How to design algorithms efficiently

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2012 ; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS 667-101 : Spring 2012 . All rights reserved.

Informal Definition: *Algorithm* is a "computational recipe" that transforms an input into an output.

Formal Definition: An **algorithm** is a well-defined (e.g. correct) sequence of computational steps that takes some value or set of values as input and generates some value or set of values as an output.

An algorithm is thus a method/tool for solving (usually) well-specified computational problems.

An algorithm is **correct** if **for every input instance** it halts with the correct output.

A (computational) *problem* defines an input-output relationship.

An **algorithm** describes a specific procedure for achieving this relationship, i.e. for each problem we may have more than one algorithms.

In this course we will be interested in **designing efficient algorithms**, i.e. algorithms whose performance is as good as we can get and can not be improved further.

For example, for the sorting problem we wouldn't be interested in bubble-sort, selection-sort as such algorithms are inefficient in terms of worst-case performance in general.

TYPES

There are many types of algorithms

- Software vs Hardware algorithms.
- Sequential vs Parallel algorithms.
- Deterministic vs Randomized vs Probabilistic algorithms.

Most of the algorithms that you have examined in other classes are **deterministic**. If an algorithm has access to a random number generator, and uses it to make decisions then it becomes a **randomized** algorithm. For example, ordinary quicksort is a deterministic algorithm; if however the splitter key is chosen at random to be one of the input keys, then this variant of quicksort becomes a randomized algorithm.

There are some problems like primality testing for which very fast **probabilistic** algorithms exist; such algorithms may "lie" in few cases, i.e. they may characterize a number as prime even if it is not.

(c) Copyright A. Gerbessiotis. All rights reserved.

Topics to discuss

Problems that will be discussed and examined in this course include

- Data compression
- Algebra (e.g. expression evaluation)
- Randomization
- Matrix operations
- Sorting networks and parallelism
- Parallel Algorithms
- Arithmetic problems
- Number Theory
- Cryptography
- String matching
- Computational Geometry
- Web-page Ranking algorithms (Kleinberg, Google's PageRank)

(c) Copyright A. Gerbessiotis. All rights reserved.

Random Access Machine: A simplistic model

The computer model is as simplistic/optimistic (in resource usage) as possible. Our intention is to show that if an algorithm requires A computation resources under such a model it would require at least A computation resources under more realistic models.

What do we measure?

Resources that are required by an algorithm may include **space** (memory), communication bandwidth, logic gates etc. The two most important resources are **time** (how fast an algorithm is run) and **space** (how much memory/space it uses). As space tends to become quite inexpensive, **time** becomes the resource whose usage needs to be minimized.

What is the size of the input? What is problem size?

It depends on the application. For sorting it is the size of the input sequence n. For the multiplication of two integers a and b, it is not the number of integers (two) but the length in bits of the two integers. This is also problem size. So for such cases problem size and input size are the same and can be used interchangeably.

For matrix-related operations involving say matrix A of geometry $n \times m$, where n is the number of rows and m the number of columns, one could say that n, m are problem size (or say $\max(n, m)$, but the size of the input is the number of elements of A i.e. $n \cdot m$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Incremental. The **incremental approach** to designing algorithms involves the building of the output from the input one element at a time.

Examples.
Sorting algorithms (insertion, selection) are examples of using the incremental approach.

Divide-and-conquer. In the design technique called divide-and-conquer a large problem is split into smaller subproblems, the smaller-size subproblems are then solved (say, recursively or using the same algorithm) and then the solutions of the subproblems are combined to give the solution of the original problem.

Examples.
Sorting algorithms such as merge-sort, quick-sort are examples of using the divide-and-conquer method. In quick-sort the combining phase does nothing, where as in merge-sort it involves the merging of sorted sequences. On the other hand, in quick-sort the dividing phase is involved and requires the picking of a pivot (splitter) keys and splitting of the input keys around the pivot, whereas in merge-sort splitting is trivial: the first half of the keys fall into one set and the remaining half (plus or minus one key) fall into the second set.

The **greedy** technique applied to designing greedy algorithms has been used for example in the design of spanning tree algorithms (Prim's and Kruskal's method) and shortest path (i.e. Dijkstra's algorithm).

The **dynamic programming** technique has been used and applied to the all-pair shortest path problem and similar other problems (i.e. Floyd-Warshall algorithm for the all-pair shortest path, nested parenthesis problem, Chomsky Normal Form recognition, transitive closure problem etc).

(c) Copyright A. Gerbessiotis. All rights reserved.

Asymptotic Notation

For large input sizes we are concerned only with rate of growth of running time as it dominates low order terms or constants. We are thus interested in how running time increases with problem size **in the limit**. The best algorithm is the one with the smallest asymptotic growth. Let $\lg n$ be the base two logarithm of n. Which element of the following pairs of functions is ASYMPTOTICALLY larger?

- **Remember:** $2^{\lg n} = n$ and $a^{\lg n} = n^{\lg a}$ and $n^{1/\lg n} = 2$.
- **Notation:** $\lg^2 n = (\lg n)^2$ and $\lg^k n = (\lg n)^k$.
- **Fact:** In general $n^k > \lg^l n$ for any positive constant k, l an d large enough n.
- **Fact:** $2^n > n^k$ for any positive constant k and large enough n.
- **Fact:** $n! > n^k$ for any constant k and large enough n.
- **Stirling's formula:** $n! \approx (n/e)^n \sqrt{2\pi n}$.

(c) Copyright A. Gerbessiotis. All rights reserved.

For function $g(n)$ we denote by $\Theta(g(n))$ the **set of functions**

$$\Theta(g(n)) = \{f(n) : \exists \text{ positive constants } c_1, c_2, n_0 : \ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \ \ \forall\, n \geq n_0\}$$

If this is the case, we say that $g(n)$ **is an asymptotic tight bound** for $f(n)$, i.e. we can "sandwich $f(n)$ between two multiples of $g(n)$, for big enough n: Abusing notation, however, we would write $f(n) = \Theta(g(n))$ to denote the membership of $f(n)$ in the set $\Theta(g(n))$.

For a given function $g(n)$ we denote by $O(g(n))$, the **set of functions**

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c, n_0 : \ 0 \leq f(n) \leq cg(n) \ \ \forall\, n \geq n_0\}$$

we then write $f(n) = O(g(n))$.

For a given function $g(n)$ we denote by $\Omega(g(n))$, the **set of functions**

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c, n_0 : \ 0 \leq cg(n) \leq f(n) \ \ \forall\, n \geq n_0\}$$

We then write $f(n) = \Omega(g(n))$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Asymptotic Notation: o, ω

For a given function $g(n)$ we denote by $o(g(n))$, the **set of functions**

$$o(g(n)) = \{f(n) : \forall \text{ constant } c > 0 \ \ \exists \text{ constant } n_0 > 0 : \quad 0 \leq f(n) \leq cg(n), \ \ \forall\, n \geq n_0\}$$

or equivalently

$$o(g(n)) = \{f(n) : \ \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \ \ \}$$

For a member $f(n)$ of $o(g(n))$, we express this membership by writing $f(n) = o(g(n))$.
For a given function $g(n)$ we denote by $\omega(g(n))$, the **set of functions**

$$\omega(g(n)) = \{f(n) : \forall \text{ constant } c > 0 \ \ \exists \text{ constant } n_0 > 0 : \quad 0 \leq cg(n) \leq f(n), \ \forall\, n \geq n_0\}$$

or equivalently

$$\omega(g(n)) = \{f(n) : \ \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \ \ \}$$

For a member $f(n)$ of $\omega(g(n))$, we express this membership by writing $f(n) = \omega(g(n))$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Recurrence Solution Methods

When an algorithm like mergesort is recursive, its running time is described by a recurrence. A recurrence is an equation that expresses a function in terms of the same function but on smaller values. For the case of merge sort, we derived the following recurrence for running time.

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

In order to solve this recurrence we need a boundary condition for the base case. The base case for the running time of merge-sort is the running time for an input of size 1 or in general, of some small constant size.In this case, we may assume that $T(1) = \Theta(1)$, i.e. the sorting of a one-key sequence takes constant time (e.g an one-key sequence is always sorted). The solution for this recurrence was claimed to be $T(n) = \Theta(n \lg n)$,

We have **three methods** to solve recurrences.

- **Substitution or "guess and check"** method: Guess and verify/check (proof by induction).
- **RecursionTree/Iteration** method: Unfold recurrence by turning it into a sum.
- **Master** method: Provides a solution formula for recurrences of certain form such as $T(n) = aT(n/b) + f(n)$, for constant $a \geq 1$ and $b > 1$ and asymptotically positive $f(n)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Master Method

The master method provides an easy solution method for a certain class of recurrences that are of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$ and $b > 1$ are constant and $f(n)$ is an asymptotically positive function.

Note that n/b can be either a floor(n/b) or a ceiling(n/b).

Theorem. For such a recurrence, $T(n)$ is bounded as follows.

1 If $f(n) = O(n^{\lg_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\lg_b a})$.

2 If $f(n) = \Theta(n^{\lg_b a})$, then $T(n) = \Theta(n^{\lg_b a} \lg n)$.

3 If $f(n) = \Omega(n^{\lg_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and for large n, then $T(n) = \Theta(f(n))$.

(c) Copyright A. Gerbessiotis. All rights reserved.

In this section we prove that no matter what algorithm is to be used for comparison and exchange sorting, the worst-case running time of the algorithm will be $\Omega(n \lg n)$ **for some input instance**. In order to prove this bound we shall introduce a tree called the decision tree.

Sorting Problem

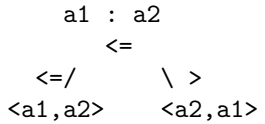
Input. A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output. A reordering (i.e. permutation) $\langle a_{j_1}, a_{j_2}, \ldots, a_{j_n} \rangle$ of the input sequence so that $a_{j_1} \leq a_{j_2} \leq \ldots \leq a_{j_n}$.

Example.
Input: $\langle 10\ 2\ 1\ 5 \rangle$
Output: $\langle 1\ 2\ 5\ 10 \rangle$
An input sequence like the $\langle\ 10\ 2\ 1\ 5\ \rangle$ is an **instance** of the sorting problem. An instance of the problem consists of all the inputs needed to compute a solution to the problem.

Sorting algorithms that we have examined before such as bubble-sort, insertion-sort, merge-sort, quick-sort, heap-sort, selection-sort are **comparison** and **exchange/swap** based. This means that in order to sort keys, a pair of input keys are compared and depending on the outcome of the comparison they may or may not be exchanged. (An interesting question is what happens if the two keys are equal: swap or not.)

Suppose we have two keys $\langle a_1, a_2 \rangle$ to sort. The problem is easy. In one comparison denoted by $a_1 : a_2$ we decide the smaller of the two keys. Sometimes when we compare we ask questions $\leq$ or $\geq$ or $<$ or $>$ or $==$. We need to standardize our questioning, so we will ask $\leq$. The outcomes are shown below. When we compare a_1 and a_2 if $\leq$ applies i.e. $a_1 \leq a_2$ the left branch is followed and this implies an output $\langle a_1, a_2 \rangle$; for the other case $a_1 > a2$ the right branch is followed and it implies an output $\langle a_2, a_1 \rangle$. The leaves of the tree is the number of possible output for this sorting problem i.e. the number of permutations 2!.



Without loss of generality we may assume that the keys are distinct and that the comparison performed is not of the type $\leq$ but just $<$. (We can make keys distinct by creating an extended key: thus key k extends to key (k, i) where i is the index of key k in the input sequence and thus $\langle a_1, a_2 \rangle$ would become $\langle (a_1, 1), (a_2, 2) \rangle$.) Therefore $=$ tests can be avoided, and the remaining operations are equivalent ie. we only need $\leq$ based comparisons to draw conclusions.

The decision tree model

For an input of given size n, a **decision tree** is a full binary tree that represents the comparisons between keys performed by a particular sorting algorithm; control, data exchanges and other details of the algorithm are ignored. The decision tree is thus a binary tree whose internal nodes are pairs $a_i : a_j$ for some $i, j \leq n$. Each leaf is a permutation of the input indices (not keys) $\langle p(1), p(2), \ldots, p(n) \rangle$.

The execution of the sorting algorithm on some input of size n is a path from the root of the decision tree to a leaf. The path shows the comparisons that were performed between any two keys.

If at some node a left path was taken this means that $a_i \leq a_j$ was decided, otherwise $a_i > a_j$.

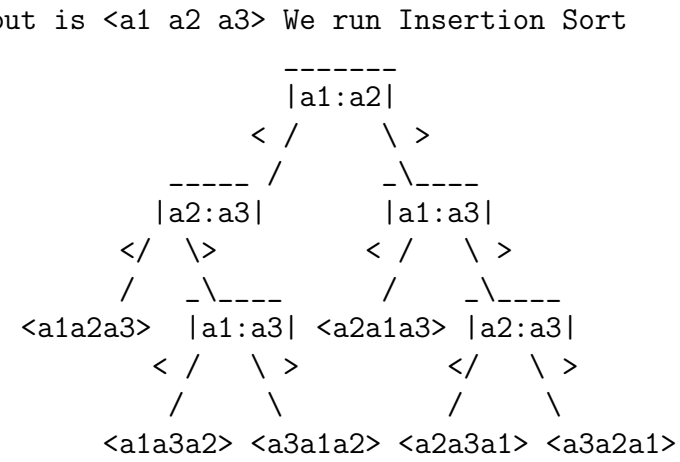
When a leaf is reached, this means that the output sequence has been decided and the sorting algorithm has established say the ordering $\langle a_{p(1)}, a_{p(2)}, \ldots, a_{p(n)} \rangle$ for the input (we reestablish the output sequence from the output permutation).

As the number of permutations on n inputs is $n!$ and each permutation must appear as a leaf (provided that the sorting algorithm is correct i.e. it DOES sort) of the decision tree we conclude that any decision tree for sorting SHOULD have at least $n!$ leaves.

Note. For the case of binary search for example on an array of n keys a decision tree would have at least $n + 1$ leaves, n for each one of the n indexes that indicate the position in the array of the searched keys, and the $n + 1$-st would account for the 'not-found' case.

Note. Consider weighing 2 coins to determine a lighter fake coin. The decision tree would have two leaves for each coin can be a fake. Yet if it is not known whether the fake coin is lighter or heavier, there would be 4 possible leaves for each case of the two coins and the two cases of a give coing being lighter or heavier.

A three-key decision tree for InsSort



All 3!=6 permutations of the input sequence appear as leaves.
If <a1 a2 a3> is the reached leaf it means that in the input
<a1 a2 a3> we have that a1 < a2 < a3
If however <a3 a2 a1> is the reached leaf it means that in the
input a3 < a2 < a1

(c) Copyright A. Gerbessiotis. All rights reserved.

Worst Case bound

As a path from the root to a leaf gives a computation trace of the algorithm for some input, the longest path represents the worst case number of comparison steps required to sort an input sequence by that number of steps. (In addition the path provides us with the worst-case input as well).

Therefore the worst case bound for a comparison based algorithm is the height of its decision tree. We are going to establish a lower bound on the height of decision trees of at least $n!$ leaves to derive a lower bound for comparison based sorting algorithms.

Theorem. Any decision tree that sorts n keys (using comparisons) has height $\Omega(n \lg n)$.

Proof. A decision tree that sorts n keys must have at least $n!$ leaves, since n (distinct) keys generate $n!$ permutations representing a distinct sorted sequence. A binary tree of height h has at most 2^h leaves. Therefore we need that $2^h \geq n!$ ie $h \geq \lg(n!)$. From Stirling's formula for the factorial (2.11), we get $n! \geq (n/e)^n$. This implies that $h \geq n \lg n - n \lg e = \Omega(n \lg n)$.

Corollary. Heapsort and mergesort are asymptotically optimal.

(c) Copyright A. Gerbessiotis. All rights reserved.

Adversarial Arguments

Game. Play a guessing game between A (Alice) and B (Bob). Alice picks a number and Bob can ask questions trying to figure out the number Alice guessed. (Alice is not allowed to cheat; Alice does not need to pick the number in advance as long as she gives correct/consistent answers.)

Objective. Alice's objective is to force Bob to ask as many questions as possible. Bob's objective is to find the answer in as few questions as possible.

Alice's methodology. Construct the number based on the questions asked by Bob. That is, Alice could generate a worst-case number/input, that is consistent with all of Bob's questions. If alice can force at least $f(n)$ questions for a number in the range $1 \ldots n$, then $f(n)$ is the lower bound!

(c) Copyright A. Gerbessiotis. CS 667-101 : Spring 2012 . All rights reserved.

Most of the algorithms that one has been dealing with are deterministic algorithm, algorithms that make fixed choices dependent or non-dependent on the input. There is also another class of algorithms whose decisions rely on the existence of a random number generator. The random number generator generates random numbers and depending on the value of a drawn number a decision is made or an action is taken. Such algorithms are classified as **randomized algorithms**.

Sometimes the differences between a deterministic and a randomized algorithms are so small that one may think that a randomized algorithm is indeed a deterministic one. Consider one of the most celebrated algorithms in computer science, quicksort.

How quicksort works A sequence S of n keys $x_1, \ldots, x_n$ is to be sorted. The algorithm picks a splitter key among the n keys and let us call this key s. The splitter splits the input sequence into two sets S_1 and S_2: S_1 contains all the keys smaller than s and S_2 all the keys greater than x. Then the same splitting process (splitter selection, and splitting) is repeated recursively for S_1 and S_2.

Deterministic quicksort The splitter is $s = x_1$ (first key), $s = x_n$ (last key), $s = x_{\frac{1+n}{2}}$ (middle key): such choices makes quicksort a deterministic algorithm.

Randomized quicksort. The splitter is $s = x_{random(1,n)}$, where an index between 1 and n is chosen uniformly at random by random(1,n). Such and other sampling-based methods give rise to randomized versions of quicksort. Random sampling can be generalized in quicksort to give a more fine-balanced variant where S_1 and S_2 are of about the same size, with high confidence. In such a context, high confidence or high probability or polynomially small probability of failure means that the probability that the claim is true is at least $1 - 1/n^c$, for some constant $c > 0$, or equivalently the probability of failure (i.e. claim is not true) is at most $1/n^c$. The "polynomially" sometimes is strengthened by claiming "exponentially" small probability of failure i.e. $1/2^{cn}$, where c is a constant.

In brief, **random sampling** the concept that a random sample from a population is representative of the population as a whole is a popular theme in randomized algorithms.

In some other cases an algorithm is required to determine whether an input (say a number x) has a certain property (for example, "is x a prime/composite number?"). It does so by finding a witness that x has this property. The difficulty in finding such a witness may lie in the fact that the search space is huge to be searched exhaustively. If however the space contains many witnesses (say, a quarter), then it often suffices to choose an element of that space at random to find a witness. Number theory applications (including primality testing already examined in the form of the Miller-Rabin algorithm) makes use of witnesses.

Fingerprinting and hashing A long string may be represented by a short fingerprint using a random mapping (cf. Rabin-Karm string matching). In some cases one may be able to prove that two strings are likely to be identical if their fingerprints are identical. This idea underlies hashing where a small set of keys drawn (identifiers of a program) from a huge universe of keys (all alphanumeric strings of length at most 64) is mapped into a smaller universe with a guarantee that distinct elements of the set are likely to be mapped to distinct elements of the smaller universe.

In **symmetry breaking** a collection of processors in a distributed system needs to break deadlock and arrive at a consensus. This is another example of a problem that can be solved through randomization.

(c) Copyright A. Gerbessiotis. All rights reserved.

Introduction

Let us reconsider the case of quicksort. If we could find a splitter s efficiently in $O(n)$ steps so that both set S_1 and S_2 have about the same size $n/2$, then the recurrence for the running time of quicksort would be

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

which leads to $T(n) = O(n \lg n)$. Note that if we are not so lucky with the splitting and we get not a $n/2 - n/2$ but say a worst-case $n/4 - 3n/4$ split all the time (i.e. in all recursive calls), the recurrence slightly changes to

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

with the same asymptotic solution as before.

The simplest answer in how to find s is to choose one at random. In the remainder we will assume that a random choice takes constant time.

This gives rise to the following quicksort variant called RandomQSort.

```
    RandomQsort(S,n)
1.  choose an element y uniformly at random from S (i.e. so that every
    element of S has the same probability of being chosen).
2.  Split S into S1 and S2 based on y.
3.  RandomQsort(S1, |S1|);
4.  RandomQsort(S2, |S2|);
```

(c) Copyright A. Gerbessiotis. All rights reserved.

Theorem RandomQsort The expected number of comparisons in an execution of RandomQsort is $2nH_n$.

Some preliminaries. The series

$$H_n = 1 + 1/2 + 1/3 + \ldots + 1/n \approx \ln n + \gamma$$

is called the harmonic series of order n. In general $H_n = O(\ln n)$.

Let $x_1, \ldots, x_n$ be the input keys of S and let $x_{(1)}, \ldots, x_{(n)}$ be the smallest, second-smallest, etc , n-smallest (i.e. largest) keys of S.

We can then define a random variable X_{ij} to assume the value of 1 if $x_{(i)}$ and $x_{(j)}$ are compared in an execution and the value 0 otherwise (if they are not).

Therefore random variable X_{ij} is an indicator of the comparison between $x_{(i)}$ and $x_{(j)}$ and the total number of comparisons (a.k.a running time) is $\sum_{i=1}^{n} \sum_{j>i} X_{ij}$. We are interested in the expected number of comparisons performed i.e.,

$$E[\sum_{i=1}^{n} \sum_{j>i} X_{ij}] = \sum_{i=1}^{n} \sum_{j>i} E[X_{ij}].$$

This equation uses an important property of **expectation** referred to as **linearity of expectation** namely that the expectation of a sum of random variables is equal to the sum of the expectations of the variables.

Let p_{ij} be the probability that $x_{(i)}$ and $x_{(j)}$ are compared. Since X_{ij} is a binary discrete variable that takes values 0 or 1 only, we have that

$$E[X_{ij}] = p_{ij} \cdot 1 + (1 - p_{ij})0 = p_{ij}$$

(c) Copyright A. Gerbessiotis. All rights reserved.

In order to find p_{ij} we visualize RandomQsort as a binary tree T each node of which is labeled with a distinct element of S.

The root of the tree is y, the first splitter to be chosen in an execution of the algorithm, the left subtree of y containing elements of S_1 and the right subtree elements of S_2, and the structures of the two subtrees defined recursively.

Since y is a splitter, elements of S_1 are compared to it and so are elements of S_2. However, no comparison occurs between elements of S_1 and elements of S_2.

Therefore a comparison between two keys $x_{(i)}$ and $x_{(j)}$ occurs **if one is an ancestor of the other in the so formed tree**.

If an in order traversal of T is initiated, the elements of S are output in sorted order : this is also the output of RandomQSort. In fact tree T is **a random binary tree**.

A **level order traversal of the nodes** of a tree is one obtained by visiting the nodes in a left-to-right top-to-bottom order: the i-th level of the tree is the set of nodes at distance i (edges) from the root.

(c) Copyright A. Gerbessiotis. All rights reserved.

Let **lot** be the level order traversal of a tree of a RandomQsort execution.

a. If $x_{(i)}$ and $x_{(j)}$ are on different subtrees of a splitter, they are never compared.

b. Thus $x_{(i)}$ and $x_{(j)}$ are compared if they are in the same subtree of a splitter and one is ancestor of the other.

Consider the output sequence $x_{(i)}, x_{(i+1)}, \ldots, x_{(j-1)}, x_{(j)}$. When do we compare $x_{(i)}$ and $x_{(j)}$?

a. If $x_{(k)}$, where $i < k < j$ appears first in the lot, then this key/splitter separates $x_{(i)}$ and $x_{(j)}$ and the keys are never compared.

b. Therefore $x_{(i)}$ or $x_{(j)}$ must appear in lot before any of their intermediate keys.

What is the probability that among $x_{(i)}, \ldots, x_{(j)}$ the first key to appear is $x_{(i)}$ or $x_{(j)}$? Since each key is equally likely to appear first, this probability is $1/(j-i+1) + 1/(j-i+1) = 2/(j-i+1)$. Therefore, $p_{ij} = 2/(j-i+1)$.

Then

$$\sum_i \sum_{j>i} p_{ij} = \sum_i \sum_{j>i} \frac{2}{j-i+1} \leq \sum_i \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2 \sum_i \sum_{k=1}^{n} \frac{1}{k} = 2 \sum_i H_n = 2nH_n$$

Since $H_n = \ln n + \Theta(1)$, we obtain that $2H_n n = 2n \ln n + \Theta(n)$.

Theorem RandomQsort The expected number of comparisons in an execution of RandomQsort is at most $2nH_n$.

When an input sequence of keys is sorted, as a byproduct of sorting we also know what the maximum and minimum keys of the sorted sequence are. They are elements $A[1]$ (minimum) and $A[n]$ (maximum) of the sorted output array $A[1..n]$.

We also know the i-th smallest key: it is $A[i]$. This is also the $n-i$ largest key (the 1-st smallest is the minimum and 1-st largest is the maximum).

The i-th smallest of a set of keys is also known as the **i-th order statistic**.

One of the interesting statistics is what is called the **median**, The **lower median or median** is the $\lfloor \frac{n+1}{2} \rfloor$ statistic. The **upper median** is the $\lceil \frac{n+1}{2} \rceil$ statistic.

The **selection problem** is defined as follows.
Input. A set S of n (distinct) keys and a number i such that $1 \leq i \leq n$.
Output The element $s \in S$ such that s is larger than $i-1$ other elements of S.

One way to solve the Selection problem is by sorting and then reading off the i-th sorted element (from the left, where leftmost is the smallest of the n elements).

(c) Copyright A. Gerbessiotis. All rights reserved.

Problem 1. Find the minimum of n keys?

Answer: An elementary upper bound on the number of comparisons is $n-1$. This is also a lower bound as all the other keys need to be read and ruled out before a conclusion on the minimum can be drawn. This can also be shown by a tournament argument where every key but the winner must lose a match (i.e. a comparison must result in this element not being the minimum).

Problem 2. Solve the MINMAX problem i.e. find both the minimum and the maximum at the same time.

Answer: A relevant question is how long it takes to find **both** the minimum and the maximum. An easy upper bound is $2n-2$ comparisons by using the previous algorithm twice. Under this algorithm every key is compared twice, once with the current minimum and once with the current maximum.

A better algorithm is the following

> Take the first two elements of the set and make the smallest of the two the current minimum and the largest the current maximum.
>
> Take the next two elements, compare them, and then compare the minimum with the current minimum and the maximum of the two with the current maximum. The total number of comparisons performed is 3 instead of 4 (2 per key).

Problem View the same problem as a divide-and-conquer problem.

Question How many comparisons do you need to find simultaneously the min and max of n keys?

Answer. At most $3n/2-2$.

Question. Suppose we have found the minimum. How many additional comparisons are required to find the second smallest key of the original n keys?

Answer. $\lceil \lg n \rceil - 1$.

or **Question.** Find the second smallest of n keys is $n + \lceil \lg n \rceil - 2$ comparisons.

(c) Copyright A. Gerbessiotis. All rights reserved.

Expected linear time algorithm

The idea of the algorithm is to use recursive partitioning (divide and conquer) just as quick sort did. The difference from quick sort is that in the recursive step only one of the two partitions needs to be further searched. This difference leads to an $\Theta(n)$ expected time algorithm rather than $\Theta(n \lg n)$.

The algorithm uses **Randomized-Partition** which is similar to **Partition** except that the splitter is chosen at random (instead of being the first element of the subsequence passed as an argument). Then **Randomized-Select** works as follows.

```
 Randomized-Select (A,l,r,i)
1. if (l=r) the return A[l]
2. m=Randomized-Partition(A,l,r);
3. k=m-l+1;// Line 3 computes the size of the left partition
4. if (i== k)  // Is statistic  A[m]?
5.  return(A[m]); //yes return it
5. elseif (i<k) //no, is it on left side?
6.   return Randomized-Select(A,l,m-1,i)  // yes it is
7. else
8.   return Randomized-Select(A,m+1,r,i-k) //no it is not
```

The worst case running time is $\Theta(n^2)$. For the average case the recurrence to solve is

$$T(n) \leq 1/n(T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k))) + O(n)$$

whose solution is $T(n) = O(n)$.

This is so because if splitter is the minimum, then the left partition is of size 1 (prob $1/n$ this happening). If otherwise rank(splitter) ≥ 2, the left set is $1, 2, \ldots n-1$ with probability $1/n$ for each case as well. Therefore probability left set is 1 is $2/n$ (two cases this happening).

(c) Copyright A. Gerbessiotis. All rights reserved.

Worst case linear time

We describe an algorithm whose worst case running time is $O(n)$. We achieve this by splitting the input set more accurately than using Randomized-Partition. The algorithm below describes procedure **Select(A,i)**.

```
   Select (A,i) // Find i-th smallest in set A of n keys.
1. Group n keys into groups of  5 keys.
2. Sort the keys in each group with insertion sort.
3. Identify the median of each group.
4. Find the median x of the medians of the groups
   recursively using Select.
5. Partition A into
   Aleft  = { a: a belongs to A and a <  x }
   Aright = { a: a belongs to A and a >= x }
6. Let |Aleft|=k and |Aright|=n-k.
7. if (i<=k)
8.    Select(Aleft,k);
9. else
10.Select(Aright,i-k);
```

1. Split the n keys into groups of 5 keys. There are ceiling($n/5$) many groups of which floor(n/5) have 5 keys and the remainder the remaining keys of the input set (less than 5).

2,3. Find the median of each group of 5 keys by sorting each group with say, insertion sort. This requires constant time $O(5^2) = O(1)$ per group and $O((n/5))$ overall, since insertion sort on 5 keys requires $O(1)$ time per group and thus $O(n)$ for all groups. The median of 5 keys is the third smallest (or largest) of the 5 keys.

4. Use Select recursively to find the median x of the ceiling(n/5) medians of step 3.

5. Partition the input keys around x. Let k be the number of elements less than x and $n - k$ the number of elements greater than or equal to x forming respectively sets Aleft and Aright.

7-10. Use Select recursively to find the i-th smallest key in Aleft if $i \leq k$ or the $(i - k)$-th smallest key in Aright otherwise.

(c) Copyright A. Gerbessiotis. All rights reserved.

The first step in the analysis of Select is the finding of the size of the set of elements which are larger than x. At least half of the medians in step 3 are greater than or equal to x. For each such median there are three elements (including the median element) out of 5 that are greater than x except for the last group (which may have fewer than 5 elements, i.e fewer than 3 greater than x) and the group containing x (which has 2 keys greater than x), plus the two elements greater than x. Therefore THE NUMBER OF KEYS GREATER THAN x is AT LEAST

$$3((1/2)\lceil n/5\rceil - 2) + 2 \geq 3n/10 - 4$$

Then, the NUMBER OF KEYS LESS THAN x is AT MOST

$$n - (3n/10 - 4) \leq 7n/10 + 4$$

Similarly, THE NUMBER OF KEYS LESS THAN x is AT LEAST $3n/10 - 4$ and NUMBER OF KEYS GREATER THAN x is AT MOST $7n/10 + 4$. Therefore in step 5 and in steps 8-10 Select is called on a set whose maximum size is at most $7n/10 + 4$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Graphical View of the keys

Assume we have sorted the medians of the groups of 5 left to right (smallest-to-largest) and also the keys within a group bottom-to-top (smallest to largest). Then the n keys look like.

```
                      -----------------------   Call the median of the medians x(=100)
       100   300   1010 |140  122  144  162
        70   200   1000 |110  121  143  161
      -----------------------                 >=x   The top   right area is the set of keys >=x
        30    60     80 \100| 120  140  160          The bottom left area is the set of keys <=x
<=x                       --/------------------     #keys (<=x) >= 3n/10 [bottom-left]
        29    10     70   /5   10    4    5          #keys (>=x) >= 3n/10 [top-right]
        19     5     20  / 1    2    3    4          HOWEVER WE WANT TO KNOW AN UPPER BOUND
     -------------------/                           ON THE NUMBER OF KEYS not the 3n/10 lower
                                                     bound.
```

If we use [bottom-left] we establish the upper-bound for the [top-right] area
Use #keys (<=x) + # keys (>x) =n i.e. #keys (>x) = n - #keys(<=x) <= n - 3n/10 = 7n/10

If we use [top-right]   we establish the upper-bound for the [bottom-left] area
Use #keys (<=x) + # keys (>x) =n i.e. #keys (<x) = n - #keys(>=x) <= n - 3n/10 = 7n/10

(c) Copyright A. Gerbessiotis. All rights reserved.

Performance Analysis

We then analyze the performance of Algorithm Select in detail.

Steps 1,2 and 3 require $O(n)$ steps. Step 4 takes $T(\lceil(\rceil n/5))$ steps and step 5-10 requires $T(7n/10+4)$ steps. Therefore If $n > 100$,

$$T(n) = T(n/5+1) + T(7n/10+4) + n \quad , \quad n \geq 100$$

else if $n < 100$, $T(n) = \Theta(1)$ i.e. it takes constant time to find the median or other statistic of a constant number (up to 100) number of keys. We solve the recurrence using the subistitution method. Let us guess $T(n) \leq cn$ for $n \geq 100$. Then

$$\begin{aligned} T(n) &= T(n/5+1) + T(7n/10+4) + n \leq c(n/5+1) + c(7n/10+4) + n \\ &= 9cn/10 + 5c + n \end{aligned}$$

We observe that $5c + n \leq cn/10$ for say $c = 20$ as long as $n \geq 100$. Large values of c are OK as well. Therefore, for $n \geq 100$

$$T(n) \leq 9cn/10 + 5c + n \leq 20n$$

which implies that $T(n) = O(n)$, as needed.

(c) Copyright A. Gerbessiotis. All rights reserved.