>);  // D[0..k-1] is the intersection
```

■

Exercise 247.

Give an $\Theta(n \lg n)$-time algorithm that, given a set A of n integers determines whether or not there exist two integers x, y of A whose sum $x+y$ is exactly 123.

Proof. We need to determine x, y in A such that $a+b=123$. Consider an element x of A; if $y=123-x$ is also an element of A, then we have found such a pair, x and $y=123-x$, since $x+y=x+123-x=123$. For every element x of A, we check whether the result $123-x$ appears in A. The latter search procedure is facilitated by sorting A (line 1); then the search of $123-x$ into A can be done by binary search (line 4) and takes $O(\lg n)$ per element x searched. Since there are n elements x in A, and we go through them through the loop of lines 2-7, the total search time is n times $O(\lg n)$, i.e. $O(n \lg n)$ (loop of lines 2-7). In addition, sorting the n elements of A also requires $\Theta(n \lg n)$ if we use heapsort (line 1). Therefore total time is $\Theta(n \lg n)$.

```
  Searchfor(A[0..n-1],x,y) {
1.  A=HeapSort(A);
2.  for(i=0;i<n-1;i++) {
3.     x=A[i];
4.     index= BinarySearch(123-x,A,n) ;
5.     if (index != -1)  // -1 : NOT FOUND
6       return(YES:<x,A[index]>);
7.  }
8.  return(NO);
```

■

Exercise 248.

We want to find the t largest of n keys in reverse sorted order using a comparison-based algorithm.
Find the algorithm that implements each one of the following methods with the best asymptotic worst-case performance, analyze its worst-case running time, and express it in terms of both n and t.

(a) Sort the n keys, and list the t largest of them.

(b) Build a Max-Heap out of the n keys, and call ExtractMax t times.

Proof.
(i) If we sort n keys using heapsort this takes $O(n \lg n)$ times. We can find the t largest keys in the sorted output $A[1..n]$ by just printing the keys of the subarray $A[n-t+1..n]$ in $O(i)$ time larger to smaller. Total running time is $O(n \lg n + t)$ where $+t$ counts for printing the t largest keys.
(ii) Build a MaxHeap of the n keys in $O(n)$ time. Call then ExtractMax t times to print the t largest keys requires $O(\lg(n-t+1)) = O(\lg n)$ time for the t-th largest key, i.e. a total of $O(t \lg n)$. The total running time is now $O(n + t \lg n)$.

```
  AlgorithmPartA (A,n,i)
1. HeapSort(A,n);
2. for(i=n;i>=n-i+1;i--)
3.  print("(n-i+1)-st largest is",
          A[i] );
```

```
AlgorithmPartB  (A,n,i)
1. BuildMaxHeap(A,n);
2. for(i=n-i+1;i>=n-i+1;i--)
3.  print("(n-i+1)-st largest is",
          ExtractMaxHeap(A));
```

If $t = o(n)$, the non-HeapSort algorithm has performance $O(n + t \lg n) = o(n \lg n)$, where as the HeapSort-based one has $\Omega(n \lg n)$. Otherwise both are equivalent ■

Exercise 249.

Array $A[1..n]$ contains n keys that are comparable. We would like to determine the $\lg n$ largest of the n keys and display them in the output in reverse sorted order (larger to smaller). Give a space and time efficient algorithm, explain the algorithm, analyze its worst-case running time and space complexity. Make sure your algorithm does not do more than $2n + o(n)$ comparisons. Justify your arguments. You may assume that n is a power of two and all the input keys are distinct and in no particular order.

Proof.

```
   HereItis(A[1...n],n)
1. BuildMaxHeap(A,n)        // 2n comparisons max, Theta(n) time.
2. for(i=1; i <= lgn ; i++) // O(lgn) per Extract x lgn Extract
3.   print(ExtractMax(A,n);
```

We first run BuildMaxHeap in time $\Theta(n)$ or at most $2n$ comparisons total. Then we perform $\lg n$ ExtractMax operations. This would take $(\lg n) \cdot O(\lg n)$ i.e. a total of $O(\lg^2 n)$ time. Total time is $\Theta(n)$. Comparison-count wise $2n$ comparisons are because of BuildMaxHeap. Each one of $\lg n$ ExtractMax operations takes no more that $2\lg n$ comparison for DownHeap (a.k.a Heapify), i.e. at most $2\lg^2 n$. Thus the total comparison count is upper bounded by $2n + 2\lg^2 n = 2n + o(n)$ as needed. ■

Exercise 250.
Array $A[1..n]$ contains n keys that are comparable. Find the smallest, third smallest and n-th smallest of the n keys by performing no more that $3n+o(n)$ comparisons. Give briefly the algorithm, reason about its correctness, and describe the comparison-count as accurately as possible to conclude that it is indeed $3n+o(n)$. Justify your arguments. You may assume that n is a power of two and all the input keys are distinct and in no particular order.

Proof.
Build a MinHeap of the n keys in at most $2n$ comparisons. Then perform three ExtractMin operations. We collect the smallest (first ExtractMin) and the third smallest keys (third ExtractMin) and discard the second ExtractMin (and second smallest key). The three ExtractMin operations are $O(\lg n)$, each or collectively all of them, which is $o(n)$ as needed. The n-th smallest is the Maximum. We find the maximum of the n keys separately in $n-1$ comparisons (in fact $n-4$ as the three smallest keys do not participate in this step)). Total number of comparisons is thus at most $2n+o(n)+n-4=3n+o(n)$ as needed. ■

Exercise 251.
Find the $n/\lg n$ smallest of n keys in no more than $4n$ comparisons.

Proof.
Build a MinHeap in $2n$ comparisons (upper bound) and then perform $n/\lg n$ ExtractMin operations each requiring at most $2\lg n$ comparisons for a total of no more than

$$2n + (n/\lg n)2\lg n = 4n,$$

comparisons. ■

Exercise 252.
Find the 4 smallest of n keys in no more than $2n+2\lg n$ comparisons.
Note: Can be done also in $2n+2\lg n-c$, where $c=2,3,4$.

Proof.
Find the smallest and second smallest of n keys in $n+\lceil \lg n \rceil -2$ comparisons. Do likewise for the third and fourth smallest, in $n-2+\lceil \lg (n-2) \rceil -2$ comparisons. The total is no more than

$$2n+2\lg n-4,$$

comparisons. ■

Exercise 253.

Give a **TRUE** or **FALSE** for each one of the statements below. Any answer other than a full TRUE or FALSE will be considered wrong. All algorithms refer to the referenced ones.

(1) MergeSort is stable.

(2) MerseSort sorts in-place.

(3) QuickSort is stable.

(4) QuickSort sorts in-place.

(5) HeapSort is stable.

(6) HeapSort sorts in-place.

(7) CountSort is stable.

(8) CountSort sorts in-place.

Proof.
TRUE, FALSE (reference algorithm), FALSE, FALSE, FALSE, TRUE, TRUE, FALSE. ■

Exercise 254.

Give a **TRUE** or **FALSE** for each one of the statements below. Any answer other than a full TRUE or FALSE will be considered wrong. All algorithms refer to the referenced ones.

(1) Among SelectionSort, MergeSort and QuickSort the asymptotically fastest algorithm is MergeSort.

(2) Sorting n keys in the range $0, \ldots, n^2 - 1$, RadixSort is asymptotically faster than HeapSort.

(3) Sorting n keys in the range $0, \ldots, 2^n - 1$, RadixSort is asymptotically faster than MergeSort.

(4) Sorting n keys in the range $0, \ldots, n^7 - 1$, RadixSort is asymptotically faster than MergeSort.

(5) Sorting n keys in the range $0, \ldots, n^n - 1$, RadixSort is asymptotically faster than HeapSort.

Proof.
TRUE, TRUE, FALSE, TRUE, FALSE. ■

Exercise 255.

Give a **TRUE** or **FALSE** for each one of the statements below. Any answer other than a full TRUE or FALSE will be considered wrong. All algorithms refer to the referenced ones.

(1) On the increasing input sequence $\langle 1,2,\ldots,n\rangle$, QuickSort is asymptotically faster than InsertionSort.

(2) On the decreasing input sequence $\langle n,n-1,\ldots,1\rangle$, QuickSort is asymptotically faster than InsertionSort.

(3) On the increasing input sequence of integers $\langle 1,2,\ldots,n\rangle$, InsertionSort is asymptotically faster than CountSort.

(4) On the decreasing input sequence $\langle n,n-1,\ldots,1\rangle$, InsertionSort is asymptotically faster than CountSort.

(5) On the decreasing input sequence $\langle n,n-1,\ldots,1\rangle$, InsertionSort is asymptotically faster than MergeSort.

(6) On the decreasing input sequence of integers $\langle n,n-1,\ldots,1\rangle$, MergeSort is asymptotically faster than CountSort.

(7) On the increasing input sequence $\langle 1,2,\ldots,n\rangle$, InsertionSort is asymptotically faster than HeapSort.

(8) On the decreasing input sequence of integers $\langle n,n-1,\ldots,1\rangle$, HeapSort is asymptotically faster than QuickSort.

(9) On the increasing input sequence of integers $\langle 1,2,\ldots,n\rangle$, MergeSort is asymptotically faster than CountSort.

(10) On the decreasing input sequence of integers $\langle n,n-1,\ldots,1\rangle$, MergeSort is asymptotically faster than CountSort.

(11) On the reverse-sorted sequence $\langle n,n-1,\ldots,1\rangle$, QuickSort is asymptotically faster than MergeSort.

(12) On the decreasing input sequence $\langle n,n-1,\ldots,1\rangle$, RadixSort is asymptotically faster than HeapSort.

Proof.
FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE, FALSE, FALSE, TRUE. ■

Exercise 256.
Give a **TRUE** or **FALSE** for each one of the statements below. Any answer other than a full TRUE or FALSE will be considered wrong. All algorithms refer to the referenced ones.

(1) Sorting n keys in the range $0, \ldots, n^7 - 1$, RadixSort is asymptotically faster than MergeSort.

(2) Sorting n keys in the range $0, \ldots, n^2 - 1$, RadixSort is asymptotically faster than HeapSort.

(3) Sorting n keys in the range $0, \ldots, n^n - 1$, RadixSort is asymptotically faster than HeapSort.

(4) Sorting n keys in the range $0, \ldots, 2^n - 1$, RadixSort is asymptotically faster than MergeSort.

Proof.
TRUE, TRUE, FALSE, FALSE. ■

Exercise 257.
TRUE or FALSE?

- QuickSort is stable.
- RadixSort is stable.
- CountSort is stable.
- CountSort sorts in-place.
- QuickSort sorts in-place.
- We can optimally in-place sort n keys in the range $0,\ldots,n-1$ by using count-sort (i.e. one round of radix-sort).
- Among count-sort, insertion-sort, quick-sort, and merge-sort the most efficient way to in-place sort n keys in the range $0,\ldots,n-1$ is insertion-sort.
- Among merge-sort, quick-sort, insertion-sort and heap-sort the most efficient way to in-place and stably sort n keys in the range $0,\ldots,n-1$ is by using quick-sort.
- Among radix-sort, insertion-sort, quick-sort, and heap-sort the most efficient way to stable-sort n keys in the range $0,\ldots,n^2-1$ is radix-sort.
- Among merge-sort, quick-sort, insertion-sort and count-sort the most efficient way to in-place and stably sort n keys in the range $0,\ldots,n-1$ is by using count-sort.
- A quicksort algorithm, where the splitter is chosen to be the median of the sequence, runs in time $\Omega(n^2)$ in the worst case.
- A comparison-based sort of 5 keys requires at least 7 comparisons (in the worst case).
- Suppose that in bucketsort we sort the buckets using Quicksort (the one where the splitter is the last key of the subsequence). Then bucket sort still runs in linear expected time but its worst case time becomes $O(n\lg n)$.

Proof.
FALSE, TRUE, TRUE, FALSE, FALSE, FALSE, TRUE, FALSE, TRUE, FALSE, FALSE, TRUE, FALSE. ■

Exercise 258.

How could you modify quicksort (give pseudocode or describe convincingly the changes) so that its worst-case performance is $O(n \lg n)$? Describe your changes, analyze the worst-case running time of the modified quicksort and show that it is indeed $O(n \lg n)$ in the worst-case. It is more suited as a review problem at the end of the semester!)

Proof. Pick the splitter (pivot) key to be the median of the keys of the input participating in the recursive call. Finding the median requires linear time (though with a very large constant). However it guarantees a 50-50 split (plus or minus one key). Thus the recurrence for the running time of this modified quick-sort will be (plus or minus one key): $T(n) = 2T(n/2) + Cn$, where C is a big constant. The asymptotic solution is $\Theta(n \lg n)$, which is acceptable and asymptotically faster than $\Theta(n^2)$ in the worst-case. Pseudocode-wise we can just change line 1 of Partition slightly.

```
OLD CODE
1.  splitter= A[r];
NEW CODE
    // DSelect 'returns' the index of the statistic not the statistic itself
1a. splitter-index=DSelect(A,floor((r-l+1)/2),l,r);
1b. swap(A[r],A[splitter-index]);
1c. splitter=A[r];
```

■

Exercise 259.

Professor I. M. Good devised a variant of Partition, called GoodPartition that guarantess a 10%-90% split around the splitter no matter what the input sequence is. The new version of quicksort that uses Good Partition is shown below.

```
  GoodQuickSort(A,n)
1. GoodQuicksort(A,0,n-1);

 GoodQuicksort(A,l,r);
1. if (l<r) {
2.  m=GoodPartition(A,l,r);  // splitter has become A[m]
3.  GoodQuicksort(A,l,m-1);  // sort recursively all keys <= A[m]
4.  GoodQuicksort(A,m+1,r);  // sort recursively all keys >= A[m]
   }
```

What is the running time of GoodQuicksort on any input sequence of n keys? Justify your answer.

Show that the minimum depth of a leaf in the recursion tree of quicksort is approximately $-\lg n/\lg 0.1$ and the maximum depth is approximately $-\lg n/\lg(0.9)$. Don't worry about integer round-off.

Proof. Then the running time of GoodQuickSort would be $T(n)$ where $T(n) = T(n/10) + T(9n/10) + \Theta(n)$. Using the substituion method, and the proofs earlier, we obtain $T(n) = \Theta(n \lg n)$. To summarize using $a = 0.1$ below, for the recursion tree based solution, the leftmost path of the recursion tree involves nodes with problem size $n, an, a^2n, a^3n, \ldots$ and the rightmost path $n, (1-a)n, (1-a)^2n, (1-a)^3n, \ldots$.
How long is the leftmost path? It is L long if $a^L n = 1$. Taking logarithms base two $L \lg a + \lg n = 0$, i.e. $L = -\lg n/\lg a$. Substitute 0.1 for a and we get the desired answer.
How long is the rightmost path? It is R long if $(1-a)^R n = 1$. Taking logarithms base two $R \lg(1-a) + \lg n = 0$, i.e. $R = -\lg n/\lg(1-a)$. Subsitute 0.1 for a or 0.9 for $1-a$ and we get the desired answer. If the depth of the recursion tree is $O(\lg n)$ and at every node the time is $\Theta(n)$, the result follows. ■

Exercise 260.
a. A new version of quicksort has been suggested by Professor I. M. CRAZY called CRAZYSORT in which an adversary picks the splitter in Partition in the worst possible way to maximize the running time of CRAZYSORT, that is, line 1 of Partition is determined by the adversary in constant time $O(1)$. What is the running time of CRAZYSORT? Justify your answer.
b. A new version of quicksort is proposed by Professor I. M. PARANOID called PARANOIDSORT in which an adversary picks the best, instead of the worst, possible splitter in line 1 of Partition *every other time* (i.e. he alternates between best and worst splitter). What is the running time of PARANOIDSORT? Justify your answer.

The bounds for the running time in parts (a) and (b) MUST be asymptotically tight.

Proof. Easy. ■

Exercise 261.

Why do we analyze the average-case performance of a randomized algorithm and not its worst-case performance? Explain.

Proof. There is no reason to analyze its worst-case performance. Randomized quick-sort for example has the same worst-case performance as ordinary quick-sort and randomization does not change this. If the choices of the random number generator lead to the worst-case, that's it, we get worst-case performance. The introduction of randomness, however, makes this (possibility of observing worst-case performance) **highly unlikely**. Much more interesting is an average-case analysis that analyzes the performance that "most likely" will be observed. ■

Exercise 262.

If we have some prior information about the ordering of the keys of the input, i.e. if they are "almost sorted" we could readjust quicksort to take this into consideration. When quicksort operates on a subsequence with fewer than k keys, let it just return without sorting the subsequence. As soon as the quicksort returns from its top-level call (the one with arguments 0 and $n-1$), invoke insertion sort on the entire "output" sequence. Argue that this two-stage sorting algorithm i.e. the modified quick-sort followed by insertion-sort runs in expected time $O(nk+n\lg(n/k))$. Find an m that minimizes this expected time.

Proof. We can change the high-level quicksort algorithm into the new one below as follows; execute in step 1 the modified quick sort that stops at k keys (called QUICKSORT2) and then run InsertionSort on the output of QUICKSORT2 (all keys). At recursion depth 1, the original input is split into two halves of expected size $n/2$. At recursion depth i, the input is split into 2^i sets each one having an expected $n/2^i$ keys. If $i=\lg(n/k)$ then we have n/k sets ($2^i=n/k$ for $i=\lg(n/k)$) with expected number of keys per set of $n/2^i=n/(n/k)=k$. The sequences of these keys are left unsorted and dealt with by Insertion Sort. What is the expected running time of step 1 of QUICKSORT? Each level of QUICKSORT takes $O(n)$ steps and there are $\lg(n/k)$ recursive calls for a total of $O(n\lg(n/k))$ expected running time for step 1 of QUICKSORT. InsertionSort takes over and sorts the keys of the n/k sets. Let us call the subsets of A $s_1,s_2,\ldots,s_{n/k}$. Because of the way quicksort works, all keys in s_1 are less than all keys in s_2 and all keys in s_2 are less than all keys in s_3 and so on. So every key is at most k positions away from its destination within its own set. Therefore insertion sort will spend time $O(k)$ per key (a key in set s_i can only move at most k positions in that set since the set is of size k), and since there are $O(n)$ keys the total time of Insertion sort (step 2 of QUICKSORT) will only be $O(nk)$ not $O(n^2)$. Combining the contributions of steps 1 and 2 of QUICKSORT we get that the expected running time is $O(nk+n\lg(n/k))$. Due to the first term we want k to be up to $\lg n$ so that the first term becomes $O(n\lg n)$. The second term is for this range of k also $O(n\lg n)$. Values of $k=\omega(\lg n)$ make the expected running time of QUICKSORT (modified version) $\omega(n\lg n)$; these values are disregarded. In practice the best value of k is chosen by trial and error trying values from $k=1$ all the way to $\lg n$ or some $A\lg n$ for some constant $A>1$, and the best one is used after experimentation.

```
 QUICKSORT(A,n)
1. QUICKSORT2(A,0,n-1);
2. InsertionSort(A,n);

QUICKSORT2 (A,l,r)
1.  if (r-l+1) >= k {      //A[l..r] has r-l+1 keys;do nothing if r-l+1 < k.
2.    m = PARTITION(A,l,r) //        otherwise sort as before.
3.    QUICKSORT2(A,l,m-1)
4.    QUICKSORT2(A,m+1,r)
    }
```

■

Exercise 263.

What is the running time of NutSort?
What is the average case running time of NutSort?

```
  Nutsort(A,n,k) // A[0..n-1]
1. NUTSORT(A,0,n-1,k);
2. InsertionSort(A,n);


  NUTSORT(A,l,r,k)
1. if (l<r) {
2    m = Partition(A,l,r);
3.   if (m-l >= k ) NUTSORT(A,l,m-1,k);
4.   if (r-m >= k ) NUTSORT(A,m+1,r,k);
```

Proof.
Lines 3 and 4 of NUTSORT are executed if problem size is k or more, and in line 2 of top-level Nutsort, InsertionSort is performed as instructed by I.M.Nuts. In the worst-case we can still have an uneven $(0,n-1)$ split, i.e. the maximum or the minimum is picked at each recursive call. Such a split can go on for $\Theta(n)$ recursive calls and gets resolved when the sequence becomes of size (at most) k. Then after one additional $(0,k-1)$ split we stop. Up to that point $O(n-k)$ recursive calls have been completed giving a running time of $\Theta(n(n-k)) = \Theta(n^2)$. Thus the worst-case of NutSort is as bad as that of QuickSort, a quadratic algorithm.
On the average we expect an approximately even split thus after i steps we expect 2^i subsequences each one having approximately $n/2^i$ keys or after $i = \lg(n/k) = \lg n - \lg k$ iterations we expect n/k subsequences each one having k keys (expected size). The $i = \lg(n/k)$ recursive calls cost altogether $O(i \cdot n) = O(n \lg(n/k))$. This is the expected time of line 1 of Nutsort. When recursion thus stops after an expected $\lg(n/k)$ iterations the subsequences are slightly less than about k and are left unsorted. Then InsertionSort of Nutsort takes over. We analyze now line 2 of Nutsort. The final Insertion sort only rearranges keys within each subsequence. Keys will never move outside a subsequence since because of the way quicksort works all keys in the j subsequence are less than or equal to the keys in subsequences $j+1,\ldots$ and greater than or equal to keys in subsequences $1,\ldots,j-1$. Thus InsertionSort will take time $O(k^2)$ for a given subsequence of expected size k and since we have n/k (expected) subsequences the overall InsertionSort time will be $O(k^2 \cdot n/k) = O(nk)$. Collectively we get as running time the sum of the two contributions of lines 1 and 2 of Nutsort i.e. $O(nk + n\lg(n/k))$. If we pick k around $\lg n$ i.e. $k = \Theta(\lg n)$, the expected running time of this approach is $O(n \lg n)$. ■

Exercise 264.

LunaticSort. A new version of quicksort is proposed by Professor I. M. NJIT, called LunaticSort in which an adversary picks the best, instead of the worst, possible splitter in line 1 of Partition *every 5-th time*, that is four times in a row a worst possible splitter is picked but then the fifth time the best possible splitter is picked. What is the running time of LunaticSort? Justify your answer.

Proof. LunaticSort? Didn't we encounter CrazySort or ParanoidSort before? ■

Exercise 265.

```
       NutSort  (A,i,j)
 1. if A[i] > A[j]
 2.    then exchange A[i] and A[j]
 3. if (i+1) >= j
 4.    then return;
 5. k= floor( (j-i+1)/3 );     // Round down
 6. NutSort( A, i , j-k );     // First two-thirds
 7. NutSort( A,i+k, j   );     // Last  two-thirds
 8. NutSort( A, i , j-k );     // First two-thirds AGAIN!
```

```
        NutSort (A,i,j)
1.  if A[i] > A[j]
2.     swap(A[i], A[j]);
3.  if ((i+1) >= j)
4.     then return;
5.  k= floor( (j-i+1)/3 ); // Round down
6.  NutSort( A, i , j-k );  // First two-thirds
7.  NutSort( A,i+k, j );    // Last two-thirds
8.  NutSort( A, i , j-k );  // First two-thirds AGAIN!
```

Professor I.M. NUTS suggests that `NutSort` sorts n keys and is the "best of the best".
(a) Argue that $NutSort(A,1,n)$ correctly sorts A.
(b) Give a recurrence and a tight asymptotic bound for the running time of NutSort.

Proof.
(a) Lines 1-4 is the base case of an inductive proof. In line 1 if two keys are out of order they are sorted (i.e. smallest on the left side). Let us deal with the inductive step below to show that `NutSort` sorts. The inductive hypothesis we are going to use is: If `NutSort` sorts any array of less than n keys, then it sorts an array of n keys. Let us mentally split A into three areas A1, A2, A3 the first, second, and third one-third of A each accommodating $n/3$ keys (Forget for the moment cases where n is not a multiple of 3; the floor deals with it). At the end of the algorithm the n keys will occupy these areas. The $n/3$ smallest will occupy area $A1$, the $n/3$ largest keys area $A3$ and so on. The largest, after the sorting, will be $A[n]$, and the smallest $A[1]$. Let us call the $n/3$ smallest keys of A collectively $X1$, the largest $n/3$ keys $X3$ and the remaining $X2$. Originally the keys are arbitrarily distributed in A. After the sorting the $X1$ keys are in area $A1$, the $X2$ in $A2$ and the $X3$ in $A3$ and sorted. The induction (inductive hypothesis) is used below for sorting sizes of about $2n/3$ i.e. less than n.

Step 1: Line 6 call. Consider the first call to `NutSort` in line 6. It deals with keys in areas $A1$ and $A2$. These areas can have $X1, X3$ keys. After the "sorting" $X1$ keys will move left and $X3$ keys will move right in those areas. **That is $X1$ keys will move to $A1$ and $X3$ keys will move to $A2$. There will be no $X1$ keys in $A2$ since there is a total of $n/3$ $X1$ keys and area $A1$ can accommodate all of them. Similarly there will be no $X3$ keys in $A1$.**

Step 2: Line 7 call. Consider the second call to `NutSort` in line 7. It deals with keys in aread $A2$ and $A3$. These areas can have $X3$ keys dispersed in both. Area $A2$ cannot have $X1$ keys because of the previous call of line 6 (Step 1); area $A3$ however can have $X1$ keys that were not moved by the call of line 6 that did not deal with them. After the "sorting" $X1$ keys will move left to area $A2$ and ALL $X3$ keys from $A2$ or $A3$ will move further to the right to area $A3$ and become sorted. That is area $A3$ will only have $X3$ keys in sorted order. The "sorted" claims here are by the inductive hypothesis as already noted before Step 1.

Step 3: Line 8 call. Consider the third call to `NutSort` in line 8. It deals with keys in aread $A1$ and $A2$ again (similarly to Step 1). These areas can NOT have $X3$ keys any more since these are in $A3$. So by the inductive hypothesis the $X1$ and $X2$ keys are sorted and the former move to $A1$ and the latter to $A2$. This completes the inductive step. The steps are also depicted below. For the algorithm to work we assume that n is a power of 3 i.e. the three areas are of the same size. This condition however can be dropped, as long as area A2 is wide

enough to accept the X3 keys of step 1, and the X1 keys of step 2. That's why we use floor in line 5 and not ceiling instead.

```
Beginning.
                              Area  A1         Area  A2         Area  A3
                           [ X1 , X2, X3  |  X1 , X2 , X3  |  X1 , X2 , X3 ]


Step 1. After Line 6       <---------sort this area------->
                              Area  A1         Area  A2         Area  A3
At the end no X1 in A2     [ X1 , X2      |        X2 , X3 |  X1 , X2 , X3 ]


Step 2. After  Line 7                      <-------------sort this area---->
At the end no X3 in A2        Area  A1           Area  A2           Area  A3
At the end no X1 in A3     [ X1 , X2       |  X1,  X2      |            X3 ]


Step 3. After  Line 8       <--------------sort here --------->
At the end X1 in A1           Area  A1           Area  A2           Area  A3
At the end X2 in A2        [ X1            |        X2     |            X3 ]
At the end X3 in A3
```

(b) The three calls of lines 6,7,8 involve approximately $2n/3$ keys. Lines 1-5 take $\Theta(1)$ time. Therefore, $T(n) = 3T(2n/3) + \Theta(1)$. Then apply master method case 1 with $a = 3$ and $b = 3/2$. We have $T(n) = \Theta(n^{\log_{3/2} 3}) = \Theta(n^{2.7})$, or about so if one uses a calculator to find the exact value of $\log_{3/2} 3$ more accurately than the indicated 2.7. That is `NutSort` is slower than any other sorting algorithm that we have examined (except for ExhaustiveSort). ■

Exercise 266.

Does NewQuickSort sort? What is its performance ? Explain. What is the depth of the recursion (recursion tree)?

NewQuickSort.

```
NewQuickSort(A,p,r)
1. while (p<r) {
2.     //partition and sort smallest array first NOT necessarily left
3.     q = Partition(A,p,r) ;
4.     if (q-p+1) < (r-q)
5.           NewQuickSort(A,p,q) ;
6.           p=q+1 ;
7.        else
8.           NewQuickSort(A,q+1,r) ;
9.           r=q ;
   }
```

Proof.

(a) NewQuicksort has the same effect as Quicksort, as line 6 that sets $p = q+1$ forces in the following iteration a call to NewQuicksort $(A, p = (q+1), r)$, just as Quicksort does. (b) The stack depth of Quicksort becomes $\Theta(n)$ if we have so many nested calls in line 5. An input that forces such a case is one that looks like this $\langle n, 1, 2, \ldots, n-1\rangle$, where the largest key is first and the remaining keys form a sorted sequence. Partition will return two subsequences $\langle n-1, 1, 2, \ldots, n-2\rangle$ and $\langle n\rangle$.

(c) The problem with part (b) is that the first call in line 5 is always performed on the largest of the two partitions. The solution is to call first NewQuicksort on the smallest of the two partitions and iterate on the largest. This way, in each recursive call the size of the sequence NewQuicksort operates on is halved. Therefore the maximum recursion depth will be $O(\lg n)$. The new function is shown above. Note that this function generates the same partitions, performs the same comparisons as before and sorts the same sequences.

■

Exercise 267.

You are given n non-zero integer keys (positive or negative but not zero). Give an efficient algorithm with worst-case running time $o(n \lg n)$ that works in-place and rearranges the input keys in such a way that the negative keys are on the left of the positive keys. For example, if the input was $A = [1, \quad -2, \quad 3, \quad 5, \quad -8]$, then the output could look like $A = [-2, \quad -8, \quad 3, \quad 5, \quad 1]$.

Proof.
Modify the Partition used in QuickSort by making the splitter to be equal to 0. Then Partition will move to the left the negative values and to the right the positive values. The minimal changes of Partition are shown in MyPartition below

```
 MyPartition(A,l,r); // Sort A[l ... r]
1. splitter= 0;
2. i=l-1;
3. for j=l to r  do {
4.   if A[j] <=splitter {
5.     then  i=i+1;
6.           exchange A[j] and A[i];
     }
   }

   /or a C-like code
      MyPartition(A[1..n],n) // No argument just a single call
 1.  splitter=0;
 2.  i=0;
 3.  for(j=1;j<=n; j++) {
 4.    if A[j] <= splitter // = is NEVER the CASE and CAN THUS BE REMOVED
 5.      then  i=i+1;
 6.            swap A[j] and A[i];
     }
 7.  return(i+1);  // first key > 0 if such exists */


1. //Calling the first version
2. MyPartition(A,1,n) // If A[1..n]
```

What if we have 0, 1, 2 key values and wish to split them? Assume the input contains only 0,1, and 2. Give a partition that 3-way splits the input.

```
   3-way-Partition(A, n) {
        l = 0;
        m = 0;
        r = n;
        splitter = 1;

        while (m <= r) {
          if (A[m] < splitter){       // A[m] == 0
            swap A[l], A[m] ;
            l++; m++;
          }
          else if (A[m] > splitter){  // A[m] == 2
                  swap A[m], A[r];
                  r--;
               }
               else{
                  m++;
                }
        }
}
```

■

Exercise 268.

Which algorithm between merge-sort and radix-sort/count-sort is asymptotically faster for each one of the following sorting problems. Justify your answer.
(a) Sorting of n keys, where each key takes values in the range $0 \ldots n^2 - 1$.
(b) Sorting of n keys, where each key takes values in the range $0 \ldots n^{2\lg n} - 1$.
(c) Sorting of n keys, where each key takes values in the range $0 \ldots n^n - 1$.

Proof.
Non-negative integer x (and also, all intermediate integers $0, \ldots, x-1$) can be represented by $\lfloor \lg x \rfloor + 1 = \lceil \lg(x+1) \rceil$ bits. On the other hand, y bit can represent all positive integers $0, \ldots, 2^y - 1$, where $y > 0$. Let us assume that n is a power of two; this assumption can be removed as we use asymptotic notation to describe algorithm performance. Then number $n-1$ requires $\lg n$ bits for its representation Number $n^2 - 1$ of part (a) requires $2\lg n$ bits, $n^{2\lg n} - 1$ of part (b) requires for its representation $2\lg^2 n$ bits, and $n^n - 1$ of part (c) requires $n \lg n$ bits. A $2\lg n$-bit integer can be viewed as the concatenation of two $\lg n$-bit sequences, i.e. as a two-digit radix-$2^{\lg n}$ number. Similarly, a $2\lg^2 n$-bit integer can be considered to be a $2\lg n$-digit radix-$2^{\lg n}$ number, and an $n \lg n$-bit integer can be considered to be an n-digit radix-$2^{\lg n}$ bit number. We have thus expressed all three integers into a common $2^{\lg n} = n$ radix. The key to the solution is that n radix-k numbers can be sorted using CountSort in $O(n+k)$ time. If $k = 2^{\lg n}$, then $O(n+k) = O(n)$. Then, n l-digit radix-k numbers can be sorted using Radix Sort in $O(l(n+k))$ time by using l times (once for each digit) the CountSort algorithm.
(a) (a) If we sort the n keys using Radix-Sort (i.e. perform two rounds of Count-Sort) then for $l = 2$, $k = n$, the running time is $O(2(n+k)) = O(n)$, i.e. CountSort/Radix-Sort(Count-Sort) beats mergesort whose running time is $O(n \lg n)$.
(b) If we sort the n keys using Radix-Sort (i.e. $2\lg n$ rounds of CountSort) for $l = 2\lg n$, $k = n$ the running time is $O(l(k+n)) = O(\lg n(n+n)) = O(n \lg n)$. Therefore RadixSort and MergeSort have the same asymptotic performance.
(c) If we sort the n keys using Radix-Sort (i.e. n rounds of CountSort) for $l = n$, $k = n$ running time is $O(l(k+n)) = O(n(n+n)) = O(n^2)$. Therefore MergeSort is asymptotically faster than Radix-Sort. ■

Exercise 269.

A search engine retrieves the following three doclists for terms A, B, C. The doclist of A for example contains all the docID (document IDs) that contain term A. The doclists for A, B and C are given in the form of arrays also named A, B, and C. Each of the three arrays is sorted based on docID. (A docID is a positive integer number.) Let the lengths of A, B and C be $\lg n$, $\sqrt{n}$ and n respectively.
The output for the problems listed below should be in the same form: in an array that however does not need to be sorted.
(i) We want to find all docs that contain all three terms A, B and C.
(ii) We want to find all docs that contain exactly two of A, B, and C.
(iii) We want to find all unique docs that contain A or B or C (i.e. no duplicated docIDs in the output).
Show how to solve each one of those problems (time) efficiently. Analyze their worst-case running time. Make them as space efficient as possible. Justify your answers.
Hashing is not a solution.

Proof.
(i) We want to find a docID that is in A and in B and in C i.e. in the intersection $A \cap B \cap C$. The intersection is of size no more than the smaller set i.e. no more than that of A, which is $\lg n$. Thus perform a binary search operation as follows: for each $a \in A$ binary search a into B and C in time $O(\lg \sqrt{n}) = O(\lg n)$ and $O(\lg n)$ respectively. (Both B and C are sorted.) If a belongs to both B and C it is part of the answer, call it D and collect a in D. Total time is $O(\lg^2 n)$ since the number of elements a is just $\lg n$ and binary search on B and C takes a combined $O(\lg n)$. Space-wise binary search uses only $O(1)$ additional space; this is in addition to the output array whose size is no more than $\lg n$. Note: we can run a dry run that only counts the elements $a \in A$ that are in B AND C. This way we can allocate for D the exact number of elements of $A \cap B \cap C$.
(ii) The output E cannot more elements (docIDs) than the two smaller of the three doclists and thus its length would be no more that $\lg n + \sqrt{n}$. The cost of a binary search into A is $O(\lg \lg n) = o(\lg n)$, into B is $O(\lg \sqrt{n}) = O(\lg n)$, and into C is $O(\lg n) = O(\lg n)$, i.e. all are $O(\lg n)$. (a) Do a BinarySearch of all elements of A into B and then into C. If there are two hits for a given a of A then this can not be in E, if a belongs either to B or C (but not to both!) we are done, a is in E. Do the same for elements of B. The only tricky part is not to add twice in E elements that are in A and B but not in C; such elements were added during the binary search of A into B,C. One needs to make sure that they are added once. The contribution for the A's binary search operations into B or C is thus $\lg n \cdot O(\lg n)$. This is $O(\lg^2 n)$. The contribution for the B's binary search operations into A or C is thus $\sqrt{n} \cdot O(\lg n)$. This is $O(\sqrt{n} \lg n)$. Adding both up we get a total running time of $O(\sqrt{n} \lg n)$. Space-wise we use only $O(1)$ additional space for the counter `number`, indexes `i,j` and flags `foundB, foundA, foundC` in the outline below.

```
  Find(A,B,C,lgn,sqrt(n),n)
0. j=0;
1. for(i=1;i<size(A);i++) {
2.   number=0;
3.   foundB = BinSearch(A[i],B,sqrt(n));
4.   if (foundB ==1) number++;
5.   foundC = BinSearch(A[i],C,n);
6.   if (foundC ==1) number++;
7.   if (number == 1) {
8.      E[j]= A[i]; j++;
9.   }
10.}
11.for(i=1;i<size(B);i++) {
12.  number=0;
13.  foundA = BinSearch(B[i],A,log(n));
14.  if (foundA ==1) number++;
15.  foundC = BinSearch(B[i],C,n);
16.  if (foundC ==1) number++;
17.  if (number == 1) && (foundA !=1) {
18.     E[j]= B[i]; j++;
19.  }
20.}
```

(iii) If we merge the three lists the same docID appear in consecutive order. However merging requires $\Omega(n)$ time and extra space of $\Omega(n)$ to accommodate the longest of the three lists. This is not (space) efficient. In space efficiency consideration we do not know the space for the output F. Consider the following approach. **From-A-to-BC**: We perform a binary search of each a in A into B and C. If a is neither in B nor in C it is reported in the output F. This requires overall time $O(\lg^2 n)$. Otherwise the x is not reported in the output. **From-B-to-C**: We then perform a binary search of b in B into C. This requires $O(\sqrt{n}\lg n)$ time. If a b is not in C, then it is reported in F, otherwise it is not reported at all. **From-C**: We dump all elements of C into F. This takes n time. Total times is $O(\lg^2 n+\sqrt{n}\lg n+n)$ i.e. $O(n)$ or $n+O(\sqrt{n}\lg n)$. Space is the space of F plus constant more.
Why does it work? Let's rank A, B and C with 0,1,2 and these are also the ranks of their elements. C has the highest and A the lowest rank. An element e of the output has rank the highest rank of the set that contains it. If e has rank 2 it will be reported into F by From-C; neither From-A-to-BC, nor From-B-to-C will report it to F. IF e has rank 1 it will be reported into F from From-B-to-C; From-A-to-BC will not report it to F. If e has rank 0 it will be report into F from FromA-to-BC.

■

Exercise 270.

t-way Merging.

Show (give algorithm, explain it and analyze its running time) how to merge t sorted sequences each one of size n/t into a sorted sequence of size n in time proportional to $n\lg t$. Ignore constants. Assume n is a power of t if needed or if it is convenient.

Proof.

We provide a different solution that the one given previously.

We give first one solution that utilizes heaps. If we had two sequences as it was the case in MergeSort by keeping two pointers to the current smallest elements of the left and right sequences was enough to insert the smallest of the two sequences into the output. In this problem however we have t sequences not just 2. We maintain the current smallest element of each one of the t sequences in a MIN-Heap. Initially we pick the smallest elements, one per sequence, for a total of t and we BuildHeap them in $O(t)$ time. The winner is the smallest that is ExtractMin removed from the Heap and inserted into the output sequence and the heap now gets $t-1$ elements. We fill the heap to t elements by HeapInserting one more element to the heap; the t-th element comes from the sequence of the ExtractMin element if there is such an element. Otherwise (if the sequence of that element is empty) no element is inserted and the heap size decreases.

Step 1. Let $j=1$ be the elements of the output sorted sequence. Let the head (smallest element) of sorted sequence i be denoted by H_i, $i=1,\ldots,t$. Take the current t heads H_i, remove each one from corresponding sequence, build a heap (BuildMinHeap) based on the key values of the t heads in $O(t)$ time, and then update the H_i to point to the second smallest elements of the sorted sequences (as the smallest elements are in the heap but not in the sequence any more).

Step 2. ExtractMin the smallest element M of the heap in $O(\lg t)$ and place it in the j-th position of the output sequence. Increment j by 1. If M came off the l-th sorted sequence (i.e. it was before step 1, $M=H_l$) then remove the current head of the l-th sequence and insert it into the $t-1$ element heap in $O(\lg t)$, and update H_l appropriately.

Step 3. Repeat Step 2 until $j=n$.

Step 1 takes $O(t)$ time. In step 2, every time M is removed, it is smaller than any of the remaining heap or sorted sequence elements. Time for step 2 is $O(\lg t)$ for ExtractMin M or for HeapInsert of the next smallest element of M's sequence. As there are a maximum of n elements that are HeapInsert-ed or ExtracMin-ed in/from the heap the total running is $O(n\lg t)$.

■

Exercise 271.

A generalized merge-sort algorithm, instead of splitting a sequence of length n into two subsequences of size $n/2$ each, it splits the input sequence into k equal size subsequences (k may not be a constant). In the combining phase, a merging of k sorted sequences needs to be performed. Suppose the recurrence for the running time of this modified merge-sort algorithm is as follows.

$$T(n) = kT(n/k) + n\lg k,$$

with $T(1) = 1$. For simplicity you may also assume that n is a power of k.
Is there a value of k that minimizes $T(n)$? Justify your answer.

The explanation of the additive $n\lg k$ term is the previous problem for $t = k$.

Proof.
Note that since k may not be a constant **the master method cannot be used** as $a = b = k$ may not be constant. We solve the recurrence with the iteration method.

$$\begin{aligned} T(n) &= kT\left(\frac{n}{k}\right) + n\lg k \\ &= k\left(kT\left(\frac{n}{k^2}\right) + \frac{n}{k}\lg k\right) + n\lg k \\ &= k^2T\left(\frac{n}{k^2}\right) + n\lg k + n\lg k \\ &= k^2T\left(\frac{n}{k^2}\right) + 2n\lg k \\ &= \dots \\ &= k^iT\left(\frac{n}{k^i}\right) + in\lg k \end{aligned}$$

This is similar to the merge-sort recurrence. The only difference is that the 2 has become a k. The base case is $T(1) = 1$. For the base case to occur $\frac{n}{k^i} = 1$ i.e. $n = k^i$, i.e. $i = \log_k n$ or $i = \lg n/\lg k$. For that value of i we have $n = k^i$ (the first term in the expression above), $T\left(\frac{n}{k^i}\right) = T(1) = 1$ (the second term multiplied by the first) and $in\lg k = (\lg n/\lg k)n\lg k = n\lg n$ (the third term). Note also, that $k^{1/\lg k} = 2$ below, and there are i terms $n\lg k$ below (equal to the exponent of k^i). Therefore,

$$\begin{aligned} T(n) &= k^iT\left(\frac{n}{k^i}\right) + in\lg k \\ &= k^{\lg n/\lg k}T(1) + \left(\frac{\lg n}{\lg k}\right)n\lg k \\ &= 2^{\lg n}T(1) + \left(\frac{\lg n}{\lg k}\right)n\lg k \\ &= n + n\lg n \end{aligned}$$

Therefore, the answer to the problem is : NO, as $T(n)$ is independent of k, there is no value of k that minimizes $T(n)$.

■

Exercise 272.

The input is two sequences of $A[1..n]$ and $B[1..n]$ of n integers each and an additional integer number x. (The integers in A and B exhibit no particular ordering.) Give a time and space efficient algorithm that runs in $o(n^2)$ time that determines whether an integer a in A and an integer b in B exist such that $a+b=x$. If such a pair (a,b) exists the first such pair is output and the algorithm terminates. Otherwise a NO gets printed to indicate that no such pair exists. Be succinct, to the point and justify your arguments.

Proof.

We first sort one sequences say B in worst case running time $O(n\lg n)$. We then perform a binary search for each element a of A by trying to locate element $x-a$ in B. If for $a=A[i]$ we find a key in B such that $b=x-A[i]$, then $a+b=A[i]+(x-A[i])=x$ as needed. The binary search of step 5 takes $O(\lg n)$ steps for every key $A[i]$ of A for a total of $O(n\lg n)$ as well, as there are n keys in A. Total time is therefore $O(n\lg n)$ in the worst case. (HeapSort is space efficient and time efficient.)

```
1. Find(A[1..n],B[1..n],x)
2.   HeapSort(B,n);
3.   for(i=1;i<=n,i++) {
4.    key = x- A[i];
5.    found = BinSearch(key,B[1..n]);
6.    if found == -1 continue;
7.     else return ( A[i], B[found]);
8.   }
9.   return (NO);
```

■

Exercise 273.

For this problem you may assume that $\lg n$ is an integer and furthermore n is a multiple of $\lg n$: in plain English, ignore floors or ceilings. You are given two arrays A and B of m and $n/\lg n$ keys respectively, where $m < n/\lg n$. The elements of A and B are numbers positive, negative or zero and exhibit no particular order (i.e. they are 'random' or 'arbitrary'). We want to determine whether there exists an a of A and there exists a b of B such that $a+b=610$. Give an $O(n)$ time algorithm that finds such a pair (a,b). One pair (a,b) suffices, if there are multiple answers. If no such pair exists, your algorithm should output a NO.

Proof.
Description and Proof of correctness. Sort one of the arrays. It does not matter which one asymptotically. Say we sort B in O(n/lgn *lgn)=O(n) time using heapsort. We then do a binary search of every $A[i]-610$ into B. If we find a $B[j]$ such that $610-A[i]=B[j]$ the $A[i]+B[j]=610$, and thus $a=A[i]$ and $b=B[j]$ are as needed and those are returned by line 5. If no such pair can be found a NO is returned through line 6.

```
 Find(A,B,m,nlgn)  // A,B are [0...] arrays
1. HeapSort(B,n/lgn)
2. for(i=0;i<m;i++) {
3.   if (j=BinSearch(A[i]-610,B,n/lgn)) != -1
4.     return((a=A[i],b=B[j]));
5. }
6. return(NO);
```

Line 1 requires $\Theta((n/\lg n)*\lg(n/\lg n)=\Theta(n)$ i.e $O(n)$. Line2-5: require $O(m\lg n)=O(n)$ since $m<n/\lg n$. Line 6 is $\Theta(1)$. Therefore $T(n,m)=\Theta(n)$. ■

Exercise 274.

You are given three arrays A[0..n-1], B[0..n-1], C[0..n-1] each one containing n integers in the range $-n\lg n..0..n\lg n$. We want to determine whethere there are three keys keys a,b,c respectively from A, B, C so that a+b+c = 0. Give a $O(n^2)$ algorithm to solve this problem.

Proof.

```
    Find(A,B,C,n)            // Running time is O(n^2 )
    allocate D = n^2 keys;   //O(n^2)
1.a k=0;
1.b for(i=0; i<n ; i++ )    //O(n^2)
1.c   for(j=0 ; j<n ; j++)
1.d.    D.value [ k ] = A[i] + B[j];
1.e.    D.info  [ k ] = {i,j};

1.e Adjust(D.value+2*n*lgn)      //values[-2nlgn,2nlgn]->[0..4nlgn]
1.f RadixSort(D, 2 rounds);  // O(n^2)
1.g ReAdjust(D.value-2nlgn);   // values[0,2nlgn]->[-2nlgn.2nlgn]


1.i for(m=1; m< n ; m++) {   // O(n \lg{n})
1.j    c = C[m]      // a+b+c=0 <=> (a+b)=-c <=> -c= D[k];
1.k    if  ( k = BinSearch(-c,D)) !=-1
1.l        return  (A[ D.info[k].i ] , B[D.info[k].j], C[m]);
1.m  }
1.n return(NOT-FOUND)
```

■

Exercise 275.

You may assume that $\lg n$ and $\sqrt{n}$ are both integer in this problem. You are given two sorted sequences stored in arrays A and B of $\lg n$ and $\sqrt{n}$ keys respectively. We are interested in forming the union $C = A \cup B$, i.e. an array C containing all the keys of A and B; common keys however, should appear only once in C. Give a space and time efficient algorithm for solving this problem. (Time efficient means its running time cannot be asymptotically improved upon; space efficient means likewise for space requirements.) Justify your arguments.

Proof.
Since we are interested in the union, the size of C can be at least $\sqrt{n}$ and at most $\lg n + \sqrt{n}$ and both are $\Theta(\sqrt{n})$. C includes all keys of B and those keys of A that are NOT in B. We resolve the latter by determining which A keys are in B by performing a binary search of A into B, and those keys are not included into C. Each binary search of an A key takes $O(\lg \sqrt{n}) = O(\lg n)$ and collectively the $\lg n$ binary search operations (lines 3-8) require $O(\lg^2 n)$ time. Yet the running time of the algorithm is $O(\sqrt{n})$ dominated by lines 9-10 (in fact it is $\Theta(\sqrt{n})$). Space-wise, we use $\lg n + \sqrt{n}$ memory for C and $\Theta(1)$ for other variables. We allocate at most $\lg n + \sqrt{n}$ space in line 1, since the actual space is at least $\sqrt{n}$, and refining to the exact value wont't improve space efficiency asymptotically, since $\sqrt{n}$ and $\lg n + \sqrt{n}$ are asymptotically the same.

```
 Union(A,B,lgn,sqrt(n),n)
0  c= sqrt(n)+lgn;
1. C = allocate space [1... c];
2. c = 0;
3. for (i=1 ; i<=lgn ; i++) {
4.   result = BinSearch(A[i], B); // if A[i] not in B add it to C
5.   if result == -1 {
6.     c++; C[c] = A[i];
7.   }
8. }
9. for (i=1 ; i<= sqrt(n); i++) {
10.  c++; C[c] = B[i];    // Add all keys of B to the union
11.}
12. return(C);
```

■

Exercise 276.

You may assume that $\lg n$ is integer in this problem. You are given two sorted sequences stored in arrays A and B of $n\lg n$ and n^2 keys respectively. We are interested in forming the intersection $C = A \cap B$, i.e. an array C containing the common keys of A and B; common keys appear once in C. Give a space and time efficient algorithm for solving this problem. (Time efficient means its running time cannot be asymptotically improved upon; space efficient means likewise for space requirements.) Justify your arguments.

Proof.

Note that the two sets have sizes $n\lg n$ and n^2 respectively. The intersection can be no larger than the smaller of the two sets, i.e. $n\lg n$. We thus perform a binary search of every key of A into B to find the common keys. This takes time $n\lg n \cdot O(\lg n^2)$, which is $O(n\lg^2 n)$. Space-wise, we allocate in line 1 too much space (all other space used is $\Theta(1)$ however). We could allocate the amount of space needed, if we find the NUMBER of keys of the intersection (rather than the intersection itself) first. The commented lines are in essence a replica of lines 2-7 and find the cardinality of C and can replace line 0 and their uncommenting turn the algorithm into space efficient.

```
 Intersection(A,B,nlgn,n**2,n)
// c=0;
// for (i=1 ; i<=nlgn ; i++)
//   if ( BinSearch(A[i], B) != -1)  c++;
// Delete the following line 0 then
0. c= nlgn;
1. C = allocate space [1...c];
2. c = 0;
3. for (i=1 ; i<=nlgn ; i++) {
4.   result = BinSearch(A[i], B);
5.   if result != -1 {
6.     c++; C[c] = A[i];
7.   }
8. }
9. return(C)
```

■

Exercise 277.
How can you turn a not-stable sorting algorithms into a stable one?

Proof.
Make sure all the keys are distinct! Suppose the input sequence $x_1, \ldots, x_n$ is to be sorted in non-decreasing order. As some x_i's may be equal we create for key x_i a new key which is the pair (x_i, i). The first element of the pair is x_i and the latter is the index/position of x_i in the input sequence. The input to the sorting algorithm is going to be not $x_1, \ldots, x_n$ but $(x_1, 1), \ldots, (x_n, n)$. Note that the keys of the newly formed input are always distinct; even if $x_i = x_j$ for some $i \neq j$, the second elements of the two pairs are distinct as $i \neq j$. As any of the sorting algorithms works fine with distinct keys (ie stability is no issue then), it will sort the n augmented keys. If key x_i was before x_j in the input sequence, it means $i < j$ and therefore $(x_i, i) < (x_j, j)$, and therefore in the output sequence (x_i, i) will appear before (x_j, j). When the output sequence is obtained, we reobtain the original keys from the pairs (by reading off the first element of the pair) to get the output sequence for the original problem.

■

Exercise 278.

You are given n keys of an array $A[1..n]$. The keys are unordered. We would like to determine whether there exists a key in the array that appears $n/2+1$ times or more. Give a space and time efficient algorithm whose worst case running time is $o(n^2)$ that solves this problem. The algorithm should find such a key if it exists or returns a "NO" if such a key does not exist.

Proof.
If a key is in "majority" i.e. it exists $n/2+1$ times or more then in the sorted output it would span the "middle" position $(n+1)/2$. We first sort the keys. (MergeSort is a no-go as it is not space efficient.) Then we isolate the "middle" key value and check in linear time (lines 3-4) whether it exists $n/2+1$ times or more. If it is so it is output, else a NO is returned. Total time is dominated by HeapSort i.e. it is $O(n \lg n)$.

```
 Majority(A[1..n],n)
1. HeapSort(A,n);  // ''space-efficient'' excludes MergeSort
2. c=0; majority = A[(n+1)/2]; // candidate for majority;
3. for(i=1;i<=n;i++)
4.  if A[i] == majority  c++;
5. if (c >= n/2+1)  return(majority);
6.   else           return(NO);
```

■

Is it time efficient? NO. Replace line 1 (sorting) with an $O(n)$ solution.

Exercise 279.

You are given n arbitrary integers in array $A[0..n-1]$, where every $A[i]$ is in the range 0 to $m-1$, i.e. $A[i] \in [0, m-1]$. $[X,Y]$ denotes all integers between X and Y inclusive of X and Y; moreoever, $Y \geq X$; if $Y < X$ the interval is empty.

We would like to set up a data structure H so that we can answer efficiently in the worst-case queries of the form:
"For given integer inputs X,Y, how many of the n integers of $A[0..n-1]$ are in range $[X,Y]$?" We thus need to implement an efficient function `HowMany(H,X,Y)`. In order to do that we need first to transform A into an efficient data structure H by also implementing `Process(A,H,n,m)` that does so, and then using the constructed H to realize `HowMany(H,X,Y)`.
(a) Show how to implement `Process(A,H,n,m)` that transforms A into H in no more than $\Theta(n+m)$ running time;
(b) so that in the resulting structure H, queries of the form `HowMany(H,X,Y)` can be answered in $O(1)$ running time.
(c) How much space do you need for H? Express your answer in tight asymptotic
notation. Show that indeed the running time of the two functions is $\Theta(n+m)$ and $O(1)$ respectively.

Proof.

```
Process (A,H,n,m)
1. for i=0 i<m i++ H[i]=0;
2. for j=0; j<n j++ H[A[i]]++
3. for i=1 i<m i++ H[i]=H[i]+H[i-1];

Howmany(H,X,Y)
1. if x=0 return H[Y]
2.  else return(H[Y]-Y[X-1])
```

■

Exercise 280.

FancySort is proposed as an alternative to QuickSort. In order to sort an array A of n distinct and comparable keys, FancySort first selects a sample of $2\sqrt{n}+1$ arbitrarily picked keys out of the keys of the array, and sorts those sampled keys with InsertionSort. Then, it picks the median M of the sorted sample (the $\sqrt{n}+1$-st smallest key of the sample) and uses it as a splitter (i.e. pivot key) to partition A into two split subsequences, that with keys less than or equal to M, and that with keys gre ater than M. FancySort then recursively sorts the two split subsequences.

a. Let $T(n)$ be the worst-case running time of FancySort on an array A of n distinct comparable keys. Show that the recurrence for the worst-case running time $T(n)$ of FancySort is $T(n) = T(n-\sqrt{n}) + T(\sqrt{n}) + \Theta(n)$.

b. Let $h(n)$ be the depth of the recursion of FancySort for the worst case. Show that $h(n) = h(n-\sqrt{n}) + 1$.

for $h(n)$? Pr ovide as tight a bound as possible.

d. After you complete (c), solve (a) for $T(n)$ using part (b) and pa rt (c). Give the worst-case running time of FancySort in as tight asymptotic notation as possible.

Proof.

a. In the worst-case the sample contains the $2\sqrt{n}+1$ smallest or the $2\sqrt{n}+1$ largest keys of A. The choice of M would then lead in the worst-case to one side containing sqrt(n) keys (smallest or largest) plus the splitter M, and the other side containing all of the remaining n-sqrt(n) keys. In the worst-case the smaller side split leads into lg(lg(n)) iterations of FancySort. $n \to n^{1/2} \to n^{1/2^2} \ldots \to n^{1/2^i}$. For $n^{1/2^i} < 2 => 1/2^i \lg n < 1$ implies $\lg n < 2^i$ which implies $i > \lg(\lg n)$.

b. In the worst case a top-level call involving n keys, leads to a next level call, involving in the worst-case n-sqrt(n) keys. Let h(n) be the depth of the recursion for the call on A on n keys. Then after the first iteration we have $h(n) = h(n - sqrt(n)) + 1$, and the worst depth of the recursion is determined by the worst-case split. Note the amount of work performed to complete this one recursive call is linear to the number of keys involved. Sorting using InsertionSort $2\sqrt{n}+1$ keys takes $\Theta(n)$ time. Picking a splitter takes $\Theta(1)$ out of the sorted sample, and partitioning the input keys around the splitter M takes another $\Theta(n)$ time. Total running time is Theta of the number of keys involved in the recursive call. The recursion for the running time will then be $T(n) = T(\sqrt{n} + T(n-\sqrt{n}) + \Theta(n)$..

c. We shall show that $h(n) \leq 2\sqrt{n}$. Show by induction $h(n) \leq c\sqrt{n}$ for some $c > 0$ and constant. Proof (induction).

$$\begin{aligned} h(n) &= h(n-\sqrt{n}) + 1 \\ &\leq c\sqrt{n-\sqrt{n}} + 1 \\ &\leq c\sqrt{n-\sqrt{n}+1/4} + 1 \\ &\leq c\sqrt{\sqrt{n}-1/2)^2} + 1 \\ &\leq c\sqrt{n} - c/2 + 1 \\ &\leq c\sqrt{n} \end{aligned}$$

The last inequality is true if $1 - c/2 \leq 0$ i.e. $c \geq 2$.

d. From parts (a) and (b), amount of work per level for partition is $\Theta(n)$. Number of levels i.e. depth of recursion in the worst case by part (c) is $\Theta(\sqrt{n})$. Thus $T(n) = O(n^{1.5})$ for the running time, or $T(n) = \Theta(n^{1.5})$ for the worst-case running time.

■

Exercise 281.

An array A[0..n-1] contains n integer keys. Each key is either a 0, 1 or 2. Give an efficient algorithm that sorts the n keys.

Proof.
Use QuickSort partition type of a function that uses as splitter a 'key' 1. Note that GTPartition puts on the 'left' keys 0,1 and on the right keys 1,2 so a cleanup round is then required to move 0 on the left of the left side and 1 to the right of the left side, and likewise for 1,2. Or you develop a partition specific to this problem. Time is $\Theta(n)$. It is time and space efficient as well. See below.

```
z=-1; o=0; t=n; /* zero, one, two */
while( o <= t) {
  if A[o] == 0 {
     z++; swap(A[z],A[o]); o++;
  }
  if A[o] == 1 {
     o++;
  ]
  if A[o] == 2 {
     t--; swap(A[o],A[t]);
  }
```

■

Exercise 282.
TRUE or FALSE?

1. By reducing the problem of sorting n keys with comparisons and exchanges to the problem of building a heap on n keys, one can prove that the time it takes to build a heap on n keys is $\Omega(n \lg n)$ in the worst case.
2. The $\lg^2 n$-th smallest key of an unsorted sequence of n keys can be determined in $o(n \lg n)$ worst case time.
3. For any $k \leq n$, we can find the k smallest keys and have them output in sorted order in time $\Theta(n + k \lg k)$.

Proof. 1. FALSE! What the heck is this?
2. TRUE.
3. TRUE. ■

Exercise 283.

Array $A[0..n-1]$ contains n integers. We would like to determine whether A contains at least one pair of integers (x, x**2) i.e. an integer x and its square x**2. If one such pair exists it is output and the algorithm terminated. If no such pair exists, a NO gets printed. Give a time and space efficient algorithm that solves this problem. Analyze its worst-case running time and given an expression for the running time in tight asymptotic notation that describes it.

Proof.
Time is $O(n \lg n)$. Details omitted.

```
Find(A,n)
1. HeapSort(A,n)
2. for (i=0;i<n;i++)
3.  if BSe(A[i]*A[i],A,0,n-1)
4.    return(A[i],A[i]*A[i])
5. return("NO")
BSe(k,A,l,r)
1. if l==r  ret A[l]==k
2. m=l+r / 2  // l+(r-l)/2
3. if A[m]==k return 1
4. else if A[m]< k  ret Bse(k,A,m+1,r)
5. else             ret Bse(k,A,l,m-1)
```

■

Exercise 284.

Derive the expected running time of Randomized QuickSort.

Proof.
Let $A[1..n]$ be the output of Randomized Quicksort on n distinct input keys. Let X_{ij} be an indicator random variable such that

$$X_{ij} = \begin{cases} 1 & \text{A[i] and A[j] were compared,} \\ 0 & \text{were not compared.} \end{cases}$$

Then random variable X defined as follows counts the number of comparisons in a run of Randomized QuickSort.

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}.$$

The expected running time of Randomized Quicksort is proportional to the expected number of comparisons performed i.e. the expectation $E(X)$ of X.

$$E(X) = E(\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}). = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E(X_{ij}). \tag{5.1}$$

Note that

$$E(X_{ij}) = 1 \cdot P(X_{ij} = 1) + 0 \cdot P(X_{ij} = 0) = P(X_{ij} = 1). \tag{5.2}$$

From Eq.(5.1) by way of Eq.(5.2) we have

$$E(X) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} P(X_{ij} = 1). \tag{5.3}$$

Two keys of the output A_i and A_j are compared if one of them is chosen as the splitter. The splitter does not participate in recursive calls, and thus they will never be compared again. If the splitter is in between A_i and A_j say at index k so that $i < k < j$ then A_k (the splitter) will split A_i and A_j and thus the two keys A_i and A_j will never be compared later. If the splitter is less than A_i or greater than A_j then the keys end up on the same side, and another splitter will be picked later. Thus considering keys $A_i A_{i+1} \ldots A_k \ldots A_{j-1} A_j$, the probability that A_i and A_j are compared is the probability the splitter is two of the $j-i+1$ keys in between A_i and A_j that is, $P(X_{ij} = 1)$ is equal to $2/(j-i+1)$ i.e.

$$P(X_{ij} = 1) = \frac{2}{j-i+1}. \tag{5.4}$$

Equation Eq.(5.3) by way of Eq.(5.4) becomes.

$$E(X) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = 2 \sum_{i=1}^{n} \left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n+1-i} \right) \leq 2 \sum_{i=1}^{n} (H_n - 1) = 2n(H_n - 1) \leq 2n \ln n \leq 2n \ln n. \tag{5.5}$$

where γ is Euler's constant, and $H_n \approx \ln n + \gamma$ and thus, $H_n - 1 \leq \ln n$. ■

5.12 Decision trees

Exercise 285.

We have 9 coins all of the same weight except one that is a fake and weighs less than the others. We also have a balance scale; any number of coins can be put on one or the other side of the scale at any one time and the scale will tell us whether the two sides weigh the same or which side weighs more (less).
Find the fake coin is 2 weighings.

Proof.
Split the coins into 3 piles of 3 coins each (A,B,C). Weigh two piles against each other (say A, B). Either A or B is identified containing the fake coin or in case of a tie we deduce it is in C. Pick the one containing the fake coin (lighter group of three coins). Once more spliting one of A,B,C into 3 piles of 1 coin and the fake coin will be determined. THe recurrence for finding the number of weighings is $T(n) = T(n/3) + 1$ for n a power of three.

■

Exercise 286.

We have four coins all of the same weight except one that is fake and may weigh LESS or MORE than the other coins (we don't know what the case is). We also have a balance scale; any number of coins can be put on one or the other side of the scale at any one time and the scale will tell us whether the two sides weigh the same or which side weighs more (or less). Can you find the fake coin with only 2 weighings? Explain (and justify your answer).

Proof.

This is a decision-tree based problem. Call the coins A,B,C,D. If we weigh A and B and show it as in $A:B$ we can determine whether the fake coin is one of C,D or one of A,B. A second weighing is required by weighing for example $A:C$. The figure below is a binary tree, with the root on the left and extending rightwards. Note that for the weighings, we only need the `equal`, `not-equal` outcome; that one side might weigh less or more is not used in the solution.

```
A:B  ---Equal---> (C or D is fake) -- A:C ---Equal     D is fake; A,B,C genuine
     |                                    |
     |                                    ---Not Equal C is fake; A,B,D genuine
     --NotEqual-> (A or B is fake) -- A:C ---Equal     B is fake; A,C,D genuine
                                          |
                                          ---Not Equal A is fake; B,C,D genuine
```

■

Exercise 287.

In this problem we reinterpret and regenerate the lower bound on the complexity of sorting (using comparison-based sorting methods) as an interaction between two parties A and B where A runs a sorting algorithms but the control of the compare function $cmp(x,y)$ belongs to B. B will set the input keys to values consistent with the $cmp(x,y)$ output that will force A's algorithm to worst-case performance of $\lg(n!)$ or more.

We are going to describe the lower bound in terms of interactions between party A that generates (simulates) the comparisons of a sorting algorithm and thus interacts with party B. Party B is the adversary that will convince A that no matter what sorting algorithm's steps A runs (simulates), party B will force the algorithm of A to take $\lg(n!)$ comparisons in the worst-case and no less. In other words A runs the sorting algorithm but for the invocations of the compare function $cmp(x,y)$ between two key values A relies on B on the answers. As expected n will be the number of keys of the input that will be involved in the interaction.

The input consists of n (say distinct keys to keep things simple) and they are $x_1,\ldots,x_n$. A does not know the key values, B will assign values to the x_i's consistently through the course of the interaction and at the end B will present the values of the x_i's to A so that A can verify that the output generated is sorted, and the comparisons performed and their outcomes consistent with the assigned values.

What is going to be the decision principle to B to force worst case behavior from A when A asks but for $cmp(x,y)$. Use the InsertionSort algorithm as an example. Note that $cmp(x,y)$ has output -1 if $x<y$ and $+1$ if $x>y$ as all keys are distinct.

Hint. At the very very beginning all possible permutations of the n keys $x_1,\ldots,x_n$ are candidate for the output of sorting. At the end based on the key comparisons generated by A

Proof.
The example will be for three keys $n=3$. A will run InsertionSort in this example. You may explore other possibilities for n or the sorting algorithm. B will use a greedy principle and for $cmp(x,y)$ will say $+1$ or -1 depending on the appearance of x,y in the permutations of the input keys consistent with the outcomes of the comparisons used so far by A and by B's answers. At the very very beginning the input is $\langle x_1,x_2,x_3\rangle$. No comparison request has been generate by A and likewise no comparison outcome by B. Thus any one of the $n!$ permutations can be eventually in the output
Step 0.

$$R_0=\{x_1x_2x_3,x_1x_3x_2,x_2x_1x_3,x_2x_3x_1,x_3x_1x_2,x_3x_2x_1\}$$

Step 1. Consistent with InsertionSort A asks $cmp(x_1,x_2)$. B consults R and realizes that half of the permutations have x_1 followed by x_2 implying $x_1<x_2$ and half implying $x_1>x_2$. At that point B says -1 implying $x_1<x_2$. With that answer B is committing as possible outputs R_1, set R_1 for which x_1 preceded x_2 i.e. $x_1<x_2$.

$$R_1=\{x_1x_2x_3,x_1x_3x_2,x_3x_1x_2\}$$

Step 2. Consistent with InsertionSort A asks then $cmp(x_2,x_3)$. B consults R_1 and realizes that more than half of the permutations have x_3 followed by x_2 implying $x_2>x_3$. At that point B says 1 implying $x_2>x_3$. With that answer B is committing as possible outputs R_2, set R_2 for which x_3 precedes x_2 i.e. $x_2>x_3$.

$$R_2=\{x_1x_3x_2,x_3x_1x_2\}$$

Step 3. Consistent with InsertionSort A asks then $cmp(x_1,x_3)$. B consults R_2 and realizes that half of the permutations have x_1 followed by x_3 and half the other way around. At that point B says 1 implying $x_1>x_3$ but could have said -1 instead. With that answer B is committing as possible outputs R_3, where $R_3=\{x_1x_3x_2\}$ and thus the output is $\langle x_1x_3x_2\rangle$. Moreover B assigns values $x_1=10,x_3=20,x_2=30$. and returns them to A. At that point A confirms that the comparisons performed and their outcome is consistent with these values.
Why $\lg(n!)$? Party B by choosing the outcome that generates R_i such that $|R_i|\geq|R_{i-1}|/2$ for $i=1,2,\ldots$ make

sure that

$$\begin{array}{rcl} |R_i| & \geq & |R_{i-1}|/2 \\ |R_{i-1}| & \geq & |R_{i-2}|/2 \\ & \cdots & \\ |R_3| & \geq & |R_2|/2 \\ |R_2| & \geq & |R_1|/2 \\ |R_1| & \geq & |R_0|/2 \end{array}$$

Multiplying the inequalities we conclude that $|R_i| \geq |R_0|/2^i$. Since $|R_0| = n!$, this implies $|R_i| \geq n!/2^i$. When do we stop? We stop when $|R_i| \leq 1$. This implies

$$n!/2^i \leq |R_i| \leq 1.$$

Solving for i we find $i \geq \lg(n!)$ and since i is an integer this also means $i \geq \lceil \lg(n!) \rceil$. ■

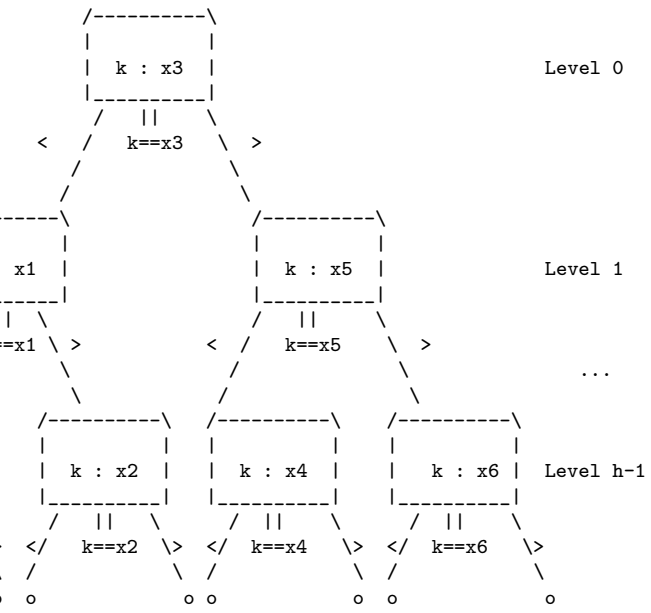
Exercise 288.

Show the lower bound for BinarySearch i.e. at least $\lceil \lg(n+1) \rceil$ comparisons are needed in the worst case to determine whether a key k is in a sorted sequence A of n keys, $A[0..n-1]$ or not.

Proof.

To keep things simple let the input keys be a power of two minus one and key k is the key that we looking for. Every node of the decision tree has three branches for $<$, $=$, and $>$. The number of nodes (in fact internal nodes) is equal to n the number of input keys. The nodes branch out for $<$ and $>$ as the other branch maps to FOUND (i.e. k is equal to some x_i). The number of leaves indicate with an o is $n+1$. Each such leaf maps to an interval which is the range of values for k if k is not in the input.

$$(-\infty, x_0), (x_0, x_1), (x_1, x_2), \ldots (x_{n-1}, x_n), (x_n, \infty).$$



Let h be the height of the decision tree. The number of nodes at level i is 2^i, $i = 0$ through $i = h-1$ are internal nodes, and at level h are the leaves o. Thus the total number of internal nodes at level 0 through $h-1$ is

$$2^0 + 2^1 + \ldots + 2^{h-1} = 2^h - 1,$$

and on each internal node a comparison with a key of the input is performed. Note that the number of comparisons in the worst case is h: one comparison betweek k and a node (single node) of level 0, one with a node of level 1 and so one one with a node of level $h-1$. All input keys must appear as internal nodes and therefore since their number is n we must have enough internal nodes $2^h - 1$ for the n keys to be accommodated i.e.

$$2^h - 1 \geq n$$

Solving for h we find $h \geq \lg(n+1)$, and since h must be integer this means $h \geq \lceil \lg(n+1) \rceil$. ■

Exercise 289.

You may assume that n is a multiple of k. Let n keys have the following property. The first k keys are less than or equal to the next k keys, and so on. Show that sorting them will take $\Omega(n \lg k)$ comparisons in the worst-case.

Proof.
The n sorted keys would be a leaf in a decision tree. There $k!$ ways the first k keys can appear in the input, likewise $k!$ ways for the next subsequence, and so on. Thus we must have as leaves at least $(k!)^{\frac{n}{k}}$ possible output (permutations /combinations) of the inputs. The height of a any decision tree for sorting the n keys should be at least

$$\lg (k!)^{\frac{n}{k}} \geq \frac{n}{k} \lg (k!) \approx \frac{n}{k} \lg (k/e)^k \approx n \lg k - n \lg e.$$

We can elimiate $\approx$ by

$$\lg (k!)^{\frac{n}{k}} \geq \frac{n}{k} \lg (k!) \geq \frac{n}{k} \lg (k/2)^{k/2} \approx n \lg k/2 - n/2.$$

■

Exercise 290.

Can you sort any 4 distinct keys with 5 comparisons only? Explain.
Note: This and the following problem need to be cleaned up.

Proof.
Easy answer: sort the first three keys with InsertionSort. For the fourth key do BinarySearchInsertion into the sorted sequenced of the first three sorted keys. The first three keys are sorted in 3 comparisons; we are done with the fourth key within two more comparisons. The lower bound for sorting with comparisons only indicates that the minimum order of comparisons performed in the worst case to sort n keys is $\lg(n!)$ which for our case $\lg(4!) = \lg 24 = 4.xx$, and the smallest integer greater than $\lg 24$ is 5. How can we sort 4 keys with 5 comparisons? Pick the first three keys and sort them using the comparison sequence of the decision tree depicted on the textbook and also done in class. This is insertion sort, and sorts 3 keys with 3 comparisons. For the fourth key binary search/insert it in the sorted sequence of the first three keys. So if t_1, t_2, t_3 is the first three keys sorted and d is the fourth work as follows with the remaining two comparisons. Compare d to t_2. If d is smaller then compare d to t_1. If d is smaller, output d, t_1, t_2, t_3, otherwise output t_1, d, t_2, t_3. If d is larger than t_2 compare d to t_3. If d is smaller output t_1, t_2, d, t_3 else output t_1, t_2, t_3, d. Total number of comparisons is 3 for insertion sort of the first three keys (done in class) plus two more (comparison of d to t_2 the first and the final is comparison of d to t_1 or t_3) for a total of

```
  Sort(<a,b,c,d>) // Four keys distinct
1.  if a< b   { t1=  a; t2=  b; }                        // 1st comparison
2.   else     { t1=  b; t2=  a; }
3.  if c> t2  { t3= c ; }                                // 2nd comparison
4.   else { if c> t1 { t3=t2; t2=c; t1=t1;}               // 3rd comparison
5.           else    { t3=t2; t2=t1;t1=c; }
6.        }
7.  if d < t2 {                                          // 4th comparison
8.   if d< t1 {t4=t3;t3=t2;t2=t1;t1=d;}                  // This
9.    else    {t4=t3;t3=t2;t2=d;}
10.  }
11. }
12. else {
13.  if d< t3 { t4=t3;t3=d;}                             //or than one
14.   else    { t4=d;}                                   // is 5th.
15. }
16. Output (<t1,t2,t3,t4>);
```

■

```
  Sort(a,b,c,d)
1. if (a < b)    /* Find smallest/largest of first two keys */  COMPARISON  1.
2.     m1 = a , M1= b
3. else
4.     m1 = b , M1 = a
5. if (c < d)   /* Find smallest/largest of next  two keys */   COMPARISON  2.
6.     m2 = c , M2= d
7. else
8.     m2 = d , M2 = c
9. if (m1 < m2)  /* z1 : smallest of smallest */               COMPARISON  3.
10.   z1 = m1 , z2 = m2
11. else
12.   z1 = m2 , z2 = m1.
13.if (M1 > M2)  /* y1 : largest of largest */                 COMPARISON  4.
14.   y1 = M1 , y2 = M2
15. else
16.   y1 = M2 , y2 = M1.
17. /* Minimum in z1  , Maximum in y1 */
18. if z2 < y2                                                COMPARISON  5.
19.   Output < z1 z2 y2  y1 >
20. else
21.   Output < z1 y2 z2  y1 >
```

Exercise 291.

What is a lower bound for the worst-case running time of sorting 5 keys with a comparison-based sorting algorithm? Explain.

Proof.
$\lceil \lg(5!) \rceil = \lceil 120 \rceil = 7$. Can we sort 5 keys with 7 comparisons ? (in the worst case).

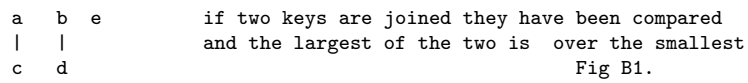


Fig B1.

Step b2. We next compare a to b in **1 comparison**. We have two outcomes depending on whether a or b is larger

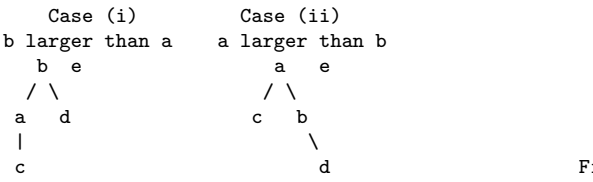


Fig B2.

Step b3. We next deal with key e. We binary-search insert key e into the longest of the two chains, i.e. with referenc to Fig B2. we insert e into $b-a-c$ in Case (i), or in $a-b-d$ in Case (ii). Binary search-based insertion of e into a sorted list of three keys requires **2 comparisons**. For example if case (i) applies, e is compared first to a and if larger than a, then it is compared to b and its position is determined in chain $b-a-c$; otherwise e is smaller than a and must be compared to c in the second comparison. The other case is treated similarly. **Step b4.** The relative position of the keys following the insertion of e looks like:

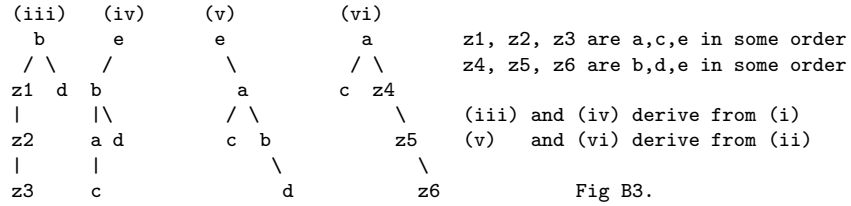


Fig B3.

Everything looks "almost sorted" except for d in cases (iii) and (iv) and c in cases (v) and (vi). Consider case (iii): The position of d versus b is known. The positions of d versus z_1, z_2, z_3 can be determined through binary search in *2 comparisons*, the first comparison involves d and z_2 and depending on the outcome, d is then compared to either z_1 or z_3 and its position in the chain $d-z_1-z_2-z_3$ is fully determined, i.e. the five keys are sorted. Consider case (iv): We know d is less than b and e, so we only need to determine the relative position of d to a and c. This also requires *2 comparisons*. Consider case (vi): This is symmetric to case (iv). Consider case (vi): This is symmetric to case (iii). ■

Exercise 292.

Show that the merging of two sorted sequences of n comparable keys each requires at least $2n-1$ comparisons in the worst-case, matching the upper bound which is also $2n-1$.

Proof.
Let us start with the upper bound. If we have two sorted sequences $a_1, a_2, \ldots$, $b_1, b_2, \ldots$ of distinct keys, where

$$a_1 < b_1 < a_2 < b_2 < \ldots < a_i < b_i < \ldots < a_n < b_n.$$

All of the comparisons above (implied by a $<$) must be performed, to merge those two sequences. If one comparison is missing (not performed) say this is between $a_i : b_i$. We can't then decide whether a_i preceded b_i or not as the only info we have is that $b_{i-1} < a_i ? b_i < a_{i+1} < b_{i+1}$ that cannot determine the relative order of a_i, b_i.
Let us continue with the lower bound. Suppose someone observes that $2n-2$ comparisons can merge the two sequences. Fix $a_i = 2i-1$ and $b_i = 2i$ i.e. $a_i < b_i$ for all $i = 1, \ldots, n$ and run them with the $2n-2$ merging algorithm. This means that one comparison is missing and it can be $b_{i-1} < a_i ? b_i < a_{i+1} < b_{i+1}$ or $b_{i-1} < a_i < b_i < a_{i+1} ? b_{i+1}$. and let that be (as before) $a_i : b_i$. Swap now a_i, b_i so that $a_i = 2i$ and $b_i = 2i-1$ and run again. Same comparisons performed as before and same output. The output will not be correct as $a_i = 2i$ will precede $b_i = 2i-1$ now, but now this is wrong.
What if we use the lower bound technique that utilizes a decision tree? Building a decision tree for the merging problem, the tree must have at least

$$\binom{2n}{n} = \frac{(2n)!}{n! \cdot n!}$$

leaves. This is because out of the $2n$ keys of the output, we can choose in $\binom{2n}{n}$ ways the keys of a_i (in the output), and the remaining keys are the b_i keys. Taking the logarithm of the number of leaves establishes the lower bound on the number of comparisons for merging two n-key sorted sequences.

$$\lg\left(\frac{(2n)!}{n! \cdot n!}\right).$$

Consider

$$\begin{aligned}
\frac{(2n)!}{n! \cdot n!} &= \frac{(2n) \cdot (2n-1) \cdot \ldots \cdot (n+1)}{n \cdot (n-1) \cdot \ldots \cdot 1} \\
&= \frac{(2n) \cdot (2n-1) \cdot \ldots \cdot (2n-i) \cdot \ldots \cdot (2n-(n-1))}{n \cdot (n-1) \cdot \ldots \cdot (n-i) \cdot \ldots \cdot (n-(n-1))} \\
&= \frac{(2n)}{n} \cdot \frac{(2n-1)}{n-1} \cdot \ldots \cdot \frac{(2n-i)}{n-i} \cdot \ldots \cdot \frac{2n-(n-1)}{n-(n-1)} \\
&\geq 2 \cdot \frac{n}{n} \cdot 2 \cdot \frac{(n-1/2)}{n-1} \cdot \ldots \cdot 2 \cdot \frac{(n-i/2)}{n-i} \cdot \ldots \cdot 2 \cdot \frac{n-(n-1)/2}{n-(n-1)} \\
&\geq 2^n
\end{aligned}$$

The latter is due to the fact that

$$\frac{n-i/2}{n-i} \geq 1 \Leftrightarrow n - i/2 \geq n - i \Leftrightarrow i/2 \leq i.$$

Therefore

$$\lg\left(\frac{(2n)!}{n! \cdot n!}\right) \geq \lg 2^n = n$$

This provides a linear bound on the number of comparisons for merging, which is not as good as the previously derived $2n-1$ bound.

■

Exercise 293.

Assume that all keys referenced below are distinct. Let $k \geq l \geq 1$ or rename the variable and let $k+l=n$. For two sorted sequences $A = \langle a_1, \ldots, a_k \rangle$, and $B = \langle b_1, \ldots, b_l \rangle$, with k and l keys each so that $k+l=n$,
(a) Show first that a decision tree for this problem (merging) has

$$\binom{k+l}{k} = \binom{n}{k}$$

leaves.
(b) Deduce a lower bound for merging of

$$\left\lceil \lg \binom{k+l}{l} \right\rceil \geq \max\left(l, \lceil 2n - 1 - 0.5 \lg n \rceil, \left\lceil l \lg\left(1 + \frac{k}{l}\right) \right\rceil \right). \tag{5.6}$$

Proof.
(a) For the output of $k+l=n$ keys, there are k positions that can be occupied by the elements of A and thus the remaining by the elements of B. Therefore the number of ways we can pick the k indexes to fill them (the corresponding elements of the output) with A's keys are

$$\binom{k+l}{k} = \binom{n}{k} = \binom{n}{l} = \binom{k+l}{l},$$

and then we fill them with a_1 the first smallest index and so on. The non-picked are filled with B's elements, b_1 first and so on. Here is an example with $A = \langle a_1, a_2, a_3 \rangle$, and $B = \langle b_1, b_2 \rangle$, Here are the $\binom{5}{3}$ possibilities for the A's elements. The dotted positions are for b_1 first b_2 next.

```
[ a1 a2 a3 .. .. ]
[ a1 a2 .. a3 .. ]
[ a1 a2 .. .. a3 ]
[ a1 .. a2 a3 .. ]
[ a1 .. a2 .. a3 ]
[ a1 .. .. a2 a3 ]
[ .. a1 a2 a3 .. ]
[ .. a1 a2 .. a3 ]
[ .. a1 .. a2 a3 ]
[ .. .. a1 a2 a3 ]
```

(b) We observe first the following.

$$\frac{k+l-i}{l-i} \geq \frac{k+l}{l} \Leftrightarrow kl + l^2 - il \geq kl + l^2 - ik - il \Leftrightarrow 0 \geq -ik \Leftrightarrow ik \geq 0.$$

which is used in the third line below to establishe the first inequality.

$$\begin{aligned}
\binom{k+l}{k} &= \frac{(k+l)\cdot(k+l-1)\cdot\ldots\cdot(k+2)\cdot(k+1)}{l\cdot l-1\cdot\ldots\cdot 2\cdot 1} \\
&= \prod_{i=0}^{l-1} \frac{k+l-i}{l-i} \\
&\geq \prod_{i=0}^{l-1} \frac{k+l}{l} \\
&= \left(\frac{k+l}{l}\right)^l.
\end{aligned}$$

A decision tree (for merging) with $\binom{k+l}{k}$ leaves has minimal height at least $\lg\binom{k+l}{k}$ and therefore utilizing the previous result,

$$\lg\left(\begin{array}{c}k+l\\k\end{array}\right) \geq \lg\left(\frac{k+l}{l}\right)^{l} \geq l\lg\left(1+\frac{k}{l}\right).$$

Since the height is always an integer equivalently we have a height of at least

$$\left\lceil \lg\left(\begin{array}{c}k+l\\k\end{array}\right)\right\rceil \geq \left\lceil \left(l\lg\left(1+\frac{k}{l}\right)\right)\right\rceil .$$

This establishes the third part of Eq.(5.6). Given that $k \geq l$, consider again, the minimal height of a decision tree for merging but utilize inequality Eq.(I.3.6) i.e. $\binom{2n}{n} \geq 2^n$,

$$\left\lceil \lg\left(\begin{array}{c}k+l\\k\end{array}\right)\right\rceil \left\lceil \lg\left(\begin{array}{c}k+l\\l\end{array}\right)\right\rceil \geq \left\lceil \lg\left(\begin{array}{c}l+l\\l\end{array}\right)\right\rceil = \left\lceil \lg\left(\begin{array}{c}2l\\l\end{array}\right)\right\rceil \geq \left\lceil \left(\lg 2^l\right)\right\rceil = \left\lceil 2^l\right\rceil .$$

This establishes the first part of Eq.(5.6). Given that $k \geq l$, consider again, the minimal height of a decision tree for merging but utilize inequality Eq.(I.3.5) i.e. $\frac{2^{2n}}{2\sqrt{n}} \leq \binom{2n}{n}$,

$$\left\lceil \lg\left(\begin{array}{c}k+l\\k\end{array}\right)\right\rceil \geq \left\lceil \lg\left(\begin{array}{c}2l\\l\end{array}\right)\right\rceil \geq \left\lceil \left(\frac{2^{2n}}{2\sqrt{n}}\right)\right\rceil = \lceil 2n-1-0.5\lg n\rceil .$$

This establishes the second part of Eq.(5.6). Thus we have established all parts of Eq.(5.6).
Some observations follow:
(a) If $k=l=n$, the second part of Eq.(5.6) is close to the $2n-1$ lower bound of the previous problem.
(b) If $k=n-1$ and $l=1$,

$$\left\lceil \lg\left(\begin{array}{c}k+l\\k\end{array}\right)\right\rceil = \left\lceil \lg\left(\begin{array}{c}n-1+1\\n-1\end{array}\right)\right\rceil \lceil \lg n\rceil .$$

■

Exercise 294.

Show that there can be no comparison-based sorting algorithm whose running-time is at most $10n$ for at least 10% of the $n!$ inputs of length n.

Proof.
The decision tree of height $10n$ has 2^{10n} leaves that need to accommodate the $0.10n!$ permutations (inputs). This is not possible as it is impossible for $2^{10n} > 0.10n!$. For a binary tree to accommodate $0.10n!$ inputs as leaves it must have height

$$\lg(0.10n!) \approx \lg(n!) + \lg 0.10 \geq n\lg n - n\lg e + \lg 0.10 > 10n.$$

■

Exercise 295.

Show that the median of n distinct integer requires at least $n-1$ comparisons.

Proof.
If we perform fewer than $n-1$ comparisons there might be keys not compared that can be the median. Pictorial proof by example. On graph with the keys, represent the key comparisons with edges. Graph with n vertices and $n-2$ edges is not connected. Identify the two or more connected components. Call one A and everything else B. Create two instances of input from the original input call it x: I_1 and I_2 where the vertices in B are either added D or subtracted D. In the former case we make them larger than all the elements of x, in the latter smalles than all the elements of x. Then it can't be the comparison-based median algorithm to give correct answers to x, I_1 and I_2 by performing the same comparisons in all three cases.

```
 10       20      Number of keys is n=9; number of comb is n-2=7
  |        |      B.T.W median   is 50
 30-------40      Min is 10
                  Max is 90
 50       60      D= Max-Min = 80
  |        |      Graph has two connected components
 70------ 80      A with 10,20,30,40 and
  |               B with 50,60,70,80,90
 90
```

Create the following input instances I_1 and I_2

```
 10       20      Add to B 80
  |        |      B with 50,60,70,80,90
 30-------40
                 I1
 130     140
  |        |
150------160
  |
170

 10       20     Subtract from B 80
  |        |     B with 50,60,70,80,90
 30-------40

-30      -20
  |        |
-10------  0
  |
 10
```

Note that there are better lower bounds for the median than this naive one. ■

Exercise 296.

Generalized binary search. You are given a sorted array of n keys $A[0..n-1]$ and two keys k_1 and k_2, where $k_1 < k_2$. Find all the keys $< k_1$, $\geq k_1$ AND $\leq k_2$, and $> k_2$. In other words count the number of keys in each group. Key k_1, k_2 might not be in A.
(a) Give an algorithm for this problem that is $o(n)$ in terms of number of comparisons.
(b) In an attempt to determine whether the algorithm in (a) is "optimal", we are going to count in a decision tree for a comparison-based solution to this problem the number of leaves in an algorithm for this problem. How many leaves does it have? (Hint: Use a bin and ball argument with some labeling.)
(c) Decude then a lower bound on the number of comparisons similarly to the technique used for sorting.

Proof.
(a) Do a binary search of k_1 into A requiring $\lceil \lg(n+1) \rceil$ keys and record indices i_1 and i_2 such that $A[i_1] \leq k < A[i_2]$. (if there is no i_2 use n instead.) Then do a binary search of k_2 in A or better in $A[i_2..n-1]$ and thus determine indices i_3 and i_4 accordingly. Skipping boundary details return three values as needed. Total time $O(\lg n)$.
(b) The decision tree has leaves of the form $0^i 1^j 2^k$, where $i+j+k=n$ and $i,j,k \geq 0$. These $0,1,2$ indicate the tree areas of A split by k_1 and k_2 as containing i, j and k keys respectively. For example the left-most area contains $A[0],\ldots,A[i-1]$. The number of leaves is the number of ways we can place 2 markers on $n+2$ slots. After assigning a marker to 2 slots, the three area are clearly disambiguated. In how many ways can we divide the n keys? The answer is

$$\binom{n+2}{2} = \frac{(n+2)(n+1)}{2}.$$

```
  Input        :   10 20 30 40 50 60 70

  Slots        : -  -  -  -  -  -  -  - -
  Marked Slots :       x     x
Implication    : 0  0  x  1  x  2  2  2 2
Implication    :10 20  x 30  x 40 50 60 70
Possibility
for k1, k2           k1=25  k2=35
Implication       2,     1,     4  keys <25  , >=25 <= 35, > 35
```

(c) The lower bound for sorting leads to a lower bound of

$$\lceil \lg \frac{(n+2)(n+1)}{2} \rceil$$

for this problem. This leads to

$$\lceil \lg(n+2) + \lg(n+1) - 2 \rceil.$$

■

Chapter 6

Hashing

6.1 Hash tables and their properties

More interesting problems related to hashing are in the probability section.

Exercise 297.

How many words are there in the textbook? Justify your answer. (BTW GT has 708 pages.) Make sure your answer has lots of trailing zeroes. Thus 1,000,000 has trailing zeroes, but 123,456 does not have. Even if the correct answer is 100,000 the former will get you 0 points but the latter will get you a 0!

Proof. 708pages and roughly 45 lines per page and roughly 80 characters per page gives 2.4M characters. The average length of an English word in a text is 5 thus roughly 500,000 words. ■

Exercise 298.
TRUE or FALSE?

(1) The worst-case running time of an Insert for Hashing by open-addressing is $\Theta(n)$, when inserting a new key into a table T of N slots containing n keys, $n < N$. (No overflow.)

(2) The running time of a Search for Hashing by open-addressing is $O(n)$, when inserting a new key into a table T of N slots containing n keys, $n < N$. (No overflow.)

Proof.
(1) FALSE, it is $\Theta(N)$.
(2) FALSE, it is $O(N)$. ■

Exercise 299.
Provide short answers to the following questions.

(1) (1) Give the running time for Search in hashing with collision resolution by open-addressing for n keys maintained in a table with N slots.
(2) (2) Give the worst-case running time for Search in hashing with collision resolution by open-addressing for n keys maintained in a table with N slots, where $n < N$.
(3) (3) Give the worst-case running time for Search in hashing with collision resolution by chaining (or by chains) for n keys maintained in a table with N slots, where $n < N$.
(4) (4) Give the running time of Search in hashing with collision resolution by chaining (or by chains) for n keys maintained in a table with N slots, where $n < N$.

Proof.
(1) $T(n,N) = O(N)$
(2) $T(n,N) = \Theta(N)$
(3) $T(n,N) = \Theta(n)$
(4) $T(n,N) = O(n)$

■

Exercise 300.

Answer the following questions by justifying your answer. If necessary provide asymptotically tight bounds. Expected times are under the SIMPLE UNIFORM HASHING (function) ASSUMPTION.
(a) What's the running time of Hash Table Search for collision resolution by chaining?
(b) What's the running time of Hash Table Unsuccessful Search for collision resolution by chaining?
(c) What's the running time of Hash Table Delete(z) for collision resolution by chaining? (You know the element containing the key).
(d) What's the running time of Hash Table Delete(k) for collision resolution by chaining? (You know the key value only.)
(e) What's the expected running time of Hash Table Insert for collision resolution by chaining? (You know the key value only.)
(f) What's the expected running time of Hash Table Successful Search for collision resolution by chaining? (You know the key value only.)
(g) What's the expected running time of Hash Table Unsuccessful Search for collision resolution by chaining? (You know the key value only.)

Proof. (a) $O(n)$. (b) $\Theta(n)$, (c) $\Theta(1)$. (d)-(f) Skipped. (g) $\Theta(1+a)$.

■

Exercise 301.

Answer the following questions by justifying your answer. If necessary provide asymptotically tight bounds. For expected running time it is under Uniform Hashing assumption.
(a) What's the running time of Hash Table Successful Search for collision resolution by open-addressing?
(b) What's the running time of Hash Table Unsuccessful Search for collision resolution by open-addressing?
(c) What's the running time of Hash Table Insert for collision resolution by open-addressing assuming the key is not already in the table?
(d) What's the expected running time of Hash Table Successful Search for collision resolution by open-addressing?
(e) What's the expected running time of Hash Table Unsuccessful Search for collision resolution by open-addressing?
(f) What's the expected running time of Hash Table Insert for collision resolution by open-addressing assuming the key is not already in the table?
(g) What's the expected running time of Hash Table Insert for collision resolution by open-addressing assuming the key is not already in the table? The table contains 50 keys before the insertion and it has 1000 slots.
(h) What's the expected running time of Hash Table Delete for collision resolution by open-addressing assuming the key is not already in the table? The load factor prior to deletion is $1-1/e$ where e is the base of the Neperian algorithms. $(1-1/e$ is roughly 60%. Which of the following is close to the average number of probes (i) 1.6, (ii) 3.5, (iii) a, (iv) 5.5, (v) e^3?

Proof. (a) $O(N)$. (b) $\Theta(N)$, (c) $O(N)$. (d)-(g) Skipped. (h) (i)

■

Exercise 302.

Consider HashInsert and HashSearch as shown below for a table T and collision resolution by chaining. Assume T has N slots indexed 0 through $N-1$, and the hash function is $h(k,i)$ where i is the probing index and runs from $i=0$ to $i=N-1$.

```
    HashInsert(k,T,N)
1.   i = 0;
2.   do {
3.     j = h(k,i);
4.     if ( T[j] == NULL ) {
5.        T[j] = k ;
6.        return(j) ;
7      }
8.     i=i+1 ;
9.   } while ( i< N );
10.  error("Overflow");
```

```
    HashSearch(k,T,N)
1.   i = 0;
2.   do {
3.     j = h(k,i);
4.     if ( T[j] == k ) {
5.
6.        return(j) ;
7      }
8.     i=i+1 ;
9.   } while ( T[j] != NULL AND  i< N );
10. return (-1);
```

What would the code for HashDelete be? Can it be as shown below? Explain or explain not.

```
    HashDelete(k,T,N)
1.   i = 0;
2.   do {
3.     j = h(k,i);
4.     if ( T[j] == k ) {
5.        T[j] = NULL ;
6.        return(j) ;
7      }
8.     i=i+1 ;
9.   } while ( T[j] != NULL AND  i< N );
10. return (-1);
```

Proof.

The code for HashSearch terminates when condition $T[j] == NULL$ is satisfied i.e. the probing sequence $h(k,0),\ldots,j=h(k,i)$ is examined. In a naive implementation of HashDelete (shown), deletion of slot j would result in setting $T[j]=NULL$ (See line 5). As a result subsequent calls to HashSearch starting at $h(k,0)$ would never probe slots $h(k,i+1),\ldots$ as $h(k,i)=NULL$ and the conditional of line 9 of HashSearch becomes false and the loop terminates. To avoid such an erratic behavior, instead of marking table slots with a single symbol NULL we mark slots with one of two symbols: NULL (to mean initially FREE, as before) and DELETED to mean that a slot becomes available for insertion after a HashDelete operation. The code for HashInsert and HashDelete is to be modified as follows.

```
    HashInsert(k,T,N)
1.   i = 0;
2.   do {
3.     j = h(k,i);
4.     if ( ( T[j] == NULL ) OR ( T[j] == DELETED) ) {
5.        T[j] = k ;
6.        return(j) ;
7      }
8.     i=i+1 ;
```

```
9.   } while ( i< N );
10.  error("Overflow");

    HashDelete(k,T,N)
1.   i = 0;
2.   do {
3.     j = h(k,i);
4.     if ( T[j] == k ) {
5.        T[j] = DELETED ;
6.        return(j) ;
7      }
8.     i=i+1 ;
9.   } while ( T[j] != NULL AND  i< N );
10. return (-1);
```

■

6.2 Hashing examples

Exercise 303.

(a) Consider inserting keys 25,19,13,7,1 into a hash table of length $N = 6$ using open addressing with the auxiliary hash function $h(k) = k \bmod N$. Illustrate the result of inserting these keys using (i) linear probing with hash function $h(k,i) = (h(k)+i) \bmod N$, (ii) quadratic probing with hash function $h(k,i) = (h(k)+i^2) \bmod N$.
(b) Consider inserting keys 7,13,1,37,25. (Same set up as before.)
(c) Consider inserting keys 11, 22, 20, 6, 26, 16, 7 into a hash table of length $N = 9$; same setup otherwise as before.
(d) Consider inserting keys 1, 3, 5, 6, 12, 23, 10, 17, 34 into a hash table of length $N = 10$ using open addressing with the primary hash function $h(k) = k \bmod N$. Illustrate the result of inserting these keys using (i) linear probing with hash function $h(k,i) = (h(k)+i) \bmod N$, (ii) quadratic probing with hash function $h(k,i) = (h(k)+i^2) \bmod N$.

Proof. **(a) and (b).**

```
                          n=5 m=6
     @a,b,c, etc  means collisions at slot a,b,c, etc in that sequence
                   for i=0, i=1, i=2, etc
        Part (a)                                  Part (b)
    25,19,13,7,1                              7,13,1,37,25

   0:                                         0:
   1: 25                                      1: 7
   2: 19 @1                                   2: 13 @ 1
   3:                                         3:
   4:  7 @1,2,5                               4: 37 @ 1,2,5
   5: 13 @1,2                                 5: 1  @ 1,2

1 @ 1,2,5,4,5,2                           25 @ 1,2,5,4,5,2
overflow!                                 overflow!

#Collisions: 12                           #Collisions : 12
```

(c). Skipped.
(d). Shows an alternate description of collisions

```
         LinearProbing        Quadratic Probing
Slot
 0 :   10                   10
 1 :    1                    1
 2 :   12                   12
 3 :    3                    3
 4 :   23@3                 23@3
 5 :    5                    5
 6 :    6                    6
 7 :   17                   17
 8 :   34@4,5,6,7           34@4,5
 9 :
  k@a,b,c means key k had collision at a,b,c before inserted here
```

In the Linear probing case you can observe the long cluster that causes problems to 34. Observe that for quadratic probing the collisions are fewer (by two) for the same key.

■

Exercise 304.

Consider inserting keys 1, 3, 5, 6, 12, 23, 10, 17, 34 into a hash table with $N = 11$ slots using open addressing with the primary hash function $h'(k) = k \bmod N$. Illustrate the result of inserting these keys using (i) linear probing with hash function $h(k,i) = (h'(k) + i) \bmod N$, (ii) quadratic probing with hash function $h(k,i) = (h'(k) + i^2) \bmod N$.

What is the total number of collisions observed? ($\bmod$ is the $\%$ operator.)

coll@a,b,c means a,b,c keys caused collision in the indicated slot.

Proof.

Slot	LinearProbing	Quadratic Probing
0 :		10
1 :	1 coll@12,23,34	1 coll@12,23,34
2 :	12 coll@23,34	12 coll@23,34
3 :	3 coll@23,34	3
4 :	23 coll@34	34
5 :	5 coll@34	5 coll@23,34
6 :	6 coll@17,34	6 coll@17,34
7 :	17 coll@34	17
8 :	34	
9 :		
10 :	10	23 coll@10,34

In the Linear probing case you can observe the long cluster that causes problems to 34. Observe that for quadratic probing the collisions are fewer (by one). ■

Exercise 305.

Consider inserting $n = 5$ keys in this specific order 31,25,19,13,7 into a hash table with $N = 6$ slots using open addressing with the auxiliary hash function $h'(k) = k \bmod N$. Illustrate the result of inserting these keys using quadratic probing with hash function $h(k,i) = (h'(k) + i^2) \bmod N$. What is the total number of collisions observed? ($\bmod$ is the $\%$ operator.)

0	
1	
2	
3	
4	
5	

Proof.

```
[      collision symbol is @         ]
[ Only slot contents AND #collisions ]
[ @x, @y indicates key had collisions]
[ at slot x and y before insertion   ]
|0:
|1: 31      #collisions: 12
|2: 25 @1
|3:
|4: 13 @1, @2, @5
|5: 19 @1, @2
key 7: @1, @2, @5,@4,@5,@2
```

■

6.3 Some challenging problems

Exercise 306.

(Cormen Leiserson Rivest and Stein Problem 11-3.)
Suppose that we are given a key k to search for in a hash table with positions $0,\ldots,N-1$, and suppose that we have a hash function h mapping the key space into the set $\{0,\ldots,N-1\}$. The search scheme is as follows.
1. Compute the value $i=h(k)$ and set $j=0$.
2. Probe in position i for the desired k. If you find it, or if this position is empty, terminate the search.
3. Set $j=(j+1) \bmod N$ and $i=(i+j) \bmod N$ and return to step 2.

Assume that m is a power of 2.

a. Show that this scheme is an instance of the general quadratic probing scheme by exhibiting the appropriate constant c_1,c_2 for $h(k,i)=(h(k)+c_1 i+c_2 i^2) \bmod N$.

The hashing scheme searches entries $h(k), h(k)+1, h(k)+1+2, h(k)+1+2+3, \ldots, h(k)+1+2+\ldots+j \bmod N$, where N is a power of two.

The value of i after j iterations (i.e. after $j+1$ probes) is $i=h(k,j)=h(k)+1+2+3+\ldots+j=h(k)+j(j+1)/2=h(k)+1/2j+1/2j^2 \bmod N$, i.e. we indeed have a quadratic probing scheme with $c_1=c_2=1/2$ and hash function $h(k,j)$.

b. Prove that this algorithm examines every table position in the worst case.

Proof. a. The hashing scheme searches entries $h(k), h(k)+1, h(k)+1+2, h(k)+1+2+3, \ldots, h(k)+1+2+\ldots+j \bmod N$, where N is a power of two.
The value of i after j iterations (i.e. after j probes) is $i=h(k,j)=h(k)+1+2+3+\ldots+j=h(k)+j(j+1)/2=h(k)+1/2j+1/2j^2 \bmod N$, i.e. we indeed have a quadratic probing scheme with $c_1=c_2=1/2$ and hash function $h(k,j)$.

b. (Proof by contradiction). Suppose that the algorithm does not examine every table position in the m probes $h(k,j)$, $j=0,\ldots,N-1$. This means that at least one table entry is visited at least twice (and at least one is not visited at all) i.e. there exist $0 \le l,r \le N-1$ such that $h(k,l)=h(k,r)$, $l \ne r$ and therefore $h(k,l)-h(k,r) \equiv 0 \pmod N$. Without loss of generality, we assume $l<r$.

We shall then show that for l,r as above we have that $h(k,l) \ne h(k,r)$, and therefore $h(k,l)-h(k,r) \not\equiv 0 \pmod N$, i.e. a contradiction. It is true that $h(k,r)-h(k,l)=r(r+1)/2-l(l+1)/2=(r-l)(r+l+1)/2 \equiv 0 \pmod m$.

Consider $(r-l)(r+l+1)/2 \equiv 0 \pmod m$ or in the other words. $(r-l)(r+l+1)/2=AN$ for some integer A. For this to be true, N, which is a power of two, or a power of two smaller than N must divide $r-l$ or $r+l+1$ or both. We distinguish two major cases.

1. Let $r-l$ be even (i.e. r,l are both odd or both even). Then, $r+l+1$ is odd and it is not divisible by N or 2 as both N (a power of 2) and 2 are even and $r+l+1$ was derived to be odd. On the other hand, $r-l \le N-1$ as $r,l \le N-1<N$. Then $(r-l)/2<N$ and therefore N cannot divide $(r-l)/2$ either. This means that $(r-l)(r+l+1)/2 \not\equiv 0 \pmod N$, since N can divide neither $r+l+1$ nor $r-l$.
2. Let $r-l$ be odd. Then, if one of r,l is odd, the other one is even, and the sum $r+l+1$ is even. As $r-l$ is odd it is not divisible by N or 2. We shall show that $(r+l+1)/2$ is not divisible by N either. As $r,l \le N-1$, $r+l+1 \le 2(N-1)+1=2N-1$. Since $r+l+1$ is even, $2N-1$ is odd, thus $r+l+1$ can not be equal to $Nm-1$ and therefore $r+l+1 \le 2N-2=2(N-1)$. Then $(r+l+1)/2 \le N-1<N$, and therefore it is not divisible by N (as this would imply that $(r+l+1)/2 \ge N$). This means that $(r-l)(r+l+1)/2 \not\equiv 0 \pmod N$ as well.

For both cases we derived a contradiction $h(k,l)-h(k,r)=(r-l)(r+l+1)/2 \not\equiv 0 \pmod N$ to the assumption that $h(k,l)-h(k,r) \equiv 0 \pmod N$. Since no two values among the 0 through $N-1$ collide with each other, they cover all the values/indices/slots of the hash table as required. ∎

Exercise 307.

Another variation.

Suppose that we are given a key k to search for in a hash table with positions $0,\ldots,m-1$, and suppose that we have a hash function h mapping the key space into the set $\{0,\ldots,m-1\}$. The search scheme is as follows.

1. Compute the value $i=h(k)$ and set $j=0$.
2. Probe in position i for the desired k. If you find it, or if this position is empty, terminate the search.
3. Set $j=(j+1) \bmod m$ and $i=(i+j) \bmod m$ and return to step 2.

Assume that m is a power of 2.

a. Show that this scheme is an instance of the general quadratic probing scheme by exhibiting the appropriate constant c_1,c_2 for $h(k,i)=(h(k)+c_1 i+c_2 i^2) \bmod m$.

The hashing scheme searches entries $h(k),h(k)+1,h(k)+1+2,h(k)+1+2+3,\ldots,h(k)+1+2+\ldots+(j-1) \bmod m$, where m is a power of two.

The value of i after j iterations (i.e. after j probes) is $i=h(k,j)=h(k)+1+2+3+\ldots+j-1=h(k)+j(j-1)/2=h(k)-1/2j+1/2j^2 \bmod m$, i.e. we indeed have a quadratic probing scheme with $-c_1=c_2=1/2$ and hash function $h(k,j)$.

b. Prove that this algorithm examines every table position in the worst case.

Proof. a. The hashing scheme searches entries $h(k),h(k)+1,h(k)+1+2,h(k)+1+2+3,\ldots,h(k)+1+2+\ldots+j-1 \bmod m$, where m is a power of two.

The value of i after j iterations (i.e. after j probes) is $i=h(k,j)=h(k)+1+2+3+\ldots+j=h(k)+j(j-1)/2=h(k)-1/2j+1/2j^2 \bmod m$, i.e. we indeed have a quadratic probing scheme with $-c_1=+c_2=1/2$ and hash function $h(k,j)$.

b. (Proof by contradiction). Suppose that the algorithm does not examine every table position in the m probes $h(k,j)$, $j=0,\ldots,m-1$. This means that at least one table entry is visited at least twice (and at least one is not visited at all) i.e. there exist $0\le l,r\le m-1$ such that $h(k,l)=h(k,r)$, $l\neq r$ and therefore $h(k,l)-h(k,r)\equiv 0 \pmod m$. Without loss of generality, we assume $l<r$. We shall then show that for l,r as above we have that $h(k,l)\neq h(k,r)$, and therefore $h(k,l)-h(k,r)\not\equiv 0 \pmod m$, i.e. a contradiction. It is true that $h(k,r)-h(k,l)=r(r-1)/2-l(l-1)/2=(r-l)(r+l-1)/2\equiv 0 \pmod m$.

Consider $(r-l)(r+l-1)/2\equiv 0 \pmod m$ or in the other words. $(r-l)(r+l-1)/2=Am$ for some integer A. For this to be true, m, which is a power of two, or a power of two much smaller than m must divide $r-l$ or $r+l-1$ or both. We distinguish two major cases.

1. Let $r-l$ be even (i.e. r,l are both odd or both even). Then, $r+l-1$ is odd and it is not divisible by m or 2 as both m (a power of 2) and 2 are even and $r+l-1$ was derived to be odd. On the other hand, $r-l\le m-1$ as $r,l\le m-1<m$. Then $(r-l)/2<m$ and therefore m cannot divide $(r-l)/2$ either. This means that $(r-l)(r+l-1)/2\not\equiv 0 \pmod m$, since m can divide neither $r+l+1$ nor $r-l$.
2. Let $r-l$ be odd. Then, if one of r,l is odd, the other one is even, and the sum $r+l-1$ is even. As $r-l$ is odd it is not divisible by m or 2. We shall show that $(r+l-1)/2$ is not divisible by m as well. As $r,l\le m-1$, $r+l-1\le 2(m-1)-1=2m-3$. Since $r+l+1$ is even, $2m-3$ is odd, thus $r+l+1$ can not be equal to $2m-3$ and therefore $r+l-1\le 2m-4=2(m-2)$. Then $(r+l-1)/2\le m-2<m$, and therefore it is not divisible by m (as this would imply that $(r+l-1)/2\ge m$). This means that $(r-l)(r+l-1)/2\not\equiv 0 \pmod m$ as well.

For both cases we derived a contradiction $h(k,l)-h(k,r)=(r-l)(r+l-1)/2\not\equiv 0 \pmod m$ to the assumption that $h(k,l)-h(k,r)\equiv 0 \pmod m$. Since no two values among the 0 through $m-1$ collide with each other, they cover all the values/indices/slots of the hash table as required. ■

Exercise 308.

In open addressing under the uniform hashing assumption, let $p(n,N)$ be the probability that there are no collisions, during the insertion of n keys into a hash table T with N slots that is originally empty.
Show by induction that

$$p(n,N) \leq \exp(-\frac{n(n-1}{2N}).$$

Deduce that for $n > \sqrt{N}$ the probability of having a collision grows to close to one.
Note that $\exp x = e^x \geq 1+x$ **for any real** x**.**

Proof.
Consider $p(n+1,N)$ and argue similarly to the derivation of the number of probes for Insertion. For the $n+1$-st key not to cause any collision in a hash table with n keys inserted with no collisions, it must hit an empty slot an event that occurs with probability $1-n/N$. This assumes that the insertion of the $n+1$-st key is independent of the previous insertions. Therefore $p(n+1,N) = (1-n/N)p(n,N)$, since the probability that the insertion of say $n+1$ cause not collision is the probability that the insertion of the $n+1$-st key causes no collisions and the prior insertion of the remaining n keys caused no collisions; the latter is $p(n,N)$. Therefore

$$\begin{aligned} p(n+1,N) &= (1-n/N)p(n,N) \\ &= (1-n/N)(1-(n-1)/N)p(n-1,N) \\ &= (1-n/N)(1-(n-1)/N)\ldots(1-1/N)(1-0/N)p(0,N) \\ &= (1-n/N)(1-(n-1)/m)\ldots(1-1/N) \end{aligned}$$

Since $e^x \geq 1+x$, setting $x = -i/N$ we obtain that $1-i/N \leq e^{-i/N}$. Therefore

$$\begin{aligned} p(n+1,N) &= (1-n/N)(1-(n-1)/N)\ldots(1-1/N) \\ &\leq \exp(-n/N)\exp(-(n-1)/N)\ldots\exp(-1/N) \\ &\leq \exp(-(1+2+\ldots+n)/N) \\ &= \exp(-n(n+1)/2N) \end{aligned}$$

Therefore $p(n,N) \leq \exp(-(n-1)n/(2N))$ as required. Consider $n \gg \sqrt{N}$, the $n(n-1)/2N \gg 1$, and thus $p(n,N)$ is upper bounded by $\exp(-A)$, where A is large and positive. However $\exp(-A)$ goes to zero and thus $p(n,N)$ goes to zero as well.
Note. One might remark that the probability of having a collision is $0+1/N+2/N+\ldots+n/N = n(n-1)/(2N)$. This is not correct. This bound is not exact but an upper bound. Thus, if one uses

$$P(\text{Nocollision}) + P(\text{Collision}) = 1$$

given that

$$P(\text{Collision}) \leq n(n-1)/(2N)$$

one can derive a lower bound for

$$P(\text{Nocollision}) \geq 1 - n(n-1)/(2N)$$

However this is not useful. To understand why the bound above is not exact think of the case of three keys only.

```
CP=CollisionProbability   NCP= Probability of NO collisions
                |--> CP=0
No key-> 1stkey-                  |--> CP= 1/N
                |-->NCP=1-->2ndkey-                       |-> CP= 2/N
                                  |-->NCP= 1-1/N-->3rdkey-
                                                        |->NCP=1-2/N
```

Think of the diagram as a binary tree with root on the left. Let's see for what probability we reach a leaf a CP or NCP and add the probabilities of reaching all CP leaves and all NCP leaves. There is one NCP leaf on the rightmost side. We reach this with probability $(1-1/N)(1-2/N)$ which is the probability of no collisions after three keys have been inserted. The probability bound is the product of probabilities on all intermediate nodes. There are three CP nodes. The topmost one is 0, the second one is reached with probability $1/N$ and the last one with probability $(1-1/N)2/N$. Thus the probability of having a collision after three keys are inserted is $1/N+2/N-(1/N)(2/N)$ which is close to $1/N+2/N$ but not equal to. This problem is a variant of a problem known as the Birthday problem. Let $N=365$ (forget about leap years). Suppose you have n people in a room. What is the probability of having no two with the same birthday? For what value of n is the probability of having two people with the same birthday close to 1? One can see that for $n=\sqrt{N}$ collisions start to show up with a significant (non negligible) probability, i.e. people exist with the same birthdays.
This problem is also known as the Birthday problem. Let $m=365$ (forget about leap years). Suppose you have n people in a room. What is the probability of having no two with the same birthday? For what value of n is the probability of having two people with the same birthday close to 1? One can see that for $n=\sqrt{m}$ collisions start to show up with a significant (non negligible) probability, i.e. people exist with the same birthday. ■

Exercise 309.

We would like to determine whether $k \in A$ or not. We would attempt to examine the problem from a deterministic and randomized point of view.
(a) Give a deterministic algorithm for this problem and its
(b) Say there is a randomized algorithm that determines membership in A i.e. whether $k \in A$ or $k \notin A$. The randomized algorithm performs a test $T(k)$ that take time $\Theta(1)$ i.e. constant time. If $k \in A$ then $T(k)$ is always a YES indicating indeed $k \in A$. If $k \notin A$ then $T(k)$ is a YES appears with probability $1-p$ indicating $k \in A$ (false positive) and a NO appears with probability p indicating $k \notin A$ correctly. A test $T(k)$ is independent of any previous tests involving k or other keys. Thus it is possible for the same key k to have a YES followed by a NO. The randomized algorithm is repeated N times for each key k. What is its running time?
(c) if $k \notin A$ what is the probability that k passes all m tests?
(d) Let Q be some fixed parameter. How large should m be so that a key k is incorrectly labeled is at most Q? (You may assume $0 < Q < 1$.)
(e) How many tests do we need until we can conclude with cetainty $k \notin A$? Give the expected number of tests $E[T]$.

Proof.
(a) Linear search in $O(n)$ time.
(b) Obviously $\Theta(N)$.
(c) It is $(1-p)^N$.
(d) The probability for a key k such that $k \notin A$ we have to have a false positive conclusion is $(1-p)^N$. THis is $(1-p)^N \leq Q$ for $N \leq \lg Q / \lg(1-p)$. Thus for $N > \lg Q / \lg(1-p)$ the probability of false positive is Q or less.
(e) This is a geometric distribution. The probability that we have a proof of non-membership in i tests is $P(T=i) = (1-p)^{i-1}p$. Then the expected number of tests $E[T]$ is

$$\begin{aligned} E[T] &= \sum_{i=0}^{\infty} i(1-p)^{i-1}p \\ &= p/(1-p) \cdot \sum_{i=0}^{\infty} i(1-p)^i \\ &= \frac{p}{1-p} \cdot \frac{1-p}{(1-(1-p))^2} \\ &= \frac{p}{1-p} \cdot \frac{1-p}{p^2} \\ &= \frac{1}{p}. \end{aligned}$$

The latter step come from anohter problem when $x \to \infty$ and $x < 1$. We then substitute $1-p$ for x. ■

Exercise 310.

The following question relates to the results of the previous problem.
You have a fair coin where in a coin toss the probability of Heads is $p_H = 1/2$ and so is the probability of Tails $p_T = 1/2$. (From now on we use H or T to indicate one or the other outcome.)
(a) What is the expected number of tosses for a T? (For example, in the experiment HHHT we have four tosses to a T.)
(b) What is the expected number of tosses to get TH? (For example, in the experiment TTTH we have four tosses to a TH.)

Proof.
(a) It is a geometric distribution with expected number of tosses equal to $1/p_T$, where $p_T = 1/2$, i.e. two tosses.
(b) First in an expected number of two tosses i.e. $1/p_T$ we get a T. From that point on in an expected number of $1/p_H$ tosses we get an H. By the sum of expectations $E[A+B] = E[A] + E[B]$ we get that the answer is the sum $1/p_T + 1/p_H = 4$. ■

Chapter 7

Selection and complexity of sorting

7.1 Selection algorithm properties

Exercise 311.
Answer the following questions briefly by justifying your answers. If needed provide asymptotically tight bounds as answers.
(a) What's the running time of RandSelect(A,n,i)?
(b) What's the worst-case running time of RandSelect(A,n,i)?
(c) What's the expected running time of RandSelect(A,n,i)?
(d) What's the worst-case running time of (deterministic) Select(A,n,i) ?
(e) What's the running time of (deterministic) Select(A,n,i) ?

Proof. (a) $T(n) = O(n^2)$
(b) $T(n) = \Theta(n^2)$
(c) $T(n) = \Theta(n)$
(d) $T(n) = \Theta(n)$
(e) $T(n) = \Theta(n)$

■

Exercise 312.

Select with groups of 13 or may be 3. In worst-case linear time selection algorithm Select(A,n,i) as described in Subject 7 and also a problem in the textbook, we used in class groups of 5 keys. If instead we used groups of 13 what changes in the derivation for the running time of the algorithm? Explain and justify your answer.

Oops typo.. What if we picked group of 3 instead?

And by now you know 2, 4, 6, 8, 10, etc are not suitable numbers. They are ... 'odd' numbers, even if they are even, i.e. not odd!

Proof. Won't be able to prove a suitable upper bound for size of L or R. ■

7.2 Selection algorithm applications

Exercise 313.

Google ranks web-pages with an arbitrary numerical value. One cannot make any assumptions about the range of the rank. Suppose that a query returns n pages (n is part of the input) and information about rank is stored for the i-th page in $A[i]$. We are interested in determining the top 20%, the bottom 40% and the middle 40% of all pages. For that we need to determine two values m and M, the values for which 40% of the pages have ranks less than m, 20% more than M and those whose rank is between m and M. Give an efficient algorithm (whose performance cannot be improved upon) that solves this problem, i.e. it finds m, and M given n and A.

Proof. Selection. ■

Exercise 314.

Close $2k+1$ keys. Let A be an array of n arbitrarily ordered keys that are comparable. Let $k \leq n/2$ be an input parameter. We would like to find $2k+1$ keys of A whose median is the same as the median of A itself. Give an algorithm for `Find(A,n,k)` whose worst-case running time is $o(n)$ (little-oh). (You just need to find some set of $2k+1$ keys; if there are many candidates for such sets, return just one of them.)

Proof. Time efficient means skip sorting. Skip sorting mean linear-time. Linear-time means selection. Proof by example. Say the keys are the integer from 1 to 20. Median is 10. For a set of $2k+1=9$ keys that has median also 10. I'll pick 4 smaller than 10, 4 larger than 10, and of course 10!. I can pick the 4 smaller to be $1,2,3,4$ or $1,7,8,6$, or you guessed it, didn't you? ■

Exercise 315.

There are two types of NJIT professors: `nice` that give 100% of their students an `A` and `mean` that gives 80% of their students an `A` and 20% a `B+`. You need to determine whether professor `I.M.Cute` is `nice` or `mean`. Towards this you have access to his grading this past semester in the form of array $A[1...n]$, $n > 25$. An entry $A[i]$, $1 \leq i \leq n$, can be an `A` or a `B+`. However reading entry $A[i]$ for some arbitrary i, is going to cost you \$1. If your budget is only \$20, can you determine whether `I.M.Cute` is `nice` or `mean` with 90% or more confidence and at a cost of no more than \$20? Explain.

Disclaimer. Do not take seriously the definition of a `mean` NJIT Professor :-) I am neither `nice` nor `mean`!

Proof. That's the Google problem! ■

Exercise 316.

How could you modify quicksort (give pseudocode or describe convincingly the changes) so that its worst-case performance is $O(n \lg n)$? Describe your changes, analyze the worst-case running time of the modified quicksort and show that it is indeed $O(n \lg n)$ in the worst-case.

Proof. Pick the splitter (pivot) key to be the median of the keys of the input participating in the recursive call. Finding the median requires linear time (though with a very large constant). However it guarantees a 50-50 split (plus or minus one key). Thus the recurrence for the running time of this modified quick-sort will be (plus or minus one key): $T(n) = 2T(n/2) + Cn$, where C is a big constant. The asymptotic solution is $\Theta(n \lg n)$, which is acceptable and asymptotically faster than $\Theta(n^2)$ in the worst-case. Pseudocode-wise we can just change line 1 of Partition slightly.

```
OLD CODE
1.  splitter= A[r];
NEW CODE
    // DSelect 'returns' the index of the statistic not the statistic itself
1a. splitter-index=DSelect(A,floor((r-l+1)/2),l,r);
1b. swap(A[r],A[splitter-index]);
1c. splitter=A[r];
```

■

Exercise 317.

For a set of $n = 2^m$ unordered keys we are interested in finding the 1-st, 2-nd, 4-th, ..., 2^{m-2}-th, 2^{m-1}-th order statistic. Give an algorithm whose running time is $O(n)$ in the worst case that finds all m statistics. **Hint: Be careful. If you use Select m times on the set of n keys, it takes $O(n)$ per statistic for a total of $O(nm) = O(n \lg n)$, which is too much.**

Proof.

```
   FindAllStatistics(A[1..n],n,m)
1. S= A[1..n];
2. for (i=m-1 ; i >=0 ; i--) {
3.    m = Median(S);
4.    Sleft= { all x of S  such that x <=m };
5.    Output('2**i-th statistic is  m');
6.    S=Sleft;
7. }
```

Start from the right statistic (the rightmost one). Solve the first selection problem. Filter out the unneeded keys and continue with the next problem and so on. In the beginning of the for loop, $S = A$.
An inductive argument proves that in the i-th iteration S contains the $2 \cdot 2^i$ smallest keys of A, and therefore median m of S is the 2^i-th order statistic of A. Lines 4-6 prepare S for the following iteration by maintaining in S the 2^i smallest keys of A.
In the i-th iteration, S contains the $2 \cdot 2^i$ smallest keys of set A. We then find the median of S which is m and identify the 2^i smallest keys of S, i.e. the 2^i smallest keys of A itself. This also means that m is the 2^i-th order statistic of A. These keys are temporarily stored in Sleft and then transferred (copied) into S before the next iteration.
Analysis of the Algorithm. Step 1 takes $O(n)$ steps for copying n keys. The body of the loop is executed m times, but the running time each time is different. In the i-th execution, S has size $2 \cdot 2^i$, and therefore median finding takes worst case $O(2^i)$ time using Select, Step 4 takes the same amount of time as well, and so is Step 6; Step 5 takes $O(1)$ time. Therefore the i-th execution of the loops takes time $O(2^i)$. Therefore as $n = 2^m$, we have that

$$\sum_{i=0}^{m-1} O(2^i) = O(\sum_{i=0}^{m-1} 2^i) = O(2^m - 1/(2-1)) = O(2^m) = O(n).$$

■

Exercise 318.

For n distinct numbers $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the **weighted median** is the key x_k satisfying

$$\sum_{x_i<x_k} w_i \leq 1/2, \qquad \sum_{x_i>x_k} w_i \leq 1/2$$

i. Argue that the median of $x_1, x_2, \ldots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, \ldots n$.

ii. Show how to compute the weighted median of n keys in $O(n \lg n)$ worst-case time.

Proof. (i) Let the median of the n keys be M. Then M is the $\lfloor (n+1)/2 \rfloor$ smallest of the n keys. If all keys have the same weight the $\lfloor (n+1)/2 \rfloor - 1$ keys smaller than the median have weight less than $1/2$, as $\lfloor (n+1)/2 \rfloor - 1 \leq (n+1)/2 - 1 = (n-1)/2$ and $(n-1)/2 \cdot 1/n < 1/2$. Similarly the keys greater than the median have total weight less than or equal to $1/2$ as well. (One easy way to prove it is to consider two cases.)
Case 1. $n = 2k+1$ i.e. n is odd. Then the number of keys greater than M is k for a total weight of $k/(2k+1) < 1/2$.
Case 2. $n = 2k$ i.e. n is even. Then the number of keys greater than M is also k for a total weight of $k/(2k) = 1/2$.
)
(ii) The algorithm for finding M is given below.

```
   WM( A[1..n])
   //A[i].key is the key A[i].weight is the weight of key A[i].key
1.  Y = MergeSort(A,n);
2.  Z = empty-set; // Z will contain keys less than weighted median M
3.  sum = 0;       //stores weight of Z
4.  for (i=1;i<=n;i++){
5.     Z = Z U { Y[i] }
6.     sum = sum + Y[i].weight;
7.     if sum < 1/2 continue;
8.       else {
9.        Z = Z - { Y[i] };
10.       report Y[i] as weighted median
11.       exit;
12.      }
13. }
```

The time for step 1 is $O(n \lg n)$, for 2-3 is $O(1)$, and for the loop of lines 5-13 is $O(n)$. Briefly, we sort the keys in increasing order and if for the first $i-1$ keys the total weight is $< 1/2$ but the weight of the first i keys is greater than or equal to $1/2$, this means that the i-th sorted key is M and it is reported in line 10. A better algorithm is as follows. Find median M in $O(n)$ time. It splits the keys into two sets. If the weighted median of each set is $< 1/2$, the M is also the weighted median. Otherwise the weighted median is in the set $> 1/2$. In that case recur on the side with $> 1/2$ that includes the median M, and adjust its weight to include the weights of the excluded keys. Running time total is thus $O(n + n/2 + n/4 + \ldots) = O(2n) = O(n)$. ■

Exercise 319.

Find the median of two sorted sequences X and Y. To make things easier assume they are of the same length n, and n is a power of two. Those two assumptions can be lifted, but they make things easier.

Proof.
First we deal with the basis of the recursion, or the simple case. If the two sequences have one key only, the median is the smallest of the two or either (in case of equality). Otherwise we proceed as follows. Compare first the medians of X and Y. Let $x = median(X)$ and $y = median(Y)$, and xx and yy the indices of the medians (initially xx=yy = floor((n+1)/2)-1.) In other words $x = median(X) = X[xx]$. Let mX and MX be 0 and n-1 respectively. Let mY and MY be 0 and n-1 respectively.
Case 1. If $x == y$ return x, and we are done.
If $x > y$, then recur on X[mx:xx] and Y[yy+1:MY]
If $x < y$, then recur on X[xx+1:MX] and Y[mY:yy]

```
   Medianof2Seq(X, Y, n, n) {  // X[0..n-1] , Y[0..n-1]
0.   if (n==1) return(X[0]<= Y[0]? X[0]: Y[0]);
1.   xx = yy = floor((n+1)/2) - 1;
2.   mX=mY=0; MX=MY=n-1;
3.   x= X[xx]; y= Y[yy]; // x= Median(X); y=Median(Y)
4.   if (x == y)
5.     return(xx);
6.   else {
7.     if (x > y ) {
8.       Medianof2Seq(X[mx:xx],Y[yy+1:MY),n/2,n/2);
9.     }
10.    else {
11.      Medianof2Seq(X[xx+1:MX],Y[mY:yy),n/2,n/2);
12.    }
13.  }
14.}
```

It is clear that the running time is $O(\lg n)$. Details re proof of correctness are left as an exercise. An example follows.

```
X:  1  3  5 10 12 14 16 18    mX=0 MX=7 xx=3  x=10
Y:  2  4  6  8  9 10 11 99    mY=0 MY=7 yy=3  y= 8

  x=10 > y= 8 line  8 applies
X:  1  3  5 10                          new   x=3
Y:  9 10 11 99                          new   y=10

  x=3  < y=10 line 11 applies
X:  5 10                                new   x=5
Y:  9 10                                new   y=9

  x=5  < y=9  line 11 applies
X: 10
Y:  9
              line 1  applies      median = 9
```

■

Exercise 320.

Given a black box worst-case linear time median function, give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

Proof. Let's call the black box function $B(X,n)$ that if given a set X of size n returns the median of X. Let the arbitrary statistic we are interested in finding be k. The idea behind the solution is simple. We are going to use a binary search motivated procedure to find the k statistic. Binary search searches the middle key of a sorted set (which happens to be the median). As X in our case is not sorted we call B(X,n) to find the median. If $k = m$, where $m = \lfloor (n+1)/2 \rfloor$ then we are done as the k statistic is the median which is returned by $B(X,n)$. Otherwise we split X into two sets X1 and X2 around the median of X and decide which one contains the k statistic (depending on whether k is less than or greater than m). We repeat recursively the same method on either X1 or X2. In case X2 is searched, we are then interested in finding not the k statistic of X2 but the $k-m$ statistic which is the k statistic of X.

```
 FindStatWithBinSearch(X,k,n)
// find k statistic in X of size n
1. begin
2.    if k = floor((n+1)/2) then return B(X,n)
3.     else
4.      if k < floor((n+1)/2) then {
5.         temp=B(X,n)
6.         collect in X1 all keys of X less than temp
7.         let m=floor((n+1)/2)
8.         FindStatWithBinSearch(X1,k,m-1)
9.       }else{
10.        temp=B(X,n)
11.        collect in X2 all keys of X greater than temp
12.        let m=floor((n+1)/2)
13.        FindStatWithBinSearch(X2,k-m,n-m)
14.      }
15.end
```

Let $T(n)$ be the time for finding the k statistic of n keys using the previously defined procedure. Sets X1 and X2 are of about the same size and each one is of size at most $n/2$.
Step 2 takes $O(n)$ worst case time (median finding, by assumption on the properties of the black box) and so do Steps 5 and 11. Step 6 or 11 takes time $O(n)$ and 7 or 12 $O(1)$. Step 8 or 13 takes time $T(n/2)$.

Note that either step 1 is executed, or steps 5-8, or steps 10-13. Therefore the worst case running time $T(n)$ is given by the following recurrence.

$$T(n) = T(n/2) + O(n)$$

The solution to this recurrence is $T(n) = O(n)$ as required (solution by any of the three known methods). ■

Exercise 321.

The input is an array A of n keys in no particular order. We are interested in determinining whether there exists a key x in A that appears more than $n/3$ times in A. Give a worst-case linear-time algorithm that determines whether such a key exist s or not. If such a key exists, you return the key otherwise you return "NO KEY". Give the algorithm in pseudocode, explain why it works, and analyze (correctly) its worst-case running time.

```
   FindKey(A,n,n/3)
1. a2= Select(A,n,ceiling(n/3));
2. a3= Select(A,n,2ceiling(n/3));
3. f2=Find-Frequency (a2,A,n);
4. f3=Find-Frequency (a3,A,n);
5. if (f2 > n/3) return(a2);
6. if (f3 > n/3) return(a3);
7. return(NO-KEY);
  Find-Frequency(key,A,n)
1.int count=0;
2.for(i=1;i<=n;i++)
3. if (A[i]== key) count++;
4 return(count);
```

Example.

				a2		a3				
n=8	Indices :	1	2	3	4	5	6	7	8	n/3 = 2
3keys	:	x	x	x						Case 1
			x	x	x					Case 2
				x	x	x				Case 3
					x	x	x			Case 4
						x	x	x		Case 5
							x	x	x	Case 6

ceil(n/3) times replicated

Proof. We provide some intuition about the solution. If you have difficult understanding the fifth smallest of $< 1,1,1,1,1,1 >$, read the Subject. Consider the keys in A in sorted order; we are not going to sort but say someone did it for us for free! If a key in A appears more than $n/3$ times, then in the sorted sequence of these keys a block of $n/3$ such consecutive values or more will span the $\lceil n/3 \rceil$ and $2\lceil n/3 \rceil$ positions (we assume a Pascal indexing). Call the keys in those index positions $a2$ and $a3$ respectively. Thus if a key appears more than $n/3$ times it will be $a2$ or $a3$ or possibly both. Our first action is thus to determine the $\lceil n/3 \rceil$-th and $2\lceil n/3 \rceil$-th order statistic of the (unordered) A i.e. determine $a2$, $a3$. These two keys that are found in lines 1 and 2 of FindKey. It suffices to find these two keys, there is not need to sort all of A to find those keys! So after computing these two keys, in lines 3 and 4 we determine how often these two keys appear in the unsorted A by scanning the input sequence left-to-right and determining the frequency of $a2$ and $a3$ respectively. If one of them appears more than $n/3$ times such a value of $a2$ or $a3$ is returned through lines 5 and 6. If neither $a2$ nor $a3$ appears more than $n/3$ times, this means there is no key that appears that often. Line 7 returns NO-KEY then. The running time of lines 1 and 2 is $O(n)$ if we ran the worst-case-linear-time selection algorithm, the running time of lines 3 and 4 is obviously $O(n)$ (observe the code of Find-Frequency) and lines 5,6,7 take $O(1)$ time. Overall $O(n)$ is the asymptotic worst-case running time of FindKey. We give an example with $n = 8$ and thus $n/8 \approx 2.6$ (i.e. determining whether a keys exists more than twice, i.e. 3 times or more). If such a key exists, it is either the $n/3$-rd order statistic (i.e. $a2$) or the $2n/3$-rd order statistic (i.e. $a3$). ■

Exercise 322.

Give an $O(n)$ worst case running time algorithm that, given a set S of n distinct numbers and a positive integer k, $k \leq n$, determines the k numbers in S that are closest to the median of S.

Proof. We first find the median of the n number stored originally in A. This takes $O(n)$ worst-case running time and is performed in line 1. So far we have found the median m. Next we create a new array that stores the n numbers in its value field. Attached to each such value is a key field. The key field indicates the distance between $A[i]$ and the median. It does not matter whether $A[i]$ is less or greater than m, and thus we take absolute values to determine the distance. This creation of a new array B takes place in lines 2-4. In line 3, the $A[]$ values are copied and the distance to the median m is computed in line 4. Total running time for lines 2-5 is also $O(n)$.
In line 6, we select the k-th smallest key of B, i.e. find the k-th statistic of B by using as keys the distance/key values previously created NOT the original numbers. This has the effect of finding the k-th number closest to m. We record its distance to m in s. This takes $O(n)$ time.
In lines 7-9 we use s to find the k numbers that are close to the median m by checking in turn which of the n numbers (value fields) are at distance s or closer to m. This also takes $O(n)$ time. The overall time complexity is also $O(n)$.
Note. The algorithm above assumes that all keys and also distances are distinct. If they are not then we can modify line 8 to output only the first k matches. We just need to add a counter in line 8. As soon as k results are output the loop is terminated.

```
  Closest(A[1..n],k)
1. m= Select(A[1..n],n,floor((n+1)/2))       //Find the median m of A[1...n]
2. for(i=1;i<=n;i++) {                       //Find the distance of A[i] from m
3.  B[i].value = A[i];                       //(distance is absolute value of
difference
)
4.  B[i].key   = |A[i]-m|; // |x| is the absolute value of x.
5. }
6. s= Select(B[1..n].key,n,k,);
   // select the k-th statistic use .key values for the selection

7. for(i=1;i<=n;i++) {        // Find the k keys with smallest distance from median m
8.  if B[i].key <= s          // i.e. the keys with distance at most s.
      print(B[i].value);      // The values of these k keys is the ANSWER.
9  }
```

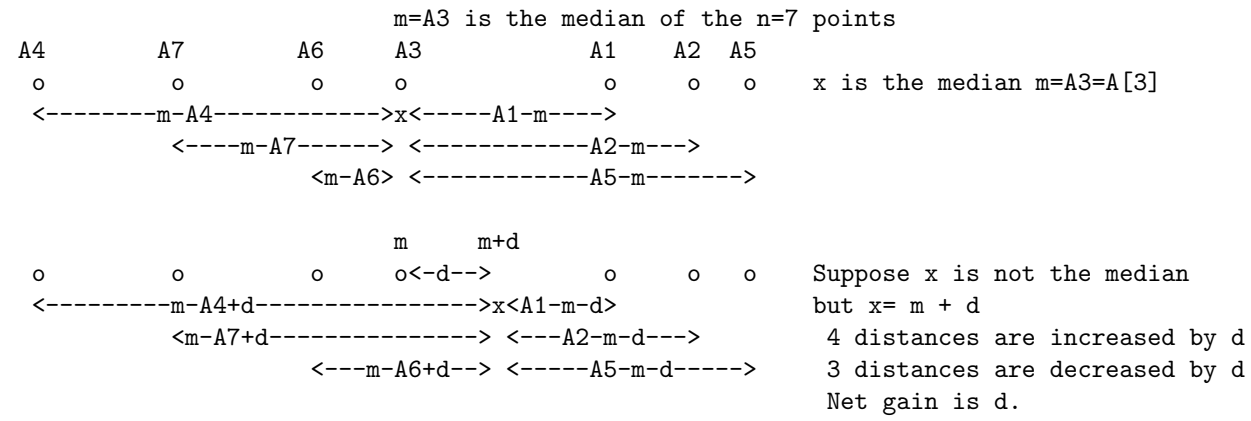
■

Exercise 323.

You are given n numbers in the form of array $A[0..n-1]$. Each number can be an integer in an arbitrary range or similarly, a real; no generic assumptions thus can be made about the range and type of these numbers. You are asked to give a linear-time algorithm that finds the number x that minimizes $\sum_{i=0}^{n-1}|A[i]-x|$ where $|y|$ is the absolute value of y. The output x can be one of the input values, or can be computed from a subset of the input values. Justify the correctness of your algorithm and analyze its worst-case running time to verify the linear-time claim.

Proof. **Case 1.** $n=2k+1$.

"Order" the numbers. Let x be the median m of the n numbers. We claim that $D(x)=\sum_{i=0}^{n-1}|A[i]-x|$ is then minimized for $x=m$, i.e. $D(m)$ is minimal. Why?



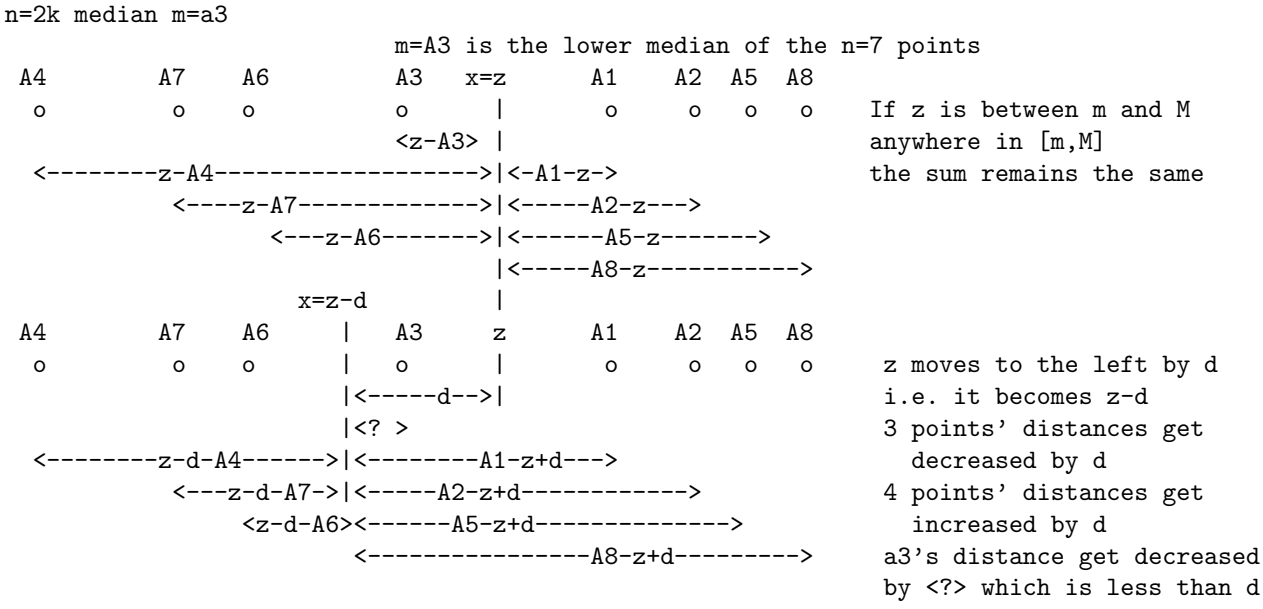
Suppose that x is not the median but a value $x=m+d$ to the right of the median by a discplacement d. Then the distance $|A[i]-x|$ of (at least) $k+1$ points (including the median) has been increased by d and the distance of (at most) k points has been decreased by d. The end result is that the sum $\sum_{i=0}^{n-1}|A[i]-x|$ has been increased by at least d, i.e. $D(m+d)\geq D(m)+d$. If x is on the left of the median m we have a similar situation and an increase by at least d of the sum (i.e. $D(m-d)\geq D(m)+d$. Therefore the median m is the optimal x because it gives the lowest value for the sum.

We do not need to sort the numbers to find the median, i.e. x. We just use the linear-time Select algorithm. This takes $\Theta(n)$.

Case 2. $n=2k$. In this case we have two medians, the lower m and the upper M median. Let us choose $x=z$ to be between m and M i.e. $m\leq z\leq M$. Let the sum for this $x=z$ be $D=D(z)$. If we move x by d to the left (i.e. $x=z-d$) but still in that interval or by d to the right (i.e. $x=z+d$) but still in that interval k points' displacement is increased by d and k points' displacement is reduced by d. So the net-gain/loss is 0 and the sum remains equal to D, i.e. $D(z+d)=D(z-d)=D=D(z)$ as long as $m\leq z,z+d,z-d\leq M$. The d is an arbitrarily chosen value.

However if x is moved to the left of m or to the right of M then the sum gets increased. For example if x moves to the left by d and falls on the left of m, for $k-1$ points or less the displacement gets smaller by d but for k points or more the displacement gets bigger by d and for m the displacement gets decreased by less than d because x moves from the right to the left of x. This is shown in the figure.

Therefore for Case 2, the optimal x is any value z such that $m \leq z \leq M$. We could report $x = m$ just as we did in Case 1, or use the upper median or any other value in-between.



Exercise 324.

The input is a sorted sequence of numbers $A[0..n-1]$ (positive, negative or zero, real or integer of an arbitrarily large range), and an integer x. Give an efficient algorithm that determines the minimum valu e attained by $|A[i]+x|$, where i is such that $0 \leq i < n$. Note that $|x|$ is the absolute value of x. "Efficient algorithm" means an algorithm whose worst-case performance can not be improved upon. So if you suggest an algorithm whose running time is $O(n)$ but there exists an algorithm whose running time is $o(n)$ your algorithm is not efficient and you will get 0 points.

Proof.
We use binary search on the sorted array A. We search for $-x$. If there is a k such that $A[k] = -x$, this means that $A[k]+x = 0$ i.e. we have found our i to be the k returned by BinarySearch. If however a $-x$ is not in the array we have one of three cases at the completion of BinarySearch:
(a) Case 1: $-x < A[0]$. Then the closest A to $-x$ is $A[0]$. We return $A[0]+x$. If $-x < A[0]$ it also means that $-x < A[0] < A[1] < \ldots$. Therefore $x+A[0] < x+A[j]$, $j > 0$, i.e. $x+A[0]$ is the smallest possible. Because the sum is positive, it is also equal to its absolute value.
(b) Case 2: $A[n-1] < -x$. Since then $A[0] < \ldots < A[n-1] < -x$, we have that $-x-A[j] > -x-A[n-1]$, $j < n-1$, i.e. $-x-A[n-1]$ is the smallest possible. But $|x+A[n-1]| = -x-A[n-1]$ given that $A[n-1] < -x$ implies $A[n-1]+x < 0$ (is negative). The closest A to $-x$ is $A[n-1]$ and the value returned is $-x-A[n-1]$.
(c) Case 3: $A[j] < -x < A[j+1]$. Then $-x$ is closer to $A[j]$ or $A[j+1]$ than anything else since everything else is smaller that $A[j]$ or greater than $A[j+1]$. We determine in constant time whether $-x-A[j]$ or $A[j+1]+x$ (both distances are positive) is the smallest one. The smallest is the one returned and is the smallest $|A[k]+x|$. The running time in any of the three cases is dominated by the BinarySearch time. Therefore we can determine the smallest $|A[k]+x|$ in $O(\lg n)$ time. ■

7.3 Some challenging problems

Exercise 325.

We want to find the t largest of n keys in sorted order using a comparison-based algorithm.
Find the algorithm that implements each one of the following methods with the best asymptotic worst-case performance, analyze its worst-case running time, and express it in terms of both n and t.
(a) Sort the n keys, and list the t largest of them.
(b) Build a Max-Heap out of the n keys, and call ExtractMax t times.

Proof.
(a) If we sort n keys using heapsort this takes $O(n \lg n)$ times. We can find the t largest keys in the sorted output $A[1..n]$ by just printing the keys of the subarray $A[n-t+1..n]$ in $O(i)$ time. Total running time is $O(n \lg n + t)$ where $+t$ counts for printing the t largest keys.
(b) Build a MaxHeap of the n keys in $O(n)$ time. Call then ExtractMax t times to print the t largest keys requires $O(\lg(n-t+1)) = O(\lg n)$ time for the t-th largest key, i.e. a total of $O(t \lg n)$. The total running time is now $O(n + t \lg n)$.
There is a better method. Run selection i.e. DetSelect(A,n,n-t+1) in other words with $k = n-t+1$. This is going to return the $n+t+1$-st largest key. Call it K. Scan A and find the t keys $\geq K$ and sort them using HeapSort. This would take $O(n + t \lg t)$, including the cost of deterministic linear time selection. We encounter this problem earlier. If $t = o(n)$, the non-HeapSort algorithm is better than the HeapSort-based one. The Selection-based algorithm is better than the other two for $t = o(\lg n)$, and equivalent to the performance of non-HeapSort algorithm otherwise. ■

Exercise 326.

An MIT exam question rephrased and simplified.
NJayIT, a misspelled Institution that shall remain misspelled, discovered n sources of a potentially dangerous liquid substance (water). Authorities suspect that t of those n sources are unhealthy (i.e. poisonous), where $t \ll n$. Let the sources be $\{s_1, s_2, \ldots, s_n\}$. NJayIT wants eventually to detect which t of those sources are poisonous and which are not! NJayIT asks some students for assistance. Any student subject can drink only one cup of water; that cup of water can contain water of any (one or more) of the n sources. If any one (or more) of the sources in the cup of water is poisonous, the subject gets sick and we detect the poisoning (but not necessarily the source of poisoning). If the subject stays healthy, subject is a nervous wreck and cannot be used further in this study.

The intent of NJayIT is to detect the t poisonous sources and thus the $n-t$ non-poisonous ones as well. A very simple test is to ask n subjects to test one different source of water each (say subject i tastes source s_i). t of them will get "sick" and $n-t$ will be "nervous wrecks". That's too bad public relations-wise. We want to minimize the number of "sick" and "nervous wrecks" (the combination, that is) to something much less than n.

Our job is to analyze the complexity of this problem. **Show formally** that any algorithm for poison-source detection requires at least $t \lg(n/t)$ tests/subjects (aka cup of water tastings). Your job is to establish the complexity of the problem only!

Proof.
Think decision trees. ■

Exercise 327.

Suppose there is a comparison-based data structure S where insertion e.g. Insert(S,k) takes $O(1)$ per key k. Moreover the data structure supports operation FindMedian(S) that also takes $\Theta(1)$. Show that on the comparison-model you can then sort in $\Theta(n)$ time the keys of an input array $A[0..n-1]$ (whose keys are comparable).
Deduce then that such an S is NOT possible.

Proof.
Suppose S is possible. Find the minimum m and the maximum M or A in $\Theta(n)$ time. Then form L with n keys smaller than m, and form R with $2n$ keys larger than M. Starting with an empty S, insert first all elements of A, and L. This will take $\Theta(n)$ time. Insert then $R[0]$ in $\Theta(1)$ doing Insert(S,R[0]), and invoke FindMedian(S). There are n keys less than m (those of L), n keys greater than m. Thus FindMedian(S) returns m, the smallest key. This is the first key of the output that will form Sort(A).
In the next step we do Insert(S,R[1]), Insert(S,R[2]). A FindMedian(S) will find the second smallest key of A, and we output it. This is because there are $n+1$ keys smaller than the second smallest (the keys of L plus m), and $n+1$ larger ($n-2$ keys of A plus the three keys of R). Inserting every time two more keys of R and calling FindMedian, we eventually sort A. Total time is $3 \times \Theta(n) = \Theta(n)$.
Comparison-based sorting yields a $\Omega(n \lg n)$ lower-bound. This means that the data structure S is a pipe dream!

■

Exercise 328.

Build a data structure S that can support median queries in constant time and median deletions in $O(\lg n)$ time. The following operations will be supported.
A. Operation Build(A,n) that will build data structure S from the comparable elements of an array A of length n in bulk; the created data structure S will have the previously mentioned properties and be constructed from A in $\Theta(n)$ time. Thus $S = Build(A,n)$ builds S, as needed,
B. operation Start(k) that will build data structure S from a single element k (think of $A = \{k\}$); thus $S = Start(k)$ builds S, as needed in constant time,
C. operation findMedian(S) finds the median of S in constant time,

D. operation insert(S,k) inserts k into S in $O(\lg n$ time,

E. operation deleteMedian(S) deletes the median of S in $O(\lg n)$ time.

Show how one can implement such a data structure S by also implementing operations A through E within the noted time bounds.

Proof.
We provide pseudocode for the variety of operations. The data structure S will be a triplet

$$S = \langle m, L, R \rangle$$

consiting of the median m of S, a MaxHeap L consisting of the elements of $S \leq m$, and a MinHeap R consisting of the elements of $S \geq m$. Moreover $||L| - |R|| \leq 1$, where $L.length = |L|$ and $R.length = |R|$.
Operation Build(A,n) is implemented as follows in $\Theta(n)$ time.

```
Build(A,n)  {  // Assumes all distinct
  m = FindMedian(A,n);
  A = RemoveMedian(A,n,m);
  L = Filter(A, < m);
  R = Filter(A, > m);
  BuildMaxHeap(L);
  BuildMinHeap(U);
  S.m = m;
  S.L = L;
  S.R = R;
  return(S);
}
```

Operation Start(k) is implemented as follows in $\Theta(1)$ time.

```
   start(x) {
   S.m = x;
   S.L = empty;
   S.R = empty;
   return(S)
}
```

Operation findMedian(S) is implemented as follows in $\Theta(1)$ time.

```
findmedian(S) {
   return(S.m);
 }
```

Operation insert(S,k) inserts k into S in $O(\lg n$ time, and is implemented as follows.

```
insert(S,x)
  cmp = compare(x,findmedian(S));
  if (cmp == 0) { // equal
    if L.length < R.length  insertMaxHeap(L,x);
       else             insertMinHeap(R,x);
  }
  if (cmp >  0) {  // x > findmedian(S)
     insertMinHeap(R,x);
     insertMaxHeap(L,findmedian(S));
     S.m = ExtractMinHeap(R);
  }
  else {           // x < findmedian(S)
     if L.length < R.length {
       insertMaxHeap(L,x);
       insertMinHeap(R,findmedian(S));
       S.m = ExtractMaxHeap(L);
     }
     else {
       insertMaxHeap(L,x);
       insertMaxHeap(R,findmedian(S));
       S.m = ExtractMaxHeap(L);
     }

  }
```

Operation deleteMedian(S) deletes the median of S in $O(\lg n)$ time, and is implemented as

```
deleteM(S) {
  temp = S.m;
  if (L.length < R.length ) {
     S.m = ExtractMin(R);
     return(temp);
  }
  else {
     S.m = ExtractMax(L);
     return(temp);
  }
```

■

Exercise 329.

The k-th quantiles of an n-element set are the $k-1$-st order statistics that divide the sorted set into k equal-sized sets (to within 1). To make things easy let n be a multiple of k. The k-th quantiles of an n-element sorted array $B[1..n]$ are: $B[n/k], B[2n/k], \ldots B[(k-1)n/k]$.
We have as input an array $A[1..n]$ of n keys that are arbitrarily (i.e. unsorted) ordered. You may assume the keys are comparable and distinct and n is a multiple of k. Array $C[1..k-1]$ of length $k-1$ would contain the k-th quantiles of A.
Give an efficient algorithm that output C the k-th quantiles of A. Show its corectness and analyze its running time.

Proof.
A high-level description follows. At the top level we need to find $k-1$ values that are the k-th quantiles of A. These are the $(i \cdot n/k)$-th order statistics of A, for $i = 1, \ldots, k-1$. We will split A into two halves A_l, A_r that will contain the first $i = 1, \ldots, \lfloor \frac{(k-1)+1}{2} \rfloor$ $(i \cdot n/k)$-th order statistics of A, and the remaining $i = \lfloor \frac{(k-1)+1}{2} + 1 \rfloor, \ldots k-1$, $(i \cdot n/k)$-th order statistics of A respectively. This requires calling DetSelect(A,n,m), where $m = (\lfloor \frac{(k-1)+1}{2} \rfloor) n/k$. This requires $\Theta(n)$ time. Subsequently we split a problem on an input of n keys into two smaller subproblems each containing roughly $n/2$ keys (plus or minus n/k), and for which roughly $k/2$-th quantiles are to be computed that will be mapped to the k-th quantiles of the original set A. The depth of the recursion is about $\lg k$. The work per recursion level is $O(n)$. Therefore the total amoutn of work is $O(n \cdot \lg k)$. ■

Exercise 330.

You are given a sequence of n unordered keys and an arbitrary integer k such that $0 < k < n$. Give an algorithm that finds the k smallest of the n input keys and prints these k keys in an arbitrary order in worst case $\Theta(n)$ time . "Give" means describe the algorithm in pseudocode (or English or Java/C/C++), show that it works as claimed, and analyze its worst case running time.

Proof.

We use Selection (deterministic algorithm) to find the k-th order statistic `key`. (k-th smallest of n keys of input A). Then we scan input A and isolate the keys smaller than or equal to `key`. Running time is $\Theta(n)$ that includes both the time of deterministic Select (line 1) and the running time of the filtering process (lines 2-5).

```
 FindSmallestandPrint(A,k,n)
1. key = DetSelect(A,n,k);  // A[1..n]
2. for(i=1;i<=n;i++) {
3.  if A[i] <= key
4.    print A[i]
5. }
```

■

Exercise 331.

The input is an array A of n keys in no particular order. We are interested in determinining whether there exists a key x in A that appears more than $n/3$ times in A. Give a worst-case linear-time algorithm that determines whether such a key exists or not. If such a key exists, you return the key otherwise you return "NO KEY". Give the algorithm in pseudocode, explain why it works, and analyze (correctly) its worst-case running time.

Proof.

Although we are not going to sort, consider the keys in A in sorted order. If a key in A appears more than $n/3$ times, then in the sorted sequence of these keys a block of $n/3$ such consecutive values or more will span the $\lceil n/3 \rceil$ and $2\lceil n/3 \rceil$ positions. Call the keys in those positions $a2$ and $a3$ respectively. Thus if a key appears more than $n/3$ times it will be $a2$ or $a3$ or possibly both. Our first action is thus to determine the $\lceil n/3 \rceil$-th and $2\lceil n/3 \rceil$-th smallest keys of the (unordered) A ($a2$, $a3$) and it is these keys that are computed in lines 1 and 2 of FindKey. We only need to find these two keys, not sort the whole of A. So after computing these two keys, in lines 3 and 4 we determine how often these two keys appear in the unsorted A by scanning the input sequence left-to-right and determining the frequency of $a2$ and $a3$ respectively. If one of them appear more than $n/3$ times such a value of $a2$ or $a3$ is returned throught lines 5 and 6. If neither $a2$ nor $a3$ appears more than $n/3$ times, this means there is no key that appears that often. Line 7 returns NO-KEY then. The running time of lines 1 and 2 is $O(n)$ if we ran the worst-case-linear-time selection algorithm, the running time of lines 3 and 4 is obviously $O(n)$ (observe the code of Find-Frequency) and lines 5,6,7 take $O(1)$ time. Overall $O(n)$ is the asymptotic worst-case running time of FindKey. We give an example with $n = 8$ and thus $n/8 = 2$ (i.e. determining whether a keys exists more than twice, i.e. 3 times or more). In the sorted input if such a key exists then the position of the first three occurrences (of the three or more) of the key are going to follow the pattern of one of the six cases depicted. All cases touch the $a2$ and $a3$ i.e. the $\lceil n/3 \rceil$-th and $2\lceil n/3 \rceil$-th smallest of the n keys.

```
   FindKey(A,n,n/3)
1. a2= Select(A,n,ceiling(n/3));
2. a3= Select(A,n,2ceiling(n/3));
3. f2=Find-Frequency (a2,A,n);
4. f3=Find-Frequency (a3,A,n);
5. if (f2 > n/3) return(a2);
6. if (f3 > n/3) return(a3);
7. return(NO-KEY);
  Find-Frequency(key,A,n)
1.int count=0;
2.for(i=1;i<=n;i++)
3. if (A[i]== key) count++;
4 return(count);

Example. Consider the sorted sequence of A i.e. Sort(A)
                          a2       a3
  n=8    Indices :   1  2  3  4  5  6  7  8
  > n/3          :   x  x  x                    Case 1
  means                 x  x  x                 Case 2
  > 8/3 = 2.6              x  x  x              Case 3
  i.e.                        x  x  x           Case 4
  3 times or more                x  x  x        Case 5
                                    x  x  x     Case 6
Sort(A)=[ 10 10 20 20 30 30 30 50 ]
  n=8    Indices :   1  2  3  4  5  6  7  8

Say  A =           [ 50 30 30 10 20 10 20 30]
                          a2       a3
Sort(A)=           [ 10 10 20 20 30 30 30 50]

Either 20 or 30 can appear more than n/3 times in A
It is indeed 30 (appearing exactly 3 times i.e. > n/3=8/3)
```

■

Exercise 332.
Majority revisited. Do it in $O(n)$ time.

Proof.

```
 Majority(A,n)
1. maj-index = DeterministicSelect(A,floor((n+1)/2),n);
   //Returns an index into A of median
2. count=0;
3. for(i=1;i<=n;i++) {
4.  if A[i] == A[maj-index]
5.    count++;
6. }
7. if count > n/2
8.   print A[maj-index]
9. else
10.  print "NO"
```

■

7.4 Review search, sort selection

Exercise 333.

For this problem input size is specified separately for each case and thus `Row-7,Column-1` indicates a problem size of n keys for MergeSort. Find the worst-case running time of every (efficient) algorithm in `Column-1` by using the time bounds in `Column-2`; use only class-introduced algorithms. If multiple answers are possible, use the asymptotically tightest bound. The same answer in `Column-2` can be reused or not used at all. Row 7 is filled for you; you should fill rows 1 through 7. If you see $\lg^t n$, it means $(\lg n)^t$.

	Column-1	Answer	Column-2
1:	HeapSort of n^2 keys	?	A. $O(\lg n)$
2:	QuickSort of n^2 keys	?	B. $O(\sqrt{n})$
3:	HeapSort of n/lgn keys	?	C. $O(n)$
4:	InsertionSort of n^2 sorted keys	?	D. $O(n \lg n)$
5:	SelectionSort of n^2 keys	?	E. $O(n^2)$
6:	MergeSort of n^2 keys	?	F. $O(n^2 \lg n)$
7:	MergeSort of n keys	**D**	G. $O(n^2 \lg^2 n)$
			H. $O(n^4)$

Proof.

	Column-1	Answer	Column-2
1:	HeapSort of n^2 keys	F	A. $O(\lg n)$
2:	QuickSort of n^2 keys	H	B. $O(\sqrt{n})$
3:	HeapSort of n/lgn keys	C	C. $O(n)$
4:	InsertionSort of n^2 sorted keys	E	D. $O(n \lg n)$
5:	SelectionSort of n^2 keys	H	E. $O(n^2)$
6:	MergeSort of n^2 keys	E	F. $O(n^2 \lg n)$
7:	MergeSort of n keys	**D**	G. $O(n^2 \lg^2 n)$
			H. $O(n^4)$

■

Exercise 334.

For this problem input size is specified separately for each case and thus `Row-7,Column-1` indicates a problem size of n keys for MergeSort. Find the worst-case running time of every (efficient) algorithm in `Column-1` by using the time bounds in `Column-2`; use only class-introduced algorithms. If multiple answers are possible, use the asymptotically tightest bound. The same answer in `Column-2` can be reused or not used at all. Row 7 is filled for you; you should fill rows 1 through 7. If you see $\lg^t n$, it means $(\lg n)^t$.

	Column-1	Answer	Column-2
1:	HeapSort of n^2 keys	?	A. $O(\lg n)$
2:	QuickSort of n^2 keys	?	B. $O(\sqrt{n})$
3:	QuickSort of $n \lg n$ keys	?	C. $O(n)$
4:	InsertionSort on n^2 sorted keys	?	D. $O(n \lg n)$
5:	SelectionSort on n^2 keys	?	E. $O(n^2)$
6:	MergeSort of n^2 keys	?	F. $O(n^2 \lg n)$
7:	MergeSort of n keys	**D**	G. $O(n^2 \lg^2 n)$
			H. $O(n^4)$

Proof.

	Column-1	Answer	Column-2
1:	HeapSort of n^2 keys	F	A. $O(\lg n)$
2:	QuickSort of n^2 keys	H	B. $O(\sqrt{n})$
3:	QuickSort of $n \lg n$ keys	G	C. $O(n)$
4:	InsertionSort on n^2 sorted keys	E	D. $O(n \lg n)$
5:	SelectionSort on n^2 keys	H	E. $O(n^2)$
6:	MergeSort of n^2 keys	F	F. $O(n^2 \lg n)$
7:	MergeSort of n keys	**D**	G. $O(n^2 \lg^2 n)$
			H. $O(n^4)$

■

Exercise 335.

For this problem input size is specified separately for each case and thus `Row-7,Column-1` indicates a problem size of n keys for MergeSort. Find the worst-case running time of every (efficient) algorithm in `Column-1` by using the time bounds in `Column-2`; use only class-introduced algorithms. If multiple answers are possible, use the asymptotically tightest bound. The same answer in `Column-2` can be reused or not used at all. Row 7 is filled for you; you should fill rows 1 through 7. If you see $\lg^t n$, it means $(\lg n)^t$.

	`Column-1`	`Answer`	`Column-2`
1:	InsertionSort of $n \lg n$ keys	**?**	A. $O(\lg n)$
2:	QuickSort of $n/\lg n$ keys	**?**	B. $O(\sqrt{n})$
3:	HeapSort of $n \lg n$ keys	**?**	C. $O(n)$
4:	InsertionSort on $\sqrt{n}$ keys	**?**	D. $O(n \lg n)$
5:	MergeSort of $\sqrt{n}$ keys	**?**	E. $O(2^{n-1})$
6:	MergeSort of $n^2 \lg n$ keys	**?**	F. $O(n \lg^2 n)$
7:	MergeSort of n keys SAMPLE	`D`	G. $O(n^2 \lg^2 n)$
			H. $O(\lg(\lg n))$

Proof.

	Column-1	Answer	Column-2
1:	InsertionSort of $n \lg n$ keys	**G**	A. $O(\lg n)$
2:	HeapSort of $n/\lg n$ keys	**C**	B. $O(\sqrt{n})$
3:	QuickSort of $n \lg n$ keys	**G**	C. $O(n)$
4:	InsertionSort on $\sqrt{n}$ keys	**C**	D. $O(n \lg n)$
5:	MergeSort of $\sqrt{n}$ keys	**C**	E. $O(2^{n-1})$
6:	MergeSort of $n^2 \lg n$ keys	**G**	F. $O(n \lg^2 n)$
7:	MergeSort of n keys SAMPLE	D	G. $O(n^2 \lg^2 n)$
			H. $O(\lg(\lg n))$

■

Exercise 336.

For this problem input size is specified separately for each case and thus `Row-7,Column-1` indicates a problem size of n keys for MergeSort. Find the worst-case running time of every (efficient) algorithm in `Column-1` by using the time bounds in `Column-2`; use only class-introduced algorithms. If multiple answers are possible, use the asymptotically tightest bound. The same answer in `Column-2` can be reused or not used at all. Row 7 is filled for you; you should fill rows 1 through 7. If you see $\lg^t n$, it means $(\lg n)^t$.

	`Column-1`	`Answer`	`Column-2`
1:	HeapSort of n^2 keys	?	A. $O(\lg n)$
2:	QuickSort of $\sqrt{n}$ keys	?	B. $O(\sqrt{n})$
3:	Sort n keys in rage $0...n^n-1$	?	C. $O(n)$
4:	InsertionSort of $n\lg n$ sorted keys	?	D. $O(n\lg n)$
5:	HeapSort of $n^2\lg n$ keys	?	E. $O(n^2)$
6:	Sort n keys in range $0..n\lg n-1$	?	F. $O(n^2\lg n)$
7:	MergeSort of n keys	**D**	G. $O(n^2\lg^2 n)$
			H. $O(n^4)$

Proof.

	Column-1	Answer	Column-2
1:	Sort $\sqrt{n}$ keys in range $0..n-1$	B	A. $O(\lg n)$
2:	QuickSort of $\sqrt{n}$ keys	C	B. $O(\sqrt{n})$
3:	Sort n keys in rage $0...n^n-1$	D	C. $O(n)$
4:	Sort $n\lg n$ keys in range $0..n\lg n$	D	D. $O(n\lg n)$
5:	HeapSort of $n^2\lg n$ keys	G	E. $O(n^2)$
6:	Sort n keys in range $0..n\lg n-1$	C	F. $O(n^2\lg n)$
7:	MergeSort of n keys	**D**	G. $O(n^2\lg^2 n)$
			H. $O(n^4)$

■

Exercise 337.

For this problem input size is specified separately for each case and thus `Row-7,Column-1` indicates a problem size of n keys for MergeSort. Find the worst-case running time of every (efficient) algorithm in `Column-1` by using the time bounds in `Column-2`; use only class-introduced algorithms. If multiple answers are possible, use the asymptotically tightest bound. The same answer in `Column-2` can be reused or not used at all. Row 7 is filled for you; you should fill rows 1 through 7. If you see $\lg^t n$, it means $(\lg n)^t$.

	`Column-1`	**Ans**	`Column-2`
1:	BuildMaxHeap on $n^2 \lg n$ keys		A. $O(\lg n)$
2:	BinSearch of $n^2 \lg n$ sorted keys		B. $O(\sqrt{n})$
3:	BinSearch on 2^n sorted keys		C. $O(n)$
4:	Search on a HashTable (open addr.)		D. $O(n \lg n)$
5:	Search on a HashTable (chaining)		E. $O(n^2)$
6:	Find Median of $n \lg n$ keys		F. $O(n^2 \lg n)$
7:	MergeSort of n keys	**D**	G. $O(n^2 \lg^2 n)$
			H. $O(1+a)$ K. $O(n \lg^2 n)$
			L. $O(N)$ M. $O(2^n)$

Proof.

	Column-1	**Ans**	Column-2
1:	BuildMaxHeap on $n^2 \lg n$ keys	F	A. $O(\lg n)$
2:	BinSearch of $n^2 \lg n$ sorted keys	A	B. $O(\sqrt{n})$
3:	BinSearch on 2^n sorted keys	C	C. $O(n)$
4:	Search on a HashTable (open addr.)	L	D. $O(n \lg n)$
5:	Search on a HashTable (chaining)	C	E. $O(n^2)$
6:	Find Median of $n \lg n$ keys	C	F. $O(n^2 \lg n)$
7:	MergeSort of n keys	**D**	G. $O(n^2 \lg^2 n)$
			H. $O(1+a)$ K. $O(n \lg^2 n)$
			L. $O(N)$ M. $O(2^n)$

■

Chapter 8

More on graph searches

8.1 Graphs and their representations

Exercise 338.

TRUE or FALSE? Anything other than a TRUE or FALSE is considered wrong!

1. It is always asymptotically faster to compute the outdeg(u) using an adjacency matrix than an adjacency list representation.
2. It is always asymptotically faster to compute the indeg(u) using an adjacency matrix than an adjacency list representation.
3. It is always asymptotically faster to compute m using an adjacency matrix than an adjacency list representation.
4. It is always asymptotically faster to generate the adjacency list of u using an adjacency matrix than an adjacency list representation.
5. Let $G = (V,E)$ be a simple undirected graph with weights on its edges and with more than three vertices and unique edge weights. Let M be a minimum cost spanning tree of G. Then the path from u to v in M is a shortest path from u to v in G as well.
6. Let $G = (V,E)$ be a simple undirected graph with weights on its edges and with more than three vertices and unique edge weight. Then the heaviest edge in G is in NO minimum-cost spanning tree.
7. In a sparse graph G=(V,E) represented through an adjacency list finding the indegree of vertex 25, where 25 is a vertex of V takes time that is $\Theta(n+m)$.
8. In a dense graph G=(V,E) represented through an adjacency list finding the outdegree of vertex 25, where 25 is a vertex of V takes time that is $\Theta(m)$.
9. In a sparse graph G=(V,E) represented through an adjacency matrix finding the indegree and outdegree of vertex 25, where 25 is a vertex of V takes time that is $\Theta(m)$.
10. On a Graph $G = (V,E)$ with $|V| = n, |E| = m$, represented through an adjacency matrix, the running time of determining the in-degree of a vertex is $\Theta(n)$.
11. On a Graph $G = (V,E)$ with $|V| = n, |E| = m$, represented through an adjacency matrix, the running time of determining the out-degree of vertex 25, $25 \in V$, $\Theta(n)$.
12. On a Graph $G = (V,E)$ with $|V| = n, |E| = m$, represented through an adjacency list, the running time of determining the out-degree of a vertex w is $O(deg(w))$.
13. In a sparse graph G=(V,E) represented through an adjacency list finding the indegree and outdegree of vertex 25, where 25 is a vertex of V takes time that is $\Theta(n+m)$.

Proof. 1. FALSE, 2. TRUE, 3. FALSE, 4. FALSE, 5. FALSE, 6. FALSE, 7. TRUE, 8. FALSE (it is $O(out(25)) = O(n)$ where as dense implies $m = \Theta(n^2)$ or $n = o(m)$), 9. FALSE (it is $\Theta(n)$ and not necessarily $\Theta(m)$, 10. TRUE, 11. TRUE, 12. TRUE, 13. TRUE, ■

8.2 Elementary graph algorithm properties

Exercise 339.

Provide answers to the following question.

(1) On a graph $G = (V,E)$, with n vertices and m edges represented through an adjacency matrix, give the running time of determining the out-degree of one given vertex u.

(2) On a directed graph G=(V,E), with n vertices and m edges represented through an adjacency list A, give the running time of deciding whether the graph is strongly connected.

(3) Give the running time of Kruskal's algorithm on an undirected graph G=(V,E), where n is the number of vertices, m is the number of edges, $m > n$, the graph is represented through an adjacency list A, and in the linked (adjacency) list of A(i) information about the distance of an edge (i,j) is also recorded by way of d(i,j) along with j.

(4) Give the running time of topological sorting on an acyclic connected undirected graph $G = (V,E)$ with n vertices and m edges represented through an adjacency list.

(5) Give the running time of deciding whether a graph is strongly connected $G = (V,E)$ with n vertices and m edges (represented by an adjacency matrix).

(6) Give the running time of deciding whether a graph is strongly connected $G = (V,E)$ with n vertices and m edges (represented by an adjacency list).

(7) Give the running time of Prim's algorithm on an undirected graph G=(V,E), where n is the number of vertices, m is the number of edges, $m > n$, the graph is represented through an adjacency list A, where $A(i)$ is the adjaceny list of vertex i and in A information in element j of the linked list not only confirms the existence of the edge (i, j) but also records in the form of $d(i, j)$ the distance of edge (i, j).

Proof. 1. $T(n,m) = \Theta(n)$, 2. $T(n,m) = \Theta(n+m)$, 3. $T(n,m) = O(m\lg m + n\lg n + m) = O(m\lg m)$, 4. $T(n,m) = \Theta(n)$ (it is a tree), 5. $T(n,m) = \Theta(n^2)$, 6. $T(n,m) = \Theta(n+m)$. 7. $T(n,m) = \Theta(n\lg n + m\lg n)$. If $m > n$, it can be simplified accordingly. ■

Exercise 340.

An alternative method to compute the topological sorting of a directed acyclic graph is the following one.
While there is a vertex u with in-degree zero, output u in the topological sorting output, and for every (u,v) delete (u,v) from G. The last step implies that the in-degree of vertices such as v will be lowered. Repeat this for the remainder graph G.

Proof.
One way to explain also an implementation of this method is as follows.
Step 1 (initialization): In $O(n+m$ time we find the in-degree $in(u)$ for every vertex u. We scan the vertices u and those whose in-degree is 0 are added to a linked list L (with head and tail pointer). We insert u at the tail of the list (think also a FIFO queue).
Step 2 (repetition): As long as there is a vertex in L we remove that vertex say u from the head of L and print it (topological sorting output). We then and only then and only onece do we adjust the in-degree of every vertex v associated with edge (u,v). This takes time $O(out(u))$ and is done once per vertex u during the deletion of u from L. If a vertex v attains in-degree 0, it is inserted at the tail of L then and only once as well. We keep on repeating step 2 until L is empty.
A vertex is removed from L exactly once. The work we do then is $O(out(u))$ to adjust the in-degree of every vertex v in the adjacency list of u i.e. $(u,v) \in E$. The total work performed during deletion is $\sum_u out(u) = \Theta(m)$. Every vertex is inserted into L once. Either in Step 1 for those u who in-degree are initially zero (source vertices), or during Step 2 when a vertex v's in-degree is lowered during the deletion of u from L and the graph along wiht allo of its associated edges. This is $O(n)$. Total time is $O(m+n)$. Note that when a vertex v appears in the output of topological sorting this means is in-degree is zero then, thus all vertices u with $(u,v) \in E$ have already been output before. ■

8.3 Graph algorithm applications

Exercise 341.

Assume a graph $G = (V, E)$ with $|V| = n, |E| = m$. A universal sink is a vertex with in-degree equal to $n-1$ and out-degree zero.
Find if G has a universal sink or not in $O(n)$ time. Graph G is represented through its adjacency matrix A.

Proof.
Observation 1. If v is a sink row v is full of zeroes and column v is full of ones (minus the $A(v, v)$).
Observation 2. Consider $A(i, j)$ in G. If $A(i, j) = 1$ it means i cannot be a (universal) sink. If on the other hand $A(i, j) = 0$ it means j cannot be a (universal) sink.
Thus starting with set S containing all vertices of V (i.e. $S = V$), we pick two vertices x and y. Applying the previous Observation 2, either x or y gets removed from S. Thus after $n-1$ removals one vertex remains in S. We check its row and column per Observation 1. It is either the universal sink or the graph has no such!. Note that we poked $(n-1)$ and $2n-1$ elements of A in total using Observation 2 and Observation 1 respectively. Total time per peek is $O(1)$ and thus total time is $\Theta(n)$. ■

Exercise 342.
A directed graph $G = (V,E)$ is singly connected if there is at most one directed path from u to v for all vertices $u,v \in V$. Give an algorithm to determine whether G is singly connected (decision algorithm). Assume an adjacency list representation.

Proof.
For each $u \in V$, run depth first search $dfs(G,u)$. If the dfs's reveal no cross or forward edges the graph is singly connected, otherwise no. Total time is $O(n(n+m)$, using an adjacency list. ■

Exercise 343.
You are given an acyclic connected and undirected graph $G = (V, E)$. Can you determine in $O(n)$ time whether it has or not a cyle (note that the bound involves only n but not m)?

Proof. Yes! G is a tree and thus $m = n - 1$. Therefore depth-first-search solves this problem in $O(n + m)$ time which is $O(n)$ as well. ■

Exercise 344.
Each of DFS(G) and BFS(G) create a tree. If G is an undirected and connected graph, then is it true that the height of the tree of DFS(G) is higher than that of BFS(G)? Explain.

Proof.
It depends on the starting position. If you have a windmill like graph with the center node being 1, and the radial nodes $2, \ldots, n$, starting DFS with 1 leads to a tree of height one, but then starting BFS on a radial node e.g. 2 leads to a height two tree. ■

Exercise 345.

You are given an undirected graph $G = (V, E)$. Can you determine in $O(n)$ time whether it has or not a cyle (note that the bound involves only n but not m)? Assume an adjacency list representation of the graph.

Proof. Yes! Run depth first search by also using two counters for each connected component discovered during the top-level call. One counts the vertices discovered and the other the edges discovered. If edges discovered is equal or more than vertices discovered. we have a cycle by the simple observation that in a tree edges are one less than vertices, and if in a connected component searched through depth-first-search we have discovered a number of edges equal to or more than the number of vertices, a cycle should exist there.

Note that in this algorithm the running time is $\sum_i O(n_i + m_i)$, where n_i is the number of vertices in the i-th component searched and m_i the number of edges searched in that component. Naturally $\sum_i n_i = n$. Given that the search process stops as soons as $m_i \geq n_i$, we have $m_i \leq n_i$. Therefore $\sum_i m_i = O(n)$. Combining the two we obtain that $\sum_i O(n_i + m_i) = O(n)$ as well. ■

Exercise 346.
Give an algorithm that detects a negative cost cycle.

Proof. Run Floyd-Warshall. At every iteration inspect the diagonal elements. Initially, they are all zero. If a value along the diagonal changes it can only grow smaller, i.e. negative. A negative $a_{ii}^k < 0$ show the existence of a cycle with end-points i whose all intermediate vertices are k or less. ■

Exercise 347.
You are given a graph $G = (V, E)$. We want to determine $\forall u, v$ whether there exists a path from u to v. Give three different algorithms that solve this problem and their worst case running time in terms of n and m.

Proof. Improvise. No solutions for this problem.

■

Exercise 348.

You are given an undirected graph $G = (V,E)$ with n vertices and m edges. The graph is represented through an adjacency list representation. Let the vertices be labelled $0,1,\ldots,n-1$. A[i] is the adjacency list of vertex i i.e. it contains all vertices j such that there exists an edge incident on i and j. A(i) is organized as a doubly-linked list with a head and tail pointer.

We want to find all triangles of G. A triangle is a tiplet (u,v,w) of vertices such that all u,v,w are distinct from each other, and $(u,v),(v,w)$ and (w,u) are edges.

Give an $O(n**3)$ algorithm that finds and prints all the triangles of G. Justify your answer.

Proof.

Convert the adjacency list into an adjacency matrix in Theta(n**2+m) time. The latter is Theta(n**2). Then three for loops give the answer.

```
    Triangle(G,V,E,A)
1. A[.][.] = n x n matrix initialized to zeroes
2. for (i=0;i<n;i++)
3.   A[i][j] =  scan A(i) and fill as needed A[i][0]...A[i][n-1].
4. for (i=0 ; i< n;i++)
5.   for (j=i+1 ; j< n;j++)
6.     for (k=j+1 ; k< n;k++)
7.      if i != j!=k and k!=i  and A[i][j]==1 and A[j][k]==1 and A[k][i]==1
8.          output( <i,j,k>);
```

Line 1 is Theta(n**2). Lines 2-3 requires Theta(n**2+n+m) i.e. Theta(n**2). Lines 4-8 are Theta(n**3) which satisfies the O(n**3) requirement. [No duplicate triangles are printed; see for-loop 4-5-6 i¡j¡k] ■

Exercise 349.

Lollipop graphs. Jerry is in a candy store for the first time. He tries to rank lollipops but he is disoriented. He generates statements of the form "I prefer (lollipop) i over (lollipop) j". Statements are transitive i.e. "I prefer i over j", and "I prefer j over k", this implies "I prefer i over k",

(a) Help Jerry to determine if his choices make sense.

(b) Help then Jerry rank his choices.

Proof.

Create a lollipop graph and make sure it has no cycles. Do so in $O(n+m)$. Do not ask how!

Topological sorting. Do not ask!

■

Exercise 350.

Find the longest path of a directed acyclic graph $G = (V,E)$. Assume an adjacency list represention A, where $A(i)$ points to the adjacency list of i.

Proof.

```
// Graph can also have weights on edges
// For this problem longest means max # of edges
1. Find topological sorting : let v1 , ... , vn are the vertices
2. for  u in V  let d[u]=0;
3. for  e in E
4.  if w(e) exists  cost(from(e),to(e))= w(e) // Alternative problem
5.     else         cost(from(e),to(e))= 1
3. For  u in <v1 , v2, ... , vn>
4.   for v in A(u)
5.      if d[v] <  d[u] + cost[u,v]
6.         d[v] =  d[u] + cost[u,v]
```

Exercise: Prove correctness. Show also running time is $O(n+m)$. ■

Exercise 351.

We have n boxes B with each box B_i has dimensions $l_i \times w_i \times h_i$ (length, width, height). No rotations of boxes are allowed e.g. one can not consider the box B_i to be $w_i \times h_i \times l_i$. You may assume all boxes are different (different dimensions).

B_i fits into B_j if and only if all $l_i \leq l_j$ and $w_i \leq w_j$ and $h_i \leq h_j$ are satisfied. Then we write $B_i \leq B_j$.

Find max k such that

$$B_{j_1} \leq \ldots \leq B_{j_k}$$

Derive the running time and express it in asymptotic notation.

Proof.

Form graph G with $V = \{B_i\}$ and E as follows. If $B_i \leq B_j$ (as previously defined) then adjacency list A(i) includes j. This is graph G constructed so far. Two more vertices will be added to G along their corresponding edges. Add to G a source vertex s and sink t as follows. Add edge (s, Bj) where $in(Bj) = 0$. Add edge (Bj, t) for all j. Find the longest path from s to t per the previous problem. Return length of the longest path deduced by two. Running time analysis: Create graph in $O(n^2)$ since n Boxes need to be checked against each other. Longest path requires $O(n+m)$ for topological sorting and BFS related activity. Total running time is thus $O(n^2+m)$. ■

Exercise 352.

You are given an undirected graph $G = (V, E)$. Its vertices are to be colored with two colors: red and black so that no two adjacent vertices are colored with the same color. Give an $O(n+m)$ time algorithm which either determines such a coloring or says NO to indicate that such a coloring does not exist.

To make the problem easier show the following.

(a) Show that if G has a cycle of odd length (counting edges), there is no such coloring.

(b) If (a) is not applicable, run BFS to find a coloring.

Proof.

Assume that the graph has an odd cycle, and let it be $u_1, \ldots, u_k$ where k is odd, AND a valid coloring C. Then $u_1, u_3, \ldots$ have the same color say a (red or black), and $u_2, u_4, \ldots$ have the other color say b (black or red). The u_k is colored a under C, and so is u_1 by the observation above. This violated the valid coloring C since adjacent vertices v_k, v_1 are of the same color. Note that given coloring C where a is red and b is black we can deduce coloring C' where a is black and b is red.

We assume that G is connected. Otherwise repeat (a) and this step (b) for all its connected components. Run BFS starting from an arbitrary vertex say vertex 1. Assume color of 1 is a (red or black). All 1's neighbors on level one are to be colored b, and so on. The running time of BFS is $O(n+m)$. We show that this algorithm works assuming no odd cycle. If the coloring produced is NOT a valid coloring, then there is an edge $e = (u, v)$ were u and v are of the same color. If u, v are of the same color it means their distance from 1 is of the same parity (odd and are colored b, or even and are colored a). Then this means the path from 1 to u, edge e, and the path from v to 1 is of length odd and it is a cycle, since the sum of two even or two odd number is always even and edge e turns the length of the path/cycle into an odd cycle. This contradicts the non-existence of an odd cycle.

■

Exercise 353.
True or False? If a directed graph G includes a path from u to v, and if DiscoveryTime[u] is less than DiscoveryTime[v] (also known as D[v]) in the DFS run, then v is a descendant of u in the DFS tree/forest generated (array p).

Proof.

Consider a graph $(a,u),(u,a),(a,v)$ with three vertices and three edges. Start a DFS on a i.e. $dfs(G,a)$. Then $D[a]=1$, $D[u]=2$, and $D[v]=3$. Obviously there is a path from u to v, $D[u]<D[v]$, but v is not a descendant of u in the tree constructed that includes $(a,u),(a,v)$. ■

8.4 Dijkstra's algorithm

Exercise 354.

We use Dijkstra's algorithm to find the shortest paths from vertex s for the problem $(G=(V,E),d,s)$ where d is the distance matrix (list) contains positive values. We then modify every edge distance i.e. $d'(u,v)=2*d(u,v)$. How does the solution for $(G=(V,E),d,s)$ relate to the solution for the problem $(G=(V,E),d',s)$? Explain briefly and justify your answer.

Proof.

The paths are the same but the cost of any such path now is doubled. ■

Exercise 355.
We use Dijkstra's algorithm to find the shortest paths from vertex s for the problem $(G=(V,E),d,s)$ where d is the distance matrix (list) contains positive values. We then add to every edge distance a 2 i.e. $d'(u,v)=2+d(u,v)$. How does the solution for $(G=(V,E),d,s)$ relate to the solution for the problem $(G=(V,E),d',s)$? Explain briefly and justify your answer.

Proof.
Think.

■

Exercise 356.
Show a lower bound of $\Omega(n \lg n)$ for the running time of Dijkstra.

Proof.
Relate it to a sorting problem instance. ■

8.5 Floyd-Warshall's algorithm

Exercise 357.

You are given a directed graph with positive weights on its edges. Show how you can solve the minimum cost directed cycle problem in that graph in time faster than $o(n^4)$ (this is a little-oh) in the worst-case. A directed cycle is a directed path that starts and ends at the same vertex and has at least one edge. A minimum cost directed cycle is one for which the sum of the weights of the edges of the cycle is minimal.

Proof.
Set $d_{ii} = \infty$. Run Floyd-Warshall. Then d_{ii}^n will give the length of the smallest cycle from i to i. Find the minimum among all a_{ii}^n for all i.

■

Exercise 358.
How would you detect the existence of a negative-weight cycle in Floyd-Warshall?

Proof.
At every iteration inspect the diagonal elements. Initially, they are all zero. If a value along the diagonal changes it can only grow smaller, i.e. negative. A negative $a_{ii}^k < 0$ shows the existence of a cycle with end-points i whose all intermediate vertices are k or less. ■

Exercise 359.
For the graph indicated apply Floyd Warshall.

Proof.

$$D^{(0)} = \begin{bmatrix} 0 & 5 & 40 & 50 \\ 5 & 0 & 8 & 60 \\ 40 & 8 & 0 & 7 \\ 50 & 60 & 7 & 0 \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} 0 & 5 & 40 & 50 \\ 5 & 0 & 8 & \mathbf{55} \\ 40 & 8 & 0 & 7 \\ 50 & \mathbf{55} & 7 & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 5 & \mathbf{13} & 50 \\ 5 & 0 & 8 & 55 \\ \mathbf{13} & 8 & 0 & 7 \\ 50 & 55 & 7 & 0 \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & 5 & 13 & \mathbf{20} \\ 5 & 0 & 8 & \mathbf{15} \\ 13 & 8 & 0 & 7 \\ \mathbf{20} & \mathbf{15} & 7 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 5 & 13 & 20 \\ 5 & 0 & 8 & 15 \\ 13 & 8 & 0 & 7 \\ 20 & 15 & 7 & 0 \end{bmatrix}$$

■

8.6 Spanning tree and extremes of

Exercise 360.

The maximum-cost spanning tree problem is defined similarly to the minimum-cost spanning tree problem except that we want to maximize the sum of the weights of the edges of the tree. Give an algorithm that finds the maximum-spanning tree of an undirected graph $G = (V, E, W)$ where edges have weights represented in the weight matrix W. What is the running time of the algorithm?

Proof. If the graph is $G = (V, E, W)$ consider graph $G' = (V, E, -W)$ i.e. we negate the weights of all the edges in G'. A maximum spanning tree T in G is a minimum spanning tree in G'. Note that the conversion of W into $-W$ takes time $O(m)$. If G is represented by an adjacency-matrix we find all the edges and negate the corresponding weights in $O(n^2)$ time. If G is represented by an adjacency-list we go through all the edges in time $O(m)$ and negate the corresponding weights.

Whatever the cost is $O(m \lg m)$, the running time of Kruskal's algorithm will contribute more.

■

Exercise 361.

We have an undirected weighed graph whose edges have weights that are only 10 or 20. Can you find the minimum spanning tree of this graph in $O(n+m)$ time? Explain.

Proof. Yes! We only need to maintain a heap whose operations are $O(1)$ rather than the traditional $O(\lg n)$. Note that in Prim's algorithm the key in a heap is the edge weight. We can only have two values 10 or 20. Therefore we can implement a heap with an array of size two that stores two (doubly) linked lists. We attach to element 0 a linked list of all edges of weight 10, and to element 1 a linked list of all edges of weight 20. We only insert or delete from the head of the list to maintain $O(1)$ time operations.
Insert inserts the weight of edge (u,v) into the Heap as the weight of v. We insert either to the linked list attached to element 0 or to the one attached to element 1. Either time is constant. At the same time we point $point[v]$ to this location (we can also choose to record in (u,v) information that this edge is related to v).
ExtractMin takes $O(1)$ time. Check list 0 first. If list 0 is not empty, extract its element pointed by head. If list 0 is empty go to list 1 and extract an element through the head pointer. If both lists 0 and 1 are empty return "underflow".
Decrease takes constant time because we use doubly linked lists. We might move an element (edge) from one list to the other. Note that in accordance to Prim's algorithm for every vertex in an array of size n we store a pointer $point[v]$ that points to the position of the vertex in the linked list/heap. Therefore DecreaseMin(v) (lines 10-12), first locates v in the corresponding linked list using $point[v]$ (and this should be the list attached to element 1) and if the value needs to be decreased (i.e. from 20 to 10) then the element pointed to by $point[v]$ is deleted in constant time, a new element is inserted in the linked list of element 0 and $point[v]$ points to this new location (and also the new location points to v). ■

Exercise 362.

You are given an undirected connected graph G=(V,E), where $|V| = n$ and $|E| = m$. G is represented through an adjacency list, and A(i) points to the adjacency list of vertex i, i.e. to all vertices j such that $(i, j) \in E$. We are interested in finding a spanning tree of graph G. (A reminder: a spanning tree of a connected undirected graph is a tree that spans i.e. touches all of its n vertices.) Justify your arguments is implicit.
(i.) Give a time efficient algorithm for finding a spanning tree of G. ('a' means if there are many, find ONE).
(ii.) Provide the running time of your solution given in as tight asymptotic notation as possible.

Proof.
(i) Run DFS in $T(n,m) = O(n+m)$ or $\Theta(n+m)$. The p array contains the spanning tree (the collection of t edges)
(b) $T(n,m) = \Theta(n+m)$. ■

Exercise 363.
What if we use Kruskal's algorithm on a graph where edge costs from 1 to 1000? What is its running time then?

Proof.
Sorting then (using CountSort, or in other cases RadixSort) would only take $O(m)$ time rather than $O(m \lg m)$. Time is reduced. ■

Exercise 364.
What if we use Prim's algorithm on a graph where edge costs from 1 to 1000? What is its running time then?

Proof.
This is more interesting. We can reduce running time to $O(n+m)$. Think! ■

Exercise 365.

You are given a directed graph with positive weights on its edges. Show how you can solve the minimum cost directed cycle problem in that graph in time faster than $o(n^4)$ (this is a little-oh) in the worst-case. A directed cycle is a directed path that starts and ends in the same vertex and has at least one edge. A minimum cost directed cycle is one for which the sum of the weights of the edges of the cycle is minimal.

Proof. Set $a_{ii} = \infty$. Run Floyd-Warshall. Then a_{ii}^n will give the length of the smallest cycle from i to i. Find the minimum among all a_{ii}^n for all i. ■

Exercise 366.
A graph G=(V,E) is given; both the adjacency list and adjacency matrix are provided. Find a cycle of length n that contains all vertices in ... jokingly ... polynomial time.

Proof. Kidding. This is the Hamiltonian cycle problem. The decision version is NP-complete. ■

Exercise 367.

Find the diameter of a tree $T = (V,E)$. That is T is an undirected, connected and acyclic graph. The diameter is the longest path in it.

Proof.

Modify depth first search. We compute for every vertex u in T $P[u] =$ length of longest path in subtree of u i.e. $dfs(G,u)$, $Q[u] =$ length of longest path in subtree of u to u. Thus if u has two children r,s or if it has more than two r,s are the ones with highest $Q[r]$, and $Q[s]$, then $P[u] = Q[r]+Q[s]+2$. Among the children of u

```
DFS(G,V,E)
 for each u      color[u]=W, P[u]=0; Q[u]=0;
 for each (u,v)  t[u,v]  = unlabeled
 dfs(G,u);

 /* Standard DFS */
 dfs(G,u) {
 color[u]= G
 for all e=(u,v) in E
  if color[v] == W {
   t[u,v] = tree; // aka t
   dfs(G,v);
  }
  else {
   t[u,v] = back; // aka b
  }
 }


 /* Modified DFS */
 dfs(G,u) {
 1.  color[u]= G;
 2.  A[u]=B[u]=0;
 3.  m=M=0;
 4.  for all e=(u,v) in E
 5.   if color[v] == W {
 6.    t[u,v] = tree; // aka t
 7.    dfs(G,v);
 8.    Q[u] = max(Q[u],Q[v]+1);
 9.    update m,M with Q[v] and prior information (u,w) w encountered before v
10.    P[u] ;
11.   }
12.   else {
13.    t[u,v] = back; // aka b
14.   }
15.   P[u] = max(P[u], m+M+2);
16.  }
```

■

8.7 Triangles

Exercise 368.

(Method 1: Naive to less naive.)
Give a naive algorithm that finds the number of triangles in an undirected graph $G = (V,E)$. Running time $O(n^3)$ expected. Adjacency matrix or list representation for G.
A triangle i,j,k is a subgraph where $(i,j) \in E, \quad (j,k) \in E, \quad (k,i) \in E$. We can call that triangle i,j,k triangle. Note that for an undirected graph, a triangle i,j,k is the same with j,k,i and k,i,j and i,k,j and k,j,i and j,i,k. It should be reported once not 6 times!

Proof.
First convert an adjacencly list into an adjacency matrix representation; the cost is $O(n+m)+O(n^2) = O(m+n^2)$, if the graph is not already in an adjacency matrix representation.
(a) Three nested for loops for a total of $\binom{n}{3}$ time, making sure $i < j < k$ to avoid reporting a triangle 6 times. The constant time checking is to make sure $A(i,j) = A(j,k) = A(k,i) = 1$. Time is $O(n^3)$.
(b) Use matrix multiplication. Explore $D = (A \times A) \times$ and deduce the meaning of $trace(D)/6$ where $trace(D) = \sum_i D_{ii}$.

■

Exercise 369.

(Method 2: Less than less naive .)
Give a non-naive algorithm that finds the number of triangles in an undirected graph $G=(V,E)$. Running time $o(n^3)$ should be expected for some problems or $O(mn)$ or $O(mD)$, where $D=\max_i d(i)$ and $d(i)$ is the degree of vertex i. Adjacency matrix representation for G.

Proof.
Find the degree $d(i)$ of every vertex i. This is $\Theta(n^2)$. For each vertex i pick vertices j and k in its adjacency list if $d(i)>1$. There are $\binom{d(i)}{2}$ such pairs of vertices, and fewer if $i<j<k$ is too be maintained (see previous problem). If $(j,k)\in E$ along with the fact that $(i,j)\in E,(i,k)\in E$, leads to a triangle. Count it if $i<j<k$ i.e. then `count = count +1 ;` Let $D=\max_i d(i)$, i.e. D is the maximum degree of a vertex of a graph. (Some books might then use the term valence to refer to D, but usually valency or valence refers to the term we use for the degree of a vertex and thus identifies $d(i)$ for vertex i.) The running time of this algorithm is (we drop O notation usage until the very end). Moreover, $n=|V|$ and $m=|E|$.

$$\begin{aligned}
T(n) &= \sum_i \binom{d(i)}{2} \\
&= \sum_i \frac{d^2(i)-d(i)}{2} \\
&= \sum_i \frac{d^2(i)-d(i)}{2} \\
&= \sum_i \frac{d^2(i)}{2} - \sum_i \frac{d(i)}{2} \\
&= \sum_i \frac{d^2(i)}{2} - \frac{2m}{2} \\
&= \sum_i \frac{d^2(i)}{2} - m
\end{aligned}$$

One possibility for the last sum is $d()\leq n$ and thus

$$T(n)=O(\sum_i \frac{d^2(i)}{2}) = O(n\times d(i)\times d(i)) = O(n\times n\times n) = O(n^3).$$

Another possibility for the last sum is $d()\leq n$ and thus

$$T(n)=O(\sum_i \frac{d^2(i)}{2}) = O(n\times d(i)\times d(i)) = O(n\times D\times D) = O(nD^2).$$

Another possibility for the last sum is $d()\leq n$ and thus

$$T(n)=O(\sum_i \frac{d^2(i)}{2}) = O(\sum_i \frac{D\cdot d(i)}{2}) = O(D\times \sum_i (d(i)/2)) = O(D\times 2m/2) = O(mD).$$

■

Exercise 370.

(Method 3: more refined .)
Give a refined algorithm that finds the number of triangles in an undirected graph $G = (V,E)$. Running time $O(m^{1.5})$ should be expected for some problems. Adjacency matrix representation for G.
(Such a method is referred to papers by N. Alon or U. Zwick or N. Alon and U. Zwick.)

Proof.
Separate the vertices V of G into $V_L(B)$ and $V_H(B)$, where $V_L(B) = \{v \in V : d(v) <= B\}$, $V_H(B) = \{v \in V : d(v) > B\}$, for some value B. B can be set to $B = m^{\frac{g-1}{g+1}}$ for some chosen g; in our case $g = 3$ and thus $B = \sqrt{m}$. And thus in the remainder $V_L(B) = V_L$ and $V_H(B) = V_H$. Let $|V_L| = n_L$ and Let $|V_H| = n_H$.
Case 1. For V_L and thus $B = \sqrt{m}$, which implies that for every $i \in V_L$, we have $d(i) \leq B$ we count triangles using the previous method. The running time would be

$$T(n) = O(mD)$$

where D for the vertices in D_L is $D = B$ and thus $D = \sqrt{m}$. Therefore

$$T(n) = O(m^{3/2}).$$

Case 2. For V_H, every vertex i such that $i \in V_H$ has $d(i) > B$. Therefore

$$\begin{aligned}
\sum_{i \in V_H} d(i) &\leq \sum_{i \in V_H \cup V_L} d() = 2m \\
\sum_{i \in V_H} d(i) &\leq 2m \\
|V_H| \cdot B &\leq \sum_{i \in V_H} d(i) \leq 2m \\
n_H \cdot B &\leq 2m \\
n_H &\leq 2m/B \\
n_H &\leq 2\sqrt{m}
\end{aligned}$$

We then find the number of triangles over V_H whose $n_H \leq 2\sqrt{m}$ using matrix multiplication in time $O((\sqrt{m})^3)$ using Binet's algorithms, or in time $O((\sqrt{m})^{\lg 7})$ using Strassen's algorithm. In the former case $T(n) = O(m^{3/2})$ in the latter $T(n) = O(m^{\lg 7/2})$.
Combining cases 1 and 2 we still have $Tn) = O(m^{3/2})$.

■

8.8 Miscellanea review

Exercise 371.

For an undirected graph $G = (V, E)$ that is connected the tree generated by DFS(G) will have height larger than that generated by BFS(G). True or False? Explain

Proof.
It depends on the starting vertex. For a windmill like graph DFS generates a tree of height one if DFS starts from the center. If BFS starts from a peripheral vertex, it will be of height two.

■

Exercise 372.

For an undirected graph $G = (V,E)$ that is connected the tree generated by DFS(G) is the same as the one generated by BFS(G) if and only if G is a tree. True or False? Explain

Proof.
(a) Let BFSTree and DFSTree are the same T but T is not G. Then there should exist an edge (u,v) in G that is not in T. In the DFSTree edge (u,v) was not picked because v was a gray node or the other way around, i.e. one of u,v is an ancestor of the other v,u. Moreoved in the BFSTree u and v's levels differ by one. This means the edge connecting them MUST be in T.

■

Exercise 373.

How many traversals do we need to reconstruct a BST T? Assume you do not know T but you can ask any number of preorder, postorder, inorder traversals. What is the running time of the reconstruction of T for a minimal number of traversals requested?
First you minimize the traversals requested and then minimize reconstruction time.

Proof.
If we have or request a preorder traversal P, the inorder traversal I can be generated in $O(n \lg n)$ time. The starting node of the preorder is the root. Do a binary search of the root in $r = P[0]$ into I in $O(\lg n)$ time. Then I is split around r in the left and right subtrees. Running time is $O(n \lg n)$. Postorder is similar but start from $O[n-1]$ not $O[0]$ instead.

■

Exercise 374.

Yet another shortest path algorithm (Bellman Ford). Graph must not have negative cost cycle.

```
BellmanFord(G,D,W,s) { // |V|=n , |E|=m  , source vertex s
 for each u in V
    D[u] = oo ;
 D[s] = 0;
 for(i=1 ; i < n ; i++) {
   for each (u,v) in E {
      // D[v] = min( D[v] , D[u] + W[u][v]
      if D[v] > D[u] + W[u][v]
         D[v] = D[u] + W[u][v];
      p(v)= u
   }
 }
 return(D);
}
```

Proof.
The running time is obviously $O(nm)$. Let $d[v]$ is the cost of the shortest path from s to u. The algorithm finds a $D[v]$. works as claimed.
Let $D_i[v]$ be $D[v]$ at the conclusion of iteration i. Then we shall prove that $D_{n-1}[v] = d[v]$ for each v in V. Moreover we shall also prove that $D_i[v]$ is the shortest path from s to v that uses no more than i edges (i.e. $\leq i$ edges). This will prove the result assuming no negative cost cycles that will force BellmanFor to follow such a cycle multiple times.
Base case $k = 0$ is satisfies since then $D[v] = D_0[s] =$ for $v = s$ and $D_0[v] = \infty$ for $s \neq v$.
Inductive hypothesis/step. Suppose $D_i[v]$ is the shortest path from s to v that uses no more than i edges (i.e. $\leq i$ edges). SHow it for $i+1$.
Let P be the shortest path from s to v with $i+1$ edges or less. Let u be the vertex incident to v on P. Let P_u be the path from s to u. This path has i edges and by the inductive hypothesis P_u must be the shortest path form s to u with at most i edges. Or if the latter was not the case we could replace P_u with an alternative shorter path Q_u that would then lead to a shorter path Q from s to v than P. By the inductive hypothesis the cost of P_u is $D[u] = cost(P_u)$ i.e. $D_i[u]$.
At iteration $i+1$, we update $D_{i+1}[v] = \min\{D_i[v], D_i[u] + W[u][v]\}$. $D_i[u] + W[u][v] = cost(P_u) + W[u][v] = cost(P)$ and therefore $D_{i+1}[v] \leq cost(P)$. Moreover $D_i[v]$ is by the induction hypothesis the cost of the shortest path from s to v using at most i vertieces, and $D_i[v] \geq cost(P)$. Therefore $D_{i+1}[v] = cost(P)$.

■

Exercise 375.

Show how you can detect a negative cost cycle through Bellman Ford.

Proof.
Suppose that there is a negative cost cycle $v_0, \dots v_t = v_0$. This means

$$\sum_{i=1}^{t} W[v_{i-1}][v_i] < 0$$

Suppose that for all u, v we have instead $D[v] \leq D[u] + W[u][v]$. Then

$$D[v_i] \leq D[v_{i-1}] + W[v_{i-1}][v_i].$$

Summing up for all i of the cycle we have

$$\sum_{i=1}^{t} D[v_i] \leq \sum_{i=1}^{t} D[v_{i-1}] + \sum_{i=1}^{t} W[v_{i-1}][v_i].$$

The first and second sum cancell out since $D[v_0] = D[v_t]$ since $v_t = v_0$. Thus $\sum_{i=1}^{t} W[v_{i-1}][v_i] \geq 0$, contradicting that the cycle is of negative cost.

```
BellmanFord(G,D,w,s);
for each (u,v) in E
  if D[v] > D[u] +W[u][v]
      return("Negative cost cycle exists");
```

■

Exercise 376.
Given an undirected graph $G = (V, E)$ determine whether it is a bipartite graph i.e. you can split V into A and B, where edges go from A to B and no edge has both end points in A and no edge has both end points in B.

Proof.
Color the vertices with two colors. Start say a DFS or BFS from some vertex u. If color of u was set to R (call it red), then all edges $(u, *)$ that will be discovered will be G (call it green). Edges that are labeled b in say BFS are edges of cycle. Its two end-points must be one in A and the other in B or graph G is not bipartite. Fill in the details. ■

Exercise 377.

If $P = NP$ then there is a polynomial time algorithm for the Hamiltonian cycle problem. Use it to find the Hamiltonian cycle (in apparently polynomial time).

Hamiltonian cycle (also, circuit): For a given graph $G = (V,E)$ does G contain a sequence of vertices $\langle u_1, \ldots, u_n \rangle$ where $n = |V|$ such that $(u_i, u_{i+1}) \in E$ for all i such that $1 \leq i \leq n-1$ and also $(u_n, u_1) \in E$? Call it HC in short.

Note the decision problem above outputs a YES or NO. It is this version that is NP-complete.

Proof.

Sometimes we call this method self-reducibility. If $P = NP$ there is a Turing Machine that recognized in poly time whether a graph G has a Hamiltonian circuit or NOT (YES or NO). Call it $Oracle(G)$. Start with $G = (V,E)$, enumerates its edges (i,j) say in increasing i and then increasing j for the same i. Have a pointer to the first edge, call the pointer T. Pick one edge $e = (u,v)$ and constract $G_e = (V, E - \{e\})$. Run and Record $Oracle(G)$ and $Oracle(G_e)$. Some cases

(a) If $Oracle(G)$ says NO, the graph has no HC, and we done

(b) if $Oracle(G)$ says YES, and $Oracle(G_e)$ says YES, then recur on graph G_e that has one fewer edge.

(c) if $Oracle(G)$ says YES, and $Oracle(G_e)$ says NO, then the edge e removes is an important edge, put it back and recur on G and advance T.

Easy to show that $O(n^2)$ calls to Oracle are needed to have remaining in the leftover of G the n edges of the cycle, unless of course algorithm exited because of case (a)!

Total time polynomial. ■

Exercise 378.
If $P = NP$ then there is a polynomial time algorithm for the Clique problem. Use it to find the Hamiltonian cycle (in apparently polynomial time).

Clique: For a given graph $G = (V, E)$, and integer $k \leq n$, where $n = |V|$, does G contain a clique of size k or more ? A clique is a subset $V' \subseteq V$ of the vertices of V such that $|V| \geq k$ and every two vertices in V' are connected by an edge of E. The problem is a YES or NO decision problem.

Note the decision problem above outputs a YES or NO. It is this version that is NP-complete.

Proof.
If $P = NP$, may be adjust the problem so that it gives a clique of max k. (Do binary search.) Use vertices, not edges and call an oracle on G and $G - \{u\}$ as needed. ■

Part III

Numbers, polynomials, and matrices

Chapter 9

Arithmetic problems

9.1 Integers

Exercise 379.

You are given an integer n. We would like to determine whether n is the integer power of an integer i.e. whether there exist two integer numbers a, b, $a > 1$ such that $n = a^b$. Give an algorithm that determines whether n is the integer power of an integer (and in that case one set a, b is printed out) or not in time that is $O(\lg^3 n)$. Use the word model for operation counting i.e. the cost of one addition, subtraction, multiplication is one. Divisions are not allowed unless they are a division by two (i.e. a shift).

Hint: If $n = a^b$ how big can b be relative to n ? And if you have spare time, can you beat the $O(\lg^3 n)$ time bound and by how much?

Proof. Binary search on the integer range from 1 to n! ■

Exercise 380.

The Fibonacci sequence is defined as $F(n) = F(n-1) + F(n-2)$, for $n \geq 2$, and $F(1) = F(0) = 1$ otherwise.
(a) Write a recursive program (pseudocode) that finds the n-th Fibonacci number (i.e. $F(n)$).
(b) Write an iterative program (pseudocode) for $F(n)$.
(c) Which of the two programs is asymptotically faster? Explain.
(d) Give an algorithm that is $O(\lg n)$ in the WORD model.
(e) Show how to compute the n-th Fibonacci number with the minimal number of bit operation in the BIT model. Say $O(n^2)$ bit operations.

Hint: $F_n \leq 2^n$, if we recall induction and its applications of Subject 1.

Proof.

```
  Fibonacci-Recur(n) {
1. if (n==0) return(1);
2. if (n==1) return(1);
3. return(Fibonacci-Recur(n-1)+Fibonacci-Recur(n-2));
  }
```

(b)

```
  Fibonacci-Iterate(n) {
0 if (n==0) or (n==1) return(1)
1. f_n_minus2 = 1;
2. f_n_minus1 = 1;
3. result= 0;
4. for (i=2;i<=n;i++) {
5.   result = f_n_minus1 + f_n_minus2;
6.   f_n_minus2= f_n_minus_1;
7.   f_n_minus1= result;
   }
8. return(result);
  }
```

■

This is Introductory section material. By now you should be able to solve (a), (b) and (c). For (d) think! The running time of Fibonacci-Iterate is easy to find. The for loop runs for $\Theta(n)$ iterations for a total time of $\Theta(n)$ operations. The running time of Fibonacci-Recur can be estimated as follows. To find $F(1)$ or $F(0)$ it takes one operation (line 1 or 2). For $F(2)$ we need to find recursively $F(1)$ and $F(0)$, i.e. two operations total. For $F(3)$ we need to find recursively $F(2)$ and $F(1)$. Finding $F(1)$ takes $F(1) = 1$ operations. Finding $F(2)$ also takes $F(2) = 2$ operations. Therefore by an inductive argument, finding $F(n)$ takes $F(n)$ operations. From a discrete math book $F(n) = \Theta(((1+\sqrt{5})/2)^n) = \Theta((1.61803)^n)$. This is a tighter bound than the one we derived in class or Subject 1. The latter number is asymptotically larger than n. Therefore Fibonacci-Iterate is asymptotically faster. Of course memoization can improve the recursive algorithm.

Exercise 381.

What does the following recursive program do?

```
int  Findme ( int x , int y) {
 if ( y == 0)  return (0);
   else
    return( Findme( 2*x, floor(y/2)) + x * (y % 2 ) );
}
```

Proof. Suppose we want to divide an integer y by 2. y is the dividend and 2 is the divisor. By the elementary school division we have that

$$y = 2Q + R$$

where Q is the quotient and R is the remainder. R is 0 or 1. The quotient is $Q = floor(y/2)$, and the remainder is $R = y\%2$. Therefore,

$$y = 2Q + R = 2floor(y/2) + (y\%2)$$

If we multiply by x both sides of the equality we get that

$$x * y = x * (2floor(y/2)) + x * (y\%2) = (2 * x) * (floor(y/2)) + x * (y\%2).$$

Therefore the product $x * y$ is equal to the sum of the product of $2x$ times $floor(y/2)$ and $x * (y\%2)$. Note that the product $x * (y\%2)$ is easy to find. If the $y\%2$ is 0, the product is 0, if $y\%2$ is 1 the product is x.
Therefore Findme finds the product $x * y$ recursively. ■

Exercise 382.
(a) You multiply $x = x_1 \dots n_2 \dots x_k$, k (non-negative) integers $x_1, \dots, x_k$. In the bit model asymptotically estimate the number of bit operations and show it is $O((\lg x)^2)$, i.e. dependent on the result x and independent of the intermediate numbers.
(b) In the bit model estimate the number of bit operations for $x = y^n$. Input is y and n and output is x. Assume all x, y, n are non-negative integer.
(c) You are given x and you are asked to determine whether x is of the form $x = y^n$ for some integers y, n. Give an efficient algorithm for that. (Testing for all y and all n is not an efficient solution.)

Proof. No solutions provided for the other problem(s). Think! and practice! ■

Exercise 383.

Theorem 9.1

Division is equivalent to multiplication (bit model of computation).

Proof. Let y be a real number. We would use Newton-Raphson to find $x = 1/y$. Let us consider $f(x) = y - 1/x$. The solution of $f(x) = 0$ for x is $y - 1/x = 0$, therefore the solution $x = 1/y$ is the inverse of y. Applying the Newton-Raphson method, we get that $f'(x) = 1/x^2$. If the first approximation to $1/y$ is x_0 (we will discuss how to choose such a x_0 later). Then $x_1 = x_0 - f(x_0)/f'(x_0) = x_0 - (y - 1/x_0)/(1/x_0^2)$ and then we obtain that $x_1 = 2x_0 - yx_0^2$. Therefore,
Claim 1: The inverse of y is the solution of $f(x) = 0$, where $f(x) = y - 1/x$.
Claim 2: If x_0 is an initial approximation to $1/y$, then the sequence of Newton-Raphson approximations is given by : $x_i = 2x_{i-1} - yx_{i-1}^2$.
Claim 3: If x_0 is an initial approximation to $1/y$, then at iteration i of Newton-Raphson the error is

$$1/y - x_i \quad \leq \quad y(1/y - x_0)^{2^i}. \tag{9.1}$$

Consider first the error of approximating $1/y$ by x_1. We obtain the following error bound.

$$1/y - x_1 \quad = \quad 1/y - 2x_0 + yx_0^2 = y(1/y^2 - 2x_0(1/y) + x_0^2) = y(1/y - x_0)^2 \tag{9.2}$$

Induction completes the rest of the proof.
If $y \leq 1$ and $\mid 1/y - x_0 \mid \leq 1/2$, then after i iterations the error of approximating $1/y$ by x_i is at most $1/2^{2^i}$. For $i = \lg n$, the error is less than $1/2^n$.

Lemma NR. If $Y \leq 1$, then an approximation to $1/Y$ can be computed by Equation (9.1) with $Y = y$. After $\lg n$ iterations, the error of this approximation is at most $1/2^n$, i.e. the value returned $x_{\lg n}$ is such that $\mid 1/Y - x_{\lg n} \mid \leq 1/2^n$. ■

Exercise 384.

You are given an a-bit positive integer A and an b-bit positive integer B. You may assume $A \geq B$. How fast can you compute $A+B$, $A-B$, $A \cdot B$, and find Q, R such that $A = BQ+R$, $0 \leq R < B$?

Express your answer in terms of a and b.

Proof.

Addition and subtraction can be done in $O(a+b)$ time. Since $A \geq B$ we have that $a \geq b$ and thus running time is also $O(a)$. The bound is tight i.e. $\Theta(a)$ since we need to read the a bits of A and b bits of B as well.

Multiplications takes time $O(ab)$ (ordinary elementary school method). Karatsuba-Ofman offers an improved bound of $O(a^{\lg 3})$.

Division based on the elementary school long division algorithm requires $a-b$ steps each one involving a subtraction of a b or $b+1$-bit number from a portion of A. This takes $O(b)$ from part (i). Therefore the total time is $O((a-b)b) = O(ab)$ or just $O(a^2)$. ■

9.2 Polynomials

Exercise 385.

You are given six polynomials $f_1, \ldots f_6$ of degrees $1,2,3,1,4,5$ respectively. We are interested in finding the product $f = f_1 f_2 f_3 f_4 f_5 f_6$. Assume that the cost of multiplying two polynomials of degree a and b is $a \cdot b$. Find a schedule for multiplying the six polynomials that is of the lowest possible total cost.

Proof. Multiply them any way you like. The final result is a polynomial of degree 16 and the total cost is 100. Note that 100 is the sum of all products $d_i \cdot d_j$ for every combination of two polynomials f_i and f_j among the six and d_i and d_j are respectively the degrees of f_i and f_j. In our example we have $6(6-1)/2$ such pairs of polynomials.

$$\begin{aligned} 100 &= d_1(d_2+d_3+d_4+d_5+d_6)+d_2(d_3+d_4+d_5+d_6)+d_3(d_4+d_5+d_6)+d_4(d_5+d_6)+d_5d_6 \\ &= 1(2+3+1+4+5)+2(3+1+4+5)+3(1+4+5)+1(4+5)+4\cdot 5 \\ &= 15+26+30+9+20 \end{aligned}$$

This sum is independent of the order of the evaluation of the products; an easy proof is by induction (eg. with reference to the 6 polynomials above, the product of five of them $f_2, \ldots, f_6$ takes time equal to the 4 summands except the first one $d_1(d_2+d_3+d_4+d_5+d_6)$; in the last step of an inductive proof the polynomial f_1 whose degree is d_1 is multiplied by the result $f_2 \ldots f_6$ whose degree is $d_2+\ldots+d_6$ and the cost of that multiplication is the missing/absent (first) term. Permute the polynomial and the result remains the same except that the pairs of the products $d_i d_j$ get rearranged.).

Note. One of the reasons that this is so is that polynomials satisfy commutativity therefore $f_1 f_2 = f_2 f_1$. What happens if instead of polynomials you have matrices and the operation is matrix multiplication? The cost then depends on the specific schedule that you choose. This is the matrix chain product; on how to determine the optimal schedule using dynamic programming. ■

Exercise 386.

For the remainder of this problem we assume that the cost of multiplying two polynomials of degrees m and n respectively, is $m+n$. You are given six polynomials $f_1, f_2, f_3, f_4, f_5, f_6$ of degrees $1,2,3,1,4,5$ respectively. We are interested in finding the product $f = f_1 f_2 f_3 f_4 f_5 f_6$ by minimizing the total cost of all intermediate polynomial multiplications involved in this computation. The total cost of finding the product is thus the sum of the costs of all polynomial multiplications involved. What is the best schedule to find f and what is the cost associated with this best schedule? Explain.

Example Let g_1, g_2, g_3 be three polynomials of degree 3,2,6 respectively. If we first multiply $h_1 = g_2 \cdot g_3$ the cost of this multiplication is $8 = 2+6$, because the two polynomials g_2, g_3 are of degrees 2 and 6 respectively. If we then find $h_2 = h_1 \cdot g_1 = g_1 \cdot g_2 \cdot g_3$ the cost of this second multiplication is $11 = 3+8$, because the degree of g_1 is 3 and the degree of h_1 is $8 = 2+6$, i.e. the sum of the degrees of g_2, g_3. This schedule results to a total cost of $8+11 = 19$ for computing the product $g_1 \cdot g_2 \cdot g_3$. It remains to be seen whether it is the best schedule or not.

Proof. Multiply f_1 by f_4. The result with f_2. Then say we multiply f_3 with f_5. The product of f_1, f_4, f_2 with f_6, and then the last remaining polynomials together. This is a greedy, ala Huffman approach. You fill in the details. ■

9.3 Matrices

Exercise 387.

TRUE or FALSE? Anything other than a TRUE or FALSE is considered wrong!

1. If we have t^2 matrices $M_1, M_2, \ldots$, of geometry $r_1, r_2, \ldots$ we can determine the optimal schedule to multiply these matrices using the dynamic programming algorithm for the matix chain product available in the textbook and Subject 9 in time $O(t^3)$.
2. If we have n matrices M_i, $i = 1, \ldots n$, each one of geometry $n \times n$, we can find the optimal schedule (requiring the minimal amount of scalar operations) to compute the product $M_1 \times M_2 \times \ldots \times M_n$ of the n matrices in $\Theta(n^3)$ time.
3. If we have the same n matrices M_i, $i = 1, \ldots n$, each one of geometry $n \times n$, the optimal schedule to compute the product $M_1 \times M_2 \times \ldots \times M_n$ requires $O(n^4)$ scalar operations.
4. If we have n matrices M_i, $i = 1, \ldots n$, each one of geometry $n \times n$, the optimal schedule to compute the product $M_1 \times M_2 \times \ldots \times M_n$ can be done in $O(n^{3.9})$ scalar operations.

Proof. 1. FALSE, 2.FALSE, 3. TRUE, 4. TRUE. ■

Exercise 388.

Either from CLRS [6] or a variation of a problem there.

a. What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?

b. V. Pan has discovered a way of multiplying 68×68 matrices using 132464 multiplications, 70×70 matrices using 143640 multiplications, 72×72 matrices using 155424 multiplications. Which method yields the best asymtotic running time when used in a divide and conquer matrix multiplication algorithm? How does it compare to Strassen's algorithm?

Proof. a. Strassen's algorithm uses the pattern of multiplication for 2×2 matrices and divide-and-conquer by splitting a matrix multiplication of $n \times n$ matrices into a matrix multiplication of $n/2 \times n/2$ matrices to derive a time bound for matrix multiplication given by the recurrence $T(n) = 7T(n/2) + O(n^2)$. In such a recurrence the 7 indicates the number of scalar multiplications in multiplying 2×2 matrices or the number of $n/2 \times n/2$ matrix multiplications in the recursive (divide-and-conquer) decomposition. Note that Strassen's method on 2×2 matrices can not be improved. We cannot multiply 2×2 matrices with say 6 (scalar) multiplications.

Now if we can multiply 3×3 matrices and k is the number of scalar multiplications required and we use a recursive decomposition of an $n \times n$ matrix into $n/3 \times n/3$ matrices we get a recurrence for the running time of this method of

$$T(n) = kT(n/3) + O(n^2).$$

The solution of this recurrence (master method) yields $T(n) = \Theta(n^{\log_3 k})$. Therefore if we compare the exponent $\log_3 k$ to $\lg 7$, then for $k = 21$ we have $\log_3 21 \approx 2.771 < 2.8073 = \lg 7$.

For that value of $k = 21$ or smaller a by 3 decomposition beats Strassen's method.

b. This is similar to part (a). If we decompose not by-2 or by-3, but by-t, where t now is $68, 70, 72$ we obtain that the running time for such a decomposition is given by the recurrence,

$$T(n) = kT(n/t) + O(n^2).$$

with an asymptotically tight solution given by $T(n) = \Theta(n^{\log_t k})$. Therefore if we compare the exponent $\log_t k$ to $\lg 7$, then for $t = 68$, $\log_{68} 132464 = 2.795128$, $t = 70$, $\log_{70} 143640 = 2.795122$, $t = 72$, $\log_{72} 155424 = 2.795147$. All three of them beat Strassen's exponent. The best of the three is the $t = 70$ split. Calculations used the Unix `bc` calculator and hopefully are correct! ■

Exercise 389.
(a) What's the best way to find the product $M_0 \times M_1 \times M_2 \times M_3$ where the matrices are of dimension 100×200, 200×500 , 500×10, and 10×1000 respectively? (In a prior problem, you had to deal with polynomials. The polynomials have now become matrices.)
(b) You are given 4 matrices M_0, M_1, M_2, M_3 and you are asked to determine the optimal schedule for the product $M_0 \times M_1 \times M_2 \times M_3$ that minimizes the number of operations (multiplications) involved. The dimensions of the four matrices are respectively 10×1, 1×2, 2×1, 1×10. What is the best (cheapest) schedule to multiply all the matrices together and compute $M_0 \times M_1 \times M_2 \times M_3$? What is the total cost for this schedule?
(c) You are given 4 matrices M_1, M_2, M_3, M_4 and you are asked to determine the optimal schedule for the product $M_1 \times M_2 \times M_3 \times M_4$ that minimizes the number of operations (addition/multiplication) involved. The dimensions of the four matrices are respectively 100×20, 20×500, 500×30, and 30×100. What is the best (cheapest) schedule to multiply all the matrices together and compute $M_1 \times M_2 \times M_3 \times M_4$? What is the total cost for this schedule?
(d) You are given 4 matrices M_1, M_2, M_3, M_4 and you are asked to determine the optimal schedule for the product $M_1 \times M_2 \times M_3 \times M_4$ that minimizes the number of operations (addition/multiplication) involved. The dimensions of the four matrices are respectively 100×50, 50×200, 200×50, and 50×10. What is the best (cheapest) schedule to multiply all the matrices together and compute $M_1 \times M_2 \times M_3 \times M_4$? What is the total cost for this schedule?

Proof. (a) May be the cost is 2,200,000. You find the schedule! Practice with the other ones, if you need to! ■

Exercise 390.

(a) Show that if we can invert a positive definite matrix A then we can invert an arbitrary matrix.
(b) For an $n \times n$ matrix A, the inverse of A is $i(A) = A^{-1}$ which is a matrix such that $i(A) \times A = A \times i(A) = I$, where I is the identity matrix (diagonals one, everything else zero). The cost of inversion of A is let it be $I(n)$. If we have a recursive approach that has cost

$$I(3n) \leq 20I(n) + 31n^2$$

what would the cost of $i(A)$ be ?

Proof.
(a) Consider matrix A that is invertible. Then $A'A$ is positive definite. Then $A^{-1} = (A'A)^{-1}A'$ is the inverse of A. Thus if $I(n)$ is the time to invert an arbitrary matrix and $J(n)$ is the time to invert a positive definite matrix B we have the following.

$$I(n) \leq 2M(n) + J(n),$$

where $M(n)$ is the cost of matrix multiplication for $A'A$ and also $(A'A)^{-1}$ and A' where $J(n)$ is the cost of the inverse of $(A'A)$. Note that if X be a symmetric, positive definite matrix $n \times n$.
(b) No solution provided. Master method. $I(n) = O(n^{\log_3^{20} n})$. ■

9.4 Miscellanea

Exercise 391.

SomeIT decides to introduce its own coinage with three different types of coins: 1 cent, 5 cent, and 6 cents. We would like to know what is the minimum number of coins we can use if we pay for an item worth n cents. Give an efficient algorithm that if given n as input, it prints as output the minimum set of coins that has value exactly n. Analyze the time and space requirements of your algorithm. Prove its correctness. For example, you can pay an item worth 30 cents by giving five 6-cent coins; other alternatives is six 5-cent coints, or thirty 1-cent coins, or say three 6-cent, two 5-cent, and two 1-cent coins.

Proof.

Let $m[i]$ be the minimum number of coins used to form i cents. Let $s[i]$ be the type of the last coin used to form i cents. Initially $m[1] = 1, m[2] = 2, m[3] = 3, m[4] = 4, m[5] = 1, m[6] = 1$ and $s[1] = s[2] = s[3] = s[4] = 1, s[5] = 5, s[6] = 6$. We use dynamic programming to solve this problem $m[i] = \min m[i-1], m[i-5], m[i-6] + 1$, and $s[i] = k$ if $m[i]$ got assigned the value of $m[i-k]$. The algorithm for finding the $m[.]$ and $s[.]$ arrays is as follows

```
          Find (m[], s[], n)
1. for(i=1;i<=n;i++)  //Forget about braces; indentation replaces them
2.   m[i] = n+1;  // MAX value
3.   s[i] = -1;   // MIN value
4. Initialize m[1..6] and s[1..6] as described previously;
5. for(i=7;i<=n;i++)
6.    x = m[i-6]+1;
7.    if (x < m[i])
8.      m[i]=x ; s[i] = 6;
9.    x = m[i-5]+1;
10.   if (x < m[i])
11.     m[i]=x ; s[i] = 5;
12.   x = m[i-1]+1;
13.   if (x < m[i])
14.     m[i]=x ; s[i] = 1;
15. PrintSolution(n);

         PrintSolution(n)
1.  c1=0; c5=0; c6=0; c=n;
2.  while (c>0)
3.    if s[c] == 1
4.       c1++;
5.    if s[c] == 5
6.       c5++;
7.    if s[c] == 6
8.       c6++;
9.    c = c - s[c];
10  print c1,c5,c6
\end{verbatim}
```

It is obvious that the running time of the algorithm and the amount of space used is $O(n)$. ■

Part IV

Trees of bounded height and depth

Chapter 10

Binary search trees: bounded height or not

10.1 Binary search trees: operations and properties

Exercise 392.
Delete the root from the following BST.

Proof.

■

Exercise 393.

CLRS [6] problem or influenced from.

a. Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which ones of the following sequences (one or more) could NOT be the sequence of nodes examined?

i. 2, 242, 301, 398, 330, 399, 363.

ii. 924, 229, 911, 244, 898, 258, 243, 363.

iii. 925, 202, 911, 240, 900, 245, 363.

iv. 2, 390, 383, 219, 265, 380, 379, 278, 363.

Justify your answer.

Proof. Sequence (i) cannot be a search sequence as 399 is in the left subtree of 398. Sequence (ii) cannot also be a search sequence as 243 is in the right subtree of 244.

■

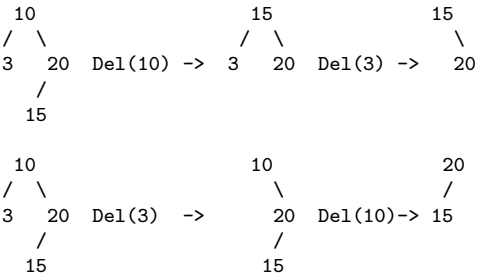
Exercise 394.
What are the minimal number of Traversals (among inorder, preorder, postorder) one can use to reconstruct a BST tree?

Proof.
Two traversals: inorder and one of preorder and postorder are needed. Inorder generates a sorted list of keys. The preorder points to the root (first element), whereas the postorder points to the root (last element) of the BST tree. After a split we continue recursively. $T(n) = O(n \lg n)$. ■

Exercise 395.

Is deletion from a BST commutative? That is if we delete x first and then y, is this the same (i.e. does it lead to the same tree) as if we first delete y and then x?

Proof.
No it is not. See example below.



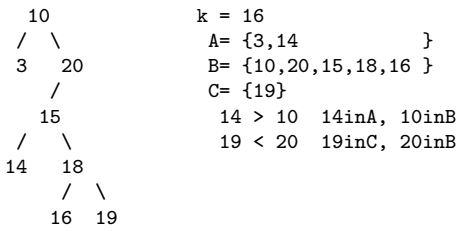
■

Exercise 396.

Follow the search path for k to a leaf. Then split a BST into three sets: B the set of keys of the path, A the set of keys on the left of the path, and C the rest of the keys (on the right of the path).

Is it true that $\forall a \in A, \forall b \in B, \forall c \in C, a \leq b \leq c$?

Proof.
No it is not.



■

Exercise 397.

TRUE or FALSE? Anything other than a TRUE or FALSE is wrong! References are to a tree containing n keys and of height h. How h varies with n depends on the individual question.

1. Every binary search tree on n nodes has height $O(\lg n)$.
2. BSTInsert is $O(\lg n)$.
3. BSTInsert is $O(h)$.
4. BSTInsert is $O(n)$.

Proof. 1. FALSE?
2. FALSE
3. TRUE
4. TRUE

■

10.2 Balanced binary search trees: operations and properties

Exercise 398.

TRUE or FALSE? Anything other than a TRUE or FALSE is wrong! References are to a tree containing n keys and of height h. How h varies with n depends on the individual question.

1. Every binary search tree on n nodes has height $O(\lg n)$.
2. An AVL tree has height between $\lg n$ and $\lg n + 10$.
3. AVLInsert is $O(n)$.
4. AVLInsert is $O(\lg n)$.
5. AVLInsert is $\Omega(n)$.

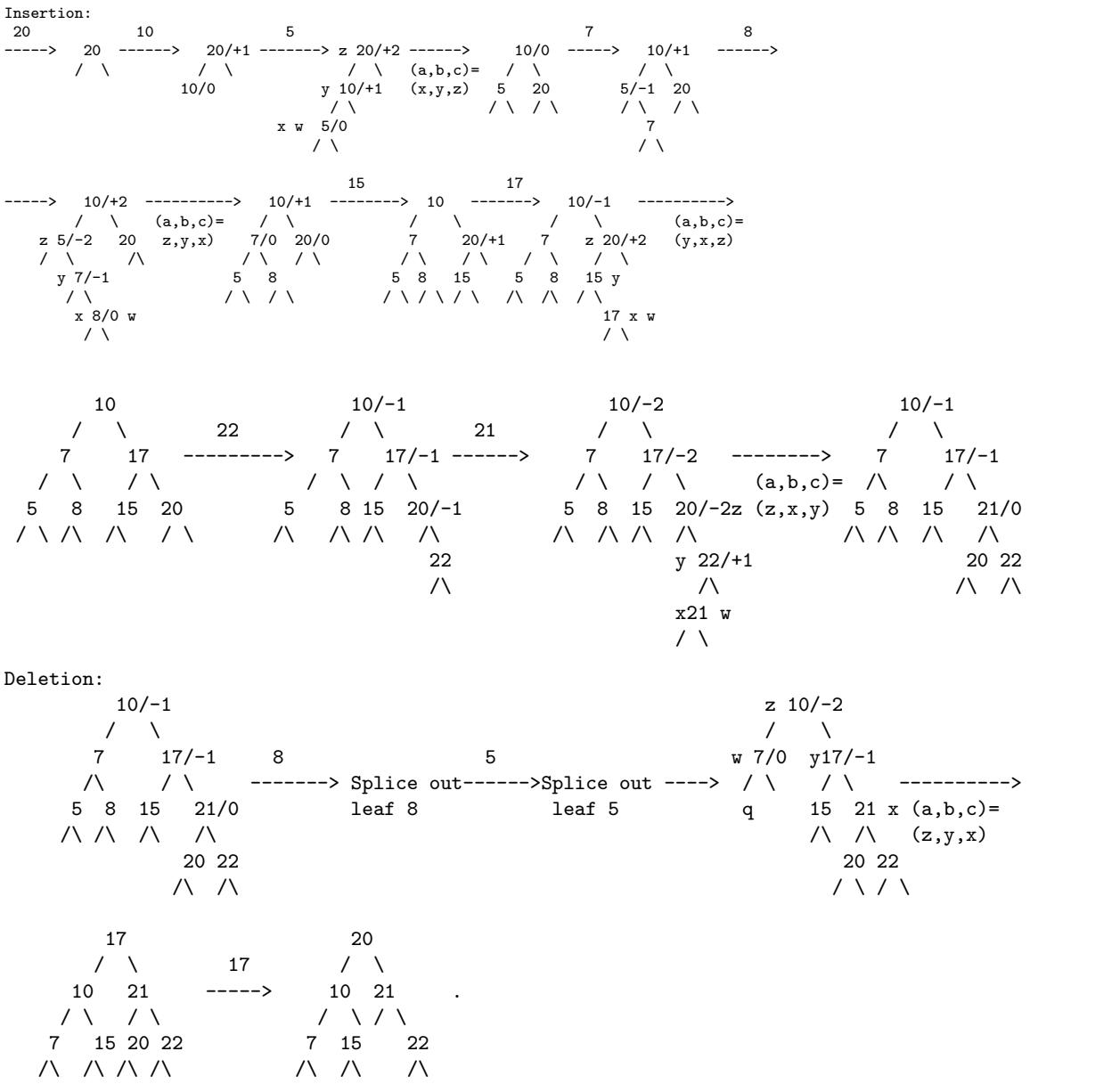
Proof. 1. FALSE?
2. FALSE!
3. TRUE
4. TRUE
5. FALSE. ■

Exercise 399.

$$20, 10, 5, 7, 8, 15, 17, 22, 21$$

Starting with an empty AVL tree insert the following keys in the order shown (leftmost key first). Show the state of the tree (a) after the key is inserted and before a rebalancing takes place, and (b) after the key is inserted and a rebalancing operation caused by the insertion is completed. (c) Following the insertions, delete $8, 5, 17$. Then delete $8, 5, 17$.

Proof.



■

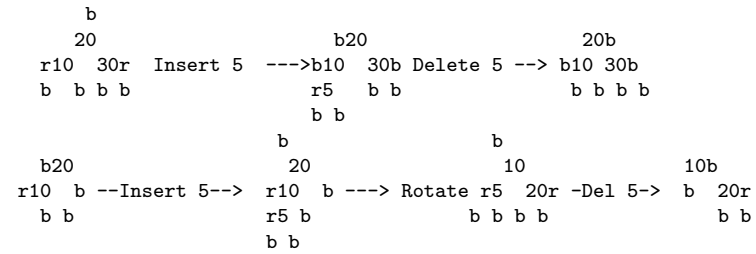
Exercise 400.

In any red-black tree T an Insert(T,x) followed by a Delete(T,x) leaves the tree in exactly the same state it was before x was Inserted and Deleted.

Note. x is not in tree T before the insertion is performed. "same state" means that nodes have same keys and same children/orientation before and after the two operations are performed.

Proof.

False. A counterexample follows. In the former the tree remains the same, in the latter the tree changes (note that "changes" is independent of coloring as per note).



■

Exercise 401.
For an Insertion operation into an AVL tree, where the tree maintains n keys prior to the Insertion operation, what is an upper bound on the number of single rotations that need to be performed to make sure it is still an AVL tree after the Insertion operation?

Proof. Two. A single or a double rotation needs to be performed; the latter is two single rotations! ■

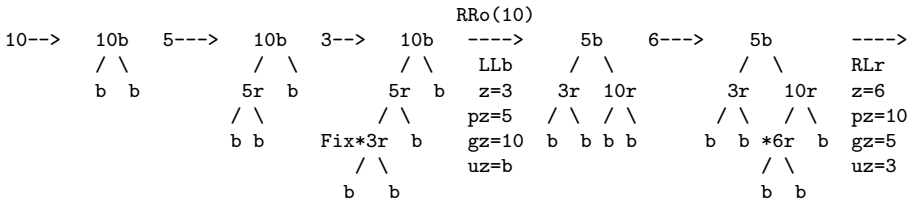
Exercise 402.
For the tree of a prior problem after deletion, label each vertex with a number from $1, \ldots 6$ that shows the order this vertex is visited in (i) a preorder, (ii) postorder, and (iii) inorder tree traversal.

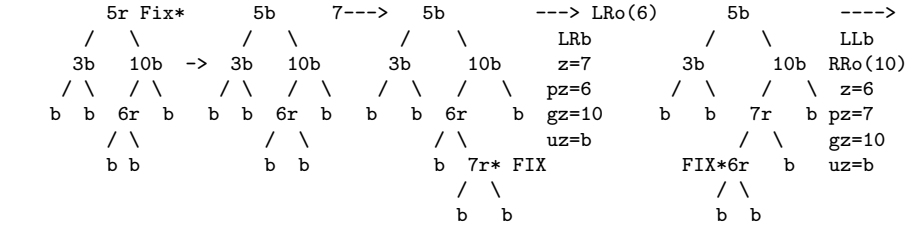
Proof. Skipped. ■

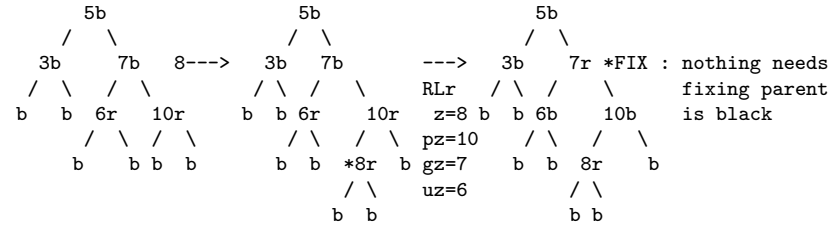
Exercise 403.

Starting with an empty red-black tree Insert one after the other the following integer keys: 10, 5, 3, 6, 7, 8.

Proof.







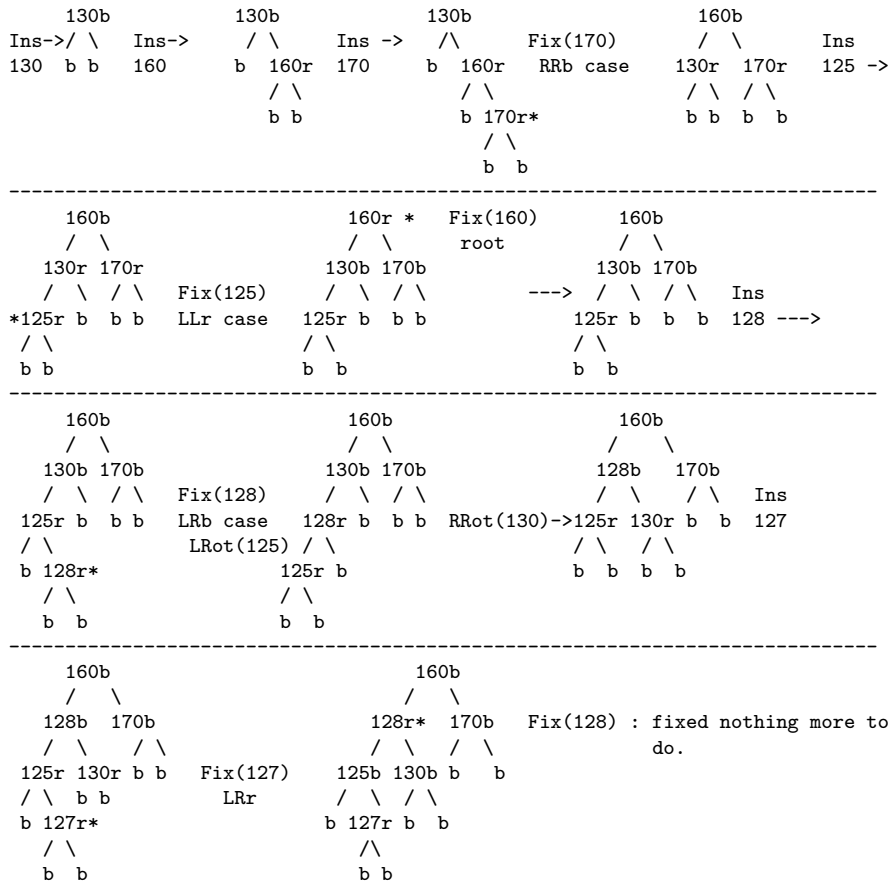
■

Exercise 404.

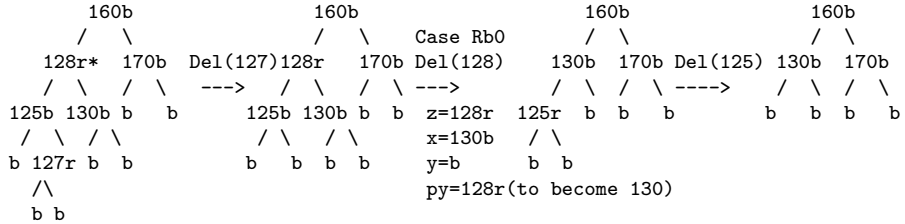
Starting with an empty red-black tree insert one after the other the following integer keys: 130, 160, 170, 125, 128, 127. Show all steps including the intermediate steps before and after a FIX-Insert operation (if such is necessary).

We use the notation introduced the class, where NULL nodes are represented not collectively as a single SENTINEL node, but as individual nodes labeled b. Every other node has a key value and r or b on its right. A $*$ indicates the node to be FIXed (i.e. it corresponds to a red node with a red parent).

Proof.



For the deletion part,



■

10.3 Miscellanea

Exercise 405.

Before you solve this problem understand carefully the semantics of the operations. In particular in Delete(S,z) z is a pointer to an element NOT a KEY VALUE. Let n be the size of the data structure before the operation is performed.

(a) Insertion of a key x in a data structure S, i.e. operation Insert(S,x).
(b) Deletion of a record of S pointed by pointer z, i.e. operation Delete(S,z),
(c) Search of a key x, i.e. operation Search(S,x),
(d) FindMin of S i.e. operation FindMin(S),
(e) ExtractMax of S i.e. operation ExtractMax(S),

Note: All keys in S are distinct. If S is sorted, it is sorted in increasing order. If a linked list (singly or doubly linked) is used, assume the existence of a head pointer; the existence or not of a tail pointer depends on the description of the data structure. **Express the running time in terms of the number of elements n in the data structure just before the application of the operation**.

Proof.
(1) All $O(n)$.
(2) All $O(\lg n)$. ■

Exercise 406.
Starting with an empty binary search tree insert 1, then 5, then 3, then 4, and finally 2. Show the tree after each insertion. (We don't ask you to build an AVL tree.) Then perform inorder, preorder and postorder traversals on the resulting binary search tree. List the nodes in the order visited for each traversal.

Proof. Practice makes perfect! You practice. ■

Exercise 407.

Argue that since sorting n keys using comparisons, takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary sequence of n keys takes $\Omega(n \lg n)$ time in the worst case.

Proof. Suppose we can construct a binary search tree on n nodes in T time. Then we can run an inorder traversal of the BST in $\Theta(n)$ time and thus get in total time of $T + \Theta(n)$ the nodes in sorted form, since we know that an inorder traversal of a BST produces a sorted sequence of the keys stored in the tree.
If $T = o(n \lg n)$ this means we can sort n keys in $o(n \lg n) + \Theta(n) = o(n \lg n)$ time: first build a binary search tree and then perform an inorder traversal of its nodes. However we know from the lower bound for sorting that comparison-based sorting can not be performed in $o(n \lg n)$ time, i.e. it requires $\Omega(n \lg n)$. Therefore we can not have $T = o(n \lg n)$, i.e. we must have $T = \Omega(n \lg n)$.
Thus any algorithm for creating a BST of n keys must have running time (at least) $T = \Omega(n \lg n)$. This bound is tight for AVL trees (and also red-black trees not covered in class but material related to them is available in the textbook) since the time an AVLTreeInsert requires to insert n keys in an empty tree is also tight $\Theta(n \lg n)$. ■

Exercise 408.

Give a data structure S that implements a MIN priority queue (operations ExtractMin and Insert are implemented efficiently) and a MAX priority queue (operations ExtractMax, Insert) efficiently and simultaneously (i.e. in worst case $O(\lg n)$ time). You therefore need to implement efficiently three operations: Insert(x), ExtractMin, and ExtractMax.

A complete solution first describes the data structure S, and then presents the algorithms for the three operations. Finally an analysis of the worst case running time is performed and shown to be $O(\lg n)$, where n is the number of elements in the data structure before the operation is performed.

Note: You are allowed to use without proof all operations that have been introduced in this class. If for example you wish to use MaxHeapify, you do not need to present its code, you just call it using the class described interface MaxHeapify(A,i).

Proof. Use an AVL tree as the data structure. Insert is an AVL tree insert, ExtractMin, ExtractMax, are implemented by first issuing a FindMin or FindMax and then a Delete on the found min or max key. Insert, Delete on AVL trees are in the notes/textbook, and so are FindMin, FindMax. Worst case performance of all these operations on an AVL tree of n nodes is $O(\lg n)$. ■

Exercise 409.
You want to sort the n keys stored into an AVL tree. How fast can you do it? Explain.

Proof. Linear time to the number of keys $\Theta(n)$. Run an inorder travesal! ■

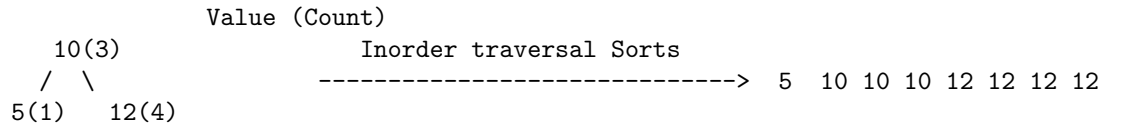
Exercise 410.
(Precursor to the previous problem.) You have n keys and you want to use an AVL tree to sort them. Do so and explain so.

Proof. Insert the keys into an AVL tree and then run the algorithm of Problem 5 above. The latter step is $O(n)$. The former step is $O(\lg n)$ per key or $O(n \lg n)$ total. Total running time is $O(n \lg n)$ as well. ■

Exercise 411.

We have n keys that take k distinct values (integer or not). Give an algorithm that sorts those keys in $O(n \lg k)$ time. (Assume $n \gg k$.)

Proof. Use an AVL treek of k nodes. Each node has a linked list attached to it containing all the keys with that value. Radix-sort does not (necessarily) work. We don't know the range of the key values. Let alone the fact that the keys might not be integer. Build an AVL tree and for each node maintain also a counter (how many keys have that specific value). Maintaining this AVL is $O(\lg k)$ per operation as the number of distinct values is k. Thus run the algorithm in Problem 6, noting that only k keys are inserted into the AVL tree, even if we run n Search operations. In addition, when you do the inorder traversal and you are about to print the value of node x you print that value as many times as indicated by the counter. (See example below.)



■

Exercise 412.

CLRS problem or based on one [6].
Given two strings $a = a_0a_1 \dots a_k$ and $b = b_0b_1 \dots b_m$, where each a_i and b_j belongs to some ordered set of characters (e.g. english alphabet, binary set) we say that string a is lexicographically less than string b if either

1. there exists an integer j, where $0 \leq j \leq \min(k,m)$, such that $a_i = b_i$ for all $i = 0, \dots, j-1$ and $a_j < b_j$, or
2. $k < m$ and $a_i = b_i$ for all $i = 0, \dots, k$.

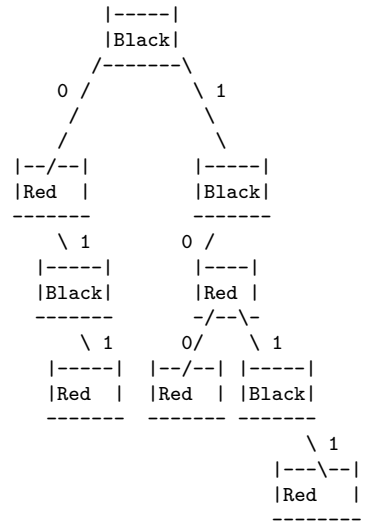


Figure: A binary radix tree.

For example, 010 is lexicographically less than 0100 which is less than 0101. A binary radix tree is a tree built out of a collection of binary strings and one is depicted in the attached Figure. In a binary radix tree an edge leading to a left child can be labeled 0 and to a right child can be labeled 1. A node is either black or red; a black node indicates that the string (using the implied edge labels) corresponding to the path from the root to the black node is NOT in the tree, whereas a red node indicates that the string is in the tree. When searching for a in such a tree we traverse a left child at depth i if $a_i = 0$, and a right child if $a_i = 1$. If a red node is reached, a is in the tree, otherwise it is not. Searching for 101 leads to a black node and for 1011 to a red node: the former string does not exist, the latter does.

Let A be a set of distinct binary strings whose total length sum is n. Show how to use a radix tree to sort A in lexicographic order in $\Theta(n)$ time. Sorting the tree above with $n = 13$ would give the strings 0, 011, 10, 100, 1011 in lexicographic order.

Proof. First insert the strings and then perform a preorder traversal. In the preorder traversal a string is printed only for the red nodes (not for the black nodes that do not correspond to strings).
Note that the preorder is the correct traversal as a node is printed before its descendants. A printed node x is a red node and the string corresponding to that node should be printed before the strings corresponding to its descendants (in the left or right subtree of x). Strings corresponding to nodes of x's left subtree must be printed before the ones corresponding to nodes of x's right subtree as up to node x both strings are the same, but just after x the strings in the left subtree have a leading zero and the ones in the right subtree have a leading 1; the former must be printed before the latter ones. Preorder traversal guarantees that.
Running time of algorithm is $O(n)$, as insertion takes time proportional to length of individual string being inserted and sum of all strings is n. ■

Exercise 413.
We have three AVL trees T_1, T_2, T_3 of a, b, c keys each. We would like to create one AVL tree T that stores all the keys of T_1, T_2, T_3 in it i.e. $T = T_1 \cup T_2 \cup T_3$ of size $a+b+c$. Give an algorithm that can do that whose worst-case running time is $O(a+b+c)$. Explain.

Proof.
3 in-order plus intelligent AVL creation after a double merging. ■

Exercise 414.
We have a complete binary tree that is a BST. Edges are from a parent to its children. Do a DFS on it starting from its root. Collect topological search information in list L.

(a) Which node is inserted into L first?
(b) Which node is inserted into L last?

Proof.
Left most leaf node, and root respectively. ■

Exercise 415.

Suppose we have a BST with node x having left and right childrens and that the successor of x is y. Node's x deletion requires $key(x) = key(y)$ and then spliceout of y.

What if there is another data structure that points to y or to y's value. During the spliceout of y the value can be lost (in fact it will have been lost earlier during the $key(x) = key(y)$ step). How can we make sure this is not so?

Proof.

Store the key values into Key array. Thus let Key[k] store key(y). Then y used to store an index to Key rather that key(y) i.e. y used to store k rather key(y). The step $key(x) = key(y)$ will copy indexes i.e. the key field of x would store index k. Moreover, any data structure that used to store k will be left untouched.

■

Exercise 416.
Show that a BST tree T_1 can be transformed into anothe BST tree T_2 using $O(n)$ rotations.

Proof.
If a node x has a left subtree, we can right rotate around x. The total number of nodex like x is no more than n. Likewise for left rorations.

■

Exercise 417.
Build a data structure S that supports in $O(\lg n)$ the following operations: Min(S), Max(S), ExtractMin(S), ExtractMax(S), Insert(S,k).

Proof.
Build an AVL tree or a red-black tree T. Then $S = T$.

■

Exercise 418.

We define a Binary-Search-Tree (BST) with n nodes to be **nicely-balanced** if its height is $O(\lg n)$. We have two arbitrary (not necessarily balanced nor nicely balanced) BST trees T_1 and T_2 with n_1 and n_2 nodes, where $n = n_1 + n_2$. Assume all n nodes have distinct key values. We need to create a **nicely-balanced** BST T which contains all n keys. That is, T is the union of the key values of T_1 and T_2 and is nicely-balanced. Give an efficient algorithm for this task whose worst-case running time is faster than $O(n \lg n)$. Prove your clai ms.

Proof.

```
A1=Inord(T1);
A2=Inord(T2);
A=Merge(A1,A2);
T=Build(A,0,n-1);

 Build(A,l,r)
1. r = A[(l+r)/2]
2.  x = build node with key r;
3. left(x)= Build(A,l,l+r / 2 -1)
4. right(x)= Build(A,l+r / 2 +1,r)
5. ret x;
```

■

Exercise 419.

Johnny Be Good is given a binary search tree T and is ordered to perform an in-order traversal to it. The library code for in-order traversal is not working, but the code for MIN, MAX, Predecessor ($pred(z)$), Successor ($succ(z)$) is working fine.

Help him implement in-order by using some of the four functions mentioned.

Proof.
Do a MIN and then $n-1$ Successor invocations. The MIN and a number of Successor operations equal to the number of keys minus one of the left subtree of the root will complete the inorder traversal of the left subtree of the root. The last successor results in the reach of the root and the a recursive traversal of the right subtree of the root. Total time is $\Theta(n)$. Every edge is traversed twice either on the way up or down. For the MIN operation we have traversals down the tree. The two cases of importance of Successor as in Successor(z) is when z has a right child, or it does not have a right child (and neither it is the left child of its parent $p(z)$ as for that case the successor of z is trivial).
(a) z has a right child: edges are to be traversed on the way down to find $succ(z)$ in the subtree rooted at $right(z)$. When $succ(z)$ is located, that's where the next invocation will start later.
(b) z has no right child, and the loop trying to find an ancestor x of z who is the left child of its parent $p(x)$ is to be used. Edges are traversed upwards and reaching $p(x)$ we are done and ready for the next successor operation that will start later.
In every call for $succ(z)$ an edge will traverse reaching to $succ(z)$. There are as many edges as nodes identified for $succ(z)$ (the first node was identified as MIN), one such edge leading to $succ(i)$ per invocation. An inductive argument on the height of the tree can formalize the arguments above.

■

Exercise 420.

You are given two sequences of length n, $A[0..n-1]$ and $B[0..n-1]$. The elements of A and B are comparable. Sequence A may or may not contain duplicate elements (elements that appear more than once). Likewise for B. Give an algorithm that determines whether $A \cap B = \emptyset$ or not. The algorithm is a decision algorithm i.e. it outputs a YES or NOT. YES means intersection is empty, NO means otherwise. Your algorithm runtime must be $o(n^2)$.

Proof.
Insert the elements of A into an AVL (or red-black tree) tree T in $O(n \lg n)$. The search each element of B in T in $O(n \lg n)$. If a search is successfule answer NO; otherwise if no search generates a NO, say YES. ■

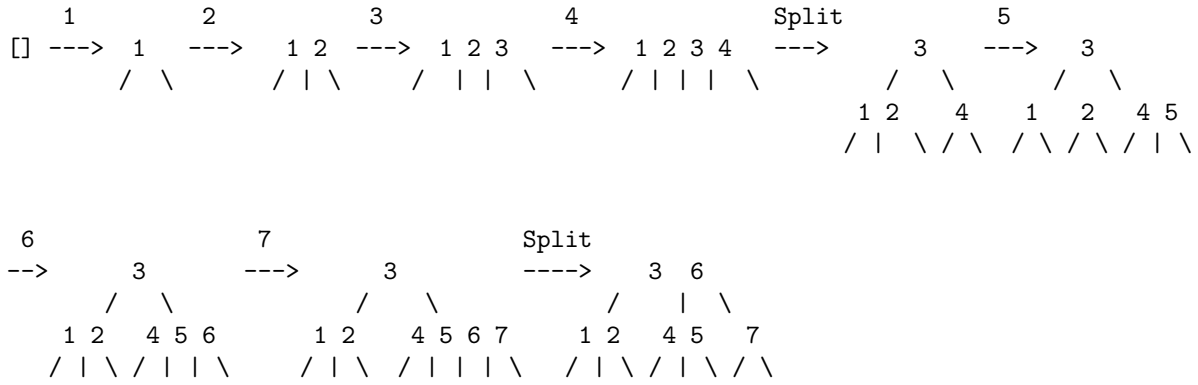
Chapter 11

Non-binary search trees: bounded depth

11.1 2-3-4 trees aka B-trees with $B = 4$

Exercise 421.
Insert 1, 2, 3, 4, 5, 6, 7. Show the steps. Then in Problem 2 delete 7.

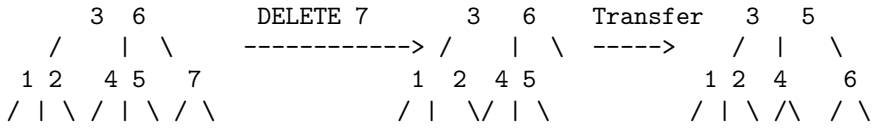
Proof.



■

Exercise 422.
Delete 7 from the tree of the previous problem.

Proof.

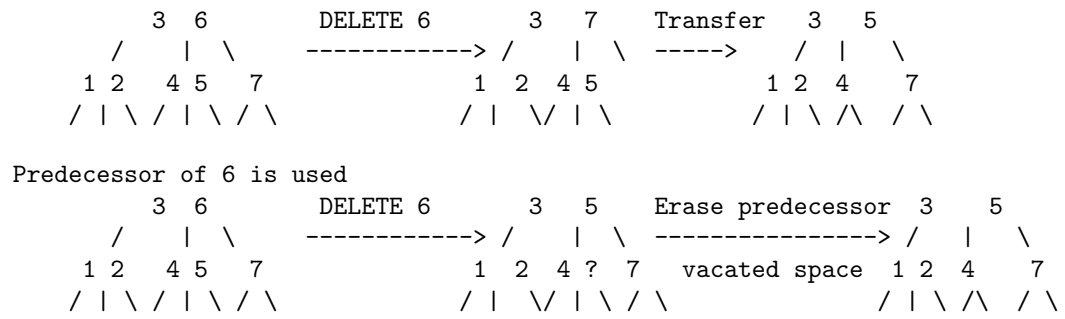


■

Exercise 423.

Delete 6 from the tree of the previous to the previous problem.

Proof. Either the successor or predecessor of z can be used!



■

Exercise 424.

Find the minimal and the maximal height for a 3-ary tree with n keys. The root is allowed to have two children. All NULL at the same level.

Proof.
The minimal 3-ary tree has 2 nodes at level 1, and $2 \cdot 3^{h-1}$ at level h. The $n+1$ NULLs at level h must be accommodated by the number of nodes available i.e. $n+1 \geq 2 \cdot 3^{h-1}$. Solving for h we have

$$h \leq \frac{\lg(n-1)-1}{\lg 3} + 1.$$

This is Theorem 1 of page 13 with $a = b = 3$. The maximal 3-ary tree has 3 nodes at level 1, and 3^h at level h. Thus $n+1 \leq 3^h$ leads to

$$h \geq \frac{\lg(n+1)}{\lg 3}.$$

■

Part V

Strings

Chapter 12

String matching

12.1 Naive, Boyer-Moore, KMP

Exercise 425.
Let the pattern p be

ABRACADABRA

and the text t be

ABRACAZABRACADABRMABRACADABRA

(a) Illustrate the working of the Boyer-Moore algorithm. What is the total number of comparisons (text vs pattern) performed?
(b) Illustrate the working of the Knuth-Morris-Pratt algorithm. What is the f[] array. What is the total number of comparisons (text vs pattern) performed?
(c) Which of the two algorithms is faster for this pattern? Explain.

Proof.
(a) For Boyer-Moore we have.

```
= is a match    ; comparison performed
x is a mismatch ; comparison performed
ABRACAZABRACADABRMABRACADABRA
ABRACADABRA
      x====
       ABRACADABRA
                 x===========
                  ABRACADABRA
```

Total number of comparisons performed to derive a match is 17.

(b) For KMP we have.

```
   A B R A C A D A B R A
f=[0 0 0 1 0 1 0 1 2 3 4 ]

= is a match    ; comparison performed
x is a mismatch ; comparison performed
ABRACAZABRACADABRMABRACADABRA
ABRACADABRA                    : Z != D; shift Pat to match the A before Z;
======x                          do not compare A vs A again!
     ABRACADABRA               : Z != B; shift Pat right once
      x
      ABRACADABRA              : Z != A; shift Pat right once again.
      x
       ABRACADABRA             : Tough luck! M !=A in the very last character
       ==========x             of the pattern Shift pattern 7 positions to the
              ABRACADABRA      :right and  continue with M vs A a mismatch!
                 x
                 ABRACADABRA   : Shift pattern 3 pos rights.Still touch luck
                 x
                  ABRACADABRA  : Shift right once and all of luck. A Match!
                  ===========
```

33 comparisons performed: Count all x and = above.

(c) 17 is better than 33. Boyer-Moore wins. ■

Exercise 426.
(a) Give an input T, P for which NaiveMatching exhibits worst-case performance over a binary alphabet of letters a, b.
(b) Give an input T, P for which Boyer-Moore exhibits worst-case performance over a binary alphabet of letters a, b.
(c) Give an input T, P for which Boyer-Moore exhibits best-case performance over a binary alphabet of letters a, b.

Proof.
(a) Let $P = a^{m-1}b$ and $T = a^n$. The it is obvious that m comparisons need to be performed to conclude that P does not match the text starting at position i, $i \leq n - m$. Thus the total number of character comparisons is $O(m(n-m)) = O(nm)$, as required.
(b) $P = ab^{m-1}$ and $T = b^n$. Reasons follow similarly to case (a).

■

Exercise 427.
(a) Give an input T, P for which NaiveMatching exhibits worst-case performance over a binary alphabet of letters a, b.

(b) Give an input T, P for which Boyer-Moore exhibits worst-case performance over a binary alphabet of letters a, b.

(c) How many comparisons does BM perform if $P = a^m$ and $T = (a^{m-1}b)^{n/m}$. Assume m/n and a^m means m consecutive occurrences of a.

Proof. Practice! ■

Exercise 428.
How many comparisons does Boyer Moore perform if $P = a^m$ and $T = (a^{m-1}b)^{n/m}$. Assume m/n and a^m means m consecutive occurrences of a.

Proof.
n/m comparisons. Since b is not part of the Pattern the whole text is skipped right. This is repeated n/m times.

■

Exercise 429.
Let the pattern P be and the text T be respectively

```
ABRACADABRA
ABRACAVABRACALABRAABRACADABRA
```

(a) Illustrate the working of the Boyer-Moore algorithm. What is the total number of comparisons (text vs pattern) performed?
(b) Illustrate the working of the Knuth-Morris-Pratt algorithm. What is the f[] array. What is the total number of comparisons (text vs pattern) performed?
(c) Which of the two algorithms is faster? Explain.

Proof. Practice! ■

Exercise 430.
Give an example of a pattern and a text for which Boyer-Moore exhibits worst case $O(nm)$ performance (n is text length, and m is pattern length). Explain.

Proof. Practice! ■

Exercise 431.
Let P=TRAINWRECK and text T=TRAMYWRECKZZTRAINWRECK. Explore Boyer-Moore.

Proof.

```
17 character comparisons of which 10 for the match
between the pattern and text. An equal below
indicates a character match, an exclamation mark
a mismatch!
TRAMYWRECKZZTRAINWRECK
    !=====
TRAINWRECK
             !
     TRAINWRECK
            ==========
            TRAINWRECK
```

■

Exercise 432.

Explain KMP for T=ALGGALGGALT and P=ALGGALT

Proof.
There is a mismatch between P and T at $P[j] != T[i]$ where $i = j = 6$, yet $P[0..j-1] = T[i-j..i-1]$.

```
   0 1 2 3 4 5 6 7 8 9 0
T: A L G G A L G G A L T
   = = = = = = !
P: A L G G A L T
   0 1 2 3 4 5 6
```

Action need: find longest prefix of $P[0..j-1]$ that is a suffix of $P[1..j-1]$ and record its length in $f[j-1]$. Note that we state suffix of $P[1..j-1]$ and not suffix of $P[0..j-1]$ because then the answer would be the nonsense $P[0..j-1]$.

```
     <--------->
     A L G G A L  ===> j=6  f[j-1]=f[5]= 2
     ---
j-f[j-1]     ---
             f[j-1]
```

We then shift pattern to the right $j - f[j-1]$ characters.

```
   0 1 2 3 4 5 6 7 8 9 0
T: A L G G A L G G A L T
   = = = = = = !
P:         A L G G A L T  P Shifted j-f[j-1]=6-2=4 char
```

And reset i to i and j to $f[j-1]$

```
   0 1 2 3 4 5 6 7 8 9 0
T: A L G G A L G G A L T
           = = ?          reset i <- i      =>  i <- 6
P:         A L G G A L T  reset j <-f[j-1]  =>  j <- 2
           0 1 2 3 4 5 6
```

■

Exercise 433.
Find f for P=ABCRACADBRA, P=ABABABABCA.

Proof.
Form $f[.]$ where $f[j]$ is the longest prefix of $P[0..j]$ that is a suffix of $P[1..j]$.

```
      0 1 2 3 4 5 6 7 8 9 10
P:    A B R A C A D A B R A
f:    0 0 0 1 0 1 0 1 2 3 4

P:    A B A B A B A B C A
f:    0 0 1 2 3 4 5 6 0 1
```

■

Exercise 434.
Find f for $P = ABACABA$.

Proof.
Form $f[.]$ where $f[j]$ is the longest prefix of $P[0..j]$ that is a suffix of $P[1..j]$.

```
    0 1 2 3 4 5 6
P:  A B A C A B A
f:  0 0 1 0 1 2 3

    0 1 2 3 4 5 6
    A B A C A B A   i j
    0               1 0
    0 0             2 0
    0 0 1           3 1
    0 0 1 0         4 0
    0 0 1 0 1       5 1
    0 0 1 0 1 2     6 2
    0 0 1 0 1 2 3   7 3
```

■

12.2 Rabin-Karp's fingerprinting method

Exercise 435.

Working modulo 11, how many spurious hits does the Rabin-Karp algorithm encounter in the text T below when looking for pattern $P = 84$ (assume that the algorithm exits after the first match)?

2718281828459045

Proof.

```
27 % 11 = 5                     P=84 % 11 = 7
71 % 11 = 5
18 % 11 = 7 --->spurious hit
82 % 11 = 5
28 % 11 = 6
81 % 11 = 4
18 % 11 = 7 --> spurious hit
82 % 11 = 5
28 % 11 = 6
84 % 11 = 7 ---- MATCH
```

■

Exercise 436.

Same as before but $P = 35$ and work modulo 10 now.

Proof.

```
00000000011111111112 :Position inder read top to bottom
01234567890123456789
                     : Hash of pattern % 10 is 5
27182818284590452536
          =          : Hash of 45 @ position 10,11 is 5
                       Compare naively 4 to 3 to detect mismatch
              =      : Same thing again
                 =   : Hashes are the same; Two more comparisons of
                       3 vs 3 and 5 vs 5 and detect a match.
```

■

Exercise 437.

In Rabin-Karp, let the pattern be $p = 35$ and all arithmetic operations be performed modulo q where $q = 10$. Are there any spurious (false) matches if we search for the pattern p in text t shown below? What is the number of character comparisons (not fingerprint!) performed to find all occurrences of the pattern in the text?

$$27182818284590452353 6$$

Explain.

Proof.

```
Search pattern P=35  with fingerprint function   % 10
       00000000011111111112 :Position inder read top to bottom
       01234567890123456789
                            : Hash of pattern % 10 is 5
       27182818284590452353 6
                 =          : Hash of 45 @ position 10,11 is 5
                              Compare naively 4 to 3 to detect mismatch
                     =      : Same thing again
                        =   : Hashes are the same; Two more comparisons of
                              3 vs 3 and 5 vs 5 and detect a match.
```

Total number of character comparisons is 4 plus all fingerprinting operations. ■

12.3 Miscellanea

Exercise 438.

Approximate String Matching. Match $P[0..m-1]$ with $T[0..n-1]$ with at most k errors. We can formulate this problem as the Levenstein distance between two strings P and T. The string operations allowed (the 'errors') are Insert, Delete, and Replace (a single character with another one). Thus for `kitten` we can get `mitten` by replacing one character, for `bidden` and `biden` an insertion of a d is required or a deletion of a d (other way around). A dynaming programming solutions requires $O(nm)$ time for a brute-force solution. Build a matrix M of geometry $m \times n$ to record information about a prefix $P[1..i]$ of the pattern to a suffix of the text $T[1..j]$ at position j and record this information at $M[i,j]$. A match is reported if $M[m,j] \leq k$.

Find the Levenshtein distance between $P[1..m] = survey$ and $T[1..n] = surgery$.

(b) Show matrix M for the strings of example (a) above.

Proof.
See below.

		T[1:n] n=7						
		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
P	s	1	0	1	2	3	4	5
[1:m]	u	2	1	0	1	2	3	4
	r	3	2	1	0	1	2	3
m=6	v	4	3	2	1	1	2	3
	e	5	4	3	2	2	1	2
	y	6	5	4	3	3	2	2

■

Exercise 439.

The prime number theorem states that the number of primes less than x is about $x/\ln x$. Therefore the density of primes is $1/\ln x$. In other words if we pick uniformly at random an integer z between 1 and N, with probability $1/\ln N$ we choose a prime number, and with probability $1-1/\ln N$ we choose a composite number.

Let $P = P_m \dots P_1$ be an m-bit integer. Let $Q = Q_m \dots T_1$ be an m-bit integer. Then $P = P_m 2^{m-1} + \dots + P_1$ and $Q = Q_m 2^{m-1} + \dots + Q_1$.

(a) Is $P \leq 2^m$? Explain. Is $Q \leq 2^m$? Explain.

(b) If $p/(P-Q)$ then show that $P \bmod p \equiv Q \bmod p$. Then show that if $P \bmod p \equiv Q \bmod p$, then $p/(P-Q)$.

(c) Show P or Q can have at most m distinct prime divisors.

(d) Let us pick a random number prime number p between 1 and $R = 4n^2 m \ln 4n^2 m$, where $n > m$. Show that such a p divides $P-Q$ with probability at most $O(1/n^2)$.

Proof.

(a) Since $P = P_m \dots P_1$ we have that $P = P_m 2^{m-1} + \dots + P_1$. Since $P_i \leq 1$, we thus have that $P \leq 2^{m-1} + \dots + 1 = 2^m - 1 < 2^m$. The same applies for Q as well, i.e. $Q < 2^m$.

(b) Let $P = ap + b$ and $Q = cp + d$, where $0 \leq b < p$ and $0 \leq d < p$. Since $P - Q = (a-c)p + (b-d)$ we have the following if $p/P-Q$: (i) $p/(a-c)p$ since the latter is a multiple of p, and (ii) $p/P-Q$ by assumption (which also implies $P \equiv Q \pmod p$). Thus by way of (i) and (ii), p must divide the difference of $P-Q$ and $(a-c)p$ i.e. $p/(b-d)$. Since b,d are less than p the only way p can divide their difference which is less than p is for $b-d$ to be 0. Thus $b = d$. But then $P \bmod p$ and $Q \bmod p$ are equal to b and d respectively and equal to each other i.e. $P \bmod p = Q \bmod p$. This further implies that $P \bmod p \equiv Q \bmod p$. If $P \bmod p \equiv Q \bmod p$, by definition $p/P \bmod p - Q \bmod p$, i.e. $p/(b-d)$. Then $p/(a-c)p + (b-d)$ and thus $p/P-Q$ as needed.

(c) Either P or Q by way of (a) is less than 2^m, i.e. P,Q are m-bit integers. Let $P = g_1 \dots g_k$ be the decomposition of P into primes. g_i may or may not be distinct. What we do know about g_i is that $g_i \geq 2$ for all i. Thus $P = g_1 \dots g_k \geq 2^k$. Since $P \leq 2^m$, we have $2^k \leq P \leq 2^m$ and thus $k \leq m$, i.e. P can have no more than m prime divisors (duplicate or not).

(d) The number of prime numbers between 1 (or 2) and R is by the prime number theorem quoted equal to $R/\ln R = 4n^2 m \ln(4n^2 m) / \left(\ln(4n^2 m) + \ln\ln(4n^2 m)\right)$. The denominator $\left(\ln(4n^2 m) + \ln\ln(4n^2 m)\right) \leq 2\ln(4n^2 m)$. Therefore $R/\ln R \geq 4n^2 m \ln(4n^2 m)/2\ln(4n^2 m) \geq 2n^2 m$. If P,Q are chosen so that $P,Q < 2^m$, then $P - Q < 2^m$ so the difference is divided by no more than m prime divisors (and they can be no more than m distinct prime divisors as well). The probability that the chosen p in the range 2 to R is one of those m divisors is no more than $m/2n^2 m \leq 1/(2n^2)$. This is indeed $O(1/n^2)$. ∎

Exercise 440.

In Rabin-Karp let P be the pattern of length m and T be the text of length n. Choose p as in part (d) of the previous problem.
(a) Show that the probability a spurious hit occurs in position i of the text is $O(1/n^2)$.
(b) Show that the probability a spurious hit occurs anywhere in the text is $O(1/n)$.
(c) Show that the expected running time of the algorithm is $O(n)$.

Proof.
q below is the prime number involved in all $\bmod$ computations. This prime number q was chosen uniformly at random in the intervale $2..N$ and thus has properties such as those described in parts (a)-(d) of the previous problem. P is the pattern which we are matching to text T. The fingerprint of the pattern P will be denoted by P'. The m-character text at position i (i.e. the m characters starting at position i will be denoted by t_i and its fingerprint by T_i).
(a) The probability that we have a spurious hit at position i is the probability that $P' = P \bmod q$ is equal to $T_i = t_i \bmod q$. By part (b) of the previous problem we have that $q/P - t_i$. This however, part (d) of the previous question and the choice of the prime number q (name p in the previous problem) can occur with probability at most $1/n^2$.
(b) Note that we assume in the remainder that the algorithm exits at the first successful match (standard algorithm behavior of the notes) and thus we analyze the running time under such a case or the end of text is reached (with no match). The probability that we have a spurious hit anywhere in the text is the probability that we a hit in position 1, or position 2, or etc or position $n-m$ which is at most $n-m$ times the probabilityh $1/n^2$, established in (b) above. This leads to a probability upper bound which is at most $(n-m)1/n^2 \leq 1/n$. We have used here the the fundamental probability result that

$$P(A_1 \cup A_2 \cup \ldots \cup A_{n-m}) \leq P(A_1) + P(A_2) + \ldots + P(A_{n-m}).$$

(c) If we do not have spurious hits running time of Rabin-Karp is $O(n)$; otherwise it may be $O(nm)$. The latter can occur with probability at most $1/n$ by part (b) and the former with probability at least $1-1/n$. Thus, the expected running time is $O(n)(1-1/n)+1/nO(nm) = O(n+m)$. Since $m < n$, we have $O(n+m) = O(n)$ as requested. ■

Part VI

Advanced topics

Chapter 13

More arithmetic problems

13.1 Integers or not

Exercise 441.

For this problem ignore issues about ceilings and floors for simplicity. This is quite a complicated problem in terms of counting operations. You need to be extra careful. Plan accordingly.
(a) You are given an integer N. Give an algorithm that finds $\lg N$ in polynomial time.
(b) You are given an integer N. Give an algorithm that finds an integer $A > \sqrt{N}$ (an upper bound to the square root) such that $A = \Theta(\sqrt{N})$ in polynomial time. You don't need to find $\sqrt{N}$ exactly. An approximation in the form of A suffices. Hint: How much is $2^{\lg n/2}$?
(c) Suppose that N is a perfect square $N = x^2$. Give a polynomial time algorithm that finds x.
(d) If integer N is an integer power of an integer, find integers x and y, if there exist, such that $N = x^y$ in polynomial time.
Note. For (d) your algorithm may fail to work properly if N is not of the form $N = x^y$. If n can be expressed in the form $N = x^y$, then your algorithm must print x, y. Polynomial time in all cases, means that the running time is $O((\lg N)^k)$ for some constant k.

Proof.
(a) Integer n has $\lfloor \lg n \rfloor + 1$ bits. Shifting integer n, i times to the right is equivalent to a division by 2^i thus producing the sequence $n/2$, $n/2^2$, ..., $n/2^i$ after each shift right. When the result is equal to zero, then $n/2^i < 1$, i.e. $i > \lg n$, i.e. $i \geq \lfloor \lg n \rfloor + 1$. The first time a shift produces a zero we stop and output the number of shifts which is $\lfloor \lg n \rfloor + 1$. Runing time of the algorithm is obviously $O(\lg^2 n)$, i.e. polynomial (in the number of bit of n).

```
 int  lgn (unsigned int n)
1. int i=0;
2. int result=0;
3. result =n;
4. while (result > 0) {
5.  n = (n>>1);
6.  i++;
7. }
8. return(i);
```

(b) Compute from part (a) $X = \lfloor \lg n \rfloor + 1 \geq \lceil \lg n \rceil$ in $O(\lg n)$ time. Divide the result X by two (a single shift right) in $O(1)$ time getting $Y = X/2$. Shift integer $W = 1$ left Y positions to get Z. It is apparent that $Z = 2^Y = 2^{X/2} = 2^{\lfloor \lg n \rfloor/2+1/2}$. Obviously

$$Z = 2^{\lfloor \lg n \rfloor/2+1/2} \leq 2^{\lg n/2+1/2} = O(\sqrt{n}).$$

Also

$$Z = 2^{\lfloor \lg n \rfloor/2+1/2} \geq 2^{\lceil \lg n \rceil/2} = 2^{\lg n/2} = \Omega(\sqrt{n}).$$

Time to find Z is polynomial $O(\lg n)$ as only shift operations are used a number of $O(\lg n)$ times.
(c) Since $n = x^2$, we conclude that $x \leq \sqrt{n}$. We binary search for x in the sorted sequence of integers $[1, 2, \ldots, \lfloor \sqrt{n} \rfloor + 1]$. The upper bound does not need to be accurate; an upper bound of it suffices including the generous one that n is. During the binary search procedure we do not peek on an array of consecutive integers. We just use two variables for the left most bound 2 (rather than 1) and the right most bound. Let the left boundary of the interval be indicated by l and the right by r. Initially $l = 2$ and $r = \lfloor \sqrt{n} \rfloor + 1$. ($x = 1$ serves as a solution for $n = 1$ only that can be dismissed during the initialization steps.) We set $m = (l + r)/2$. If $m^2 = n$, we are done. Otherwise if $m^2 > n$, then we search for x in $[l, m]$ or if $m^2 < n$ we search for x in $[m, r]$. This is binary search. The time for search is $O(\lg \sqrt{n})$;each time we perform a multiplication of two $\lg n$-bit integers for a cost of $O(\lg^2 n)$ and a comparison that is absorbed in that cost ; the total cost of the algorithm is $O((\lg n)^3)$ bit operations.

```
   SqSearch(n) // Find pos. integer x such that x*x = n ,n pos. integer
    if (n==1)
      return(1);
    else
      Qsearch(n);

   Qsearch(A)
1. l=2, r=A;                      //Search [l,r] for m such that m*m=A
2. Middle point  m= l+(r-l)/2;  //Pick middle value
3. if (m==l)  return(m,"not found);
4. Update rule  : if (m*m) >  A then  r=m-1; // m large; search [l..m-1]
                  if (m*m) <  A then  l=m+1; // m small; search [m+1..r]
                  if (m*m) == A then  return(m);// or return(m,l,r);
5. Goto 2
```

(d) If $n = x^y$, by taking logarithms we conclude $\lg n = y \lg x$, i.e. $y \leq \lg n$. We find $\lg n$ or $F = \lfloor \lg n \rfloor + 1$ from Problem 1, part (a) in polynomial time. For every integer y from 1 to F, we check whether $n = x^y$ using the (binary-search based) method of Part (c). The only difference with part c is the time it takes to raise x^y; in part (c) x^2 took $O(\lg^2 n)$. x^y since $x^y \leq n$ also takes the same time. Therefore the running time of part (d) is $\lg n$ times more than part (c) to account for $\lg n$ possibilities of the exponent y, for a total of $O(\lg^4 n)$.
Further optimization. We search for increasing exponents y. Thus if the search with $y = 2$ was not successful, the returned value m is going to be the starting r value for the bin search for $y = 3$. There were $\lg n$ binary search operations that concluded unsuccessfully for $y = 2$. Value $r \approx \sqrt{n}$ at the end of the $y = 2$ search, thus yield $\approx \lg(\sqrt{n} \approx \lg n/2$ iterations for the binary search for $y = 3$. Likewise $\lg n/3, \ldots$, for the remaining ones. The total is

$$\lg n + \lg n/2 + \lg n/3 + \ldots \approx \lg n \cdot \lg \lg n.$$

For a nice survey paper see "Detecting perfect powers in essentially linear time" by D. J. Bernstein, Mathematics of Computation, Vol 67, Number 223, Jul 1998, pp. 1253-1283, or a subsequent paper by D.J.Bernstein, H.W.Lenstra, J.Pila "Detecting Perfect Powers by factoring into coprimes". Detecting Perfect Powers is the first step of the polynomial time primality testing algorithm. ■

Exercise 442.

You are given an a-bit positive integer A and an b-bit positive integer B. You may assume $a \geq b$. How fast can you compute $A+B$, $A-B$, $A \cdot B$, and find Q,R such that $A = BQ+R$, $0 \leq R < B$? Express your answer in terms of a and b and justify it by pointing to the appropriate reference or give the algorithm.

Proof.
(a) Addition and subtraction can be done in $O(a+b)$ time. Since $A \geq B$ we have that $a \geq b$ and thus running time is also $O(a)$. The bound is tight i.e. $\Theta(a)$ since we need to read the a bits of A and b bits of B as well.
(b) Multiplications takes time $O(ab)$ (ordinary elementary school method). Karatsuba-Ofman can give $O(a^{\lg 3})$.
(c) Division based on the elementary school long division algorithm requires $a-b$ steps each one involving a subtraction of a b or $b+1$-bit number from a portion of A. This takes $O(b)$ from part (i). Therefore the total time is $O((a-b)b) = O(ab)$. However if we do Newton-Raphson based division this requires $O(M(n)) = O(a^{\lg 3})$ to find the inverse $1/B$ and the final multiplication $A \cdot 1/B$ requires also that time, i.e. a total of $O(a^{\lg 3})$. ■

Exercise 443.

(a) We want to compute the product

$$m = m_1 \cdot m_2 \cdot \ldots \cdot m_k.$$

Show that m can be computed in $O((\lg m)^2)$ bit operations.
(b) Use part (a) to show how to compute $n!$ (n factorial) in $O((n \lg n)^2)$ bit operations.
(c) Given integers a and b how long (in bit, NOT word operations) does it take to efficiently compute integer $n = a^b$, i.e. raise a to the power of b? Express the number of bit operations required in asymptotic notation as a function of n in an efficient solution to this problem. **Note**. An answer of the form: "*At most* $2 \lg n - 1$ *multiplications*" is not suitable as an answer to this problem.
This problem in essence states that the number of bit operations depends only on the number of bits of the result. It does not matter how we multiply the numbers. We are going to use the observation that the multiplication of an a-bit number with a b-bit number requires order of ab bit operations. This is similar to the polynomial multiplication cost function: two polynomials of degree bounds a and b can be multiplied in order of ab operations with a naive polynomial multiplication algorithm. Similarly, if we modify notation, in order to multiply numbers A and B we need in the order of $\lg A \lg B$ bit operations.

Proof.
We are going to choose the left to right ordering (associativity) to multiply the k numbers.

m	$=$	$m_1 \cdot m_2 \cdot \ldots m_k$		
t_1	$=$	m_1,	0 ops	t_1 has $\lg m_1$ bit
t_2	$=$	$t_1 \cdot m_2$,	$\lg m_1 \cdot \lg m_2$ ops	t_2 has $\lg m_1 + \lg m_2$ bit
t_3	$=$	$t_2 \cdot m_3$,	$(\lg m_1 + \lg m_2) \cdot \lg m_3$ ops	t_3 has $\lg m_1 + \lg m_2 + \lg m_3$ bit
$\ldots$		$\ldots$		
t_i	$=$	$t_{i-1} \cdot m_i$,	$(\lg m_1 + \lg m_2 + \ldots + \lg m_{i-1}) \cdot \lg m_i$ ops	t_i has $\lg m_1 + \lg m_2 + \ldots + \lg m_i$ bit
$\ldots$		$\ldots$		
t_k	$=$	$t_{k-1} \cdot m_k$,	$(\lg m_1 + \lg m_2 + \ldots + \lg m_{k-1}) \cdot \lg m_k$ ops	t_k has $\lg m_1 + \lg m_2 + \ldots + \lg m_k$ bit
m	$=$	t_k		

Do the multiplication left to right by computing $t_i = t_{i-1} m_i$ where $t_1 = m_1$. For t_2 (base case) we perform order of $\lg m_1 \lg m_2$ bit operations and the resulting t_2 is $\lg m_1 + \lg m_2$ bit long (to understand the duality, consider the multiplication of two polynomials of degrees a and b whose product has degree $a+b$ although the number of operations is of the order of ab). Thus for t_i the cost of multiplying t_{i-1} and m_i is $(\lg m_1 + \ldots + \lg m_{i-1}$ (the bit of t_{i-1}) times $\lg m_i$ and the result t_i is approximately $\lg m_1 + \ldots + \lg m_i$ bit long. Substitute k for i and you get the cost of the last multiplication when m is computed. The sum of all the costs is the cost of finding m. Note that $\lg m = \lg m_1 + \lg m_2 + \ldots + \lg m_k$. The total number of operations is thus

$$\begin{aligned}
\text{Time} &= \lg m_2 (\lg m_1) \\
&+ \lg m_3 (\lg m_1 + \ldots + \lg m_2) \\
&+ \lg m_4 (\lg m_1 + \ldots + \lg m_3) \\
&\ldots \\
&+ \lg m_i (\lg m_1 + \ldots + \lg m_{i-1}) \\
&\ldots \\
&+ \lg m_k (\lg m_1 + \ldots + \lg m_{k-1})
\end{aligned}$$

Consider any one of the parenthesized expressions on the far right side. Each one has some of the terms of the

sum $\lg m_1 + \lg m_2 + \ldots + \lg m_k$. Thus each one of them is at most $\lg m$.

$$\begin{aligned}
\text{Time} &\leq \lg m_2(\lg m) \\
&+ \lg m_3(\lg m) \\
&+ \lg m_4(\lg m) \\
&\ldots \\
&+ \lg m_i(\lg m) \\
&\ldots \\
&+ \lg m_k(\lg m) \\
&= (\lg m_2 + \ldots + \lg m_k)\lg m \\
&\leq \lg m \lg m = \lg^2 m
\end{aligned}$$

Then add all the terms up and the sum on the left (the equation before the last one above) is also at most $\lg m$. The result is proven.

We could also calculate the result right to left as follows. We calculate first $t_{k-1} = m_{k-1}m_k$ at a cost of $O(\lg m_{k-1} \lg m_k)$ with a result consisting of $O(\lg m_{k-1} + \lg m_k)$ bits. We calculate then $t_{k-2} = m_{k-2}t_{k-1}$ at a cost of $O(\lg m_{k-2} \lg t_{k-1}) = O(\lg m_{k-2}(\lg m_{k-1} + \lg m_k))$ with a result consisting of $O(\lg m_{k-2} + \lg m_{k-1} + \lg m_k)$ bits. We calculate then $t_{k-3} = m_{k-3}t_{k-2}$ at a cost of $O(\lg m_{k-3} \lg t_{k-2}) = O(\lg m_{k-3}(\lg m_{k-2} + \lg m_{k-1} + \lg m_k))$ with a result consisting of $O(\lg m_{k-3} + \lg m_{k-2} + \lg m_{k-1} + \lg m_k)$ bits. We go on until at the very end we calculate $t_1 = m_1 t_2$ at a cost of $O(\lg m_1 \lg t_2) = O(\lg m_1(\lg m_2 + \ldots + \lg m_k))$ with a result consisting of $O(\lg m_1 + \lg m_2 + \ldots + \lg m_{k-1} + \lg m_k) = O(\lg m)$ bits, which is integer m. The total running time (dropping for the time the O(.) notation) is therefore given by the following expression, working from t_1 all the way up to t_{k-1}.

$$\begin{aligned}
\text{Time} &= \lg m_1(\lg m_2 + \ldots + \lg m_k) \\
&+ \lg m_2(\lg m_3 + \ldots + \lg m_k) \\
&+ \lg m_3(\lg m_4 + \ldots + \lg m_k) \\
&+ \lg m_{k-1}(\lg m_k)
\end{aligned}$$

Since $(\lg m_i + \ldots + \lg m_k) \leq \lg m$ for any i, this can be written as

$$\begin{aligned}
\text{Time} &\leq \lg m_1 \lg m + \lg m_2 \lg m \ldots + \lg m_{k-1} \lg m \\
&= (\lg m_1 + \ldots + \lg m_{k-1}) \lg m
\end{aligned}$$

We apply the same observation for this latter expression, to obtain that the running time for finding m is $O(\lg m \lg m)$.

(b) Observe from part (a) above that for $m = n!$, $m_i = 1$, $k = n$, the cost of calculating $n!$ is $O(\lg^2(n!))$. Since $n! \approx (n/e)^n$ then, $\lg(n!) = \Theta(n \lg n)$ this means that the number of bit operations is $O(n^2 \lg^2 n)$.

Conclusion. Not only can one use Part (a) for the $n!$ computation but also for the integer power a^b result. The simplicity of the conclusion of Part (a) is that **it does not matter how many terms you have, it only matters how large, in terms of number of bit, the result/answer is!**.

(c) The answer is $O(\lg^2 n)$, since for $n = a^b$, use $m = n$, $m_i = a$, $k = b$, and from part (a), the result follows. ■

Exercise 444.

Show that the GCD computation requires $O(\lg a \lg b)$ bit operations.

Proof.
For any $a \geq 0, b > 0, a > b$, we have that $gcd(a,b) = gcd(b, a \bmod b)$. Euclid's algorithm (circa 300BC) finds the gcd of a and b as follows recursively.

```
   Euclid(a,b)
1. if (b==0) return(a);
2.  else return(Euclid(b,a mod b));
```

Let $a_0 = a$ and $a_1 = b$, and $a_2 = a \bmod b = a_0 \bmod a_1$. It is obvious that $a_2 < a_1$. Then $gcd(a_0, a_1) = gcd(a_1, a_2)$. Continuing the same way and defining $a_i = a_{i-2} \bmod a_{i-1}$ with $a_i < a_{i-1}$, we obtain that

$$gcd(a_0, a_1) = gcd(a_1, a_2) = gcd(a_2, a_3) = \ldots = gcd(a_{i-2}, a_{i-1}) = gcd(a_{i-1}, a_i) = \ldots = gcd(a_n, 0) = a_n.$$

Because $a_i < a_{i-1}$ we have that $a_i < a_{i-1} < a_{i-2} < \ldots < a_2 < a_1$. Given that a_1 is finite, and the sequence of the a_i's is strictly decreasing, there will be an $a_{n+1} = 0$ after at most $a_1 = b$ steps.

$$\begin{aligned} a_0 &= a_1 q_1 + a_2 \\ a_1 &= a_2 q_2 + a_3 \\ a_2 &= a_3 q_3 + a_4 \\ \ldots & \\ a_{n-2} &= a_{n-1} q_{n-1} + a_n \\ a_{n-1} &= a_n q_n \end{aligned} \tag{13.1}$$

Give eventuall $a_{n+1} = 0$, we have that $gcd(a,b) = gcd(a_{n-1}, a_n) = gcd(a_n, 0) = a_n$. We also have that $q_i = \lfloor a_{i-1}/a_i \rfloor$, i.e. $q_i \leq a_{i-1}$. In addition $q_n \geq 2$, since otherwise $q_n = 1$, $a_{n-1} = a_n$, and then the remainder of the division of a_{n-2} by a_{n-1} is $a_n = a_{n-1}$ which contradicts the fundamental $a_n < a_{n-1}$ fact. Let F_n be the n-th Fibonacci number, i.e. $F_n = F_{n-1} + F_{n-2}$, $n > 1$, where $F_0 = 0$ and $F_1 = 1$. It is known that $F_{n+1} \approx \phi^{n+1}/\sqrt{5}$ where $\phi = (1+\sqrt{5})/2$ is the golden ratio. Therefore if $b = F_n$, then $n = O(\lg b)$ as well. In Euclid's algorithm for $a > b \geq 1$, we have that $a \geq F_{n+2}$ and $b \geq F_{n+1}$, where n is the number of division steps of the algorithm. The proof is by induction. For $n = 1$, we show that in an one-step division algorithm it is true that $a \geq F_3$ and $b \geq F_2$. Note that $F_2 = F_1 + F_0 = 1$ and $F_3 = F_2 + F_1 = 2$. Since $b \geq 1$ we have that $b \geq F_2 = 1$. Since $a > b$, we have that $a \geq 2 = F_3$. This proves the base case of the induction. We then prove the inductive step, i.e. that if an i division step GCD computation of A, B, requires $A \geq F_{i+2}$ and $B \geq F_{i+1}$, for any $i = 1, \ldots, n-1$, then an n division step GCD computation of a, b requires $a \geq F_{n+2}$ and $b \geq F_{n+1}$. An n step division consists of the first step and the $n-1$ remaining steps. The first step of an n step division algorithm is say $a_0 = a = bq_1 + a_2 = a_1 q_1 + a_2$ Subsequently the gcd of a_1 and a_2 is computed in the remaining $n-1$ steps. The inductive hypothesis for $i = n-1$, would yield $b = a_1 \geq F_{n+1}$ and $a_2 \geq F_n$. Since $q_1 = \lfloor a/b \rfloor$ and $a > b$ we have that $q_1 \geq 1$ and thus $a_0 = a_1 q_1 + a_2 \geq a_1 \cdot 1 + a_2 \geq F_{n+1} + F_n = F_{n+2}$. Therefore, $a = a_0 \geq F_{n+2}$ and we also have by the inductive hypothesis that $b = a_1 \geq F_{n+1}$ as required. This proves the inductive step. It is straightforward to show that the bit complexity of GCD is $O(\lg b M(\lg a, \lg a))$ which is also $O(\lg a M(\lg a, \lg a))$. The number of steps of GCD is $\Theta(\lg b)$ and each a_i, q_i is no more than a, i.e. all intermediate results are $O(\lg a)$ bits long and of course $\lg b \leq \lg a$. Division and/or Multiplication requires $O(M(n,n))$ bit operations for n bit numbers. In the i-th division step $a_{i-1} = a_i q_i + a_{i+1}$, The multiplication involved requires $\lg a_i \lg q_i \leq \lg b \lg q_i \leq \lg b (\lg a_{i-1} - \lg a_i)$ bit operations, since $q_i \leq a_{i-1}/a_i$.
From the division steps (see steps above) we have that the complexity of the i-th division step is $O(\lg a_i \lg q_i)$ for the first multiplication and $O(\lg a_{i-1})$ for the subtraction of $a_i q_i$ from a_{i-1} to derive a_{i+1}. Ignoring for the time

being the O(.) notation we have

$$\begin{array}{rcl} \text{Step1} & : & \lg a_1 \lg q_1 + \lg a_0 \\ \text{Step2} & : & \lg a_2 \lg q_2 + \lg a_1 \\ \ldots & : & \ldots \\ \text{Step}i & : & \lg a_i \lg q_i + \lg a_{i-1} \\ \ldots & : & \ldots \\ \text{Step}n & : & \lg a_n \lg q_n + \lg a_{n-1} \end{array}$$

We treat the first step separately from the others.

$$\text{Step1} \quad : \quad \lg a_1 \lg q_1 + \lg a_0$$

Note that the complexity of finding q_i is also $\lg a_i \lg q_i$. Consider the division of $a_0 = a$ by $a_1 = b$ to find q_1 and a_2. Do the "elementary school" algorithm i.e. repeated subtractions of b from a portion of a which is 1 bit longer than b. Each such subtraction requires $O(\lg b)$ bit operations. The number of subtractions is the number of bits of q_1 i.e. $\lg q_1$. The total is thus $O(\lg q_1 \lg b) = O(\lg a_1 \lg q_1)$. These terms not just for the first quotient but also for all subsequent quotients are also included in the first term of the running time of each step above. Treating the first step separately, we conclude that the complexity of the first step is $O(\lg a \lg b)$ as $a_1 = b$, $a_0 = a$ and $q_1 \leq a$. We then add up the contributions of the remaining $n-1$ division steps to obtain the following time bound, and then add to this result the contributions of the first step already derived.

$$\begin{array}{rcl} \text{Time} & = & \lg a_2 \lg q_2 + \lg a_1 \\ & \ldots & \\ & + & \lg a_i \lg q_i + \lg a_{i-1} \\ & \ldots & \\ & + & \lg a_n \lg q_n + \lg a_{n-1} \end{array}$$

We note that $b = a_1 > a_2 > \ldots > a_n$, i.e. $\lg a_i \leq \lg b$. Also $q_i = a_{i-1}/a_i$, i.e. $\lg q_i \leq \lg a_{i-1} - \lg a_i$. Therefore,

$$\begin{array}{rcl} \text{Time} & = & \sum_{i=2}^{n} (\lg a_i \lg q_i + \lg a_{i-1}) \\ & \leq & \sum_{i=2}^{n} (\lg a_2 \lg q_i + \lg a_1) \\ & \leq & \sum_{i=2}^{n} (\lg a_1 \lg q_i + \lg a_1) \\ & \leq & \sum_{i=2}^{n} (\lg b (\lg a_{i-1} - \lg a_i) + \lg b) \\ & \leq & \lg b (\lg a_1 - \lg a_n) + n \lg b \\ & \leq & \lg b \lg b + n \lg b \\ & = & O(\lg^2 b + n \lg b) \end{array}$$

The number of division steps is $O(\lg b)$. Thus, $n = O(\lg b)$. The last expression thus gives: $O(\lg^2 b)$. Combining this with the running time of Step 1, we obtain the desired $O(\lg a \lg b)$ bound since $\lg b \leq \lg a$. The term $\lg a$ in the running time is due to the first division! All remaining divisions are $O(\lg^2 b)$.

Alternative proof:

We use the notes: $q_i = \lfloor a_{i-1}/a_i \rfloor$, i.e. $q_i \leq a_{i-1}/a_i$, for $i = 1, \ldots$. Thus $q_1 q_2 q_3 \ldots \leq (a_0/a_1)(a_1/a_2)(a_2/a_3)\ldots \leq a_0$.

In addition $a_1 = b > a_2 > a_3 > \ldots$. We have also from the discussion above that n, the number of division steps, is $O(\lg b)$. Therefore the running time is

$$\begin{aligned}
\sum_{i=1}^{i=n} \lg a_i \lg q_i &\leq \sum_{i=1}^{i=n} \lg b \lg q_i \\
&\leq \lg b \sum_{i=1}^{i=n} \lg q_i \\
&\leq \lg b (n + \lg \left(\prod_{i=1}^{i=n} q_i \right)) \\
&\leq \lg b (\lg b + \lg \frac{a_0 a_1 a_2 \ldots}{a_1 a_2 a_3 \ldots}) \\
&\leq \lg b (\lg b + \lg a_0) \\
&\leq \lg b (\lg b + \lg a) \\
&= O(\lg b \lg a).
\end{aligned}$$

What is left is to explain how the division can be done in the order of $\lg a_i \lg q_i$ steps. This is the long division algorithm. Long division (elementary school method) requires the order of $\lg q_i$ steps, the number of bits of the quotient. Take for example $a = a_1 q_1 + a_2 = b q_1 + a_2$. In each step we subtract $a_1 = b$ from a portion of a to form one more bit of q_1. Thus the running time is $\lg q_1$ times $\lg b = \lg a_1$. This is approximately $\lg q_1 \lg a_1$. There is nothing specific about this first division; the result can generalize for any other division step as well. From the division steps we have that the complexity of the i-th division step is $O(\lg a_i \lg q_i)$ for the first multiplication and $O(\lg a_{i-1})$ for the subtraction of $a_i q_i$ from a_{i-1} to derive a_{i+1}. Ignoring for the time being the O(.) notation we have

$$\begin{aligned}
\text{Step1} &: \lg a_1 \lg q_1 + \lg a_0 \\
\text{Step2} &: \lg a_2 \lg q_2 + \lg a_1 \\
\ldots &: \ldots \\
\text{Step}i &: \lg a_i \lg q_i + \lg a_{i-1} \\
\ldots &: \ldots \\
\text{Step}n &: \lg a_n \lg q_n + \lg a_{n-1}
\end{aligned}$$

We treat the first step separately from the others.

$$\text{Step1} : \lg a_1 \lg q_1 + \lg a_0$$

Note that the complexity of finding q_i is also $\lg a_i \lg q_i$. Consider the division of $a_0 = a$ by $a_1 = b$ to find q_1 and a_2. Do the "elementary school" algorithm i.e. repeated subtractions of b from a portion of a which is 1 bit longer than b. Each such subtraction requires $O(\lg b)$ bit operations. The number of subtractions is the number of bits of q_1 i.e. $\lg q_1$. The total is thus $O(\lg q_1 \lg b) = O(\lg a_1 \lg q_1)$. These terms not just for the first quotient but also for all subsequent quotients are also included in the first term of the running time of each step above. Treating the first step separately, we conclude that the complexity of the first step is $O(\lg a \lg b)$ as $a_1 = b$, $a_0 = a$ and $q_1 \leq a$. We first add up the contributions of the remaining $n-1$ division steps to obtain the following, and then add to this result the contributions of the first step.

$$\begin{aligned}
\text{Time} &= \lg a_2 \lg q_2 + \lg a_1 \\
&\ldots \\
&+ \lg a_i \lg q_i + \lg a_{i-1} \\
&\ldots \\
&+ \lg a_n \lg q_n + \lg a_{n-1}
\end{aligned}$$

We note that $b = a_1 > a_2 > \ldots > a_n$, i.e. $\lg a_i \leq \lg b$. Also $q_i = a_{i-1}/a_i$, i.e. $\lg q_i \leq \lg a_{i-1} - \lg a_i$. Therefore,

$$\begin{aligned}
\text{Time} &= \sum_{i=2}^{n} (\lg a_i \lg q_i + \lg a_{i-1}) \\
&\leq \sum_{i=2}^{n} (\lg a_2 \lg q_i + \lg a_1) \\
&\leq \sum_{i=2}^{n} (\lg a_1 \lg q_i + \lg a_1) \\
&\leq \sum_{i=2}^{n} (\lg b(\lg a_{i-1} - \lg a_i) + \lg b) \\
&\leq \lg b(\lg a_1 - \lg a_n) + n \lg b \\
&\leq \lg b \lg b + n \lg b \\
&= O(\lg^2 b + n \lg b)
\end{aligned}$$

The number of division steps is $O(\lg b)$. Thus, $n = O(\lg b)$. The last expression thus gives: $O(\lg^2 b)$. Combining this with the running time of Step 1, we obtain the desired $O(\lg a \lg b)$ bound.

■

Exercise 445.

Show how you can determine that n is a perfect power or not in $O(\lg^k n)$ bit operations. Keep k (strictly) smaller than 4. In other words find integer x and y such that $x^y = n$.

Note 1. You count bit operations and the only input you are given is n in binary.

Note 2. It is possible to solve this problem for $k = 2$. It is non-trivial and requires the use of Newton-Raphson methods.

Proof.

Consider the previous problem, with $m = n$ and $m_i = x$ and $k = y$. It is straightforward from the result of that problem that computing x^y requires $O(\lg^2 n)$ bit operations. This requires the knowledge of x, y though. Note that division by two that requires is a right shift operation and can be done in $O(\lg n)$ time for a number z that is up to n. We note that the smallest value for x is $x = 2$. This suggest the largest value for y i.e. $y = \lg n$. One possibility is to try all possible values of y from two to $\lg n$. This implies an $O(\lg n)$ base loop. The second step is for fixed y, to do a binary search in the integer range $[1, n]$ to find an appropriate x such that $x^y = n$ binary ■

Exercise 446.

Finding x^2 in time $O(S(n))$ plus other linear terms is no harder than reciprocal finding in time $O(R(n))$, i.e. $S(n) = O(R(n)+n)$. Assume you have a black box that for input y computes $1/y = INV(y)$. Use this black box and additions/subtractions ONLY to show that indeed $S(n) = O(R(n)+n)$.

Hint. You need to use a transformation similar to $ab = ((a+b)^2 - (a-b)^2)/4$ to reduce squaring to reciprocal finding. Arguments of the form "reciprocal finding requires at least one multiplication, thus $R(n) \geq M(n)$ and i can find x^2 with a single multiplication" are NOT VALID arguments. You can only use INV, $+$ and $-$ operations.

Proof.
Observe that

$$x^2 = \frac{1}{\frac{1}{x-1} - \frac{1}{x}} + x.$$

To compute x^2 do the following.

```
z1 = Inv(x)
z2 = Inv(x-1)
z3 = x2-z1
z4 = Inv(z3)
x5 = x4 +x.
return(x5) // x5 is x*x
```

Note that all intermediate steps require 3 inverses on numbers that are at most x, and 3 additions/subtractions. What is the running time? Note that all inversions are to a number that is at most x. Let x have n bits. Thus the 3 inversions cost $3R(n)$. The three additions cost $O(n)$. Total is $O(R(n)+n)$ as needed. Note that one does not know how $R(n+1)$ relates to $R(n)$, so one needs to be careful if in for z2, one tries to find the inverse of $x+1$ rather than $x-1$. For examle if $R(n) = (n)^3$, then $R(n+1) = O(R(n)+n^2)$. ■

Exercise 447.
Fibonacci Revisited. Show that F_n can be computed in $O(n^2)$ bit operations even if n-bit multiplication can only be done in $\Theta(n^2)$ bit operations.

Proof.
Show trivially by induction that $F_n \leq 2^n$. To find F_n do the iterative algorithm using the recurrence $F_n = F_{n-1} + F_{n-2}$. It requires n additions. Each Fibonacci number up to F_n is at most 2^n i.e. n bits long. Thus the n additions require $O(n^2)$ bits. ■

Exercise 448.

Let A and B be two m-bit integers.
The prime number theorem states that the number of primes less than x is about $x/\ln x$. Therefore the density of primes is $1/\ln x$. In other words if we pick uniformly at random an integer z between 1 and N, with probability $1/\ln N$ we choose a prime number, and with probability $1-1/\ln N$ we choose a composite number. Let $A = A_m \ldots A_1$ be an m-bit integer. Let $B = B_m \ldots B_1$ be an m-bit integer. Then $A = A_m 2^{m-1} + \ldots + A_1$ and $B = B_m 2^{m-1} + \ldots + B_1$.
(a) Is $A \leq 2^m$? Explain. Is $B \leq 2^m$? Explain.
(b) If $p/(A-B)$ then show that $A \bmod p \equiv B \bmod p$. Then show that if $A \bmod p \equiv B \bmod p$, then $p/(A-B)$.
(c) Show A or B can have at most m distinct prime divisors.
(d) Let us pick a random number prime number p between 1 and $4n^2 m \ln n$, where $n > m$. Show that such a p divides $A-B$ with probability at most $O(1/n^2)$.

Proof.
(a) Since $A = A_m \ldots A_1$ we have that $A = A_m 2^{m-1} + \ldots + A_1$. Since $A_i \leq 1$, we thus have that $A \leq 2^{m-1} + \ldots + 1 = 2^m - 1 < 2^m$. Similarly for B.
(b) If $p/A-B$ then $A - B = kp$ for some integer k i.e. $A - B \equiv 0 \bmod p$ i.e. $A \equiv B \bmod p$, which implies that $A \bmod p \equiv B \bmod p$.
(c) Let $c < 2^m$, i.e. c is an m-bit number. Let $c = p_1 p_2 \ldots p_k$ be the decomposition of c in terms of primes (note that the primes $p_1, p_2, \ldots$ may not be distinct. For example $c = 60$ has a decomposition $p_1 = 2$, $p_2 = 2$, $p_3 = 3$, $p_4 = 5$. Any prime number p_i is at least 2. We have that

$$c = p_1 p_2 \ldots p_k \geq 2 \cdot 2 \ldots 2 = 2^k$$

Since $2^m > c \geq 2^k$ we have that $k \leq m$. Therefore c has at most m (distinct or indistinct) prime divisors. And thus A, B also have at most m prime divisors.
(d) Pick a random prime number between 2 (rather than one) and $N = 4n^2 m \lg n$, where $n > m$. Using the prime number theorem, the number of primes in the interval less than N is $N/\ln N = 4n^2 m \ln n/(\ln 4 + 2\ln n + \ln m + \ln\ln n \approx mn^2$. Since $A, B < 2^m$ the difference $c = A - B$ is also at most 2^m, and by part (c) $A, B, A-B$ have each one at most m prime divisors. Thus the probability p is one of them is at most $m/mn^2 = O(1/n^2)$. ■

Exercise 449.

Word model Fibonacci.

The Fibonacci sequence is given by the following recurrence $F_{n+1} = F_n + F_{n-1}$ for $n \geq 1$ and $F_0 = F_1 = 1$.
(i) Show how to compute F_n in $O(n)$ time.
(ii) Given an $n \times n$ matrix A show how you can find A^n in $O(n^3 \lg n)$ time.
(iii) Can you improve the obvious time bound in (a)? In particular prove that F_n can be computed in $O(\lg n)$ time.

Proof.
(i) This is straightforward.

```
   Fibonacci(int n)
1. int F[0..n];
2. F[0]=F[1]=1;
3. for(i=2;i<=n;i++)
4.  F[i]=F[i-1]+F[i-2];
5. return(F[n]);
```

(ii) This is the x^n problem in a different context. If A is a matrix, the exponentiation A^n requires $O(\lg n)$ matrix multiplications. Each multiplication for a $n \times n$ matrix requires $O(n^3)$ scalar operations (A/S/M). Thus the total cost is $O(n^3 \lg n)$ as claimed. Strassen can improve upon the n^3 part as well.
(iii) We first note that

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix}$$

and thus by unfolding the recurrence $n-1$ times by similarly expanding the vector of F_{n-1}, F_{n-2} we obtain the following.

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = A^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

where

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

It is known that $F_0 = 0$ and $F_1 = 1$. Thus applying part (b) with $n = 2$, we have in $O(\lg n)$ time. Subsequently we multiply this result with

$$\begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

in $O(1)$ time to find F_n in overall time still $O(\lg n)$, after reading off the first component (element) of the resulting product.

■

Exercise 450.

We are interested in determining whether n is a perfect power i.e. whether there exist integer x, y such that $n = x^y$ and if such x, y exist also determining the values of x, y. We will use in the remainder the word-model for operation counting.
(a) If n is indeed a perfect power, how large can y be? Express your answer as a function of n only.
(b) Suppose you have a black box `YRoot(N,Y)` that given N and Y it finds the integer Y-th root of N (i.e. $N^{1/Y}$) if such exists or returns −1 if such root does not exist. For example `YRoot(8,3)` would return 2 since $8^{1/3} = 2$, whereas `YRoot(8,2)` would return -1. Suppose that the time it takes for `YRoot` to return an answer is $O(1)$. How could you use `YRoot` to determine whether n is a perfect power or not? How many times do you need to call `YRoot` in an efficient determination of whether n is a perfect power or not?
(c) `YRoot` in $O(1)$ time is rather unrealistic. How fast could you implement it? Does a solution that is $O(\log^3 n)$ (or $o(\log^3 n)$) exist? Explain. Can you do it in $o(\log^2 n)$ time?

Proof.
(a) Since $n = x^y$ we have that $\lg n = y \lg x$. Therefore $y \leq \lg n$.
(b) We try in turn all possible integer exponents y as determined by part (a) i.e. all $y \leq \lg n$. If for some exponent y we get a value of X that is not equal to -1 i.e. an X such that $X^Y = n$, then we have found the pair (X, Y) and we print it througl line 4 of the pseudocode. The number of calls to `YRoot` is $O(\lg n)$. Note that $\lg n$ is easy to compute, if one needs to do so. It's the number of bits used to represent n that can be retrieved by dividing n by 2 successively until we reach 0 (or equivalently shifting right).

```
    PerfectPower(n)
1.  for(Y=1; Y<= lg(n); Y++) {
2.       X= YRoot(n,Y);
3.       if (X != -1 ) then
4.         print "X,Y : X**Y = N "
5.  }
```

(c) An implementation of `YRoot` involves binary search. BinarySearch all the integer numbers $1, \ldots, n$ to find if there is an integer X such that $X^Y = n$ for given n, Y. A very high-level pseudocode might look like the following code fragment.

```
1.   YRoot(n,Y)
2.     YRoot(n,Y,1,n);

1.   YRoot(n,Y,L,R)
2.   while (L<=R) {
3.    M= L+ (R-L)/2;
4.    if ( L == M)   return (-1);   // return("no progress; does not exist");
5.    if ( M**Y == n )   return (M); // x= M , y= Y
6.    if ( M**Y  > n )     R = M;
7.    if ( M**Y  < n )     L = M;
7.   }
```

The number of iterations in `YRoot` is $O(\lg n)$. Each one involves an exponentiation operation. Raising M^Y can be done naively in $Y - 1$ multiplications; with successive doubling it can be done in $O(\lg Y)$ multiplications. Since $Y = O(\lg n)$, this means that exponentiation requires $O(\lg \lg n)$ multiplications and thus `YRoot` takes time $O(\lg n \cdot \lg \lg n)$ overall. This implicitly means that the original problem (i.e. perfect power) can be solved in $O(\lg^2 n \cdot \lg \lg n)$ time since `PerfectPower` iterates $O(\lg n)$ times. ■

Exercise 451.

Can you further optimize YRoot by say reducing $\lg^2 n$ into $\lg n \lg\lg n$ instead?

Proof.
Consider the first iteration of the algorithm that uses $y = 2$ and tries to establish whether n is a perfect square. After $O(\lg n)$ binary search operations we establish whether there exists integer x such that $x^2 = n$ or not. If there is such an x, we are done and stop. If there is no such x, then we have also found an l such that $l^2 < n$ and an $r = l + 1$ such that $r^2 > n$. The subsequent binary search operation for $y = 3$ does not need to start with $l = 1, r = n$ as did the first, but it can start with $l = 1$ and initialize r to the value of r last established in the previous binary search operation (the one that had $r^2 > n$). This is because that if $r^2 > n$, then of course $r^3 > n$ as well, so our current r is large enough to contain the possible integer cubic root of n. Therefore with this observation each binary search operation starts with the r of the previous one and all of them would require not $O(\lg^2 n)$ but less than that. How much is this time? For the first iteration (i.e. exponent 2) it will be $\lg n$ steps, for the next iteration (i.e. exponent 3) since we start from an approximation of $n^{1/2}$, the number of steps is going to be $\lg n/2$ and in the i iteration (i.e. exponent $i+1$) it will be $\lg n/i$. i can be up to $y \leq \lg n$. Thus the number of iterations is $\lg n \sum_{i=1}^{y} 1/i = \lg n \lg y = \lg n(\ln\lg n + \gamma)$, since the sum is the harmonic series. Thus $O(\lg^2 n)$ means $O(\lg n \lg\lg n)$. **Example.** Let $n = 32$. We start looking whether $32 = x^2$, i.e. start with $y = 2$. Initially $l = 1, r = 32$ and thus $m = (l + r)/2 = 16$. We go on with binary search to establish that the square root is between $l = 5$ and $r = 6$, which gives $m = 5$ again which is also equal to l. Then we start the $y = 3$ search with $l = 1$ and $r = 6$, rather than $r = 32$ since $n^{1/3} \leq n^{1/2} < r = 6$.
Time resolution. The only thing that we did not mention in this argument is the fact that every binary search operation requires also an x^y computation. This exponentiation could be done in $O(\lg y)$ multiplications. Since $y \leq \lg n$, this is up $O(\lg\lg n)$. Thus each one of the fewer than $O(\lg^2 n)$ binary searches costs $O(\lg\lg n)$ time for a total of $O(\lg n \lg\lg n)$ times $O(\lg\lg n)$ which is $O(\lg^2 n)$ or also $o(\lg^2 n)$ as well. For a nice survey paper see "Detecting perfect powers in essentially linear time" by D. J. Bernstein, Mathematics of Computation, Vol 67, Number 223, Jul 1998, pp. 1253-1283, or a subsequent paper by D.J.Bernstein, H.W.Lenstra, J.Pila "Detecting Perfect Powers by factoring into coprimes". ■

Exercise 452.

(a) Show that X^n, where X is an $m \times m$ array such that $m = \Theta(1)$, requires $O(\lg n)$ time to compute in the word model.

Hint: How many bits to represent n?

Proof.

(a) Let $k = \lfloor \lg n \rfloor$. We first compute all power of two products, X^{2^i}, $i = 1, \ldots, k$.

$$x, y_1 = X^2, y_2 = y_1^2 = X^4, y_3 = y_2^2 = X^8, y_4 = y_3^2 \ldots = X^{2^{k-1}}, y_k = y_{k-1}^2 = X^{2^k}$$

This requires k multiplications by performing repeated doubling. Note that $y_k \leq X^n \leq y_k^2$. We then find the binary representation of n i.e. $n = b_0 + b_1 2^1 + \ldots + b_k 2^k$, where $b_i = 0, 1$. This could require shifting operations; the number of such bit operations and also the number of bits is $O(\lg n)$. Then

$$X^n = X^{b_0 + b_1 2^1 + \ldots + b_k 2^k} = X^{b_0} X^{b_1 2^1} \ldots X^{b_k 2^k}$$

Thus if b_1 is not zero (it is one) we include in the product $X^{2^1} = y_1$, otherwise, not. The product above has k terms; to compute it we need to form k more products.

```
   Power( X, n)
1.  k = floor(lgn); X[0]=X;
2.  for (i=1;i<=k;i++)
3.   X[i] = X[i-1]*X[i-1] // X[i]=X**(2^i)
4.  n= B[k]B[k-1]...B[0] //binary representation of n
5.  if (B[0] == 1)  P=X;
6.   else           P=1;
7.  for(i=1;i<=k;i++);
8.   if (B[i]==1)  P= P* X[i];
9.  return(P);
```

Therefore $2\lg n$ multiplications are required, each one takes $O(m^3) = O(1)$ time, thus proving the $O(\lg n)$ upper bound. ■

Exercise 453.

n choose k. $\binom{n}{k}$ is defined as

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

where $k!$, the factorial, is defined as $k! = k(k-1)(k-2)\ldots 1$.
Find $\binom{n}{k}$ in $O(n^2)$ additions/subtractions. Multiplications and divisions are not allowed.

Proof.
Let $T(n,k) = \binom{n}{k}$ be written as the sum of

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

thus giving

$$T(n,k) = T(n-1,k) + T(n-1,k-1).$$

The former or the latter expression can be derived by combinatorial interpretation. $T(n,k)$ is the number of ways k objects are chosen out of n objects. Let X be one of the n objects. If X is one of the k objects the remaining $n-1$ objects will contribute $k-1$ objects which along with X give the k objects. The combinations of $k-1$ of $n-1$ objects is $T(n-1,k-1) = \binom{n-1}{k-1}$. On the other hand if X is not one of the k objects, then $T(n,k)$ will include a term $T(n-1,k)$ which is the number of combinations of k objects out of the remaining $n-1$ objects (excluding X). $T(n,k)$, $k \leq n$, is easy to compute as long as the base case is given. $T(n,0) = T(n,n) = 1$ for all n. Obviously its running time is $O(n^2)$ and no operations other than additions are performed. FindT(n,k) below does this.

```
  FindT(n,k)
1. int T[0..n][0..k];
2  T[0][0]=1;
3. for(N=1;N<=n;N++)
3.  for(K=0;K<=MIN(N,k);K++)
4.   if (K==0 || K==N) T[N][K]=1;
5.    else             T[N][K]=T[N-1][K-1]+T[N-1][K];
6. return(T[n][k]);
```

■

Exercise 454.

Karatsuba-Ofman for integers.

Proof.
In order to simplify exposition, without loss of generality, let n be an even power of two. Let a,b be the two n-bit numbers. The n bit number a consists of $n/2$ least significant bits and $n/2$ most significant bits respectively. Let a_l, a_m be the integer representation of these $n/2$ bits. Then $a = a_m 2^{n/2} + a_l$ and similarly $b = b_m 2^{n/2} + b_l$ Then

$$\begin{aligned} ab &= (a_m 2^{n/2} + a_l)(b_m 2^{n/2} + b_l) \\ &= a_m b_m 2^n + (a_m b_l + a_l b_m) 2^{n/2} + a_l b_l \\ &= a_m b_m 2^n + ((a_m + a_l)(b_m + b_l) - a_l b_l - a_m b_m) 2^{n/2} + a_l b_l \end{aligned}$$

```
   IntegerMult   (a,b)
1. (a_m a_l) =a;
2. (b_m b_l) =b;
3. t1 = IntegerMult   ((a_m + a_l ),(b_m + b_l));
4. t2 = IntegerMult   (a_m , b_m);
5. t3 = IntegerMult   (a_l , b_l);
6. t4 = t1 - t2 -t3;
7. t2 = t2 << n; // t2 =t2 * 2**n
8. t4 = t4 << (n/2); // t4 = t4 * 2**(n/2)
6. return ( t2 + t4 + t3);
```

The analysis is that of part (b). Note that a tricky point in line 3 is that $a_m + a_l$ are of $n/2+1$ bits. This can be fixed by step (a). ■

13.2 Polynomial

Exercise 455.

Word model. In the remainder A, S, M represent scalar (two integer) additions, subtractions and multiplications respectively. There is no difference between addition and subtraction in general as $a+(-b)=a-b$.
Let $f(x)=a+bx$, and $g(x)=c+dx$, where $a,b,c,d\neq 0$, are (for the time being) integers and f,g are two polynomials on indeterminate (unknown) x. Show how you can find the product $f(x)g(x)$ that is, the coefficients of all powers of x doing only three integer multiplications. The product $f(x)g(x)$ is $f(x)g(x)=bdx^2+(ad+bc)x+ac$, and the coefficients are ac, $ad+bc$ and bd.

Proof.
$f(x)g(x)=bdx^2+(ad+bc)x+ac$ can be rewritten as $f(x)g(x)=bdx^2+((a+b)(c+d)-ac-bd)x+ac$. Therefore only three multiplications are required $m_1=ac$, $m_2=bd$, and $m_3=(a+b)(c+d)$. The number of additions however increased from one to 4 (additions and subtractions) $a_1=a+b$, $a_2=c+d$, $a_3=m_3-m_1$, $a_4=a_3-m_2$. ■

Exercise 456.

A complex number z is represented by $z = a + ib$, where a, b are real numbers and symbol i is such that $i^2 = -1$. Sometimes we represent $a + ib$ by the pair (a, b). You may relate z to a polynomial $f(x) = a + xb$ with $x = i$. To be in line with a prior discussion consider a, b, c, d integers rather than reals.
For two complex numbers $z_1 = a + ib$ and $z_2 = c + id$, $z_3 = z_1 + z_2 = (a + c) + i(b + d)$. We can represent z_1, z_2, z_3 with the pair $(a, b), (c, d), (a + c, b + d)$ respectively.
The number of scalar real (integer) additions or subtractions to add the two complex numbers is two ($a + c$ and $b + d$). The product of z_1 and z_2 is $z_4 = z_1 z_2 = (ac - bd) + i(ad + bc)$. (Treat the two complex number as polynomial with the difference that i^2 resolves to -1. Total number of (scalar) real operations to multiply z_1 and z_2 include two real (integer) additions and four real (integer) multiplications to compute $ac, bd, ad, bc, ac - bd, ad + bc$; a subtraction is an addition involving the negative of a number.
(a) One can readily use the previous problem to reduce the number of multiplications from four to three with some increase of the number of addition and subtractions. Show how to multiply two complex numbers using only three integer (or real) multiplications.
(b) Now let us focus our attention to division. Let $z_1 = a + ib$ and $z_2 = c + id$ be two complex numbers. You may assume that $z_2 \neq 0$, i.e. $c^2 + d^2 \neq 0$, since otherwise the division $z_1/z_2 = z_1/0$ can not be performed. The division of two complex numbers is defined as follows.

$$\begin{aligned} \frac{a+ib}{c+id} &= \frac{(a+ib)(c-id)}{(c+id)(c-id)} \\ &= \frac{ac+ibc-iad+bd}{c^2+d^2} \\ &= \frac{ac+bd}{c^2+d^2} + \frac{(bc-ad)i}{c^2+d^2} \end{aligned}$$

Apparently the z_1/z_2 is a complex number z_3 such that $z_3 = \frac{ac+bd}{c^2+d^2} + \frac{(bc-ad)i}{c^2+d^2}$. Thus to compute z_3 from z_1 and z_2 we apparently need 6 multiplications (Ms) and 2 divisions (Ds) for a total of 8 M/Ds. Can you decrease this bound further? Explain.

Proof.
(a) One way to solve this problem is as follows. First we perform 2 real S/A's and 3 real M's to find m_1, m_2, m_3.

$$\begin{aligned} m_1 &= (a-b)*(c+d) \\ m_2 &= a*d \\ m_3 &= b*c \end{aligned}$$

Then, since $m_1 = ac + ad - bc - bd$, we can find $Re(z) = s_1$ and $Im(z) = s_2$ by performing three additional S/A's as follows.

$$\begin{aligned} s_1 &= m_1 - m_2 + m_3 \\ s_2 &= m_2 + m_3 \end{aligned}$$

It is straightforward to verify that $s_1 = ac - bd$ and $s_2 = ad + bc$ as required. Total number of operations is 3M's and 5S/A's. Note that you cannot find the answer with 2M's; there is a lower bound proof that 3 is the minimum (see Aho, Hopcroft and Ullman "The design and analysis of computer algorithms").
(b) We want to find A, B as defined below.

$$\begin{aligned} A = \frac{ac+bd}{c^2+d^2} \quad , \quad B = \frac{(bc-ad)}{c^2+d^2} \\ A = \frac{ac/d+b}{c^2/d+d} \quad , \quad B = \frac{(bc/d-a)}{c^2/d+d} \end{aligned}$$

Let us start as follows.

```
T1 = c/d ;     // c/d
T2 = T1*a      // a*c/d
T3 = c *T1     // c*c/d
N1 = T2 +b;   // (ac/d+b)
D1 = T3+d;    // (c**2/d+d)
A  = N1 / D1; // (ac/d+b)/(c**2/d+b) is A.
T4 = T1*b;    // bc/d
N2 = T4-a;    // bc/d-a
B  = N2/D1    // B
```

Total number of Ms/Ds is 6 and As/Ss is 3 compared to naive approach that requires 8 and 3 respectively. Of course the claimed schedule works provided that $D \neq 0$. If $D == 0$, we interchange the roles of C, D because C should be $C \neq 0$. Otherwise both C, D are zero and we are done (we can not divide a complex number with zero). ■

Exercise 457.

You are given six polynomials $f_1, \ldots, f_6$ of degrees $1,3,2,3,4,5$ respectively. We are interested in finding the product $f = f_1 f_2 f_3 f_4 f_5 f_6$ by performing pairwise multiplications. Assume that the cost of multiplying two polynomials of degree a and b is $a \cdot b$. Find a schedule for multiplying the six polynomials that is of the lowest possible total cost (total cost is the sum of the costs of all multiplications performed to determine f).

Show (i.e. give a proof) that the schedule you have chosen can not be improved in any way.

Proof.

Multiply them any way you like. The final result is a polynomial of degree 18 and the total cost is 130. Note that 130 is the sum of all products $d_i \cdot d_j$ for every combination of two polynomials f_i and f_j among the six and d_i and d_j are respectively the degrees of f_i and f_j. In our example we have $6(6-1)/2$ such pairs of polynomials.

$$130 = d_1(d_2+d_3+d_4+d_5+d_6) + d_2(d_3+d_4+d_5+d_6) + d_3(d_4+d_5+d_6) + d_4(d_5+d_6) + d_5 d_6$$

This sum is independent of the order of the evaluation of the products; an easy proof is by induction. **Note.** One of the reasons that this is so is that polynomials satisfy commutativity therefore $f_1 f_2 = f_2 f_1$. What happens if instead of polynomials you have matrices and the operation is matrix multiplication? The cost then depends on the specific schedule that you choose. ■

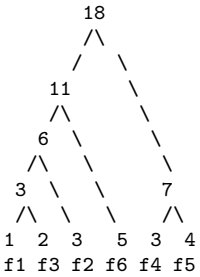
Exercise 458.

You are given six polynomials $f_1, \ldots, f_6$ of degrees $1,3,2,3,4,5$ respectively. We are interested in finding the product $f = f_1 f_2 f_3 f_4 f_5 f_6$ by performing pairwise multiplications. Assume that the cost of multiplying two polynomials of degree a and b is $a+b$, i.e. it is proportional to the space required to store the product which is a polynomial of degree $a+b$. Find a schedule for multiplying the six polynomials that is of the lowest possible total cost (total cost is the sum of the costs of all multiplications performed to determine f) for this non-traditional definition of a cost function.

Example. For three polynomials g_1, g_2, g_3 of degrees $1,2,3$ respectively, you first compute $g_2 g_3$ and then multiply the result by g_1. The cost of the first multiplication is 5 $(2+3)$ and the cost of the second multiplication is 6 since you multiply the result, a degree 5 polynomial, to a degree one polynomial. Total cost is 11.

Proof.

We use a greedy algorithm to find the optimal schedule a-la Huffman. Proofs for optimality are those of Huffman coding. Total cost is 45.



■

Exercise 459.

You are given n polynomials $f_1(x),\ldots,f_n(x)$ each one of degree $n-1$, i.e. each polynomial look similar to the generic polynomial $f(x)=a_{n-1}x^{n-1}+\ldots+a_1x+a_0$. You are also given n points $c_1,c_2,\ldots,c_n$. Give an algorithm that evaluates the n polynomials in the n points in $O(n^{\lg 7})$ time.

Proof.

Form a matrix F that is $n\times n$ whose i-th row consists of the polynomial coefficients of the i-th polynomial. For example if the i-th polynomial is $f(x)=a_{n-1}x^{n-1}+\ldots+a_1x+a_0$, the i-th row of F looks like $(a_{n-1}\ldots a_1a_0)$. F can be formed in $O(n^2)$ time. Similarly we form a matrix G that is $n\times n$ whose i-th column is $(1c_ic_i^2\ldots c_i^{n-1})^T$. The time it takes to form a particular column i given c_i is $O(n)$ as the n power c_i^j need to formed. The total cost is n times as much. Thus G can be formed in $O(n^2)$ time. After we form F and G polynomial evaluation is equivalent to finding the product $F\times G$. This is a matrix product and by Strassen's algorithm it requires $O(n^{\lg 7})$ time. The total cost is thus $O(n^{\lg 7}+n^2)=O(n^{\lg 7})$ as needed.

■

Exercise 460.

CLRS [6] problem.

Consider two sets A and B each having n integers in the range from 0 to $10n$. We wish to compute the Cartesian sum of A and B defined by

$$C = \{x+y : x \in A \ \ and \ \ y \in B\}$$

Note that the integers in C are in the range from 0 to 20n. We want to find the elements of C and the number of times each element of C is realized as a sum of elements in A and B. Show that if the product of two degree bound n polynomials can be computed in $O(n \lg n)$ time, then this problem can also be solved in $O(n \lg n)$ time.
Hint. Represent A and B as polynomials of degree at most $10n$.

Proof.
For set A consisting of numbers $a_1, \ldots, a_n$ consider the polynomial

$$A(x) = x^{a_1} + \ldots + x^{a_n}$$

and similarly for set B consider

$$B(x) = x^{b_1} + \ldots + x^{b_n}$$

In the product $C(x) = A(x)B(x)$ a term $c_i x^{s_i}$ indicates the existence of an a_{j_1} and b_{k_1} such that $a_{j_1} + b_{k_1} = s_i$ as $x^{a_{j_1}} \cdot x^{b_{k_1}} = x^{s_i}$. The coefficient c_i is an indicator of how many pairs (j_l, k_l) exist such that $a_{j_l} + b_{k_l} = s_i$. The number of pairs $l = c_i$. Note that since a_i, b_i are at most 10n $A(x)$, $B(x)$ are of degree at most 10n and $C(x)$ of degree at most 20n. Therefore, if an algorithm exists that multiplies two polynomials of degree n in $O(n \lg n)$ time the product $A(x)B(x)$ can be computed in $O(20n \lg(20n)) = O(n \lg n)$ time as well.

■

Exercise 461.

Evaluating a polynomial A(x) of degree-bound n at a given point x_0 can also be done by dividing $A(x)$ by the polynomial $(x-x_0)$ to obtain the quotient polynomial $q(x)$ of degree-bound $n-1$ and a remainder r, such that

$$A(x) = q(x)(x-x_0)+r$$

Clearly $A(x_0) = q(x_0)(x_0-x_0)+r = r$. Show how to compute the remainder r and the coefficients of $q(x)$ in time $\Theta(n)$ from x_0 and the coefficients $a_0, a_1, \ldots s$, of A.

Proof.
By inspection, since $A(x) = q(x)(x-x_0)+r$, we obtain that $A(x_0) = r$. Therefore in order to compute r we use polynomial evaluation of $A(x)$ at x_0 at a cost of $O(n)$ by using Horner's method. Since $A(x)$ is of degree bound n, its degree is at most $n-1$, and therefore $A(x) = a_{n-1}x^{n-1} + \ldots a_1x + a_0$. Since $q(x)$ is of degree $n-2$, we obtain that $q(x)$ looks like $q(x) = q_{n-2}x^{n-2} + \ldots q_1x + q_0$. Therefore

$$\begin{aligned} A(x) &= q(x)(x-x_0)+r \\ a_{n-1}x_{n-1} + \ldots a_1x + a_0 &= (q_{n-2}x^{n-2} + \ldots q_1x + q_0)(x-x_0)+r \\ a_{n-1}x^{n-1} + \ldots a_1x + a_0 &= q_{n-2}x^{n-1} + q_{n-3}x^{n-2} + \ldots + q_1x^2 + q_0x + r \\ &= -x_0q_{n-2}x^{n-2} - \ldots - x_0q_1x - x_0q_0. \end{aligned}$$

By equalizing the coefficient of x^i on both sides we obtain.

$$\begin{aligned} a_0 &= r - x_0q_0 \\ a_1 &= q_0 - x_0q_1 \\ a_2 &= q_1 - x_0q_2 \\ a_{n-2} &= q_{n-3} - x_0q_{n-2} \\ a_{n-1} &= q_{n-2} \end{aligned}$$

From the last $q_{n-2} = a_{n-1}$ and from the equation before the last, $q_{n-3} = a_{n-2} + x_0q_{n-2}$. Similarly $q_i = a_{i+1} + x_0q_{i+1}$ for all i.

```
q[n-2] = a[n-1]
for(i=n-3;i>=0;i--)
     q[i] = a[i+1] + x_0 * q[i+1]
```

Therefore it takes $O(n)$ time to find the coefficient of $q(x)$. Total time is thus $O(n)$ and since we need $\Omega(n)$ time just to read $A(x)$ the results follows.

■

Exercise 462.

Show that interpolation can be done in $O(n^2)$ time.

Proof.

We are going to refer to Lagrange's interpolation formula as needed. Let $T(x) = (x-x_1)\dots(x-x_n)$. Each of these monomials $x-x_i$ is of degree 1! The time it takes to find $T(x)$ is by by the previous problem equal to $n(n-1)/2$ since we have n monomials and pairs of their degrees lead to $n(n-1)/2$ terms $d_i d_j$ where $d_i = d_j = 1$. Therefore computing $T(x)$ requires $O(n^2)$ time.

A. Time to find $T(x)$ is $O(n^2)$.

Consider $g_i(x)$ one of the n Lagrange's polynomials. The numerator of $g_i(x)$ is $(x-x_1)(x-x_2)\dots(x-x_{i-1})(x-x_{i+1})\dots(x-x_n)$ and the denominator is $(x_i-x_1)(x_i-x_2)\dots(x_i-x_{i-1})(x_i-x_{i+1})\dots(x_i-x_n)$. The denominator can be computed with at most n multiplications and n additions since it involves scalars. The nominator however of $g_i(x)$ is $T(x)/(x-x_i)$ in other words it is the polynomial $q(x) = g_i(x)$ such that $T(x) = q(x)(x-x_i)$. By problem 1 we can find $q(x)$ (i.e. $g_i(x)$) in $O(n)$ time. Therefore $g_i(x)$ can be found in $O(n)$ time. All g_i's can thus be found in $O(n^2)$ time.

B. Time to find $g_i(x)$ for all i is $O(n^2)$.

The polynomial $g(x) = b_1 g_1(x) + \dots + b_n g_n(x)$, where $((x_1,b_1),\dots,(x_n,b_n))$ is its point value representation can be found by adding n polynomials of degree at most n in $O(n^2)$ time. Each product $b_i g_i$ requires $O(n)$ time, therefore $g(x)$ can be found in $O(n^2)$ time.

C. Time to find $b_i g_i(x)$ for all i is $O(n^2)$.

D. Time to find $\sum_i b_i g_i(x)$ is $O(n^2)$.

Total time is thus $O(n^2)$ as required. We are going to refer to the notes of the Subject related to Lagrange's interpolation formula. Let $T(x) = (x-x_1)\dots(x-x_n)$. Each of these monomials $x-x_i$ is of degree 1! The time it takes to find $T(x)$ is by Homework 1/Problem 4 equal to $n(n-1)/2$ since we have n monomials and pairs of their degrees lead to $n(n-1)/2$ terms $d_i d_j$ where $d_i = d_j = 1$. Therefore computing $T(x)$ requires $O(n^2)$ time.

■

Exercise 463.

Karatsuba-Ofman for polynomials.

Proof.
A polynomial of degree bound n is of degree at most $n-1$. Let $f(x)$ (and similarly for $g(x)$) be two polynomial splits around the low order and the high order terms as follows, assuming $m = \lceil n/2 \rceil$.

$$\begin{aligned} f(x) &= f_0 + \ldots + f_{m-1}x^{m-1} + f_m x^m + \ldots + f_{n-1}x^{n-1} \\ &= f_L(x) + f_H(x)x^m \end{aligned}$$

where f_L and f_H are defined as follows:

$$f_L(x) = f_0 + f_1 x + \ldots + f_{m-1}x^{m-1},$$

and

$$f_H(x) = f_m + f_{m+1}x + \ldots + f_{n-1}x^{n-1-m}.$$

Note that $n-1-m = n-1-\lceil n/2 \rceil \leq \lceil n/2 \rceil - 1 = m-1$. Then by

$$fg = f_L g_L + (f_L g_H + g_L f_H)x^m + f_H g_h x^{2m} \tag{13.2}$$

we observe that

$$(f_L g_H + g_L f_H) = (f_L - f_H)(g_H - g_L) + f_L g_L + f_H g_H$$

Therefore if

$$h_1 = f_L g_L, h_2 = f_H g_H, h_3 = (f_L - f_H)(g_H - g_L)$$

Then

$$fg = h_1 + (h_3 + h_1 + h_2)x^m + h_2 x^{2m}. \tag{13.3}$$

Therefore polynomial multiplication between f and g of degree bounds n (or degrees at most $n-1$) involves 3 polynomial multiplications of polynomials of degree bounds m (or degree at most $m-1$), plus additions which are $O(m)$ operations since all polynomials are degree bound m; note that multiplication of a "shift" of the coefficients of $g(x)$ i.e. it requires linear time $O(m)$. Therefore we obtain the following recurrence for the running time. Assume n is even.

$$M(n) = 3M(m) + O(m) = 3M(n/2) + O(n)$$

providing a solution of $M(n) = \Theta(n^{\lg 3})$. The odd-even split is similar. We skip the details except to note that for the FFT problem we are going to establish it again.

■

Exercise 464.

You are given three polynomials $f(x), g(x), h(x)$ of degree bounds $n+1$, $n+1$, and $2n+1$ respectively. We would like to verify whether $f(x)g(x) = h(x)$ very fast but not with absolute certainty. The verification algorithm does not need to be deterministic; it can be a randomized algorithm.

One naive approach is to find the product $f(x)g(x)$ and compare it to $h(x)$. Finding the product takes $O(n^2)$ time (or $O(n \lg n)$ using FFT-based methods that we are going to discuss in due time) and comparison of $f(x)g(x)$ and $h(x)$ takes $O(n)$ time. The total amount of time is, however, $\omega(n)$.

Give a randomized algorithm that determines in $O(n)$ time whether $h(x)$ is $f(x)g(x)$ with confidence at least $1 - 1/n^{10}$. In addition the number of random bits you use should not exceed $O(\lg n)$. For example if you want to draw a random number in the interval $[1, N]$ it will cost you $\lg N$ bits because the largest such number requires about $\lg N$ bits to be represented.

Proof.

Choose a random integer number between 1 and n^{11}. Call this number z. Evaluate $f(z), g(z)$ and $h(z)$ in $O(n)$ overall. Then multiply $f(z)g(z)$ and compare the result to $h(z)$. Note that $f(z), g(z), h(z), f(z) \cdot g(z)$ are all (possibly integer) numbers!

Case 1. $f(z)g(z) \neq h(z)$. It is obvious that $f(x)g(x) \neq h(x)$ because otherwise $f(z)g(z) = h(z)$. The return statement in line 4 of Check-1 below is always correct.

Case 2. $f(z)g(z) = h(z)$. It might be the case that $f(x)g(x) = h(x)$ or it might be the case that z is a root of the polynomial $B(x) = f(x)g(x) - h(x)$. If z is such a root, then $B(z) = 0$ and consequently $f(z)g(z) = h(z)$. Now $B(x)$ is of degree n, therefore it has at most n distinct roots (integer or not). What is the probability that a random integer in the selected range is a root? This is at most $n/n^{11} = 1/n^{10}$. Thus the return in line 5 of Check-1 is almost always correct and reliable. It might fail to give the correct answer with probability at most $\leq 1/n^{10}$, or equivalently it will give the correct answer with probability at least $\geq 1 - 1/n^{10}$.

```
   Check-1 (f,g,h,n)
1. Pick at random z in [1..n**11];
2. Compute f(z), g(z), and h(z)
3. Computer B(z) = f(z)g(z) - h(z)
4. if (B(z) != 0) return ("f . g != h") //Returns correct answer
5. if (B(z) == 0) return ("f . g == h") // Returns an unreliable answer
```

How many random bits did we use? $\lg n^{11} = \Theta(\lg n)$. What if we use the following algorithm ? Show that it is as good as the previous one probabilisticallly. In addition it use $10 \lg n^2$ random bits, less than the previous version; asymptotically speaking however there is not much difference.

```
   NewCheck (f,g,h,n)
1. Pick at random z in [1..n^2];
2. Compute f(z), g(z), and h(z)
3. Computer B(z) = f(z)g(z) - h(z)
4. if (B(z) != 0) return ("f . g != h") //Returns correct answer
5. if (B(z) == 0) return ("") // Returns but with

   Check-2(f,g,h,n)
1. for(i=1;i<=10;i++)
2.  NewCheck(f,g,h,n);
3. return("f.g == h");
```

■

Exercise 465.

We want to perform polynomial division. Suppose we are given a polynomial $f(x)$ of degree n and a polynomial $g(x)$ of degree-bound $n+1$ (i.e. its degree is at most n). We want to find the polynomials $q(x) = f(x)/g(x)$ and $r(x)$ such that

$$f(x) = g(x)q(x) + r(x)$$

Polynomials $q(x)$ and $r(x)$ are known as the quotient and the remainder of the division. Since both $f(x)$ and $g(x)$ are of degree at most n and based also on the properties of the division, we conclude that $q(x)$ and $r(x)$ are of degree at most n as well.

Inverse of a polynomial

For a polynomial $g(x)$ the inverse of $g_i(x)$ of $g(x)$ is the polynomial $g_i(x) = 1/g(x)$.

We can reduce polynomial division to multiplication by observing that $q(x) = f(x)/g(x) = f(x) \cdot (1/g(x)) = f(x) \cdot g_i(x)$. By the division properties, each one of $q(x)$ and $f(x)$ has degree at most n. Therefore the degree of $g_i(x)$ can be limited to n.

How do we find the inverse of a polynomial?

Let $g(x) = (1-x)$. Then $g_i(x) = 1+x+x^2+\ldots$ since

$$\begin{aligned}(1-x)(1+x+x^2+\ldots &= (1+x+x^2+\ldots)\\ &= 1.\end{aligned}$$

Similarly if $g(x) = 1+g_1x+\ldots$, i.e. its constant coefficient is 1, then

$$1/g(x) = 1/(1-(-g_1x-\ldots) = 1/(1-G(x)) = 1+G(x)+G^2(x)+\ldots.$$

Using these observations do the following.

a. Let $f(x) = 1+2x+x^2-4x^3-6x^4$. Let $g(x) = 1-2x^2$. What is the degree bound of $q(x)$ and $r(x)$ the quotient and remainder of the division of $f(x)$ and $g(x)$. Find $g_i(x) = 1/g(x)$.

b. Since $f(x)(1/g(x)) = f(x)g_i(x) = q(x)$ and the degree bound of $q(x)$ is n disregard all higher order term in $g_i(x)$, i.e. find $G_i(x) = g_i(x) \bmod x^5$.

c. Find the product $Q_i(x) = f(x)G_i(x)$. Since the product is the quotient of $f(x)/g(x)$ and the quotient is of degree bound n, disregard the higher order terms i.e. find $q(x)$ such that $q(x) = Q_i(x) \bmod x^5$.

d. Find $r(x)$, the remainder of the division of $f(x)$ and $g(x)$.

e. Give $q(x), r(x)$ such that $f(x) = g(x)q(x) + r(x)$.

Proof.

This problem is straightforward.

(a) The quotient has degree at most 2, the difference of the degrees of $f(x), g(x)$ i.e. its degree bound is 3. The remainder has degree less than the degree of $g(x)$, i.e. its degree bound is 2. We note that $1/g(x) = 1/(1-2x^2)$. Then,

$$\frac{1}{1-h(x)} = 1+h(x)+h^2(x)+h^3(x)+\ldots+h^i(x)+\ldots$$

and therefore

$$g_i(x) = \frac{1}{1-2x^2} = 1+2x^2+4x^4+\ldots+2^ix^{2i}+\ldots$$

It is also easy to verify that $g(x)g_i(x) = 1$.

(b) We have that

$$G_i(x) = g_i(x) \bmod x^5 = 1+2x^2+4x^4$$

(c) $Q_i(x)$ is given by the following expression.

$$Q_i(x) = 1+2x+3x^2-8x^6-16x^7-24x^8$$

and thus $q(x)$ is given by the following expression.

$$q(x) = Q_i(x) \bmod x^5 = 1 + 2x + 3x^2.$$

(d) Then $r(x) = f(x) - g(x)q(x) = 0$.
(e) $q(x) = 1 + 2x + 3x^2$ and $r(x) = 0$, as required.

■

Exercise 466.

a. Let $f(x) = 1 + 2x - 4x^2 - 6x^3 + 3x^4 + 4x^5$. Let $g(x) = 1 - x^2$. What is the degree bound of $q(x)$ and $r(x)$ the quotient and remainder of the division of $f(x)$ and $g(x)$. Find $g_i(x) = 1/g(x)$.
b. Since $f(x)(1/g(x)) = f(x)g_i(x) = q(x)$ and the degree bound of $q(x)$ is n disregard all higher order term in $g_i(x)$, i.e. find $G_i(x) = g_i(x) \bmod x^6$.
c. Find the product $Q_i(x) = f(x)G_i(x)$. Since the product is the quotient of $f(x)/g(x)$ and the quotient is of degree bound n, disregard the higher order terms i.e. find $q(x)$ such that $q(x) = Q_i(x) \bmod x^6$.
d. Find $r(x)$, the remainder of the division of $f(x)$ and $g(x)$.
e. Give $q(x), r(x)$ such that $f(x) = g(x)q(x) + r(x)$.

Proof.
This problem is straightforward.
a. The degree bounds for q, r are 4 and 2 respectively. $g_i(x) = 1 + x^2 + x^4 + \ldots + x^{2i} + \ldots$.
b. $G_i(x) = 1 + x^2 + x^4$.
c. $Q_i(x) = f(x)G_i(x) = 1 + 2x - 3x^2 - 4x^3 - x^6 - 2x^7 + 3x^8 + 4x^9$. Thus $q(x) = 1 + 2x - 3x^2 - 4x^3$.
d. Then $r(x)$ is $r(x) = 0$.
e. $q(x) = 1 + 2x - 3x^2 - 4x^3$ and $r(x) = 0$. That's how computers divide polynomials.

■

Exercise 467.

Derive a point value representation for $A^{rev}(x) = \sum_{j=0}^{n-1} a_{n-1-j}x^j$ from a point-value representation for $A(x) = \sum_{j=0}^{n-1} a_j x^j$, assuming that none of the points in the point-value representation of $A(x)$ is 0.

Proof.
Let

$$A(x) = a_0 + a_1 x + \ldots + a_i x^i + \ldots + a_{n-1}x^{n-1}$$

Then

$$A^{rev}(x) = a_{n-1} + a_{n-2}x + \ldots + a_i x^{n-1-i} + \ldots + a_0 x^{n-1}$$

We substitute $1/y$ for x in the latter (assuming of course $y \neq 0$). Then

$$A^{rev}(1/y) = a_{n-1} + a_{n-2}(1/y) + \ldots + a_i(1/y)^{n-1-i} + \ldots + a_0(1/y)^{n-1}$$

By multiplying the latter with y^{n-1} we obtain that

$$y^{n-1}A^{rev}(1/y) = a_{n-1}y^{n-1} + a_{n-2}y^{n-2} + \ldots + a_i y^i + \ldots + a_0$$

The right hand side is however $A(y)$. Therefore

$$y^{n-1}A^{rev}(1/y) = A(y) \tag{13.4}$$

$$A^{rev}(1/y) = \frac{1}{y^{n-1}}A(y) \tag{13.5}$$

$$A^{rev}(x) = x^{n-1}A(1/x) \tag{13.6}$$

A point-value representation of $A(y)$ means that there are available n pairs $(x_i, A(x_i))$. Let

$$(x_i, A(x_i))$$

be then a point-value representation of $A(x)$ at the x_i, $i = 1, \ldots, n$. A point-value representation of $A(x)$ means that there are available n pairs $(y_i, A(y_i))$. Let

$$(y_i, A(y_i))$$

be then a point-value representation of $A(x)$ at the y_i, $i = 1, \ldots, n$, such that all $y_i \neq 0$. Since by equation Eq. 13.4 we have that for $y = y_i$, $A(y_i) = y_i^{n-1}A^{rev}(1/y_i)$ we conclude that $A^{rev}(1/y_i) = A(y_i)/y_i^{n-1}$. Therefore a point-value representation of $A^{rev}(x)$ can be derived from a point-value representation of $A(x)$ as follows

$$(x_i, A^{rev}(x_i)) = (1/y_i, A(y_i)/y_i^{n-1})$$

For the latter point-value representation to exist we require that the two divisions be possible, i.e. $y_i \neq 0$ for all $i = 1, \ldots, n$. Therefore a point-value representation for $A(x)$ at y_i yields a point-value representation for $A^{rev}(x)$ at $1/y_i$.

■

Exercise 468.

Evaluate the following polynomial of degree n, at $x = c$ using $o(n)$ additions/subtractions and $o(n)$ multiplications (you are not allowed to use divisions or any operations other than $+, -, *$).

$$f(x) = \sum_{k=0}^{n} \binom{n}{k} x^k$$

[Note:

$f(n) = o(g(n))$ if and only if $\lim_{n\to\infty} f(n)/g(n) = 0$.
$\binom{n}{k} = n!/(k!(n-k)!)$
$n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$].

Proof.
From discrete math,

$$(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$$

Therefore after performing one addition to obtain $X = 1 + x = 1 + c$, we need to compute X^n in $O(\lg n)$ multiplications using the power algorithm. The total operator count of $O(\lg n)$ is $o(n)$.

■

Exercise 469.

Evaluate the following polynomial of degree n, where $n = 2m+1$, at $x = c$ using $(n+1)/2+O(1)$ additions/subtractions and $(n+1)/2+O(1)$ multiplications (you are not allowed to used divisions or any operations other than $+,-,*$).

$$f(x) = 1+x+x^2+\ldots+x^n$$

Hint: Use ideas from the operation scheduling of the Munro-Paterson algorithm.

Proof.
Since $n = 2m+1$ and $f(x)$ can be rewritten as follows

$$f(x) = (1+x)+(1+x)x^2+(1+x)x^4+\ldots+(1+x)x^{2m}$$

We will utilize the very first step in the Munro-Paterson computation. A straight line program that evaluates $f(x)$ at $x = c$ is the following one.

```
I1:  t= c * c;          //  Compute x*x at x=c
I2:  s= 1+c;            //  Compute (1+x) at x=c
I3:  M= s;              //  Initialize M= 1+x at x=c
I4:  F= s;              //  Initialize P= 1+x at x=c


M1:  M= M*t;            //  M= (1+x) x**2
A1:  F= F+M             //  F= (1+x) + (1+x) x**2


M2:  M= M*t;            //  M= (1+x) x**4
A2:  F= F+M             //  F= (1+x) + (1+x) x**2 + (1+x) x**4


M3:  M= M*t;            //  M= (1+x) x**6
A3:  F= F+M             //  F= (1+x) + (1+x) x**2 + ... + (1+x) x**6


Mi:  M= M*t;            //  M= (1+x) x**(2*i)
Ai:  F= F+M             //  F= (1+x) + (1+x) x**2 + ... (1+x) x** (2*i)


   .....


Mm:  M= M*t;            //  M= (1+x) x**(2*m)
Am:  F= F+M             //  F= (1+x) + (1+x) x**2 + ... (1+x) x** (2*m)
```

In the first four initialization steps $I1...I4$ one 1M and 1A are performed. In the remaining 2m steps m M's are performed in the M steps and m $A's$ are performed in the A steps. Total number of additions is $m+1$ and total number of multiplications is also $m+1$. Therefore since $m+1 = (n+1)/2$, we perform $(n+1)/2$ M's and so many $(n+1)/2$ A's. ■

13.3 Matrices

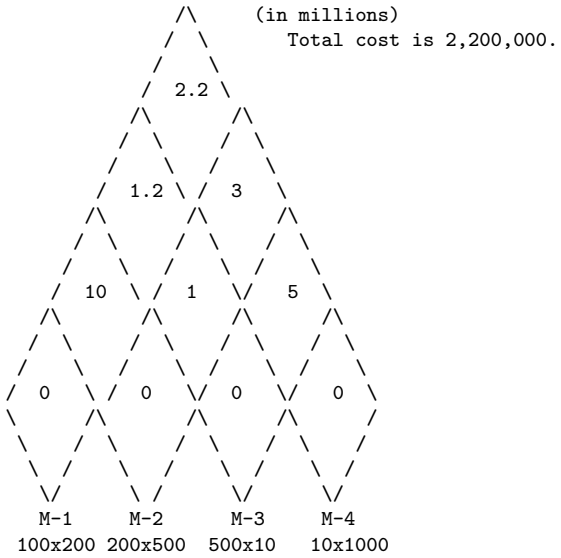
Exercise 470.

What's the best way to find the product $M_1 \times M_2 \times M_3 \times M_4$ where the matrices are of dimension 100×200, 200×500 , 500×10, and 10×1000 respectively?

Proof.
Easy.

$$((M_1\,(M_2 \times M_3)) \times M_4)$$

at a cost of 2,200,000.



■

Exercise 471.

CLRS [6] or based on.

What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?

Proof.

Strassen's algorithm uses the pattern of multiplication for 2×2 matrices and divide-and-conquer by splitting a matrix multiplication of $n \times n$ matrices into a matrix multiplication of $n/2 \times n/2$ matrices to derive a time bound for matrix multiplication given by the recurrence $T(n) = 7T(n/2) + O(n^2)$. In such a recurrence the 7 indicates the number of scalar multiplications in multiplying 2×2 matrices or the number of $n/2 \times n/2$ matrix multiplications in the recursive (divide-and-conquer) decomposition. Note that Strassen's method on 2×2 matrices can not be improved. We cannot multiply 2×2 matrices with say 6 (scalar) multiplications.

Now if we can multiply 3×3 matrices and k is the number of scalar multiplications required and we use a recursive decomposition of an $n \times n$ matrix into $n/3 \times n/3$ matrices we get a recurrence for the running time of this method of

$$T(n) = kT(n/3) + O(n^2).$$

The solution of this recurrence (master method) yields $T(n) = \Theta(n^{\log_3 k})$. Therefore if we compare the exponent $\log_3 k$ to $\lg 7$, then for $k = 21$ we have $\log_3 21 \approx 2.771 < 2.8073 = \lg 7$. For that value of $k = 21$ or smaller a by-3 decomposition beats Strassen's method of by-2 decomposition.

■

Exercise 472.

CLRS [6] or based on.

What is the largest k such that if you can multiply 5×5 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?

Proof.

Strassen's algorithm uses the pattern of multiplication for 2×2 matrices and divide-and-conquer by splitting a matrix multiplication of $n \times n$ matrices into a matrix multiplication of $n/2 \times n/2$ matrices to derive a time bound for matrix multiplication given by the recurrence $T(n) = 7T(n/2) + O(n^2)$. In such a recurrence the 7 indicates the number of scalar multiplications in multiplying 2×2 matrices or the number of $n/2 \times n/2$ matrix multiplications in the recursive (divide-and-conquer) decomposition. Note that Strassen's method on 2×2 matrices can not be improved. We cannot multiply 2×2 matrices with say 6 (scalar) multiplications.

Now if we can multiply 3×3 matrices and k is the number of scalar multiplications required and we use a recursive decomposition of an $n \times n$ matrix into $n/3 \times n/3$ matrices we get a recurrence for the running time of this method of

$$T(n) = kT(n/5) + O(n^2).$$

The solution of this recurrence (master method) yields $T(n) = \Theta(n^{\log_5 k})$. Therefore if we compare the exponent $\log_5 k$ to $\lg 7$, then for $k = 91$ we have $\log_5 91 \approx 2.802 < 2.8073 = \lg 7$, but for $k = 92$ we have $\log_5 92 \approx 2.809 > 2.8073$. Therefore the largest such k is 91, and the algorithm would have an exponent 2.802 rather than 2.8073.

■

Exercise 473.

CLRS [6] or based on. Might have appeared earlier.
V. Pan has discovered a way of multiplying 68×68 matrices using 132464 multiplications, 70×70 matrices using 143640 multiplications, 72×72 matrices using 155424 multiplications. Which method yields the best asymtotic running time when used in a divide and conquer matrix multiplication algorithm? How does it compare to Strassen's algorithm?

Proof.
This is similar to the previous problem. If we decompose not by-2 or by-3, but by-t, where t now is $68, 70, 72$ we obtain that the running time for such a decomposition is given by the recurrence,

$$T(n) = kT(n/t) + O(n^2).$$

with an asymptotically tight solution given by $T(n) = \Theta(n^{\log_t k})$. Therefore if we compare the exponent $\log_t k$ to $\lg 7$, then for $t = 68$, $\log_{68} 132464 = 2.795128$, $t = 70$, $\log_{70} 143640 = 2.795122$, $t = 72$, $\log_{72} 155424 = 2.795147$. All three of them beat Strassen's exponent. The best of the three is the $t = 70$ split. Calculations used the Unix `bc` calculator and hopefully are correct!

■

Exercise 474.

Let $M(n)$ be the time to multiply two $n \times n$ matrices and let $S(n)$ be the time to square an $n \times n$ matrix. Show that multiplying and squaring have essentially the same difficulty: i.e. an $M(n)$ matrix multiplication algorithms implies an $O(M(n))$ squaring algorithm and an $S(n)$ squaring algorithms implies an $O(S(n))$ matrix multiplication algorithm.

Proof.
One direction is easy, the other is tricky.

Build a heftier matrix with more rows and columns! Let $M(n)$ be the time to multiply two $n \times n$ matrices and let $S(n)$ be the time to square an $n \times n$ matrix. Show that multiplying and squaring have essentially the same complexity: i.e. an $M(n)$ matrix multiplication algorithm implies an $O(M(n))$ squaring algorithm and and an $S(n)$ squaring algorithms implies an $O(S(n))$ matrix multiplication algorithm.

(a) One way is easy to show. If we can multiply we can square since $S(A) = A * A$. Therefore $S(n) = M(n)$.

(b) The other way requires the transformation (or other similar ones)

$$X = \begin{pmatrix} A & B \\ 0 & 0 \end{pmatrix}$$

The following matrix also works

$$Y = \begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}$$

Then

$$X^2 = \begin{pmatrix} A^2 & AB \\ 0 & 0 \end{pmatrix}$$

and

$$Y = \begin{pmatrix} 0 & BA \\ AB & 0 \end{pmatrix}$$

Note that the dimension of X is $2n \times 2n$ now. Then $M(n)$ is given by the expression,

$$M(n) = S(2n) + O(n^2)$$

where the first term is the time to square the $2n \times 2n$ X and $O(n^2)$ is the time to form X and retrieve AB from X^2. For both $S(n)$ and $S(2n)$ we have $S(n) = S(2n) = \Omega(n^2)$. If in addition $S(2n) = O(S(n))$ then the result follows.

Theorem. Matrix multiplication is no harder than squaring as long $S(n) = \Omega(n^2)$ and $S(n) = O(S(2n))$.

Note. The transformation $((A+B)^2 - (A-B)^2)/4$ does not work. If A, B are numbers indeed $AB = ((A+B)^2 - (A-B)^2)/4$. However if A, B are matrices that are not commutative (i.e. $AB \neq BA$), then $((A+B)^2 - (A-B)^2)/2 = AB + BA$ which might not be $2AB$. From there, it is not easy to retrieve AB unless of course we know BA or A, B are commutative. Therefore this transformation/reduction for AB holds for commutative matrices only.

■

Exercise 475.

(a) Solve the recurrence for the number of operations in Strassen's method exactly, i.e. solve

$$M(n) = 7M(n/2) + (18/4)n^2.$$

What is the base case? Explain. How many operations are required for the base case? Explain. Does it matter whether one chooses the base case to be $n = 1$ or $n = 2$?

(b) What happens for the questions above if one uses the modification of the original Strassen method with the recurrence becoming

$$M(n) = 7M(n/2) + (15/4)n^2?$$

Proof.

Instead of solving the indicated recurrence we are going to solve the more general recurrence

$$M(n) = 7M(n/2) + an^2$$

with $M(x) = b$, for any constant a, b. For Strassen's method $a = 18/4 = 4.5$. The parameter a decides the number of additions performed on the dimension $n/2$ submatrices. The constant b determines how we multiply the 2×2 matrices of the base case. The implicit assumption in Strassen's method is that n is a power of two.

Claim 1. Strassen's method performance remains asymptotically the same no matter how we perform the multiplication of the base case 2×2 matrices. In essence, we can use any method to multiply such base case matrices. This is because $M(2) = \Theta(1)$.

Claim 2. Binet's method requires $M(1) = 1$, $M(2) = 12$, $M(4) = 112$, $M(8) = 960$, $M(16) = 7936$ and in genereal $M(n) = n^3 + n^2(n-1) = 2n^3 - n^2$ operations. Strassen for 2×2 matrices uses 25 (18 or 15 As/Ss, 7 Ms) operations, vs Binet's 12 (4As, 8Ms). The base case $n = 2$ is better in general than $b = 1$.

Claim 3. The first case where Strassen beats Binet's method in total operations is $M(16)$ where 8×8 matrices are multiplied by Binet's method. This is because $M(16) = 7M(8) + 4.5 \cdot 16^2 = 7 \cdot 960 + 4.5 \cdot 16^2 = 7872$ and this is better than 7936. On the other hand $M(8) = 7M(4) + 4.5 * 64 = 7 * 112 + 4.5 * 64 = 1072$ which is greater than 960. Therefore base case is $n = 8$ and we use Binet's method for 960 operations then, i.e. $x = 8$ and $b = 960$.

We start the analysis of the recurrence in the general case.

$$\begin{aligned}
M(n) &= 7M(n/2) + an^2 \\
&= 7(7M(n/2^2) + a(n/2)^2) + an^2 \\
&= 7^2 M(n/2^2) + 7^1 a(n/2)^2 + 7^0 an^2 \\
&= 7^i M(n/2^i) + 7^{i-1} a(n/2^{i-1})^2 + \ldots + 7^1 a(n/2^1)^2 + 7^0 a(n/2^0)^2 \\
&= 7^i M(n/2^i) + an^2(7/4)^{i-1} + \ldots + an^2(7/4)^1 + an^2(7/4)^0 \\
&= 7^i M(n/2^i) + an^2 \frac{(7/4)^i - 1}{7/4 - 1}
\end{aligned}$$

For $i = \lg n - 1$ we have $n/2^i = 2$ and thus $M(n/2^i) = M(2) = b$ Then the expression for $M(n)$ becomes (note that $n^2 n^{\lg(7/4)} = n^{\lg 7}$).

$$\begin{aligned}
M(n) &= 7^i M(n/2^i) + an^2 \frac{(7/4)^i - 1}{7/4 - 1} \\
&= 7^{\lg n - 1} M(2) + an^2 \frac{(7/4)^{\lg n - 1} - 1}{3/4} \\
&= 7^{\lg n} M(2)/7 + (4an^2/3)((7/4)^{\lg n - 1} - 1) \\
&= 2^{\lg 7 \lg n} M(2)/7 + (4an^2/3)\left((2)^{\lg(7/4)\lg n}/(7/4) - 1\right) \\
&= n^{\lg 7} M(2)/7 + (4an^2/3)\left((4/7)n^{\lg(7/4)} - 1\right) \\
&= n^{\lg 7} b/7 + (16a/21)n^{\lg 7} - 4an^2/3.
\end{aligned}$$

Therefore,

$$M(n) = (b/7 + 16a/21)n^{\lg 7} - 4an^2/3$$

Therefore for any constant a, b we have that $M(n) = \Theta(n^{\lg 7})$.
Base case $n = 2$**.** If $a = 18/4 = 4.5$ (as it is in the recursive Strassen steps) and $b = 12$ (Binet's / standard method for multiplying base case 2×2 matrices, with $M(2) = 12$), then the constant of the high order term becomes ≈ 5.14.

$$M(n) \approx 5.14n^{\lg 7} - 6n^2$$

If one uses the modification of Strassen's method of the textbook, the number of additions is 15 instead of 18 which further reduces a to $a = 15/4$ instead of $a = 18/4$ thus giving

$$M(n) \approx 4.58n^{\lg 7} - 5n^2.$$

Had we used Strassen in the base case, b would have been 25 instead of 12 which would have given worse figures.
Base case for $n = 1$**.** If one decides that the base case is not $M(2)$ but $M(1)$ then $M(1) = 1$ and $i = \lg n$. Then the solution one derives is

$$M(n) = (1 + 4a/3)n^{\lg 7} - 4an^2/3.$$

For Strassen, $a = 18/4$ and for the book's modification $a = 15/4$. The corresponding results are

$$M(n) \approx 7n^{\lg 7} - 6n^2,$$

and

$$M(n) \approx 6n^{\lg 7} - 5n^2$$

respectively. Both however are worse than the results obtained from having a base case $n = 2$ (i.e. $M(2)$).

■

Exercise 476.

Strassen's method works provided that the dimension n of the two matrices is a power of two. If n however is not a power of two, augmenting the two matrices so that their dimension becomes a power of two i.e. the two $n \times n$ matrices become $2^{\lceil \lg n \rceil} \times 2^{\lceil \lg n \rceil}$ creates matrices that may be too large. In such a case the constant hidden in $O(n^{\lg 7})$ is too large. A better method is one that increases n to $2^p m \geq n$, where p, m are integers such that $p \leq \lfloor \lg n \rfloor$ and such that $2^p m$ is much closer to n than $2^{\lceil \lg n \rceil}$.

Describe in pseudocode this new variant of Strassen's algorithm. What is the base case? How do you implement the base case? Give the recurrence for the running time for this new algorithm and provide an asymptotic solution for its running time.

Proof.

If n is not a power of two we can pad zeroes to the matrix so that the matrix becomes a power of two, i.e. 2^k, where 2^k is the smallest power of two so that $2^k > n$, i.e. $k = \lceil \lg n \rceil$. This in the worst case almost doubles the dimension i.e. quadruples the size of the matrix. In general, this is too expensive. Instead we can make the dimension of the padded matrix be $2^p m \geq n$, so that $2^p m$ is much closer to n that $2^{\lceil \lg n \rceil}$. For example if $n = 150$ padding to a power of two creates an array of dimension 256 since $\lceil \lg n \rceil = 8$. However we observe that $2^5 \cdot 5 = 160$. Therefore by choosing $p = 5$ and $m = 5$ we get a matrix of dimension 160 which is much closer to $n = 150$ than 256. In general we are going to choose $p = \lfloor \lg n - d \rfloor \leq \lg n$ and then m so that $m = \lceil n/2^p \rceil$. This way the product $2^p m = 2^p \lceil n/2^p \rceil \geq n$ since $x + 1 \geq \lceil x \rceil \geq x$ and $x \geq \lfloor x \rfloor \geq x - 1$. The parameter d is a small positive integer constant that will be determined at the end of the proof. This would make m a small constant as well. By the properties of the floor and ceiling functions we obtain that since $p = \lfloor \lg n - d \rfloor$

$$\lg n - d - 1 \leq p \leq \lg n - d$$

Similarly since $m = \lceil n/2^p \rceil$ we obtain that

$$n/2^p \leq m = \lceil n/2^p \rceil \leq \lceil n/2^{\lg n - d - 1} \rceil$$

This can be further refined as

$$2^d \leq n/2^{\lg n - d} \leq m \leq \lceil 2^{d+1} \rceil = 2^{d+1}$$

since d is assumed to be an integer and thus 2^{d+1} is an integer as well. Summarizing the bounds for m and p we have

$$2^d \leq m \leq 2^{d+1}$$

and

$$\lg n - d - 1 \leq p \leq \lg n - d$$

Therefore for such choices of p, m, we follow the standard Strassen decomposition if the dimension is divisible by two. After p decompositions the original problem of multiplying $2^p m \times 2^p m$ matrices has been reduced to the problem of multiplying $m \times m$ matrices. Since we do not know the exact values for m we cannot use this piece of information to optimize the multiplication of $m \times m$ matrices they way Strassen did for 2×2 matrices. However we can use the Claim of Problem 4, and thus multiply $m \times m$ matrices greedily in $\Theta(m^3)$ time (or at most $2m^3$ operations) using the standard (Binet's) method. The recurrence we are going to solve for the running time is

$$M(2^p m) = 7M(2^{p-1} m) + a(2^p m)^2$$

with boundary condition $M(m) = 2m^3$. In the last step we divide and multiply all the terms other than the first

by 4^p.

$$\begin{aligned}
M(2^p m) &= 7M(2^{p-1}m) + a(2^p m)^2 \\
&= 7^2 M(2^{p-2}m) + 7a(2^{p-1}m)^2 + a(2^p m)^2 \\
&= 7^p M(m) + 7^{p-1}a(2^1 m)^2 + \ldots + 7a(2^{p-1}m)^2 + a(2^p m)^2 \\
&= 7^p M(m) + 7^{p-1}a(2^1 m)^2 + \ldots + 7a(2^{p-1}m)^2 + 7^0 a(2^p m)^2 \\
&= 7^p M(m) + am^2 4^p \left((7/4)^{p-1} + \ldots + (7/4)^1 + (7/4)^0\right) \\
&= 7^p M(m) + am^2 4^p \left(((7/4)^p - 1)/(7/4 - 1)\right) \\
&= 7^p M(m) + (4/3)am^2 4^p \left((7/4)^p - 1\right)
\end{aligned}$$

We examine the two terms in the derived expressions for $M(2^p m)$ separately and we use the fact that $p \leq \lg n - d$ and $m \leq 2^{d+1}$ and $M(m) = 2m^3$.

$$\begin{aligned}
7^p M(m) &= 2^{p \lg 7} M(m) \\
&\leq 2^{(\lg n - d)\lg 7} M(m) \\
&= \frac{2^{\lg n \lg 7}}{2^{d \lg 7}} M(m) \\
&= \frac{n^{\lg 7}}{2^{d \lg 7}} M(m) \\
&= \frac{n^{\lg 7}}{2^{d \lg 7}} (2m^3) \\
&\leq \frac{n^{\lg 7}}{2^{d \lg 7}} (2 \cdot 8 \cdot 2^{3d}) \\
&\leq \frac{n^{\lg 7}}{7^d} (16 \cdot 2^{3d}) \\
&\leq (16(8/7)^d) n^{\lg 7}.
\end{aligned}$$

For the latter term we perform similar operations to derive the following upper bound.

$$\begin{aligned}
(4/3)am^2 4^p((7/4)^p - 1) &\leq (4/3)am^2 4^p (7/4)^p \\
&= (4/3)am^2 7^p \\
&\leq (4/3)am^2 7^{\lg n - d} \\
&\leq (4/3)a2^{2(d+1)}(7^{\lg n}/7^d) \\
&= (4/3)a2^{2(d+1)}(n^{\lg 7}/7^d) \\
&= (16/3)a4^d(n^{\lg 7}/7^d) \\
&= (16/3)a(4/7)^d n^{\lg 7}
\end{aligned}$$

Therefore we obtain that.

$$M(2^p m) \leq (16(8/7)^d + (16/3)a(4/7)^d) n^{\lg 7}$$

For $a = 4.5$ (Strassen) we obtain

$$M(2^p m) \leq (16(8/7)^d + 24(4/7)^d) n^{\lg 7}$$

If $a = 15/4$ (textbook improvement on Strassen) then we get a slightly improved bound of

$$M(2^p m) \leq (16(8/7)^d + 20(4/7)^d) n^{\lg 7}$$

Trying $d = 1,2,3,4$ we obtain that the constant term is minimized for $d = 3$ and thus we obtain $M(2^p m) \leq 28.36 n^{\lg 7}$ for p, m chosen in terms of n according to the formulae introduced earlier. The textbook improvement on strassen's algorithm yields

Comment. The analysis of the algorithm is very rough and non-optimal in the last steps when we bound the two terms of $M(2^p m)$. One could take as a lower bound of $m = \lceil n/2^p \rceil$ not 2^{d+1} as we did here but the more accurate $n/2^p + 1$. This could lead to a reduction of the constant 28.36 to a constant c that is between 4.3 and 4.7. Then this variant would be as good, if not slightly better, than Strassen's algorithm that assumes that n is a power of two.

Final Comment. In prior problems better results were obtained when the base case was treated the standard way, i.e. the standard (Binet's) matrix multiplication method was used rather than Strassen's method and variants.

■

Exercise 477.

A Toeplitz matrix is an $n \times n$ matrix $A = a_{ij}$ such that $a_{ij} = a_{i-1,j-1}$ for $2 \leq i, j \leq n$.
(a) Is the sum of two Toeplitz matrices a Toeplitz matrix?
(b) Describe how to represent a Toeplitz matrix so that two $n \times n$ Toeplitz matrices can be added in $O(n)$ time.
(c) Give an $O(n \lg n)$-time algorithm for multiplying an $n \times n$ Toeplitz matrix by a vector of length n. Use your representation from part (b). Hint: FFT, Polynomial-multiplication may help.
(d) Given an efficient algorithm for multiplying two $n \times n$ Toeplitz matrices. Analyze its running time.

Proof.
Let for the sake of an example A be a Toeplitz matrix. Since for an $n \times n$ matrix $2n-1$ entries are uniques.

$$A = \begin{pmatrix} a_3 & a_4 & a_5 & a_6 \\ a_2 & a_3 & a_4 & a_5 \\ a_1 & a_2 & a_3 & a_4 \\ a_0 & a_1 & a_2 & a_3 \end{pmatrix}$$

we can represent a Toeplitz matrix by a vector $a = (a_0 \dots a_{2n-2})$ of length $2n-1$. In particular for our example $a = (a_0, \dots, a_6)$ suffices. We can also represent A by a polynomial of degree $2n-2$ as follows $A(x) = a_0 + a_1 x + \dots + a_{2n-2} x^{2n-2}$. We use the vector or polynomial representation of matrix A interchangeably in the remainder.
(a) Obviously yes; proof by inspection.
(b) To manipulate Toeplitz matrices use the function form; to add $A+B$ add $A(x)+B(x)$ in $O(n)$ time and we are done.
Consider vector $b^t = (b_3 b_2 b_1 b_0)$, an $n \times 1$ vector. We represent the vector as a polynomial of degree $n-1$ as follows $b(x) = b_0 + b_1 x + \dots + b_{n-1} x^{n-1}$. Then the matrix-vector product $y = Ab$ reduces to a polynomial multiplication problem $A(x)b(x)$.

$$y = Ab = \begin{pmatrix} a_3 & a_4 & a_5 & a_6 \\ a_2 & a_3 & a_4 & a_5 \\ a_1 & a_2 & a_3 & a_4 \\ a_0 & a_1 & a_2 & a_3 \end{pmatrix} \times \begin{pmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{pmatrix} = \begin{pmatrix} a_3 b_3 + a_4 b_2 + a_5 b_1 + a_6 b_0 \\ \dots \\ \dots \\ a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 \end{pmatrix}$$

It is straightforward to observe that the i-th entry (count of entries starts from 0 and the bottom of the vector NOT THE TOP) of the product $y = Ab$ is the coefficient of x^{n-1+i} of $A(x)b(x)$ where $0 \leq i \leq n-1$. Therefore if A is in the polynomial form we turn vector b in polynomial form as shown above, and the multiplication $A(x)b(x)$ in $O(n \lg n)$ time using FFT gives the matrix vector product Ab. The n entries of $y = Ab$ are the coefficient of x^{n-1+i} of $A(x)b(x)$ filling y bottom to top.
(d) One could employ the algorithm of part (c) to n times (number of column vectors of B) to derive an $O(n^2 \lg n)$ algorithm for this problem. However, there is a simpler solution that does not necessarily require FFT that finds the product in optimal $O(n^2)$. Initially we find the first column and row of the product; one could use 2 FFT computations for $O(nlgn)$ or the standard algorithm (Binet's method) in $O(n^2)$ time. Then the remainder of the results is being built row by row. Each element however uses the results of the first row/column; the additional computational load is one subtraction and one addition per entry for a total of $O(n^2)$ additional operations; therefore the total cost is $O(n^2)$ as claimed. For example let $A(x)$ and $B(x)$ are as before and $a(x)$ and $b(x)$ are defined as $a(x) = a_{n-1} + a_n x + \dots + a_{2n-2} x^{2n-2}$, and $b(x) = b_0 + \dots + b_{n-1} x^{n-1}$. Then the first row of the results can be obtained by reading off some coefficients of $a(x)B(b)$ and the first column by reading off coefficients of $b(x)A(x)$. The element at position (i, j) (top to bottom, 0 to $n-1$ number) is the one. IF this is confusing use the following notation. Let the element of the first row and column of A be a_{1j} and a_{i1} in the standard notation and similarly for B. After the first row and column of AB have been determined the remaining positions of $C = AB$ are determined by the following $c_{i,j} = c_{i-1,j-1} - a_{1,n+2-j} b_{n+2-i} + a_{i,1} b1j$.

■

13.4 Some challenging problems

Exercise 478.

Given y, m find $\lfloor y^{1/m} \rfloor$.

Proof.
We use Newton-Raphson approximation. Let $f(x) = y - x^m$. The solution for $f(x) = 0$ is $x = y^{1/m}$. We also have that $f'(x) = -mx^{m-1}$.
Let us use Newton-Raphson with

$$x_{i+1} = x_i - f(x_i)/f'(x_i) = x_i + \frac{y - x_i^m}{m x_i^{m-1}}. \tag{13.7}$$

In order to find an integer approximation, we modify Eq.13.7 so that we get integer values at various iterations.

$$x_{i+1} = x_i + \lfloor \frac{y - x_i^m}{m x_i^{m-1}} \rfloor. \tag{13.8}$$

This provides an integer sequence as long as x_0 the initial condition is also an integer. We choose x_0 such that

$$x_0 = 2^{1+\lfloor \lceil \lg y \rceil / m \rfloor} \geq 2^{\lceil \lceil \lg y \rceil / m \rceil} \geq 2^{\lg Y/m} = y^{1/m}, \tag{13.9}$$

i.e. $x_0 \geq y^{1/m}$. At the same time,

$$x_0 = 2^{1+\lfloor \lceil \lg y \rceil / m \rfloor} \leq 2 \cdot 2^{\lceil \lg y \rceil / m} \geq 4 \cdot 2^{\lg Y/m} = 4y^{1/m}, \tag{13.10}$$

and therefore $y^{1/m} \leq x_0 \leq 4y^{1/m}$.
Note 1. Starting with y, finding $\lceil \lg y \rceil$ is easy as it requires shift operations and can be done in $O(\lg n)$ shifts. Therefore x_0 is easy to compute.
Consider Eq. 13.9. For $x_0 \geq y^{1/m}$ we have that $x_0^m - y \geq 0$ and thus

$$y - x_0^m \leq 0. \tag{13.11}$$

By eq 13.8 because of eq 13.11 for $i = 1$ we obtain that $x_1 = x_0 + \lfloor < 0 \rfloor < x_0$, i.e. sequence $x_0, x_1, x_2, \ldots$ at its first step at least, is a decreasing sequence.
Consider again Eq. 13.8.

$$\begin{aligned} x_{i+1} &= x_i + \lfloor \frac{y - x_i^m}{m x_i^{m-1}} \rfloor \\ &\geq x_i + \frac{y - x_i^m}{m x_i^{m-1}} - 1 \end{aligned} \tag{13.12}$$

Let $y^{1/m} = Y$ and thus $y = Y^m$.

$$y = Y^m \tag{13.13}$$

Eq 13.12 because of Eq 13.13 becomes.

$$x_{i+1} \geq x_i + \frac{Y^m - x_i^m}{m x_i^{m-1}} - 1 \tag{13.14}$$

Suppose now that $x_i > Y = y^m$ which is true at least for the first few terms of the sequence because of Eq 13.9. We then obtain that noting that $x_i^j Y m - 1 - j \leq x_i^{m-1}$.

$$\begin{aligned} x_i^m - Y^m &= (x_i - Y)(x_i^{m-1} + x_i^{m-2} Y + \ldots + Y^{m-1}) \\ &= (x_i - Y) m x_i^{m-1}, \end{aligned} \tag{13.15}$$

i.e. $x_i^m - Y^m \leq (x_i - Y)mx_i^{m-1}$. Therefore, if $x_i > Y$, eq 13.14 because of Eq. 13.15 gives.

$$\begin{aligned} x_{i+1} &\geq x_i + \frac{Y^m - x_i^m}{mx_i^{m-1}} - 1 \\ &\geq x_i + \frac{(Y - x_i)mx_i^{m-1}}{mx_i^{m-1}} - 1 \\ &\geq x_i + (Y - x_i) - 1 \\ &\geq Y - 1 \\ &\geq y^{1/m} - 1 \\ &\geq \lfloor y^{1/m} \rfloor - 1. \end{aligned} \tag{13.16}$$

We have just proved that $x_{i+1} \geq \lfloor y^{1/m} \rfloor - 1$ provided that $x_i > Y = y1/m$.
Lemma 1. If $x_i > y^{1/m}$ then $x_{i+1} \geq \lfloor y^{1/m} \rfloor - 1$.
Lemma 2. If $x_i > y^{1/m}$ then $x_{i+1} \leq x_i$.
Therefore the sequence $x_0, x_1, x_2, \ldots$ may end up in one of two cases.
Case 1. $x_{n+1} = x_n$ occurs for the first time after a decreasing sequence i.e. we have $x_0 > x_1 > \ldots > x_n = x_{n+1}$. Note that $x_{i+1} = x_i + \lfloor < 0 \rfloor < x_i - 1$. In order to have $x_{n+1} = x_n$ we must have that x_n is such that $y - x_n^m \geq 0$, i.e. $x_n^m \leq y$ and thus $x_n \leq y^{1/m}$ i.e. we have that $x_{n-1} > y^{1/m} > x_n$.
Note that by Lemma 2 for $i = n-1$ we have that $x_{n-1} > y^{1/m}$. By the above arguments we have obtained that $y^{1/m} \geq x_n$. And by Lemma 1 we know that $x_n \geq \lfloor y^{1/m} \rfloor - 1$. Therefore we can find $\lfloor y^{1/m} \rfloor$ by just considering one of x_n and $x_n + 1$; one of these terms is $\lfloor y^{1/m} \rfloor$.
Case 2. We have that $x_{n+1} > x_n$; again this case indicates that $y - x_n^m > 0$ i.e. $x_n \leq y^{1/m}$. Since by Lemma 2 for $i = n-1$ we have that $x_{n-1} > y^{1/m}$, and we just showed that $y^{1/m} \geq x_n$, and $x_n \geq \lfloor y^{1/m} \rfloor - 1$ by Lemma 1, $\lfloor y^{1/m} \rfloor$ is one of x_n or $x_n + 1$. Also, if $x_{n+1} > x_n$, then $x_{n+2} > x_{n+1}$. This concludes the discussion.

■

Exercise 479.
Let us be given y,m and we want to check whether there exists integer x such that $x^m = y$

Proof.
Consider a prime divisor p of m. Since y is given, if $y = x^m = (x^{m/p})^p \geq 2^p$ and therefore $p \leq \lg y$. Consider all factors of m that are prime and less than $\lg y$. If among them there is a p such that $y = z^p$ we are done, we have found the pair (m,x) to be (p,z).

■

Exercise 480.

Given y check whether y is the integer power of an integer or not, i.e. whether there exist x,m such that $x^m = y$.

Proof.

If there are such x,m by taking logarithms of both sides we have that $m \lg x = \lg y$ and thus $m \leq \lg y$. We thus use the technique of the previous two-three problems

```
   Nth-root ( y) //Check whether y= x**m for intefer x,m
1. Compute A=lg(y); // m <= lgy
2. Generate all prime numbers less than A;call them p[1]...p[t].
3. for (i=1;i<=t;i++) {
4.   x = floor( y**(1/A[i]));
5.   if ( x**A[i] == y)  return (TRUE)
6. }
7. return(FALSE);
```

■

Exercise 481.
If we can lower triangular matrix multiply then we can multiply arbitrary matrices.

Proof.
No comments.

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & A & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 0 & 0 \\ B & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ AB & 0 & 0 \end{pmatrix}$$

or use the matrix below and square it,

$$\begin{pmatrix} 0 & 0 & 0 \\ B & 0 & 0 \\ 0 & A & 0 \end{pmatrix}$$

thus leading to if we can square lower triangular matrixes we can multiply arbitrary matrices!

■

Exercise 482.

If we can invert lower triangular matrices then we can multiply arbitrary matrices.

Proof.
No comments. Let matrix C be

$$C = \begin{pmatrix} I & 0 & 0 \\ B & I & 0 \\ 0 & A & I \end{pmatrix}$$

Then, we have for the inverse of C, which is $C^{-1} = inv(C)$.

$$C^{-1} = \begin{pmatrix} I & 0 & 0 \\ -B & I & 0 \\ AB & -A & I \end{pmatrix}$$

■

Exercise 483.
An easy one: If we can square polynomials then we can multiply polynomial (plus $O(n)$) other ops.

Proof.
This is the same result as for integers (or numbers)

$$f \cdot g = ((f+g)^2 - f^2 - g^2)/2.$$

■

Exercise 484.

Suppose three polynomials $f(x)$, $g(x)$, and $h(x)$ are given of degree bounds $n+1$, $n+1$, and $2n+1$ respectively. Verify whether $f(x) \cdot g(x) = h(x)$ faster than $O(n \lg n)$ worst-case time (i.e. in $o(n \lg n)$ time). Your confidence in the answer should be at least $1 - 1/n^k$, where k is an integer strictly greater than 1 i.e. $k > 1$. The verification algorithm does not need to be deterministic; it can be a randomized algorithm.

One naive approach is to find the product $f(x)g(x)$ and compare it to $h(x)$. Finding the product takes $O(n^2)$ time (or $O(n \lg n)$ using FFT-based methods) and comparison of $f(x)g(x)$ and $h(x)$ takes $O(n)$ time. The total amount of time is, however, $\omega(n)$.

Give a randomized algorithm that determines in $O(n)$ time whether $h(x)$ is $f(x)g(x)$ with confidence at least $1 - 1/n^{10}$. In addition the number of random bits you use should not exceed $O(\lg n)$. For example if you want to draw a random number in the interval $[1, N]$ it will cost you $\lg N$ bits because the largest such number requires about $\lg N$ bits to be represented.

Note that if you pick a random integer between integers 0 and x, it will cost you $O(\lg x)$ random bits at a cost of $O(\lg x)$ in the running time. If x is a real number or you pick a real number between 0 and x, it will cost infinite random bits at an infinite cost in running time.

Hint: Consider $z(x) = f(x)g(x) - h(x)$. For arbritrary integer m, how often is $z(m) = 0$? How many such m exist?

Proof.
Skipped.

■

Exercise 485.

You are given t and you are asked to compute t^{30} using only 6 multiplications in the word model. Show how to do it (use a STRAIGHT LINE PROG RAM) or show that it cannot be done. (No divisions are allowed.)

Proof.

```
1-2-3-5-10-20-30  or 1-2-3-6-12-24-30
They are known as chains.
       a1= t  * t    //t^2
       a2= a1 * a1   //t^4
       a3= a2 * a1   //t^6
       a4= a3 * a3   //t^12
       a5= a4 * a4   //t^24
       a6= a5 * a3   //t^30
```

■

Chapter 14

Parallel algorithms

14.1 Fundamentals

Exercise 486.

Under which circumstances is speed more important than efficiency when designing a parallel algorithm? (Give an example to highlight your answer).

Proof.
You are commander of say an antiaircraft missile battery whose command processor is a sequential computer with response time to a missile threat of 10 seconds. A new missile threat requires a response time of only 5 seconds. You are given two options. (a) Wait 2-3 years until a faster command processor comes out with a two-fold increase in performance. (b) Order a parallel machine that uses the current generation of command processors. An eight processor machine running a parallel version of the control program would achieve a speedup of 4 (i.e. efficiency 50%).
Question: What do you do? Do you wait until option (1) becomes available or take up option (2) even if the efficiency of the parallel implementations is not impressive? This is an example of a time-critical application where efficiency does not matter. What it really matters is whether a significant increase in performance can be obtained. ■

Exercise 487.

From F.T. Leighton's book [15]. Consider two algorithms for solving a problem of size N, one that runs in N steps on an N-processor machine and one that runs in $\sqrt{N}$ steps on an N^2 processor machine. Which algorithm is more efficient?

Proof.

We use the definition of efficiency $e = T_{seq}/W$, where $W = T_{par}P$, where T_{seq}, T_{par}, P is the sequential time, parallel time and processor size of the algorithm implementation for a fixed problem size N. On the first machine $e_1 = T_{seq}/(N \cdot N) = T_{seq}/N^2$. On the second machine, $e_2 = T_{seq}/(\sqrt{N} \cdot N^2) = T_{seq}/N^{5/2}$. It is immediate that $e_1/e_2 = \sqrt{N} > 1$ for $N > 1$ and the first implementation is the most efficient of the two. ■

Exercise 488.
Construct a 4-bit Gray code.

Proof.
0010, 0000, 1000, 1010, 1110, 1100, 0100, 0110, 0011, 0001, 1001, 1011, 1111, 1101, 0101, 0111. ■

Exercise 489.

You are given a non-negative integer number x, a positive integer n and 2 processors. Design a parallel algorithm that computes x^n in $\lg n + O(1)$ parallel multiplication steps. You are only allowed to use the real/integer operations $+$, - , * , and the integer operations SHR, LSB. SHR(x,k) shifts x right k positions; SHR(x) is SHR(x,1). Likewise for SHL(x,k) and SHL(x). LSB(x) returns the least significant bit of x.

For example if $x = 1111$, the after SHR(x) x becomes $x = 111$, and after SHL(x) it becomes $x = 11110$.

Proof.

The sequential $O(\lg n)$ algorithm is hard to beat asymptotically.

A. Uniprocessor case. The algorithm works as follows. First compute all powers x^{2^i}, for all $2^i \leq n$, and if x^n is none of them, find the binary representation of n. Then form a product of all x^{2^j}'s for which the j-th bit in the binary representation of n is an 1. This will get you x^n.

```
   Power( x, n)
1.  k = floor(lgn); X[0]=x;
2.  for (i=1;i<=k;i++)
3.   X[i] = X[i-1]*X[i-1] // Form all powers X[i]=x**(2^i)
4.  n= B[k]B[k-1]...B[0]  //binary representation of n
5.  if (B[0] == 1)  P=X;
6.   else           P=1;
7.  for(i=1;i<=k;i++);
//multiply all power X[i] for which B[i] is an one
8.   if (B[i]==1)  P= P* X[i];
9.  return(P); //Done
```

If you analyze carefully the algorithm above running-time is $2\lfloor \lg n \rfloor$. $P = 1$, $W = W_2 = \Theta(\lg n)$, $T = 2\lfloor \lg n \rfloor$.

The two-processor case. What if we have two processors? We label processors as zero and one. One can (almost) halve the time as follows: One of the two processors computes the x^{2^i} (or the $X[i]$ terms in our pseudocode), one per step, and a second processor decides, based on bit manipulations of n, whether to include x^{2^i} in the product that will compute x^n. So one processor does lines 2-3 and the other lines 7-8 of the sequential POWER; everything works fine because the computation of the two processors is interlaced (i.e. one step of one, is followed by one step of the other). We show this program POWER_2 in the following figure. Initally $P = 1$. Then by determining the binary representation of n one bit at a time, rightmost to leftmost, it computes the partial product P, i.e. the product of all x^{2^i}, for which the i-th rightmost bit of n is a one. Determining the i-th most bit is done using the extraction of the rightmost bit LSB(n) followed by a shift right operation. Processor 1 needs one extra step since it uses the result obtained by processor 0 in the previous step. The total running time of the parallel algorithm is $k+1$, where $k = \lfloor \lg n \rfloor$. Therefore 2-parallelism cuts the execution time by half of the serial (sequential) algorithm.

```
Step             Proc 0          | Proc 1
0:             X[0]=x            | P=1 ; n;
1      :       X[1] = X[0] * X[0] | x=X[0];
            // X[1] = x**(2**1)   | if (LSB(n)==1) P=x  //Determine B[0]
                                  |   else         P=1
                                  | n=SHR(n);
                                  | if (n==0) return(P);


2      :       X[2] = X[1] * X[1] | if (LSB(n)==1) P= P*X[1];//Determine B[1]
            // X[2] = x**(2**2)   | n=SHR(n);
                                  | if (n==0) return(P);
                                  |
                                .....
                                  |
i      :       X[i]=X[i-1]*X[i-1] | if (LSB(n)==1) P= P*X[i-1];//DetermineB[i-1]
           //  X[i]= x**(2**i)    | n=SHR(n);
                                  | if (n==0) return(P);
                                  |
                                  |
k=lgn  :       X[k]=X[k-1]*X[k-1] | if (LSB(n)==1) P= P*X[k-1];//Det. B[k-1]
           //  X[k]= x**(2**k)    | n=SHR(n)
                                  | if (n==0) return(P)
                                  |
k+1    :       No-operation       | if (LSB(n)==1) P= P*X[k]; //Det B[k]
                                    return(P);
```

Running time of POWER_2 is $T = \lfloor \lg n \rfloor + 1$. but we use two processors. $P = 2$, $W = W_2 = \Theta(\lg n)$, $T = \lfloor \lg n \rfloor + 1$.
Can we do exponentiation faster or with less work? Any other parallel algorithm will do more work, or spend more time.
(Not required, but) One can prove by a simple inductive argument, if only multiplications are used in the computation of x^n, that in the i-th step all powers up to x^{2^i} can only be computed. Thus x^n requires at least $\lg n$ steps. ■

Exercise 490.

Find the n-th Fibonacci number using n processors in $O(\lg n)$ time.

Proof.
Simple answer is that we need only one processor! If we use the matrix representation familiar from another problem We note that

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

and thus by unfolding the recurrence $n-1$ times by similarly expanding the vector of F_{n-1}, F_{n-2} we obtain that.

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

We know F_1, F_0 by the base case. Therefore computing

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

in $O(\lg n)$ time and then multiply this matrix with the vector

$$\begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

in $O(1)$ calculates F_n in $O(\lg n)$ overall time. Using parallel prefix or parallel matrix multiply is an overkill. ■

Exercise 491.

Compute F_n, the order n Fibonacci number. Input is the order n. Show how to solve this problem in time $O(\lg n)$ on an EREW PRAM with n processors. Assume that one word of memory is long enough to hold F_n. All arithmetic operations (add, subtract, multiply) take constant time. Recall that F_n is defined by the following recurrence: $F_0 = 0, F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$.

Proof.
Ignoring the discussion of the previous problem. We just need one processor if we use the 2×2 matrix trick (matrix A) and calculation of A^{n-1}.

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = A^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Using a prior problem we can design also a 2-processor algorithm requiring at most $\lg n + 1$ steps only for the A^{n-1} calculation. An of course one case parallel multiplication or a parallel prefix operation as well. ■

14.2 PRAMs

Exercise 492.

Parallel copy: broadcasting. A word is stored in memory location M[0] of the shared memory. We would like this word to be read by all n processors of a PRAM (naturally, a non CR PRAM).

Proof.
On a CRCW or CREW PRAM this problem is trivial. On an EREW PRAM broadcasting can be performed in $O(\lg n)$ steps. The structure of the algorithm is the reverse of the one for the parallel sum outlined in the next problem to follow. In $\lg n$ steps the word is broadcast as follows. In step i each processor with index j less than 2^i reads the contents of cell $M[j]$ and copies it into cell $M[j+2^i]$. After $\lg n$ steps each processor i reads the message by reading the contents of cell i. The EREW PRAM algorithm that solves the broadcasting problem has

```
begin BROADCAST (M)
//   pid() ∈ {0,...,P−1}
//   returns the id of the processor issuing the call.
1.   i = 0 ; j = pid(); M[0]=M;
2.   while (2^i < P)
3.       if (j < 2^i)
5.           M[j+2^i] = M[j];
6.       i = i+1;
6.   end
7.   Processor j reads M from M[j].
end BROADCAST
```

performance $P = n/2 = O(n)$, $T = O(\lg n)$, and $W = O(n \lg n)$, $W_2 = O(n)$. Broadcasting on a hypercube and a butterfly. How does it work? (Hint: Base your solution on the BROADCAST algorithm). ■

Exercise 493.

Do Parallel sum in $O(\lg n)$ time for $x_0, \ldots, x_{n-1}$, that is evaluate $x_0 + \ldots + x_{n-1}$.

Proof.

Evaluate $x_0 + \ldots + x_{n-1}$. For a PRAM implementation, value x_i is initially stored in shared memory cell $M[i]$. The sum $x_0 + x_1 + \ldots + x_{n-1}$ is to be computed in $T = \lg n$ parallel steps.The algorithm consists of $\lg n$ steps. In step i, processor $j < n/2^i$ reads shared-memory cells $M[2j]$ and $M[2j+1]$ combines (sums) these values and stores the result into memory cell $M[j]$. After $\lg n$ steps the sum resides in cell 0. Algorithm PARALLEL_SUM has $T = O(\lg n)$, $P = n/2$ and $W = P \times T = O(n \lg n)$, and $W_2 = O(n)$ (actual work done).

```
//   pid() ∈ {0,...,P−1}
//   returns the id of the processor issuing the call.
begin PARALLEL_SUM (n)
1.   i = 1 ; j = pid();
2.   while (j < n/2^i)
3.       a = M[2j];
4.       b = M[2j+1];
5.       M[j] = a+b;
6.       i = i+1;
7.   end
end PARALLEL_SUM
```

M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]	M[7]	
x0	x1	x2	x3	x4	x5	x6	x7	t=0
x0+x1	x2+x3	x4+x5	x6+x7					t=1
x0+...+x3	x4+...+x7							t=2
x0+...+x7								t=3

Algorithm PARALLEL_SUM can be easily extended to include the case where n is not a power of two.

PARALLEL_SUM is the first instance of a sequential problem that has a trivial sequential but more complex parallel solution. Instead of operator *Sum* other operators like *Multiply*, *Maximum*, *Minimum*, or in general, any associative operator could have been used. As associative operator $\otimes$ is one such that $(a \otimes b) \otimes c = a \otimes (b \otimes c)$. Can you improve PARALLEL_SUM so that T remains the same, $P = O(n/\lg n)$, and $W = O(n)$? Explain. What if we have p processors where $p < n$? See a follow-up problem. (You may assume that n is a multiple of p.) Generalize the Parallel Sum algorithm to any associative operator. ■

Exercise 494.

(a) Let $X_1 \ldots, X_n$ be binary/boolean values. Find $X = X_1 \wedge X_2 \wedge \ldots \wedge X_n$ (i.e. logical AND also known as conjunctins) using an EREW PRAM.
(b) Do so using a CRCW PRAM.

Proof.
(a) An EREW PRAM algorithm solution for this problem works the same way as the PARALLEL SUM algorithm and its performance is $P = O(n), T = O(\lg n), W = O(n \lg n)$ along with the improvements in P and W mentioned for the PARALLEL SUM algorithm.
(b) In the remainder we will investigate a CRCW PRAM algorithm. Let binary value X_i reside in the shared memory location C_i. We can find $X = X_1 \wedge X_2 \wedge \ldots \wedge X_n$ in constant time on a CRCW PRAM. Processor 1 first writes an 1 in shared memory cell C_0. If $X_i = 0$, processor i writes a 0 in memory cell C_0. The result X is then stored in this memory cell. The result stored in cell 0 is 1 (TRUE) unless a processor writes a 0 in cell 0; then

```
begin LOGICAL_AND (X_1 ... X_n)
1.  Proc 1 writes an 1 in cell C_0.
2.  if X_i = 0 processor i writes 0 into cell C_0.
end LOGICAL_AND
```

one of the X_i is 0 (FALSE) and the result X should be FALSE, as it is. Practice: Give an $O(1)$ CRCW algorithm for logical OR (disjunction or inclusive disjunction) before you move to the next problem. ■

Exercise 495.
You are given n binary values $X_1, \ldots, X_n$. Find the logical OR of these n values in constant time on a CRCW PRAM with n processors.

Proof.
Let value X_i reside in cell C_i of the shared memory of the PRAM. Let C_0 be the cell where the logical OR of these n values will be stored. In the first step processor 1 writes (exclusively) a 0 in cell C_0. In the second step processor i reads cell C_i. If the value read by processor i is an 1, processor i writes an 1 into cell C_0. At the following time step the result in C_0 becomes available to every processor and it is the logical OR of the n values. If no processor writes into C_0, then all X_i are 0 and the value written in step 1 by processor 1, i.e. a 0, is the result. Otherwise, processors that had read an 1 would write into C_0 an 1. No matter what the arbitration rule for concurrent write operations is, one processor will succeed in overwriting the 0 in cell C_0 by an 1, the final result. ■

Exercise 496.

A sequence S of n keys $x_1, \ldots, x_n$ is given. For a given input key y, the *rank* of y with respect to S is the number of keys in S whose key-values are less than or equal to the key value of y. Given $y, x_1, \ldots, x_n$ describe an $O(\lg n)$ time algorithm that ranks y in S using n processors of a EREW PRAM.

Proof.

Let cell C_i of the shared memory store x_i and let cell C_0 store y. Processor 1 reads value y and broadcasts it to the remaining processors in $O(\lg n)$ steps. Subsequently, processor i reads x_i and compares it to y. If x_i is less than or equal to y an one is written in register r_i (initialized to 0) of processor i. In the following step, a parallel sum of the r_i values is initiated. The sum indicates the rank of y in the sequence of x_i's. The running time of the algorithm is $O(\lg n)$. ■

Exercise 497.
You are given n comparable keys (i.e. a total order is defined on them). Find the rank of each key time efficiently on a EREW PRAM with $p = n^2$ processors, and then sort the keys.

Proof.
On an EREW PRAM the same bound (time and processor) can be derived provided that Steps 1 and 2 (the concurrent steps) become broadcasting steps and thus require $O(\lg n)$ time rather than $O(1)$ time. In step 1, x_i is broadcast to $(i, *)$ processors and in step 2, x_j to $(*, j)$ processors. Given the $\lg n$ time of Step 4, the asymptotics are not affected by this modification. The same time/processor bounds are derived. In all cases we can reduce processor size by a logarithmic factor by using the standard pipelining method. ■

Exercise 498.

You are given n comparable keys (i.e. a total order is defined on them). Find the rank of each key time efficiently on a CREW PRAM with $p = n^2$ processors.

A sequence S of n keys $x_1, \ldots, x_n$ is given. For a given input key y, the *rank* of y with respect to S is the number of keys in S whose key-values are less than or equal to the key value of y. Assuming n distinct keys, the rank of x_i is the number of keys less that x_i. Thus the maximum has rank $n-1$ and the minimum rank 0.

Then sort the n keys.

Proof.

```
    Rank(x[1..n],n) //Use n**2 processors , indexed P(i,j)
1.   Processor P(i,j), i != j, reads x[i]  into register a
2.   Processor P(i,j), i != j, reads x[j]  into register b
3.   If (a=x[i]) > (b=x[j])    M[i][j]=1;
        else                   M[i][j]=0;
     // Do one of 4  or  4a and 4b.
4.   SegmentedPrefix
     (+,M[1][1]...M[1][n]|M[2][1]..M[2][n]|...|M[n][1]...M[n][n],n);
or
4a.  In parallel over P(i,n)
4b.    ppf(+, M[i][1]...M[i][n]);
5.   Processor P(i,1) writes M[i][n] into Rank[i].
6.   return(Rank[i]) // for all i=1 , ... , n.
```

Let the input be $x_1, \ldots, x_n$. Let $P = n^2$. Let the n processors be labeled $P(i,j)$, $i = 1, \ldots, n$ and $j = 1, \ldots, n$. Processor (i,j), $i \neq j$ is going to read in step 1 value x_i, in step 2 value x_j, and in step 3 it is going to compare the two values as in $x_i > x_j$ and if this comparison is true, in step 3, it will store 1 in memory cell $M(i,j)$, otherwise it will store a 0. A parallel sum of the $M(i,*)$ values among the $(i,*)$ processors is computed for each $i = 1, \ldots, n$ in step 4 with the end result (sum) stored in $M(i,n)$. Because we have n parallel and independent sums, we can also formulate them as one segmented scan with n segments being the n independent parallel sums. One way is line 4, the other possibility are lines (4a) and (4b). Value $M(i,n)$ at the end of the scan is the rank of x_i since it counts the ones in $M(i,*)$ i.e. the number of keys less than x_i (the rank of the maximum is $n-1$).
It is obvious that the algorithm has the following properties: $P = n^2$, $T = O(\lg n)$, $W = W_2 = O(n^2)$.
Sorting. One extra line is need for sorting: the generation of the output. We rewrite instead the pseudocode to include the output array y.

```
    Rank(x[1..n],y[1..n],n) //Use n**2 processors , indexed P(i,j)
1.   Processor P(i,j), i != j, reads x[i]  into register a
2.   Processor P(i,j), i != j, reads x[j]  into register b
3.   If (a=x[i]) > (b=x[j])    M[i][j]=1;
        else                   M[i][j]=0;
4a.  In parallel over P(i,n)
4b.    ppf(+, M[i][1]...M[i][n]);
5.   Processor P(i,1) writes M[i][n] into Rank[i].
      // Rank[.] takes values         0..n-1
      // Arrays here use index values 1..n
6.    y[Rank[i]+1]= x[i];
7.    return(y);
      // One can sort "in place" if
      // 6. M[i][1]      = x[i];
      // 7. x[Rank[i]+1] = M[i][1];
      // 8. return(x);
```

Performance remains the same as before. ■

Exercise 499.

Let $S = \langle x_1, x_2, \ldots, x_n \rangle$ be a sequence of n distinct keys. The rank of x_1 in the sequence S or $r(x_1, S)$ is the number of keys less than x_1 in S. The problem of sorting is equivalently the problem of determining the rank of each one of the n input keys.
Determine the rank of all keys in S in $O(\lg n)$ time with a CRCW PRAM. How many processors did you use?
Sort the n keys in the same time with a CRCW PRAM. How many processors did you use?
Can you repeat the two questions above for an EREW PRAM? How would the answers change? Explain.
Note that Step 1-5 in the algorithm(s) below are required to determine the ranks of all keys. Step 6 concludes the sorting operation after the ranking is performed.

Proof.
Let $P = n^2$. Processor (i, j), $i \neq j$ is going to read in step 1, x_i, in step 2, x_j, and in step 3, it will compare the two values as in $x_i > x_j$ and if the result of this comparison is true, in step 4, it will store 1 in memory cell $M(i, j)$, otherwise it will store 0. A parallel sum of the $M(i, *)$ values among the $(i, *)$ processors is computed for each $i = 1, \ldots, n$ in step 5 with the end result (sum) stored in $M(i)$. $*$ means this position takes all $i = 1, \ldots, n$ values. This sum stored in $M(i)$ is the rank of x_i. Note that step 5 may take more than one time units.
In step 6, Processor i reads $M(i)$, then reads x_i and writes x_i in the $M(i)$-th position $O[M(i)]$ of the output sequence $O[..]$.
Note that in step 1, all processors $(i, *)$ read x_i (a concurrent read step), and in step 2 all processors $(*, j)$ read x_j. Let us examine the computation of processors $(i, *)$. They compare x_i to every other key. The number of 1's they compute in step 4 is the rank of x_i in the output sequence that is computed as a parallel sum in step 5. If this is r then the index of the output sequence that will store x_i will be r (we assume that the minimum of rank 0 will be stored in position 0 and the maximum of rank $n-1$ in position $n-1$).
Steps 1,2,3,4 each take $O(1)$ step, and step 5 takes $O(\lg n)$ steps. Step 6 takes $O(1)$ time. Total is $O(\lg n)$.
Remark. We were generous in allocating memory and using various names for it. $M(i, j)$ cell can be mapped to real address $j * n + i$ for example uniquely. $M(i)$ can be mapped to $n^2 + i$ and so on. Thus these naming conventions used for convenience can be made unique if necessary.
So far, ranking required $T = O(\lg n)$, $P = n^2$ (in fact $n(n-1)$) and so did sorting on a CRCW PRAM (sorting in addition involved step 6).
On an EREW PRAM the same bound (time and processor) can be derived provided that
Steps 1 and 2 (the concurrent steps) become broadcasting steps and thus require $O(\lg n)$ time rather than $O(1)$ time. In step 1, x_i is broadcast to $(i, *)$ processors and in step 2, x_j to $(*, j)$ processors. Given the $\lg n$ time of Step 6, the asymptotics are not affected by this modification. The same time/processor bounds are derived.
Can we sort in $O(\lg n)$ time using fewer processors on a CRCW or EREW PRAM? Yes we could sort at the stated time bound in $O(\lg n)$ using a parallelization the merge sort algorithm. The resulting algorithm however is more involved. Details are available in R. Cole, Parallel Merge Sort, SIAM J. Computing, 17(4) (1988), 770–785. ■

Exercise 500.
Show how to find the MAX of n comparable keys in $O(1)$ using n^{1+t} processors for some constant $t < 1$.

Proof.
Let $t = 1/k$, for some integer k. If there is no such k choose an integer k (the smallest possible) such that $1/k < t$. Let $m = n^{1/k}$. We are going to find the MAX of $m^k = n$ values by grouping. We first group the m^k values into m^{k-1} groups of m values. This reduces the original problem to that of MAX finding m^{k-1} values. We then group m^{k-1} values in m^{k-2} groups of m values each. After k iteration we want to find the MAX of m values. Note that each group has m values. We know (class) how to MAX m values in constant time with m^2 processors. That's what we do. The number of parallel rounds to find MAX will collectively be k. The most processor consuming step is the base step. We have $n = m^k$ values arranged in m^{k-1} groups of m values and we assign m^2 processors to each such group. If this is possible we obtain m^{k-1} partial results in $O(1)$ time. A total of $m^{k-1} \cdot m^2 = m^{k+1}$ processors are utilized; since $m^{k+1} = m^k \cdot m = n \cdot n^{1/k} = n^{1+1/k}$ and $1/k$ was chosen so that $1/k \leq t$, we have $n^{1+1/k} < n^{1+t}$ and thus the processor pool is sufficient for this step. At any other intermediate step fewer processors are required as there are fewer groups of m values. Thus $O(k) = O(1/t)$ steps are needed to compute the MAX. Since k (or t) is a constant, this is $O(1)$, as required. ■

Exercise 501.

Can you find the MAX of n keys with $n^{7/6}$ processors in $\Theta(1)$ time? Explain.

Proof.
That's a previous problem rephrased sparsely with $t = 1/6$! So will be the solution. We use the idea of doubly-logarithmic depth tree. If we have n^{2t} processors, we can find the maximum of n^t elements in $O(1)$ number of steps.

Lemma 1. If we have $n^{2/6}$ processors we can find the maximum of $n^{1/6}$ keys in constant time.

Split the keys into groups of $n^{1/6}$ keys. Find the maximum of each group. We have $n^{5/6}$ partial results. Group again into groups of $n^{1/6}$ keys. Theare are only $n^{4/6}$ of them. Find the maximum of each group, and repeat until we are left with one group.

`Step 1.` Split the n keys into $n^{5/6}$ groups each group consisting of $n^{1/6}$ keys. Assign to each group $n^{2/6}$ processors. Use Lemma 1 to find the maximum of each group in $O(1)$ time. Since $n^{5/6}$ groups use $n^{2/6}$ processors, overal this step utilizes $n^{7/6}$ processors. We have enough for this step. We need now to determine the maximum of $n^{5/6}$ values to determine the maximum of the original n values.

`Step 2.` The number of maxima left, $n^{5/6}$ are split into $n^{4/6}$ groups of $n^{1/6}$ keys each. Assign to each group $n^{2/6}$ processors. Use Lemma 1 to find the maximum of each group in $O(1)$ time. Since $n^{4/6}$ groups use $n^{2/6}$ processors, overall this step utilizes $n < n^{7/6}$ processors. We use only a fraction of the available processors.

`Step 3,4,5` work similarly and the number of maxima that will determine the maximum of the n original keys is further reduced from $n^{4/6}$ to $n^{3/6}$ to $n^{2/6}$ down to $n^{1/6}$.

`Step 6` The maximum of $n^{1/6}$ keys needs to be computed. Use Lemma 1 to perform this step in $O(1)$ time. Overall, 6 steps of $O(1)$ time are used for a total of $O(6) = \Theta(1)$ time.

■

Exercise 502.

Give an efficient algorithm for $ParallelSum(X[0..n-1])$ if we have p processors, where $p < n$? (You may assume that n is a multiple of p, and n is sufficiently larger than p, say $n/\lg^2 n > p$.)

Proof.

Let n be a multiple of p. Let $X[0..n-1]$ be the array whose sum we are interested in computing. Split it into p even pieces each one of size n/p. Each processor works on its own piece `X[pid()*n/p+(pid()+1)n/p-1]` (i.e. a consecutive block of the shared memory) and computes a partial sum in time $n/p-1$. The p partial results are then summed up PRAM-style using a binary tree. Total time is $n/p-1+\lg p$ assuming p is a power of two or $n/p+O(\lg p)$, otherwise. Given than $n/\lg^2 n > p$, this is $n/p+o(n/p)$.

■

Exercise 503.

Show how to merge two sorted sequences in the CREW or CRCW PRAM. Optimize the number of processors used.

Proof.
The algorithm below works for the CRCW PRAM and also for the CREW PRAM. (CRCW is to mean one of the reasonable choices of a CRCW PRAM.) We don't need concurrent write operations at all. We have n keys and let $m = n/2$. Assume that n is even otherwise pad the array with one extra key to do so. Let the two sorted arrays be $A[1..m]$ and $B[1..m]$. Copy them into $X[1..n]$ so that $X[1..m]$ is A and $X[m+1..n]$ is B. This can be done in $O(1)$ time with $O(n)$ processors or $O(\lg n)$ time with $O(n/\lg n)$ processors. Then perform the following algorithm.

```
   Merge(X[1..m m+1...n],Y[1..n],m,n)
    //Merge X[1..m] and X[m+1..n] on a CREW PRAM
   //Result in Y[1..n]
1. Processor i, where 1 <= i <= m
2.  r[i]  = BinarySearch(X[i],  X[m+1..n]);
    //r[i] : X[r[i]] < X[i] <= X[r[i]+1]
4. Y[(r[i]-m)+i] = X[i];
5. Processor j, where 1 <= j <= m
6.  r[j+m]= BinarySearch(X[j+m],X[1..m]); //
7. Y[r[j+m]+j]=X[j+m]
```

Consider key $a = A[i]$. a is stored in position $X[i]$ of the array. The rank of a in A is known, it is $i-1$ since A is already sorted. Thus $rank(a:A) = rank(a:X[1..m]) = i-1$. The only thing left is to find the rank of a in B i.e. the rank of a in $X[m+1..n]$. This is done with the binary search operation of lines 1-2. This binary search returns an index between m and n however. To convert it to rank we need to subtract m, i.e. the rank $rank(a:B) = r[i]-m$. Therefore the final destination position of element a, i.e. its rank in the sorted sequence Y plus one will be $i+(r[i]-m)$. This position is filled with $X[i]$ in line 4 of MERGE. Similarly, we find the rank of an element $b = B[j]$ of B which is stored in position $X[m+j]$ of the array X. Rank $rank(b:X[m..n]) = j-1$. The Binary search operation of line 6, is combined with the rank of b in B and line 7 inserts $X[j+m]$ in the right position of the output array. The only concurrent operation is CR operations during BinarySearch. Each processor works for $O(\lg n)$ time. However we must use n processors as rescheduling of the work is not possible (due to binary search). Thus $P = n$, $T = O(\lg n)$, $W = W_2 = O(n \lg n)$.
Can we reduce the work load to linear? We could still use a CREW algorithm and have the same time performance but for a reduced W and W_2 that are both linear, thus matching the sequential work/time performance.

Below, we outline the algorithm. Pick every $\lg n$-th element of A and do the same for B. Call the corresponding elements a_i and b_i respectively, i.e. $a_i = A[i * \lg n]$, $b_i = B[i * \lg n]$. In either case $i \leq n/\lg n$. We call these elements the samples. One processor will be responsible for a_i and the keys between a_{i-1} (exclusive) and a_i (inclusive). One processor will be responsible for b_i and the keys between b_{i-1} (exclusive) and b_i (inclusive). What we have effectively done is reduce the processor load by a factor of $\lg n$. Each processor will be responsible not only for a_i, b_i but also for ranking all keys from a_{i-1} through a_i and b_{i-1} through b_i. There are $O(\lg n)$ keys altogether assigned to each processor. Other than that, our algorithm resembles the already mentioned CREW algorithm. a_i is already ranked within A. It can be ranked within B similarly to our previous algorithm. b_i is ranked in A using the same method. If rank(a_i:B) is found, then ra_i = rank(a_i: $b_1, b_2, b_3 \ldots$) can be found in constant time (note that the latter sequence is not B, just the samples of B). This is ra_i = rank(a_i:B) $/\lg n$; call rb_i the corresponding rank for b_i elements. [Note: From this brief description we ignore issues related to the endpoints, and also ± 1 indexing issues]. Now that we have ranked the sample keys a_i and b_i we can rank the remaining keys. The keys assigned to processor i are all the keys between a_{i-1} (exclusive) and a_i (inclusive). In order to rank these keys we need only look to a fraction of B, not all B. What is this fraction? It is $B[ra_{i-1}] \ldots B[ra_i]$. Note that for distinct a_i these ranges are non-overlapping. Is the description over? No, the b range $B[ra_{i-1}] \ldots B[ra_i]$ might include more than $\lg n$ keys. So how do we deal with this? Depending on how big the b-range is we assign a number of processors to it proportional to its size. We then rank the B samples of this set into the $a_{i-1} \ldots a_i$ range. Can we reduce the CREW requirement into an EREW requirement?
Merging on an EREW PRAM: non-optimal time and work. An easy method is to emulate the CR operation

by a broadcasting or segemented broadcasting (i.e. use segmented prefix where the associative operator is copy == broadcast). Then every CR requires $O(\lg n)$ steps and the running time becomes $O(\lg^2 n)$.

Merging on an EREW PRAM: $O(\lg n)$ but work is not optimal. One such method is the bitonic merger network. This would merge the n keys of A and the n keys of B in about $\lg n + 1$ steps (as long as n is a power of two) using $n \lg n$ comparisons (i.e. work). The merging algorithm is **oblivious** i.e. the same comparisons are performed (e.g. key in position i compared to key in position j in the first step and so on) no matter what the values of the keys are. Naturally the work of this method is non-optimal.

Merging on an EREW PRAM: $O(\lg n)$ but work is optimal. A work optimal solution is a refinement (complicated however) of the CREW work optimal algorithm and an Information Processing Letters paper of 1989, Vol 33(4),pp 181-185 by Hagerup and Ruub. It involves merging the sorted samples by another non-optimal sorting method (eg. Batcher's bitonic merger); its workings however are based on the outlined CREW work-optimal algorithm. ■

Exercise 504.

Give an EREW PRAM algorithm that merges two sorted arrays of size $n/2$ with $P = n/2$ in $O(\lg n)$ time. You may assume that n is a power of two, and you may of course reuse prior results.

Proof.

As noted in a prior problem, use the bitonic merging network as used in bitonic-based merge-sort and perform its operations on the PRAM. (The next section, alas, discusses sorting and merging networks!) $P = n/2$ processors can perform the operations of $n/2$ comparators in parallel in one parallel step. Given that merging (not sorting) requires only a $\lg n$ comparison network, the result is immediate.

■

Exercise 505.

Given a set of n values $x_0, x_1, \ldots, x_{n-1}$ and an associative operator, say $+$, the *parallel prefix* problem (ppf in short) is to compute the following n results/"sums". The parallel prefix operation is also known as *prefix sums* or as a *scan* operation.

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 + x_1 \\ y_2 &= x_0 + x_1 + x_2 \\ &\ldots \\ y_i &= x_0 + \ldots + x_i = \sum_{j=0}^{i} x_i \\ &\ldots \\ y_{n-1} &= x_0 + \ldots + x_{n-1} = \sum_{j=0}^{n-1} x_i. \end{aligned}$$

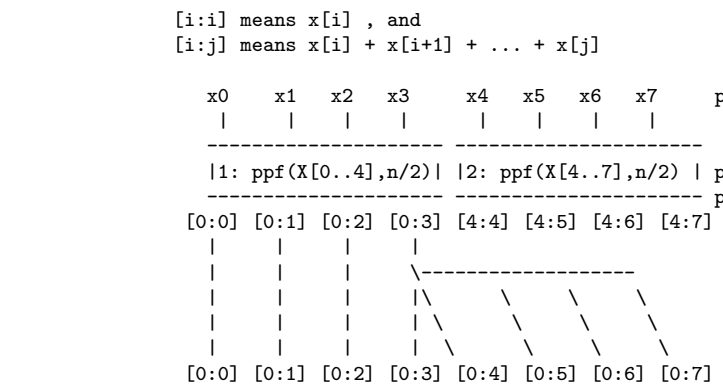
The operation has many uses in parallel computing such as in load-balancing the work assigned to processors, and compacting data structures such as arrays.

Show that computing the prefix sums is no more (asymptotically) difficult than computing the single sum $x_0 + \ldots + x_{n-1}$.

Proof.

An algorithm for parallel prefix on an EREW PRAM would require $\lg n$ phases. In phase i, processor j reads the contents of cells j and $j - 2^i$ (if it exists) combines them and stores the result in cell j. The EREW PRAM algorithm that solves the parallel prefix problem has performance $P = O(n)$, $T = O(\lg n)$, and $W = O(n \lg n)$, $W_2 = O(n)$. We present a variety of approaches.

Approach 1: Divide and conquer. Let $ppf(X[0..n-1], n)$ calculate the parallel prefix of the elements of array $X[0..n-1]$. Let $ppf(X[0..n/2-1], n/2)$, and $ppf(X[0..n/2-1], n/2)$ utilizing divide and conquer computes the ppf of half of X respectively.



It is quite straightforward that $T(n) = T(n/2) + 1$ thus allowing $T = O(\lg n)$. Moreover $P = n$, and $W(n) = P(n)T(n) = O(n \lg n)$. Furthermore, $W_2(n) = 2W_2(n/2) + n/2$ implying $W_2(n) = O(n \lg n)$.

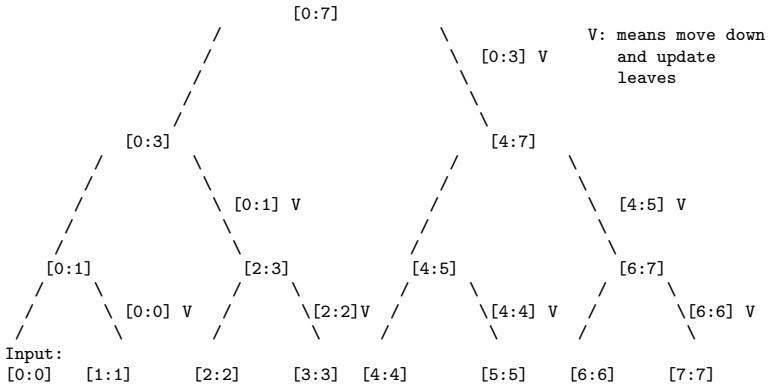
Approach 2: Better

```
// Input :  M[j]= x[j]=[j:j]
// Output:  M[j]= x[0]+...+X[j] = [0:j] for j= 0, ... , n-1
    ppf2(X,n) //
1.i=0;
2.while (2**i < n ) {
3. j=pid();
4. if (j-2**i > 0 ) {   // j- 2**i +1  >=0
5.  a=M[j];             // M[j]      holds [j-2**i+1 : j]
6.  b=M[j-2**i];        // M[j-2**i] holds [j-2**i-2**i +1 : j-2**i]
7.  M[j]=a+b;           // associative operator +
8. }                    // M[j] holds [j-2**i-2**i +1 : j-2**i]+
9. i+= 1;               //            [j-2**i+1 : j]
  }                     //          = [j- 2**(i+1)+1  : j]
```

```
   x0      x1        x2        x3         x4        x5      x6       x7
0. [0:0]  [1:1]     [2:2]     [3:3]      [4:4]     [5:5]   [6:6]    [7:7]
1.  |     [0:1]     [1:2]     [2:3]      [3:4]     [4:5]   [5:6]    [6:7]
2.  |       |       [0:2]     [0:3]      [1:4]     [2:5]   [3:6]    [4:7]
3.  |       |         |                  [0:4]     [0:5]   [0:6]    [0:7]
```

Using induction, At step 5, memory location j is read, and in step 6, memory location $j-2^{i-1}$ is read (as long as it exists). The two values are combined. By inductions the former value $M[j]$ is $[j-2**i+1:j]$ i.e. $x_{j-2^i+1}+\ldots+x_j$ and the other value $M[j-2^i]$ is $[j-2^i-2^i+1:j-2^i]$ i.e. $x_{j-2^i-2^i+1}+\ldots+x_{j-2^i}$ and after step 7 is realized $M[j]$ would hold value $[j-2^{i+1}+1:j]$ and the inductive step is complete. The termination condition is $j \leq 2^i$ i.e. $i \geq \lg(j)$. Given that $j \leq n-1$ we have that for $i \geq \lg(n-1)$ the if loop is ignored. The while loop is executed while $i < \lg n$, i.e. $\lceil \lg n \rceil$ times.

Approach 3: The tree approach.



The parallel prefix algorithm of the previous page (tree-based) requires about $2\lg n+1$ parallel steps, $P=n$ processors and work $W=\Theta(n\lg n)$, and $W_2=\Theta(n)$. One could describe that version due to Ladner and Fischer as follows. The algorithm start from the bottom (leaves). The leaves are assigned a processor and hold one input value each. Two sibling nodes in the tree combine their values: the sum (if the associative operator is plus) is sent to their parent and the left sibling value is sent through the parent to the subtree of the right sibling. In the example above $[4:4]$ and $[5:5]$ are combined $[4:5]$ which is sent to their parent and $[4:4]$ is sent to the right subtree of their parent: the only node is the one initially holding $[5:5]$, that combines next $[4:4]$ and $[5:5]$ into

a $[4:5]$. Thus in first iteration there is one movement up (to parents of leaves) and one movement down (from parent to right child, a leaf). Next time this is repeated with one movement up, but when the left child value of the new parent is sent to the right subtree this would move two levels down to reach a leaf note. When are are close to the root there is one movement up but the left sibling (or a.k.a left child of the root) value will move down the height of the tree number of levels towards the leaves. The left sibling value is thus transmitted 2^i times through its sibling's right subtree where i is the height of the left or right sibling (a leaf has height 0). Note that when a node receives a value from above (i.e. through its parent from its left sibling) it transmits it to both of its children and so on. By rescheduling the computation and using $P = n/\lg n$ processors, while maintaining $T = O(\lg n)$, the work can be reduced to linear $W = W_2 = \Theta(n)$. ■

```
begin PPF3  (X[0..n-1],Y[0..n-1],p = 0..n-1)
1.   Y [0] = X [0];
2.   if  n > 1      then
3.       ∀ i = 0,...,n-1      dopar
4.           A[i] = X[2i] + X[2i+1];
5.       enddo
6.       B= ppf3 (A[0..n/2-1],B[0..n/2-1],p = 0..n/2-1);
7.       ∀ i = 0,...,n/2-1    dopar
8.           Y [2i+1]=B[i];
9.       enddo
10.      ∀ i = 1,...,n/2-1    dopar
11.          Y [2i]=B[i-1]+A[2i];
12.      enddo
13.  endif
end PPF3
```

Exercise 506.

(a) Give an algorithm for multiplying two $n \times n$ matrices on a CREW PRAM with time complexity $T = O(\lg n)$ and $P = n^3$.
(b) Give an algorithm for multiplying two $n \times n$ matrices on a EREW PRAM with time complexity $T = O(\lg n)$ and $P = n^3$.

Proof.
For convenience, processors are indexed as triples (i, j, k), where $i, j, k = 1, \ldots, n$. In the first step processor (i, j, k) concurrently reads a_{ij} and b_{jk} and performs the multiplication $a_{ij}b_{jk}$. In the following steps, for all i, k the results $(i, *, k)$ are combined, using the parallel sum algorithm to form $c_{ik} = \sum_j a_{ij}b_{jk}$. After $\lg n$ steps, the result c_{ik} is thus computed. The same algorithm also works on the EREW PRAM with the same time and processor complexity. The first step of the CREW algorithm need to be changed only. We avoid concurrency by broadcasting element a_{ij} to processors $(i, j, *)$ using the broadcasting algorithm of the EREW PRAM in $O(\lg n)$ steps. Similarly, b_{jk} is broadcast to processors $(*, j, k)$.

```
                                          CREW    EREW
1. aij to all            (i,j,*) procs    O(1)    O(lgn)
   bjk to all            (*,j,k) procs    O(1)    O(lgn)
2. aij*bjk at            (i,j,k) proc     O(1)    O(1)
3. parallel sum aij *bjk (i,*,k) procs    O(lgn)  O(lgn)  n procs participate
             j

4. cik = sum aij*bjk                      O(1)    O(1)
           j
```

In conclusion $T(n) = O(\lg n)$, $P = O(n^3)$, $W = TP = O(n^3 \lg n)$, and $W_2 = O(n^3)$.

■

Exercise 507.

Let $X_1 \ldots, X_n$ be n keys. Find $X = \max\{X_1, X_2, \ldots, X_n\}$.

(a) On an EREW PRAM.

(b) On a CRCW PRAM with n^2 procs.

(c) On a CRCW PRAM with n procs.

Proof.

(a) Do as in the PARALLEL SUM algorithm and its performance is $P = O(n), T = O(\lg n), W = O(n \lg n), W_2 = O(n)$ along with the improvements in P and W mentioned for the PARALLEL SUM algorithm.

(b) In the remainder we will investigate a CRCW PRAM algorithm. Let binary value X_i reside in the local memory of processor i.

(b.1) Algorithm $\mathrm{MAX1}$**.**

The CRCW PRAM algorithm $\mathrm{MAX1}$ to be presented has performance $T = O(1)$, $P = O(N^2)$, and work $W_2 = W = O(N^2)$. The second algorithm to be presented in the following pages utilizes what is called a doubly-logarithmic depth tree and achieves $T = O(\lg \lg N)$, $P = O(N)$ and $W = W_2 = O(N \lg \lg N)$.

The third algorithm is a combination of the EREW PRAM algorithm and the CRCW doubly-logarithmic depth tree-based algorithm and requires $T = O(\lg \lg N)$, $P = O(N)$ and $W_2 = O(N)$.

begin MAX1 $(X_1 \ldots X_N)$
1. **in proc** (i, j) **if** $X_i \geq X_j$ **then** $x_{ij} = 1$;
2. **else** $x_{ij} = 0$;
3. $Y_i = x_{i1} \wedge \ldots \wedge x_{in}$;
4. Processor i reads Y_i ;
5. **if** $Y_i = 1$ processor i writes i into cell 0.

end MAX1

(b.2) Algorithm $\mathrm{MAX2}$**.**

The CRCW PRAM algorithm $\mathrm{MAX1}$ has performance $T = O(1)$, $P = O(N^2)$, and work $W_2 = W = O(N^2)$.

Let $N = 2^{2^n}$, for some integer n. A *doubly logarithmic-depth* tree with N leaves is defined as follows.

begin MAX2 $(X_1 \ldots X_N)$
0. The i-th cell contains X_i; $N = 2^{2^n}$.
1. **for** $(i = n-1; i \geq 0; i--)$ **do**
2. **begin**
3. Assign to each node u of level i, $(2^{2^{n-i-1}})^2$ processor s (i.e. the square of its children in level $i+1$.) ;
4. Use algorithm $\mathrm{MAX1}$ and these processors to find the maximum
of the values stored at the children of u and store the result at u;
5. Node u ceases to hold an UNDEFINED.
6. **end**

end MAX2

(1) The root of the tree at level 0 has $2^{2^{n-1}} = N^{1/2}$ children in level 1.

(2) Each node at level 1 has $2^{2^{n-2}} = N^{1/2^2}$ children in level 2.

(3) Each node of level i has $2^{2^{n-(i+1)}} = N^{1/2^{i+1}}$ children in level $i+1$.

(4) Each node of level $n-1$ (the level before the last) has $2^{2^{n-n}} = N^{1/2^n} = 2$ children in level $n = \lg \lg N$.

(5) The nodes of level n are the **leaves** of the tree.

Some properties of a *doubly logarithmic-depth* tree are listed below.

(1) The **height** of the tree is $n = \lg\lg N$.

(2) A node of level i has $2^{2^{n-(i+1)}}$ children in level $i+1$.

(3) The TOTAL number of level i nodes is $2^{2^{n-1}} 2^{2^{n-2}} \ldots 2^{2^{n-i}} = 2^{2^n - 2^{n-i}}$.

(4) The Product $(2^{2^{n-i-1}})^2 \times 2^{2^n - 2^{n-i}}$ is $O(2^{2^n}) = O(N)$.

Algorithm $\mathrm{MAX2}$ below achieves better work performance than $\mathrm{MAX1}$: a $T = O(\lg\lg N)$, $P = O(N)$, and $W = W_2 = O(N \lg\lg N)$. Algorithm $\mathrm{MAX2}$ works as follows.

(1) Initially, items X_i are on the N leaves of the tree.

(2) The root will hold the result at the completion of the algorithm.

(3) Processors are assigned to the nodes of the tree in some predetermined fashion.

(4) All nodes of the tree other than the leaves hold an UNDEFINED value in the beginning of the execution.

(5) If a node u at level i holds an UNDEFINED value and its M children hold some intermediate results M^2 processors are assigned to u to find the maximum of M numbers (the partial results of the children of u) using $\mathrm{MAX1}$ in constant time. Node u then holds the computed maximum (and ceases to hold an UNDEFINED value).

(b.3) Algorithm $\mathrm{MAX3}$

Algorithm $\mathrm{MAX2}$ has $W = W_2 = O(N \lg\lg N)$. Algorithm $\mathrm{MAX3}$ below has $W_2 = O(N)$. It uses $\mathrm{MAX2}$ and the EREW PRAM algorithm as subroutines. The EREW algorithm finds the SUM or the MAXIMUM of N numbers by working from the leaves to the root of a binary tree, ie $\lg N$ levels. If we stop the computation after i levels, we have $N/2^i$ partial results, each result being the MAXIMUM of 2^i numbers initially stored in the leavs of the subtree rooted at the partial result. **Max3** first runs the EREW PRAM algorithm for $i = \lg\lg\lg N$ levels so that a total of $N/\lg\lg N$ partial MAXIMA are computed. Then it applies $\mathrm{MAX2}$ where N in $\mathrm{MAX2}$ is equal to the number of partial results ie $N/\lg\lg N$. Step 2 of algorithm $\mathrm{MAX3}$ requires $T = O(\lg\lg\lg N)$, $P = O(N)$ and

begin $\mathrm{MAX3}$ $(X_1 \ldots X_N)$
0. The i-th cell contains X_i;
1 **begin**
2. Use the EREW PRAM algorithm on a complete binary tree on N leaves to reduce the original problem to computing the maximum of $N/\lg\lg N$ values (ie proceed from the leaves up to the nodes of level $\lg N - \lg\lg\lg N$);
3. Use $\mathrm{MAX2}$ to find the maximum of the $N/\lg\lg N$ values of Step2;
4. **end**
end $\mathrm{MAX3}$

$O(N)$ work (number of comparisons is at most the number of edges of the tree). Step 3 requires $P = O(N)$, $T = O(\lg\lg N)$ and total work $W = W_2 = O(N)$ by the analysis of $\mathrm{MAX2}$. ■

Exercise 508.

Add two n-bit binary numbers in $2\lg n+1$ steps using an n-leaf c.b.t.

Proof.
$(a+b)_i = a_i \oplus b_i \oplus c_{i-1}$, where $\oplus = XOR$.

	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
a	0	1	0	1	1	1	0	0	1	0	0	1	0	0	1	0	
b	0	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	
	s	g	p	p	g	p	s	s	p	p	s	g	p	p	p	s	s
c																	
$a+b$																	

Table 14.1: Binary addition example

- s: stops a carry bit $(0+0)$
- g: generates a carry bit $(1+1)$
- p: propagates a carry bit $(0+1$ or $1+0)$.

In order to compute the k-th bit the $k-1$-st carry needs to be computed as well. There exists a non-trivial non-obvious parallel solution. We shall try for each bit position to find the carry bit required for addition so that all bit positions can be added in parallel. We shall show that carry computation takes $\Theta(\lg n)$ time on a binary tree with a computation that is known to us: parallel prefix. The i-th carry bit is one if the leftmost non-p to the right of the i-th bit is a g. The previous observation takes the following algorithmic form.

```
Scan for j=i, ... , 0
  if p ignore else
    if g
       carry=1
       exit;
    else
       carry=0
       exit;
```

Such a computation requires $O(n)$ time for $j=n$ (n-th bit). Let the i-th bit position symbol (p,s,g) be denoted by x_i. Then $c_0 = x_0 = s$
$c_1 = x_0 \otimes x_1$
$c_2 = x_0 \otimes x_1 \otimes x_2$
etc $c_{16} = x_0 \otimes \ldots \otimes x_{16}$, where

$\otimes$	s	p	g
s	s	s	g
p	s	p	g
g	s	g	g

Step 1. Compute symbol $(\{s,p,g\})$ for i bit in parallel for all i.
Step 2. Perform a parallel prefix computation on the n symbols plus 0-th symbol s in parallel where operator is defined as in previous table.
Step 3. Combine (exclusive OR) the carry bit from bit position $i-1$ (interpret g as an 1 and an s as a 0) with the exclusive OR of bits in position i to find the i-th bit of the sum.
Steps 1 and 3 require constant time. Step 2, on a complete binary tree on n leaves would require $2\lg n$ steps.

Conclusion: $T = 1+1+2\lg n$. $P = 2n-1 = O(n)$.

■

Exercise 509.

A segmented prefix (scan) computation consists of a sequence of disjoint prefix computations. Let the x_{ij} below take values from a set X and let $\oplus$ be an associative operator defined on the elements of set X. Then the segmented prefix computation for

$$x_{11}x_{12}\ldots x_{1k_1} \mid x_{21}x_{22}\ldots x_{2k_2} \mid \ldots \mid x_{m1}x_{m2}\ldots x_{mk_m} \mid$$

requires the computation of all

$$p_{ij} = x_{i1} \oplus x_{i2} \oplus \ldots \oplus x_{ij} \;\; \forall\, 1 \leq qi \leq m,\, 1 \leq j \leq k_i$$

In brief the segment separator | terminates one prefix operation and starts another one.

Show that it can be realized as a ppf operation.

Proof.

One way to deal with a segmented prefix computation in parallel is to extend $(X, \oplus)$ into $(X', \otimes)$ so that

$$X' = X \cup \{|\} \cup \{|x : x \in X\}$$

i.e. X' has more than twice the elements of X: it has all the element s of X, the segment separator | and a new element $|x$ which consists of the segment separator and x. The new operator $\otimes$ is associative if we define it as follows.

Now, if the length of the segmented prefix formula is n we can assign n processors to solve the problem with

$\mid \otimes \mid = \mid$,	$\mid \otimes x = \mid x$,	$\mid \otimes \mid x = \mid x$,
$x \otimes \mid = \mid$,	$\mid x \otimes \mid = \mid$,	$x \otimes y = x \oplus y$
$\mid x \otimes y = \mid (x \oplus y)$	$x \otimes \mid y = \mid y$	$\mid x \otimes \mid y = \mid y$

parallel prefix in asymptotically the same time. Note that an element in X' requires for its representation no more than 2 extra bits of the storage size of an element of X. If an $\oplus$ computation takes $O(1)$ time so does an $\otimes$ computation. ■

Exercise 510.

Show that the problem of computing in parallel the product of n matrices, each one of them being an $n \times n$ matrix can be reduced to that of inverting a polynomial (in n) size matrix.

Proof.

Let $A_1, \ldots, A_n$ be $n \times n$ matrices. Form an $n(n+1) \times n(n+1)$ matrix B viewed as an $(n+1) \times (n+1)$ matrix whose elements are $n \times n$ matrices. The diagonal elements of this matrix are I, the $n \times n$ identity matrix. Element $b_{i+1,i}$ is A_i. All other elements are equal to the 0 matrix. The inverse of B is an $n(n+1) \times n(n+1)$ matrix C which if viewed as an $(n+1) \times (n+1)$ matrix whose elements are $n \times n$ matrices i s such that its $j+1$ row is $(-1)^{(j+2)} A_1 \cdot \ldots \cdot A_j, (-1)^{(j+3)} A_2 \cdot \ldots \cdot A_j, \ldots, (-1) A_j, I, 0, \ldots, 0$. Element $c_{n+1,1}$ of C is matrix $A_1 \cdot \ldots \cdot A_n$. This shows that the product of n $n \times n$ matrices can be reduced to that of inverting a matrix whose dimension is a polynomial in n and reading off an $n \times n$ submatrix out of its bottom-leftmost corner. ■

Exercise 511.

In ppf (parallel prefix) over associative operator $+$, over the $x_1,\ldots,x_n$, all $x_1+\ldots+x_i$, $i=1,\ldots,n$ are computed; one of them is also $x_1+\ldots+x_n$. If the processor storing this info subsequently broadcasts it to remaining processors than all processors will receive it in additional $\lg n$ time steps. From now on, we will assume that this is so every time a ppf operation over $+$ is performed.

Let n items are stored so that item i is stored in $A[i]$ (think $A[i]$ a value associated with item i). Let a flag array f contains $f[i]=0$ if value $A[i]$ is irrelevant and $f[i]=1$ if value $A[i]$ is relevant for the next step operation. Let $m<n$ be the number of items with $f[i]=1$. Store such items in $B[1..m]$ in $O(\lg n)$ time. Furthermore Store items with $f[i]=0$ in $B[m+1..n]$ in $O(\lg n)$ time as well.

We can refer to such an operation as compaction of A into B.

Proof.
Suppose we have m 1's and $n-m$ 0's. An item $A[k]$ whose flag $f[k]$ is 1 will occupy one of the position $B[1..m]$; items with flag 0 will occupy $B[m+1..n]$. An unoptimized version is shown below.

```
         Compact (A[1..n],f[1...n],n,B[1..n])
1.  if (f[i] == 1 )  c[i]=1;
       else          c[i]=0;
2.  Parallel_Prefix (c[1..n],d[1..n],n);
3.  if (f[i]==1)
4.   B[ d[i] ] = A[i]; // d[i] is destination address for item i.
    // Broadcast d[n] (number of zeroes) everywhere.
    // Proc n broadcasts d[n] into variable Total
    // (one copy per processor, consider Total[i] copy of proc i)
5.  Broadcast(n,d[n],Total)
    // Do the same for f[i]=0 items.
6.  if (f[i] == 1 )  c[i]=0;
       else          c[i]=1;
7.  Parallel_Prefix (c[1..n],d[1..n],n);
8.  if (f[i]==0)
9.   B[Total[i] + d[i]] = A[i];
```

Lines 1-2. Every processor i is assigned one element of the input, i.e. an $A[i]$ and $f[i]$. If $f[i]$ is a 1, a variable $c[i]$ is set to one, otherwise it is set to zero. This step seems superfluous and it is. A parallel prefix operation is initiated on the $c[1..n]$ and the output is a $d[1..n]$ array.

Lines 3-4. Processors i has input $c[i]$ and output $d[i]$. What is $d[i]$? $d[i]$ is the number of zeroes found in the subarray $f[1...i]$. Therefore $d[i]$ is the destination position for item $A[i]$.

Line 5 The $d[n]$ the number of items with $f[i]=1$ is also computed in the ppf by processors n and this value is broadcast in line 5. We assume that variable Total (one copy per processor) has this value. The copy of processor i is Total$[i]$.

Lines 6-9. The same thing is repeated for the items with $f[i]=1$. Note that these items will be stored in B not in $B[1..Total[i]]$ but in $B[Total[i]+1...n]$. Therefore in the destination address computed in line 9, we must include $Total[i]$ (i.e compute the right offset).

Steps 2,5,7. Suppose we use $P=n$ processors. The running time of steps 1,3,4,6,8,9 all require $O(1)$ and steps 2,5,6 require $O(\lg n)$ steps (parallel prefix or broadcasting). The work is that of ppf or broadcasting. If however we use $P=n/\lg n$ processors both ppf and broadcasting can still be done in $O(\lg n)$ with work $W=W_2=\Theta(n)$. Steps 1,3,4,6,8,9 however will be done in $O(\lg n)$ time as each processor i will work on $\lg n)$ items. The running time and work-load remains the same.

Question. Can we do better? We can shave off one parallel prefix operation.

```
        Compact2(A[1..n],f[1...n],n,B[1..n])
1.  if (f[i] == 0 )  c[i]=1;
       else          c[i]=0;
2.  Parallel_Prefix (c[1..n],d[1..n],n);
3.  Broadcast(n,d[n],Total);
4.  if (f[i]==0)
5.   B[Total[i] + d[i]] = A[i];
6.  if (f[i]==1)
7.   B[i-d[i]]= A[i];
```

COMPACT2 does one less parallel prefix. Lines 1 and 2 are a copy of lines 6 and 7 and line 3 does the same thing as line 5 of COMPACT. Lines 4 and 5 move the items with flags $f[i]$ that are 0 similarly to lines 8 and 9 of COMPACT. The new thing is lines 6 and 7 of the code that move the items with flags $f[i]$ that are ones. After the parallel prefix of line 2, the value $d[i]$ for an item i with $f[i] = 1$ shows the number of 1's in $A[1...i]$. If we have $d[i]$ items with flags 1's in $A[1..i]$, then we must have $i - d[i]$ items with flags zeroes in $A[1..i]$. One of these is of course $A[i]$. What is then the destination address for $A[i]$ in $B[]$? It is just $i - d[i]$! That's what we do in line 7 of COMPACT2. The asymptotics of COMPACT2 are those of COMPACT1. ■

Exercise 512.

There is an obvious way to represent a positive integer n in binary. For example, 82 in binary with 8 bits is 01010010. A question in computer architecture is how to represent negative numbers; one way to represent for example -82 is 2's complement representation. An example on how to do 2's complement follows.

```
  2's complement (X,n)                    2's complement (-82, 8)
//n-bit 2's complement for X; X negative
1. Form n 1's (i.e. 2**n -1)              1.  11111111  // 2**8 -1 = 255
2. Subtract |X| (absolute value of $X$    2. -01010010  // this is 82
3. Add 1                                     ---------
4. return result                              10101101  // bits inverted
                                          3. +       1  // add 1
                                             ---------
                                              10101110  // done
```

A mnemonic to do 2's complement is the following

```
                                                      <-------
1. Scan number from right to the left                 01010010

2. If a   bit is zero leave it zero.                  01010010
3. If you see the first  one leave it,                ------10
   but invert all bit past that one until             10101110
   you reach the left side.
```

Say you have a n-bit number and n processors. How can you find 2's complement in $O(\lg n)$ time with $O(n/\lg n)$ processors with a EREW PRAM (1-bit operation takes 1 step)? Explain.

Proof.
That's a ppf problem. There are many ways to solve this problem. We give two such methods below. Let the input sequence be $x = x_n \dots x_1$. Create $z = z_n \dots z_1 = x$ i.e. duplicate x. Each processor flips its bit, i.e. $z_j = 1 - z_j$. Let x_i be the first one so that $x_1 \dots x_{i-1}$ are all zero. Then $z_i = 0$ and $z_1 \dots z_{i-1}$ are all one. A parallel prefix operation is then performed on the z's with the operation being logical AND. Let the results of the ppf be the sequence $y_1, \dots, y_n$. Then $y_1 = \dots = y_{i-1} = 1$ and $y_i = \dots = y_n = 0$. Assign processor j to the j-th input bit. Processors can determine the value of i as follows: they read in parallel y_j in step 1 and then y_{j-1} in step 2. For only one processor, $y_j = 0$ and $y_{j-1} = 1$ and this determines i. The rest is history. If y_j is 1, x_j maintains its value, if y_j is 0, $x_j = 1 - x_j$ with the exception of x_i that retains its value.
A second solution follows. Define an associative operator $\oplus$ and define $a \oplus b$ as follows. $\oplus$ will operate (and modify) z not the original input x. Note that although the input z is a 0 or 1 the application of $\oplus$ might also create a 2. After the parallel prefix a processor checks x_i and z_i. If z_i is 0 or 1, the processor does nothing on the x_i. If

$\oplus$	$b=0$	$b=1$	$b=2$
$a=0$	0	1	2
$a=1$	2	2	2
$a=2$	2	2	2

z_i is 2, the processor does $x_i = 1 - x_i$ that effectively flips the i-th bit. The modified x is returned. The purpose of the associative operator is two-fold. It leaves bits as they are if they are 0 or it is the first 1 encountered. As soon as an 1, is detected, force it to propagate a 2 in the computation. A 2 means flip corresponding bit of x. The running time and processor/work requirements are those of ppf for either of the two methods. ■

Exercise 513.
Show how the comparison of two n-bit numbers can be computed by a parallel prefix computation.

Proof.
Let $a = a_1 \dots a_n$ and $b = b_1 \dots b_n$ be two n-bit numbers (most significant bit position is position 1). Let c_i be the result of the comparison of a_i and b_i (i.e. the result is $L, G, =$). A parallel prefix on the c_i's is then performed and the result stored on the processor holding the combined result $c_1 + c_2 + \dots + c_n$ is the result of the comparison, where $+$ is an associative operator on $\{L, G, =\}$ with the following properties: $x + y$ is x if $x = L, G$, and y otherwise. ■

14.3 Network algorithms and more

Exercise 514.

How many disjoint s-dimensional hypercubes are contained in an r-dimensional hypercube for $r \geq s$? (For example, a two-dimensional hypercube contains two one-dimensional hypercubes.)

Proof.

The vertices of an r dimensional hypercube are represented by binary strings of length r. Those of an s dimensional hypercube by strings of length s. The r bits of a sequence can be split into an s bit sequence and an $(r-s)$-bit sequence. Then for a fixed $(r-s)$-bit string (fixed values of the $r-s$ bits), the values of the remaining s-bit sequence correspond to a hypercube of dimension s. There are 2^{r-s} different values for an $(r-s)$-bit binary string, and each such value gives a hypercube of dimension s. ■

Exercise 515.

Linear array. Add n, n-bit integers using a linear array of size $O(n)$.

Proof.
The algorithm is the elementary school addition algorithm. As the sum of n, n-bit numbers is $n+\lg n$-bit (or so) long, we use so many processors for a linear array. Note that if x is n-bit long then $x < 2^n$ and thus $nx < n2^n$ or $nx \leq n2^n - 1$ and therefore $\lceil \lg n2^n - 1 + 1 \rceil = n + \lg n$, assuming n is a power of two.
The n rightmost processors are used for input. Let the processors be indexed from right to left $0, 1, \ldots$ (0 is the least significant bit). The i-th least significant bit of the n integers is fed one bit after the other into processor i after an initial delay of i steps (Question: Is it required?). Each processor i then performs the operations of a binary adder for the input bits of the same arity it received from the n integer numbers (one bit at a time), if any (the left-most $\lg n$ processors have no such input), the carry bit, if any, it received from its right processor (processor 0 receives no such input) and the stored value (initially zero). The msb of the sum of the three bit values (right, input and stored) is then sent to the processor on its left and the lsb is stored locally as a stored value. The longest computation path is the 0-bit of the n-th number. It takes n steps to be fed into the 0-th processor, and $n + \lg n$ steps for the carry bit it might generate to possibly propagate to the left-most processor. Therefore, after $2n + O(\lg n)$ steps, processors store the sum of the n input numbers. ■

Exercise 516.

(a) Show how to add 2 n-bit numbers on an $\sqrt{n} \times \sqrt{n}$ mesh in $O(\sqrt{n})$ steps
(b) How long does it take on an $n^{1/3} \times n^{1/3} \times n^{1/3}$ mesh? Justify your answer(s).

Proof.
(a) It suffices to prove that parallel prefix can be performed in $O(\sqrt{n})$ steps, as carry lookahead addition completes the proof. For the 3d-mesh a parallel prefix bound in $O(\text{diameter})$ time, which is $O(n^{1/3})$, needs to be established. The algorithm emulates the actions performed on a complete binary tree and only highlights are given. We assume a row-oriented distribution of the values on which parallel prefix will be performed so that x_n resides in the top-left processor and x_1 in the bottom-right processor.
Step 1. In parallel for all rows perform independent parallel prefix operations on the rows only, where the direction of the parallel prefix is from low to high indexed x's. This takes $O(\sqrt{n})$ time (implying an $O(n)$ bound for parallel prefix on a n-cell linear array).
Step 2. Processors of the first column contain the combined result of the values on the corresponding row of the array. A parallel prefix is performed on this column. This takes $O(\sqrt{n})$ time as in step 1.
Step 3. The value stored in the first column after the ppf of Step 2 is distributed to the processors of the corresponding row. This takes $O(\sqrt{n})$ time. This value is then combined with the value obtained in Step 1 in a single step.
The overall time of the algorithm is $O(\sqrt{n})$. Proof of correctness is immediate.

3d-case.
(b) For the 3d-mesh case, the algorithm is similar to that of the 2d-case. Step 1 as before is performed on the rows of dimension 1. The partial results which correspond to the combined results of the row values reside in a $n^{1/3} \times n^{1/3}$ 2d-mesh extending over dimensions 2 and 3. The 2d-mesh prefix algorithm is then applied which requires $O(\text{diameter})$ time i.e. $O(n^{1/3})$. Finally, a step similar to Step 3 before is performed in $O(n^{1/3})$ to complete the parallel prefix operation. (If partial results are incorrectly oriented after Step 1, then a correction in their orientation before Step 2 is performed requires $O(n^{1/3})$ time). ■

Exercise 517.
Show how to add two $n\lg n$-bit integers a and b using a CBT (complete binary tree) with n leaves in $O(\lg n)$ time.

Proof.
Each of the n leaves of the cbt will compute $\lg n$ bits of the sum (the leftmost leaf may compute $\lg n+1$ bits, the extra bit being the carry out of the msb of $a+b$) as well as the carry bits for $\lg n$ bits of the two numbers a and b. In the first phase that requires $O(\lg n)$ steps, leaf i computes $\lg n$ lsb's of a_j+b_j and the symbols g,p,s for these bits as well, as in algorithm CLA. In the second phase, a parallel prefix operation on $n\lg n$ inputs needs to be performed first, in $O(\lg n)$ steps. As $\lg n$ inputs are assigned to each leaf, first, each processor performs sequential (parallel) prefix on these inputs. The final result (rightmost parallel prefix result computed on each leaf processor) is utilized as an input to this leaf for a subsequent n-input parallel prefix on a cbt that requires $O(\lg n)$ steps. At the end of this computation the parallel prefix result on leaf i is combined with each of the $\lg n$ results of the sequential prefix operation so that the prefix on an $n\lg n$-input problem is computed. Total time for this phase is $O(\lg .)$. The carry bits computed in the second phase are combined with the sums a_j+b_j of the first phase to give the sum of a and b. As each leaf contains $\lg N$ carry bits and sums this step requires $O(\lg n)$ steps. ■

Exercise 518.
Show how to sort on a CBT with n leaves, n comparable keys in $O(n)$ time. Each key is input and output through a leaf.

Proof.
Each key is assigned to a leaf of the cbt. Each internal node of the tree finds the minimum of its two children and forwards it to its father. If one of its children has no keys, it forwards the key of its other child. If none has a key, it stops forwarding keys. When this procedure is used, the minimum of the n keys comes out of the root in $\lg n$ steps. In the following step $\lg n+1$, the second smallest step comes out of the root of the cbt and in general, in the $\lg n+i-1$-th step, the i-th smallest key is output. The root can also redirect the i-th smallest key to arrive in the i-th leftmost leaf of the tree. The largest of the n keys arrives in the root in step $\lg n+n-1$ and to the rightmost leaf $O(\lg n)$ steps later for a total of $O(n)$ operations. ■

Exercise 519.

On a 2^n-node hypercube processor 0 possesses one word of information. Present an algorithm for broadcasting this information to the remaining vertices of the hypercube in time proportional to the diameter of the network. (You may assume that each processor knows its processor number and the geometry of the network).

What if message size is not one word but $M > n$ words? How fast can you broadcast on the hypercube then?

Hint: Split M into pieces and broadcast pieces independently.

Proof.

Let $N = 2^n$. We use the $\lg N$ step PRAM algorithm. In round i, processor $j < 2^i$ transmits the message to processor $j + 2^i$. Processors j and $j + 2^i$ differ in the i-th lsb, i.e. a dimension $n - i$ edge of the hypercube connects them and so this transmission is possible. For the optional part try to split the message into $\lg N$ pieces and use an algorithm similar to the one described (you are allowed to communicate along all the links of a single processor in unit time).

■

Exercise 520.
Give an algorithm for broadcasting an one-word message on the following networks. You may assume that links can transfer a whole word in a single cycle/clock-tick. What is the corresponding running time? Use $O(.)$ notation to express time bounds.
(i) n-cell linear array. Message is held by leftmost processor.
(ii) n-leaf complete binary tree. Message is held by root.
(iii) $(n \times n)$-cell 2d-array. Message is held by top-leftmost processor.

Proof.
(i) In the i-th step, processor $i-1$ sends the received message from processor $i-1$ (or stored message if $i=0$) to processor i. Broadcasting completes in $n-1$ steps.
(ii) Each node that receives the message from its father (for root, it is locally stored), it sends it to its children. Broadcasting completes in $\lg n$ steps (i.e. depth of the tree).
(iii) Top-leftmost processor broadcasts message using algorithm in (i) to all the processors of the first row. Each one of these processors broadcasts received message to all processors of the same column using algorithm in (i) and viewing each column as a linear array. Total broadcast time is $2n-2$.

■

Exercise 521.

Show how to multiply two $n \times n$ matrices A and B in $O(\lg n)$ steps on an $O(n^3)$-node shuffle-exchange graph. Be brief but explicit, i.e. identify where the (i,j) entry of each matrix goes (including that of the product) and where the $a_{ij}b_{jk}$ product is computed.

Proof.

We present only highlights of the algorithm (the solution is not the most elegant one for this problem). We are going to use a $2^{3\lg n}$-size shuffle-exchange graph (n is assumed to be a power of two). We represent the processors by triplets (i,j,k). We assume that processor $(0,i,j)$ holds a_{ij} and b_{ij} of A and B respectively. The result $C = A \times B$ is stored so that processor $(j,i,0)$ will store c_{ij}. In the discussion below, when we refer to shuffle edges we shall mean the ones corresponding to a left-shift of a processor (not a right-shift). In step 1 we broadcast a_{ij} and b_{ij} to processors $(i,j,*)$ ("*" is to mean any valid processor number for that position). This is achieved by copying elements of A and then of B along the shuffle (left-shift) edges first and then along the exchange edges to neighboring processors. This step takes $4\lg n$ steps. In step 2 we rearrange A so that element a_{ij} is stored in processors $(j,*,i)$. This is achieved in $\lg n$ steps by utilizing the shuffle edges (left-shift) for copying the elements. After this step, processor (i,j,k) holds a_{ki} (for all j) and b_{ij} (for all k). Note that in the previous sentence we renamed indices. In step 3, processor (i,j,k) computes the product $a_{ki}b_{ij}$. In order to produce element c_{kj}, the sum $\sum_i a_{ki}b_{ij}$ needs to be computed. In step 4, the sum $\sum_i a_{ki}b_{ij}$ is computed in processor $(j,k,0)$. We add products generated in step 3, by first copying through the shuffle-edges (left shifts) and then summing results held on the adjacent processors that are incident on exchange edges and storing the partial sum in the processor having (its binary representation) lsb 0. This way processor $(j,k,0)$ gets c_{kj} or by renaming $(j,i,0)$ gets c_{ij}. Total running time is $O(\lg n)$. ■

Exercise 522.
Give a CREW algorithm for solving the problem of multiplying an $n \times n$ matrix A and vector x in $O(\lg n)$ time. How many processors does your algorithm require? How much work does it require? Comparing your algorithm to its sequential counterpart, what is its efficiency and speedup? (Use $O(.)$ notation to describe various results).

Proof.
Let an $n \times n$ matrix A and vector x be given. An n^2 processor CREW PRAM will be used. We name the processors of the PRAM (i, j), $1 \leq i, j \leq n$. Processor $(*, j)$ (a $*$ indicates "any processor" (k, j) such that $1 \leq k \leq n$) reads vector element x_j. A concurrent read is required for this operation. Processor (i, j) reads element a_{ij} of A (an exclusive read operation). Then, processor (i, j) computes $a_{ij}x_j$ and stores the result in a local register. Subsequently, processors $(i, *)$ use the parallel sum PRAM algorithm to compute in time $\lg n$ the sum $\sum_j a_{ij}x_j$ which is y_i of the result vector $y = Ax$. Note that the n parallel sums utilize distinct sets (of n) processors. The running time of the algorithm is thus $T = O(\lg n)$ (constant time for the read operations and the multiplication, and $\lg n$ time for the parallel sum). For this algorithm, $P = n^2$ and $W = PT = O(n^2 \lg n)$. A sequential algorithm that solves the same problem runs in time $T_{seq} = O(n^2)$ (if multiplications and additions are taken into consideration in the running time) for a speedup $s = O(n^2/\lg n)$ and an efficiency $e = O(1/\lg n)$. ■

Exercise 523.
Make the algorithm above to work on an EREW PRAM. What is its running time?

Proof.
The matrix vector multiplication algorithm can be made to run on an EREW PRAM if the concurrent read operation involving the reading of x_j is eliminated. This can be achieved by, for example, making processor $(0, j)$ first reading x_j and then broadcasting x_j to the remaining $n-1$ processors $(*, j)$, an operation that requires $\lg n$ steps on an EREW PRAM. Therefore, the EREW PRAM variant has $T = O(\lg n)$ (though the hidden constant becomes two from one previously). ■

Exercise 524.

For the algorithm in the two previous problems, let us assume that we have only p processors available where $p < n$ and n is a multiple of p. How fast can we solve the matrix-vector multiplication problem on an EREW PRAM? Explain. Express parallel time, speedup and efficiency in terms of n and p.

Proof.

The are various ways to perform this multiplication. We sketch one (not necessarily the best) that divides the rows of matrix A into p groups of n/p rows each and assigns group i to processor i. Similarly, vector x is split into p groups X_i of n/p elements each. The algorithm works in p phases. In phase j processor i finds those partial products that are related to the multiplication of group i of A with $X_{(i+j)\%p}$ an operation that requires $O(n^2/p^2)$ time. As there are p phases, the total time of all phases for processor i is $O(n^2/p)$ as required. At the end processor i will have computed n/p elements of vector $y = Ax$ (group Y_i of y defined similarly to X_i). What happens if we split the column of A into p groups? What happens if we split A into p submatrices of dimension $n/\sqrt{p} \times n/\sqrt{p}$ each so that each column and row of A belongs to $\sqrt{p}$ submatrices. ■

Exercise 525.
The discrete Fourier transform of an n vector $x_1,\ldots,x_n$ is defined to be $y_1,\ldots,y_n$, where

$$y_i = \sum_{j=1}^{n} w^{(i-1)(j-1)} x_j$$

for $1 \leq i \leq n$, and w is an n-th primitive root of unity. Design an $O(n)$-step algorithm for computing the discrete Fourier transform of an n-vector in $O(n)$ steps of an n-cell linear array. Data can only be input or output from one end of the array, but you may assume that w is provided as part of the input but you must compute the powers of w as part of your algorithm.

Proof.
Let the n cells of the n-cell linear array be labeled $1,2,\ldots,n$ for a left to right labeling. Input is fed into processor 1 (leftmost). Input w is fed first, then x_1, x_2 and finally, x_n. As in the convolution problem, each processor i transfers simultaneously to its right one not only w, when needed, but also w^{i-1} as well. Processors perform two tasks. When processor i receives w and w_{i-2} (except for processor 1) it initializes register $A = 0$ stores $w^{i-2}w = w^{i-1}$ into register B and forwards to its right processor w and w^{i-1}. A third register C is also initialized ($C = 1$). As w moves from left to right, at step i it enters processor i that then stores w and computes and stores w^{i-1}. At time $i+j+1$, processor i receives x_j. and does the following computations: $A = A + C * x_j$, $C = C * B$. Register C thus computes $w^{(i-1)j}$. At time step $i+n+1$ processor i has computed y_i. Total time is $2n+1$ steps (for processor n to compute y_n). ■

Exercise 526.
Show that sorting the rows and then the columns of a mesh leaves the rows in sorted order.

Proof.
We are going to use the 0-1 Sorting Lemma. After row and column sorting consider an arbitrary row i of mesh a and say that two keys in column j and k are not in proper order i.e $a_{ij} = 1$ and $a_{ik} = 0$. Since $a_{ik} = 0$, all $a_{lk} = 0$, $l \leq i$ as well, and since before column sorting the rows were sorted, this means that at least i rows of the mesh must have at least k 0's each. When the columns are sorted these 0's are moving up and thus the submesh of dimension $i \times k$ with top-left end the top-left end of the mesh must contain 0's (the ones of the at least i rows of at least k 0's each). This forces $a_{ij} = 0$ (as it belongs to that submesh), a contradiction. ■

14.4 Uni vs Multi

Exercise 527.

In this problem we show the effects of parallelizing a "slow" sequential sorting algorithm. The net effect is to obtain a superlinear speedup of more than p. The input to the problem is n keys $X = \langle x_0, \ldots, x_{n-1} \rangle$, where n is a multiple of p.

Algorithm ALG1. Let ALG1 be a sequential $O(n^2)$ running time sorting algorithm of the selection-sort/bubble-sort type. Assume for the remainder that its running time is $T_1(n) = n^2$. This is the baseline algorithm used for speedup purposes.

Algorithm ALG2. ALG2 is the following modification of ALG1. It is still a sequential algorithm that will be used to derive a parallel version of ALG1 in the form of ALG3 below.

```
  Alg2 (<x0..xn-1>)
1. Split  <x0..xn-1> into p equally sized groups,
   each one consisting of n/p keys.
2. Call Alg1 on each one of the p subsequences.
3. Merge the p sorted subsequences of step 2.
```

Show how to do step 3 in $O(n \lg p)$. Then analyze the remaining steps of the algorithm and derive an expression of its worst-case running time in term of n and p. For what values of p is the running time $o(n^2)$? Explain. (A little-oh was used, not a Big-oh!)

Algorithm ALG3. ALG3 is the following EREW PRAM parallelization of ALG2.

```
  Alg3 (<x0..xn-1>)
1. Processor $i$ is assigned the $i$-th subgroup of
   consecutive keys <x_in/p .. x_in/p+n/p-1>
   each one consisting of n/p keys.
2. Processor i sorts the i-th group using Alg1 in
   parallel with all other procs.
3. The  p sorted subsequences  are then merge by processor 0.
```

What is the parallel running time of ALG3? What is its processor count P? Explain. For what values of p is the speedup strong superlinear (i.e. $\omega(p)$)? (ω is not Ω). For which of the following cases is the speedup strong superlinear? (i) $P = \lg n$, (ii) $P = \sqrt{n}/\sqrt{\lg n}$, (iii) $P = \sqrt{n}$, (iv) $P = n/\sqrt{\lg n}$, (v) $P = n$. Explain. How could you characterize (v)? Explain.

Proof.

(a) Algorithm `Alg1` has run-time n^2 (per problem statement), which becomes the baseline for speedup reporting.

(b) Split n into p groups of size n/p. Each group is sorted by `Alg1` in time $(n/p)^2$ per group. Since we have p groups total time is $p \cdot n^2/p^2 = n^2/p$ This is the time of lines 1 and 2 of `Alg2`. The p sorted sequences are then merged bottom up by reducing their number from p to $p/2$ to $p/4$ etc in $\lg p$ phases. Each phase takes time n for a total of $n \lg p$ overall $\lg p$ phases. [Consider for example the first phase that includes $p/2$ merges of pairs of sorted sequences, each pair involving two sorted sequences of n/p keys. Each merge takes $2n/p - 1 \leq 2n/p$ comparisons for a total of $p/2 \cdot 2n/p = n$ comparisons. In the next step we have $p/4$ merges but each sorted sequence of the pair is now $2n/p$-key long.] The overall running time is thus $n^2/p + n \lg p$. If $p = \omega(1)$ but $p \leq n$, t he running time is $n^2/p + n \lg p = o(n^2) + O(n \lg n) = o(n^2)$.

(c) Algorithm `Alg3` is a parallelization of Alg 2. The p sorting operations are done independently and in parallel thus at a cost of n^2/p^2 in time NOT in n^2/p. Everything else remains the same $n \lg p$. Total time is now $T_2(n,p) = n^2/p^2 + n \lg p$, if p processors are used.

Let's answer questions (i)-(v) first. The baseline time for speedup results is n^2. Let the speedup be indicated by s.

(i) $P = \lg n$, i.e. $s = \Theta(n^2/(n^2/p^2)) = \Theta(p^2)$ and the speedup is strong superlinear since $\Theta(p^2) = \Theta(\lg^2 n) = \omega(\lg n) = \omega(p)$.

(ii) $P = \sqrt{n}/\sqrt{\lg n}$. Then $s = \Theta(n^2/(n^2/p^2 + n \lg p)) = \Theta(n^2/n \lg n) = \Theta(n/\lg n = \Theta(p^2) = \omega(p)$. The speedup is still strong superlinear.

(iii) $P = \sqrt{n}$, then $s = \Theta(n^2/(n^2/p^2 + n\lg p)) = \Theta(n/\lg n) = \Theta(p^2/\lg n) = \omega(p)$. The speedup is still strong superlinear.
(iv) $P = n/\sqrt{\lg n}$, then $s = \Theta(n^2/n\lg n) = \Theta(n/\lg n$ which is $o(n/\sqrt{\lg n}) = o(p)$, and thus the speedup is sublinear.
(v) $P = n$, then $s = \Theta(n/\lg n)$ and thus again the speedup is sublinear.
All five cases can be analyzed summarily as follows. The speedup s is $s = n^2/T_2(n,p) = \frac{n^2}{n^2/p^2+n\lg p}$. Consider the denominator. If $n^2/p^2 \geq n\lg p$, the first term dominates the sum otherwise the second term dominates the sum (and disregard the non-dominating term). Thus $n^2/p^2 \geq n\lg p$ is equivalent to (roughly) $p \leq \sqrt{n}/\sqrt{\lg n}$, and for those cases the speedup is $s = \Theta(n^2/n^2/p^2) = \Theta(p^2)$. Such cases are (i), (ii) above. For all other cases $p > \sqrt{n}/\sqrt{\lg n}$, and the speedup is $s = n^2/n\lg p = n/\lg p = \Theta(n/\lg n)$. Such cases are (iii), (iv), and (v) above. One of the reasons for the "supelinear speedups" is that the original `Alg1` quadratic algorithm has been infected in `Alg2` by elements of merging, an $O(n\lg n)$ approach. In fact question (v) with $P = n$ if you exclude the trivial local sort of 1 key per processor turns outs to be mergesort. Thus a more fair speedup comparison would be with `Alg2` with $p = n$ i.e. merge-sort. Thus an "apples-to-oranges" comparison might yield interesting results. ■

14.5 Slack greater than one

Exercise 528.

Add n numbers on the p processors of a PRAM in $O(n/p + \lg p - 1)$.

Proof.

Split the input into p groups of n/p keys each. Processor i is assigned the i-th group and adds the numbers sequentially in $n/p - 1$ additions. This is dones in lines 1-4 below. p sums are formed and summed up using the parallel sum algorithm in $\lg p$ time in step 5. Total is $n/p + \lg p - 1$. If n is not a multiple of p and $\lg p$ is not an integer, two more steps are required.

```
   Sum(n,p,A[0..n-1])
1  mypid = pid(); //pid = 0...p-1
2  sum[pid]=0;
3  for(i=0;i<n/p-1;i++)
4   sum[pid] = sum[pid] + A[pid*n/p+i];
5  ParallelSum(+,sum,Sum); //Parallel sum on p inputs
6  Return(Sum)
```

■

Exercise 529.

Parallel BubbleSort or SelectionSort. Explore the possibilities for sorting n keys on p processors, where $p < n$. You may assume p divides n, and p is a power of two, if needed. Assume that the cost of communication is 1 relative to the cost of memory access.

What if $p = \sqrt{n}$?

Proof.

Step 1. We split the n keys into p groups of n/p keys. Each group is sorted by Bubble/Selection-Sort in $O((n/p)^2)$ time.

Step 2. We then merge the p sorted sequences. We utilize $p/2, p/4, \ldots, 2, 1$ processors for the merging. Each parallel round takes $O(\frac{n}{p} \cdot 2^i)$ parallel time, where $i = 1, \ldots, \lg p$. Total parallel time is

$$O\left(\frac{n^2}{p^2} + \sum_i \frac{n}{p} 2^i\right) = O\left(\frac{n^2}{p^2} + \sum_{i=1}^{\lg p} \frac{n}{p} 2^i\right) = O\left(\frac{n^2}{p^2} + n\right)$$

Step 2 alternative. Send all the results to processor 1 to merge the p sorteq sequences. This takes $O(n \lg p)$. Total parallel time is

$$O\left(\frac{n^2}{p^2} + n \lg p\right)$$

Step 2 another alternative. Use bitonic merging or just parallel merging. Analyze its time complexity.

Finally if $p = \sqrt{n}$ then $O\left(\frac{n^2}{p^2} + n\right) = O(n)$ and $O\left(\frac{n^2}{p^2} + n \lg p\right) = O(n \lg n)$ respectively. ■

Exercise 530.

Design an algorithm for sorting n numbers on an $O(\log n)$-processor complete binary tree that has $\Theta(1)$ efficiency. (You may assume that each processor can process and store $O(n/\log n)$ numbers, and you can take advantage of the fact that a single sequential processor can sort N numbers in $O(N \log N)$ steps. You may also allow I/O at each processor of the network).

Proof.
The algorithm borrows itself from sequential mergesort and is pipelined in nature. A complete binary tree on $\lg n$ leaves is constructed. Each leaf is assigned $n/\lg n$ keys which are sorted locally in $O(n)$ time using say, sequential mergesort. Each internal processor thereafter works as follows. It receives from each of its two sons a key, and sends the minimum of the two to its father and requests a new key from the son who provided the minimum. The root redistributes keys so that the leftmost leaf receives the $O(n/\lg n)$ smallest keys and so on. Upward and downward movement is independent of each other (it can occur in alternate steps). The minimum of the n keys is output from the root in $O(\lg n)$ steps, while the maximum after $O(n+\lg n)$ steps. The overall time is thus $O(n)$ utilizing $O(\lg n)$ processors for an efficiency of $\Theta(1)$. ■

14.6 Challenging problems

Exercise 531.

From F. T. Leighton's book [15]. Given a $\sqrt{n} \times \sqrt{n}$ matrix of inputs $X = (x_{ij})$, a two-dimensional FFT on X consists of computing an FFT on the $\sqrt{n}$ values in each column of X, and then computing an FFT on the $\sqrt{N}$ values in each row of the resulting matrix.. Show how to compute a two-dimensional FFT in $\lg n$ steps with one pass through a $\lg n$-dimensional butterfly when n is a power of four. How fast can a two-dimensional FFT be computed on an n-node hypercube?

Proof.

We are going to use the FFT algorithm and thus we arrange the butterfly so that its last level appears as input and its first level as output (reverse butterfly). As $N = 4^m$, $\sqrt{N} = 2^m$ i.e. there exists a butterfly on $\sqrt{N}$ rows.

Step 1. The FFTs of the columns of the butterfly will be computed on $\sqrt{N}$-row butterflies. We thus note that the last $\lg N/2$ levels of a butterfly form $\sqrt{N}$ consecutive $\sqrt{N}$-row butterflies revealed by deleting levels 0 through $\lg N/2$. On a $\sqrt{N}$-row butterfly the input to row i is, as in the FFT, the element of row reverse(i) (the reverse of the binary representation of i).

Step 2. On levels $\log N/2$ through 0 we compute the FFT on the rows of the butterfly on the outputs generated in Step 1. We note that the constants involved (i.e. powers of primitive roots of unity) are the same for both steps and involve $\sqrt{N}$-th primitive roots of unity. The $\sqrt{N}$ butterflies of this step, as opposed to those of Step 2, are interleaved which means that the first butterfly receives as input the first element of all columns (output of step 1), i.e. row 1 of the matrix, as required. We note that the j-th input to each such butterfly, by the FFT computation, should be an element of the reverse(j) column. In order to achieve that, the input to the j-th butterfly in step 1 is the reverse(j) column of the matrix (and from the discussion of step 1, the input to row i of this butterfly is the element of row reverse(i)). The total time of the algorithm is proportional to the number of levels of the butterfly. As the 2-d FFT algorithm operates on a leveled network, it can be readily ported as a fully normal algorithm on the hypercube.

■

Exercise 532.

Definition. An embedding of a graph $G = (V,E)$ into a graph $G' = (V',E')$ is a function ϕ from V to V'.

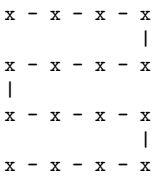
Definition. The dilation of the embedding ϕ is defined as follows. $dil(\phi) = max\{dist(\phi(u),\phi(v)) : (u,v) \in E\}$, where $dist(a,b)$ is the distance in edges between $a,b \in V'$.

(a) Show that a ring can be embedded into a 2d-array with dilation 1, if and only if the number of vertices of the array is even.

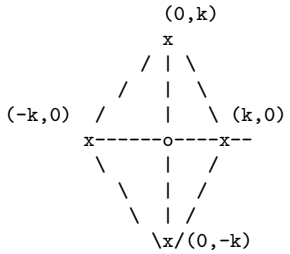
(b) Show that in a 2d-array the number of vertices which are within distance k from any vertex is at most $2k^2+2k+1$.

(c) A complete binary tree can not be embedded into a 2d-mesh with dilation 1 for any $k > 4$.

Proof.
(a)



An embedding of a 16-proc linear array into a 4x4 2d-array with dilation 1 is shown above.
(b)



All the nodes in the square are within distance k from the origin (0,0). Note distance is Manhattan distance horizontal-vertical steps not Euclidean distance. Number of nodes in each quarter is $1+2+...+k = k(k+1)/2$ excluding $(0,0)$. Total is $4k(k+1)/2+1$ with the plus one accounting for the $(0,0)$ contribution.
(c) A binary tree of depth k has $2^{k+1}-1$ vertices. On a 2d-mesh, within distance k from any vertex, there are at most $2k^2+2k+1$ vertices. As $2^{k+1}-1 > 2k^2+2k+1$, $k > 4$, it is evident that no such embedding exists. ■

Exercise 533.

Show that a complete binary tree of $2n-1$ nodes (i.e. n leaves) cannot be embedded with dilation one into a 2d-array with at least $2n-1$ nodes, $n>16$.

Hint: Show that the number of nodes in the array within distance k from some array node is at most $2k^2+2k+1$ which is less than than $2^{k+1}-1$, $k>4$, the number of nodes within the same distance in the tree.

Proof.

Let O be a processor of a 2d-array. The number of processors at distance k from O are within the area (and the perimeter) of ABCD of the rectangle of side $2k$ shown in that figure. All the processors that are distance k from O are on the vertical lines shown in that figure. The central vertical line contains $k+1+k$ processors. Each of the two surrounding it contains $(k-1)+1+(k-1)$ processors and so on. The total number of these processors is (noting that $1+2+\ldots+i=i(i+1)/2$).

$$(k+1+k)+2((k-1)+1+(k-1))+\ldots+2(0+1+0)=(2k+1)+2k+2(k-1)k/2+2(k-1)k/2=2k^2+2k+1.$$

The number of processors withing distance k from the root of a CBT (complete binary tree) is $2^{k+1}-1$ (including internal nodes). Had it been possible for a CBT to be embedded into a 2d-array, then the root of the tree it would have been mapped to a node of the array. Call that node of the array O. By the previous results, there are $2k^2+2k+1$ nodes in the array within distance k from O and it is $2^{k+1}-1>2k^2+2k+1$ $(k>4)$. Therefore such an embedding with dilation one is not possible. ■

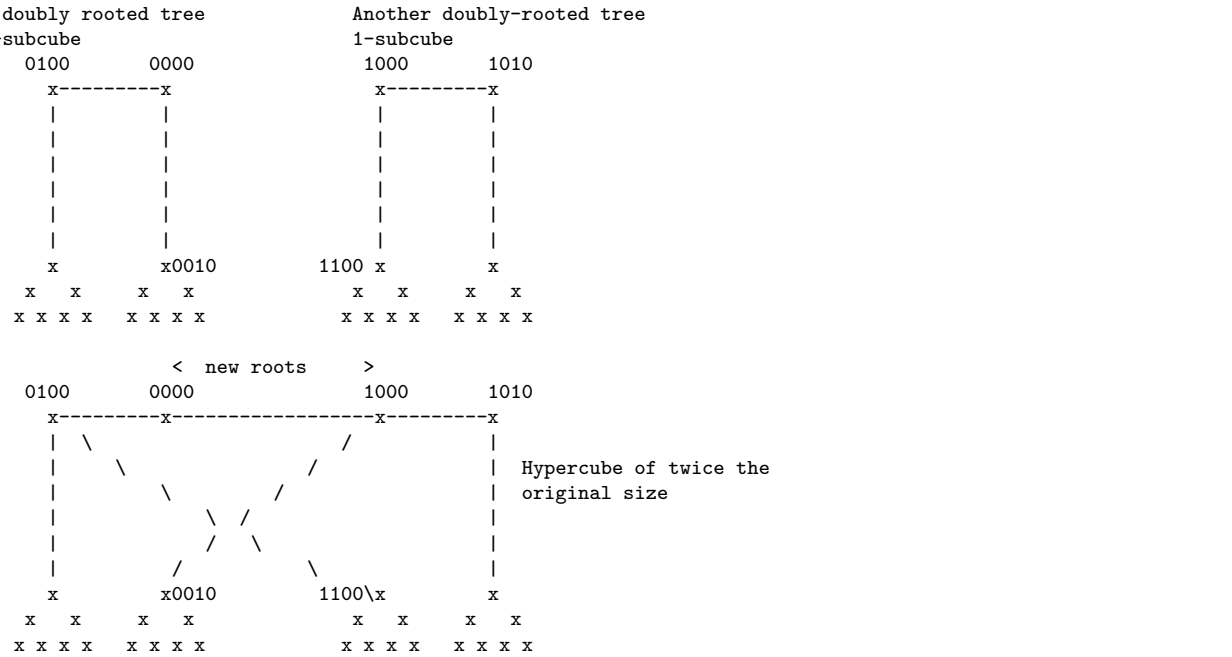
Exercise 534.

An N-vertex double-rooted complete binary tree is a subgraph of an N-vertex hypercube.

And therefore, an $N/2$ vertex complete binary tree can be embedded into an N vertex hypercube.

Proof.

In a doublty rooted tree the two roots would be mapped to opposite parity vertices and therefore two vertices at distance i from roots r_1 and r_2 respectively are mapped to vertices of opposite parity. This intuitive argument is proved by (pictorial) induction below.



■

Exercise 535.

Show that the N-node hypercube can be bisected by removing $O(N/\sqrt{\lg N})$ nodes, or by removing $N/2$ edges. (Stirling's approximation formula for the factorial is $n! = (n/e)\sqrt{2\pi n}$.

Proof.

As far as the $N/2$ edges are concerned, this is a direct consequence of the recursive/inductive definition of the hypercube i.e. removal of all the edges of any dimension i gives such a bisection as it splits a hypercube of dimension n into two of dimension $n-1$. Let u be a hypercube node. Let $size(u)$ be the number of 1's in its binary representation. For a hypercube node u of size k, u is connected by an edge to hypercube nodes of size $k-1$ or $k+1$ (flipping a bit in some dimension to find a neighbor of u either increases size by 1, if a 0 becomes an 1, or decreases it by 1). We claim that we can bisect the hypercube if we remove nodes that have size $\lfloor \lg N/2 \rfloor$ and $\lceil \lg N/2 \rceil$ (note that if $\lg N$ is even only nodes of size $\lg N/2$ need to be removed). If we remove these nodes, then no node of size $k < \lfloor n/2 \rfloor$ can be connected to a node of size $l > \lceil n/2 \rceil$ as $l - k > 1$. The number of nodes of size k is $\binom{n}{k}$ (i.e. number of ways of having k 1's in a bit sequence of length n). By straightforward properties of the choice function $\binom{n}{k} = \binom{n}{n-k}$. This proves that the above defined splitting of the vertices is a bisection as the number of vertices of size $k < \lfloor n/2 \rfloor$ is equal to the number of vertices of size $n - k > \lceil n/2 \rceil$. We now estimate the number of the removed vertices.

Case 1. n is even, i.e. $n = 2x$. Then we have, after using Stirling's approximation formula for the factorial.

$$\binom{2x}{x} = \frac{(2x)!}{x!x!} \approx \frac{2^{2x}}{\sqrt{\pi x}} = O(N/\sqrt{\lg N}).$$

Case 2. n is odd i.e. $n = 2x+1$. Then we have, by using the fact that $2x+1 \leq 2(x+1)$ and $(1+1/x)^x \leq e$.

$$\binom{2x+1}{x} + \binom{2x+1}{x+1} = 2 \cdot \binom{2x+1}{x} = O(N/\sqrt{\lg N}).$$

■

Exercise 536.
Show that the bisection width of an $\sqrt{n} \times \sqrt{n}$ mesh is at least $\sqrt{n}$

Proof.
The proof is by contradiction. Suppose that the bisection width W of the mesh is less than $\sqrt{n}$ (i.e. at most $\sqrt{n}-1$). Let us label the four sides of the mesh A, B, C, D. If $W \leq \sqrt{n}-1$, a bisection starting from say side A cannot end on opposite side C (similarly, for any other origin) since that would imply $W \geq \sqrt{n}$. It would thus end in B or D. Say, without loss of generality it ends in B. Let the number of cuts of links parallel to sides A/C be x and parallel to B/D be y. Then $x+y=W$. The number of processors enclosed in the region surrounded by the bisection and sides A and B is at most xy (the bisection line moves at most x links away from A and is no more than y away from B). Let us maximize xy (processor size on one side of bisection) under the constraint $x+y \leq \sqrt{n}-1$. Elementary arguments show that xy is maximized when $x=y=(\sqrt{n}-1)/2$. Then, $xy=(n-2\sqrt{n}+1)/4 < n/2-1$, for any $n \geq 3$. This implies that on one side of the bisection width there are at most $n/2-2$ processors and on the other at least $n/2+2$, which contradicts the definition of bisection width (two sides differ by at most 1). ■

Exercise 537.

May be [15].

Show that r perfect outshuffles return a deck of n cards, where n is even, to its original order, if and only if r is the smallest integer such that $2^r = 1 \bmod (n-1)$. How many shuffles are needed when $n = 2^k$? How many for a regular deck ($n = 52$)? How many for a regular deck with two jokers ($n = 54$)?

Proof.

A card in position $u < N/2$ after an outshuffle ends in position $2u \bmod (N-1)$, whereas a card in position $u \geq N/2$ ends in position $2(u - N/2) + 1 \bmod (N-1) = 2u - (N-1) \bmod (N-1) \equiv 2u \bmod (N-1)$, i.e. in both cases, the destination is $2u \bmod (N-1)$. Applying this rule r times, a card at position j after r outshuffles ends in position $2^r j \bmod (N-1)$. We note that the first and the last cards do not move. It suffices then to examine the case $0 < j < N-1$. After r outshuffles a card returns to its original position if $2^r j \equiv j \bmod (N-1)$. This means that $N-1$ must divide $(2^r - 1)j$. As $j < N-1$ this means that $N-1$ must divide $2^r - 1$ which is equivalent to $2^r \equiv 1 \bmod (N-1)$. We also note that r outshuffles are also necessary as card 1 (the second card) would otherwise not return in its original position.

(a) If $n = 2^k$ then the minimum r so that $2^r \equiv 1 \bmod (2^k - 1)$, is $r = k$

(b) If $n = 52$, $2^8 \equiv 256 \equiv 1 \bmod 51$, then $r = 8$.

(c) If $n = 54$, $2^{52} \equiv 1 \bmod 53$, then $r = 52$. [One may observe that 53 is a prime number and by Fermat's theorem for p a prime, $2^{p-1} \equiv 1 \bmod p$. r is minimum because $2^{52/p} \not\equiv 1 \bmod 53$ for any p that divides 52. This means that 2 is a generator of the cyclic group Z_{53}^*]. ■

Exercise 538.
You are given a hypercube with N nodes where $N = 2^{3n}$. Describe a matrix multiplication algorithm of two $M \times M$ matrices A and B on the hypercube, where $M = 2^n = N^{1/3}$. You need to explain which processor holds each element of A and B, where each computation is performed, and where each element of $C = A \times B$ is stored at the completion of the algorithm. (Hint: Use a triplet (a,b,c) to represent a processor id).

Proof.
The nodes of the hypercube are represented by a binary string of length $3n$. Such a string can be broken into 3 substrings of length n as well, i.e. $x = x_1 \dots x_n x_{n+1} \dots x_{2n} x_{2n+1} \dots x_{3n}$ is the same as $x = abc$ where, $a = x_1 \dots x_n$, $b = x_{n+1} \dots x_{2n}$, and $c = x_{2n+1} \dots x_{3n}$. Let us further represent string $x = abc$ as $x = (a,b,c)$. Each of a, b and c represents 2^n possible values. Use the EREW PRAM matrix multiplication algorithm for matrix multiplication on this hypercube observing that "processor i reads from /writes into location j" in the PRAM algorithm corresponds to "processor i reads from/writes into memory of processor j" of the hypercube algorithm. ■

Exercise 539.

Can we find the maximum of n comparable keys with a CRCW PRAM in one (comparison-related) step using fewer than $n(n-1)/2$ processors?

Proof.
MAX can be found in ONE parallel step with $n(n-1)/2$ processors. This involves finding the key whose rank is $n-1$, i.e. it has all the remaining $n-1$ keys smaller than it. A set of n keys establish $n(n-1)/2$ pairs of keys and thus comparisons to be realized to obtain all the information required to find the MAXIMUM using the rank operation described. This concludes the proof that an upper bound of $n(n-1)/2$ processors is needed for the one comparison-related step.
One step MAX requires at least $n(n-1)/2$ processors. To prove the lower bound on processors, we use a proof by contradiction. Suppose we can do it with one fewer processor that is, $P = n(n-1)/2 - 1$. Since there are $n(n-1)/2$ pairs of keys to compare, one such pair is not compared. Call the keys of the pair x, y. What we are going to show, by playing the role of an adversary, is that we can set up the values of the n input keys so that the missing comparison of the x, y is the crucial one to establish the maximum. We thus play the role of an adversary whose only mission is to make the algorithm that uses P processors to fail. To do so, we set the results of the comparisons in such a way that x, y are the MAX and SECOND MAX keys. Thus the comparison between x, y (that is not being performed) is CRUCIAL in determining the MAXIMUM of the n keys. Since it is not performed we cannot find the MAX with P processors. Contradiction is established.
Can we find the MAX of n keys in 2 steps say using $O(n^{3/2})$ processors ? YES, we can, as it was shown elsewhere. What if we have n processors only and of course n keys? Before we prove Valiant's lower bound that at least $\lg\lg n$ parallel steps are needed to computer the MAX of n keys with $p \leq n$ processors, we establish a graph theory related result: Turan's theorem. ■

Exercise 540.

On an undirected graph $G=(V,E)$, an **independent set** is a set of vertices such that no two vertices are connected by an edge. On an undirected graph $G=(V,E)$, a **clique** is a set of vertices such that any two vertices are connected by an edge. A clique on n vertices is the "complement" of an independent set on the same n vertices. The complete graph on n vertices is a clique on n vertices by default. A clique of n vertices has $n(n-1)/2$ edges.

Turan's theorem (version 1). Let $G=(V,E)$ be an undirected graph, where $|V|=n$ and $|E|=m$. Graph G has an independent set of size at least

$$\frac{n^2}{2m+n}.$$

.

Turan's theorem (version 2). Let $G=(V,E)$ be an undirected graph, where $|V|=n$ and $|E|=m$. If graph G has no p clique then it has at most

$$\frac{(1-\frac{1}{p-1})n^2}{2}$$

edges.

Proof.

Case 1. If $n \leq p-1$ then G does not have obviously a p-clique and G has at most $n(n-1)/2$ edges. It is obvious that $n(n-1)/2 \leq (1-1/(p-1))n^2/2$ by elementary calculations.

Case 2. Thus the interesting case left is $n \geq p$. If graph G has the maximum number of edges but does not have a p-clique it must have a $(p-1)$-clique. This is because otherwise we could add edges to G to create such a $(p-1)$-clique; this would contradict the maximality of edges of G. Call C a $(p-1)$-clique of G. C has $(p-1)(p-2)/2$ edges. Call G' the graph G without C, i.e. $G'=G-C$. Graph G' has m' edges. Let k be the number of edges that go from G' to C. By induction on G' we have that $m' \leq (1-1/(p-1))(n-p+1)^2/2$. Since G does not have a p-clique every vertex of G' is connected to at most $p-2$ vertices of C (since if it were connected to all the vertices of C a p-clique would have been formed). Thus $k \leq (p-2)(n-p+1)$. Therefore the number of edges m of G is

$$m \leq (p-1)(p-2)/2+(1-1/(p-1))(n-p+1)^2/2+(p-2)(n-p+1) \leq (1-1/(p-1))n^2/2.$$

If we solve this inequality for p we get

$$p-1 \geq \frac{n^2}{n^2-2m}$$

Therefore an equivalent formulation is that G has a $p-1$ clique of size at least $\frac{n^2}{n^2-2m}$. Now take the complementary graph (where an edge becomes a non-edge and a non-edge becomes an edge). The complement of G has $N=n$ vertices and $M=n(n-1)/2-m$ edges. From the latter we get that $2m=n(n-1)-2M=n^2-n-2M$. A $p-1$ clique in G becomes an independent set in its complement whose size is at least $\frac{n^2}{n^2-2m}=\frac{N^2}{N^2-2m}=\frac{N^2}{N^2-n^2+n+2M}=\frac{N^2}{N+2M}$, noting that $N=n$. This latter bound is the expression in the first version of Turan's theorem. An easy corollary is that a graph with n/k vertices, $k \geq 1$, and n edges has an independent set of size at least $n/(3k^2)$. ■

Exercise 541.

(L. G. Valiant). Computing the maximum of n keys requires at least $\lg\lg n$ parallel steps with $p \leq n$ processors.

Proof.

Preliminaries: parallel decision tree model. The lower bound is for the parallel version of the decision tree model where we allow p processors to work at any time step each one performing a single comparison between two keys. The decision tree model in the case $p = 1$ was used to prove the lower bound $\lceil \lg(n!) \rceil$ on the number of comparisons for comparison-based sorting. Note that this model deals with information gathering to compute the MAXIMUM. One also needs to process this information to derive the MAXIMUM. The model assumes that processing is **for free**. We can do so because we plan to establish a lower-bound (minimum possible running time) not to establish an algorithm for realizing this bound.

By induction. The lower bound is proved by what we call **an adversary argument** through induction. An **adversary** for this problem observes the actions of the algorithm and is allowed to choose the input keys by modifying their values in such a way so as to force the algorithm to run for at least $\lg\lg n$ steps, i.e. exhibit worst-case performance. These modifications should not invalidate, however, the operations of the algorithm already performed.

Rounds of algorithm-adversary interaction. The interaction between the adversary and the algorithm consists of a number of rounds. At the end of round $j-1$ the adversary can specify an input such that the maximum lies in a set of keys I_j of cardinality $n_j = |I_j|$ with the following (inductive step) properties.

(1) no two keys of I_j have been compared before,

(2) and

$$|I_j| = n_j \geq \frac{n_{j-1}^2}{n_{j-1}+2p} \geq \frac{n_{j-1}^2}{n_{j-1}+2n}.$$

Base case of inductive proof . For $i = 0$ let $n_0 = n$. Each one of the n input keys becomes part of set V_0 i.e. $|V_0| = n_0 = n$. Any parallel algorithm for finding the maximum could perform, in step $i = 0$, at most $p \leq n$ comparisons between the input keys that is, at most one comparison per processor. These comparisons are then mapped to the edges of a graph $G_0 = (V_0, E_0)$ that will be formed with vertex set V_0. The edge set E_0 will contain an edge joining two of the vertices of V_0 if the corresponding keys were compared in this step $i = 0$ by the parallel algorithm. G_0 has $n_0 = n$ vertices and at most $p \leq n$ edges and by Turan's theorem an independent set I_1 of size n_1, where $|I_1| = n_1$ and such that

$$|I_1| = n_1 \geq \frac{n_0^2}{n_0+2p} \geq \frac{n_0^2}{n_0+2n},$$

thus satisfying inductive step (2) property. Furthermore,

$$|I_1| = n_1 \geq \frac{n_0^2}{n_0+2p} \geq \frac{n_0^2}{n_0+2n}, \geq \frac{n^2}{n+2n} \geq \frac{n}{3} \geq \frac{n}{4} \geq \frac{n}{k},$$

where $k = 4$ is a more suitable choice than the obvious choice 3. Since the vertices of I_1 form an independent set the corresponding key values have never been compared before, thus satisfying the inductive step (1) property as well. Let $a_1, \ldots, a_{n_1}$ be the vertices of I_1. The adversary can then choose the keys in such a way that a_k, for any k, is larger than all of its neighbors in G_0. This is possible as a_k is not connected to any a_l (by the independent set property of the vertices of I_1). The actions of the adversary would force an optimal algorithm to work on the keys of I_1 next in step $i = 1$. This completes the base case.

Inductive step $j-1$ to j. In step $j-1$ a graph $G_{j-1} = (V_{j-1}, E_{j-1})$ was formed with $G_{j-1} = G_{j-2} \cap I_{j-1}$. Thus $V_{j-1} \subseteq I_{j-1}$, and by the inductive step (1) property, it is such that no two keys of I_{j-1} and thus of V_{j-1} have been compared before. Moreover the keys of V_{j-1} have values larger than the other input keys. Thus a parallel algorithm that is optimal in step $j-1$ will perform comparisons between these keys and E_{j-1} reflects the $p \leq n$ comparisons the parallel algorithm will perform in step $j-1$. At the completion of step $j-1$, an independent set I_j is identified as before with cardinality $n_j = |I_j|$ and let $x_{j_1}, \ldots, x_{j_{n_1}}$ be its vertices. An adversary can then

choose the keys x_i's in such a way that x_{j_k} is larger than all its neighbors. This is possible as x_{j_k} is not connected to any x_{j_l} (by the independent set property of the vertices of I_j and the properties of V_{j-1}). Set I_j satisfies the inductive step (1) property. By Turan's theorem,

$$|I_j| = n_j \geq \frac{n_{j-1}^2}{n_{j-1}+2p} \geq \frac{n_{j-1}^2}{n_{j-1}+2n}. \tag{14.1}$$

thus satisfying the inductive step (2) property The inductive step is complete.
We can then solve Eq.(14.1) that holds for for $j \geq 1$ and $n_0 = n$. By using the iteration recurrence solution method, we can show the following.

$$|I_j| = n_j \geq \frac{n}{4^{2^{j-1}-1}k^{2^{j-1}}}, \tag{14.2}$$

Moreover $k = 4$. Substituting into Eq.(14.2) for $k = 4$ we have the following.

$$|I_j| = n_j \geq \frac{n}{4^{2^j-1}} \geq \frac{n}{4^{2^j}}, \tag{14.3}$$

The algorithm terminates when $|I_j| = n_j \leq 1$.

$$\begin{aligned} \frac{n}{4^{2^j}} &\leq 1 \\ n &\leq 4^{2^j} \\ \lg n &\leq 2 \cdot 2^j \\ \lg n/2 &\leq 2^j \\ \lg\lg n - 1 &\leq j. \end{aligned}$$

which implies that for a j such that $j \geq \lg\lg n$, the comparison steps is $\Omega(\lg\lg n)$, as needed. Then at the end of step j an adversary can specify an input such that the maximum is the key value of I_j. ■

Exercise 542.

Show that in order to find the max of n keys in two rounds of parallel comparisons, we need $\Omega(n^{4/3})$ processors.

Proof.
We use Valiant's lower bound argument.
Let us have n keys whose maximum MAX_n we must establish in two rounds using p processors. Let us assume that $p \geq n$, since otherwise we can use extra processors for free. The number of parallel comparisons that can be performed in the first round is p. We form a graph with vertices the keys and edges the comparisons established. In the so formed graph there is an independendent set of size at least $P = n^2/(2p+n)$ by Turan's theorem. We can force a second round on the algorithm by fixing these P keys so that they become candidates for the maximum. In the second round concentrate on these P keys only. A number of p parallel comparisons can determine the Maximum of the P keys (and of course of the original n keys) as long as $p \geq P^2 \geq \binom{P}{2}$ (we use the "CRCW" or the 1-round algorithm). We thus have

$$\begin{aligned}
p &\geq P^2 \geq \binom{P}{2} \\
p &\geq \left(\frac{n^2}{2p+n}\right)^2 \\
p(2p+n)^2 &\geq n^4 \\
p(2p+p)^2 \geq p(2p+n)^2 &\geq n^4 \\
p(2p+p)^2 &\geq n^4 \\
9p^3 &\geq n^4 \\
p &= \Omega\left(n^{4/3}\right).
\end{aligned}$$

■

Exercise 543.

Find the n most significant bits of the quotient $1/y$ of an n-bit integer y on an n-cell linear array in $O(n)$ steps.

Proof.
The algorithm is based on Newton's iteration formula and computes the n bits of the reciprocal of an n bit integer on an n-cell linear array in $O(n)$ steps.
Let $f(x)$ be a polynomial with derivative $f'(x)$ and let x_i be an i-th approximation to a root of f. A better approximation x_{i+1} can be obtained by the following formula.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

If $1/y$ is to be computed, then $f(x) = 1 - xy$ and thus $f'(x) = -y$ and therefore the iterative formula becomes.

$$x_{i+1} = x_i - \frac{1 - yx_i}{-y} = x_i + \frac{1}{y}(1 - yx_i).$$

This formula poses a problem. It is expressed in terms of $1/y$ but this is the quotient that needs to be computed. Replacing $1/y$ by the current best approximation, namely x_i, resolves this issue and the following iterative formula is thus obtained.
Method 1 (not optimal). Use the formula,

$$x_{i+1} = x_i + x_i(1 - yx_i) = 2x_i - yx_i^2,$$

making sure that during initialization the following two operations are peformed.

(a) Rescale y so that $1/2 \leq y \leq 1$. This is done by shifting (the number of shifts is recorded).

(b) Choose an appropriate initial value for x_0. Choose $x_0 = 3/2$ in decimal ($x_0 = 1.1$ in binary). The choice will become apparent at the end of the discussion (it makes sense to choose the best value at the very end of the calculations).

It is shown that successive approximations are getting closer to $1/y$ with the following claim established.
Claim 1. For any i, $|\, 1/y - x_i \,| \leq 2^{-2^i}$, and an n-bit approximation requires, $T = O(n \lg n)$, $P = n$ on a linear array (bit model).
Proof of Claim 1. By induction on i. As $1/2 \leq y \leq 1$, $1 \leq 1/y \leq 2$. As $x_0 = 1.5$ the base of the induction ($i = 0$) follows ($|\, 1/y - x_0 \,| \leq 1/2$). Let $\varepsilon_i = 1/y - x_i$. By the inductive hypothesis let $|\, \varepsilon_i \,| \leq 2^{-2^i}$. Then, $x_{i+1} = 2x_i - yx_i^2 = 1/y - y\varepsilon_i^2$, which implies that $1/y - x_{i+1} = \varepsilon_{i+1} = y\varepsilon_i^2 \leq 2^{-2^{i+1}}$. This means that n bits of $1/y$ can be obtained with $\lg n$ iterations of the approximation formula. Each iterative step requires a number of multiplications, additions and subtractions which take $O(n)$ steps on an n-cell linear array (bit model).
This approach is not optimal: the reciprocal calculation would require $T = O(n \lg n)$, $P = n$ on a linear array (bit model).
Method 2 (optimal). An improvement is possible if at iteration i instead of using n-bit arithmetic operations, $O(2^i)$-bit arithmetic operations are utilized. Therefore the time of iteration i will be $O(2^i)$ instead of $O(n)$ and total time would be $O(n)$ steps ($\sum_i O(2^i)$) instead of $O(n \lg n)$ steps ($\sum_i O(n)$). The extended algorithm works as follows.
Rule 1. Use the iterative formula

$$x_{i+1} = 2x_i - y_i x_i^2,$$

where y_i is y rounded down to the $2^{i+1} + 4$ bit position (the choice will become apparent at the end of the discussion below).
Rule 2. Maintain $2^{i+1} + 1$ bits of precision of x_{i+1} by rounding up.
Claim 2. For any i, $|\, 1/y - x_i \,| \leq 2^{-2^i}$, and an n-bit approximation requires, $T = O(n)$, $P = n$ on a linear array (bit model).

Proof of Claim 2. From $x_{i+1} = 2x_i - y_i x_i^2$, $1/2 \leq y < 1$ and $1 < y \leq 2$. Let $\varepsilon_i = 1/y - x_i$. It will be shown by induction that $\mid \varepsilon_i \mid \leq 2^{-2^i}$. The base case is immediate (as in previous algorithm). Let the bound be try for any index up to and including i. it suffices to be proven for $i+1$. It is the case that $1/y - x_{i+1} = y_i\varepsilon_i^2 - \frac{y-y_i}{y^2}(1-2\varepsilon_i y)$. First, x_{i+1} is rounded up to the $2^{i+1}+1$ bit of precision (i.e. this adds at most $2^{-2^{i+1}-1}$ to the current value). Therefore $\mid 1/y - x_{i+1} \mid = \max\{y_i\varepsilon_i^2, \frac{y-y_i}{y^2}(1+2 \mid \varepsilon_i \mid y) + 2^{-2^{i+1}-1}\}$. As $y_i \leq y$, $y_i\varepsilon_i^2 \leq 2^{-2^{i+1}}$. It is also the case that $\frac{y-y_i}{y^2}(1+2 \mid \varepsilon_i \mid y) + 2^{-2^{i+1}-1} \leq 2^{-2^{i+1}-4}(2^2)(1+2(1/2)1) + 2^{-2^{i+1}-1} = 2^{-2^{i+1}}$. Therefore, it is the case that $\mid \varepsilon_{i+1} \mid \leq 2^{-2^{i+1}}$ and the same error bound holds for the new approximation formula. For the description of the algorithm to be complete information related to the shifted $1/y$ needs to be maintained as well as the number of bits of approximation used in each iteration (eg for the approximation y_i of y). ■

Exercise 544.

Find the n most significant bit of the normalized quotient of x/y in a linear array with $O(n)$ processors.

Proof. Division x/y can be reduced to the problem of finding the n msb's of the reciprocal $1/y$ of y, and then realizing $x*(1/y)$.

A Newton method-based solution for division would yield an algorithm that requires $\lg n$ iterations to find the n most significant bits of the normalized quotient of x/y. Each iteration requires a number of multiplications, additions and subtractions, i.e. $O(\lg n)$ time steps for a total of $O(\lg^2 n)$. Below we summarize a method that is more efficient (asymptotically). It is, however, impractical to implement.

Step 1: Normalize Y. First normalize $Y = y \cdot 2^r$ so that $1/2 \leq y \leq 1$ by shifting (its binary representation). Each such shift corresponds to a division by 2. Let $y = 1-\varepsilon$, where $0 \leq \varepsilon \leq 1/2$. Then

$$\frac{1}{Y} = 2^{-r}\frac{1}{y} = 2^{-r}\frac{1}{1-\varepsilon} = 1+\varepsilon+\varepsilon^2+\ldots$$

Let $X_i = 1+\varepsilon+\varepsilon^2+\ldots+\varepsilon^i$. Observe then that

$$|1/y - X_i| = \varepsilon^{i+1}+\ldots \leq 1/2^i.$$

Therefore X_n provides an n-bit approximation of $1/y$.

Given all ε^i, $i \leq n$, X_n can be computed in $O(\lg n)$ step by CSA (Carry-Save-Addition) in $O(\lg n)$ steps. Note that $n+\lg n$ bits of ε^i must be available, as by summing n such numbers $\lg n$ bits of precision will be lost (due to an increase in the number of most significant bits).

Objective 1. Calculate $(n+\lg n)$ bits (of precision) for ε^j for all $1 \leq j \leq n$ in $\Theta(\lg n)$ steps using $\Theta(n^c)$ processors, for some constant $c > 1$ (and preferably $c < 3$). In order to achieve Objective 1 an easier Objective 2 will be sought.

Objective 2. For any $j \leq n$, calculate $(n+\lg n)$ bits for ε^j, where $0 < \varepsilon < 1/2$, in $O(\lg n)$ steps. How do we calculate ε^j? A Naive approach is Objective 3.

Objective 3. Calculate ε^j with repeated squaring in $O(\lg^2 n)$ time steps.

■

Exercise 545.

Show how to perform a parallel prefix operation on the n nodes of a wrapped $\log n$-dimensional butterfly in $\lg n$ steps (instead of the usual $2\lg n$ steps).

Proof.

We shall describe the algorithm in terms of a non-wrapped butterfly. The input x_i to the prefix problem will be input to output row i and the computation will proceed from the back to the front i.e. from level $\lg N+1$ to level 0. In fact the input will be a pair (x_i, x_i). What is sent in the beginning along a cross edge is x_i (a value, not a pair). A processor is fed through the level edge pair (a,b) and through the cross edge value c. We distinguish two major cases. (i) the cross edge comes from below. Then the output on the level edge is (a,bc) and on the cross edge bc. (ii) the cross edge comes from above. Then the output on the level edge is (ac,bc) and on the cross edge bc. The result of the parallel prefix is the left member of the pair on a processor of level 0 after $\log n$ steps (in each step at most two multiplications are performed). The proof of correctness is by induction noting that the removal of nodes of level 0 splits the butterfly into two of half the size, and that each node of level i computes in the right member of the pair the combined "product" of $2^{\lg N-i}$ inputs of the rows of the butterfly of size $2^{\lg N-i}$ it belongs to. We shall describe the algorithm in terms of a non-wrapped butterfly. The input x_i to the prefix problem will be input to output row i and the computation will proceed from the back to the front i.e. from level $\lg N+1$ to level 0. In fact the input will be a pair (x_i, x_i). What is sent in the beginning along a cross edge is x_i (a value, not a pair). ■

Exercise 546.
Find the MAX of n comparable keys with n processors in $O(\lg\lg n)$ time.

Proof.
Consider n keys. Split them into $n/3$ groups of 3 keys each. For each group we can detemine the MAX in $3(3-1)/2 = 3$ comparisons using 3 procs per group. Total number of processors used is $n/3 \cdot 3 = n$. Thus we are left with determining the max of $n/3$ keys. Take the $n/3$ Maxima, and split them into groups of 7. We have $n/(3 \cdot 7)$ groups of 7 keys. Each group requires $7(7-1)/2 = 21$ processors to find the MAX of the group. Total number of processors used is $n/21 \cdot 21 = n$, and we can afford to use that many processors. Thus after the second second it suffices to find the max of $n/21$ keys to determine the MAX of the original n keys. How do we split the $n/21$ keys next? What is the pattern? Say that at some point we end up having n/s keys. We split them into groups of t keys so that $t(t-1)/2$ comparisons/processors are assigned per group to determine the group maximum in one step(round). This would give a total of assigned processors equal to $(n/(st))t(t-1)/2$ which must be at most n (number of available processors). For this to hold, if we do the math we end up with $t = 2s+1$. Thus we should split the n/s keys into $n/(s(2s+1))$ groups of $2s+1$ keys. Or for $n/21$ keys, into groups of 43 keys in the following iteration. What is the pattern? The denominator s in the next step become $s(2s+1)$ or at least $2^1 \cdot s^{2^1}$. In the next step, it becomes $2(2s^2)^2 = 2^{1+2}s^{2^2}$. In the following step, it becomes $2 \cdot 2^{2+4}s^8 = 2^{1+2+4}s^{2^3}$. In the i-th step it will become $2^{1+2+2^2+\ldots+2^{i-1}}s^{2^i} = 2^{2^i-1}s^{2^i}$. This denominator becomes greather than n, i.e. the number of groups is 1 or less and thus MAX can be determined uniquely for i about $\lg\lg n$. ■

Exercise 547.

Show how to compute the first n bits of the square root of an n-bit number in $O(n)$ steps on an n-cell linear array of bit processors. Give a high-level description of the algorithm but make sure you prove that it works as claimed. If an iterative algorithm is used, make sure you also consider the problem of the initial approximation.

Proof.
The algorithm is similar to the reciprocal algorithm and most of the details (such as running time, number of operations) are omitted in this outline. We only show the error bounds and the approximations used. Suppose we need to find the square root of y. We first shift y to scale it so that $1/2 \leq y < 1$. Note that rescaling at the end may require the knowledge of $\sqrt{2}$ (if number of shifts is odd) in order to compute $\sqrt{y}$. We compute $\sqrt{2}$ only once using our algorithm (an initial two-shift, an even number, may be required) and thereafter the result is used as a constant. Choose $f(x) = x^2 - y$, $f'(x) = 2x$. Then by Newton's formula we have $X_{i+1} = (1/(2X_i))(X_i^2 + y)$. We use instead the formula $x_{i+1} = g_i(x_i^2 + y_i)$, where g_i is the $(2^{i+1} + a)$-bit approximation (we set $a = 3$ later) of $1/(2x_i)$, i.e. $1/(2x_i) - 2^{-2^{i+1}-a} < g_i < 1/(2x_i)$. This is done utilizing the algorithm presented in class for the reciprocal. Similarly, y_i is the $(2^{i+1} + b)$-bit approximation (we set $b = 2$) of y, i.e. $y - 2^{-2^{i+1}-b} < y_i < y$. Let $x_i - \sqrt{y} = \varepsilon_i$. We prove by induction that $|\varepsilon_i| \leq 2^{-2^i}$, i.e. in $\lg n$ iterative rounds n bits of $\sqrt{y}$ can be obtained utilizing only $O(2^i)$-bit arithmetic in round i, for a total of $O(n)$ in the number of bit steps. The base case of induction is trivial, provided $1 \geq x_0 \geq \sqrt{y}$ (initial choice for x_0). Then, by the fact that $y_i \leq y$,

$$x_{i+1} - \sqrt{y} \quad = \quad g_i(x_i^2 + y_i) - \sqrt{y} \leq g_i\varepsilon_i^2 + \sqrt{y}(2g_ix_i - 1).$$

As $2g_ix_i - 1 \leq 0$ and $y \leq 1$,

$$\begin{aligned} |x_{i+1} - \sqrt{y}| &\leq \max\{g_i\varepsilon_i^2, \sqrt{y}|2g_ix_i - 1|\} \\ &\leq \max\{g_i 2^{-2^{i+1}}, 2^{-2^{i+1}-a}\} \end{aligned}$$

by the inductive hypothesis. It suffices to show that $g_i \leq 1$ and the inductive proof will be complete after setting $a = 3$ (first term will dominate the maximum). We first show that $x_{i+1} \leq 1$.

$$x_{i+1} = g_i(x_i^2 + y_i) \quad \leq \quad (1/(2x_i))(x_i^2 + y).$$

The right hand side is the original recurrence, $X_{i+1} = (1/2X_i)(X_i^2 + y)$, with $X_0 = x_0$, and $\sqrt{y} \leq x_0 \leq 1$. It can be proven by induction on i that $\sqrt{y} \leq X_i \leq 1$, for all i. Then $x_{i+1} \leq X_{i+1} \leq 1$ as required. We now derive a lower bound for x_{i+1}.

$$\begin{aligned} x_{i+1} = g_i(x_i^2 + y_i) &\geq (1/(2x_i) - 2^{-2^{i+1}-a})(x_i^2 + y_i) \\ &\geq (x_i/2 + y_i/2x_i) - 2^{-2^{i+1}-a}(x_i^2 + y_i) \\ &\geq \sqrt{y_i} - 2^{-2^{i+1}-a+1}, \end{aligned}$$

for all $i \geq 0$, as $x_i \leq 1$, $y_i \leq y \leq 1$ and $(x_i/2 + y_i/2x_i) \geq \sqrt{y_i}$. Therefore $x_{i+1} \geq \sqrt{y_i} - 2^{-4}$, if we fix $i = 0$ and $a = 3$. We then show that $\sqrt{y_i} \geq \sqrt{y} - 2^{-4}$. This, combined with the previous result, would yield $x_{i+1} \geq \sqrt{y} - 2^{-3}$. We use the fact that $y - 2^{-2^{i+1}-b} < y_i$ and $y \geq 1/2$, and show that for $b = 2$, $y - 2^{-2-b} > y + 2^{-8} - 2\,2^{-4}/\sqrt{2} > (\sqrt{y} - 2^{-4})^2$, which is immediate. Given that $x_{i+1} \geq \sqrt{y} - 2^{-3}$ and $y \geq 1/2$ we get that $x_{i+1} \geq \sqrt{2} - 1/8 \geq 1/2$, for all i. Then, $g_i \leq 1/(2x_i) \leq 1$ and the induction is completed by proving that $|\varepsilon_{i+1}| \leq 2^{-2^{i+1}}$. ■

Exercise 548.

Given an N-row butterfly, with a packets per level-0 processor, in the average case (random routing) of the greedy algorithm, the congestion at every node will be less than $C = \Theta(a) + o(\lg N)$, with high probability. With high probability, the running time T of the algorithm is

$$T = O(C) + \lg N + o(\lg N).$$

Proof.
Consider a level i node u. It is reachable from 2^i nodes of level 0 and therefore it is the intermediate node of $a2^i$ input packets if these packets are destined to any of the $2^{\lg N - i}$ nodes of level $\lg N$ that are reachable from this node. For a given packet that may go through u, the probability that one of the $2^{\lg N - i}$ destination nodes is chosen among $2^{\lg N} = N$ is $2^{\lg N - i}/2^{\lg N} = 1/2^i$. The number of packets that may go through u is $a2^i$. Therefore the probability that at least r of these packets cross u is given by the following expression.

$$Pr(\textit{at least r crossing } u) \quad = \quad \binom{a2^i}{r}(2^{-i})^r \leq (\frac{ae}{r})^r$$

$$Pr(\textit{at least r packets crossing at least one of } N\lg N \textit{ nodes}) \quad \leq \quad (N\lg N)(\frac{ae}{r})^r = P$$

We distinguish two cases:
Case 1: If $a \geq \lg N/2$ then choose $r = 2ea = \Theta(a)$. Then, $P \leq 1/N^{3/2}$.
Case 2: If $a \leq \lg N/2$ then choose $r = 2e\lg N/\lg(\lg(N/a))$. Then, $P \leq 1/N^{3/2}$.
Therefore, the probability of success (ie. node congestion at most r on EVERY NODE) is given by the following expression

$$Pr(\textit{at most r packets crossing ANY vertex of } N\lg N) = 1 - Pr(\textit{at least r packets crossing some node}) \geq 1 - 1/N^{3/2}.$$

For the sake of an example if $N = 100$ this probability of success is at least $1 - 1/1000 = 99.9\%$. If $N = 10000$, it is at least 99.9999%. The bound on congestion C covers both cases and is larger than the value of C established in each case. The implicit assumption made is that the a packets per processor are initially ordered, i.e. we can refer to the 1st, 2nd, ..., a-th packet of some input (level 0) row i processor of the butterfly. The problem provides a bound on node congestion only. We can turn it into a bound on the running time T if we first explain how the greedy algorithm resolves the case where two packets attempt to exit a node through the same link simultaneously. We resolve this case by using a random-rank protocol, that is, each packet is assigned a random priority key from an interval ($[1, O(a + \lg N)]$) and the packet with the minimum key crosses the link first. In case, and it is likely, that two packets have the same priority key, we assume the one with the lowest priority is the one originating from a lower indexed row. In case of a further tie, we use as a priority key the initial ordering of the packets per processor. One can prove an upper bound for the running time of the greedy algorithm. Although the average case analysis of the random routing guarantees, almost surely, acceptable performance on the average case it fails however to assure such performance in the worst case. In practice, there are input that suffer from much worse performance than the average one. Such input problems like transposition, bit-reversal are observed quite often in practice as the routing patterns of various problems. Therefore the guarantees offered by random routing (or average case arguments) are not very strong. Although the average case analysis of the random routing guarantees, almost surely, acceptable performance on the average case it fails however to assure such performance in the worst case. In practice, there are input that suffer from much worse performance than the average one. Such input problems like transposition, bit-reversal are observed quite often in practice as the routing patterns of various problems. Therefore the guarantees offered by random routing (or average case arguments) are not very strong.

We are interested in describing the behavior of a routing algorithm under worst case conditions. For this we are going to introduce randomized routing. Under randomized routing, no matter what the input problem is (eg a transposition or bit-reversal permutation or the "average" routing instance) the running time of the algorithm will

be within the claimed performance. I.e randomized algorithm works on any problem, EVEN the bad ones. From time to time, it may fail however to exhibit the claimed performance. This however would occur very rarely (i.e with very small probability) that is, on the average, perhaps once (one routing) in N^c runs of the algorithm. This would mean not that the routing instance was bad but that the choices made by the random number generator of the randomized algorithm were poor. One solution to such a case would be to rerun the algorithm on the same input instance by resetting the random number generator to some new seed value. We need only rerun the algorithm a couple of times before success is observed. The only disadvantage of using a randomized algorithm is that its running time is twice as much as that of the overage case of the deterministic algorithm. The question that arises is how we can turn a random routing problem using a deterministic (and greedy) algorithm into a randomized routing algorithm. ■

Exercise 549.

L. G. Valiant: Reduce randomized routing to two Random Routing Problems.

Routing Problem For any packet i, let the source and destination be described by the pair $(s(i),d(i))$. Realize routing described by the set of pairs $(s(i),d(i))$, $\forall i$.

You might encounter high congestion using that approach. Randomized routing can alleviate such problems. Describe Valiant' randomized routing paradigm.

Valiant's Randomized Algorithm (1982) **Routing Problem** For any packet i, let the source and destination be described by the pair $(s(i),d(i))$. Realize randomized routing described by the set of pairs $(s(i),d(i))$, $\forall i$.

Proof.

Valiant's method follows

Step 1. For every packet i, choose uniformly at random an intermediate destination $r(i)$.

Step 2. Route packet i from source $s(i)$ to intermediate destination $r(i)$ i.e. realize routing $(s(i),r(i))$.

Step 3. Route packet i from intermediate destination $r(i)$ to final destination $d(i)$ i.e. realize routing $(r(i),d(i))$.

The algorithm is randomized because in Step 1 random choices are made. Step 2 by itself is an instance of a random routing problem whose performance on the butterfly is well studied (Theorems 1 and 2). Step 3 is a reverse (mirror image) of a random routing problem and therefore its running time is that of Step2 as well. Overall, with high probability, Time (randomized routing) $=$ 2 T(random routing). The following theorem can then be derived: Given an N-row butterfly, with a packets per level-0 processor, there exists a randomized routing algorithm whose running time is, with high probability,

$$T' = O(a) + 2\lg N + o(\lg N).$$

Note that the congestion in this algorithm can be quite high $C = \Theta(a)$. In fact, we can route on the butterfly using constant-sized queues by using Ranade's algorithm: Given an N-row butterfly, with a packets per level-0 processor, there exists a randomized routing algorithm whose running time is, with high probability,

$$T' = O(a) + \lg N + o(\lg N),$$

The queues at each processor are of maximum size $Q \geq 1$. Note that maximum queue size per processor is independent of congestion C, whereas in the previous algorithms Q could grow as high as C. The constants hidden in the $O(.)$ and $o(.)$ notations are, however, larger. ■

Exercise 550.

A greedy **routing** algorithm is an algorithm that follows a shortest path. Given any routing problem on an N-row butterfly for which at most one packets originates from each level 0 processor and at most 1 packet ends in a $\lg N$-level processor, the greedy algorithm will route all the packets to their destination in $O(\sqrt{N})$ time.

Proof.
Let u be a node of row 0 and level i of the butterfly. Exactly 2^{i-1} input processors of level 0 lead to u and $2^{\lg N-i}$ output processors of level $\lg N$ are reachable from u. Let n_i be the number of greedy paths that traverse u. It is $n_i \leq 2^i$ and $n_i \leq 2^{\lg N-i}$. Therefore the total running time of the greedy algorithm will be

$$\begin{aligned} Totaltime &= \sum_i delays\ on\ each\ node\ of\ level\ i\ of\ the\ path\ from\ source\ to\ destination \\ &= \sum_i (n_i - 1) = \sum_{i=1}^{\lg N/2} 2^i + \sum_{i=\lg N/2+1}^{\lg N/2} 2^{\lg N-i} - \lg N = O(\sqrt{N}). \end{aligned}$$

The maximum queue size required at any processor is also $\Theta(\sqrt{N})$ (the case for $i = \lg N/2$). Conclusion: There exists a routing problem that requires at least $\Omega(\sqrt{N})$ steps. Examples off such problems are the following ones. **A transposition** is a permutation of the form: $\pi(u_1 \dots u_{\lg N/2} u_{\lg N/2+1} \dots u_{\lg N}) = u_{\lg N/2+1} \dots u_{\lg N} u_1 \dots u_{\lg N/2}$. **A bit-reversal** is a permutation of the form: $\pi(u_1 \dots u_{\lg N}) = u_{\lg N} \dots u_1$. Routing an 1-relation requires at most $N-1$ steps on an N-cell linear array (bi-directional links are available). This is because

- There is never contention on using the same edge in the same direction.
- Whenever a packet wants to move it moves.
- At no time are 2 packets at the same processor (unless one is already on its own destination).
- packet reaches destination at d steps if d is distance between source and destination.
- packet moves unless it has reached its destination.

Routing an 1-relation on a $\sqrt{N} \times \sqrt{N}$ 2d-array requires $2\sqrt{N}-2$ steps. For this, use greedy algorithm. Move a packet from (s_1, s_2) to (d_1, d_2) as follows.

1. Correct column first, i.e. move from (s_1, s_2) to (s_1, d_2).
2. Correct row then, i.e. move from (s_1, d_2) to (d_1, d_2).

Step 1. is the time for linear array ($\sqrt{N}$ independent routing problems). Routing is performed in $\sqrt{N}-1$ steps (at most). Step 2. is not symmetric; columns packets may pile up at a processor. Each processor has more than one source packets. **However each processor** is the destinati on of ONE AND ONLY ONE packet. Use the following Rule: Give priority to packets going farthest away. Under this rule, step 2 requires $\sqrt{N}-1$ steps. As a conclusion the routing can be realized in $2\sqrt{N}-2$ parallel steps. ■

Exercise 551.
An N-cell linear array (with wrap-around edges, a.k.a ring) is a *subgraph* of any N-vertex hypercube (i.e it contains a Hamiltonian path that traverses all vertices exactly once) for any $N \geq 4$.

Proof.
(by induction) Base case: $N = 4$ true by inspection. Assume inductive hypothesis is true. Take an N-vertex hypercube. Remove dimension n edges. It is split into two hypercubes of dimension $n-1$. By induction, construct two identical Hamiltonian paths/cycles for the two halves. Let $a_1 a_2 x \ldots y a_1$, $a'_1 a'_2 x' \ldots y' a'_1$ be the two paths/cycles. Then, construct the following cycle $a_1 a'_1 y' \ldots x' a'_2 a_2 x \ldots y a_1$. ■

Exercise 552.

An n-bit Gray code is an ordering of all n-bit binary numbers so that consecutive numbers differ in precisely one bit position.

(a) The $2^{d_1} \times 2^{d_2} \times \ldots \times 2^{d_r}$-cell r dimensional array is a subgraph of the $2^{d_1+\ldots+d_r}$-vertex hypercube.

(b) The $N-1$ vertex complete binary tree **cannot** be embedded in the N-vertex hypercube, $N \geq 8$.

Proof.

(a) (By construction.) Map each 2^{d_i}-cell array to a d_i-dimension hypercube. Let g_i be this mapping. Map cell $(i_1, i_2, \ldots i_r)$ of the array to a $d_1 + \ldots d_r$-long binary string $g(i_1)\ldots g(i_r)$, where $g(i_i)$ is a binary string of d_i bits.

(b) (By contradiction.) In an n-bit binary sequence the number of strings with an odd number of 1's is equal to the number of strings with an even number of 1's. Let us assume that such an embedding is possible. Let the root r of the tree (a level 0 vertex) be mapped to some vertex v of the hypercube. Let the parity of v be even (i.e. its string contains an even number of 1's). Adjacent vertices of v in the hypercube must have oppositive parity (ie. odd) as two vertices are adjacent if they differ in exactly one bit (if this bit is 1 in v it must be 0 in its neighbor, a decrease by one of the 1's, if it is 0 in v it must be 1 in its neightbor, an increase by one of the 1's, i.e. odd parity for the neighbor of v in any of the two cases). If r (level 0) is mapped to an even-parity vertex v, then level-1 vertices of the tree are mapped to odd-parity vertices in the hypercube (neighbors in the hypercube of even-parity vertices). Similarly level-2 vertices in the tree are mapped to even-parity vertices in the hypercube and so on. Let us consider the parity of the leaves of the tree. Let it be odd. Then so is the parity of their grandparents. On an $N-1$-vertex binary tree the number of leaves and their grandparents is $N/2+N/8$. These odd-parity vertices are mapped to odd parity vertices in the hypercude. Therefore the n-dimension hypercube contains at least $N/2+N/8 = 5N/8$ odd-parity vertices and at most $3N/8$ even parity vertices contradicting to the observation. ■

14.7 BSP

Exercise 553.

(a) In the BSP model, compare the performance of the naive algorithm (e.g. processor 0 sends message to $1,\ldots,p-1$) vs the tree $\lg p$ superstep algorithm.

(b) Give a formula that expresses the communication performance of the two-phase broadcasting algorithm (Alg. 4) that broadcasts n words of information, where $n > p$ (be sure to indicate number of rounds).

Proof.

(a) The performance of the first (say Algorithm 1) is given by $\max L,(p-1)g$ and of the second (call it Algorithm 2) by $\lg p\ \max L,g$. We distinguish three cases: (i) $L > (p-1)g > g$: Alg. 1 is more efficient than Alg. 2, (ii) $L < g$: Alg. 2 is the best of the two, and (iii) $g < L < (p-1)g$: Alg. 2 is best if $(p-1)/\lg p > L/g$, otherwise Alg. 1 is better.

(b) The two-phase broadcasting algorithm requires two rounds of communication. Each round requires communication time at most $\max L,gn$ (or more accurately $\max L,g(p-1)n/p$) for a total cost of $2\max L,gn$.

■

Exercise 554.

Present in detail an algorithm (Algorithm 3) for broadcasting one word of information, where the degree of replication per round is a variable k. Describe the communication performance of the algorithm in terms of L, g, p and k. (You should provide enough detail to show which processor transmits messages in every round, and the processors these messages are destined for, similarly to Algorithm 2 above. Feel free to make any reasonable assumptions related to the divisibility of the various parameters involved).

Proof.

Let k be the replication parameter. The broadcasting algorithm consists of $\lg p/\lg k$ rounds (we assume p is a power of k so that the number of rounds is an integer). In round $i = 0, 1, \ldots$, every processor j with index $j < k^i$ sends the message to $k-1$ distinct processors numbered $j + k^i \cdot l$, where $l = 1, \ldots k-1$. By induction, at the end of round i (the $(i+1)$-st round), the message is broadcast to $k^i \cdot (k-1) + k^i = k^{i+1}$ processors. The number of rounds required is the minimum integer r such that $k^r \geq p$, i.e. $r = \lg p/\lg k$ (or $\lceil \lg p/\lg k \rceil$, if $\lg p/\lg k$ is not an integer).

■

Exercise 555.

Let X_j, $1 \leq j \leq n$, be n vectors $X_j = (x_{1j}, \ldots x_{nj})$ whose components are of length one word. The prefices of vectors X_j, that is, all partial sums $X_1 + \ldots + X_i$, $1 \leq i \leq n$, need to be computed in parallel (operator $+$ is defined as the component-wise sum of two vectors). Present an optimal parallel prefix algorithm on $p < n$ processors (assume a fully connected network) and express its computation and communication performance (assume that the addition of two vector components – not vectors! – takes one step) in terms of the machine parameters L and g, processor size p and problem size n. [If in a communication round the maximum amount of computation performed by any processor is x steps, and at most m words/messages are transmitted by any processor, then the cost assigned to this round (computation and communication) is taken to be $\max\{L, x + gm\}$.]

Proof.
As the number of vectors is larger than the number of processors, n/p (assume n is multiple of p) vectors will be assigned to each processor. Let processor $0 \leq i \leq p-1$ hold vector X_k such that $\lfloor k/(n/p) \rfloor = i$ (assuming k runs from 0 through $n-1$). Processor i splits each vector X_k into p pieces of equal size (just as in the two-phase broadcasting algorithm), each piece consisting of n/p components of vector X_k. Let X_{jk} be the j-th piece of vector X_k stored in processor i and is itself a vector as well. In the first round of parallel prefix processor i sends X_{jk} to processor j. Processor j thus receives the j-th pieces of all n vectors. The cost of communication is $\max L, gn^2/p$. Each processor j thus computes the sequential prefices on the pieces of vectors it received. This operation requires at most n steps for each component for a total of n^2/p time, and all sums $Y_{jk} = \sum_{l=1}^{l=k} X_{jl}$, $0 \leq k < n$, are thus computed. This way all prefices related to the components that form the j-th piece of all vectors have been computed in processor j. Processor j then sends vector piece (on n/p components) Y_{jk}, to processor i such that $\lfloor k/(n/p) \rfloor = i$ which is the processor that sent X_{jk}. This communication is the inverse of the communication performed in the first round. The total time for this round (computation and communication) is $\max L, n^2(g+1)/p$. Finally processor i reconstructs the resultant vector Y_k from all pieces Y_{jk}, $0 \leq j < p$. Then $Y_k = \sum_{l=1}^{k} X_l$, as required. This final round is pure computation and takes an extra $\max L, n^2/p$.

■

Chapter 15

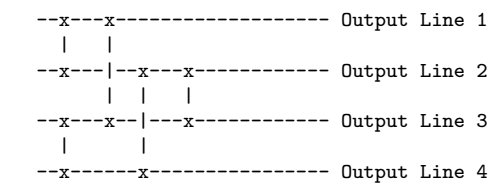
Sorting networks

15.1 Properties

Exercise 556.

Give a sorting network that sorts 4 keys with 5 comparators.

Proof.



Proof of correctness by inspection or the 0-1 Sorting Lemma. If the network looks similar to a merging-based one, it is not by coincidence! In the particular network if the last comparator is removed the remainder calculates the MIN-MAX of the 4 inputs in $3n/2-2$ comparisons. Min and Max are output through line 1 and line 4 respectively. This explains why the fifth comparator is needed to deal with the remaining two keys. ■

Exercise 557.

(a) Show that any sorting network must have $\Omega(\lg n)$ depth, and $\Omega(n \lg n)$ comparators.
(b) Then show that any soring network must have depth at least $\lg n$.
Note: In (a) we don't care about constants; in (b) we do care.
The idea for the first part is that if you can have better bounds for the sorting networks, then you can improve the performance of comparison-based sorting algorithms.
For the latter part it suffices to show that a $< \lg n$ depth sorting network can compute the maximum or the minimum faster than in $\lg n$ depth.

Proof.
(a) If a sorting network does not have $\Omega(\lg n)$ depth, i.e. it has $o(\lg n)$ depth, then given that $n/2$ comparisons can be done in parallel, the total number of comparators used would be $o(n(\lg n))$. Use (or simulate) the sequence of comparisons performed by the sorting network to derive a comparison-based sorting algorithm that performs $o(n \lg n)$ comparisons to sort n keys; this is impossible by the lower bound for comparison-based sorting. Therefore, any sorting network also requires $\Omega(n \lg n)$ comparators.
(b) Proof by contradiction. Suppose the sorting network has depth D less than $\lg n$. In a sorting network the n-th output wire carries the maximum. Since the depth $D < \lg n$, the maximum can be found in less than $\lg n$ time. We prove that this is impossible since one level of comparators at most doubles the number of distinct input wires whose values are considered for the maximum and in order for all n values to be considered the 2^D considered values after D levels must be at least n, i.e. $2^D \geq n$ which means that $D \geq \lceil \lg n \rceil \geq \lg n$. This contradicts the assumption that $D < \lg n$, and the claim is proved.
Another way to show this is as follows by induction on D. Base case: The MAX output wire of a single comparator considers the values of at most two input wires. True by default. Inductive Step: Walk leftward on output wire n until we hit a comparator. This is at level at most D. Since a comparator has two input wires the output wire n depends on the input values considered by the two input wires to that comparator. Each input wire to this D-level comparator is at depth at most $D-1$ and thus by the inductive hypothesis dependent on at most 2^{D-1} inputs. Therefore the MAXIMUM value on the n-th output wire is dependent on at most $2^{D-1} + 2^{D-1} = 2^D$ input values. ■

15.2 Sorting networks

Exercise 558.

The odd-even transposition sorting algorithm for sorting n keys has n rounds. In an odd round the keys at odd-indexed position get compared to the even-indexed positioned keys to their right and swapped if greater than them.

In an even round the keys at even-indexed position get compared to the odd-indexed positioned keys to their right and swapped if greater than them.

Show that the sorter sorts.

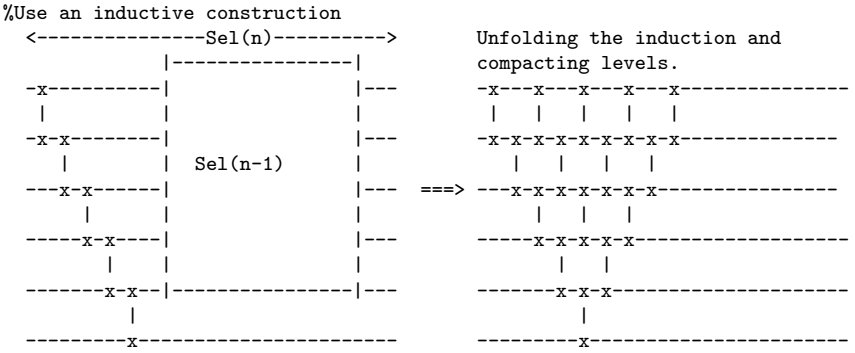
Proof.

Use the 0-1 Sorting Lemma. Let us say that there are t ones. The rightmost one if it is in odd position, will start moving immediately and keep on moving until it reaches its destination. If it is in an even position it will start moving with the next round. It will need at most $n-t+1$ rounds to beat and thus get swapped with potential zeroes since its earliest starting position is t and no less.. The 2-rnd rightmost one can be delayed once by the right-most one and once by it being in an even postion after the second round it will keep on moving towards index $n-1$, a total of $(n-1)-(t-1)+2=n-t+2$ moves. In general the i-th righmost key will move $n-t+i$ rounds. It is $i \leq t$ and thus within $n-t+i \leq n$ rounds all ones will be properly placed. ■

Exercise 559.

Max-based SelectionSort network. Provide details.

Proof.
The $n-1$ comparators of stage 1 find the maximum of n keys and the max is settled on wire n at the completion and tranferred to the output. For the remaining $n-1$ keys (the $n-1$ smallest of the input) we use a recursive/inductive construction in a similar way. If we unfold the recursion/induction we get the Figure on the right. The depth of the network is $2n-3$ as the longest path from an input to an output is the stair case path of stage 1 from line 1 to line n of length $n-1$ plus the upward moving reflection of length $n-2$ from line n to line 1 (we do not count the n to $n-1$ comparator twice but only once in both paths).
NOTE: For selection, the MAX of a number of keys is selected at every round of the algorithm. For Bubble-sort that is implemented and described in some other references leads to a network that is identical to the sorting network based on selection sort and also on insertion sort.

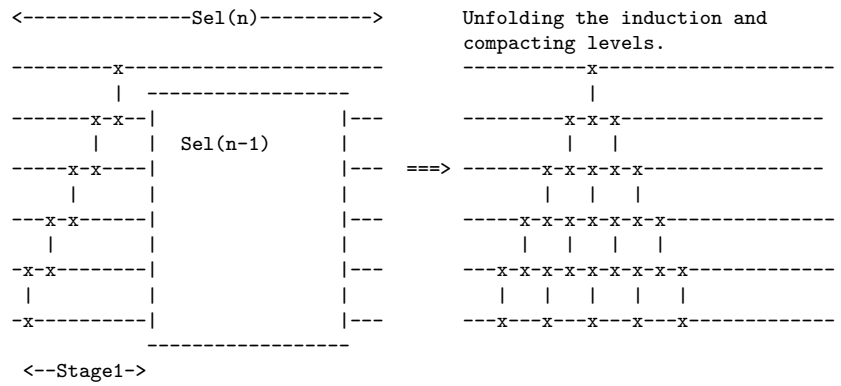


■

Exercise 560.

Min-based SelectionSort network. Provide details.

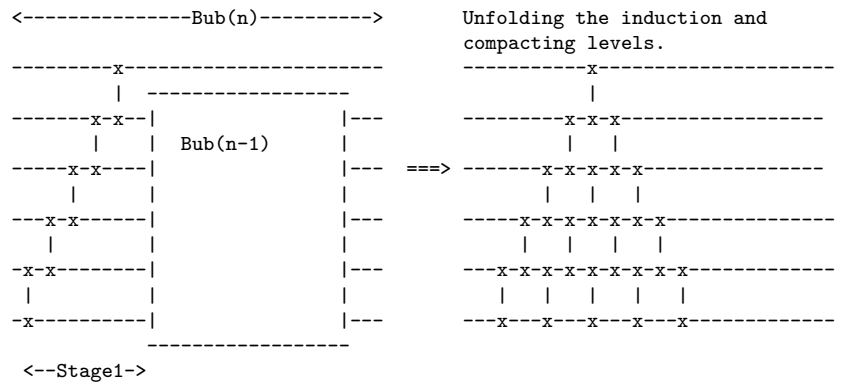
Proof.



The $n-1$ comparators of stage 1 find the minimum of n keys and the min is settled on wire 1 at the completion and tranferred to the output. For the remaining $n-1$ keys (the $n-1$ largest of the input) we use a recursive/inductive construction in a similar way. If we unfold the recursion/induction we get the Figure on the right. The depth of the network is $2n-3$ as the longest path from an input to an output is the stair case path of stage 1 from line n to line 1 of length $n-1$ plus the upward moving reflection of length $n-2$ from line 1 to line n (we do not count the 1 to 2 comparator twice but only once in both paths). ■

Exercise 561.
Bubble-sort network. Provide details.

Proof.



The $n-1$ comparators of stage 1 "bubble-up" the minimum of the n keys which is settled on wire 1 at the completion and tranferred to the output. For the remaining $n-1$ keys (the $n-1$ largest keys of the input) we use a recursive/inductive construction in a similar way. If we unfold the recursion/induction we get the Figure on the right that is the same as the selection-sort/insertion-sort based sorting network inverted. For the same reason as before the depth of the network is $2n-3$. ■

Exercise 562.

Show that an n-input sorting network must contain at least one comparator between the i-th and the $i+1$-st wire for all i such that $1 \leq i \leq n-1$.

Proof.

Proof by contradiction. Suppose that there is a sorting network T that has no comparator between wire m and wire $m+1$ for some m such that $1 \leq m \leq n-1$. We will prove that T CAN NOT BE a sorting network. We will prove this contradiction by using the 0-1 Sorting Lemma, on an input that is 0 for lines $1,\ldots,m-1$, is 1 for line m, 0 for line $m+1$, and 1 for lines $m+2,\ldots,n$. Just by inspection it is easy to verify that the only way the 1 of line m can move down and the 0 of line $m+1$ to move up is by having a comparator between the two since the following claims otherwise apply.

Claim 1. The 0's of lines $1,\ldots,m-1$ cannot move down and remain in position in the output.

Proof (Claim 1). Lines m and $m+2,\ldots,n$ have 1. Definitely any comparison between the lines that have zeroes and these lines can not move the zeroes down. The only other line that may facilitate a move-zero-down is line $m+1$ that has a zeo. A comparison between a zero and zero has the zeroes remain in their original position.

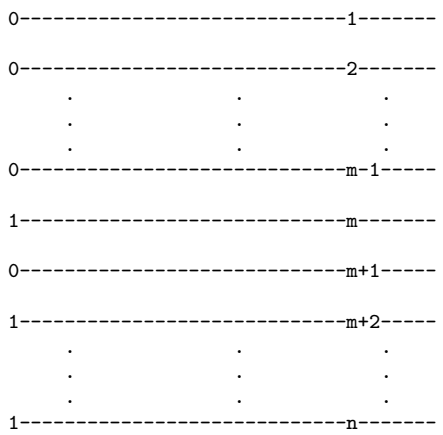
Claim 2. The 1's of lines $m+2,\ldots,n$ cannot move up and remain in position in the output.

Proof (Claim 2). Symmetric to Proof of Claim 1.

Claim 3. Lines $1,\ldots,m-1,m+1,\ldots,n$ remain unchanged in the output.

Proof of Claim 3. Because of Claims 1 and 2, the first $m-1$ zeroes remain in position in the output and the $n-m-1$ ones also remain in position in the output.

By claim 3, the only way the 1 of line m to move down is by comparing it to the 0 of lime m+1. There is no such comparator however. Therefore in the output the line m outputs an 1 before the zero of line $m+1$ and thus T CANNOT be a sorting network. Contradiction has been established.

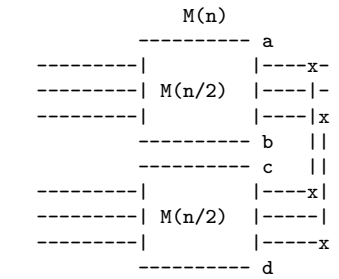


■

Exercise 563.

Let n be a power of two. Show how to construct an n-input and an n-output comparison network of depth $\lg n$ in which the top output wire always carries the minimum input value and the bottom output wire always carries the maximum input value. What is the size of the network in term of comparators? Given an exact answer. Show that it can be done in $o(n \lg n)$ comparators.

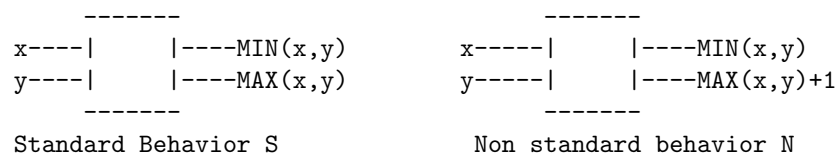
Proof.



$M(n)$ is the network that finds the MIN and MAX of n keys. It can be constructed recursively by first constructing a network $M(n/2)$ that finds the MIN and MAX of the first $n/2$ wires with the MIN coming out wire a and MAX coming out of wire b and then use another copy of $M(n/2)$ to find the MIN and MAX of the bottom $n/2$ wires. Then we establish the MIN of the n wires by comparing the two mins of line a and c and the MAX by comparing b and d with two comparators (shown). Total network depth is $D(n)$ for the $M(n)$ network and the recurrence $D(n) = D(n/2) + 1$, with boundary condition $D(2) = 1$ gives a solution $D(n) = \lg n$. The size of the network $S(n) = 2S(n/2) + 2$ giving as a solution $S(n) = 3n - 2$. The network corresponds to the divide and conquer method of finding the MIN and MAX of n keys simultaneously. ■

Exercise 564.

In a traditional comparison network every comparator behaves the standard way (let us call it S) that is outlined below. In addition to the standard behavior each comparator may behave as in the non-standard way N also outlined below. For every comparison network C we can flip a switch to decide whether every one of the comparators of C behaves the standard way or the non-standard way.



You are given a comparison network C enclosed in a black box and the only elements of the comparison network that are accessible by you are its input and output lines (i.e. you can set the input lines to specific inputs and read the outputs on the output lines but you can not see what the comparison network looks like) and a switch which you can set it to position S (standard behavior) or N (non-standard behavior).

Show how you can find the size (in terms of number of comparators) of network C even if you are not allowed to peek through the black box.

Proof.
Switch to mode N and assign 0 to all inputs. Observe the output lines, and sum the values of the output lines. The sum gives the size of the network. The proof is obvious. For input of zeroes, when two values are fed in any comparator the sum of the output lines of the comparator is 1 more than the sum of its input lines, i.e. each comparator adds 1 to the original input of zeroes. Therefore the sum of the values of the output lines gives the size of network C as each comparator contributes 1 to this sum.

■

15.3 Bitonic sorting

Exercise 565.

A(n alternative definition of) bitonic sequence is of the form

$$0^i 1^j 0^k,$$

where $n > i, j > 1$ or of the form

$$1^i 0^j 1^k,$$

where $n > i, j > 1$ or of the form

$$0^n$$

or of the form

$$1^n$$

(The last two 0^n and 1^n can be treated as special cases of the former two for $i = n,\ j = k = 0$ for example. This however requires the dropping of the conditions $n > i, j$ and $i, j > 1$.)

How many bitonic sequences are there (as defined)?

Proof.

```
0   1   2   3   4   5                   n-3         n
  x   x   x   x   x   ....   x   x   x   ....  x   x   x
```

Consider the first one $0^i 1^j 0^k$ with $i, j > 1$. We can place two separators A, B on any two of the $x's$ positions above: first A is placed anywhere and then B on the remaining x's between A and the rightmost x. Everything on the left of A will be 0 and associated with i, between A, B will be 1's and associated with j and on the right of B will be an 0's as well associated with k. How many choices do we have for A? Given that $i > 1$, we have $n-1$ choices for A/i from 1 to $n-1$, and $n-i$ choices for B/j from $i+1$ through n i.e. a total of $n-(i+1)+1 = n-i$. This also fixes the choices for k which will be n minus the value of j. Thus the number of $0^i 1^j 0^k$ sequences is

$$\sum_{i=0}^{n-1} (n-i) = n(n-1)/2.$$

Similarly the number of $1^i 0^j 1^k$ sequences is $\sum_{i=0}^{n-1}(n-i) = n(n-1)/2$. If we add the two sequences 0^n and 1^n we have $n(n-1)/2 + n(n-1)/2 + 2 = n^2 - n + 2$. ■

Exercise 566.

Find the exact solution of the recurrence representing the size of the sorting network based on bitonic sort

$$S(n) = 2S(n/2) + n\lg n$$

where $S(2) = 1$. Then find the exact solution of the following recurrence for the depth of such a network.

$$D(n) = D(n/2) + \lg n$$

where $D(2) = 1$. Use the iteration method and assume n is a power of two.

Proof.
We solve first $D(n) = D(n/2) + \lg n$ using the iteration method.

$$\begin{aligned} D(n) &= D(n/2) + \lg n \\ &= D(n/4) + \lg(n/2) + \lg n \\ &= D(n/2^i) + \lg(n/2^{i-1}) + \ldots + \lg n \end{aligned}$$

The boundary condition is $D(2) = 1$ and for $i = \lg n - 1$ we have $n/2^i = 2$ and thus $D(n/2^i) = 1$. Note that we can write 1 as $\lg 2 = \lg(n/2^{\lg n - 1})$ as well. Then we obtain.

$$\begin{aligned} D(n) &= D(n/2^i) + \lg(n/2^{i-1}) + \ldots + \lg n \\ &= 1 + \lg(n/2^{\lg n - 1 - 1}) + \ldots + \lg(n/2^0) \\ &= \lg(n/2^{\lg n - 1}) + \lg(n/2^{\lg n - 1 - 1}) + \ldots + \lg(n/2^0) \\ &= \lg n \cdot \lg n - ((\lg n - 1) + \ldots + 1 + 0) \\ &= \lg n \cdot \lg n - (\lg n - 1)\lg n/2 \\ &= (\lg n + 1)\lg n/2. \end{aligned}$$

Therefore $D(n) = (\lg n + 1)\lg n/2$.
For the recurrence $S(n) = 2S(n/2) + n\lg n/2$ we follow the same method as well.

$$\begin{aligned} S(n) &= 2S(n/2) + (n/2)\lg n \\ &= 2^2 S(n/2^2) + (n/2)\lg(n/2) + (n/2)\lg n \\ &= 2^i S(n/2^i) + (n/2)\lg(n/2^{i-1}) + \ldots (n/2) + \lg(n/2^0) \end{aligned}$$

The boundary condition is $S(2) = 1$ and for $i = \lg n - 1$ we have $n/2^i = 2$ and thus $2^i S(n/2^i) = 2^i S(2) = n/2$. Note that we can write $n/2$ as $(n/2)\lg 2 = (n/2)\lg(n/2^{\lg n - 1})$ as well.

$$\begin{aligned} S(n) &= 2^i S(n/2^i) + (n/2)\lg(n/2^{i-1}) + \ldots + (n/2)\lg(n/2^0) \\ &= n/2 + (n/2)\lg(n/2^{\lg n - 1 - 1}) + \ldots + (n/2)\lg(n/2^0) \\ &= (n/2)\lg n/2^{\lg n - 1} + (n/2)\lg(n/2^{\lg n - 1 - 1}) + \ldots + (n/2)\lg(n/2^0) \\ &= (n/2)\lg n \lg n - (n/2)((\lg n - 1) + \ldots + 1 + 0) \\ &= (n/2)\lg n \lg n - (n/2)(\lg n - 1)\lg n/2 \\ &= n(\lg n + 1)\lg n/4. \end{aligned}$$

■

Exercise 567.

(a) Let n be a power of 2. Give an expression for the number of comparators in a bitonic sorter (that sorts bitonic sequences) of n keys.
(b) Let n be a power of 2. Give an expression for the number of comparators of the bitonic-based sorter that sorts n general keys.
(c) Let n be a power of two. Find the depth of the network in (b).
Note. Give exact expressions. Do not use asymptotic notation.

Proof.
(a) Number of levels is $\lg n$ and number of comparators per level is $n/2$. Therefore $n\lg n/2$ is the answer. $S(n) = n\lg n/2$ follows from $S(n) = 2S(n/2) + n/2$ and $S(2) = 1$.
(b) Number of levels is $\sum_{i=1}^{\lg n} i = \lg n(\lg n + 1)/2$ and number of comparators per level is $n/2$ therefore the total is $n\lg n(\lg n + 1)/4$.
(c) Obvious.

■

Exercise 568.

Prove the bitonic-sort analog of the 0-1 sorting principle i.e. if a comparison network sorts any bitonic sequence of 0's and 1's it can then sort any bitonic sequence of arbitrary numbers.

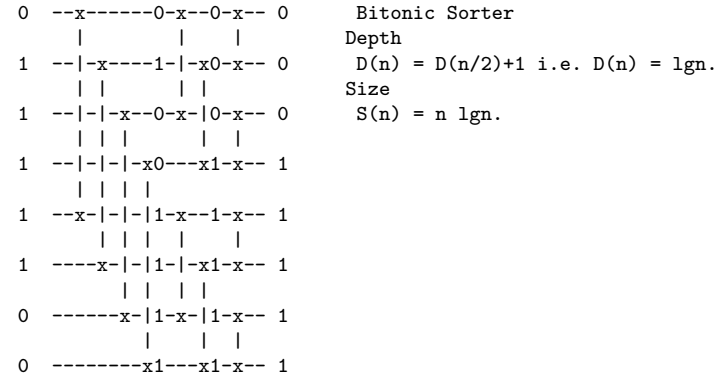
Proof.

The proof is identical to the 0-1 Sorting Lemma. The only thing we need to prove is that if $\langle a_1, \ldots, a_n \rangle$ is a bitonic sequence so is $\langle f(a_1), \ldots, f(a_n) \rangle$. If the latter input sequence is not bitonic nothing no contradiction can be proven as we don't know the behavior of the bitonic sorting network on non bitonic input sequences. First note that all 0-1 bitonic sequences have the form $0^i 1^j 0^k$ or $1^i 0^j 1^k$, for any $i \geq 0$, $j \geq 0$ and $k \geq 0$. Since $A = \langle a_1, \ldots, a_n \rangle$ is bitonic, then A is $\nearrow\searrow$ or a rotation of it. We prove below that $f(A)$ is bitonic. If A is a rotation then let the rotated sequence which is $\nearrow\searrow$ be $R = \langle a_t, \ldots, a_n, a_1, \ldots a_{t-1} \rangle$. We then prove that $f(R)$ by way of the previous case and a rotation of a bitonic sequence of zeroes and ones is a bitonic sequence as well. ■

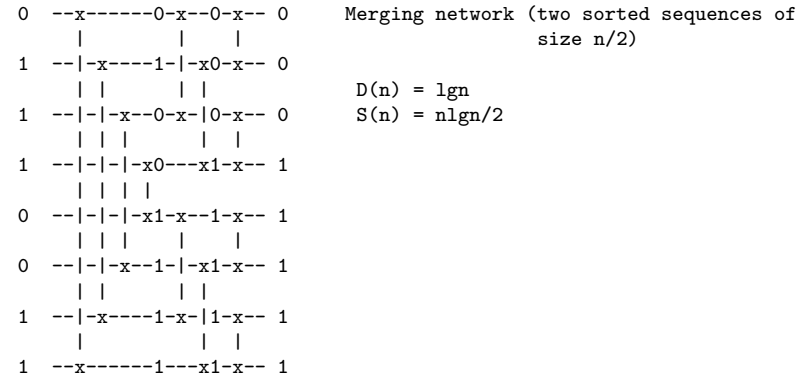
Exercise 569.

Modify a bitonic sorter to merge two sorted sequences. Provide time space information.

Proof.
Two sorted sequence can be turned into a sorted sequence followed by a the reverse of the second sorted sequence. The composition is a bitonic sequence. Use the bitonic sorter to merge it (aka sort it). That's a bitonic sorter



That is a merger of two sorted sequences.



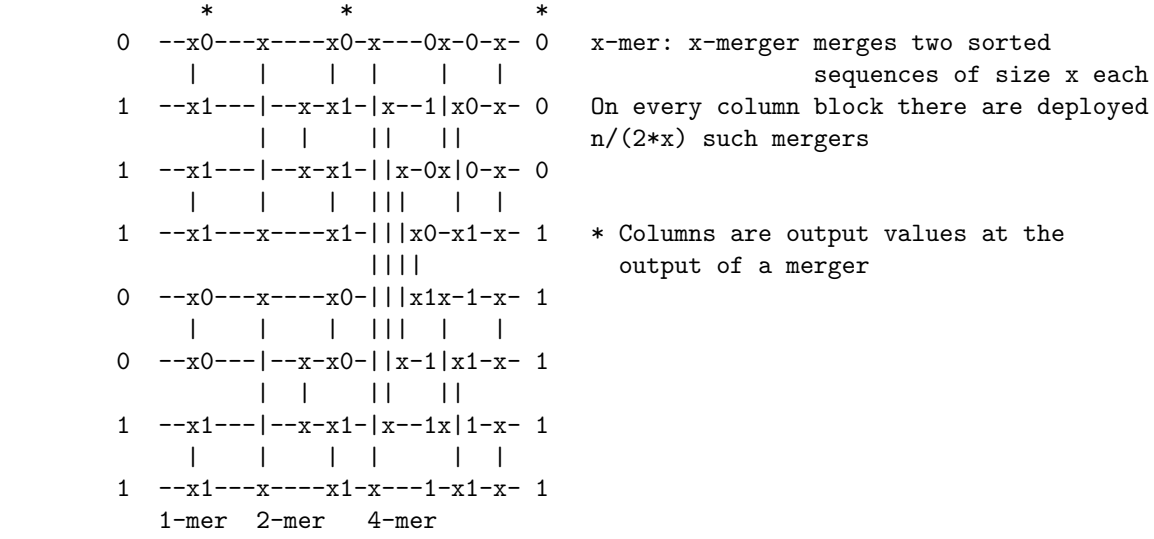
This will take $\lg n$ phases. Each phase consists of $n/2$ comparators. The depth of the network is $\lg n$ and its size $n \lg n/2$. $D(n) = \lg n$ and $S(n) = n \lg n/2$. ■

Exercise 570.

Use the previous result to sort arbitrary sequences. Derive $T(n)$, $S(n)$.

Proof.

Use MergeSort bottom up to merge sorted sequence of length 1, then length 2, and so on. Example below.



The depth and size of such a network can be found as follows.

$$D(n) = D(n/2) + \lg n$$

This gives $D(n) = O(\lg^2 n)$ by the master method and $\lg n(1+\lg n)/2$ by the iteration method.

$$S(n) = 2S(n/2) + O(n \lg n)$$

This gives $S(n) = O(n \lg^2 n)$ by the master method. ■

Exercise 571.

Provide pseudocode for Batcher's odd-even merge.

Proof.
The variant given here slightly differs from the original Batcher paper. In that paper Odd(A) with Odd(B) and

begin OddEvenMerge ($A = \langle a_0 \dots a_{m-1} \rangle$,$B = \langle b_0 \dots b_{m-1} \rangle$,m)
1a. Even(A) $= \langle a_0, a_2, \dots, a_{m-2} \rangle$;
1b. Odd(A) $= \langle a_1, a_3, \dots, a_{m-1} \rangle$;
1c. Even(B) $= \langle b_0, b_2, \dots, b_{m-2} \rangle$;
1d. Odd(B) $= \langle b_1, b_3, \dots, b_{m-1} \rangle$;
2a. C= OddEvenMerge(Even(A),Odd (B),m/2);
2b. D= OddEvenMerge(Odd (A),Even(B),m/2);
3. L_{pre} $= \langle c_0, d_0, c_1, d_1, c_2, d_2, \dots, c_{m-1}, d_{m-1} \rangle$;
// x:y means compare x and y and place so that the left of the two
// is the smallest of the two
4. L $= \langle c_0 : d_0, c_1 : d_1, c_2 : d_2, \dots, c_{m-1} : d_{m-1} \rangle$;
//Perform a comparison side-by-side between c_i and d_i.
5. Return(L);
end OddEvenMerge

Even(A) with Even(B) are merged. That however, would require the comparison of d_i with c_{i+1} in line 4 instead, with c_0 and d_{m-1} not participating in that step (left unchnaged). The proof of correctness is by the 0-1 sorting Lemma. Let A consists of a 0's and $m-a$ 1's, and B of b 0's and $m-b$ 1's. After Line1,
$odd(A)$ has $\lfloor a/2 \rfloor$ 0's, and
$even(A)$ has $\lceil a/2 \rceil$ 0's.
Similarly,
$odd(B)$ has $\lfloor b/2 \rfloor$ 0's, and
$even(B)$ has $\lceil b/2 \rceil$ 0's. Then, after Line2, C has
$c' = \lceil a/2 \rceil + \lfloor b/2 \rfloor$ 0's
and D has
$d' = \lceil b/2 \rceil + \lfloor a/2 \rfloor$ 0's,
where $|c' - d'| \leq 1$. Then, after Line3, we get
a. If $c = d$ or $c = d+1$,

$$L = L_{pre} \quad \underbrace{0 \quad 0 \quad \dots \quad 0}_{c+d \text{ 0's}} \quad 1 \quad 1 \quad 1 \quad 1$$

b. If $c = d-1$,

$$L = L_{pre} \quad \underbrace{0 \quad 0 \quad \dots \quad 0}_{2c \text{ 0's}} \quad 1 \quad \underbrace{0}_{\text{flip}} \quad 1 \quad 1$$

This completes the proof. ■

Exercise 572.

Use the previous merging method to sort.

Proof.
The running time of OddEvenMergeSort is, however, more complicated. Lines 1a and 1b are done in parallel.

```
begin OddEvenMergeSort (X = ⟨x_0 ... x_{n-1}⟩,n)
1a.     Left(A)  = ⟨x_0, x_1, ..., x_{n/2-1}⟩;
1b.     Right(A)= ⟨x_{n/2}, x_{n/2+1}, ..., x_{n-1}⟩;
2a.     L(A) = OddEvenMergeSort(Left (A),n/2);
2b.     R(A) = OddEvenMergeSort(Right(A),n/2);
3.      Y= OddEvenMerge(L(A),R(A),n/2);
5.      Return(Y);
end OddEvenMergeSort
```

Lines 2a and 2b are done in parallel. Line 3 is $T_{merge}(n/2)$.

$$\begin{aligned} T_{sort}(n) &= T_{sort}(n/2) + T_{merge}(n/2) \\ T_{sort}(n) &= O(\lg^2 n). \end{aligned}$$

A proof that OE merge-sort works utilizes the 0-1 sorting lemma. By induction let us assume that OE merge-sort works for sizes less than or equal to $n-1$. Therefore in order to sort n keys, we split them into 2 halves of size $n/2$ each. By the inductive hypothesis, o-e merge-sort sorts independently the two halves. It remains to be shown that the merging algorithm so described merges the two sorted sequences and the theorem is proved. ■

Part VII

Probability and randomness

Chapter 16

Probability review problems

16.1 Balls and bins

Exercise 573.
(a) Calculate $P(n,k)$ the number of permutations of n objects taken k at a time.
(b) Calculate $C(n,k)$ the number of combinations of n objects taken k at a time. Order in the set (of a combination) makes no difference.

Proof.
(a) We have n possibilities for the first element of the permutation, $n-1$ for the second, etc. Total is

$$P(n,k) = n(n-1)\dots(n-(k-1)) = \frac{n!}{(n-k)!}$$

(b) We have $P(n,k)$ possibilities but then need to adjust because order is not important by dividing with $k!$. Total is

$$C(n,k) = P(n,k)/k! = n(n-1)\dots(n-(k-1)/k! = \frac{n!}{(n-k)!k!} = \binom{n}{k}$$

■

Exercise 574.

Find the number of ways you can throw n balls into N bins for the following scenarios.
(a) Distinct balls into distinct bins.
(b) Indistinct balls into distinct bins.
(c) Indistinct balls into distinct bins with at least one ball per bin.
(d) Indistinct balls into distinct bins with at least k_i balls into bin i?
(e) Indistinct balls into distinct bins at most one ball per bin.
(f) Distinct balls into distinct bins at most one ball per bin.
(g) Distinct balls into indistinc bins at least one ball per bin.
(h) Distinct balls into indistict bins at most one ball per bin.
(i) Indistinct balls into indistict bins at most one ball per bin.

Proof.
(a) We describe the distribution as a n-digit sequence, each digit taking one of N values representing a bin's ID (e.g. from 1 to N). There are

$$\underbrace{N \times N \times \ldots \times \ldots N}_{\text{since there are } \ n \ \text{ digits}} = N^n$$

(b) This is equivalent to n_i the number of balls in bin i: $\sum_{i=1}^{N} n_i = n$. In other words, the number of solutions n_i of this equation in the non-negative integers is the answer to the original question. This is $\binom{n+N-1}{N-1} = \binom{n+N-1}{n}$. Think also $N+1$ vertical lines (inclusive of first and last). First and last are fixed, they can be ignored thus the position of the remaining $N-1$ varies. Total number is out of $n+N-1$ marked positions we pick $N-1$ and turn them into vertical line markers $\binom{N+n-1}{N-1}$. Between the markers we count balls per corresponding bin.
(c) Place one indistinct ball per bin. The remaining $n-N$ balls remain to be distributed into the N bins. By case (b) we have $\binom{n-1}{N-1}$. Equivalently, the n balls create $n-1$ gaps between them. Pick $k-1$ out of the $n-1$ gaps in $\binom{n-1}{N-1}$.
(d) Similar to (c). Put k_i balls as neede into bin i. The remaining balls are handled as in (b) above.

$$\binom{n-\sum_i k_i}{N-1}.$$

(e) Implied is that $N \geq n$. We need to choose which n of the N urns will be assigned a ball! Thus $\binom{N}{n}$.
(f) N choice for first ball, $N-1$ for second and so on. Total $N(N-1)\ldots(N-n+1)$. Note that if $N=n$, then the answer is $n!$. Moreover, $n > N$, answer is 0.
(g) That's a Stirling number $\left\{ \begin{matrix} N \\ n \end{matrix} \right\}$.
(h) Not many choices: if $n < N$ the answer is 1; otherwise it is zero!
(i) See the previous question. Not many choices: if $n < N$ the answer is 1; otherwise it is zero! ■

Exercise 575.

We throw n (indistinguishable) balls into N (indistinguishable) bins. Label them $1,\ldots,n$ and $1,\ldots,N$ respectively.
(a) What is the probability all balls fall into bin k? What is the probability that bin k is empty?
(b) What is the expected number of balls in bin k?
(c) What is the expected number of empty bins ?

Proof.
(a) All balls fall into bin k with probability $(1/N)^n$. Bin k remains empty with probability $(1-1/N)^n$.
(b) Let us use a random variable X_i is 1 if ball i goes to bin k, and 0 if it goes to another bin. Then $Y_k = \sum_{i=1} nX_i$ counts the number of balls into bin k. We have

$$E(X_i) = 1 \cdot 1/N + 0 \cdot (1-1/N) = 1/N.$$

Then using the linearity of expectation.

$$E(Y_k) = E(\sum_{i=1}^{n} X_i) = \sum_i E(X_i) = n \cdot 1/N = n/N.$$

(c) Let A_k be the event that bin k is empty. Then, $p(A_k)$ is the probability bin k is empty. Let B_k be a random variable that is 1 if bin k is empty and 0 otherwise. Then, $B = \sum_k B_k$ counts the number of empty bins. First,

$$P(A_k) = (1-1/N)^n$$

Then by linearity of expectation.

$$E(B) = E(\sum_k B_k) = \sum_k E(B_k)$$

Finally

$$E(B_k) = 1 \cdot P(A_k) + 0 \cdot (1 - P(A_k) = (1-1/N)^n$$

We conclude that

$$E(B) = \sum_k E(B_k) = N(1-1/N)^n.$$

The latter is $\approx Ne^{-n/N}$.
If $n = N$, it says that $1/e$ of the N bins are empty. If we throw n more balls $1/e \cdot 1/e \cdot N$ of the bins will be empty. How many times m do we need to repeat this so that $(1/e)^m N < 1$? Solving for m we find $m > \ln(N)$.
Let C be the event that there is an empty bin. Then we have the following.

$$P(C) = P(\exists eptybin) = P(\cup_{k=1}^{N} empty - bink) \leq \sum_k P(empty - bink) \leq N(1-1/N)^n.$$

We could have proved the same by Markov's inequality.

$$P(B \geq 1) \leq E(B)/1 = N(1-1/N)^n.$$

What is n so that $P(C)$ is w.h.p? Let us say we want $P(C) \leq 1/N$ or equivalently we want $Ne^{-n/N} \leq 1/n$. Set $n = 2N\ln N$. Then $Ne^{-n/N} \leq 1/N$ indeed. ■

Exercise 576.

We throw n (indistinguishable) balls into N (indistinguishable) bins. Label them $1,\ldots,n$ and $1,\ldots,N$ respectively. Continuing the previous problem for what value of n do we expect to see two balls in one bin, say bin k? This assumes there is one ball already into bin k.

Proof. The probability p of no second ball is as follows. We use $e^x \geq 1+x$ for all $x \in R$ in the form $e^{-x} \geq 1-x$.

$$1 \cdot (1-\frac{1}{N}) \cdot (1-\frac{2}{N}) \cdot (1-\frac{3}{N}) \ldots \cdot (1-\frac{n-1}{N}) \leq e^{-1/N} e^{-2/N} \ldots e^{-(n-1)/N} = e^{-n(n-1)/(2N)}$$

The birthday paradox says that if we bound this probability by $1/2$

$$e^{-n(n-1)/(2N)} \leq 1/2$$

the probability of a collision becomes $\geq 1/2$. Solving the equation above for n we have $n \approx \sqrt{2\ln(2)N}$. For a planet (Earth) with 365 days on a non-leap year this means that in a group of 23-24 people there would be two with same birthday (with probability at least 0.5). ■

Exercise 577.

Assume $n = N$. What is the probability that bin k has at least t balls in it?
We provide the following definition

Definition 16.1

We say an event E dependent on n occurs with high probability, if $P(E) \leq 1 - 1/n^c$ for some constant $c > 1$.

Instead of writing "with high probability" we shall write "w.h.p." or "whp" instead.

Proof. Our current scenario simplifies to n balls into n bins the fullest bin has $O(\lg n \ \lg\lg n)$ balls w.h.p.

Let bin k has t balls. The probability this is the case is $B(n,p;t)$. A reminder

$$B(n,p;t) = \binom{n}{t} p^t (1-p)^{n-t}.$$

By Eq.(I.1) with $t = \lceil upn \rceil$, $p = 1/n$ and $t = \lceil u \rceil$, $u > 1$. In Eq.(I.1) the $u/(u-1)$ becomes at most 1.5 for $u > 3$ or equivalently $t > 3$. Therefore for $t > 3$

$$P(Bin\ k \geq t) = P(S_n \geq t) \leq \frac{u}{u-1} B(n,p;t) \leq 1.5 \binom{n}{t} (1/n)^t. \tag{16.1}$$

This bound makes sense if we can put $(1-p)^{n-t}$ into use that we obviously did not do.

An alternative is to argue as follows (for bin k that is).

$$P(Bin\ k \geq t) \leq \sum_{S_n \subseteq \{1,\ldots,n\}, |S_n| = t} \prod_i P(\text{ball i } \in S_n \text{is in bin } k) \leq \binom{n}{t} p^t = \binom{n}{t} (1/n)^t \tag{16.2}$$

Note that in this set-up, we guarantee the balls of S_n fall into k, but there might be other balls falling into it; we do not use $(1-p)^{n-k}$ and thus this effectively become 1^{n-k} or the other balls are free agents: they can fall into k or not.

The difference between Eq.(16.1) and Eq.(16.2) is a 1.5. We proceed discarding 1.5 as follows.

Using inequality I.3.3 we obtain from Eq.(16.2) the following.

$$P(Bin\ k \geq t) \leq \left(\frac{ne}{tn}\right)^t \leq \left(\frac{e}{t}\right)^t.$$

Assum $n \geq e^2$ implying a $\ln n \geq 2$. Furthermore choose $t = c\ln(n)$, with $c = e^2$. Then if $t = e^2 \ln(n)$ we have the following. Note that $2c = 2e^2 \approx 14.778 > 14$

$$\begin{aligned}
P(Bin\ k \geq t) &\leq \left(\frac{e}{t}\right)^{c\ln n} \\
&\leq \left(\frac{e}{e^2 \ln n}\right)^{c\ln n} \\
&\leq \left(\frac{1}{e\ln n}\right)^{c\ln n} \\
&\leq \left(\frac{1}{e}\right)^{c\ln n} \cdot \left(\frac{1}{\ln n}\right)^{c\ln n} \\
&\leq \frac{1}{n^c} \cdot \frac{1}{n^c} \\
&\leq \frac{1}{n^{2c}} \leq \frac{1}{n^{14}}.
\end{aligned}$$

Then

$$\begin{aligned}
P(\text{number of balls any bin } \geq t) &= P((\text{number of balls of bin } 1 \geq t) \\
&\quad \cup (\text{number of balls of bin } 2 \geq t) \\
&\quad \cup \quad \ldots \\
&\quad \cup (\text{number of balls of bin k} \geq t) \\
&\quad \cup \quad \ldots \\
&\quad \cup (\text{number of balls of bin n} \geq t)) \\
&\leq \sum_i P((\text{number of balls of bin k} \geq t) \\
&= n \cdot P((\text{number of balls of bin k} \geq t) \\
&= n \cdot P((\text{Bin k} \geq t) \\
&\leq n \cdot \frac{1}{n^{14}} \\
&= \cdot \frac{1}{n^{13}},
\end{aligned}$$

since $c = e^2$. This means $t = e^2 \ln(n)$ is too generous. We can make it smaller.

The bound $t = c \ln n / \ln \ln n$ for $c = 2e^2$ gives. First let's bring our bound in an exponential form that we did not do so before.

$$\begin{aligned}
P(\text{number of balls any bin } \geq t) &\leq n \cdot P((\text{number of balls of bin k} \geq t) \\
&\leq n \cdot \frac{e^t}{t^t} \\
&\leq \exp(\ln n + t - t \ln t) \\
&\leq \frac{1}{n^6}.
\end{aligned}$$

The bound holds for $n > 4000000 > e^{e^{e^1}}$. This is to make sure $\ln \ln \ln n / \ln \ln n$ is small enough. Thus the result has been proven w.h.p i.e. that with probability at least $1 - 1/n^6$ every bit has no more than $2e^2 \ln n / \ln \ln n = \Theta(\ln n / \ln \ln n)$ balls. ■

Exercise 578.

If we throw n balls into n bins, what is,
a) the expected number of empty bins?
b) the expected number of bins with exactly one ball?
c) the expected number of bins with exactly two balls?

Proof.
Part (a) has been answered earlier for the general case. Set $N = n$ to obtain the answer in (a) below. For all the parts below, let X be a random variable that takes values 1 or 0 depending on whether a bin is empty (for part (a)), contains exactly one ball (for part (b)), or exactly two balls (for part (c)). Then let Y be a random variable that counts the number of empty bins for part (a), or the balls with exactly one/two ball(s) for parts (b) and (c) respectively. It is clear that $E(Y) = nE(X)$.
a) Let's consider one bin and call it for convenience only, bin 1. The probability that a ball falls in it is $1/n$ (and therefore the probability that a ball does not fall in it is $1-1/n$). The probability that none of the n balls falls into this bin is thus $(1-1/n)^n$. Then for the X variable we get that $E(X) = (1-1/n)^n$. Summing for the n bins we get that Y is such that: $E(Y) = n(1-1/n)^n \leq n/e$.
b) Again, for bin 1, we find the probability that exactly one ball falls into that bin. Among n balls, the probability that exactly one falls into bin 1 is equal to $\binom{n}{1}(1/n)(1-1/n)^{n-1} = (1-1/n)^{n-1}$. As in part (a), we get $E(X) = 1 \cdot Pr(X = 1) + 0 \cdot Pr(X = 0) = (1-1/n)^{n-1}$, and therefore for Y (that gives the number of bins with exactly one ball) we find that $E(Y) = n(1-1/n)^{n-1} = \frac{n^2}{n-1}(1-1/n)^n \leq \frac{n^2}{e(n-1)}$.
c) As in parts (a), (b), the probability that bin 1 has exactly two balls is now: $\binom{n}{2}(1/n)^2(1-1/n)^{n-2}$, and therefore the X (the random variable that take values 1 or 0 depending on whether a given bin has exactly two ball or not) has expectation $E(X) = \binom{n}{2}(1/n)^2(1-1/n)^{n-2}$. Then for Y, we get that $E(Y) = nE(X) = n\frac{n(n-1)}{2}\frac{1}{n^2}\frac{(1-1/n)^n}{(1-1/n)^2} \leq \frac{n^2}{2e(n-1)}$.

■

Exercise 579.

We have m balls and n bins.
(a) What is the probability that two balls fall into the same bin?
(b) What is the probability that bin 5 is empty? (Implicit in the statement above is that we have enough bins to allow for bin 5 to exist!)
(c) What is the expected number of (ball) collisions?
(d) What is the expected number of empty bins.
(e) What is the probability bin 5 has k balls? Show that it is $\leq 1/k!$ for $m = n$.
(f) What is the probability that there is an empty bin (whether it is 5 or some other bin or more than one)?
(g) What is the probability bin 5 has AT LEAST k balls?
(h) What is the probability that the maximum number of balls in a bin is at least k ?

Proof.
(a) Obviously it is $1/n$. Pick two arbitrary balls i and j. The throws are independent of each other. Let e_{ij} be the event that they are in the same bin.

$$P(e_{ij}) = \sum_k P(\text{Ball i into bin k, AND Ball j into bin k}) = \sum_k \frac{1}{n} \cdot \frac{1}{n} = \frac{n}{n^2} = 1/n.$$

(b) There is nothing specific about bin 5. We just want to stress that the question is for a specific bin. The possible outcomes is n^m. We can write down an m-digit sequence, and every 'digit' is a value from 1 to n indicating a bin number (that identifies a signle bin). The number of outcomes is n^m as each one of the m digits take n values. The event we are interested in is the one where the m-digit sequence contains no 5 at all. We have $(n-1)^m$ such sequences (all possible digits other than 5). Thus the probability of bin 5 being empty is $(n-1)^m$ divided by n^m i.e.

$$\frac{(n-1)^m}{n^m} = \left(1 - \frac{1}{n}\right)^m.$$

Note that $1 - 1/n \leq e^{-1/n}$ and therefore,

$$\left(1 - \frac{1}{n}\right)^m \leq e^{-m/n}.$$

If $n = m$ this is at most $1/e$. One also could prove the same using indicator random variables. Here let us define Y_5 being 1 if bin 5 is empty and 0 otherwise. Obviously

$$E(Y_5) = P(\text{bin 5 is empty}) = \prod_{j=1}^{m} (1 - P(\text{ball j fall into 5}) = \prod_{j=1}^{m} (1 - 1/n) = (1 - 1/n)^m$$

(c) This derives from (a). The probability of a collision $P(e_{ij})$ is $1/n$. Let X_{ij} be an indicator random variable that is 1 if there is a collision between i and j and 0 otherwise. Then

$$E(X_{ij}) = 1 \cdot P(e_{ij}) + 0 \cdot \text{Irrelevant} = P(e_{ij}).$$

The number of collisions X is given by

$$X = \sum_i \sum_{j, i \neq j} X_{ij}$$

Therefore,

$$E(X) = E(\sum_i \sum_{j, i \neq j} X_{ij}) = \sum_i \sum_{j, i \neq j} E(X_{ij}) = \frac{m(m-1)}{2} E(X_{ij}) = \frac{m(m-1)}{2} \frac{1}{n} = \frac{m(m-1)}{2n}.$$

(d) Directly from (b). The probability a bin is empty is $\left(1 - \frac{1}{n}\right)^m$. Let Y_j be an indicator variable for bin Y_j, and

let $Y = \sum_{j=1}^{n} Y_j$ counts the number of empty bins. Using the same trick-methodology from the previous step we have, after noting $E(Y_i) = E(Y_5)$ for all j.

$$E(Y) = \sum_{i=1}^{n} E(Y_i) = n \cdot \left(1 - \frac{1}{n}\right)^m$$

For the same reasons as in (b) this is at most $ne^{-m/n}$. And for $m = n$ it is at most n/e.
(e) This is a binomial distribution $B(m, p; k) = B(m, 1/n; k)$.

$$B(m, 1/n; k) = \binom{m}{k} \left(\frac{1}{n}\right)^k (1 - 1/n)^{m-k} \leq \binom{m}{k} \frac{1}{n^k} \leq \left(\frac{m}{n}\right)^k \frac{1}{k!}.$$

Obviously the last expression is at most $1/k!$ for $m = n$.
(f)

$$\begin{aligned} P(\text{some bin is empty} &= P(\text{bin 1 is empty} \cup \text{bin 2 is empty} \cup \ldots \cup \text{bin n is empty}) \\ &\leq \sum_i P(\text{bin i is empty}) \\ &\leq \sum_i (1 - 1/n)^m \\ &= n(1 - 1/n)^m. \end{aligned}$$

(g) Pick k out of the m balls heading for bin 5. There are $\binom{n}{k}$ choices. And the chosen k balls ARE heading to bin 5 with probability $1/n^k$. We do not care about the remaining balls (they might be heading to bin 5 or not). Therefore the probability bin 5 has k or more balls is overestimated to be

$$P(\text{bin 5 has at least k balls}) \leq \binom{m}{k} \cdot (1/n^k)$$

Not bad! Compare it to the bound of (e) prior to the last step there of upper bounding.
(h) Let X_i be the number of balls in bin i.

$$P(\max_i X_i \geq k) = P(\exists i : X_i \geq k) \leq \sum_i P(X_i \geq k) = nP(X_i \geq k).$$

■

Exercise 580.
Let us throw t balls into n bins. What is the probability that bins 5, 8 is empty?
Implicit in the statement above is that we have enough bins to allow for bin 5, 8 (i.e. if we label them $1,2,,3,\ldots,5,\ldots,8,\ldots$), we have 8 bins or more!

Proof.
It is not difficult to conclude that

$$\frac{(n-2)^t}{n^t} = \left(1-\frac{2}{n}\right)^t.$$

■

Exercise 581.

Let us throw balls into n bins one by one. Let Y_i be a random variable that counts the number of balls needed to fill one more bin, the i-th bin (not necessarily bin i). Obviously $Y_1 = 1$, since before the throw of the first ball we assume all bins are empty.

Examine the properties of random variable Y, where $Y = \sum_{i=1}^{n} Y_i$.

The problem is also known as the coupon's collector problem. Consider also an alternative view let us call it baseball-player cards. Bob has decided to to build a collection baseball cards of n players. For that he purchases individual enveloped baseball cards; one card is contained in an opaque envelope. The chance that he gets one of any of the n players of his election is $1/n$. How many cards should he gather to make sure that he has at least one of its one of the n players?

Proof.

For $Y_1 = 1$. Random variable Y_2 follows a geometric distribution with $Y_2 \sim g(p_2)$, where $p_2 = (n-1)/n$. We can even say that $Y_1 \sim g(p_1)$, whre $p_1 = n/n = 1$. In general $p_i = (n-(i-1))/n$. The expected number of balls thrown to fill a second bin is $E[Y_2]$, etc to fill the i-th bin is $E[Y_i]$ and so on. Since Y_i $g(p_i)$ with $p_=(n-(i-1))/n$ the $E[Y_i] = 1/p_i$. Therefore, using also the linearity of expectation,

$$E(Y) = E(\sum_i Y_i) = \sum_i E(Y_i) = sum_{i=1}^{n} \frac{n}{n-(i-1} = n(1+1/2+\ldots+1/n)nH_n,$$

where H_n is the harmonic series of order n. $H_n = \ln(n) + \gamma$, where γ is Euler's constant. Furthermore,

$$\begin{aligned} var(Y) = var\sum_i Y_i) &= \sum_i var(Y_i) \\ &= \sum_{i=1} n(1-p_i)/p_i^2 \\ &\leq \sum_{i=1} n1/p_i^2 \\ &= \sum_{i=1} n\frac{n^2}{(n-i+1)^2} \\ &= n^2 \sum_{i=1} n\frac{1}{(n-i+1)^2} \\ &= n^2 \sum_{i=1} n\frac{1}{i^2} \\ &= n^2\pi^2/6, \end{aligned}$$

utilizing the Basel problem's sum due to Euler that says

$$\sum_{i=1} n\frac{1}{i^2} = \pi^2/6.$$

■

Exercise 582.

We throw n identical balls into n distinct bins with equal probability. Let X_i be the number of balls in bin i. Obviously $0 \leq X_i \leq n$ and $n = \sum_i X_i$. Moreover, $X_i \sim B(n,p;k)$ with $p = 1/n$, $q = 1-p = 1-1/n$. Furthermore, $E(X_i) = np = n(1/n) = 1$ and $var(X_i) = npq = n(1/n)(1-1/n) = 1-1/n = 1-q$.
(a) What is the probability that a given bin i gets more than $2\sqrt{n}$ balls? Use Chebyshev and Chernoff or Hoeffding bounds.
(b) What is the probability that any bin gets more than $2\sqrt{n}$ balls or equivalenty every bin has no more than $2\sqrt{n}$ balls?

Proof.
(a.1) Let X_i be the r.v. associated with the number of balls in bin i. Noting that $E(X_i) = 1$, and using Markov's inequality we have the following.

$$\begin{aligned} P(X_i > 2\sqrt{n}) &= P(X_i \geq 1 + 2\sqrt{n}) \\ &= P(X_i \geq E(X_i) + 2\sqrt{n}) \\ &= P(X_i - E(X_i) \geq 2\sqrt{n}) \\ &= P((X_i - E(X_i))^2 \geq 4n) \\ &\leq E((X_i - E(X_i))^2)/4n \\ &\leq var(X_i)/4n \\ &\leq (1-1/n)/4n \\ &\leq 1/4n. \end{aligned}$$

(a.2) We first utilize the Chernoff bound of Corollary I.16. We pick $\delta = 2\sqrt{n} > 2e-1$ for sufficienly large n, and remind ourselves $pn = 1$.

$$P(X_i > 2\sqrt{n}) = P(X_i \geq 1 + 2\sqrt{n}) = P(X_i \geq np + 2\sqrt{n}) = P(X_i \geq (1+\delta)np) = 2^{-\delta pn} = 2^{-2\sqrt{n}}$$

We next pick Lemma I.4 with $r = 2/sqrtn$. The bound is an even better $\approx 1/n^{\sqrt{n}}$. Hoeffding provides the same bound through Theorem I.26. Corollary I.25 or Corollary I.26 of Hoeffding's theorem provide lesser bound $\exp(-8n)$.
(b) Let E_i be the event that a given bin i gets more than $2\sqrt{n}$ balls? By part (a) we proved $P(E_i) \leq 1/(4n$. We want to calculate the probabibility of event E where E is the event that any bin get more than $2\sqrt{n}$ balls. Then

$$P(E) = P(E_1 \cup E_2 \cup \ldots \cup E_n) \leq \sum_i P(E_i) \leq nP(E_i)$$

Using Chebyshev's bound $nP(E_i) \leq n(1/4n) \leq 1/4$

One could prove the latter as follows. Let Y_i be a random variable that is 1 if bin i has more than $2\sqrt{n}$ balls and 0 otherwise. Then $E(Y_i) = 1 \cdot P(X_i > 2\sqrt{n}) \leq 1/(4n)$. Let us then define random variable Y such that $Y = \sum_i Y_i$ i.e. it counts the number of bins with more than $2\sqrt{n}$ balls. Obviously $E(Y) = nE(Y_i)$ $P(E)$ is $P(Y \geq 1)$. The latter can be computed through Markov's inequality

$$P(Y \geq 1) \leq E(Y) \leq nE(Y_i) \leq n(1/(4n)) = 1/4.$$

■

16.2 Binomial properties

Exercise 583.

Examine the monotonicity of the binomial process $B(n,p)$.

Proof.
Let $B(n,p;k) = \binom{n}{k} p^k (1-p)^{n-k}$. Consider

$$\frac{B(n,p;k)}{B(n,p;k-1)} = 1 + \frac{(n+1)p-k}{k(1-p)}.$$

If $k < (n+1)p$ the binomial terms form an increasing sequence. If $k > (n+1)p$ the binomial terms form a decreasing sequence. If $(n+1)p = t$ then $B(n,p;k) = B(n,p;k-1)$, since the fraction above is 1. Otherwise there exists only one integer m such that $(n+1)p - 1 < m \leq (n+1)p$. ■

Exercise 584.

Let $S_n = X_1 + \ldots + X_n$, where X_i are individually independent Bernoulli processes with $X_i \sim b(p)$. Then $S_n \sim B(n,p)$. Given the expectation, and the variance of S_n and show that

$$P(S_n \geq t) \leq t(1-p)/(t-np)^2$$

for $t > np$.

Proof.
if $t > np$ this implies $t \geq np+1 \geq (n+1)p$ since $p \leq 1$. Let $B(n,p;k) = \binom{n}{k} p^k (1-p)^{n-k}$. Consider

$$\frac{B(n,p;k)}{B(n,p;k-1)} = 1 + \frac{(n+1)p-k}{k(1-p)}.$$

If $k < (n+1)p$ the binomial terms form an increasing sequence. If $k > (n+1)p$ the binomial terms form a decreasing sequence. If $(n+1)p = t$ then $B(n,p;k) = B(n,p;k-1)$, since the fraction above is 1. Otherwise there exists only one integer m such that $(n+1)p - 1 < m \leq (n+1)p$. ■

Exercise 585.
Let $S_n = \sum_{i=1}^{n} X_i$, where X_i are individually independent Bernoulli processes with $X_i \sim b(p)$. Then $S_n \sim B(n,p)$. Show that for $6np < n$ we have the following

$$P(S_n \geq 6np) \leq 2^{-6pn},$$

using Chernoff bounds.

Proof.
Consider Corollary I.16. For $\delta > 2e-1$, the following applies.

$$P(S_n \geq (1+\delta)pn) \leq 2^{-(1+\delta)pn}.$$

If we pick $(1+\delta) = 6$ the result follows. ■

Exercise 586.

Let $S_n = \sum_{i=1}^{n} X_i$, where X_i are individually independent Bernoulli processes with $X_i \sim b(p)$. Then $S_n \sim B(n,p)$. Show that for $r > np$,

$$P(S_n - np \geq r) \leq \left(\frac{npe}{r}\right)^r.$$

Proof.
We use the Chernoff trick with the moment generating function. For this we note the following.

$$E(\exp t(X_i - p)) = p \cdot e^{t(1-p)} + (1-p) \cdot e^0 = pe^{t(1-p)} + 1 - p \leq pe^t + 1 \leq e^{pe^t}.$$

In the application of the Chernoff trick, the first inequality is by Markov's inequality's application. The last step uses the result above

$$\begin{aligned} P(S_n - np \geq r) &= P(t(S_n - np) \geq tr) \\ &= P(\exp t(S_n - np) \geq \exp tr) \\ &\leq e^{-tr} E(\exp(t(S_n - np))) \\ &= e^{-tr} E(\exp(t(\sum_i X_i - np))) \\ &= e^{-tr} \prod_i E(\exp(t(X_i - p))) \\ &= e^{-tr} \prod_i e^{(pe^t)}) \end{aligned}$$

We have thus concluded the following.

$$\begin{aligned} P(S_n - np \geq r) &\leq e^{-tr} \prod_i e^{pe^t} \\ &= e^{-tr} e^{npe^t} \\ &= e^{-tr + npe^t} \end{aligned}$$

The exponent is $f(t) = -tr + npe^t$. We find $f'(t) = -r + npe^t$. We then equate it to zero and solve for t that is, $f'(t) = -r + npe^t = 0$ implies $e^t = r/(np)$ and consequently $t = \ln(r/(np))$. Deriving $f''(t) = npe^t = np(r/(np)) = r > 0$. Thus $f(t)$ has a minimum at $t = \ln(r/(np))$. Substituting for t the right-hand side value we have.

$$e^{-\ln(r/(np))r + np(r/(np))}) = (\frac{np}{r})^r e^r = (\frac{npe}{r})^r.$$

■

Exercise 587.

Let $S_n = \sum_{i=1}^{n} X_i$, where X_i are individually independent Bernoulli processes with $X_i \sim b(p)$. Then $S_n \sim B(n,p)$. Show the following for $1 \leq r < n$.

$$P(S_n \geq r) \leq \left(\frac{npe}{r}\right)^r.$$

$$P(S_n < r) \leq \binom{n}{n-r}(1-p)^{n-r}.$$

Proof.
We prove a variation of the previous problem i.e.

$$P(S_n \geq r) \leq \left(\frac{npe}{r}\right)^r.$$

Consider all subsets S of cardinality r of $\{1,2,\ldots,n\}$. There are $\binom{n}{r}$ of them and for $j \in S$ we consider experiment X_j a success. We call the event related to the given S, event E_S (where all experiments described by S are a success). $P(E_S) = p^r$. Thus $P(S_n \geq r)$ is upper bounded by the union of all such events possible.

$$P(S_n \geq r) \leq P(\cup E_S) \leq \sum_{S,|S|=r} P(E_S) \leq \binom{n}{r} p^r. \tag{16.3}$$

However, the problem in question asks for a bound of the following.

$$P(S_n < r)$$

The probability of having fewer than r successes is also the probability of having at least $n-r$ failures. This is Eq.(16.3) substituting failure for success and thus probability of failure $1-p$ for probability of success p. Therefore

$$P(S_n < r) = P(B(n,p;k) < r)P(B(n,(1-p);k \geq n-r) \leq \binom{n}{n-r}(1-p)^{n-r}.$$

■

Exercise 588.

Let $S_n = \sum_{i=1}^n X_i$, where X_i are individually independent Bernoulli processes with $X_i \sim b(p)$. Then $S_n \sim B(n,p)$. Show that for $r \leq np$,

$$P(S_n < r) \leq \frac{r(1-p)}{np-r} B(n,p;r).$$

Proof.
Consider

$$P(S_n < r) = P(B(n,p) < r) = \sum_{i=0}^{r-1} B(n,p;i).$$

Now take the ration

$$\begin{aligned} \frac{B(n,p;i-1)}{B(n,p;i} &= \frac{i}{n-i+1}\frac{1-p}{p} \\ &\leq \frac{r}{n-r}\frac{1-p}{p} \\ &= \frac{rq}{(n-r)p}. \end{aligned}$$

The inequality above follows from $1/(n-i+1) < 1/(n-r) \Leftrightarrow r > i$. We then call a the ratio $a = rq/((n-r)p)$. Note that $a \leq 1$ since $r(1-p) \leq (n-r)p \Leftrightarrow r \leq pn$, with the latter given as a condition in the problem statement. Therefore for all $i = 0, \ldots, r-1$

$$B(n,p;i-1) \leq aB(n,p;i),$$

which implies

$$\begin{aligned} \sum_{i=0}^{r-1} B(n,p;i) &\leq \sum_{i=1}^{r} a^r B(n,p;r) \\ &\leq B(n,p;r)\sum_{i=1}^{r} a^r B(n,p;r) \\ &\leq B(n,p;r)\sum_{i=1}^{\infty} a^r B(n,p;r) \\ &\leq \left(\frac{1}{1-a} - 1\right) B(n,p;r). \\ &\leq \frac{a}{1-a} B(n,p;r). \end{aligned}$$

Note that $a/(1-a) = rq/(np-r)$. ∎

16.3 Binomial properties and Chernoff/Hoeffding bounds

Exercise 589.

Let $S_n = \sum_{i=1}^{n} X_i$, where X_i are individually independent Bernoulli processes with $X_i \sim b(1/2)$. Then $S_n \sim B(n, 1/2)$.

In n coin tosses ($p = 1/2$ of a fair coin), what is the probability of having r Heads, where $r < n/2$?

$$P(r\ Heads,\ r < n/2)?$$

Use a Chernoff or Hoeffding bound. Do not browse later questions.

Use the bound to answer the following question: What is the probability that in 100 coin tosses we have 25 or fewer heads?

Proof.

Use Chernoff Corollary I.21

$$P(S_n \leq (1-\delta)np) \leq \exp(-\delta^2 np/2).$$

with $(1-\delta)n/2 = m$ i.e. $\delta = 1 - 2m/n$. Now $n = 100$, $m = 25$, $\delta = 1 - 50/100 = 1/2$ we have

$$P(S_n \leq 25) \leq \exp -(1/4)25 < 2/1000.$$

■

Exercise 590.

(Coin tossing continues.) Let $S_n = \sum_{i=1}^{n} X_i$, where X_i are individually independent Bernoulli processes with $X_i \sim b(1/2)$. Then $S_n \sim B(n, 1/2)$.

In n coin tosses ($p = 1/2$ of a fair coin), what is the probability of having r Heads, where $r < n/2$?

$$P(|S_n/n - 1/2| \geq \delta)$$

Use a Chebyshev and then a Chernoff or Hoeffding bound.

Consider the cases $\delta = 1/2$, $\delta = \sqrt{1.5\ln(n)/n}$ and $\delta = 2\sqrt{1.5\ln(n)/n}$. Observe variation when delta doubles for the last two choices.

Proof.
Chebyshev first.

$$P(|S_n/n - 1/2| \geq \delta) = P(|S_n - n/2| \geq \delta n) \leq \frac{var(S_n)}{\delta^2 n^2} \leq \frac{1}{4\delta^2 n}.$$

Note that $var(S_n) = npq = n(1/2)(1/2) = n/4$.
Let us go for the familiar Corollary I.23. The term below (n/2) is introduced to form $np = n/2$ next to δ per Corollary I.23.

$$P(|S_n/n - 1/2| \geq \delta) = P(|S_n - n/2| \geq \delta n) = P(|S_n - n/2| \geq 2\delta(n/2)) \leq 2\exp(-(n/2)4\delta^2/3) \leq 2\exp(-(2/3)n\delta^2).$$

The latter one
(a) for $\delta = 1/2$ becomes $2e^{-n/6}$,
(b) for $\delta = \sqrt{1.5\ln(n)/n}$ becomes $2/n$, and
(c) for $\delta = 2\sqrt{1.5\ln(n)/n}$ becomes $2/n^4$.
Consider now a Hoeffding bound (e.g. Corollary I.33.

$$P(|S_n/n - 1/2| \geq \delta) = P(|S_n - n/2| \geq 2\delta(n/2)) \leq 2\exp(-2n\delta^2).$$

The latter one
(a) for $\delta = 1/2$ becomes $2e^{-n/2}$,
(b) for $\delta = \sqrt{1.5\ln(n)/n}$ becomes $2/n^3$, and
(c) for $\delta = 2\sqrt{1.5\ln(n)/n}$ becomes $2/n^{12}$. ■

16.4 Miscellanea

Exercise 591.

Give the number of derangements of permutations on n elements $\{1,2,\ldots,n\}$.

A permutation π is a derangement if and only if it does not have a fixed point. Point i is a fixed point of a permutation π if $\pi(i)=i$. Therefore in a derangement π we have $\pi(i)\neq i \quad \forall i,\ i=1,\ldots,n$.

Proof. Let S be the set of all permutations. Let $A_i=\{\pi\in S:\pi(i)=i\}$. A permutation π is a derangement if and only if

$$\pi\in\overline{A_1\cup\ldots\cup A_n}$$

Moreover A_I, $I\subseteq\{1,2,\ldots,n\}$ with $|I|=k$ contains all permutations with k fixed points. There are $|A_I|=(n-k)!$ of them. Let us try to count instead

$$n!-A_1\cup\ldots\cup A_n$$

or

$$A_1\cup\ldots\cup A_n$$

By the principle of inclusion exclusion we get

$$|A_1\cup\ldots\cup A_n|\sum_i|A_i|-\sum i<j|A_i\cap A_j|+\sum i<j<k|A_i\cap A_j\cap A_k|-\ldots=\sum_{i=1}^{n}(-1)^{i+1}\binom{n}{i}(n-i)!=n!\sum_{i=1}^{n}(-1)^{i+1}/i!.$$

Thefore $n!-|A_1\cup\ldots\cup A_n|=\sum_{i=0}^{n}(-1)^{i+1}/i!$. ■

Exercise 592.

We pick random points P_i in the unit square i.e. $P_i = (x_i, y_i)$, with $-1/2 \leq x_i, y_i \leq +1/2$. We then check whether P_i is inside the circle C with center $(0,0)$ i.e. $x^2 + y^2 \leq 1$. Then $p_i = P(P_i \in C) = \pi/4$, as the probability that P_i falls into C is proportional to the area of the circle relative to that of the square. We define random variable X_i that is equal to one if $P_i \in C$, and 0 otherwise.

We generate n random points P_i. Let S_n be another random variable with $S_n = \sum_i X_i$. We have $X_i \sim b(p_i)$. It is $p_i = p = \pi/4$. Furthermore $E(S_n) = nE(X_i) = np = n\pi/4$. Consider then another random variable $Y_n = (4/n) \cdot S_n$. Then

$$E(Y_n) = (4/n)E(S_n) = (4/n)(n\pi/4) = \pi$$

Therefore $Y_n = (4/\pi)S_n \to \pi$.

Compute

$$P(|Y_n - E(Y_n)| \leq \varepsilon\pi)$$

for some $0 < \varepsilon < 1$. Use a Chebyshev inequality and a Chernoff bound.

Proof.
Let us find some properties of the relevant random variables.

$$E(Y_n) = E((4/n)S_n) = (4/n)E(S_n) = (4/n)(n\pi/4) = \pi.$$

The we find the variance of Y_n; we note $p = \pi/4$ and $q = 1 - p = 1 - \pi/4$.

$$var(Y_n) = var((4/n)S_n) = (16/n^2)var(S_n) = (16/n^2)(npq) = 16pq/n.$$

We use Chebyshev's inequality below

$$\begin{aligned} P(|Y_n - E(Y_n)| \geq \varepsilon\pi) &\leq \frac{var(Y_n)}{\varepsilon^2\pi^2} \\ &\leq \frac{16pq}{n\varepsilon^2\pi^2} \\ &\leq \frac{4-\pi}{n\pi\varepsilon^2}. \end{aligned}$$

Therfore,

$$P(|Y_n - E(Y_n)| \leq \varepsilon\pi) \geq 1 - P(|Y_n - E(Y_n)| \geq \varepsilon\pi) \geq 1 - \frac{4-\pi}{n\pi\varepsilon^2}.$$

We use Chernoff's bound below in the form of Corollary I.23. Note that $pn = \pi n/4$ and we use $\delta = \varepsilon$.

$$\begin{aligned} P(|Y_n - E(Y_n)| \geq \varepsilon\pi) &= P(|\frac{4}{n}S_n - \frac{4}{n}E(S_n)| \geq \varepsilon\pi) \\ &= P(|S_n - E(S_n)| \geq n\varepsilon\pi/4) \\ &\leq 2\exp\left(-\frac{\varepsilon^2\pi n}{3\cdot 4}\right) \\ &\leq 2\exp\left(-\frac{\delta^2\pi n}{12}\right). \end{aligned}$$

■

For $P(|Y_n - E(Y_n)| \leq \varepsilon\pi)$ we do as before.

Exercise 593.

Birthday Problem

We have m people that have birthdays that take n values, and let for siplicity they are drawn from the set $\{1,2,\ldots,n\}$. The probability that all m have different birthdays is

$$p = \frac{n(n-1)\ldots(n-m+1}{n^m}$$

How large should m be so that this p at least $1/2 = 1-1/2$, $1-1/4$, $1-1/8$, etc $1-1/2^{10}$.

Proof.
Let $m-1 = k\sqrt{n}$ where k is some small integer. We can also use that for $x \leq 1/2$ we have $e^{-2x} \leq 1-x$ or equivalently $1-x \geq e^{-2x}$. Then $(1-\frac{i}{n}) \geq e^{-2i/n}$.

$$\begin{aligned} p &= (1-\frac{1}{n})(1-\frac{2}{n})(1-\frac{3}{n})\ldots(1-\frac{m-1}{n}) \\ &\geq \exp(-2\cdot 1/n)\cdot\exp(-2\cdot 2/n)\cdot\exp(-2\cdot 3/n)\cdot\ldots\exp(-2\cdot(m-1)/n) \\ &\geq \exp(\frac{-2m(m-1)}{2n}) \geq \exp(-k^2) \end{aligned}$$

■

Exercise 594.

Coupon Collector

Let $\{1,2,\ldots,n\}$ be a set of n cards (coupons). In a collection of m coupons how large should m be sort that there is at least one instance of each one of the n coupons? We will show that $m=(1+\varepsilon)n\ln n$ for some $\varepsilon>0$.

Proof.
For any fixed $i\in\{1,2,\ldots,n\}$, the probability i is not chosen in m choices is given by p_i

$$p_i=\left(1-\frac{1}{n}\right)^m=\exp(-\frac{m}{n})$$

Then the probability $p_1\vee p_2\vee p_3\vee\ldots\vee p_n$ is at most $\sum_i p_i=n\exp(-\frac{m}{n})$. The latter probability for $m=(1+\varepsilon)n\ln n$.

$$n\exp(-\frac{m}{n})=n\exp(-(1+\varepsilon)\ln n)=1/n^{\varepsilon}.$$

Anothe way to view this problem is by introducing an indicator variable $X_i=1$ if coupon i is never drawn and $X_i=0$ otherwise. The number of coupons NOT drawn is $X=\sum_i X_i$. The expected number of coupons NOT drawn is

$$E[X]=E[\sum_i X_i]=\sum_i E[X_i]$$

But $E[X_i]=1\cdot P(X_i=1)=(1-1/n)^m$. Thus

$$E[X]=E[\sum_i X_i]=\sum_i E[X_i]=n(1-1/n)^m$$

For $m=(1+\varepsilon)n\ln n$ this becomes $E[X]=1/n^{\varepsilon}$. Therefore if $E[X]<1$ there is a way that all coupons have been drawn.

■

Exercise 595.

Toss a coin $n = 2m$ times. What is the probability of exactly $n/2 = m$ heads?

Proof.
Let $p_k = \binom{n}{k} p^k (1-p)^{n-k}$ Assuming the coin is fair $p = q = 1-p = 1/2$ and thus $n = 2m$ and $n/2 = m$, we have

$$p = \binom{n}{k} p^k (1-p)^{n-k} \binom{2m}{m} p^m (1-p)^{2m-m} = \binom{2m}{m} (1/2)^m (1/2)^{2m-m} = \binom{2m}{m} (1/2)^{2m}. = \binom{n}{n/2} (1/2)^n.$$

Furthermore we use Stirling's formula for $n > 10$ i.e. $n! \approx \sqrt{2\pi n}(n/e)^n$

$$p = \frac{n!}{(n/2)!(n/2)!} \frac{1}{2^n} = \frac{\sqrt{2\pi n}(n/e)^n}{\sqrt{2\pi n/2}(n/2e)^{n/2} \sqrt{2\pi n/2}(n/2e)^{n/2}} \frac{1}{2^n} = \frac{\sqrt{2}}{\sqrt{\pi n}}$$

■

Exercise 596.

Hashing with chaining continues with the following problem which also serves as a generalization. Assume uniform hashing i.e. a key x is equally likely to get hashed into any of the n slots. A total of n keys get hashed into n slots of table T. Let n_i be the number of keys hashed into slot i. Answer the following.
(a) What is the probability p_k that k keys get hashed into slot i?
(b) Let $k > 1$. Show that

$$p_k/p_{k+1} > 1.$$

(c) Show that for $k > 1$,

$$P(n_i \geq k) = \sum_{j=k}^{n} P(n_i = j) \leq nP(n_i = k).$$

(d) Show that for $n \geq 20$, $A = e^2 + 1$, and $k = A\frac{\ln n}{\ln\ln n}$,

$$P(n_i = k) \leq \frac{1}{n^7}.$$

(e) Conclude that $P(n_i \geq k) \leq \frac{1}{n^6}$.
(f) Provide an bound bound for the expected size of n_i, i.e. show that

$$E[n_i] \leq k + 1,$$

where k is as defined in (d) above.
(g) After all keys are inserted into the slots randomly (uniformly at random), let D be the maximum number of keys in a slot. Show that q_k, the probability D is k is

$$q_k \leq np_k.$$

(h) Find the expected value of $E[D]$. (Hint: use the rationale of (f) above.)

Proof.
(a) The size of slot i, n_i, follows a binomial distribution that is, $n_i \sim B(n, p; k)$, where $p = 1/n$. Therefore,

$$p_k = P(n_i = k) = B(n, 1/n, k) = \binom{n}{k}(1/n)^k(1 - 1/n)^{n-k}.$$

(b) Let $k > np = n(1/n) = 1$. Consider p_k/p_{k+1}.

$$\frac{p_k}{p_{k+1}} = \frac{P(n_i = k)}{P(n_i = k+1)} = \frac{\binom{n}{k}(1/n)^k(1-1/n)^{n-k}}{\binom{n}{k+1}(1/n)^{k+1}(1-1/n)^{n-k-1}} = \frac{(k+1)(n-1)}{n-k}.$$

We claim that for $k > 1$, $(k+1)(n-1)/(n-k) > 1$. This is true as long as $kn > 1$. The latter is true as $k > 1$ and by implication $n \geq k > 1$ as well.
(c) We have that $P(n_i \geq k) = \sum_{j=k}^{n} P(n_i = j) = \sum_{j=k}^{n} p_j$. Since $p_k/p_{k+1} \geq 1$ by part (b), the terms p_j , $j \geq k$ are bounded above by p_k. Therefore

$$P(n_i \geq k) = \sum_{j=k}^{n} p_j \leq (n - k + 1)p_k \leq np_k.$$

Moreover $p_k = P(n_i = k)$.
(d) From (a) above we have

$$p_k = \binom{n}{k}(1/n)^k(1 - 1/n)^{n-k},$$

and $(1-1/n)^{n-k} \leq 1$. Since $\binom{n}{k} \leq (ne/k)^k$, we then obtain the following.

$$\begin{aligned} p_k &\leq \binom{n}{k}(1/n)^k \\ &\leq (ne/k)^k(1/n)^k \\ &\leq e^k/k^k \\ &\leq \exp\left(\ln(e^k/k^k)\right) \\ &\leq \exp(\ln Z), \end{aligned}$$

where $Z = \ln(e^k/k^k)$. In the remainder we are going to first form $\ln Z = \ln(e^k/k^k) = k - k\ln k$. Then we are going to find an upper bound for $\ln Z \leq -7\ln n$, which would then translate into an upper bound for $\exp(\ln Z)$. This way we shall prove the following.

$$\begin{aligned} p_k &\leq \exp(\ln Z) \\ &\leq \exp(-7\ln n) \\ &\leq \frac{1}{n^7} \end{aligned}$$

We start with $\ln Z$ after choosing, $A = e^2 + 1$, and

$$k = A \cdot \frac{\ln n}{\ln\ln n}.$$

Then

$$\begin{aligned} \ln Z &= \ln(e^k/k^k) = k - k\ln k \\ &= \frac{A\ln n}{\ln\ln n} - \frac{A\ln n}{\ln\ln n} \cdot \ln\frac{A\ln n}{\ln\ln n} \\ &= \frac{A\ln n}{\ln\ln n} - \frac{A\ln n}{\ln\ln n} \cdot (\ln A + \ln\ln n - \ln\ln\ln n) \\ &= A(1-\ln A)\frac{\ln n}{\ln\ln n} - A\ln n + A\ln n \cdot \frac{\ln\ln\ln n}{\ln\ln n} \\ &= A(1-\ln A)\frac{\ln n}{\ln\ln n} - (A-1)\ln n + A\ln n \cdot \left(-1 + \frac{\ln\ln\ln n}{\ln\ln n}\right) \\ &= Q_1 - (A-1)\ln n + Q_2. \end{aligned}$$

Quantity Q_1 is such that

$$Q_1 = A(1-\ln A)\frac{\ln n}{\ln\ln n}.$$

Since $A = e^2 + 1$ and thus $\ln A > 2$, we conclude $Q_1 < 0$. Furthermore, quantity Q_2 is such that

$$Q_2 = A\ln n \cdot \left(-1 + \frac{\ln\ln\ln n}{\ln\ln n}\right).$$

$Q_2 < 0$ i.e. $1 \geq \frac{\ln\ln\ln n}{\ln\ln n}$ for $n \geq 20$. Therefore we have the following noting that $A = e^2 + 1$ and thus $A - 1 = e^2 \geq 7.38$.

$$\begin{aligned} \ln Z &= Q_1 - (A-1)\ln n + Q_2 \\ &\leq 0 - (A-1)\ln n + 0 \\ &\leq 0 - 7.38\ln n + 0 \\ &\leq 0 - 7\ln n + 0. \end{aligned}$$

We then conlude the desired result.

$$\begin{aligned} p_k &\leq \exp(\ln Z) \\ &\leq \exp(-7\ln n) \\ &\Leftrightarrow \\ p_k = P(n_i = k) &\leq \frac{1}{n^7}. \end{aligned}$$

(e) From (c) above, using the upper bound for p_k derived from (d) above we have that

$$P(n_i \geq k) \leq np_k \leq n\frac{1}{n^7} \leq \frac{1}{n^6}$$

(f) We are going to estimate $E[n_i]$ for any $i = 1, \ldots, n$. We have, note also from (b) that $p_k/p_{k+1} > 1$. The value of k used is

$$\begin{aligned} E[n_i] &= \sum_{j=0}^{n} j \cdot P(n_i = j) \\ &= \sum_{j=1}^{n} j \cdot P(n_i = j) \\ &= \sum_{j=1}^{k-1} j \cdot P(n_i = j) + \sum_{j=k}^{n} j \cdot P(n_i = j) \\ &\leq \sum_{j=1}^{k-1} k \cdot P(n_i = j) + \sum_{j=k}^{n} n \cdot P(n_i = j) \\ &\leq k \cdot \sum_{j=1}^{k-1} P(n_i = j) + n \cdot \sum_{j=k}^{n} \cdot P(n_i = j) \\ &\leq k \cdot 1 + n \cdot \sum_{j=k}^{n} \cdot P(n_i = k) \\ &\leq k + n \cdot n \cdot P(n_i = k) \end{aligned}$$

Combining with the result from (e) we conclude that

$$\begin{aligned} E[n_i] &\leq k + n \cdot n \cdot P(n_i = k) \\ &\leq k + n^2/n^6 < k + 1. \end{aligned}$$

The value of $A = e^2 + 1$ and k is given as described in (d) above.

$$k = A \cdot \frac{\ln n}{\ln \ln n}.$$

(g) Below k is as defined earlier in (d) above. Let $D = \max_{i=1}^{n} n_i = \max\{n_1, n_2, \ldots, n_n\}$. We utilize also (b) that

states $P(n_i = k) \leq 1/n^7$ and this is true for all i, $i = 1, \ldots, n$.

$$\begin{aligned} P(D = k) &= P(\max_{i=1}^{n} n_i = k) \\ &= P(\exists\ i : n_i = k \wedge n_l \leq k \forall l \neq i) \\ &\leq P(\exists\ i : n_i = k) \\ &\leq P(n_1 = k \cup n_2 = k \cup \ldots \cup n_n = k) \\ &\leq nP(n_i = k) \\ &\leq n \cdot (1/n^7) = \frac{1}{n^6}. \end{aligned}$$

(h) For the expected value of $E[D]$ we work similarly as before for the $E[n_i]$. The conclusion is the same $E[D] < k+1$. ■

Exercise 597.

Hashing with chaining. A total of n keys are hashed into an $N = n$ slot table T. Answer the following.
(a) What is the expected number of keys per slot?
(b) Show that for $n \to \infty$, $P(\max_i n_i \geq 2\ln n) = 0$.
(c) What is the expected number of empty slots?
(d) What is the expected number of slots with one key?

Proof.
(a) Let n_i be the keys per slot i. Then n_i follows a binomial distribution with parameters n and $p = 1/n$ i.e. $B(n,p;k)$. Let $q = 1-p$, i.e. $p+q = 1$. Let n_i $B(n,p;k)$. Then $E[n_i]$.

$$\begin{aligned}
B(n,p;k) &= \binom{n}{k} p^k q^{n-k} \Rightarrow \\
E[n_i] &= \sum_{k=0}^{n} k \cdot \binom{n}{k} p^k q^{n-k} \\
E[n_i] &= \sum_{k=1}^{n} k \cdot \binom{n}{k} p^k q^{n-k} \\
E[n_i] &= \sum_{k=1}^{n} k \cdot \frac{n!}{k!(n-k)!} p^k q^{n-k} \\
E[n_i] &= \sum_{k=1}^{n} \cdot \frac{n!}{(k-1)!(n-k)!} p^k q^{n-k} \\
E[n_i] &= np \sum_{k=1}^{n} \cdot \frac{(n-1)!}{(k-1)!(n-k)!} p^{k-1} q^{n-k} \\
E[n_i] &= np \sum_{k=1}^{n} \cdot \binom{n-1}{k-1} p^{k-1} q^{n-k} \\
E[n_i] &= np \sum_{k=0}^{n-1} \cdot \binom{n-1}{k} p^k q^{n-1-k} \\
E[n_i] &= np(p+q)^{n-1} \\
E[n_i] &= np
\end{aligned}$$

Another way to prove the same is the n keys follow a Bernoulli distribution with $p = 1/n$. Thus let X_j be 1 if key k_j falls into slot i and 0 otherwise. Then $n_i = \sum_j X_j$ and therefore

$$E[n_i] = E[\sum_j X_j] = nE[X_j] = np.$$

Theorem 16.1 **Chernoff**

Suppose $X_1, \ldots X_n$ are independent random variables taking values in $\{0,1\}$. Let $X = sum_j X_j$ denote their sum and let $m = E[X]$ denote the sum's expected value. Then for any $\delta > 0$,

$$P(X \geq (1+\delta)m) \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}} \right)^m$$

(b) For $\delta = 2\ln n - 1$, $m = np = n(1/n) = 1$, we have

$$\begin{aligned} P(n_i \geq 2\ln n) &\leq \frac{e^{\delta}}{(1+\delta)^{1+\delta}} \\ &\leq \frac{e^{2\ln n - 1}}{(2\ln n)^{2\ln n}} \\ &\leq \frac{n^2}{e \cdot n^2 \cdot n^{2\ln\ln n}} \\ &\leq \frac{1}{e \cdot n^{2\ln\ln n}} \end{aligned}$$

Then

$$\begin{aligned} P(max_i(n_i) \geq 2\ln n) &\leq nP(n_i \geq 2\ln n) \\ &\leq \frac{n}{e \cdot n^{2\ln\ln n}} \\ &\leq \frac{1}{n^{2\ln\ln n - 1}} \end{aligned}$$

The latter upper bound goes to 0 as $n \to \infty$. Moreover if $n > e^3$, we have $2\ln\ln n - 1 \geq 1.0$, and thus the probability is at most $1/n$.
(c) Since $n_i \sim B(n,p;k)$, we have that $P(n_i = 0) = \binom{n}{0} p^0 (1-p)^n = (1-p)^n$. Let indicator random varible B_i be 1 if $n_i = 0$, and 0 otherwise. Then $C = \sum_i B_i$ is the number of empty slots.

$$E[C] = E[\sum_i B_i] = \sum_i E[B_i] = \sum_i P(n_i = 0) = n(1-p)^n$$

If $p = 1/n$ then $E[C] = n(1-1/n)^n \approx n/e$.
(d) Since $n_i \sim B(n,p;k)$, we have $P(n_i = 1) = np(1-p)^{n-1}$. Defining a random variable just like before and call D, the expected number of slots with one key, we derive similarly,

$$E[D] = n^2 p(1-p)^{n-1}$$

If $p = 1/n$ this is $E[D] = n(1-1/n)^{n-1} \approx n^2/(e(n-1))$. ■

Exercise 598.
(a) In hashing with chaining (uniform hashing assumption) for n keys and N slots, what is the probability that two keys collide?
(b) What is the exepcted number of collision for key x_i, $1 \leq i \leq n$?

Proof.
(a) From the ball and bin setup, the answer is $1/N$.
(b) Using an indicator random variable, $(n-1)/N$. ■

Exercise 599.

Hashing with chaining. A total of n keys are hashed into an $N = n^2$ slot table T. Answer the following for simple uniform hashing (assumption).
(a) What is the probability that slot i is empty after hashing n keys?
(b) What is the probability that slot i has two or more keys?

Proof.
(a) The probability that slot i is empty is the probability that all n keys missed it. In general this is $(1-1/N)^n$. Probability $1/N$ is for a key k_j to hit i or $1-1/N$ is the probability that k_j misses i. Given that we have n keys the formula derives. One can prove the first derivation below by induction on n. It also follows by expanding the power on the left side

$$(1-\frac{1}{N})^n \quad \geq \quad (1-\frac{n}{N})$$

If $N > n$ then $\geq$ becomes $>$.
(b) This is a binomial distribution's term: $b(n,k;p)$, where $k = 1$, $p = 1/N$ and thus

$$\binom{n}{1} p^k (1-p)^{n-k} \quad = \quad \frac{n}{N}(1-1/N)^{n-1}$$

Given that $N \geq n$, we have $N > n-1$. Thus from the observation and deriviation in (a) we have

$$\begin{aligned} \frac{n}{N}(1-1/N)^{n-1} &> \frac{n}{N}\left(1-\frac{n-1}{N}\right) \\ &= \frac{n}{N} - \frac{n(n-1)}{N^2} \end{aligned}$$

(c) The probability that a slot has two or more keys is one minus the probability it has zero minus the probability it has one key. From (a) and (b) above we have

$$\begin{aligned} P(n_i \geq 2) &\leq 1-(1-\frac{n}{N})-\frac{n}{N}+\frac{n(n-1)}{N^2} \\ P(n_i \geq 2) &\leq \frac{n(n-1)}{N^2} \end{aligned}$$

If we substitute $N = n^2$ we see that the probability a slot i has 2 or more keys is at most $1/n^2$. The probability that there exists a slot (0 or 1 or etc or i or etc $N-1$) that has two or more keys is at most $n \cdot 1/n^2 = 1/n$. Equivalently all slots have zero or one keys with probability at least $1-1/n$. If we increase N from $N = n^2$ to $N = n^k$, $k > 2$ we can further decrease the bound $1/n$ or increase $1-1/n$. ■

16.5 Randomness

Exercise 600.

You have a fair dice. You want to generate an integer 1 through 5 throwing this dice uniformly at random.
(a) Show how to do it; make sure it always comes with an answer i.e. it terminates.
(b) Give the expected number of throws you require.
Now you have an unfair coin what over a toss it comes H (heads) with probability p, where $0 < p < 1$, and T (tails) with probability $1 - p$.
(c) How can you generate a fair toss for H (heads) and T (tails) with this unfair coin?
(d) Give the expected number of tosses you require.

Proof.
(a) Throw the dice until it comes 1-5 i.e. ignore 6. If it comes x, where $x \neq 6$ report x. Otherwise continue.
(b) The number of throws n follows a geometric distribution with parameter $p = 5/6$. The expected number of throws is $1/p = 6/5$. i
(c) Toss the coin twice. It HT with probability $p(1-p)$ and you report T, or TH with probability $(1-p)p$ and you report H. If it comes HH or TT you ignore the outcomes. T and H reporting are equiprobable $p(1-p)$.
(d) Experiment of (c) resembles a geometric distribution with $P = 2p(1-p)$. Expectation of eventual success is $1/P = 0.5/(p(1-p))$. Moreover the number of coin tosses is $2 \cdot (1/P) = 1/(p(1-p))$ as each experiment involves two of them. ■

Exercise 601.

We have the following function call `RandomBit()` that returns a zero or one with equal probability $(1/2)$. We use if the as follows.

```
  flip(int n) {
1.  m = 0;
2.  for(i=0 ; i < n ; i++){
3.    m = 2 * m + RandomBit();
4.  }
5.  return(m)
```

(a) What is the minimum and the maximum value returned by m?
(b) Does `flip(n)` generate a uniformly at random drawn integer?

Proof.
(a) Obviously min m is 0 and max m is 2^n-1. One can prove it inductively. Prior to $i=0$, we have $m=0$. At the conclusion of $i=0$, the only possible values for RandomBit() are 0 and 1, and thus of m 0 or 1 respectively. If by induction at the conclusion of the $i=k$ iteration the min and max value of m are 0 and 2^k-1 respectively, then at the conclusion of the $i=k+1$ iteration m is minimally 0 (one more RandomBit() that is 0), and the maximum value is

$$m = 2*m+1 = 2*(2^k-1)+1 = 2^{k+1}-1.$$

Thus at the conclusion of $i=n-1$, we have that the maximum value of m is 2^n-1.
(b) When m is equal to N at the conclusion of iteration $n-1$ then, m has generated the righmost bit of N. In other words, m generates in iteration i the i-leftmost bit of N. The range of values for N is 0 and $2^0+\ldots+2n-1=2^n-1$, and each such value is generated uniformly at random with probability $\underbrace{1/2\times 1/2\times\ldots\times 1/2}_{n \text{ times}}$.

■

Exercise 602.

In order to find the minimum (MIN) of n keys of $A[0..n-1]$, the following algorithm can be used.

```
  Min(A[0..n-1],n)
1. min = A[0];
2. for(i=1 ; i < n ; i++){
3.   if A[i] < min
4.     min=A[i];
5. }
6. return(min);
```

Assume all elements of A are distinct
(a) What is the probability that $A[i]$ is the MIN?
(b) What is the probability that line 4 is executed?
(c) What is the expected number of times line 4 is executed?

Proof.
(a) The probability that $A[i]$ is the MIN is $1/n$, as there are n numbers distributed over the n slots of A.
(b) The probability that line 4 is executed is the probability that $A[i]$ is the minimum of the i keys $A[0],\ldots A[i]$ i.e. $1/(i+1)$ using the same argument as in (a) above.
(c) Let X_i be a random variable that is 1 if line 4 is executed and 0 otherwise. Then $X = \sum_i X_i$ is the number of times line 4 is executed. We are interested in finding

$$E[X] = E[\sum_i X_i] = \sum_i E[X_i]$$

We note that

$$E[X_i] = 1 \cdot P(X_i = 1) + 0 \cdot P(X_i = 0) = P(X_i = 1)$$

From part (b) P ($X_i = 1$) = $1/(i+1)$. Therefore,

$$E[X] = \sum_i E[X_i] = \sum_{i=0}^{n-1} \frac{1}{i+1}.$$

The latter sum is the harmonic series H_n thus

$$E[X] = \ln n + \gamma.$$

■

Exercise 603.

Alice and Bob have each received a sealed envelope and an assurance that each contains some money and that one contains exactly twice as much as the other. They are given the option of making the following agreement: they both open their envelopes and whoever has the more money gives it to the other person.

Alice convinces herself that taking up this option is advantageous to her by the following argument: Assume her envelope contains x amount of money. Then Bob's envelope contains either $2x$ or $x/2$, each possibility being with probability one half. Hence Alice's expected gain in taking up the option is

$$\frac{1}{2} \cdot 2x - \frac{1}{2} \cdot x = x/2 > 0$$

Since she has a positive expected gain it is worth her while to play the game. By an analogous argument Bob also concludes that taking up the option gives him an expected gain. Surely this is a contradiction.

Identify the fallacy in the previous paragraph.

Proof.

Alice and Bob are mistaken in assuming that the probabilities that the other envelope contains $2x$ and $x/2$ are both one half. That depends on the distribution of how the envelopes were originally filled, which is unknown and by no means necessarily uniform. So for different values of x, the probability that x is the larger of the two amounts is likely to vary, in which case this expected value calculation is not correct.

■

Exercise 604.
Show that $n! \nmid n^n$ for $n > 2$.

Proof.
It suffices to show that $(n-1)! \nmid n^{n-1}$. Assume by way of contradiction that $(n-1)!|n^{n-1}$.
We have two cases two consider.
(a) n is a prime number. Since $(n-1)|(n-1)!$ and $(n-1)!|n^{n-1}$ we have $(n-1)|n^{n-1}$. We have two subcases then.
(a.1) $n-1$ is a prime number. Since $(n-1)|n^{n-1}$ and n is also a prime number (case (a)), we have $n-1|n$. Since $n-1|n-1$ then $n-1|n-(n-1)$, i.e. $n=2$ but this is impossible since $n>2$.
(a.2) $n-1$ is NOT a prime number. From the decomposition theorem, $n-1 = p_1^{a_1} \dots p_k^{a_k}$, where $1 < p_i < n-1$. Since $p_i^{a_i}|(n-1)$ and $(n-1)|n^{n-1}$, we obtain $p_i^{a_i}|n^{n-1}$ i.e. $p_i = n$. This is a contradiction since $p_i < n-1 < n$.
(b) n is NOT a prime number. Consider its decomposition $n = q_1^{b_1} \dots q_t^{b_t}$. Say $n-1 = p_1^{a_1} \dots p_k^{a_k}$ as in case (a.2). Then $p_i|n-1$ and $(n-1)!|n^{n-1}$ i.e. p_i divides one of the q_j. Then $p_i = q_j$ and this is true for all i. Pick any one of them arbitrarily. Say it is p_1. Therefore since $p_1|n-1$ and $p_1 = q_j$ we have that $p_1|n$ i.e. $p_1|(n-(n-1))$ i.e. $p_1|1$ i.e. $p_1 = 1$. This contradicts the primeness of p_1. ■

Exercise 605.

```
   PermutE(A,n) // A is an array A[1..n]
1.   for(i=1;i<=n;i++) {
2.       //random(1,n)  returns
3.       //a uniformly at random integer between 1 and n
4.     swap(A[i], A[random(1,n)]);
5.   }
6.   return(A);
```

(a) Does PermutE(A,n) generate a permutation?
(b) Does PermutE(A,n) generate a random permutation?

Proof.
YES and NO.
(a) YES, one can show easily that a permutation is returned.
(b) NO. Algorithm `PermutE` generates permutations but not uniformly at random. Consider for example the case of $n = 3$. On 3 keys, there are $3! = 6$ different permutations. Algorithm `PermutE` can generate $3^3 = 27$ possible outcomes each one corresponding to a permutation; however not all of them are distinct from each other. Given that there are only 6 distinct permutations on 3 items, and that 6 does not divide 27, some of these 6 permutations will be generated more often than the others, i.e. there is a bias towards certain permutations over others. If you exhaustively generate all possible 27 outcomes of the algorithm, you can see that three permutations show up 5 times and 3 show up 4 times, i.e. there is a bias in favor of certain permutations such as $(2,3,1)$. You can see in the drawing below the possible outcomes after only the first two swaps have been performed. Out of the 9 possible permutations that can be generated so far $(2,1,3)$, $(1,2,3)$ and $(2,3,1)$ appear twice already. In general the "decision tree" corresponding to the outcomes has n^n leaves, each one reached with the same probability. The number of permutations for n items is however $n!$. The leaves can not be divided equally among the permutations since n^n is not a multiple of $n!$ for $n > 2$. Thus each permuation is not available with the same equal probability $1/n!$ and thus the algorithm is incorrect.

```
Starting Input A[i]=i for i=1,2,3

 1                            9             27

          1st           2nd (2,1,3)
        ------> (1,2,3) --- (1,2,3)
        |                   (1,3,2)
        |
        | 1st           2nd (1,2,3)   3rd  not shown
(1,2,3) ------> (2,1,3) --- (2,1,3)  ....  and so on    ...
        |                   (2,3,1)
        |
        | 1st           2nd (2,3,1)
        ------> (3,2,1) --- (3,2,1)
                            (3,1,2)
1st = Possible outcomes after swap ( A[1], A[random(1,3)] ) is done
2nd = Possible outcomes after swap ( A[2], A[random(1,3)] ) is done
3rd = Possible outcomes after swap ( A[3], A[random(1,3)] ) is done
```

■

Exercise 606.

Consider the following algorithm.

```
  FunPermute(A,n) // A is an array A[1..n]
1. for(i=1; i<= n-1 ; i++){
2. swap(A[i], A[random(i+1,n)];
3. }
4. return(A)
```

Does this code generate any permutation ? Does it produce a uniform at random non-identity permutation?

Proof.
No or not if $n > 2$. The element in the first position is swapped out at $i = 1$; in the remainder it cannot be re-occupy $A[1]$.

■

Exercise 607.

You are given RAND(a,b) that returns a random integer number between a (inclusive) and b (inclusive) with uniform probability. Show how to use this RAND function to generate a random permutation in $O(n)$ time of n distinct objects stored in array $X[1..n]$. You may assume that a call to RAND takes constant time $O(1)$.

For example if X has three items $< 10,20,30 >$ then after your algorithm is run it is equally likely that the output is $< 10,20,30 >$ or $< 20,10,30 >$ or any one of the remaining 4 permutations of the three distinct objects 10, 20, 30.

Proof.

```
   Permute(A,n) // A is an array A[1..n]
1.   for(i=1;i<=n;i++) {
2.       //random(1,n)  returns
3.       //a uniformly at random integer between 1 and n
4.     swap(A[i], A[random(i,n)]);
5.   }
6.   return(A);
  or
   Permute(A,n) // A is an array A[1..n]
1.   for(i=n; i>=1 ;i--) {
4.     swap(A[i], A[random(1,i)]);
5.   }
6.   return(A);
```

The algorithm (second version) works in-place. At the start, at the conclusion of iteration $I = 1$, where $i = n+1-I$, the items at index $i = n+1-I$ stores any one of the items in position $n+1-I, n-I, \ldots, 1$, a choice among $n+1-I = n$ items. By induction, in iteration I, where $i = n+1-I$, the items at index $i = n+1-I$ stores any one of the items in positions $n+1-I, n-I, \ldots 1$, a choice among i of them. Therefore for iteration $I = 1$ (equivalent to $i = n$), there are n outcomes for $A[i]$, for iteration $I = 2$ (equivalent to $i = n-1$) there are $n-1$ outcomes for $A[i]$), and so on. The decision tree has $n!$ leaves, corresponding to the outcomes of Permute. Each of the outcomes is generate with the same probability $1/(n(n-1)\ldots 1) = 1/n!$.
The running time is obviously $O(n)$ if a single call to RAND is $O(1)$.
Inductive hypothesis: Before the start of iteration i, $A[1..i-1]$ contains an $(i-1)$-permutation ($i-1$ of the n keys) with probability $(n-i+1)!/n!$. (For a base case this is true as well by default.)
Inductive step. An i-permuations is an $(i-1)$-permutation followed, in the i-th iteration by the element picked for $A[i]$. Let the $(i-1)$-permuation of the induction step be p. Its probability of occurence is by the inductive hypothesis $(n-i+1)!/n!$. The even that a given item is picked among the $A[i..n]$ and swapped at $A[i]$ is $1/(n-i+1)$. Thus the probability that the i-permuation generated at the end of the i iteration is as is is $1/(n-i+1) \cdot (n-i+1)!/n! = (n-i)!/n!$. At the completion of the $i = n$ iteration, a given permutation formed appears with probability $(n-n)!/n! = 1/n!$, as required. ■

Exercise 608.

The `Higlander Casino` plays a game of random permutations on 4 items as follows.

- The client bets on a permutation of his/her choice.
- The casino uses the incorrect one of the algorithms (i.e. PermutE) to generate a random permutation.
- The casino pays the client the fair amount of money if the client won (i.e. guessed right). Fair money means in the long run neither the client nor the casino would make any money IF their algorithm chose correctly and fairly a permutation (i.e. for n items the casino pays n dollars for every dollar bet right).

Since the casino's algorithm is incorrect, what methodology could you follow to gain over the casino? What edge (percentage-wise) do you expect to gain over the casino? Explain.

Proof.
The casino is using PermutE. With 4 items the decision tree has $4^4 = 256$ leaves. There are however $4! = 24$ permutations on 4 items and these 24 permutations correspond to the 256 leaves. So, on the average $256/24$ leaves correspond to each permutation (or a given permutation appears on the average that often as outcome of PermutE). The ratio $256/24$ is equal to 10.666. A permutation can appear as a leaf 10 or 11 times but not 10.666 times. Therefore there must be a permutation that appears at least 11 times since otherwise all permutations appear only 10 times and therefore the 24 of them can only appear $24 \times 10 = 240$ times as leaves but there are 256 leaves permutation generated by PermutE. Therefore our algorithm in beating the casino is to identify this permutation (keeping track of what shows up as a result of successive games and histograming these outcomes) and after doing so, to bet on it time after time. Our expected probability of winning would thus be (at least) $11/256$. Since the casino pays fair money, if we bet 1 we win 24\$. So our expected return would be $11/256 * 24 = 264/256 = 1.03125$. So our edge over the casino is 3.125%.

■

Exercise 609.

You are given a random vector $a = (a_1, \ldots a_n)$, where a_i is equally likely and independently to be 0 or 1, i.e. $P(a_i = 1) = P(a_i = 0) = 1/2$. Answer the following questions.
(a) **(Warmup)** What's the probability that a is the all zero vector (3 points)?
(b) Suppose that a, b are two 0-1 vectors of length n whose components were chosen uniformly at random as discussed previously. What is the expected value of the inner product $a \cdot b = \sum_{i=1}^{n} a_i b_i$? Explain (17 points).
(c) Let d be a vector of integers mod p (i.e. items of d are $0, \ldots p-1$), where p is a prime. Let a be a random vector of 0-1's chosen as before. What is an upper bound on the probability that $\sum d_i a_i \equiv 0 \bmod p$? Explain.

Proof.
(a) $1/2^n$.
(b) $n/4$.

$$E[c] = E[ab] = E[\sum_i a_i b_i] = \sum_i E[a_i b_i]$$

$a_i b_i$ is 0 with probability $3/4$ and 1 with probability $1/4$ (when both $a_i = b_i = 1$).

$$E[a_i b_i] = 0(3/4) + 1(1/4) = 1/4$$

Therefore

$$E[c] = E[ab] = E[\sum_i a_i b_i] = \sum_i E[a_i b_i] = n(1/4)$$

(c) The vector d is given (and is not necessarily random). Assume $d \neq 0$, i.e. at least one component of the vector is non-zero since otherwise the problem is trivial. $\sum d_i a_i \equiv 0 \bmod p$.

$$\begin{aligned} \sum_i d_i a_i &\equiv 0 \bmod p \\ d_1 a_1 + d_2 a_2 + \ldots + d_n a_n &\equiv 0 \bmod p \\ d_1 a_1 &\equiv (-d_2 a_2 - \ldots - d_n a_n) \bmod p \\ d_1 a_1 &\equiv Z \bmod p \end{aligned}$$

where $Z \equiv (-d_2 a_2 - \ldots - d_n a_n) \bmod p$. Then,

$$\begin{aligned} \sum_i d_i a_i &\equiv 0 \bmod p \\ d_1 a_1 &\equiv Z \bmod p \\ a_1 &\equiv (d_1)^{-1} Z \bmod p \\ a_1 &\equiv B \bmod p \end{aligned}$$

The inverse of d_1 exists since $d_1 x \equiv 1 \bmod p$ has a single solution for x by the fact that $d_1 < p$ is such that $(d_1, p) = 1$ and p is prime. $B = (d_1)^{-1} Z \bmod p$ is an integer in $0, \ldots, p-1$. a_1 is a (uniformly at) random (chosen) 0,1. What is the probability that the random a_1 is B? Naturally this probability is at most $1/2$, as after we fix B, a_1 can agree with this fixed value of B half of the time only. If the a_i's were not binary but ternary, then the probability bound would be $1/3$ instead. ■

Exercise 610.

SomeIT has two types of professors: `mean` and `nice`. A generous professor gives A's to all 100% of the students in his class. A mean professor gives A's to 80% of the students and $B+$ to the remaining students. Let's assume that all course/class grades are available in the form of arrays and all arrays have the same size n. How fast can you determine reliably (i.e. you are confident on your estimate 51% of the time or more) whether the grades of a given professor correspond to a mean or a nice one?
Note that your algorithm does not need to be correct all the time. However it needs to be fast. An obvious $O(n)$ solution that is correct 100% of the time is to go through array G and if we find a $B+$ we return `mean` otherwise we return `generous`.
This is too slow however.

Proof.
We look at say 11 random indices $i_1, \ldots i_{11}$.

```
1. SomeIT(A.roster,n,A.instructor)
2.  for(j=1;j<=11;j++){
3.    k=ChooseRandom(1,n) //Choose random index in 1..n
4.    i[j]=k;
5.    if A[k] == "B+'
6.        printf("mean");
7.        return;
8.  }
9.  printf("nice");
10. return;
```

SomeIT is an instance of a probabilistic algorithm. It always returns an answer, but sometimes the answer might be incorrect (unreliable). When SomeIT returns `mean` through lines 6-7, we know that the instructor is indeed `"mean"`, since a `B+` grade has been found in the roster. When SomeIT returns `nice` through lines 9-10 we might have of two cases.
Case 1. The instructor in question is `nice` and there was no chance of finding a `B+` whether 11 or n of the grades were to be examined. The answer given is thus correct.
Case 2. The instructor in question is `mean` and thus the answer given is misleading and unreliable. SomeIT failed to detect a `B+` in 11 trials, i.e. it detected an `A` in 11 trials.
What are the chances that the answer is unreliable i.e. Case 2 applies? The chance that we get an `A` answer to a lookup given a `mean` professor is 0.8 (80%). The chance that we get an `A` 11 times in a row in randomly picked probes is $0.8^11 < 0.086 < 0.10$. Therefore $> 1 - 0.10 = 90\%$ of the time a `mean` professor would be detected through lines 6-7, but only $\leq 8.6\%$ of the time a potentially unreliable answer will be given through lines 9-10.
A probabilistic algorithm similar to SomeIT always tells the truth when it says `mean`. Every time it says `nice` the result is reliable with confidence 90% or more. SomeIT is what we call a `Monte-Carlo` algorithm. ■

Part VIII

Brief math review

Appendix A

Notation

A.1 Notation

Some preliminaries.

Definition A.1 **colon symbol : and pipe symbol |**

The colon symbol : stands for **such that**. The pipe symbol | also stands for **such that** or alternatively for **where**.

Definition A.2 **universal quantifier ∀**

The ∀ symbol is also known as the **universal quantifier**. It reads as **for all**.

Definition A.3 **existential quantifier ∃**

The ∃ symbol is also called the **existential quantifier**. It reads as **there exists** in singular or in plural as **there exist**.

Definition A.4 **set membership**

Symbol ∈ is the **belongs to** set membership symbol.

Definition A.5 **implication**

$X \Rightarrow Y$ is also known as implication and can be stated otherwise as "X implies Y". Y is then necessary for X, and X is sufficient for Y.

Definition A.6 **Unknown, Variable, Indeterminate**

For $f(x)$ or $\log(x)$ the x inside the parentheses is what is traditionally known as an unknown. Compu-speak we might call it a "variable" but we have not yet defined a variable formally. In Math we also call it an indeterminate.

An **operator** in mathematics and also in computing is a symbol that indicates an **operation**. The object of the operator and its operation is known as the **operand**. An operation denoted by an operator (and thus a function) can have one or more operands and is then known as a unary operator/operation (one operand), binary (two operands), etc.

Definition A.7 **Unary operators and operations**

A unary operator is a symbol that indicates a unary operation, that is, the application of a mathematical or computing function on one single value. The value is known as the operand of the operator and the corresponding operation.

When a unary operator is used, it precedes (or surrounds) its operand.

Any one of the trigonometric functions such as sin is a unary operator. In $sin(x)$, the sin is a unary operator, the x is the operand and $sin(x)$ is the unary opeation that involves the application of the sine trigonometric function on operand x.

Thus **operator is a symbol**, the **operation is the mathematical or comuting function** implied by the symbol (operator) and **the object of the operation or operator is the operand**.

Another unary operator is the absolute value function $||$. Thus $|5|$ is a 5. We also have unary operators $+$ and $-$ to assign a positive or negative sign to a number. Thus in $+5$ and -5 the operator also precedes its corresponding single operand.

Besides **unary operators** we also have **binary operators** that denote a **binary operation**.

Definition A.8 **Binary operators, and operations**

A binary operator is a symbol that indicates a binary operation i.e. the application of a mathematical or computing function on two values. The two values are known as the operands of the operator and its corresponding operation. The first one (from the left) is known as the left operand and the second one as the right operand.

The $+$, the plus-symbol, is the (additive) binary operator that indicates the operation known as addition. In $5+3$, the operator is the plus $(+)$, the operation is addition as implied by the presence of the additive operator plus. Operation addition is a binary operation and requires two operands present. Numbers 5 and 3 are the two operands that will participate in the operation: 5 is the left operand and 3 the right operand. In this context, operation addition is a binary operation and $+$ is a binary operator because they require two operands. Previously, the plus-symbol indicated a unary operator and operation. The presence of one or two operands resolves the type of the operator/operation (unary vs binary).

Definition A.9 **Operator overload**

The same operator can indicate one or more operations: one unary and one binary. The presence of one or two operands, that is the context, can be used to resolve the type of the operator and its coresponding operation.

In expression $5-3$ the dash-symbol (also known as the minus-symbol) $-$ denotes operation subtraction and it is a binary operator. Operation subtraction is then a binary operation. The plus-symbol and similarly the dash-symbol are also unary operators and each one indicates the sign assigned to an integer.

Definition A.10 **Prefix, postfix and infix notation**

A unary operator requires one operand, a binary operator two operands. In the former case the operator precedes the operand. In the latter case the operator can precede, follow or be in-between the operands. Thus $+5\ 3$ or $5\ 3+$ or $5+3$ indicate the same addition operation in prefix, postfix and infix notation. In all cases 5 is the left operand and 3 is the right operand.

We are used to using infix notation in describing operations.

Definition A.11 **Integer vs Real ideterminate names**

In functions defined hereafter we will shall more often use n instead of x. Indeterminate n implies a non-negative or positive integer. Indeterminate x implies a real number. We describe a discrete math universe of non-negative integers.

Definition A.12 **Algorithms: Problem size, Input size and names**

Indeterminate n will usually denote problem size or input size. Thus for a sequence of n keys, n represents the number of elements or keys in the sequence, the problem size and also to some degree the input size. For a 2d-array (aka matrix) $n \times n$, n represents the problem size denoting the number of its rows or columns i.e. its geometry or shape of the matrix. In this latter example the input size is the size of the matrix i.e. its number of elements which is n^2.

Definition A.13 **Sums and Sigma notation**

The sum $a_0 + a_1 + \ldots + a_n$ can be represented in compact form as

$$\sum_{i=0}^{i=n} a_i = \sum_{i=0}^{n} a_i = a_0 + \ldots + a_n$$

Variable i has values that vary between a smallest value as indicated under the sum's Sigma symbol and it is $i = 0$ in this example and its largest value as indicated over the sum's Sigma symbol and it is $i = n$). It also assumes all integer number values between 0 and n (inclusive of the end points). The variable's name is available beneath the Sigma and can be omitted over it as shown in the second formulation of the sum. The terms of the sum are usually members of a sequence and in this case $a_0, a_1, \ldots, a_n$. The general member of the sequence is a_i as described in the sigma / sum formulation. If the sequence is simple such as $a_i = i$ instead of a_i we use directly i; likewise for $a_i = i^2$, $a_i = i^3$.

Definition A.14

Let

$$S_k(n) = 1^k + 2^k + \ldots + n^k = \sum_{i=1}^{n} i^k$$

for any integer $k > 0$.

The latter term $\sum_{i=1}^{n} i^k$ indicates a sum that runs from $i = 1$ through (inclusive) $i = n$. The i-th term of the sum is i^k.

Definition A.15 **Integer numbers**

An **integer number** is a number that takes integer values. It can be positive, negative or zero. For a positive integer number we might or might not place a plus sign $+$ before its magnitude. For a negative integer number we always place a negative sign $-$ before its magnitude.

Definition A.16 **Non-negative integer numbers**

A **non-negative integer number** can be positive or zero.

Definition A.17 **Natural (integer) numbers: unsigned integers**

A **natural (integer) number** is an integer number that is a *positive integer number*. However this definition varies and it might also mean a *non-negative integer number*. We also call it an **unsigned integer**.

Most numbers listed below would be natural numbers (one way or the other). When we start talking about negative numbers this will be made very clear (and the discussion will be brief).

Example A.1.1

Integer 13 is a positive integer and so is $+13$. Integer -13 is a negative integer. Integer 0 is neither positive nor negative. Ordinarily, there should be no sign preceding a 0.

Definition A.18 **Integer Numbers: Signed Integers**

In general, a **(signed) integer** number can be positive, negative or zero.

A zero does not have a sign. A 5 or $+5$ mean the same thing: a positive sign for 5. Then a -5 means a negative sign for five i.e. minus five.

Definition A.19 **Real Numbers: Floating-point Numbers**

A real number that includes integer digits, possibly a decimal point, and decimal digits is called a `floating-point` number.

Thus 12.1 or 12.10 or $1.21 \cdot 10^1$ all represent the same real number.

Definition A.20 **Exponential notation**

A real number can be expressed in exponential notation in the form $a \times 10^b$ or equivalently as aEb or aeb.

Thus 5.1×10^3 is $5.1e3$ or $5.1E3$.

Definition A.21 **Magnitude of a number real or integer**

The magnitude of an integer or real number is its absolute value.

Example A.1.2 **Magnitude vs value**

For a negative number such as -5 its magnitude is 5 and its value is -5. Thus the 'we always place a negative sign $-$ before its magnitude' above makes sense.

A.2 Boolean function

Definition A.22 **conjunction and disjunction**

We read $\wedge$ as 'conjunction' i.e. 'AND' and $\vee$ as 'disjunction' i.e. 'OR'.

We provide a bit more information about Boolean functions. All operations are defined primarily for Boolean arguments. Note that an operator is a symbol that denotes the corresponding operation. In general $+$ is a symbol that denotes primarily addition of numbers, but by using operator overaloading it can denote other operations including a disjunction.

Definition A.23 **Boolean**

Boolean variables take two values true or false. For true, we might use `true` or `t` or `1`. Likewise we might use for false, `false` , or `f` , or `0`. Moreover in programming languages any non-zero value evaluates to true and a zero value evaluates to false. (Thus a non-zero value is true or `1` whether it is 1 or some other non-zero value.)

Definition A.24 **Logical Operation: Conjunction.**

The symbol $\wedge$ or $\cdot$ or & or . indicates a conjunction of two boolean variables. The result if true if and only if both evaluate to true. We can write $x \wedge y$ or $x \cdot y$ or $x\&y$ or sometimes $AND(x,y)$ or $x\ AND\ y$.

Definition A.25 **Logical Operation: (Inclusive) Disjunction.**

The symbol $\vee$ or $+$ or $|$ indicates a disjunction of two boolean variables. The result if true if and only if at least one evaluates to true. We can write $x \vee y$ or $x + y$ or $x|y$ or sometimes $OR(x,y)$ or $x\ OR\ y$.

Definition A.26 **Logical Operation: Negation.**

The symbol $\neg$ or $\sim$ indicates the negation of a single boolean variable. The result is true or false depending on whether the variable is false or true respectively. We can write $\bar{x}$ or $\sim x$ or $NOT(x)$.

Definition A.27 **Logical Operation: Exclusive Disjunction.**

The symbol XOR or $\oplus$ indicates an exclusive disjunction of two boolean variables. The result if true if and only if exactly one is true and exactly one is false. We can write $x\ XOR\ y$ or $XOR(x,y)$ or $x \oplus y$.

A.3 Axioms, Propositions, Theorems

In computer science we prove statements. Such statements need to be expressed/stated precisely.

Definition A.28 **Proposition**

A proposition is a statement that is either true or false.

An axiom forms the basis for logically deducing other statements.

Definition A.29 **Axiom**

An axiom is a statement accepted or assumed to be true without proof. An axiom is thus a proposition.

The first proposition below is obviously true; the next one is false. (Integers shown in this section are traditional denary integers also known as decimal i.e. base-10 or radix-10 integers.)

Proposition A.1

$1 + 1 = 2$

Proposition A.2

$1 + 1 = 1$

Proposition A.3 **Goldbach's Conjecture**

Every even integer greater than two is the sum of two prime numbers.

Proposition A.1 is true and Proposition A.2 is false. According to Wikipedia, Proposition A.3 is true for all integers up to about $4 \cdot 10^{18}$; no one knows whether Proposition A.3 is indeed true as stated.

Definition A.30 **Predicate**

A predicate is a proposition whose truth is dependent on the value of one or more variables.

Proposition A.4 below is defined in the form of predicate $P(n)$

Proposition A.4

P(n): For all positive integer (also known as a natural number) n, integer n^2+7 is prime.

Another way to form this predicate is by writing P(n): $\forall n \in N$. n^2+7 is prime.
Here we use the universal quantifier to state **for all**. Moreoever the set of natural numbers (i.e. positive integers) is denoted by N.

Definition A.31 **Proof**

A proof is a verification of a proposition by a sequence of logical deductions derived from a base set of axioms.

Definition A.32 **Theorem**

A theorem is a proposition along with a proof of its correctness.

Definition A.33 **Lemma, Corollary**

A lemma is a preliminary or simpler theorem useful to proving a proposition that yields a theorem. A corollary is a proposition that follows from a theorem in few and usually simple logical deductions.

A lemma precedes, and a corollary follows a theorem.

Proposition A.4 can be easily shown to be false by providing a counterexample.

Proof. (That Proposition A.4 is false.) For $n=3$, we have that $n^2+7=3^2+7=16$ and one divisor of 16 other than 1 and 16, is 2. Therefore 16 is not a prime number, it is in fact a composite number and therefore this simple counterexample shows that Proposition A.4 is false because it is not true for $n=3$. □ ■

Counterexample. To prove that this proposition is false it suffices to find a single integer of the form n^2+7 that is not a prime number. Thus determining for $n=3$ that 3^2+7 is not a prime number completes the proof that the proposition is FALSE.

Proposition A.5

$\exists n \in N$ such that n^2+7 is prime.

In order to prove that Proposition A.5 is true, we only need prove it for a single value of n. For $n=2$, we can easily establish that $2^2+7=11$ is a prime number. Proposition A.5 is not, however, very interesting.

Appendix B

Functions

B.1 Exponentials and logarithms

Definition B.1 **Powers of 2**

The expression 2^n means the multiplication of n twos.

Therefore, $2^2 = 2 \cdot 2$ is a 4, $2^8 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$ is 256, and $2^{10} = 1024$. Moreover, $2^1 = 2$ and $2^0 = 1$. Several times one might write $2 * *n$ or $2\hat{}n$ for 2^n ($\hat{}$ is the hat/caret symbol usually co-located with the numeric-6 keyboard key).

Power	Value
2^0	1
2^1	2
2^4	16
2^8	256
2^{10}	1024
2^{16}	65536
2^{20}	1048576
2^{30}	1073741824
2^{40}	1099511627776
2^{50}	1125899906842624

Figure B.1: Powers of two

Prefix	Name	Multiplier
d	deca	$10^1 = 10$
h	hecto	$10^2 = 100$
k	kilo	$10^3 = 1000$
M	mega	10^6
G	giga	10^9
T	tera	10^{12}
P	peta	10^{15}
E	exa	10^{18}
d	deci	10^{-1}
c	centi	10^{-2}
m	milli	10^{-3}
μ	micro	10^{-6}
n	nano	10^{-9}
p	pico	10^{-12}
f	femto	10^{-15}

Figure B.2: SI system prefixes

Prefix	Name	Multiplier
Ki	kibi or kilobinary	2^{10}
Mi	mebi or megabinary	2^{20}
Gi	gibi or gigabinary	2^{30}
Ti	tebi or terabinary	2^{40}
Pi	pebi or petabinary	2^{50}

Figure B.3: SI binary prefixes

Definition B.2 **Properties of powers**

- (Multiplication.) $2^m \cdot 2^n = 2^m\, 2^n = 2^{m+n}$. (Dot $\cdot$ optional.)
- (Division.) $2^m / 2^n = 2^{m-n}$. (The symbol / is the slash symbol)
- (Exponentiation.) $(2^m)^n = 2^{m \cdot n}$.

Example B.1.1 **Approximations for 2^{10} and 2^{20} and 2^{30}**

Since $2^{10} = 1024 \approx 1000 = 10^3$, we have that $2^{20} = \left(2^{10}\right)^2 \approx 1000^2 = 10^6$, and likewise, $2^{30} = \left(2^{10}\right)^3 \approx 1000^3 = 10^9$.

The last number, a one followed by nine zeroes, we call it a billion in American English; in (British) English a billion is a million millions (aka trillion). If one writes 10^9 or 10^{12} no confusion is possible; therefore avoid saying "billion" or might hear a joke about millions, billions and trillions.

Note B.1.1

A kilo uses a lower case **k**. A capital case **K** stands for Kelvin, as in degrees Kelvin.

B.2 Logarithms base two (and e, and 10)

Definition B.3 **Logarithm base two of n is $\lg(n)$**

The logarithm base two of n is formally denoted by $y = \lg(n)$ or if we drop the parentheses, $y = \lg n$, and is defined as the power y that we need to raise integer 2 to get n.

$$\text{That is, } y = \lg(n) \iff 2^y = 2^{\lg(n)} = n.$$

From now on we will be using the informal form $y = \lg n$ without parentheses instead of $y = \lg(n)$. Another way to write both is $y = \log_2 n$ or $y = \log_2(n)$. The two writings: $\lg^k n = (\lg n)^k$ are equivalent. We sometimes write $\lg\lg n$ to denote $\lg(\lg(n))$ and the nesting can go on. Note that $\lg^{(k)} n$ with a parenthesized exponent means something else (it is the iterated logarithm function).

Definition B.4 **Other Logarithms**

The other logarithms: $\log_{10}(x)$ or $\log_{10} x$ and $\ln(x)$ or $\ln x$ or $\log_e n$, are to the base 10 or to the base $e = 2.7172\ldots$ of the Neperian logarithms respectively. If one writes $\log n$, then the writing may be ambiguous. Note that if we tilt towards calculus we use x as in $\lg(x)$ but if we tilt towards computing or discrete mathematics we use n as in $\lg(n)$ for the indeterminate's i.e. variable's name.

Expression	Value	Explanation
$\lg(n)$	y	since $2^y = 2^{\lg n} = n$ (**by definition**)
$\lg(1)$	0	since $2^0 = 1$
$\lg(2)$	1	since $2^1 = 2$
$\lg(256)$	8	since $2^8 = 256$
$\lg(1024)$	10	since $2^{10} = 1024$
$\lg(1048576)$	20	since $2^{20} = 1048576$
$\lg(1073741824)$	30	and so on

Figure B.4: Logarithms: Base two

Example B.2.1

$\lg 2$ is one since $2^1 = 2$. $\lg(256)$ is 8 since $2^8 = 256$. $\lg(1)$ is 0 since $2^0 = 1$.

Theorem B.1 **Properties of Logarithms.**

In general, $2^{\lg(n)} = n$ *and thus,*

i. (Multiplication.) $\lg(n \cdot m) = \lg n + \lg m$.

ii. (Division.) $\lg(n/m) = \lg n - \lg m$.

iii. (Exponentiation.) $\lg(n^m) = m \cdot \lg n$.

iv. (Change of base.) $n^{\lg m} = m^{\lg n}$. *Moreover* $\lg a = \frac{\log a}{\log 2}$ *(the* log *denotes a logarithm to an arbitrary base).*

Example B.2.2

Since $2^{20} = 2^{10} \cdot 2^{10}$ we have that $\lg(2^{20}) = \lg(2^{10} \cdot 2^{10}) = \lg(2^{10}) + \lg(2^{10}) = 10 + 10 = 20$. Likewise $\lg(2^{30}) = 30$. Drawing from the exercise of the previous page, $\lg(1,000) \approx 10$, $\lg(1,000,000) \approx 20$ and $\lg(1,000,000,000) \approx 30$.

Example B.2.3

How much is $n^{1/\lg n}$**?** Let $z = n^{1/\lg n}$. Then by taking logs of both sides (and using the exponentiation rule for logarithms) $\lg z = (1/\lg n)\lg n = 1$, we have $\lg z = 1$ which implies $z = 2$.

Example B.2.4

Exponential function: A Lower bound. For all real x, $e^x \geq 1 + x$, where $e = 2.7172\ldots$.

Example B.2.5

Exponential function: An upper bound (and the lower bound above combined). For all x such that $|x| < 1$, we have that $1 + x \leq e^x \leq 1 + x + x^2$.

B.2.1 Other functions

Definition B.5 **The floor function:** $\lfloor x \rfloor$

The floor function is defined as follows: $\lfloor x \rfloor$ is the largest integer less than or equal to x.

Definition B.6 **The ceiling function:** $\lceil x \rceil$

The ceiling function is defined as follows: $\lceil x \rceil$ is the smallest integer greater than or equal to x.

Example B.2.6 **The floor of** $\lfloor 10.1 \rfloor$ **and** $\lfloor -10.1 \rfloor$

We have that $\lfloor 10 \rfloor$ is 10 itself. Moreover $\lfloor 10.1 \rfloor$ is 10 as well. Note that $\lfloor -10.1 \rfloor$ is -11.

Example B.2.7 **The ceiling of** $\lceil 10.1 \rceil$ **and** $\lceil -10.1 \rceil$

We have that $\lceil 10 \rceil$ is 10 itself. Moreover $\lceil 10.1 \rceil$ is 11 as well. Note that $\lceil -10.1 \rceil$ is -10.

B.2.2 Inequalities

For a discrete function the indeterminate (variable, uknown) will more often than not be denoted by n. For continuous functions by x. Thus $A(n)$ is a discrete function and $B(x)$ is a continuous variable function (unless otherwise noted). There are several ways one can use to comparae two functions $f(n)$ and $g(n)$. First we note some properties of integer and real numbers as summarized below.

Fact B.1 **Inequalities**

$$\begin{aligned}
\textbf{Transitivity} &: & a<b, \quad \wedge \quad b<c \quad &\Rightarrow & a<c.\\
\textbf{R1}. &: & a<b, \quad \forall q \quad &\Rightarrow & a+q<b+q\\
\textbf{R2}. &: & a<b, \quad c<d \quad &\Rightarrow & a+c<b+d\\
\textbf{R3}. &: & a<b, \quad c>0 \quad &\Rightarrow & a\cdot c<b\cdot c\\
\textbf{R4}. &: & a<b, \quad c<0 \quad &\Rightarrow & a\cdot c>b\cdot c\\
\textbf{R5}. &: & 0<a<b \quad &\Rightarrow & 1/a>1/b\\
\textbf{R6}. &: & \textbf{Flip } < \textbf{ into } > \textbf{ and } > \textbf{ into } <
\end{aligned}$$

Example B.2.8
Show that for $0<a<b$ we have that $a^2<b^2$.

Proof.

$$\begin{aligned}
\textbf{Use R3 with } c=a\wedge a>0: \quad & a<b \Rightarrow & a^2<ab \qquad (B.1)\\
\textbf{Use R3 with } c=b\wedge b>0: \quad & a<b \Rightarrow & ab<b^2 \qquad (B.2)\\
\textbf{Use Transitivity and } (1) \textbf{ and } (2) \textbf{ above}: \quad & a^2<ab\wedge ab<b^2 \Rightarrow & a^2<b^2
\end{aligned}$$

■

Example B.2.9
Show that for $0<a<b$ we have that $a^3<b^3$.

Example B.2.10
Show that $\sum_{i=1}^{n} i^3 \leq n^4$.

Proof.

$$\sum_{i=1}^{n} i^3 = 1^3+2^3+\ldots+i^3+\ldots+(n-1)^3+n^3.$$

Since i runs from 1 to n we have, using the examples before,

$$\begin{aligned}
1\leq i\leq n: 1\leq n \Rightarrow \quad & 1^3\leq n^3\\
\ldots \Rightarrow \quad & \ldots\\
i\leq n \Rightarrow \quad & i^3\leq n^3\\
\ldots \Rightarrow \quad & \ldots\\
n\leq n \Rightarrow \quad & n^3\leq n^3\\
\textbf{Add both sides of inequalities} \quad & 1^3+2^3+\ldots+i^3+\ldots+n^3\leq n^3+n^3+\ldots+n^3+\ldots+n^3\\
& \sum_{i=1}^{n} i^3 \leq n\cdot n^3 = n^4
\end{aligned}$$

■

B.3 The factorial and Stirling's approximation formula

Definition B.7 **Factorial $n!$ definition**

The factorial $n!$ is defined as follows:

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

By definition $0! = 1$ and $1! = 1$. Thus $2! = 2$, $3! = 6$ and so on.

Definition B.8 **Choose symbol**

The choose symbol as in *n choose k* or *choose k objects out of n*.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Theorem B.2 **Upper bound for factorial**

For every $n \geq 1$, we have that $n! \leq n^n$.

Proof. The first result is a straighforward upper bound of each term of the factorial with n.

$$\begin{aligned} n! &= \overbrace{1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n}^{n \text{ terms}} \\ &\leq \overbrace{n \cdot n \cdot \ldots \cdot n \cdot n}^{n \text{ terms}} \\ &\leq n^n \end{aligned}$$

■

Corollary B.1 **Logarithmic Upper bound**

If $n! \leq n^n$ by taking logarithms $\lg(n!) \leq n \lg n$.

Theorem B.3 **Lower bound for factorial**

For every $n \geq 1$, hwe have $n! > (n/2)^{(n/2)}$.

Proof. The smallest half of the terms of the factorial are at least one (lower bound). The largest half of the terms of the factorial are at least $n/2$. Moreover $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$ and $\lfloor n \rfloor \leq n \leq \lceil n \rceil$. Therefore,

$$\begin{aligned} n! &= \overbrace{1 \cdot 2 \cdot \ldots \cdot \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil \cdot \ldots \cdot (n-1) \cdot n}^{n \text{ terms}} \\ &\geq \overbrace{1 \cdot 1 \cdot \ldots \cdot 1 \cdot \lceil n/2 \rceil \cdot \ldots \cdot \lceil n/2 \rceil \cdot \lceil n/2 \rceil}^{n \text{ terms}} \\ &\geq \overbrace{1 \ldots 1}^{\lfloor n/2 \rfloor \text{ terms}} \overbrace{\lceil n/2 \rceil \ldots \lceil n/2 \rceil}^{\lceil n/2 \rceil \text{ terms}} \\ &\geq (\lceil n/2 \rceil)^{\lceil n/2 \rceil} \\ &\geq (n/2)^{(n/2)}. \end{aligned}$$

■

Corollary B.2 **Logarithmic Lower bound**

If $n! \geq (n/2)^{(n/2)}$ by taking logarithms $\lg(n!) \geq (n/2)\lg(n/2)$.

Example B.3.1

For the floor or ceiling manipulations consider $n = 6$ with $n/2 = 3$ and $\lceil n/3 \rceil = \lfloor n/3 \rfloor = 2$.

$$6! \quad = \quad 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \geq 1 \cdot 1 \cdot 1 \cdot 3 \cdot 3 \cdot 3 = 3^3 = (6/2)^{(6/2)}$$

Likewise, for $n = 5$ with $n/2 = 2.5$ and $\lfloor 5/2 \rfloor = 2$, $\lceil 5/2 \rceil = 3$.

$$5! \quad = \quad 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \geq 1 \cdot 1 \cdot 3 \cdot 3 \cdot 3 = 3^3 \geq (5/2)^{(5/2)}$$

Theorem B.4 **Stirling's approximation formula for n!.**

For all $n \geq 10$, we have that

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right).$$

Corollary B.3 **Simplified Stirling's formula**

For $n \geq 10$ we have that $n! \approx \left(\frac{n}{e}\right)^n$.

Corollary B.4

A corollary is that $\lg(n!) = \Theta(n \lg n)$ after the Θ is formally introduced.

Proof. From either Stirling's theorem or its Corollary, we have that

$$\lg(n!) \approx n \lg n - n \lg e = \Theta(n \lg n).$$

■

B.4 Limits, Derivatives and Integrals

Fact B.2 **Limits in computing**

In computing limits are used for asymptotic considerations and are for $n \to \infty$ i.e. for asymptotically large values of some integer parameter n. The parameter n is sometimes referred to as problem size and less often as input size. Derivatives are thus $\frac{d}{dn}$.

Fact B.3 **Simple derivatives**

For any constant $k > 0$, we have that

$$(k)' = 0, \quad \left(n^k\right)' = k \cdot n^{k-1}, \quad (e^n)' = e^n, \quad (a^n)' = a^n \cdot \ln(a), \quad (\ln(n))' = \frac{1}{n},$$

$$(2^n)' = 2^n \cdot \ln(2) \approx 2^n, \quad (\lg(n))' \approx \frac{1}{n}$$

Fact B.4 **Multiplication and Division**

$$(f(n)g(n))' = f'(n)g(n) + f(n)g'(n), \qquad \left(\frac{f(n)}{g(n)}\right)' = \frac{f'(n)g(n) - f(n)g'(n)}{g^2(n)},$$

Fact B.5 **Integrals**

$$\int n^k dn = n^{k+1}/(k+1) + c, \qquad \int (1/n)dn = \ln(|n|) + c, \qquad \int (e^n)dn = e^n + c, \qquad \int (a^n)dn = a^n/\ln(a) + c,$$

Fact B.6 **Limits**

For any constant $k > 0$,

$$\lim_{n\to\infty} n = \infty, \qquad \lim_{n\to\infty} n\lg n = \infty, \qquad \lim_{n\to\infty} n^k = \infty, \qquad \lim_{n\to\infty} \lg n = \infty, \qquad \lim_{n\to\infty} 2^n = \infty$$

Fact B.7

$$\lim_{n\to\infty} (a/b)^n = \begin{cases} 0 & a < b \\ 1 & a == b \\ \infty & a > b \end{cases}$$

Fact B.8

$$\lim_{n\to\infty} \frac{2^n}{n!} = 0$$

Example B.4.1 **L'Hopital**

For infinite limits, if L'Hopital is to be applied, constants from base conversions are less of an issue and can be ignored without affecting the end result. Thus in the third equality do not spend time thinking of whether $1/n$ must be multiplied with $\ln 2$ or divided by $\ln 2$ or something else.

$$\lim_{n\to\infty} \frac{n\lg n}{n^2} = \lim_{n\to\infty} \frac{\lg n}{n} = \lim_{n\to\infty} \frac{(\lg n)'}{(n)'} = \lim_{n\to\infty} \frac{(1/n)}{1} = 0$$

B.5 Minimum and Maximum of a function: f(n)

Fact B.9 **Extremes**

Finding minima or maxima of a function involves taking the first derivative, equating it to zero and solving for the indeterminate to determine the critical point(s) of the function. Then finding the sign of the second derivative at the critical points.

Example B.5.1 **Maximum**

What is the maximum of $f(t) = t^2 + (n-1-t)^2$ in the interval $[0, n-1]$?

Proof. Find the first derivative $f'(t)$, equate it to zero and solve for the indeterminate t to determine the critical point. $f'(t) = 2t + 2(n-1-t)(-1) \Rightarrow f'(t) = 0 \Rightarrow t = (n-1)/2$.

The second derivative $f''(t) = 2 + 2(-1)(-1) = 4 > 0$. The function has a minimum at $t = (n-1)/2$. We are not interested in its minimum but rather in its maximum. It is a parabola and thus it is symmetric in the range $[0, n-1]$. The maximum would be at the extreme $t = 0$ or $t = n-1$ or there might be two maxima at $t = 0, t = n-1$. Indeed

$$f(0) = f(n-1) = 0^2 + (n-1-0)^2 = (n-1)^2 + (n-1-(n-1))^2 = (n-1)^2.$$

■

Example B.5.2 **Minimum**

What is the minimum of $f(t) = t \lg(t) + (n-1-t) \lg(n-1-t)$ in the interval $[0, n-1]$?

Proof. Find the first derivative $f'(t)$.

$$f'(t) = 1 \cdot \lg t + t \cdot (1/t) + (-1) \lg(n-1-t) + (n-1-t)(-1/(n-1-t)) = \lg(t) - \lg(n-1-t) \Rightarrow$$

$$f'(t) = \lg(t/(n-1-t)).$$

Solving for t in $f'(t) = 0$ relates to $\lg(t) - \lg(n-1-t) = 0$ i.e. $t = n-1-t \Rightarrow t = (n-1)/2$. Finding the second derivative of $f(t)$ i.e. the derivative of $\lg(t) - \lg(n-1-t)$ gives

$$f''(t) = (\lg(t) - \lg(n-1-t))' = 1/t - (-1)/(n-1-t) = 1/t + 1/(n-1-t) > 0$$

Thus the $t = (n-1)/2$ is a minimum.

As a conclusion

$$f(\frac{n-1}{2}) = (n-1) \lg((n-1)/2) = (n-1) \lg(n-1) - (n-1).$$

■

Appendix C

Collections : sets and sequences

C.1 Collections

C.1.1 Sets

A collection of elements is a grouping of its elements. Two collections are of note: **sets** and **sequences**.

Definition C.1 **Set**

A set is a collection of elements in no particular order. The elements of a set are listed one after the other in no particular order separated by commas; curly braces { and } are used to denote and define the set. Thus $\{10,30,20\}$ is a set of three elements.

Definition C.2 **Set ordering, and the belongs-to $\in$ symbol.**

In a set the order of its elements does not mater. Moreover a set does not contain duplicate elements thus $\{10,10\}$ is not a set. We say an element x belongs to A if and only if A contains x; we write $x \in A$.

Definition C.3 **Empty Set**

A set with no elements is empty. Thus $\{\}$ is an emptyset. We also denote an empty set as $\varnothing$ or $\emptyset$. Thus $\varnothing = \emptyset = \{\}$.

Definition C.4 **Set enumeration and set comprehension; ellipsis**

The elements of a set can be enumerated individually and completely as in $\{10,30,20\}$, or through a comprehension that describes a property $P(.)$ that is satisfied by them. Then the set of elements x satisfied by property P i.e. $P(x)$ is described as $\{x : P(x)\}$ or $\{x \mid P(x)\}$.

Infinite sets can use an ellipsis (three periods) as in $\{1,2,3,\ldots\}$ or $\{\ldots,1,2,3,\ldots\}$ or $\{\ldots,-2,-1,0,1,2,\ldots\}$ for example.

Definition C.5 **Cardinality of a set**

The cardinality of a set is the number of its elements. The cardinality of set A is denoted as $|A|$ or $\mathfrak{c}A$ or $\mathfrak{c}(A)$.

The empty set has carbinality zero. The cardinality of $\{10,30,20\}$ or $A = \{10,30,20\}$ is three. We write $|A| = 3$ or $\mathfrak{c}(A) = 3$ or $\mathfrak{c}A = 3$.

Definition C.6 **Set equality.**

Two sets A and B are equal to each other and we write $A = B$ or $B = A$ if and only if they have the same elements. Thus set $\{10,30,20\}$ is equal to $\{10,20,30\}$: both represent the same set containing elements 10, 20, and 30 and thus $\{10,30,20\} = \{10,20,30\}$.

Definition C.7 **Proper subsets and subsets**

A set A is called proper subset of B and denoted by $A \subset B$ if every element of A is in B and B has at least one element not in A. If the two sets are allowed to be equal we say A is a subset of B and write $A \subseteq B$. Moreovera $A \subseteq B$ and $B \subseteq A$ is equivalent to $A = B$. Not a subset is denoted by $\not\subset$ and $\not\subseteq$ as needed.

Definition C.8 **Intersection and disjoint sets.**

The intersection C of A and B is denoted by $C = A \cap B$ and denotes the set that contains the common elements of A and B. Thus $C = A \cap B = \{x : x \in A \wedge x \in B\}$. Two sets E, F are disjoint if they have no common elements thus $E \cap F = \emptyset$.

Definition C.9 **Union of sets.**

The union D of A and B is denoted by $D = A \cup B$ and denotes the set that contains all the elements of A and B, but the common ones are listed once to comply with the definition of a set. Thus $D = A \cup B = \{x : x \in A \vee x \in B\}$.

Definition C.10 **PowerSet**

The Powerset of A signified as $\mathbb{P}(A)$ or $\mathscr{P}(A)$ is the set of all possible subsets of set A. Thus $|\mathbb{P}(A)| = 2^{|A|}$.

For set A as previously defined $\mathbb{P}(A) = \{\emptyset, \{10\}, \{20\}, \{30\}, \{10,20\}, \{10,30\}, \{20,30\}, \{10,20,30\}\}$.

Definition C.11 **Complement of a set**

The complement of B sometimes denoted $\bar{B}$ or B' or $B^{\complement}$ is the set of elements not in B.

For this to make sense, there should be a reference set call it A and all sets such as B are subsets of A. Then we can define the complement of B relative to A. Formally $B^{\complement} = \{x \in A : x \notin B\}$.

Definition C.12 **(Relative) complement and Absolute complement**

For two sets A, B we define the relative complement of B relative to A to be the difference $A - B$ or $A \setminus B$ that is defined as $A - B = A \setminus B = \{x : x \in A \wedge x \notin B\}$. Thus $A - B = A \cap B^{\complement}$. The absolute complement of B relative to a reference set that contains all sets including B is $B^{\complement} = F - B$, where F is the reference set. Sometimes the latter is defined equivalently as $A + B = (A - B) \cup (B - A)$.

Definition C.13 **De Morgan's Law**

For every two sets that are subsets of a reference set F, we have $(A \cup B)^{\complement} = A^{\complement} \cap B^{\complement}$ and $(A \cap B)^{\complement} = A^{\complement} \cup B^{\complement}$.

Definition C.14 **Complement Law and Involution**

Moreover by De Morgan's Law we can also derive that $A \cup A^{\complement} = F$, $A \cap A^{\complement} = \emptyset$ and $\emptyset^{\complement} = F$ and $F^{\complement} = \emptyset$. Moreover $A \subseteq B$ implies $B^{\complement} \subseteq A^{\complement}$. Finally $A^{\complement^{\complement}} = A$.

Definition C.15 **Ordered Pair, Triples and n-tuple.**

For a pair of elements a,b order matters. Thus (a,b) means that a is the first element of the pair and b is the second. Moreover $(a,b) \neq (b,a)$. Likewise (a,b,c) and $(x_1,\ldots,x_n)$ are triples/triplets and n-tuples respectively.

Definition C.16 **Cartesian product; cartesian plane.**

For two sets A and B the cartesian product $C = A \times B$ is defined as $C = A \times B = \{(a,b) : a \in A, b \in B\}$. A cartesian product is generalizable to n sets, not just two. A cartesian plane is one where $A = B = \mathbb{R}$.

Definition C.17 **Natural numbers and Integer Numbers and Real Numbers**

The set of integer numbers $\mathbb{Z}$ includes positive and negative integers and zero. The set of Natural numbers or ordinal numbers includes the positive numbers and (but sometimes not) zero, i.e. it is the set of the non-negative integers or the set of positive integers. Thus to accommodate zero in the definition of natural numbers, we use $\mathbb{N}$ and $\mathbb{N}^+$ to denote $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. Likewise $\mathbb{R}$ is the set of real numbers.

C.1.2 Sequences

We can generalize the definition of a pair, triple or triplet, and n-tuple by defining a sequence in which elements can be repeated.

In a sequence, as opposed to a set, order matters, i.e. the elements of a sequence have some order.

Definition C.18 **Sequence**

A sequence is a collection of elements in which order matters i.e. it matters which element is first, second, etc. The elements of a sequence are listed according to their order and separated by commas; angular brackets $\langle$ and $\rangle$ are used to denote and define a sequence.

Example C.1.1

The first elements of sequence $\langle 10,30,20\rangle$ is a 10, the second a 30, and the third a 20. This sequence is different from sequence $\langle 10,20,30\rangle$. The two are different because for example the second element of the former is a 30, and the second element of the latter sequence is a 20. Thus those two sequences differ in their second element position. (They also differ in their third element position anyway.) Moreover we write $\langle 10,30,20\rangle \neq \langle 10,20,30\rangle$ to indicate that they are not the same.

Sets include unique elements; sequences not necessarily. Note the other distinction between a **set** and a **sequence**. The $\{10,10,20\}$ is incorrect as in a set each element appears only once. The correct way to write this set is $\{10,20\}$. For a sequence repetition is allowed thus $\langle 10,10,10\rangle$ is OK.

Sequences with too many elements to write down: ellipsis to the rescue. We can use the three period symbol known as an ellipsis (plural form, ellipses) that was introduced in the context of sets.

Definition C.19 **Sequence enumeration**

An infinite or finite sequence can also be described by a sequence comprehension or enumeration. For example $\langle a_i\rangle$ describes sequence $a_1,a_2,\ldots,a_n$, or summarily a_i, for $i = 1,\ldots,n$, if we drop the angular brackets.

We can define sequence $a_i = i$. This is sequence $1,2,\ldots,n$ more correctly writtens as $\langle 1,2,\ldots,n\rangle$.

Definition C.20 **Sum of sequence and product of sequences**

The elements of a sequence can be summed or multiplied. Thus

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} i$$

or inline $\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} i$ and similarly $\prod_{i=1}^{n} a_i = \prod_{i=1}^{n} i$ show how to denote this for the sequence of the previous example.

C.1.3 Sums of sequences: finite and infinite

Definition C.21 **Sums and Sigma notation**

The sum $a_0 + a_1 + \ldots + a_n$ can be represented in compact form as

$$\sum_{i=0}^{i=n} a_i = \sum_{i=0}^{n} a_i = a_0 + \ldots + a_n$$

Variable i has values that vary between a smallest value as indicated under the sum's Sigma symbol and it is $i = 0$ in this example and its largest value as indicated over the sum's Sigma symbol and it is $i = n$). It also assumes all integer number values between 0 and n (inclusive of the end points). The variable's name is available beneath the Sigma and can be omitted over it as shown in the second formulation of the sum. The terms of the sum are usually members of a sequence and in this case $a_0, a_1, \ldots, a_n$. The general member of the sequence is a_i as described in the sigma / sum formulation. If the sequence is simple such as $a_i = i$ instead of a_i we use directly i; likewise for $a_i = i^2$, $a_i = i^3$.

Definition C.22

Let

$$S_k(n) = 1^k + 2^k + \ldots + n^k = \sum_{i=1}^{n} i^k$$

for any integer $k > 0$.

The latter term $\sum_{i=1}^{n} i^k$ indicates a sum that runs from $i = 1$ through (inclusive) $i = n$. The i-th term of the sum is i^k.

Fact C.1 **Some constants**

The base e of the Neperian (also known as natural) logarithms, π, Euler's gamma constant, Golden ratio phi ϕ and its Fibonacci-related counter-part are shown.

$$e \approx 2.718281, \quad \pi \approx 3.14159, \quad \gamma \approx 0.57721, \quad \phi = \frac{1+\sqrt{5}}{2} \approx 1.61803, \quad \hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -.61803.$$

Theorem C.1 **Binomial Theorem**

For all $n \in \mathbb{N}$ and a, b we have

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{k} a^k b^{n-k} = a^n + na^{n-1}b + \frac{n(n-1)}{2} a^{n-2}b^2 + \ldots + \binom{n}{k} a^k b^{n-k} + \ldots + nab^{n-1} + b^n.$$

A series is the sum of a sequence.

Fact C.2 **Arithmetic series**

The arithmetic series is the sum of the (terms of the) arithmetic sequence i.e. the sum of the consecutive integers starting from one (or zero) to an upper bound usually denoted by n. A closed-form expression is shown.

$$A(n) = A_n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Fact C.3 **Quadratic and Cubic series**

They are defined analogously. A closed-form expression is shown.

$$Q(n) = Q_n = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \qquad C(n) = C_n = \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}.$$

Fact C.4 **Harmonic series**

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \int (1/x)dx \approx \ln(n) + \gamma.$$

Fact C.5 **Derivative formulae from harmonic series**

Let $H_n = \sum_{i=1}^{n} 1/i$. Then we have the following equations.

$$\sum_{i=1}^{n} H_i = (n+1)H_n - n, \qquad \sum_{i=1}^{n} iH_i = \frac{n(n+1)}{2}H_n - \frac{n(n-1)}{4}.$$

The following identities can be derived from the following theorem that is proved by induction, as sppecial cases for $n = 2$ and $n = 3$, and with the third identity deriving from the second by substituting $-b$ for b.

Fact C.6

For all a, b

$$a^2 - b^2 = (a-b)(a+b), \qquad a^3 - b^3 = (a-b)(a^2+ab+b^2), \qquad a^3 + b^3 = (a+b)(a^2-ab+b^2).$$

Theorem C.2

For all $n \in \mathbb{N}$ and a, b we have

$$(a^n - b^n) = (a-b)\left(a^{n-1} + a^{n-2}b + a^{n-3}b^2 + \ldots + ab^{n-2} + b\right).$$

Theorem C.3 **Geometric series**

For all $n \in \mathbb{N}$ and a we have

$$(a^n - 1) = (a-1)\left(a^{n-1} + a^{n-2} + a^{n-3} + \ldots + a + 1\right).$$

This is the previous theorem for $b = 1$ and can be rewritten in the form of the geometric series provided that $a \neq 1$.

$$G(n,a) = G_n^a = \sum_{i=0}^{n-1} a^i = 1 + a + a^2 + \ldots + a^{n-1} = \frac{a^n - 1}{a-1}.$$

Theorem C.4 **Infinite geometric series**

If $|a| < 1$ to the limit $n \to \infty$ we have the following

$$\lim_{n\to\infty} 1 + a + a^2 + \ldots + a^{n-1} = \lim_{n\to\infty} \frac{a^n - 1}{a-1} = \frac{1}{1-a}.$$

Therefore

$$G(a) = G^a = \sum_{i=0}^{\infty} a^i = \frac{1}{1-a}.$$

Fact C.7

For $a \neq 1$ we have the following finite sum.

$$I(n,a) = I_n^a = \sum_{i=0}^{n-1} i a^i = \frac{(n-1)a^{n+1} - na^n + a}{(1-a)^2}.$$

Correspondingly, the infinite sum is derived as follows.

Fact C.8

For $|a| < 1$ we have the following sum.

$$I(a) = I^a = \sum_{i=1}^{\infty} i a^i = \frac{a}{(1-a)^2}.$$

Corollary C.1 $I(n, 1/2)$

Moreover, we have that for $I(n, 1/2)$,

$$I(n, 1/2) = \sum_{i=0}^{n-1} i/2^i = 2 - \frac{2n+2}{2^n}.$$

$$I(1/2) = \sum_{i=0}^{\infty} i/2^i = 2.$$

Fact C.9

For $|a| < 1$.

$$e^a = 1 + a + \frac{a^2}{2!} + \ldots + \frac{a^i}{i!} + \ldots = \sum_{i=0}^{\infty} \frac{a^i}{i!}.$$

$$\ln(1+a) = a - \frac{a^2}{2} + \ldots + \frac{(-1)^{i+1} a^i}{i} + \ldots = \sum_{i=1}^{\infty} (-1)^{i+1} \frac{a^i}{i}.$$

$$\ln \frac{1}{1-a} = a + \frac{a^2}{2} + \ldots + \frac{a^i}{i} + \ldots = \sum_{i=1}^{\infty} \frac{a^i}{i}.$$

Fact C.10

For $|a| < 1$, and F_i the Fibonacci sequence,

$$\frac{1}{(1-a)^{n+1}} = \sum_{i=0}^{\infty} \binom{i+n}{i} a^i, \qquad \frac{1}{\sqrt{1-4a}} = \sum_{i=0}^{\infty} \binom{2i}{i} a^i, \quad \frac{a}{1-a-a^2} = a + a^2 + 2a^3 + 3a^4 + \ldots = \sum_{i=0}^{\infty} F_i a^i$$

Fact C.11

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \qquad \sum_{k=0}^{n} \binom{n}{k} = 2^n, \qquad \binom{n}{k} = \binom{n}{n-k}, \qquad \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

Proof. **(Theorem C.2)** Since $A(n) = A_n = 1 + 2 + \ldots + (n-1) + n$. Writing A_n forwards and backwards we add up the corresponding terms.

$$\begin{aligned}
A_n &= 1 + 2 + \ldots + (n-1) + n \\
A_n &= n + (n-1) + \ldots + 2 + 1 \quad \text{Add up this and the previous equation} \\
2A_n &= (n+1) + (n+1) + \ldots + (n+1) + (n+1) \\
2A_n &= n(n+1) \\
A_n &= n(n+1)/2
\end{aligned}$$

■

Proof. **(Theorem C.2,Theorem C.3)** We can use the following method to find sums of the following form

$$S_k = \sum_{i=1}^{n} i^k.$$

First consider $(i+1)^{k+1}$ and expand it. Substitute in the expansion $i = 1$, $i = 2, \ldots$, $i = n$, a total of n times and write the resulting n equalities one after the other. Then, sum these n equalities by summing up the left hand sides and the right hand sides. Solve for S_k and S_k can then be found as a function of n.

For the sum in question $k=1$. Therefore we consider

$$(i+1)^2 = i^2+2i+1$$

We substitute for $i=1,2,\ldots,n$ writing one equality after the other

$$\begin{aligned}
(1+1)^2 &= 1^2+2\cdot 1+1\\
(2+1)^2 &= 2^2+2\cdot 2+1\\
(3+1)^2 &= 3^2+2\cdot 3+1\\
(4+1)^2 &= 4^2+2\cdot 4+1\\
\ldots &= \ldots\\
(n+1)^2 &= n^2+2\cdot n+1
\end{aligned}$$

When we sum up the n equalities we realize that say, $(3+1)^2$ of the third line is equal to 4^2 of the fourth line and therefore.

$$(1+1)^2+(2+1)^2+\ldots+(n+1)^2 = (1^2+2^2+3^2+\ldots+n^2)+2\cdot(1+2+\ldots+n)+(1+\ldots+1)$$

We note that $2\cdot(1+2+\ldots+n)=2S_1$ and $(1+\ldots+1)=n$ (number of ones is number of equations). Then,

$$(n+1)^2 = 1+2S_1+n$$

Solving for S_1 we get that $S_1=((n+1)^2-n-1)/2$, ie $S_1=(n^2+2n+1-n-1)/2=(n^2+n)/2=n(n+1)/2$, which is $A(n)=A_n$. ■

Proof. **(Fact C.8, Corollary C.1)** We show that

$$I(1/2)=I^{1/2}=\sum_{i=0}^{\infty} i/2^i = 2.$$

We start with the geometric series and in particular the infinite geometric series. Getting its first derivative with respect to a and then multiplying both sides with a yields, almost, the result. In the last step we substitute $1/2$ for a. In all cases $|a|<1$.

$$\begin{aligned}
G(n,a)=\sum_{i=0}^{n-1} a^i &= 1+a+\ldots+a^i+\ldots+a^{n-1}=\frac{a^n-1}{a-1}\\
G(a)=\sum_{i=0}^{\infty} a^i &= 1+a+\ldots+a^i+\ldots=\frac{1}{1-a}\\
I(a)=\sum_{i=0}^{\infty} i\cdot a^i &= 1\cdot a^1+2\cdot a^2+\ldots+i\cdot a^i+\ldots=?
\end{aligned}$$

For any $|a| < 1$, we have.

$$\begin{aligned}
\sum_{i=0}^{\infty} a^i &= \frac{1}{1-a} \\
1 + a + \ldots + a^i + \ldots &= \frac{1}{1-a} \\
\left(1 + a + \ldots + a^i + \ldots\right)' &= \left(\frac{1}{1-a}\right)' \\
0 + 1 \cdot a^0 + \ldots + i \cdot a^{i-1} + \ldots &= \frac{1}{(1-a)^2} \\
0 \cdot a + 1 \cdot a^1 + \ldots + i \cdot a^i + \ldots &= \frac{a}{(1-a)^2} \\
I(a) = \sum_{i=0}^{\infty} i \cdot a^i &= \frac{a}{(1-a)^2}
\end{aligned}$$

From the last one for $a = 1/2$ we get $I(1/2) = 2$. ■

Appendix D

Comparison of functions

D.1 Comparison of (discrete or continuous) functions

For a discrete function the indeterminate (variable, uknown) will more often than not be denoted by n. For continuous functions by x. Thus $A(n)$ is a discrete function and $B(x)$ is a continuous variable function (unless otherwise noted). There are several ways one can use to comparae two functions $f(n)$ and $g(n)$.

Method D.1 **Using direct or indirect methods and transitivity: from $F(n$ to $G(n))$**

One can compare two functions $F(n)$ and $G(n)$, by using inequalities and starting with one (say, $F(n)$) the other (i.e. $G(n)$) is derived (see previous page). Or indirectly as follows.

(1) Show first that $F(n) < F_1(n)$ where $F_1(n)$ is an 'easier to deal' function.

(2) Then show that $G_1(n) < G(n)$ likewise.

(3) Then show also directly or using latter methods or indirectly (recursively) that $F_1(n) < G_1(n)$.

(4) Two applications of transitivity show that $F(n) < F_1(n)$ and $F_1(n) < G_1(n)$ imply $F(n) < G_1(n)$. This and $G_1(n) < G(n)$ imply $F(n) < G(n)$.

Method D.2 **Difference $F(n - G(n))$**

One can compare two functions $F(n)$ and $G(n)$, by taking their difference $F(n) - G(n)$ (or $G(n) - F(n)$) and determining the sign of the difference.

Method D.3 **Ratio $F(n/G(n))$**

One can compare two functions $F(n)$ and $G(n)$, by taking their difference $F(n)/G(n)$ (or $G(n)/F(n)$) and determining whether it is greater, equal or less than one.

Method D.4 **Raise using common base and then compare i.e. $2^{F(n)} : 2^{G(n)}$**

One can compare two functions $F(n)$ and $G(n)$, by comparing $2^{F(n)}$ and $2^{G(n)}$.

Method D.5 **Take logarithms to same base and then compare i.e. $\lg F(n) : \lg G(n)$**

One can compare two functions $F(n)$ and $G(n)$, by comparing $\lg F(n)$ to $\lg G(n)$.

Example D.1.1 **Transitivity**

The two functions $f(n) = 25n^2$ and $g(n) = 25n^2 + 10$ can be compared by starting say from $f(n)$

$$f(n) = 25n^2 < 25n^2 + 1 < 25n^2 + 2 < \ldots < 25n^2 + 10 = g(n)$$

(Transitivity is implicitly used $A < B < C$ means $A < B$ and $B < C$ and thus by transitivity $A < C$.)

Example D.1.2 **Difference**

The two functions $f(n) = 25n^2$ and $g(n) = 25n^2 + 10$ can be compared by taking the difference $g(n) - f(n) = 25n^2 + 10 - 25n^2 = 10 > 0$ and observing that $g(n) - f(n)$ is positive i.e. $g(n) > f(n)$. Equivalently $f(n) - g(n) < 0$ and also $f(n) < g(n)$.

Example D.1.3 **Ratio**

The two functions $f(n) = 25n^2$ and $g(n) = 25n^2 + 10$ can be compared by taking the ratio $g(n)/f(n) = (25n^2 + 10)/25n^2 \geq 1$ and observing that $g(n)/f(n) > 1$ i.e. $g(n) > f(n)$. Equivalently $f(n)/g(n) < 1$ and also $f(n) < g(n)$.

Example D.1.4 **Exponentiation**

Let $a(n) = \ln(25n^2)$ and $b(n) = \ln(25n^2 + 10)$. Then $f(n) = e^{a(n)} = 25n^2$ and $g(n) = e^{b(n)} = 25n^2 + 10$ The resulting functions $f(n)$ and $g(n)$ have already been compared and found to be such that $f(n) < g(n)$. Thus $a(n) < b(n)$ as well by monotonicity!

Example D.1.5 **Logarithms**

The two functions $f(n) = 25n^2$ and $g(n) = 25n^2 + 10$ can be compared by taking their logarithms $a(n) = \ln(25n^2)$ and $b(n) = \ln(25n^2 + 10)$. We just proved above that $a(n) < b(n)$ and thus $f(n) < g(n)$ by monotonicity. (Admittedly, not a very interesting example!)

Example D.1.6

Compare $n^{1/\lg n}$ and 2.

Proof. See also Exercise B.2.3 for solution or

$$\begin{aligned} n^{1/\lg n} &: 2 \\ \lg(n^{1/\lg n}) &: \lg 2 \\ (1/\lg n) \cdot \lg n &: 1 \\ 1 &: 1. \end{aligned}$$

Obviously $1 = 1$ and this $n^{1/\lg n}$ is equal to 2. ■

Fact D.1 **Polynomial functions**

A function n^m for any positive constant integer $m > 0$ is a polynomial in n i.e. a polynomial function. The linear combination of polynomial functions is also a polynomial function.

Fact D.2 **Polylogarithmic functions**

A function $\lg^m n$ for any positive constant integer $m > 0$ means $(\lg n)^m$ and is a polylogarithmic function.

Fact D.3 **Large enough means asymptotically large**

The expression "**for large enough** n " means "**there is a positive constant** n_0 **such that for all** $n > n_0$ ", that is for "asymptotically large values of n".

Fact D.4 **Polynomial vs Polylogarithmic**

For any positive constant integer $k, m > 0$ and integer $n > 0$, we have that $n^m > \lg^k n$ for large enough n. *This means that every polynomial function is asymptotically larger than any logarithmic function of the same variable n.*

Fact D.5 **Exponential vs Polynomial**

For any positive constant integer m and integer $n > 0$, we have that $2^n > n^m$ for large enough n. *This means that every exponential function (base two) is asymptotically larger than any polynomial function of the same variable n.*

The results above are true evan for non-integer but positive constant values for k, m. Any such constant value is bounded between two integer constant values.

Fact D.6 **Linear, Log-linear, Quadratic, Cubic**

A linear functions is n, a quadratic function is n^2 and cubic n^3. A log-linear functions is $n \lg n$ the product of a linear and a logarithmic function. A linear function is asymptotically smaller than a log-linear function which is likewise smaller than a quadratic function and likewise (asymptotically) smaller than a cubic function.

A log-linear function can be considered a polynomial function as $n < n \lg n < n^2$ for large enough n. Note also that $n \lg n = n^{1+\lg\lg n / \lg n}$.

Appendix E

Mathematical induction

E.1 Mathematical induction

Every proposition is true or false with reference to a space we first define axiomatically and then build on by establishing more theorems.

Definition E.1 **Axioms of Arithmetic: Peano's axioms**

Peano's axioms, on whose inductive proofs are based, define natural numbers. The set of natural numbers is denoted by N or $\mathbb{N}$.

Axiom 1 (Peano). 0 is a natural number.

Axiom 2 (Peano). If n is a natural number, then its successor $s(n)$ is also a natural number.
(We prefer to write $n+1$ for the successor $s(n)$ of n.)

Theorems in mathematics are true because the space to which these theorems apply are based on simple axioms that are usually true.

Theorem E.1 **Well Ordered Set Principle**

Every non-empty subset of $\mathbb{N}$ has a minimal element.

Theorem E.2 **Mathematical Induction**

Let $A \subseteq N$. Let $1 \in A$ and whenever $k \in A$ then $k+1 \in A$. This implies $A = N$.

Proof. Let $A^{\complement}$ be the complement of A over $\mathbb{N}$.

$$A^{\complement} = N - A = \{k \in N | k \notin A\}.$$

Suppose $A \neq N$. Then $A^{\complement}$ is a non-empty set and by the well ordered set principle it has a minimal element and call that element a. Since $1 \in A$ it is obviously $a \neq 1$. This mean $a > 1$ and thus $a-1$ is positive. Because a is the mimimal element of $A^{\complement}$, $a-1$ belongs to A. From the mathematical induction principle set $k = a-1$. $k \in A$ implies $k+1 = a \in A$. This contradicts the membership of a in the complement of A! Thus $A^{\complement}$ cannot have a minimal element i.e. it must be empty. This implies that $A = N$. ■

In inductive proofs we try to prove that the set of integers A that satisfy a given property or proposition is all of $\mathbb{N}$ i.e. $A = \mathbb{N}$.

Theorem E.3

If $a,b \in \mathbb{N}$ then $ab \geq a$. Equality is applicable if and only if $b = 1$.

Proof. Use induction on b. Let $a \in \mathbb{N}$.

STEP1: Base case. is $b = 1$. Clearly $a \cdot 1 \geq a$ as the former is a by way of $a \cdot 1 = a$; moreover $a \cdot 1 = a$.
Auxiliary step. For the inductive step, since $a \geq 1$, we have $a > 0$ and thus $2a = a + a > a + 0$, i.e. $2a > a$.

STEP 2: Inductive hypothesis. Suppose that $ab > a$ for some $b \in \mathbb{N}$.

STEP 3: Inductive step. We then have to show that $a(b+1) > a$ to prove the inductive step. This is

$$a(b+1) = ab + a > a + a > a$$

The first $>$ is by the inductive hypothesis; the second $>$ is by the auxiliary step where $a > 0$ implies $2a > a$.

STEP 4: Conclusion. It follows that $ab > a$ for all $b \geq 2$. (And of course $ab = a$ for $b = 1$ thus concluding the theorem.) ■

Theorem E.4 **Strong Induction**

Let $A \subseteq N$. Let $1 \in A$ and whenever $\{1,2,\ldots,k\} \subseteq A$ then $k+1 \in A$. This implies $A = N$.

Strong vs Weak form of induction. Given that Theorem E.4 establishes strong induction, we sometimes refer to Theorem E.2 as weak induction. Thus induction or mathematical induction or weak induction or also ordinary induction are all synonymous and refer to Theorem E.2. There is one and only one name for strong induction of Theorem E.4.

Induction as a proof method. In inductive proofs we try to prove that the set of integers A that satisfy a given property or proposition is all of $\mathbb{N}$ i.e. $A = \mathbb{N}$. In the remainder we ignore properties and focus on propositions. Of particular interest are proposition that depends on an integer variable n and thus we would establish that the range A of n is indeed $\mathbb{N}$ i.e. $A = \mathbb{N}$. Note that we will use the term "integer variable" as a misnomer for natural number. A natural number of always a positive integer, whereas our "integer variable" would allow the possibility of including 0 (but not negative numbers). Thus "integer variable" is an alias for "non-negative integer", "natural numbers", or "natural numbers including 0".

To wrap up all these assumptions let $P(n)$ be a generic proposition that depends on integer variable n. Whether we call the indeterminate k or n or p does not matter. What it matters is its range of values which is usually all of N (i.e. $\mathbb{N}$).

Note E.1.1

(Mathematical Induction: the principle). Let $P(n)$ be a proposition that depends on a natural (integer) number n. Weak induction is structured as follows.

(Base case): $P(0)$ is true, and
(Inductive Step): $P(n)$ implies $P(n+1)$ for all natural numbers n, then
(Conclusion): $P(n)$ is true for all natural numbers n, including 0 (base case value).

Note E.1.2

(Why does induction work?) The base case establishes $P(0)$, i.e. the truthness of $P(n)$ for the base case value: the base case value is the smallest value for which $P(.)$ can be proven true. For this generic description 0 is the base case value and thus for the base case we show $P(0)$ is true. (Repetition was intentional in the last two sentences.) The Inductive step establishes the chain of implications,

$$P(0) \Rightarrow P(1), \quad P(1) \Rightarrow P(2), \quad P(2) \Rightarrow P(3), \quad \ldots \quad , P(n) \Rightarrow P(n+1),$$

To fire-up the chain reaction of implications, we start with the left-hand side of the first implication that contains $P(0)$. The base case established the truthness of $P(0)$. The first implication then established the truthness of $P(1)$, the second of $P(2)$ and so on! Collectively we say that $P(n)$ is true for all $n \geq 0$. This is equivalent to saying $0 \in A$, $1 \in A$, and so using the terminology of Theorem E.2.

Note E.1.3

(**Strong Induction**) Let $P(n)$ be a proposition that depends on a natural (integer) number n. Strong induction is structured as follows.

(Base case): $P(0)$ is true, and
(Inductive Step): $P(0), P(1), \ldots, P(n)$ jointly imply $P(n+1)$ for all natural numbers n, then
(Conclusion): $P(n)$ is true for all natural numbers n, including 0.

Note E.1.4

(A base case other than $P(0)$). If $P(0)$ cannot be proven true, find the smallest value a, where a is a natural number such that $P(a)$ is true. The conclusion would then be true for all $P(k)$ such that $k \geq a$. For Strong induction the changes for the Inductive step and Conclusion are shown below.

(Base case): $P(a)$ is true, and
(Inductive Step): $P(a), P(a+1), \ldots, P(n)$ jointly imply $P(n+1)$ for all natural numbers $n \geq a$, then
(Conclusion): $P(n)$ is true for all natural numbers $n \geq a$.

Let $S_k(n) = \sum_{i=1}^{n} i^k$. We shall show that $S_1(n) = 1 + \ldots + n$ is equal to $n(n+1)/2$. We sometimes denote $S_1(n)$ as $A(n)$ for the sum of the terms of the arithmetic sequence.

Example E.1.1

(Arithmetic sequence sum.)

Proposition E.1

For all natural number $n \geq 1$, we have that $S_1(n) = 1 + 2 + \ldots + n = n(n+1)/2$.

The first step in induction is to identify in a proposition a predicate that depends on a natural-valued variable and also that same natural-valued variable.

Proof. **Let us call $P(n)$ the predicate $S_1(n) = n(n+1)/2$.** We are going to show that $P(n)$ is true for all (integer $n \geq 1$). That is we shall show that $S_1(n) = n(n+1)/2$. The proof is by induction.

1. Base case: Show that $P(1)$ is true. Thus we shall show that $S_1(n)|_{n=1} = n(n+1)/2|_{n=1}$. The left hand side sum of $P(1)$ is $S_1(1)$ i.e. the sum of one term (and that term is 1), which is 1. The right hand side of $P(1)$

is $1(1+1)/2$ which is also 1. Therefore $P(1)$ is true since the left and right hand sides of $=$ are equal to one and equal to each other obviously.

2. Inductive Step: $\forall n \in N\ P(n) \Rightarrow P(n+1)$**.** We need to establish the implication i.e. $P(n) \Rightarrow P(n+1)$ for all $n \geq 1$, i.e. we need to show that $P(n)$ implies $P(n+1)$. We move from the left hand of the implication to the right hand side.

2.a Induction hypothesis: $P(n)$**.** $P(n)$ true, implies $S_1(n) = 1+2+\ldots+n = \sum_{i=1}^{n} i = n(n+1)/2$.

2.b Is $P(n+1)$ **implied?** To show $P(n+1)$ we need to show that $S_1(n+1) = 1+2+\ldots+(n+1) = \sum_{i=1}^{n+1} i = (n+1)(n+2)/2$.

Show 2.b using 2.a. We start from the left-hand side of the latter equality to derive the right-hand side utilizing the induction hypothesis, i.e. the assumption that $P(n)$ is true which is equivalent to $S_1(n) = n(n+1)/2$: we use to derive the third equality from the second one.

$$\begin{aligned} 1+2+\ldots+(n+1) &= \sum_{i=1}^{n+1} i = \left(\sum_{i=1}^{n} i\right) + (n+1) \\ &= n(n+1)/2 + (n+1) = n(n+1)/2 + 2(n+1)/2 = (n+1)(n+2)/2 \end{aligned}$$

This completes the induction. We proved two things

- We first proved that $P(1)$ is true.
- and then showed $P(n) \Rightarrow P(n+1)$ for all $n \geq 1$.

■

An alternative would have been to show (substituting $n-1$ for n)

- that $P(1)$ is true,
- and then show that $P(n-1) \Rightarrow P(n)$ for all $n \geq 2$.

Example E.1.2

(Geometric sequence sum.) The sum below is known as the geometric series or sum of the geometric sequence. The i-th term of the sum is x^i. (We use m to deviate from the standard notation.)

Theorem E.5

Show that for all integer $m \geq 0$, and for any $x \neq 1$,

$$\sum_{i=0}^{m} x^i = 1+x+\ldots+x^m = \frac{x^{m+1}-1}{x-1}.$$

Proof. We prove the theorem **by induction**. The natural variable in the theorem is m. The predicate $P(m)$ in the theorem that depends on m is

$P(m)$: $\quad 1+x+\ldots+x^m = \frac{x^{m+1}-1}{x-1}$, for any $x \neq 1$.

An equivalent way is to write

$P(m)$: $\quad \sum_{i=0}^{m} x^i = \frac{x^{m+1}-1}{x-1}$, for any $x \neq 1$.

1. Base case: $P(1)$ is true. We can show either $P(0)$ or $P(1)$. To show $P(1)$ we need to show that $1+x$ is equal to $\frac{x^{1+1}-1}{x-1}$ which is obviously true as the latter is a stealthy way of writing $(x^2-1)/(x-1)$ which is indeed $x+1$ as long $x \neq 1$ which is so anyway.

2. Inductive Step: $P(m) \Rightarrow P(m+1)$. Let us show first what $P(m)$ is/implies, i.e. the induction hypothesis. It is equivalent to

$$P(m): \qquad 1+x+\ldots+x^m = \frac{x^{m+1}-1}{x-1}$$

Then $P(m+1)$ is equivalent to

$$P(m+1): \qquad 1+x+\ldots+x^{m+1} = \frac{x^{m+2}-1}{x-1}$$

Similarly to the previous example E.1.1 we start from the latter's left-hand side to conclude its right hand-side by using the former result for $P(m)$. Between the second and third inequality we use the induction hypothesis for $P(m)$ above.

$$\begin{aligned} 1+x+\ldots+x^{m+1} &= 1+x+\ldots+x^m+x^{m+1} = (1+x+\ldots+x^m)+x^{m+1} = \frac{x^{m+1}-1}{x-1}+x^{m+1} \\ &= \frac{(x^{m+1}-1)+x^{m+1}(x-1)}{x-1} = \frac{x^{m+2}-1}{x-1}, \end{aligned}$$

which proves that $P(m+1)$ is true. Base case and inductive step conclude the induction. ■

Just because we set up the induction description by using n to represent the natural number, it doesn't mean that we should always use n or have n in a Proposition. In this example m was the natural number, and the proposition dependend on m (i.e. we had $P(m)$). Moreover the base case use was $P(1)$; an alternative could have been $P(0)$.

A **recursive function** is a function that invokes itself. In **direct recursion** a recursive function f invokes directly itself, whereas in **indirect recursion** function f invokes function g that invokes f. A quite well-known recursive function from discrete mathematics is the Fibonacci function F_n or more widely know as the Fibonacci Sequence F_n. The name sequence implies that it includes all the terms $F_0, F_1, \ldots, F_{n-1}, F_n, \ldots$. The n indexed term is given by the following recursive formulation.

$$F_n = F_{n-1} + F_{n-2} \text{ if } n > 1$$

where

$$F_0 = 0 \text{ and } F_1 = 1$$

Example E.1.3 **(Strong induction example.)**

Proposition E.2 **Fibonacci sequence**

For any $n \geq 0$, we have that $F_n \leq 2^n$.

Proof. Proof is by strong induction. Let the predicate $P(n)$ be $F_n \leq 2^n$. Given that we have F_0 we shall use a base case $n = 0$. Otherwise, if we use a $n = 1$ base case, we could prove a lesser result that for $n \geq 1$ $P(n)$ is true.

1. Base case: Show $P(0)$ is true. To show $P(0)$ we need to show that $F_0 \leq 2^0$. Since F_0 is 0 and $2^0 = 1$ we have obviously $F_0 \leq 1 = 2^0$, and $P(0)$ follows directly.

The base case $n = 0$ was easy to prove to establish $P(0)$. No need to move on to $P(1)$.

2. Inductive Step. Show that $P(0) \wedge \ldots \wedge P(n-1) \Rightarrow P(n)$. The inductive step is of the (strong induction) type "$0, 1, \ldots, n-1$, induces n" and not of the type "$1, 2, \ldots, n$ induces $n+1$".

2.a Induction hypothesis: $P(0) \wedge \ldots \wedge P(n-1)$ **which is equivalent to** $\forall i, 0 \leq i < n : P(i)$**.**

We need to show that $\forall i, 0 \leq i < n : P(i) \Rightarrow P(n)$. $P(i)$ is equivalent to $F_i \leq 2^i$. Among the i less than n two are of use: $P(n-1)$ and $P(n-2)$ that relate to F_{n-1} and F_{n-2} respectively.

$$P(n-2): \qquad F_{n-2} \leq 2^{n-2} \qquad \text{and} \qquad P(n-1): \qquad F_{n-1} \leq 2^{n-1}$$

We complete the inductive step by starting with the recurrence and then claiming $P(n-1)$ and $P(n-2)$ for F_{n-1} and F_{n-2} in its right hand side; this involves turning an equal into less than or equal.

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \leq 2^{n-1} + F_{n-2} \leq 2^{n-1} + 2^{n-2} \\ &\leq 2^{n-1} + 2^{n-1} \\ &= 2 \cdot 2^{n-1} \\ &= 2^n \end{aligned}$$

This completes the strong induction. ■

Appendix F

Asymptotic notation

F.1 Asymptotic Comparison: informal and error-prone approach

Definition F.1 **Asymptotic comparison of functions: an informal overview**

An asymptotic comparison of two functions is not an exact comparison. We do not care to establish what happens between $f(n)$ and $g(n)$ for all values of n. We care only how the two functions compare asymptotically i.e. for large n. This means that asymptotic comparisons is of interest for $n \to \infty$.

Definition F.2 **Asymptotic comparison of functions: an informal definition**

For two functions $f(n)$ and $g(n)$ that are always positive, the $\lim_{n\to\infty} f(n)/g(n)$ establishes whether $f(n)$ is asymptotically larger, equal or smaller than $g(n)$ depending on whether the limit is ∞, a constant other than zero , and zero respectively.

Definition F.3 **Informal (no limits asymptotic comparison)**

A "naive way" to compare asymptotically two functions $f(n), g(n)$ involves eliminating low-order terms and multiplicative constant terms. The left-overs say $F(n)$ and $G(n)$ are then directly compared using $<$ or $>$ or $=$ or $\leq$ and $\geq$. A direct comparison of the $F(n)$ and $G(n)$ does not imply a same-like comparison for $f(n)$ and $g(n)$: only an asymptotic comparison is then possible. Therefore special symbols are introduced as asymptotic equivalents to $<, >, =, \leq, \geq$. These are $o, \omega, \Theta, O, \Omega$ respectively.

S1. Eliminate low order terms first	$f(n) = 256n^2 + 20n + 3$	:	$g(n) = 128n^2 - 3$
Starting with constant terms first	$256n^2 + 20n$	:	$128n^2$
Then with higher order terms	$256n^2$	:	$128n^2$
S2. Multiplicative constants become 1	$1 \cdot n^2$	:	$1 \cdot n^2$
S3. Compare directly the leftovers	$F(n) = 1 \cdot n^2$	$=$	$G(n) = 1 \cdot n^2$
And from column 2 below find	$F(n)$	$=$	$G(n)$
S4. asymptotic symbol in column 1	$f(n)$	$=$	$\Theta(G(n))$
S4. (alternative formulation)	$f(n)$	$\in$	$\Theta(G(n))$

AsymptoticComparison	DirectComparison
$f(n) : g(n)$	$F(n) : G(n)$
Θ	$=$
O	$\leq$
Ω	$\geq$
o	$<$
ω	$>$

F.2 Asymptotic Comparison: formal definitions

Two functions $f(n)$ and $g(n)$ are formally asymptotically compared as follows

Definition F.4 **Asymptotic comparison of $f(n$ and $g(n)$ using o, ω, Θ)**

In order to asymptotically compare two always positive functions $f(n)$ and $g(n)$ with the three symbols o, ω, Θ, we first find the limit $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ and then determine which case is applicable as follows. (c is a positive constant).

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \Rightarrow f(n) = o(g(n)) \\ \infty & \Rightarrow f(n) = \omega(g(n)) \\ c \neq 0 & \Rightarrow f(n) = \Theta(g(n)) \end{cases}$$

Corollary F.1 **$o \Rightarrow O$ implication**

If $f(n) = o(g(n))$, then $f(n) = O(g(n))$.

Corollary F.2 **$\omega \Rightarrow \Omega$ implication**

If $f(n) = \omega(g(n))$, then $f(n) = \Omega(g(n))$.

Corollary F.3 **$\Theta \Leftrightarrow \Omega \bigwedge O$**

$f(n) = \Theta(g(n))$, if and only if $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$.

Fact F.1 **Asymptotically smaller, equal, large**

When $f(n) = o(g(n))$ we say that $f(n)$ is asymptotically smaller than $g(n)$.
When $f(n) = \omega(g(n))$ we say that $f(n)$ is asymptotically larger than $g(n)$.
When $f(n) = \Theta(g(n))$ we say that $f(n)$ is asymptotically equal to $g(n)$.
Naturally in the former case $g(n)$ is asymptotically larger than $f(n)$, in the second case above $g(n)$ is asymptotically smaller than $f(n)$. In the third case $g(n)$ is also asymptotically equal to $f(n)$.

Fact F.2 **Comparison vs Asymptotic comparison**

When we compare two functions $f(n)$ and $g(n)$ we directly compare them using $<, >$ etc. When we asymptotically compare two functions $f(n)$ and $g(n)$ we take the limit of $\lim_{n\to\infty} f(n)/g(n)$ and then we use the asymptotic symbols o, ω, Θ depending on whether the limit is $0, \infty$ or some non-zero (and positive) constant.

Example F.2.1 **Asymptotically equal does not mean equal**

5 and 6 as number are not the same, 5 is smaller than 6 i.e. 5 is not equal to 6. But $f(n) = 5$ and $g(n) = 6$ means that '5' and '6' are both constant functions. They are asymptotically equal as $\lim_{n\to\infty} 5/6$ is a non-zero constant. Thus functions 5 and 6 are asymptotically equal (and by functions we mean $f(n) = 5$ and $g(n) = 6$).

Example F.2.2

Functions $5n$ and $6n$ are asymptotically equal. Their limit is 5/6 (or 6/5 the other way around). They are both linear functions.

Example F.2.3

Functions $5n^2$ and $6n^2$ are asymptotically equal. Their limit is 5/6 (or 6/5 the other way around). They are both quadratic functions.

Example F.2.4

Function n is asymptoticall larger than $1,000,000,000,000$. Their limit is ∞. (It does not matter what happens if $n = 1$ or $n = 1000$ or $n = 1,000,000$. It only matters what happens for $n \to \infty$.)

F.3 Asymptotic notation

Definition F.5 **Little-oh**

$f(n) = o(g(n))$, iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Definition F.6 **Little-omega**

$f(n) = \omega(g(n))$, iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Definition F.7 **Big-Oh**

$f(n) = O(g(n))$ iff $\exists$ positive constants $c_2, n_0 : \ 0 \le f(n) \le c_2 g(n) \ \forall\, n \ge n_0$.

Corollary F.4 **Big-Oh: If little-oh then Big-Oh.**

If $f(n) = o(g(n))$, then $f(n) = O(g(n))$.

Definition F.8 **Big-Omega**

$f(n) = \Omega(g(n))$ iff $\exists$ positive constants $c_1, n_0 : \ 0 \le c_1 g(n) \le f(n) \ \forall\, n \ge n_0$.

Corollary F.5 **Big-Omega: If little-omega then Big-Omega.**

If $f(n) = \omega(g(n))$, then $f(n) = \Omega(g(n))$.

Definition F.9 **Theta**

$f(n) = \Theta(g(n))$ iff $\exists$ positive constants $c_1, c_2, n_0 : \ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \ \forall\, n \ge n_0$.

Definition F.10 **Theta –Limit definition**

$f(n) = \Theta(g(n))$, iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, where $c > 0$ is a (positive) constant (and other than zero).

Corollary F.6 **Theta: Big-Oh and Big-Omega if-and-onlyif Theta.**

$f(n) = \Theta(g(n))$, iff $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$.

Note F.3.1 **Set theoretic view of $O, o, \Theta, \omega, \Omega$**

When we say $n^2 + 10 = O(n^2)$ we view $O(n^2)$ as the set containing all quadratic functions. Thus it also contains $n^2 + 10$ which is indeed a quadratic function. Likewise $5 = O(1)$ means 5 belongs to the set of all constant functions. Some textbooks are using the notation $n^2 + 10 \in O(n^2)$ or $5 \in O(1)$ instead of using the equal sign. And as a side note, $5 = \Theta(1)$ or $5 \in \Theta(1)$ is a better tighter description and so is $n^2 + 10 = \Theta(n^2)$ or $n^2 + 10 \in \Theta(n^2)$. $O(n^2)$ includes not only quadratic functions, but also linear, log-linear, logarithmic and everything else in-between or asymptotically smaller.

Note F.3.2 **Non-commutative use of symbols**

$\Theta(1) = 5$ means nothing. The symbols as defined earlier appear always to the right of the equal $=$ sign. The set theoretic view make it easy to argue that $\Theta(1) \in 5$ does not make sense.

Note F.3.3 **Tomatoes vs Potatoes or $<$ and O etc**

$1 \leq O(n)$ is meaningless. One can say $1 = O(n)$ and this was properly defined above, or $1 = o(n)$. Nothing else was defined involving the $<, >, \leq, \geq$ and the five letter symbols!

Fact F.3 **Polynomial vs Polylogarithmic: an asymptotic comparison**

Some obvious results (constant $m, k > 0$): $n^m = \omega(\lg^k n)$. This derives from the fact that in general $n^m > \lg^k n$ for any positive constant m, k and large enough n.

Fact F.4 **Exponential vs polynomial: an asymptotic comparison**

Some obvious results (constant $m > 0$): $2^n = \omega(n^m)$. This derives from the fact that in general $2^n > n^m$ for any positive constant m and large enough n.

Fact F.5 **Factorial**

Some obvious results (constant $m > 0$): $n! = \omega(n^m)$. This derives from the fact that $n! > n^m$ for any constant m and large enough n.

Fact F.6 **Log-factorial**

Some obvious results (constant k): $\lg(n!) = \Theta(n \lg n)$. Also, a result of Stirling's approximation formula for the factorial.

Example F.3.1

Which of $a_0 + a_1 n + a_2 n^2 + a_3 n^3$ and n^2 is asymptotically larger, where $a_i > 0$ for all i?

Proof. Consider $a_0 + a_1 n + a_2 n^2 + a_3 n^3$. As all a_i are positive, then $a_0 > 0$ and $a_1 n > 0$ and $a_2 n^2 > 0$ and thus $a_0 + a_1 n + a_2 n^2 + a_3 n^3 > a_3 n^3$.

We now show that our lower bound $a_3 n^3$ is $\Omega(n^2)$. As $a_3 > 0$, it is obvious that $a_3 n^3 > 1 \cdot n^2$ for any $n > 1/a_3$ (note that $a_3 > 0$ DOES NOT MEAN THAT $a_3 > 1$, as a_3 is real and not necessarily an integer).

Therefore for $c = 1$ and $n_0 = 1/a_3$ we have shown that $a_0 + a_1 n + a_2 n^2 + a_3 n^3 = \Omega(n^2)$. ■

Example F.3.2

Which of the two functions is asymptotically larger $a_0 + a_1 n + a_2 n^2 + a_3 n^3$ or n^4, where $a_i > 0$ for all i?

Proof. **Hint.** When we intend to prove $f(n) = O(g(n))$, as is the case here, it sometimes helps to find an upper bound $h(n)$ for $f(n)$ ie one such that $f(n) \leq h(n)$ and then show that $h(n) = O(g(n))$. Since in $a_0 + a_1 n + a_2 n^2 + a_3 n^3$ all a_i are positive we take the maximum of all a_i and we call it A. Then we have that $a_i < A$ for all i. Also, $An^i < An^3$ for $i \leq 3$. Then

$$a_0 + a_1 n + a_2 n^2 + a_3 n^3 \leq A + An + An^2 + An^3 \leq An^3 + An^3 + An^3 + An^3 = 4An^3$$

Finally $4An^3 \leq n^4$ for all $n > 4A$.

We have shown that

$$a_0 + a_1 n + a_2 n^2 + a_3 n^3 \leq 4An^3 \leq 1 \cdot n^4$$

for all $n \geq n_0 = 4A$, where A is the maximum of a_0, a_1, a_2, a_3. As all a_i are constant, so is $4A$. Therefore the constants in the O definition are $c = 1$ and $n_0 = 4A$, where $A = \max\{a_0, a_1, a_2, a_3\}$. ■

Appendix G

Divisibility

G.1 Divisibility and Division

Definition G.1 **Divisibility**

For $a, b \in \mathbb{Z}$, we say that a **divides** b, or b **is divided by** a, and we write $a|b$, if and only if there exists an integer $q \in \mathbb{Z}$ such that $b = aq$.

For a, b as in the definition, integer b **is a multiple of** a and consequently a is a **divisor or fctor** of b.

Example G.1.1

Every integer a is a divisor of 0. 0 divides itself and only itself.

Proof. Since $0 = a0$ the first claim follows. Since $0 \cdot q = 0$, integer 0 is a divisor of 0. There is no way for any $d \neq 0$ to have $0 \cdot q = d$. Thus 0 cannot be the divisor of any $d \neq 0$. Thus 0 only divides 0; moreover, 0 is a multiple of every integer! ■

Example G.1.2

Both 5 and -5 are divisors of 5. Both 1 and -1 are divisors of 5.

Proof. It is $5 = 5 \cdot 1$ and $5 = (-5)(-1)$. Replacing 5 by a, for every integer a, both $\pm a$ and ± 1 are divisors of a. Thus integer 5 has four divisors $+1, -1, +5, -5$. And so so does -5. And in fct any $b \neq 0, 1, -1$ has four divisors. 0 has an infinite number of divisors (in fcts its set of divisors is $\mathbb{Z}$). Only $+1, -1$ have two divisors each (including the other one of the pair). $+1$ is the positive unit and -1 the negative unit and collectively are known as the **units**. ■

Example G.1.3

If $a|b$ then $-a|b, \quad a|-b, \quad a|-b$.

Proof. If $b = aq$ then $b = (-a)(-q)$ and $-b = a(-q)$ and $-b = a(-q)$. ■

Theorem G.1 **Properties of divisibility**

If $a, b, c \in \mathbb{Z}$, then

(i) *if $a|b$ then for every $d \in \mathbb{Z}$, $a|db$.*

(ii) *if $a|b$ and $b|c$ then for $a|c$.*

(iii) *if $a|b$ and $a|c$ then for $a|b+c$.*

(iv) *if $a|b$ and $a|c$ then for $a|b-c$.*

(v) *if $a|b$ and $b \neq 0$ then for $|a| \leq |b|$.*

(vi) *if $a|b$ and $b|a$ then $|a| = |b|$ or equivalently*

(vii) *if $a|b$ and $a|c$ then $a|bx+cy$ for every $x, y \in \mathbb{Z}$.*

Proof. (i) If $a|b$, then $b = aq$. Therefore $bd = aqd = a(qd)$ and the result follows.

(ii) If $a|b$, then $b = aq$, for some $q \in \mathbb{Z}$. If $b|c$, then $c = br$, for some $r \in \mathbb{Z}$. Then $b = aq$ and $c = br$ imply $c = br = (aq)r = a(qr)$. Thus $a|c$.

(iii) If $a|b$, then $b = aq$, for some $q \in \mathbb{Z}$. If $a|c$, then $c = ar$, for some $r \in \mathbb{Z}$. From $b = aq$ and $c = ar$ we conclude that $b+c = aq+ar = a(q+r)$, where $q+r \in \mathbb{Z}$. Thus $a|b+c$.

(iv) Similarly to (iii).

(v) If $a|b$ then $b = aq$. Then $|b| = |aq| = |a| \cdot |q|$. If $b \neq 0$ then $|b| > 0$. (Absolute values are positive or zero.) This implies that $|a| > 0$ and $|q| > 0$. The latter is equivalent to $|q| \geq 1$. Then $|b| = |a| \cdot |q| \geq |a| \cdot 1 \geq |a|$. Equivalently $|a| \leq |b|$.

(vi) Given that $a|b$ and $b|a$, if $b = 0$ since $b|a$, it means $a = 0$ as well. Likewise if $a = 0$ then $b = 0$ as well. (So the case one of a,b is zero implying that the other is zero and thus $|a| = |b|$ is resolved.) For the remainder, let us assume that $a \neq 0, b \neq 0$. Then by way of (v) we have $|a| \leq |b|$ and swapping the roles of a and b, also $|b| \leq |a|$. Both imply that $|a| = |b|$ and thus $a = \pm b$ or $b = \pm a$.

(vii) A consequence of (i) and (iii). (Use $d = x$ and $b = c, d = y$.) ■

Theorem G.2 **Uniqueness**

For every $a \in \mathbb{Z}^*, b \in \mathbb{Z}$, if $a|b$, there is a unique integer $q \in \mathbb{Z}$ such that $b = aq$.

Proof. Suppose that $a|b$ i.e. $b = aq$ with $q \in \mathbb{Z}$ and $a \in \mathbb{Z}^*$ i.e. $a \neq 0$. If q is not unique, then there might exist a q_1 such that $b = aq_1$ and $q \neq q_1$. Then $b = aq = aq_1$ implies $a(q - q_1) = 0$. Since $a \neq 0$, it must be $q - q_1 = 0$. Then $q = q_1$ but this contradicts to the existence of $q_1 \neq q$. ■

In the following definition a 1 is a prime number and has two divisors, the two units (+1, -1), and so does -1; every other non-zero prime number has four divisors. In the follow-up definition we exclude the units from being prime numbers, and thus every prime has four divisors.

Definition G.2 **Prime (units are primes)**

An integer $p \in \mathbb{Z}^*$ is a prime (number) if and only if its only divisors are the units and p, and $-p$.

Definition G.3 **Prime (units are not primes)**

An integer $p \in \mathbb{Z}^*$ is a prime (number) if and only if $p \neq \pm 1$ and its only divisors are the units and p, and $-p$ (i.e. four integers for an integer $p \neq 0, -1, +1$)).

Definition G.4 **Prime**

An integer $p \in \mathbb{Z}^*$ is a prime (number) if and only if $p \neq \pm 1$ and its only positive divisors are 1 and p.

Definition G.5 **Composite**

An integer $n \in \mathbb{Z}^*$ that is not a unit, it is either a prime or a composite (integer) number.

Theorem G.3 **Division**

For $a \in \mathbb{Z}$ and $b \in \mathbb{Z}^*$ there exist unique integers $q \in \mathbb{Z}$ and $r \in \mathbb{Z}$ such that

$$a = bq + r \quad \textbf{and} \quad 0 \leq r < |b|.$$

Proof. **Case 1 ($a \geq 0, b > 0$).** Let A be the set of all non-negative integers $a - bi$, where i is such that $a - bi \geq 0$. For $i = 0$, a belongs to A, and thus A is not empty. By the Well-ordered set principle A has a minimum and let it be r. Since r is in A, we have $r \geq 0$ and $r = a - bi$ for some integer i. All it remains to show is that $r < |b| = b$. Say that $r \geq b$ instead. Then $r - b \geq 0$. Moreover

$$r - b = (a - bi) - b = a - b(i + 1).$$

Thus $r - b \geq 0$ and is of the form $a - bi'$, with $i' = i + 1$. Thus it belongs to A. Moreover $r - b$ is less than r, $r - b < r$ since b is positive. We have found an element smaller than the minimum element of A. This contradicts to the choice of r as being the minimum; we reached contradiction because we assume $r \geq b$. Thus $r < b$. This thus establishes that $0 \leq r < b$.

We have yet to prove the pair (q, r) is unique. Let is not be and let another pair be (q', r'). Then $a = bq + r = bq' + r'$ where $0 \leq r, r' < b$.

This gives $b(q - q') = r' - r$, and $|b(q - q')| = |r' - r|$. Adding $0 \leq r$ and $r' < b$ we get $r' < r + b$ or equivalently $r' - r < b$. Thus $|b(q - q')| < b$. As $b > 0$ we have $|q - q'| < 1$. This can only be possible, since q, q' are integer if $q = q'$. This means that $r = r'$. This show the uniqueness of the pair (q, r).

Case 2 ($a < 0, b > 0$). Similarly as before A is not empty because $a - bi \geq 0$ contains at least one element $a - ba \geq 0$, and the rest of the discussion is similar to case 1.

Case 3 ($a \in \mathbb{Z}, b < 0$). Then $|b| > 0$, and of course $|b| = -b$. By way of cases 1 and 2 for a and $|b|$ we have that $a = |b|q + r$, where $0 \leq r < |b|$. This is equivalent to $a = (-b)q + r$, where $0 \leq r < |b|$. This is equivalent to $a = b(-q) + r$, where $0 \leq r < |b|$. The claim is satisfied for case 3 as well.

For a pair $(a, b \neq 0)$ there is a unique pair (q, r) with $a = bq + r$ and $0 \leq r < |b|$. ∎

Theorem G.4 **Division Results**

For a, b as in division ($b \neq 0$), we have

(i) if $d|a$ and $d|b$ then $d|r$.

(ii) if $d|r$ and $d|b$ then $d|a$.

Proof. (i) If $d|a, d|b$ then $d|bq$ and thus $d|a - bq$ thus leading (given that $a = bq + r$) to $d|r$.

(ii) If $d|r, d|b$ the $d|bq$ and thus $d|bq + r$ which leads to $d|a$. ∎

Theorem G.5 **Remainder of division by b**

For $a, A \in \mathbb{Z}$, let them be divided by b. The remainders of those two divisions are the same if and only if $a-A$ is a multiple of b.

Proof. $\Rightarrow$. If $a = bq + r$ and $A = bQ + R$, and we are given $r = R$, the $a - A = bq + r - qQ - r = b(q - Q)$. Since $q - Q \in \mathbb{Z}$, we conclude that $b|a - A$.
$\Leftarrow$. Say that $b|a-A$. Then let $a = bq + r$ and $A = bQ + R$, where $0 \leq r, R < |b|$. We have $a - A = b(q-Q) + (r-R)$. Since obviously $b|b(q-Q)$ and also by assumption $b|a-A$ we conclude that $b|(a-A) - b(q-Q)$ i.e. $b|r-R$ and also $|b||r-R$. This would imply that $|b| \leq |r-R|$ or $|r-R| = 0$. But $r, R < |b|$ which means that $r - R$ or $|r-R|$ are such that $|r-R| < |b|$. Thus the former $|b| \leq |r-R|$ is false and we must have $|r-R| = 0$ which implies $r = R$.

Another way to prove $\Leftarrow$ it is directly as follows. Say that $b|a-A$, i.e. $a - A = bm$ for some $m \in \mathbb{Z}$. Then let $a = bq + r$ and $A = bQ + R$, where $0 \leq r, R < |b|$. From the former $a - A = bm$ and $A = bQ + R$ we have $a - (bQ + R) = bm$ implying $a = b(m + Q) + R$. This latter equality implies $q = m + Q$ and $r = R$ as R is such that $0 \leq R < |b|$). Result is proven. ■

Theorem G.6 **Representation of integers**

The representation of integer a in a base b mean $a = a_0 + a_1 b + a_2 b^2 + \ldots + a_k b^k$. Integer $k+1$ is the number of digits in the representation. The base b is also known as radix-b. Moreover $0 \leq a_i < b$ for all $0 \leq i \leq k$. We also write $a = (a_k \ldots a_1 a_0)$. a_0 is the least significant digit (or rightmost digit) and a_k is the most significant digit (or leftmost digit).

Example G.1.4

In order to convert say integer 140 into a base 2 (aka binary) integer we can perform repeated division

Proof.

$$\begin{aligned} 140 &= 2 \cdot 70 + 0 \\ 70 &= 2 \cdot 35 + 0 \\ 35 &= 2 \cdot 17 + 1 \\ 17 &= 2 \cdot 8 + 1 \\ 8 &= 2 \cdot 4 + 0 \\ 4 &= 2 \cdot 2 + 0 \\ 2 &= 2 \cdot 1 + 0 \\ 1 &= 2 \cdot 0 + 1 \end{aligned}$$

Reading the remainders bottom to top and writing them left to write we claim that $140 = (10001100)_2$. It is not difficult to confirm this. ■

G.2 Greatest Common Divisor

Definition G.6 **Greatest Common Divisor**

For integers $a,b \in \mathbb{Z}$, $\gcd(a,b)$ is the **greatest common divisor** of a and b if and only if $\gcd(a,b)|a$ and $\gcd(a,b)|b$ and every other divisor c of a,b is such that $c \leq \gcd(a,b)$. Thus

(i) $\gcd(a,b)|a$ and $\gcd(a,b)|b$,

(ii) $c|a$ and $c|b \Rightarrow c \leq \gcd(a,b)$.

Note that if $c|a$ then by Theorem G.1(v) we have $|c| \leq |a|$. Likewise if $c|b$ we have $|c| \leq |b|$. Combining the two we have $|c| \leq \max(|a|,|b|)$. Thus the set of common divisors of a,b is finite. A finite set of integers always has a maximum, and thus there is a uniquest largest integer $d > 0$ such that $d|a$ and $d|b$. We call d the **greatest common divisor** of a,b and denote it by $\gcd(a,b)$ thus $d = \gcd(a,b)$.

Note G.2.1

The $\gcd(0,0)$ is not defined as the set of common divisors is an infinite set and it does not have a maximum. In the remainder when $\gcd(a,b)$ is considered, we would assume that $a \neq 0$ or $b \neq 0$ (or both). Thus it cannot be that both a and b are zero.

Definition G.7

Let $S(a)$ be the set of divisors of A.

Fact G.1 **Simple GCD facts**

Let $a,b \in \mathbb{Z}$ not both zero. Then

(i) $\gcd(a,b) > 0$ and also $\gcd(a,b) \geq 1$.

(ii) $\gcd(a,b) = \gcd(|a|,|b|)$.

(iii) $\gcd(a,b) = \gcd(b,a)$.

(iv) $\gcd(a,1) = 1$.

(v) $\gcd(a,0) = |a|$ for all $a \neq 0$.

(vi) $\gcd(a,b) = |a|$ if and only if $a|b$.

Proof. (i) Obviously. $gcd(a,b)$ is the maximum of the common divisors of a and b i.e. the maximum elements of $S(a) \cap S(b)$. One positive element of this set of common divisors is 1 and thus $gcd(a,b) \geq 1$ in addition to $\gcd(a,b) > 0$.

(ii) Since $S(a) = S(|a|)$ and $S(b) = S(|b|)$ we have that $S(a) \cap S(b) = S(|a|) \cap S(|b|)$. Thus $\gcd(a,b)$ is equal to $\gcd(|a|,|b|)$ since the set of common divisors are equal to each other.

(iii) It is a consequence of the fct that $S(a) \cap S(b) = S(b) \cap S(a)$.

(iv) Since $1|a$ trivially, and the largest divisor of 1 is 1 itself the result follows. (Note that $S(1) = \{-1,+1\}$.)

(v) $S(a) \cap S(0) = S(a) \cap \mathbb{Z} = S(a)$. The result follows as the largest element of $S(a)$ is $|a|$.

(vi) $\Rightarrow$. If $\gcd(a,b) = |a|$ then $|a||a$ and $|a||b$. For the latter there exist q such that $b = q|a|$. If a is non-negative, the $b = qa$ as well. If a is negative $b = q(-a) = (-q)a$. The former concludes $a|b$ and so does the latter.

$\Leftarrow$. If $a|b$ then $-a|b$ and thus $|a||b$. Since trivially $a|a$ we conclude that $-a|a$ and thus $|a||a$. Then $|a||\gcd(a,b)$. This by Theorem G.1(v) implies $|a| \leq \gcd(a,b)$ (the gcd is always positive thus no absolute value sign around it

is needed). By definition $\gcd(a,b)|a$ and thus $\gcd(a,b)|-a$. Thus $|\gcd(a,b)| \leq |a|$. Combining $|\gcd(a,b)| \leq |a|$ and $|a| \leq \gcd(a,b)$ the result follows. ■

Theorem G.7

If $x, y \in \mathbb{Z}$, then

$$\gcd(x,y) = \gcd(x, xq+y), \quad \gcd(x, xq+y) = \gcd(x,y),$$

for all integers $q \in \mathbb{Z}$.

Proof. Let $d = \gcd(x,y)$. Let $d_1 = \gcd(x, xq+y)$. If $d|x$ and $d|y$ by the gcd definition, we have $d|xq$ and $d|y$. Thus $d|xq+y$ in addition to $d|x$. Thus d is a common divisor fo $xq+y$ and x. Thus $d \leq d_1 = \gcd(x, xq+y)$. For $d_1 = \gcd(x, xq+y)$ we have $d_1|x$ and $d_1|xq+y$. From the former, we conclude that $d_1|xq$; combining it with the latter we have $d_1|xq+y-xq$ i.e. $d_1|y$. Thus $d_1|x$ and $d_1|y$ and therefore d_1 is a common divisor of x, y. Thus $d_1 \leq d = \gcd(x,y)$.

By way of $d_1 \leq d$ and $d \leq d_1$ we conclude $d = d_1$. ■

From Theorem G.3 in order to compute the $\gcd(a,b)$ we formulate the divsion operation.

$$a = bq + r$$

where $0 \leq r < |a|$. Then by way of Theorem G.7

$$\gcd(a,b) = \gcd(b,a) = \gcd(b, bq+r) \overset{Th.\ G.7}{=} \gcd(b,r).$$

Note G.2.2

If $a = b$ then $\gcd(a,b) = a = b$. Thus in general we need to compute $\gcd(a,b)$ for $a > b$ or $b > a$.

Note G.2.3

If we assume, **without loss of generality**, that $a > b$, if $b = 0$ $\gcd(a,b) = a$. Thus in general we need to compute $\gcd(a,b)$ for $a > b$ and $b \neq 0$.

At a minimum $|a| + |b| \neq 0$ i.e. we can't determine the gcd of two numbers that are both 0. The set of common divisors is then $\mathbb{Z}$ and has no maximum. GCD is also known as Euclid's algorithm.

Theorem G.8 **GCD**

Applying Theorem G.3 for $a > b$, $b \neq 0$ by prior discussion if $a = bq + r$ we have $\gcd(a,b) = \gcd(b,r)$. Moreover since r is the remainder of the division of a by b we have $0 \leq r < |b|$. If the remainder is not 0, we continue the division until a remainder is drawn that is 0. The **last non-zero remainder** *is the gcd i.e. $\gcd(a,b) = r_k$.*

$$\begin{aligned}
a &= bq + r & 0 \leq r < |b| & & \gcd(a,b) &= \gcd(b,r) \\
b &= rq_1 + r_1 & 0 \leq r_1 < r & & \gcd(b,r) &= \gcd(r,r_1) \\
r &= r_1 q_2 + r_2 & 0 \leq r_2 < r_1 & & \gcd(r,r_1) &= \gcd(r_1,r_2) \\
&\dots & & & & \\
r_{k-2} &= r_{k-1} q_k + r_k & 0 \leq r_k < r_{k-1} & & \gcd(r_{k-2},r_{k-1}) &= \gcd(r_{k-1},r_k) \\
r_{k-1} &= r_k q_{k+1} + 0 & & & \gcd(r_{k-1},r_k) &= \gcd(r_k,0) = r_k
\end{aligned}$$

Note that $a > b$ and $|b| > r > r_1 > r_2 > \ldots > r_{k-1} > r_k$.

Proof. It is obvious from the statement that

$$\gcd(a,b) = \gcd(b,r) = \gcd(r,r_1) = \gcd(r_1,r_2) = \ldots = \gcd(r_{i-1},r_i) = \ldots = \gcd(r_{k-1},r_k) = \gcd(r_k,0) = r_k$$

That is, the last, non-zero remainder, is the gcd. Moreover the sequence of remainders purely decreases i.e. $|b| > r > r_1 > r_2 > \ldots > r_{k-1} > r_k$. Thus after a finite number of no more than b steps r_k will be determined. ■

Another way to prove this Theorem G.8 is the following direct method.

Proof. Starting from the bottom $r_k|r_{k-1}$. Then from the penultimate equation we have r_k dividing both itself and r_{k-1} and thus $r_k|r_{k-1}q_k + r_k$. Thus $r_k|r_{k-2}$. Working likewise we show that r_k divides r and b of the first equation and thus also divides a. Thus r_k divides both a,b. Pick an arbitrary integer d dividing a and b. Working downwards we show that d divides r_k as well. Thus $d \leq r_k$. That is r_k is the gcd of a,b as any common divisor of a,b is at most r_k. ■

The worst-case (i.e. longest) division is for two consecutive Fibonacci numbers $F_n = F_{n-1} + F_{n-2}$, $n > 1$, with $F_0 = 0$ and $F_1 = 1$ generate a quotient of 1 every iteration! For F_n the number of divisions is $n-2$ as the last division is $F_2 = F_1 + F_0 = F_1 + 0$. The n-th Fibonacci number exceeds ϕ^{n-2}.

Corollary G.1

For $a,b < N$ the number of divisions in Euclid's Algorithm will be less than $\lg N / \lg \phi \approx 1.48 \lg N$.

Proof. We can do a bit better by bounding the number of divisions in terms of b only. In the worst case, for an n step division $a \geq F_{n+2}$ and $b \geq F_{n+1}$. But $F_{n+1} \geq \phi^{n-1}$. Thus $n-1 \leq \lg b / \lg \phi$. $\lg \phi \approx 0.687$ and thus $n-1 \leq 1.471 \lg b$ i.e. $n \leq 1.471 \lg N + 1$. ■

Theorem G.9 **GCD-Divided**

If $d = \gcd(a,b)$ *then* $\gcd(\frac{a}{d}, \frac{b}{d}) = 1$.

Proof. If $d = \gcd(a,b)$ then by Theorem G.8 we have that $d = r_k$. By the last equation of gcd's we have $\gcd(r_{k-1}, r_k) = r_k = d$. Given that d divides itself ($r_k = d$) we also have that $d|r_{k-1}$. From the prior expression, now that we have $d|r_k$ and $d|r_{k-1}$ using $\gcd(r_{k-2}, r_{k-1}) = \gcd(r_{k-1}, r_k) = r_k = d$ we have that $d|r_{k-2}$ as well. Continuing likewise we have $d|r_2$ and $d|r_1$. Likewise from the third equation of Theorem G.8 we have $d|r$ and from the second $d|b$. Concluding from the first (division) equation we have $d|a$. We can thus divide by d all equations. This shows that $\gcd(\frac{a}{d}, \frac{b}{d}) = \frac{d}{d} = 1$ as all remainders will be divided by $d = r_k$ as well. ■

Definition G.8

For two integers $a,b \in \mathbb{Z}$ if $\gcd(a,b) = 1$ the two integers a and b are called **relatively prime**.

Theorem G.10 **Extended-GCD**

Let $a,b \in \mathbb{Z}$, *where* $|a| + |b| \neq 0$. *Let* $d = \gcd(a,b)$ *then there exist integers* x,y *such that*

$$d = ax + by$$

Moreover, d *is the smallest positive integer that can be written as a linear combination of* a,b.

Proof. Consider

$$A = \{au + bv | u,v \in \mathbb{Z} \textbf{ and } au + bv > 0\}.$$

Note that for $u = a$, $v = b$, $a \cdot a + b \cdot b > 0$ since $|a| + |b| \neq 0$. Set A has thus at least one element and it is not empty. Thus from the well-ordered set principle there must exist a minimum element for A and let it be d'. Since $d' \in A$ there exist x,y such that

$$ax + by = d'.$$

Show d' is a divisor of a. We first show that $d'|a$. Let us form the division operation for the two integers i.e.

$$a = d'q' + r'$$

where $0 \leq r' < d'$. Then we have that

$$r' = a - d'q' = a - (ax+by)q' = a \cdot (1 - xq') + b \cdot (-yq').$$

Note that decause of the division of a by d' we know that $0 \leq r^p rime$.

$0 < r'$ **case.** If $r' > 0$ then it is a member of A since $r' = a \cdot (1 - xq') + b \cdot (-yq')$. But this cannot happen since $r^p rime < d'$ and d' is the minimum element of A. It would imply the existence of an element of A (i.e. r') smaller than the minimum element of A (i.e. d')!

$0 = r'$ **case.** The only other possibility is that $r' = 0$. But then, $a = d'q' + r'$ implies $a = d'q'$ i.e. d' is a divisort of a.

Show d' is a divisor of b. Similar to the case of a.

Thus d' divides a and b and is a common divisor of a, b. It should be that $d' \leq d$ since d is the gcd of a and b and is the greatest common divisor of a, b. If g is another common divisor of a and b we conclude from $ax+by = d'$ that $g|ax+by$ i.e. $g|d'$. Thus $g \leq d'$. Thus any common divisor of a, b such as g is no more than d'. Such a common divisor also includes the gcd of a and b and thus $d \leq d'$. This along with the $d' \leq d$ shows that $d = d'$. (We also proved that every common divisor of a, b also divided the gcd of a, b.)

In conclusion

$$d = \gcd(a,b) = d' = ax + by.$$

■

Corollary G.2

If $d = \gcd(a,b)$ then $S(a) \cap S(b) = S(d)$. (In other words, $m|a$ and $m|b \iff m|d$.)

Proof. Every common divisor of a and b is a divisor of $d = \gcd(a,b)$ since $d = d' = ax + by$ derived and used in Theorem G.10. That is $S(a) \cap S(b) \subseteq S(d)$.

Moreover for $t \in S(d)$ then $t|d$ and since $d|a$ and $d|b$ by transitivity we have $t|a$ and $t|b$. The former show that $t \in S(a)$ and the latter that $t \in S(b)$. Both of them show that $t \in S(a) \cap S(b)$. Thus $S(d) \subseteq S(a) \cap S(b)$. ■

Corollary G.3

Let $a, b \in \mathbb{Z}$, $|a| + |b| \neq 0$. Let $m \in \mathbb{Z}_+^*$. Then

$$\gcd(ma, mb) = m \gcd(a,b).$$

Proof. Let $d = \gcd(a,b)$. Since $d|a$ we have $a = dq$ and thus $ma = (md)q$. Thus $md|ma$. Likewise $md|mb$. Thus $md \leq \gcd(mb, ma)$.

Since $m|ma$ and $m|mb$ we have $m|\gcd(ma,mb)$. Thus $\gcd(ma,mb) = mq$ for some q. Moreover $\gcd ma, mb|ma$ implies $mq|ma$ i.e $q|a$. Likewise $q|b$. Thus $q|d$ i.e. $q \leq d$. We conclude that $\gcd(ma,mb) = mq \leq md$. From $md \leq \gcd(mb,ma)$ previously and $\gcd(ma,mb) = mq \leq md$ the corollary follows. ■

Theorem G.11

For three $a, b, c \in \mathbb{Z}$, we have $\gcd(a,b,c) = \gcd((a,b),c) = \gcd(d,c)$, *where* $d = \gcd(a,b)$.

Proof. $\gcd(a,b,c)$ belongs to $S(a) \cap S(b) \cap S(c) = (S(a) \cap S(b)) \cap S(c)$. The result follows. ■

Theorem G.12

For p a prime, and $a \in \mathbb{Z}^*$, p DOES NOT DIVIDE a if and only if $\gcd(a,p) = 1$.

Proof. If p is a prime $S(p) = \{-1,+1,-p,+p\}$, then p does not divide a means that neither p nor $-p$ are divisors of a. The only possible divisors are $+1$ and -1. Thus $S(a) \cap S(p) = \{+1,-1\}$. Thus the gcd is 1, and thus $\gcd(a,p) = 1$ as needed.

If $\gcd(a,p) = 1$, then p cannot divide a. This is because if $p|a$ since obviously $p|p$ we would have from a prior property that $p|\gcd(a,p)$ and (in fct $\gcd(a,p) = |p|$). For this to happen p should be -1 or $+1$. But p is prime and this can't happen. ■

Theorem G.13

For $a,b,m \in \mathbb{Z}^*$ and $\gcd(a,b) = 1$ *and $a|bm$ then $a|m$.*

Proof. Since $\gcd(a,b) = 1$ we have $1 = ax+by$ for some x,y. Multiplying by m we get $m = axm+bmy$. Obviously $a|axm$. Since $a|m$ we have $a|bmy$ as well. Thus $a|axm+bmy$ i.e. $a|m$. ■

Theorem G.14

For $a,b \in \mathbb{Z}^*$ and prime p is such that $p|ab$, then p divides either a or b.

Proof. Say p does not divide a. By a prior thm since p is prime this means $\gcd(p,a) = \gcd(a,p) = 1$. Since $p|ab$ by the previous thm we have $p|b$. ■

Theorem G.15

If $\gcd(a,b) = 1$ *and $a|c$ and $b|c$ then $ab|c$.*

Proof. If $\gcd(a,b) = 1$ we have $1 = ax+by$. Then $c = acx+bcy$. We have $a|c$ i.e. $c = aq$. Then $cb = abq$ and thus $bcy = ab(qy)$. The latter implies $ab|bcy$.

We also have $b|c$ i.e. $c = br$. Then $ac = abr$. Thus $acx = ab(rx)$. The latter implies $ab|acx$.

Thus $ab|bcy$ and $ab|acx$ imply $ab|acx+bcy$. Therefore $ab|c$. ■

G.3 Least Common Multiplier

For $a,b \in \mathbb{N}$ we denote by $\operatorname{lcm}(a,b)$ the **least common multiplier** of a and b. For $a,b \in \mathbb{Z}$ only the positive multipliers are considered.

Moreover $\operatorname{lcm}(ma,mb) = m\operatorname{lcm}(a,b)$.

Theorem G.16

Let $a,b \in \mathbb{N}$. *If* $\gcd(a,b) = 1$ *then* $\operatorname{lcm}(a,b) = ab$.

Proof. Let $m = \operatorname{lcm}(a,b)$. Since $\gcd(a,b) = 1$ we have $1 = ax + by$. Then $m = axm + bym$. Since $a|m$ and $b|m$. Then $m = ap$ and $m = bq$. Moreover substituting to the previous equation we have $m = ax(bq) + by(ap) = ab(xq) + ab(yp) = ab(xq + yp)$. Thus $ab|m$. This implie $ab \leq m$. Moreover m is the least common multiple of a,b. One such multiple is ab. Thus $ab \geq m$. The $ab \leq m$ and the just shown $ab \geq m$ implies $m = ab$. ■

Theorem G.17

Let $a,b \in \mathbb{N}$. *Then* $\gcd(a,b) \cdot \operatorname{lcm}(a,b) = ab$.

Proof. Let $d = \gcd(a,b)$. Let $a_1 = a/d$ and $b_1 = b/d$. Then $\gcd(a/d, b/d) = \gcd(a_1, b_1) = 1$. From the previous thm we have $\operatorname{lcm}(a_1,b_1) = \operatorname{lcm}(a/d, b/d) = ab/d^2$. ■

If $T(a)$ is the set of (positive) multiples of a, then $T(a) = T(-a) = T(|a|)$. Moreover $T(a) \cap T(b) = T(|a|) \cap T(|b|)$. Therefore $\operatorname{lcm}(a,b) = \operatorname{lcm}(|a|,|b|)$.

Corollary G.4

For $a,b \in \mathbb{Z}^*$ if $a|b$, then $\operatorname{lcm}(a,b) = b$.

Proof. If $a|b$, then any element $t \in T(b)$ is $t \geq b$. Moreover b or $|b|$ is in $T(a)$. The result follows. ■

Appendix H

String Matching

H.1 String Matching

H.1.1 Naive approach

The **string matching** problem is defined as follows. For a text $T[..]$ of length n and a pattern $P[..]$ of length m and the characters of T and P drawn from a finite alphabet Σ, we say that the pattern P starts at position $s+1$ in text T if $T[s+1..s+m]$ is equal to $P[1..m]$ (or to $P[0..m-1]$ if indices start from 0 rather than 1).

A string of length 0 is called the null string and represented by ε. A substring of a string S is a string $S[i...j]$, where $j \geq i$. A substring $S[i..j]$, with $i > j$ is considered to be the null string. The length of string S is denoted by $|S|$. For two strings x and y the concatenation of x and y is string xy of length $|x|+|y|$. A string w is a prefix of x if $x = wy$. A string q is a suffix of a string r if $r = tq$.

The naive way for string matching works as follows.

```
  NaiveMatching(P[0..m-1],T[0..n-1])
1. for (i=0;i<=n-m;i++) {// i indicates T[i...] is checked vs P[..]
2.  j=0; // runs through the pattern
3.  while ((j<m) && (T[i+j] == P[j])) {
4.   j++;
    }
5.  if (j==m) // a match has been found
6.    return(i); //T[i..i+m-1] is P[0..m-1]
   }
7. reurn(-1); //-1 means no match.
}
```

Namely, in the NaiveMatching algorithm the pattern is overlayed with the text for each starting position of the text (indexed by i). For each such overlay we compare the pattern to the text starting at position i. This is performed through lines 2-4. If a match is found $j == m$ is true and the algorithm returns with the index of the match (starting at i). Otherwise i is incremented by 1 and the algorithm continues. When the for..loop exits no match exists and no match (-1) is returned.

The running time of the algorithm is $O(nm)$ as the loop of line 1 is executed $O(n)$ times and the while loop of lines 3-4 $O(m)$ times.

H.1.2 Rabin-Karp

The Rabin-Karp algorithm is another approach to the matching problem, performs quite well in practice and can also be generalized for two dimensional (i.e. image) matching. Although its worst case running time is $O(nm)$ its average case running time is quite good (Do not forget that in computer science the most practical algorithms are not optimal in terms of worst case performance: hashing and quick-sort are some notorious examples).

The algorithm uses the notion of "hashing" and elementary number-theoretic notions. Suppose that the characters come from an alphabet Σ. We view each character as a digit in radix-d notation where $d = |\Sigma|$. For example a character from $\Sigma = \{a, b, \ldots, z\}$ can be viewed as a digit of radix-26. Therefore the text T and pattern P can be viewed as an n-digit and m-digit number respectively. For the rest of this discussion we are going to make the assumption that text and pattern are strings of decimal digits.

Therefore for a pattern $P[..]$ of length m we denote by p its decimal value. For a text $T[..]$ we denote by t_i the decimal value of the length m substring starting at i, i.e. of the string $T[i..i+m-1]$. Therefore the pattern matching problem is reduced to deciding whether there exists an index j such that $p = t_j$. If we could compute p in $O(m)$ time and the n numbers t_i in $O(n)$ time, then we could solve the string matching problem in additional $O(n)$ time. We note that

$$p = P[m-1] + 10P[m-2] + \ldots + P[0]10^{m-1}$$

can be computed using Horner's rule in $O(m)$ time as p is the value of the corresponding polynomial at $x = 10$. As soon as t_0 is computed for $T[0..m-1]$ in a similar way the sequence of $t_1, t_2, \ldots,$ can be computed in $O(n-m)$ time by the following recurrence relation that computes t_{i+1} from t_i, i.e. the decimal representation of $T[i+1..i+m]$ from $T[i..i+m-1]$.

$$t_{i+1} = 10(t_i - 10^{m-1}T[i]) + T[i+m]$$

Subtraction of the $10^{m-1}T[i]$ term removes the contribution of the $T[i]$ from the computation of the representation $T[i+1..i+m]$, the multiplication by 10 "shifts" the $T[i+1..i+m-1]$ one position to the left and the last term adds to that the contribution of the next term $T[i+m]$ thus giving the decimal representation of $T[i+1..i+m]$ as required. Given that 10^{m-1} can computed once (precomputed) and used forever, the reccurrence for t_{i+1} takes constant time.

The only problem with this approach is that t_i in general can be a very long integer and the assumption of constant time for the computation of t_i may make no sense at all. We can eliminate this problem if we perform all computations modulo q , where q is a very large prime number. In practice we may choose q large enough such that $10q$ can be represented by a single words (for example in C/C++ $10q$ is less than the largest unsigned 32-bit int which is $2^{32}-1$).

If instead of using a decimal alphabet, we have characters in radix-d, the recurrence above is modified accordingly to reflect the cardinality (size) of the alphabet Σ.

$$t_{i+1} = (d(t_i - (d^{m-1} \bmod q)T[i]) + T[i+m]) \bmod q.$$

The problem of using modulo arithmetic is that finding a $t_i \bmod q = (p \bmod q)$ does not mean that $t_i = p$. This may or may not be true. If we have such a "spurious hit" we can then perform the check $T[i..i+m-1] = P[0..m-1]$ to check whether $t_i = p$ indeed.

```
   Rabin-Karp (T,P,d,q)
1. h = d**(m-1) mod q;
2. p = 0;
3. t[0] = 0;
4. for(j=0;j<m;j++) {
5.   p= (dp+P[j]) mod q;
6.   t[0]=(dt[0]+T[j])mod q;
7. }
8. for(i=0;i<=n-m;i++){
9.   if (p == t[i]) {
10.    k=0; // runs through the pattern
11.    while ((k<m) && (T[i+k] == P[k])) {
12.      k++;
13.    }
14.    if (k==m) // a match has been found
15.       return(i); //T[i..i+m-1] is P[0..m-1]
```

```
16.  }
17.  if (i < n-m)
18.   t[i+1] = (d(t[i]-T[i]h)+T[i+m]) mod q;
19 }
```

In the worst case Rabin-Karp is $\Theta(nm)$. However, on the average as long as $q > m$ we expect the running time of Rabin-Karp to be $O(n+m)$. Overviewing Rabin-Karp we point out the main idea of the algorithm is the idea of fingerprinting (hashing).

Problem. In the Rabin-Karp algorithm pattern $P = P[0..m-1]$ is matched to text $T = T[0..n-1]$ by creating a finger print of the pattern P to be denoted by p and fingerprints $t_i = T[i..i+m-1]$ of text T and thus matching first p to t_i for all $i = 0,\ldots n-m$. If $p \neq t_i$ there can be no match of P at position i. If they are equal there may be a match but this requires a comparison of the actual pattern $P[0..m-1]$ to the actual text $T_i = T[i..i+m-1]$; note this latter comparison is not a fingerprint comparison but character to character comparison.

Simple FingerPrints. We describe what the fingerprints look like in Rabin-Karp's algorithm. Let us assume for the sake of simplicity that both the text and the pattern are binary 0-1 strings. The pattern P atains a fingerprint $p = P[m-1] + 2P[m-2] + \ldots + 2^{m-1}P[0]$. Similarly, fingerprint t_i is given by. $t_i = T[i+m-1] + \ldots + 2^k T[i+m-1-k] + \ldots + 2^{m-1}T[i]$. Note that $t_{i+1} = 2(t_i - 2^{m-1}T[i]) + T[i+m]$. Therefore starting with t_0 that can be computed with $O(m)$ computational operations, and progressing inductively, if we know t_i, then t_{i+1} can be found in $O(1)$ time with operations that involve subtraction/addition and binary shifting left/right. Thus establishing all t_i for all i requires $O(n+m)$ time; $O(m)$ is the contribution of finding t_0. In addition $O(m)$ time is required to find p. Total remains $O(n+m)$.

If we compute fingerprints this way, the t_i can grow large if m is large (say larger than 32 bits). Other than that the problem of matching strings has been reduced to the problem of matching fingerprints.

Elaborate Fingerprints. As fingerprints can grow large and can't possibly fit into a single word we deal with this issue by performing all operations mod q where q is a large prime number. Thus the update formula for text fingerprints becomes

$$t_{i+1} = 2(t_i - (2^{m-1} \bmod q) * T[i]) + T[i+m] \bmod q.$$

Note that $(2^{m-1} \bmod q$ is a constant and can be precomputed just once. If q, or $2q$, is kept smaller than the maximum integer available, then the update operations cause no overflows. A problem however that appears with $\bmod q$ calculations is whether $t_i = p$ really indicates a match or a false match also known as false positive or **spurious match/hit**. For spurious matches/hits we need to go back to the actual text and compare P against $T[i...i+m-1]$ for matching.

Non-binary Fingerprints. The discussion of binary fingerprints can be generalized to any alphabet; Page 3 or 4 show this for decimal and general alphabets.

How frequent are false positives? For $\bmod q$ operations, we have two matches: (a) **real match** when $t_i = p$ and $T_i = P$, and (b) **a spurious match/hit** when $t_i = p$ but $T_i \neq P$. We naturally have misses/mismatches when $t_i \neq p$.

In practice we choose q so that it is a prime and such that $q = Am\ln Am$, for some A. Why q should be so and how one can choose A will become apparent later in this discussion, and in a relevant homework.

$$Pr(\text{spurious match at } i) \quad = \quad Pr(p \bmod q = t_i \bmod q)$$

What is the probability that $p \bmod q = t_i \bmod q$? This is equivalent to $(p - t_i) \equiv 0 \bmod q$ or equivalently that q divides $p - t_i$. R is a prime number. Both p and t_i, by the way they are defined they are at most $1 + 2 + 2^2 + \ldots + 2^{m-1} \leq 2^m$. The number of distinct prime numbers that divide $p - t_i$, given that this differnce is also at most 2^m is at most m. This is because if $z_1, \ldots, z_s$ are these distinct prime divisors of the difference $z_i \leq 2$ and thus $2^s \leq z_1 \cdot \ldots \cdot z_s \leq 2^m$, which implies that $s \leq m$. The number of primes between 1 and q however, by the Prime Number Theorem is about $q/\ln q$ which is $(Am\ln Am)/(\ln Am + \ln\ln Am) \leq Am/2$. The probability that q is one of these s numbers is at most $s/Am/2 \leq 2/A$.

We have thus established.

$$\begin{aligned} Pr(\text{spurious match a t}i) &= Pr(p \bmod R = t_i \bmod q) \\ &\leq 2/A. \end{aligned}$$

Then the probability that there is a spurious match somewhere in the text is

$$\begin{aligned} Pr(\text{spurious match in text }) &= Pr(\cup_{i=0}^{i=n-1} \text{spurious match at } i) \\ &= \sum_{i=0}^{n-1} Pr(\text{spurious match at } i) \\ &\leq 2n/A. \end{aligned}$$

If we choose A such that $A = 2n^2$, then the probability $Pr(\text{spuriousmatchintext})$ is at most $1/n$. Choosing A larger further decreases this probability.

If thereis a spurious match the running time may go as high as $O(nm)$. In all other cases it remains linear $O(n+m)$. The expected running time is thus $T(n) = O(nm)(1/n) + O(n+m)(1-1/n) = O(n+m)$.

H.1.3 Boyer-Moore

If the pattern P is relatively long (say more than 4-5 characters) and the alphabet is reasonably large (i.e. strings are non-binary strings), then an algorithm due to R.S. Boyer and J.S.Moore is perhaps the most efficient string matching algorithm.

The Boyer-Moore algorithm differs from the brute-force naive algorithm in that it checks the pattern against the text not left-to-right but right-to-left. If a mismatch is found the action performed on the pattern differs from the approach of the naive algorithm (i.e. shift pattern one position to the right). In the Boyer-Moore algorithm, if the mismatched character $c = T[i]$ of the text that was compared to $P[j]$ does not appear in the pattern P, then the pattern is shifted right j positions, and thus a comparison of $P[..]$ against $T[i+1..i+m]$ will then be performed. This is because c does not appear in P and therefore no character in the portion of the pattern $P[..j-1]$ will ever match $T[i]$ and thus the portion of the text $T[..i]$ can be skipped.

If however, c appears in $P[..j-1]$, we shift P to the right by an appropriate number of positions. Let t be the maximum index in $..j-1$ that a c appears. Then we shift P to the right $j-t$ positions, so that $c = P[t]$ matches $T[i] = c$. If c does not appear in $P[..j-1]$ but it does appear in $P[j+1..]$, then and only then do we shift right one position only (i.e. Boyer-Moore behaves like the naive algorithm). The actions performed due to the mismatched c describe a method that is known as **bad-character heuristic** or the **character-jump**. How do we compute the last occurrence of a character c in P? We use function BadCharacter (or Compute-Last-Occurrence-Function in CLR, first edition). Array last computes the rightmost occurrence of a character of the alphabet Σ.

```
  BadCharacter(P,m,Alphabet)
1. for(i=0;i<|Alphabet|;i++)
2.  last[i]= Left-1;       //Left=0, Right=m for   P[0..m-1]
3. for(j=Left;j<Right;j++){//Left=1, Right=m+1 for P[1..m]
4.   last[P[j]]=j;
5. return(last[]);
```

The running time of the algorithm is $O(m+|\Sigma|)$ and thus can be performed quite efficiently. The Boyer-Moore algorithm is described below.

```
   Boyer-Moore(P[0..m-1],T[0..n-1],Alphabet)
0. last[] = BadCharacter(P,m,Alphabet);
1. i=m-1; // runs through the text
2. j=m-1; // runs through the pattern
3. while (i <= n-1) {
4.  if (P[j]==T[i]) {
5.   if (j==0)
6.     return("Match at i");
```

```
7.   else {
8.     i--;
9.     j--;
10.  }
11. }
12. else { //mismatch P[j] differs from c=T[i]
13.  i=i+m- min(j, 1+last[T[i]]);
14.  j=m-1; //restart from right end of P
15. }
16.}
17.return("no match found");
```

Why step 13 is performed the way shown there becomes clearer by the example cases below.

```
 i  T[i-j]   c=T[i]           T[i-j+m-1]
               !   =  =  =  =   =
 j  P[0]       P[j]           P[m-1]

first mismatch

   c= T[i] != P[j]

CASE 1:  c nowhere in P[0..m-1]
ACTION : Shift P to the right j positition past T[i] so that
               T[i] T[i+1]           T[i+m]
                    P[0]             P[m-1]
 reset
   new i = i+m = i+m - min(j,t+1)
   new j = m-1
   t = rightmost index such that P[t]=c ; since c not in P t=-1

CASE 2: c in P[0..j-1] and rightmost t such that P[t]=c  is t<j
  ACTION : SHift P j-t positions to the right (j-t>0)
                  j-t = j-Last[c]

          T[i-t]  c=T[i]           T[i-t+m-1]
          P[0]    c=P[t]            P[m-1
  j-t> 0  ==> j > t  ==> j >= t+1  i.e. min(j,t+1) = t+1
  reset
    new i = i-t+m-1 = i+m - min(j,t+1)
    new j = m-1

CASE 3:  c in P[j+1 .. m-1]
        i.e. P[t]=c and t >=j+1 >= j
 ACTION Shift P to the right one character
  T[i-j+1]   T[i]               T[i-j+m]
   P[0]      P[j-1] P[j]         P[m-1]
t+1 >= t >= j+1 > j   t>=j+1 ==> min(t+1,j) =j
 reset
   new i = i+m -j   = i+m -min(t+1,j)
   new j = m-1
```

The best case for Boyer-Moore is an all "a" text and an all "b" pattern. Boyer-Moore will only compare n/m characters, before failure is declared.

What is a worst case text/pattern senario in Boyer-Moore? In the worst-case Boyer-Moore is an $O(nm)$ time algorithm. Can we improve the performance of Boyer-Moore?

H.1.4 Knuth-Morris-Pratt

The Knuth-Morris-Pratt algorithm is a linear time algorithm (in the worst-case). Note that Boyer-Moore as described before may not be linear in certain cases (e.g. binary stings of short length). For every pattern some

precomputation needs to be performed in the KMP algorithm. KMP works left-to-right as opposed to BM that works right to left.

The precomputation involves the computing of the failure function f, that is used when the comparison of $P[j]$ to $T[i]$ fails, and the pattern/text need to be shifted. This means that although $P[..j-1]$ matches $T[..i-1]$, $P[j] \neq T[i]$. Let us call P_k the prefix of P consisting of the first k characters of P and P_{-k} the suffix of P consisting of the last k characters of P.

The question that arises then is by using the lessons learned from the BM algorithm how to shift P to the right by more than one position. If we shift right $P[]$ by say d positions we have the following case.

```
(a)                                i
T:                          abcdefghX
                            0       j     m-1
P:                          ABCDEFGHYIJKLMN
A mismatch occurs at j : This means that a=A b=B ... h=H and X!=Y
-----------------------------------------------------------------
This is equivalent to having,
(b)                                i
T:                          ABCDEFGHX
                            0       j     m-1
P:                          ABCDEFGHYIJKLMN
-----------------------------------------------------------------
If we shift P to the right d=4 positions we have
(c)                                i
T:                          ABCDEFGHX
P:                              ABCDEFGHYIJKLMN
                                   j
For the shift to be helpful:  E=A, F=B, G=C, H=D, X=E.
```

From cases (a) and (b) a mismatch at position j of the pattern indicates that $P[0..j-1]$ appears in the text just before position i. Therefore a d shift makes sense if a prefix $P[0..k-1]$ of P (like ABCD in Case 3) for some k, is a suffix of $P[1..j-1]$. Therefore if a mismatch or failed match at position j occurs, the algorithm first reviews the failure function $f[j-1]$ to figure out whether there is a prefix of P that is a suffix of $P[1..j-1]$. If $f[j-1]$ maintains the length of the longest prefix of P that is a suffix of P_{j-1} we are done. We shift P right $j-f[j-1]$ positions, and we continue our string matching at position j as before.

```
  KMP(P,T)
1.  f[]=LongestPrefix(P);
2.  i=0;
3.  j=0;
4.  while (i <= n-1) {
5.    if P[j] == T[i] {
6.     if (j==m-1){
7.      return("match at i-m+1");
8.     }
9.     i++; //P[j]==T[i] advance i
10.    j++; //     and j
11.   }
12.   else { // mismatch at j but P[0..j-1] matches T[..i-1]
13.    if (j>0) //try to advance pattern a little
14.     j=f[j-1];
15.    else     // no luck; advance text one character to the right.
16.     i++;
17.   }
18.  }
19. return("no match");
```

How do we compute array $f[]$? Element $f[j]$ is the length of the longest prefix $P[0..]$ of P that is a suffix of $P[1..j]$. For the boundary case $f[0]=0$. For example let $P = ababababca$. Then $f = [0,0,1,2,3,4,5,6,0,1]$. We find $f[]$ by using KMP but we match the pattern against a text which is the pattern P! The array f can thus be computed as follows.

```
   LongestPrefix (P);
1.  f[0]=0; // boundary case
2.  i=1; // i-j >=1
3.  j=0;
4.  while (i <= m-1) {
5.    if P[j] == P[i] { //P[0..j] matches P[i-j..i]
6.     f[i]=j+1;        // j+1 matched characters
7.     i++; //P[j]==T[i] advance i and j
8.     j++; //     maintains i-j >=1
9.    }
10.   else { // mismatch at j but P[0..j-1] matches P[i-j..i-1]
11.    if (j>0)
12.     j=f[j-1];
13.    else {    //  no luck; no match
14.     f[i]=0;
15.     i++;
15a.   }
16.   }
17.  }
18. return(f[]);
```

The analysis of the Knuth-Morris-Pratt algorithm consistst of examining the three possible cases of matches/mismatches and also examine how the difference $i-j$ behaves. Let us call $D = i-j$, i.e. the difference in shift between the text and the pattern. Naturally, $D \leq n$.

The three cases that we have are as follows.

Case 1. If $T[i] == P[j]$ then, i increases by one and j increases by one and D remains the same $D = (i+1)-(j+1) = i-j$.

Case 2. If $T[i] \neq P[j]$ and $j > 0$, then i remains the same, j however changes from j into $f[j-1]$ a decrease by $j - f[j-1]$. Then D changes from $i-j$ into $i-f[j-1]$. Since $f[j-1] < j$, this means that D is increased by $j-f[j-1]$, i.e. by at least one.

Case 3. If $T[i] \neq P[j]$ and $j == 0$, then i increases by 1 and j remains the same. This leads to $D = (i+1)-j$, i.e. an increase by 1 in the value of D.

As a conclusion,

- i increases by at least one in cases 1 and 3.
- D increases by at least one in cases 2 and 3.

Therefore at every iteration of the loop, either i or D increases by at least one. The maximum value for either i or D is n. Therefore the loop of KMP is executed at most $2n$ times.

In order to complete the analysis we need to examine the running time of line 1. By a similar argument it is $O(2m) = O(m)$. Therefore KMP has running time $O(n+m)$.

H.1.5 Approximate matching

Match $P[0..m-1]$ with $T[0..n-1]$ by minimizing the number of edits. We can formulate this problem as the Levenstein distance between two strings P and T. The string operations allowed (the 'edits') are Insert, Delete, and Replace (a single character with another one). Thus for `kitten` we can get `mitten` by replacing one character, for `bidden` and `biden` an insertion of a d is required or a deletion of a d (other way around). A dynaming programming solutions requires $O(nm)$ time for a brute-force algorithm.

Build a matrix M of geometry $m \times n$ to record information about a prefix $P[1..i]$ of the pattern to a suffix of the text $T[1..j]$ at position j and record this information at $M[i,j]$.

```
   Levenshtein(P[1..m], T[1..n])
0.  Allocate M as (m+1) x (n+1) matrix
1.  if m == 0 return(n);
2.  if n==0  return m;
3.  for each j
4.      M[0,j] = j;
5.  for each i
6.      M[i,0] = i;
7.  for each j
8.     for each i
9.         if T[j] == P[i]  cost = 0;
10.           else          cost = 1;
11.        // check three cells
12.        // above, left, diagonal left-up
13.        //  x  z
14.        //  y  *
15.        z= M[i-1,  j] ;
16.        y= M[i  ,j-1] ;
17.        x= M[i-1,j-1] ;
18.        M[i , j] = min { x+ cost , y+1, z+1 }
20.        // if M[i,j] = x + cost => replace
21.        // if M[i,j] = y + 1    => del from T
22.        // if M[i,j] = z + 1    => ins into T
23.  print (M[m,n]; //output from bottom right corner
```

Problem H.1.1

What if $P = survey$ and $T = surgery$?

Part IX

Brief probability theory review

Appendix I

Probability theory

A function $f(x)$ of real values x to real values y, is a mapping such that to every x there corresponds a value y and we describe this by writing $y = f(x)$.

The infimum of a set is the largest lower bound of a set and is the largest value that is smaller than or equal to all elements of the set.

The supremum of a set is the smallest upper bound of a set and is the smallest value that is larger than or equal to all elements of the set.

I.1 Experiments and random processes

An experiment can have a set of outcomes. Flipping or tossing a coin can result with one of two possible outcomes: H for heads or T for tails depending on whether the coin comes heads or tails respectively.

Definition I.1 **Experiment or trial**

An experiment (or trial) is any procedure that can be repeated and generate a well-defined set of outcomes.

Definition I.2 **Sample Space**

The set of all possible outcomes of an experiment is known as sample space. We denote a sample space with the symbol S.

Thus for the coin experiment, $S = \{H, T\}$. Note that in bibliography Ω is used to indicate a sample space.

Definition I.3 **Sample point**

An element of S is known as a sample point.

Definition I.4 **Event**

An event A is a set of outcomes and is a subset of S.

Sometimes we refer to the outcomes of S as elementary events. Let us throw a coin three times. The sample space now is

$$S = \{HHH, HHT, HTH, THH, HTT, THT, TTH, TTT\}$$

The sample space is of cardinality 2^n, where n is the number of repetitions of the coin experiment. An event $E_1 = \{HHT, HTH, THH\}$ indicates an event in which only one Tails was encountered in the experiment. One can also describe this event as the event where the number of Heads is an even positive integer, or an even where the number of Heads is two! There are three possible outcomes of S associated with this event. Another event $E_2 = \{HHH\}$ is a no Tails event.

Definition I.5 **Mutually exclusive events**

Two events A, B are mutually exclusive if and only if $A \cap B = \emptyset$. Three events or more are mutually exclusive if every two of them are mutually exclusive.

Continuing our most recent example, obviously E_1, E_2 are mutually exclusive events since $E_1 \cap E_2 = \emptyset$. We can now provide some formal definitions, including that of a probability space.

In the remainder, a countable sequence is meant to be a finite sequence or a countably infinite sequence.

I.1.1 Event spaces and measurable spaces

Definition I.6 **Event space**

An event space T is a set of events that is, a collection of subsets of a sample space S.

An event space T is usually a σ-algebra, also known as a σ-field.

Definition I.7 **σ-algebra or σ-field**

T is a σ-algebra or σ-field if it contains a collection of subsets of a sample space S that satisfy the following.

1. $S \in T$,
2. $A \in T$ implies that $A' = S - A \in T$ (closedness under complement), and
3. for a countable sequence $A_1, A_2, \ldots, A_i$, where $A_i \in T$ implies $\cup_i A_i \in T$ (closedness under union).

Using DeMorgan's Law $((A \cap B)' = A' \cup B', or (A \cup B)' = A' \cap B')$, one can show also that for a countable sequence A_i, $A_i \in T$ implies $\cap_i A_i \in T$ (closedness under intersection).

Definition I.8 **(S,T): measurable space**

A pair (S, T) is known as a measurable space: it consists of a sample space S of outcomes, and a set of events T that is a σ-algebra.

The pair (S, T) is called measurable space because it is possible to put a measure on it.

Definition I.9 **Topology**

Y is a topology (or topological space), if it contains a collection of subsets of a space S that satisfy the following.

1. $S \in Y$, $\emptyset \in Y$,
2. for an arbitrary sequence A_i, $A_i \in Y$ implies $\cup_i A_i \in Y$ (closedness under union),
3. for a finite sequence A_i, $A_i \in Y$ implies $\cap_i A_i \in Y$ (closedness under intersection).

Thus in a σ-algebra we have a countable sequence of unions (and by implication intersections) plus complementation, whereas in a topology we have an arbitrary sequence of unions and a finite sequence of intersections. The magic words are countable and arbitrary and finite.

A σ-algebra requires the complement of a set to be in T; a topology does not require it, though this is the case for the two sets S and $\emptyset$. A topology is associated with closeness (neighborhood) whereas a σ-algebra with length (measure or weight). With a topology we intend to determine whether a function is continuous or not. In probability spaces (σ-algebra) we want to assign values to sets (event probabilities). For every topology one can associate it with a σ-algebra to it through the notion of a Borel-space or algebra.

Example I.1.1

A (S,T) is a topological space where $S=\{a,b,c\}$ and $T=\{\emptyset,S,\{a,b\},\{b,c\},\{b\}\}$. But it is not a σ-algebra since the complement of $\{a,b\}$ is not in T for example.

Example I.1.2

A finite intersection of open intervals $\cap_{m\in\{1,3,5\}}(a-\frac{1}{m},b+\frac{1}{m})$ preserves the open interval notion. On the other hand, a countable intersection of intervals $\cap_{m\in N}(a-\frac{1}{m},b+\frac{1}{m})$ is a closed interval $[a,b]$. Consider also the case $a=b=0$ or $a=0,b=1$.

A σ-algebra is ready to be measured. This is why (S,T) is also known as a measurable space. Every element A of T can be assigned a number, a probability.

Example I.1.3

For a sample space S of a measurable space (S,T), and $s\in S$, we call $\{s\}$ an elementary event. Moreover $\emptyset$ and S are also events of T. The impossible event or null event is the $\emptyset$. If A,B are events so are $A\cap B$ and $A\cup B$ or A' for example.

A measure m is a non-negative function on $\mathbb{R}$ that can be applied to elements of the event space T of (S,T).

Definition I.10 **Measure m**

A measure m is a function on T defined as $m:T\mapsto\mathbb{R}$ that satisfies the following.

1. $m(A)\geq m(\emptyset)=0$ for $\forall A\in T$,
2. if $A_i\in T$ is a countable sequence of pairwise disjoint sets, $m(\cup_i A_i)=\sum_i m(A_i)$.

I.1.2 Probability spaces

A probability measure denoted by P is a measure were $\mathbb{R}$ is replaced by the closed interval $[0,1]$ and has one additional property: $P(S)=1$.

Definition I.11 **Probability measure P**

A probability measure P is a function on T defined as $P:T\mapsto[0,1]$ that satisfies the following.

1. $P(A)\geq P(\emptyset)=0$ for $\forall A\in T$,
2. if $A_i\in T$ is a countable sequence of pairwise disjoint sets, $P(\cup_i A_i)=\sum_i P(A_i)$, and
3. $P(S)=1$.

In case (2) above we say that P is countably additive. We use $P(A)=\sum_{s\in A}P(s)$, where $P(s)\geq 0$.

A probability space is the triplet (S,T,P) as defined earlier in this and previous sections. Those definitions are summarized below.

Several times when we use the term probability we mean probability space and we thus imply the existence of the triplet (S,T,P). So several times the existence of (S,T,P) is implicily given.

Definition I.12 **Probability space**

A probability space is a triplet (S,T,P), where

- S is a sample space that is, a set of outcomes,
- $T \subseteq 2^S$ is a σ-algebra on S, that is, a collection of subsets containing S and closed under complement, closed under union (of a countable number of sets), and by DeMorgan implication closed under intersection (of a countable number of sets), and
- P is a probability measure on T with $P(S) = 1$.

The set S is known as the sample space and the elements of S are known as outcomes or elemetary events. For an event $A \in 2^S$, we define $P(A)$ the probability of event A. The probability of an event A is the sum of the probabilities of the elementary events of A.

If S is finite, (S,T,P) is a finite probability space. If S is finite and $T = 2^S$ the probability measure is determined by its values on elementary events. Thus the probability space assigns through $p : S \to [0,1]$ a probability $p(s)$ to every element of s of S such that $p(s) \geq 0$, with $\sum_{s \in S} p(s) = 1$. Then for an event $A \in 2^S$, the probability of the event A is the sum of the probabilities of the elements of S in A i.e. the sum of the probabilities of the elementary events, or in other words $P(A) = \sum_{s \in A} p(s)$.

Theorem I.1

Let (S,T,P) be a probability space. The following apply to events, $A, B, \ldots$ of T.

1. *Monotonicity: If $A \subseteq B$, then $P(A) \leq P(B)$.*
2. *Subadditivity: If $A \subseteq \cup_i A_i$, then $P(A) \leq \sum_i P(A_i)$.*
3. *Continuity below: If $A_1 \subset A_2 \subset \ldots$ and $\cup_i A_i = A$ then $\lim_{i \to \infty} P(A_i) = P(A)$.*
4. *Continuity above: If $A_1 \supset A_2 \supset \ldots$ and $\cap_i A_i = A$ with $P(A_i) < \infty$, then $\lim_{i \to \infty} P(A_i) = P(A)$.*

Definition I.13 **Discrete probability space**

A discrete probability space has S that is a countable set. Then $P(A) = \sum_{s \in A} p(s)$, where $p(s) \geq 0$, and $\sum_{s \in S} p(s) = 1$.

Sometimes we refer to it as finite probability space. Let S be a finite sample space $S = \{s_1, \ldots, s_n\}$. A probability model or finite probability space is obtained by assigning to each point $s_i \in S$ a real number p_i (or $p(i)$), the probability of s_i, that satisfies the following properties.

- Each p_i is nonegative $p_i \geq 0$, and
- the sum of p_i is one i.e. $\sum_{1 \leq i \leq n} p_i = p_1 + p_2 + \ldots + p_n = 1$.

Definition I.14 **Equiprobable space**

For a finite probability space S of n sample points, if each sample point has the same probability as any other one, the sample space is called equiprobable space.

I.1.3 Probability of events

Lemma I.1

For any collection of events $A_1, \ldots, A_n$,

$$P(A_1 \cup \ldots \cup A_n) \leq P(A_1) + P(A_2) + \ldots + P(A_n) = \sum_{i=1}^{n} P(A_i).$$

Lemma I.2 **Independent Events**

Two events A and B are independent if

$$P(A \cap B) = P(A)P(B)$$

Theorem I.2 **Properties of events**

Consider two events A, B of probability space (S, T, P).

$$P(\emptyset) = 0,$$

$$P(A - B) = P(A) - P(A \cap B),$$

and for $A \subseteq B$ we have

$$P(A) \leq P(B).$$

Moreover,

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

The latter equality is also known as the inclusion-exclusion property or principle. It can be generalized to three or more events (with a proof by induction). It is also straightforwars to derive the following.

Corollary I.1

Given two events A, B we have

$$P(A \cup B) \leq P(A) + P(B)$$

Theorem I.3 **Inclusion-Exclusion of three events**

Given three events A_1, A_2, A_3 we have the following

$$\begin{aligned} P(A_1 \cup A_2 \cup A_3) &= P(A_1) + P(A_2) + P(A_3) \\ &\quad - P(A_1 \cap A_2) - P(A_2 \cap A_3) - P(A_1 \cap A_3) \\ &\quad + P(A_1 \cap A_2 \cap A_3). \end{aligned}$$

Example I.1.4

An experiment is performed by throwing (tossing) a coin. The experiment is repeated twice. The combination of the results of the two individual experiments is the experiment in question. The sample space S is then $S = \{HH, HT, TH, TT\}$ and indicates the outcomes of the first and second toss of the coin: H indicates heads, T indicates tail as an outcome. AB indicates that A is the outcome of the first experiment, B is the outcome of the second experiment, where A, B is H or T. Event X is $X = \{HH, TT\}$ i.e. even number of H or T. Event Y is $Y = \{HH, HT, TH\}$ i.e. at least one H.

I.1.4 Conditional probabilities

Theorem I.4 **Conditional Probability**

Let A, B be two events of a finite probability space (S, T, P) and $P(B) > 0$. The probability that an event A occurs conditional on event B had already occurred is known as the conditional probability of A given B and denoted $P(A|B)$. It is given by

$$P(A|B) = P(A \cap B)/P(B).$$

Moreover

$$P(A \cap B) = P(A|B)P(B).$$

Theorem I.5 **Generalization of conditional probability**

$$P(A_1 \cap A_2 \cap \ldots \cap A_n) = P(A_1) \cdot P(A_2|A_1) \cdot P(A_3|A_1 \cap A_2) \cdot P(A_4|A_1 \cap A_2 \cap A_3) \cdot \ldots \cdot P(A_n|A_1 \cap A_2 \cap A_3 \cap \ldots \cap A_{n-1})$$

Theorem I.6

If sample space is partitioned into events $A_1, \ldots A_n$ then for some event A we have,

$$P(A) = \sum_{i=1}^{n} P(A/A_i)P(A_i).$$

We refer to the following as Bayes' rule.

Theorem I.7

If sample space is partitioned into events $A_1, \ldots A_n$ then for some event A we have,

$$P(A_i/A) = \frac{P(A \cap A_i)}{P(A)} = \frac{P(A/A_i)P(A_i)}{\sum_i P(A/A_i)P(A_i)}$$

Moreover

Theorem I.8

$P(A_1/A_2) = \frac{P(A_2/A_1)P(A_1)}{P(A_2)}$.

Definition I.15 **Properties of independent events**

Let A, B be two events of finite probability space (S, T, P). A is independent of B if

$$\text{A is independent of B } \Leftrightarrow P(A) = P(A|B).$$

Corollary I.2

From $P(A|B) = P(A \cap B)/P(B)$ since $P(A|B) = P(A)$ we have $P(A) = P(A \cap B)/P(B)$ which implies that $P(A \cap B) = P(A)P(B)$. Moreover $P(B|A) = P(B)$.

Theorem I.9

For any event A, we have

$$E(I_A) = P(A).$$

Definition I.16 **Independent Repeated Experiments**

Let (S, T, P) be a finite probability space. A space of n independent trials is space S_n consisting of ordered n-tuples of elements of S with the probability of an n-tuple defined to be the product of the probabilities of its components.

$$P((s_1, \ldots, s_n)) = P(s_1) \ldots P(s_n).$$

The sample space S of an experiment or trial or process has outcomes that might be number or might not be numbers. Think about the experiment of tossing a coin. Sometimes instead of using symbolic values or names to

an outcome such as H or T we prefer to use numeric values such as 1 and 0 respectively. This mapping is known as random variable.

Definition I.17

Let S be a sample space and $A_i \subseteq S$ events. For each non empty subjet $I \subseteq \{1,\ldots,n\}$ we define

$$A_I = \cap_{i\in I} A_i.$$

By default $A_0 = S$.

Theorem I.10 **Inclusion-Exclusion**

We have

$$|A_1 \cup \ldots \cup A_n| = \sum_{I\subseteq\{1,\ldots,n\}} (-1)^{|I|+1}|A_I|. = \sum_{I\subseteq\{1,\ldots,n\}} (-1)^{|I|+1}|\cap_{i\in I} A_i|.$$

By DeMorgan's Law the $\cap$ and $\cup$ can be swapped without affecting the result otherwise.
When needed we would like to calculate

$$|A| - \sum_{I\subseteq\{1,\ldots,n+1\}} (-1)^{|I|+1}|\cap_{i\in I} A_i|,$$

or

$$|A| - \sum_{I\subseteq\{1,\ldots,n+1\}} (-1)^{|I|+1}|\cup_{i\in I} A_i|.$$

I.1.5 Random variables

As we mentioned earlier, a probability space is a triplet (S,T,P). In the simplest of the cases, S is a finite set, and $T = 2^S = P(S)$, that is, T is the powerset of S, the set of all posible subsets of S and thus T has cardinality $|T| = 2^{|S|}$. The probability measure P, $P : T \mapsto [0,1]$, essentially becomes, $P : S \mapsto [0,1]$, i.e. it is defined on the elementary events, and by extension to any event subset of T. Then

$$P(A) = \sum_{s\in A} P(\{s\}),$$

and

$$\sum_{s\in S} P(\{s\}) = 1.$$

We can then define arbitrary functions on events.

Definition I.18 **Random variable: simple definition**

A random variable (later, r.v.) X defined on a probability space (S,T,P) is a real-valued measurable function on S. Random variable X is then a mapping

$$X : S \mapsto \mathbb{R}.$$

We can add few more details on this definition.

Definition I.19 **Random variable: more detailed definition**

A random variable (later, r.v.) X defined on a probability space (S,T,P) is a real-valued measurable function that assigns a real value $X(s)$ to an elementary event s of S so that for every x, where $-\infty < x < \infty$, set $\{s : X(s) \leq x\}$ is an event contained in T.

For a random variable we can informally write $P(X = 5)$ to actually mean $P(\{s \in S : X(s) = 5\}$. Moreover, when we informally write $P(5 \leq X \leq 11)$ we actually mean $P(\{s \in S : 5 \leq X(s) \leq 11\})$.

Definition I.20 **Convention**

For a random variable X, when we write $X \in A$ for $A \subseteq \mathbb{R}$, we mean $\{s \in S : X(s) \in A\}$. Then,

$$P(X \in A) = P(\{s \in S : X(s) \in A\}).$$

A random variable and can be discrete or continuous.

Discrete random variables

The discussion below applies to a discrete random variable.

Definition I.21 **Discrete r.v.**

A random variable is discrete if its domain consists of a finite or countably infinite set of outcomes.

We usually denote an event involving random variable X by writing $X = x$. This indicates event $E = \{s \in S : X(s) = x\}$. Moreover, it should be the case that $E \subseteq T$.

Definition I.22 **Probability mass function**

A probability mass function (p.m.f.) of a discrete r.v. X is the function f_X that associates a probability to each $s \in S$.

In other words the p.m.f. of r.v. X is a function that returns $P(X = x)$ for each s in the domain of X with $X(s) = x$. Then $f_X(x) = f(x) = P(X = x) = P(E) = \sum_{s \in E} P(s)$. It is $f_X(x) \geq 0$ and $\sum_{s \in S} f_X(s) = 1$.

For a discrete random variable X that also takes discrete values $x_1, \ldots, x_n$, we have that

$$P(X = x_i) = P(\{s \in S : X(s) = x_i\}) = f(x_i),$$

Theorem I.11 **Range**

The range $R(S,X)$ of random variable X is the set of numeric values assigned to outcomes of a sample space S by random variable X. For function X its range is $R(S,X)$.

Example I.1.5

We have two dice. The sample space of throwing two dice is of cardinality 36, as there are so many pairs (d_1, d_2) where d_1 is one of the outcomes of one die, and likewise for d_2. Use the definitions to calculate $P(X = 4)$, where X is a random variable defined to be the sum $d_1 + d_2$.

Continuous random variables

The discussion below applies to a continuous random variable.

Definition I.23 **Continuous r.v.**

A random variable is continuous if its domain is uncountably infinite.

Definition I.24 **Distribution of a random variable**

The cumulative distribution function of a continuous r.v. X is described as the distribution of a random variable and is denoted by $F_X(x)$ (sometimes, just $F(x)$).

$$F_X(x) = F(x) = P(X \leq x) = P(X^{-1}((-\infty, x])).$$

We note that $0 \leq F_X(x) \leq 1$.

Given X the cumulative distribution function F can be defined as $F_X(x) = P(X \leq x)$, for $-\infty < x < \infty$. One can also define $F_X(x) = P(A)$, where $A = (-\infty, x]$, where $x \in \mathbb{R}$. Function $F_X(x)$ or simply $F(x)$ when it implicitly infers to X, is a monotonically increasing continuous from left function with

$$\lim_{x \to -\infty} F(x) = 0, \qquad \lim_{x \to \infty} F(x) = 1.$$

Furthermore, if there is $f(t)$ such that

$$F(x) = \int_{-\infty}^{x} f(t)dt,$$

then $f(t)$ is the probability density function of X. Function $f(t)$ to exist it means that $F(x)$ is differentiable. In the remainder, as needed, it will be assumed that $F(x)$ is differentiable. An interpretation of the density function is

$$f(x)dx \approx P(x < X \leq x + dx).$$

Theorem I.12

Let (S, T, P) be a probability space. The (cumulative) distribution function $F_X(x)$ of r.v. X has the following properties.

1. *F is nondecreasing.*
2. *$\lim_{x \to \infty} F(x) = 1$, $\lim_{x \to -\infty} F(x) = 0$,*
3. *F is right continuous $\lim_{y \to x^+} F(y) = F(x)$,*
4. *$P(X < x) = \lim_{y \to x^-} F(y)$,*
5. *$P(X = x) = F(x) - P(X < x)$.*

The distribution of X is called continuous if there is a function $f_X(x)$ (called the probability density function) such that

$$F_X(B) = \int_B f_X(x)dx,$$

for every interval B (or in mathematical analysis terms, for every Borel set B) and in that case F_X is continuous.

Definition I.25 **Probability density function**

When the distribution function $F_X(x) = P(X \leq x)$ has the form

$$P(X \leq x) = \int_{-\infty}^{x} f_X(t)dt,$$

we refer to $f_X(x)$ as the probability density function of X and sometimes we denote it $f(x)$ implicitly referring to r.v. X. Moreover,

$$f_X(x) = \frac{dF_X(x)}{dx}.$$

If $F(x)$ is differentiable, then $f(x)$ exists. It is possible however that $F(x)$ is not differentiable.
We can also define a cumulative distribution function for a discrete r.v. X. Then $F(x) = \sum_{t=X(s) \leq x, s \in S} f(t)$.

Composition of random variables

Definition I.26 **Sum and product of a random variables**

Let X, Y be two random variables on the same probability space (S, T, P). Then $X+Y$, $X \cdot Y$ and $a \cdot X$, $a \in \mathbb{R}$ are also random variables and functions on S defined as follows.

$$\forall s \in S : (X+Y)(s) = X(s) + Y(s), \quad \forall s \in S : (X \cdot Y)(s) = X(s) \cdot Y(s), \quad \forall s \in S, a \in \mathbb{R} : (a \cdot X)(s) = aX(s).$$

Likewise for any polynomial function $f(x, y, \ldots, z)$ we define $f(X, Y, \ldots, Z)$ to be a function on S defined analogously.

$$\forall s \in S : f(X, Y, \ldots, Z)(s) = f(X(s), Y(s), \ldots, Z(s)).$$

I.1.6 Expectation of a random variable

The term expectation or expected value is associated with a random variable. The terms mean and average can be used interchangeably with (mathematical) expectation, but the term mean usually refers to a distribution. The expectation of a random variable X would be denoted by $E(X)$ though it is more often to encounter $E[X]$. (Do not be surprised if by accident you see both instances in the remainder.) The term for mean is usually μ or $\bar{X}$.

For the discrete case we have.

Definition I.27 **Expectation of a discrete r.v. X**

Let X be a discrete random variable on probability space (S, T, P) that also takes values $x_1, \ldots$ with $P(X = x_i) = f_X(i)$. Then the expectation (or expected value) of r.v. X is defined as follows.

$$E(X) = \mu = \sum_{x \in X} x f_X(x) = \sum_i x_i f_X(x_i).$$

If the series (sum) converges, then X has finite expectation. If $\sum_i |x_i| f(x_i)$ diverges, then X has no finite expectation.

Bear in mind that dropping subscripts or introducing subscripts, $P_X(X = x_i) = P(X = x_i) = f(x_i) = f_X(i)$. If it does not cause ambiguity, we will be using f for f_X or P for P_X.

Definition I.28 **Expectation of a continuous r.v. X**

Let X be a continuous and differentiable random variable on probability space (S, T, P), with probability density function $f_X(x)$. Then the expectation or (expected value) of r.v. X is defined as follows.

$$E(X) = \int_{-\infty}^{\infty} x f_X(x) d(x).$$

If $E(X) = -\infty$ or $E(X) = \infty$ we say the expectation $E(X)$ does not exist.

Theorem I.13

Let X, Y be r.v. and $a, b \in \mathbb{R}$. Then

$$E(aX + bY) = aE(X) + bE(Y).$$

Corollary I.3

For X and Y r.v. as before,

$$E(X+Y) = E(X) + E(Y),$$

obtainable by setting $a = b = 1$.

The result below applied for any probability space.

Theorem I.14

For any random variables $X_1,\ldots,X_n$ of a probability space, we have that

$$E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} E(X_i).$$

I.1.7 Independent random variables

We can describe independence for the case of a discrete and continuous variable as follows.

Definition I.29 **Independent r.v.**

Let X, Y be two discrete random variables taking also discrete values $x_1,\ldots,x_i,\ldots$ and $y_1,\ldots,y_j,\ldots$. The two random variables X and Y are independent if for all x_i and y_j,

$$P(X = x_i \cap Y = y_j) = P(X = x_i) \cdot P(Y = y_j).$$

In the general case,

$$P(X \in A \cap Y \in B) = P(X \in A) \cdot P(Y \in B).$$

By induction, this can generalize to n random variables. Note that for two events A, B we defined the conditional probability of A on B as $P(A|B) = P(A \cap B)/P(B)$ and defined A is independent of B when $P(A) = P(A/B)$. Substituting the left-hand side of the latter for the left-hand side of the former we conclude $P(A/B) = P(A) = P(A \cap B)/P(B)$ from which we further derive $P(A \cap B) = P(A)P(B)$.

Definition I.30 **Independent r.v.: continuous case**

Let X and Y be two random variables with $F_X(x)$ and $F_Y(y)$ distribution functions respectively. Let $f_{X,Y}(x,y)$ be their joint distribution function. Then X and Y are independent r.v. if, for all x, y,

$$f_{X,Y}(x,y) = f_X(x) \cdot f_Y(y).$$

Theorem I.15

If X and Y are two discrete random variables for a probability space (S,T,P) then if X and Y are independent we have the following.

$$E(XY) = E(X) \cdot E(Y),$$

The result extends for continuous r.v. X and Y.

Example I.1.6

From a prior example of dice tossing. Random variable Y is defined as follows

$$Y : S \to \mathbb{R} \quad \textbf{where } Y((a,b)) = \min(a,b).$$

Find $E(Y)$. Thus for an element $s = (a,b)$ of S, random variable Y assigns a value to this element that is equal to the minimum of the two values of the two faces of the two dice. The range of Y is $R(S,Y) = \{1,2,3,4,5,6\}$. We can compute

$$P(Y = 1) = 11/36, P(Y = 2) = 9/36, P(Y = 3) = 7/36, P(Y = 4) = 5/36, P(Y = 5) = 3/36, P(Y = 6) = 1/36$$

For example, $P(Y=1)$ is derived from the fact that the tosses $(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(2,1),(3,1),(4,1),(5,1),(6,1)$ have 1 as the minimum value. Furthermore the following outcomes $(4,4),(4,5),(4,6),(5,4),(6,4)$ have all minimum value equal to 4, and so on. Thus

$$\mu = E(Y) = 1\cdot 11/36 + 2\cdot 9/36 + 3\cdot 7/36 + 4\cdot 5/36 + 5\cdot 3/36 + 6\cdot 1/36 = \frac{11+18+21+20+15+6}{36} = 2.5277.$$

I.1.8 Indicator random variable

Definition I.31 **Indicator Random Variable**

For an event A we define the indicator random variable I_A as follows.

- $I_A(s) = 1$ if $s \in A$, and
- $I_A(s) = 0$ if $s \notin A$.

The indicator random variable sometimes uses 1 for I. It is also known as the characteristic function of A.

Theorem I.16

Let $X = I_{A_1} + \ldots + I_{A_n}$. Then

$$E(X) = E(I_{A_1} + \ldots + I_{A_n}) = \sum_i P(A_i).$$

Example I.1.7

The expected number of fixed points on a random permutation p on $\{1,\ldots,n\}$ is one. We define a random variable A with

$$A(p) = |\{i : p(i) = i\}|$$

Then we generate $A_i(p) = 1$ if $p(i) = i$ and 0 otherwise. Then $A(p) = \sum_i A_i(p)$.

$$E(A_i) = P[p(i) = i] = 1/n.$$

and

$$E(A) = \sum_i E(A_i) = n\cdot 1/n = 1.$$

Example I.1.8

A coin is tossed twice. The sample space $S = \{HH, TH, HT, TT\}$. We count the number of heads and this becomes random variable X. Thus $R(S,X) = \{0,1,2\}$. The probabilities assigned by function $f(x_i) = P(X = x_i)$ are as follows.

$$P(X=0) = 1/4, P(X=1) = 1/2, P(X=2) = 1/4.$$

Then the expection of r.v. X becomes

$$\mu = E(X) = 0\cdot P(x=0) + 1\cdot P(X=1) + 2\cdot P(X=2) = 0\cdot 1/4 + 1\cdot 1/2 + 2\cdot 1/4 = 1$$

(We thus expect half of the tosses to be H.)

I.1.9 More on expectation

Definition I.32 **Conditional expectation: discrete case**

For two discrete random variables X and Y, the conditional expectation of X given Y is

$$E(X|Y=y) = \sum_{x\in X} xP(X=x|Y=y) = \sum_{x\in X} x\frac{P(X=x,Y=y)}{P(Y=y)},$$

where $P(X=x,Y=y)$ is the joint probability (mass) function of X and Y. An alternative formulation is as follows.

$$E(X|Y=y_j) = \sum_{x_i\in X} xP(X=x_i|Y=y_j) = \sum_{x_i\in X} x_i \cdot \frac{P(X=x_i,Y=y_j)}{P(Y=y_j)},$$

Definition I.33 **Conditional expectation: continuous case**

For two continuous random variables X and Y, with joint density $f_{X,Y}(x,y)$ the conditional expectation of X given Y is given below. Note that $f_{X,Y}(x,y) = f_{X|Y}(x|y)\cdot f_Y(y)$.

$$E(X|Y=y) = \int_{-\infty}^{\infty} xf_{X|Y}(x|y)dx = \frac{1}{f_Y(y)}\int_{-\infty}^{\infty} xf_{X,Y}(x,y)dx.$$

I.1.10 Variance of random variables

The variance of a random variable is the expected value of the squared deviation from the mean of the random variable. The variance of X is denoted $var(X)$ or $Var(X)$ or $\sigma^2(X)$ or just σ^2 if the reference to X is obvious. Then $\sqrt{\sigma^2(X)} = \sigma(X)$ or σ is the standard deviation of X.

Definition I.34 **Variance of r.v. X**

Let X be a random variable on a probability space S i.e. (S,T,P). Then the variance $\sigma^2(X) = var(X)$ of r.v. X is defined as follows.

$$\sigma^2(X) = var(X) = E((X-\mu)^2) = E((X-E(X))^2).$$

The standard deviation of X is defined as $\sqrt{var(X)} = \sigma(X)$.

The variance exists if the second momdent $E(X^2)$ exists. The latter implies that the first moment $E(X)$ also exists.

Corollary I.4

The variance $var(X)$ of a random variable X we have

$$var(X) = E(X^2) - (E(X))^2.$$

Corollary I.5

Let X be a r.v. and for any $a,b\in\mathbb{R}$, we have the following.

$$var(aX+b) = a^2 var(X).$$

Definition I.35

Let X and Y be r.v. then, the covariance of X and Y is defined as follows.

$$cov(X,Y) = E((X - E(X)) \cdot (Y - E(Y))) = E(XY) - E(X)E(Y).$$

Corollary I.6

For r.v. X, Y, we have the following.

$$var(X+Y) = var(X) + var(Y) + 2cov(X,Y).$$

If X and Y are indepdendent $E(XY) = E(X)E(Y)$ and thus $cov(X,Y) = 0$. We then obtain the following.

Corollary I.7

Let X, Y be r.v. that are independent. We have the following.

$$var(X+Y) = var(X) + var(Y).$$

The results can be generalized for $n > 2$ variables. Proofs are by induction on n.

Example I.1.9

Let X be a r.v. such that $P(a \leq X \leq b) = 1$ in other words $a \leq X \leq b$. Show that

$$var(X) \leq \frac{(b-a)^2}{4}.$$

I.1.11 Standard distributions

Bernoulli distribution

Definition I.36 **Bernoulli Trials**

A Bernoulli trial is an experiment that has two outcomes. One outcome is called a success and the other a failure. Let p be the probability of success and $q = 1 - p$ be the probability of failure. We denote such a Bernoulli trial with $b(p)$.

Several times 1 indicates success and 0 indicates a failure in the context of a Bernoulli trial. If however $1/2$ indicates success and $-1/2$ a failure we call the experiment a Rademacher trial.

Corollary I.8

Let X be a random variable that follows a Bernoulli distribution with parameter p that is, $X \sim b(p)$. Sometimes we use $q = 1 - p$. The probability of success and failure are $P(X = 1) = p$ and $P(X = 0) = 1 - p = q$ respectively. The expected value $E(X)$ of X and its variance $var(X) = \sigma^2$ are as follows.

$$E(X) = p, \qquad var(X) = pq = p(1-p).$$

Geometric distribution

Definition I.37

A geometric distribution can be described as the number of Bernoullis trials to get the first success, or equivalently the number of failures before the first success. If random variable X describes the number of Bernoulli trials and variable Y the number of failure to the first success, then $X = Y + 1$. We then say $X \sim g(p)$, where p is the probability of success in a Bernoulli trial.

Corollary I.9

Let X be a random variable that follows a geometric distribution with parameter p that is, $X \sim g(p)$. Sometimes we use $q = 1 - p$. The probability of the first success after $k-1$ failures is $P(X = k) = (1-p)^{k-1} \cdot p$. The expected value $E(X)$ of X and its variance $var(X) = \sigma^2$ are as follows.

$$E(X) = 1/p, \qquad var(X) = (1-p)/p^2.$$

Binomial distribution

Definition I.38 **Binomial process or trial**

A Binomial process or trial is the independent repetition of identical Bernoulli trials. Independent means the outcome of a Bernoulli trial (or experiment) is not dependent of previous outcomes. We denote such a Binomial process with $B(n,p)$, where n indicates the number of Bernoulli trials $b(p)$, and p the property of the Bernoulli trial (probability of success).

Let $B(n,p)$ denote a binomial process or trial or experiment of n independent Bernoulli trials each one $b(p)$.

Theorem I.17

The probability of k successes in a binomial process $B(n,p)$ is denoted by $B(n,p;k)$ and given by

$$B(n,p;k) = \binom{n}{k} p^k (1-p)^{n-k}.$$

Corollary I.10

The probability of one or more success is $1 - B(n,p;0) = 1 - (1-p)^n = 1 - q^n$, where $q = 1 - p$ is the probability of failure.

Corollary I.11

We toss a (fair) coin 8 times. Thus $p = q = 1/2$. The probability of no heads is $1/2^8 = 1/256$. The probability of at least one head is is $1 - (1 - 1/2)^8 = 1 - 1/256$.

Theorem I.18

Let X be a random variable that follows a binomial distribution with parameters n and p that is, $X \sim B(n,p)$. Sometimes we use $q = 1 - p$. The probability of k successes in n trials is B(n,p;k). The expected value $E(X)$ of X and its variance $var(X) = \sigma^2$ are as follows.

$$E(X) = np, \qquad var(X) = \sigma^2(X) = npq.$$

Example I.1.10

Let X_i, $i = 1,\ldots,n$ be n independent Bernoulli variables representing the outcomes of a sequence of n coin tosses with bias p, where $0 < p < 1$ that is, $X_i \sim b(p)$. Consider $A_n = \frac{1}{n}\sum_{i=1}^{n} X_i$. The latter is the fraction of Heads in the n trials i.e. in space $X^n = (X_1,\ldots,X_n)$. Show the following.

$$E(A_n) = p, \quad var(A_n) = p(1-p)/n.$$

In this and later sections we will interchangeably use the following notation for the exponential function: $\exp(Z)$ will denote e^Z.

I.1.12 Moment generating function

Definition I.39

The moment generating function $M_X(t)$ of a random variable X, with cumulative distribution function F_X is defined as follows, provided that the expectation $E(\exp(tX))$ is defined for t in some neighborhood of 0, in other words there exists an $h > 0$ such that for all t in $[-h,h]$, $E(\exp(tX))$ exists.

$$M_X(t) = E(\exp(tX)) = \int_{-\infty}^{\infty} e^{tx} f_X(x)dx.$$

Note that using Taylor's expansion

$$\exp(tX) = 1 + tX + \frac{t^2X^2}{2!} + \frac{t^3X^3}{3!} + \ldots + \frac{t^nX^n}{n!} + \ldots.$$

Taking expectations of both sides we derive the following.

$$E(\exp(tX)) = 1 + tE(X) + \frac{t^2E(X^2)}{2!} + \frac{t^3E(X^3)}{3!} + \ldots + \frac{t^nE(X^n)}{n!} + \ldots.$$

Differentiating n times $M_X(t)$ with respect to t and setting $t = 0$ we obtain the n-th moment $m_n = E(X^n)$ of r.v. X.

Example I.1.11

The Bernoulli process $b(p)$ has $M_X(t) = 1 - p + pe^t$.

Corollary I.12

For a Bernoulli process $b(p)$,

$$M_X(t) = E(\exp(tX)) \leq \exp(p(e^t - 1)).$$

Theorem I.19

Let X_i be independent random variables, with MGF $M_{X_i}(t)$ respectively, $i = 1,\ldots,n$. Let $S_n = \sum_{i=1}^{n} X_i$. Then the following applies for the MGF of random variable S_n.

$$M_{S_n}(t) = M_{X_1}(t)\ldots M_{X_n}(t) = \prod_{i=1}^{n} M_{X_i}(t).$$

Furthermore, if X_i are identically distributed to random variable X, then

$$M_{S_n}(t) = (M_X(t))^n.$$

Example I.1.12

The Uniform distribution $U(a,b)$ has

$$M_X(t) = \frac{e^{tb} - e^{ta}}{t(b-a)}.$$

Example I.1.13

The binomial distribution $B(n,p)$ has

$$M_X(t) = (1-p+pe^t)^n.$$

Example I.1.14

The normal distribution $N(\mu,\sigma^2)$ has

$$M_X(t) = \exp(\mu t + 0.5\sigma^2 t^2).$$

Example I.1.15

The Poisson distribution $Poisson(\lambda)$ has

$$M_X(t) = \exp(\lambda(e^t - 1)).$$

Example I.1.16

The geometric distribution $g(p)$ with parameter p, has

$$M_X(t) = \frac{pe^t}{1-(1-p)e^t} = \frac{pe^t}{1-e^t+pe^t}.$$

I.1.13 Probability inequalities

Definition I.40 **Convex function**

Let $f(x)$ be a real valued function on some interval $[a,b]$ with $b \geq a$. Function f is called convex if for all $x_1 \neq x_2 \in [a,b]$, and for all λ such that $0 \leq \lambda \leq 1$ we have

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2).$$

In other words, a function f is convex if the line segment between any two distinct points on the graph of the function lies above or on the graph between the two points.

Equivalently, A differentiable function of one variable is convex on an interval if and only if its graph lies above all of its tangents i.e. $f(x) \geq f(y) + f'(y)(x-y)$. If function f on one variable is twice differentiable, then f is convex if $f''(x) \geq 0$.

Theorem I.20 **Markov's inequality**

For a non-negative random variable X with expectation μ that is thus non negative and every positive $a > 0$ we have

$$P(X \geq a) \leq \frac{E(X)}{a}.$$

Theorem I.21 **Cauchy-Schwartz inequality**

Let A and B be two random variables. with $E(A^2) < \infty$ and $E(B^2) < \infty$. Then

$$|E(AB)| \leq \sqrt{E(A^2)}\sqrt{E(B^2)}.$$

Theorem I.22 **Chebyshev's inequality**

Let X be a random variable X with finite expected value μ and non-zero and finite standard deviation σ (variance σ^2). Then for every positive $t > 0$,

$$P(|X - \mu| \geq \sigma t) \leq \frac{1}{t^2}.$$

If one replaces σt with t the following alternative form is derived.

Corollary I.13 **Chebyshev variation 1**

An alternative form is the following bound.

$$P(|X - \mu| \geq t) \leq \frac{var(X)}{t^2} = \frac{\sigma^2}{t^2}.$$

The following form starts from $P(|X| \geq t)$ instead of $P(|X - \mu| \geq t)$ or $P(|X - \mu| \geq \sigma t)$.

Corollary I.14 **Chebyshev variation 2**

An alternative form is the following bound.

$$P(|X| \geq t) \leq \frac{E(X^2)}{t^2}.$$

One can obtain sharper bounds for small variance by considering $P(|X - \mu|^k \geq \sigma t)$, where $k > 2$.

Example I.1.17

Let S be a random variable that is defined as $S_n = X_1 + \ldots + X_n$, where X_i are independent random variables. Then $var(S_n) = \sum_i var(X_i)$. Chebyshev's inequality is then given in the forn below

$$P\left(\frac{1}{n}|\sum_{i=1}^{n}(X_i - E[X_i])| \geq t\right) \leq \frac{\sigma^2}{nt^2},$$

where $\sigma^2 = (1/n)\sum_i var(X_i)$.

If we repeat an experiment n times, we can consider the outcome of each experiment a random variable and thus describe the n experiments with r.v. $X_1, X_2, \ldots, X_n$, where X_i is the random variable associated with the i-the outcome. If the experiments are independent of each other, then $P(X_i = x_k, X_j = x_l) = P(X_i = x_k)P(X_j = x_l)$.

Example I.1.18

Let X_i, $i = 1, \ldots, n$, be n (independent) Bernoulli trials with $X_i \sim b(1/2)$. (X_i are 0 or 1.) Let $S_n = \sum_i X_i$ as before. Show that $Pr(S_n \geq 3n/4) \leq 2/n$.

Theorem I.23 **Jensen's inequality**

Let g be a convex function on $\mathbb{R}$ that is, $g:\mathbb{R}\mapsto\mathbb{R}$. Let X be a random variable, and $Y=g(X)$ another random variable defined. The the following applies

$$E(g(X))\geq g(E(X)).$$

Definition I.41 **Sample mean or sample average**

Let X be a r.v. with expectation (mean) $E(X)=\mu$ and standard deviation σ. Let an experiment is repeated n times with repetitions independent of each other. Let X_i be the random variable associated with the i-th repetition $X_i\sim X$. Then the sample mean or sample average of $X_1,\ldots,X_n$ is defined as follows.

$$\bar{X}=\frac{X_1+X_2+\ldots+X_n}{n}.$$

The sample mean $\bar{X}$ is also a random variable.

Definition I.42 **Law of Large Numbers**

For any $r>0$ the probablity that the sample mean $\bar{X}$ of n independent experiments has a value in the interval $[\mu-r,\mu+r]$ is to the limit for large n equal to 1.

$$lim_{n\to\infty}P(|X-\mu|\leq r)=0$$

The following result in the form of Equations (I.2) and (I.3) appears in Feller [9] and in its complete form including Equation (I.1) as Theorem 1.1 in Bóllobás [3], where $B(n,p;m)$ is expanded and bounded,

Theorem I.24 **Feller[9], Bóllobás[3]**

Let random variables X_i be inpdependent and follow a Bernoulli distribution that is, $X_i\sim b(p)$, and let $S_n=\sum_{i=1}^{n}X_i$. Then for $m=\lceil upn\rceil$, $u>1$,

$$P(S_n\geq m)\leq\frac{u}{u-1}B(n,p;m). \tag{I.1}$$

Furthermore,

$$P(S_n\geq m)\leq\frac{m(1-p)}{(m-np)^2}. \tag{I.2}$$

and

$$P(S_n\leq m)\leq\frac{(n-m)p}{(np-m)^2}. \tag{I.3}$$

The first form appears in [3]; the other two in [9].

I.1.14 Example

Example I.1.19

Let $X_1,\ldots X_n$ be bernoulli $b(p)$ random variables. Then $S_n=\sum_i X_i$ is such that $S_n\sim B(n,p)$. Then $E(S_n)=np=\mu$ and $var(S_n)=\sum_i var(X_i)=npq$ as noted earlier. Let us reformulate this as follows: let $nX\sim B(n,p)$ that is $S_n=nX$. For any $\varepsilon>0$ we have the following using Chebyshev's inequality.

$$P(\frac{1}{n}|S_n-E(S_n)|>t)=P(|X-E(X)|>t)\leq pq/nt^2$$

Proof. By Chebyshev's inequality

$$P(|S_n - E(S_n)| > t) \leq var(X)/t^2$$

Starting with

$$P(\frac{1}{n}|S_n - E(S_n)| > t) = P(|S_n - E(S_n)| > n \cdot t)$$

and the latter by Chebyshev's inequality

$$P(|S_n - E(S_n)| > n \cdot t) \leq var(S_n)/n^2t^2$$

Therefore

$$P(\frac{1}{n}|S_n - E(S_n)| > t) \leq var(S_n)/n^2t^2 = \sum_i var(X_i)/n^2t^2 = npq/n^2t^2 = pq/nt^2.$$

Furthermore we note, $S_n/n = X$ and $E(S_n)/n = E(X)$ and the result follows. ■

I.2 Chernoff's method

In [4] bounds on the tails of a set of Bernoulli trials are discussed in the form of Theorem 1. Theorem 1 of [4] is restated below as Theorem I.25 after some relevant definitions. One can extract a variety of bounds out of Theorem I.25. Such bounds can lead to a sequence of lemmas such as Lemma I.3, Lemma I.4, Lemma I.5, and Lemma I.6 for the right tails, and Lemma I.7, and Lemma I.8, and Lemma I.9 for the left tails, and their associated corollaries along with some obvious concentration bounds (left and right tail bounds).

Definition I.43

Let X_i, $i = 1, \ldots, n$ be independent random variables identical in distribution to random variable X whose moment generating function is $M_X(t) = E(e^{tX})$, and its cumulative distribution function is $F_X(x) = P(X \leq x)$. Let $E(X) = p = \mu$ and define $S_n = \sum_{i=1}^{n} X_i$ and $E(S_n) = nE(X) = n\mu$, and

$$m(r) = \inf E(e^{t(X-r)}) = \inf e^{-rt} M_X(t) = \inf e^{-rt} E(e^{tX}).$$

The infimum is with respect to the t's values. $M_X(t)$ attains a minimum value $m(0)$.

Skipping some other details, the t value for the minimum is finite unless $P(X > 0) = 0$ or $P(X < 0) = 0$ and then $m(0) = P(X = 0)$. If $P(X \leq 0) > 0$ and $P(X \geq 0) > 0$ then $m(0) > 0$. The following is Theorem 1 of Chernoff [4]

Theorem I.25 **Chernoff [4]**

If $E(X) < \infty$ and $r \geq E(X)$ then

$$P(S_n \geq nr) \leq (m(r))^n. \tag{I.4}$$

If $E(X) > -\infty$ and $r \leq E(X)$ then

$$P(S_n \leq nr) \leq (m(r))^n. \tag{I.5}$$

If $0 < u < m(r)$ and $E(X)$ might not exist,

$$\lim_{n \to \infty} \frac{(m(r) - u)^n}{P(S_n \leq nr)} = 0. \tag{I.6}$$

In most of the discussion to follow we will assume that X_i and X follow a Bernoulli distribution i.e. $X_i \sim b(p)$ and $X \sim b(p)$ and thus $E(X_i) = E(X) = p$, where $0 < p < 1$. Then, $S_n \sim B(n, p)$. Therefore we have the following.

$$M_X(t) = E(e^{tX}) = e^t \cdot p + 1 \cdot (1 - p) = 1 + p(e^t - 1) \leq e^{pe^t - p}.$$

The last part is because for all x, we have $e^x \geq 1 + x$.

Definition I.44 **Kullback-Leibler**

The Kullback-Leibler divergence D_{KL} or just simply D is defined as follows for two distributions of n elements $P = \cup_i \{p_i\}$, $p_i \geq 0$, $Q = \cup_i \{q_i\}$, $q_i \geq 0$, $i = 1, \ldots, n$ such that $\sum_i p_i = \sum_i q_i = 1$.

$$D_{KL}(P||Q) = D(P||Q) = \sum_i p_i \ln \frac{p_i}{q_i}. \tag{I.7}$$

I.2.1 Derived right tails

We derive the first Chernoff bound for a Binomial r.v. S_n which is the sum of n Bernoulli random variables, using Eq.(I.4).

Lemma I.3

Let X_i be independent random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any r such that $p < r < 1$ we have the following.

$$P(S_n \geq rn) \leq \exp(-D(r||p)n) = \left[\left(\frac{p}{r}\right)^r \left(\frac{1-p}{1-r}\right)^{1-r} \right]^n. \tag{I.8}$$

Sometimes $r = p + t$ and the bound $p < r < 1$ becomes $p < p + t < 1$ or equivalently $0 < t < 1 - p$. We report this variant in Corollary I.20.

We generate one more bound from Eq.(I.4). The bound is stronger than the Corollaries to follow that bring it into an easier to deal with form.

Lemma I.4

Let X_i be independent random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any r such that $p < r < 1$ we have the following, after we substiture $r = (1+\delta)p$, $\delta > 0$.

$$P(S_n \geq rn) = P(S_n \geq (1+\delta)pn) \leq \left(\frac{e^r \cdot p^r}{e^p \cdot r^r} \right)^n = \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}} \right)^{pn} \tag{I.9}$$

A simpler proof of Eq. (I.8) for binomial random variables is found in [5] and the proof is presented below.

Lemma I.5 **[5]**

Let X_i be independent Bernoulli random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any r such that $p < r < 1$ we have the following.

$$P(S_n \geq rn) \leq \exp(-D(r||p)n) = \left[\left(\frac{p}{r}\right)^r \left(\frac{1-p}{1-r}\right)^{1-r} \right]^n. \tag{I.10}$$

We present an alternative Chernoff bound formulation which follows the simple case of Hoeffding bounds, where $a_i = a = 0$ and $b_i = b = 1$. Naturally it applies to Bernoulli or binary random variables (e.g. Rademacher).

Lemma I.6

Let X_i be independent Bernoulli random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any r such that $p < r < 1$ we have the following.

$$P(S_n \geq rn) \leq \exp\left(-2n(r-p)^2\right). \tag{I.11}$$

The following corollary is more widely known than Lemma I.6.

Corollary I.15

Let X_i be independent Bernoulli random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any r such that $p < r < 1$ we have the following.

$$P(S_n - E(S_n) \geq rn) \leq \exp\left(-2nr^2\right). \tag{I.12}$$

The following Corollary can be obtained from Lemma I.4. It provides a more tangible upper bound than the generic one of the Lemma.

Corollary I.16

Let X_i be independent random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any $\delta > 2e - 1$. we have the following,

$$P(S_n \geq rn) = P(S_n \geq (1+\delta)pn) \leq 2^{-(1+\delta)pn}. \tag{I.13}$$

The following Corollary is also obtained from Lemma I.4. Note that $\delta > 0$ is better than the δ used in Corollary I.19 that follows it.

Corollary I.17

Let X_i be independent random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any $\delta > 0$ we have the following.

$$P(S_n \geq (1+\delta)pn) \leq \exp\left(\frac{-\delta^2}{2+\delta} \cdot pn\right). \tag{I.14}$$

A small improvement has been proposed by McDiarmid in the following form.

Corollary I.18 **[17]**

Let X_i be independent random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any $\delta > 0$ we have the following.

$$P(S_n \geq (1+\delta)pn) \leq \exp\left(\frac{-\delta^2}{2+2\cdot\delta/3} \cdot pn\right). \tag{I.15}$$

The following Corollary can also be obtained from Lemma I.4. It is similar to the one in [2] for the binomial case.

Corollary I.19

Let X_i be independent random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any δ such that $0 < \delta < 1$ we have the following.

$$P(S_n \geq rn) = P(S_n \geq (1+\delta)pn) \leq \exp\left(\frac{-\delta^2}{3} \cdot pn\right). \tag{I.16}$$

We report below a corollary variant of Lemma I.3. This is Corollary I.20.

Corollary I.20

Let X_i be independent random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any t such that $0 < t < 1-p$ we have the following.

$$P(S_n \geq (p+t)n) \leq \exp\left(-D((p+t)||p)n\right) = \left[\left(\frac{p}{p+t}\right)^{p+t}\left(\frac{1-p}{1-p-t}\right)^{1-p-t}\right]^n. \tag{I.17}$$

I.2.2 Derived left tails

We proceed to deriving a Lemma identical to Lemma I.3 for the left tails. This is stated next.

Lemma I.7

Let X_i be independent random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any r such that $0 < r < p$ we have the following.

$$P(S_n \leq rn) \leq \exp\left(-D(r||p)n\right) = \left[\left(\frac{p}{r}\right)^{r}\left(\frac{1-p}{1-r}\right)^{1-r}\right]^n. \tag{I.18}$$

Sometimes $r = p - t$, and the bound $0 < r < p$ becomes $0 < p - t < p$ or equivalently $0 < t < p$.
We generate one more bound below.

Lemma I.8

Let X_i be independent random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any r such that $0 < r < p$ or, equivalently, for any δ such that $0 < \delta < 1$ we have the following.

$$P(S_n \leq rn) = P(S_n \leq (1-\delta)pn) \leq \left(\frac{e^r \cdot p^r}{e^p \cdot r^r}\right)^n = \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{pn} \tag{I.19}$$

There is a weaker but more easier to deal bound for small p. This is shown next. It is similar to the one in [2] for the binomial case.

Corollary I.21

Let X_i be independent random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any δ such that $0 < \delta < 1$ we have the following.

$$P(S_n \leq (1-\delta)pn) \leq \exp\left(\frac{-\delta^2}{2} \cdot pn\right). \tag{I.20}$$

The symmetric case for the left tails to Lemma I.6 is stated below.

Lemma I.9

Let X_i be independent Bernoulli random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any r such that $0 < r < p$ we have the following.

$$P(S_n \leq rn) \leq \exp\left(-2n(r-p)^2\right). \tag{I.21}$$

The following corollary is then evident.

Corollary I.22

Let X_i be independent Bernoulli random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any r such that $0 < r < p$ we have the following.

$$P(S_n - E(S_n) \leq rn) \leq \exp\left(-2nr^2\right). \tag{I.22}$$

I.2.3 Derived concentration bounds

Finally the following Corollary can also be obtained from Corollary I.19 and Corollary I.21.

Corollary I.23

Let X_i be independent random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any δ such that $0 < \delta < 1$ we have the following,

$$P(|S_n - np| \geq \delta pn) \leq 2 \cdot \exp\left(\frac{-\delta^2}{3} \cdot pn\right). \tag{I.23}$$

The following is a direct consequence of Corollary I.15 and Corollary I.22.

Corollary I.24

Let X_i be independent Bernoulli random variables following the same distribution as random variable X, $X \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$. Then, for any r such that $0 < r < p$ we have the following.

$$P(|S_n - E(S_n)| \geq rn) \leq 2 \cdot \exp\left(-2n(r-p)^2\right). \tag{I.24}$$

I.2.4 Hoeffding's method

We now present Theorem 1 and Theorem 2 of Hoeffding [13]. The work [13] deals with random variables that are not necessarily Bernoulli but are bounded e.g. $0 \leq X_i \leq 1$ or $a_i \leq X_i \leq b_i$. Theorem 1 [13] for the case of Bernoulli trials takes the form of Corollary I.20 associated with a Chernoff bound.

That's why several times the terms Chernoff and Hoeffding refer to the same bound. Theorem 1 of [13] is the strongest among the Hoeffding bounds. The bounds are sufficient for large deviations. Otherwise one can use bounds available in [3] and [9]. Angluin-Valiant bounds [2] are weaker but more useful; the latter are or may be more useful for small p.

I.2.5 Derived right tails

Note that all random variables X in the two theorems that follow are to have finite first and second moments. This Theorem 1 of [13] follows.

Theorem I.26

Let X_i be an independent random variable, $i = 1,\ldots,n$. Let $S_n = \sum_i X_i$, $\bar{X} = S_n/n$ and $p = E(\bar{X})$. Then for any h such that $0 < h < 1-p$ we have the following.

$$\begin{aligned} P(\bar{X} - p \geq h)) &= P(S_n \geq (p+h)n) \\ &\leq \exp(-D((p+h)||p)n) \\ &\leq \left[\left(\frac{p}{p+h}\right)^{p+h}\left(\frac{1-p}{1-p-h}\right)^{1-p-h}\right]^n. \end{aligned} \tag{I.25}$$

Note that if $h > 1-p$, then Eq. (I.25) remains true, and for $h = 1-p$, the right-hand side can be replaced by the limit $h \to 1-p$ which is p^n.

We provide a proof below for the Theorem following the techniques of the Chernoff's method also attributed to Cramér [7]. The theorem also appears as Theorem 5.1 in [16].

Theorem 1 of [13] includes (weaker) upper bounds of Eq.(I.25) of the form $\exp(-nh^2k(p))$, where

$$k(p) = \frac{1}{1-2p}\ln\frac{1-p}{p},$$

for $0 < p < 1/2$, and

$$k(p) = \frac{1}{2p(1-p)},$$

for $1/2 \leq p < 1$. The proof arguments are tedious and we refer to [13]. Furthermore, a weaker bound of the form $\exp(-2nh^2)$ can also be derived. This can be established also through Theorem 2 of [13] that is stated and proved below.

Theorem 2 of [13], utilizes the following result that is proven separately.

Proposition I.1 **Hoeffding [13], Eq (4.16)**

For a random variable X such that $a \leq X \leq b$ with $E(X) = 0$ and for any $t > 0$, we have the following.

$$E[e^{tX}] \leq \exp\left(\frac{t^2(b-a)^2}{8}\right).$$

Theorem 2 of [13] is stated and proved below. Whereas in Theorem 1 of [13] variables X_i were bounded in range by 0 and 1, the bounds next are variable, in the sense that $a_i \leq X_i \leq b_i$.

Theorem I.27

Let X_i be an independent random variable, $i = 1,\ldots,n$. Let $S_n = \sum_i X_i$, $\bar{X} = S_n/n$, where $a_i \leq X_i \leq b_i$ and $p = E(\bar{X})$ and $E(S_n) = np$. Then for any h such that $0 < h < 1-p$ we have the following.

$$P(\bar{X} - p \geq h)) = P(S_n \geq (p+h)n) \leq \exp\left(\frac{-2n^2h^2}{\sum_{i=1}^n (b_i - a_i)^2}\right). \tag{I.26}$$

By symmetry one can also prove the following e.g. by $Y_i = -X_i$.

Theorem I.28

Let X_i be an independent random variable, $i = 1,\ldots,n$. Let $S_n = \sum_i X_i$, $\bar{X} = S_n/n$, where $a_i \leq X_i \leq b_i$ and $p = E(\bar{X})$ and $E(S_n) = np$. Then for any h such that $0 < h$ we have the following.

$$P(\bar{X} - p \leq -h)) = P(S_n \leq (p-h)n) \leq \exp\left(\frac{-2n^2h^2}{\sum_{i=1}^n (b_i - a_i)^2}\right). \tag{I.27}$$

In [16], Theorem I.26 appears as Theorem 5.1, and Theorem I.27 and Theorem I.28 appear as Theorem 5.7 there.

There are variants of Theorem I.26 and Theorem I.27. These include the following ones.

Corollary I.25

Let X_i be an independent random variable, $i = 1,\ldots,n$, such that $a \leq X_i \leq b$. Let $S_n = \sum_i X_i$ and $E(S_n) = np$. Then for any δ such that $0 < \delta < (1-p)/p$ we have the following.

$$P(S_n \geq (1+\delta)pn) \quad \leq \quad \exp\left(\frac{-2n^2\delta^2p^2}{n(b-a)^2}\right). \tag{I.28}$$

A similar corollary to Corollary I.25 can be proven for $P(S_n \leq (1-\delta)pn)$, if one bounds the number of failures rather than successes with Theorem I.27 or if $a_i = 0$ and $b_i = 1$, equivalently consider $Y_i = -X_i$. The upper bound would be identical then.

Corollary I.26 appears in [16] as 5.3 in Corollary 5.2 for $a = 0$ and $b = 1$.

Corollary I.26

Let X_i be an independent random variable, $i = 1,\ldots,n$, such that $a \leq X_i \leq b$. Let $S_n = \sum_i X_i$ and $E(S_n) = np$. Then for any h such that $0 < h < n - np$ we have the following.

$$P(S_n \geq pn + h) \quad \leq \quad \exp\left(\frac{-2h^2}{\sum_{i=1}^n (b_i - a_i)^2}\right). \tag{I.29}$$

Moreover one can combine Corollary I.26 and Corollary I.30 to bound $P(|S_n - pn| \geq h)$.

The following is due to [2]. It also appears as Corollary I.19 and Corollary I.21

Theorem I.29 **Angluin-Valiant[2]**

For every n, p, b with $0 \leq p \leq 1$ and $0 \leq b \leq 1$, we have the following.

$$\sum_{k=0}^{k=\lfloor (1-b)np \rfloor} B(n,p;k) \leq \exp(-b^2np/2),$$

and

$$\sum_{k=\lceil (1+b)np \rceil}^{n} B(n,p;k) \leq \exp(-b^2np/3).$$

For the case of a binomial distribution of Bernouli trials the following becomes available.

Corollary I.27

Let X_i be an independent random variable, $i = 1, \ldots, n$, such that $X_i \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$ and $E(S_n) = np$. Then for any δ such that $0 < \delta < (1-p)/p$ we have the following.

$$P(S_n \geq (1+\delta)pn) \quad \leq \quad \exp\left(-2n\delta^2 p^2\right). \tag{I.30}$$

Corollary I.28

Let X_i be an independent random variable, $i = 1, \ldots, n$, such that $X_i \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$ and $E(S_n) = np$. Then for any h such that $0 < h < n - np$ we have the following.

$$P(S_n \geq pn + h) \quad \leq \quad \exp\left(\frac{-2h^2}{n}\right). \tag{I.31}$$

I.2.6 Derived left tails

Corollary I.29

Let X_i be an independent random variable, $i = 1, \ldots, n$, such that $a \leq X_i \leq b$. Let $S_n = \sum_i X_i$ and $E(S_n) = np$. Then for any δ such that $0 < \delta < 1$ we have the following.

$$P(S_n \leq (1-\delta)pn) \quad \leq \quad \exp\left(\frac{-2n^2\delta^2 p^2}{n(b-a)^2}\right). \tag{I.32}$$

Corollary I.30 appears in [16] as 5.4 in Corollary 5.2 for $a = 0$ and $b = 1$.

Corollary I.30

Let X_i be an independent random variable, $i = 1, \ldots, n$, such that $a \leq X_i \leq b$. Let $S_n = \sum_i X_i$ and $E(S_n) = np$. Then for any h such that $0 < h < 1 - p$ we have the following.

$$P(S_n \leq pn - h) \quad \leq \quad \exp\left(\frac{-2h^2}{\sum_{i=1}^n (b_i - a_i)^2}\right). \tag{I.33}$$

For the case of Bernoulli trials we have the following simplifications.

Corollary I.31

Let X_i be an independent random variable, $i = 1, \ldots, n$, such that $X_i \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$ and $E(S_n) = np$. Then for any δ such that $0 < \delta < 1$ we have the following.

$$P(S_n \leq (1-\delta)pn) \quad \leq \quad \exp\left(-2n\delta^2 p^2\right). \tag{I.34}$$

Corollary I.32

Let X_i be an independent random variable, $i = 1, \ldots, n$, such that $\leq X_i \sim b(p)$, where $0 < p < 1$. Let $S_n = \sum_i X_i$ and $E(S_n) = np$. Then for any h such that $0 < h < 1 - p$ we have the following.

$$P(S_n \leq pn - h) \quad \leq \quad \exp\left(\frac{-2h^2}{n}\right). \tag{I.35}$$

I.2.7 Derived concentration bounds

The following Corollary combines Corollary I.25 with Corollary I.29.

Corollary I.33

Let X_i be an independent random variable, $i = 1, \ldots, n$, such that $a \leq X_i \leq b$. Let $S_n = \sum_i X_i$ and $E(S_n) = np$. Then for any δ such that $0 < \delta < 1$ we have the following.

$$P(|S_n - np| \geq \delta pn) \quad \leq \quad 2\exp\left(\frac{-2n^2\delta^2 p^2}{n(b-a)^2}\right). \tag{I.36}$$

The following Corollary combines Corollary I.26 with Corollary I.30 instead.

Corollary I.34

Let X_i be an independent random variable, $i = 1, \ldots, n$, such that $a \leq X_i \leq b$. Let $S_n = \sum_i X_i$ and $E(S_n) = np$. Then for any h such that $0 < h < 1 - p$ we have the following.

$$P(|S_n - pn| \geq h) \quad \leq \quad 2\exp\left(\frac{-2h^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right). \tag{I.37}$$

I.3 Identities and Inequalities

Inequality I.3.1
For any integer $n >$ we have the following.

$$n! \leq n^n, \quad \text{and} \quad n! \geq (n/2)^{n/2}.$$

We also have the following.

$$n! \leq \left(\frac{n}{2}\right)^{\frac{n}{2}} n^{n/2} \quad \text{and} \quad n! \geq n^{\frac{n}{2}-\frac{\sqrt{n}}{2}}.$$

Stirling's approximation for the factorial.

Inequality I.3.2
For any integer $n > 0$ we have the following.

$$(n/e)^n \leq n! \leq en(n/e)^n.$$

Inequality I.3.3
For any integer $n > 0$ and $1 \leq k \leq n$ we have the following.

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{ne}{k}\right)^k.$$

Inequality I.3.4
For any integer $n > 0$ and $k \leq n$ we have the following.

$$\binom{n}{k} \leq 2^n.$$

Inequality I.3.5
For any integer $n > 0$ we have the following.

$$\frac{2^{2n}}{2\sqrt{n}} \leq \binom{2n}{n} \leq \frac{2^{2m}}{\sqrt{2m}}.$$

Inequality I.3.6
The following holds.

$$\binom{2n}{n} = \frac{(2n)!}{n! \cdot n!} \geq 2^n.$$

Inequality I.3.7 **[3]**
Let $b \leq b+x < a$ and $0 \leq y < b \leq a$. The following hold.

$$\left(\frac{a-b-x}{a-x}\right)^x \leq \binom{a-x}{b}\binom{a}{b}^{-1} \leq \left(\frac{a-b}{a}\right)^x \leq e^{-(b/a)x}.$$

The following also holds.

$$\left(\frac{b-y}{a-y}\right)^y \leq \binom{a-y}{b-y}\binom{a}{b}^{-1} \leq \left(\frac{b}{a}\right)^y \leq e^{-(1-b/a)y}.$$

I.3.1 Series

The primary source for the listed inequalities is [1], [18].

Inequality I.3.8 **Mercator-Newton**

For every $-1 < x \leq 1$ we have

$$\ln(1+x) = \sum_{i=1}^{\infty}(-1)^{i+1}\frac{x^i}{i} = x - \frac{x^2}{2} + \frac{x^3}{3} - \dots$$

Inequality I.3.9

For $-1 \leq x < 1$ we have the following equality by substituting $-x$ for x in Inequality (I.3.8).

$$\ln(1-x) = \sum_{i=1}^{\infty}(-1)\frac{x^i}{i} = -x - \frac{x^2}{2} - \frac{x^3}{3} - \dots$$

Moreover we obtain the following.

$$\ln(n+1) - \ln(n-1) = 2\left(\frac{1}{n} + \frac{1}{3n^3} + \frac{1}{5n^5} + \dots\right).$$

Furthermore we can also conclude that for a finite t we have the following.

$$\ln(1-x) \leq \sum_{i=1}^{t}(-1)\frac{x^i}{i},$$

and the following trivially holds.

$$\ln(1-x) \geq \sum_{i=1}^{t}(-1)\frac{x^i}{i} - \frac{x^t}{t}.$$

Inequality I.3.10

For $x > 1/2$ we have as follows.

$$\ln(x) = \sum_{i=1}^{\infty}\frac{1}{i}\cdot\left(\frac{x-1}{x}\right)^i = \frac{x-1}{x} + \frac{1}{2}\left(\frac{x-1}{x}\right)^2 + \frac{1}{3}\left(\frac{x-1}{x}\right)^3 + \dots$$

Inequality I.3.11

For every $2 \geq x > 0$ we have the following.

$$\ln(x) \geq \sum_{i=1}^{\infty}(-1)^{i+1}(x-1)^i/i$$

Inequality I.3.12

For we have as follows.

$$\exp(-\frac{1}{n}) = \sum_{i=1}^{\infty}(-1)^i \frac{1}{i! \cdot n^i} = 1 - \frac{1}{n} + \frac{1}{2! \cdot n^2} - \frac{1}{3! \cdot n^3} \geq 1 - \frac{1}{n}.$$

I.3.2 Exponential inequalities

The primary source for the listed inequalities is [1], [18].

Inequality I.3.13

Note that for positive integer $n > 0$, we have the following.

$$(1+1/n)^n \leq e, \quad (1+1/n)^{n-1} \geq e, \quad (1-1/n)^{n-1} \geq e^{-1}.$$

They can proved by taking logarithms of both sides, then set $t = 1/n$, $t > 0$, and finally clear determine the monotonicity of a function $f(t)$ of t by calculating its first derivative and $f(0)$.

Inequality I.3.14

The following are true.

$$e^x \geq 1 + x \quad \forall x \in R.$$

For integer $k \geq 1$, we have that

$$e^{1/k} \geq \frac{k+1}{k}, \quad \text{and} \quad e^{-1/(k+1)} \geq \frac{k}{k+1}.$$

In other words,

$$e \geq (1+1/k)^k, \quad \text{and} \quad e^{-1/(k+1)} \geq \frac{k}{k+1}.$$

Inequality I.3.15

For all $x \in R$ and different from zero and $x < 1$.

$$\exp(-\frac{x}{1-x}) < 1 - x < \exp(-x), \quad \frac{1}{1-x} > \exp(x),$$

Furthermore,

$$x < e^x - 1 < \frac{x}{1-x}$$

Inequality I.3.16

For every $x > -1$ we have the following.

$$\frac{x}{1+x} < (1 - e^{-x}) < x.$$

For every $x > -1$ we have the following.

$$1 + x > \exp(\frac{x}{1+x}).$$

For every $0 < x \leq 1.5936$ we have the following.

$$e^{-x} < 1 - (x/2).$$

Inequality I.3.17
For every $x>0, y>0$ we have the following.

$$e^x > (1+\frac{x}{y})^y > e^{\frac{xy}{x+y}}.$$

Inequality I.3.18
For all $x \in R$ and different from zero and $x > -1$

$$1+x > \exp(\frac{x}{1+x}),$$

and

$$\frac{x}{1+x} < 1-\exp(-x) < x.$$

Inequality I.3.19
It is $(1-1/N)^n \geq (1-n/N)$ for any integer $n \geq 1$ and $N > 0$.

I.3.3 Logarithmic and other inequalities

The primary source for the listed inequalities is [1], [18], [2] , [11].

Inequality I.3.20
For every $x > -1$ and $x \neq 0$ we have the following.

$$\frac{x}{1+x} < \ln(1+x) < x.$$

For every $x > 0$ we have the following.

$$\ln(x) \leq x-1.$$

For every $x < 1$ and $x \neq 0$ we have the following.

$$x < -\ln(1-x) < \frac{x}{1-x}.$$

The following is true.

Inequality I.3.21
For every $x > -1$ and $x \neq 0$ we have the following.

$$\frac{x}{1+x} < \ln(1+x) \leq \frac{x(6+x)}{6+4x} < x.$$

Inequality I.3.22
For every $x > 0$ we have the following.

$$\ln(1+x) \geq \frac{x}{1+x/2}.$$

Proof. Consider $f(x) = \ln(1+x) - x/(1+x/2)$. It is $f(0) = 0$. Take the first derivative, and it is positive, for all $x > -1$. $f'(x) = x^2/((2+x)^2(x+1))$, therefore $f(x)$ is increasing for $x \geq 0$. ■

Inequality I.3.23

For every $\infty > x \geq 0$ we have the following.

$$\frac{2x}{2+x} \leq \ln(1+x) \leq \frac{x(x+2)}{2\cdot(x+1)}.$$

For every $-1 < x \leq 0$ we have the following.

$$\frac{2x}{2+x} \geq \ln(1+x) \geq \frac{x(x+2)}{2\cdot(x+1)}$$

Inequality I.3.24

For every $x \geq 0$ we have, as a consequence of Inequality I.3.8, the following hold.

$$\ln(1+x) \leq x - x^2/2 + x^3/3,$$

and

$$\ln(1+x) \geq x - x^2/2.$$

Inequality I.3.25

For every $0 < x < 0.5828$ we have the following.

$$|\ln(1-x)| < 1.5x.$$

Inequality I.3.26

For every $0 \leq \delta < 1$ we have the following.

$$(1-\delta)\ln(1-\delta) \geq -\delta + \delta^2/2.$$

Proof. For $0 \leq x < 1$,

$$\begin{aligned}
\ln(1-x) &= -\sum_{i=1}^{\infty} x^i/i \\
&= -x - x^2/2 - x^3/3 - x^4/4 - \ldots \\
(1-x)\ln(1-x) &= -x - x^2/2 - x^3/3 - x^4/4 - \ldots \\
&+ +x^2 + x^3/2 + x^4/3 + x^5/4 + \ldots \\
(1-x)\ln(1-x) &= -x + x^2/2 + x^2/6 + \ldots
\end{aligned}$$

The missing terms in the last form of the equation are positive. We can then conclude

$$(1-x)\ln(1-x) \geq -x + x^2/2 + x^2/6 \geq -x + x^2/2$$

■

Inequality I.3.27
For every $0 \leq \delta \leq 1$ we have the following.

$$(1+\delta)\ln(1+\delta) \geq \delta + \delta^2/3.$$

Proof. For $0 \leq x \leq 1$,

$$\begin{aligned} \ln(1+x) &= x - x^2/2 + x^3/3 - x^4/4 + \dots \\ (1+x)\ln(1+x) &= x - x^2/2 + x^3/3 - x^4/4 + \dots \\ &+ x^2 - x^3/2 + x^4/3 - x^5/4 + \dots \\ (1+x)\ln(1+x) &\geq x + x^2/2 - x^3/6 \end{aligned}$$

We can then conclude since $0 \leq x \leq 1$ that

$$(1+x)\ln(1+x) \geq x + x^2/2 - x^3/6 \geq x + x^2/2 - x^2/6 \geq x + x^2/3.$$

Set $x = \delta$ and the result follows. ■

Inequality I.3.28
For every $2e - 1 < \delta$ we have the following.

$$(1+\delta)\ln(1+\delta) \geq (2e)^{2e}.$$

Inequality I.3.29

(a) For integer $n > 0$ the following holds.

$$\frac{1}{n+1} \leq \ln(1+\frac{1}{n}) \leq \frac{1}{n}$$

(b) For integer $n > 1$ the following holds.

$$\frac{1}{n} \leq -\ln(1-\frac{1}{n}) \leq \frac{1}{n-1}$$

Inequality I.3.30 **[3]**
For every $0 < t < 0.45$ we have the following.

$$\ln(1+t) > t - t^2/2 + t^3/4.$$

For every $0 < t < 0.69$ we have the following.

$$\ln(1-t) > -t - t^2.$$

For every $0 < t < 0.431$ we have the following.

$$\ln(1-t) > -t - t^2 - t^3/3.$$

I.3.4 Combinatorial equalities

All numbers are non-negative integers unless otherwise stated.

Lemma I.10

For any n,k we have the following.

$$\binom{n}{k} = \binom{n}{n-k}$$

Proof. The number of ways to select k objects out of n is the number of ways to UNselect $n-k$ objects out of n thus selecting the remaining k objects. ■

Lemma I.11

For any n,k we have the following.

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Proof. The number of ways to select k objects out of n is equal to the number of selections of k objects that contain 1 plus the number of selections of k objects that DO NOT contain 1. If the selection of k objects contains 1 the number of such selections is to select out of the (out of 1) remaining $n-1$ objects $k-1$ of them and add to the mix 1. This is $\binom{n-1}{k-1}$. If the selection of k objects does not contain 1 the number of such selections is to select out of the (out of 1) remaining $n-1$ objects k of them. This is $\binom{n-1}{k}$. ■

Theorem I.30

For any $x,y \in R$, and n we have the following.

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^n y^{n-k}.$$

Corollary I.35

We have the following.

$$(1+1)^n = 2^n = \sum_{k=0}^{n} \binom{n}{k}.$$

Corollary I.36

$$(1-1)^n = 0 = \sum_{k=0}^{n} \binom{n}{k} (-1)^k.$$

Corollary I.37

The number of subsets of $\{1,2,\ldots,n\}$ of even cardinality is equal to the number of subsets of $\{1,2,\ldots,n\}$ of odd cardinality and thus we have the following.

$$(1-1)^n = 0 = \sum_{k=0}^{n} \binom{n}{k} (-1)^k.$$

Proof.

$$(1-1)^n = 0 = \sum_{k=0}^{n} \binom{n}{k} (-1)^k.$$

implies

$$\binom{n}{0} + \binom{n}{2} + \ldots = \binom{n}{1} + \binom{n}{3} + \ldots$$

■

Corollary I.38

For n and a, we have the following.

$$(1+b)^n = 0 = \sum_{k=0}^{n} \binom{n}{k} b^{n-k}.$$

$$(b+1)^n = 0 = \sum_{k=0}^{n} \binom{n}{k} b^k.$$

Lemma I.12

For any n,k,m, with $k \leq r \leq n$, we have the following.

$$\binom{n}{r}\binom{r}{k} = \binom{n}{k} + \binom{n-k}{r-k}$$

Proof. The number of ways to choose k balls for bin RED and $r-k$ balls for bin BLUE is to select first r balls out of n and then out of the r balls choose the k balls to go to bin RED with the unchosen going to bin BLUE. This is the first left-hand side. Equivalently, First pick out of n the k RED bin balls. From the remaining $n-k$ ones pick the $r-k$ BLUE bin balls. ■

Lemma I.13

For any p,q, and r we have the following.

$$\sum_{k=0}^{r} \binom{a}{k}\binom{b}{r-k} = \binom{a+b}{r}$$

Proof. We have $a+b$ call it n balls, a are RED and b are BLUE. In how many ways can we pick r balls out of n? ■

Lemma I.14

For n and m, where $m \leq n$ we have the following.

$$\sum_{k=0}^{m} \binom{n}{k} (-1)^k = (-1)^m \binom{n-1}{m}.$$

Proof. Use induction on m. ■

Bibliography

[1] M. Abramowitz and I. A. Stegun. Handbook of mathematical functions with formulas, graphs, and mathematical tables. New York: Dover Publications. Ninth printing.

[2] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. Journal of Computer and System Sciences. Volume 18, Issue 2, 1979, Pages 155-193.

[3] B. Béla Bollobás. Random Graphs. Academic Press 1985.

[4] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. The Annals of Mathematical Statistics. 23 (4):493-507, 1952.

[5] V.Chvátal. The tail of the hypergeometric distribution. Discrete Mathematics. 25(3):285-287, 1979, Elsevier.

[6] T. C. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms. 3rd edition. MIT Press.

[7] H. Cramér. Sus un nouveau thèoréme-limite de la théorie des probabilités. Actualités Scientifiques et Industrielles. No 736. Paris 1938.

[8] P. Erdos and J. Spencer. Probabilistic methods in combinatorics. Academic Press. New York, 1974.

[9] W. Feller. An introduction to probability theory and its applications. Vol. 1. 3rd Edition. John Wiley & Sons. New York, 1968.

[10] W. Feller. An introduction to probability theory and its applications. Vol. 2. John Wiley & Sons. New York, 1971.

[11] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. Journal of Parallel and Distributed Computing. 22(1994), pp. 251-267, 1994.

[12] M. T. Goodrich and R. Tamassia. Algorithm design and applications. Wiley. 2014, ISBN 978-1-118-33591-8.

[13] W. Hoeffding. Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association. 58 (301):13-30, March 1963.

[14] S. Kullback and R. A. Leibler. On information and sufficiency. The Annals of Mathematical Statistics. 22 (1):79-86, 1952.

[15] F. T. Leighton, Introduction to Parallel Algorithms and Architectures: Array - Trees - Hypercubes". Morgan Kauffman Publishers, 1992.

[16] C. McDiarmid . On the method of bounded differences. In: Siemons J, ed. Surveys in Combinatorics, 1989: Invited Papers at the Twelfth British Combinatorial Conference. London Mathematical Society Lecture Note Series. Cambridge University Press; 1989:148-188.

[17] C. McDiarmid . Concentration. In: Probabilistic methods for algorithmic discrete mathematics. M. Habib, C. McDiarmid, J. Ramirez-Alfonsin, B. Reed (Eds), Algorithms and Combinatorics 16, Springer, 1998.

[18] National Institute of Standards and Technology. https://dlmf.nist.gov/4.5