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];
    }
    else
    {
        mid = ⌊(i+j)/2⌋          // Divide — O(1)
        (max1, min1) = DAC-maxmin(a, i, mid);        // conquer
        (max2, min2) = DAC-maxmin(a, mid+1, j);
        if( max1<max2 )          // combine
            max = max2;
        else
            max = max1;
        if( min1<min2 )
            min = min1;
        else
            min = min2;
    }
    return(max,min);
}
```

small → O(1)

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

combine ⇒ O(1)

Big

for findmaxmin

Recurrence Relation:-

1.) $T(n)$ = Time Complexity

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

$= O(n)$

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

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

$= 1.5n-2$

$= O(n)$.

Space Complexity :- i/p + Extra

n Bytes + stack

$\Downarrow$

$\log n$

$O(n)$

NOTE :- TC for find an element neither max or min $= O(1)$.

TC for find an element neither 2nd min nor max $= O(1)$.

2.) POWER OF AN ELEMENT:-

```
DAC_power (a,n)        ⇒T(n)
{
  if(n==1)
     return a;
  else
  {  mid = ⌊n/2⌋          ⇒ O(1)      — Divide
     b   = DACpower (a, mid) ⇒T(n/2)  — Conquer
     c   = b × b          ⇒ O(1)      — Combine
     return (c)
  }
}
```

Recurrence Relation :- ① $T(n) = TC$ to find a^n.

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

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

$$= O(\log n)$$

② $T(n)$ = No. of Multiplications for above Program.

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 0 + T\left(\frac{n}{2}\right) + 1 & \text{if } n>1 \end{cases} \quad \begin{matrix} \text{Conq} \\ \text{Combine} \end{matrix} \quad = O(\log n).$$

NOTE:- for odd no :- $c = b \times b$; $c = c \times a$;

$T(n) = TC$

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

$T(n)$ = No. of Multiplications

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

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 element + element [x]
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)  ⟶ 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)              } C
       else
       if (x<a[mid])
    return BS(a, p, mid-1, x)  —— T(n/2)
  else return BS(a, mid+1, q, x) —— T(n/2)
  }
}
```

Recurrence Relation:-

$T(n) = TC$

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

WC & AC $= O(\log n)$

SC = n + log n
= n.

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

Merge Sort Algorithm:-

```
MergeSort (a,P,q)
{
  if (P==q)                                   O(1)
    return (a[P])
  else
  {  mid = ⌊(P+q)/2⌋          — 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≤mid && K≤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(n²) – inplace

NOTE:- TC of merge Algo is : O(m+n) or O(n) → for outplace

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

(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.

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

$= O(n^2)$.

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

i/p ⇓ n Byte

extra → Stack (log n) + Merge (n Byte)

(log n + n)

$= O(n)$.

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

← m → ← n → for sorted unequal elements

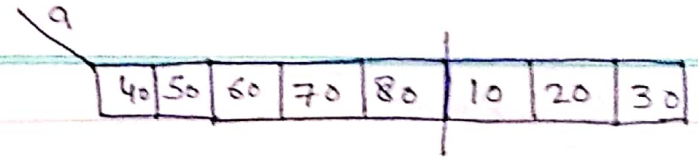
$\frac{n}{2}$ $\frac{n}{2}$ → for sorted equal elements

	Comparisons	Moves
4, 4	4+4-1	4, 4 → 4+4
$\frac{n}{2}, \frac{n}{2}$	n-1	$\frac{n}{2}, \frac{n}{2} \rightarrow n$
m, n	m+n-1	m, n ⇒ m+n

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

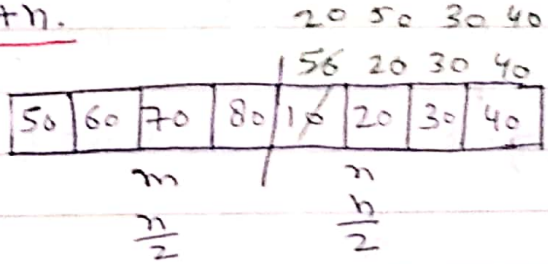
↳ moves ⇒ 4, 4 ⇒ 4+4

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



(ii.) Bestcase outplace

Comparison	Moves
$4,4 = 4$	$4,4 \Rightarrow 8$
$\frac{n}{2}, \frac{n}{2} = \frac{n}{2}$	$\frac{n}{2}, \frac{n}{2} \Rightarrow n$
$m,n \Rightarrow \min(m,n)$	$m,n \Rightarrow m+n.$



iii.) Worst case (Inplace)

m { 50, 10 $\Rightarrow$ 10 $\Rightarrow n$
60, 20 $\Rightarrow$ 20 $\Rightarrow n$

n

$m \cdot n$

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

NOTE:- Merging two ~~sorted~~ subarrays each of size m & n will take $O(m+n)$ 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)  ——  k = partition (a,p,q)      — Pivot element
T(k-p) ——  Quicksort (a,p, k-1);     — Left sort
T(q-k) ——  quicksort (a, k+1, q);    — Right sort
    }
}
```

Partition Algorithm :-

```
Partition (a, p, q)
{
    x = a[p]
    i = p;
    for(j = p+1 ; j≤q ; j++)
    {   if (a[j] ≤ 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]);        ⇒ O(1)
  else
  {
    m = Partition(a, p, q);   — O(n)
    if (m==K)
      return (a[K]);
    else
    if (K<m)
    return (SP(a, p, m-1, K))   — T(m-p)
    else
    return (SP(a, m+1, q, K))   — T(q-m)
  }}
```

$T(n) = TC$,

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

Best case

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

$\downarrow \log n$

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

$= \boxed{O(n)}$

Worst case

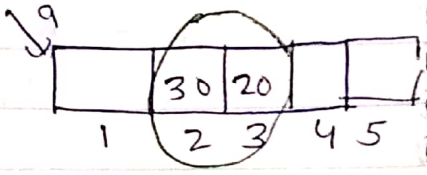
$T(n) = n + T(n-1)$

$= 1 + 2 + \cdots + n$

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

Counting Number of Inversion:-

i/p – An array of n elements
o/p – count inversions



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

```
while(p≤mid && k≤q)
{
   if(a[p]>a[k])
   {
      b[i] = a[k];
      i++; k++;
   NI = NI + mid - p + 1
   }
   else
   { b[i] = a[p];
     i++ ; p++;
   }
} copy remaining element to B.
```

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

$2T(n/2) + n$

$O(n \log n)$.

Strassen's Matrix Multiplication:-

Without DAC

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

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

With DAC

$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$

$= 7T\left(\frac{n}{2}\right) + 4.5n^2$

$O\left(n^{2.81}\right)$

Recurrence Relations & Time Complexity

1.) finding max min 2.) Linear search	$T(n) = \{2T(n/2) + c\}$	$TC = O(n)$
3.) Power of an element 4.) Binary search	$T(n) = \{T(n/2) + c\}$	$TC = O(\log n)$
5.) Merge Sort 6.) ~~Selection Procedure~~ Counting No. of Inversion	$T(n) = \{2T(n/2) + n\}$	$TC = O(n \log n)$
7.) Quick Sort (Randomized Quick sort)	B.C - $T(n) = n^2 + T(m-p) + T(q-m)$ $= n + T(n/2) + T(n/2)$ $= 2T(n/2) + n$ $= O(n \log n)$	W.C $T(n) = n + T(n-1)$ $= O(n^2)$.
8.) Selection Procedure	$T(n) = \{n + T(m-p)\}$ $T(n) = n + T(\frac{n}{2})$ $= n + \frac{n}{2} + \frac{n}{2^2} + \cdots \frac{n}{2^{\log n}}$ $= O(n)$	$T(n) = n + T(n-1)$ $\downarrow n$ $1 + 2 + \cdots n$ $= O(n^2)$
9.) Strassen's Multiplication	without strassen $T(n) = 8T(\frac{n}{2}) + n^2$ $= O(n^3)$	with strassen $T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2$ $= O(n^{2.8})$

with comparison	BC (for quite n^2)	WC	AC	without comparison
1. Bubble Sort	n^2	n^2	n^2	1. counting sort
2. Selection Sort	n^2	n^2	n^2	2. Radix sort
3. Insertion Sort	n	n^2	n^2	3. Bucket sort
4. Merge Sort (out place)	$n \log n$	$n \log n$	$n \log n$	$\Downarrow$
5. Quick Sort (constant factor)	$n \log n$	n^2	$n \log n$	$O(n)$ in Every case
6. Heap Sort (in place)	$n \log n$	$n \log n$	$n \log n$	

TC = swap + comparison, whichever greater will come.

Algorithm	Comparison	Swaps/ Moves
1.) Merge Algorithm		
i.) BC outplace	n/2	n
TC = n	min(m,n)	m+n
ii.) WC outplace	n-1	n
TC = n	m+n-1	m+n
iii.) WC inplace	n-1	m·n
TC = n^2	m+n-1	$\frac{n}{2}\cdot\frac{n}{2}=\frac{n^2}{4}$
2. Build Heap :-	= 2 × swap	$\frac{n}{2}$
TC = n	= n	
	TC = n	
3. Maxheap Inversion	TC = O(logn)	
i.) Linear Search	logn	logn
ii.) Binary Search	logn	logn
4.) Bubble Sort	n^2	zero (min)
If swap=0, then stop algo, because array is sorted, BC = O(n). AC, WC = $O(n^2)$		$\frac{n(n-1)}{2}$ (max)
5.) Selection Sort	$O(n^2)$ ($\frac{n(n-1)}{2}$)	n-1
6.) Insertion Sort		
(i) BC O(n)	n-1	0
(ii) WC & AC $O(n^2)$	n^2	n^2

Dynamic Programming

1.) Fibonacci Series

	without DP	with DP
TC :	$O(2^n)$	O(n)
SC :	O(n)	O(n)
DFC :		n+1

2. LCS :-

	without DP	with DP
TC : (WC)	$O(2^m 2^n)$	O(mn)
SC :	O(m+n)	O(m+n)
DFC :		(m+1)(n+1)

3. Knapsack & Sum of Subset Problem

	without DP	with DP
TC -	$O(2^n)$	O(mn)
SC -	O(n)	O(mn)
DFC -		mn

$(2^n \simeq mn)$ (NP complete)

4. Matrix chain multiplication

	without DP	with DP
TC :	$O(n^n)$	$O(n^3)$
SC :-	O(n)	$O(n^2)$
DFC :-		n^2

5. floyd - warshell 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

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

Height = Level - 1

$H = \log_2 (n+1) - 1$

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

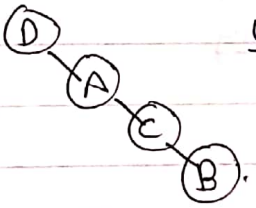
NOTE:- If node is stored in 1st place of the array then.

i.) Parent of $i = \lfloor i/2 \rfloor$
ii.) Left child $(i) = 2 \times i$
iii.) Right child $(i) = (2 \times i) + 1$

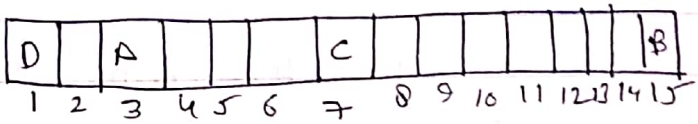
} Array starting from 1. {different when index change}

NOTE:- Array is better for complete & almost complete Binary Tree.
Linked List is better idea for any gap in Nodes.

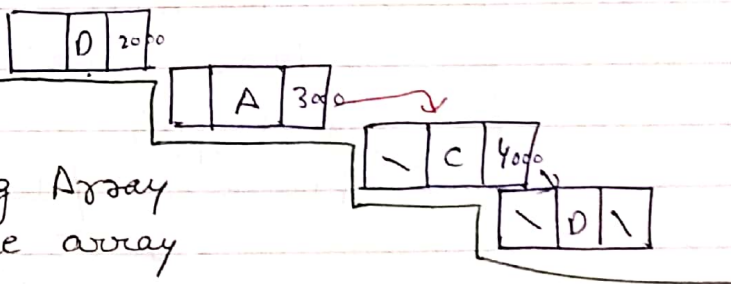
for ex:-



Using Array:-



Using LL:-



Q.) BT - n nodes represented using Array

Min^m size array
n
$\therefore$ In case of ACBT

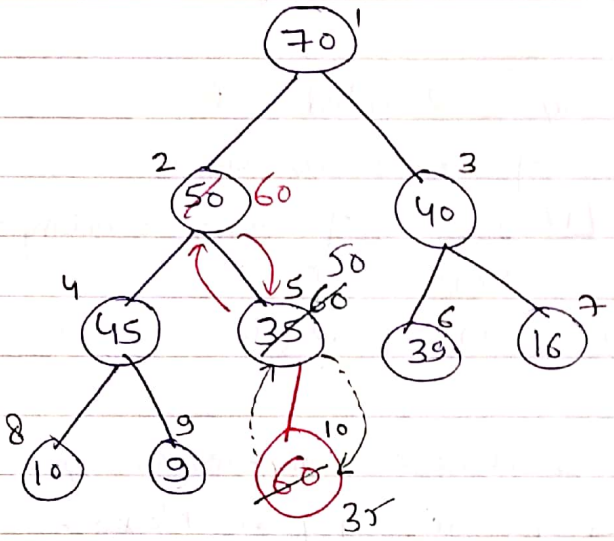
Max^m size array
$2^n - 1$
In case of BT (non ACBT).

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 comparison cost is $O(1)$.

Insertion in Maxheap:- (Bottom to top Approach)



A	70	50	40	45	35 60	39	16	10	9	60 35
	1	2	3	4	5	6	7	8	9	10

Parent of 10 $= \lfloor \frac{i}{2} \rfloor = 5$

	70	60 50	40	45	60 50	39	16	10	9	35
	1	2	3	4	5	6	7	8	9	10

Parent of 5 $= \lfloor \frac{5}{2} \rfloor = 2$

70	60	40	45	50	39	16	10	9	35

Parent of 2 is $= \lfloor \frac{2}{2} \rfloor = 1$.

No swapping.

```
insertHeap(A, n, value)
{   n = n+1;
    A[n] = value;
    i = n;
    while (i > 1)
    { Parent = ⌊i/2⌋
      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 logn.

for n elements → Best case - O(1), WC & AC - O(logn).

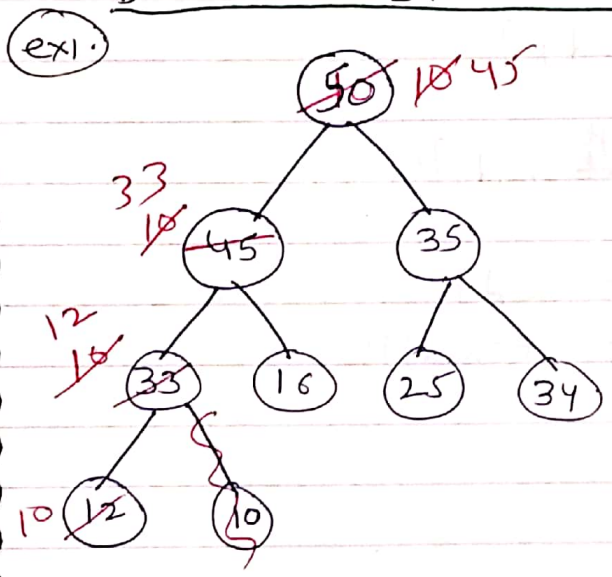
for logn elements - log logn.

Q.) Max heap of $n2^n$ element then insert 1 element }

$\log(n2^n) = \log n + \log 2^n = \log n + n = O(n)$.

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

ex:-



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

child of 1 (10) = 45

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

child of 10 (2) = 33

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

child of 10 (4) = 12

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

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 ⇒ Max heap Tree.

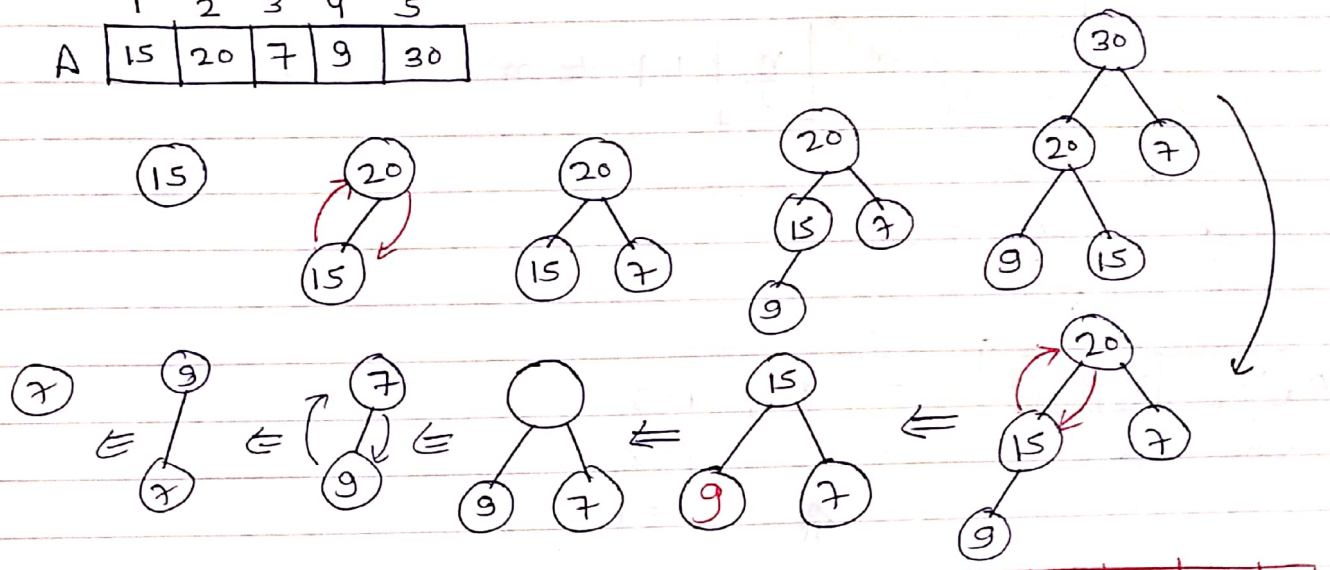
```
Deleteheap (A, n, value)
{
        value = a[i].
    a[i] = a[n]
     n = n-1


  Left child =  2×i  ;
  Right  child = (2×i)+1;

  if ( A[i] < A[Left child] )
   {
       swap (A[i], A[Left child]);
   }
 else if (A[i] < A[Right child])
   {   swap (A[i], A[Right child]);
   }
 else
    { return ;
    }
}
```

Heap Sort:-

	1	2	3	4	5
A	15	20	7	9	30



	1	2	3	4	5
A	7	9	15	20	30

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

$$O(n \log n)$$

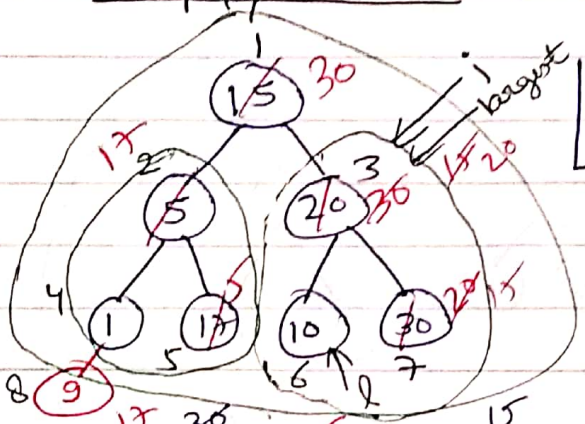
$$O(2n \log n)$$

$$\equiv O(n \log n)$$

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.



$\left\lfloor \frac{n}{2} \right\rfloor + 1$ to n

$O(n)$

	1	2	3	4	5	6	7	8
A	15	5	20	1	17	10	30	9

i=3
i=2
i=1

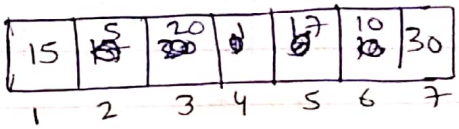
30	17	20	1	5	10	15

```
MaxHeapify (A, n, i)
{   int largest = i ;
    int l = (2×i)   ;
    int r = (2×i)+i ;
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)
{
```

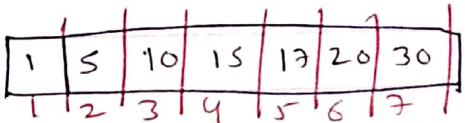
Build Max heap

```
  for (i = n/2 ; i >= 1 ; i--)
    { Maxheapify (A, n, i);
    }
```



Delete

```
  for (i = n ; i >= 1; i--)
    { swap (A[1], A[i]);
      Maxheapify (A, i-1, 1);
    }
```



↳ we are always deleting data from root node thats why always 1.

BUBBLE SORT :-

A	15	16	6	8	5
	0	1	2	3	4

$n = 5$

Pass 1

15 16 6 8 5
15 6 16 8 5
15 6 8 16 5
15 6 8 5 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 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 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 $= (n-1)+(n-2)+ \cdots +1 = \frac{(n-1)}{2} n = O(n^2)$ {Avg}
- Total swaps $= O(\min), \frac{(n-1)n}{2} = O(n^2)$ (max)

$TC = C + S$
$= n^2 + \_\_\_$
$= O(n^2)$.

- 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

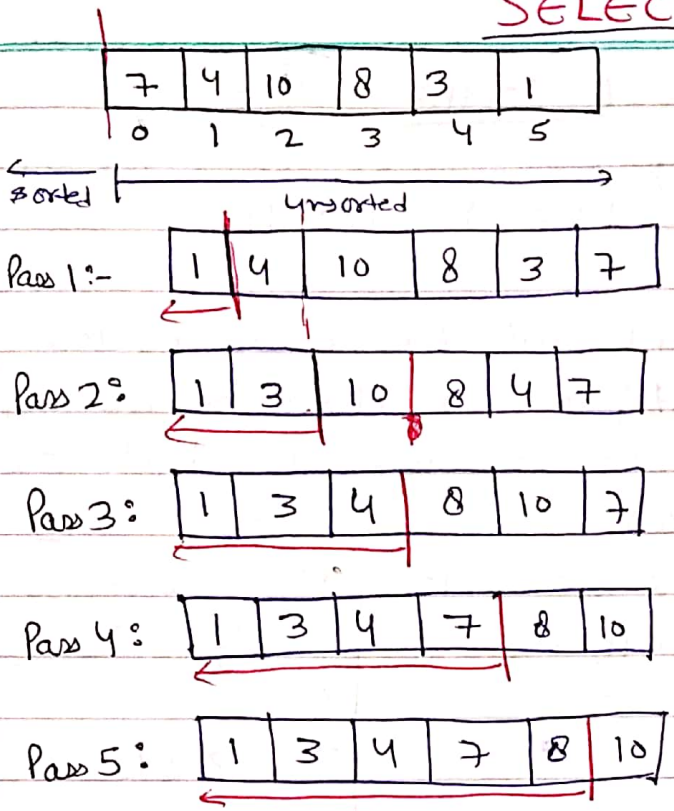
0	1	2	3	4	5
5	4	10	1	6	2

$n = 6$

```
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 = $1+2+\dots+n-1 = n^2$. swaps,
- Average case (Half Asc + Half Desc) = $\frac{n}{2}$ BC + $\frac{n}{2}$ WC = $O(n^2)$.
- If Array size is very small, use Insertion sort.

SELECTION SORT



$n = 6$

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

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

NOTE:- Selection Sort speciality is no. of swaps are (n-1) in every pass $\underline{O(n-1)}$.

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, 109, 011

0: 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:	3 321
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 I^st pass, sort Acc. to unit digit (i.e --0, --0, --1, --2)
- After I^st pass, sort Acc. to 2^nd digit (-0-, -1-, -2-)
- After 2^nd pass, sort Acc. to 3^rd digit (0--, 1--, 2--)

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

(n → no.)

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

	BC	WC	AC
BST	O(1)	O(n)	O(logn)
BT	O(n)	O(n)	O(n)
AVL	O(logn)	O(logn)	O(logn)
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(logn)
BT (array)	O(1)	O(n)	O(n)
AVL	O(1)	O(logn)	O(logn)
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(logn)
BT	O(n)	O(n)	O(n)
AVL	O(logn)	O(logn)	O(logn)
Max	O(1)	O(n)	O(n)

DYNAMIC PROGRAMMING

APPLICATIONS:-

1. Fibonacci Series
2. Longest common subsequence (LCS)
3. 0/1 Knapsack
4. Matric 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)
  }C
}
```

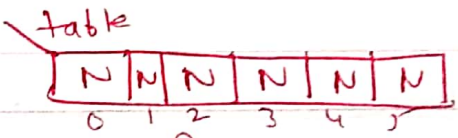
$$T(n) = \begin{cases} O(1) & \text{if } n \le 1 \\ T(n-1)+T(n-2) & n > 1 \end{cases}$$

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

SC:- i/p + Extra

$= 2B + nB = \boxed{O(n)}$

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])
}
```

$\boxed{TC = O(n)}$ {n+1} DFC.

SC : i/p + extra

$2B + Stack(nB) + Tabk(nB)$

$= 2B + 2nB = \boxed{O(n).}$

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

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

$n+1 = O(n) \to$ Time Complexity.

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

- DP is best when repeating func, If no repetion, dont go for DP.

Longest Common Subsequence (LCS):-

ex1: $x = (B\ A\ B\ A\ B\ A)$

$y = (B\ A\ B\ A\ A\ B\ A)$

$LCS(6,7) = 1 + LCS(5,6)$ — 6

$1 + LCS(4,5)$ — 5

$1 + LCS(3,4)$ — 4

$\max \{ LCS(2,4) \Rightarrow 2,\ LCS(3,3) \Rightarrow 3 \}$

Recursive Program without DP:-

```
LCS(m,n)
{
  if (m == 0 || n == 0)
     return (0)
  else
  if (x[m] == y[n])
  return (1 + LCS([m-1],[n-1])        BC → min(m,n)
  else
{  a = LCS(m-1,n)   — T(m-1,n)
   b = LCS(m,n-1)   — T(m,n-1)
   c = max(a,b)
       return (c)          ⇒ 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 $\Downarrow (m+n)$

$O(m+n)$

Dynamic Program:-

```
LCS(m,n)
{   if (m==0 || n==0)
        return (0)
    else
    if (x[m] == y[n])
        if (Table[m-1,n-1] == NULL)
            Table[m-1,n-1] = LCS(m-1,n-1);
        Table[m,n] = 1 + Table[m-1,n-1]
    return (Table[m,n])
    else
    {   if (Table[m-1,n] == NULL)
            Table[m-1,n] = LCS(m-1,n)
        if (Table[m,n-1] == NULL)
            Table[m,n-1] = LCS(m,n-1)
        Table[m,n] = max(Table(m-1,n), Table(m,n-1));
        return (Table[m,n]);
    }
}
```

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

Ans.) LCS(5, 10)

m: 5 4 3 2 1 0
n: 10 9 8 7 6 5 4 3 2 1 0

(5+1)(10+1) = DFC

DFC = (m+1)(n+1)
= m·n

TC = O(mn).

Space Complexity — i/p + extra
= (m+n) + stack + Table
= (m+n) + (m+n) + (m·n)
= O(m·n).

with DP
TC = mn
SC = mn

without DP
2^n
O(n)

Because of less repetitions TC of 0/1 KS using DP is approx equal to 2^n. = NP complete problem.

0/1 Knapsack Problem:- (Same for Sum of Subset Problem)

without DP

```
OIKS (m,n)
{
  if (m==0 || 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.) $OIKs(m,n) = \begin{cases} 0 & \text{if } m=0 \text{ or } n=0 \\ OIKs(m,n-1) & \text{if } w[n]>m \\ \max\{OIKs(m-w[n],n-1)+P[n],\ OIKs(m,n-1)\} & \text{if } w[n]\le m \end{cases}$

RR-TC-WC:-

$T(m,n) = T(m-w[n],n-1) + T(m,n-1) + c$

$= c \cdot 2^n$

$= O(2^n)$.

SC:- $nB + nB$

$= O(n)$.

Q.) How many PFC in 0/1 Knapsack with DP?

Ans.) OIKs (m, n) (5, 10)

4, 10
3, 9
2, 8
1

m+1, n+1

TC = O(mn).

SC:- i/p. + stack + Table

$= n + (n+1) + mn$

$= O(mn)$.

without DP = TC = $O(n^n)$
SC = $O(n)$.

Matrix chain Multiplication:-

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

Ans:) MCM (1, n)

2 — 3(n-1)
3 — 2(n-2)
4 — 1
⋮

$\underline{n} \quad \underline{n} = n^2 \rightarrow DFC$

TC = $O(n^3)$

SC :- $n + n + n^2$

SC = $O(n^2)$.

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

Master theorem for Decreasing function:-

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

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. $T(n) = T(n-1) + 1 \rightarrow O(n)$
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. $T(n) = T(n-100) + n \rightarrow \frac{n}{100} \times n \rightarrow O(n^2)$

} $a=1$

7. $T(n) = 2T(n-1) + 1 \rightarrow O(2^n)$
8. $T(n) = 3T(n-1) + 1 \rightarrow O(3^n)$
9. $T(n) = 2T(n-1) + n \rightarrow O(n2^n)$
10. $T(n) = 2T(n-1) + \log n \rightarrow O(2^n \log n)$
11. $T(n) = 2T(n-2) + n \rightarrow O(n\, 2^{n/2})$

} $a>1$

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

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

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

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

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

case 1:→ Comparing $f(n)$ and $n^{\log_b a}$, $n^{\log_a b}$ is polynomial time greater than $f(n)$. (n^ϵ, ϵ is constant).

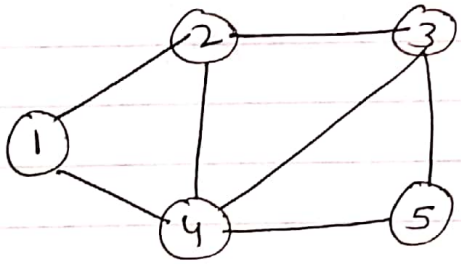
GREEDY TECHNIQUE

APPLICATIONS OF GREEDY TECHNIQUES:-

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

Graph Representation:-

1.) Adjancency Matrix :- It is a matrix A[V][V], where V is no. of vertices & { a[i][j] = 1 , if i & j are adjacent
= 0 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

5×5

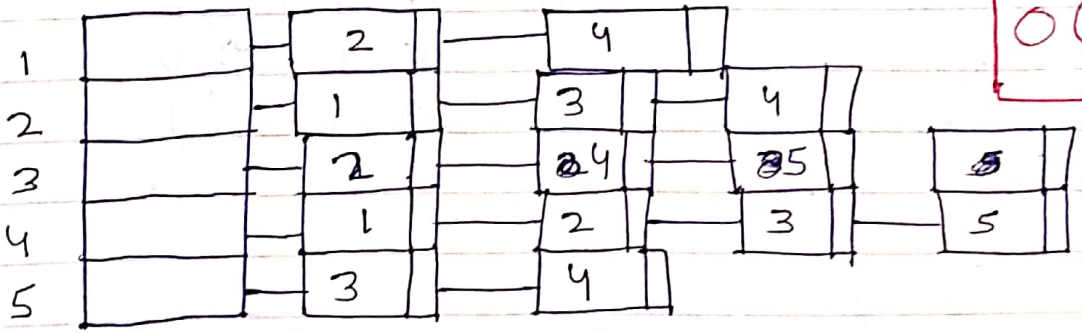
$O(V^2)$ space

{ $\in C$ }

written 2 times → (1,2) & (2,1)

$O(V+2E)$ space

2) Adjancency List :-



- Dense graph - (complete graph) — Use Adjancency Matrix.
- Less edges - sparse graph — Use Adjancency List.
- By Default Adjancency 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(nlogn) {EC}
2.) Job Sequencing :- O(nlogn) {BC}
$n^2 + n\log n \Rightarrow O(n^2)$ (WC)
3.) Bellmand ford :- O(mn) - (WC).

DIJKSTRA'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 $= O[E\log V]$ $E >> V$

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
TC $= O(E + V\log V)$

KRUSKAL'S ALGORITHM :-

BC: $= E + (V-1)\log E$
$= E + V\log V$
$= O(E + V\log V)$

complete graph → O(E)
null graph → O(VlogV)

WC :- $E + E\log E$
$E + E\log V$
$= O(E\log V)$

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