Design And Analysis of Algorithms

[BCSE 2350]

Ankit Verma
Batch

MODULE I

INTRODUCTION TO DAA

Algorithm

Algorithm is a step by step Method to solve a problem. It refers to a set of rules/instructions that step by step define how a work is to be Executed upon inorder to get the Expected results.

Characteristics of Algorithm

* Input :- Algorithm must contain '0' or more input
* Output :- Algorithm must Contain '1' or more output.
* Finiteness :- Algorithm must Complete after finite No. of steps
* Definiteness :- The step of the algorithm must be defined precisely or clearly (unambiguous)
* Effectiveness :- The step of an algorithm must be Effective (can be done successfully)

Analysis of Algorithm

Analysis of algorithm depend upon the time & Memory taken by the algorithm.

i.e
* Time Complexity
* Space Complexity.

Time Complexity

The amount of time required by the algorithm is called time complexity.

Space Complexity

The amount of space/Memory required by the algorithm is called as space complexity.

Application of Algorithm

- Bioinformatics
- Internet
- Ecommerce
- Finding shortest root in road map
- Product of Matrices.

Growth functions

	2^n	$n!$
$n=1$	2	1
$n=2$	4	2
$n=3$	8	6
$n=4$	16	24
$n=5$	32	120

$n! > 2^n \forall n \geq 4$

$n!$ has higher growth rate than 2^n

Arrange the following according to increasing growth rate order.

$n, 2n, n\log n, n!, n^3, n^5, n^2, 1$

$\Rightarrow$ constant $\leq$ logarithm $\leq$ polynomial $\leq$ Exponential $\leq$ factorial

Ans: $1, n, n\log n, n^2, n^3, n^5, 2^n, n!$

Asymptotic Notation

These Notations are used to describe the asymptotic running time (when the input size n is very large i.e $n \to \infty$) of an algorithm which defines in terms of functions whose domains are the set of Natural Numbers.

⇒ The time complexity of a function can be represented by asymptotic notations.

⇒ The different asymptotic Notations are :-

* Big Oh (O)
* Big omega (Ω)
* Big theta (Θ)
* little Oh (o)
* little omega (ω)

Big Oh Notation, O

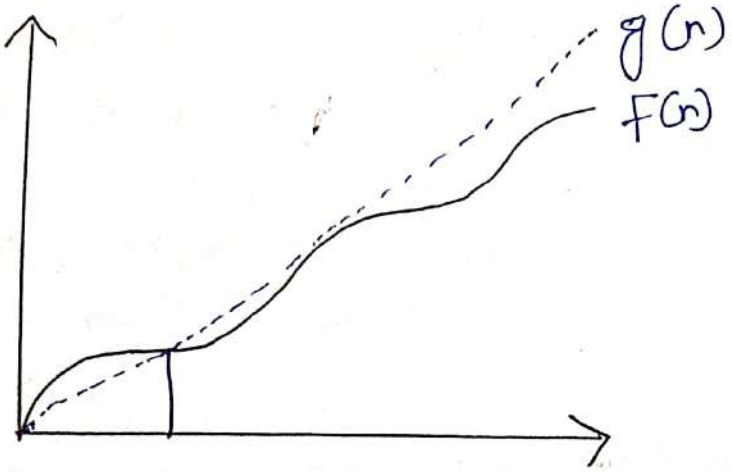
The Notation $O(n)$ is the formal way to Express the upper bound of an algorithm's running time. It measures the worst case time Complexity or the longest amount of time an algorithm can possibly take to Complete

A function $f(n)$ can be represented is the order of $g(n)$ that is $O(g(n))$, if there Exists a value of positive integer n as n_0 and a positive Constant C such that

$$f(n) \leq C \cdot g(n) \quad \text{for } n > n_0 \text{ in all cases}$$

$g(n) \rightarrow$ upper bound for $f(n)$

$$f(n) \in O(g(n))$$



$f(n) \leq c \cdot g(n)$

Example:- $f(n) = 4 \cdot n^3 + 10 \cdot n^2 + 5 \cdot n + 1$

Let $g(n) = n^3$ $\{c = 4\}$

$f(n) \leq 5 \cdot g(n)$ for all values of $n > 2$

Complexity of $f(n) \Rightarrow O(g(n))$

i.e $O(n^3)$

Big Omega (Ω)

The Notation $\Omega(n)$ is the Normal way to Express the lower bound of an algorithm's running time. It Measures the best case time Complexity or the best amount of time an algorithm can possibly take to Complete.

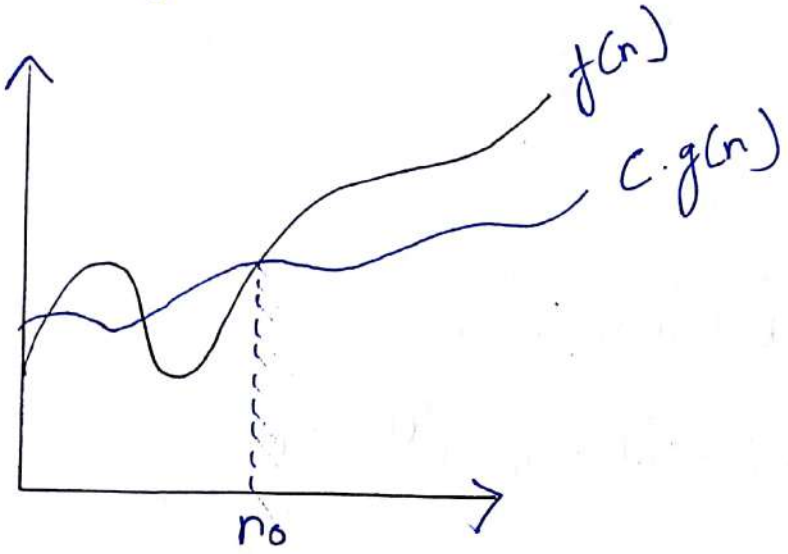
$f(n) = \Omega(g(n))$ when there Exists constant

i.e $\boxed{f(n) \geq c \cdot g(n)}$ for larger value of n.

where g is a lower bound for fun f

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

$f(n) \in \Omega(g(n))$



Example:- $f(n) = 4 \cdot n^3 + 10 \cdot n^2 + 5 \cdot n + 1$

$g(n) = n^3$

$f(n) \geq 4 \cdot g(n) \quad \forall \quad n > 0$

Complexity of $f(n) = \Omega(g(n))$

i.e $\Omega(n^3)$

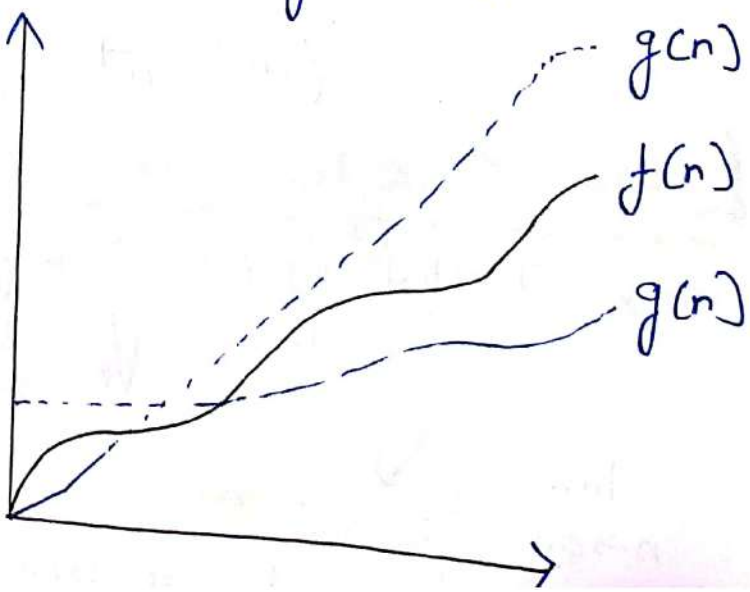
Big Theta (Θ)

The Notation $\Theta(n)$ is the Normal way to Express both the lower bound and the upper bound of an algorithm's running time.

$f(n) = \Theta(g(n))$

when there Exist constant c_1 & c_2 such that

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$



Example:- $f(n) = 4.n^3 + 10.n^2 + 5n + 1$

$g(n) = n^3$, $4g(n) \leq f(n) \leq 5.g(n)$

Complexity of $f(n) = O(g(n))$

i.e $O(n^3)$.

Little Oh Notation (o)

O notation is used to denote an upper bound

$O(g(n)) = \{ f(n)$: for any +ve const. $c > 0$

if a const $n_0 > 0$

such that $0 \leq f(n) < c.g(n)$

$\forall n \geq n_0$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Example:- $f(n) = n^4 + 35n^2 + 84$

$g(n) = n^4$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n^4 + 35n^2 + 84}{n^4}$$

$$\lim_{n \to \infty} \frac{n^4}{n^4} + \lim_{n \to \infty} \frac{35n^2}{n^4} + \lim_{n \to \infty} \frac{84}{n^4}$$

$1 + 0 + 0 = 1$

$f(n) \neq n^4$

Little Omega (ω)

It is denoted by ω. The fun^n $f(n) = \omega(g(n))$

iff $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

It defines the lower bound of Algo. which is not asympotatically tight

Divide & Conquer Approach

In divide and conquer approach, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

Generally divide & conquer algorithms have 3 parts :-

(*) Divide the problem :- Divide the problem into a number of sub problems that are smaller instances of the same problem.

(*) Conquer the sub-problems :- Conquer the sub problems by solving them recursively. If they are small enough solve the sub-problems as base cases.

(*) Combine the solutions :- Combine the sub-problems into the solution for the original problem.

Application of Divide and Conquer Approach

Following are some problems, which are solved using divide & conquer approach :—

(*) Finding the Maximum & minimum of a sequence of numbers.

(*) Strassen's Matrix Multiplication

(*) Merge Sort.

(*) Binary search.

MERGE SORT

→ Merge Means Combine

→ Sorting means arranging Elements in increasing or decreasing order.

→ Merge sort has ~~two~~ the operations :-

(i) Divide (ii) Conquer

(iii) Merge

Divide :- Divide the n-elements sequence to be sorted into two subsequences of n/2 Elements Each.

Conquer :- Sort the subsequences recursively using Merge Sort.

Combine :- Merge the two sorted subsequences to produce the sorted answer.

Algorithm

A [0 ..... n-1], low, mid, high]

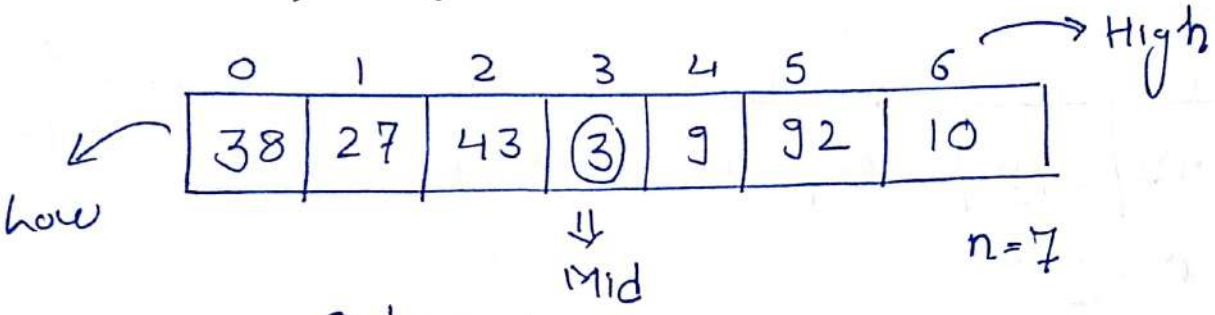
// **input :-** Array A of unsorted Elements, low as begining pointer Array A and high as End of pointer Array A.

// **output :-** Sorted Array A[0 .... n-1]

```
If (low < high) then
    mid ← (low + high)/2
Merge sort (A, low, mid)        // left sublist
Merge sort (A, mid+1, high)     // right sublist
Sorted Merge sort (A, mid, low, high)
```

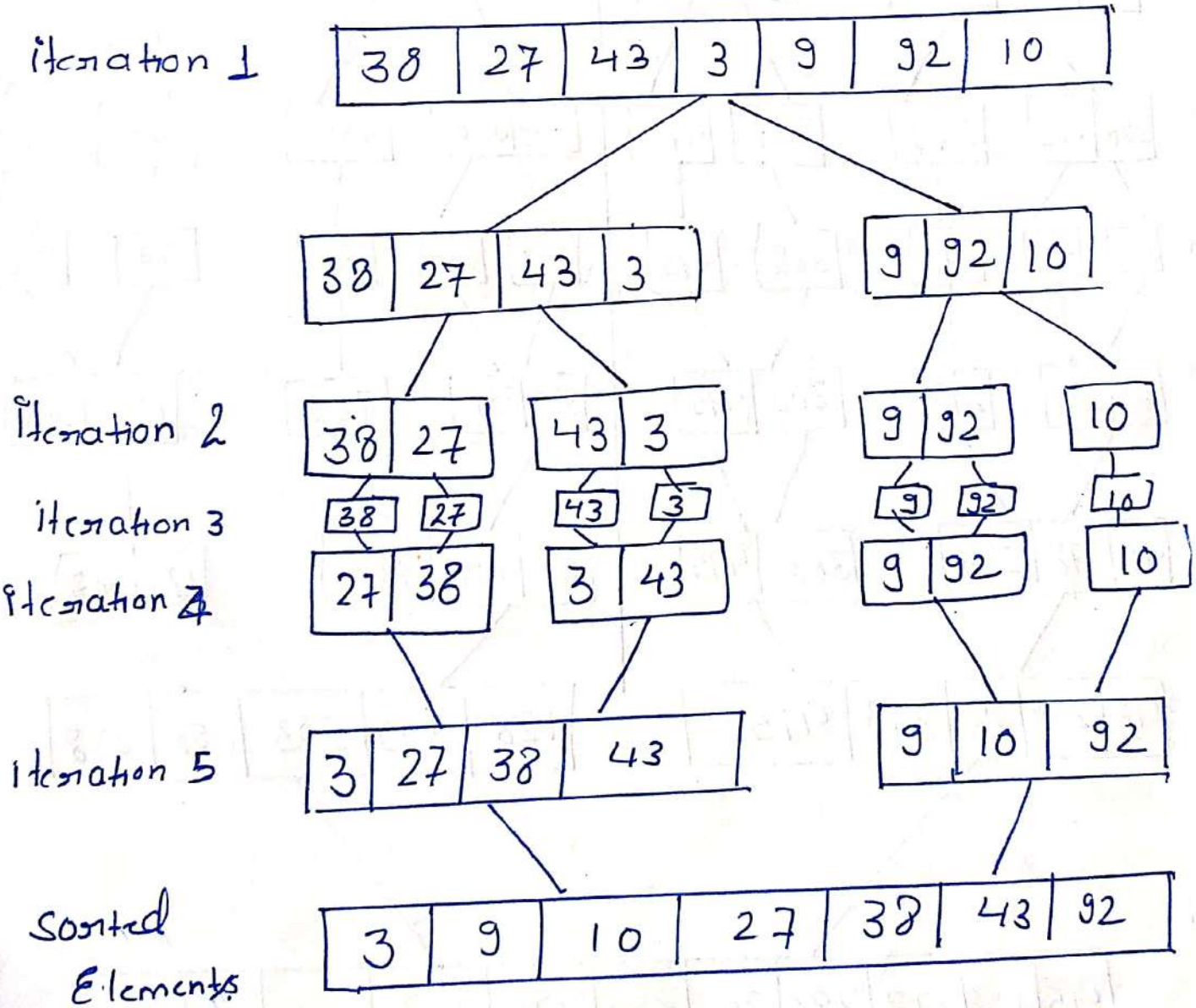
Sort the following using Merge Sort :-

38, 27, 43, 3, 9, 92, 10



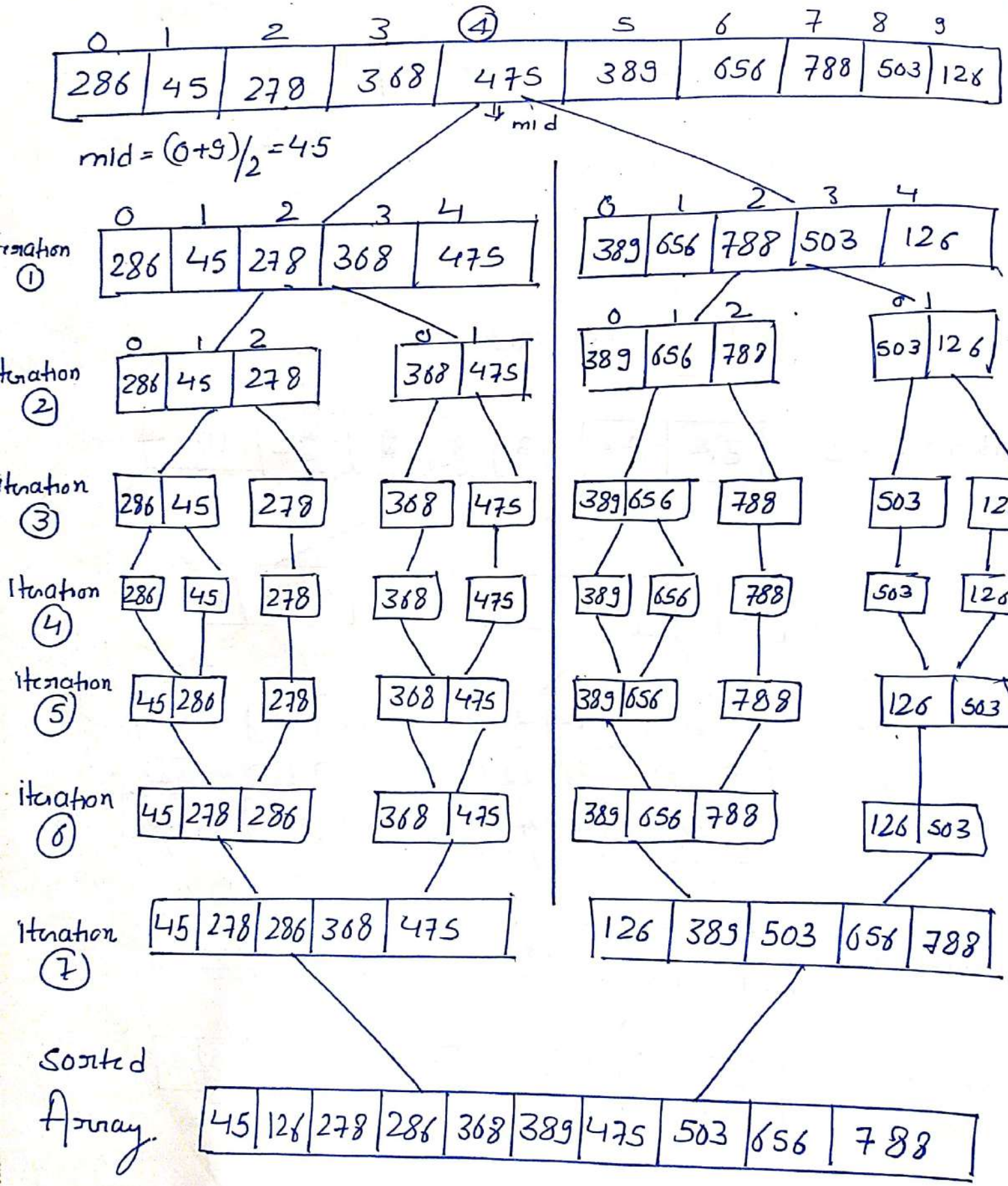
0 to n-1
0 to 7-1
0 to 6

Now $(0+6)/2 = 3$ Mid



Sort the following using Merge Sort.

286, 45, 278, 368, 475, 389, 656, 788, 503, 126



Time Complexity

For Each algorithm we can Calculate the time Complexity by using three Methods —

(i) Master Method
(ii) Tree Method
(iii) Substitution Method.

Substitution Method

We make a guess for the soln & then we use Mathematical induction to prove the guess is Correct or incorrect

Tree Method

In this Method we draw a recurrence tree & Calculate the time taken by Every level of tree. Finally we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmatic or geometric series.

Master Method

Master Method is a direct way to get the solution The Master method works only for the following type of recurrences or for the recurrences that can be transformed to the following type.

$$T(n) = aT(n/b) + f(n) \quad \text{where } a \geq 1, \; b > 1$$

Time Complexity for Merge Sort.

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

- Recurrence Relation
- $T(n/2)$: Time taken for left sublist
- $T(n/2)$: Time taken for Right sublist
- Cn: Merge left & Right sublist

$$T(n) = 2T(n/2) + Cn \quad \text{—— (1)}$$

$$T(n) = 2T(n/2) + C \cdot n$$

Substitute $n = 2^k$

$$T(2^k) = 2T(2^k/2) + C \cdot 2^k \quad \text{—— (2)}$$

$$= 2T(2^{k-1}) + C \cdot 2^k$$

If we put $k = k-1$

$$2[2T(2^{k-2}) + C \cdot 2^{k-1}] + C \cdot 2^k$$

$$4T(2^{k-2}) + 2 \cdot C \cdot 2^{k-1} + C \cdot 2^k$$

$$4T(2^{k-2}) + 2 \cdot C \cdot 2^k/2 + C \cdot 2^k$$

$$4T(2^{k-2}) + C \cdot 2^k + C \cdot 2^k$$

$$4T(2^{k-2}) + 2 \cdot C \cdot 2^k$$

$$2^2 T(2^{k-2}) + 2 \cdot C \cdot 2^k$$

$$2^2 T(2^{k-2}) + 2 \cdot C \cdot 2^k$$

$$2^3 T(2^{k-3}) + 3 \cdot C \cdot 2^k$$

$$2^4 T(2^{k-4}) + 4 \cdot C \cdot 2^k$$

$$2^k \cdot T(2^{k-k}) + k \cdot C \cdot 2^k$$

$$2^k T(2^0) + k C \cdot 2^k$$

$2^k T(1) + k \cdot C \cdot 2^k$

$2^k (0 + k \cdot C \cdot 2^k)$

$T(2^k) = k \cdot C \cdot 2^k \longrightarrow (3)$

$n = 2^k$ logarithm on both sides :-

$$\boxed{\log_2 n = k}$$

Substitute $k = \log_2 n$ in eq^n (3)

$\log_2 n \; C \cdot n$

Time Complexity for Merge Sort

$O(n \log_2 n)$

QUICK SORT

Quick Sort is also based on the Concept of Divide and Conquer, just like Merge Sort. But in Quick Sort all the heavy lifting (heavy work) is done while dividing the array into subarrays, while in case of Merge Sort, all the real work happens during Merging the subarrays. In case of Quick sort, the Combine step does absolutely nothing.

Divide :- Split the array Elements into two sub-arrays based on pivot Element. Each Element in the left array should be less than or Equal than the pivot Element Each Element in the Right subarray should be greater than the pivot Elem

Conquer :- Sort the subarrays.

Combine :- Combine the left subarray and right subarray Elements into a single group.

We will use the following Rules in QUICK SORT

(i) If A[pivot] > A[I] increment i-position

(ii) If A[pivot] < A[J] decrement j-positior

(iii) If i and j cannot moved then swap A[i] & A[j]

(iv) Swap A[pivot] and A[j] if 'i' crosses 'j'

Example :- 44 33 11 55 77 90 40 60 99 22 88

let 44 be the pivot element & scanning done from right to left

Comparing 44 to the right side Elements. & if right side Elements are smaller than 44, then swap it. As 22 is smaller than 44 so swap them.

22	33	11	55	77	90	40	60	99	44	88

Now Comparing 44 from left side Element & the Element must be greater than 44 then swap them. As 55 are greater than 44 so swap them

22	33	11	44	77	90	40	60	99	55	88

Recursively repeating step ① & ② until we get two lists one left from pivot Element 44 & one right from pivot element.

22	33	11	40	77	90	(44)	60	99	55	88

Swap with 77

22	33	11	40	(44)	90	77	60	99	55	88

Now the Elements on the right side and left side are greater & smaller than 44.

⇒ Now we get two sorted lists

22	33	11	40	44

Sublist ①

90	77	66	99	55	88

Sublist ②

And these are sorted under the same process as Above done.

(22) 33 11 40 44 (90) 77 60 99 55 88

Pivot

pivot

11 33 (22) 40 88 77 60 99 55 (90)

11 22 33 40 88 77 60 (90) 55 99

First sorted list

88 77 60 (55) (90) 99

pivot

Sublist 3 Sublist 4

55 77 60 88 90 99

Sorted

55 77 60

55 60 77

Sorted.

Now Merging all sublistss

11	22	33	40	44	55	60	77	88	90	99

Quick Sort worst Case Complexity :- $T(n) = n^2$

Quick Sort Average Case Complexity $T(n) = n \log n$

SHELL SORT

Shell sort is a highly Efficient sorting algorithm and is based on insertion Sort algorithm. This algorithm avoids large shifts as in case of Insertion Sort, if the smaller value is to the far right has to be moved to the far right & has to be moved to the far left.

This algorithm uses insertion sort on a widely spread Elements, first to sort them and then sorts the less widely spaced Elements. This spacing is termed as interval.

The Shell Sort sometimes called the "Diminishing Increment Sort" improves On the insertion sort by breaking the original list into a Number of smaller sublists. Each of which sorted using an insertion sort.

Algorithm for Shell Sort

Step (1) Initialize the value of Elements

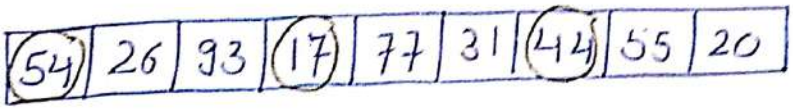
Step (2) Divide the list into smaller sub-list of Equal interval

Step (3) Sort these sublists using insertion Sort

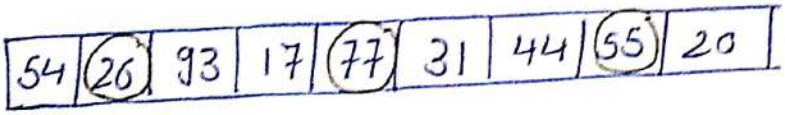
Step (4) Repeat Until Complete list is sorted

Sort the following using Shell Sort

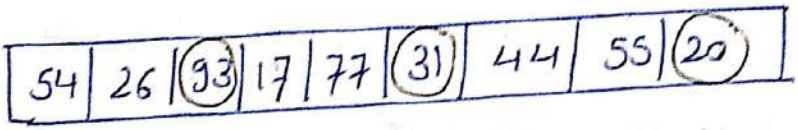
54, 26 93, 17, 77, 31, 44, 55, 20



sublist 1

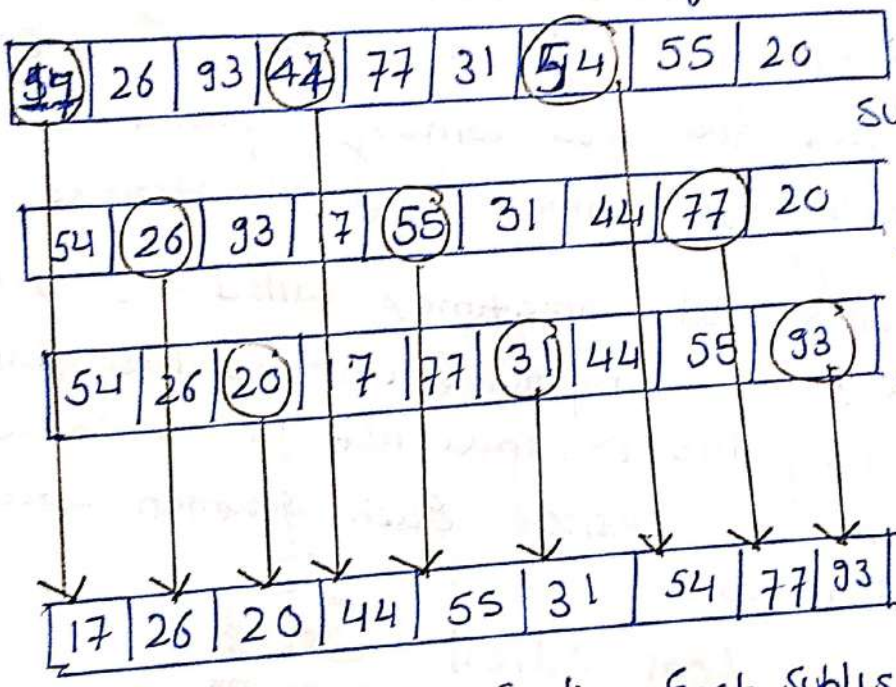


sublist 2



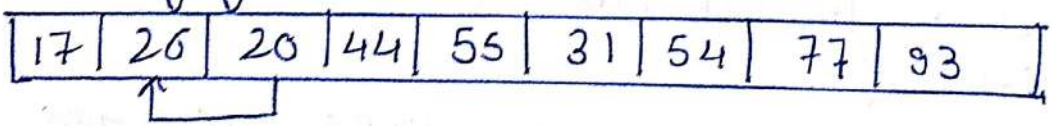
sublist 3

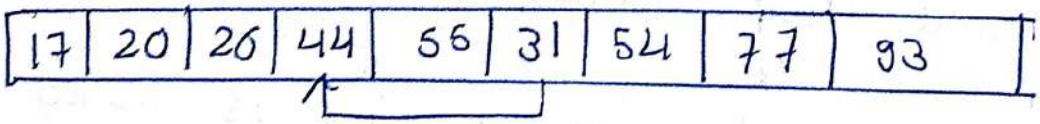
A shell sort with increament of three

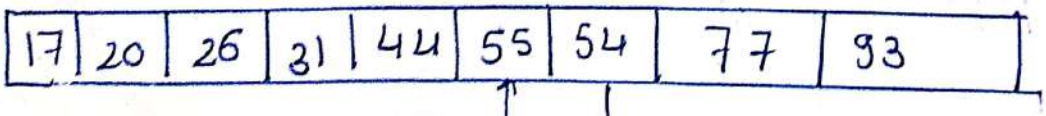


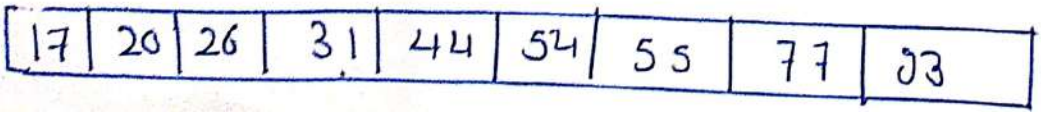
A shell sort after sorting each sublist.

Now Applying Insertion sort:-









Recurrence Equation

Recurrence Equation is an Equation that defines Sequence Recursively.

It is normally in the form of :-

$$T(n) = T(n-1) + n \quad \text{——} \quad (1)$$

$$T(0) = 0 \quad \text{——} \quad (2)$$

Equation (1) is called Recurrence Relation &
Equation (2) is called initial condition

Solving Recurrence Equation

There are three Methods to solve Recurrence Eq^n :-

(i) Substitution Method
(ii) Trace Method
(iii) Master Method

Substitution Method

In this Method of substition a guess for the solution is made :-

~~Formal~~ → Forward } substitution method.
→ Backward

Forward Substitution

This method make use of initial condition in the initial term & value for the next is generate. This process continued until some formula is guessed.

Backward Substitution

In this Method backward values are substitued Recursively in order to derive a formula

Example of Forward Substitution

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

n = 1 $\quad T(1) = T(1-1) + 1 \qquad n > 0$

$$T(0) + 1$$

$$n(1) = 0 + 1 = 1$$

n = 2 $\quad T(2) = T(2-1) + 2$

$$= T(1) + 2$$

$$= 1 + 2$$

$$T(2) = 3$$

n = 3 $\quad T(3) = T(3-1) + 3$

$$= T(2) + 3$$

$$= 3 + 3$$

$$T(3) = 6$$

Formed Formula

$$\Rightarrow \frac{n(n+1)}{2} + n^2/2 + n/2$$

Complexity :- $O(n^2)$

Example of Backward Substitution

$$T(n) = T(n-1) + n \quad \text{—— (1)}$$

$\downarrow$

$n = n-1$

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

$$T(n-1) = T(n-2) + n-1 \quad \text{—— (2)}$$

$$T(n) = T(n-2) + n-1 + n \quad \text{—— (3)}$$

$n = n-2$ in Eqn (1)

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

$T(n-2) = T(n-3)+n-2$

$T(n) = T(n-3)+n-2+n-1+n$ — (4)

$n = k$

$T(n) = T(n-k)+n-k+1+n-k+2+n$

$= T(k-k)+k-k+1+k-k+2+n$

$= T(0)+1+2+n$

$= 0+1+2+n$

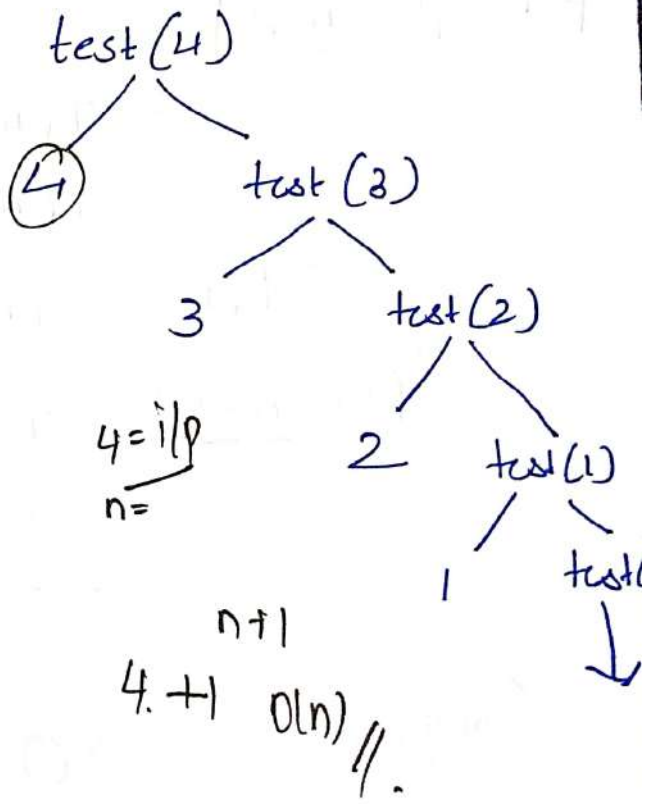
$= n(n+1)/2$

$= n^2$

Complexity = $O(n^2)$

Tree Method

```
void test (int n)
{ if (n>0)
{ print ("%d", n)
  test(n-1)
}
}
```



Find Complexity for the following by Substitution Method.

```
T(n) = void test (int n)
       {
         if (n>0)
         {
           print ("%d ", n)
           test (n-1)
         }
       }
```

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

$T(0) = 1$

Soln:-

$$T(n) = T(n-1) + 2 \rightarrow (1)$$

put $n = n-1$ $\Rightarrow T(n-1-1)+2$ $\quad n = n-1$

$= T(n-2)+2$

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

Substitute in (1)

$T(n) = T(n-2)+2+2$ $\quad = T(n-2)+2+2$

$T(n) = T(n-2)+4 \rightarrow (2)$

put $n = n-2$

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

$T(n-3)+2$

$T(n) = T(n-3)+6$

put $n = K$ $\quad = T(n-K)+K$

$\Rightarrow T(n-n)+n$

$= T(0)+n$

$= 1+n$

Complexity $\Rightarrow O(n)$

Find the Complexity of below Recurrence.

$$T(n) = \begin{cases} 3T(n-1), & \text{if } n>0 \\ 1 & \text{otherwise} \end{cases}$$

Let us solve using ~~Recursion~~ Substitution :-

$$T(n) = 3T(n-1)$$
$$3(3T(n-2))$$
$$3^2T(n-2)$$
$$3^3T(n-3)$$
$$3^4T(n-4)$$
$$\vdots$$
$$3^nT(n-n)$$
$$3^nT(0)$$
$$= 3^n$$

This shows that the Complexity of the fun^n

Find recurrence eqn using substitution method :-

```
int recursive (int n)
{ if (n<2)
  return 1;
  else
  return recursive (n-1) + recursive (n-1)
```

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

$T(n) = 2T(n-1) + c$ ——— (1)

$2\{2T(n-2) + c\} + c$ ——— (2)

$T(n) = 2^2 T(n-2) + 3c$

$2^3 T(n-3) + 7c$ --- 3rd

$\vdots$

$T(n) = 2^3 T(n-3) + (2^3 - 1)c$ --- 3rd

$\vdots$

$T(n) = 2^k T(n-k) + (2^k - 1)c$ --- k^{th}

Here $n-k = 1$ $\{(n-1) < 2\}$

$K = n-1$

Substitute k in (*)

$T(n) = 2^{n-1} T(1) + (2^{n-1} - 1)c$

$2^{n-1} + (2^{n-1} - 1)c$

$O(2^{n-1})$

$O(2^n/2$

$\Rightarrow O(2^n)$

Heap Sort

This Sorting Method discovered by J.N. J. Williams
It consist of two stages :—

Heap —
- Heap construction
- Deletion of Minimum key

Heap Construction :- First we have to construct a heap for given Numbers.

Deletion of Minimum Key :- Delete Root key always (n-1) to Remaining heap

For an array implementation of Heap. Delete the Element from the heap and push the deleted Element in the last position of the array. Thus after deleting all the Elements from the heap we will get the Elements in ascending or Descending order.

Heap

Heap is a Complete Binary tree or atmost binary tree in which parent node be Either greater or lesser than its Child Nodes.

Types of Heap

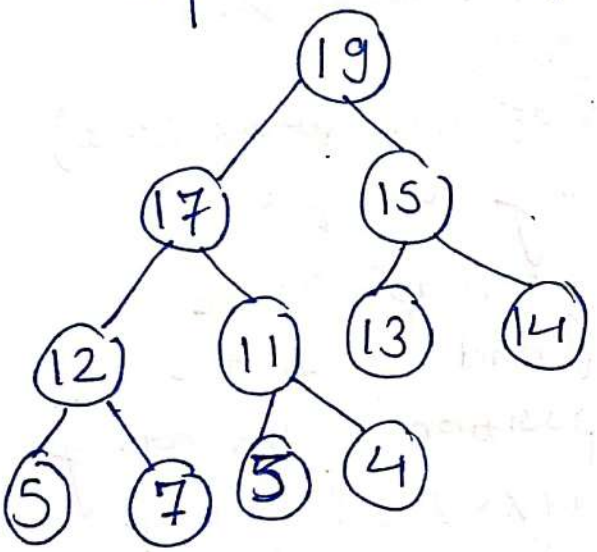
- Max Heap (Descending order) — Ascending
- Min Heap (Ascending order) — Descending.

(*) Max Heap is a tree in which value of each Node is smaller than the parent Node.

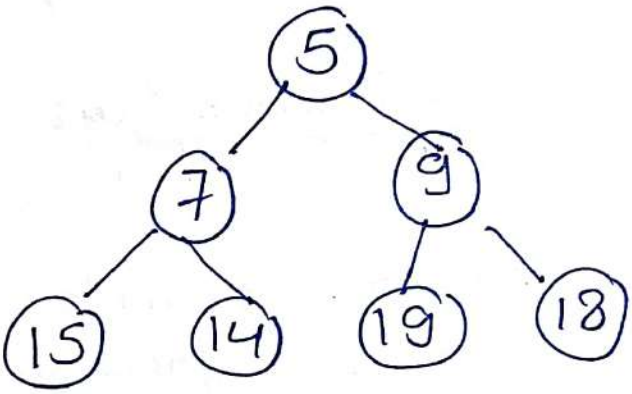
(*) Min Heap the value of Child Node is greater than of Parent Node.

Heap —
- It should be Complete Binary tree
- parented properly

Maximum Heap



Minimum Heap



Algorithm for Max-heap

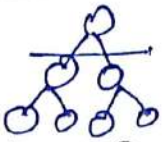
Maxheap : parent > child

```
for (i = (n/2); i>0; i--)
{
```

it takes O(n/2) times

∴ O(n)

Max Heap

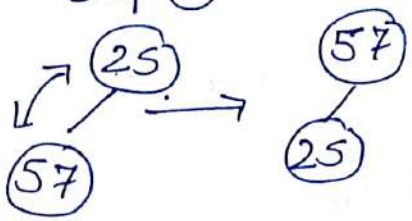


Binary tree

Example:- 25, 57, 48, 37, 12, 92, 86, 33

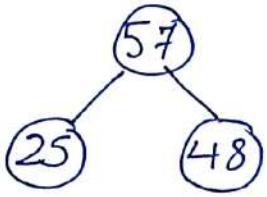
Step (1)

Step (2)

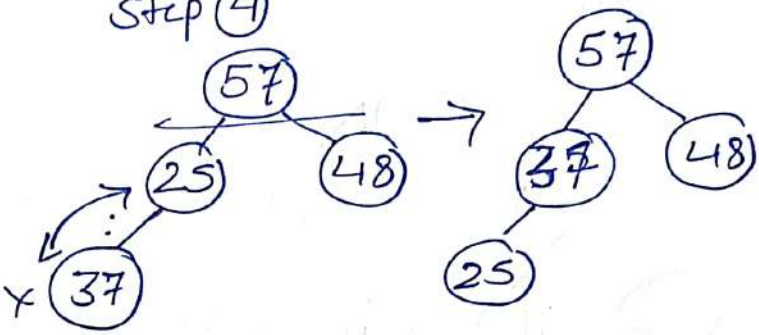


Max Heap
each value ≥ child Nodes.

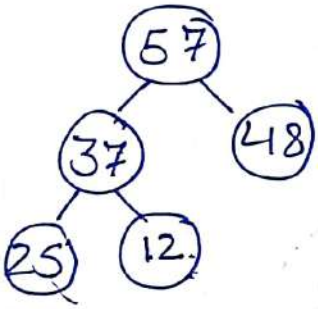
Step (3)



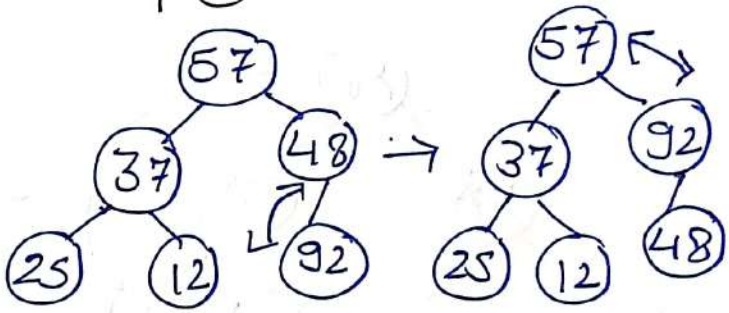
Step (4)

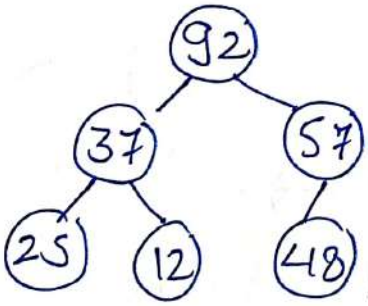


Step (5)

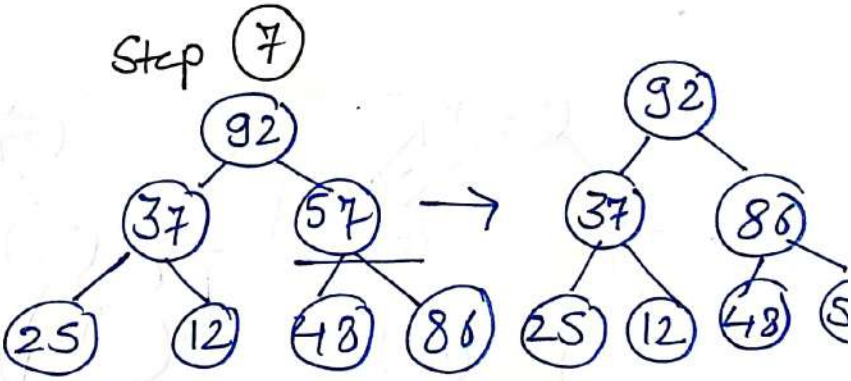


Step (6)

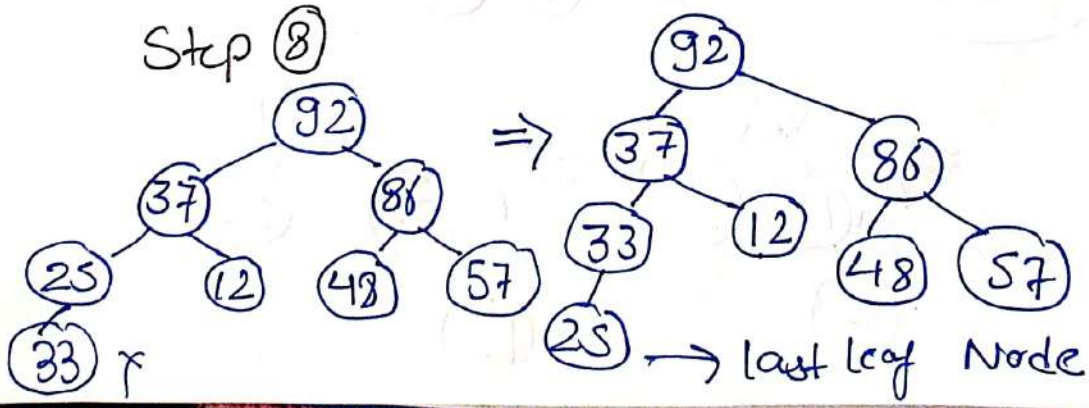


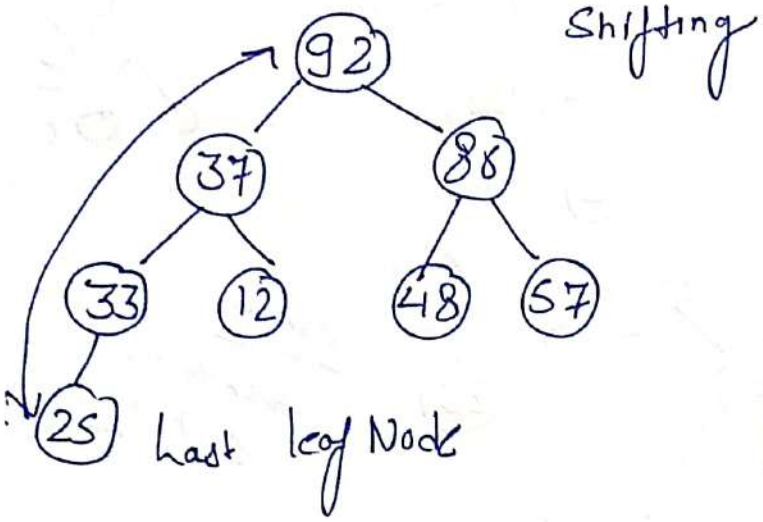


Step (7)

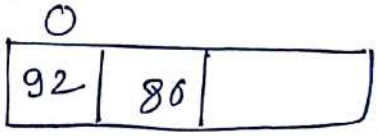


Step (8)

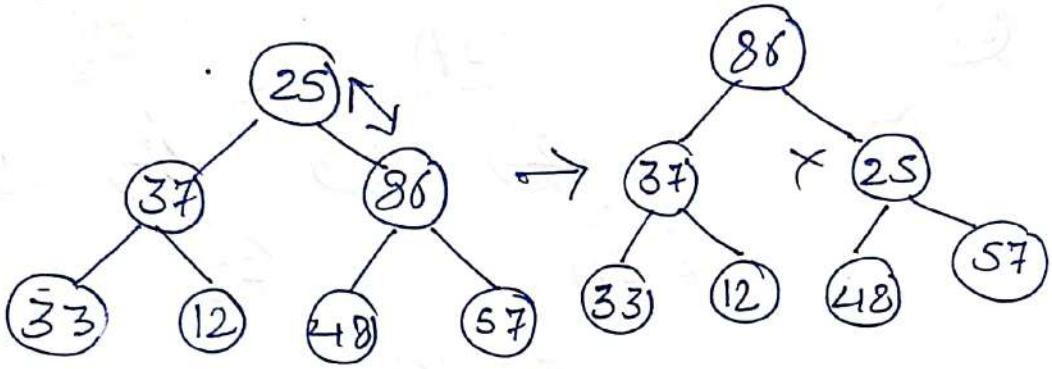


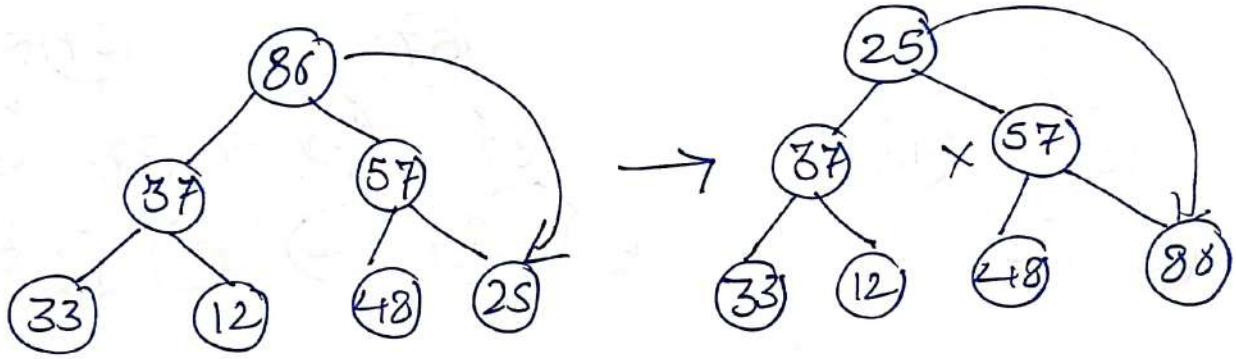


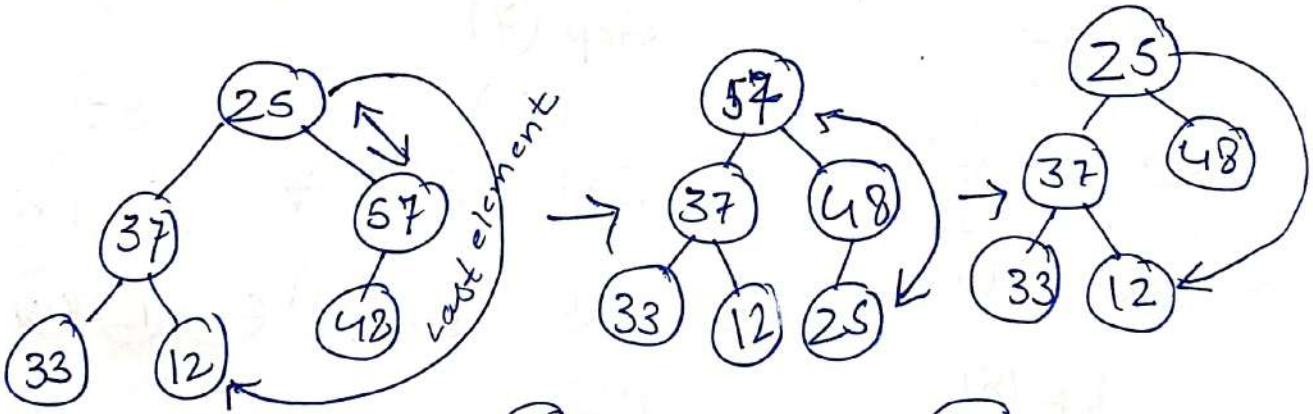
Shifting

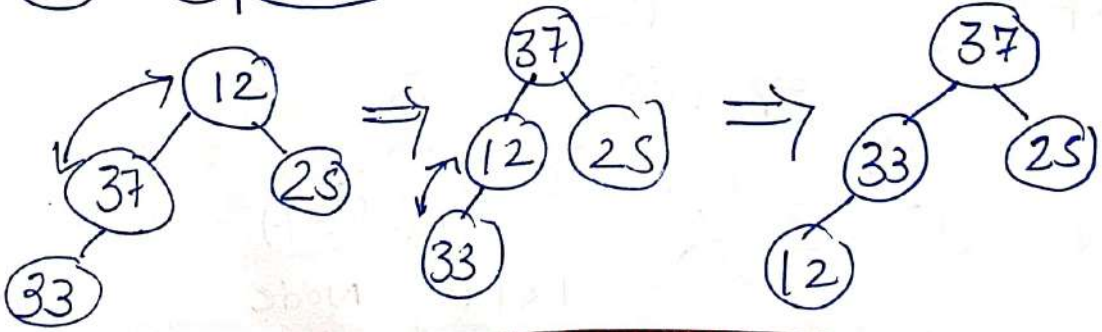


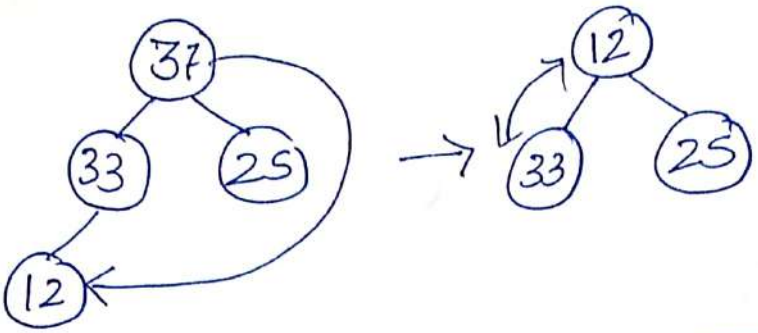
~~Sorts~~



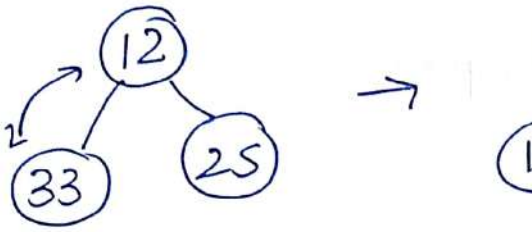




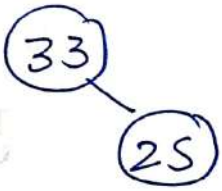




12	25	33	37	48	56	86	92



12	25	33	37	48	56	86	92



Sorted Array

12	25	33	37	48	56	86	92

Complexity of Heap Sort

Worst Case Time Complexity : $O(n*\log n)$

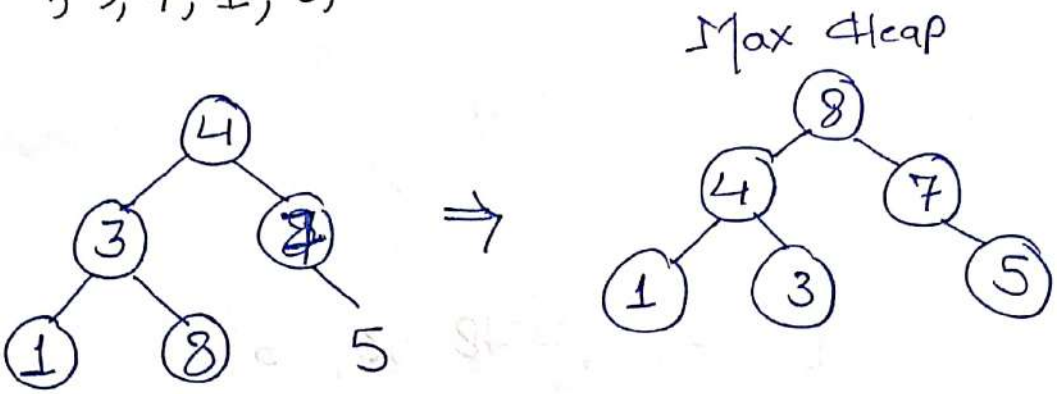
Best Case Time Complexity : $O(n*\log n)$

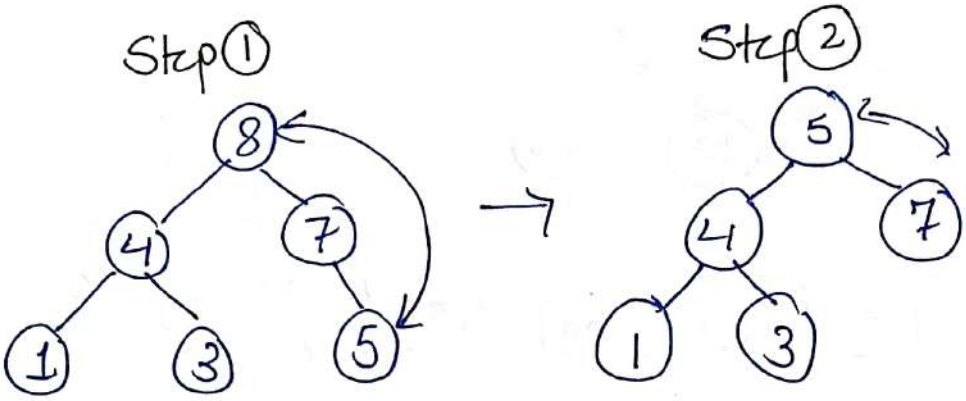
Average Time Complexity : $O(n*\log n)$

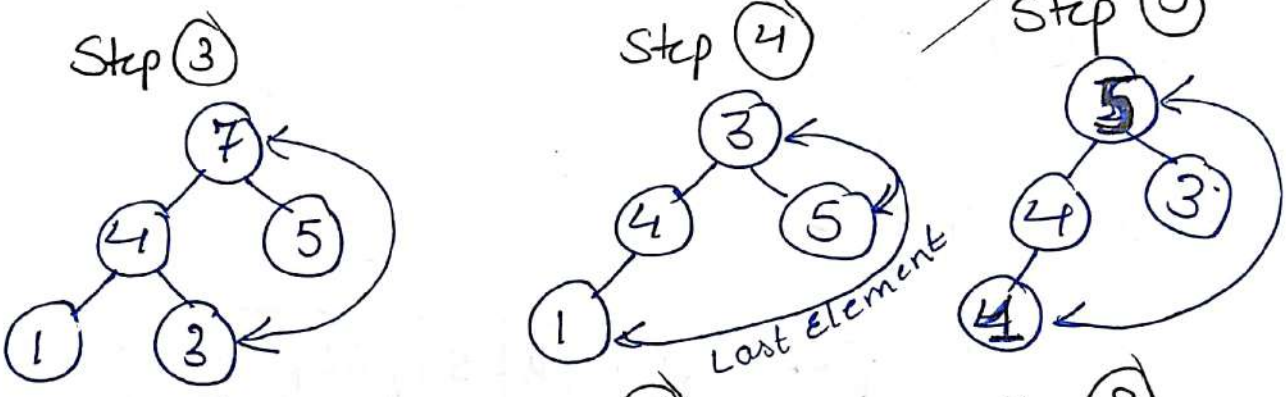
Space Complexity : $O(1)$

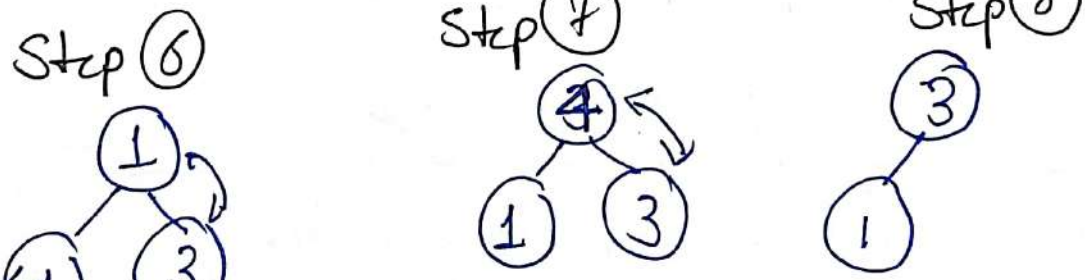
* Heap Sort is not a stable Sort and requires a Constant space for Sorting a list.

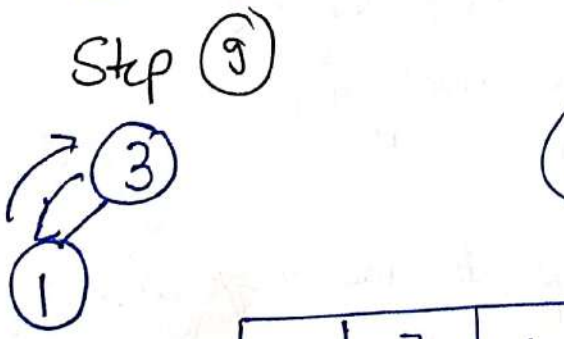
4, 3, 7, 1, 8, 5







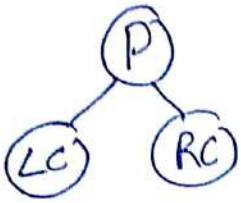




1	3	4	5	7	

Min_Heap Algorithm

Min_Heap Algorithm :- Decreasing order



$P \leq LC$ & Rc

P → Parent Node

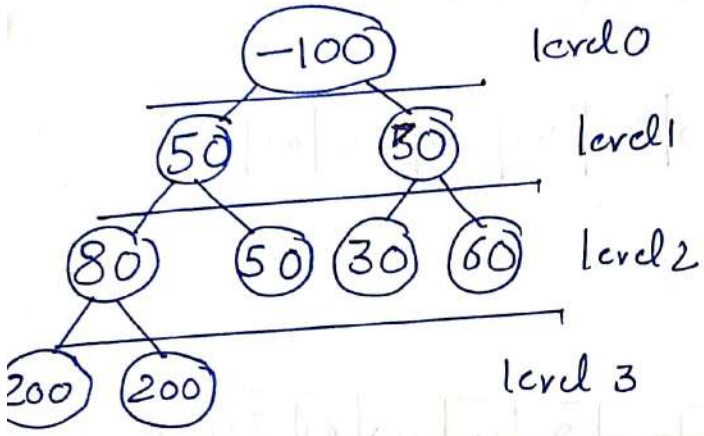
where LC - Left child

Rc - Right child.

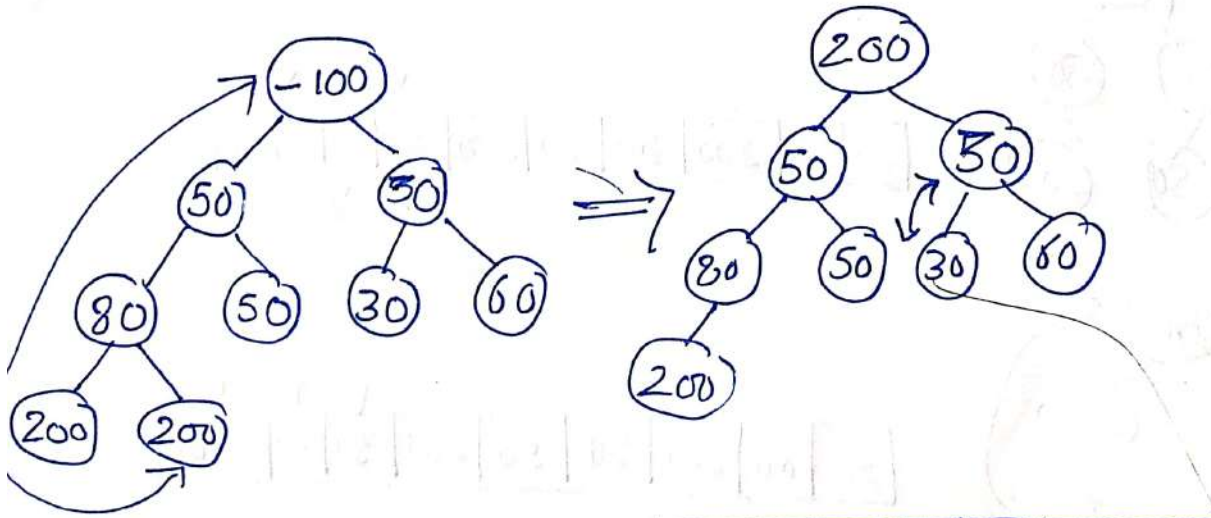
Example ①

Show the tree of Heapsort algorithm for the following input data.

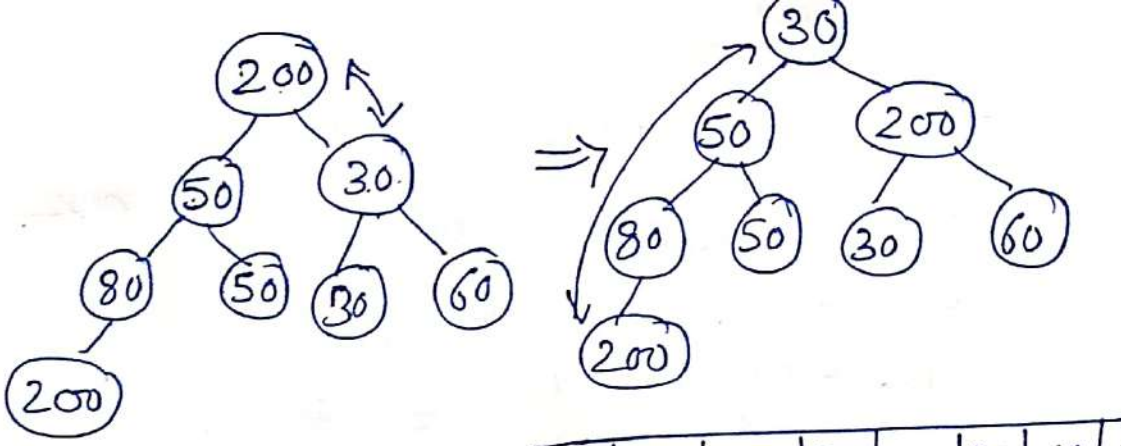
30, 50, -100, 200, 50, 30, 60, 80, 200 in order



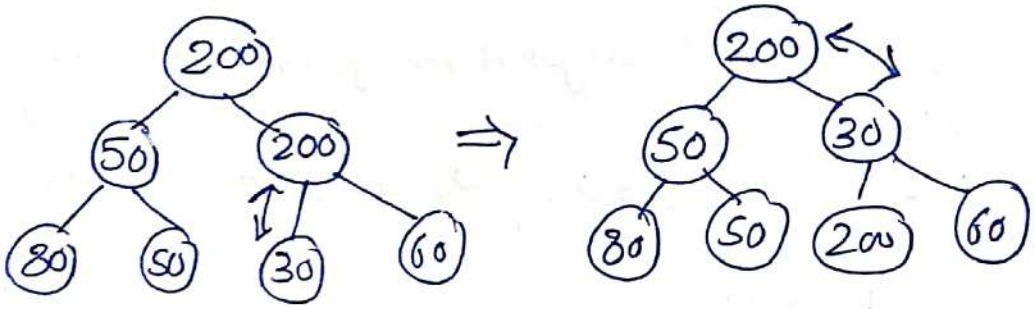
-100	50	30	80	50	30	60	200	200
0	1	2	3	4	5	6	7	8



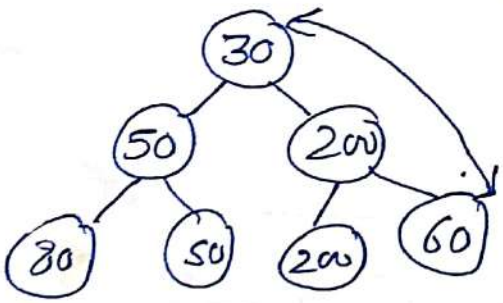
200	50	30	80	50	30	60	200	-100



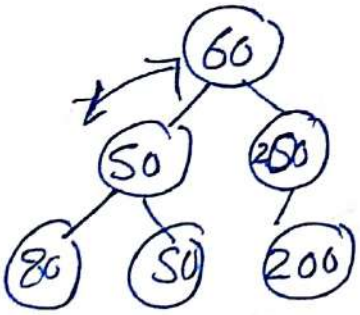
30	50	200	80	50	30	60	200	-100



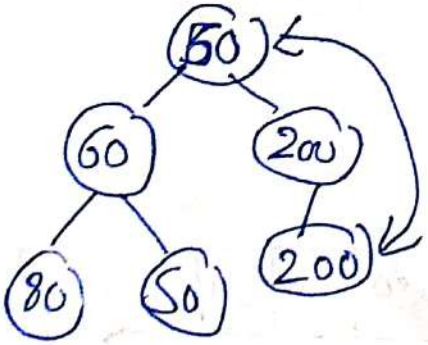
200	50	30	80	50	200	60	30	-100
							x	



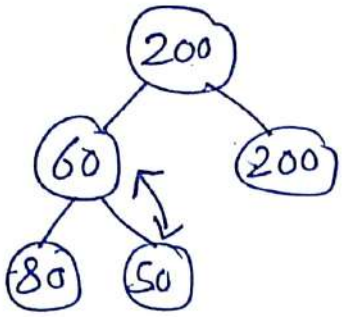
30	50	200	80	50	200	60	30	-100

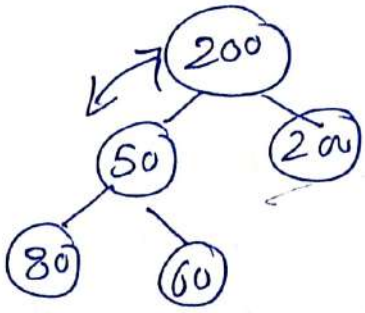


						✓	✓	✓	
60	50	200	80	50	200	30	30	-100	
						x	y	x	



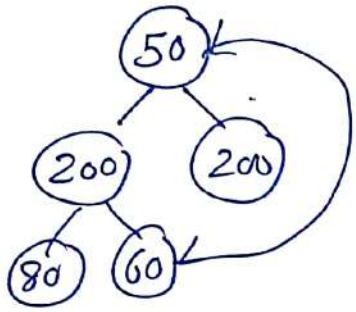
							✓	✓	✓
50	60	200	80	50	200	30	30	-100	



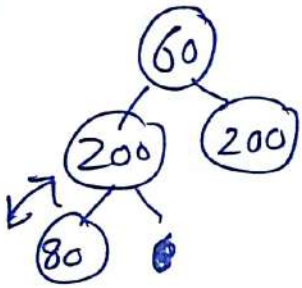


200	60	200	80	50	50	30	30	-100

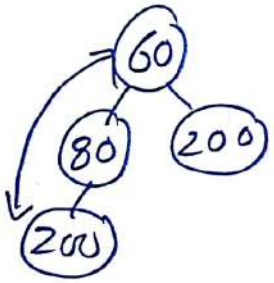
200	50	200	80	60	50	30	30	-100



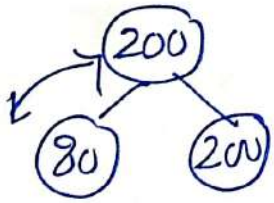
50	200	200	80	60	50	30	30	-100



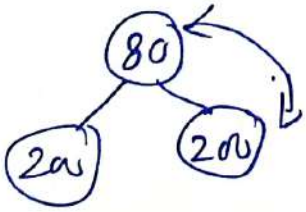
60	200	200	80	50	50	30	30	-100

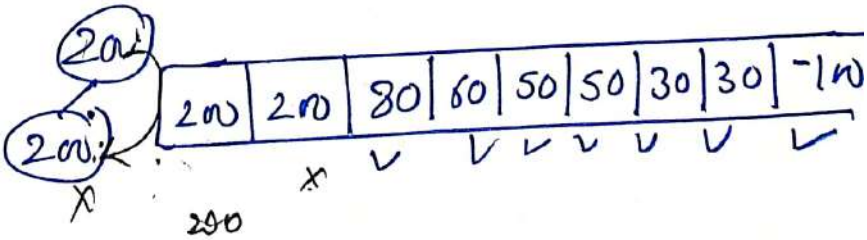


60	80	200	200	50	50	30	30	-100



200	80	200	60	50	50	30	30	-100





200

Sorting

* The Sorting problem is to rearrange the items of a given list an specified order.
* Sorting can be done on numbers, characters, strings or Records.
* A sorting algorithm is called Stable if it preserves the Relative order of any two Equal Elements in its input.
* Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order. A collection of records called a list where Every record has one or More fields.

Categories of Sorting

Internal Sorting:- If all the data that is to be sorted can be adjusted at a time in the main Memory, the internal sorting Method is being performed.

External Sorting:- When the data is to be sorted cannot be accommodated in the Memory at the time & some has to be Kept in Auxiliary Memory.

Several Sortings :-

Bubble Sort:- A sorting algorithm which compares one Element to its next Element to its next Element & if requires it swaps like a bubble.

Selection Sort:- A sorting algorithm which selects a position in the Elements & Compares it to the rest of the positions one by one.

Insertion Sort:- It is the sorting Mechanism where the sorted array is built having one item a time.

Quick Sort:- A sorting algorithm which divides the Elements into two subsets & again sorts recursively.

Heap Sort:- A sorting algorithm which is a comparison based sorting technique based on Binary Heap data structure.

Merge Sort:- A sorting algorithm which divides the elements to subgroups & then merges back to make a sorted.

Radix Sort:- A sorting algorithm used for numbers. It sorted the elements by Rank of the individual digits

Comparison of Sorting techniques

Algorithm	Time Complexity			Extra space
	Best Case	worst Case	Average Case	
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	1
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	1
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	1
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	n (worst case)
Quick Sort	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	$\log n$ (avg.), n (worst case)
Radix Sort	$O(s.d.n)$	$O(s.d.n)$	$O(s.d.n)$	n
Shell Sort	$O(n)$	$n\log^2 n$	$O(n\log^2 n)$ or $O(n^{7/6})$	1

* Different Sorting techniques have different uses according to their behaviour for different inputs Every sorting technique has own complexity. ~~humans & Quick sort is poorly sorted for humans.~~ (↳ ~~Merge sort is poorly used for~~)

* For restricted data (data in a fixed interval) radix sort is useful.

* Bubble sort is rarely used, but found in teaching & theoretical Expressions.

* Nowdays Shell sort is rarely used in serious applications. It does not use stack.

* Some implementations of the Quick sort fun in the C. Standard library targeted at Embedded systems. Use it instead of Quick Sort.

* Selection Sort is useful where swapping is costly.

* Insertion Sort is use for small data sets. Merge Sort & Quick sort are useful for large data sets

Linear Time Sorting

* There are sorting algorithms that run faster than O(nlogn) time but they require special assumptions about the input sequence to be Sort.

Examples of Sorting algorithms that run in linear time are Counting sort, radix sort and bucket sort.

* Despite of linear time Usually these algorithms are not very desirable from practical point of view.

(*) The Efficiency of Linear time algorithms depend on the keys randomly ordered. If this condition is not satisfied the result is the degrading in performance

(*) These algorithms require Extra space proportional to the size of the array being sorted, so if we are dealing with large files.

(*) The inner loop of these algorithms Contain quite a few instructions so Even though they are linear they would not be as faster than Quick sort.