Advanced Algorithms

Dr. Malek Mouhoub

Mathematics and Computer Science Department

University of Lethbridge

Spring 2001

3. Sorting and Order-Statistics

1. Introduction
2. Sorting methods & analysis
3. A general lower bound for sorting
4. External Sorting
5. Order statistics

1. Introduction

The **sorting problem** consists in the following :

Input : a sequence of n elements $(a_1, a_2, \ldots, a_n)$.

Output : a permutation $(a'_1, a'_2, \ldots, a'_n)$ of the initial sequence, sorted given an ordering relation $\leq$: $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example :

(8,1,6,3,6,4) → Sorting Algorithm → (1,3,4,6,6,8)

2. Sorting methods & analysis

Insertion sort : $\Theta(n^2)$ in the worst case.

Heapsort : $\Theta(n \log\ n)$ in the worst case.

Devide and Conquer algorithms :

Mergesort : $\Theta(n \log\ n)$ but don't sort in place.

Quicksort : $\Theta(n^2)$ in the worst case but $\Theta(n \log\ n)$ in the average case.

When extra information are available

- Bucketsort : elements are positive integers smaller than m : $O(m + n)$

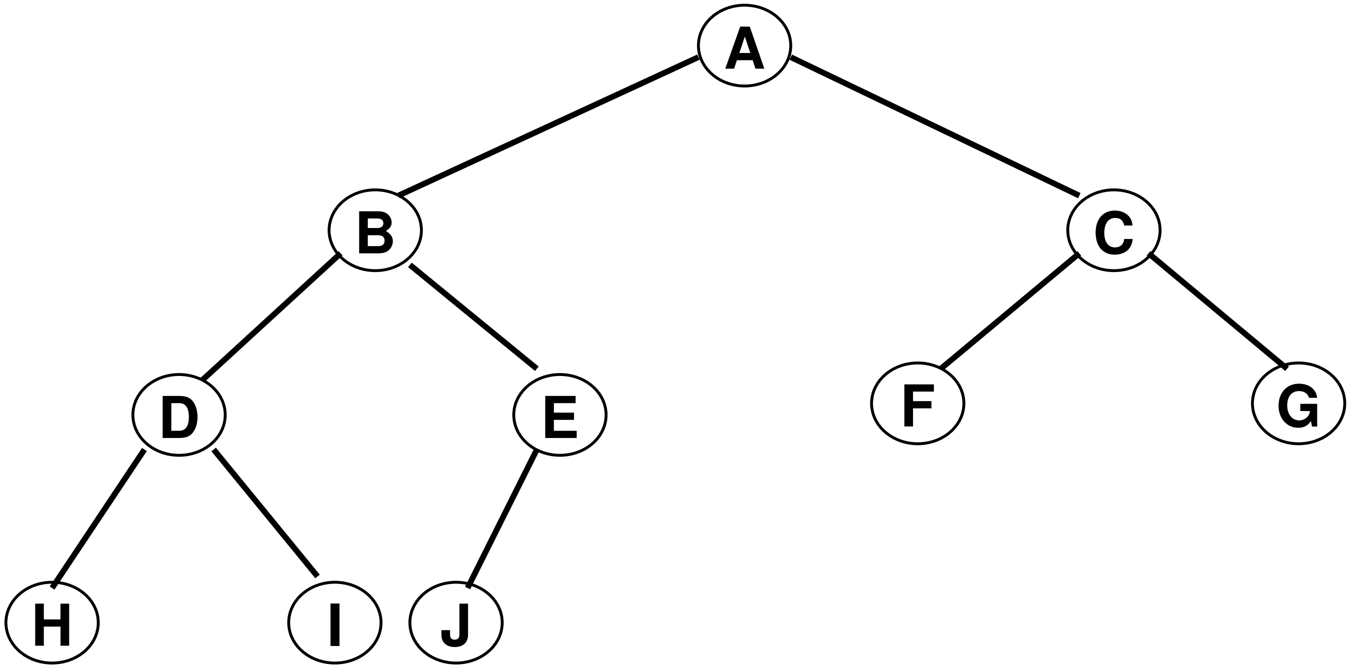
Insertion Sort

- Efficient for a small number of values.
- The intuition behind this algorithm is the principle used by the card players to sort a hand of cards (in the Bridge or Tarot).
  - *We generally start with an empty left hand and at each time we take a card, we try to place it at the good position by comparing it with the other cards.*
- Consists of $N-1$ passes. For each pass $p\,(1 \leq p \leq N-1)$ insertion sort ensures that the elements in position 0 through p are in sorted order.
- Best case : presorted elements. $\Theta(N)$
- Worst case : elements in reverse order. $\Theta(N^2)$

Heapsort

Binary heap

- A binary heap is a complete binary tree : a tree that is completely filled, with possible exception of the bottom level, which is filled from left to right.
- Can easily be represented by an array (and an integer representing the current heap size) instead of a linked list.
- For any element in array position i :
  - the left child is in position $2i$,
  - the right child is in position $2i + 1$,
  - and the parent is in position $[i/2]$.
- The operations required to traverse the tree are very simple and are performed quickly (because of the heap-order property).

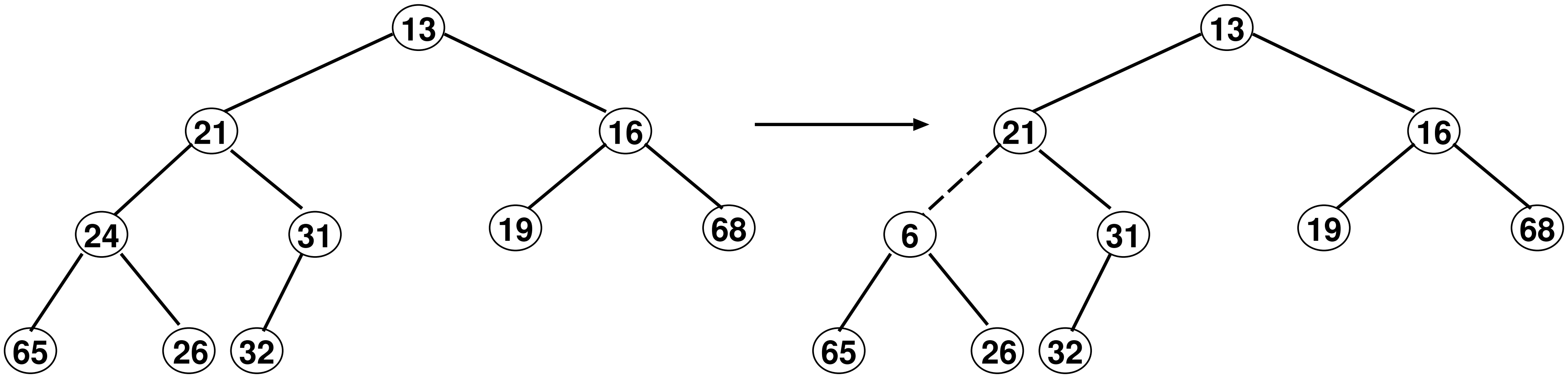


	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap-Order Property

- The smallest element should be at the root.
- Any node should be smaller than all of its descendants (any subtree should also be a heap).
- **Heap-order property** :

  For every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the exception of the root (which has no parent).



Two complete trees (only the left is a heap)

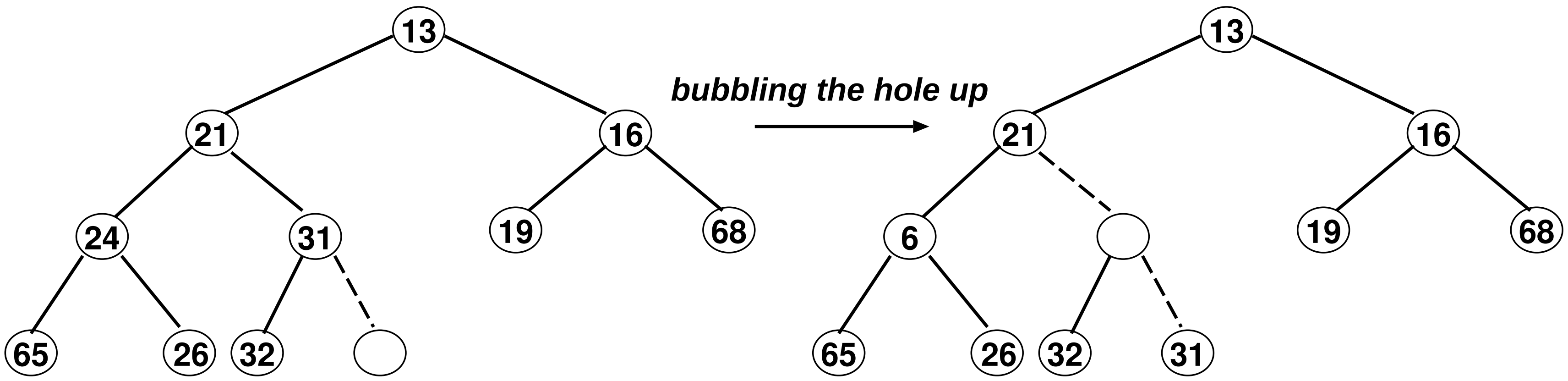
Basic Heap Operations

- **Insert (perlocate up):** $\Theta(\log n)$ in the worst case.
- **DeleteMin (perlocate down):** $\Theta(\log n)$ in the average and worst case.

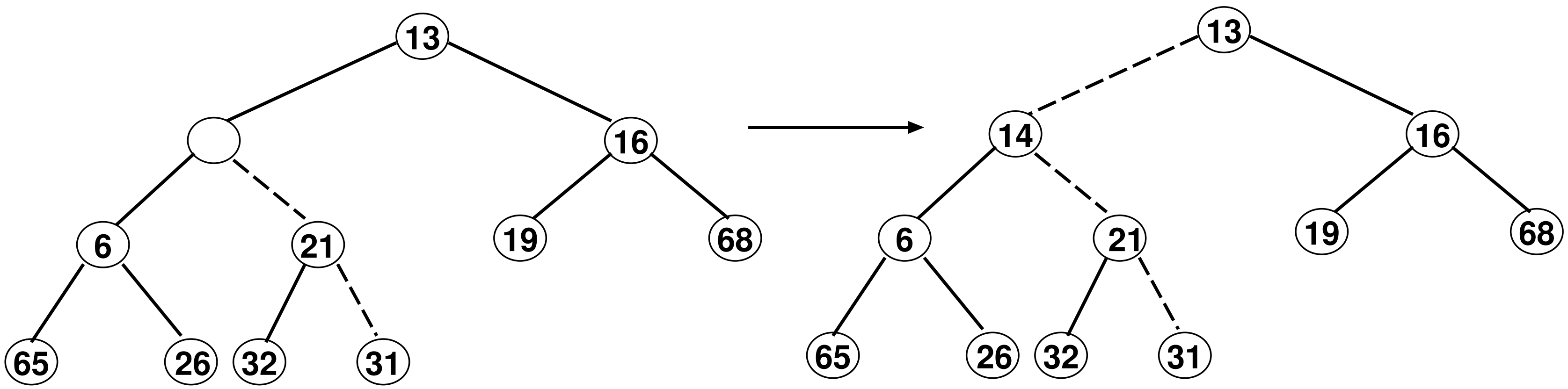
Basic Heap Operations

Insert

1. Create a hole in the next available location.
2. Place the element to be inserted in the hole if it does not violate the heap order.
3. Otherwise slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. Continue the process until the element can be placed in the hole.


13
21
16
24
31
19
68
65
26
32
bubbling the hole up
13
21
16
6
19
68
65
26
32
31

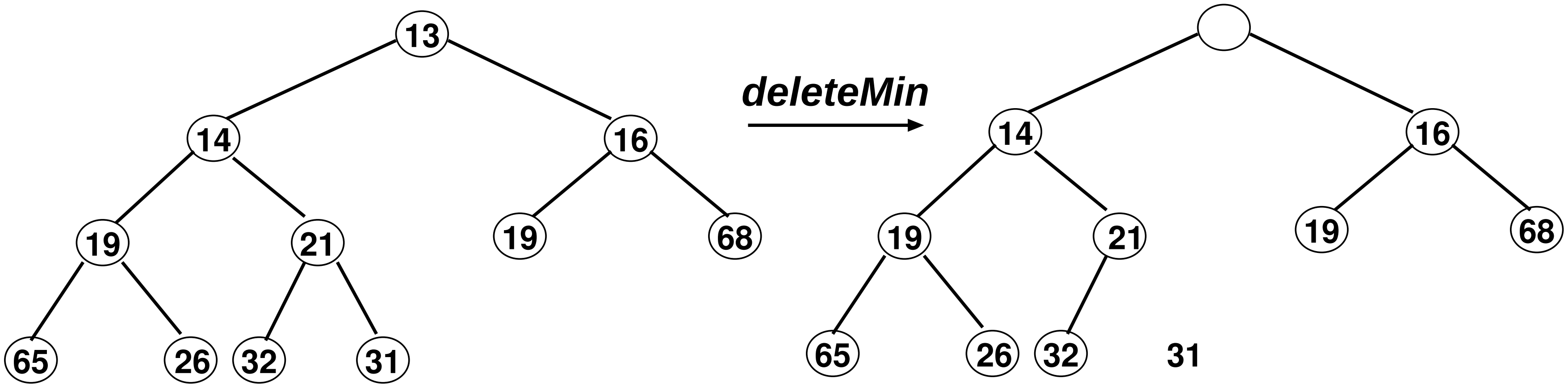
Insert 14



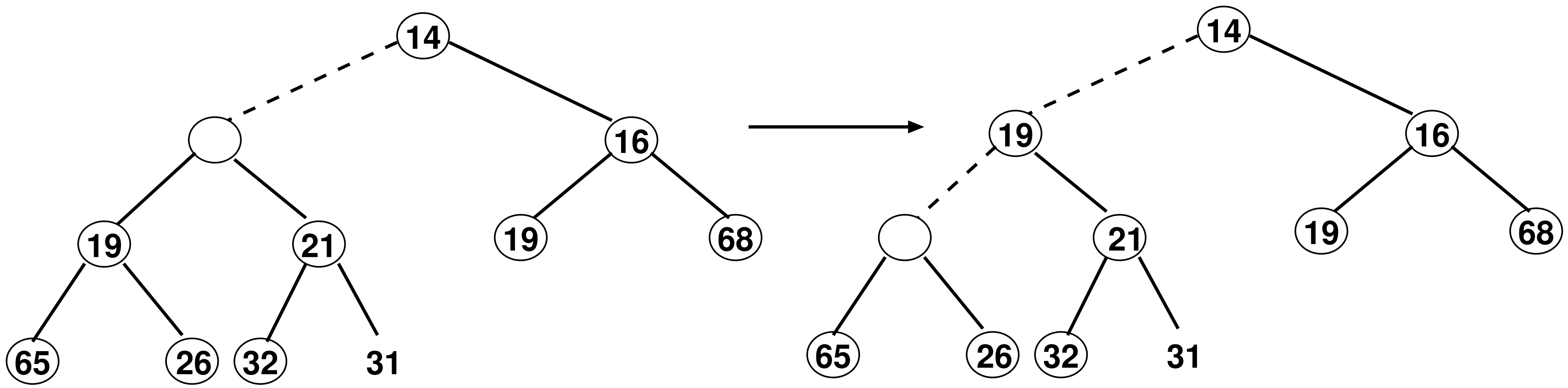
Insert 14

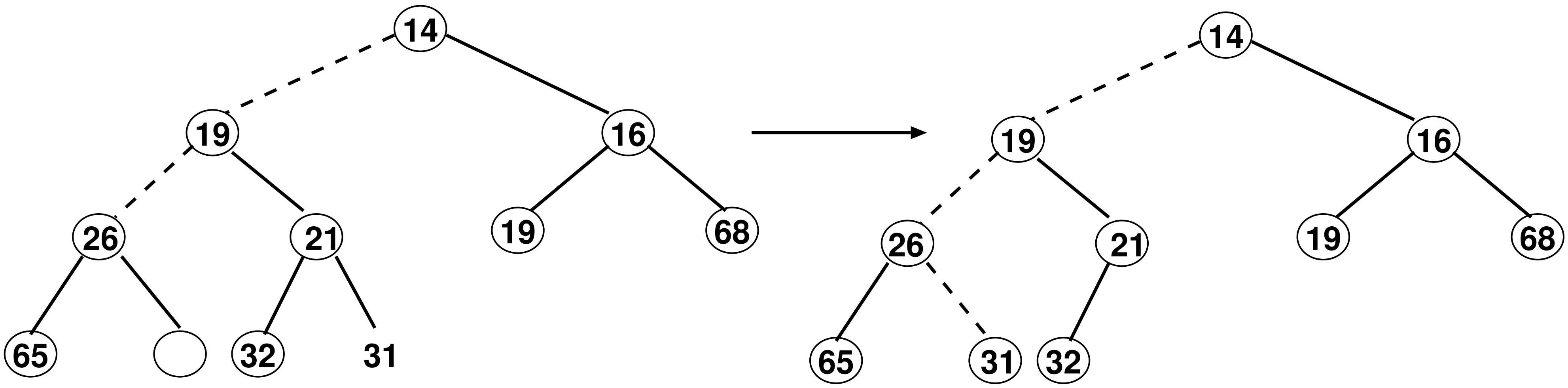
deleteMin

1. Create a hole at the root.
2. The last element X in the heap must move somewhere in the heap.
3. Slide the smaller of the hole's children into the hole, thus pushing the hole down one level. Repeat this step until X can be placed in the hole.



Creation of the hole at the root


14
16
19
21
19
68
65
26
32
31
14
19
16
21
19
68
65
26
32
31


14
19
16
26
21
19
68
65
32
31
14
19
16
26
21
19
68
65
31
32

Other heap operations

Buildheap

- **First method**
  - Takes as input N items and places them into an empty heap.
  - N successive `inserts`

  $\Rightarrow O(N)$ average case but $O(N \log N)$ worst-case.
- **Second method**
  - Establish heap order property from an arbitrary arrangement of items.
  - $O(n)$ in the worst case.

Heapsort

1st Method

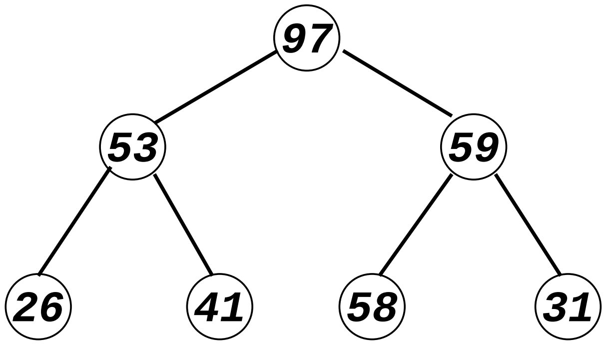
1. Build a binary heap ($O(N)$).
2. Perform N `deleteMin` operations copy them in a second array and then copy the array back ($N \log N$).

$\Rightarrow$ waste in space : an extra array is needed.

Heapsort

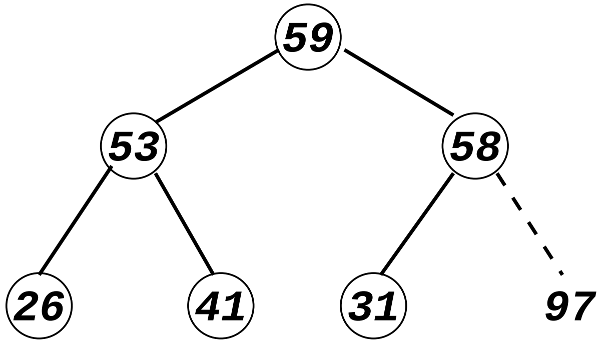
2nd Method

- Avoid using a second array : after each `deleteMin` the cell that was last in the heap can be used to store the element that was just deleted.
- After the last `deleteMin` the array will contain the elements in **decreasing** order.
- We can change the ordering property (max heap) if we want the elements in **increasing** order.
- $O(N \log N)$ time complexity. Why ?



0	1	2	3	4	5	6	7	8	9	10
	97	53	59	26	41	58	31			

First deleteMax



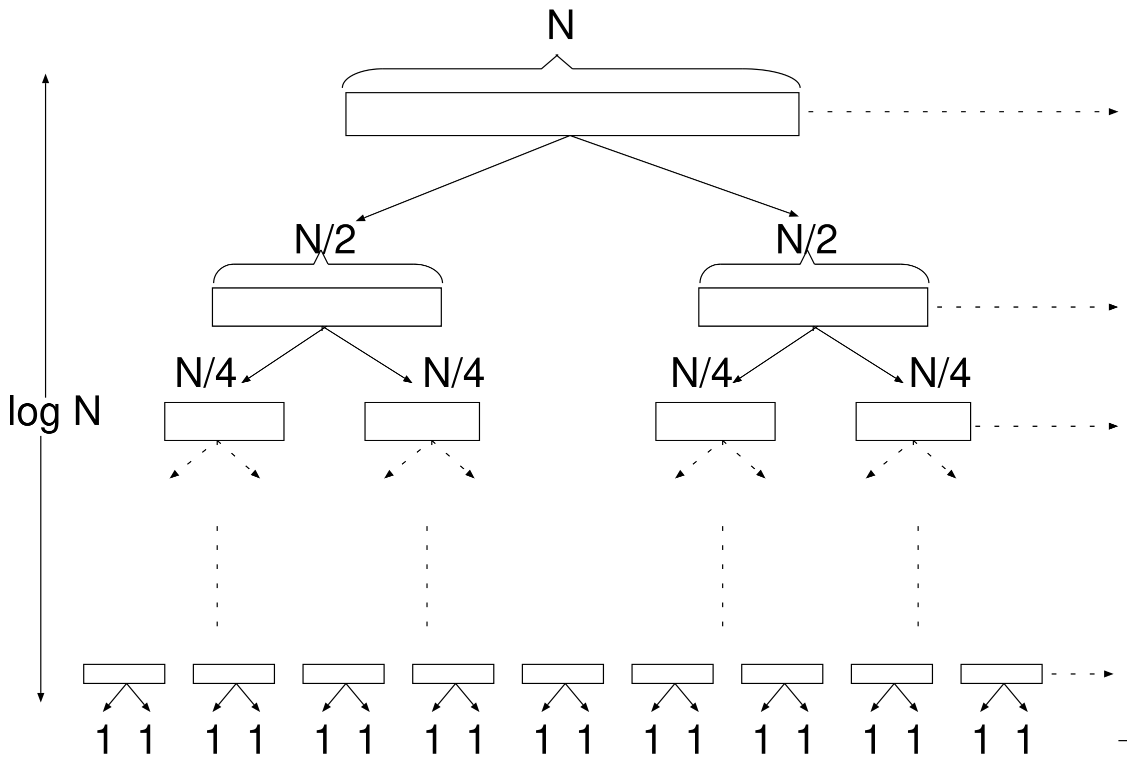
0	1	2	3	4	5	6	7	8	9	10
	59	53	58	26	41	31	97			

Mergesort

Recursive algorithm :

- If $N = 1$, there is only one element to sort.
- Otherwise, recursively mergesort the first half and the second half. Merge together the two sorted halves using the merging algorithm.
- Merging two sorted lists can be done in one pass through the input, if the output is put in a third list. At most $N - 1$ comparisons are made.

Analysis of Mergesort



$$T(N) = 2T(N/2) + cN$$

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + c$$

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + c$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + c$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c$$

$$\frac{T(N)}{N} = \frac{T(1)}{1} + c \log N$$

$$T(N) = cN \log N + N$$
$$= O(N \log N)$$

The master method

The master method provides a "cookbook" method for solving reccurences of the form

$$T(n) = a\,T(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

The master theorem

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

Quicksort

Given an array $A[p \ldots r]$, Quicksort works as follows :

Divide : the array $A[p \ldots r]$ is divided in two non empty subarrays $A[p \ldots q]$ and $A[q + 1 \ldots r]$.

Conquer : the two subarrays are recursively sorted.

Quicksort

$Quicksort(A, p, r)$

$\quad pivot \leftarrow SelectPivot(A, p, r)$

$\quad q \leftarrow Partitioning(A, p, r, pivot)$

$\quad Quicksort(A, p, q - 1)$

$\quad Quicksort(A, q + 1, r)$

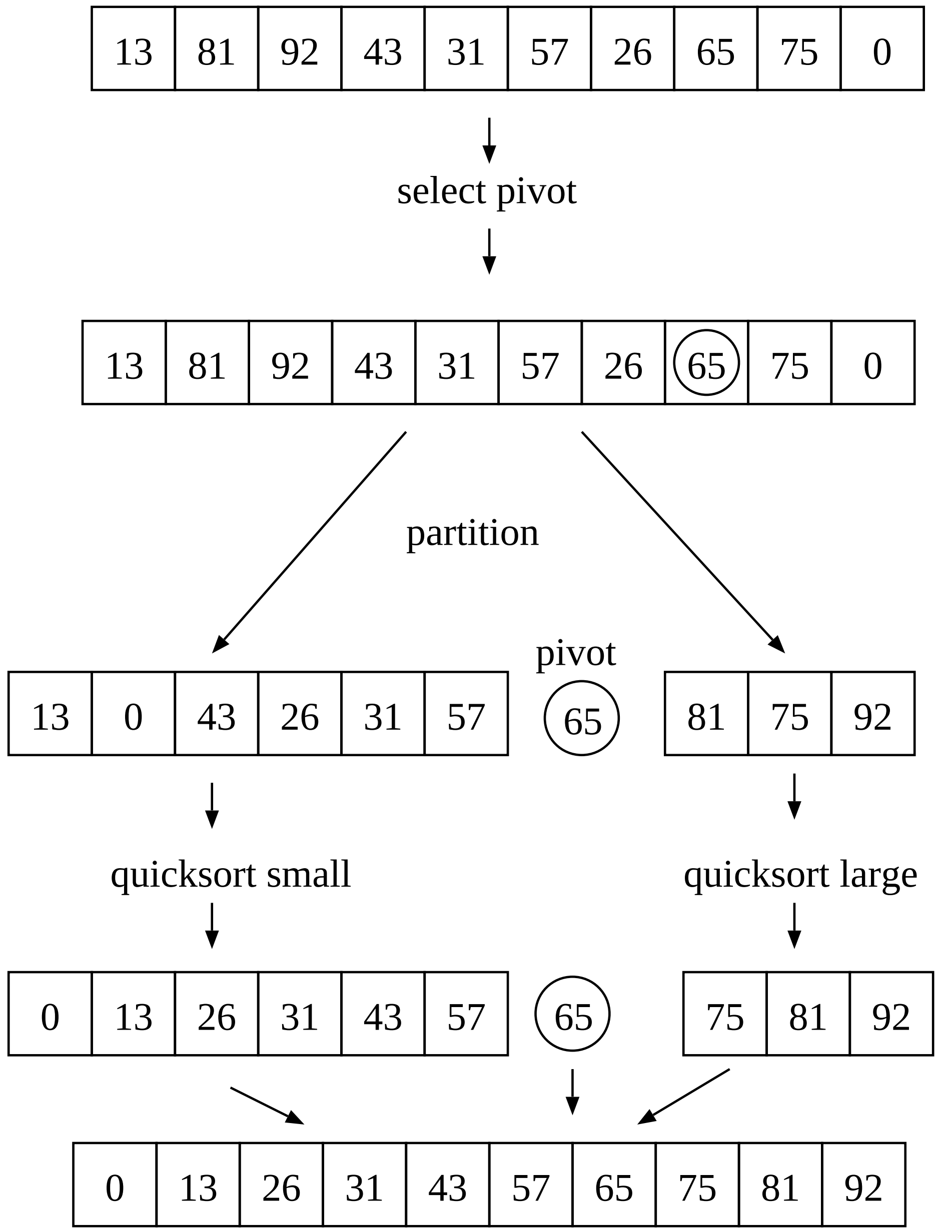
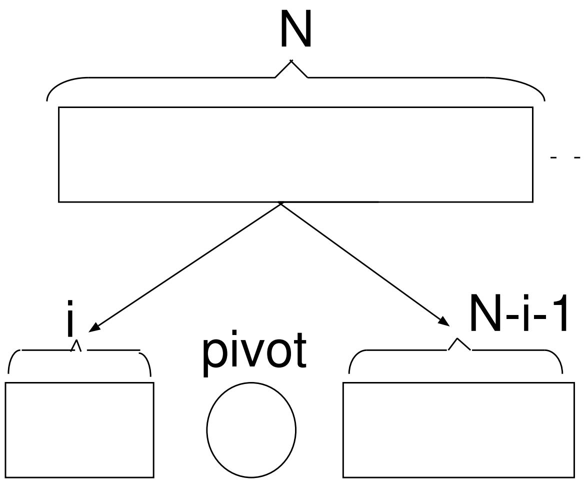


Figure 1: Quicksort steps illustrated by example

Analysis of Quicksort

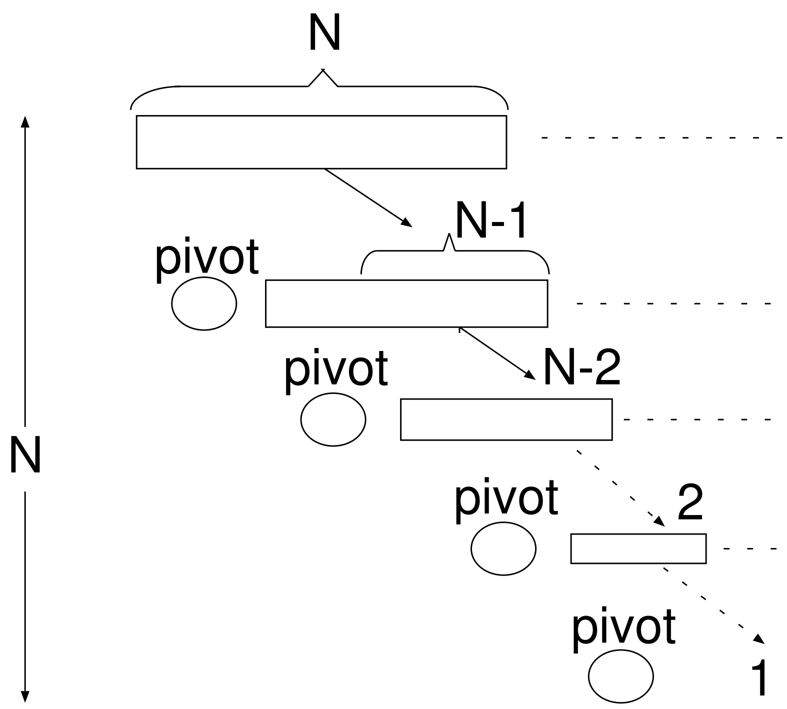


T(N) = T(i) + T(N-i-1) + cN

Assumptions :

- Random pivot.
- No cutoff for small arrays.
- $T(0) = T(1) = 1.$

Worst-case Analysis



$$T(N) = T(N-1) + cN$$

$$T(N-1) = T(N-2) + c(N-1)$$

$$T(N-2) = T(N-3) + c(N-2)$$

$$\vdots$$

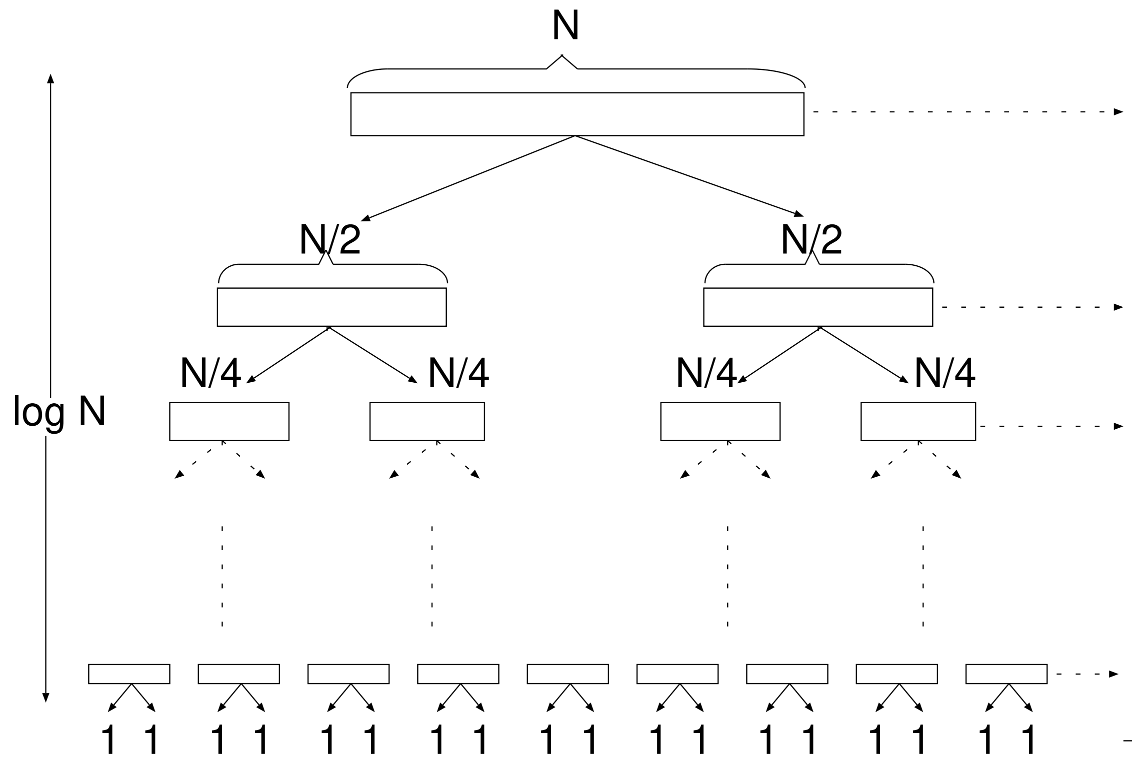
$$T(2) = T(1) + c(2)$$

$$T(N) = T(1) + c\sum_{i=2}^{N} i$$

$$T(N) = 1 + c\,(N-1)(N+2)/2$$

$$= O(N^2)$$

Best Case Analysis



$$T(N) = 2T(N/2) + cN$$

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + c$$

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + c$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + c$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c$$

$$\frac{T(N)}{N} = \frac{T(1)}{1} + c \log N$$

$$T(N) = cN \log N + N$$
$$= O(N \log N)$$

Average-Case Analysis

Assumptions :

- The possible sizes of the subarrays have the same probability (1/N where N is the number of elements of the array).

$$T(N) = T(i) + T(N - i - 1) + cN \tag{1}$$

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

$$T(N) = \frac{2}{N} \sum_{j=0}^{N-1} T(j) + cN \tag{3}$$

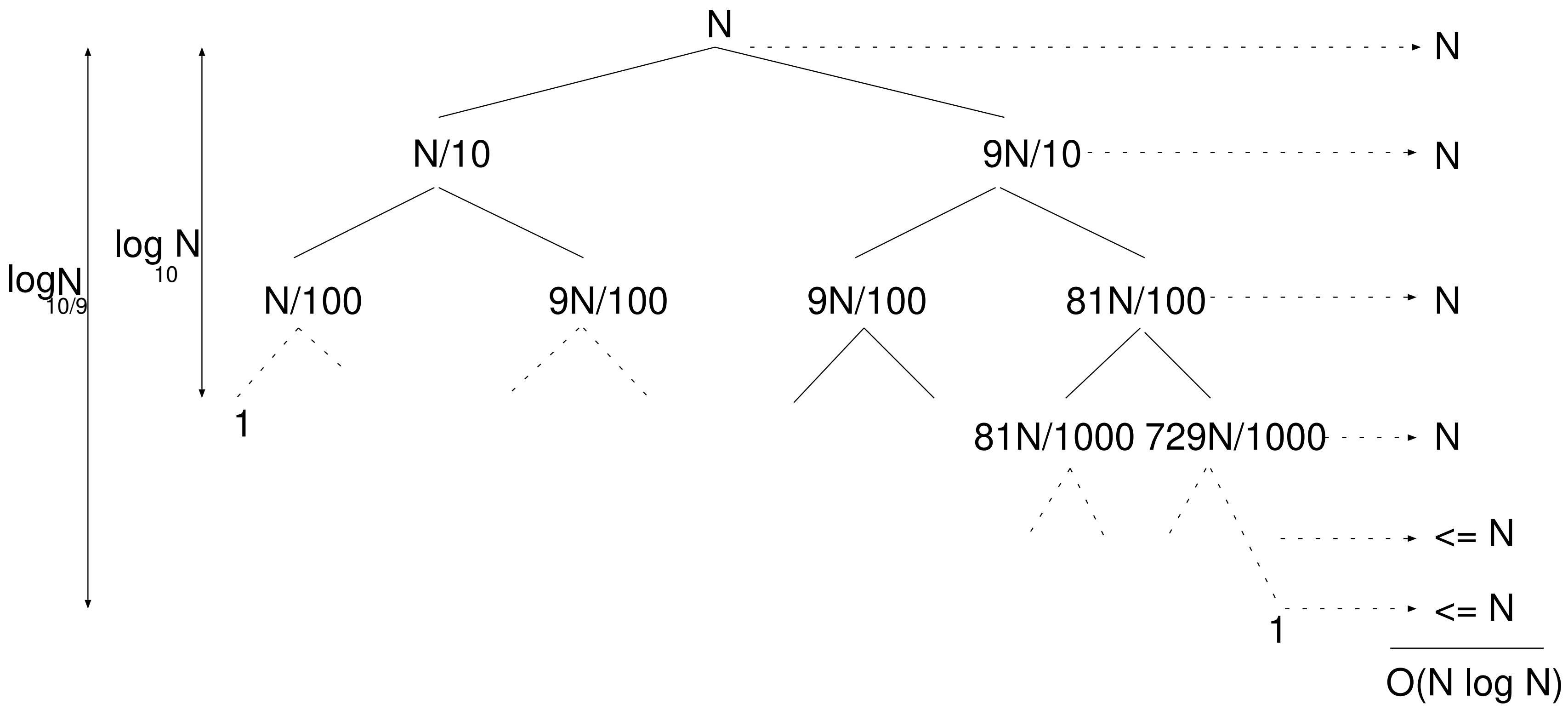
$$NT(N) = 2 \sum_{j=0}^{N-1} T(j) + cN^2 \tag{4}$$

To remove the summation we telescope with one equation :

$$(N - 1)T(N - 1) = 2 \sum_{j=0}^{N-2} T(j) + c(N - 1)^2 \tag{5}$$

4 - 5 yields :

$$
\begin{aligned}
NT(N) - (N-1)T(N-1) &= 2T(N-1) + 2cN - c \\
NT(N) &= (N+1)T(N-1) + 2cN \\
\frac{T(N)}{N+1} &= \frac{T(N-1)}{N} + \frac{2c}{N+1} \\
\frac{T(N-1)}{N} &= \frac{T(N-2)}{N-1} + \frac{2c}{N} \\
\frac{T(N-2)}{N-1} &= \frac{T(N-1)}{N-2} + \frac{2c}{N-1} \\
&\vdots \\
\frac{T(2)}{3} &= \frac{T(1)}{2} + \frac{2c}{3} \\
\frac{T(N)}{N+1} &= \frac{T(1)}{2} + 2c \sum_{i=3}^{N+1} \frac{1}{i} \\
\frac{T(N)}{N+1} &= \Theta(logN) \\
T(N) &= \Theta(NlogN)
\end{aligned}
$$


N
N
N/10
9N/10
N
log N
10
logN
10/9
N/100
9N/100
9N/100
81N/100
N
1
81N/1000
729N/1000
N
<= N
1
<= N
O(N log N)

Bucket sort

- General sorting algorithms using only comparisons require $\Omega(N \log N)$ time in the worst case.
- In some special cases it is possible to sort in linear time.
- If the input $A_1, A_2, \ldots, A_N$ consists of only positive integers smaller than M, bucket sort can be applied.
  1. Keep an array called `count`, of size M (M buckets), which is initialized to all 0s.
  2. When A_i is read, increment `count[`A_i`]` by 1.
  3. After all the input is read, scan the `count` array, printing out the a representation of the sorted list.
  4. The algorithm takes $O(M+N)$. If M is $O(N)$, then the total is $O(N)$.
  5. Useful algorithm when the input is only small integers.

Radix sort

- Input : the keys are all nonnegative integers in base 10 and having the same number of digits.
- 2 ways to sort the keys :
  - **Method 1 :** Sort on the most significant digit first (leftmost digit first). The ith step of the method consists in distributing the keys into distinct piles based on the values of the ith digit from the left.

  $\Rightarrow$ a variable number of piles is required.
  - **Method 2 :** Sort on the least significant digit first. We can use 10 piles (one for each decimal digit).
- $\Theta(N)$ in the best case but $\Theta(N^2)$ in the worst case.

3. A general lower bound for sorting

Prove that any algorithm for sorting that uses only comparisons requires

- $\Omega(n \log n)$ comparisons in the worst case

  $\Rightarrow$ Merge sort and Heap sort are optimal to within a constant factor

- and $\Omega(n \log n)$ comparisons in the average case

  $\Rightarrow$ quick sort is optimal on average within a constant factor

Decision Trees

- A decision tree is an abstraction used to prove lower bounds.
- Every algorithm that sorts by using only comparisons can be represented by a decision tree.
- The number of comparisons used by the sorting algorithm is equal to the depth of the deepest leaf.

Lemma 1 Let T be a binary tree of depth d. Then T has at most 2^d leaves.

Lemma 2 A binary tree with L leaves must have depth at least $\lceil \log L \rceil$.

Theorem 1 Any sorting algorithm that uses only comparisons between elements requires at least $\lceil \log N! \rceil$ comparisons in the worst case.

Theorem 2 Any sorting algorithm that uses only comparisons between elements requires $\Omega(N \log N)$ comparisons.

4. External Sorting

- Most of the internal sorting algorithms take advantage of the fact that memory is directly addressable

  $\Rightarrow$ comparing elements is done in constant number of time units.

- This is not the case if the data is on tape or on a disk.

Model for external sorting

- Sort data stored on tape.
- We assume that at least 3 tape drives are available (otherwise any sorting algorithm will require $\Omega(N^2)$.

The simple algorithm

- Algorithm based on the merge sort principle.
- 4 tapes are used. 2 input and 2 output tapes.
- **First step :** read M records (M is the number of records the main memory can hold) at a time from the input tape, sort the records internally and write the sorted records on one of the output tapes. Read M other records, sort them and write the sorted records on the other tape. Repeat the process until all records are processed.
- Each set of records is called a **run**.
- The algorithm will require $\lceil \log(N/M) \rceil$.

Multi-way Merge

- Use 2k tapes. k input tapes and k output tapes.
- The algorithm will require $\lceil \log_k (N/M) \rceil$.

5. Order Statistics

- The **ith order statistic** of a set of n elements is the ith smallest element.
  - The **minimum** of a set of elements is the first order statistic.
  - The **maximum** is the nth order statistic.
  - the **median** is the element in the middle of a sorted list of elements.
- The **selection problem** consists in selecting the ith order statistic from a set of n distinct numbers.

The selection Problem

- **Algorithm 1A :** read the elements into an array and sort them, returning the appropriate element.

  $\Rightarrow$ assuming a simple sorting algorithm, the running time is $O(N^2)$ ($O(N \log N)$ if merge sort of heap sort are used).

- **Algorithm 1B : find the kth largest element**

  1. read k elements into an array and sort them. The smallest of these is in the kth position.
  2. Process the remaining elements one by one. As an element arrives, it is compared with the kth element in the array. If it is larger, then the kth element is removed, and the new element is placed in the correct place among the remaining $k - 1$ elements.

  $\Rightarrow$ $O(N\dot{k})$ running time. Why ?

- If $k = N/2$ then both algorithms are $O(N^2)$. k is known as the median in this case.

- The following algorithms run in $O(N \log N)$ in the extreme case of $k = N/2$.

Algorithm 6A

- **Algorithm for finding the kth smallest element**
  1. Read N elements into an array.
  2. Apply the `buildHeap` algorithm to this array.
  3. Perform k `deleteMin` operations. The last element extracted from the heap is the answer.
- Complexity : $O(N + k \log N)$ in the worst case.
  - $k = O(N/\log N) \Rightarrow O(N)$
  - $k = N/2 \Rightarrow N\Theta(N \log N)$
  - For large values of k : $O(k \log N)$
  - $k = N \Rightarrow O(N \log N)$ (Idea of the *heapsort*).
- By changing the heap-order property, we will solve the problem of finding the kth largest element.

Algorithm 6B

- **Find the kth largest element**

  1. Same idea as algorithm 1B.
  2. At any point in time, maintain a set S of the k largest elements.
  3. After the first k elements are read, when a new element is read it is compared with the kth largest element, which we denote by S_k (S_k is the smallest element in S).
     - If the new element is larger, then it replaces S_k in S.
  4. At the end of the input, we find the smallest element in S and return it as the answer.

- $O(k + (N - k)\log k) = O(N \log k)$ in the worst case. Why ?

Using quick sort for Selection

$Quickselect(A, p, r, k)$

$pivot \leftarrow SelectPivot(A, p, r)$

$q \leftarrow Partitioning(A, p, r, pivot)$

If $(k < q)$ **then**

  Else If `(k>q)` $Quickselect(A, q + 1, r, k)$

    Else `return` $A[q]$

$O(N^2)$ in the worst case but $O(N)$ in the average case.

Selection in expected linear time

$Randomizedselect(A, p, r, i)$

if p=r

  then return $A[p]$

$q \leftarrow Randomizedpartition(A, p, r)$

$k \leftarrow q - p + 1$

If $(i \leq k)$

  then return $Randomizedselect(A, p, q, i)$

  else return $Randomizedselect(A, q + 1, r, i - k)$

Selection in average-case linear time

$Randomizedpartition$ produces a partition whose low side has 1 element with probability $2/N$ and i elements with probability $1/N$ for $i = 2, 3, \ldots, n-1$.

$$
\begin{aligned}
T(N) &\leq 1/N(T(max(1, N-1)) + \sum_{k=1}^{N-1} T(max(k, N-k))) + O(N) \\
&\leq 1/N(T(N-1) + 2 \sum_{k=\lceil n/2 \rceil}^{n-1} T(k)) + O(N) \\
&= 2/N \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(N)
\end{aligned}
$$

The recurrence can be solved by substitution (assuming that $T(N) \leq cN$ for some constant c): $T(N) \leq cN \Rightarrow T(N) = O(N)$

Selection in worst-case linear time

Idea of the Select algorithm : Guarantee a good split when the array is partitioned.

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining n mod 5 elements.

2. Find the median of each of the $\lceil n/5 \rceil$ groups by insertion sorting the elements of each group and taking its middle element.

3. Use Select recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2.

4. Partition the input array around the median-of-medians x using a modified version of the Partition procedure. Let k be the number of elements on the low side of the partition, so that $n - k$ is the number of elements on the high side.

5. Use Select recursively to find the ith smallest element on the low side if $i \leq k$, or the $(i - k)$th smallest element on the high side if $i > k$.

Analysis of the Select algorithm

The number of elements greater than x is at least :

$$3(\lceil 1/2 \lceil n/5 \rceil \rceil - 2) \geq 3n/10 - 6$$

- if $n \leq 80$ then $T(n) \leq \Theta(1)$
- if $n > 80$ then $T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$

The recurrence can be solved by substitution (assuming that $T(N) \leq cN$ for some constant c) :

$$T(N) \leq cN \Rightarrow T(N) = O(N)$$