DAA

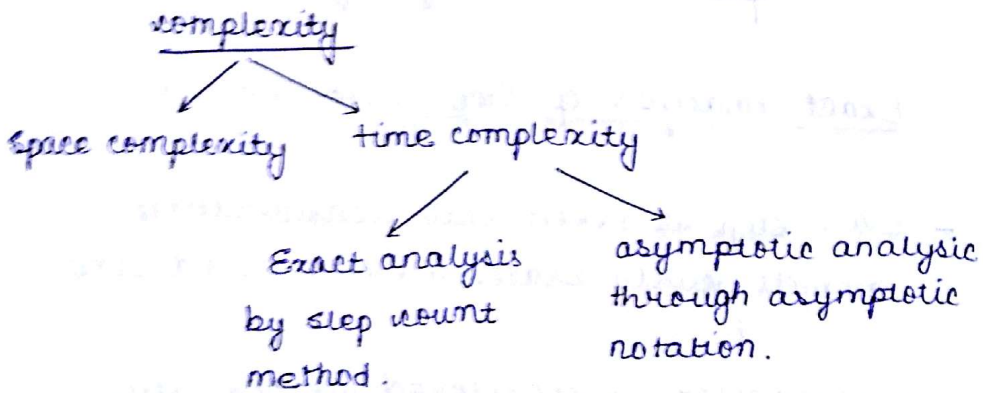
Algorithm:
- Input : zero or more quantities may be supplied externally.
- Output : must have one output
- Finiteness : must terminate after a finite set of steps.
- Effectiveness : By principle of coincidence it gives the output
- No ambiguous instructions

An expression can be expressed in following way :

a. English
b. Pseudo code
c. Real programming language.

Complexity of an Algorithm :

complexity of each algorithm determines how efficiently it will be executed. Complexity of the amount of time and space required by an algorithm for an input of size (let n).



Space complexity : Amt. of space required by an algorithm to run to completion.

expressed as : $S(P) = C + S_p$.

$C \longrightarrow$ constant (no. of distinct var.)

$S_p \longrightarrow$ Instance characteristics

Example 1:

```
Algorithm sum (x,y,z)
{
    z = x + y + z + z * x + y
    return z
}
```

$S(sum) = C + S_{sum}$

$= 3 + 0$

$= 3.$

Example 2 :

```
Algorithm sum1 (a, n)
{
    s ← 0
    for i ← 1 to n
        s ← s + a[i]
    return i ;
}
```

$S(sum1) = C + S_{sum1}$

$= 2 + n$

Time Complexity: It is the amount of time taken by an algorithm to run into completion.

- Here time is expressed as the number of steps/ operations taken by algorithm to be carried out.

Exact Analysis of Step Count Method:

- Each step is taken into consideration.
- comments / curly braces are considered zero.
- assignment is considered as one step.
- s/e : steps per execution

Example 1:

```
Algorithm Sum (x, y, z)
{
    z = x+y+z + z * x + y ;
    return z ;
}
```

s/e	freq	total
1	1	1
1	1	1

total time = 2

Example 2:

```
Algorithm Sum (a, n)
{
    S ← 0 ;
    for i ← 1 to n
        S = S + a[i] ;
    return s ;
}
```

s/e	freq	Total.
1	1	1
1	n+1	n+1
1	n	n
1	1	1

total time = $2n+3$

Example 3:

Finding an element x in an array by linear search

Best time : 1.

worst case : time = n or n+1.

Average case : time = $\frac{n(n+1)}{2n} = \frac{n+1}{2}$.

Q) Arrange in increasing order of growth rate.

~~logn~~.

→ 1 ⟶ const.

logn ⟶ logarithmic

n ⟶ linear

nlogn.

n^2 ⟶ quadratic

n^3 ⟶ cubic

2^n ⟶ exp.

n! ⟶ factorial.

Q) 2^{2^n}, $2^{\sqrt{n}}$, $\sqrt{n}\log n$, $n\log\sqrt{n}$, $n!$, $\log(n!)$, $\log\sqrt{n}$.

$\log\sqrt{n}$

$\sqrt{n}\log n$.

$n\log\sqrt{n}$

$\log(n!)$

$2^{\sqrt{n}}$.

n!

2^{2^n}.

TIME COMPLEXITY: (Approximate Analysis)

Asymptotic notation

a) Big-oh (O) Notation

b) Big-omega (Ω) Notation

c) Big-theta (θ) Notation

d) small-oh (o) Notation

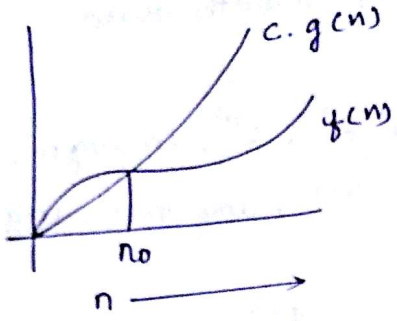
e) small-omega (ω) Notation.

O-Notation

Let $f(n)$ & $g(n)$ are two +ve functions. we can say $f(n)$ is said to be big-oh of $g(n)$ if there exists two +ve constants c and n_0 and

$$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0.$$

$\Rightarrow$ $\boxed{f(n) = O(g(n))}$



Eg: Represent in O-Notation

a) $f(n) = 2n + 3$

b) $f(n) = 3n^2 + 2n + 5$

c) $f(n) = 3n^2 + \log n + n$

d) $f(n) = 3n^2 + 2^n + 5$

e) $f(n) = 2n^2 + |n| + 1$

\# $f(n) = 2n + 3$

$2n + 3 \leq 2n + 3$.

$2n + 3 \leq 2n + n$ $(\because n \geq 3)$

$2n + 3 \leq 3n$.

$f(n)$ c $g(n)$

$\Rightarrow f(n) = O(n)$ where $c = 3$ & $\forall n \geq 3$.

\# $f(n) = 3n^2 + 2n + 5$.

$3n^2 + 2n + 5 \leq 3n^2 + 2n + 5$

$3n^2 + 2n + 5 \leq 3n^2 + 2n + n$ $(\because n \geq 5)$.

$3n^2 + 2n + 5 \leq 3(n^2 + n)$ $3n^2 + n^2$ $(\because n^2 \geq 3n)$

$3n^2 + 2n + 5 \leq 4n^2$

$f(n)$ c $g(n)$

$f(n) = O(n^2)$

eg $f(n) = 2^n + n! + 1 = O(n!)$

2.) $2^{2n} = O(2^n)$? Justify

Method 1 : let $2^{2n} = O(2^n)$ ⟶ Valid

$\Rightarrow 2^{2n} \leq c \cdot 2^n$.

$\Rightarrow 2^n \cdot 2^n \leq c \cdot 2^n$.

$\Rightarrow 2^n \leq c$. ⟵ Invalid

~~Method 2~~

3.) $n! = O(n^n)$? Justify

⟶ :

$1 \leq n$

$2 \leq n$

$3 \leq n$

$\vdots$

$n \leq n$

$n! \leq n^n$

$\Rightarrow n! \leq 1 \cdot n^n$

$f(n)$ ↑ c ↑ $g(n)$

$n! = O(n^n)$.

for $c = 1$ & $n_0 = 1$.

4.) $2^{2n+1} = O(2^n)$? Justify

$2^{n+1} \leq c \cdot 2^n$. ⟶ Valid

$2^n \cdot 2 \leq c \cdot 2^n$.

$2 \leq c$ ⟶ Valid

$c = 1$ $n_0 = 1$.

5) $3n+1 = O(n^2)$ T/F

Eg

→ Let T

$3n+1 = O(n^2)$

$3n+1 \leq cn^2$

$\frac{3}{n} + \frac{1}{n^2} \leq c$ $\quad c = 3$, $n_0 = 2$

for some value of C and taking values on $n = 1, 2 \ldots$ it will be val

d.f $f(n) = 3n^2 + 2^n + 5$

$3n^2 + 2^n + 5n^2 \leq 3n^2 + 2^n + 5n^2$ $\quad (5n^2 \geq 5)$

$\leq 8n^2 + 2^n$

$\leq 2^{2n} + 2^n$ $\quad (2^{2n} \geq 8n^2)$

$\leq 2^n (8+1)$

$\leq 9 \cdot 2^n$

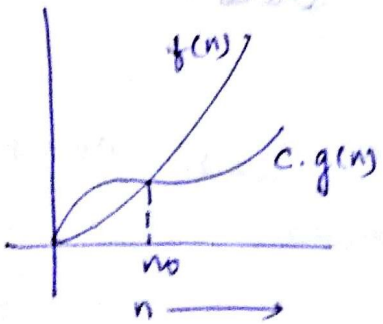
$\hookrightarrow c$

7/12/16

Ω - Notation :

For any two functions $f(n)$ and $g(n)$ which are non-negative for all $n \geq 0$, $f(n)$ is said to be big omega of $g(n)$ $f(n) = \Omega g(n)$ if there exists two +ve constants

$$0 \leq c \cdot g(n_0) \leq f(n) \quad \text{for all } n \geq n_0$$



It represents lower bound or best case time complexity.

Eg 1)) $f(n) = 3n^2 + 2n + 1$ represent in Ω notation ?

$3n^2 + 2n + 1 \geq 3n^2 + 2n + 1$

$\Rightarrow 3n^2 + 2n + 1 \geq 3n^2 + 2n$.

$\Rightarrow 3n^2 + 2n + 1 \geq 3n^2$ [∵ growth rate of $n^2 > n$ & $n^2 > 1$]

$f(n) \geq c \quad g(n)$

$f(n) = \Omega(n^2)$

2)) $f(n) = 2^n + n\log n + 5$ in Ω notation ?

$f(n) = \Omega(2^n)$

$f(n) = \Omega(n \log n)$ ✓ [∵ lower values of 2^n are permitted]

$f(n) = \Omega(1)$ ✓

3)) $2^{2n} = \Omega(2^n)$ Justify ?

let $c \cdot 2^n \leq 2^{2n}$ ⟶ Valid ✓

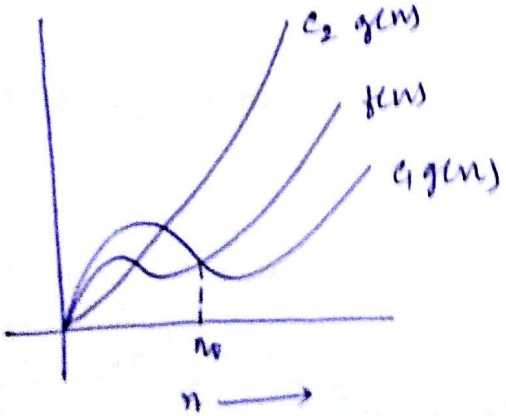
$c \cdot 2^n \leq 2^n \cdot 2^n$

$c \leq 2^n$ ⟶ Valid ✓

θ - Notation Here a specific value is found out for $g(n)$ that satisfies both conditions.

$f(n) = \theta\, g(n)$

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.



Q) $f(n) = 3n^2 + 2n + 3$

$3n^2 + 2n + 3 \leq 3n^2 + 2n + 3$

$3n^2 + 2n + 3 \leq 3n^2 + 3n$

$\leq 3n^2 + n^2$

$\leq 4n^2$

c_2 , $g(n)$

$f(n) \leq O(n^2)$ $\quad f(n) \leq 4\, g(n)$

$3n^2 + 2n + 3 \geq 3n^2 + 2n + 3$

$\geq 3n^2 + 2n$

$\geq 3n^2$

c_2 , $g(n)$

$f(n) = \Omega(n^2)$ $\quad f(n) \geq 3\, g(n)$

$3\, g(n) \leq f(n) \leq 4\, g(n)$

$f(n) = \Theta(n^2)$

Insertion Sort:

8/12/16

```
/* Assume array starts at index 1 */
INSERTION-SORT (A, n)                                   cost        freq
{
    for j ← 2 to n ------------------------------------ c1          n
    {
        key ← A[j] ---------------------------------- c2          n-1
        i ← j-1 ------------------------------------- c3          n-1
        while (i>0 and a[i] > key) ------------------ c4          Σ_{j=2}^{n} tj
        {
            A[i+1] = A[i] ---------------------------- c5          Σ_{j=2}^{n} (tj-1)
        }
        A[i+1] = key --------------------------------- c6          n-1
    }
}
```

Analysis:

Time = $T(n) = \sum_{\text{all terms}}$ (cost of each statement). (freq. of each statement)

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6(n-1)$$

Best case:

- This occurs when data are already sorted.

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} 1. \qquad [t_j = 1] \qquad \sum_{j=2}^{n} (t_j - 1) = 0.$$

$$= n - 1.$$

$$\therefore T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \cdot (n-1) + c_6(n-1)$$

$$= c_1 n + (n-1)[c_2 + c_3 + c_4 + c_6].$$

$$= n(c_1 + c_2 + c_3 + c_4 + c_6) - (c_2 + c_3 + c_4 + c_6).$$

$$= An - B.$$

$$= \Omega(n)$$

worst case :

This occurs when data is already sorted in reverse order.

$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} (j)$ $\quad \sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1)$.

~~$T_n = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 (n-1) + c_5 (n-2) + c_6 (n-1)$~~

$$T_n = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \left(\frac{n(n+1)}{2} - 1 \right) + c_5 \left(\frac{n(n-1)}{2} \right) + c_6 (n-1).$$

$$= n(c_1 + c_2 + c_3 + c_4 - c_5 + c_6) + n^2 \left(\frac{c_4}{2} + \frac{c_5}{2} \right) - (c_2 + c_3 + c_6)$$

$$= An^2 + Bn + C$$

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

9/12/16 Re

→ runn
of w

Sowing v

a) Iterat
b) Recu
c) Maste
d) Subt
e) chan

a) Itera

→ Here
we

→ The
com

a)

b)

9/12/16 **Recurrence** : it is an equation or inequality that describes a function in terms of itself with smaller inputs.

DMS

→ running time of recurrence can be expressed in terms of recurrence.

Solving recurrence : methods :

a) Iterative method
b) Recurrence tree method
c) Master theorem.
d) substitution method.
e) change of variables.

a) **Iterative method** :-

→ Here the recurrence terms in def^n part are expanded till we find a regular pattern.

→ The regular pattern may refer to some a.p or g.p or may come under the following mathematical formulae.

a) $a^{\log_b c} = c^{\log_b a}$

b) $\sum_{k=0}^{n} a^k = \frac{a^n - 1}{a - 1}$ if $a > 1$.

$= \frac{1 - a^n}{1 - a}$ if $a < 1$.

Example (Iterative method)

a) $T(n) = T(n-1) + c$, $T(0) = 0$

↑ recurr — ↑ base case.

Soln

$T(n) = T(n-1) + c$

$= [T(n-2) + c] + c$

$= [[T(n-3) + c] + c] + c.$

$= T(n-n) + nc$

$= T(0) + nc$

$= nc$

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

$= 2[2T(n/2^2) + 1] + 1$

$= 2^2 T(n/2^2) + 2 + 1.$

$= 2^2 [2T(n/2^3) + 1] + 1.$

$= 2^3 T(n/2^3) + 4 + 2 + 1.$

$= 2^3 [2T(n/2^4) + 1] + 4 + 2 + 1.$

$= 2^4 T(n/2^4) + 8 + 4 + 2 + 1.$

$= 2^i T(n/2^i) + 2^{i-1} + \ldots\ldots + 1.$

$= 2^{\log_2 n}(1) + \frac{2^{i-1} - 1}{2 - 1}$

$= 2^{\log_2 n} + \frac{2^{\log_2 n - 1} - 1}{1}$

$= \frac{3n - 2}{2}.$

8) $T(n) = 3T(n/2) + n$, $T(1) = 1$

$= 3\left[3T(n/2^2) + n/2\right] + n$

$= 9T(n^2/2^2) + \frac{3n}{2} + n$

$= 9\left[3T(n/2^3) + n/2^2\right] + \frac{3n}{2} + n$

$= 3^3 T(n/2^3) + 3^2\frac{n}{2^2} + \frac{3n}{2} + n$

$= 3^i T(n/2^i) + \left[\left(\frac{3}{2}\right)^{i-1} + \left(\frac{3}{2}\right)^{i-2} + \ldots + \left(\frac{3}{2}\right)^1 + \left(\frac{3}{2}\right)^0\right] n$

let $n/2^i = 1$ $\quad i = \log_2 n$

$= 3^{\log_2 n} T(1) + n \sum_{k=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^k$

$= n^{\log_2 3} + n\left[\frac{(3/2)^{\log_2 n} - 1}{3/2 - 1}\right]$

$= n^{1.5} + 2n\left[\frac{3^{\log_2 n} - 2^{\log_2 n}}{2^{\log_2 n}}\right]$

$= n^{1.5} + 2n\left[\frac{n^{1.5} - n}{n}\right]$

$= 2n^{1.5} - 2n$.

$= O(n^{1.5}) = O(n^2)$.

16/12/16

4) $T(n) = 4T(n/2) + n \qquad T(1) = 1.$

$= 4\left[4T(n/2^2) + n/2\right] + n.$

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

$= 4^2\left[4T(n/2^3) + n/2^2\right] + n/2 + n.$

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

$= 4^i T(n/2^i) + n \sum_{n=0}^{\log_2^n - 1} (2)^k$

$= n^{\log_2^4} + n \sum_{n=0}^{\log_2^n - 1} (2)^k$

$= n^2 + n\left(2^{\frac{\log n - 1}{2-1}}\right)$

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

$= O(n^2)$

0 —

1 —

2 —

3 —

i —

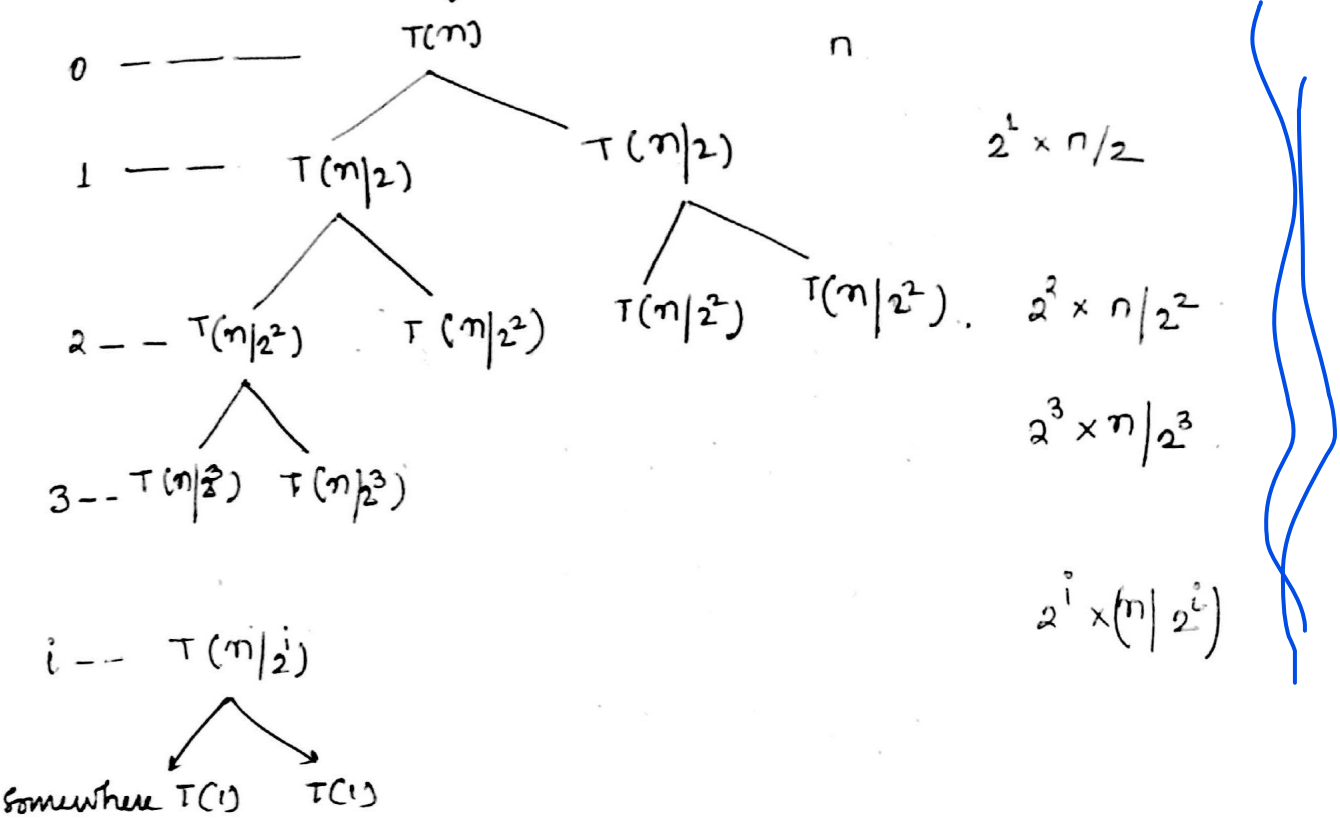
Somewh

⇒

T(n)

16/12/16 RECURRENCE TREE

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



$\Rightarrow \frac{n}{2^i} = 1 \Rightarrow i = \log_2 n$

$T(n)$ = Recc cost at last level + non recc cost

= last level cost + cost from level 0 to last level − 1

$= 2^{\log_2 n} \times T(1) + (n + n + n \ldots\ldots + \log_2 n)$

$= n + \sum_{i=0}^{\log_2(n-1)} 2^i = n + \frac{2^{\log_2 n} - 1}{2-1} = n + n - 1 = 2n - 1.$

5/1/17

Q) Solve. $T(n) = 2T(n/2) + n^2$

Soln By iterative method.

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

$$= 2\left[2T(n/2^2) + (n/2)^2\right] + n^2$$

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

$$= 2^2\left[2T(n/2^3) + (n/2^2)^2\right] + (n/2)^2$$

$$= 2^3 T(n/2^3) + 2^2(n/2^2)^2 + 2(n/2)^2 + n^2$$

$$= 2^i T(n/2^i) + \left(\frac{1}{2^{i-1}} + \frac{1}{2^{i-2}} + \cdots \frac{1}{2^1} + \frac{1}{2^0}\right) n^2$$

$$= 2^i T(n/2^i) + n^2 \sum_{n=0}^{i-1} (1/2)^k.$$

$$\frac{n}{2^i} = 1. \Rightarrow i = \log_2 n$$

Putting value of i

$$= 2^{\log_2 n} T(1) + n^2 \sum_{k=0}^{\log_2 n - 1} (1/2)^k.$$

$$= n + 2n^2\left(\frac{1 - (1/2)^{\log_2 n - 1 + 1}}{1 - 1/2}\right)$$

$$= n + 2n^2\left(\frac{2^{\log_2 n} - 1}{2^{\log_2 n}}\right)$$

$$= n + 2n^2\left[\frac{n-1}{n}\right]_2$$

$$= n + 2n^2 - 2n$$

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

~~By recursive tree~~

HA a) $T(n) = 3T(n/2) + n \quad T(1) = 1$

b) $T(n) = 3T(n/2) + n^2$

c) $T(n) = 3T(n/4) + n^2$

d) $T(n) = 2T(n/4) + n^2$

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

MASTER THEOREM: (To solve recurrence).

10/1/17.

It applies to the recurrence of following form:

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

where $a \geq 1$ and $b > 1$ are constants

$f(n)$ is asymptotically +ve function

Then $T(n)$ can be asymptotically bounded as follows:

— There are three cases:

case 1. if $f(n) = O(n^{\log_b a - \epsilon})$ for $\epsilon > 0$

then $T(n) = \theta(n^{\log_b a})$

case 2: if $f(n) = \theta(n^{\log_b a})$ then

$T(n) = \theta(n^{\log_b a} \log n)$

case 3: if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$

and $a f(a/b) \leq c \cdot f(n)$ where $c < 1$

then $T(n) = \theta(f(n))$.

Ex 1 Solve $T(n) = 9T(n/3) + n$

Ans finding $n^{\log_b^a} = n^{\log_3^9} = n^2$

comparing $n^{\log_b^a}$ with $f(n)$ we get

$n^{\log_a^b}$ is asymptotically larger Hence.

As per case 1.

$f(n) = O(n^{\log_b^a - \epsilon})$

$\Rightarrow n \leq c \cdot n^{2-\epsilon}$

$\Rightarrow n \leq 2 \cdot n^{1.5}$ [Taking value $c=2$ & $\epsilon=0.5$]

$$\boxed{T(n) = \theta(n^{\log_a^b}) = \theta(n^2)}$$

Ex 2 Solve $T(n) = T(2n/3) + 1$

Ans $n^{\log_b^a} = n^{\log_{3/2}^1} = n^0 = 1$.

On comparing we get $n^{\log_b^a} = f(n)$

According to case 2. $f(n) = \theta(n^{\log_b^a})$

$\Rightarrow f(n) \geq c \cdot n^{\log_b^a}$

$1 \geq c \cdot 1$

$T(n) = \theta(n^{\log_b^a} \log n)$

$$\boxed{T(n) = \theta(\log n)}$$

Ex 3 $T(n) = 3T(n/4) + n \log n$

Ans $n^{\log_b a} = n^{\log_4 3} = n^{0.78}$

$f(n)$ is asymptotically larger

According to case 3 $\therefore f(n) = \Omega(n^{\log_b a + \epsilon})$

$f(n) \geq c \cdot n^{0.78 + 0.22}$

$n \log n \geq 2n$

Now $a f(n/b) \leq c \cdot f(n)$

$\Rightarrow \frac{3n}{4} \log \frac{n}{4} \leq c \cdot n \log n$

$\Rightarrow \log \frac{n}{4} \leq \log n$. valid

$\therefore T(n) = f(n)$

$$\boxed{T(n) = \theta(n \log n)}$$

HA · (Any)

① $T(n) = 7T(n/2) + n^2$

② $T(n) = 3T(n/2) + n \log n$.

③ $T(n) = 16T(n/4) + n^2$

④ $T(n) = 2T(n/4) + \sqrt{n}$

⑤ $T(n) = 2T(n/2) + n^3$

⑥ $T(n) = T(9n/10) + n$.

⑦ $T(n) = 7T(n/3) + n^2$

⑧ $T(n) = 4T(n/2) + n^2\sqrt{n}$.

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

$a = 2^n$, $b = 2$

$n^{\log_b^a} = n^{\log_2 2^n} = n^n$.

$n^{\log_b^a} = f(n)$

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

~~$T(n) = \Theta(n^n \log n)$~~.

~~$T(n) = 2T(\sqrt{n})$~~.

$T(n) = 2T(n/2) + \frac{n}{\log n}$.

$n^{\log_2^2} = n$

$\frac{n}{\log n} < f(n)$ $\quad \frac{n}{\log n} > f(n)$

$\frac{n}{\log n} < f(n)$.

$f(n) = O(n^{\log_b^a - \epsilon})$

$\frac{n}{\log n} < c \cdot n^{1-\epsilon}$.

$\frac{1}{\log n} < \frac{1}{n^\epsilon}$.

$n^\epsilon < \log n$

$f(n) = \Omega(n^{\log_b^a + \epsilon})$.

$f(n) \geq c \cdot n^{1+\epsilon}$

$f(n) \geq c \cdot n^{1.5}$

$\frac{n}{\log n} \geq n \cdot n^\epsilon$.

$\frac{1}{\log n} \geq n^\epsilon$.

Solving recurrence by changing variable method

Ex: $T(n) = 2T(\sqrt{n}) + \log n$.

$m = \log n \Rightarrow 2^m = n \Rightarrow 2^{m/2} = n^{1/2} = \sqrt{n}$.

$T(2^m) = 2T(2^{m/2}) + m$

$S(m) = T(2^m)$

$S(m/2) = T(2^{m/2})$

Now $S(m) = 2S(m/2) + m$.

$a = 2 \quad b = 2 \quad f(m) = m$.

$m^{\log_b a} = m$.

$S(m) = \theta(m^{\log_b a} \log m) = \theta(m \log m)$.

$T(2^m) = \theta(m \log m)$.

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

Q. $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$.

→ Dividing by n.

$\frac{T(n)}{n} = \frac{1}{\sqrt{n}} T(\sqrt{n}) + 1$.

let $m = \log n \Rightarrow 2^m = n \Rightarrow 2^{m/2} = n^{1/2} = \sqrt{n}$.

Now $\frac{T(2^m)}{2^m} = \frac{T(2^{m/2})}{2^{m/2}} + 1$.

let $\frac{T(2^m)}{2^m} = S(m)$

then $S(m) = S(m/2) + 1$.

$m^{\log_b a} = m^0 = 1$

$n^{\log_b a} = f(n)$.

~~$S(n) = \theta(n^{\log_b a} \log n)$~~

~~$1 \le c \cdot (n^{\log_b a - \epsilon})$~~

~~$1 \le c \cdot (n^{0 - \epsilon})$~~

$S(m) = \theta(m^{\log_b a} \log m)$

$\Rightarrow \frac{T(2^m)}{2^m} = \theta(\log m)$

$\Rightarrow \boxed{T(n) = \theta(n \log\log n)}$

Substitution method

a) First guess the form of solution

b) applying substitution to find constants.

Ex $T(n) = 2T(n/2) + n.$

a) let solution be $T(n) = O(n \log n)$

b) assuming that solution works for $n/2$

$\therefore T(n/2) \leq c \cdot \frac{n}{2} \log \frac{n}{2}$

Now checking for n.

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

$= 2.\left[\frac{n}{2} \cdot c \cdot \log \frac{n}{2}\right] + n.$

$= c.n \log \frac{n}{2} + n.$

$= cn \log n - cn \log 2 + n$

$= cn \log n - cn + n.$

$T(n) \leq cn \log n.$

$\boxed{T(n) = O(n \log n)}$

DIVIDE

The app

Duivide

Conque

combi

Psuedoc

Binary

{ if

{

q

if

u

e

u

}.

let

num

DIVIDE and CONQUER Method

The approach is

Divide the problem into subproblems

Conquer the subproblems by solving the subproblems recursively

combine ~~conte~~ the solutions into one.

Psuedocode:

Array name

lower bound.

upper bound.

```
Binary_search (A, P, r, key)
{   if (P ≤ r)
    {
    q ← (P+r) /2
    if (A [q] == key)
    return q ;
    else if (key < A[q])
    return binary_search (A, P, q-1, key);
    else
    return binary_search (A, q+1, r, key);
}.
```

Let Binary_search (A, P, r, key) takes T(n) Time n is the number of elements

$P = 0 \qquad r = n-1.$

No. of elements $= r - P + 1 = n - 1 + 0 + 1 = n.$

$T(n) = 2T(n/2) + 1$

(solving this we get $T(n) = O(\log n)$.

Q) Let A & B are two arrays having m & n elements respectively. Both sorted in ascending order. Write the merge procedure to combine the two arrays A & B into C in ascending order.

```
MERGE (A, M, B, N, C)
{
  i ← 1 ; j = 1 , k = 1 ;
  while (i ≤ m & j ≤ n)
  {
    if (a[i] < b[j]
    {
      c[n] ← a[i];
      i = i + 1 ;
      k = k + 1 ;
    }
    else
    {
      c[n] ← b[j]
      j = j + 1;
      k++;
    }
  }
  while (i ≤ n)
  {
    c[n] ← A[i];
      i++ ;
      k++;
  }
  while ( j ≤ n)
  {
    c[n] ← b[i];
     j++;
     k++;
  }
}
```

Q) Elements of array a is sorted in ascending order, array b is sorted in descending order. write an array c to merge a & b and input value in descending order.

MERGE SORT

To sort an array A[p....r]

a) Divide : By splitting the array into two subarrays A[p....q] and A[q+1....r] where q is the halfway point.

b) Conquer : By sorting the subArrays A[p...q] and A[q+1...r]

c) Combine : the sorted subarrays A[p...q] and A[q+1...r] into one sorted subarray.

```
MERGESORT (A, P, r)
{  if (P<r)
   { q ← (P+r)/2
     MERGESORT (A, P, q);
     MERGESORT (A, q+1, r);
     MERGE (A, P, q, r);
   }
}
```

divide

```
MERGE (A, P, q, r)
{
     n1 ← q-P+1
     n2 ← r-q

  // create two temp. arrays L(n1+1) & R[n2+1]
        for i ← 1 + n1
        L[i] ← A[p+i-1]
```

```
for j ← 1 to n2
    R[j] ← A[q+j]
L[n1+1] = ∞
R[n2+1] = ∞
i ← 1
j ← 1.

for k = p to r
{
    if L[i] < R[j]
    {
        A[k] = L[i];
        i ← i+1;
    }
    else
    {
        A[k] = R[j]
        j = j+1;
    }
}
```

17/1/16

QUICK SORT

To sort an

Divide P

S

CONQUER

PARTITIO

{

i ←

x ←

for

{

{

A[

ret

}

QUICK

{

}

}

H4

Rewrite merge procedure without taking any large value.

17/1/16

QUICK SORT

To sort an array A[P...r]

Divide Partition the arrays into A[P, q-1] and A[q+1, r] such that

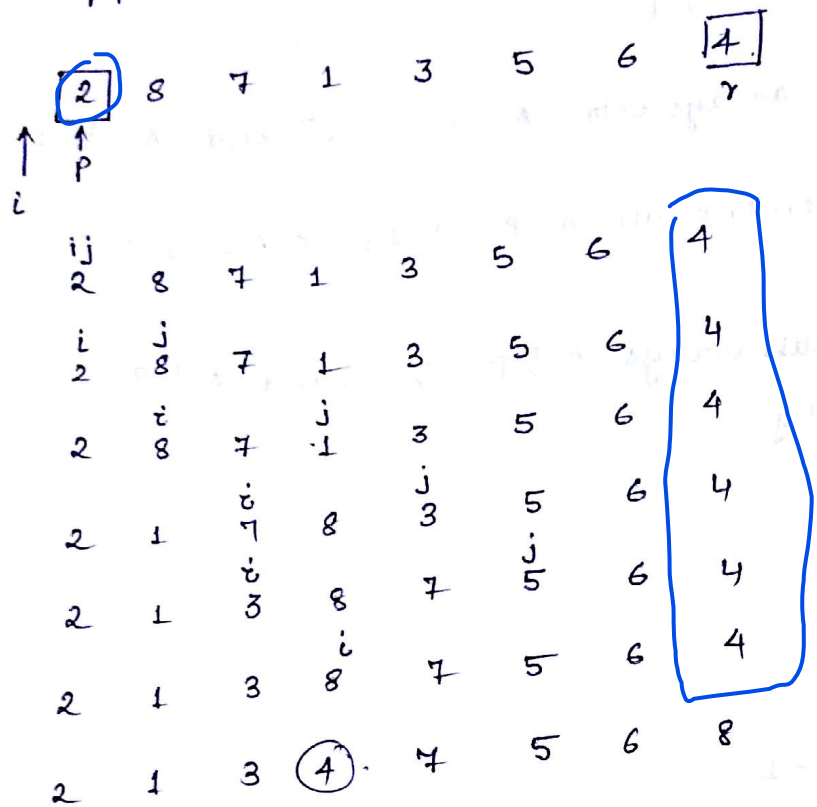
Any element in A[P..q-1] ≤ A[q] ≤

CONQUER Sort the subarrays A[P..q-1] and A[q+1...r] recursively.

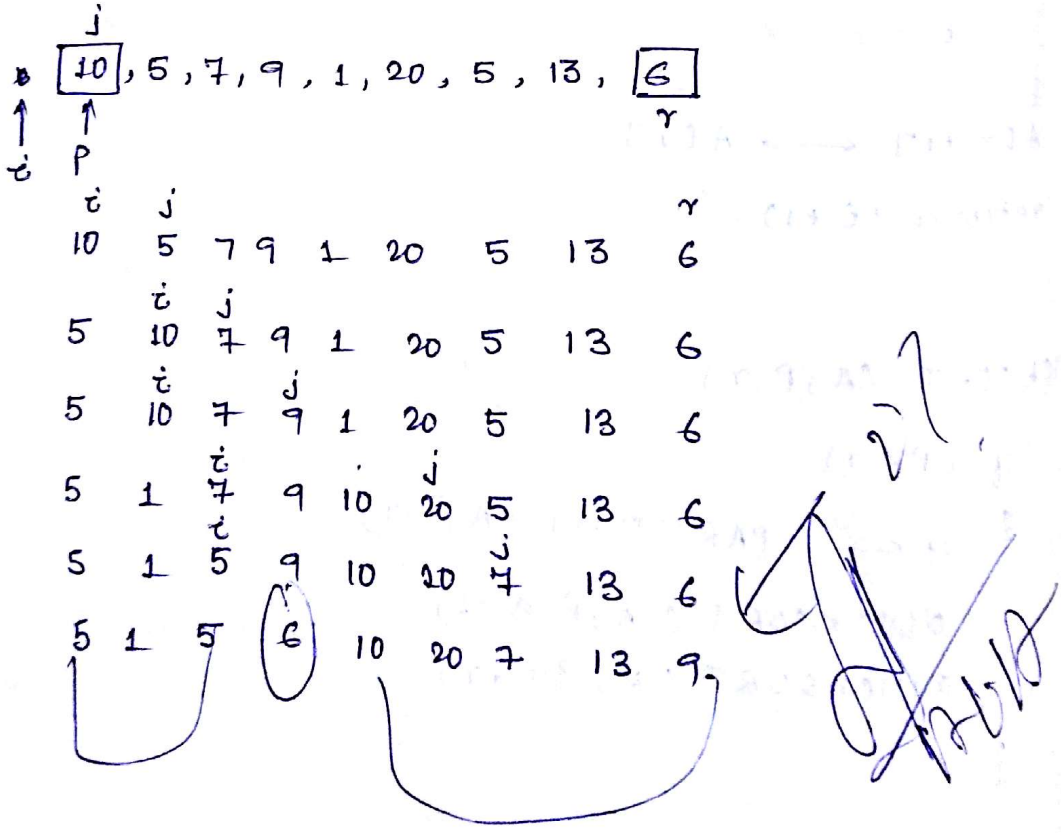
```
PARTITION (A, P, r)
{
    i ← P-1
    x ← A[r]
    for j ← P to r-1
    {
        if A[j] ≤ x
        {
            i ← i+1
            A[i] ↔ A[j]
        }
    }
    A[i+1] ↔ A[r]
    return (i+1)
}
```

```
QUICKSORT (A, P, r)
{
    if (P < r)
    {
        q ← PARTITION (A, P, r)
        QUICKSORT (A, P, q-1)
        QUICKSORT (A, q+1, r)
    }
}
```

Q.) show diff steps of partition method of pass 1 when applied to following data.



Q.) show step by step the partition method of pass 1 of quick sort with data.



Time complexity :

worst case

if data is already sorted.

1 2 3 4 5 [6] ← T(n)

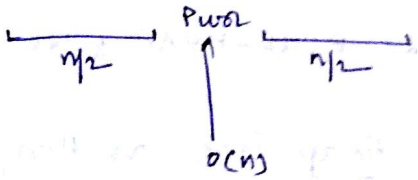
T(n-1)

$T(n) = T(n-1) + cn$.

$\boxed{T(n) = O(n^2)}$

Best case

if pivot is always at middle



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

$\boxed{T(n) = O(n \log n)}$

HA

- rewrite the partition method by taking 1st element as pivot.
- taking middle element as pivot.

18/1/17

HEAP SORT

1. uses BUILD-HEAP and HEAPIFY methods
2. BUILD-HEAP first creates a heap-tree

Heap Tree or Heap

Two properties

Property 1

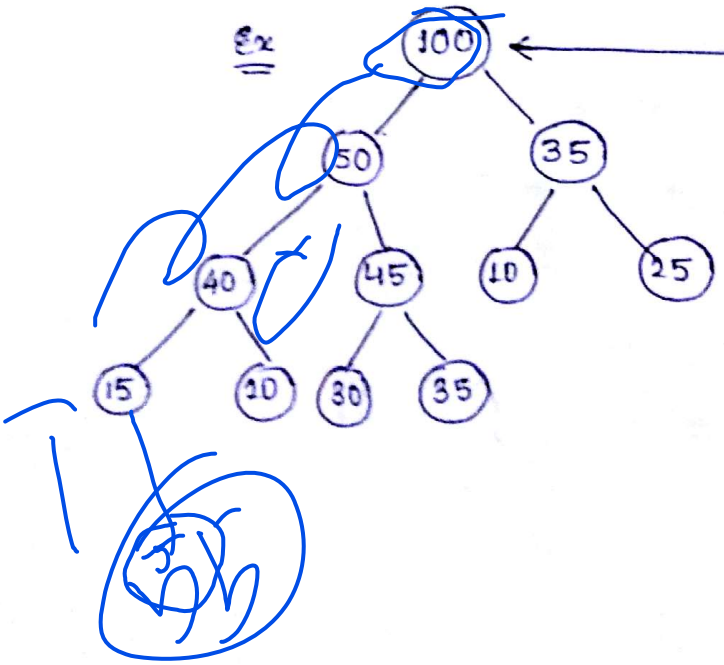
shape of heap = complete binary tree (query level is filled in LTR manner)

Property 2

For max heap : value at each node is greater than equal to its children.

For min heap : value at each node is less than or equal to its children.

Ex

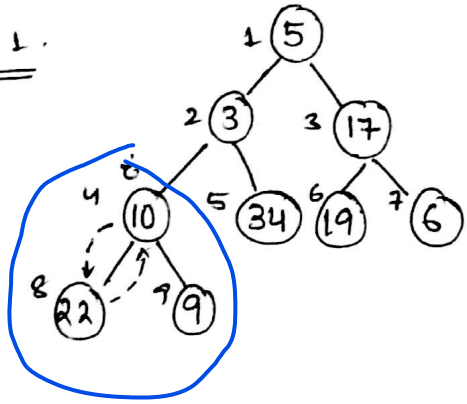


max. heap
(max value at node root)

Q) Illustrate operation of build max heap on the array

A = { 5, 3, 17, 10, 34, 19, 6, 22, 9 }

Prop 1.



if i is the index of a node then.

→ index of its left child

l = LEFT (i) = 2i

→ index of its right child

r = RIGHT (i) = 2i + 1

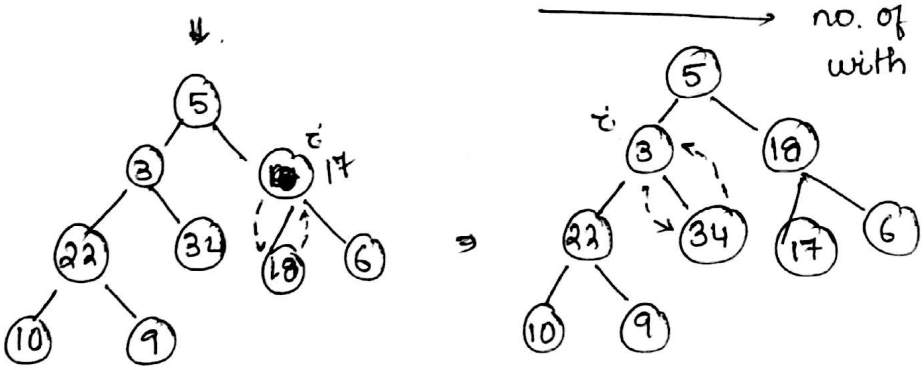
→ Index of parent is

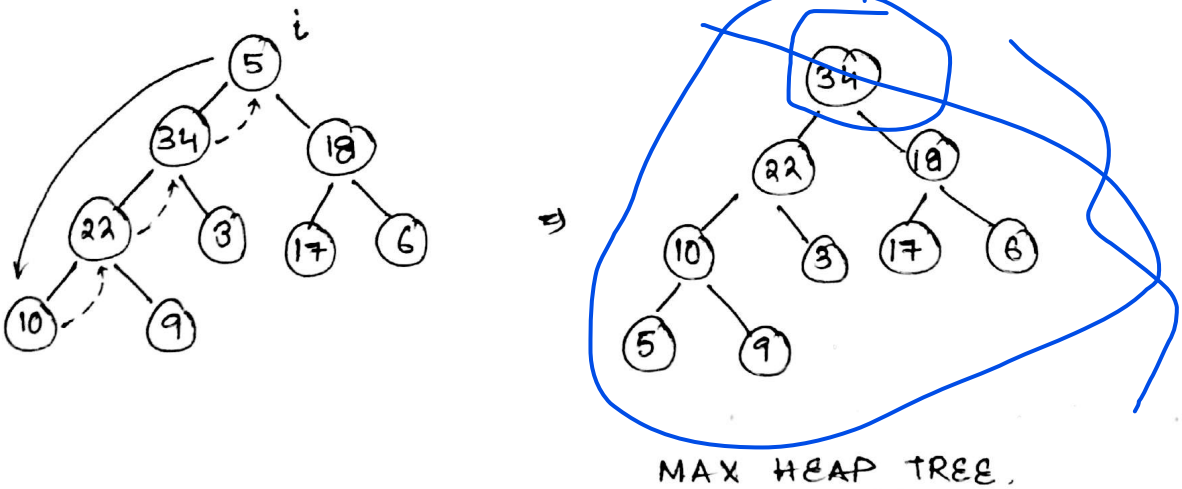
= PARENT i = [i/2].

→ no. of nodes with child = [n/2]

n = 9

i = n/2 = 9/2 = 4.





MAX HEAP TREE.

```
MAX HEAPIFY (A, i)
{
    l ⟵ LEFT(i)
    0 ⟵ RIGHT(i)
    if l ≤ heap.size[A] and A[l] > A[i]
    else largest ⟵ l
        else largest ⟵ i
    if r ≤ heapsize[n] and A[r] > A[largest]
        largest ⟵ r
    if largest ≠ i
        A[i] ⟷ A[largest]
        MAX - HEAPIFY (A, largest).
}

BUILD MAX-HEAP (A)
{ for i ⟵ [heapsize[n] /2] down to 1,
        MAX - HEAPIFY (A, i)
```

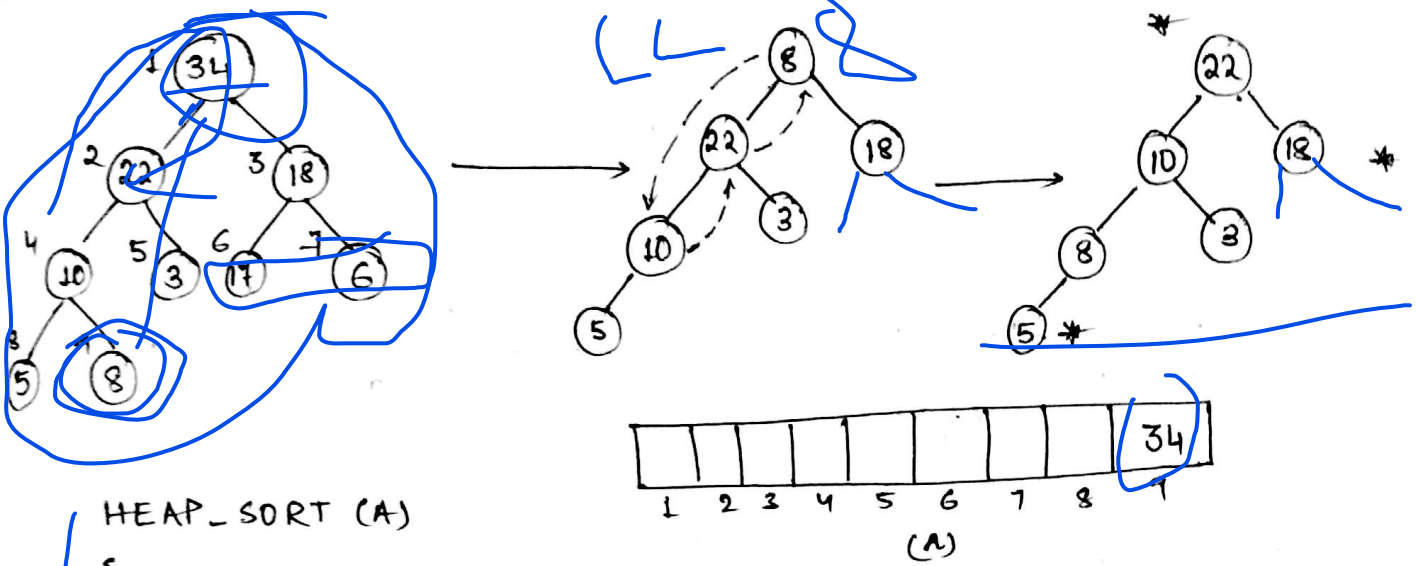
19/1/17.

Running time of:

MAX_HEAPIFY (A, i) ∋ O(logn)

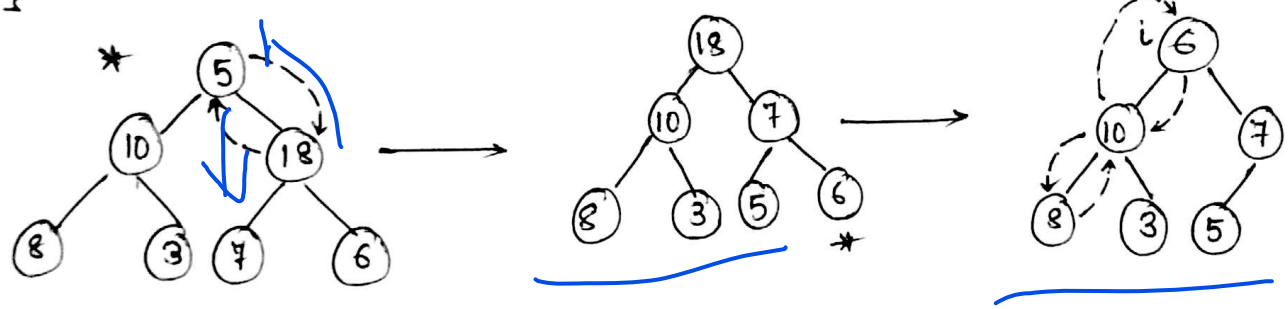
BUILD_MAX_HEAP(A) ∋ O(n)

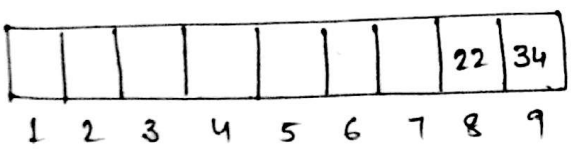
HEAP_SORT(A) ∋ O(n logn).



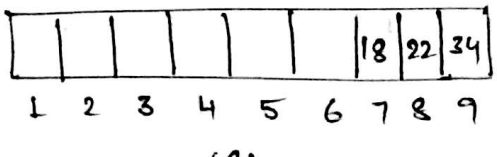
(a)

```
HEAP_SORT (A)
{
  BUILD_MAX_HEAP (A)
  for i ⟵ heapsize [A]  down to 2
  {
    A[1] ⟷ A [ heap − size(A)] // Last element is exchanged with root.
    heap_size [A] ← heap size [A]-1
    MAX_ HEAPIFY (A ,i)
  }
}
```

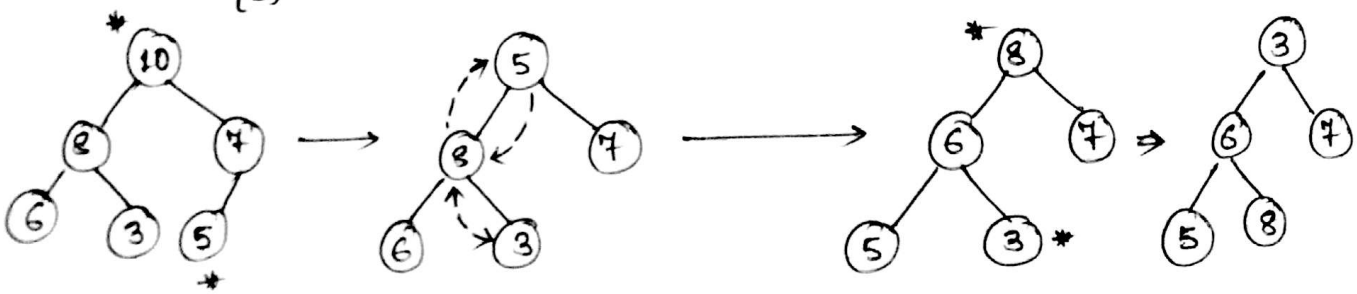




(b)

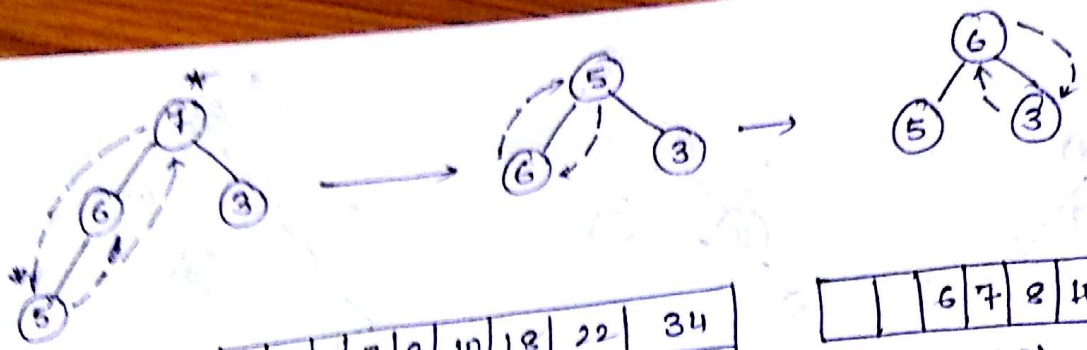


(c)



(d)

(e)

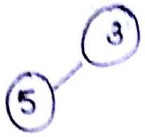


			7	8	10	18	22	34

(f)

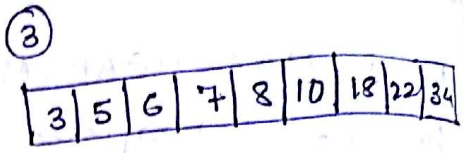
		6	7	8	10	18	22	34

(g)

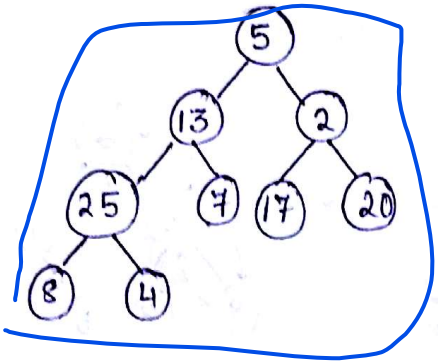


		5	6	7	8	10	18	22	34

(h)



HA1 Illustrate the operation of HEAP_SORT on the data stored in an array A where A = { 5, 13, 2, 25, 7, 17, 20, 8, 4 }



HA2 Rewrite the MIN_HEAPIFY Procedure, BUILD_MIN_HEAP and HEAP_SORT(A) for constructing w.r.t the procedure of MAX_HEAP

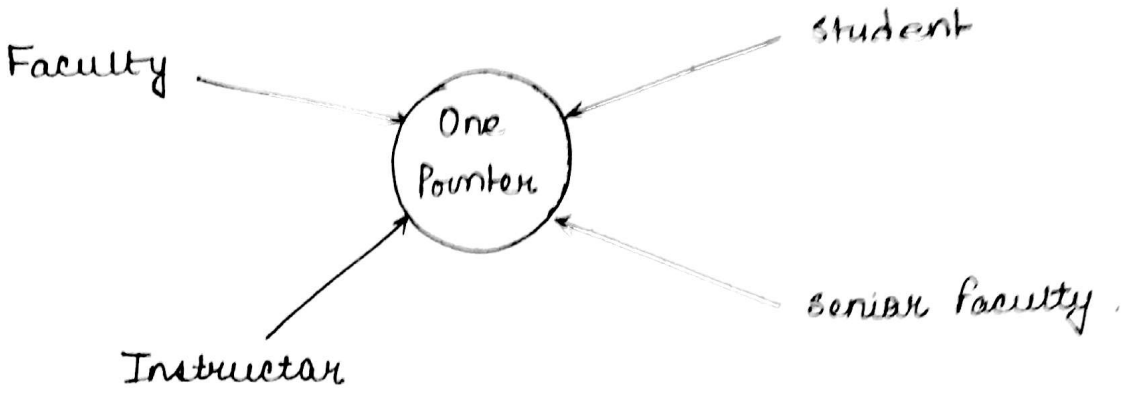
HA Solve the HA1 that will sort data in descending order.

PRIORITY QUEUES

20/1/17

- Heaps efficiently implements priority queues.
- max priority queues is implemented with max heap.
- min priority — do — min heap.

— Priority queue is a data structure that maintains a set S of elements with each associated with a ~~heap~~ key.



→ max primary queue: supports the following operations.

a) INSERT (S, x) insert a key into a set S. $S \leftarrow S \cup [x]$
(x)

b) EXTRACT MAX (S) removes and returns an element of S with largest key.

c) INCREASE KEY(A, i, Key): Here the value of S is increased to a new value key.

d) Maximum (S) returns an element of S with largest key

→ Min priority queue: supporting following operations.

a) same

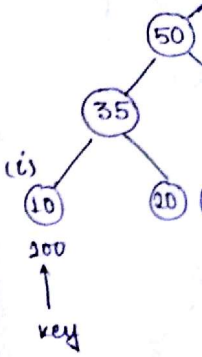
b) Extract-min (S) removes & returns an element of S with smallest key.

c) Decrease key (A, i, key) : then the ith value is decreased to a new value key.

d) Minimum (S) returns an element of S with smallest key.

```
Heap - maximum (A)
{ return A[1]; }.

Exract - Max (A)
{ if heapsize [A] < 1
  printf ("underflow of heap")
  max ← heap - maximum (A)
  A [1] ← A [heapsize [A]]
  heapsize [A] ← heapsize [A] - 1
  Max - heapify (A, 1)
  return MAX
}.
```

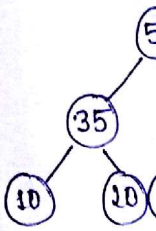


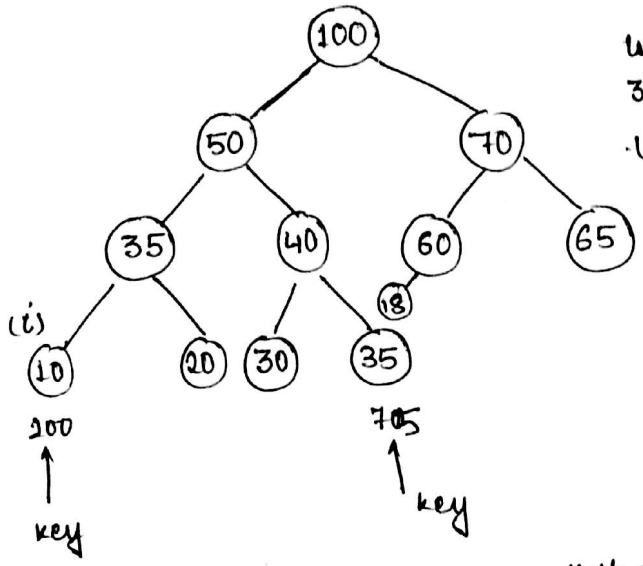
MAX_HEAP
{
heap - s
A [heap
MAX_ h
}.

24/1/17.

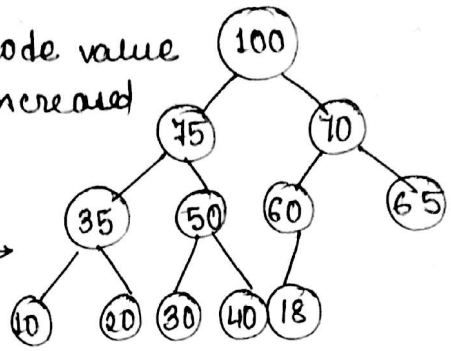
Q. Inse

```
MAX HEAP_ INCREASE KEY (A, i, key)
{   if key < A[i]
    {   print ("key < A[i], no change")
        return;
    }
    A[i] ← key
    while (i > 1 && A[i] > A [Parent [i]])
    {   A[i] ⟷ A [Parent [i]]
        i ← Parent [i]
    }
}
```

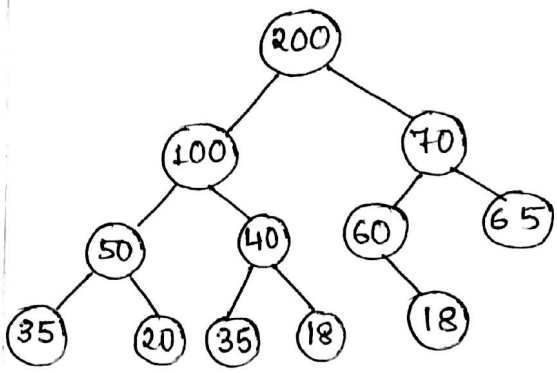




when node value 35 is increased to 75

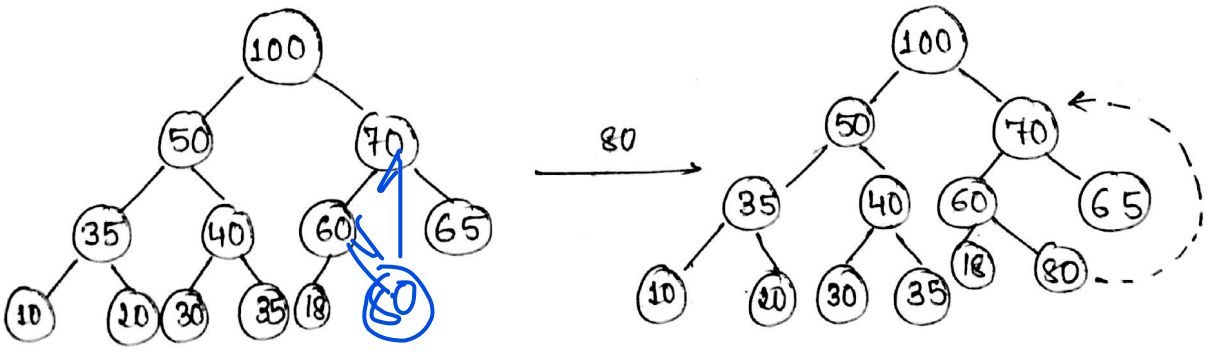


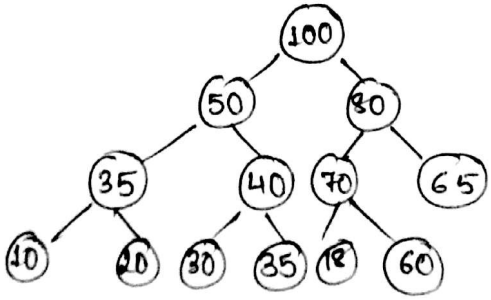
when node value 10 is increased to 200




```
MAX_HEAP_INSERT (A, X)
{
    heap_size [A] ⟵ heap_size + 1
    A [heap size [A]] = -∞
    MAX_ heap_ Increase_KEY (P, heap_size [A], X)
}.
```

Q) Insert a value 80 into the following max_heap.

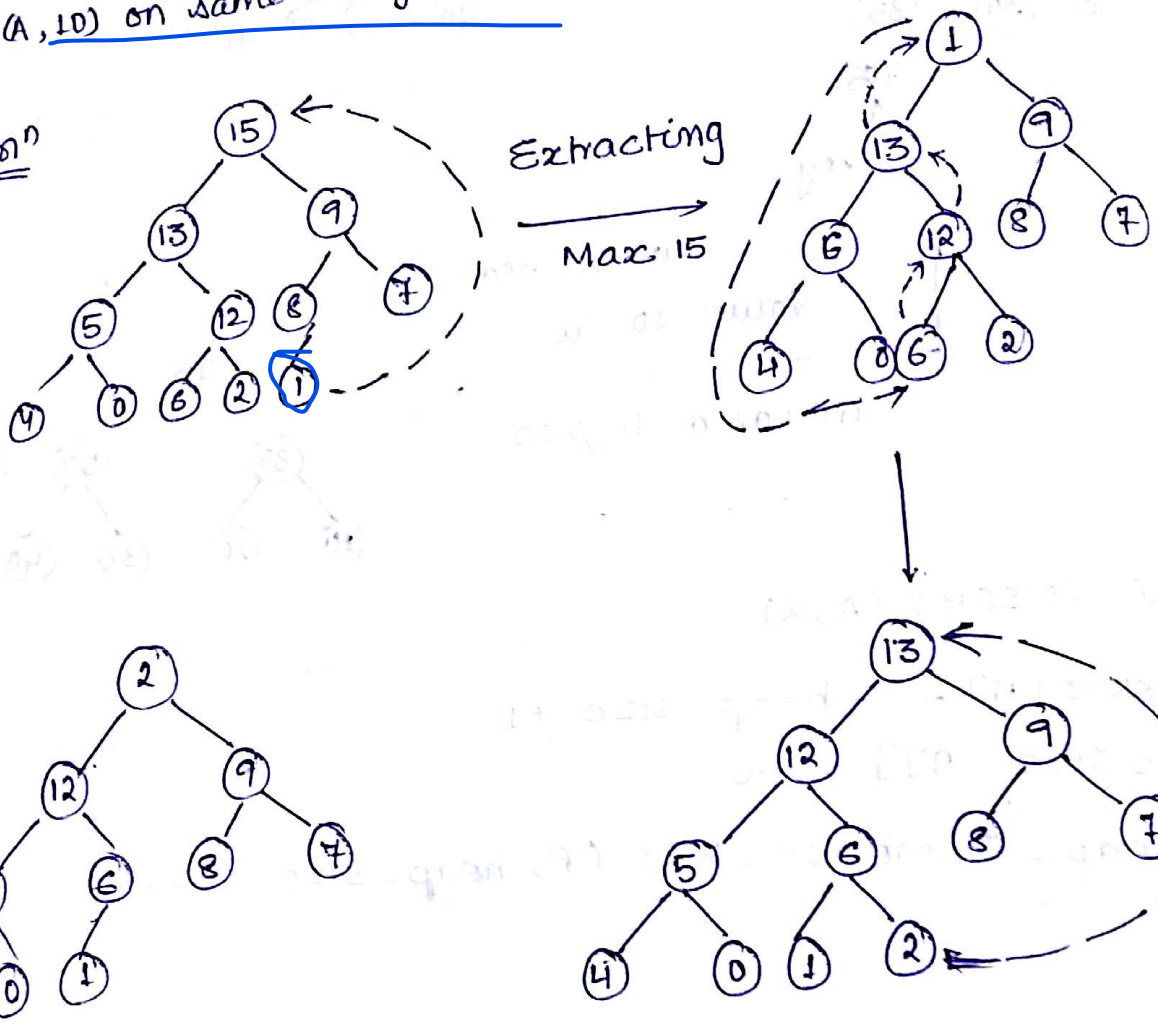




HA Illustrate the operation HEAP_EXTRACT_MAX on the heap A = {15, 13, 9, 3, 12, 8, 7, 4, 0, 6, 2, 1}

HA. Illustrate the operation of MAX_HEAP_INSERT (A, 10) on same array in Q1.

Soln



HA. write pseudocode for the procedure HEAP_MINIMUM HEAP_EXTRACT_MIN, HEAP_DECREASE_KEY and MIN_HEAP_INSERT that implements a minimum priority queue with MIN-HEAP.

25/1/17

MAX - MIN PROBLEM by D-n-C

An array A[1..n] or A[lb... ub] is having n elements. To find out max and min. value stored in an array.

Approach 1: Assuming 1st element as max & min and then compairing these values with the rest.

```
STRAIGHT_MAX_MIN (A, lb, ub, min, max)
{
    min = max = A[ub];
    for i ← ub+1 to ub
    {
        if A[i] > max.
        max ← A[i];
        if A[i] < min
        min ← A[i]
    }
}
```

$T(n) = O(n)$

Approach 2:

Dividing the arrays into two halfs. Find out max and min in both the halfs. Now finally compare two max, and to mins to get the overall max and min.

Two small cases

$n = 1$ max = min = A[lb]

$n = 2$ if A[lb] > A[lb + 1] (lb → ub)

then max = A[lb]

min = A[ub or lb + 1]

```
DNC_MAX_MIN (A, lb, ub, max, min)
{   if (lb == ub)      /* hang one element */
    {
      max = min = A[lb]
    }
    else if (lb = ub-1)  /* hang two elements */
    {
      if A[lb] > A[ub]
          max ← A[lb]

      else
          max ← A[ub]

      if  A[ub] < A[ub]
          min ← A[ub]

      else
          min ← A[ub]
    }
    else.        // Divide
    {
        mid ← (lb + ub)/2

        DNC_MAX_MIN(A, lb, mid, max, min)
        DNC_MAX_MIN(A, mid+1, ub, max1, min1)

    // compare

    if max1 > max      max = max1;
    if min1 < min      min = min1;

}}
```

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

Q.) Compare the straight max-min with DNC max-min in terms of time complexity in both cases.

⟶ when n is small, straight method.
n is large, Asymptotic method.