UNIT - 1
INTRODUCTION

Algorithm :→ An algorithm is a sequence of computational steps that transform the input into output.

Basically, it is a finite set of instructions that specify a sequence of operation to be carried out in order to solve a specific problem.

Properties of Algorithm :→ There are 5 basic properties of algorithm :→

1) Input :→ Each algorithm should receive some set of input variables to work upon.

2) Output :→ Each algorithm should produce a set of output after performing its computations on set of input variables.

3) Definite :→ Each algorithm should terminate after a finite no. of steps.

4) Effectiveness :→ All the instructions specified in the algorithm should be as much effective so that it can be carried out manually.

5) Unambiguous :→ Each & every instruction in the algorithm should be clearly specified without any chance of duplicacy or ambiguity for ex:- If an instruction states add X to 2 then the value & data type of x should be clearly defined.

Designing strategies of Algorithms :→ The following strategies are

used to design the algorithms :→

1) Divide & Conquer Technique :- In this technique, the original problem is divided into set of some problems which are then solved individually.

After that, the sol^n of the subprograms are combined to get the sol^n of the original problem.

for ex :→ Quick sort, binary search, merge sort.

2) Greedy Approach :- These algorithms are used to optimize the function which are locally based but may not generate the global optimal sol^n. However it generally produces the sol^n that are closed to the optimal sol^n. for ex :→ Activity selection problem, Travelling salesman problem etc.

3) Dynamic programming :- In this technique, the global optimal sol^n is obtained bcoz the subproblems are worked out for all the possible sol^n & the best out of all the possible sol^n is chosen.

Imp :→ for ex :→ 1) Matrix Chain Multiplication (MCM)
2) Longest Common Subsequence (LCS)

4) Backtracking Approach :- In this approach, we tried each possibility until the right one is found.

During the search, if an alternative doesn't go, the search is backtrack to the choice point, a new alternative is tried until & unless the final sol^n is

obtained.

for ex:→ Graph coloring, N-queens problems.

5) Branch & Bound Approach:→ In this technique, the various possible soln are represented are rep in the form of subproblems that are solved individually & non-convenient problem.

for ex:- LC & FIFO Branch Bound.

21/08/17 DAA - Lecture

Algorithms Conventions:- Some conventions are required to be followed by writing a pseudocode.

1 Comments begin with // & continued until the end of line. (//)

2. Blocks are indicated with the set of curly braces. ({ })

3 The datatype of variable is declared as:→
datatype data1; ⇒ int a;

4. Assignment of the variable is done using:-
variable := expression. a := 5

5. Comparison of two variables is done using :→
variable = expression eg ⇒ a = 10.

6. Elements of multi-Dimensional array are repres

ented as:-
A[i, j] ⇒ i = row & j = column.

7. The for loop in the algorithm is expressed as:-
for variable := value 1 to value 2 step no' do
{
statement 1;
⋮
statement n;
}

8. A conditional statement is expressed as:-
if (condition)
then expression 1
else expression 2.

9. Input & output are done using read & write instruction respectively.

10. Algorithm keyword is always prefixed before starting any algorithm & represented as:-

Algorithm name (parameter list)

where name = represents name of procedure.
& parameter list = the variables that will be used while designing the algorithm.

Performance Analysis of algorithms:- These are of 2 types:-
1) Time Complexity
2) Space "

Time Complexity :- The time complexity indicates the total time that will be taken by the algorithm to run to its completion. It can be denoted as:-

$$t = t_c + t_r$$

where t_c = Compile time
t_r = Running time of algorithm.

The compile time is usually fixed & does not depend on instance characteristics. So, while computing the total time only running time of the algorithm is taken into account.

$$t = t_r$$

It has two methods:→

1) Count Method 2) Step Table method.

Count Method :- 1) In this method, a global variable count is taken & is initialize to zero (0).

2) The count is incremented on the basis of time for which that particular statement will be executed.

3) No significant time is taken into account for comment, braces & algorithm name.

Example 1:→ Sum of n natural number:-

```
Algorithm Sum (a, n) → global count=0
{
    S=0.0;
    count= count+1;        1
```

* Here a: is an array of n natural no whose sum S is to be executed

```
for (i=1 to n) do                    n+1 → To check the false condition
{  count = count+1;
   S = S+a[i];  → count = count+1;    n
}
return S;  → count = count+1;         1
}
```

Time complexity, $T(n) = 2n+3$

$T(n) \approx O(n)$ → Linear Running Time

Example 2:- Addition of two matrix:-

```
Algorithm matrix add (a,b,c,m,n)
{      global count = 0;
1      for (i:=1 to m) do  → count = count+1;      m+1
2      {
3          for (j:=1 to n) do  → count = count+1;      m(n+1)
4          {
5              c[i,j] = a[i,j] + b[i,j]
6              count = count+1;                        mn
7          }}
```

Time Complexity, $T(n) = (m+1)+m(n+1)+mn$

$= m+1+mn+m+mn$

$T(n) = 2m+2mn+1$

$= 2m(\because 1+n)+1$

$T(n) \cong O(mn)$

DAA-Lecture.

Example 3:→ RSum (Recursion Sum) ⇒ Sum of n natural no.

Algorithm RSum (a, n)

global count = 0;

1. {

2 if (n≤0) then
count = count+1;

3 retun 0.0;
count = count+1;

4 else

5 return (Rsum (a, n-1) + a[n]);
count = count+1;

6. }

	n=-1 n≤0	n>0
2	1	1
3	1	—
5	—	$t_{RSum(n-1)} + 1$

Let $T_{Rsum(n)}$ denotes the total time to compute the sum of n natural no. using recursion. It involves the total tym e.g. for computation procedure calling passing of parameters into stack & transfer of control from one procedure to another. Since, recursive sum is called itself (n-1) times. So, the total tym required is denoted by :-

$$\boxed{T_{Rsum(n-1)} + 1} \rightarrow \text{final return of sum}$$

$$T_{Rsum(n)} = \begin{cases} 2 & \text{if } n \leq 0 \\ 2 + T_{Rsum(n-1)} & \text{if } n > 0 \end{cases}$$

STEP-Table Method :- It is the second method to determine the step-count of an algorithm. In this method, there are 2 main parameters :-

1. S/e ⇒ Steps per execution.
2. Frequency.

1. S/e : The steps per execution of the statement is the amount for which the account changes as a result of execution of that statement. The value zero indicates that the statement is executed but takes negligible time for execution while value 1 represents a significant amount of compiler time is taken for execution.

2. Frequency :- It denotes the total no. of execution steps of a particular statement.

Example 1:→

	Statement	S/e	Frequency	Total steps (S/e * Frequency)
1	Algorithm Sum(a,n)	0	—	—
2	{	0	—	—
3	S=0.0;	1	1	1
4	for(i=1 to n) do	1	n+1	n+1
5	S=S+a[i]	1	n	n
6	return S;	1	1	1
7.	}	0	—	

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

Example 2:-

	Statement	S/e	Frequency	Total Steps.
	Algorithm Matrix Add (a,b,c,m,n)	0	—	—
	{	0	—	—
	for(i=1 to n) do	1	m+1	m+1
	{	0	—	—

for (j=1 to n) do	1	m(n+1)	mn+m
{	0	—	—
c[i,j] = a[i,j] + b[i,j]	1	mn	mn
}	0	—	—
}	0	—	—
}	0	—	—

$T(n) = 2mn + 2m + 1$

Example 3 :→

	Statement	S/e	Frequency n≤0	Frequency n>0	Total Steps n≤0	Total Steps n>0
1	Algorithm RSum(a,n)	0	—	—	—	—
2	{	0	—	—	—	—
3	if (n≤0)	1	1	1	1	1
4	return 0;	1	1	—	1	—
5	else return RSum (a,n-1)+a[n];	1	—	$t_{RSum(n-1)}$	—	$1+t_{RSum(n-1)}$
6.	}	0	—	—	—	—
					2	$2+t_{RSum(n-1)}$

$$T_{RSum(n)} = \begin{cases} 2 & \text{if } n \leq 0 \\ 2 + t_{RSum(n-1)} & \text{if } n > 0 \end{cases}$$

Analysis of an algorithm indicates the total no. of resources that are required for an algorithm to complete its execution. It is normally defined in terms of time & space required by an algorithm.

Space Complexity :→ It is the total amount of space required by an algorithm to run to its completion. The total space consumed is denoted by :-

$$S = S_c + S_v$$

where S_c = the amount of space taken by constants or the variables that are defined & computed at compile tym.

S_v = the amount of space taken by variables i.e usually defined at run tym on the basis of instances characteristics provided by the user.

So, while computing total space ~~complexity~~, S_c is usually ignored & in the final total space complexity, only S_v will take into considerations.

$$\boxed{S = S_v}$$

Example 1

Algo Calculate (a,b,c)
{
1 int a=2, b=3, c;
2 c=a+b
3 return c
}

Sv = 0
S = 0

Example 2

Algorithm Sum (a,n)
{
1 S=0.0;
2 for (i=1 to n) do
3 S=S+a[i];
4. Return S;
}

Sv = 3
S = n+3

⇒ S, i, n

Asymptotic Notations:- There are 5 types of AN:-

1 Big Oh Notation (O)
2 Big Omega Notation (Ω)
3 Theta Notation (Θ)
4 Little Oh notation (o)
5. Little Omega notation (ω)

AN:- These notations are used to judge the order of growth of the running tym in accordance to the input size i.e when n is increased, the proportion-

ate running time growth is computed. An algorithm i.e asymptotically more efficient will be the best choice for the set of inputs.

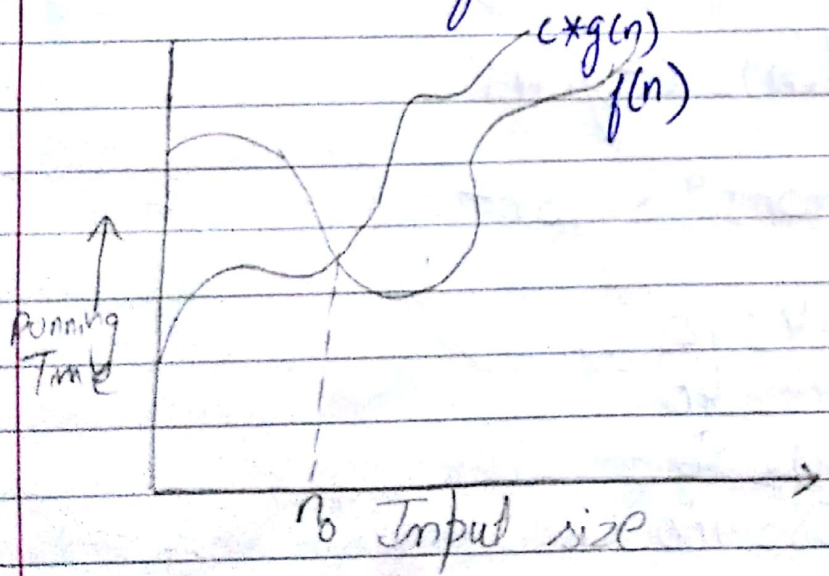
1 O:→ This notation is used to provide an upper bound to the ~~not~~ algorithm with a constant factor i.e when an asymptotic upper bound needs to be computed, this notation is used.

for a given function, $f(n)$ & $g(n)$, the big Oh notation can be represented as:

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

for the +ive constants c & n_0 such that :→

$$0 \leq f(n) \leq c * g(n)$$
$$\text{for } c \geq 0,\ n \geq n_0.$$



Steps to compute O:→

Step 1:→ Given a function $f(n)$, $g(n)$ is computed to be the highest degree n.

Step 3:- The value of c is equal to the coefficient of highest degree 'n' + 1.

Example1:- $f(n) = 3n+2$.

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

$3n+2 \leq 4n$

$n=1$, $5 \not\leq 4$

$n=2$, $8 \leq 8$

$\boxed{f(n) = O(g(n))}$ for $n \geq n_0$ & $c \geq 0$

$\Rightarrow \boxed{f(n) = O(n) \text{ for } n \geq 2, c = 4}$

Example2:- $f(n) = 9n^2+3n+2$.

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

$9n^2+3n+2 \leq 10n^2$.

$n=1$ $\quad 14 \not\leq 10$

$n=2$ $\quad 44 \not\leq 40$

$n=3$ $\quad 92 \not\leq 90$

$n=4$ $\quad 158 \leq 160$

$\Rightarrow \boxed{f(n) = O(n^2) \text{ for } n \geq 4 \quad c = 10}$

Example 3 $f(n) = 300n^0$

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

$300 \leq 301 * 1 * n^0$

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

$\Rightarrow \boxed{f(n) = O(1)} \Rightarrow$ for any value.

Example 4:- $f(n) = 12 \log n + 3$

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

$12 \log n + 3 \leq 13 \log n$

$n = 1 \qquad 3 \leq 0$

$n = 2$

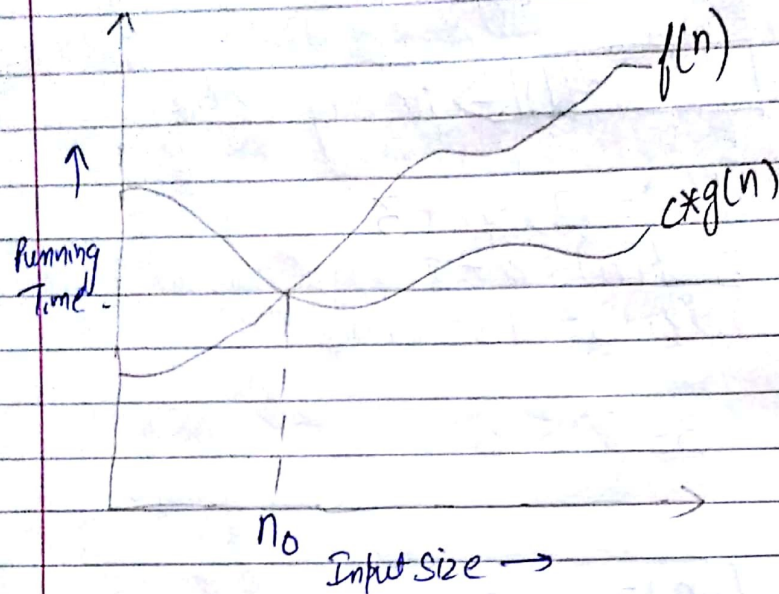
$$\boxed{f(n) = O(\log n) \text{ for } n \geq \quad c = 13}$$

Big O is the formal method of representing the upper bound of an algo. i.e it is the measure of longest amount of time that an algorithm could possibly take for its completion. i.e. worst case.

2) Big Omega notation (Ω):⇒ It represents the lower bound of an algorithm i.e the minimum tym req. by the algorithm to run to its completion i.e the best case.

For non-negetive functions, $f(n)$ & $g(n)$ the omega notation can be represented as:⇒

$f(n) \geq c * g(n)$
for $n \geq n_0$ & $c \geq 0$
& $f(n) = \Omega(g(n))$



Steps to compute Omega notation :-

Step 1:- For a given function, $g(n)$ is the highest degree n.

Step 2:- If the function contains Σ sign (+), then the value of c is equal to the coefficient of highest degree n.

Step 3:- If the function contains (-) sign, then the value of c is equal to the highest degree coefficient of 'n' ± 1.

Step 4:- If the function contains both +ive & -ive sign, then the sign b/w 2 highest degree 'n' is considered to compute the value of c.

Example 1:- $f(n) = 3n + 2$

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

$3n + 2 \geq 3n$

$n = 1 \quad 5 \geq 3$

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

$\Rightarrow$ **$f(n) = \Omega(n)$ for $n \geq 1$, $c = 3$**

Example 2:- $f(n) = 2n - 3$

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

$2n - 3 \geq 1n$

$n = 1 \quad -1 \not\geq 1$

$n = 2 \quad 1 \not\geq 2$

$n = 3 \quad 3 \geq 3$

$f(n) = \Omega(n)$ for $n \geq 3$, $c = 1$

Example 3:- $f(n) = 2n^3 + 3n^2 - n + 2$

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

$2n^3 + 3n^2 - n + 2 \geq 2n^3$

$f(n) = \Omega(n^3)$ for $n \geq$, $c = 2$

DAA - Lecture

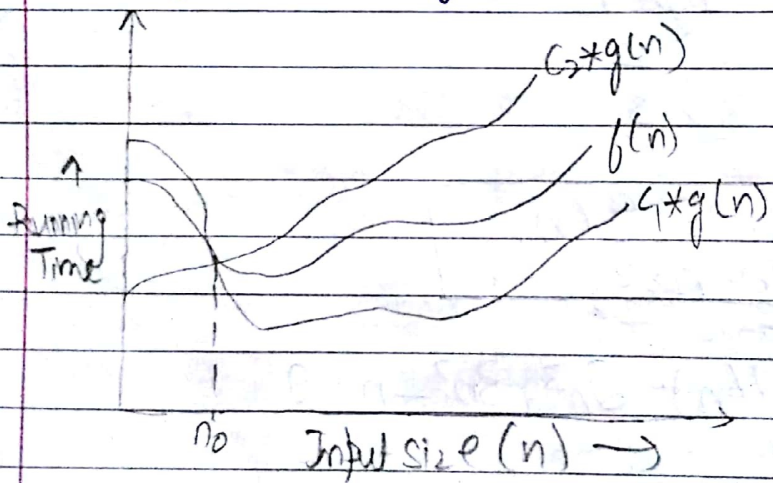
3) Big Theta Notation (Θ):- This notation is used to represent the lower as well as upper bound of an algorithm i.e it is the formal method of representing an avg case of an algorithm.

For non-negative function, $f(n)$ & $g(n)$ there exists an integer n_0 & constants C_1 & C_2 such that:

$$C_1 * g(n) \leq f(n) \leq C_2 * g(n)$$

for $n \geq n_0$, $C_1, C_2 > 0$

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



Example 1:→ $f(n) = 2n + 3$.

$$C_1 * g(n) \leq 2n+3 \leq C_2 * g(n)$$

$$2n \leq 2n+3 \leq 3n$$

$n=1$ $\quad 2 \leq 5 \not\leq 3$

$n=2$ $\quad 4 \leq 7 \not\leq 6$

$n=3$ $\quad 6 \leq 9 \leq 9$

$$f(n) = \theta(n) \text{ for } n \geq 3,\ C_1 = 2,\ C_2 = 3$$

Example 2:→ $f(n) = 10n^2 + 4n + 2$

$$C_1 * g(n) \leq f(n) \leq C_2 * g(n)$$

$$10n^2 \leq 10n^2 + 4n + 2 \leq 11n^2$$

$n=1$ $\quad 10 \leq 16 \not\leq 11$

$n=2$ $\quad 40 \leq 50 \not\leq 44$

$n=3$ $\quad 90 \leq 104 \not\leq 99$

$n=4$ $\quad 160 \leq 178 \not\leq 176$

$n=5$ $\quad 250 \leq 272 \leq 275$

$$f(n) = \theta(n^2) \text{ for } n \geq 5,\ C_1 = 10,\ C_2 = 11$$

4) Little Oh Notation (o):- It represents a loose bound version of big oh notation. It only bounds the top upper bound & does not bound from the bottom.

Let $f(n)$ & $g(n)$ are the set of non-negative function that can be represented as:-

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

$$\Rightarrow f(n) = o(g(n))$$

Example 1:- $f(n) = 5n+3$

Let $g(n) = n^2$

$$\lim_{n\to\infty}\left[\frac{5n+3}{n^2}\right] = \lim_{n\to\infty}\left[\frac{5n}{n^2}+\frac{3}{n^2}\right] = \lim_{n\to\infty}\left[\frac{5}{n}+\frac{3}{n^2}\right] = 0$$

$$\boxed{f(n) = \Theta(n^2)}$$

n should highest degree coefficient + 1.

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

Let $g(n) = n^4$

$$= \lim_{n\to\infty}\left[\frac{4n^3}{n^4}+\frac{3n^2}{n^4}-\frac{2n}{n^4}+\frac{1}{n^4}\right]$$

$$= \lim_{n\to\infty}\left[\frac{4}{n}+\frac{3}{n^2}-\frac{2}{n^3}+\frac{1}{n^4}\right] = 0 \qquad \boxed{f(n) = \Theta(n^4)}$$

5) Little Omega Notation (ω):⇒ It represents the loose lower bound of big omega notation. It bounds the function from the bottom & not from the top. This notation is suitable for those functions having a linear or constant running time. for values having quadratic or higher running time, this notation is never used.

Let $f(n)$ & $g(n)$ are the set of non-negative functions that can be for omega notation as:

$\lim_{n \to 0} f(n) \cdot g(n) = 0$
$\Rightarrow f(n) = \omega(g(n))$

Example 1 = $f(n) = 200$.
Let $g(n) = n$

$\lim_{n \to 0} f(n) \cdot g(n)$

$\lim_{n \to 0} [200 * n] = 0$ $\quad f(n) = \omega(n)$

Example 2 ⇒ $f(n) = 4n + 2$

Let $g(n) = n$

$\lim_{n \to 0} [f(n) \cdot g(n)]$

$= \lim_{n \to 0} [4n + 2] n \quad \Rightarrow \lim_{n \to 0} [4n^2 + 2n] = 0$

$f(n) = \omega(n)$

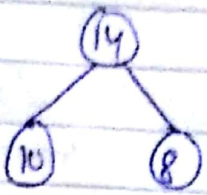
Heap Sort :- The heap is a data structure of the form of complete binary tree where each node of the tree corresponds to an element of an array.

The tree is completely filled on all the levels except at last & the elements are inserted in the order of top to bottom & from left to right.

Heap is of two types:-

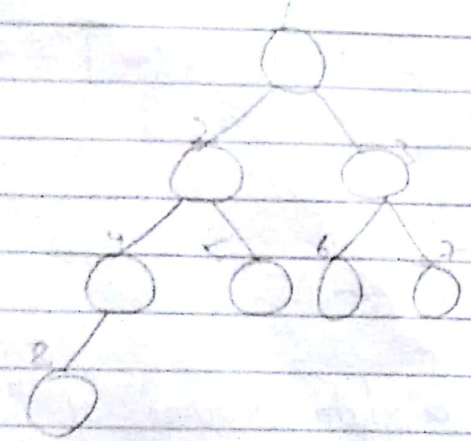
1. Max heap:- In this heap, the max heap property needs to be satisfied for every node which states that value of parent node should always be greater than child node.

$A[Parent[i]] \geq A[i]$



Min heap:- In this heap, the min heap property needs to be satisfied i.e the value of parent node should always be smaller than or equal to child node.

$A[Parent[i]] \leq A[i]$



Parent [i] ⇒
return [i/2]

Left child [i] ⇒
return 2i
Right child [i] ⇒
return 2i+1

Algorithm of heap sort :-

Algorithm Heap sort (A)

1. BUILD - MAXHEAP (A)
2. for i = length [A] down to 2

3. exchange A[1] with A[i]
4. heapsize[A] = heapsize[A]-1
5. MAX-HEAPIFY (A, 1)

} O(n log n)

⇒ BUILD-MAX-HEAP(A)

1. heapsize[A] = length[A]
2. for i = ⌊length[A]/2⌋ down to 1
3. MAX-HEAPIFY[A, i]

} O(n)

⇒ MAX-HEAPIFY[A, i]

1. l = LEFT[i], r = RIGHT[i]
2. If (l ≤ heapsize[A] & A[l] > A[i])
3. largest = l
4. else largest = i
5. If (r ≤ heapsize[A] & A[r] > A[largest])
6. largest = r
7. If (largest ≠ i)
8. exchange A[i] with A[largest]
9. MAX-HEAPIFY[A, largest)

} O(log n)

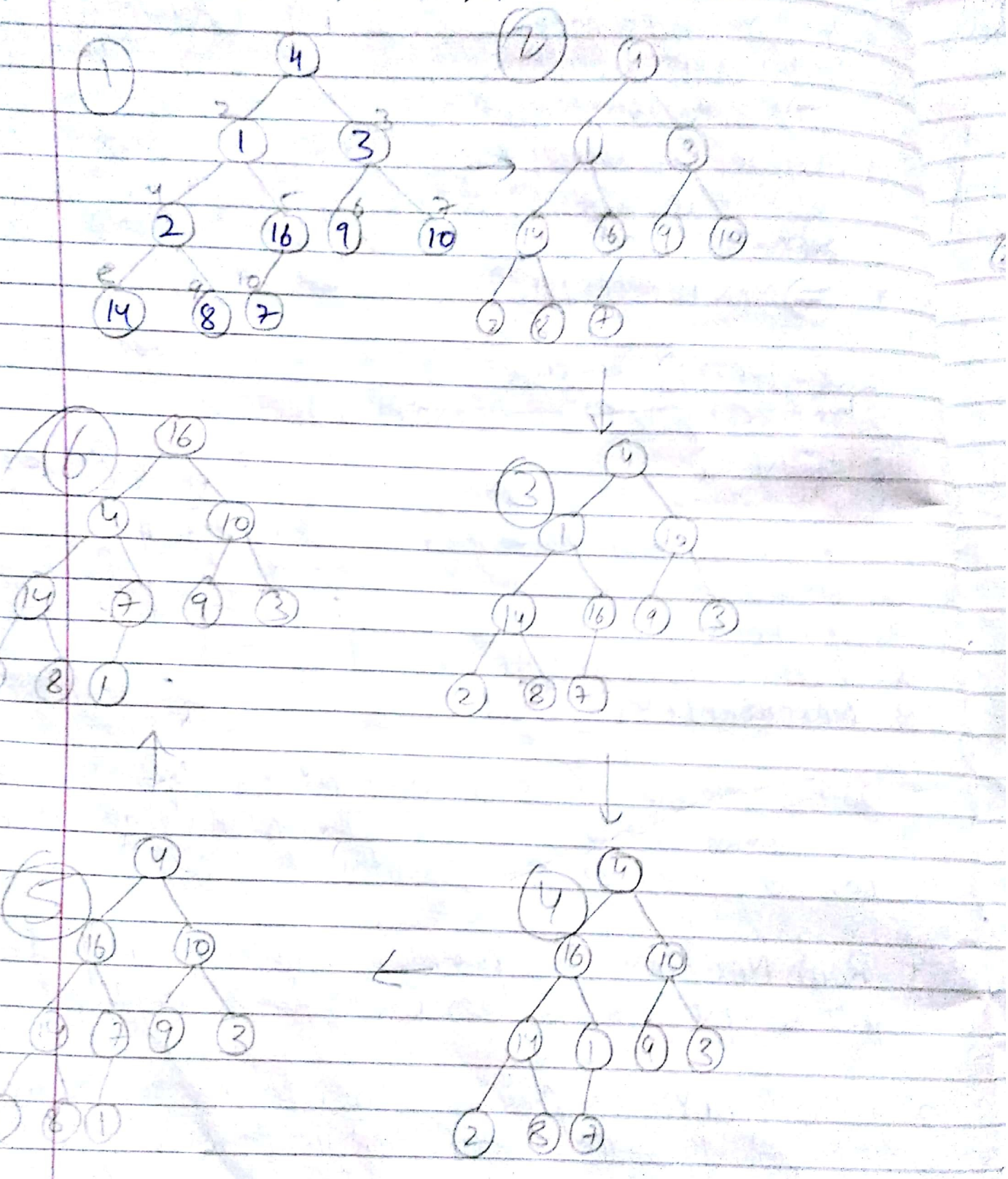
In this algorithm, A is an unsorted array which is to be sorted using heap sort. The following procedures are being used in this algorithm.

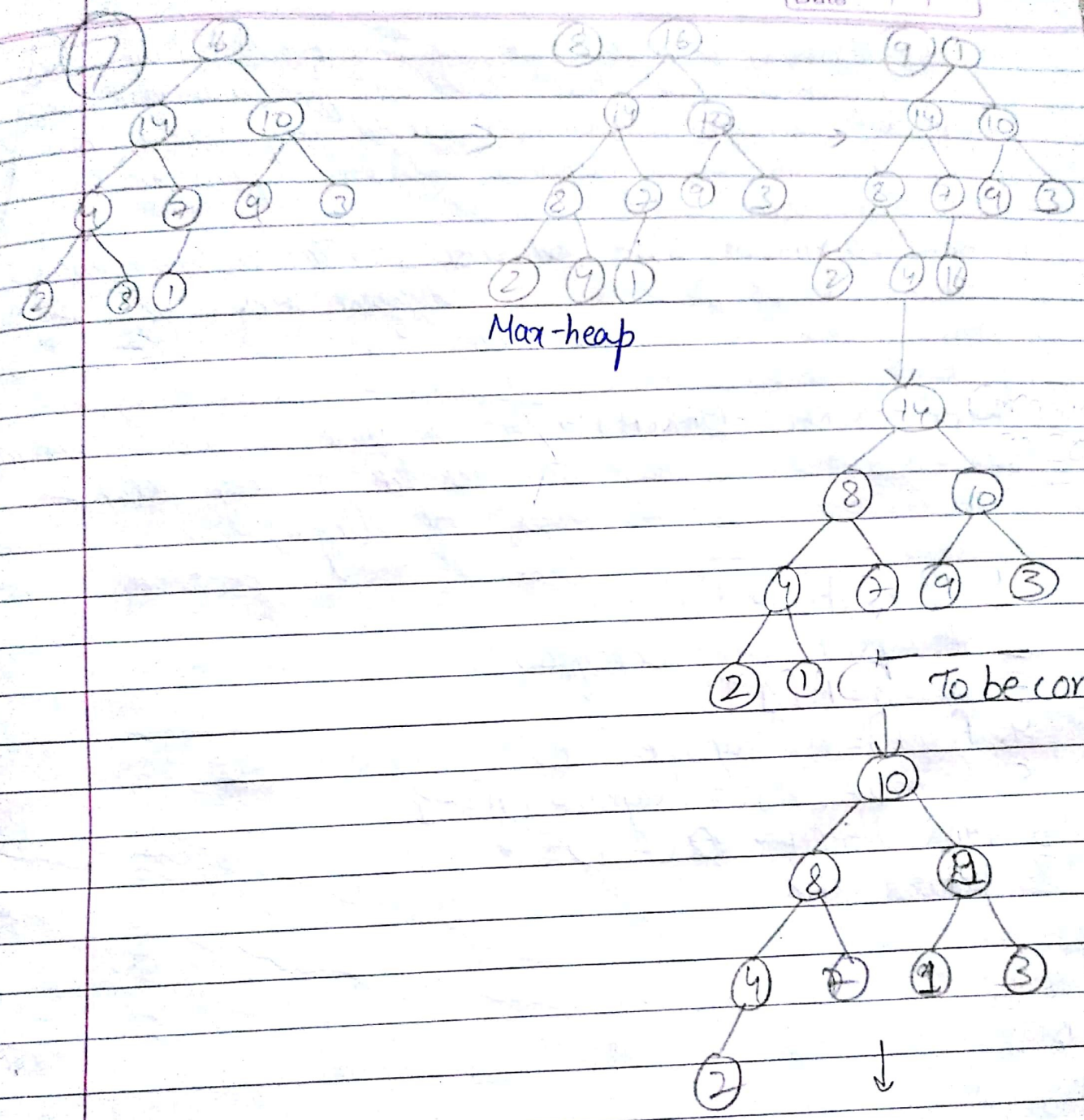
1 BUILD MAX-HEAP :- This procedure produces max heap from an unsorted input array & takes O(n) running time

2. MAX-HEAPIFY :- This procedure is used to maintain the max-heap property & the running time is the order of O(log n)

3. HEAP-SORT :- This procedure sorts an array & time complexity is of the order of O(n log n)

Example: A = 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.





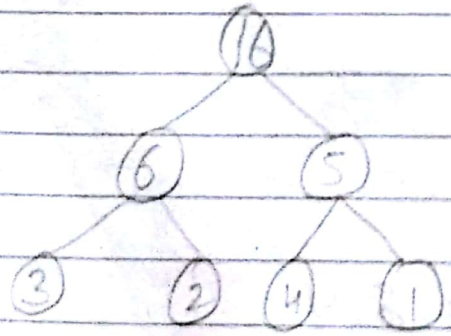
Qu 2 5, 13, 2, 25, 7, 17, 20, 8, 4.

Applications of Heap Sort : Priority Queue :→ A priority queue is a data structure that is used for maintaining a set S of element with an associated key value. A max priority queue supports following operations :-

1) HEAP-MAXIMUM (S):- returns S[1] :- This procedure returns the element of S with the largest key value & running time is of the order of O(1)

2) HEAP-EXTRACT-MAX(A) :- This procedure removes & return the element of array with the largest key value & running time is of the order of O(log n).

1. If heapsize [A] < 1
2. then print "Heap Underflow".
3. max := A[1]
4. A[1] := A[heapsize [A]]
5. heapsize [A] := heapsize [A] - 1
6. MAX-HEAPIFY (A, 1)
7. return MAX.



3 HEAP-INCREASE-KEY (A, i, key) :- In this procedure, the value of element's key is increased to a new value which is assumed to be as large as the current key value. Here, :-
i- position of the element at which key is used to increased.

A= is an array & key is the value which is to be increased & running time is of the order of O(log n).

1. If key < A[i]
2. then print "New value is smaller than old value" & Exit.
3. A[i] := key
4. while (i>1) and (A[parent[i]] < A[i])
5. Exchange A[i] & A[parent[i]]
6. i := parent[i].

4. MAX-HEAP-INSERT (A, key) :→ This algorithm inserts a new element into the existing set of element which is equivalent to following operations :-

$$A = A \cup \{\text{key value element}\}$$

& the running time is of the order of O(log n)

1. heapsize = heapsize[A] + 1
2. A[heapsize[A]] = $-\infty$
3. HEAP-INCREASE-KEY (A, heapsize[A], key).

A= is an array & key is the new value that is to be inserted in the array.

DAA - Lecture (LAB)

Quick Sort :→

```
Algorithm quick sort (p, q)
1 {If (p<q) then // If their are more than 1 element
  {
2.  j := Partition (a, p, q+1); // Divide 'p' into subproblems
    // 'j' is the position of partitioning element.
3.  Quicksort (p, j-1); // solve the subproblem
4   Quicksort (j+1, q); }}

Algorithm Partition (a, m, b)
1  v := a[m], i := m, j := b
2  repeat {
3  repeat {
4  i = i+1;
5  until (a[i] ≥ v); }
6  repeat {
7  j := j-1;
8  until (a[j] ≤ v); }
9  If (i<j)
10 then interchange (a, i, j);
11 } until (i≥j);
12 a[m] := a[j];
   a[j] := v;
   return j; }
```

i	j
2	9
3	8
4	7
5	6
6	5
4	4

do

a(1) i=1 j=

Algorithm interchange (a, i, j)

```
1  { temp := a[i];
2    a[i] := a[j];
3    a[j] := temp;
   }
```

Running Time = (n log n)

Example:-

(1) i	(2)	(3)	(4)	(5) k	(6)	(7)	(8)	(9)	(10) j
65	70	75	80	85	60	55	50	45	∞

Quick Sort :- This sorting technique is based on divide & conquer approach. In this array A[1-n] is divided into 2 subarrays which are independently sorted & the division is made in such a way so that the sorted sub array need not to be merge later on.

1/10

In this algorithm :-

p = is the index of first element
q = " " " " last "
j = " " partitioning point.

Partition procedure is used to divide the list into two sub-arrays & interchange procedure is used to swap the values of i & j.

65 45 75 80 85 60 55 50 70 ∞

65 45 50 80 85 60 55 75 70 ∞

65 45 50 55 85 60 80 75 70 ∞

65 45 50 55 60 85 80 75 70 ∞

60 45 50 55 65 85 80 75 70 ∞

55 45 50 60 65 85 80 75 70 ∞

50 45 55 60 65 85 80 75 70 ∞

45 50 55 60 65 85 80 75 70 ∞

45 50 55 60 65 70 80 75 85 ∞

45 50 55 60 65 70 75 80 85 ∞

Example:- 4, 13, 10, 9, 8, 1, 2, 5

Analysis of Time Complexity of Quick Sort:-

Let $C_A(n)$ denotes the no. of comparisons that are req. to sort an unsorted array A having n no. of elements. As no comparison is required to sort NIL or single element. Therefore, $C_A(0) = C_A(1) = 0$

$$C_A(n) = (n+1) + \frac{1}{n} \sum_{1 \le k \le n} \left[C_A(k-1) + C_A(n-k) \right] \quad \text{--- (1)}$$

k: L.L, $n-k$: R.L

Multiply (1) with 'n'

$$n C_A(n) = n(n+1) + \sum_{1 \le k \le n} [C_A(k-1) + C_A(n-k)] \quad (2)$$

Expanding the value of 'k' from 1 to n in eq" (2)

$$n C_A(n) = n(n+1) + [\{C_A(0) + C_A(1) + \dots + C_A(n-1)\} + \{C_A(n-1) + C_A(n-2) + \dots + C_A(0)\}]$$

$$n C_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \dots C_A(n-1)] \quad (3)$$

Replace 'n' with 'n-1' in eq" (3)

$$(n-1) C_A(n-1) = n-1[n-1+1] + 2[C_A(0) + C_A(1) + \dots + C_A(n-2)] \quad (4)$$

(3) − (4)

$$n C_A(n) - (n-1) C_A(n-1) = n^2 + n - n^2 + n + 2 C_A(n-1)$$

$$n C_A(n) - (n-1) C_A(n-1) = 2n + 2 C_A(n-1) \quad (5)$$

~~$n C_A(n) - (n-1) C_A(n-1)$~~

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

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

$$n C_A(n) = 2n + (n+1) \cdot C_A(n-1) \quad (6)$$

Divide (6) with $n(n+1)$

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

$$\Rightarrow \frac{C_A(n)}{n+1} = \frac{2}{n+1} + \frac{C_A(n-1)}{n} \quad -(7)$$

Replace 'n' with 'n-1' in (7)

$$\frac{C_A(n-1)}{n} = \frac{2}{n} + \frac{C_A(n-2)}{n-1} \quad -(8)$$

Substitute (8) in (7)

$$\frac{C_A(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{C_A(n-2)}{n-1} \quad -(9)$$

Replace 'n' with (n-2) in (7)

$$\frac{C_A(n-2)}{n-1} = \frac{2}{n-1} + \frac{C_A(n-3)}{n-2} \quad -(10)$$

Substitute (10) in (9)

$$\frac{C_A(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{C_A(n-3)}{n-2} \quad -(11)$$

Expanding the above eq^n upto 'n-1' terms

$$\frac{C_A(n)}{n+1} = 2\sum_{2}^{n+1}\frac{1}{K} + \frac{C_A(0)}{n-(n-1)}$$

As $C_A(0) = 0$

$$\frac{C_A(n)}{n+1} = 2\sum_{2}^{n+1}\frac{1}{K} \quad \text{---} (12)$$

The summation term can be expressed in integral form.

$$\frac{C_A(n)}{n+1} = 2\int_{2}^{n+1}\frac{1}{K}\,dK$$

$$\frac{C_A(n)}{n+1} = 2(\log K)_2^{n+1} = 2|\log(n+1) - \log 2|$$

$$C_A(n) = 2(n+1)[\log(n+1) - \log 2]$$

$$C_A(n) = 2n\log(n+1) - 2n\log 2 + 2\log(n+1) - 2\log 2$$

Ignoring the constant value & smaller powers of 'n'

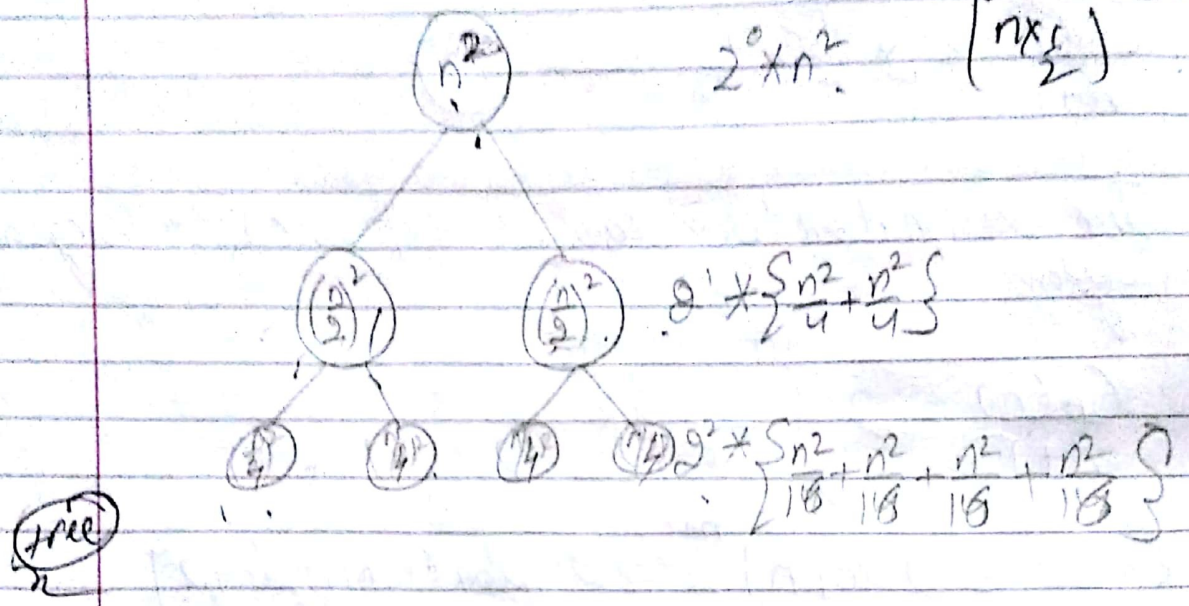
$$C_A(n) \gtrsim O(n\log n)$$

Recursion Tree :- It is the pictorial representation of alteration method in which, the eqn is represented is in the form of tree & each level of the nodes are expanded. In this method, the second term of recurrence relation is considered as root.

This method is useful when divide & conquer algorithm is

used.

Example:- $T(n) = 2T\left(\frac{n}{2}\right) + n^2$



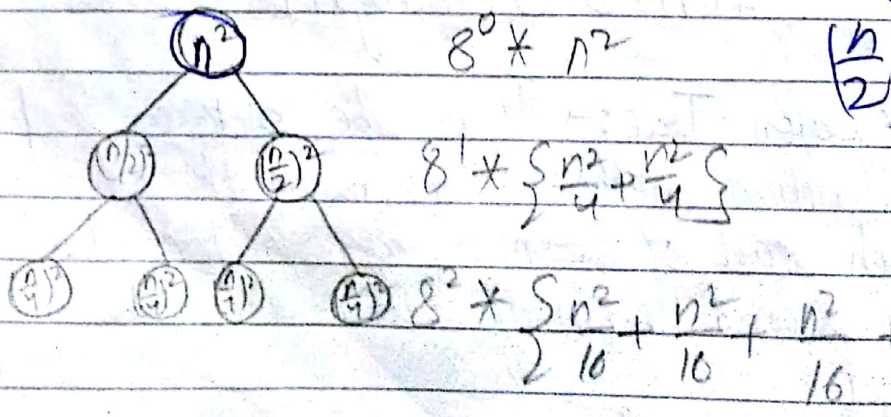
$= n^2 + \frac{4n^4}{4} + 4 * \frac{4n^2}{16}$ $= \{n^2 + n^2 + n^2 + \cdots\}$ $\log n$ time

$\in n^2 \log n$. where $\log n$ is the height of tree.

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

$n^2 \left(\frac{1}{2} \times n\right)$

$\left(\frac{n}{2}\right)^2$



$= n^2 + 8 \times \frac{2n^2}{4} + 64 \times \frac{4n^2}{16} + \cdots$

$= n^2 + 4n^2 + 16n^2 + \cdots$

$= n^2 [1 + 4 + 16 + \cdots]$

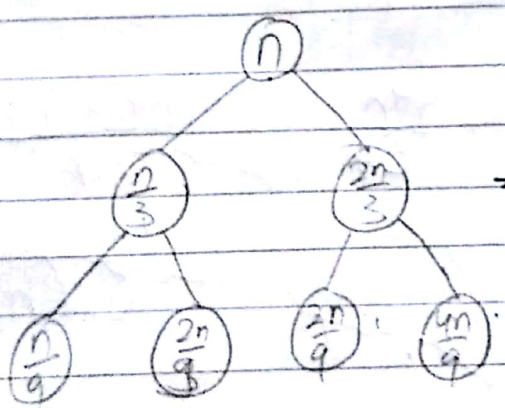
Infinite G.P series :- $\frac{a}{1-r}$

$a = 1$

$r = 4$

$= n^2 \left\{ \frac{1}{1-4} \right\} = \frac{-n^2}{3} \Rightarrow O(n^2) \log n.$

Example:- $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$



$\rightarrow \frac{2}{3} \times \frac{2n}{3}$

$= \{n + n + n + \cdots\} \log n \Rightarrow O(n \log n)$

Example:- $T(n) = 2T\left(\frac{n}{3}\right) + 3T\left(\frac{2n}{3}\right) + n$

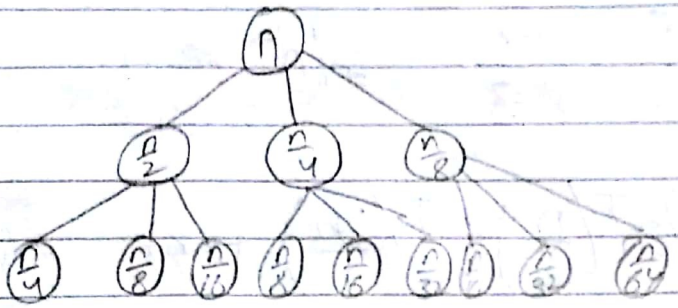
Same as above diagram

$\Rightarrow n$

$\Rightarrow 2^1 * \left(\frac{n}{3} + \frac{2n}{3}\right) + 3^1 * \left\{\frac{n}{3} + \frac{2n}{3}\right\}$

$\Rightarrow 2^2 * \left\{\frac{n}{9} + \frac{2n}{9} + \frac{2n}{9} + \frac{4n}{9}\right\} + 3^2 \left\{\frac{n}{9} + \frac{2n}{9} + \frac{2n}{9} + \frac{4n}{9}\right\}$

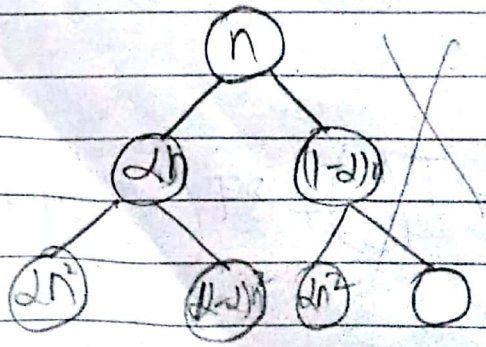
Example:- $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$

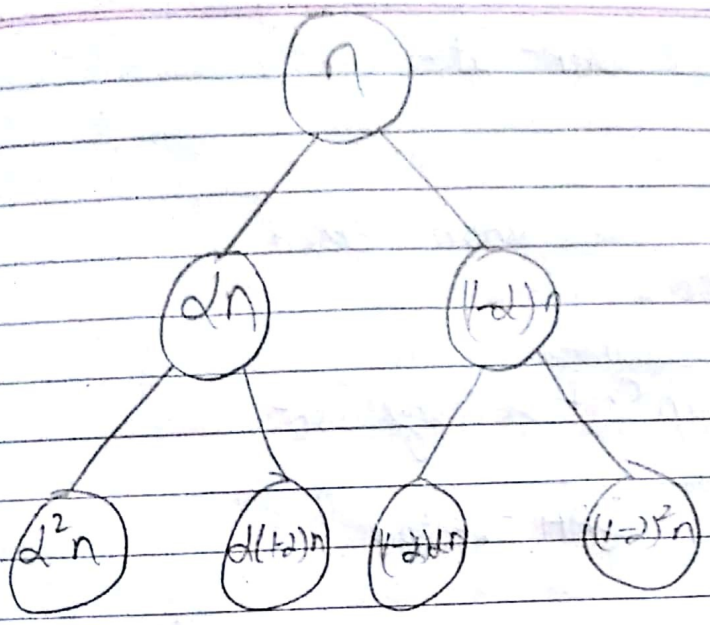


$= n + \frac{3n}{4} + \frac{9n}{16} + \cdots = \left(n + \frac{7n}{8} + \frac{49n}{64} + \cdots\right) \Rightarrow \left(\frac{64n + 56n + 49n + \cdots}{64}\right)$

$= \left(\frac{169n}{64} + \cdots\right) \log n = O(n \log n)$

Example:- $T(n) = T(\alpha n) + T(1-\alpha)n + n$





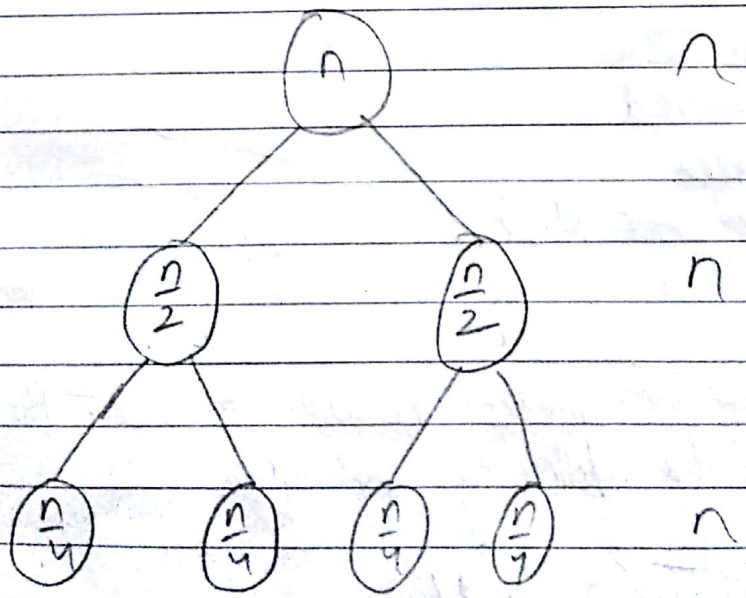
$\alpha n + n - \alpha n$

$\alpha^2 n + \alpha n - \alpha^2 n + \alpha n - \alpha^2 n + n + \alpha^2 n - 2\alpha n$

$= (n+n+n+\ldots\ldots)$ log n times

$= O(n \log n)$

Example:- $4T\left(\frac{n}{2}\right) + n$



$= (n+n+n+\ldots\ldots)$ log n times

$= O(n \log n)$

DAA-Tut:-

Recurrence Tree
Master Method, Substitution Method
Asymptotic Notations
Time Complexity Questions.
Quick Sort, Heap Sort & Merge Sort

DAA - Lecture.

Recurrence Relation:- When an algorithm calls itself. It is known as recursive call & its running time can be expressed in form of a recurrence relation.

A recurrence can be defined as an eqⁿ of inequality that described a function in terms of its smaller inputs. Following are the methods to solve a recurrence relation:-

1) Master Method
2) Substitution Method.
3) Recurrence Trees
4) Change of variable Method.
5) Iteration method

Master Method:- This method is used to solve the recurrence relation for the following eqⁿ types:-

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$ & $b > 1$

$f(n) =$ any non-ive function of n.

2) The above recurrence relation describes the problem of size 'n' which is divided into a subproblems each of size $\left(\frac{n}{b}\right)$.

3) The 'a' subproblems are solved recursively in time $T\left(\frac{n}{b}\right)$. The cost of dividing the problems & combining the result of subproblems is described by function $f(n)$.

4) $T(n)$ can be asymptotically bounded in the following 3 cases:-

Case 1:- If $f(n) = O(n^{\log_b a - \epsilon})$ where $\epsilon > 0$

then $T(n) = O(n^{\log_b a})$

Case 2:- If $f(n) = \Theta(n^{\log_b a})$

then $T(n) = \Theta(n^{\log_b a} \log n)$

Case 3:- If $f(n) = \Omega(n^{\log_b a + \epsilon})$

then $T(n) = \Theta(f(n))$

Example:- $T(n) = 4T\left(\frac{n}{2}\right) + n$

Step-1 $a = 4, b = 2, f(n) = n$

Compute $n^{\log_b a}$

$n^{\log_2 4} = n^{\log_2 2^2} = n^{2\log_2 2} = n^2$

$\log m^n = n \log m$
Same base power same = 1

Step 2:- Compare value of $n^{\log_b a}$ & $f(n)$

$n^{\log_b a} = n^2$ & $f(n) = n$

$n^{\log_b a - \epsilon} = n^{2-1} = n = f(n)$

Case 1 is satisfied

$T(n) = \Theta(n^{\log_b a}) = n^2$ $\quad T(n) = \Theta(n^2)$

Example 2:- $T(n) = 8T\left(\frac{n}{2}\right) + n^3$

Step 1 $\quad a = 8, \; b = 2 \quad f(n) = n^3$

Compute $n^{\log_b a}$

$= n^{\log_2 8} = n^{\log_2 2^3} = n^{3\log_2 2} = n^3$

Step 2:- Compare value of $n^{\log_b a}$ & $f(n)$

$n^{\log_b a} = n^3$ $\quad$ & $f(n) = n^3$

$n^{\log_b a} = f(n) \Rightarrow$ Case 2 is satisfied

$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^3 \log n)$

Example 3:- $T(n) = 2T\left(\frac{n}{4}\right) + n^2$

$a = 2 \quad b = 4 \quad f(n) = n^2$

Compute $n^{\log_b a}$

$n^{\log_4 2} = n^{\log_4 4^{1/2}} = n^{\frac{1}{2}\log_4 4} = n^{1/2}$

Compare value of $n^{\log_b a}$ & $f(n)$

$n^{\log_b 4} = n^{1/2}$ & $f(n) = n^2$

$n^{\log_b a + \epsilon} = n^{\frac{1}{2}+\frac{3}{2}}$

Case 3 is satisfied

$T(n) = \Theta(f(n)) = \Theta(n^2)$

Example 4:- $T(n) = 2T\left(\frac{n}{4}\right) + 1$ (same as above)

Example 5:- $T(n) = 3T\left(\frac{n}{5}\right) + n$

$a = 3, \; b = 5, \; f(n) = n$

$n^{\log_b a} = n^{\log_5 3} = n^{\frac{\log 3}{\log 5}} \simeq n^{0.4}$

$T(n^{\log_b a}) =$

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

Substitution Method:- This method is used to make the repeated substitution for each occurence of $T(n)$ in the right hand side until all such occurences disappear.

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

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

Put $n \Rightarrow \frac{n}{2}$ in (1)

$$T(n/2) = 2T\left(\frac{n}{4}\right)+\frac{n}{2} \quad -(2)$$

Put (2) in (1)

$$T(n) = 2\left[2T\left(\frac{n}{4}\right)+\frac{n}{2}\right]+n$$

$$T(n) = 4T\left(\frac{n}{4}+2n\right) \quad -(3)$$

Put $n \Rightarrow \frac{n}{4}$ in (3)

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right)+\frac{n}{4} \quad -(4)$$

Put (4) in (3)

$$T(n) = 4\left(2T\left(\frac{n}{8}\right)+\frac{n}{4}\right)+2n$$

$$= 8T\left(\frac{n}{8}\right)+3n \quad -(5)$$

Expanding the above relation to 2^k term

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

Let $n = 2^k \Rightarrow k = \log_2 n$

$T(n) = nT\left(\frac{n}{k}\right) + n\log_2 n$

$T(n) = nT(1) + n\log_2 n$

$T(n) = n + n\log_2 n$

$$T(n) = \begin{cases} \Theta(n) & \text{if } n \leq 10 \\ \Theta(n\log_2 n) & \text{if } n > 10 \end{cases}$$

Example 2:- $$T(n) = \begin{cases} \Theta(1) & \text{, if } n = 1 \\ 2T\left(\frac{n}{2}\right) + 1 & \text{if } n > 1 \end{cases}$$

Example:- $T(n) = 2T\left(\frac{n}{2}\right) + 1$

$$T(n) = 2T\left(\frac{n}{2}\right) + 1 \quad \text{— (1)}$$

Put $n \to n/2$ in (1)

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 1 \quad \text{— (2)}$$

Put (2) in (1)

$$T(n) = 2\left\{2T\left(\frac{n}{4}\right) + 1\right\} + 1$$

$$T(n) = 4T\left(\frac{n}{4}\right) + 3 \quad \text{— (3)}$$

Put $n \to n/4$ in (1)

$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 1$ — (4)

Put (4) in (3)

$T(n) = 8T\left(\frac{n}{8} + 7\right)$ — (5)

⋮

Expanding it upto 2^k terms.

$T(n) = 2^k T\left(\frac{n}{2^k}\right) + C \quad C > 0$

Let $n = 2^k$

$k = \log_2 n$

$T(n) = nT\left(\frac{n}{n}\right) + C = nT(1) + C$

$T(n) \approx \Theta(n)$

Merge Sort:- This algorithm is based on divide & conquer approach which states that:-

1) Divide:- It divides the n elements sequence to be sorted into subsequences of $\frac{n}{2}$ elements each.

2) Conquer:- Sort the two subsequences recursively using merge sort.

3) Combine:- Merge the two sorted subsequences to produce the final sorted array.

It follows bottom up approach & is divided till an array of one element is obtained.

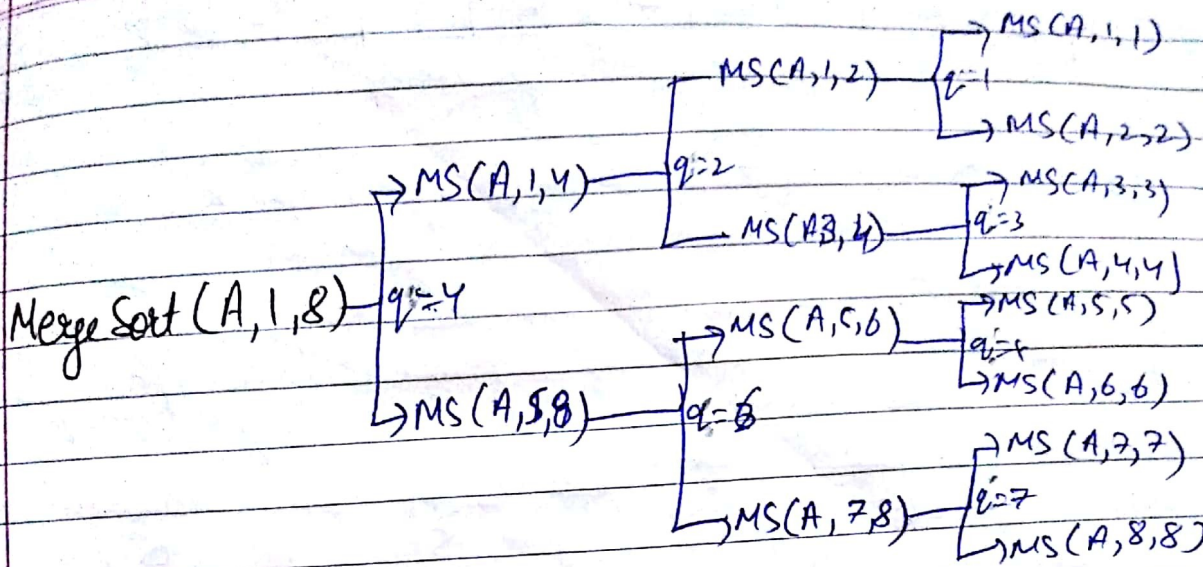
Algorithm :→

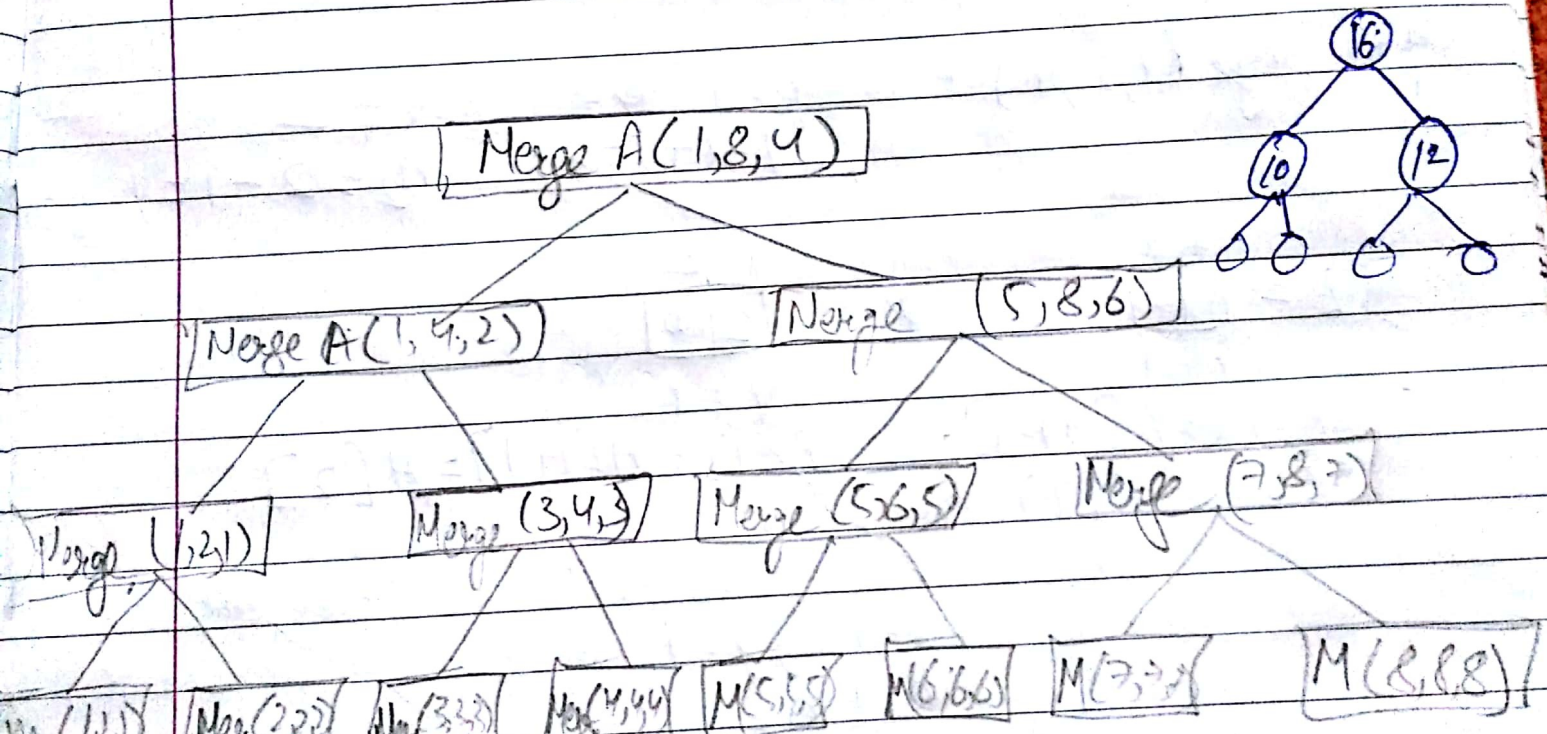
Algorithm Merge Sort (A, p, r)

1. If (p < r)
2. q = ⌊(p+r)/2⌋
3. Merge-SORT (A, p, q)
4. MERGE-SORT (A, q+1, r)
5. MERGE (A, p, q, r)

Algorithm Merge (A, p, q, r)

1. $n_1 := q-p+1$, $n_2 := r-q$
2. Let L(1 ---- n_1+1) & R(1 ---- n_2+1) be new arrays
3. for i := 1 to n_1
4. L[i] = A[p+i-1]
5. for j := 1 to n_2
6. R[j] := A[q+j]
7. $L[n_1+1] = \infty$, $R[n_2+1] = \infty$
8. i := 1 , j := 1
9. for k := p to r
10. If (L[i] ≤ R[j])
11. A[K] = L[i]
12. i := i+1
13. else A[K] := R[j]
14. j := j+1

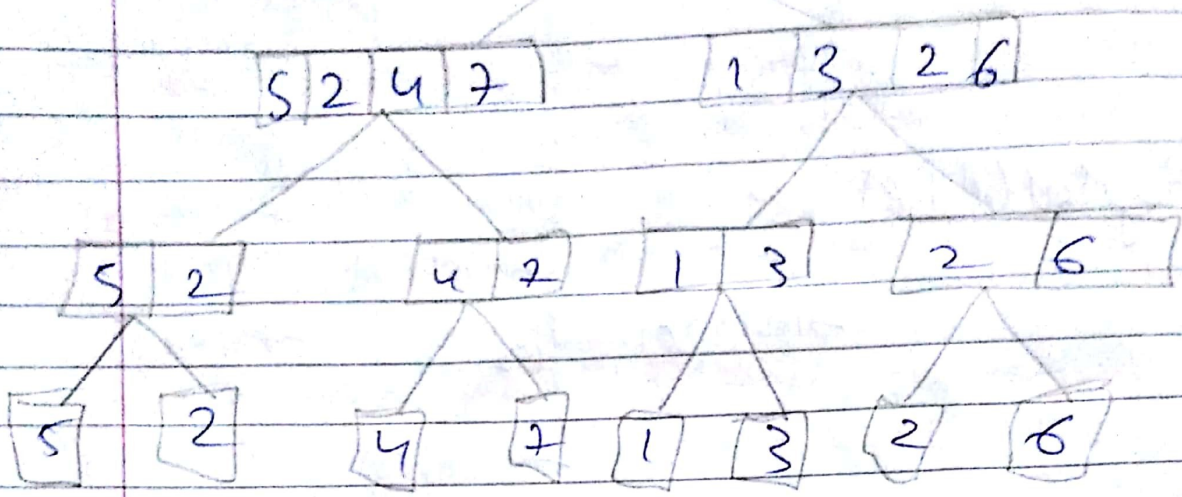




Here P = index of first element
r = " " last "
q = the middle element index at which the list will be partitioned.
L & R = are left & right subarrays respectively

Step-1 :- A =

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



Step-2 Merge (A, 1, 2, 1) = $p=1$ $q=1$ $r=2$

$n_1 = 1-1+1 = 1$ $n_2 = 2-1 = 1$

$L =$ | 5 | ∞ | $R_2 =$ | 2 | ∞ |

$i=1$ $j=1$

$L[1] = A[1+1-1]$. $R[1] = A[1+1] = A[2]$

$L[1] = A[1]$

$i=1$, $j=1$

$k=1$ to 2

$A :=$ | k | 5 | $k=1$ $i=2$ $k=2$

Step-3 Merge (A, 3, 4, 3)

$p=3$ $q=4$ $q=3$

$n_1 = 3-3+1 = 1$ $n_2 = 4-3 = 1$

$L :=$ | 4 | ∞ | $R :=$ | 7 | ∞ |

Step 4 of 12

Merge (A, 1, 4, 2)

p = 1 r = 4 q = 2

$n_1 = 2 - 1 + 1$

$n_2 = 4 - 2 = 2$

i = 1 to 2

j = 1 to 2

L: | 2 | 5 | ∞ |

R: | 4 | 7 | ∞ |

A: | 2 | 4 | 5 | 7 |

i