

## Algorithm

1) Algorithm for DAC (finding min max):-

DAC-maxmin(a, i, j)

{

    if ( $i == j$ )

        max = min =  $a[i]$ ; // only one element

    else if ( $i == j - 1$ ) // only 2 elements

    {

        if ( $a[i] < a[j]$ )

            max =  $a[j]$ ; min =  $a[i]$ ;

        else

            max =  $a[i]$ ; min =  $a[j]$ ;

    }

    small  
 $O(1)$

{

    mid =  $\lfloor (i+j)/2 \rfloor$  Divide

$- O(1)$

conquer

$T(n/2)$

~~$T(n/2)$~~

$(\max_1, \min_1) = \text{DAC-maxmin}(a, i, \text{mid});$

$(\max_2, \min_2) = \text{DAC-maxmin}(a, \text{mid}+1, j);$

$2T(n/2)$

if ( $\max_1 < \max_2$ )

    max =  $\max_2$ ;

else

    max =  $\max_1$ ;

combine

if ( $\min_1 < \min_2$ )

    min =  $\min_1$ ;

else

    min =  $\min_2$ ;

$\Rightarrow O(1)$

Big

return (max, min);

}

for find max/min

Recurrence Relation:-

1)  $T(n)$  = Time Complexity

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \text{ or } 2 \\ O(1) + 2T\left(\frac{n}{2}\right) + O(1) & \text{for } n>2 \end{cases}$$

$$= O(n)$$

2)  $T(n)$  = No. of comparisons to

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ 0 + 2T\left(\frac{n}{2}\right) + 2 & \text{if } n>2 \end{cases}$$

$$= 1.5n - 2$$
$$= O(n).$$

Space Complexity :- i/p + Extrp

$$\begin{aligned} &n \text{ Bytes} + \text{stack} \\ &\qquad\qquad\qquad\downarrow \\ &\qquad\qquad\qquad \underbrace{\log n}_{O(n)} \end{aligned}$$

NOTE :- TC for find an element neither max or min =  $O(1)$ .  
TC for find an element neither  $2^{\text{nd}}$  min or max =  $O(1)$ .

## 2.) POWER OF AN ELEMENT:-

DNC-power ( $a, n \rightarrow T(n)$ )

{

if ( $n == 1$ )

return  $a$ ;

else

{ mid =  $\lfloor n/2 \rfloor \rightarrow O(1)$  — Divide

b = DNCpower ( $a, \text{mid} \rightarrow T(n/2)$ ) — Conquer

c =  $b \times b \rightarrow O(1)$  — Combine

return ( $c$ )

}

Recurrence Relation :-  $T(n) = TC$  to find  $a^n$ .

$$T(n) = \begin{cases} O(1) & \text{for } n=1 \\ O(1) + T(n/2) & \text{for } n>1 \\ + O(1) \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right) + C$$

$$= O(\log n)$$

(2)  $T(n) = \text{No. of Multiplications}$  - for above Program.

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ O(1) + T\left(\frac{n}{2}\right) + 1 & \text{if } n>1 \end{cases} = O(\log n).$$

NOTE :-  $\left[ \text{for odd no. : } c = b \times b ; \quad T(n) = \text{No. of Multiplications} \right]$

$$c = c \times a^n \quad \left[ T(n) = \begin{cases} O(1) & \text{if } n=1 \\ O(1) + T\left(\frac{n}{2}\right) + 2 & \text{if } n>1 \end{cases} \right]$$

$$T(n) = TC$$

$$T(n) = \begin{cases} O(1) & \text{for } n=1 \\ O(1) + T\left(\frac{n}{2}\right) + O(1) & \text{for } n>1 \end{cases}$$

$$= O(\log n)$$

If same logics are used in Algo. Recursive & Non-Recursive Program will take same TC. Binary Search is partial DAC application because No combined is done in B.S.

iii.) Linear Search:- i/p - An array of n elements + element  $[n]$   
o/p - find position of x.

$$BC : O(1)$$

$$WC : O(n)$$

$$AC : \frac{1+n}{2} = O(n).$$

if found  
else  
return (-1).

and

Binary Search:- i/p - A sorted array of n elements, ele - x  
o/p - find position of x if found else -1.

Algorithm:-

$$BS(a, p, q, x) \longrightarrow T(n)$$

{

if ( $p == q$ )  
if ( $a[p] == x$ )  
return  $x$   
else  
return -1 ]  $O(1)$

else

{ mid =  $(p+q)/2$   
if ( $a[mid] == x$ )  
return (mid)  
else  
if ( $x < a[mid]$ ) ] c

return BS(a, p, mid-1, x) —  $T(n/2)$   
return BS(a, mid+1, q, x) —  $T(n/2)$

else return }

Recurrence Relation:-

$$T(n) = T_C$$

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ T(n/2) + c & \text{if } n>1 \end{cases}$$

$$AC = \underline{O(\log n)}$$

SC =  $n + \log n$   
=  $n$ .

BC =  $O(1)$ .

tail programs takes more space, non tail taking less space. 5

### Merge Sort Algorithm:-

MergeSort ( $a, p, q$ )

```
{  
    if ( $p == q$ )  
        return ( $a[p]$ )  
    else  
        {  
            mid =  $\lfloor (p+q)/2 \rfloor$  — O(1)  
            MergeSort ( $a, p, mid$ ); — T( $n/2$ )  
            MergeSort ( $a, mid+1, q$ ); — T( $n/2$ )  
            Merge ( $a, p, mid, mid+1, q$ ); — O( $n$ ) for outplace  
            return ( $a$ );  
        }  
}
```

### Merge Algorithm:-

Merge ( $a, p, mid, mid+1, q$ )

```
{  
    i = 1;  
    k = mid + 1;  
  
    while ( $p \leq mid \text{ & } k \leq q$ )  
    {  
        if ( $a[p] < a[k]$ )  
        {  
            b[i] = a[p]  
            p++, i++  
        }  
        else  
        {  
            b[i] = a[k]  
            k++, i++  
        }  
    }  
    move remaining element from A to B.  
}
```

O( $n$ ) — Outplace  
O( $n^2$ ) — Inplace

NOTE:- TC of merge Algo is :  $\frac{O(m+n)}{O(n)}$  for outplace  
 $O(m \cdot n) \text{ or } O(\frac{n \cdot n}{2})$  —  $\underline{O(n^2)}$  for inplace.

Recurrence Relation :-  $T(n) = T.C.$

(i.) for outplace

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ O(1) + 2T\left(\frac{n}{2}\right) + n & \text{if } n>1 \end{cases}$$

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

(ii.) for Inplace

$$T(n) = \begin{cases} O(1) & n=1 \\ O(1) + 2T\left(\frac{n}{2}\right) + \frac{n^2}{4} & n>1 \end{cases}$$

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

(1) Space Complexity :- i/p + extra

$$\begin{matrix} \Downarrow \\ n \text{ Byte} \end{matrix} \quad \begin{matrix} \nearrow \\ \text{stack Merge} \\ \log n \quad n \text{ Byte} \\ (\log n + n) \end{matrix}$$

$$= O(n).$$

- NOTE :-
- 1.) MergeSort is outplace, because in MergeSort algo we are using  $O(n)$  place extra.
  - 2.) MergeSort is advisable for larger size arrays. for smaller size, Insertion sort is preferable.

(i) worst case outplace (output store in another array)

10	20	30	40	11	21	31	41
----	----	----	----	----	----	----	----

$\leftarrow m \longrightarrow \leftarrow n \longrightarrow$  for sorted unequal elements

$\frac{n}{2} \quad \frac{n}{2} \longrightarrow$  for sorted equal elements

Comparisons

$$4, 4 = \underline{4+4-1}$$

Moves

$$4, 4 \rightarrow \underline{4+4}$$

$$\frac{n}{2}, \frac{n}{2} = \underline{n-1}$$

$$\frac{n}{2}, \frac{n}{2} \rightarrow \underline{n}$$

$$m, n = \underline{m+n-1}$$

$$m, n \rightarrow \underline{m+n}$$

TC :- [9n comparisons & Moves, whoever is greater is TC]

$\hookrightarrow$  moves  $\Rightarrow 4, 4 \Rightarrow 4+4$

$$\frac{n}{2}, \frac{n}{2} \Rightarrow n$$

$$m, n \Rightarrow m+n$$

(ii) Butcase outplace

Comparison

$$4, 4 = 4$$

$$\frac{n}{2}, \frac{n}{2} = \frac{n}{2}$$

$$m, n \Rightarrow \underline{\min(m, n)}$$

Moves

$$4, 4 \Rightarrow 8$$

$$\frac{n}{2}, \frac{n}{2} \Rightarrow n$$

$$m, n \Rightarrow \underline{m+n}$$

40	50	60	70	80	10	20	30
----	----	----	----	----	----	----	----

(iii) worst case (Inplace)

$$M \left\{ \begin{array}{l} 50, 10 \Rightarrow 10 \Rightarrow n \\ 60, 20 \Rightarrow 20 \Rightarrow n \end{array} \right.$$

$$\frac{n}{m}$$

$$\frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4} \quad \} \text{ moves}$$

20	50	30	40				
15	60	20	30	40			
m		n					

NOTE:- Merging two sorted subarrays each of size  $m$  &  $n$  will take  $\underline{O(m+n)}$  time. (outplace).

AMB will contain all elements with repetition  $\{1, 1, 2, 2, 3, 3, 4, 5\}$ .

Quicksort ( $a, p, q$ )  
{     if ( $p == q$ )  
        return ( $a[p]$ )

else

{

$O(n) \longrightarrow K = \text{partition}(a, p, q)$  — Pivot element  
 $T(K-p) \longrightarrow \text{Quicksort}(a, p, k-1);$  — Left sort  
 $+ (q - k) \longrightarrow \text{Quicksort}(a, k+1, q);$  — Right sort  
}

}

Partition Algorithm:-

Partition ( $a, P, q$ )

{

$x = a[p]$

$i = p;$

for ( $j = p+1; j \leq q; j++$ )

{     if ( $a[j] \leq x$ )

{          $i = i + 1;$

           swap ( $a[i], a[j]$ );

}

}

swap ( $a[i], a[p]$ )

return ( $i$ );

}

## SELECTION PROCEDURE :-

i/p - An Array of  $n$ -integers and integer  $K$ .

o/p - find  $K^{th}$  smallest element.

SP( $a, p, q, K$ )

{  
if ( $p == q$ )  
return ( $a[p]$ );  $\Rightarrow O(1)$

else

{  
 $m = \text{Partition}(a, p, q);$   $\rightarrow O(n)$

if ( $m == K$ )

return ( $a[K]$ );

else

if ( $K < m$ )

return (SP( $a, p, m-1, K$ ))  $\rightarrow T(m-p)$

else

return (SP( $a, m+1, q, K$ ))  $\rightarrow T(q-m)$

} }

$$T(n) = TC,$$

Recurrence Relation:-

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ n + T(m-p) & \text{or} \\ & \text{if } n \geq 1 \\ T(q-m) & \end{cases}$$

Best case

$$T(n) = n + T\left(\frac{n}{2}\right)$$

$\downarrow \log n$

$$= n + \frac{n}{2} + \frac{n}{2^2} + \dots + \frac{n}{2^{\log n}}$$

$$= O(n)$$

worst case

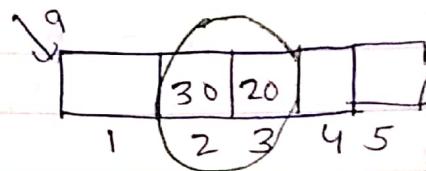
$$T(n) = n + T(n-1)$$

$$= 1 + 2 + \dots + n$$

$$= O(n^2)$$

### Counting Number of Inversion :-

i/p - An array of  $n$  elements  
 o/p - count inversions



$2 < 3$   
 but  $a[2] > a[3]$

while ( $p \leq mid \text{ } \& \& k \leq q$ )

{  
 if ( $a[p] > a[k]$ )

{  
 $b[i] = a[k];$

$i++; \quad k++;$

$$NI = NI + mid - p + 1$$

}

else

{  
 $b[i] = a[p];$

$i++; \quad p++;$

}

} copy remaining element to  $B$ .

$$\begin{aligned} T(n) = TC &= O(1) + T(n/2) + T(n/2) + n \\ &\quad 2T(n/2) + n \\ &\underline{O(n \log n)}. \end{aligned}$$

### Strassen's Matrix Multiplication :-

without DAC

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2$$

↓

$$\underline{O(n^3)} = \underline{O(n^{\log_2 8})}$$

with, DAC

$$\begin{aligned} T(n) &= 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \\ &= 7T\left(\frac{n}{2}\right) + 4.5n^2 \\ &\underline{O(n^{2.81})} \end{aligned}$$

# Recurrence Relations & Time Complexity

1.) finding max min	$T(n) = \{ 2T(n/2) + c \}$	$TC = O(n)$																					
2.) Linear search																							
3.) Power of an element	$T(n) = \{ T(n/2) + c \}$	$TC = O(\log n)$																					
4.) Binary search																							
5.) Merge Sort <small>Counting No. of inversion</small>	$T(n) = \{ 2T(n/2) + n \}$	$TC = O(n \log n)$																					
6.) Selection Procedure																							
7.) Quick Sort <small>(Randomized Quick sort)</small>	<p><u>B.C :-</u></p> $\begin{aligned} T(n) &= n^2 + T(m-p) + T(q-m) \\ &= n + T(n/2) + T(n/2) \\ &= 2T(n/2) + n \\ &= \underline{\underline{O(n \log n)}} \end{aligned}$	<p><u>W.C</u></p> $\begin{aligned} T(n) &= n + T(n-1) \\ &= \underline{\underline{O(n^2)}}. \end{aligned}$																					
8.) Selection Procedure	$T(n) = \{ n + T(m-p) \}$ $T(n) = n + T\left(\frac{n}{2}\right)$ $= n + \frac{n}{2} + \frac{n}{2^2} + \dots + \frac{n}{2^{\log n}}$ $= \underline{\underline{O(n)}}$	$T(n) = n + T(n-1)$ $\downarrow n$ $1 + 2 + \dots + n$ $= \underline{\underline{O(n^2)}}$																					
9.) Strassen's Multiplication	<p>without strassen</p> $T(n) = 8T\left(\frac{n}{2}\right) + n^2$ $= \boxed{\underline{\underline{O(n^3)}}}$	<p>with strassen</p> $T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$ $= \underline{\underline{O(n^{2.8})}}$																					
with comparison	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">BC (forget)</th> <th style="text-align: center;">WC</th> <th style="text-align: center;">AC</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;"><math>n^2</math></td> <td style="text-align: center;"><math>n^2</math></td> <td style="text-align: center;"><math>n^2</math></td> </tr> <tr> <td style="text-align: center;"><math>n^2</math></td> <td style="text-align: center;"><math>n^2</math></td> <td style="text-align: center;"><math>n^2</math></td> </tr> <tr> <td style="text-align: center;"><math>n</math></td> <td style="text-align: center;"><math>n^2</math></td> <td style="text-align: center;"><math>n^2</math></td> </tr> <tr> <td style="text-align: center;"><math>n \log n</math></td> <td style="text-align: center;"><math>n \log n</math></td> <td style="text-align: center;"><math>n \log n</math></td> </tr> <tr> <td style="text-align: center;"><math>n \log n</math></td> <td style="text-align: center;"><math>n^2</math></td> <td style="text-align: center;"><math>n \log n</math></td> </tr> <tr> <td style="text-align: center;"><math>n \log n</math></td> <td style="text-align: center;"><math>n \log n</math></td> <td style="text-align: center;"><math>n \log n</math></td> </tr> </tbody> </table>	BC (forget)	WC	AC	$n^2$	$n^2$	$n^2$	$n^2$	$n^2$	$n^2$	$n$	$n^2$	$n^2$	$n \log n$	$n \log n$	$n \log n$	$n \log n$	$n^2$	$n \log n$	$n \log n$	$n \log n$	$n \log n$	without comparison
BC (forget)	WC	AC																					
$n^2$	$n^2$	$n^2$																					
$n^2$	$n^2$	$n^2$																					
$n$	$n^2$	$n^2$																					
$n \log n$	$n \log n$	$n \log n$																					
$n \log n$	$n^2$	$n \log n$																					
$n \log n$	$n \log n$	$n \log n$																					
1. Bubble Sort	$n^2$	$n^2$	1. counting sort																				
2. Selection Sort	$n^2$	$n^2$	2. Radix sort																				
3. Insertion Sort	$n$	$n^2$	3. Bucket sort																				
4. Merge Sort	$n \log n$	$n \log n$																					
5. Quick Sort	$n \log n$	$n^2$																					
6. Heap Sort	$n \log n$	$n \log n$																					
inplace →			$\downarrow$																				
NOTE :- If range given - then go for counting sort.			$O(n)$ in every case																				
No. of digits of larger no in Array — Radix sort.																							

$TC = \text{swap} + \text{comparison}$ , whichever greater

will come.

Algorithm	Comparison	Swaps/ Moves	Dynamic Programming	
i.) Merge Algorithm			i.) Fibonacci Series without DP	with DP
ii.) BC outplace	$n/2$	$n$	$TC: O(2^n)$	$O(n)$
$TC = n$	$\min(m,n)$	$m+n$	$SC: O(n)$	$O(n)$
ii.) WC outplace	$n-1$	$n$	$DFC:$	$n+1$
$TC = n$	$m+n-1$	$m+n$		
iii.) WC inplace	$n-1$	$m \cdot n$	2. L.C.S:- without DP	with DP
$TC = n^2$	$m+n-1$	$\frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$	$TC_{WC}: O(2^m 2^n)$	$O(mn)$
			$SC: O(m+n)$	$O(m+n)$
2. Build Heap:-	$= 2 \times \text{swap}$	$\frac{n}{2}$	$DFC:$	$(m+1)(n+1)$
$TC = n$				
3. Maxheap Inversion		$TC = O(1ogn)$	3. Knapsack & Sum of subset Problem	
i.) Linear Search	$1ogn$	$1ogn$	without DP	with DP
ii.) Binary Search	$1ogn$	$1ogn$	$TC = O(2^n)$	$O(mn)$
			$SC = O(n)$	$O(mn)$
			$DFC =$	$mn$
4.) Bubble Sort If swap=0, then stop algo, because array is sorted, $BC = O(n)$ . $AC, WC = O(n^2)$	$n^2$	zero (min) $\frac{n(n-1)}{2}$ (max)	$(2^n \approx mn)$	(NP complete)
5.) Selection Sort	$\frac{n(n-1)}{2}$	$n-1$	4. Matrix chain multiplication	
	$O(n^2)$		without DP	with DP
6.) Insertion Sort			$TC: O(n^n)$	$O(n^3)$
(i) BC $O(n)$	$n-1$	$0$	$SC: O(n)$	$O(n^2)$
(ii) WC & AC $O(n^2)$	$n^2$	$n^2$	$DFC: -$	$n^2$
			5. Floyd - warshall algorithm	
				$O(n^3)$ .
			6.) Topological sorting : $O(n+m)$	

### Heap Sort:-

1. If CBT contains  $K$ -levels, Total nodes =  $2^K - 1$
2. Total leaf nodes in CBT (ACBT also) =  $\lceil n/2 \rceil$  ( $n$  = total nodes)  
Internal nodes =  $\lfloor n/2 \rfloor$
3. If CBT contains  $n$  - nodes,  $K$ -levels

$$\boxed{\text{Height} = \text{Level} - 1}$$

$$\boxed{H = \log_2(n+1) - 1}$$

$$n = 2^K - 1$$

$$2^K = n + 1$$

$$\log_2 2^K = \log_2(n+1)$$

$$K = \log_2(n+1)$$

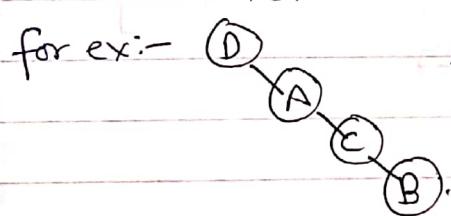
Representation of Binary Tree:- 1. Array 2. Linked List.

NOTE:- If node is stored in 1<sup>st</sup> place of the array then.

- i.) Parent of  $i = \lceil i/2 \rceil$
  - ii.) Left child ( $i$ ) =  $2 \times i$
  - iii.) Right child ( $i$ ) =  $(2 \times i) + 1$
- } Array starting from 1.  
} different when index changes

NOTE:- Array is better for complete & almost complete Binary Tree.

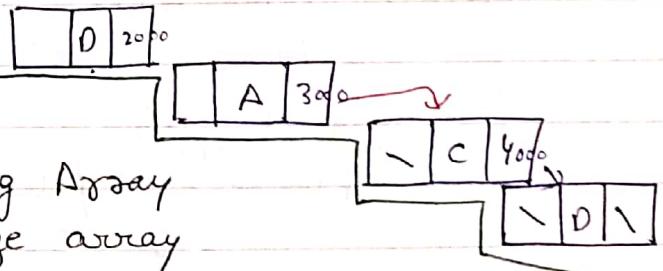
Linked List is better idea for any gap in Nodes.



Using Array:-

D	A				C							B
1	2	3	4	5	6	7	8	9	10	11	12	13

Using LL:-



Q.) BT -  $n$  nodes represented using Array

Min<sup>m</sup> size array

$n$

∴ In case of ACBT

Max<sup>m</sup> size array

$$\underline{2^n - 1}$$

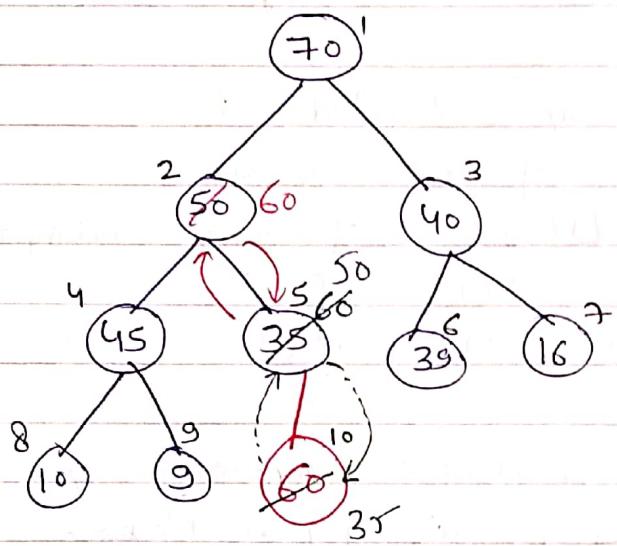
In case of BT (non ACBT).

Maxheap :- root is maximum or equal to children at every level.

for  $i^{th}$  max, Total comparison =  $\frac{i(i-1)}{2} = O(i^2)$ .

NOTE :- In Bubble sort, every pass is costly i.e  $O(n)$  but in heap sort, every <sup>comparison</sup> cost is  $O(1)$ .

Insertion in Maxheap :- (Bottom to Top Approach)



A	70	50	40	45	60	35	16	10	9	66
	1	2	3	4	5	6	7	8	9	10

$$\text{Part of } 10 = \left\lfloor \frac{10}{2} \right\rfloor = 5$$

	60	50	40	45	66	35	16	10	9	35
	1	2	3	4	5	6	7	8	9	10

parent of 5 =  $\left\lfloor \frac{5}{2} \right\rfloor = 2$

	70	60	40	45	50	35	16	10	9	35
	1	2	3	4	5	6	7	8	9	10

$$\text{parent of 2 is } \left\lfloor \frac{2}{2} \right\rfloor = 1.$$

No swapping.

insertHeap(A, n, value)

```
{
    n = n+1;
    A[n] = value;
```

```
while (i > 1) {
    Parent =  $\left\lfloor \frac{i}{2} \right\rfloor$ 
```

```
if (A[Parent] < A[i])
```

```
{
    swap (A[Parent], A[i]);
    i = Parent;
```

```
else
```

```
{
    return;
```

## & Deleting

NOTE:- Inserting a element into minheap or maxheap, which already contain  $n$  elements will take order of  $\log n$ .

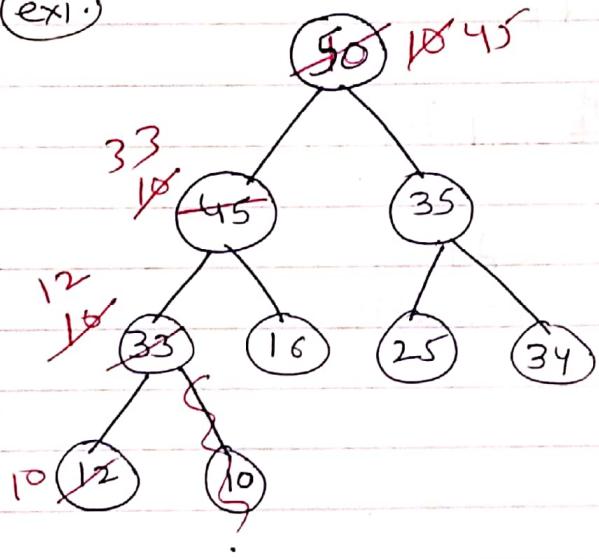
for  $n$  elements  $\rightarrow$  Best case -  $O(1)$ , WC & AC -  $O(\log n)$ .

for  $\log n$  elements -  $\log \log n$ .

Q.) Max heap of  $n^{2^n}$  element then insert 1 element }  
 $\log(n^{2^n}) = \log n + \log 2^n = \log n + n = O(n)$ .

## DELETION IN MAX HEAP :- (Top to Bottom Approach)

(ex1.)



10									empty
50	45	35	33	16	25	34	12	10	

child of 1 <sup>(1)</sup> = 45									
45	10	35	33	16	25	34	12	-	

child of 10 (2) = 33									
45	33	35	10	16	25	34	12		

child of 10 (4) = 12									
45	33	35	12	16	25	34	10	50	

NOTE:- • To delete an element, we can simply put the deleted element at last place, & reduce the loop to one place (from 1 - 8).

- As soon as we deleted the element, we will get the elements in Ascending order (because we are placing Max element at last place).

10	12	16	25	33	34	35	45	50	
1	2	3	4	5	6	7	8	9	

- By default Heap Tree  $\rightarrow$  Max heap Tree.

Deleteheap (A, n, value)

{

value = a[i]

a[~~i~~] = a[n]

n = n - 1

left child =  $2 \times i$  ;

right child =  $(2 \times i) + 1$  ;

if ( $A[i] < A[\text{left child}]$ )

{

swap ( $A[i], A[\text{left child}]$ );

}

else if ( $A[i] < A[\text{right child}]$ )

{

swap ( $A[i], A[\text{Right child}]$ );

}

else

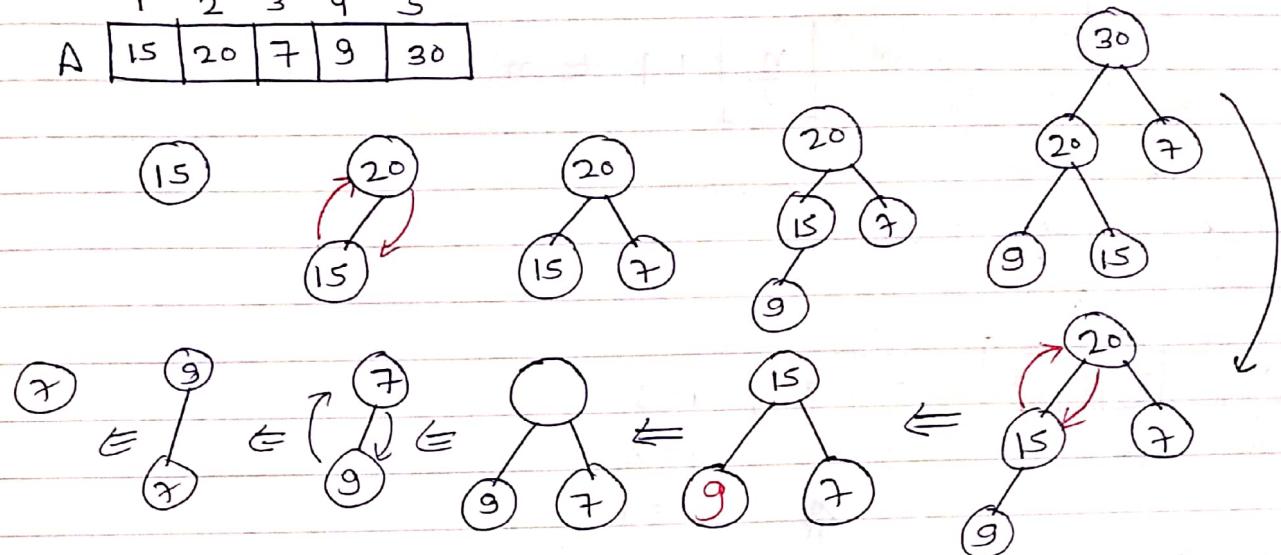
{ return;

}

}

### Heap Sort :-

1	2	3	4	5	
A	15	20	7	9	30



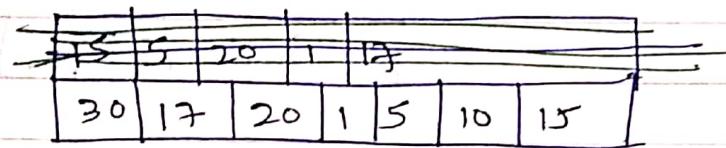
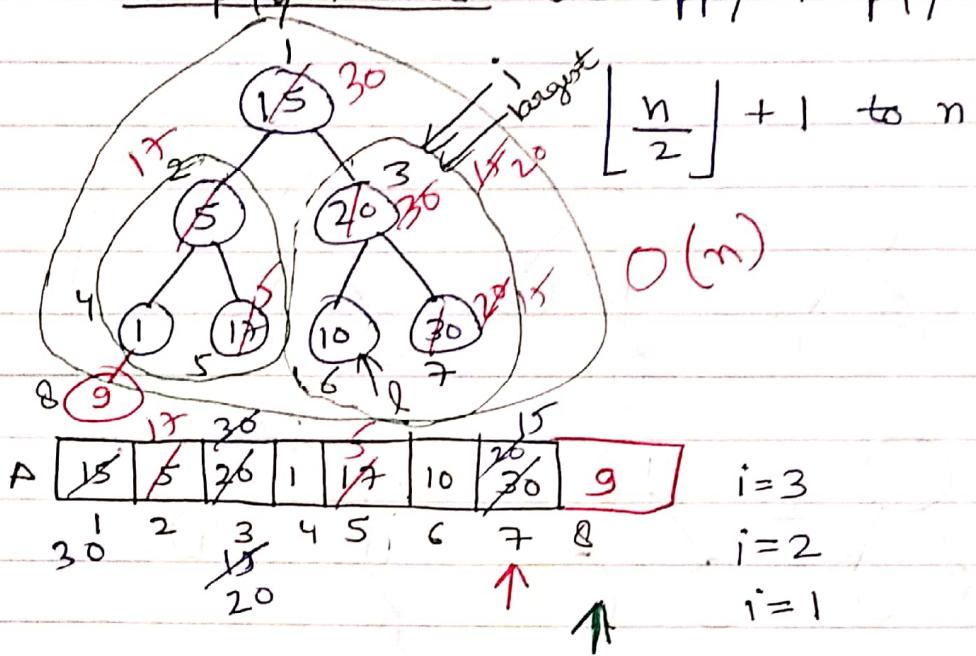
A	7	9	15	20	30
	1	2	3	4	5

- for create a max heap or min heap,  $O(1 \log n)$  times require for an single element.
  - for  $n$  elements  $= O(n \log n)$
  - for delete a max heap or min heap,  $O(1 \log n)$  times require for an single element.
  - for delete  $n$  elements  $= O(n \log n)$
  - for Heap Sort, creation & deletion requires for  $n$  elements :  $O(n \log n)$
- $O(n \log n)$   
 $O(2n \log n)$
- $\equiv O(n \log n)$

NOTE:- This method takes  $O(n \log n)$ , but there is another method which is heapify method & it takes  $O(n)$ .

from right to left

Heapify Method: - we apply Heapify method for non-leaf nodes.



Max-Heapify (A, n, i)

{ int largest = i ;

int l = (2\*i) ;

int r = (2\*i)+1 ;

while (l <= n && A[l] > A[largest])

{ largest = l ;

}

while (r <= n && A[r] > A[largest])

{ largest = r ; }

if ( largest != i )

{ swap (A[largest], A[i]);

heapify (A, n, largest);

}

}

heapsort ( $A, n$ )

{

Build

Max  
heap

{ for ( $i = n/2 ; i \geq 1 ; i--$ )  
{ Maxheapify ( $A, n, i$ );  
}

15	15	20	10	17	10	30
1	2	3	4	5	6	7

Delete

{ for ( $i = n ; i \geq 1 ; i--$ )  
{ swap ( $A[1], A[i]$ );  
Maxheapify ( $A, i-1$ );  
}

1	5	10	15	17	20	1	30
1	2	3	4	5	6	7	

we are always deleting data from  
root node that's why always 1.

## BUBBLE SORT :-

A	15	16	6	8	5	n=5
	0	1	2	3	4	

Pass 1

15	16	6	8	5
15	6	16	8	5
15	6	8	16	5
15	6	8	5	16

Pass 3

6	8	5	15	16
6	8	5	15	16
6	5	8	15	16
6	5	8	15	16
6	5	8	15	16

Pass 2

15	6	8	5	16
6	15	8	5	16
6	8	15	5	16
6	8	5	15	16
6	8	5	15	16

Pass 4

6	5	8	15	16
5	6	8	15	16
5	6	8	15	16
5	6	8	15	16
5	6	8	15	16

- No. of Required Passes =  $(n-1)$  { n elements }.
- Total Comparisons =  $(n-1) + (n-2) + \dots + 1 = \frac{(n-1)n}{2}$  =  $O(n^2)$  { EC }
- Total swaps =  $O(\min)$ ,  $\lceil \frac{(n-1)n}{2} \rceil = O(n^2)$  (max)
- TC = C + S  
 $= n^2 + \dots$   
 $= O(n^2)$ .
- 10 20 30 40 50
- $S=0$  (stop because sorted)
- Before every pass  $S=0$ , after any case, if  $S=0$ , stop.
- With this modification, B.S TC BC -  $O(n)$ , WC & AC -  $O(n^2)$ .

## OPTIMIZED BUBBLE SORT.

```

for (i=0; i < n-1; i++)
{
    flag = 0;
    for (j=0; j < n-1-i; j++)
    {
        if (A[j] > A[j+1])
        {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
            flag = 1;
        }
    }
    if (flag == 0)
        break;
}

```

$O(n)$  {Bc}.

## INSERTION SORT

0	1	2	3	4	5
5	4	10	1	6	2

$n=6$

```

for (i=1; i<n; i++) {
    temp = a[i]
    j = i-1;
    while (i>=0 && a[j]>temp) {
        a[j+1] = a[j];
        j--;
    }
    a[j+1] = temp;
}

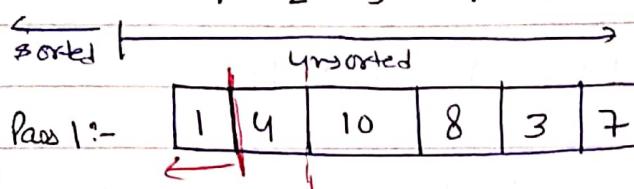
```

- Insertion Sort is useful in BC only. TC -  $O(n)$  BC.
- Comparisons -  $(n-1)$  in BC.
- If Array is Almost sorted, I.S. will take  $O(n)$ .
- No. of swap operation = No. of inversions.
- Worst case of Insertion sort (Array is in Descending order). It will take  $O(n^2)$ . Comparison =  $1+2+\dots+n-1 = \underline{\underline{n^2}}$ .
- Average case (Half Asc + Half Desc) =  $\frac{n}{2}$  BC +  $\frac{n}{2}$  WC =  $\underline{O(n^2)}$ .
- If Array size is very small, use Insertion sort.

## SELECTION SORT

7	4	10	8	3	1
0	1	2	3	4	5

$n=6$

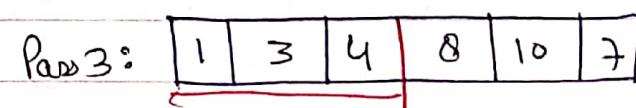
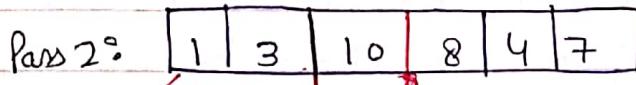


```
for (i = 0; i < n-1; i++)
```

```
{
```

```
int min = i;
```

```
for (j = i+1; j < n; j++)
```

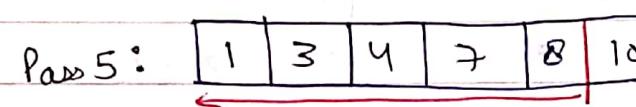
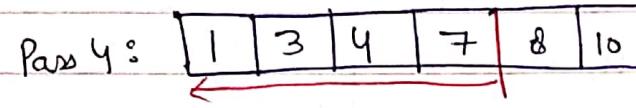


```
if (a[j] < a[min])
```

```
{
```

```
min = j;
```

```
}
```



```
if (min != i)
```

```
{
```

```
swap(a[i], a[min]);
```

```
}
```

```
}
```

- Selection Sort requires  $N-1$  passes.

- Total comparisons =  $\frac{(n-1)n}{2} = \underline{\underline{O(n^2)}}$  [ $\in C$ ], total swaps =  $\underline{(n-1)}$ .

- $TC = O(n^2)$   $\{ \in C \}$   $\{ \text{because comparisons are always } n^2 \}$ .

NOTE :- Selection Sort特别性 is no. of swaps are  $(n-1)$  in every pass  $O(n-1)$ .

## SORTING TECHNIQUES WITHOUT COMPARISON:-

## 1.) RADIX SORT:- / Bucket sorting:-

15, 1, 321, 10, 802, 2, 123, 90, 109, 11

Step 1: from all the numbers, find out the maximum one.

Step 2: Calculate how many digits are there in max digit

Step 3: we are going to make all no. a 3 digit no.

015,001, 321, 010, 802, 002, 123, 090, 103, 011

O:	010,090	0: 001,802,002,109	0:001,002,010,011
1:	001, 321, 011	1: 010, 011, 015	1: 015, 090 109, 123
2:	802, 002	2: 321, 123	2
3:	123	3:	3 2 1
4:		4:	4
5:	015	5:	5
6:		6:	6
7:		7:	7
8:		8:	8 802
9:	109	9: 090	9

**NOTE :-** The no. of passes in Radix sort is same as the no. of digits in maximum no.

Pass 1: 010, 090, 001, 321, 011, 802, 002, 123, 015, 109

Pass 2: 001, 802, 002, 109, 010, 011, 015, 321, 123, 090

Pass 3: 001, 002, 010, 011, 015, 090, 109, 123, 321, 802.

1, 2, 10, 11, 15, 90, 109, 123, 321, 803

- before 1<sup>st</sup> pass, sort Acc. to unit digit (i.e --0, --0, -1, -2)
  - ~~loop~~ After 1<sup>st</sup> pass, sort Acc. to 2<sup>nd</sup> digit (-0-, -1-, -2-)
  - After 2<sup>nd</sup> pass, sort Acc. to 3<sup>rd</sup> digit (D--, 1--, 2--)

$$O(d \times (n+6))$$

digit ←      ↗ no.  
base (10 here).

Q.) How much time it will take to find min element.

	BC	WC	AC
BST	$O(1)$	$O(n)$	$O(\log n)$
BT	$O(n)$	$O(n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$
Max heap	$O(n)$	$O(n)$	$O(n)$

Q.) How much time it will take to find element  $x$ .

	BC	WC	AC
BST	$O(1)$	$O(n)$	$O(\log n)$
BT (array)	$O(1)$	$O(n)$	$O(n)$
AVL	$O(1)$	$O(\log n)$	$O(\log n)$
Max	$O(1)$	$O(n)$	$O(n)$

Q.) How much time it will take to find element  $x$  not exist?

Ans.)

	BC	WC	AC
BST	$O(1)$	$O(n)$	$O(\log n)$
BT	$O(n)$	$O(n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$
Max	$O(1)$	$O(n)$	$O(n)$

# DYNAMIC PROGRAMMING

## APPLICATIONS:-

1. Fibonacci Series
2. Longest common subsequence (LCS)
3. 0/1 Knapsack
4. Matrix chain multiplication
5. Sum of subset problem
6. All pair shortest path.
- 7) Optimal cost BST.

Fibonacci Series :-

### 1) without DP

```

fib(n)
{
    if (n == 0 || n == 1)
        return(n)
    else
        {
            a = fib(n-1) - T(n-1)
            b = fib(n-2) - T(n-2)
            c = a + b
            return(c)
        }
}

```

$$T(n) = \begin{cases} O(1) & \text{if } n \leq 1 \\ T(n-1) + T(n-2) & n > 1 \end{cases}$$

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

$$SC := i/p + \text{extra}$$

$$= 2B + nB = O(n)$$

table

N	N	N	N	N	N
0	1	2	3	4	

with DP

```

DR-fib(n)
{
    if (n == 0 || n == 1)
        return(n)
    else
        {
            if (Table[n-1] == N)
                Table[n-1] = DP-fib(n-1)
            if (Table[n-2] == N)
                Table[n-2] = DP-fib(n-2)
            Table[n] = Table[n-1] + Table[n-2]
        }
    return(Table[n])
}

+ C = O(n) {n+1} DFC.

```

SC : i/p + extra

$$2B + \text{stack}(nB) + \text{Table}(nB)$$

$$= 2B + 2nB = O(n).$$

Q.) How many distinct func calls are there in fibonacci fib(n)?

Ans.)  $f(5) \rightarrow f(4) \rightarrow f(3) \rightarrow f(2) \rightarrow f(1) \rightarrow f(0) . = \underline{(n+1)}$

$n+1 = O(n) \rightarrow$  Time complexity.

**NOTE:-** Most function are computing again & again, so we go to DP.

• DP is best when repeating func, If no repetition, dont go for DP.

## Longest Common Subsequence (LCS) :-

(ex1)  $x = (\text{B A B A B A})$

$y = (\text{B A B A A B A})$

$$\text{LCS}(6,7) = 1 + \text{LCS}(5,6) - 6$$

$$(6) \quad 1 + \text{LCS}(4,5) - 5$$

$$\Downarrow 1 + \text{LCS}(3,4) - 4$$

$$\Downarrow \max \begin{cases} \text{LCS}(2,4) \\ \text{LCS}(3,3) \end{cases} \xrightarrow{2} 2$$

$$\Downarrow \max \begin{cases} \text{LCS}(2,4) \\ \text{LCS}(3,3) \end{cases} \xrightarrow{3} 3$$

## Recursive Program without DP :-

$\text{LCS}(m, n)$

{

if ( $m == 0$  ||  $n == 0$ )  
return (0)

else

if ( $x[m] == y[n]$ )

return ( $1 + \text{LCS}([m-1], [n-1])$ )

else

{  $a = \text{LCS}(m-1, n) - T(m-1, n)$

$b = \text{LCS}(m, n-1) - T(m, n-1)$

$c = \max(a, b)$

return ( $c$ )

}

}

## Recurrence Relation for Time (WC) :-

$$T(m, n) = T(m-1, n) + T(m, n-1) + c  
= O(2^m 2^n).$$

Space = i/p + extra

$(m+n)$  stack

$\frac{(m+n)}{(m+n)}$

$O(m+n)$

Dynamic Program:-

$LCS(m, n)$

{ if ( $m == 0 \text{ || } n == 0$ )  
    return (0)

else

if ( $x[m] == y[n]$ )

    if ( $\text{Table}[m-1, n-1] == \text{NULL}$ )

$\text{Table}[m-1, n-1] = LCS(m-1, n-1);$

$\text{Table}[m, n] = 1 + \text{Table}[m-1, n-1]$

    return ( $\text{Table}[m, n]$ )

else

{ if ( $\text{Table}[m-1, n] == \text{NULL}$ )

$\text{Table}[m-1, n] = LCS(m-1, n)$

    if ( $\text{Table}[m, n-1] == \text{NULL}$ )

$\text{Table}[m, n-1] = LCS(m, n-1)$

$\text{Table}[m, n] = \max(\text{Table}(m-1, n), \text{Table}(m, n-1));$

    return ( $\text{Table}[m, n]$ );

}

}

Q) How many distinct func calls are there in  $LCS(m, n)$ ?

Aw.)  $LCS(5, 10)$

4  
3  
2  
1  
0

$$DFC = (m+1)(n+1)$$

$$= m \cdot n$$

$$TC = O(mn).$$

$$\underline{(5+1)(10+1)} = DFC$$

Space Complexity - i/p + extra.

$$= (m+n) + \text{stack} + \text{Table}$$

$$= (m+n) + (m+n) + (m \cdot n)$$

$$= \boxed{O(m \cdot n)}$$

with DP  
 $T.C = mn$   
 $S.C = mn$

without DP  
 $\cdot 2^n$   
 $O(n)$

Because of less repetitions  
 $T.C \approx O(1) \text{ ks using DP}$   
 is approx equal to  $2^n$ ,  
 $= NP \text{ complete problem}$

## O/I Knapsack Problem:-

(Same for Sum of Subset Problem)

OIKS ( $m, n$ )

{

if ( $m == 0 \text{ || } n == 0$ )  
 return (0)

else

if ( $w[n] > m$ )  
 return (OIKS ( $m, n-1$ ))

else

{  $a = OIKS(m - w[n], n-1) + P[n]$

$b = OIKS(m, n-1)$

$c = \max(a, b)$

return (c)

}

}

Recurrence Relation:- (1.)  $OIKS(m, n) = \begin{cases} 0 & \text{if } m=0 \text{ or } n=0 \\ OIKS(m, n-1) & \text{if } w[n] > m \\ \max\{OIKS(m - w[n], n-1) + P[n] \\ OIKS(m, n-1)\} & \text{if } w[n] \leq m \end{cases}$

RF - TC - WC:-

$$\begin{aligned} T(m, n) &= T(m - w[n], n-1) + T(m, n-1) + c \\ &= C \cdot 2^n \\ &= O(2^n). \end{aligned}$$

$$\begin{aligned} S.C. &= \frac{nB + nB}{n} \\ &= O(n). \end{aligned}$$

Q.) How many DF C in O/I Knapsack with DP?

Ans.)  $OIKS(5, 10)$

4, 10  
 3, 9  
 2, 8

$\frac{1}{m+1}$        $\frac{n+1}{n+1}$

$$T.C = O(mn).$$

i/p.      stack      Table

$$\begin{aligned} S.C. &= n + (n+1) + mn \\ &= O(mn). \end{aligned}$$

$$\text{without DP} = TC = O(n^n) \\ SC = O(n)$$

35

### 2nd Matrix chain Multiplication:-

Q.) How many distinct function calls are there in  $MCM(1, N)$ ?

Ans)  $MCM(1, n)$

$$\frac{2}{3} \quad \frac{3(n-1)}{2} \\ \frac{3}{4} \quad \frac{2(n-2)}{3} \\ \vdots \quad \vdots \\ \frac{1}{n} \quad \frac{1}{n} = n^2 \rightarrow DFC$$

$$TC = O(n^3)$$

$$SC := n + n + n^2$$

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

- $TC$  of floyd-warshall algorithm :  $O(n^3)$   
 $\Rightarrow$  (positive, negative, undirected)
- Topological sorting :  $O(m+n)$ .

## Master theorem for Decreasing function:-

$$T(n) = aT(n-b) + f(n) \quad a > 0, b > 0, k \geq 0.$$

if  $f(n) = O(n^k)$ , then

1. If  $a < 1$  then  $T(n) = O(n^k)$
2. If  $a = 1$  then  $T(n) = O(n^{k+1})$
3. If  $a > 1$  then  $T(n) = O(n^k a^{n/b})$

1. $T(n) = T(n-1) + 1$	$\rightarrow O(n)$	} $a=1$
2. $T(n) = T(n-1) + n$	$\rightarrow O(n^2)$	
3. $T(n) = T(n-1) + \log n$	$\rightarrow O(n \log n)$	
4. $T(n) = T(n-1) + n^2$	$\rightarrow O(n^3)$	
5. $T(n) = T(n-2) + 1 \rightarrow n/2$	$\rightarrow O(n)$	
6. $T(n) = T(n-100) + n \rightarrow \frac{n}{100} \times n \rightarrow O(n^2)$		
7. $T(n) = 2T(n-1) + 1$	$\rightarrow O(2^n)$	} $a > 1$
8. $T(n) = 3T(n-1) + 1$	$\rightarrow O(3^n)$	
9. $T(n) = 2T(n-1) + n$	$\rightarrow O(n^2)$	
10. $T(n) = 2T(n-1) + \log n$	$\rightarrow O(2^n \log n)$	
11. $T(n) = 2T(n-2) + n$	$\rightarrow O(n 2^{n/2})$	

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

$a \geq 1, b > 1$  are constantly  $f(n)$  is the function.

case 1: If  $f(n) = O(n^{\log_b a - \epsilon})$  where  $\epsilon > 0$  is constant.  
 $T(n) = O(n^{\log_b a})$

case 2: If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  where  $\epsilon > 0$  is constant.  
 $T(n) = O(f(n))$

case 3: If  $f(n) = \Theta(n^{\log_b a} \cdot (\log n)^k)$  where  $k$  is constant  $k \geq 0$ .  
 $T(n) = O(n^{\log_b a} \cdot (\log n)^{k+1})$

case 1.) Comparing  $f(n)$  and  $n^{\log_b a}$ ,  $n^{\log_b a}$  is polynomial time greater than  $f(n)$ . ( $n^\epsilon, \epsilon$  is constant).

case 2.) Comparing  $f(n)$  and  $n^{\log_b a}$ ,  $n^{\log_b a}$  is polynomial time smaller than  $f(n)$ .

case 3.)  $f(n)$  and  $n^{\log_b a}$  are asymptotically equal.

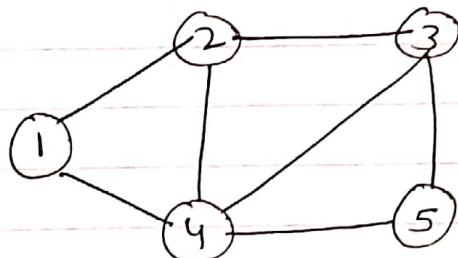
## GREEDY TECHNIQUE

### APPLICATIONS OF GREEDY TECHNIQUES:-

1. Knapsack
2. Job Sequencing with deadlines
3. Huffman Coding
4. Optimal Merge Pattern
5. Minimum Cost Spanning Tree
  - (i.) Prims
  - (ii.) Kruskal
6. Single Source Shortest Path
  - i.) Dijkstra's algo
  - ii.) Bellman Ford algorithm
  - iii.) BFT

### Graph Representation:-

- 1.) Adjacency Matrix :- It is a matrix  $A[v][v]$ , where  $v$  is no. of vertices &  $\{ a[i][j] = 1 , \text{if } i \& j \text{ are adjacent} \}$   
 $= 0 \quad \text{otherwise}$



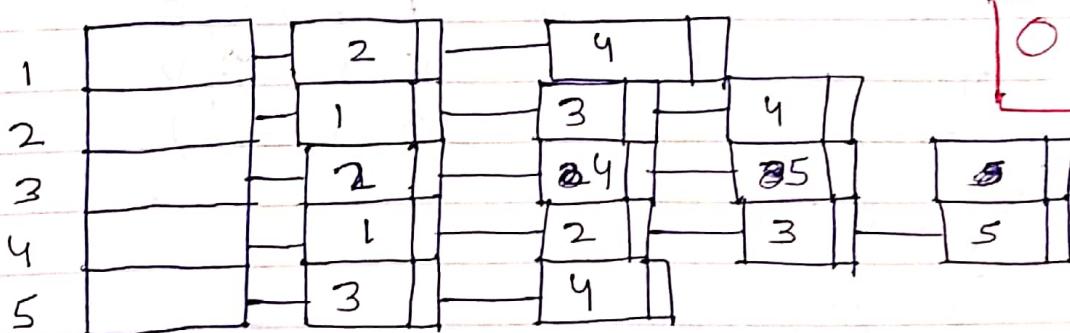
	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

$O(v^2)$  space  
{EC}

5x5

- 2.) Adjacency List :-

Written 2 times  $\rightarrow (1, 2) \& (2, 1)$



$O(v+2e)$  space

- Dense graph - (Complete graph) — Use Adjacency Matrix.
- Less edges - Sparse graph — Use Adjacency List.
- By Default Adjacency List.

V-1

- To check Adj List or not,  $O(1)$  - BC,  $O(V)$  - worst case
- To check Adj Matrix,  $O(1)$  - EC.
- Degree of Adj Matrix,  $-O(V)$  {EC}
- Adj List,  $-O(1)$  BC  
 $O(V)$  WC

- 1) Knapsack Problem :-  $O(n \log n)$  {EC}
- 2) Job Sequencing :-  $O(n \log n)$  {BC}  
 $n^2 + n \log n \Rightarrow O(n^2)$  (WC)
- 3) Bellman Ford :-  $O(mn)$  - (WC).

### DJIKSTRA'S ALGORITHM:-

- 1) Using A-list, minheap  
 $= V \log V + V + E + E \log V$   
 $= V \log V + E \log V$   
 $= O[(E+V) \log V]$

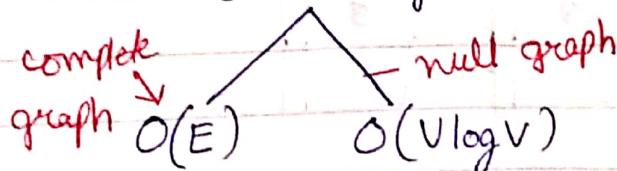
$$WC = O[E \log V] \Leftrightarrow V$$

- 2) Using A-Matrix, Minheap  
 $= V \log V + V + V^2 + E \log V$   
 $= V^2 + E \log V$   
 $= O[V^2 + E \log V]$   
 $WC = O[E \log V]$

- 3) Fibonacci heap & A-list  
 $T_C = O(E + V \log V)$

### KRUSKAL'S ALGORITHM:-

$$\begin{aligned} BC: &= E + (V-1) \log E \\ &= E + V \log V \\ &= O(E + V \log V). \end{aligned}$$



$$\begin{aligned} WC: &- E + E \log E \\ &E + E \log V \\ &= O(E \log V) \end{aligned}$$

### PRIOM'S ALGORITHM:-

1.) Using Minheap, A-list  
 $= V \log V + 2E + E \log V + V$   
 $= (V + E) \log V$   
 $= O[(V+E) \log V]$

2.) Using Minheap, A-Matrix  
 $= V \log V + V^2 + E \log V + V$   
 $= O(V^2 + E \log V)$

3.) Using sorted array, A-list  
 $= V + 2E + EV + V$   
 $= O(EV)$

4.) adj. Matrix, Sorted Double L.L.  
 $= V + V^2 + EV + V$   
 $= O(V^2 + EV)$

5.) A-list & normal unsorted array.  
 $= V^2 + 2E + E + V$   
 $= O(V^2).$

⇒ No. of simple graph possible with 'n' vertices.

$$= 2^{\text{Max edges}} = 2^{\frac{n(n-1)}{2}}, \text{ where Max.edges} = \frac{n(n-1)}{2}$$