

# INTRODUCTION TO ALGORITHMS - CHAPTERS 1-9

Tamim Sookoor

Department of Computer Science  
University of Virginia  
Charlottesville, Virginia 22904  
`sookoor@cs.virginia.edu`

July 19, 2007

## ALGORITHMS

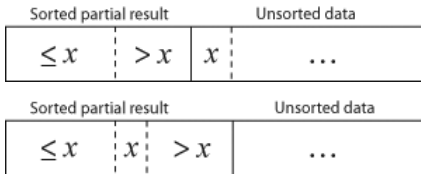
## DEFINITION

An *algorithm* is any well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*.



## INSERTION SORT

- Similar to sorting a hand of playing cards



- Best case:
  - When input is sorted
  - $O(n)$
- Worst case:
  - When input is sorted in reverse order
  - $O(n^2)$



# LOOP INVARIANTS

- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of a loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.



# LOOP INVARIANTS

- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of a loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

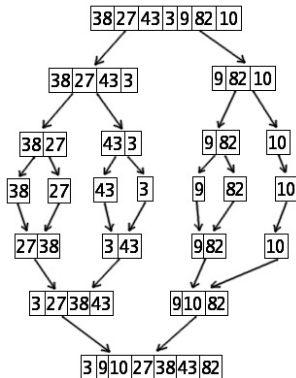
## EXAMPLE

At the start of each iteration of the **for** loop, the subarray  $A[1..j - 1]$  consists of the elements originally in  $A[1..j - 1]$  but in sorted order.



## MERGE SORT

- Divide and conquer algorithm



- Average and worst-case:  $O(n \log n)$



$\theta$ -NOTATION

## DEFINITION

$\theta(g(n)) = \{f(n) : \text{there exists positive constants } c_1, c_2,$   
and  $n_0$  such that  $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$  for all  $n \geq n_0\}$

- Asymptotically tight bound
- $\forall f(n) \in \theta(g(n))$  must be asymptotically nonnegative



## O-NOTATION

## DEFINITION

$O(g(n)) = \{f(n) : \text{there exists positive constants } c$   
and  $n_0$  such that  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0\}$

Asymptotic upper bound



# $\Omega$ -NOTATION

## DEFINITION

$\Omega(g(n)) = \{f(n) : \text{there exists positive constants } c$   
and  $n_0$  such that  $0 \leq cg(n) \leq f(n)$  for all  $n \geq n_0\}$

Asymptotic lower bound

## THEOREM

For any two functions  $f(n)$  and  $g(n)$ , we have  $f(n) = \theta(g(n))$  iff  
 $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$



## O-NOTATION

## DEFINITION

$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$



## O-NOTATION

## DEFINITION

$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$

## DEFINITION

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$



$\omega$ -NOTATION

## DEFINITION

$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$



# $\omega$ -NOTATION

## DEFINITION

$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$

## DEFINITION

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$



# $\omega$ -NOTATION

## DEFINITION

$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$

## DEFINITION

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

## DEFINITION

$f(n) \in \omega(g(n))$  iff  $g(n) \in o(f(n))$



## RECURRENCE

## DEFINITION

A *recurrence* is an equation or inequality that describes a function in terms of its value on smaller inputs

There are three methods to solve recurrences

- **Substitution method:** guess a bound and then use mathematical induction to prove the guess correct
- **Recursion-tree method:** convert a recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion
- **Master method:** provide bounds for recurrences of the form  $T(n) = aT(n/b) + f(n)$



# THE SUBSTITUTION METHOD

- 1 Guess the form of the solution
- 2 Use mathematical induction to find the constants and show that the solution works



# THE SUBSTITUTION METHOD

- 1 Guess the form of the solution
- 2 Use mathematical induction to find the constants and show that the solution works

## Making a good guess

- If a recurrence is similar to one you have seen before, guess a similar solution
- Prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty



# THE RECURSION-TREE METHOD

- Used to generate a good guess, which is then verified by the substitution method
- In a recurrence tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations
- Recursion trees are particularly useful when the recurrence describes the running time of a divide-and-conquer algorithm



## THE MASTER METHOD

$$T(n) = aT(n/b) + f(n)$$



## THE MASTER METHOD

$$T(n) = aT(n/b) + f(n)$$

- 1 If  $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$ , 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  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some constant  $\epsilon > 0$ , then  
 $T(n) = \theta(f(n))$



## PROBABILISTIC ANALYSIS

## DEFINITION

*Probabilistic analysis* is the use of probability in the analysis of problems

Indicator random variables can be used for probabilistic analysis

## EXAMPLE

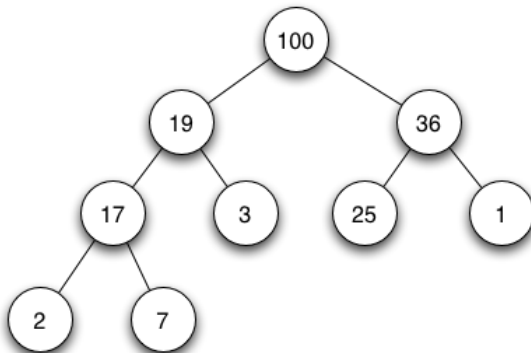
$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$



## HEAPS

## DEFINITION

A *heap* data structure is an array object that can be viewed as a nearly complete binary tree



# BINARY HEAP PROCEDURES

- MAX-HEAPIFY: runs in  $O(\lg n)$  time and maintains the max-heap property
- BUILD-MAX-HEAP: runs in linear time and produces a max-heap from an unordered input array
- HEAPSORT: runs in  $O(n \lg n)$  time and sorts an array in place



# PRIORITY QUEUE

## DEFINITION

A *priority queue* is a data structure for maintaining a set  $S$  of elements, each with an associated value called a *key*

Applications of priority queues:

- Job scheduling
- Event-driven simulator



# DESCRIPTION OF QUICKSORT

- Pick an element, called a pivot, from the array
- Partition the array so that all elements which are less than the pivot are in one subarray and all elements reater than the pivot are in the other partition
- Sort the two subarrays by recursive calls to quicksort



# PERFORMANCE OF QUICKSORT

- Worst-case
  - When partitioning routine produces one subarray with  $n - 1$  elements and the other with 0 elements
  - Running time of  $\theta(n^2)$
- Best-case
  - When each subarray has no more than  $n/2$  elements
  - Running time of  $O(n \lg n)$
- Average-case
  - Any split other than the best- or worst-case
  - Running time of  $O(n \lg n)$



# COMPARISON SORTS

- Merge sort
- Heap sort
- Quick sort



# COMPARISON SORTS

- Merge sort
- Heap sort
- Quick sort

## DEFINITION

*Comparison sorts* are algorithms where the sorted order is based only on comparisons between the input elements



# COMPARISON SORTS

- Merge sort
- Heap sort
- Quick sort

## DEFINITION

*Comparison sorts* are algorithms where the sorted order is based only on comparisons between the input elements

## THEOREM

*Any comparison sorting algorithm requires  $\Omega(n \lg n)$  comparisons in the worst case*



## COUNTING SORT

Sorts in linear time



# COUNTING SORT

## Sorts in linear time

- 1 Create an empty array,  $C$ , with size equal to the size of the input array



# COUNTING SORT

## Sorts in linear time

- 1 Create an empty array,  $C$ , with size equal to the size of the input array
- 2 Inspect each input element and, if the value of an input element in  $i$ , increment  $C[i]$



# COUNTING SORT

## Sorts in linear time

- 1 Create an empty array,  $C$ , with size equal to the size of the input array
- 2 Inspect each input element and, if the value of an input element in  $i$ , increment  $C[i]$
- 3 Accumulate the counts



# COUNTING SORT

## Sorts in linear time

- 1 Create an empty array,  $C$ , with size equal to the size of the input array
- 2 Inspect each input element and, if the value of an input element in  $i$ , increment  $C[i]$
- 3 Accumulate the counts
- 4 Place each input element in its correct sorted position in the output array,  $B$ , by indexing into  $B$  using  $C$



# COUNTING SORT

## Sorts in linear time

- 1 Create an empty array,  $C$ , with size equal to the size of the input array
- 2 Inspect each input element and, if the value of an input element in  $i$ , increment  $C[i]$
- 3 Accumulate the counts
- 4 Place each input element in its correct sorted position in the output array,  $B$ , by indexing into  $B$  using  $C$

Radix sort and bucket sort also sort in linear time



# ORDER STATISTIC

## DEFINITION

The  $i$ th *order statistic* of a set of  $n$  elements is the  $i$ th smallest element



# ORDER STATISTIC

## DEFINITION

The  $i$ th *order statistic* of a set of  $n$  elements is the  $i$ th smallest element

## EXAMPLE

The **minimum** of a set of elements is the first order statistic ( $i = 1$ )



## SELECTION PROBLEM

## DEFINITION

**Input:** A set  $A$  of  $n$  (distinct) numbers and a number  $i$ , with  $1 \leq i \leq n$ .

**Output:** The element  $x \in A$  that is larger than exactly  $i - 1$  other elements of  $A$



## SELECTION PROBLEM

## DEFINITION

**Input:** A set  $A$  of  $n$  (distinct) numbers and a number  $i$ , with  $1 \leq i \leq n$ .

**Output:** The element  $x \in A$  that is larger than exactly  $i - 1$  other elements of  $A$

- Can be solved in  $O(n \lg n)$  time by first sorting the input and then indexing the  $i$ th element



## SELECTION PROBLEM

## DEFINITION

**Input:** A set  $A$  of  $n$  (distinct) numbers and a number  $i$ , with  $1 \leq i \leq n$ .

**Output:** The element  $x \in A$  that is larger than exactly  $i - 1$  other elements of  $A$

- Can be solved in  $O(n \lg n)$  time by first sorting the input and then indexing the  $i$ th element
- Minimum and/or maximum can be selected in  $O(n)$  time



## SELECTION PROBLEM

## DEFINITION

**Input:** A set  $A$  of  $n$  (distinct) numbers and a number  $i$ , with  $1 \leq i \leq n$ .

**Output:** The element  $x \in A$  that is larger than exactly  $i - 1$  other elements of  $A$

- Can be solved in  $O(n \lg n)$  time by first sorting the input and then indexing the  $i$ th element
- Minimum and/or maximum can be selected in  $O(n)$  time
- The selection problem can be solved with an expected running time of  $\theta(n)$  using RANDOMIZED-SELECT



## SELECTION PROBLEM

## DEFINITION

**Input:** A set  $A$  of  $n$  (distinct) numbers and a number  $i$ , with  $1 \leq i \leq n$ .

**Output:** The element  $x \in A$  that is larger than exactly  $i - 1$  other elements of  $A$

- Can be solved in  $O(n \lg n)$  time by first sorting the input and then indexing the  $i$ th element
- Minimum and/or maximum can be selected in  $O(n)$  time
- The selection problem can be solved with an expected running time of  $\theta(n)$  using RANDOMIZED-SELECT
- The selection problem can be solved with a worst-case running time of  $O(n)$  using the median of medians algorithm

