

INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

Read along with Document 2 (CS Fundamentals)

Chapters 1-1.1.4, 1.2-1.3, 2.1-2.3, 12.2-12.3 pages 284-285, 305, 324 of GT

*DISCLAIMER: Report an error / typo to the author so that he resolves it.
These notes are NOT to be distributed outside of NJIT without the expressly written permission of A. Gerbessiotis*

Introduction

Data-Type

A variable is an object whose value can change. A constant is an object whose value cannot change. (Several times the value of a constant is the name of the constant.) In programming languages we define data-type(s), primitive data-type(s), composite data-type(s).

Definition 1 (Variable). An object whose value can change.

Definition 2 (Constant). An object whose value cannot change.

Definition 3 (Data-type). Every variable has a data-type, which is the set of values the variable takes.

Moreover, the data-type of a variable defines the operations that are allowable on the variable. There are **built-in (or primitive or elementary)** data types and also **composite** data types. The primitive data types of programming language C++ include `char`, `int`, `double`. The composite data types can be built on top of the primitive ones and include aggregations such as arrays, or methods that allow for composition such as structures and classes (`struct`, `class` , `int[]`). The whole set of data types and the mechanisms that allow for their aggregation define the data model of C++. C++ is a strongly-typed language.

When we write `int x` we define variable `x` to be of an integer data type. C, C++ or Java are **strongly-typed languages**. Some languages such as Python are **weakly-typed languages**. The data type of a variable depends on its value. Thus `x=10` in one line makes `x` to be of an integer data-type but `x='ab'` in the next line makes it to be a string.

Definition 4 (Definition and Declaration). The declaration of a variable includes its declaration of its name and data-type. The definition of a variable includes its declaration plus allocation of space for it.

A variable in a given scope can be defined only once. It can be declared multiple times. And its definition also serves as its first declaration.

Definition 5 (Data model). The data model is an abstraction that describes and defines how data are represented and used.

The data model offered by a programming language such as C++ includes characters, integers of different sizes, and floating-point numbers of different sizes BUT ALSO METHODS (e.g. arrays, classes, structures) that allow for their aggregation and composition (e.g. templates).

Note that the `int` data-type of C++ differs from the integer abstraction as we use it in mathematics; the latter is of unlimited precision.

In programming languages one can compare the data model of C to that of C++ and to that of Java.

Introduction

Abstract Data Type (ADT)

Definition 6 (Abstract data type). An abstract data type (ADT) is a mathematical (data) model and a collection of operations defined on that model.

Therefore in order to describe an ADT we need to complete the following two tasks.

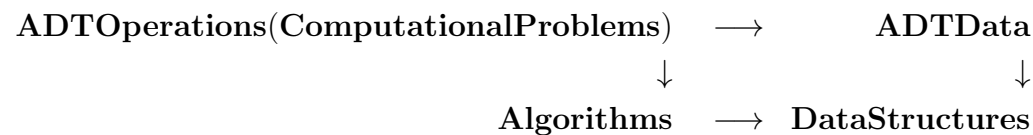
- **Task 1:** Define the data model of the ADT.
- **Task 2:** Describe the operations defined on it.

We need to formally describe the operations of an ADT in the form of a computational problem.

The realization of the data model involved in Task 1 gives rise to the data structure underlying an ADT.

The operations involved in Task 2 and have been described as computational problems generate algorithms that realize them.

The algorithms that realize the operations of an ADT operate on the data structure that has been chosen for the data model of the ADT.



Before we provide formal definitions of data structures and algorithms, let us provide some ADT examples.

ADT Example: Dictionary ADT

Example 1 (Dictionary ADT). In a Dictionary ADT

- (a) the data model consists of words and
- (b) the collection of operations defined on the data (words) are Insert, Search (Lookup) and Delete.

We do not provide the detailed semantics for the three operations as they have been covered in prerequisites. Likewise the term word is used abstractly but implicitly we note that a word is a string of characters.

In different contexts we might define for a dictionary ADT just two of the operations: Insert and Search. For some others all three can be defined.

A Dictionary ADT is an abstract data type consisting of a collection of words (data model) on which a set of operations are defined (Insert, Delete, Search).

Modifying operations: operations such as Insert and Delete might be referred to as modifying operations as they can increase or decrease the number of words in the collection (dictionary).

Probing operation: is the Search as it neither increase or decreases the number of words in the collection (dictionary) but just probes (searches) it.

Note that at this point we make no reference to words such as 'Hash Table', 'Array', 'Search Tree'. The Dictionary ADT relates to an **abstract definition**!

ADT		
Dictionary	Member(s) of Data model	Operations
Instance: D	Words w	Insert(D,w) Search(D,w) Delete(D,w)

ADT Example: Priority Queue ADT

Example 2 (Priority Queue (PQ) ADT). In a PQ ADT

- (a) the data model consists of elements (elt) with some information associated with each one of them (info) and a numeric value associated with the element that is known as the priority i.e. $\text{elt} = \langle \text{info}, \text{priority} \rangle$, and
- (b) the collection of operations defined on the data (elements) are referred to as Insert (an element into an instance of a PQ) and RemoveHigh, which is a delete operation of a specific element of the PQ: the element with the highest priority.

A Priority Queue is another ADT consisting of a collection of elements each of which has a priority and other additional information associated with it. A set of operations defined for a priority queue are Insert and RemoveHigh (or ExtractHigh, or DeleteHigh).

Operation Insert inserts an element into a priority queue; this includes not just the priority of the element (priority) but also all other relevant and auxiliary information (info) that make the element (elt).

Operation RemoveHigh finds and removes the element with the highest priority.

Whether priority 1 is high or low this is not a detail that needs to be addressed explicitly at this level.

More often than not we will be inserting only priority values and ignoring other auxiliary info.

ADT		
PriorityQueue	Member(s) of Data model	Operations
Instance: PQ	$\text{elt} = \langle \text{info}, \text{priority} \rangle$	Insert(PQ,elt) RemoveHigh(PQ)

ADT Counter Example: FIFO and LIFO

In computing a FIFO (First-In First-Out) queue is an ADT that supports two of the three fundamental operations:
an Insert, and a Delete such that the item Delete'd is the first item Insert'ed.

Likewise a LIFO (Last-In First-Out) queue is an ADT that supports two of the three fundamental operations:
an Insert, and a Delete such that the item Delete'd is the last item Insert'ed.

The operations Insert and Delete have special names for an efficient realization or implementation of a FIFO and a LIFO that describe the semantics for an insert operation into a Stack or a Queue and a delete operation into a Stack or a Queue respectively.

LIFO or STACK S	FIFO or QUEUE Q
-----	-----
Data: item t;	Date : item t
Op: Insert(S,t) also known as Push(S,t)	Op: Insert(Q,t) also known as Enqueue(Q,t)
Op: Delete(S) also known as Pop(S)	Op: Delete(Q) also known as Dequeue(Q)

There are efficient implementations of FIFO and LIFO using a Queue and a Stack. The names in this context refer to the data structure rather than the ADT. Most of you have encountered an implementation of Queue using a linked list or of a Stack using an array.

An interesting question is whether using a priority queue and a clock (timer) we can implement a STACK or QUEUE?

The answer is a YES; the solution however (in data structure and algorithm terms) might not be 'optimal'.

ADT Counter Example concluded

STACK (LIFO) S	using a Priority Queue and a timer time	QUEUE (FIFO) Q
PQmin		PQmax
Data : item t ;		Data: item t
Insert(S,t) { Insert(PQmin,< t,time >); }		Insert(Q,t) { Insert(PQmax,< t,time >); }
Delete(S) { return(RemoveHigh(PQmin)); }		Delete(Q) { return(RemoveHigh(PQmax)); }
RemoveHigh:=ExtractMinTime		RemoveHigh:=ExtractMaxTime

One wonders how the semantics can describe operations for both a STACK and a QUEUE ADT.

The trick is hidden in the details of the missing semantics.

For a PQmin priority queue HIGH priority means low (min) time (least recent time).

For a PQmax priority queue HIGH priority means high (max) time (most recent time).

If you ask the question why no one implements a STACK or QUEUE with a PRIORITY QUEUE the answer is simple: it is not (time or space) efficient algorithmically.

Thus in addition to defining DATA STRUCTURES and ALGORITHMS,
we also need to define

SPACE and TIME efficiency of operations realized by ALGORITHMS applying to DATA STRUCTURES.

Note 1 (Collections : Sequence vs Set). A collection is a sequence or a set. In a set the order of its elements does not matter. In a sequence order matters. Thus set $\{1, 2, 3\}$ is the same as $\{2, 1, 3\}$ but sequence $\langle 1, 2, 3 \rangle$ is different from $\langle 1, 3, 2 \rangle$.

Note 2 (Sorted sequence). The sequence $\langle 10, 20, 20, 30 \rangle$ is a **sorted sequence** i.e. non-decreasing or monotonically increasing. The sequence $\langle 30, 20, 10, 10 \rangle$ is a **reverse sorted sequence** i.e. non-increasing or monotonically decreasing.

Note 3 (Symbols). The symbols $:$, $\forall$, $\exists$, $\wedge$, $\vee$ read ‘such that’, ‘for all’, ‘there exists’, ‘logical-AND’, ‘logical-OR’. For definitions, see Document 2.

Note 4 (Logs and Exps). The $\log m$ refers generically to the log base 10 of m i.e. $\log_{10} m$. The Neperian logarithm is $\log_e m$ and usually denoted by $\ln m$. In computing we use instead $\log_2 m$ denoted by $\lg m$. We define by $\lg m$ the base-2 logarithm of m : the logarithm base-two of m is the value $\lg m$ such $2^{\lg m}$ is equal to m . Properties of logs and exponentials are discussed in Document 2.

Exponentiation	$2^m \cdot 2^n = 2^{m+n}$ $(2^m)^n = 2^{m \cdot n}$	$2^m / 2^n = 2^{m-n}$
LogarithmBase2	$2^{\lg n} = n$ $\lg x \cdot y = \lg x + \lg y$	$\lg x/y = \lg x - \lg y$ $\lg x^y = y \cdot \lg x$

Note 5 (Comparator). The colon operator $:$ will be used to denote a comparator. $x : y$ returns -1,0,+1 or negative, 0, positive to indicate $x < y$, $x == y$ or $x > y$ respectively.

Note 6 (Function comparison). How do we compare two functions $f(n)$ and $g(n)$ i.e. how do we realized $f(n) : g(n)$? There is a variety of ways: take the difference $f(n) - g(n)$ and check whether it is $< 0, == 0, > 0$, or consider the fraction $f(n)/g(n)$ and then check whether it is < 1 or $= 1$ or > 1 . Otherwise take $\lg f(n)$ and $\lg g(n)$ and then compare (considering the monotonicity of log) $\lg f(n) : \lg g(n)$, or take $2^{f(n)}$ and $2^{g(n)}$ and compare (considering the monotonicity of exponential) $2^{f(n)} : 2^{g(n)}$.

Question 1. How does $n^{1/\lg n}$ compare to 2? See Document 2 for an answer.

Note 7 (Bit,Byte). Computer memory stores bytes and not bit. The unit name for a bit is bit; b is nonsense and thus incorrect. 1B is 1 byte. Moreover $1KiB$, $1MiB$, etc are to be used for aggregates. See Document 2 for details.

Exponentiation	$1KiB = 2^{10}B$	$1MiB = 2^{20}B$	$1GiB = 2^{30}B$
(Document 2)	Kibi-byte	Mebi-byte	Gibi-byte

Note 8. The number of bit of (unsigned) integer $m > 0$ is denoted by $|m|$ or $\text{bit}(m)$. $|m| = \lfloor \lg m \rfloor + 1 = \lceil \lg(m+1) \rceil$ for integer $m > 0$. Thus $|x| = 3$ since 4 in binary is 100. Likewise if we use m bit we can represent 2^m consecutive unsigned integer values for example $0, \dots, 2^m - 1$.

Definition 7 (**Models of computation**). To add a and b as in $a + b$ the cost of it in the **WordModel** is one (we count operations). In the **BitModel** is proportional to $|a| + |b|$ since we count bit operations. In the WordModel every piece of data has a data type that can be accommodated into a whole word. An **SLP (Straight Line Program)** uses the WordModel to account for operations but program-wise has no for, while, do loops, no if or case statements and no function calls. It is a sequence of statements.

Here is an example of an SLP that computes x^4 for some (positive integer) x .

```
Power(x,4)    // return x**4 using an SLP
1.  t1 = x;
2.  t2 = t1 * t1;  // This is pseudocode
3.  t3 = t2 * t2;
4.  print(t3);
```

Introduction

Data Structures of Computational Problems

Definition 8 (Data structure). A data structure is a representation of the mathematical model underlying an ADT, and thus it is a systematic way of organizing and accessing data.

The representation of an ADT is done in terms of the facilities of a reference programming language: we use or extend the primitive and composite data types of such a language.

Thus the realization or representation of the data part (model) of an ADT is a data structure.

For the Dictionary ADT its data (words) can utilize a variety of data structures such as Array, Hash Table, etc.

Does it matter what data structures we use?

Yes, it matters if **efficiency**, **economy of space** and **easiness of programming** are important.

As running/execution time is paramount in some applications, we would like to access/retrieve/store data as fast as possible. For the Dictionary ADT previously defined, we might use arrays, sorted arrays, linked lists, binary search trees, balanced binary search trees, or hash tables to represent the data model of the ADT known as Dictionary. The semantics of the operations remain the same the realization of the operations and their efficiency depends on the choice of the appropriate or not data structure.

We call algorithms the computational methods that we use to operate on a data structure. That is, the operations of an ADT are realized by respective algorithms. An operation is a computational problem with input (given) and output (expected). An algorithm is the computational method that realizes the input output relationship of the computational problem defined by the corresponding operation of an ADT.

The title of this slide begs for a question.

Question 2. What is the structure of a computational problem?

An answer follows.

Introduction

Computational Problems

Definition 9. A computational problem defines an input-output relationship of computational values; it has an input (e.g. of a set of values) and an output (e.g. of a set of values) and describes how the output can be derived from the input.

Definition 10 (Decision Problem.). A decision problem is a computational problem where the output is one bit for YES and NO (or TRUE and FALSE).

Definition 11 (Optimization Problem.). An optimization problem is a computational problem where the output generates a solution for the problem.

COMPUTATIONAL DECISION PROBLEM

Input : A collection (set, sequence) of 0, 1 or more input values.

Output: A YES or a NO depending on whether a property is satisfied.

(The property depends on the input values.)

COMPUTATIONAL (OPTIMIZATION) PROBLEM

Input : A collection of 0, 1 or more input values.

Output: A collection of output values derived from the input values.

A poorly defined computational problem is given next.

Problem 1 (Addition of “numbers”).

Input. Two numbers

Output. The sum of these numbers.

Introduction

Examples of Computational Problems

A better formulation of Problem 1 follows. In such a formulation we do not need to worry how to deal with negative numbers, integer or real, nor with rational numbers, nor with complex numbers, etc. The following computation problem is an optimization problem. The one after it is a decision problem.

Problem 2 (Addition of bounded-size non-negative integers).

Input. Two n -bit non-negative integers a, b .

Output. An (at most) $(n + 1)$ -bit integer c such that $c = a + b$.

Problem 3 (Verification of addition of integers).

Input. Three non-negative integers a, b, c , where c is up to $(n + 1)$ -bit and a, b are n -bit.

Output. A YES or a NO depending on whether c IS-EQUAL-TO $a + b$.

Note that c is obviously non-negative as well! We might have used digit instead of bit, as in n -digit and $(n + 1)$ -digit. Integer c is the output value and a, b are the input values for the optimization problem. All three are input values for the decision problem. Note that n , the number of bit of a and b is also part of the input.

Definition 12 (Length vs Size). For a sequence its length is the number of elements of the sequence. For a sequence that contains elements of the same (data) type, the size of a sequence is its length multiplied by the size in bytes of the data type of one of its elements.

Example 3. An array of length 20 unsigned 32-bit integers has length 20. Its size is 80B since a 32-bit integer uses space equivalent to 4B. Note that length is dimensionless but size is expressed in B which is the unit name for a byte.

Note 9 (Problem size notation). Traditionally we represent problem size with a variable n . It is n to indicate that it is a natural number (a positive integer extending including also the zero value to become a non-negative integer). Performance functions will be denoted sometimes by $T(\cdot)$ and $S(\cdot)$. If n is the "problem size", they will become $T(n)$ and $S(n)$. But if problem size is m the use of $T(n)$ instead of $T(m)$ is wrong if not bizarre. And for graph algorithms where n is the number of vertices and m is the number of edges the use of $T(n)$ would be a misnomer for the correct $T(n, m)$.

DRAFT

Note 10 (Binary trees). In the context of a binary tree we can define a complete binary tree (CBT) to be one that is complete of nodes. Nodes can be internal nodes (non-leaves) or external nodes (leaves). The height of a node is the distance (in term of edges) from the node to a descendant leaf. The height of the tree is the height of the root. An external node has no children, an internal node IN A COMPLETE BINARY TREE has two children. Consider a complete binary tree of height h . We can label the levels top to bottom starting with 0. The number of nodes at level i is 2^i . At level h the number of nodes is 2^h and all leaves reside there. The number of internal nodes is one less $2^h - 1$ and the total number of nodes $2^{h+1} - 1$. If this is denoted by n we can solve for h and express thus h in terms of n . If $n + 1$ is a power of two $h = \lg(n + 1) - 1$ or use ceilings otherwise. Moreover a CBT with n nodes has $(n + 1)/2$ leaves and $(n - 1)/2$ internal nodes.

	Level	#nodes	Complete Binary Tree of height h
<pre> x root / \ x x /\ /\ x x x x </pre>	0	2^{**0}	internal
			#internal nodes = $2^{**0} + 2^{**1} + \dots + 2^{**h-1}$
	1	2^{**1}	internal
			= $2^{**h} - 1$
	2		#external nodes = 2^{**h}
...	i	2^{**i}	internal
<pre> x x x x /\ /\ /\ /\ x x x x x x x x </pre>	h-1	2^{**h-1}	internal
	h	2^{**h}	external also known as leaves

Definition 13 (Problem size). The problem size of a computational problem is the most important parameter identifiable directly or indirectly in the problem description of a computational problem.

Problem size several times is a parameter (variable) that would be used in determining the performance of an algorithm. It might be clearly stated and be visible in a well-defined computational problem or be not very visible in some poorly-defined computational problems. Performance measures (e.g. space, time, work) of algorithms that will later be established for this computational problem will be expressed as functions of this parameter. Going back to Definition 7, $\text{bit}(x) = |x|$ is the number of bit in the binary representation of non-negative integer x .

Some collateral definitions that sometime define problem size or are confused with problem size.

Definition 14 (Input size). It is the amount of space used by (a problem's) input.

Definition 15 (Input instance). An input instance of a problem is the sequence of values assigned to the inputs of the problem.

Definition 16 (Output size). It is the amount of space used by (a problem's) output.

Problem size and Input size

Examples

With reference to problem 2 defined earlier we note the following.

Remark 1 (to Problem 2). With reference to Problem 2, problem size is easily identifiable as n . Input size is $2n + |n| = 2n + \text{bit}(n)$. The input consists of integer a, b each one n -bit long ($\text{bit}(a) = \text{bit}(b) = n$) but also of n itself which is $\text{bit}(n)$ long. Output size is at most $n + 1$ bit.

Note that a, b, c would use arrays of length $n, n, n + 1$ respectively. We won't address the size of these three arrays! This explains the argument about dissipation discussed earlier.

Let us study the following variation of Problem 2.

Problem 4.

Input. Two non-negative integers a, b with $a > b$.

Output. An integer c such that $c = a * b$.

Though the computational problem has become multiplication, changing a times into a plus can turn it into a addition problem that is not as clearly defined as Problem 2. In this problem, problem size is not directly available. It must be indirectly derived. Given our experience with Problem 2 let us define as N the problem size. (We use N to vary the range of parameters used for problem size beyond the traditional n .) N would be the number of bit of the larger of the two integers. Thus $N = |a| = \text{bit}(a)$. In that context we would also use N bit for b packing it with leading (and harmless) zero bit to make it into N bit if it is not so. An interesting question is the number of bit of c . This would be $|a| + |b|$ but $2N$ is a close upper bound.

Introduction Algorithms

An algorithm realizes the input-output relationship of a computational problem.

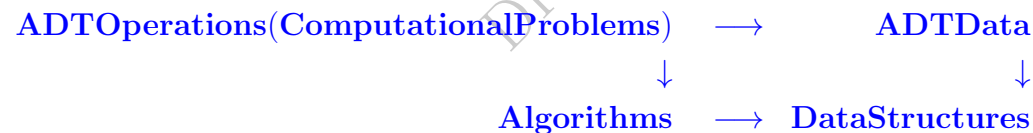
An algorithm is a sequence of computational steps.

Definition 17 (Informal algorithm definition). An algorithm is a “method” that transforms an input into an output, i.e. it realizes the input-output relationship of a computational problem.

We avoided intentionally the use of the word “recipe” !!

An ADT has data (model) and operations. In an ADT, operations operate abstractly on the data of the ADT. The ADT operations are defined as computational problems.

When the time comes to realize an ADT through its components and the abstraction is to become a bit more tangible, the data of an ADT are realized or organized through a data structure. Then the operations, the computational problems, must have their input-output relationship established through the design of an algorithm. are realized in the form of more tangible algorithms. Those algorithms operate on the data structure associated with the ADT data.



The realization of the input output relationship of those computational problems involves the design of an algorithm. The algorithm operates on the data of the data structure (ADT data).

Definition 18 (Algorithm). An **algorithm** for a computational problem is a well-defined sequence of computational steps that generates the output values of the computational problem from its input values.

Algorithm introduction

Continued

A computational problem is described through a specification that includes its input values and also its output values.

An algorithm for the computational problem establishes (realizes, uses) a well-defined sequence of computational steps that generates the (set of) output values from the (set of) input values of the computational problem.

“Well-defined” means correct. A set of values can be empty, can include only one value or more than one values!

An algorithm thus becomes a method for solving well-specified computational problems.

A (computational) problem defines an input-output relationship. An algorithm describes a specific procedure for achieving (establishing, realizing) that relationship.

For each computational problem we may have more than one algorithms.

Definition 19 (Correct algorithm). An algorithm for a problem is an “Algorithm”, i.e. a correct algorithm, if it works correctly for every input instance and halts (terminates) with the correct output(s).

At this moment we are not going to include in this definition algorithms that might give incorrect answers. We could include such algorithms in our definition under the assumption that the “problem description” allows for erroneous outputs.

An algorithm for Problem 2 is the elementary school algorithm. We learnt it for radix-10 (denary) integers. It works flawlessly for binary (radix-2) integers as well.

Remark 2 (Recipe vs Algorithm). How you cook or you brew coffee is a (cooking) recipe. An algorithm is associated with a computational problem.

Remark 3. Problem size is sometimes defined in terms of input size or input length or output size or output length.

Note 11 (Functions). In general 2^n is an exponential function (in n), n^3 is cubic function (in n), $n \lg n$ is a log-linear function, $\lg n$ is a logarithmic function, and 1 is a constant function in the sense that $f(n) = 1$ for all non-negative integer n . Variable n is non-negative in general or positive.

Note 12 (Factorial and Stirling's approximation). By default $0!=1$ and $1!=1$, with $n! = n \times (n-1)!$ From Document 2, Stirling's approximation for the factorial for $n > 10$ in its simplest form is

$$n! \approx (n/e)^n.$$

Moreover

$$n! \leq n^n, \quad (n/2)^{(n/2)} \leq n!$$

and thus

$$\lg(n!) \approx n \lg n.$$

To express later this $\approx$ relationship we shall write $\lg(n!) = \Theta(n \lg n)$. More in Document 2.

Note 13 - Note 15

Note 13 (Derivatives). Thus $(2^n)' \approx 2^n$ is a derivative with respect to n . Likewise $(\lg n)' \approx 1/n$. (Variable k is a positive constant.)

$$\begin{array}{ll} (e^n)' \approx e^n & (2^n)' \approx 2^n \\ (2)' = 0 & (n^k)' \approx n^{k-1} \\ (f \cdot g)' = f'g + fg' & (f/g)' = (f'g - fg')/g^2 \end{array}$$

Note 14 (Integrals). Note that $H_n = \sum_{i=1}^n \frac{1}{i} = 1 + 1/2 + \dots + 1/n \approx \ln n + \gamma$. The constant γ i.e. Euler's constant approximately $0.57\dots$. For more see Document 2.

$$\begin{array}{ll} \int n^k dn = \frac{n^{k+1}}{k+1} + c & \int (1/n) dn = \ln n + c \\ \sum_{i=1}^n 1/i = 1 + 1/2 + 1/3 + \dots + 1/n = & \int_{x=1}^{x=n} (1/x) dx = \ln n - \ln 1 + c = \ln n + \gamma \\ \gamma \approx 0.57 \text{ (Euler's constant)} & k > 1 \text{ constant} \end{array}$$

Note 15 (Limits). Be reminded limits of interest are for $n \rightarrow \infty$ i.e. for asymptotically large values of n . $\lim_{n \rightarrow \infty} (a/b)^n$ depends on whether $a < b$, $a = b$ or $a > b$. For more see Document 2, but the limit is 0, 1, ∞ respectively.

$$\begin{array}{ll} \lim_{n \rightarrow \infty} 2^n = \infty & \lim_{n \rightarrow \infty} n^k = \infty \\ \lim_{n \rightarrow \infty} \lg n = \infty & \end{array}$$

Definition 20 (Series). A series is the sum of the terms of a numeric sequence. The harmonic series $H(n)$ was defined earlier, the arithmetic series is

$$A(n) = \sum_{i=1}^{i=n} i = 1 + 2 + \dots + n = n(n+1)/2,$$

and the geometric series is ($x \neq 1, x > 0$)

$$G_{n,x} = \sum_{i=0}^{i=n} x^i = x^0 + x^1 + \dots + x^n = (x^{n+1} - 1)/(x - 1),$$

and the infinite geometric series G_x (infinite form $0 < x < 1$)

$$G_x = \sum_{i=0}^{i=\infty} x^i = 1/(1 - x).$$

We can define likewise for $0 < x < 1$ another series,

$$F_{n,x} = \sum_{i=0}^{i=n} i \cdot x^i = 0 \cdot x^0 + 1 \cdot x^1 + \dots + n \cdot x^n$$

and its infinite series counter-form

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

In Document 2 (also Subject 5) it is shown that $F_{1/2} = 2$.

F_x can be derived from G_x by taking the first derivative of G_x and multiplying by x .

Remark 4 (Propositions and Mathematical Induction). A Proposition P is a statement that is either true or false. We write $P(n)$ to indicate a proposition that depends on an integer variable n . ‘Integer’ means (from now on) non-negative or positive integer i.e. a natural number.

To Prove $P(n)$ for $\forall n \geq 0$ we can either prove it directly by showing $P(0), P(1), \dots, P(n), \dots$ (to be true), or by using a simpler method known as mathematical induction or simply induction that comes into two flavors: induction (also known as weak induction) and strong induction.

Definition 21 (Weak induction). (a) $P(1)$, and (b) $P(n-1) \Rightarrow P(n)$ imply $P(n) \quad \forall n \geq 1$.

To show $P(n)$, $\forall n \geq 1$ we need a firestarter (ignition) and a way to propagate a chain reaction. The ignition is known as the **base case** e.g. $P(1)$ meaning that we need to show that $P(1)$ is true. We need to show explicitly that $P(1)$ is true. The propagation (chain reaction) is provided by $P(n) \Rightarrow P(n+1)$ or $P(n-1) \Rightarrow P(n)$ an implication that is known as the **inductive step**. The left side of the implication is known as the **induction hypothesis** or **induction assumption**. Having proven the base case $P(1)$ and the implication $P(n-1) \Rightarrow P(n)$ we can show then all the $P(1), P(2), \dots$ are true by way of $P(1) \Rightarrow P(2) \Rightarrow P(3) \Rightarrow \dots \Rightarrow P(n)$. If $P(1)$ is problematic, we try to show $P(2)$ or $P(3)$ and then the chain reaction might start at $P(2)$ or $P(3)$ as needed, e.g. $P(3) \Rightarrow P(4) \Rightarrow P(5) \Rightarrow \dots \Rightarrow P(n)$.

Definition 22 (Strong Induction). (a) $P(1)$ and (b) $P(1) \wedge P(2) \wedge \dots \wedge P(n-1) \Rightarrow P(n)$ imply $P(n) \forall n \geq 1$.

In strong induction the induction hypothesis is more extensive that is, it is stronger!

Example 4 (Fibonacci Sequence). Let $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0, F_1 = 1$. We show that $P(n)$ is true where $P(n) : F_n \leq 2^n$.

Proof. Base case $P(1)$ (is true) is proven directly. $P(1)$ is true as long as $F_1 \leq 2^1 = 2$ which is so by definition. (Note that $P(0)$ is also true.) Using strong induction and the implication of $P(n-1)$ and $P(n-2)$ which imply $F_{n-1} \leq 2^{n-1}$ and $F_{n-2} \leq 2^{n-2}$ we have that

$$F_n = F_{n-1} + F_{n-2} \leq 2^{n-1} + 2^{n-2} \leq 2^{n-1} + 2^{n-1} = 2^n$$

■

DRAFT

Algorithms and Algorithm Analysis

Description and Pseudocode

We describe an algorithm in this course with *pseudocode* that looks like a procedural (aka imperative) language such as Pascal, C, C++, or Java but we are not concerned with its syntax, variable declarations or definitions and other minute details. We also use free-form English to describe more complex interactions. For example we may say 'Find the maximum of three elements' instead of describing how to find the maximum of three elements. If we have introduced an algorithm, for example QuickSort for sorting arrays or sequences of keys, we might invoke it in pseudocode without further details by issuing say a function call to QuickSort(B, m) to sort an array B of m keys with the implication being that at the completion of the function call all elements are sorted in B , even if originally B was not sorted.

We are liberal with the boundaries of an array if it fits our purposes (i.e it makes exposition of an algorithm simpler). Therefore we may declare an array of n elements more often as $A[1..n]$ but occasionally as $A[0..n-1]$. Why ? It causes confusion to say the fifth element of an array is $A[4]$ rather than the more natural and less confusing $A[5]$. We do not bother about error-checking. So if we are to design an algorithm for adding two non-negative integers, we do not bother checking for negative-numbers or real numbers or complex numbers or strings. We assume that the input is correctly formed.

Our intent is to introduce concepts around algorithms, and the ideas behind them. We do not want to spend too much time on implementation issues and divert ourselves from the important design and analysis issues. Normally, it would be easy for a competent programmer to turn the pseudocode into actual code.

We will intentionally leave a semicolon missing at the end of line 2 before or a line 3 of the code for Fibonacci (you find out in which of the three programs this is so). Furthermore, no variables would be properly defined. And Python-like indentation will be used more often than adding well-balanced braces in statements!

Algorithms and Algorithm Analysis

RAM model of computation

We analyze the computational performance of an algorithm by first introducing the computation model that will be used. Our model is a generic uniprocessor machine called **Random-Access Machine (RAM)**. In a RAM instructions are executed one after the other in unit time. Each memory cell is wide enough to accommodate a word (e.g. for Fibonacci numbers, it can store each one of the numbers however big it is in one word) and each word can be retrieved in unit time. Note that we use the generic term word to refer to keys (for sorting problems), integers (short int, int, long int, etc), floating point numbers (float, double), or arbitrary precision numbers or other data types. The computation model is as simplistic/optimistic (in resource usage) as possible. The intent is to show that if an algorithm uses A computation resources under the RAM it would also require at least A under more realistic models.

Other Models of Computation: WORD and BIT models. We have already discussed briefly (or introduced) these models. Later on when we discuss arithmetic problems we might want to differentiate between adding two integers and adding two relatively long integers. We would then talk about the **WORD** model where addition of integer a and integer b takes unit time i.e. one step, i.e. one operation (thus RAM is the Word model), and the **BIT** model where the addition of a and b would take time proportional to the number of bits of a and the number of bits of b .

Other Models of Computation: Straight Line Program. When we discuss the WORD and BIT model, we are going to define the Straight Line Program (SLP) model of computation with respect to the WORD model. In SLP we have no conditionals (if statements), no iterative statements (eg for/while loops), no function invocation, no arrays nor table lookups and no indirection (eg no pointers)!

For the integer addition problem, n is the number of bits for each one of the two input integers a and b . Input size is an atrocious two integers (WORD model) for the poorly defined version, or $2n$ bits (BIT model) plus all other complications mentioned earlier. The fact that we utilize the name n in the expression $2n$ to express input size is enough to convince us that the most important parameter of the problem is n , its problem size. For sorting things are easier. We have n input keys so problem size is indeed n . As far as input size is concerned this consists of n keys i.e. it is n keys as well. Note that for input size we need to say something about the unit of measurement.

Problem size is n but Input size is $2n$ bit for Problem 1 or $2n + \lceil \lg(n+1) \rceil$ bit, if you want to be more precise. Saying Input size is $2n$ is incorrect. The unit is missing. For Sorting, problem size is n , but Input size is n keys. Saying Input size is n without the unit of measurement would make it wrong.

Algorithms and Algorithm Analysis

Resources

Computational resources expended during the execution of an algorithm are in the form of cpu-cycles used (or wasted) and amount of memory utilized (or expended). Performance measures established to describe the performance of an algorithm can associate with them directly. In other words **TIME** and **SPACE** are of paramount importance. In more complex problems other performance measures such as throughput, latency, communication bandwidth, logic gates, efficiency or speedup might also be defined and used. With **TIME** we mean how long an algorithm runs to completion with **SPACE** how much memory/space the algorithm uses.

We then find problem size and express those performance measures as a function of problem size. How do we measure space and time? We do not measure space in bytes (or megabytes). Rather in words or in multiples of the size of the elementary data type used (eg key, number). And for time forget nanoseconds or femtoseconds. In the best of cases it would be "operations" in general (mixing up additions, assignments, read/write operations) or if we can be more discriminating, we would identify the most fundamental operation (eg comparison of keys or addition of numbers) and express time in terms of those fundamental operations! This means what we measure as time and space for example is not a precise measure. If we count number of keys instead of bytes, we need to know how many bytes we use to store a key to be precise. Likewise if the cost of an addition is one operation, to get nanoseconds we need to know clock-cycles and other properties of a CPU on which execution is to take place. This is impossible to do in a generic fashion. Thus for such an analysis we do not aim for precision. Just for good enough accuracy!

Thus multiplicative constants and low-order terms becomes less important. And when we compare algorithms make sure that operations that are compared are the same. For example if an algorithm uses $3n - 2$ multiplications and another one uses $4n$ additions, which of the two is faster?

Sorting is the computational problem. There are multiple algorithms that solve this Problem. NaiveSort (or ExhaustiveSort that describes more faithfully its behavior) is a non efficient proposal for a sorting algorithm. MergeSort and HeapSort are more efficient time-wise but the former is not efficient time-wise (for the implementation that we will propose). Other approaches include Odd-Even Transposition sort, BubbleSort, InsertionSort, SelectionSort, QuickSort. All such algorithm perform key comparisons and swaps (exchanges). There are some other sorting algorithms that do not compare key values but inspect key values. They are referred to as distribution-based sorting algorithms and they include CountSort, RadixSort. The term BucketSort refers to a solution similar to CountSort: BucketSort used LinkedLists but CountSort arrays. The term BucketSort might be used differently in some textbooks though to define a separate and different sorting problem. The 'See before you leap' is applicable: read the problem definition not the name of the (sorting) algorithm.

When we measure the time and space requirements of an algorithm we describe them in terms of functions $T(n)$ and $S(n)$ respectively, where n is problem size.

Algorithms and Algorithm Analysis

Deriving $T(n)$ and $S(n)$

When we have an algorithm for a computational problem and before we start analyzing its performance we need to do certain things. These include the following.

- (a) Determine the resource (e.g. time) whose usage will be studied (performance evaluation).
- (b) Isolate the operation that describes best the resource usage.
- (c) Determine the model of computation to use.
- (d) Find problem size and parametrize it, if needed.
- (e) Then determine the resource usage in the form of a function for the given algorithm (e.g. $T(n), S(n)$).

Example 5 (Sorting). (a) For example for the case of a sorting algorithm we might decide to study the time it takes for a sorting algorithm to complete its operation (sorting).

- (b) We determine then and isolate as the most important operation the key comparisons performed.
- (c) We then decide to use the Word Model of computation (one comparison would involve two keys and each key occupies one word).
- (d) We find problem size to be the number of keys n to be sorted.
- (e) And finally we derive an expression for the running time in terms of function $T(n)$.

With functions $T(n)$ and $S(n)$ details (low-order terms or multiplicative constants) are not important.

Some of those details are hidden in the units (eg keys vs bytes, operations vs nanoseconds). But how do we remember all the constants if we need to write a $S(n) = 2n + 3$ or $S(n) = 2n$? The easiest thing to remember is to say that " $S(n)$ is linear to n ". We would denote this as $S(n) = \Theta(n)$. Or if $S(n) \leq 3n$ to say that " $S(n)$ is upper bounded by a linear function to n " we would denote it by writing $S(n) = O(n)$. A $S(n) \geq n$ to say that " $S(n)$ is lower bounded by a linear function to n " would become $S(n) = \Omega(n)$. The symbols are known as Theta, Big-Oh and Big-Omega respectively. A $\Theta(1)$ means a constant!

Definition 23. A **sorted sequence** is an increasing sequence or non-decreasing sequence or monotonically-increasing sequence such as $\langle 1, 2, 3, 3, 4 \rangle$. A **reverse-sorted sequence** is a decreasing / non-increasing / monotonically-decreasing sequence such as $\langle 4, 3, 3, 2, 1 \rangle$.

The elements of a sequence are "comparable" that is, a "total order" is defined on them. On them antisymmetry is applicable that is, $a \leq b$ and $b \leq a$ implies $a = b$, transitivity is also applicable i.e. $a \leq b$ and $b \leq c$ implies $a \leq c$ and reflexivity $a \leq a$ and totality are also applicable i.e. $a \leq b$ or $b \leq a$.

Problem 5 (Sorting).

Input. A sequence of n comparable keys

$$\langle a_1, a_2, \dots, a_n \rangle.$$

Output. A reordering (a permutation) of the n keys $\langle a_{j_1}, a_{j_2}, \dots, a_{j_n} \rangle$ of the input sequence so that

$$a_{j_1} \leq a_{j_2} \leq \dots \leq a_{j_n}.$$

(The output is a sorted sequence.)

If the output is explicitly defined to be $a_{j_n} \geq a_{j_{n-1}} \geq \dots \geq a_{j_1}$, we call such an output a reverse-sorted sequence. Problem size is clearly n the number of input keys. An algorithm is a sorting algorithm if and only if it sorts correctly all $n!$ permutations of n keys for all permissible (possible) values of n . A permissible value is determined by the available computational resources (eg memory). Note that on n keys we have $n!$ possible permutations if the n keys are distinct.

Sorting Preliminaries

A naive approach

Example 6. For input $\langle a_1 = 10, a_2 = 2, a_3 = 1, a_4 = 5 \rangle$ an algorithm for sorting would generate an output such as $\langle a_3 = 1, a_2 = 2, a_4 = 5, a_1 = 10 \rangle$. The input keys are $\langle a_1, a_2, a_3, a_4 \rangle = \langle 10, 2, 1, 5 \rangle$, and the output keys are $\langle 1, 2, 5, 10 \rangle$. Sometimes sorting is defined in term of indexes (subscripts). In that case the input is $\langle 1, 2, 3, 4 \rangle$ since $\langle a_1 = 10, a_2 = 2, a_3 = 1, a_4 = 5 \rangle$ and the output is $\langle 3, 2, 4, 1 \rangle$ since $\langle a_3 = 1, a_2 = 2, a_4 = 5, a_1 = 10 \rangle$. We don't use the latter.

Method 1 (A solution for sorting: NaiveSort). A first solution to the problem of sorting is NaiveSort.

Solution. NaiveSort(A,B,n) // ExhaustiveSort
1. for(i=1 ; i <= n! ; i++) {
2. B = < generate next permutation of the n input keys of A>
3. If B is a sorted sequence return(B);
4. }

Proposition 1. *The time performance of NaiveSort, when the operation that best describes time is the number of permutations generated, is $T(n) \leq n!$.*

Proof. We observe the following if we count the number of permutations generated by NaiveSort in a solution. In such considerations we assume that all n keys are distinct.

Best Case. The answer is found after generating 1 permutation.

Worst Case. The answer is found after generating all $n!$ permutations.

Average Case. The number of permutations generated on the average is $(n! + 1)/2$. Why ? We might get the answer after generating 1 permutation, or 2 permutations, or ... i permutations, ... or $n!$ permutations. The average over all these possibilities is (The numerator is the arithmetic series $A(n!)$).

$$\frac{1 + 2 + \dots + n!}{n!} = \frac{n!(n! + 1)/2}{n!} = \frac{n! + 1}{2}$$

Algorithms

More on Sorting and NaiveSort

Proposition 2. *The time performance of NaiveSort when the operation that best describes time is the number of comparisons performed is $T(n) \leq n! \cdot (n - 1)$.*

Proof. Line number 3 if it is further explained, and this is probably a limitation of our pseudocode approach reveals that we can verify that B is a sorted sequence by performing at most $n - 1$ comparisons: the i -th comparison is between $B[i]$ and $B[i + 1]$.

$$B[1] \leq B[2] \leq B[3] \leq \dots \leq B[n - 1] \leq B[n]$$

We can discard a B and move to the generation of the next permutation after performing as few as one comparison or as many as $n - 1$ comparisons. (If $B[1] > B[2]$ we know that B is not a sorted sequence.) Moreover let it be noted that we use a Pascal indexing for arrays: it is convenient in this context.

Thus we have the following observations similarly to the one established earlier in the context of counting permutations for the running time.

Best Case. One permutation generated and it is sorted; $n - 1$ comparisons performed to verify B is sorted.

Worst Case. All $n!$ permutations generated. It is quite a leap of faith to say that in each permutation we need to perform $n - 1$ comparisons to realize that it is not sorted. It might or might not be the case. Try it with 4 or 5 keys! But in the worst scenario we have to deal with $n! \times (n - 1)$ comparisons.

Conclusion. The number of comparisons determines the running time $T(n)$, with n the problem size. We might use $C(n)$ for number of comparisons to write $C(n) \leq n! \times (n - 1)$. Yet later on we shall write

$$T(n) = O(n! \times n) = O(n! \cdot n)$$

The latter expression for $T(n)$ is an asymptotic expression for the running time: -1 is thrown out of the upper bound for $C(n)$ that involves a $n - 1$. Whether we use $n - 1$ or n does not make a difference; both n and $n - 1$ are linear functions. Low-order terms such as -1 do not change that nor a multiplicative constant if there was to be one such as $2n - 1$.

Finally, a question. **Is $i \leq n$ of line 1 of the code a comparison?**

The answer is NO. The term comparison as defined is a key comparison. Variables i and n are not keys but integers. In reality a compiler will resolve $i \leq n$ with a subtraction. The accumulator of the CPU will hold the result $t = i - n$ and the Processor Status Word (EFLAGS or RFLAGS for Intel CPUs) will flag the result t as positive, negative or zero. ■

Did you spot the missing semicolon from NaiveSort? The “code” of Page 15 for a solution to the Sorting problem has a semicolon missing. It is not code, it is pseudocode.

What is the time to completion for NaiveSort? How do we measure time of pseudocode? Time analysis even of actual code is tedious. We identify the most important operation and express time in terms of or in connection with it. For a sorting problem such as NaiveSort the most important operation was identified earlier to be a (generated) permutation. However this ignores the cost of execution of line 3. We can refine further and claim that the most important operation is comparison of keys.

DRAFT

Algorithms

One Problem, many algorithms

Sorting is the problem. There are multiple algorithms that solve this problem as highlighted earlier. Several of those approaches satisfy different requirements and have different performance. Thus it makes sense to compare the performance of two algorithms for the same computational problem. In fact we need to make sure that such comparison is an 'apples to apples' or 'oranges to oranges' comparison and not an 'apples to oranges' comparison.

Consider two algorithms one that requires n^2 operations and one that requires $n \lg n$ operations to sort n keys. Consider a machine that executes 10,000,000 such operations per second with a millisecond precision clock. The two algorithms solve instances of various sizes n in the following time.

	n	$= 2$	10^3	10^6	10^9
Alg A: n^2	$=$	$<0.001\text{sec}$	0.1sec	1day	300years
Alg B: $n \lg n$	$=$	$<0.001\text{sec}$	0.001sec	2sec	1hour

By comparing n^2 and $n \lg n$ we draw the conclusion that for small values the performance of the two algorithms is indistinguishable. For large values it is not. In our example comparison was easy; there were no low-order terms nor multiplicative constant involved. Consider for example the case where one algorithm requires n^2 operations and the other $20n \lg n + 1000n + 10^9$. Things become more complicated calculator-wise. To establish asymptotic performance and comparison we only need to know what happens as $n \rightarrow \infty$. For this we just need to $\lim_{n \rightarrow \infty} n^2 / n \lg n$. It is not difficult to find that the limit is infinity and thus the n^2 algorithm is too costly operation-wise (slower) than the $n \lg n$ algorithm. And the same applied to n^2 and $20n \lg n + 1000n + 10^9$.

From now on when we compare the performance of two algorithms we would be comparing the asymptotic performance. For problem size n this means a comparison for $n \rightarrow \infty$. What happens for $n = 2$ or $n = 10$ or $n = 10000000$ might be irrelevant.

Unsigned integer addition analyzed: BIT model

We reexamine Problem 2.

	Low/Little Endian	Big Endian
Input	$a=a[n] \dots a[1]$ $b=b[n] \dots b[1]$	$a=a[1] \dots a[n]$ $b=b[1] \dots b[n]$
Output	$c = a+b$ where $c=c[n+1] \ c[n] \dots c[1]$ or $c=c[1]c[2] \dots c[n+1]$	

Definition 24 (Big Endian vs Little Endian). If we use an array $a[1..n]$ to store the n -bit integer a (and likewise for b), the question that arises is do we store in $a[1]$ the leftmost bit (i.e. most significant bit) or the rightmost bit (i.e. least significant bit). In the former case we use something akin to a Big Endian representation, in the latter a Little Endian representation. The former is utilized by PowerPC (or ARM) architectures, the latter by Intel for a different purpose: to store for example the bytes of a multi-byte integer in memory locations (low index becomes low address and stores the least significant byte in LittleEndian). From a programming point of view using Low Endian the second bit from the right is a_1 or b_1 or c_1 . In Big Endian it is a_n , b_n or c_{n+1} . By the way $a = a_n \dots a_1$ and a_i is stored in $a[i]$ in the pseudocode! Likewise for b .

Proposition 3 (Unsigned Integer addition). *Integer addition of unsigned n -bit binary numbers requires $T(n) = \Theta(n)$ and $S(n) = \Theta(n)$. $\Theta(n)$ is a way to say “linear to n ”. Thus $T(n)$ is linear to n and so is $S(n)$.*

The proof is on the following page. It is based on algorithm ADD.

Proof.

```

    Add (a,b,c,n)    // Little Endian Elementary School solution
1.  R = 0 ;          // carry bit
2.  for (i=1 ; i <= n ; i++)
3.      temp = a[i] + b[i] + R;    // bit + is most IMPORTANT OPERATION
4.      c[i] = temp % 2 ;          // c[i] = lsb(temp)
5.      R    = temp / 2 ;          // c[i] = (temp > 1)? 1 : 0)
6.  c[n+1] = R;

```

We first identify the most important operation. It is in line 3. It is the **bit addition**. The mod and divide ops in line 4 and 5 are not division. It is there to simplify the code. If one checks the rightmost bit of `temp` one knows whether c_i is 0 or 1. (See comment in code.) Line 3 performs at most 2 bit additions. In the first iteration R is 0 and thus we only have 1 bit addition not involving R . The number of iterations is n thus we have at most $2n - 1$ bit additions.

The input size is $2n$ bit (forget about n itself) to account for a and b . Output size is $n + 1$ bit. R is 1 bit, and `temp` is 1 or 2 bit as it can be 0,1,2,3 in denary. Thus total space requirements are rough $3n + 4$.

Both $2n - 1$ (number of bit additions) and $3n + 4$ are linear functions. We use $\Theta(n)$ to describe either and both of them. In fact one interpretation of $\Theta(n)$ is the set of linear functions. Thus $2n - 1 \in \Theta(n)$ and likewise $3n + 4 \in \Theta(n)$. Some textbooks prefer to use Thus $2n - 1 = \Theta(n)$ and likewise $3n + 4 = \Theta(n)$. Moreover it is convenient to use such an asymptotic notation for one more reason: we ignored all other operations but we know that are proportional to the number of bit additions!

Running time $T(n)$ for Add.

$$T(n) = \Theta(n)$$

Space requirements $S(n)$ for Add.

$$S(n) = \Theta(n)$$

■

Prerequisites

Linear Search (CS505 or CS114 or CS116 material)

We provide the definition for linear search for completeness. Arrays now start from 0.

Problem 6 (Linear Search).

Input. Array/Sequence $A[0..n-1]$ of n comparable keys; key k .

Output. Print index i , where $0 \leq i \leq n-1$ such that $A[i] == k$ for $k \in A$, or print -1 to indicate that key $k \notin A$.

Solution.

```
LinearSearch(A[0..n-1],n,k) // cmp(x,y) is x:y and returns
1. for(i=0 ; i < n ; i++)    // if x<y return(-1) or negative
2.   if cmp(A[i],k) == 0     // if x==y return(0)
3.     return(i);           // if x>y return(+1) or positive
4. return(-1);              //
```

Proposition 4. *Linear search on keys has $T(n) = O(n)$.*

Proof. The most important operation is key comparison utilizing a fictitious function **cmp**. If $k \notin A$ we need to perform n comparisons and we exit through line 4. If $k \in A$ in the best case we perform 1 comparison and in the worst case n comparisons and exit through line 3. The average number of comparisons when $k \in A$, is ala NaiveSort. One case uses 1, another uses 2, etc another use n comparisons. The average of those comparison counts is $(n+1)/2$ not the misquoted $n/2$.

$$\frac{1 + 2 + \dots + n}{n} = \frac{n+1}{2}$$

Therefore, in linear search the number of comparisons is not always n a linear function as it was a linear function the number of bit operations in Add. It can be n but it can also be 1 or 2 or something in between 1 and n . We can't thus use $\Theta(n)$ that was an alias for linear-function to n . We must use something that is equivalent to "up to or no more than linear function to n ". This is represented by $O(n)$ that reads "big-Oh of n ".

$$T(n) = O(n).$$

and likewise for the comparison count $C(n)$.

$$C(n) \leq n \text{ or } C(n) = O(n).$$

Prerequisites

Linear Search (comparisons)

Question 3. Is $i < n$ a comparison (i.e. a key comparison)?

Answer. An $i < n$ is NOT a key comparison because i and n are unsigned integers and not (comparable) keys.

A compile for a programming language would resolve the $i < n$ with arithmetic operations

Is $i < n$?

```
1. temp = i-n;
2. return( ( sign(temp) - sign(-temp) ) / 2 ); //(temp>0)-(temp<0)
   // sign is +1 , 0 , -1 for positive, zero or negative argument
   // sign(temp)-sign(-temp) >> 1 or (temp<0)?-1:(temp>0))
```

In modern CPUs the value of temp is maintained by the accumulator; the accumulator's positive, or negative or zero value is reflected in the flag bit values of a register FLAGS (Intel 64-bit CPUs have a register RFLAGS). ■

Prerequisites

Asymptotic comparison of functions (CS506 or CS241).

We will revisit this section next chapter and discuss it with more mathematical rigor. For now, we remind ourselves the prior discussion related to the comparison of two functions.

In order to asymptotically compare $f(n)$ and $g(n)$ and establish a result that utilizes the commonly used symbols o, ω, Θ we first find the limit $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ and then determine which case is applicable. Object c is a constant. An implicit assumption is that all functions are positive or non-negative for large values of n a rigor that will be detailed in a following chapter.

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

1. $f(n) = o(g(n))$ implies $f(n) = O(g(n))$.
2. $f(n) = \omega(g(n))$ implies $f(n) = \Omega(g(n))$.
3. $f(n) = \Theta(g(n))$ is equivalent to $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Some books use instead $f(n) \in o(g(n))$, $f(n) \in \omega(g(n))$, and $f(n) \in \Theta(g(n))$ respectively.

Example 7. $5n = \Theta(n)$ but also $n = \Theta(5n)$. Yet $5n \neq n$ (exclamation mark follows)!

Reading greek characters. The symbol o reads little-oh or littleoh. The symbol ω reads little-omega or littleomega. The symbol Θ reads just theta. It is by default a big-theta or bigtheta. The symbol O is big-oh or bigoh. And the Ω reads big-omega or bigomega.

Prerequisites

Comparison vs Asymptotic comparison of functions

Duality: Comparison vs Asymptotic comparison. We observe the following parallels between the original comparison symbols and the introduced asymptotic ones.

Comparison Symbol $f(n) :?g(n)$	Asymptotic Comparison Symbol $f(n) =?(g(n))$
$<$	o
$>$	ω
$=$	Θ
$\leq$	O
$\geq$	Ω

DRAFT

Some examples

Example 8. Show that $n^{1/\lg n} = \Theta(2)$.

Proof. Θ is applicable and by equivalence also O and Ω . Thus $n^{1/\lg n} = \Theta(2)$, $n^{1/\lg n} = O(2)$, $n^{1/\lg n} = \Omega(2)$. ■

Example 9. Show that $2n = \Theta(3n)$.

Proof. Easy. If it is Θ it is also O and Ω . ■

Example 10. Show that $2^n = o(3^n)$.

Proof. Consider the limit of $(2/3)^n$ given $2/3 < 1$. If it is o it is also O . ■

Example 11. Show that $\lg(n!) = \Theta(n \lg n)$.

Proof. Use the discussion of Stirling's approximation for the factorial. If it is Θ it is also O and Ω . ■

Example 12. Show that $(n!) = \omega(n \lg n)$.

Proof. More Stirling related. If it is ω it is also Ω . ■

Polynomial time algorithms

Definitions

Definition 25 (Polynomial time algorithms). An algorithm is a polynomial-time algorithm if its running time $T(n)$ is a polynomial function of input size, that is $T(n) = O(n^k)$, where n is problem size and $k > 0$ is a positive integer constant. (We sometime refer to the algorithm as being poly-time in short.)

The key word in the definition is that n is not just problem size. It is input size! We have just proven that n -bit addition requires $T(n) = \Theta(n)$ time (i.e. bit operations).

Corollary 1. Unsigned integer addition (Problem 2) accepts a polynomial time algorithm.

Proof. We have shown that algorithm ADD has running time $T(n) = \Theta(n)$ where n is input size (number of bit of a or b). In fact input size is $n + n + \text{bit}(n) = \Theta(n)$ but constants are irrelevant. The next step is to show that $T(n) = O(n^1)$, but this straightforward from $\Theta \Rightarrow O, \Omega$. ■

Corollary 2. Unsigned integer multiplication (Problem 4) accepts a polynomial time algorithm.

Proof. For multiplication, problem size is N , but input size was determined in terms of $\text{bit}(a) = |a|$ as noted earlier.

Elementary school integer multiplication provides an algorithm for Problem 4 that in the bit model requires $O(N * N)$ bit operations. Given that $N = \text{bit}(a) > \text{bit}(b)$ we have that $N = O(\lg a)$. Thus

$$T(a) = O(\lg a * \lg a) \Rightarrow T(N) = O(N * N)$$

■

Problem size is N and running time is poly-time since $T(N) = O(N^2)$. Parameter a is not problem size. It is the value of one of the two integers!

Example 13. Let $a = 251$ and $b = 192$. The two integers in binary (0b is the binary notation prefix) are $a = 0b11111011$ and $b = 0b11000000$. $\text{bit}(a) = \text{bit}(b) = 8$ and thus $N = 8$. Note that $N * N = 64$ is proportional to the number of bit additions and the number of bit multiplications to compute $a * b$. The product $a * b = 0b1011110001000000$ and has $2 * N = 16$ bit.

There are various approaches one can use to design algorithms for various computational problems. We call those approaches, **methods** or **techniques**. Less often we might refer to them as principles. They include the following techniques

1. Exhaustive method.

With this method we search all possible solutions that is we perform an exhaustive search of the solution space. Sorting algorithm NaiveSort is such a case!

2. Incremental method.

The method is also known as the iterative or iteration method. Sometimes the output(s) of a problem can be computed iteratively using an iteration approach – and this can be realized with the use of a for or a while loop – that generates the output incrementally one-element at a time or by walking through the input and generating a solution one element of the input at a time incrementally. This lends the name to this method. It is known as the incremental method but several times it is referred to as the iteration or the iterative method. The iterative method name is more often used when the output is generated by (or in) the last iteration.

The elementary school algorithm for addition of n -bit unsigned integers is an example of the incremental method. The bit of c are generated one at a time starting from the rightmost bit and moving towards the leftmost bit of the result c .

3. Divide and Conquer method.

In Divide-and-Conquer, in order to solve a problem we first DIVIDE the problem into two or more sub-problems, conquer (i.e. solve) those sub-problems, and then by combining the solutions of those subproblems we get the solution of the original problem.

4. Dynamic Programming method.

The dynamic programming method is similar to but more complicated than Divide and Conquer but keeps track of the solutions (and cost of the solutions) of the subproblems generated into an array (matrix) and uses these solutions to generate the solution of the original (base) problem.

5. Greedy method.

What it look best is chosen. Different alternatives or solutions have assigned costs or benefits and smallest cost or largest benefit solution or subsolution is chosen at any time.

Algorithms

Fibonacci: Incremental Method

Problem 7 (Fibonacci).

Input. An integer $n \geq 0$.

Output. Integer F_n , the n -th Fibonacci number defined as follows: $F_n = F_{n-1} + F_{n-2}$ if $n > 1$ where $F_0 = 0$ and $F_1 = 1$.

Proposition 5 (Fibonacci incremental). *If the incremental method is used for computing the n -th order Fibonacci number then there exists a solution that requires $T(n) = \Theta(n)$. Moreover, that solution requires $S(n) = \Theta(n)$.*

Proof.

F (n)	//Incremental	#time run	time it takesd	Total
1. F[0]=0; F[1]=1;		1	c1	c1
2. for(i=2;i<=n;i++) {		n	c2	c2*n
3. F[i]=F[i-1]+F[i-2]		n-1	c3	c3(n-1)
4. }		n-1	c4	c4(n-1)
5. return(F[n])		1	c5	c5

The Fibonacci sequence definition lends itself to an incremental method solution. A simple for loop computes in iteration i the i -th Fibonacci number for $i \geq 2$. The 0-th and 1-st numbers are precomputed (line 1). This naive solution computes not just the n -th Fibonacci number but also all previous Fibonacci numbers and stores them into array F . It looks as if it is wasting resources. Amount of space used is $n + 1$ elements for array F (storing the 0-th, 1-st through n -th Fibonacci numbers) plus space for variables n and index i . We can say $n + 3$ or linear to n or proportional to n but $S(n)$ is described succinctly as $S(n) = \Theta(n)$ using the prior page discussion! Parameter n is problem size.

We utilize the WORD model (RAM). Finding the actual time of the code is next to impossible. Trying to determine is also difficult. Let for example Line i takes c_i time for one execution for some constant c_i . Lines 1 and 5 are executed once; line 3 $n - 1$ times and so is the right brace of line 4. Line 2 is executed once more than line 3; for $n - 1$ times the condition $i \leq n$ evaluates to true and the last time the n -th to false and the for loop terminates. Adding up all those contributions we get for $T(n)$ the running time for finding the n -th Fibonacci number but also all the previous ones.

$$T(n) = 1 \cdot c_1 + n \cdot c_2 + (n - 1) \cdot c_3 + (n - 1) \cdot c_4 + 1 \cdot c_5 \Rightarrow T(n) = \Theta(n).$$

■

The time bound is obviously $\Theta(n)$. There is no best-case for computing F_n . The for loop is computed completely and never stops ahead of the n -th iteration.

The most important operation in the calculation of F_n is the addition of line 3.

The most important task or tasks in analyzing the asymptotic performance of an algorithm are the following

- understand that we are to do an asymptotic performance analysis,
- pick wisely the computational model,
- determine problem size correctly, and
- perform a carefully asymptotic analysis of a performance metric (space, time or otherwise).

Obviously the number of additions in line 3 determines the running time; all other operations are constant per addition (eg assignment of $F[i]$ or indexing $i - 1$ or indexing $i - 2$ or incrementing i by one in $i++$). Moreover the initialization in Line 1 is constant $\Theta(1)$ time and so is Line 5. Therefore.

$$T(n) = \Theta(n).$$

Question 4. How would you rewrite the algorithm so that $S(n) = \Theta(1)$. Is it possible ?

Fibonacci

Divide and Conquer Method: overview

Next we show how to solve the same problem with Divide and Conquer. A natural way to describe a Divide and Conquer method is through recursion.

Definition 26 (Recursive function). A **recursive function** is a function that invokes itself.

Definition 27 (Direct and Indirect recursion). In direct recursion a recursive function f invokes directly itself, whereas in indirect recursion, function f invokes another function g that subsequently invokes f .

Using Divide-and-Conquer, in order to compute $F(n)$, the n -th Fibonacci number) we DIVIDE the problem of its computation, into two sub-problems, those of computing the $(n - 1)$ -st and $(n - 2)$ -nd Fibonacci numbers, CONQUER those problems i.e. solve them recursively this computing $F(n - 1)$ and then $F(n - 2)$ and at the end combine those two solutions to get the solution for $F(n)$. Combining in this context means addition since $F(n) = F(n - 1) + F(n - 2)$. In 3 lines a Divide and Conquer approach yields an algorithm for computing F_n through a call to $F(n)$.

Fibonacci Divide and Conquer Method

Proposition 6 (Fibonacci divide and conquer). *If the divide and conquer approach method is used for computing the n -th order Fibonacci number then there exists a solution that requires $T(n) = \omega(n)$. Moreover, that solution requires $S(n) = \omega(n)$.*

Proof. The overview of the divide and conquer solution outlined earlier is shown next.

```
F(n) // Divide + Conquer
1. if (n==0) return (0);      // F[0]
2. if (n==1) return (1);      // F[1]
3. return(F(n-1) + F(n-2) );  // F[n]
```

In general, the recursive divide-and-conquer solution is easier to design and implement. It is, however, less efficient as recursive calls are costly in terms of space and time utilization: each call requires space on the program's stack. Furthermore its analysis requires the solution of a recursive function also known as a recurrence or recurrence relation.

We keep things simple. 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 7. *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 $n > 1$.*

Proof. One could rewrite this recurrence $T(n) = T(n-1) + T(n-2) + 1$ by adding one to both sides and then using $G(n) = T(n) + 1$.

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 && \Rightarrow \\ T(n) + 1 &= T(n-1) + 1 + T(n-2) + 1 && \Rightarrow \\ G(n) &= G(n-1) + G(n-2). \end{aligned}$$

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

$$F(n) = F(n-1) + F(n-2), \quad F(2) = 1, \quad F(3) = 2, \quad \text{for } n > 3.$$

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 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. ■

Lemma 8. *The solution of $T(n) = T(n-1) + T(n-2) + 1$ with $T(0) = T(1) = 0$ is $T(n) = O(2^n)$ and $T(n) = \Omega(2^{n/2})$.*

Proof. We can prove the upper bound by induction. This was done in Example 4. The upper bound establishes the following.

$$T(n) = O(2^n).$$

Likewise we prove a corresponding lower bound as follows. It uses the monotonicity of $T(n)$ i.e. $T(n) \geq T(n-1) \geq T(n-2) \geq \dots$ beyond the first two fibonacci numbers.

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

For $n - 2i \approx 2$ we have $i = n/2 - 1$ and thus $T(n) \geq 2^{n/2-1}T(2)$. Since $T(2) = 1$ (see bottom 1.17.3.2) we have $T(n) \geq 2^{n/2-1}$ i.e. an exponential number of additions is needed. Therefore

$$T(n) = \Omega(2^{n/2})$$

This method of solving recurrences is known as the iteration method. And we used it to solve the recurrence of the running time of an algorithm that was designed not with the algorithmic design technique known as the iteration method but known as the divide and conquer method. One is a algorithm design method the other is a recurrence relation solution method! This method of solving recurrences is known as the iteration method. And we used it to solve the recurrence of the running time of an algorithm that was designed not with the algorithmic design technique known as the iteration method but known as the divide and conquer method. One is a algorithm design method the other is a recurrence relation solution method!

Go to the next page for a pictorial view of this method: the corresponding solution is still known as the iteration method or recursion tree method. ■

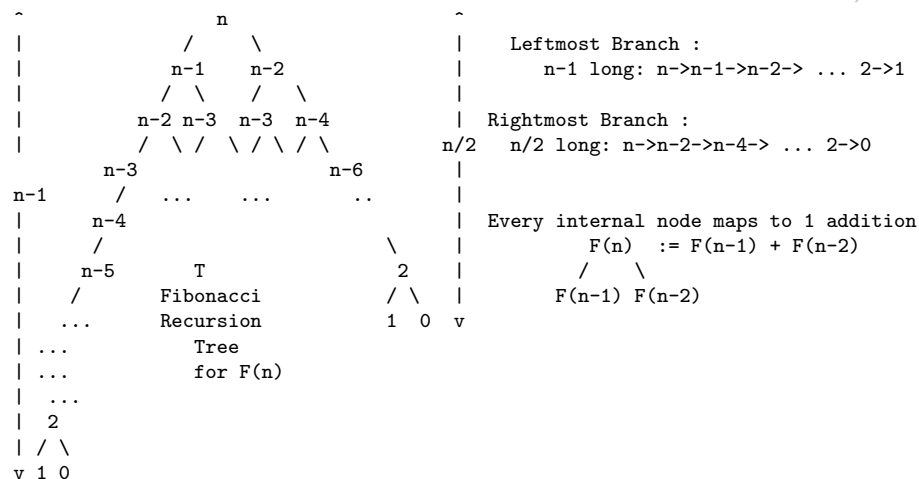
Fibonacci time derivation: Recursion tree style

Lemma 9. The solution of $T(n) = T(n-1) + T(n-2) + 1$ with $T(0) = T(1) = 0$ is

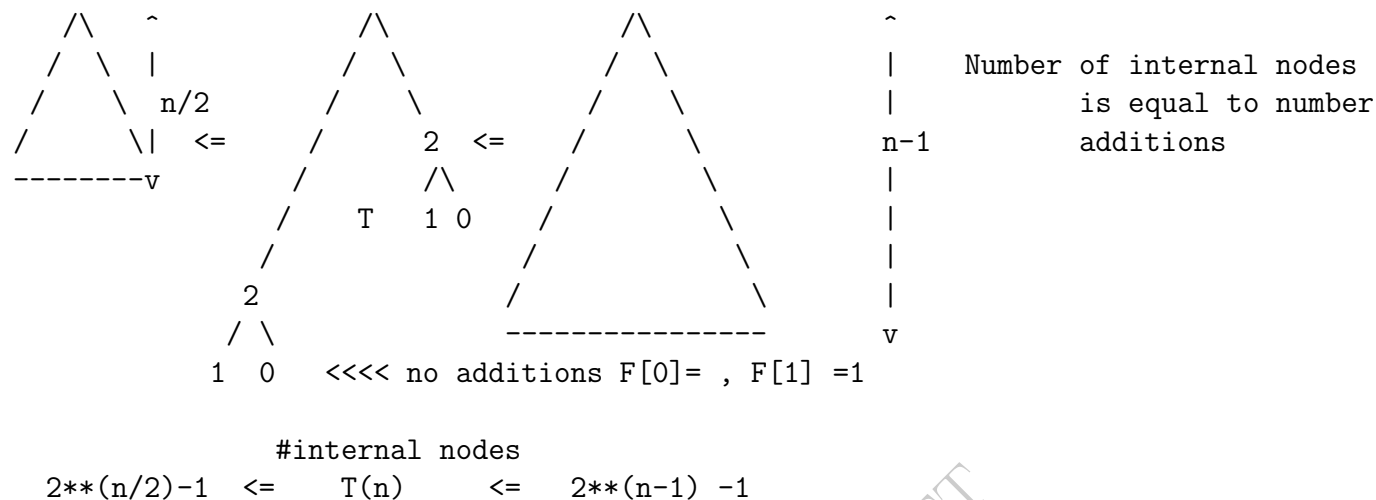
$$2^{\lceil n/2 \rceil} - 1 \leq T(n) \leq 2^{n-1} - 1.$$

Proof. We visualize the computations involved in the recursive structure of code $F(n)$ in the form of a binary tree. The label at a node is the value n in the call to $F(\cdot)$. Thus the root node is labeled n since the top level call is for $F(n)$. Inside the code of $F(n)$ we have two recursive calls to $F(n-1)$ and $F(n-2)$. The left and right children of the node labeled n are labeled $n-1$ and $n-2$ respectively. At the conclusion of the recursive call an addition of $F(n-1)$ and $F(n-2)$ will be performed. This is charged as one addition to the node labeled n . The number of internal nodes of the recursion tree with root n is the number of additions $T(n)$ to compute f_n the n -th Fibonacci number using call $F(n)$. The lop-sided Fibonacci recursion tree is upper bounded by a complete binary tree whose height is $n-1$ (the length of the leftmost branch of the recursion tree), and lower bounded by a complete binary tree of height $\lceil n/2 \rceil$. The number of internal nodes for the two boundary complete binary trees is $2^{n-1} - 1$ and $2^{\lceil n/2 \rceil} - 1$ respectively. Thus

$$2^{\lceil n/2 \rceil} - 1 \leq T(n) \leq 2^{n-1} - 1.$$



Fibonacci time derivation: Recursion tree style



If one thing is obvious about the recursive divide and conquer approach is the waste of computational resources. The running time of the divide and conquer method is exponential to n whereas the running time of the incremental method is linear!

Function n is much smaller than $2^{n/2} - 1$. Compute both functions for $n = 10, 100, 1000, \dots$ or if just take the limit $\lim_{n \rightarrow \infty} n/(2^{n/2} - 1) = 0$. The latter is what we called before an asymptotic comparison of two functions that shows that n is asymptotically smaller (i.e. running time is better) than $2^{n/2} - 1$ as well.

Fibonacci: Dynamic programming method

Recursion is a duplicating process. The number of times the recursive function is invoked with $F(1)$ or $F(2)$ is a Fibonacci number depending on n (check whether it is $F(n-1)$ or $F(n-2)$ or $\dots$). We can eliminate duplication if we keep track of the Fibonacci numbers whose values have already been computed thus computing each value ONCE and only ONCE. Subsequently we do not recompute but retrieve a previously computed value or skip the computation altogether to avoid overhead costs. For this we need to store previously computed values in a storage area (array in memory). The term **memoization** refers to the additional storage required to making it work! Thus we get this other algorithmic design technique known as **dynamic programming**. Think dynamic programming as divide and conquer without duplication of work.

Proposition 10 (Fibonacci memoization). *If the dynamic programming method is used for computing the n -th order Fibonacci number then there exists a solution that requires $T(n) = \Theta(n)$. Moreover, that solution requires $S(n) = \Theta(n)$. The method is also known as (or utilizes) memoization.*

Proof. The fundamental difference compared to divide and conquer is that we only recur say of $F(n-1)$ if the result related to $F(n-1)$ is NOT already available in table entry $T[n-1]$ i.e. the latter is UNDEFINED. Thus the computation call to $F(n-1)$ is done only once. Moreover, if you trace the calls to the recursive invocations, these involve only the nodes of the leftmost path of the recursion tree. Everything but this left-most path is pruned out and ignored in this computation.

```

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) return (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);

```

■

Algorithms

Divide and Conquer: Binary Search

Problem 8 (Binary search).

Input. Sorted sequence $A[0..n-1]$ of n comparable keys; key k .

Output. Print i , where $0 \leq i \leq n-1$, such that $A[i] == k$ for $k \in A$, or print -1 to indicate that key $k \notin A$.

Solution. The difference with LinearSearch is that the input sequence (array) A is sorted. With one comparison we can eliminate for further consideration not just one element but half of the remaining elements of the array. Thus comparing k to $A[m]$ either we have found the key at index m and it is $k == A[m]$ or half of the array $A[l..r]$ is eliminated and we recur to the half that is not eliminated.

<pre> BinarySearch(A[0..n-1],n,k) 1. BiS(A,k,0,n-1) // A[0..n-1] 0 1 m r n-1 ----- A[m] A ----- k:A[m] / \ < = > Line-7 Line-4 Line-9 </pre>	<pre> BiS(A,k,l,r) // A[l..r] 1. if (l>r) return(-1); 2. else{ 3. m= l+ (r-l)/2; //m=(l+r)/2 4. c= cmp(k,A[m]); 5. if (c==0) return(m); 6. elseif (c<0) 7. BiS(A,k,l,m-1); 8. else 9. BiS(A,k,m+1,r); } </pre>
---	--

■

Let $C(n)$ be the number of comparisons in BinarySearch for a sorted array A of n keys. This is also equal to the number of comparisons performed by BiS in line 1 and the only line of BinarySearch. In this call of BiS the bounds l and r are $l = 0$ and $r = n-1$. Entering BiS with $l = 0$ and $r = n-1$ we have an $m \approx n/2$. We perform first 1 comparison (Line 4) and then either recur on the left (line 7) or the right half (line 9) of $A[l..r]$ requiring recursively $C(n/2)$ comparisons since for $l = 0$ and $r = n-1$ the left half and the right half contain approximately $n/2$ keys. Thus $C(n) = 1 + C(n/2)$. Assuming n is a power of two and thus $n, n/2, n/4$ are all integer

$$C(n) = 1 + C(n/2) = 1 + (1 + C(n/4)) = 1 + 1 + (1 + C(n/8)) = 3C(n/2^3) = i \cdot C(n/2^i).$$

This is $C(n) = \lg n \cdot C(1)$. An array of size 1 requires one comparison and the recursive call following it exists through line 1 of BiS. Thus $C(1) = 1$. If $C(n) \approx \lg n$ then $T(n) = O(\lg n)$. We use big-oh because Binary search might end after one comparison in the best case through line 5 of BiS. This analysis so far is a bit sloppy. We perform a worst-case analysis next and for that analysis we choose a worst-case input size (length). This is for $n = 2^t - 1$ and the worst case.

Proposition 11. *For a sorted array of length $n = 2^t - 1$ the worst case number of comparisons performed by BinarySearch is $t = \lg(n + 1)$.*

Proof. In the worst case the recursion stops at the bottom of the recursion tree. Let $n = 2^t - 1$ for some integer t . Then $n + 1 = 2^t$ i.e. $t = \lg(n + 1)$. We revise the recursion for binary search from $C(n) = 1 + C(n/2)$ into $C(n) = 1 + C((n - 1)/2)$. Equivalently $C(2^t - 1) = 1 + C(2^{t-1} - 1)$ with $C(1) = 1$ as before. We substitute $t, t - 1, t - 2$, etc for t above.

$$\begin{aligned} C(2^t - 1) &= C(2^{t-1} - 1) + 1 \\ C(2^{t-1} - 1) &= C(2^{t-2} - 1) + 1 \\ &\dots = \dots \\ C(2^2 - 1) &= C(2^1 - 1) + 1 \\ C(2^1 - 1) &= 1 \end{aligned}$$

If we add up all the equations all but the first left hand side and all right handside C terms cancel out. The number of equations is t and thus the leftover number of 1s is t as well. Thus we obtain noting that $C(1) = 1$ the following.

$$C(2^t - 1) = (1 + 1 + \dots + 1) \Rightarrow C(2^t - 1) = t \Rightarrow C(n) = \lg(n + 1)$$

If n is not as specified (i.e. if $n + 1$ is not a power of two), then a ceiling will be needed.

$$C(n) = \lceil \lg(n + 1) \rceil$$

Note that $C(n)$ is the worst-case number of comparisons. If we do not have a worst-case the number of comparisons is smaller and as small as one. Thus, the number of comparisons in BinarySearch $C(n)$ is

$$C(n) \leq \lceil \lg(n + 1) \rceil$$

We then derive a running time bound $T(n)$

$$T(n) = O(C(n)) = O(\lg n)$$

The running time includes not only comparisons (i.e. line 7 of BiS) but everything else contributing to the running times (line 1-3 and line 5-9 as needed). Writing $T(n) \leq O(\lg n)$ is nonsense. We can only have $T(n) =$ or $T(n) \in$ as already discussed. Moreover in asymptotic notation we eliminate lower order term and multiplicative constants. ■

Binary Search Examples

Example 14. Party A guesses an integer x such that $0 \leq x < M$ and M a power of two (or not if we are careful with floors)). Party B knows M but does not know x .
Party B is only allowed to ask Party A questions of the type **Is x less than Z , for some $Z > 0$.**
Can Party B find x in $\lfloor \lg M \rfloor + 1$ questions? Let for this example $M = 128$. Can Party B find x in 7 questions?

```
x:  Is a 7-bit integer (values 0..127) since x< M=128
x:  x[6] x[5] x[4] x[3] x[2] x[1] x[0]
Findme (M)
1. temp = 0;
2. b= lg(M); // for M=128=> b=7
3. b--;
4. Question:  Is x < 2**b + temp ?
5. if answer == YES  then x[b]=0;
6. if answer == NO  then x[b]=1; temp+=2**b;
7. if b != goto step 3;
8  else  output(x);
```

An extension / generalization is the following example/problem.

Example 15. Party A guesses an integer x such that $0 \leq x$. Party B must find x with the same type of a supported question in twice the number of questions used before.
Is this possible?

Proof. Yes. The first $\lfloor \lg M \rfloor + 1$ questions establish the value of M . These are : Is $x < 2^0$? Is $x < 2^1$? Is $x < 2^2$? etc going on asking questions if the answer is a NO until eventually the answer is a YES for the question Is $x < 2^{\lg M}$? This would mean that we have found our unknown M . The remaining questions, having established M , are as above! ■

Properties of Sorting algorithms

In-place sorting

For completion we restate the problem of sorting. Implied is that the data structure used is an array that hosts the sequence of n keys.

Problem 9 (Key Sorting).

Input. A sequence of n comparable keys stored in array $A = \langle a_1, a_2, \dots, a_n \rangle$.

Output. A reordering, a permutation of the n keys $\langle a_{j_1}, a_{j_2}, \dots, a_{j_n} \rangle$ of the input sequence so that $a_{j_1} \leq a_{j_2} \leq \dots \leq a_{j_n}$.

Definition 28 (Property: sorting in-place algorithm). A sorting algorithm **sorts in place**, if the same space is used for input and output and the extra memory used by the algorithm is constant.

This definition becomes problematic when one needs to assess a recursive sorting algorithm: its non-recursive version is being used to determine its in-place status.

Properties of Sorting algorithms

Stable sorting

Definition 29 (Property: stable sorting algorithm). A sorting algorithm **is stable** if the relative order of same valued keys remains the same in the input and output sequences.

If the input keys are all distinct, stability is NOT an issue as there are no same valued keys.

Example 16 (Stability). Consider the problem of sorting student records, each record consisting of a college-year field (1:Freshman, 2:Sophomore, etc) and a name field. Somehow the input records are sorted by name, and we'll sort them by college-year.

Input: (sorted by name left to right)

<<2,Pat>,<2,Tim>,<2,Tom>,<1,Vic>>

Stable Algorithm gives

<<1,Vic>,<2,Pat>,<2,Tim>,<2,Tom>>

<<2,Pat>,<2,Tim>,<2,Tom>,<1,Vic>>

Non-stable algorithm may give

<<1,Vic>,<2,Tom>,<2,Pat>,<2,Tim>>

Both output sequences are sorted by college-year. The former however retains the original sorted order for same valued keys (i.e. all sophomore students are sorted by name).

Example 17 (Multi round sorting). Consider the problem of Google's sorting of triplets of the form

$\langle \text{DocID}, \text{wordID}, \text{Offset} \rangle$

where DocID is the document id of a document (i.e. a URL turned into a number), wordID is the word id of a given word (i.e. a word turned into a number) and offset is the starting position of an instance of the word in the document identified by wordID. When the Google indexer parses a single document it generates for that document triplets with the same DocID values. Because a document is parsed beginning to end, the triplets are sorted by offset for a given DocID. All such triplets of all documents are collected (and thus are sorted based on DocID as well). However the Google indexer wants to sort things based on wordID but at the same time does not want to destroy the existing orderings based on docID and also offset.

Google parser: processes one document at a time:

triplets sorted by Doc-ID and index.

$\langle 0, \text{"algorithm"}, 10 \rangle \langle 0, \text{"data"}, 20 \rangle \langle 0, \text{"algorithm"}, 30 \rangle \langle 1, \text{"data"}, 4 \rangle \langle 2, \text{"data"}, 20 \rangle \langle 2, \text{"data"}, 40 \rangle$

Stable sorting: Sort by word-ID (sorting by index is for free) ordering by index/doc-ID

$\langle 0, \text{"algorithm"}, 10 \rangle \langle 0, \text{"algorithm"}, 30 \rangle \langle 0, \text{"data"}, 20 \rangle \langle 1, \text{"data"}, 4 \rangle \langle 2, \text{"data"}, 20 \rangle \langle 2, \text{"data"}, 40 \rangle$

Not stable sorting: Sort by word-ID but previous orderings might get lost

$\langle 0, \text{"algorithm"}, 30 \rangle \langle 0, \text{"algorithm"}, 10 \rangle \langle 2, \text{"data"}, 40 \rangle \langle 1, \text{"data"}, 4 \rangle \langle 0, \text{"data"}, 20 \rangle \langle 2, \text{"data"}, 20 \rangle$

Definition 30. In the remainder we shall use the following definitions as they relate to a sorting algorithm L .

- $C(n)$ is the number of comparisons performed by L , on an input of length n (keys),
- $T(n)$ is the running time of L ,
- $S(n)$ is space requirements beyond the space used by the keys in the input array.

Note that $S(n)$ is defined differently than before. In fact the space requirements for sorting are $n + S(n)$ or $\Theta(n) + S(n)$, where the linear additive (first) term accounts for the input array.

DRAFT

Proposition 12 (NaiveSort). *Sorting algorithm NaiveSort has performance that can be summarized as follows. $C(n) \leq n! \cdot n$ and $T(n) = O(n! \cdot n)$. (Assuming we generate permutations efficiently, we will have $S(n) = \Theta(n)$ as well.)*

DRAFT

Elementary Sorting

Odd-Even transposition sort

Odd-even transposition sort is an unoptimized version of bubble sort. It is stable and sorts in-place. The algorithm is oblivious in the sense that it performs the same actions (comparisons) again and again independent of the data considered.

The description and the example that follows assume a Pascal indexing. The pseudocode we provide includes a 'Pascal indexing' version and a 'C/C++/Java' version.

Method 2 (Odd-even transposition sort). Odd-even transposition sort consists of n iterations, the even ones $2, 4, 6, \dots$ and the odd ones, $1, 3, 5, \dots$. In an odd iteration we compare keys $A[1] : A[2]$, $A[3] : A[4]$, etc and in an even iteration $A[2] : A[3]$, $A[4] : A[5]$, etc.

Example 18. An example of how odd-even transposition sort works is shown next.

	1	2	3	4	5	6	
Input	50	20	40	60	30	10	ODD-EVEN TRANSPOSITION SORT (PASCAL INDEXING)
Step 1.	50:20	40:60	30:10				
Step 2.	20	50:40	60:10	30			SWAP if left key > right key.
Step 3.	20:40	50:10	60:30				(left key is relative to comparator :)
Step 4.	20	40:10	50:30	60			
Step 5.	20:10	40:30	50:60				
Step 6	10	20:30	40:50	60			
Output	10	20	30	40	50	60	

Odd-Even transposition sort continued

Proposition 13 (OddEvenSort). *Odd-even transposition sort (OddEvenSort) is a sorting algorithm whose performance can be summarized as follows. The number of comparisons $C(n)$ performed on an input of length n is $C(n) \approx n^2/2$ and therefore $T(n) = \Theta(n^2)$, Space (additional to input) is $S(n) = \Theta(1)$. It is a stable algorithm that sorts in-place.*

Proof. `OddEvenSort(A[1..n],n) // 'Pascal'`

```
1. for(i=1 ; i<=n ; i++)
2.   for(j=odd(i); j< n; j+=2)
3.     if A[j] > A[j+1] swap(A[j],A[j+1]);
//odd(i):1 if i odd; 2 otherwise
  OddEvenSort(A[1..n])
1. for (i=1;i<=n;i++) {
2.   if (ODD(i))           // ODD(i) ?
3.     for (j=1;j<=n-1;j+=2)
4.       if A[j] > A[j+1] swap(A[j],A[j+1]);
5.   else // for even(i)
6.     for (j=2;j<=n-1;j+=2)
7.       if A[j] > A[j+1] swap(A[j],A[j+1]);
8. }
```

`OddEvenSort(A[0..n-1],n) //'C'`

```
1. for(i=0; i<n ; i++)
2.   for(j=odd(i); j< n-1 ; j+=2)
3.     if A[j] > A[j+1] swap(A[j],A[j+1]);
// odd(i):1 if i is; 0 otherwise!
```

DRAFT

Running Time and Properties. The number of iterations is $i = 1, \dots, n$. In each iteration we iterate over the keys for $j = \text{odd}(i), \text{odd}(i)+2, \dots$ and compare $A[j] : A[j+1]$ and swap if $A[j] > A[j+1]$. The running time and number of comparisons is $\#i \cdot \#j \approx n^2/2$, where $\#i, \#j$ is the number of time variable i and variable j are set in the for loop. The algorithm sorts in-place as it uses variables i, j , and temp for the swap that is $\Theta(1)$ additional space to array A . Stability is due to the swap that takes place for $A[j] > A[j+1]$. Thus a 10 : 10 will not cause a swap! ■

Elementary Sorting

Selection Sort

Method 3 (Selection sort). Selection sort is an incremental sorting algorithm. At iteration (step) i we find the minimum of the keys currently stored in $A[i..n]$ and swap that minimum with $A[i]$, if needed. The last iteration can be skipped thus algorithm runs for only $n - 1$ iterations.

The MIN of n keys requires $n - 1$ comparisons to be found as shown. In the example below we use Pascal type indexing.

Example 19. An example of how selection sort works is shown next.

	1	2	3	4	5	6	
Input		50	20	40	60	30	10
Step 1					--		Min is A[6]=10 and swapped with A[1]
		10		20	40	60	30
Step 2		--					Min is A[2]=20 and swapped? with A[2]
		10	20		40	60	30
Step 3				--			Min is A[5]=30 and swapped with A[3]
			10	20	30		60
Step 4				--			Min is A[5]=40 and swapped with A[4]
				10	20	30	40
Step 5					--		Min is A[6]=50 and swapped with A[5]
					10	20	30
Step 6						10	20
Output :						10	20

Selection Sort continued

Proposition 14 (SelectionSort). *SelectionSort is an incremental sorting algorithm whose performance can be summarized as follows. The number of comparisons $C(n)$ performed on an input of length n is $C(n) = n(n-1)/2$ and therefore $T(n) = \Theta(n^2)$. Space (additional to input) is $S(n) = \Theta(1)$. It is stable in the form of StableSelectionSort and sorts in-place. (All references to SelectionSort would refer to StableSelectionSort after this section.)*

Proof.

```
PSelectionSort(A[1..n],n) /* Pascal Indexing */ PMIN(A[1..n],n)
1. for(i=1 ; i < n ; i++)          1. t=A[1];
2.   swap A[i] with PMIN(A[i..n],n-i+1);  2. for(j=2 ; j<= n ; j++)
                                         3.   if A[j] < t  t=A[j];
                                         4. return(t)

SelectionSort(A[0..n-1],n) /* C indexing */ MIN(A[0..n-1],n)
1. for(i=0 ; i < n-1 ; i++)          1. t=A[0];
2.   swap A[i] with MIN(A[i..n-1],n-i);  2. for(j=1 ; j< n ; j++)
                                         3.   if A[j] < t  t=A[j];
                                         4. return(t)
```

Running Time and Properties. The number of steps (iterations) is $i = 0, \dots, n-2$. In each iteration we find the minimum of $A[i..n-1]$ in $n-i-1$ comparisons (see table below).

Iteration 0	$n-1$ comps to find MIN of ALL n keys
Iteration 1	$n-2$ comps to find MIN of remaining $n-1$ keys
Iteration 2	$n-3$ comps to find MIN of remaining $n-2$ keys
...	...
Iteration $n-2$	1 comps to find MIN of remaining 2 keys

Adding all comparisons for $i = 0, \dots, n-2$ gives $C(n) = (n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2$. Thus $T(n) = \Theta(n^2)$ as comparisons determine the running time. Moreover by inspection $S(n) = \Theta(1)$. This is because in addition to A we need space for i and j , and variable t for minimum finding. Moreover the swap operation requires an implied variable $temp$. ■

Selection sort Stability part 1

Question 5. Is the algorithm as described a stable one?

Proof. No it is not. Proof by counterexample.

	1	2	3
Input	20	20	10
	\		--
	\		
		\	
Iter-1	10	20	20
	--		
Iter-2	10	20	20
Output	10	20	20

The first 20 moves third and stays there!

DRAFT

■

Selection sort Stability part 1

Question 6. Is there a version of selection sort algorithm that is stable so that we can claim SelectionSort is stable? Show that there exists one.

Proof. Yes there is one. Even if the previous version is not stable, It can become stable if instead of swapping the MIN with $A[i]$ we shift the keys between index position i and the index of the MIN right one index position and then concluding by moving the MIN into $A[i]$. The properties for $C(n)$, $T(n)$ and $S(n)$ as described in the previous proposition are asymptotically still the same.

```
StableSelectionSort(A[0..n-1],n)
1. for(i=1 ; i < n ; i++)
2.   index = MIN(A[i..n-1],n-i);
3.   HalfSwap(A,index,i,n);

HalfSwap(A,index,i,n)
1. temp=A[index];
   ShiftRightbyOne(A,i,index)
2. ShiftRightbyOne(A[i]...A[index-1])
3.   for(j=index-1; j>=i; j--)
4.     A[j+1]=A[j];
5.   A[i]=temp;

MIN(A[0..n-1],n)
1. t=A[0];index=1;
2. for(j=1 ; j< n ; j++)
3.   if A[j] < t  t=A[j],index=j;
4. return(index)
```

Input: | 20 20 10*

Iter-1: 20 20 -- temp=10

 \ \

 20 20

 10 | 20* 20

Iter-2: 10 20 | 20

Output: 10 20 20

■

Elementary Sorting

Insertion Sort

Method 4 (Insertion Sort). Insertion sort is an incremental sorting algorithm. At iteration i , $key = A[i]$ is inserted into the sorted sequence formed after the sorting of the first $i - 1$ keys. Thus at iteration i the keys $A[1..i - 1]$ have already been sorted, and we are currently considering inserting $A[i]$ into the appropriate position into $A[1..i - 1]$ so that the keys $A[1..i]$ get sorted; in the process of doing so, we temporarily store $A[i]$ in variable key . The keys $A[i + 1..n]$ are still part of the input and under future consideration.

Example 20. An example of how insertion sort works is shown next. The bar separates the output already formed $A[1..i - 1]$ from the remaining input $A[i + 1..n]$. The keys under consideration is $key = A[i]$ and marked with a * on its right. The diagonal marks indicate the half-swaps that are performed (movement of a key one position right of its current position).

The pseudocode utilizes Pascal indexing.

```
InsertionSort (A[1..n],n)
1. for (i=2; i<=n; i++)
2.   key= A[i];
3.   j=i-1; // Insert A[i] in the sorted sequence A[1..i-1]
4.   while (j>0) and (A[j] > key) {
// Note           ^^^ : Stability dependence
5.     A[j+1]= A[j];
6.     j=j-1;
   }
7.   A[j+1]=key;
```

Note that step 1 of InsertionSort can be omitted: inserting $key = A[1]$ into an initially empty output is trivial ignored. The for loop starts with $i = 2$ not $i = 1$.

Insertion Sort Example

Example 21. An example of Insertion Sort for 6 keys is given below. The | indicates the boundary between left and right. Left are the keys of the input already sorted: right are the keys of the input not already sorted. At the start left side is empty; at the end right side is empty and left side contain all the n input keys sorted.

	1	2	3	4	5	6
Input	50	20	40	60	30	10
Step 1.	50*	20	40	60	30	10
Step 2.	50	20*	40	60	30	10
Step 3.	20	50	40*	60	30	10
Step 4.	20	40	50	60*	30	10
Step 5.	20	40	50	60	30*	10
Step 6.	20	30	40	50	60	10*
Output	10	20	30	40	50	60

Step 1 can be omitted.

DRAFT

Insertion Sort

Worst, Best, Average case examples

Example 22. An example that describes the worst case, the best case and the average case is shown next.

Input:	Sorted	Reverse-Sorted	Random
	<1 2 3 4 5 >	<5 4 3 2 1 >	<2 5 1 4 3 >
	<1 2 3 4 5 >	<5 4 3 2 1 >	<2 5 1 4 3 >
	<1 2 3 4 5 >	<4 5 3 2 1 >	<2 5 1 4 3 >
	<1 2 3 4 5 >	<3 4 5 2 1 >	<1 2 5 4 3 >
	<1 2 3 4 5 >	<2 3 4 5 1 >	<1 2 4 5 3 >
	<1 2 3 4 5 >	<1 2 3 4 5 >	<1 2 3 4 5 >
	4 comparisons	10 comparisons	9 comparisons ?

Fact 1. With reference to InsertionSort. The best case number of comparisons is $n - 1$; proof in Question 7. The worst case number of comparisons is $n(n - 1)/2$; proof in Question 8. The average case number of comparisons is $\approx n(n - 1)/4$; Proof in Question 9.

Insertion Sort

Worst, Best, Average performance

Question 7. How many (key) comparisons does InsertionSort perform on a sorted sequence of n keys [best case] ?

Answer. The number is $n - 1$; in iteration i , for all $i = 2, \dots, n$, we perform for $j = i - 1$, one $A[j] > key$ comparison. ■

Question 8. How many (key) comparisons does InsertionSort perform on a reverse-sorted sequence of n keys [worst case]?

Answer. The answer is $n(n - 1)/2$; in iteration i , for all $i = 2, \dots, n$, we perform $i - 1$ comparisons. ■

Question 9. How many (key) comparisons on a random sequence of n keys?

Answer. The answer is $n(n - 1)/4$ on the average; for an arbitrary (random) sequence in iteration i we perform $(i - 1)/2$ comparisons on the average, for all $i = 2, \dots, n$. We expect half of the keys to be greater than $key = A[i]$ and half smaller. (A proper argument counts the average number of inversions in an input. An inversion (p, q) is one such that $p < q$ but $A[p] > A[q]$.) ■

Insertion Sort

Overall performance

Question 10. How many (key) comparisons on n arbitrary keys?

Answer. Arbitrary does not mean random! At most the worst case number i.e. $\leq n(n-1)/2$. ■

Question 11. What comparisons are we talking about?

Answer. Of the type $A[j] > key$ involving keys, i.e. key comparisons. A $j > 0$ is NOT a comparison since neither j nor 0 is a key! ■

Question 12. What is the number of comparisons of InsertionSort?

Answer. Based on the previous question, the number $C(n)$ is bounded above by $n(n-1)/2$ and below by $n-1$, therefore $n-1 \leq C(n) \leq n(n-1)/2$. Thus $C(n) = O(n^2)$ but also $C(n) = \Omega(n)$. ■

Insertion Sort Properties

Proposition 15 (InsertionSort). *InsertionSort is an incremental sorting algorithm whose performance can be summarized as follows. The number of comparisons $C(n)$ for an input of length n is such that $n - 1 \leq C(n) \leq n(n - 1)/2$ and therefore $T(n) = O(n^2)$. Space (additional to input) is $S(n) = \Theta(1)$. InsertionSort is stable and sorts in-place.*

Proof. We have already given the answer for $C(n)$ through the questions answered earlier. We instead analyze the running time of InsertionSort by analyzing the contribution of EVERY operation rather than just that of a key comparison. We present below a C/C++ version of the code and swap i and j in it. The remaining discussion refers to this version.

InsertionSort (A[0..n-1],n)	Cost	TimesExecuted
1. for (j=1;j<n;j++) {	c1	n
2. key= A[j];	c2	n-1
3. i=j-1;// Insert A[j] in the sorted sequence A[0..j-1]	c3	n-1
4. while (i>= 0) and (A[i] > key) {	c4	t1+t2+...+tn-1
// Note ^^^ : Stability dependence		
5. A[i+1]= A[i];	c5	(t1-1)+...+(tn-1-1)
6. i=i-1;	c6	(t1-1)+...+(tn-1-1)
}		
7. A[i+1]=key;	c7	n-1
8. }		

■

Question 13. How many times is line 1 executed ?

Answer. The number is n , and not $n - 1$. ■

Question 14. How many times are lines 2, 3, 7 executed ?

Answer. The number of times is $n - 1$. Once per key other than the first one. ■

Question 15. How many times is line 4 executed ?

Answer. It depends on the input sequence that is the input. For key $A[j]$ let t_j (t_j) be that number! It is $1 \leq t_j \leq j + 1$. Note that the number of comparisons in Line 4 is one less $t_j - 1$. ■

Question 16. Running time on an input instance?

Answer. The running time of an algorithm depends on the input (input instance) and problem size and thus it is expressed as a function of n . The rest addresses this question in conjunction with the answers of the previous three questions. ■

Insertion Sort continued

Detailed analysis continues. Line k has cost some constant c_k . The running time $T(n)$ of InsertionSort for an input of size n is thus given by the following expression.

$$T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_7(n-1) + c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1} (t_j - 1) + c_6 \sum_{j=1}^{n-1} (t_j - 1)$$

Best case. For the best-case, it is $t_j = 1$ for all j . Therefore, a best-case is a case where the input is already sorted. The best-case running time is then.

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_7 + c_4)$$

Running time is linear in n , or $\Theta(n)$, since all c_i are constant. The running time of InsertionSort on n sorted keys is $T(n) = \Theta(n)$.

Worst case For the worst-case, it is $t_j = j + 1$ and therefore, a worst-case is a case where the input is reverse-sorted. The worst-case running time is then.

$$T(n) = (c_4 + c_5 + c_6)n^2/2 + (c_1 + c_2 + c_3 + c_4/2 - c_5/2 - c_6/2 + c_7)n - (c_2 + c_3 + c_7 + c_4)$$

Running time is quadratic in n , or $\Theta(n^2)$. In this case, we used the fact that The running time of InsertionSort on n reverse-sorted keys is $T(n) = \Theta(n^2)$.

Average case. On the average, key $key = A[j]$ is smaller (or bigger) than half of the j keys ($A[0..j-1]$) already considered, i.e. $t_j = j/2 + 1$ (the plus one is for the first condition of the **while** that is always executed). Running time is "half" of that of the worst case, or asymptotically speaking $\Theta(n^2)$ as well.

The running time of InsertionSort is bounded by the worst-possible running time on an input instance. Thus

$$T(n) = O(n^2).$$

This is because the running time is bounded above by the worst-case's running time. (It is also bounded below by the best-case's running time.) On an arbitrary input instance the running time is between the lower bound and the upper bound. The maximum (the upper bound) determines the running time of an algorithm. ■

Elementary Sorting

Merge sort

A **divide-and-conquer** method involves three phases (steps).

1. **Divide** the problem into a number of independent subproblems.
2. **Conquer** the independent subproblems by solving them recursively. The recursion terminates when subproblem size is so small that it can be solved directly (the $n = 0$ and $n = 1$ for Fibonacci).
3. **Combine** the subproblem solutions to get the solution of the original problem.

We describe a sorting algorithm based on divide and conquer that is known as merge sort. For the divide phase to work nicely we may (theoretically assume) that n is divided by two. A simple theoretical assumption is that n is an integer power of two. Thus the half of the half of the half etc of n is still an integer!

Method 5 (Merge sort). Merge sort is a sorting algorithm that utilizes divide and conquer. Even if divide and conquer naturally provides a recursive solution, one can eliminate recursion by providing an iterative solution for merge sort. The divide, conquer and combine phase of merge sort are highlighted below.

- **Divide phase:** divide n keys into two sequences of $n/2$ keys each.
- **Conquer phase:** Likewise (recursively) sort the two sequences generated.
- **Combine phase:** Merge the two sorted sequences of length $n/2$ to produce the sorted result. (The Merging problem is defined separately next.)

Merge sort is stable but the implementation provided does not sort in-place.

Elementary Sorting

Merging is not Merge sort

Problem 10 (Merging (of two sorted sequences)).

Input. Two sorted sequences $X[1..k]$ and $Y[1..l]$

Output. Merge X and Y into a sorted sequence $Z[1..k+l]$ keys, using at most $k+l-1$ comparisons.

Solution.

The proof is by example. The number of comparisons is at most $k+l-1$. Merging is stable because $X[i] \leq Y[j]$ moves a key from X to the output (e.g. $X[i] = Y[j]$).

Example 23 (Merging).

```
MERGING : Pick next X[i] and next Y[j]:
           If X[i]<= Y[j] move X[i] to output
           else          move Y[j] to output
X:< 1 13 14 > Y:< 2 5 6 27> ==> Z < 1          >
*              *
X:< 1 13 14 > Y:< 2 5 6 27> ==> Z < 1 2          >
* *              **
X:< 1 13 14 > Y:< 2 5 6 27> ==> Z < 1 2 5        >
* **              ***
X:< 1 13 14 > Y:< 2 5 6 27> ==> Z < 1 2 5 6      >
* ***              ****
X:< 1 13 14 > Y:< 2 5 6 27> ==> Z < 1 2 5 6 13   >
* ****              *****
X:< 1 13 14 > Y:< 2 5 6 27> ==> Z < 1 2 5 6 13 14 >
* *****          *
X:< 1 13 14 > Y:< 2 5 6 27> ==> Z < 1 2 5 6 13 14 27 >
* *****          *
* *****          *
* *****          *
```

In the example above a star under key indicates a comparison in which the key participated. The total number of stars on X keys is equal to the total number of stars on Y keys. Moreover the number of stars is $k+l-1$. All but the maximum entered Z following a less than or equal result of a comparison performed. The maximum (27) entered Z by default: there is no key larger than 27 in the input.

A Python-like pseudocode would look like as follows. ■

Merging is faster than sorting

Solution continued.

Example 24 (Merging).

```
def Merging(X,Y):
    Z = []
    i, j = 0, 0

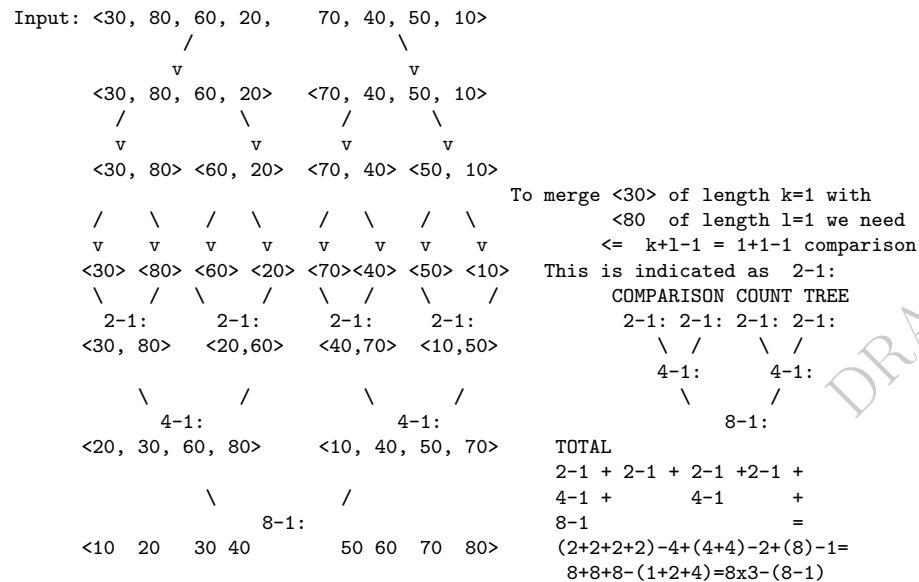
    k=len(X); l=len(Y); m=k+l;
    for t in range(m):
        if i > k-1:
            Z.append(Y[j])
            j += 1
        elif j > l-1:
            Z.append(X[i])
            i += 1
        else:
            if X[i] <= Y[j]:
                Z.append(X[i])
                i += 1
            else:
                Z.append(Y[j])
                j += 1
    return Z
```



Elementary Sorting

Merge sort

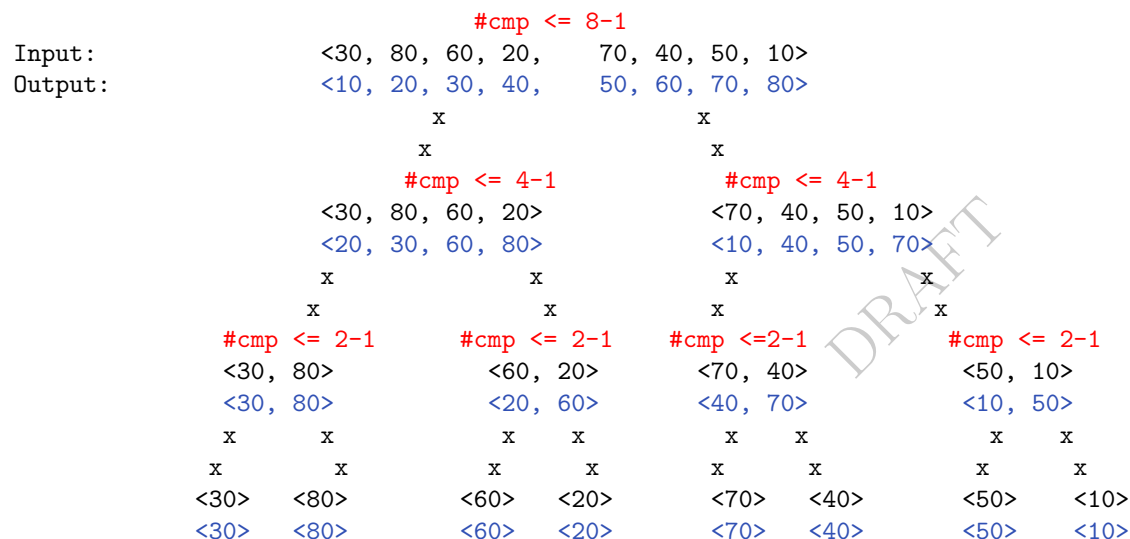
Example 25. An example of how merge sort works is shown next. The example shows also the number of comparisons performed (upper bound).



Elementary Sorting

Merge sort

The example figure below is the same as the one on the previous page. It indicates the recursion tree of MergeSort on the way down (divide stage) and also on the way up (conquer stage). The input sequences in black color indicate the input during the divide stage (steps). The input sequences in blue indicate the (partial) outputs during the conquer stage (steps). In those steps already sorted subsequences are getting merged into a (sorted) subsequence of double the length of its inputs. At the bottom subsequences are of length one and thus sorted by default. In red we indicate the (upper bound on the) number of comparisons performed to generate the corresponding output (sequence in blue two lines after). The sum of the red values of all the nodes of the recursion tree is the upper bound on the number of comparisons for this instance of merge sort.



Proposition 16 (MergeSort). *MergeSort is a divide and conquer sorting algorithm whose performance can be summarized as follows. The number of comparisons $C(n)$ for an input of length n that is a power of two is such that $C(n) \leq n \lg n - (n - 1)$ and therefore $T(n) = O(n \lg n)$. Space (additional to input) is $S(n) = \Theta(n)$. MergeSort is stable but does not sort in-place. (The running time bound can be further tightened.)*

Merge sort more

Proof.

MergeSort(A[0..n-1],n)	$T(n)$	$C(n)$
1. MS(A,0,n-1);	$T(n)$	$C(n)$
MS(A,l,r) // A[l .. r]	$l=1, r=n$	
1. if (l < r) {	$\Theta(1)$	0
2. m= l + (r-l)/2; // m=(l+r)/2	$\Theta(1)$	0
3. MS(A, l, m);	$T(n/2)$	$C(n/2)$
4. MS(A, m+1, r);	$T(n/2)$	$C(n/2)$
5. MERGING(A[l..m],A[m+1..r],A[l..r],m-l+1,r-m);	$\Theta(n)$	$\leq n-1$
6. }		

Running Time and Comparisons plus Properties. If the running time of MergeSort is $T(n)$ then so is the running time of MS on n keys. MS on n keys has $l = 0$ and $r = n - 1$. Lines 1 and 2 of MS are $\Theta(1)$. MERGING of line 5 is $\Theta(n)$ and line 3 and line 4 involves roughly $n/2$ keys in each call, each contributing $T(n/2)$ to the running time respectively. Thus $T(n)$ is given by

$$T(n) = 2T(n/2) + \Theta(n)$$

For a comparison count, if $C(n)$ is the count for MergeSort and MS on n keys, line 3 and line 4 contribute $C(n/2)$ and $C(n/2)$ respectively. Line 5 contributes at most $n - 1$. Thus $C(n)$ is given by

$$C(n) \leq 2C(n/2) + (n - 1)$$

MergeSort does not sort in-place as described; this is because of MERGING (uses Z a separate array for output, or as intermediate space before the output gets copied back to X and Y). MergeSort is stable by way of the stability of MERGING. We can solve the recurrence relations above to derive a time or comparison count for MergeSort. For the latter see the recursion tree of the next page that shows the number of comparisons for an input with $n = 8$ and also for n an arbitrary power of two. It is straightforward to derive that $C(n) \leq n \lg n - (n - 1)$, for n a power of two. (The recursion tree is shown on the next page.) Moreover the implication is that the running time is

$$T(n) = O(n \lg n).$$

■

Merge sort more

Recursion tree. Recursion level 0 through $\lg n$ maps to the Divide part where we split the original (unsorted) input into smaller inputs of half the previous level length until we get input sequences of length one at level $\lg n$. A sequence of length one is sorted by default. Thus the merging of line 5 would start merging smaller (sorted) subsequences to double the length (sorted) subsequences when recursion pops up from level $\lg n$ back to level 0. On the way up (but also on the way down) at recursion level i there are 2^i nodes in the recursion tree. Each one of them maps to a subsequence of length $n/2^i$; on the way up the subsequence becomes sorted when level i has popped up (completed). Two such sorted subsequences are then merged at the next level $i - 1$ into one of size double of the length of the two i.e. of length $n/2^{i-1}$. The number of comparisons performed during that merging would be at most $n/2^{i-1} - 1$. For the popped up level we show on the right the number of comparisons involved at a node of some level, and the total number of comparisons involved on ALL the nodes of that level. The number of comparisons of MergeSort is then equal to the sum of the number of comparisons of all levels i.e. the sum of the values of the last column indicated. It is similar to Example 25

Level			#comps per node	#node	#comps per level
0	n				
1	<div> <div>/ \</div> <div>n/2 n/2</div> </div>				
2	<div> <div>/ \ / \</div> <div>n/4 n/4 n/4 n/4</div> </div>				
	<div> <div>/ \ / \ / \ / \</div> <div>... </div> </div>				
lgn-1	<div> <div>2 2 2 2 ... 2 2 2 2</div> <div>/ \ / \ / \ / \ / \ / \ / \</div> </div>				
lgn	<div> <div>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</div> <div>\ / \ / \ / \ / \ / \ / \ / \</div> </div>				
lgn-1	<div> <div>2 2 2 2 ... 2 2 2 2</div> <div>\ / \ / \ / \ / \ / \ /</div> </div>		2-1	n/2,	n - n/2**1
lgn-2	<div> <div>4 4 4 4 ... 4 4 4 4</div> <div>\ / \ / \ / \ / \ / \ /</div> </div>		4-1	n/4,	n - n/2**2
	...				
2	<div> <div>n/4 n/4 n/4 n/4</div> <div>\ / \ / \ / \ /</div> </div>		n/4-1	2**2,	n - n/2**(lgn-2)
1	<div> <div>n/2 n/2</div> <div>\ / \ /</div> </div>		n/2-1	2**1,	n - n/2**(lgn-1)
0	n		n-1	2**0,	n - n/2**lgn

Merge sort

End of analysis

Continued.

$$\begin{aligned} C(n) &\leq \sum_{i=0}^{\lg n - 1} (n - 2^i) \leq n \cdot \lg n - (2^0 + 2^1 + \dots + 2^{\lg n - 1}) \leq n \cdot \lg n - (2^{\lg n} - 1) \\ &\leq n \cdot \lg n - (n - 1). \end{aligned}$$

■

Example 26 (MergeSort (Python)).

```
def MergeSort(A):
    return(MS(A))

def MS(A):
    if len(A) < 2:
        return A[:]
    else:
        m = (len(A)-1)//2
        L = MS(A[:m+1])
        R = MS(A[m+1:])
        return Merging(L,R)
```

Selection problems

Maximum and Minimum

Problem 11 (Maximum).

Input. A sequence $A[1..n]$ of n comparable keys.

Output. The maximum of A i.e. the key r such that $r \geq A[i]$ for all $i = 1, \dots, n$.

Problem 12 (Minimum).

Input. A sequence $A[1..n]$ of n comparable keys.

Output. The minimum of A i.e. the key r such that $r \leq A[i]$ for all $i = 1, \dots, n$.

Solution. We are interested in finding an efficient solution to this Maximum (and its symmetric Minimum) problem. We provide solutions for both (Pascal indexing) and do it counterway (for loops in opposite directions).

```
MAXIMUM( A[1..n], n)           MINIMUM( A[1..n], n)
1.  r = A[n];                   1.  r = A[1];
2.  for(i=n-1 ; i >= 1 ; i--) {  2.  for(i=2; i <= n; i++) {
3.    if A[i] > r                3.    if A[i] < r
4.      r = A[i];               4.      r = A[i];
5.  }                           5.  }
6.  return(r)                  6.  return(r)
```

■

If we have found the maximum of the last $n - i$ elements that are stored in positions $A[i + 1], \dots, A[n]$ and this maximum is maintained in variable r we then go to the next element (downwards) of the sequence which is $A[i]$. We do not need to compare this element with all previous $n - i$ elements, we just need to compare $A[i]$ with the maximum of the last $n - i$ elements already stored in r . Thus line 3 determines whether $A[i]$ is the new maximum of $A[i], A[i + 1] \dots A[n]$. If this is the case r gets updated in line 4 if a value greater than the current maximum has been encountered, otherwise line 4 is ignored.

The code for MINIMUM works similarly.

Maximum and Minimum continued

Lemma 17. *The number of comparison performed (by MAXIMUM or MINIMUM) is $n - 1$.*

Proof. **Running time of MAXIMUM.** Line 3 of the for loop is executed $n - 1$ times; Line 4 is executed at most $n - 1$ times but as few as 0, if $A[n]$ is apparently the maximum. In other words $T(n)$ is a linear function to n and thus $T(n) = \Theta(n)$.

Number of comparisons of MAXIMUM. This is easier to compute. Line 3 is executed $n - 1$ times exactly. Thus to find the maximum of n keys we need to perform $n - 1$ key comparisons (such as those between $A[i]$ and r). ■

Question 17. Is there a reason one scans the input left to right or right to left?

Answer. We can write the code for MAXIMUM one way or the other way and so can we the code for MINIMUM. However if the function is to be used by a sorting algorithm to be called StableSelectionSort, for StableSelectionSort to be stable, if all keys are equal, the minimum must be the first (of equal) keys and the maximum the last (rightmost). ■

Maximum and Minimum continued

Question 18. Can we find MAXIMUM with at most $n - 2$ comparisons in a worst-case scenario?

Question. NO. $n - 1$ is the best we can do. Every key that is not the maximum must be disqualified from being MAXIMUM. In every comparison we have one disqualification. We need to have $n - 1$ disqualifications for the $n - 1$ keys that will not be the MAXIMUM, thus we need at least $n - 1$ comparisons. Think it also the following way. If there is an algorithm that 'finds' the MAXIMUM in $n - 2$ comparisons it means that at most $n - 2$ keys got disqualified, one was decided to be the MAXIMUM and (at least) one key never got disqualified. Change the value of that key to something larger than the MAXIMUM value and repeat the comparisons of the 'n-2' algorithm. All remaining keys have the same values as before so all previous steps/comparisons would be repeated. The only problem is that the wrong MAXIMUM would be reported now! (The previous MAXIMUM is not the MAXIMUM any more as a key has a value greater than that.) ■

Question 19. Is MAXIMUM time efficient?

Question. We just proved that MAXIMUM is comparison efficient. Thus it is also time efficient. In the worst-case we can do no better than $\Theta(n)$! ■

Question 20. Is MAXIMUM space efficient?

Question. We are frugal in space allocation for MAXIMUM; other than A that holds the keys/elements, we only used i and **result** i.e. a constant number of variables (i.e. $\Theta(1)$ variables). So it is space efficient (asymptotically). No matter what you do you need one variable to hold the answer, and the difference between one and two is not asymptotically important! ■

Question 21. Is MAXIMUM efficient?

Question. Yes because it is space and time efficient! ■

Sorting

Stable sorting and more

Every sorting algorithm can be turned into a stable sorting algorithm, if it is not. To achieve this, we modify the input. If the i -th input key is k_i , we “augment the key” into (k_i, i) , where i is either the index or memory location storing k_i in the input. A comparison always resolves into $<$ and $>$ and never to $=$. For $(k_i, i) \neq (k_j, j)$ a $k_i : k_j$ resolves to $<$ or $>$ or we further compare $i : j$ to resolve to $<$ or $>$. Sorting the sequence $\langle k_1, k_2, \dots \rangle$, is equivalent to sorting the sequence $\langle (k_1, 1), (k_2, 2), \dots \rangle$; in the latter sequence however, no two keys are equal (i.e. they are all distinct) so stability problems are not possible.

Comparison-based and Distribution-based sorting

Comparison-based			
In-place	NotInPlace	Stable	Not-stable
InsertionSort	NaiveSort	InsertionSort	HeapSort
SelectionSort	MergeSort	SelectionSort	QuickSort
Odd-Even Tr. Sort	QuickSort	Odd-Even Tr. Sort	
HeapSort		MergeSort	
Distribution-based			
In-place	NotInPlace	Stable	Not-stable
	BucketSort	BucketSort	
	CountSort	CountSort	
	RadixSort	RadixSort	

(As we already mentioned, SelectionSort refers to the stable version also known as StableSelectionSort.)

All of the sorting algorithms we have examined so far are comparison-based. A function `compare` compares two keys and a negative, 0, or positive value (eg -1,0,+1) is then output. There exist some sorting algorithms that do not use key comparisons. Instead they inspect the key values because they have prior knowledge of those key values. Such algorithms are known with the names of CountSort, BucketSort and RadixSort.

Algorithms

Analysis recapture

Let A be an algorithm, and I an input instance of n keys.

1. Running time of a single input instance: $T_A(n, I)$. It is the running time of algorithm A on an input instance I of n keys, i.e. of problem size n .

2. Worst-case running time: $W_A(n)$.

$$W_A(n) = \max_{I, |I|=n} \{T_A(n, I)\}.$$

3. Best-case running time: $B_A(n)$.

$$B_A(n) = \min_{I, |I|=n} \{T_A(n, I)\}.$$

4. Running Time $T_A(n)$ or just $T(n)$.

$$T_A(n) = T(n) = O(W_A(n)).$$

If $W_A(n) = \Theta(B_A(n))$, then we can also write,

$$T_A(n) = T(n) = \Theta(W_A(n)).$$

Examples. The best-case running time of InsertionSort is attained by a sorted sequence: Thus $T_{\text{InsSort}}(n, \text{Sorted}) = \Theta(n)$. Its worst-case running time is attained by a reverse sorted sequence: $T_{\text{InsSort}}(n, \text{ReverseSorted}) = \Theta(n^2)$.

The running time bound of InsertionSort is $O(\max_I T_{\text{InsSort}}(n, I)) = O(n^2)$. This is because $\max_I T_{\text{InsSort}}(n, I) = \Theta(n^2)$ as the maximum is the running time of a reverse sorted sequence. When we say that the running time of InsertionSort is $O(n^2)$ we mean that its running time can be $\Theta(n^2)$ for some inputs, or anything in between for some of the other inputs, including $\Theta(n)$ for some other inputs. More on asymptotic notation in Subject 2.

Static and Dynamic Sets

Introduction to operations on Sets

Sets as well as operations on sets are of paramount importance in computer science. Although in a mathematical context a set is a static concept, in computer science a set becomes a dynamic entity whose size can change during the course of the execution of a program/algorithm. Such sets are called **dynamic** to distinguish them from **static** ones. In the remainder at some point we shall drop the static or dynamic: the type is to be derived from the relevant context.

For a dynamic set we need to introduce the operations that operate on that set and describe algorithms that implement these operations. Such operations can be **modifying operations** or **probing operations**.

Modifying operations. Data structures get modified through “modifying operations” that add to or delete elements from them. Thus such operations are valid or make sense for dynamic sets only. Modifying are operations such as Insert(k), Delete(z), ExtractMax() or DeleteMax(), ExtractMin() or DeleteMin(), that modify a set. Here k is a key value, and z is a pointer or reference to the element that is going to be deleted (the element is an object that includes a key value as its members.) Moreover for Delete we do not need to Search for the element z or its key value. A modifying operation on a set must leave the set in the same state prior to the operation: thus if a set satisfies a property prior to an operation, the same property must still be satisfied after the operation. With modifying operations the size of the set usually changes: it either increases or decreases. One needs to have safeguards for underflow and overflow. Underflow is an attempt to delete an element from an empty set, and overflow to insert an element into a full set (and fullness might have to do with the lack of available physical (RAM) memory. An operation ExtractMax could also be a modifying operation: it deletes the Maximum element.

Probing operations. Static data structures support “probing operations” that do not modify the data structure i.e. neither increase nor decrease the number of its elements nor change its structure. Probing operations are operations such as Search(k), Min (i.e. Minimum) or FindMin, Max (i.e. Maximum) or FindMax, Predecessor(x) and Successor(x) that do not modify the set but just probe it. Here k is a key value, and x is a pointer or reference to the element whose successor or predecessor will be determined (and x as an object contains a key value as its member data). The size of the set remains the same during and after the operation.

Dynamic Sets Operations

Efficiency of representation of a set: choice of data structure. Issues related to the efficiency of these operations need to be dealt with as well. Foremost and related to such issues is the problem of the representation and storage of a set in the memory of a computer.

In the following discussion U is a universe of elements, $S \subseteq U$ denotes a set and $x \in U$ is an element of U . Each x has associated with it a field **key** with value k . Other information may also be associated with x . Without loss of generality we shall ignore this additional information as for the operations that we shall describe for a set S , this additional information does not affect the description of the operations.

Search(S,k)	Given a set S and a key value k , decide whether there exists an element x of S whose key is k . If there exists such an x return a pointer to x , otherwise return NULL.
Insert(S,x)	or Insert(S,k). Given a set S , x is inserted into S ; an alternative is to insert just a key value k . (x usually contains k plus additional information.)
Delete(S,z)	Given a set S , and a pointer to an element z of S , element z is deleted.
Min(S)	Given a set S find the element of x with the minimum key value and return the value (or the element with the min value). ExtractMin or DeleteMin also deletes element.
Max(S)	Given a set S find the element of x with the maximum key value and return the value (or the element with the max value). ExtractMax or DeleteMax also deletes element.
Successor(S,x)	Given x and S the element of S is returned with value the next larger to the value k of x or NULL.
Predecessor(S,x)	Given x and S the element of S is returned with value the next smaller to the value k of x or NULL.

```
-----
| 50 |--->| 20 |-->| 10|-->| 30 |--> | 40 |
|    |<---|    |<---|    |<---|    |
-----
```

Predecessor and Successor. The Predecessor(x) of x is the largest element $\leq x$. Thus the predecessor of 50 is 40. The Successor(x) of x is the smallest element $\geq x$. Thus the successor of 20 is 30. None of those operations has anything to do with pointers or references (next, previous or otherwise).

Sets Dictionary

A dictionary abstract data type consists of words (data model) and operations such as Insertion of a word, Search for a word, and possibly Deletion of a word from the dictionary.

For a given dictionary An **Insert** is an operation that inserts a word into the dictionary thus increasing its size by one.

A **Search** is an operation that searches for a word in an already formed dictionary.

A **Delete** is an operation that deletes a specified word from the dictionary thus reducing its size by one.

Static Dictionary. For such a dictionary the number of its elements (words) remains the same. A static dictionary is one that does not support operations Insert and Delete. Only a Search or similar operations are allowed. (Think of a book dictionary that allows for look-ups only.)

Dynamic Dictionary. A “full-dictionary” is a *dynamic* data structure that supports operations Insert and Delete in addition to Search. Its size can thus change either on the upside or the downside.

A data-structure for the Dictionary ADT

There are many ways one can implement a dictionary. One could use an array, a sorted array, a linked-list or a sorted linked-list, or a binary-search tree to implement a dictionary ADT. (A linked-list can be singly-linked or doubly-linked.) If an array is used, Search would be a slow operation if realized through **LinearSearch**; a sorted array can take advantage of a more efficient search operation known as **BinarySearch**. Thus the data-structure chosen affects the performance of the operations and also dictates the possible algorithms that can be used to realize them.

Linked Lists

Singly or Doubly linked

Array vs Linked List structures. For an array A to find the i -th element takes $\Theta(1)$ constant time as in $A[i - 1]$ (C/C++ indexing). An **array** is a random access data structure. We can access the i -th element of the array in unit time. Whether we access the 1st, i -th or last element, the time is the same. **Random Access** means the same as the term Random Access Memory (RAM). For a linked list L take $\Theta(i)$ time as we need to traverse all prior $i - 1$ elements.

Singly vs Doubly Linked List. For the former Delete(z) is a $\Theta(i)$ operation if element is the i -th element from the start of the list (head pointer available only). For the latter Delete(z) is a constant $\Theta(1)$ time operation. Always use in practice a doubly linked list.

Singly and Doubly Linked List. Several times both a head pointer (points to first element) and a tail pointer (points to last element) are available.

In a **circular** list next of last element (tail) is the first (head).

What is a **Sentinel**? We may avoid the use of null pointers by making such elements (pointing to null) to point to a sentinel element (a default-non-valid-list-element) that points to the head of the list.

Overflow is the exception of trying to Insert into a Full structure (queue, stack, etc).

Underflow is the exception of trying to Delete from an Empty structure (queue, stack, etc).

Stacks and Queues

Quick overview

Stacks and queues are dynamic sets where the element that is being deleted by a DELETE operation is prespecified. In a **stack** this element is the most recently inserted into the stack. In a **queue** it is the element least recently inserted. For this reason the two structures are also known as LIFO (last-in first-out) and FIFO.

For a stack, the Insert and Delete operations are more often called **Push** and **Pop**.

An implementation of a stack in terms of an array is immediate. The leftmost end of the array is the bottom of the stack, the stack grow to the right and the top of the stack is pointed by a pointer **top**. A stack with no elements is an empty stack. If an operation tries to reach an element past the left end an **underflow** has occurred. If it tries to reach an element past the right end an **overflow** is flagged.

For a queue, the Insert and Delete operations are more often called **Enqueue** and **Dequeue**.

A queue has a **head** and a **tail**. A dequeue operation removes the element of the head, and an enqueue operation adds an element at the tail of the queue.

An implementation of a queue in terms of an array is easy if we use a floating pointer to denote the head and the tail of the queue. Initially head and tail point to the beginning of the queue. If $\text{head} = \text{tail} + 1$, the queue is full. When they are equal the queue is empty. Head points to the first element and tail next to the last element (i.e. the first available position). When tail or head reaches n (size of array) the value is rolled over to 1, that is, we visualize the array as a circular one.

```
EmptyStack(S)          PUSH(S,x)          POP(S)
1. if top(S)==0         top(S)=top(S)+1;    if EmptyStack(S)
2. return(TRUE);        S[top(S)] = x;      return(UNDERFLOW);
3. else                 else
4. return(FALSE);        top(S)=top(S)-1;
5.                      return(S[top(S)+1]);

1. Enqueue(Q,x)         Dequeue(Q)
2. Q[tail(Q)]=x;         x=Q[head(Q)];
3. if (tail(Q) == length(Q) if head(Q)==length(Q)
4.   tail(Q)=1;          head(Q)=1;
5. else                  else
6.   tail(Q)++;           head(Q)++;
```

Implementing a doubly linked list

Overview

If we have say a doubly-linked list with two pointers next, prev, and key, we can represent such a list by three arrays one for each of next, prev, and key. A pointer value then becomes an index value of the array.

Sometimes it makes sense to represent a list in a single array. Then, an element of the list will occupy space m which is large enough to store prev, next, and key (for the first two an array index is stored for each prev, next). A pointer points to the first position of size m allocated to the pointed linked-list element.

As a linked-list is a dynamic data structure in both representations elements are added to and deleted from the list. If the arrays/array are/is of size n a naive approach of implementing linked-list operations on these structures may cause problems after the first n operations.

We need to somehow record the freed positions (after a delete operation) and reallocate these positions whenever an Insert is issued. The **garbage-collector** is responsible for such allocation/deallocation operations.

To allow this we keep in the **free list** all available memory (ie it is a linked list). Initially the whole memory belongs to the free list. Whenever a portion of the array is deleted it is added to the top of the free list and whenever an element needs to be allocated it is allocated from the top (ie it is a stack). Every element is either in the list L or in the free store.

This is one reason why it is advised that a malloc(A), malloc(B) in the C programming language be followed by a free(B) and free(A). **Why is it so?**

The methods we used to represent lists extend to representing any homogeneous data structure including trees. We can represent a binary tree by using three fields, **left** and **right** to point to the children of the node/element of the tree and p to point to its **parent**. An element with both left and right NULL is a leaf. An element with p NULL is the root pointed to by $root(T)$.

Suppose that a tree node has unlimited degree (number of children). Then $left(x)$ points to the left child y_1 of x and $right(y_1)$ points to its sibling y_2 (second child of x) and so on until the last one y_k points to nil (no more siblings).

Implementing a doubly linked list

Example

```

----- <----- tail
head----> | Tom |---->| Jerry |--> NULL
          |   |   |   |
          NULL <--|   |<---|   |
          -----

```

NUL not NULL delimited string
 NUL is a 'character' whose
 all bit 0 i.e. 8 bit 0 in ASCII

```

10 11 12 13 14 15 16          20 21 22 23 24 25 26 27 28 29
-----
|13|24|-1|-1|20|10|          T o m 0 J e r r y 0
-----

```

-----v----- v-----
 Jerry Tom

```

          a : previous pointer (to first of
          3 addresses)
          -----
          |a |b |c | b : pointer to Name
          ----- c : next pointer (to first of
          -1 : NULL          3 addresses)

```

DRAFT

Formal definitions of binary trees are in Subject 3. In the mean time we describe some binary tree traversals. A **tree traversal** is a process of visiting each of the vertices of a rooted tree exactly once. For rooted trees preorder and postorder tree traversals are defined. For the specific case of a **binary** tree three such traversals are quite known: **inorder**, **postorder**, **preorder**. In preorder, parents are visited before their children; in postorder after their children; for binary trees, left children are visited before right children. Inorder traversal has a meaning only for binary trees: left subtree of a parent is visited before the parent followed by right subtree. As a rooted tree may or may not be ordered, inorder is not well-defined/meaningful.

```
pre-visit(u)   :: {pre[u]=pre++; print(<u,pre[u]>);} ;
in-visit(u)    :: {in[u]=in++; print(<u,in[u]>);} ;
post-visit(u)  :: {post[u]=post++; print(<u,post[u]>);} ;
```

BT-Inorder(u)	BT-Preorder(u)	BT-Postorder(u)
1. if u != NULL {	if u != NULL {	if u!= NULL {
2. BT-Inorder (left(u));	pre-visit(u);	BT-Postorder(left(u));
3. in-visit(u) ;	BT-Preorder(left(u));	BT-Postorder(right(u));
4. BT-Inorder (right(u));	BT-Preorder(right(u));	post-visit(u);
}	}	}

<pre> 10 / \ 20 30 / 40 </pre>	<pre> Pre: 10 20 30 40 In: 20 10 40 30 Post: 20 40 30 10 </pre>
---	--

Tree traversals

BFO and DFO

BFO (Bread-first Order) traversal. Another useful traversal is the **breadth-first order** (BFO) traversal obtained as follows.

```
RT-BFO(root(RT)) // RT is a rooted (not necessarily binary) tree
1. visit(root(RT)); // visit[root(RT)]=i++; print(visit[root(RT)]);
2. repeat until all vertices are visited
3. visit an unvisited child of the LEAST RECENTLY VISITED
   vertex with an unvisited child
END_BFO
```

If **LEAST RECENTLY** is replaced by **MOST RECENTLY**, a **depth-first order** (DFO) traversal is obtained.

We can extend these two traversals from rooted trees to graphs. If G is a graph and s is an arbitrary starting vertex, we use this vertex as the “root” of the traversal by visiting s first and then repeating the following step until there is no unexamined edge (u, v) such that vertex u has been visited.

Search Step. Select an unexamined edge (u, v) such that u has been visited and examine this edge, visiting vertex v if v has not been visited yet.

DFO (Depth-First Order) traversal. If u is the **MOST RECENTLY VISITED** vertex we get a **DEPTH-FIRST** order. In **BREADTH-FIRST ORDER** traversal v is the least recently visited vertex.

<pre> 10 / \ 20 30 / \ 40 60 / 50 </pre>	<pre> DFO: 10, 20, 40, 50, 60, 30 BFO: 10, 20 30, 40, 60, 50 </pre>	<pre> DFO stack BFO queue 30 ----- 60 10 20 30 40 60 50 50 ----- 40 20 Tie Breaker : Smallest lable wins 10 ----- </pre>
--	---	---

Notice

Problem Set with Solutions

These Problem Sets are not for credit. They are provided as are for your practice. Several problems are provided with solutions. These can be discussed with the instructor after a student has studied the relevant material. Some other problems are provided with either no solutions at all or a hint of a solution. They are there for your reference and will not be discussed further. For some solutions provided you might have to fill in details related to calculations or intermediate steps.

Before you try these problems: (a) Study the designated textbook, (b) use the Subject notes (typewritten notes available through the course web-page from the password-protected section C) to review your study of the designated textbook material, (c) resolve the examples done in class, (d) then practice with some of these problems PS (Problem Set), and then if you have time, (e) solve some of the problems of the designated textbook, and (f) solve more problems from the PS especially the ones with minimal solution information.

Several PS problems appeared in past exams or homeworks.

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 1.

(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 sandwiched between an A , and a B of part (a), for some integer value m . Thus

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

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 $m = \lfloor \lg L \rfloor + 1 = \lceil \lg(L+1) \rceil$. ■

Problem Set Problem 2.

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.

$$\begin{aligned} f(91) &=_{\text{Condition B}} f(f(91 + 11)) \\ &= f(f(102)) \\ &=_{\text{Condition A}} f(102 - 10) \\ &= f(92) \end{aligned}$$

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

$$\begin{aligned} f(92) &=_{\text{Condition B}} f(f(92 + 11)) \\ &= f(f(103)) \\ &=_{\text{Condition A}} f(103 - 10) \\ &= f(93) \end{aligned}$$

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 $f(91) = f(92) = \dots = f(100)$. We thus find now $f(100)$.

$$\begin{aligned} f(100) &=_{\text{Condition B}} f(f(100 + 11)) \\ &= f(f(111)) \\ &=_{\text{Condition A}} f(111 - 10) \\ &= f(101) \\ &=_{\text{Condition A}} 101 - 10 \\ &= 91 \end{aligned}$$

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

Problem Set

Problem 3.

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

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

Proof. The sum

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

can be rewritten as $\sum_{i=0}^n i^2 + \sum_{i=0}^n i^3$. The second sum is given by Document 2 $\sum_{i=0}^n i^3 = n^2(n+1)^2/4$. For the first sum, we use the technique outlined in Document 2. 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!

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

From Document 2 i.e. obtain that $S_1 = \sum_{i=0}^n i = n(n+1)/2$. Similarly to Document 2 we use the identity

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

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

$$(1+1)^3 = 1^3 + 3 \cdot 1^2 + 3 \cdot 1 + 1$$

$$(2+1)^3 = 2^3 + 3 \cdot 2^2 + 3 \cdot 2 + 1$$

$$(3+1)^3 = 3^3 + 3 \cdot 3^2 + 3 \cdot 3 + 1$$

$$\dots = \dots$$

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

Problem Set continued

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. $(i + 1)^3$ of the second equality cancels out in the sum with i^3 of third equality).

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

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

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

■

Problem Set

Problem 4.

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.

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

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 substituting 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 $Q(k-1) \implies Q(k)$. $Q(k-1)$ 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 $Q(k-1)$.

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

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 $P(n)$. ■

Problem Set
Problem 5.

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

Proof. We use the derivation shown in class, and also available in the appendix of Subject 5, or in Document 2.

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

or

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

From $\sum_{i=0}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2}$, 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 are ok since $0 < x < 1$. Result proven. ■

Problem Set
Problem 6.

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! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 7.

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

- (a) What is the value of `sum` at the completion of the algorithm? Why? Express your answer as a function of n .
- (b) What is the worst-case running time of X ? What is the best-case running time of X ? Explain. Use asymptotic notation to express your answers.

Proof. The outer loop is incremented for values of i such that $i = 1, 2, 4, \dots, n$, i.e. $\lg n + 1$ times and so many times is the loop of lines 5-6 executed. The test $i \leq n$ succeeds $\lg n + 1$ times for $i = 1, \dots, n$ and fails for $i = 2n$.

- (a) For every such successful i the inner loop is executed a FIXED number of n times. Therefore at the completion of the loop the value of `sum` is given by

$$n \cdot (1 + 1 + 1 + \dots + 1) = n \cdot (\lg n + 1).$$

The value of `sum` is therefore $n \lg n + n$.

- (b) The loops of lines 4-6 are fixed. Therefore best-case and worst-case running time are the same. The running time is $O(n \lg n)$ as line 6 is executed (exactly) $n \lg n$ times for any n , and all other contributions are $\Theta(1)$. Therefore best and worst-case running time are $O(n \lg n)$.

■

Problem Set

Problem 8.

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;
```

■

Problem Set
Problem 9.

- (a) What is the number of key comparisons of Insertion-Sort on the n -key sequence $\langle 10, 10, \dots, 10, 10, 9 \rangle$, which are all 10 but the last one which is a 9? Justify your answer.
- (b) 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. (a) 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 $2n - 3$.

- (b) Hint: Show it is not a ... linear bound!

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 10.

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.

Proof. TRUE,FALSE,TRUE,



Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page

Problem 11.

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 decreasing input sequence $\langle n, n-1, \dots, 1 \rangle$, InsertionSort has running time that is $\Theta(n^2)$.
- (2) On the input sequence $\langle 1, 2, \dots, n \rangle$, MergeSort has running time that is $\Omega(n)$.
- (3) On the input sequence $\langle 1, 2, \dots, n \rangle$, InsertionSort is asymptotically faster than Selection-sort.

Proof. TRUE, TRUE, TRUE.

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 12.

Write pseudocode for merging two sorted sequence A, B of n keys each.

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? ■

Problem Set

Problem 13.

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

```
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);
```

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem 14.

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 $\{1, 2, \dots, n\}$ 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.

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. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 15.

The insertion sort procedure outlined in class uses a linear search to scan (backward) 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

$$C(n) = \sum_{j=1}^{n-1} O(\lg j) = O\left(\sum_{j=1}^{n-1} \lg j\right) = O(\lg 1 + \lg 2 + \dots + \lg n) = O(\lg (1 \cdot 2 \cdot \dots \cdot n)) = O(\lg n!) = O(n \lg n),$$

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).

■

Problem Set
Problem 16.

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 keys?

Proof. Let the two sequences to be merged be $A = \langle a_1, \dots, a_n \rangle$ and $B = \langle b_1, \dots, b_n \rangle$. The worst possible case is when $a_1 < b_1 < a_2 < b_2 < \dots < a_n < b_n$. 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. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 17.

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);
}
```

■

Problem Set Problem 18.

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);
    }
}
```

Problem Set Continued

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 $O(n)$. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 19.

Which of the two is asymptotically larger? $2^{n/3}$ or $2^{(n/2)}$? Explain.

Proof. It can wait until the next Problem Set or not. It is the obvious one!

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 20.

You are given n keys each one of which has a key value A,B,C, or D (say A means freshman, 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 syntactic correctness of 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)
```

■

Problem Set Problem 21.

Challenging Problem. Analyze the binary entropy function $f(x) = -x \lg x - (1-x) \lg (1-x)$, with $0 < x < 1$. Find $\lim_{x \rightarrow 0} f(x)$, $\lim_{x \rightarrow 1} f(x)$, $f(1/2)$, and extrema.

Then 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 $F(n/2) \geq n \lg n - n$?

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 $\lim_{x \rightarrow 0} g(x)$?

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

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

$$\lim_{x \rightarrow 0} g(x) = \lim_{x \rightarrow 0} x \lg x = \lim_{y \rightarrow \infty} (1/y) \lg (1/y) = \lim_{y \rightarrow \infty} \frac{-\lg y}{y} \stackrel{L'Hospital}{=} \lim_{y \rightarrow \infty} \frac{-(\lg y)'}{(y)'} = \lim_{y \rightarrow \infty} \frac{-1/y}{1} = 0^-$$

Thus the limit $\lim_{x \rightarrow 0} g(x) = 0$ 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 $\lim_{x \rightarrow 1} g(x) = 0^-$ as well by STEP 1. Therefore for $f(x) = -g(x)$ we have $\lim_{x \rightarrow 1} f(x) = 0^+ = \lim_{x \rightarrow 0} f(x)$.

Problem Set Continued

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

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

■

Problem Set Problem 22.

Before you solve this problem understand carefully the semantics of the operations. In particular in $\text{Delete}(S, z)$ 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 $\text{Insert}(S, x)$.
 - (b) Deletion of a record of S pointed by pointer z , i.e. operation $\text{Delete}(S, z)$,
 - (c) Search of a key x , i.e. operation $\text{Search}(S, x)$,
 - (d) FindMin of S i.e. operation $\text{FindMin}(S)$,
 - (e) ExtractMax of S i.e. operation $\text{ExtractMax}(S)$,
- 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. PROOF CHECK...**

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), $O(n)$, $O(n)$, $O(n)$, $O(n)$. 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. ■

Problem Set

Problem 23.

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! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web page

Problem Set Problem 24.

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. We can not use the running times of InsertionSort or MergeSort because they do not provide exact running time bounds, only asymptotic. So $O(4^2)$ or $O(4 \lg 4)$ can be 5 or 6 or whatever other constant you like them to be. Moreover the worst-case comparison bound of InsertionSort is $n(n-1)/2$ which means $4(4-1)/2 = 6$. The idea 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 as all the first $i-1$ keys are already sorted among themselves. Note that we save on comparisons but not in running time. You still need to do the (right) swaps and thus create space for the i -th input key.

Let's say our keys are a, b, c, d and are all distinct. We can remove this assumption (distinctness) but let's avoid headaches for the time being. (The sorting of the first three keys can also be done by way of InsertionSort and this shown below.)

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 binary search/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. We will prove later in class that no matter what we do and what algorithm we invent, sorting 4 keys in the worst-case requires at least 5 comparisons if the only operation allowed is comparison/swapping of keys. How do 4 (number of keys) and 5 (number of comparisons) compare? Well 5 is the smallest integer greater than $\lg(4!)$.

■

Problem Set
Problem 25.

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. But then that trick over there cannot be used. If that trick was to reinvent InsertionSort, here reinvent something else!



Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 26.

What's the order of growth of the sum $\sum_{i=0}^n x^i$ for $x > 1$.

Proof. If $x > 1$ sum is $\sum_{i=0}^n x^i = (x^{n+1} - 1)/(x - 1) = \Theta(x^n)$.

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page