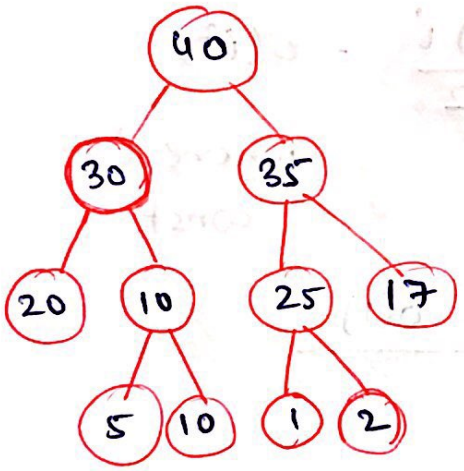
⇒ Heap - tree :-

Max Heap tree

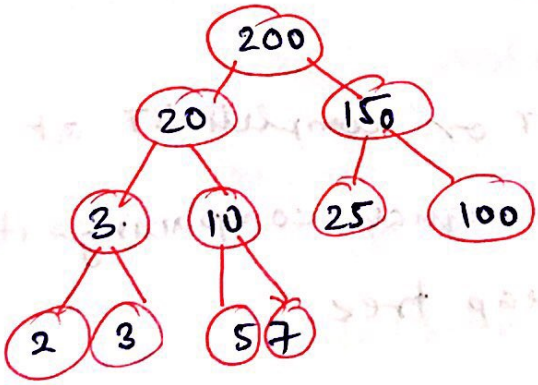
Min heap tree

= Max heap tree :-

In the given almost complete BT or complete BT at every node Root is maximum or equal comparing its children is known as max heap tree.



It is not Max heap tree as it is not ACBT. or CBT.



ACBT, so it is max heap tree

200	20	150	3	10	25	100	2	3	5	7

Max Heap :-

1st max — 1 — 0 — O(1)

2nd max — 2 — 1 — O(1)

3rd max — 3 — 2 — O(1)

⋮ ⋮

10th max — 10 — 9 — O(1)

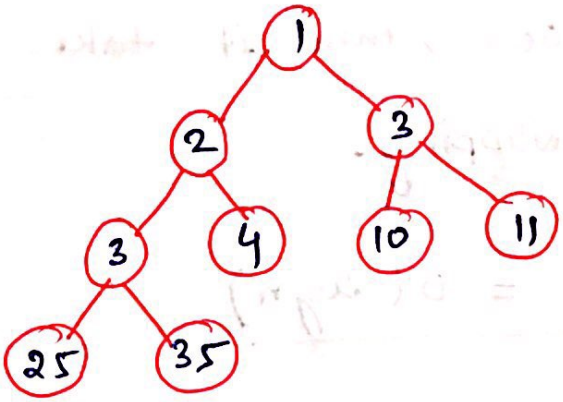
So, for i^{th} max — $\frac{(i-1)\,i}{2} = O(i^2)$

where i is const

So, $T(n) = O(1)$

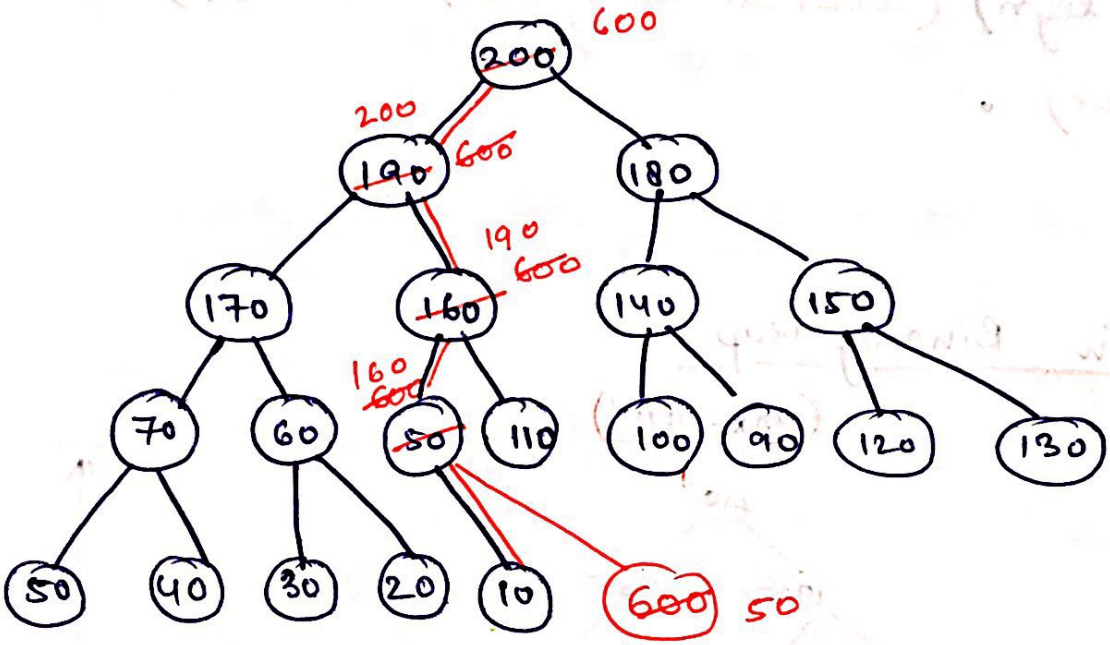
Min Heap tree :-

In the given almost complete BT or complete BT at every node Root is minimum or equal compairing its children is known as min heap tree.



Insertion in Max heap :- (Max heapify) bottom

(child is asking to parent)



n = arraysize (21)

m = last element position (20)

m = m+1

= 21

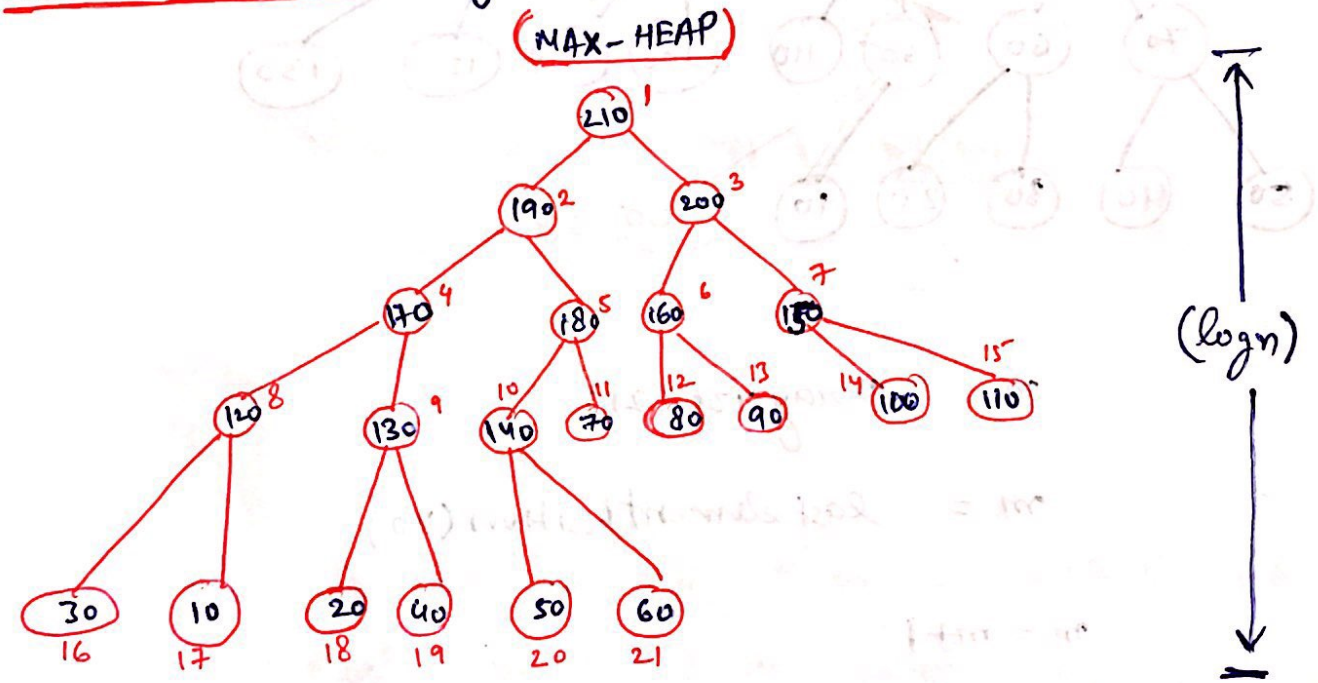
a[m] = x

so, if we insert 600, then it take $\log n$ comparison and swapping.

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

NOTE:- Inserting a element into Minheap or Maxheap which already contain n elements will take $O(\log n)$ (Worst case & Avg case) and $O(1)$ (Best case).

Deletion in Binary Heap :-



210	190	200	170	180	160	150	120	130	140	70	80	90	100	110	30	10	20

40	50	60

n = 21 (arraysize)

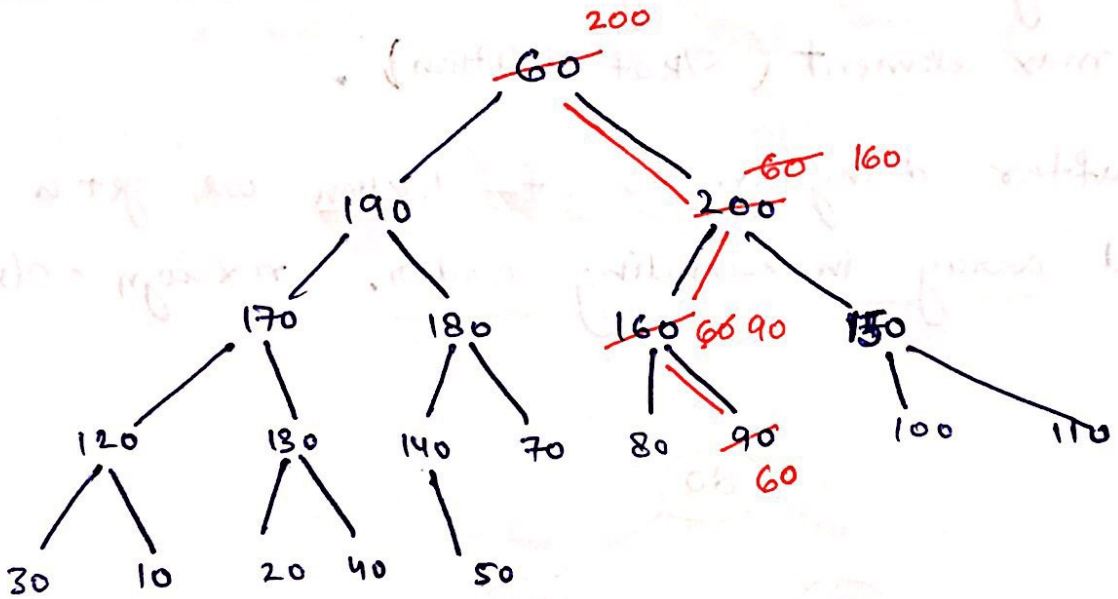
m = 21 (Last element index)

Steps :-

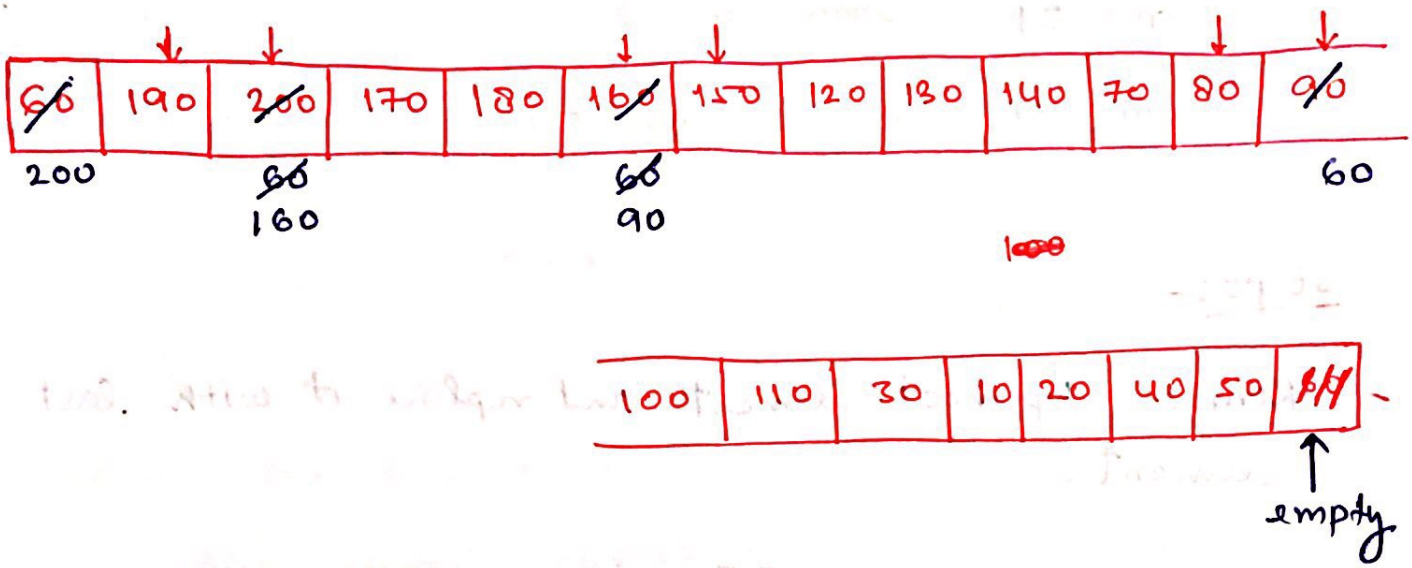
- Remove top root element and replace it with last element.

```
n = a[1];
a[1] = a[m];
```

- Now apply max-heapify top (parent asking child)



- In max-heapify top, each cycle contains 1 comparision of childs, then swapping of root element with max child. Total logn cycles will be performed.

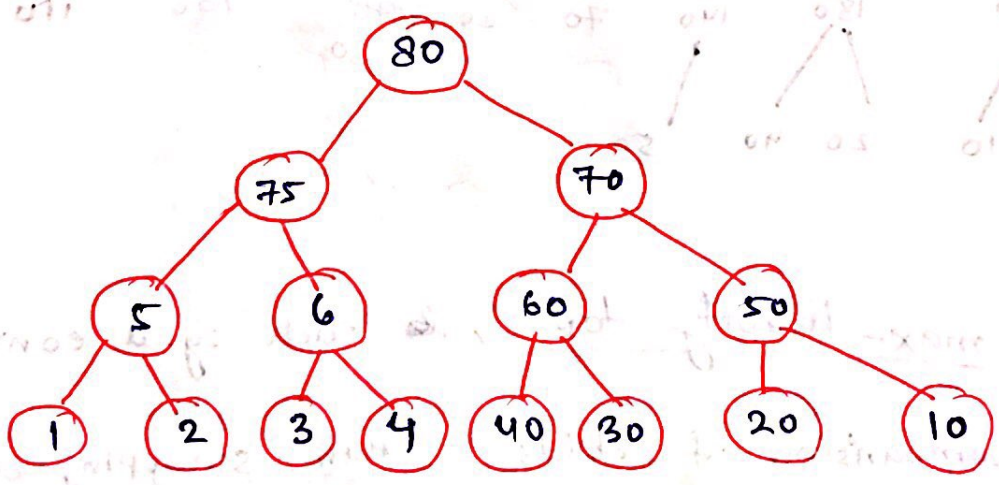


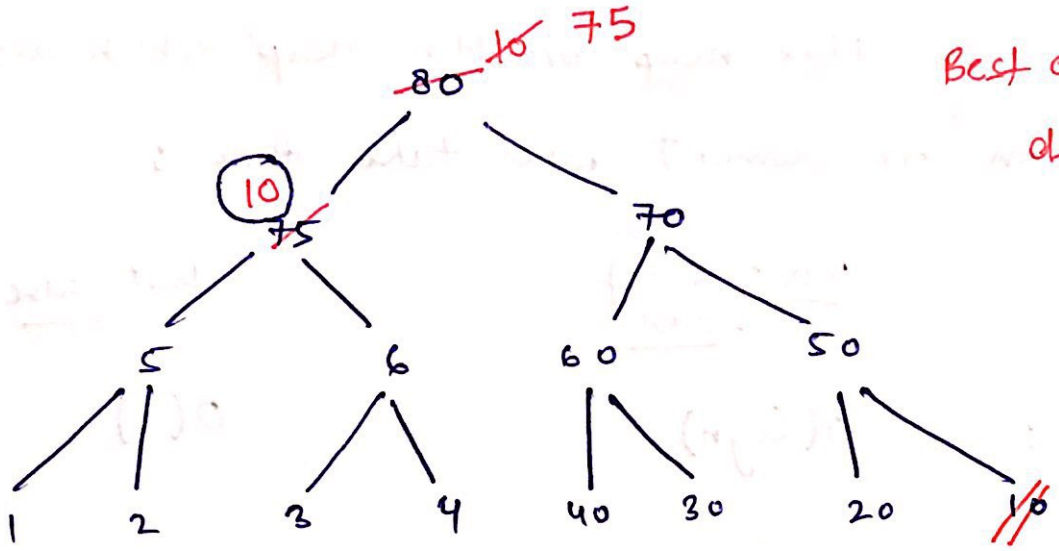
$T(n) = O(\log n)$

NOTE:- As we can see after first deletion last index in array becomes empty which can used to store the max element (o/p of deletion).

Thus after doing n no. of deletion we get a sorted array in ascending order. $n \times \log n = O(n \log n)$.

eg:-



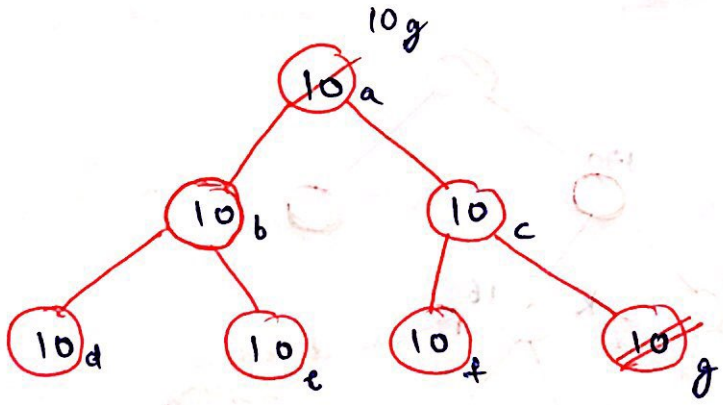


Best case of deletion.

- After one cycle in deletion position of 10 is fixed as 10 is the max element in left subtree.

- So, if the min element of right sub-tree is max element of left subtree then it will give best case with $(T(n) = O(1))$.

eg:-



This will also gives the Best case of deletion.

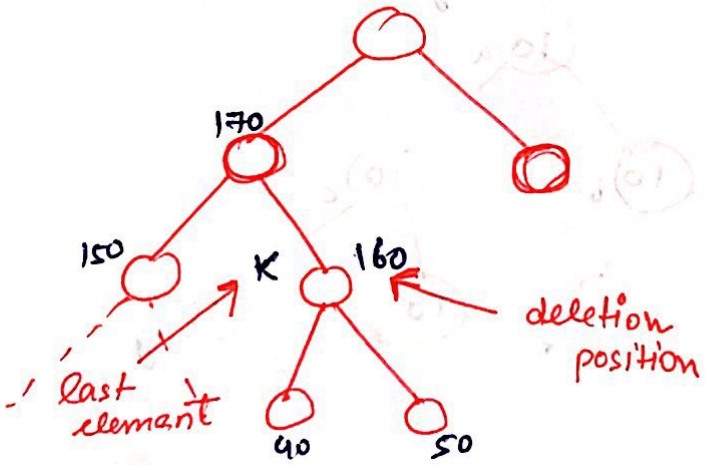
This is also unstable sorting (Heap sort) as it swap same elements.

Note :- In Max-heap or Min-Heap which already contain n element will take time :

	Worst & avg -case	Best-case
Deletion :	$O(\log n)$	$O(1)$
Insertion :	$O(\log n)$	$O(1)$

* Heap-sort is inplace and unstable with $T(n) = O(n \log n)$.

⇉ **Deletion in Heap (Max or Min) at random position :-**



if $(K = 150)$ then both childs & parent are satisfied $O(1)$

if (K = 200) then parent is not satisfied

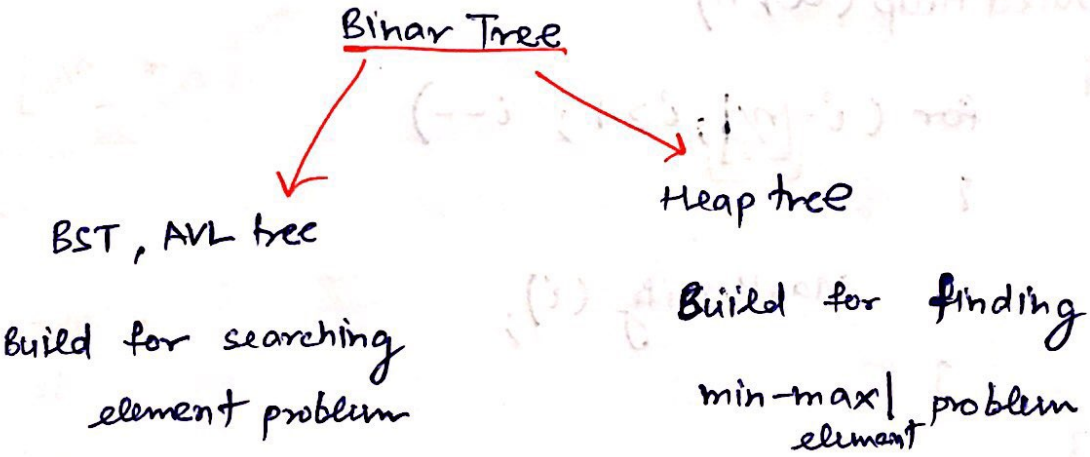
Max-heapify bottom $O(\log n/2)$

if (K = 20) then child is not satisfied

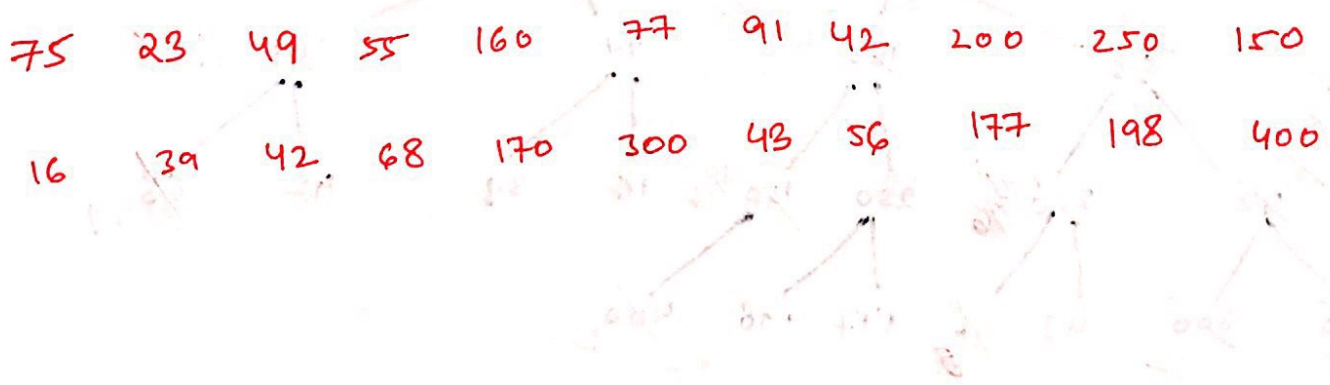
Max-heapify top $O(\log n/2)$

So, overall $T(n) = O(\log n)$

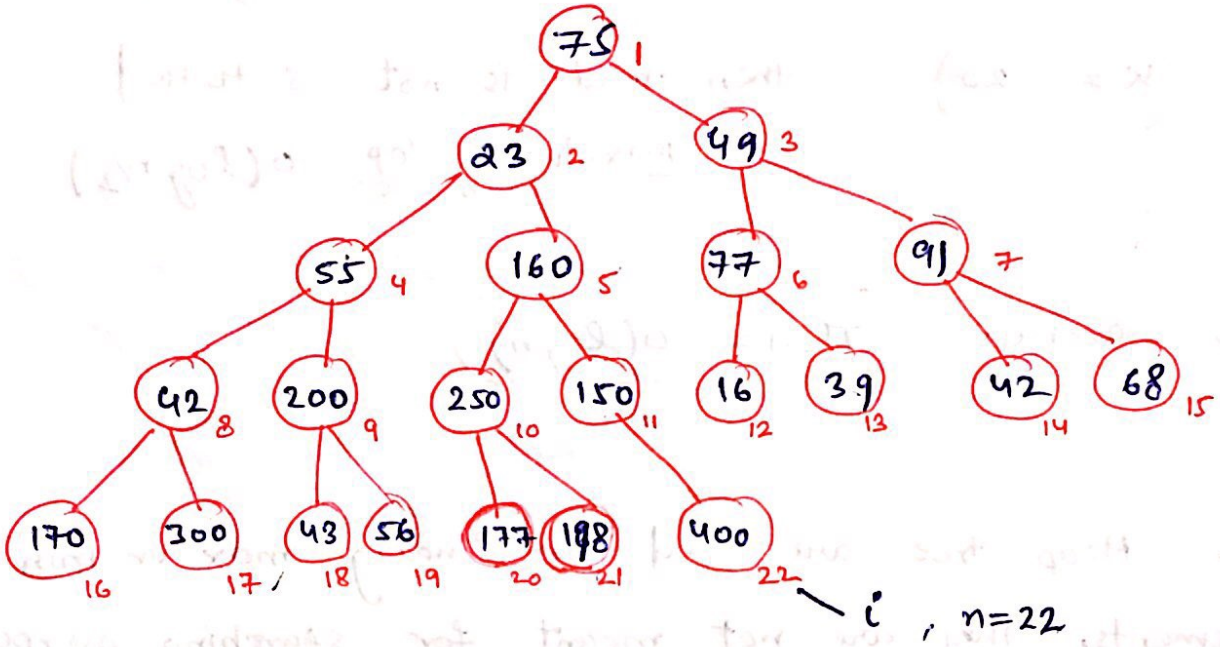
NOTE:- Heap tree are used for finding max or min elements. They are not meant for searching purpose.



Build Heap (Creating Min or Max heap in O(n)) :-



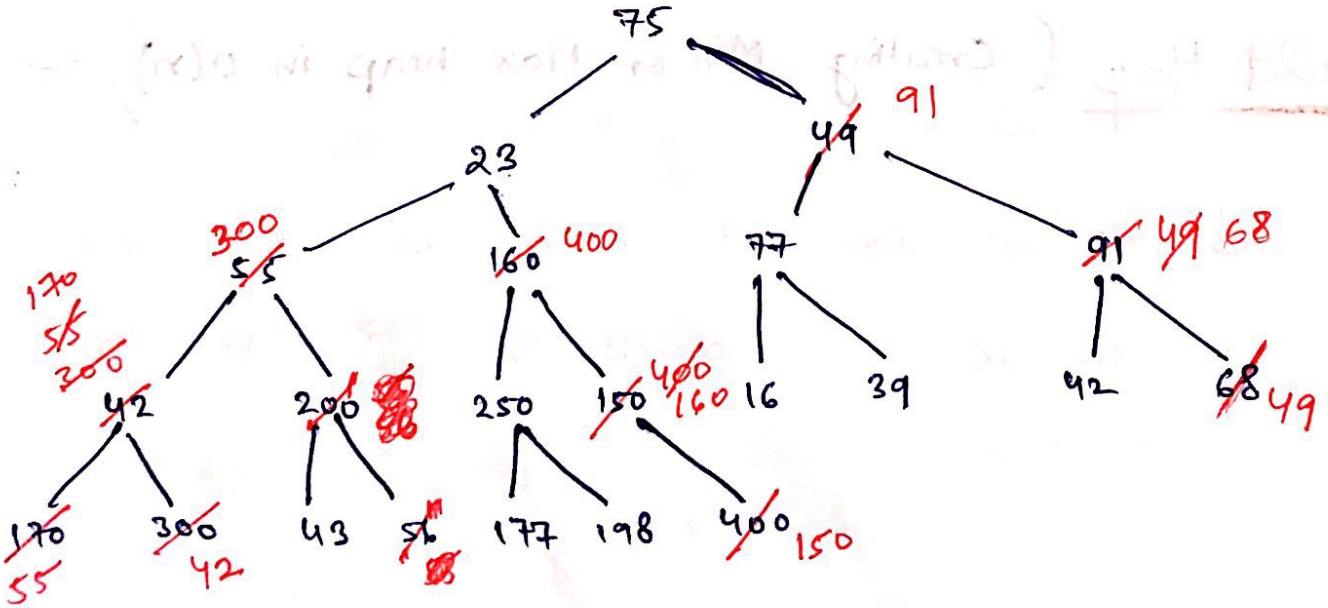
(i) Create almost complete binary tree :-

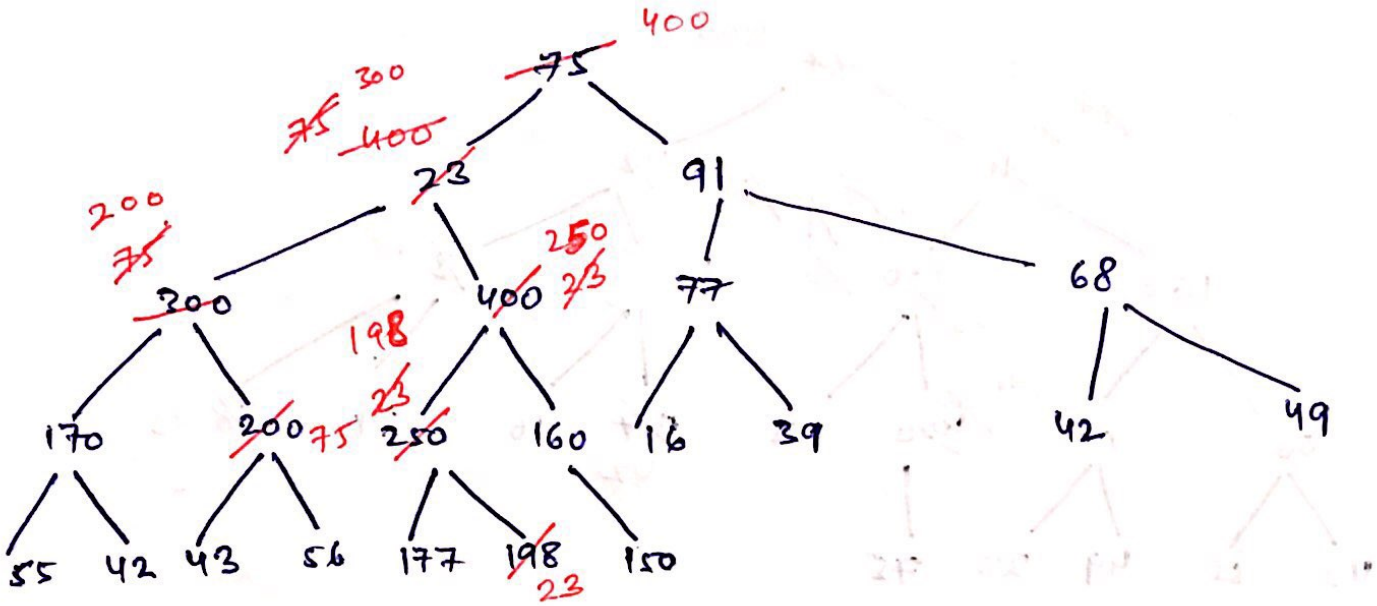


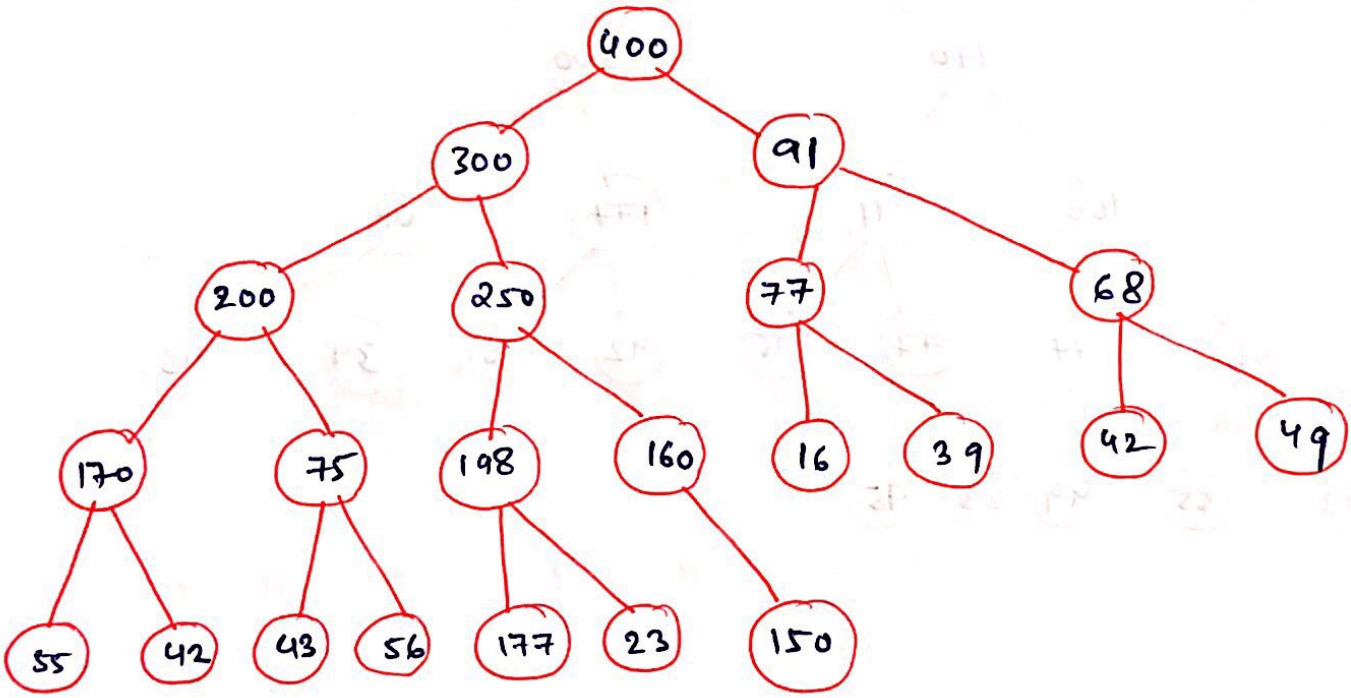

```
Build Heap (a, n)
{
    for (i = ⌊n/2⌋; i ≥ 1; i--)
    {
        MaxHeapify (i);
    }
}
```

* $\lfloor n/2 \rfloor$ ~~max~~ is the no. of non-leaf nodes.

* There is no need to apply max-heapify at leaf nodes.





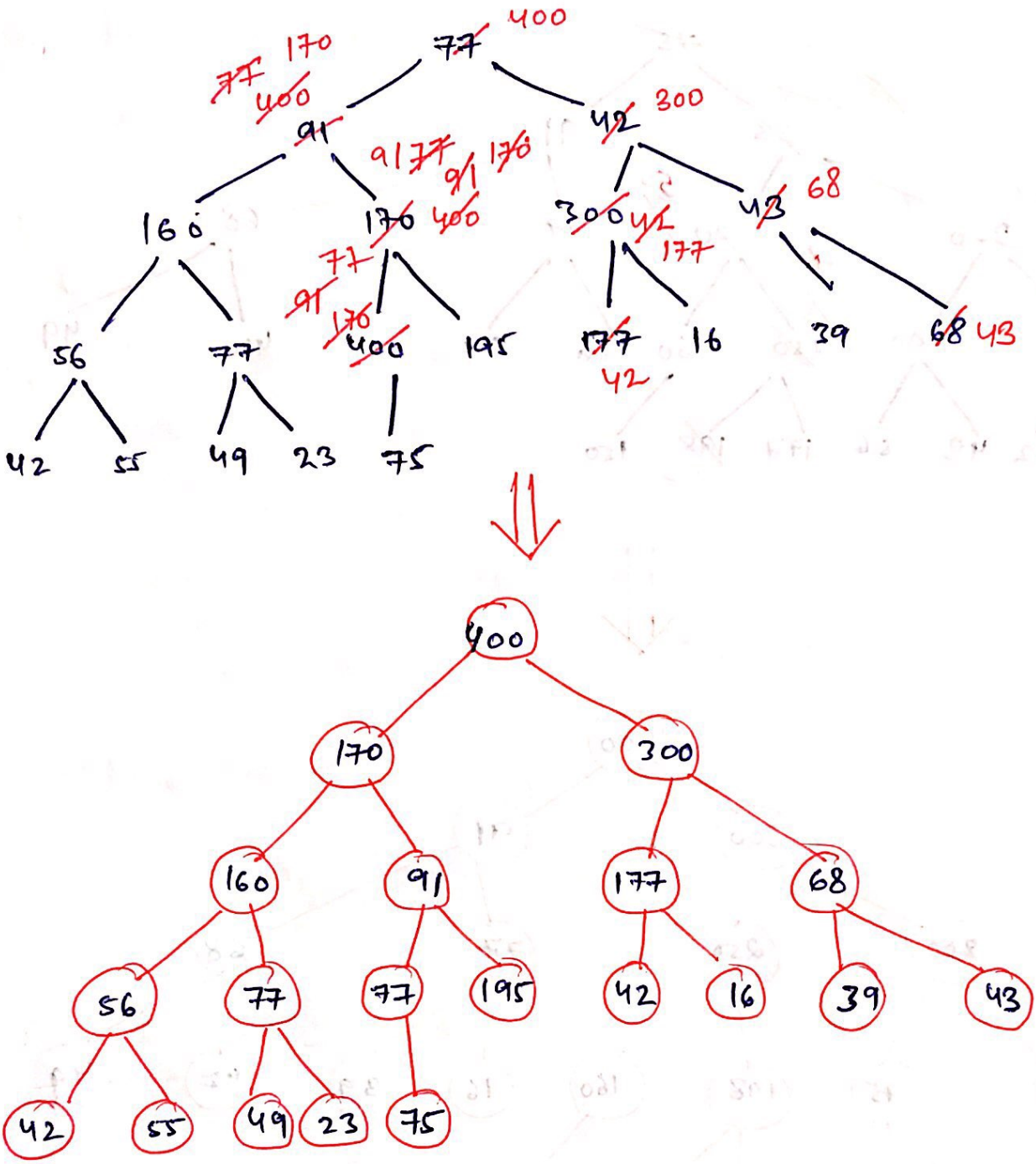


eg:

77 91 42 160 170 300 43 56 77

400 195 177 16 39 68 42 55 49

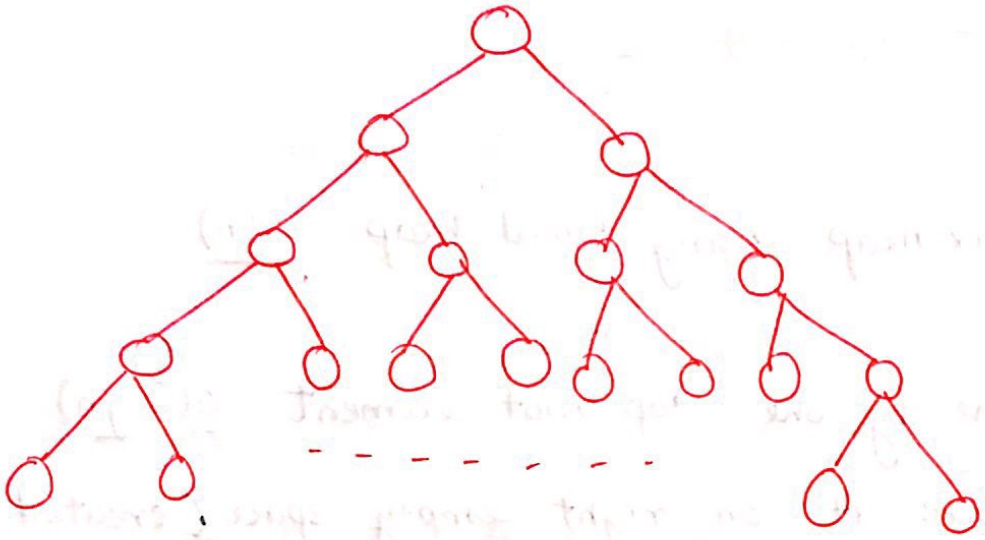
23 75



Time complexity for Build Heap :-

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

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



$n/2^{\log_2 n}$	$\log n - 1$
$n/2^4$	3
$n/2^3$	2
$n/2^2$	1
$n/2$	0

Total nodes at each level

No. of possible swaps in maxheapify

Total swaps $= n/2^2 \times 1 + n/2^3 \times 2 + n/2^4 \times 3 + \cdots + n/2^{\log_2 n} \times \log_2 n - 1$

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

$= n/2 \times \text{const}$

$= O(n)$

Since no. of swaps are the main steps in Build Heap algorithm so

$$T(n) = O(n)$$

⇒ Heap - Sort Algorithm:-

(i) Create max-heap using Build heap $O(n)$

(ii) Delete one by one top root element $O(\log n)$ and store it on right empty space (created due to shift of last element to root position)

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

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

⇒ Bubble Sort :-

I/p :- 75 25 45 90 15 12 11 .

PASS 1 : 25 45 75 15 12 11 (90)

PASS 2 : 25 45 15 12 11 (75) (90)

PASS 3 : 25 15 12 11 (45) (75) (90)

PASS 4 : 15 12 11 (25) (45) (75) (90)

PASS 5 : 12 11 (15) (25) (45) (75) (90)

PASS 6 : (11) (12) (15) (25) (45) (75) (90)

* $n-1$ passes are required only.

* It is inplace and stable

- Each pass has comparision and swap sub tune.

for $n = 7$

PASS-1 : 6 comparisions + (0,6) swapping

PASS-2 : 5 comparisions + (0,5) swappings

⋮

PASS $n-1$: 1 comparisim + (0,1) swaps

so, Time complexity = Total comparisim + Total swaps

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

$T(n) = O(n^2)$ Upper bound

swap with temp (t) variable		swap without temp variable	
space ↑ time ↓	$t = a$ $a = b$ $b = t$	$a = a+b$. $b = a-b$ $a = a-b$	space ↓ time ↑

eg:-

50 10 20 30 40

PASS 1 10 20 30 40 50 swap = 4

PASS 2 10 20 30 40 50 swap = 0

STOP

So, while performing bubble ~~so~~ sort, if swap = 0 at any pass then stop as array is already sorted. So by this modification:

$$T(n) = kn = O(n)$$ Best case

k / const

Selection Sort :-

I/p : 80 15 19 90 11 13 23

1 2 3 4 5 6 7

~~PASS 1 : 80 19 90 15 13 23 11~~

PASS 1 : i = 1

min = ~~1~~ 2 5 , Now ~~a[i]~~ =

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

(11 15 19 90 80 13 23)

PASS 2 : $i = 2$ swap (a[i], a[min])

$min = 2$ ~~2~~ 6

11	13	19	90	80	15	23
1	2	3	4	5	6	7

PASS 3 : 11 13 15 90 80 19 23 min = 6

PASS 4 : 11 13 15 19 80 90 23 min = 6

PASS 5 : 11 13 15 19 23 90 80 min = 7

PASS 6 : 11 13 15 19 23 80 90 min = 7

- In this, single swap is done at each pass.

(i) Selection sort required $n-1$ passes

(ii) Total comparision is $(n-1, n-2, \ldots\ldots 2, 1) = \frac{n(n-1)}{2}$

(iii) Total swaps is $n-1$ (all cases)

if $(i = min)$ then also we have to swap (overwrite itself)

$T(n) =$ comparisions + swap

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

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

- It is inplace & unstable.

Advantage:- Less number of swap opⁿ (Worst case)

⇉ Insertion Sort:-

I/p: 50 70 90 75 55 10 200 300 75 79

1 2 3 4 5 6 7 8 9 10

PASS 1: 50 70 90 75 55 10 200 300 75 79

PASS 2: 50 70 90 75 55 10 200 300 75 79

PASS 3: 50 70 75 90 55 10 200 300 75 79

PASS 4: 50 55 70 75 90 10 200 300 75 79

PASS 5: 10 50 55 70 75 90 200 300 75 79

PASS 6: 10 50 55 70 75 90 200 300 75 79

PASS 7: 10 50 55 70 75_a 90 200 300 75_b 79

PASS 8: 10 50 55 70 75_a 75_b 90 200 300 79

$$T(n) = \underbrace{(1+2+3+\ldots+n-1)}_{\text{comparision}} + \underbrace{(n\times-1\times1)}_{\text{swaping}}$$

$$= \frac{n(n-1)}{2} + n = O(n^2)$$ Worst & avg. case

Best Case :-

I/p : 10 20 30 40 50

		Comparision	Swap
PASS 1	10 20 30 40 50	1	0
PASS 2	10 20 30 40 50	1	0
PASS 3	10 20 30 40 50	1	0
~~PASS~~		= (n-1) × 1	0

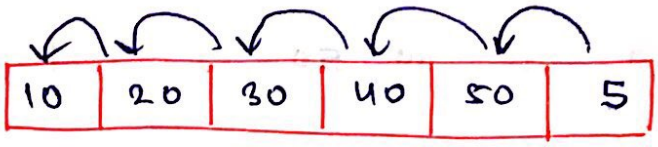
$T(n) = O(n)$ almost ascending order

- To sort n-elements it takes n-1 comparision in best case

- If array already sorted ~~then~~ or almost sorted then insertion sort algorithm will take $O(n)$ but Quicksort will take $O(n^2)$.

- No. of swap op^n = no. of inversions in the given array

- if in the given array atmost n inversions are there, then we can say array is almost sorted.



(n-1) inversions, array is almost sorted

So, $T(n) = O(n)$ ascending order

Worst case :-

I/p : 50 40 30 20 10

		CMP	SWAP
P_1 :	40 50 30 20 10	1	1
P_2 :	30 40 50 20 10	2	2
P_3 :			
:			
P_{n-1} :	10 20 30 40 50	n-1	n-1
		$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$

$T(n) = O(n^2)$ Descending order

Average case:-

$$T(n) = \frac{1}{2} \text{ Best case} + \frac{1}{2} \text{ worst case}$$

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

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

Advantage :-

	Best case
Insertion sort	$O(n)$
Merge sort	$O(n \log n)$
Quick sort	$O(n^2)$

Insertion sort worst case :-

using Linear search		using Binary search	
1-cycle	for n-cycle	i-cycle	for n-cycle
n cmp		log n cmp	
n swap	$n \times 2n$	n swap	$n(n+\log n)$
$2n$	$= O(n^2)$	$n+\log n$	$= O(n^2)$

Sorting Techniques

with comparision

	BC	AC	WC
1. Bubble sort	n^2	n^2	n^2
2. Selection sort	n^2	n^2	n^2
3. Insertion sort	n	n^2	n^2
4. Merge sort	$n\log n$	$n\log n$	$n\log n$
5. Quick sort	$n\log n$	$n\log n$	n^2
6. Heap sort	$n\log n$	$n\log n$	$n\log n$

without comparisions

1. counting sort
2. Radix sort
3. Bucket sort

Linear time sorting algo

NOTE:- In all comparision based sorting techniques, min worst case $T(n) = O(n\log n)$, whereas min best case $T(n) = O(n)$.