Sorting Techniques

Insertion Sort

```
Insertion sort (A)
{
 for j = 2 to A.length {
   Key = A[j]
  // insert A[j] into sorted sequence A[1 ... j-1]
    i = j-1;
   while ( i>0 and A[i] > Key) {
      A[i+1] = A[i]
       i = i-1 }
    A[i+1] = key
 } }
```

In Best Case

1 2 3 4 5 6 7 8 9

At	j	no of comparison	no of movement
	2	1	0
	3	1	0
	:	1 (Just 1 comparison)	:
	n	1	0

Total time = ~~1+2+3+...+(n-1)~~

= 1+1+1+1+1...(n-1)

= $\Omega(n)$

In worst Case

At	j	no of comparison	no of movement
	2	1	1
	3	2	2
	4	3	3
	:	:	:
	n	(n-1)	(n-1)

Total time = $2(1+2+\cdots+(n-1))$

$= \frac{2(n-1)n}{2}$

$= O(n^2)$

Space Complexity = $n + 3$

↳ Already given

So extra space is not required = $O(1)$

Called as Inplace Algorithm.

To reduce time we can apply binary search instead of linearsh search to $T(n) = O(\log n)$ (Comparison)

But no of movements will take $O(n)$.

On using doubly linked list

	Comparison	Movement	
Linear search	O(n)	O(1)	→ O(n)

But we can't apply binary search on doubly linked list.

So finally, for Insertion sort

Time complexity = $O(n^2)$

Space ——— = O(1).

Merge Short Algorithm

Good in terms of Time Complexity then Insertion or quick short.

Merging

Let us consider an array of n elements. Say (n=8)

A =

1	2	3	4	5	6	7	8
1	5	7	8	2	4	6	9

(p) (q) (q+1) (r)

L1 L2

Above array contains two sorted list as L1 and L2.

Copy the L1 and L2 in array L and R

L

1	5	7	8	∞

(i)

R

2	4	6	9	∞

(j)

size of newly formed array must be (+1) to add (∞) element.

Then give a pointer (i) to access array L and (j) to array R and a pointer (k) to access array 'A'.

1	2	4	5	6	7	8	9

→ Sorted Array.

k↑ ..... k↑

After Sorting (k)

(Bcz of sorting)

Algorithm (Merging Procedure)

```
Mege (A, p, q, r)
{
  n1 = q - p + 1
  n2 = r - q
  Let L[1 .... n1+1] and R[1 to n2+1] be new arrays.
  for (i = 1 to 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++
    else A[k] = R[j]
           j++
    k++ }
```

Time Complexity:

① To copy n elements from A to different arrays L and R takes $O(n)$.

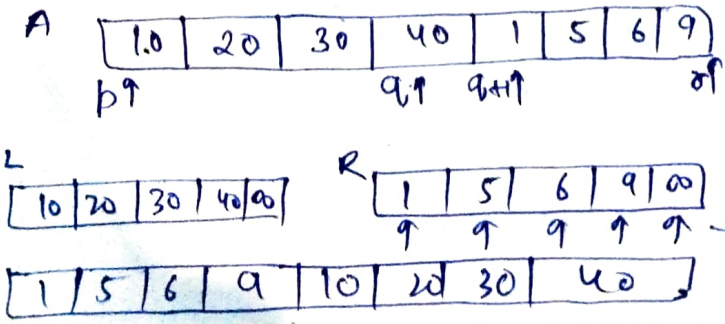
② Then from L and R to again A takes 'n' time for comparison and 'n' time for copy so total time = $n+n = 2n = O(n)$.

Overall → $O(n)$.

Space Complexity = $O(n)$ also because extra space require as array L and R equals to

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

Why add ∞ at the end?



∞ → It is ensure that the list is end and from here another list is to copy automatically at end

Merge- Sort Algorithm (Devide and conquer)

↳ Devide the problem in sub problem and solve them individually and then create a complete soln.

merge-sort (A, p, r) → T(n)

{ if p<r

q = ⌊(p+r)/2⌋ } → c

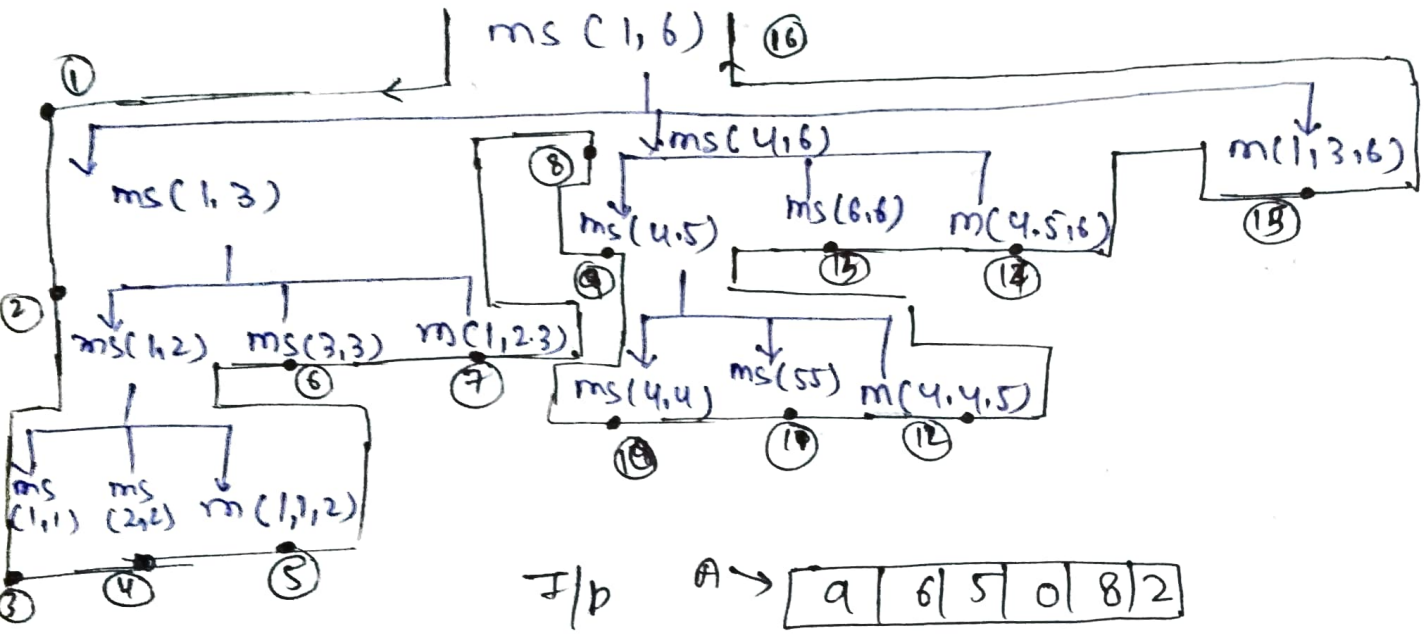
merge-sort (A, p, q) → T(n/2)

merge-sort (A, q+1, r) → T(n/2)

merge (A, p, q, r) → O(n)

}

Consider an array of 6 elements.



I/p A → | 9 | 6 | 5 | 0 | 8 | 2 |

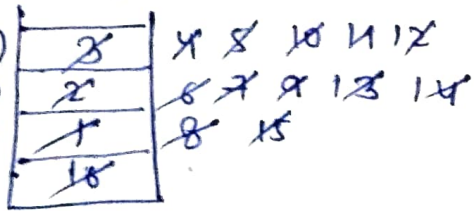
O/p A → | 0 | 2 | 5 | 6 | 8 | 9 |

Extra space required for merge = O(n).

Extra space for merge-sort

No. of cells in a stack required = 4 (here)

↳ (no. of levels)



for n elements no. of levels = (⌈log n⌉ + 1)

space for merge-sort = O(log n).

Total space complexity = $O(n + \log n)$

But dominating factor is 'n' in $(n + \log n)$

So, $S(n) = O(n)$ → finally,

Also called out of place Algorithm.

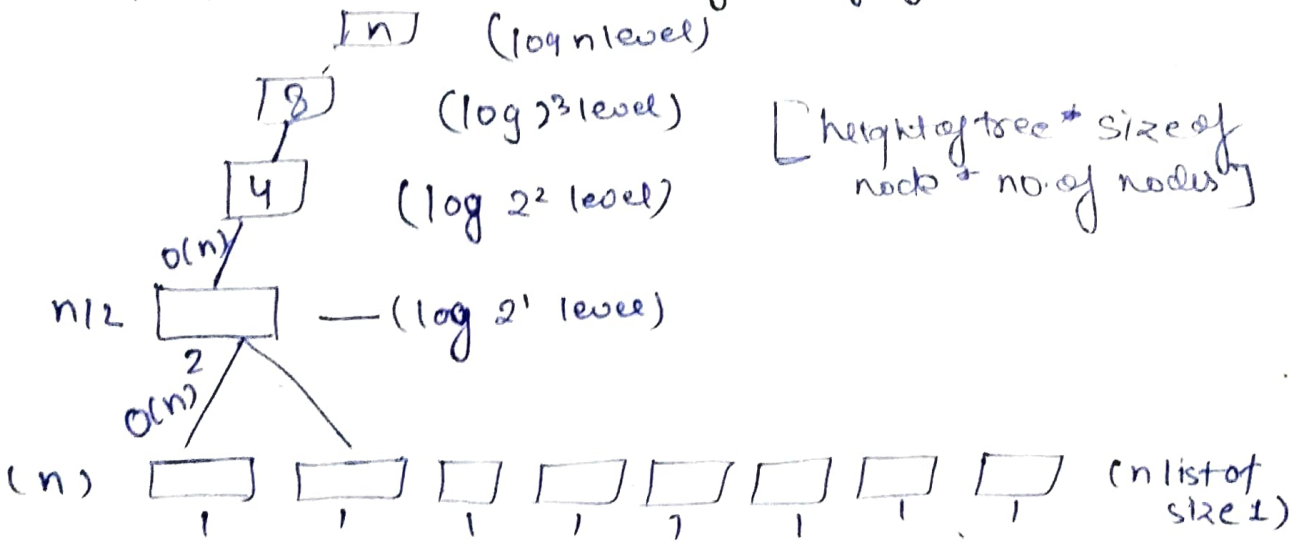
Time Complexity

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

Using Masters theorem in worst or best case

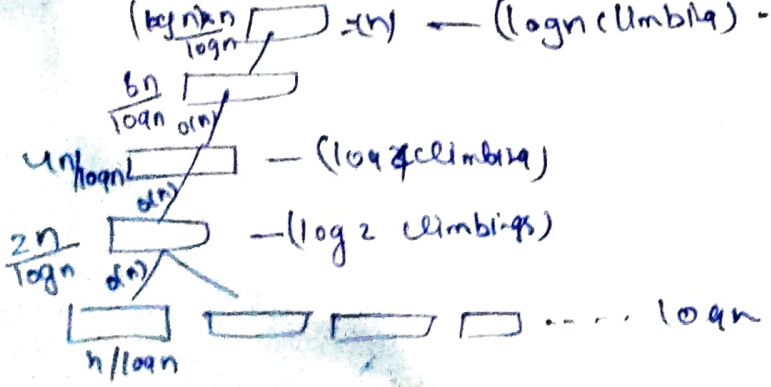
$$T(n) = O(n \log n)$$

Given n elements merge them into one sorted list using merge procedure (Two ways merging)



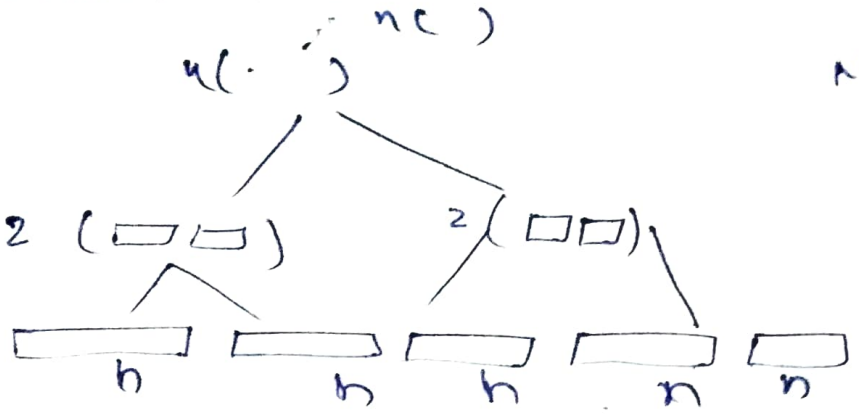
Total time taken = $O(n \log n)$

Given "log n" sorted list each of size "n/log n". What is the time required to merge them into into 1.



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

" n " strings each of length " n " are given then what is the time taken to sort the list.



No. of comparison = $n/2$

Every comparison taken = $O(n)$ time

So total time taken = $O(n^2)$.

Total time taken for copying = $O(n^2)$

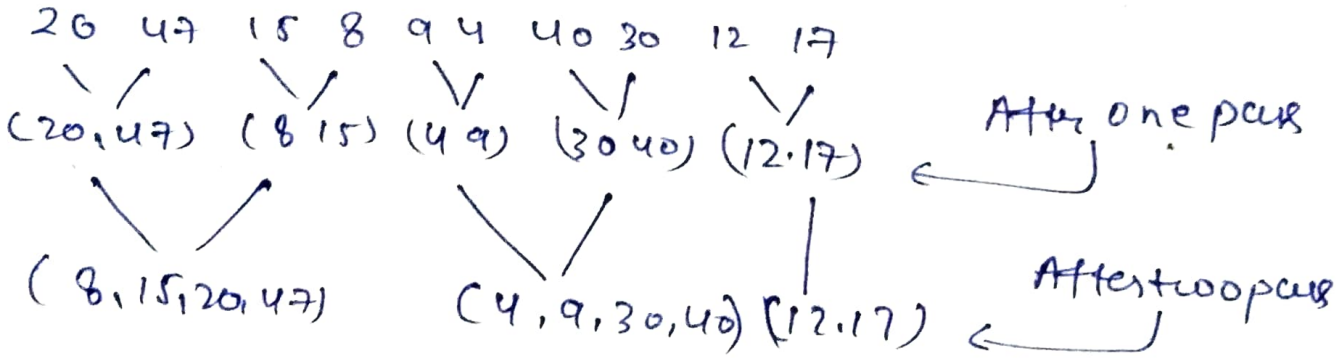
Net $T(n) = O(n^2) + O(n^2) = O(n^2)$

for n levels = $O(n^2) \times O(\lg n) = O(n^2 \log n)$

1) Merge sort uses Devide and conquer.

2) Two sorted list of size 'm' and 'n'. Total time taken = $O(m+n)$.

3)



4) Gauss Algo. Recursive eqn. for integer multiplication

$T(n) \leq 3T(n/2) + O(n)$

In choosing middle element as pivot then, quick sort to

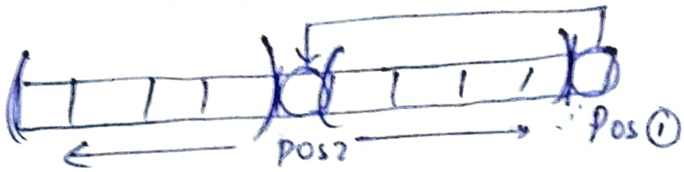
$T(n) = O(n \log n)$

Quick-Sort Algorithm → Less complexity for small no. of inputs. (Devide & Conquer)

Partition

```
Partition (A, p, r)
{
   x = A[r]        → pivot element
   i = p-1
   for (j = p to r-1)
   {
      if (A[j] <= x) {
         i = i+1
         exchange A[i] with A[j]
      }
   }
   exchange A[i+1] with A[r] }
   return i+1;
}
```

Partition is the heart of Quick sort work as.



Take an array and select any element (here we select last) and then insert the element at the pos 2 suchat on the right side all numbers greaterthan the number and on the left side small then the given numbers. Again apply the same procedure on RHS and LHS both.

Working of Partition Algorithm

Take I/P → Pivot element

9	6	5	0	8	2	4	7
1	2	3	4	5	6	7	8

i = 0, p = 1, r = 8, x = 7

Take 7 first

Two pointers i and j work as j is traverse the array and will find the number less than 7 swap it with i.

as

6 9 5 0 8 2 4 7 → 6 5 9 0 8 2 4 7

6 5 0 2 8 9 4 7 ← 6 5 0 9 8 2 4 7

6 5 0 2 4 ⑧ 9 ⑦ → 6 5 0 2 4 ⑦ 8 9

replace by i+1.

lesser ← | → greater

Sorted element.

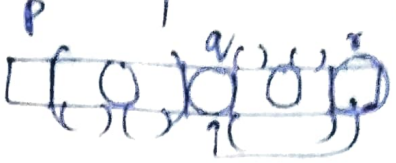
After 1 partition iteration one element get sorted.

We are varing the position of j from 1 to n-1 position So, time complexity of partioning Algorithm is **O(n)**.

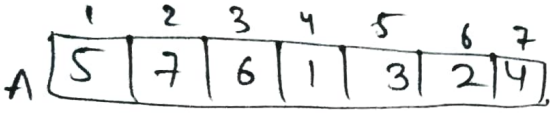
Quick Sort →

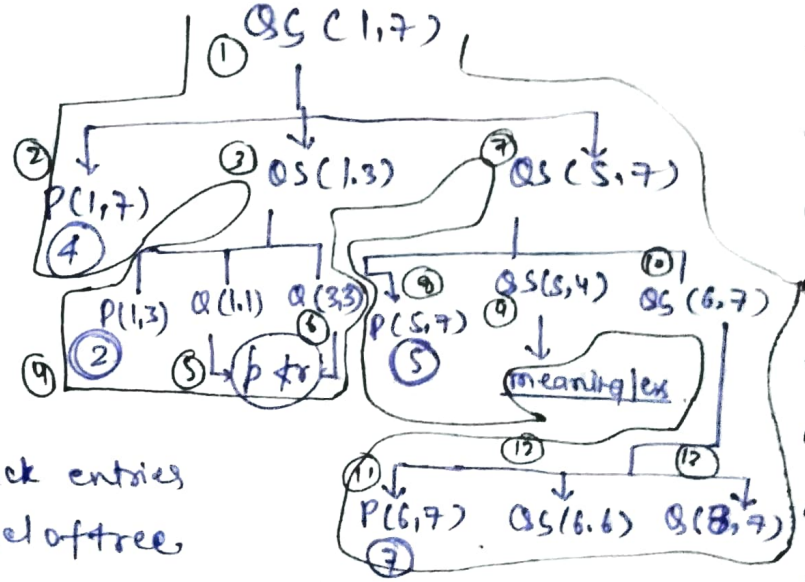
```
Quicksort (A, p, r) — T(n)
{
  if (p<r)
  {
    q = partition (A, p, r) → O(n)
    Quicksort (A, p, q-1) → T(n/2)   [Best case]   (O(logn) struck through)
    Quicksort (A, q+1, r) → T(n/2)
  }
}
```

→ Devide and Conquer.



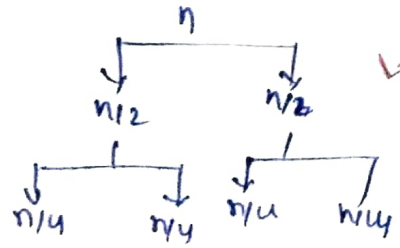
Ex





Space complexity = no. of stack entries
= no. of level of tree

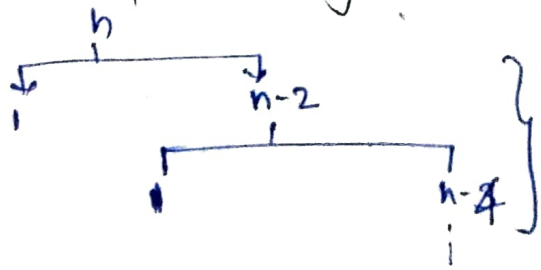
① for balanced partitioning. (Best case)



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

for partitioning not for quick sort.

② for unbalanced partitioning. (worst case



Height might go to O(n).

So, $S(n) = O(n)$

Time complexity:- $T(n) = 2T(n/2) + O(n)$

$= \boxed{O(n \log n)}$ (Best case)

↳ $\Omega(n \log n)$ also

Worst case :- $T(n) = T(0) + T(n-1) + O(n)$

$= T(n-1) + Cn$

↳ for sorting not for partition Algorithm

By Back substitution we get,

$T(n-2) + C(n-1) + Cn$

$T(n-3) C(n-2) + C(n-1) + Cn$

$= C1 + C2 + \dots + Cn$

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

In best case after a element sorted problem gets devided into two parts.

Analysing quick sort and Problem on it

1) Given input in ascending order. (1 2 3 4 5 6)

$T(n) = O(n)$ (for partitioning) $+ T(n-1)$

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

On solving we get

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

2) If input is in descending order (6 5 4 3 2 1)

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

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

3) for same elements (2 2 2 2 2)

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

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

Major time is taken by the partitioning algorithm.

I/p	Time	Sorted I/p
n	— Cn	— 1
n-1	— C(n-1)	— 2
⋮		
1	— C	— n

$n * O(n) \rightarrow O(n^2)$

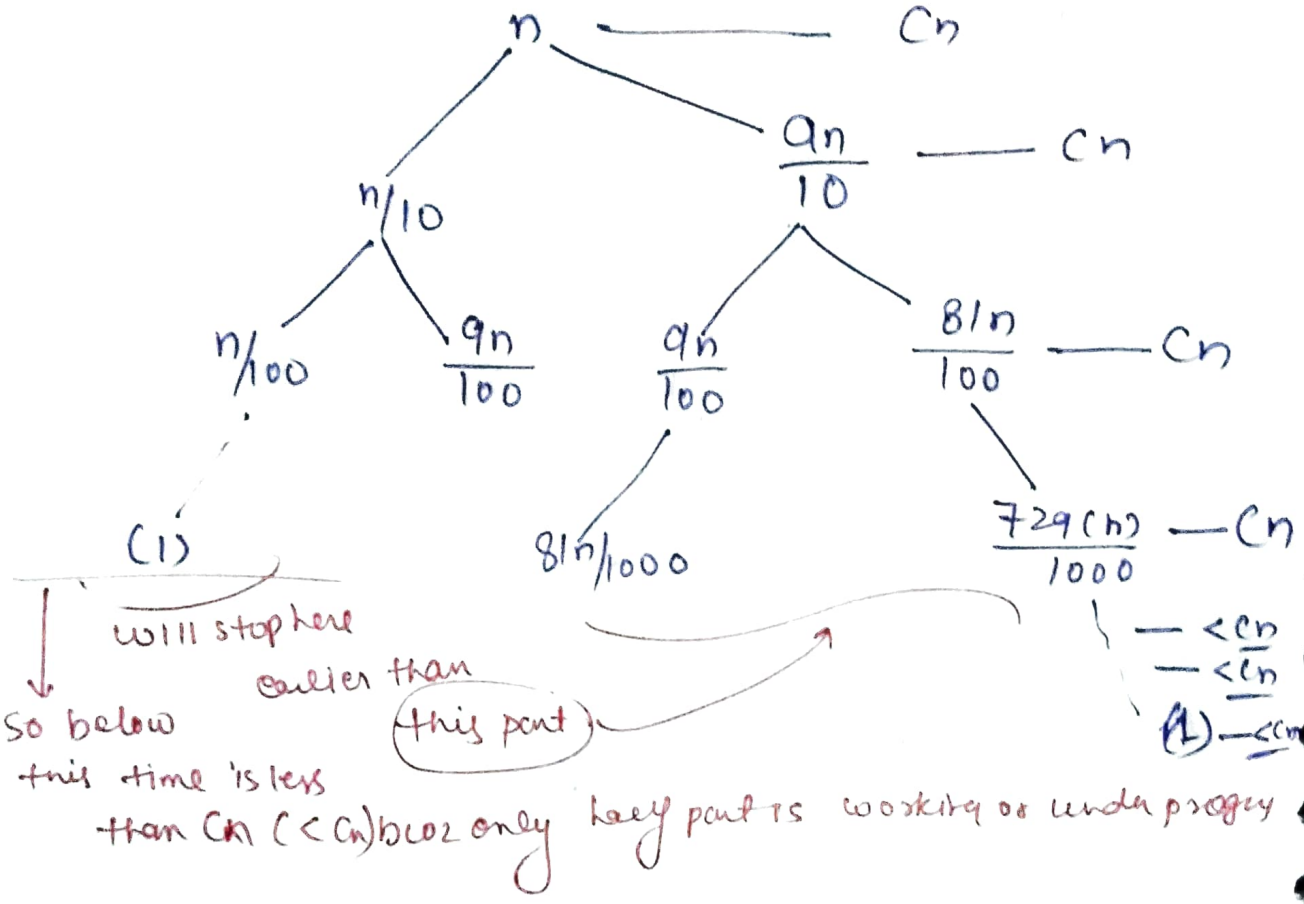
Partitioning Algo. take $O(n)$ time for sorting a single value and to sort n value will take $n * O(n) = O(n^2)$

Every time it will sort a single value.

In best case $T(n) = O(n) + 2T(n/2)$

$= \Omega(n \log n)$.

(4) Deviding in a ratio



tree is shrinking in way like like

$$n - \frac{n}{(10/9)} - \frac{n}{(10/9)^2} - \frac{n}{(10/9)^3} - \cdots 1 = \log_{10/9} n \approx \log_2 n$$

or $= O(\log_2 n)$

total no. of levels $= O(\log_2 n)$

At every level work done is $= Cn$

then total work done is $= Cn * O(\log_2 n)$

$= O(n \log_2 n)$ → Best Case

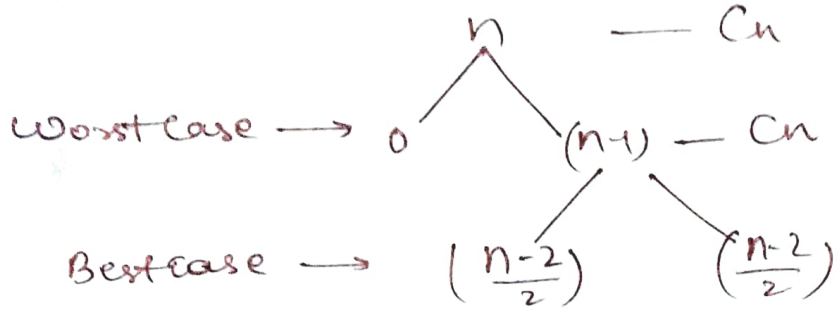
If split is as

1:9

1:99

1:999 or anything in this pattern we get

To sort n element using Quicksot in



Alternate Worst and Best

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

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

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

$\leq O(n) + 2T(n/2)$

$= O(n \log n)$ → By masters theorem.

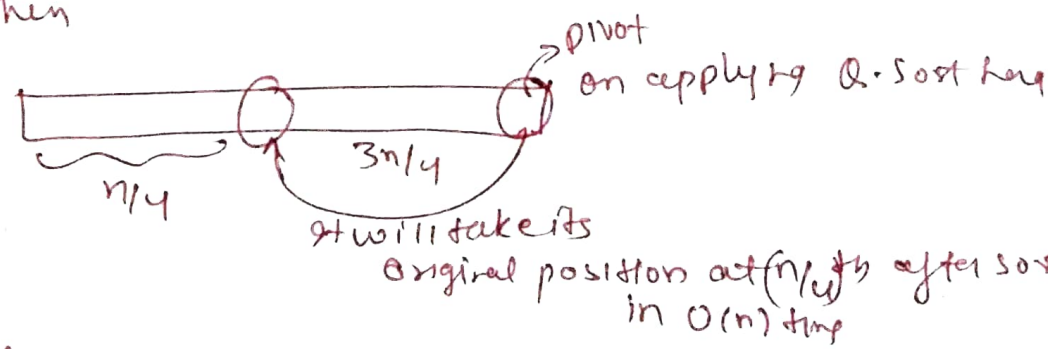
Q → In QSort, for sorting n elements, the $(n/4)^{th}$ smallest element selected as pivot using O(n) time algorithm. What is the worst case time complexity of Q. Sort.

Given → first the array is sorted and then select $n/4^{th}$ elements pivot using O(n) algorithm

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

↓ for sorting ; ↓ for swap $(n/4)^{th}$ with last to make pivot

then



So, $T(n) = O(n) + O(1) + O(n) + T(n/4) + T(3n/4)$

$= O(n) + T(n/4) + T(3n/4)$

ratio 1 : 3

So Ans will we $\boxed{O(n \log n)}$

Q Apply QS on input (1, 2, 3, 4, 5 ... n) T_1 time

① or ——— (n, (n-1) ... 1) — T_2

$T_1 = O(n^2) = T_2$

so, $\boxed{T_1 = T_2}$

② If problem is partioned in $\frac{n}{5}$ and $\frac{4n}{5}$ then find T(n)

$$T(n) = O(n) + T(n/5) + T(4n/5)$$

$$\leq O(n) + 2T\left(\frac{4n}{5}\right)$$

or $O(n \log n)$ Ans)

Introduction to Heaps

	insert	search	find min	Delete min
Unsorted Array	O(1)	O(n) → Linear search	O(n)	O(n)
Sorted array	O(n)	O(log n) → Binary Search	O(1)	O(n) ← depends on ascending / descending order.

In unsorted array for delete min O(n) for select the min then O(1) to delete and then O(n) to fill the gap create due to deleted element

So, $O(n) + O(n) + O(1) = O(n)$.

for insert into sorted array first find the position at which element needs to be inserted,

So, O(log n) to search position using binary search then to make a place vacant for element will take O(n)

So, $O(n) + O(\log n) = O(n)$.

To ~~find min~~ search in sorted array using Binary search

$$T(n) = 1 + T(n/2)$$
$$= O(\log n)$$

find min in sorted array in ascending order $O(1)$
to delete min take $O(n)$

In descending find min = $O(n)$
delete min = $O(1)$

If Input is given in Linked list :-

① Unsorted Linked List.

① Insert → $O(1)$ at beg.

② Search → $O(n)$ → Linear search

③ find Min → $O(n)$

④ Delete min → $O(n+1) = O(n)$

In array and linked the operation like Insert find / del min → इनमें से कम्से कम एक तो $O(n)$ तो रहा है।

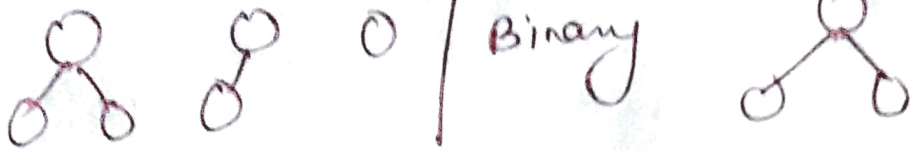
So, we need a data structure that can optimize the soln.

Heap → If any algo need only these operation then using heap execution will be faster.

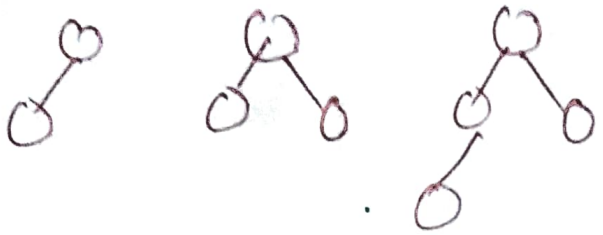
min Heap	Insert	$O(\log n)$
	find min	$O(1)$
	Del min	$O(\log n)$

Heap

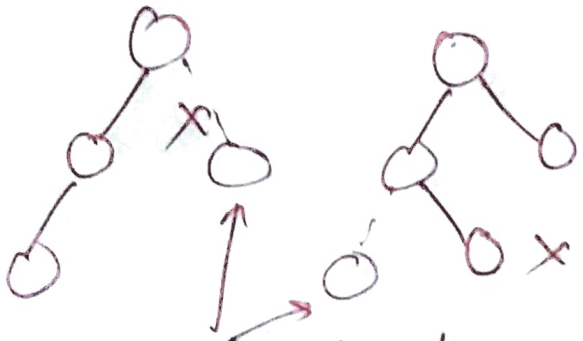
Heap is a binary, 3-ary or ... n-ary tree



Heap is almost complete binary tree.

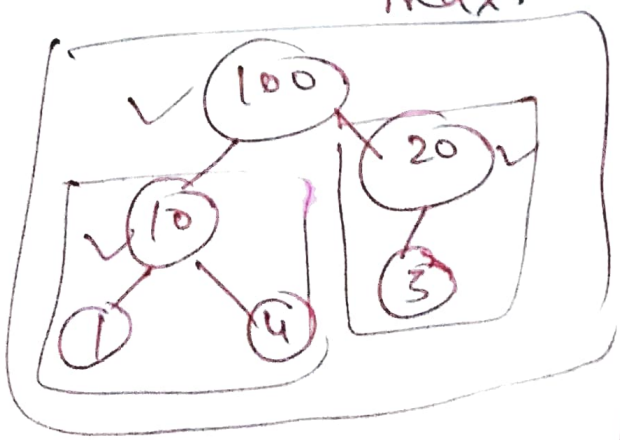


Leaf should be at end or one level less than end on filling L → R

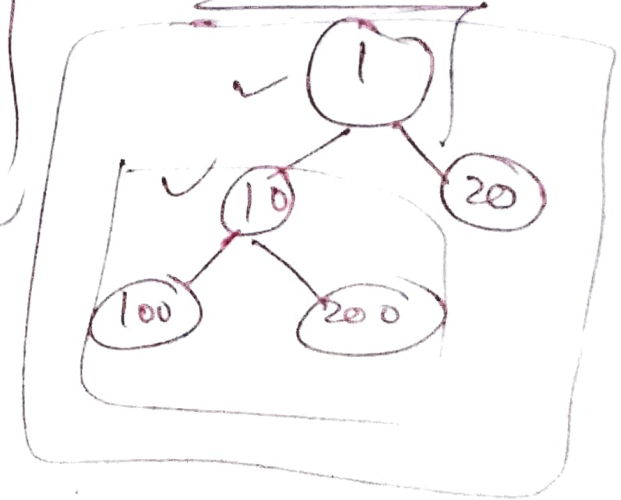


Previous level should be completely filled.

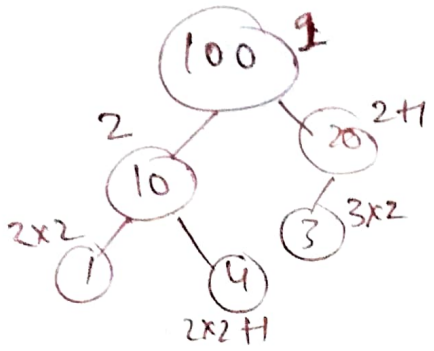
max. heap :- A binary tree in which root is max.



min heap



max. heap and min. heap property.



1	2	3	4	5	6
100	10	20	1	4	3

Array Implementation of Heap.

L child = 2 x index of parent

R Child = (2 * index of parent + 1)

To get index of parent = $\left\lfloor \frac{\text{index of child}}{2} \right\rfloor$

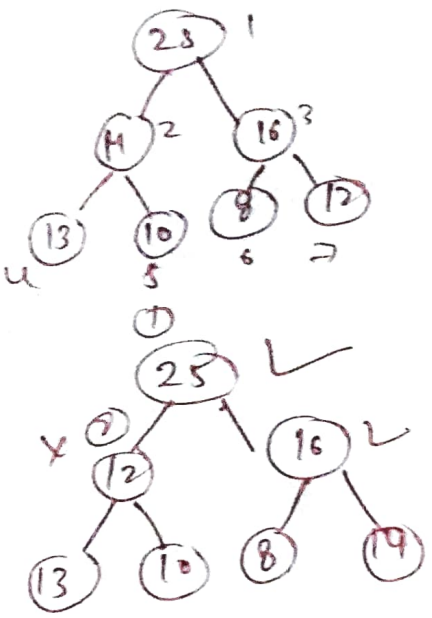
floor function.

(2 * something → Left shift by 1)

(something / 2 → right shift by 1)

Implementation is easy.

Array	A.length	A.heap size (no. of node follow heap property)
25, 12, 16, 13, 10, 8, 14	7	1
25, 14, 16, 13, 10, 8, 12	7	7
25, 14, 13, 16, 10, 8, 12	7	1
25, 14, 12, 13, 10, 8, 16	7	2
14, 13, 12, 10, 8	5	5
14, 12, 13, 8, 10	5	5
14, 13, 8, 12, 10	5	5
14, 13, 12, 8, 10	5	5
89, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70	13	2



Every node is heap and node that have no child consider as heap

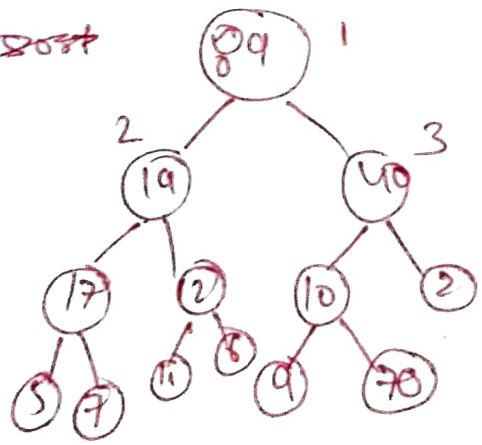
So total $3+4=7$

Here total only 1 node is heap either 16 is also a heap but since we consider L→ child first and it is not a heap so we can't go further.

If array is sorted in descending order output tree is ⟶ max heap

If array is sorted in ascending order output tree is ⟶ min heap.

~~To sort~~



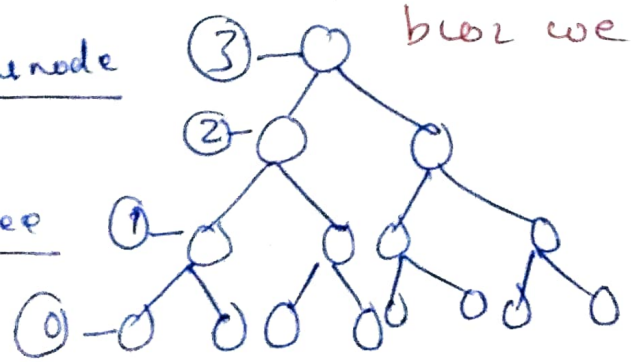
Here 1 and 2 is follow the heap property but 40 is not follow bcoz 70 is true

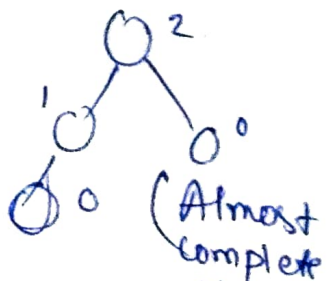
So, total no of heap nodes = 2

bcoz we will not go further

Height of the node

Complete binary tree





Almost complete Binary tree

Height of the tree = Height of the root.

Max no. of nodes in a complete binary tree of height h is (=) $\boxed{N = 2^{h+1} - 1}$

No. of nodes in 3-ary tree or n-ary tree / Todo

→ Max. no. of nodes in n-ary tree of height h = $\left(\frac{n^{h+1} - 1}{n-1}\right)$

min. height of Complete (or almost complete) binary tree with n nodes:

$$\min(h) = \lfloor \log n \rfloor$$

#

Height of any binary heap = $O(\log n)$

Build a heap from Array (max heap);

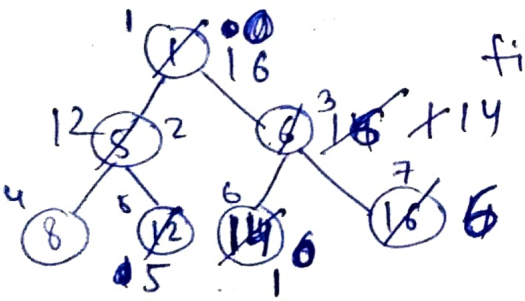
In any comparison based algorithm the best complexity we could get is → $O(n \log n)$.

To design a max heap :- you can sort the array in descending order makes ease.

But we interested in $< (n \log n)$

In a heap which complete or almost complete binary tree then, leaf will start from $(\lfloor n/2 \rfloor + 1$ to $n)$ and no n leafs 1 to $\lfloor n/2 \rfloor$

Ex 1, 5, 6, 8, 12, 14, 16 (array)



find the longest index with non leaf.

Here index = 3

Start from 3 and comes to 1, verify that node is max heap or not

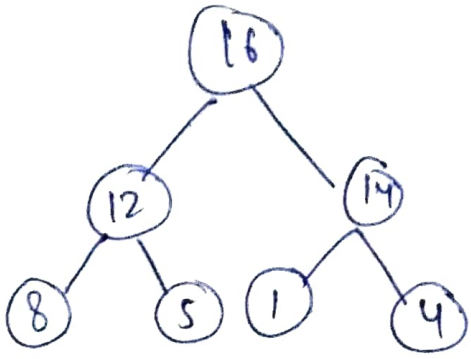
Here ③ is not replace 3 by 7 (16).

then come to ② and replace 5 by 12

then come to ① and replace 1 by 16

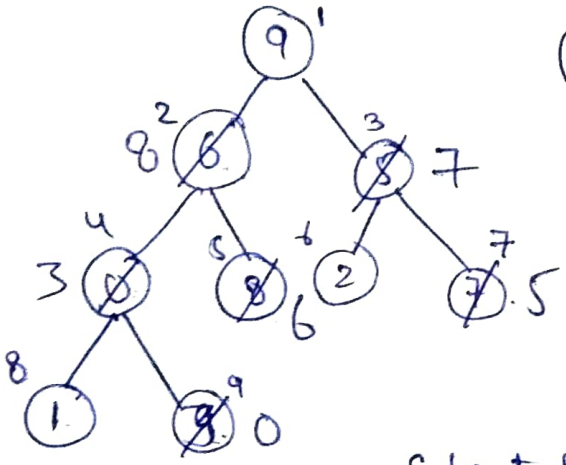
then again verify from 3 and replace 1 by 14

then final tree obtained is-1



Ex-2

9, 6, 5, 0, 8, 2, 7, 1, 3



① Leaf will start from

$[\lfloor n/2 \rfloor + 1 \text{ to } n]$

$= 5 \rightarrow 9$

longest index which is non-leaf

$= 4$

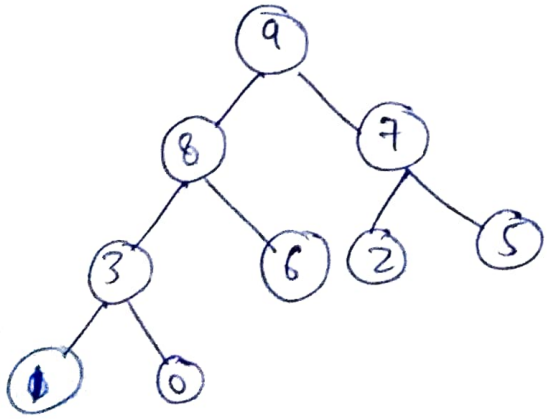
Start from 4.

① replace $0 \longleftrightarrow 3$

② replace $5 \longleftrightarrow 7$

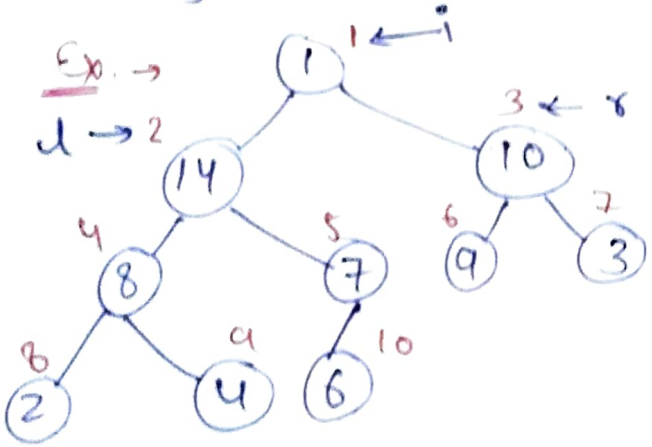
③ replace $8 \longleftrightarrow 6$

and index is already a max heap. So final tree



Contd..

```
MAX-HEAPIFY (A, i) {
    l = 2i;
    r = 2i+1;
    if (l ≤ A.heap.size and A[l] > A[i])
        largest = l;
    else largest = i;
    if (r ≤ A.heapsize and A[r] > A[largest])
        largest = r;
    if (largest ≠ i)
    exchange A[i] with A[largest]
        MAX.- HEAPIFY (A, largest)
}
```



let i = 1 (say)
l = 2
r = 3
largest = 2,
and largest ≠ i
so replace 1 by 14
and call Algo again on
(A, 2).

Above algorithm will work if left sub Tree and right subtree is already a heap but root is not a heap.

Just to move 1 level down we have to do 2 comparison. So total no. of comparison required = $2 * \log n$ | as max

So, Time complexity = $\boxed{O(\log n)}$ → Best Case
= $O(\log n)$ → worst Case

Space Complexity

Total no recursive calls = no of levels

In worst case height or level max = $\log n$

So, space complexity = $[O(\log n)]$

Also extra space is used by space is used for recursion in function call.

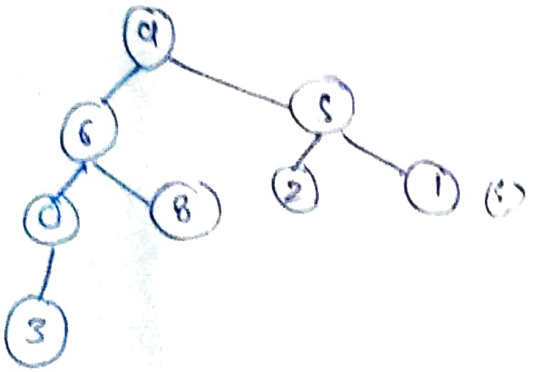
So, $[S(n) = O(\log n)]$ Worst case //

Build Max Heap Algorithm

```
Build_Max_Heap(A)
{
A.heapsize = A.length
for(i = ⌊A.length/2⌋ down to 1)
    MAX_HEPIFY(A,i)
}
```

Call Max heapify on all the internal nodes

Ex 9, 6, 5, 0, 8, 2, 1, 3



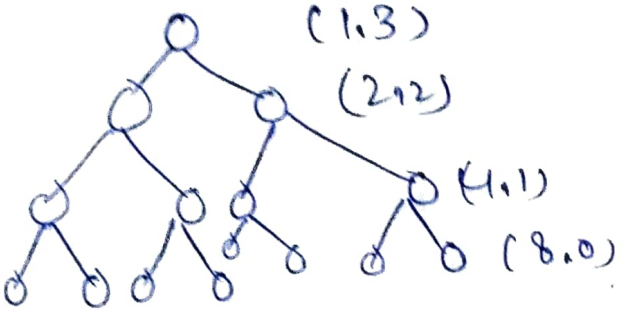
leaves start from $\lfloor 8/2 \rfloor + 1$

= 5 (nodes)

non leaves 1 to $\lfloor n/2 \rfloor$

If a tree of height (h) and no of nodes (n) then

max. nodes present at height h = $\left\lceil \frac{n}{2^{h+1}} \right\rceil$



① at height $h = 0$

$$= \left\lceil \frac{15}{2^{0+1}} \right\rceil = 8$$

② at height $h = 1$

$$\left\lceil \frac{15}{4} \right\rceil = 4$$

Time taken by max-Heapify is equal to = $O(h)$.
↳ height

Total time taken by max_heap procedure

$$= \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

$$= \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^h \cdot 2} \right\rceil (c \cdot h)$$

$$= \frac{cn}{2} \sum_{h=0}^{\log n} \left(\frac{h}{2^h} \right)$$

$\sum_{k=0}^{\infty} k x^k = \frac{x^2}{(1-x)}$

$$= O\left(\frac{cn}{2} \left(\sum_{h=0}^{\infty} \frac{h}{2^h} \right) \right)$$

→ Appendix A.8 Coreman value = 2

$$= O\left(\frac{cn}{2} * 2 \right) = O(n)$$

Space Complexity : Due to MAX_HEPIFY() only.

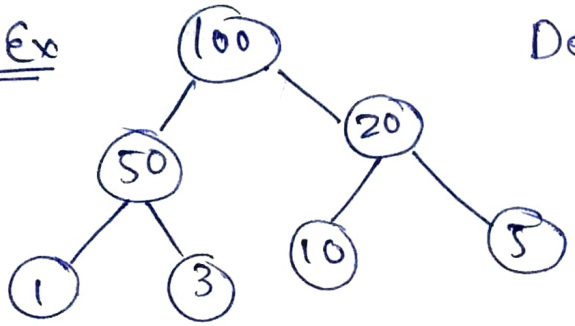
So, $S(n) = O(\log n)$

Q Implementation of priority Queue using heap. ⊢ To read

Delete or Extract the MAX Node from MAX. HEAP

```
Heap - Extract - Max (A)
{
    If A.heap_size < 1                          } T(n) = C
     error "heap underflow"
    max = A[1]
    A[1] = A[A.heap_size]
    A.heapsize = A.heap_size - 1
     MAX_HEAPIFY (A,1)                          } T(n) = O(logn)
    return max
}
```

Ex



Delete 100 and replace it by 5 then called MAX_HEPIFY To again make it a heap.

Always last element will replace the root node.

Time taken due to MAX_HEAPFY() only.

So, $T(n) = O(\log n)$] prof

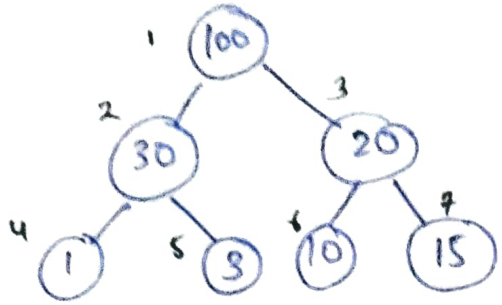
Space Complexity $= O(\log n)$

```
Heap_increase_key (A, i, key) {
    If (key < A[i])
      error
A[i] = key;
while (i > 1 and A[i/2] < A[i])
  exchange A[i] and A[i/2]
    i = i/2;
}
```

key → value to be substituted

i → index of change value

Ex → Max heap



Let us assume we want value at index 5 from 3 to 300.

then you have check the value of parent also

Replace 100 ⟷ 300
30 ⟷ 100
3 ⟷ 30

If key is greater change or exchange the value at the index and then compare with parent node.

T(n) → In worst case you may go from leaf to root So, it will take

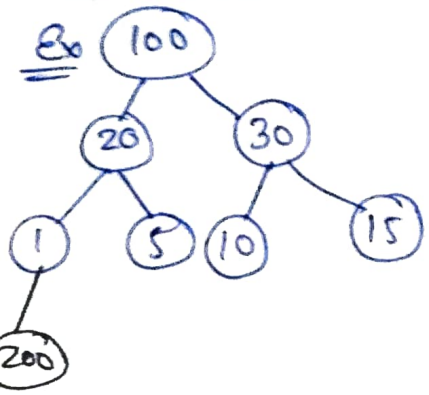
On max_heap ← O(logn) → (max. height of tree)

for decrease_key in worst case it may take

T(n) = O(logn) | On min-heap.

Insert an element into max_heap.

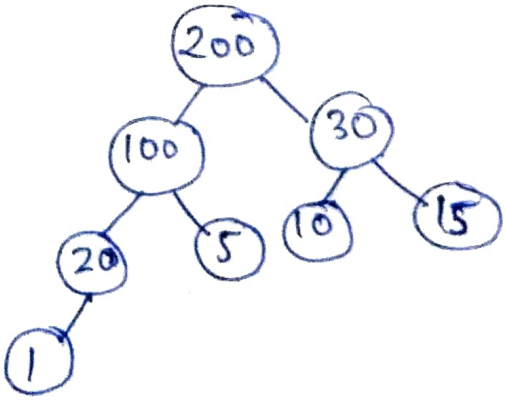
Ex



Let the inserted element is 200.

Insert at the end and if maximum then exchange it with parents.

So,



T(n) in worst case O(logn).

Linear Search

- It retrieves information stored in data structure
- It can also use in SQL select command.
- Sequential search
- works on both sorted and unsorted data.

Program

```
int linearSearch ( int *arr , int size , int target){
for (int i=0 ; i<size ; i++){
        iff (arr[i] == target)
            return i; }

    return -1; }
```

Time Complexity

Best Case → No. of iteration = 1 → O(1)

Worst Case = (n+1)

Avg case = (n+2)/2 — O(n)

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

$T(1) = 1$

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

```
Struct node *Linked List (struct node *head, int target)
if (head) {
  while (head) {
  if (head->data == target)
        return head;
     head = head->next;
} }
```