DESIGN AND ANALYSIS OF ALGORITHMS

CSE 5311

Prepared by:
Chris M. Gonsalves and Shalli Prabhakar.
(Spring 2004; Instructor: Dr. M. Kumar.)

Why Analysis ??

- Often a choice of algorithms and data structures are available.
- Different time requirements.
- Different space requirements.
- Most appropriate choice has to be made.

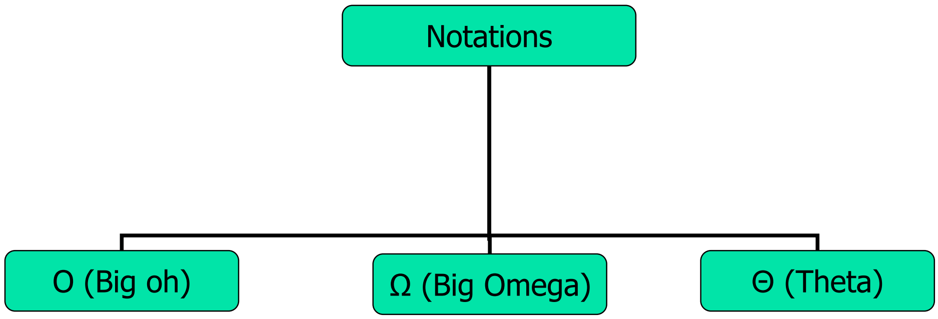
OUTLINE

- Asymptotic Analysis:
  - Growth Functions.
  - Insertion Sort / Selection Sort.
- Partially Ordered Trees and Heaps:
  - Functioning.
  - Examples.
  - Implementation.

OUTLINE

- Recurrence Relations:
  - Recurrence Relations.
  - Substitution Method.
  - Iteration Method.
  - Master Method.

Asymptotic Analysis



What is Asymptotic Analysis ?

- ***Def:*** Rate at which usage of time or space grows with input size.
- Dependent on many factors:
  particular machine, particular compiler, etc.
- We want to analyze algorithm independent of the above factors.
- Analyze rate of growth, not absolute usage.
  solution : Asymptotic Analysis
- Always assume some operations take constant time

Upper Bound ??

- When we analyze algorithms, we usually refer to "upper bounds"

  – the worst case time complexity.

Lower Bound ??

- When we analyze the complexity of a problem we usually mean "lower bounds"

  – the running time of the fastest algorithm.

Example: the lower bound time complexity of searching a value in an unsorted array of size n is $\Omega(n)$.

- lower bound ≤ upper bound.

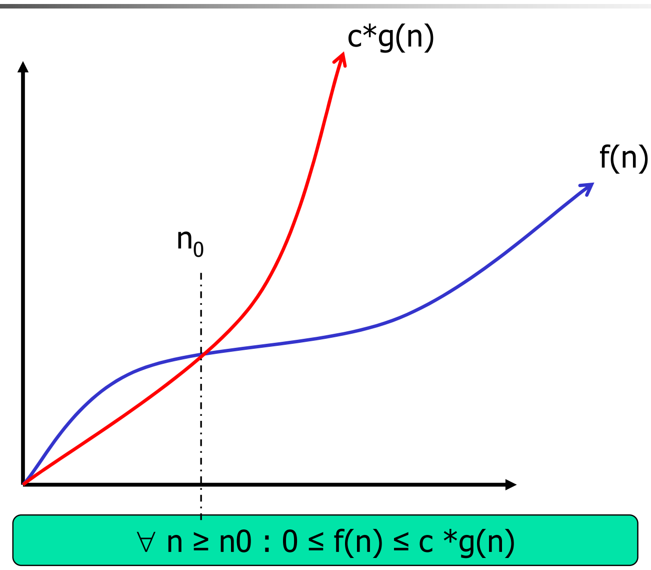
O (Big Oh) Notation

- ***Def:*** A function f(n) is said to be O(g(n)) if there are constants c and n_0 such that $f(n) \leq c*g(n)$ for all $n \geq n_0$

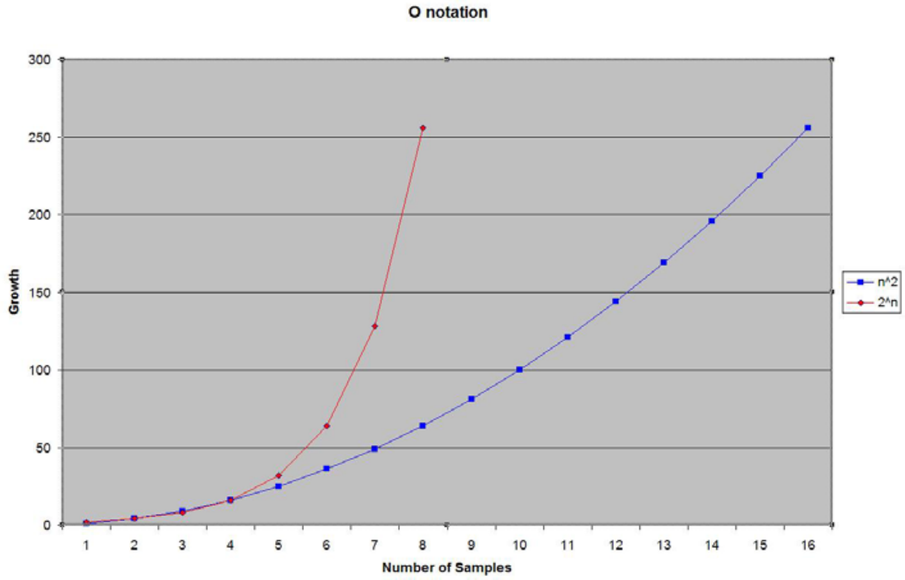
- Mathematical Formulation :

$$f(n) = O[g(n)] \Leftrightarrow \exists \text{ const } c, n_0 \quad \text{S.T. } \forall\, n \geq n_0 : 0 \leq f(n) \leq c*g(n)$$

O (Big Oh) Notation



O (Big Oh) Notation



Common Big-Oh Bounds

- O(1): constant time e.g.: Accessing an array element
- O(lg n): logarithmic time e.g.: binary search
- O(n): linear time e.g.: linear search
- O(n lg n) e.g.: merge sorts
- O(n^k): polynomial time e.g.: insertion sort is O(n^2)
- O(2^k): exponential time e.g.: recursive Fibonacci

An example algorithm

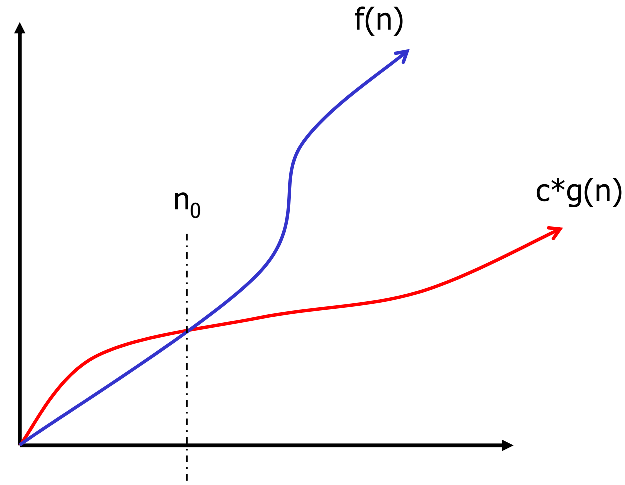
- **Suppose we designed the following algorithm to solve a problem**
  - prompt the user for a filename and read it (100 "time units")
  - open the file (50 "time units")
  - read 15 data items from the file into an array (10 "time units" per read)
  - close the file (50 "time units")
- **We could use the formula 200 + 10n where n = number of items read (here 15)**
- **to describe the algorithm's time complexity. Which term of the function really describes the growth pattern as n increases?**
  -> 10n .
- **Therefore, the time complexity for this algorithm is in O(n).**

Ω (Big Omega) Notation

- ***Def:*** A function f(n) is said to be Ω(g(n)) if there are constants c, $n_0 \geq 0$ such that $f(n) \geq c*g(n)$ for all $n \geq n_0$.
- Mathematical Formulation :

$f(n) = \Omega[g(n)] \Leftrightarrow \exists$ const c, n_0
S.T. $\forall\ n \geq n_0 : 0 \leq c*g(n) \leq f(n)$

Ω (Big Omega) Notation

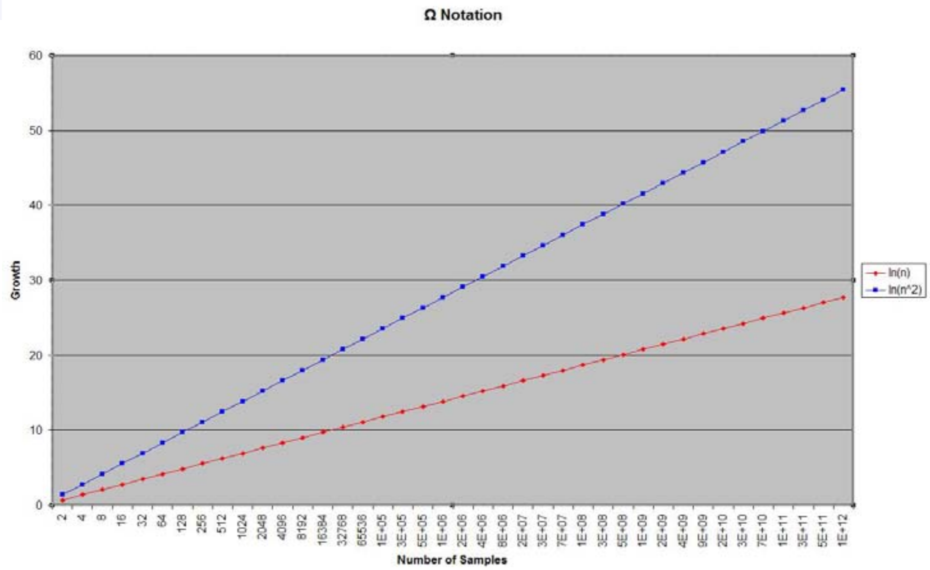


$\forall\ n \geq n0 : 0 \leq c *g(n) \leq f(n)$

Ω (Big Omega) Notation

- The Big-Omega (Ω) notation is used to indicate a "lower bound" on the time taken by an algorithm.

Ω (Big Omega) Notation



POINT TO REMEMBER

- "Big - Oh" is a bound on worst case, “Big Omega” is a bound (from below) on best case.
- e.g.: in the case of insertion sort, for all inputs n:

  T(n) = **O** (n2)
  = **Ω** (n)

- "Running time is **Ω**(g (n))" means that no matter what input the running time is at least c * g (n) for large n.

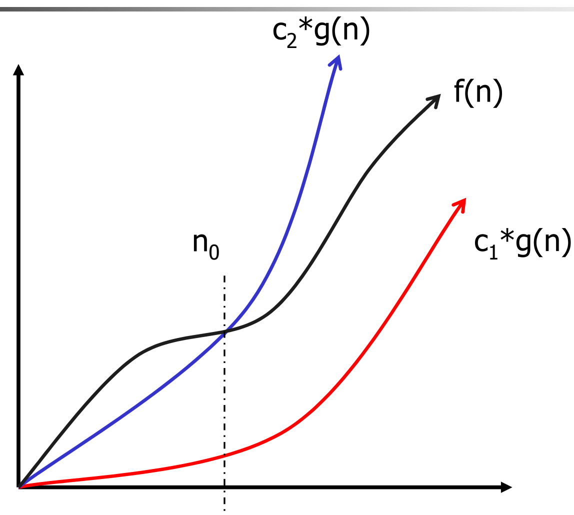
Θ (Theta) Notation

- ***Def:*** A function f(n) is said to be Θ(g(n)) if there exists constants c_1, c_2, $n_0 > 0$ such that $f(n) \geq c_2 * g(n)$ and $c_1 * g(n) \geq f(n)$ for all $n \geq n_0$.
- Mathematical Formulation :

$$f(n) = \Omega[g(n)] \Leftrightarrow \exists \text{ const } c_1, c_2, n_0$$

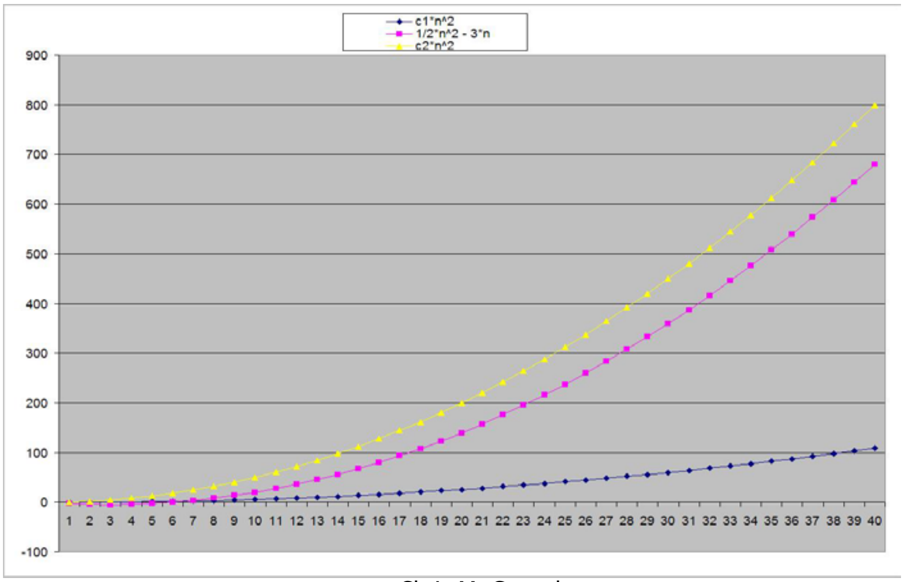
$$\text{S.T. } \forall\ n \geq n_0 : 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

Θ (Theta) Notation



$\forall$ n ≥ n0 : 0 ≤ c1 *g(n) ≤ f(n) ≤ c2 *g(n)

Θ (Theta) Notation



Θ (Theta) Notation

- Prove: $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.
- Consider :

  By Def : $c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$

  $c_1 \le \frac{1}{2} - 3/n \le c_2$

  Therefore $c_1 \le \frac{1}{2} - 3/n$, $c_2 \ge \frac{1}{2} - 3/n$ and $n_0 \ge 0$

  If $n \ge 1$ then $\frac{1}{2} - 3/n \le \frac{1}{2}$ (by making $c_2 = \frac{1}{2}$)

  $\frac{1}{2} - 3/n \ge 1/14$ when $n \ge 7$

- We have proved that n_0, c_1 and c_2 Exists.

  Hence $\frac{1}{2}n^2 - 2n = \Theta(n^2)$.

POINT TO REMEMBER

- $\mathbf{O}$ (g (n)) is the top piece of bread .
- $\boldsymbol{\Omega}$ (g (n)) is the bottom piece of bread
- in the $\boldsymbol{\Theta}$ (g (n)) sandwich

- $f(n) = \boldsymbol{\Theta}(g(n)) \Leftrightarrow$

  $f(n) = \mathbf{O}(g(n))$ and $f(n) = \boldsymbol{\Omega}(g(n))$

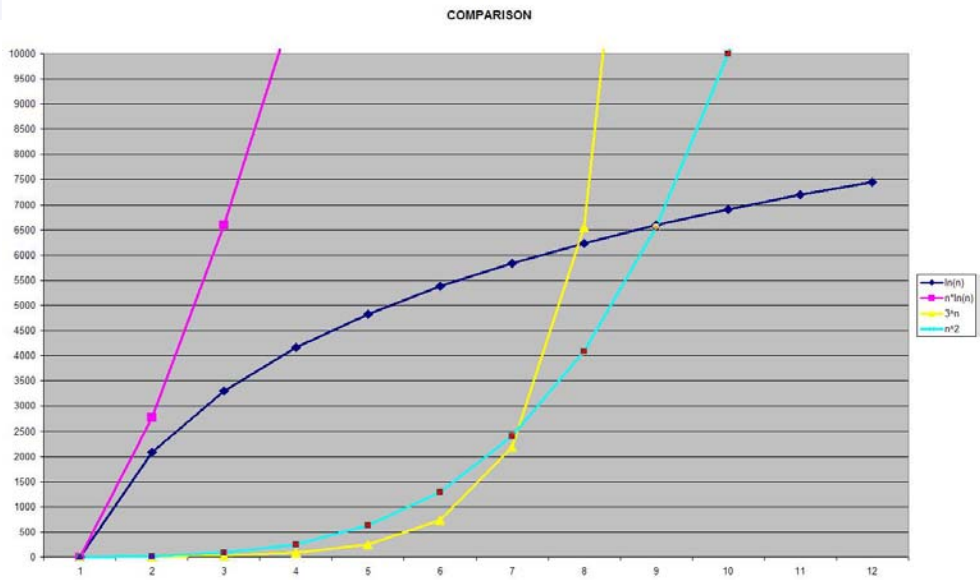
POINT TO REMEMBER

- $f(n) = \boldsymbol{\Theta}(g(n))$ and $g(n) = \boldsymbol{\Theta}(h(n)) \Leftrightarrow f(n) = \boldsymbol{\Theta}(h(n))$
- $f(n) = \mathbf{O}(g(n))$ and $g(n) = \mathbf{O}(h(n)) \Leftrightarrow f(n) = \mathbf{O}(h(n))$
- $f(n) = \boldsymbol{\Omega}(g(n))$ and $g(n) = \boldsymbol{\Omega}(h(n)) \Leftrightarrow f(n) = \boldsymbol{\Omega}(h(n))$

- $f(n) = \boldsymbol{\Theta}(g(n))$ IFF $g(n) = \boldsymbol{\Theta}(f(n))$

- $f(n) = \mathbf{O}(g(n))$ IFF $g(n) = \boldsymbol{\Omega}(f(n))$

Growth of a Functions



Simple Theorem

- Given :- f(n) = O[g(n)] and
  g(n) = O[f(n)]
- S.T. :- $f(n) = \Theta[g(n)]$.
- Proof :
  - $\exists\, n_1, c_1$:- $\forall\, n \geq n_1 : 0 \leq f(n) \leq c_1 * g(n)$
  - $\exists\, n_2, c_2$:- $\forall\, n \geq n_2 : 0 \leq g(n) \leq c_2 * f(n)$

$\Rightarrow$ $\forall\, n \geq \max(n_1, n_2) : 0 \leq 1/\, c_2 * g(n) \leq f(n) \leq c_1 * g(n)$

Binary Trees (Introduction)

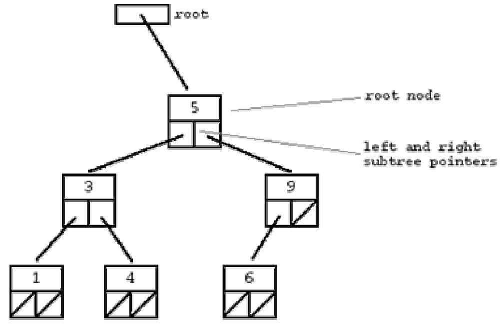
- **Binary tree is either**
  - An empty tree (no vertices), or
  - a tree consisting of two disjoint binary trees *L*, and *R* and a vertex *r* (called the root of the tree) in the following form *(L, r, R)*.

- **Binary Tree Node:**

```
struct node {
      int data;
      struct node* left;
      struct node* right;
}
```

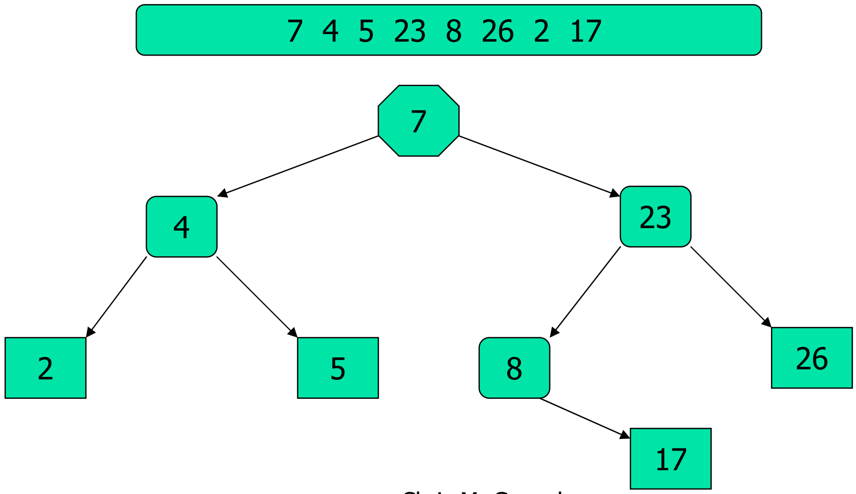

Binary Search Trees (Introduction)

- A "**B**inary **S**earch **T**ree" (BST) is a *ordered* binary tree.
- Element(leftchild) ≤ Element(root) ≤ Element(rigthchild).



Binary Search Trees

- Generation of Binary Tree :-



PARTIALLY ORDERED TREES AND HEAPS

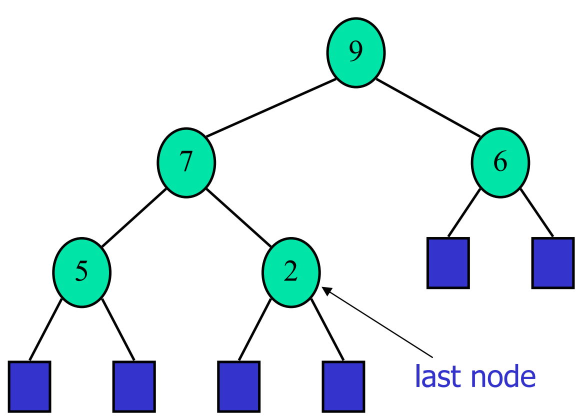
HEAPS - Outline

- What is a Heap.
- Building a Heap.
- Height of a heap.
- Insertion
- Removal
- Heap-sort
- Vector-based implementation
- Bottom-up construction

What is a HEAP

- **Def:** A heap is a *Partially Ordered Binary Tree* storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $key(parent(v)) \geq key(v)$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
    - at depth $h - 1$, the internal nodes are to the left of the external nodes

- The last node of a heap is the rightmost internal node of depth $h - 1$



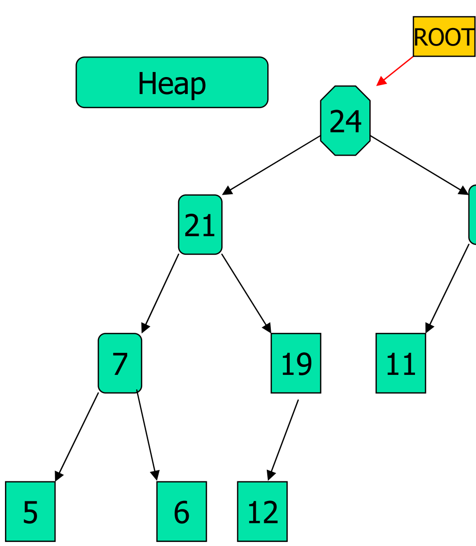
HEAPS

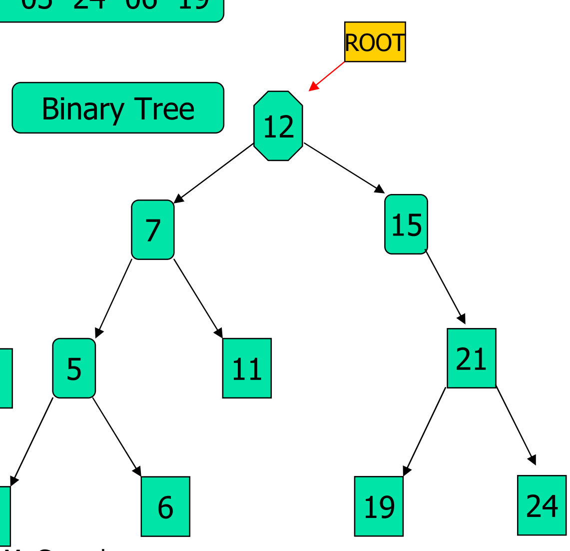
1	2	3	4	5	6	7	8	9	10
12	10	06	08	05	01	02	03	07	04

e.g. Parent(4) = floor(4/2) = 2
left(4) = 2 * 4 = 8
right(4) = 2 * 4 + 1 = 9

HEAPS v/s Binary Tree

12 7 15 05 21 11 03 24 06 19





Building a HEAPS

MAX_HEAPIFY: This procedure is applied to maintain the HEAP property: ***key*(*parent*(*v*))** ≥ ***key*(*v*).**

```
MAX_HEAPIFY(n,Array[])
{
        l = left(n); r = right(n);
        if l ≤  heap_size(Array) and Array[l] > Array[n] then
            largest = l
        else largest = n

        if r ≤ heap_size(Array) and Array[r] > Array[largest] then
            largest = r

        If largest <> n then {
            exchange Array[n] = Array[largest]
            MAX_HEAPIFY(largest,Array)
        }
}
```

T (n) = O (log n)

Building a HEAPS

BUILD_MAX_HEAPIFY: This procedure builds a HEAP of the Array modified by MAX_HEAPIFY .

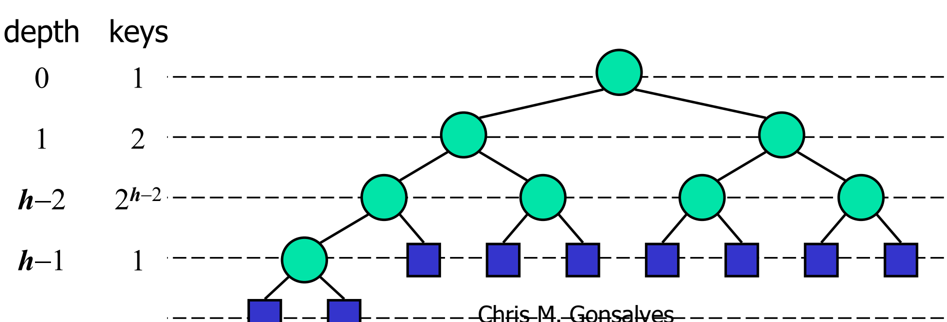
```
BUILD_MAX_HEAPIFY(Array[])
{
        heap_size(Array) = length (Array)
        for i = heap_size(Array)/2 downto 1 do
                MAX_HEAPIFY(i,Array)
}
```

T (n) = O (n)

RESULTS : HEAP.HEAPIFY

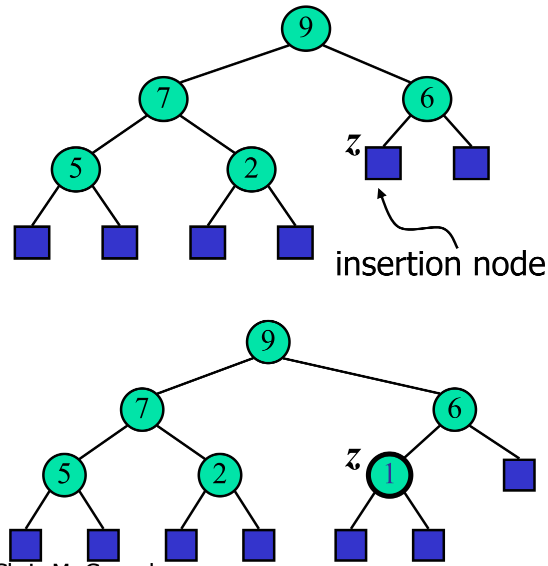
Height of a HEAP

- Theorem: A heap storing n keys has height $O(\log n)$
  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are $2i$ keys at depth $i = 0, \ldots, h - 2$ and at least one key at depth $h - 1$, we have $n \geq 1 + 2 + 4 + \ldots + 2h\text{-}2 + 1$
  - Thus, $n \geq 2h\text{-}1$, i.e., $h \leq \log n + 1$



Insertion into a HEAP

- Method *insertItem* of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - **Find** the insertion node z (the new last node)
  - **Store** k at z and expand z into an internal node
  - **Restore** the heap-order property (discussed next)



Insertion: Heap

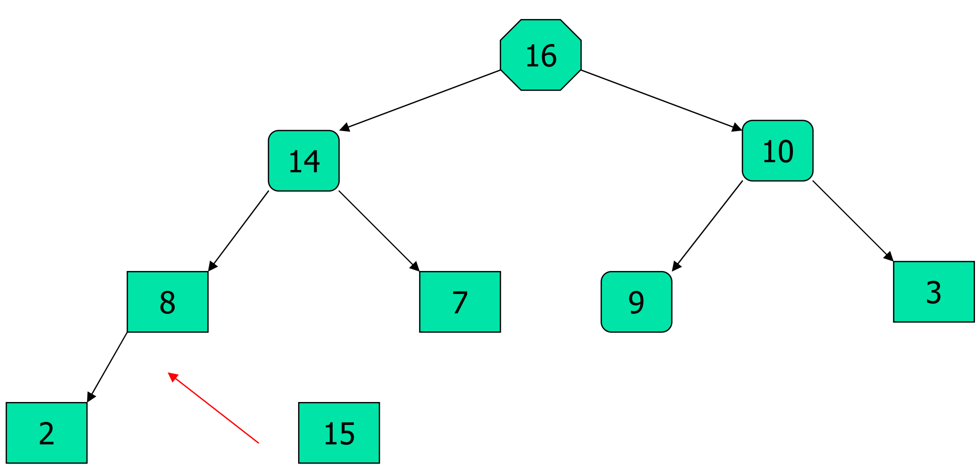
```
HEAP_INSERT_KEY(Array, key)
{
   HeapSize(Array) = HeapSize(Array) + 1
   n = HeapSize(Array)
  Array[n] = key

  while n > 1 and Array[Parent(n)] < Array[n] {
          exchange Array[n] and Array[Parent(n)]
          n = Parent(n)
          Array[n] = key
  }
}
```

$T(n) = O(\log n)$

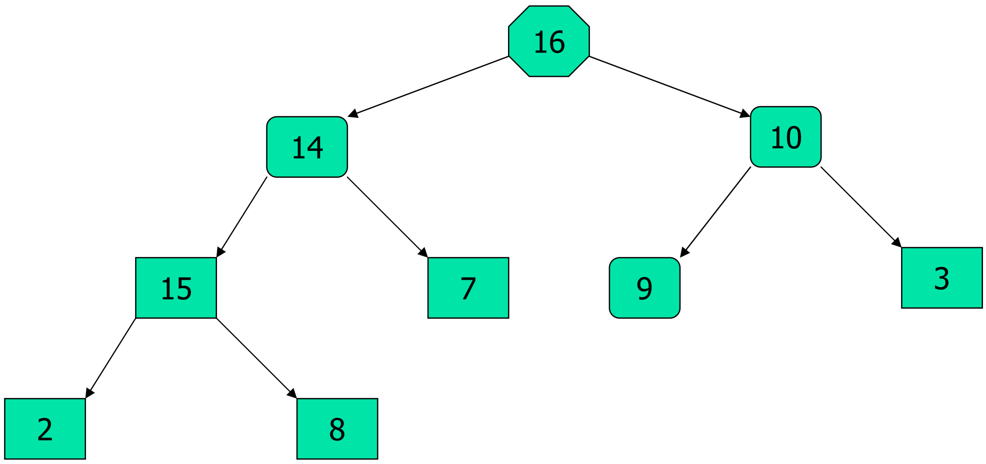
RESULTS : HEAP.INSERT

```
C:\WINDOWS\System32\cmd.exe
Input Array :
12.000
10.000
8.000
6.000
5.000
7.000
1.000
2.000
3.000
4.000
4.000
Insert Key Operation
Enter key :  15
12.000  10.000  8.000  6.000  5.000  7.000  1.000  2.000  3.000  4.000  15.000

12.000  10.000  8.000  6.000  15.000  7.000  1.000  2.000  3.000  4.000  5.000

12.000  15.000  8.000  6.000  10.000  7.000  1.000  2.000  3.000  4.000  5.000

15.000  12.000  8.000  6.000  10.000  7.000  1.000  2.000  3.000  4.000  5.000

New Heap
15.000  12.000  8.000  6.000  10.000  7.000  1.000  2.000  3.000  4.000  5.000
C:\DOCUME~1\CHRISG~1\MYDOCU~1\UTA\SPRING~1\DAA\PRESEN~1>_
```


Insertion: Heap



Insertion: Heap

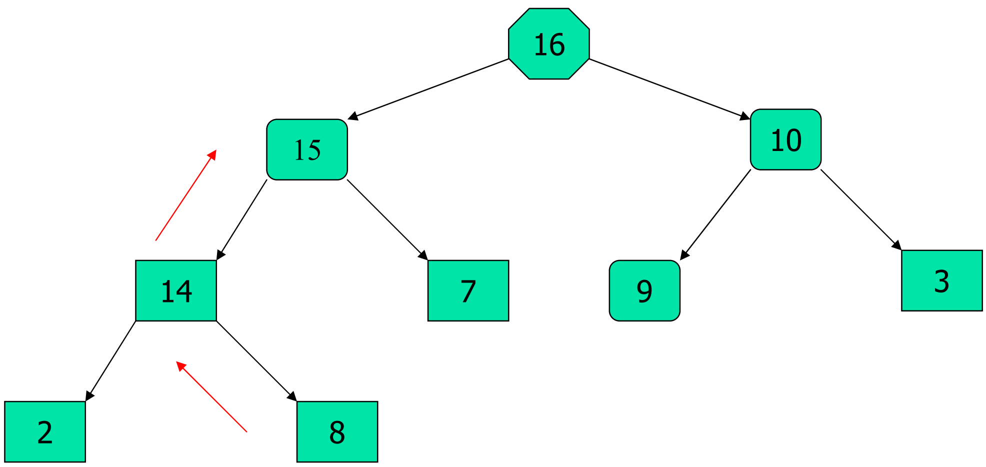


Deletion: Heap

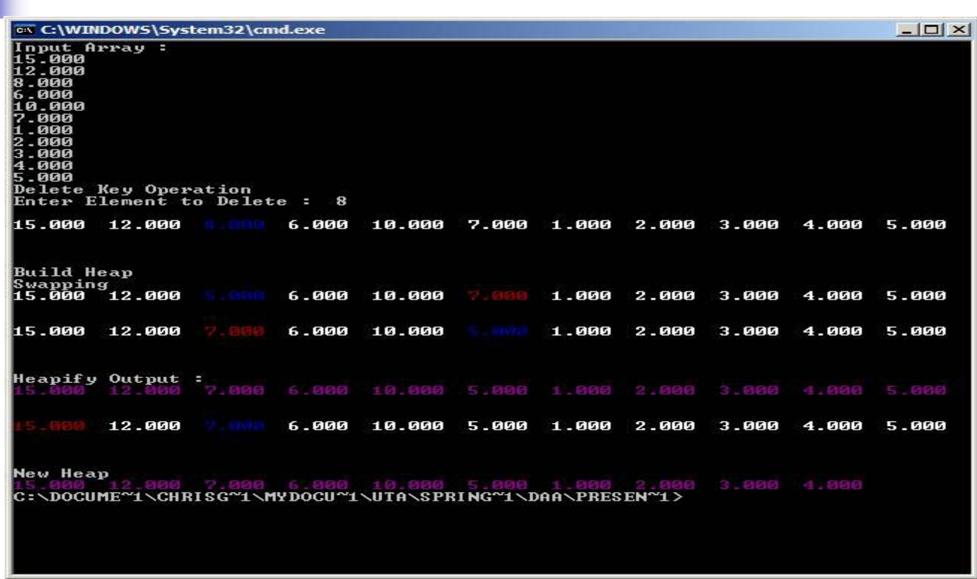
```
HEAP_DELETE_KEY(Array, key)
{
    Array[n] = Array[size]
    HeapSize(Array) = HeapSize(Array) - 1


     if(key <= Array[Parent(n)]){
          BuildHeap(size,Array)
     }
    else
          while (n > 1 and Array[Parent(n)] < key) {
                    Array[n] = Array[Parent(n)]
                    n = Parent(n)
          }
}
```

T (n) = O (log n)

Deletion: Heap



RESULTS : HEAP.DELETE



```
C:\WINDOWS\System32\cmd.exe
Input Array :
15.000
12.000
8.000
6.000
10.000
7.000
1.000
2.000
3.000
4.000
5.000
Delete Key Operation
Enter Element to Delete :  8

15.000  12.000  8.000  6.000  10.000  7.000  1.000  2.000  3.000  4.000  5.000


Build Heap
Swapping
15.000  12.000  5.000  6.000  10.000  7.000  1.000  2.000  3.000  4.000  5.000

15.000  12.000  7.000  6.000  10.000  5.000  1.000  2.000  3.000  4.000  5.000


Heapify Output :
15.000  12.000  7.000  6.000  10.000  5.000  1.000  2.000  3.000  4.000  5.000

15.000  12.000  7.000  6.000  10.000  5.000  1.000  2.000  3.000  4.000  5.000


New Heap
15.000  12.000  7.000  6.000  10.000  5.000  1.000  2.000  3.000  4.000
C:\DOCUME~1\CHRISG~1\MYDOCU~1\UTA\SPRING~1\DAA\PRESEN~1>
```

Heap Properties

- An array A representing a complete binary tree for HeapSize(A) elements satisfying the heap property:

  A[parent(i)] >= A[i]

  for every node i except the root node.

- HeapSize(A) <= length(A) .

- **parent(i)** = floor(i/2)
  left(i) = 2i ; left child
  right(i) = 2i + 1 ; right child

Partially Ordered Tree

- **Def:** Partially Ordered Tree is a *Complete Binary Tree* such that *value* of each parent node *smaller or equal* to the values in the children nodes.
- **Properties**:
  - The smallest element in a partially ordered tree (POT) is the root.
  - No conclusion can be drawn about the order of children.
- POT is just the *Opposite* of Heaps.

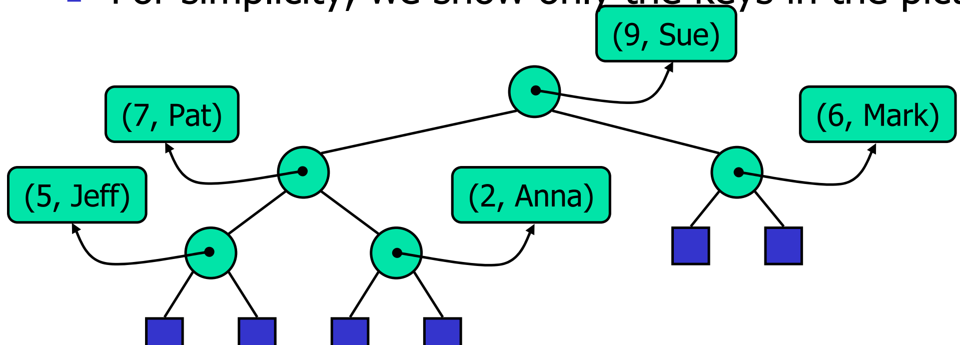
Priority Queue

- **Def:** A priority queue keeps its data in *order*, allowing the user to modify the smallest element only (e.g. minimum value).

- **Applications:**
  - Scheduling processes in operating systems.
  - Scheduling of further events in simulations.
  - Rank choices that are generated out of order.

Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we show only the keys in the pictures



Heap Search

Searching techniques :-

1. Preorder Traversal.
2. Inorder Traversal.
3. Postorder traversal.

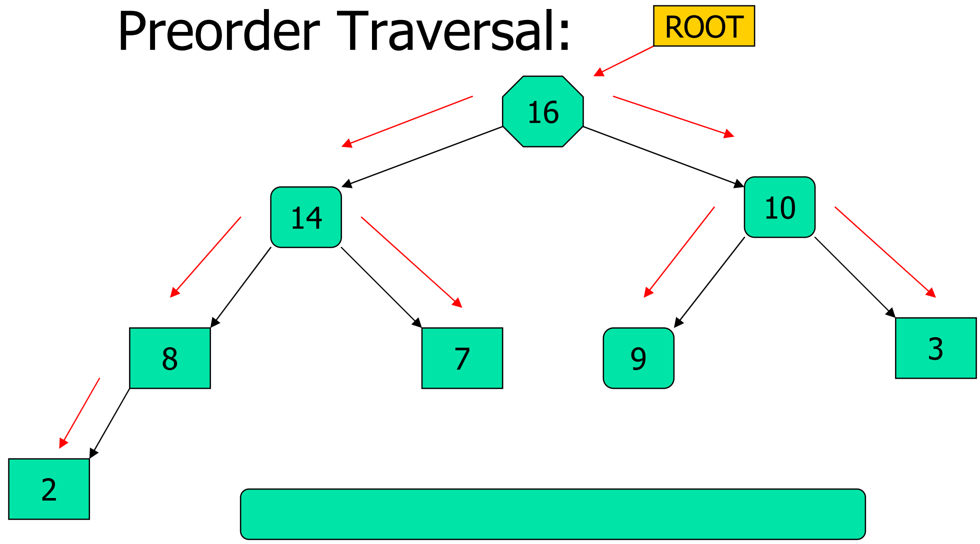
Heap Search

Preorder Traversal.

```
void Preorder (TreeNode * t)
    if (t != NULL) then
        Visit (t);
        Preorder (t->left);
        Preorder (t->right);
```

Heap Search

Preorder Traversal:



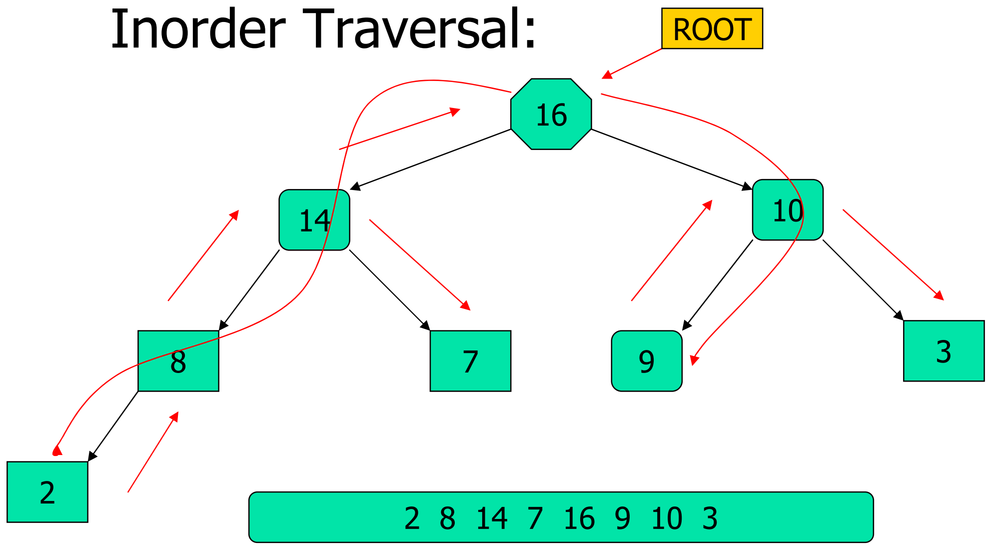
Heap Search

Inorder Traversal.

```
void Inorder (TreeNode * t)
    if (t != NULL) then
        Inorder (t->left);
        Visit (t);
        Inorder (t->right);
```

Heap Search

Inorder Traversal:



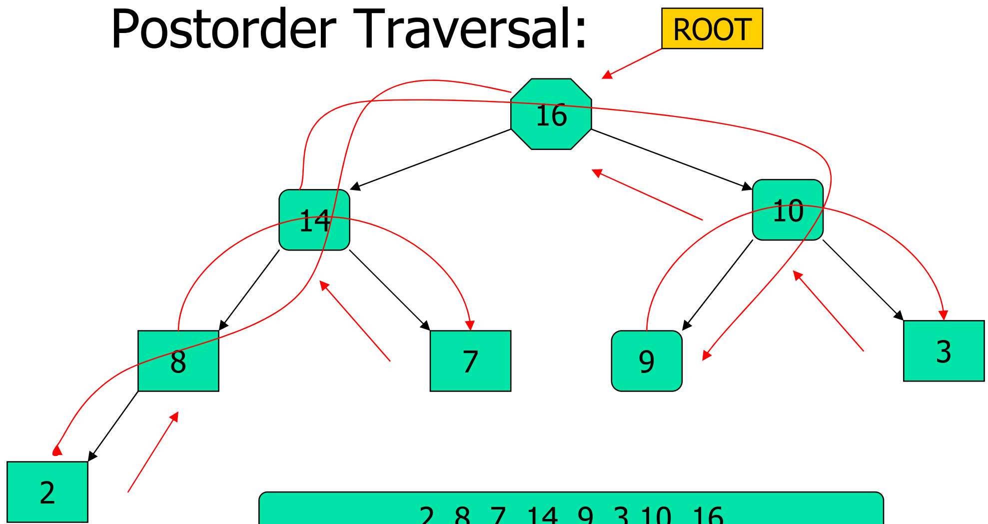
Heap Search

Postorder Traversal.

```
void Postorder (TreeNode * t)
    if (t != NULL) then
        Postorder (t->left);
        Postorder (t->right);
        Visit (t)
```

Heap Search

Postorder Traversal:



2 8 7 14 9 3 10 16

RECURRENCE RELATIONS

Recurrence Relations

- Many algorithms, particularly divide and conquer algorithms, have time complexities which are naturally modeled by recurrence relations.

- A recurrence relation is an equation which is defined in terms of itself.

Recursive Relations

- Recurrence relations are recursive definitions of mathematical functions or sequences.
  - For example, the recurrence relation

    $g(n) = g(n-1) + 2n - 1$

    $g(0) = 0$

    defines the function $f(n) = n^2$

Why are recurrences good things?

- Many natural functions are easily expressed as recurrences.
- It is often easy to find a recurrence as the solution of a counting problem. *Solving* the recurrence can be done for many special cases.

Recursion is Mathematical Induction!

- We have *general* and *boundary* conditions, with the *general* condition breaking the problem into smaller and smaller pieces.
- The *initial* or *boundary* condition terminate the recursion.
- As we will see, induction provides a useful tool to solve recurrences - guess a solution and prove it by induction.

Solving a recurrence relation

- Solution Techniques:
  - Substitution Method: Guess bound then use mathematical induction to prove.
  - Iteration Method: Convert recurrence to summation and bound summation.
  - Master Method :Simple solutions to recurrences of the form

$$T(n) = a\,T(n / b) + f(n)\ ;\ a \geq 1,\ b > 1$$

Substitution Method

- Guess a solution.
- Verify by induction.
- Solve for constants.

Substitution Method

- For example, for
  $T(n) = 2\,T(n / 2) + n$ and $T(1) = 1$
  we guess $T(n) = O(n \lg n)$

- Induction Goal:
  $T(n) \le c\, n \lg n$, for some c and all $n > n_0$

- Induction Hypothesis:
  $T(n / 2) \le c\,(n / 2) \lg (n / 2)$
- Proof of Induction Goal:
  $T(n)= 2\,T(n / 2) + n$
  $\le 2\,(c(n / 2) \lg (n / 2)) + n$
  $\le c\, n \lg (n / 2) + n$
  $= c\, n \lg n - c\, n \lg 2 + n$
  $= c\, n \lg n - c\, n + n$
  $\le c\, n \lg n$ provided $c \ge 1$

Substitution Method

- So far the restrictions on c, n_0 are only $c \ge 1$

- Base Case:
  $T(n_0) \le c\, n \lg n$
  Here, $n_0 = 1$ does not work, since $T(1) = 1$
  but $c_1 \lg 1 = 0$.

  However, taking $n_0 = 2$ we have:

  $T(2) = 2T(1) + 2 = 4$ and $4 \le c * 2 \lg 2 \le 2c \Rightarrow C \ge 2$

  $T(3) = 2T(2) + 2 = 10$ and $10 \le c * 3 \lg 3 \le 4.75c \Rightarrow C \ge 2$

  Thus $T(n) \le 2\, n \lg(n)$ for all $n \ge 2$

Iteration Method

- Express the recurrence as a summation of terms.
- Use techniques for summations.

Iteration Method

- For example, we iterate

  $T(n) = 3\,T(n / 4) + n$

  as follows:

  $T(n) = n + 3\,T(n / 4)$

  $= n + 3\,(n / 4 + 3\,T(n / 16))$

  $= n + 3\,(n / 4 + 3\,(n / 16 + 3\,T(n / 64)))$

  $= n + 3\,(n / 4) + 9\,(n / 16) + 27\,T(n / 64)$

- The i-th term in the series is $3^i(n / 4^i)$.
  We have to iterate until $(n / 4^i) = 1$, since $T(1) = \Theta(1)$, or equivalently until $i > \log_4 n$.

CSE@UTA

Iteration Method

- We continue:

$$T(n) = n + 3\,(n / 4) + 9\,(n / 16) + 27\,T(n / 64)$$

$$\le n + 3\,n / 4 + 9\,n / 16 + 27\,n / 64 + \ldots + 3^{\log_4 n}\Theta(1)$$

$$\{\text{as } a^{\log_b n} = n^{\log_b a}\}$$

$$\le n\,(\textstyle\sum i \mid 0 \le i < \infty \bullet (3/4)^i) + \Theta(n^{\log_4 3})$$

{decreasing geometric series:

$(\sum k \mid 0 \le k < \infty \bullet x^k) = 1 / (1 - x)$}

$$\le 4\,n + \Theta(n^{\log_4 3})$$

$$\{\log_4 3 < 1\}$$

$$= 4\,n + o\,(n)$$

$$= O\,(n)$$

CSE@UTA

Master Method

- Let $a \ge 1$, $b > 1$ be constants and $f(n)$ be a asymptotically positive function. Assume

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

  Then there are three common cases.

Compare $f(n)$ with $n^{\log_b a}$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$,
   - $f(n)$ grows *polynomially slower* than $n^{\log_b a}$ (by an n^{ε} factor).

   Solution: $T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a})$
   - $f(n)$ and $n^{\log_b a}$ grow at the same rate.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg n)$

Master Method

3. $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some $\varepsilon > 0$
   - *f*(*n*) grows *polynomially faster* than $n^{\log_b a}$ a (by an n^{ε} factor) and f(n) satisfies the regularity condition that $a\,f(n/b) \le c\,f(n)$ for some $c < 1$ and sufficiently large n.

   Solution : $T(n) = \Theta(f(n))$

- Note 1: This theorem can be applied to divide-and-conquer algorithms, which are all of the form
  $$T(n) = a\,T(n/b) + D(n) + C(n)$$
  where D(n) is the cost of dividing and C(n) the cost of combining.
- Note 2: Not all possible cases are covered by the theorem.

Merge Sort with the Master Method

- For arbitrary $n > 0$, the running time of Merge-Sort is
  $$T(n) = \Theta(1) \quad \text{if } n = 1$$
  $$T(n) = T(\text{floor}(n/2)) + T(\text{ceiling}(n/2)) + \Theta(n) \quad \text{if } n > 1$$

  We can approximate this from below and above by
  $$T(n) = 2\,T(\text{floor}(n/2)) + \Theta(n) \text{ if } n > 1$$
  $$T(n) = 2\,T(\text{ceiling}(n/2)) + \Theta(n) \text{ if } n > 1$$

  respectively. According to the Master Theorem, both have the same solution which we get by taking
  $$a = 2,\ b = 2,\ f(n) = \Theta(n)\ .$$

  Since $n = n^{\log_2 2}$, the second case applies and we get:
  $$T(n) = \Theta(n \lg n)$$

Binary Search with the Master Method

- The Master Theorem allows us to ignore the floor or ceiling function around n/b in T(n/b) in general.

- Binary Search has for any n > 0 a running time of

  $T(n) = T(n/2) + \Theta(1)$.

  Hence $a = 1$, $b = 2$, $f(n) = \Theta(1)$.
  Since $1 = n^{\log_2 1}$ the second case applies and we get:

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

Examples

e.g. $T(n) = 4T(n/2) + n$
$a = 4$, $b = 2$ => $n^{\log_b a} = n^2$; $f(n) = n$.
CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.
hence, $T(n) = \Theta(n^2)$.

e.g. $T(n) = 4T(n/2) + n2$
$a = 4$, $b = 2$ => $n^{\log_b a} = n^2$; $f(n) = n^2$.
CASE 2: $f(n) = \Theta(n^2\lg^0 n)$, that is, $k = 0$.
hence, $T(n) = \Theta(n^2\lg n)$.

Examples

e.g. $T(n) = 4T(n/2) + n^3$

$a = 4$, $b = 2$ => $n^{\log_b a} = n^2$; $f(n) = n^3$.

CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(cn/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

hence, $T(n) = \Theta(n^3)$.

e.g. $T(n) = 4T(n/2) + n^2/\lg n$

$a = 4$, $b = 2$ => $n^{\log_b a} = n^2$; $f(n) = n^2/\lg n$.

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\lg n)$.

Monge Array

An mxn array A of real numbers is an Monge Array if for all i, j, k and l, such that

$1 \le i \le k \le m$ and

$1 \le j \le l \le n$.

we have,

$$A[i,j] + A[k,l] \le A[i,l] + A[k,j]$$

Monge Array

		j	l	
	10	17	13	28
i	17	22	16	29
k	24	28	22	34
	11	13	6	17
	45	44	32	37

$$\boxed{A[i,j] + A[k,l] \le A[i,l] + A[k,j]}$$

Problem 4-7 (CLRS) Monge Arrays

a) Problem Statement:

Prove that an array is Monge, if and only if for all $i = 1,2,\ldots,m-1$ and $j = 1,2,\ldots,n-1$, we have

$$A[i,j] + A[i+1,j+1] \le A[i,j+1] + A[i+1,j]$$

Problem 4-7 (CLRS) Monge Arrays

Solution :

- First we prove "*only if*" part i.e. assume that array A is Monge.
  a) Set k = i+1 and l = j+1
  b) By the Monge property we have
     A[i,j] + A[i+1,j+1] ≤ A[i,j+1] + A[i+1,j]

Problem 4-7 (CLRS) Monge Arrays

- Now we prove the "*if*" part by induction on rows and columns.

We are assuming that (Induction Hypothesis)
A[i,j] + A[i+1,j+1] ≤ A[i,j+1] + A[i+1,j] is true
for all i and j in the appropriate ranges.

Want to Prove that this implies
A[i,j] + A[k,l] ≤ A[i,l] + A[k,j] ,
for all i<k and j<l

Problem 4-7 (CLRS) Monge Arrays

- First we do the row induction:
  - Trying to prove that for all i, j and k>i
    $A[i,j] + A[k,j+1] \le A[i,j+1] + A[k,j]$ …(1)

  The base case k= i+1 is true by our assumption.

  For the inductive case, assume that (1) is true for all k such that $k - i \le c$.

Problem 4-7 (CLRS) Monge Arrays

- Then we have

$$A[k,j]+A[k+1,j+1] \le A[k,j+1]+A[k+1,j]$$

and

$$A[i,j] + A[k,j+1] \le A[i,j] + A[k,j]$$

So by adding the above equations we get

$$A[i,j] + A[k+1,j+1] \le A[i,j] + A[k+1,j]$$

Problem 4-7 (CLRS) Monge Arrays

- So we have proved that for all k
  such that k - i ≤ c + 1.
- Similarly we do induction on columns and we get that for any k>i, l>j we have
  - A[i,j] + A[k,l] ≤ A[i,l] + A[k,j]
- And so A is Monge.

Problem 4-7 (CLRS) Monge Arrays

b) **Problem Statement:** Given that the adjacent array is not Monge. Change one element to make it Monge ?

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

Solution:

Changing the 7 to a 5 is one possible solution

Problem 4-7 (CLRS) Monge Arrays

c) **Problem Statement:**

Let f(i) be the index of the column containing the leftmost minimum element of row i.

Prove that:

$f(1) \leq f(2) \leq \ldots \leq f(m)$ for any m x n Monge array.

Solution:

We Solve this problem using proof by *CONTRADICTION*.

Problem 4-7 (CLRS) Monge Arrays

- Assume that the property is untrue. Let i be the smallest value for which $f(i) > f(i+1)$.

  By the properties of f we have

  $$A[i,f(i+1)] > A[i,f(i)]$$

  and $$A[i+1,f(i)] \geq A[i+1,f(i+1)]$$

- Adding the above equations together,

$$A[i,f(i+1)] + A[i+1,f(i)] > A[i,f(i)]+A[i+1,f(i+1)]$$

which is a contradiction to the Monge property.

Problem 4-7 (CLRS) Monge Arrays

d) Problem Statement:

Here is a description of a divide-and-conquer algorithm that computes the left-most minimum element in each row of an m x n Monge array A:

Construct a sub-matrix A′ of A consisting of the even-numbered rows of A. Recursively determine the leftmost minimum for each row of A′. Then compute the leftmost minimum in the odd-numbered rows of A.

Explain how to compute the leftmost minimum in the odd-numbered rows of A(given that the leftmost minimum of the even-numbered rows is known) in O(m+n) time.

Problem 4-7 (CLRS) Monge Arrays

Solution:

Say the previous step returns an array F′[1,…floor(m/2)] in which F′[i] = $f_{A'}$ (i), for all i = 1,…floor(m/2).

We want to produce F[1,…m] such that F[i] = f_A(i), for all i = 1,…m.

We do the following :

For i = 1…m

if i is even then

let F[i] = F′[i/2].

Else

scan the elements A[i, F′[(i-1)/2]] through A[i, F′[(i+1)/2]] to find the leftmost min., then set F[i] to be its column index.

Problem 4-7 (CLRS) Monge Arrays

- For the running time note that the outer loop is executed m/2 times, once for each odd i and a total of

$$\sum F'[i+1] - F'[i]+1 \leq m/2 + n$$

(for i= 1 to m/2)

Array elements are examined by the inner loop over the entire run of the algorithm.

Therefore, the running time is $O(m + n)$.

Problem 4-7 (CLRS) Monge Arrays

e) **Problem Statement:**

Finding the recurrence for running time of algorithm in part (d).

Solution: Let T(m,n) be the running time of the algorithm on an m x n array.

For some constant $c > 0$,

$$T(m,n) = T(m/2, n) + c(m+n)$$

and $T(1, n) = O(n)$ for all n. This gives

$$T(m,n) = O(m+n \log m).$$

This can be verified by *SUBSTITUTION*.

RESULTS : LAB

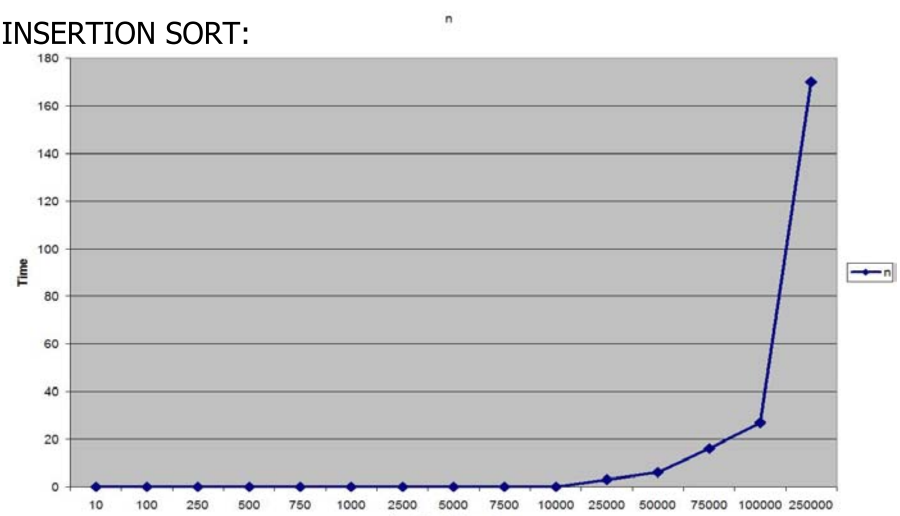
Output:

- Execution Time :
- ---

Number	Insertion Sort	Selection Sort
10	0	0
100	0	0
250	0	0
500	0	0
750	0	0
1000	0	0
2500	0	0
5000	0	0
7500	0	1
10000	0	0
25000	3	2
50000	6	7
75000	16	16
100000	27	28
250000	170	177

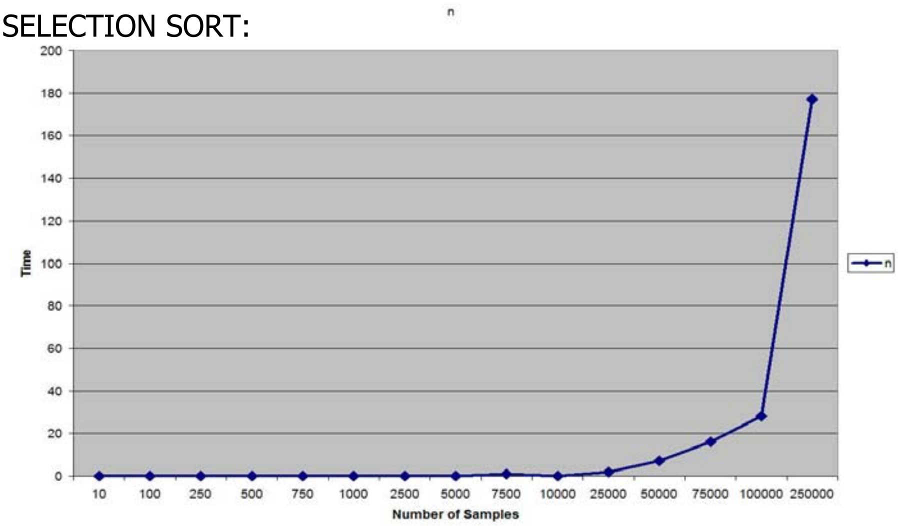
RESULTS : LAB

INSERTION SORT:



RESULTS : LAB

SELECTION SORT:



References

- http://www.ma.hw.ac.uk/~simonm/ae/ae.pdf
- http://appsrv.cse.cuhk.edu.hk/~csc2520/lecture/L11-2.doc
- http://math.hws.edu/eck/cs327/
- http://www.cs.umbc.edu/courses/undergraduate/CMSC202/fall01/202.webnotes.asymptotic_analysis.htm
- http://www.cs.umbc.edu/courses/undergraduate/CMSC202/fall02/Lectures/AA/

References

- http://www.csd.uwo.ca/courses/CS340b/bigo-4.pdf
- http://www.cs.berkeley.edu/~oakamil/su03/docs/5-1.html
- http://www.cs.fsu.edu/~cop4531/slideshow/
- http://www.cs.jhu.edu/~goodrich/dsa/trees/btree.html
- http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/heapsort.html
- http://www.cse.ucsc.edu/classes/cmps101/Spring04/recurrence.pdf

Problem 4-6 (CLRS) VLSI Chip Testing

(a) The first observation is that any decision about whether a chip is good or bad must be based on what the good chips say.

(b) Consider a chip x that we do not know is good or bad. Ask all chips in the set whether x is good or bad.

- If there is a majority (> n/2) good chips in the set and x is bad then we may safely conclude that x is in fact bad.
- If there is a majority (> n/2) good chips in the set and x is good then we may safely conclude that x is in fact good.

Problem 4-6 (CLRS) VLSI Chip Testing

- Suppose there are n/2 good chips and n/2 bad chips. The good chips do not lie, so we conclude that each good chip will say that the all other n/2 -1 good chips are in fact good and all n/2 bad chips are in fact bad.
- The bad chips on the other hand will say : each bad chip will say that all other bad chips are good and all good chips are bad.
- Since we have complete symmetry between good and bad chips and there is no majority of good chips , Professor cannot identify any chip as being either good or bad.

Problem 4-6 (CLRS) VLSI Chip Testing

b) Let m_{good} is the number of the good chips and m_{bad} is the number of bad chips.We know that

$$m_{good} + m_{bad} = n \text{ and } m_{bad} \geq m_{good}$$

Suppose that good chips constitute the set A and that bad chips could be split into two sets B and C

Problem 4-6 (CLRS) VLSI Chip Testing

- Now it is easy to see that if

$x, a_1, a_2, \ldots a_k \in B$, $b_1, b_2, \ldots b_l \in A$, $c_1, c_2, \ldots c_p \in C$ then his strategy would declare x to be a good chip because

none of the outcomes of the pairwise tests would change.

Problem 4-6 (CLRS) VLSI Chip Testing

c) Lets enumerate all outcomes of a test in order they appear in the book.

We could use the following algorithm:

- Pick any two chips and test them together
- If we have the first outcome then pick any of the two chips and throw it away. Put the other into the output set.
- If we have any of the other outcomes throw away both chips.
- Repeat this until we have atleast two chips available.
- If we are left with only one chip then add it to the output set if the number of tests with the first outcome is even and throw it away otherwise.

Problem 4-6 (CLRS) VLSI Chip Testing

d) The recurrence for the problem is
   $T(n) = T(n/2) + n/2$
   (assuming n is power of 2).
   Here a=1, b=2, $f(n) = n/2$, $f(n) = \Omega(n^{\varepsilon})$ for any $0 < \varepsilon < 1$ and regularity condition holds if, $c = 3/4$
   So we are in the third case of Master Theorem and hence
   $T(n) = \Theta(n/2) = \Theta(n)$.