DAA - Provide techaique, using those technique helps to design algorithm

classmate
Date
Page

Course Title : Design and Analysis of Algorithms

Course no : CSC-303

Credit hours : 3

Full Marks : 80+20

Pass Marks : 32+8

Course Content

UNIT 1: -10 Hrs.

1.1 Algorithm Analysis : worst, best and average cases, space and time complexities.

Mathematical background: asymptotic behaviour, solving recurrences.

1.2 Data structures Review: linear data structures, hierarchical data structures, data structures for representing graphs and their properties. Search structures: heaps, balanced trees, hash tables.

UNIT 2: -14 Hrs.

2.1 Divide and Conquer: Concepts, applications, sorting problems (quick, merge), searching (binary), median finding problem and general order statistics, matrix multiplications

2.2 Greedy Paradigm : Concepts, applications, knapsack problem, job sequencing, Huffman codes

2.3 Dynamic Programming : Concepts, applications, knapsack problem, longest common subsequence, matrix chain multiplications.

36

UNIT 3: - 21 Hrs.

3.1 Graph Algorithms: breadth-first and depth-first search and their applications, minimum spanning trees (Prim's and kruskal's algorithms), shortest path problems (Dijkstra's and flyod's algorithms), algorithm for directed acyclic graphs (DAGs). 19

3.2 Geometric Algorithms: Concepts, plolygon triangulation, Convex hull computation 11

3.3 NP Completeness: Introduction, Class P and NP, Cooks theorem, NP Complete problems: vertex cover problem. 6

3.4 Introductions: Randomized algorithms concepts, randomized quick sort, approximation algorithms concepts, vertex cover problem.

Textbook: "Introduction to Algorithms", 2nd Edition.
Reference: "Fundamentals of Algorithmics (G. Brassard and P. Bratley

Total questions: '10', each question cover 8 marks.

UNIT: 1

1.1) ALGORITHM ANALYSIS

Algorithm:

An algorithm is clearly specified step of instructions to solve the problem. Algorithm takes some values or set of values as input and produce some value or set of values as output. An algorithm is thus sequence of computational steps that transform the input into output.

A finite set of instructions that specify the sequence of operation to be carried out inorder to solve a specific problem or class of problem is called an algorithm. Algorithm are not dependent on particular machine, programming language or compiler i.e. algorithm run in same manner everywhere. So algorithm is a mathematical object where the algorithm are assured to be run under the machine with unlimited capacity.

Properties of Algorithm:

An algorithm must have following properties:

1) Input/ Output:

For an algorithm there must be some input from standard set of input and algorithm must produce some output on execution with that input.

ii) Definiteness: Step ko meaning clear hunuparxa.

Definiteness means each step must be clear and unambiguous. According to this property, each operation must be definite meaning that it must be perfectly clear what should be done.

iii) Finiteness:

Finiteness means algorithm must complete after the finite number of step.

iv) Correctness:

The algorithm must produce correct output with correct set of input i.e. an algorithm is correct if for every input instance it halt with correct output.

v) Effectiveness:

The algorithm must be effective to solve particular problem for which algorithm is designed. So, effectiveness property requires that each operation be effective i.e. each step must be carried out in finite amount of time.

Expressing Algorithm:

There are many ways of expressing algorithms. An algorithm can be specified in:

i) Natural language like english
ii) Pseudo code
iii) Real programming language syntax.

The only requirement on these expressing technique of algorithm is that the specification must provide a precise description of the computational procedure to be followed.

(2 marks) LV

Why we study algorithm?

We study an algorithm to understand the basic concept of computer science programming where the computation are done. We also study algorithm to effectively solve the problem and to understand input-output relation of the problem. We must be able to understand the step involved in getting output from the given input by studying the algorithms.

Algorithm Design:

Algorithm design is a specific method to create mathematical process in problem solving. The development of algorithm is a key step in solving problem. Once we have an algorithm, we can translate it into a computer program in some programming language. The algorithm development process consist of five major steps:

i) Obtain the description of the problem
ii) Analyze the problem
iii) Develop a high level algorithm
iv) Redefine the algorithm by adding more detail
v) Review the algorithm

Algorithm Design Techniques:

For a given problem, there are many ways to design algorithm for it. The following is a list of some popular design approach.

1) Divide and Conquer:

These are the methods of designing an algorithm that proceeds as follows:

- Given an instance of the problem to be solved, split these into several smaller sub instances of the same problem, solve the sub instance recursively and then combine the sub instance solution so as to yield the solution for the original instance.

For example: binary search, merge sort, quick sort, etc.

2) Dynamic Programming:

One disadvantage of using divide and conquer is that the process of recursively solving specific sub problems can result in the same computation being performed repeatedly since identical sub problem may arise. The dynamic programming avoid this problem by maintaining a table of sub problem results.

Dynamic Programming is bottom-up technique in which smallest sub problem are explicitly solved first and result of these sub problems are used to construct solution of large sub problem.

For example: Longest common sub sequence problem can be solved by designing the algorithm using dynamic programming approach.

3) Greedy approach:

In greedy approach, solution to the problem is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far. Each step the choice must be locally optimal, this is the central part of this technique.

Greedy techniques are mainly used to solve optimization problem. They donot always give the best solution.
Example: The problem of finding the MST using greedy approach.

2074(2) LE

Algorithm Analysis:

Analyzing an algorithm means predicting the resources that the algorithm require. Occassionally, resources such as memory, communication bandwidth or computer hardware are of primary concerned but often it is computational time that we want to measure.

The analysis of algorithm gives a good insight of the algorithm under study. Analysis of algorithm tries to answer few questions like is the algorithm correct ie. algorithm generates the required result or not? Does the algorithm terminates for all inputs under problem domain. The other issues of analysis are efficiency, optimality, etc.

Generally by analyzing the algorithm (knowing the different aspect of algorithms) on the smaller problem domain we can choose the better algorithm for our need. This can be done by knowing the resource needed for the algorithm to its

execution. The two most important resources are time and space.

The amount of time any algorithm takes almost always depends on the input size it must process i.e. the time taken by the algorithm grows with the size of input so running time of algorithm is described as a function of the size of input.

The analysis of algorithm provides general idea of how long an algorithm will take for a given problem set. For example: we might determine how many comparisions a sorting algorithm does to put list of n-values in ascending order during the analysis of sorting algorithm. There are number of algorithms that will solve a same problem so studying the analysis of algorithm gives us a tool to choose between algorithms.

fair designing ek choti ma cuta

to maintain standard of analysis of algorithms

Random Access Machine Model (RAM Model):

RAM Model is a generic one processor model of computation that we use for our study of design and analysis of algorithm in machine independent scenario. In RAM model, instructions are executed one after another with no concurrent operations. The RAM model contains instructions commonly found in real computer such as arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, substructure call and return).

This model assumes all algorithm must be implemented in a single processor computer and each primitive

euta processor hunxa, ekchoti ma euta statement execute hunxa,
assume garekko comp., jun comp. use garera fair analysis hunxa.

operations takes one step. Examples of primitive operations are assigning value to variable, performing arithmetic operation, indexing into an array, comparing two numbers, calling a method, returning from method and memory reference. Loops and subroutines are not primitive operations. In this model we can measure the runtime of algorithm by counting the steps. The following figure shows RAM model.

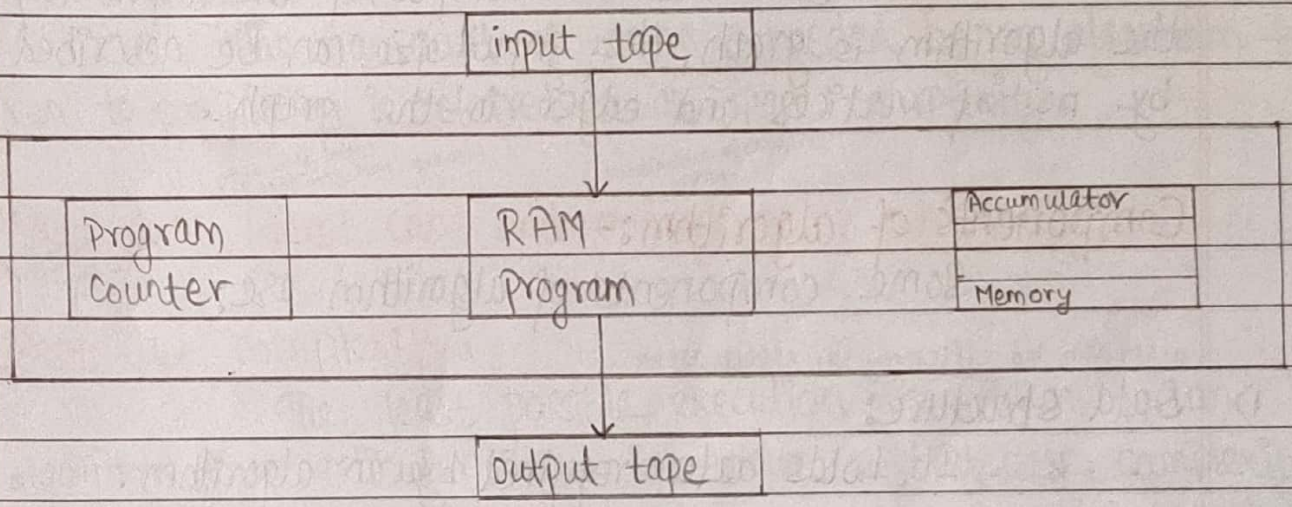


Fig: RAM Model

How to calculate running time of algorithm in RAM model:

We can calculate the running time of an algorithm by running the implementation of algorithm on RAM model. The running time of an algorithm on a particular input is the no. of primitive operation or steps executed. The running time of an algorithm grows with the size of input so it is better to describe running time of algorithm as a function of size of input.

Input size:

The input size depends on the problem being studied. For many problem such as sorting, input size means no. of items in the input. For other problem such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in binary notation. Sometimes size of input is described with two numbers rather than one. For example, if the input to the algorithm is graph, the input size can be described by no. of vertices and edges in the graph.

Components of algorithm:-

Some components of algorithm are:

algorithm ko efficiency lai effect garxa

1) Data structure:
   It holds data required by an algorithm.

2) Data Manipulation instruction:
   They allow for changes in data values.

3) Conditional Expressions:
   They are used to make decisions.

4) Control structures:
   They are used to act on decisions.

5) Modules:
   They are used for making abstractions.

time: taken by algorithm to complete job
space/memory: taken by algorithm to complete job

classmate
Date
Page

Time and Space Complexity of an algorithm:-

Time complexity referred to as amount of time required by an algorithm to run to completion. For an algorithm, time complexity depends upon the size of input, thus it is the function of input size 'n'. It should be noted that different time can arise for same algorithm. But usually deal with the best case time, average case time and worst case time for an algorithm.

The amount of memory needed by an algorithm to run to completion is referred to as space complexity.

tyo vanda badhi complexity hunai sakdaina

combination of best & worst

input ko instance ko lagi

tyo vanda lower time ma complete garna sakdaina vane best case comp.

Best case, Worst case and average case complexity

2067
2071

Best case complexity:

The least possible execution time taken by an algorithm for particular input is known as best case complexity. Best case complexity gives lower bound on running time of the algorithm for any instance of input. This indicates that the algorithm can never have lower running time than best case complexity for particular class of problem.

Worst case complexity:

The maximum possible execution time taken by an algorithm for particular input is known as worst case complexity. Worst case complexity gives an upper bound on the running time of algorithm for all instance of the input. This ensures that no input can overcome the running time limit posed by worst case complexity.

Average case complexity:

Average case complexity gives the average time required by an algorithm to solve the problem.

Mathematical foundation:-

Since mathematics can provide clear view of an algorithm so understanding the concept of mathematics is aid in the design and analysis of good algorithms. The following are some mathematical concepts that are helpful in our study.

1) Exponent:

Some of the formulas that are helpful are:

i) $x^a x^b = x^{a+b}$

ii) $x^a / x^b = x^{a-b}$

iii) $(x^a)^b = x^{ab}$

iv) $x^n + x^n = 2x^n$

v) $2^n + 2^n = 2^{n+1}$

2) Logarithms:

Some of the formulas that are useful are:

i) $\log_a b = \log_c b / \log_c a$ for all real $a>0, b>0, c>0$.

ii) $\log_c(ab) = \log_c a + \log_c b$

iii) $\log_c(a/b) = \log_c a - \log_c b$

iv) $\log a^b = b \log a$

v) $\log x < x$ for all $x>0$.

vi) $\log_2 1 = 0$, $\log_2 2 = 1$, $\log_2 1024 = 10$

vii) $a \log_b n = n \log_b a$

3) Series:

General series

i) $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

$$\boxed{\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}}$$

ii) $\sum_{i=0}^{n} a^i = \frac{1}{1-a}$ if $|a| < 1$.

iii) $\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$ if $a \neq 1$

iv) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

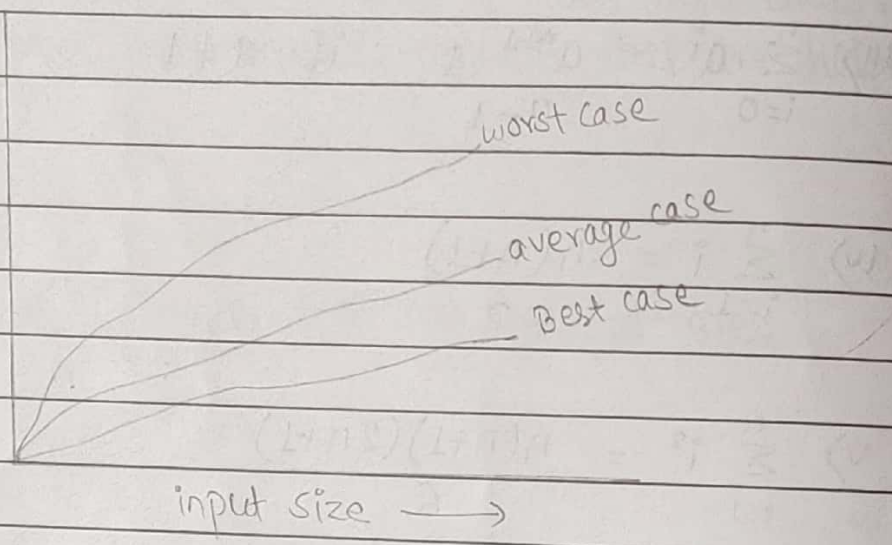
v) $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

vi) $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$

vii) $\sum_{i=1}^{n} 1/i = \log_e n$

Asymptotic Notations:

Complexity analysis of an algorithm is very hard if we try to analyze exact. We know that time complexity of an algorithm is the mathematical function of size of input. so if we analyze the algorithm in terms of bound then it would be easier. For this, we need asymptotic notation. Figure below shows upper and lower bound concept.



Why are asymptotic notations important?

i) They give a simple characterization of algorithm's efficiency.
ii) They allow the comparison of performances of various algorithms.

2067
2070

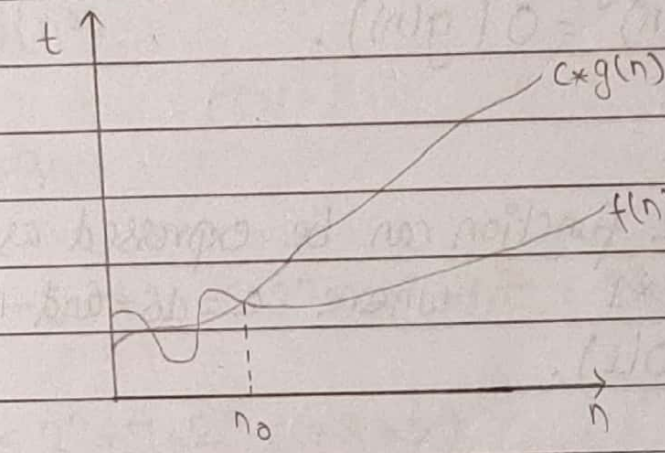
Big-Oh (O) notation:

When we have only asymptotic upper bound then we use big-oh notation.

A function $f(n) = O(g(n))$ (read as $f(n)$ is big-oh of $g(n)$) if and only if there exist two positive constants c and n_0 such that: $0 \leq f(n) \leq c * g(n) \quad \forall\ n \geq n_0$

This relation says that g(n) is upper bound of f(n). The following figure shows geometrical interpretation of big-oh notation.

O(1) is used to denote constants.



Some properties of big-oh notation:

i) Transitivity: $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$

ii) Reflexivity: $f(n) = O(f(n))$

iii) Transpose Symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

Examples:

Find big-oh notation for following functions:

1) $f(n) = 3n^2 + 4n + 1$
$g(n) = n^2$ then prove that $f(n) = O(g(n))$.

Proof: To prove $f(n) = O(g(n))$ we need to show that $0 \leq f(n) \leq c * g(n)$ if $n \geq n_0$, $c > 0$.

Let us choose c and n_0 values as 14 and 1 respectively then we have

$$0 \leq 3n^2+4n+7 \leq 14n^2 \quad \forall n \geq 1$$

This inequality is true.

Hence, $f(n) = O(g(n))$.

ii) $f(n) = 16$

- The above function can be expressed as:

$f(n) = 16 * 1$ where $c = 16$ and $n_0 = 0$.

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

iii) $f(n) = 2n+3$

- For $n \geq 3$, $f(n) = 2n+3 \leq 2n+n \leq 3n$

So, $f(n) \leq 3n \quad \forall n \geq 3$

Thus, $c = 3$, $n_0 = 3$

Hence, $f(n) = O(n)$.

iv) $f(n) = 4n^3 + 2n + 3$

- For $n \geq 3$, $f(n) = 4n^3 + 2n + 3 \leq 4n^3 + 2n + n \leq 4n^3 + 3n$

For $n^3 \geq 3n$,

$$4n^3 + 3n \leq 4n^3 + n^3 \leq 5n^3$$

Thus, $c = 5$, $n_0 = 3$.

So, $f(n) \leq 5n^3 \quad \forall n \geq 3$

Hence, $f(n) = O(n^3)$.

$2^n \geq n^2$

v) $f(n) = 10n^2 + 7$ (for $n \geq 7$, $f(n) = 10n^2 + 7 \leq 10n^2 + n^2 \leq 11n^2$)

- For $n \geq 7$, $10n^2 + 7 \leq 10n^2 + n^2 \leq 11n^2$
  Thus, $c = 11$, $n_0 = 7$
  So, $f(n) \leq 11n^2 \quad \forall n \geq 7$
  Hence, $f(n) = O(n^2)$.

vi) $f(n) = 2^n + 6n^2 + 3n$

- For $n^2 \geq 3n$,
  $f(n) = 2^n + 6n^2 + 3n \leq 2^n + 6n^2 + n^2 \leq 2^n + 7n^2$
  For $2^n \geq n^2$,
  $2^n + 7n^2 \leq 2^n + 7 * 2^n \leq 8 * 2^n$
  So,
  $c = 8, n_0 = 4$
  Hence, $f(n) = O(2^n)$.

2068
2070

Big Omega (Ω) notation:

Big omega notation gives asymptotic lower bound. A function $f(n) = \Omega(g(n))$ (read as $f(n)$ is big omega of $g(n)$) if and only if there exist two positive constants c and n_0 such that,

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

This relation says that $g(n)$ is lower bound of $f(n)$. The following figure shows geometrical interpretation of big omega (Ω) notation.

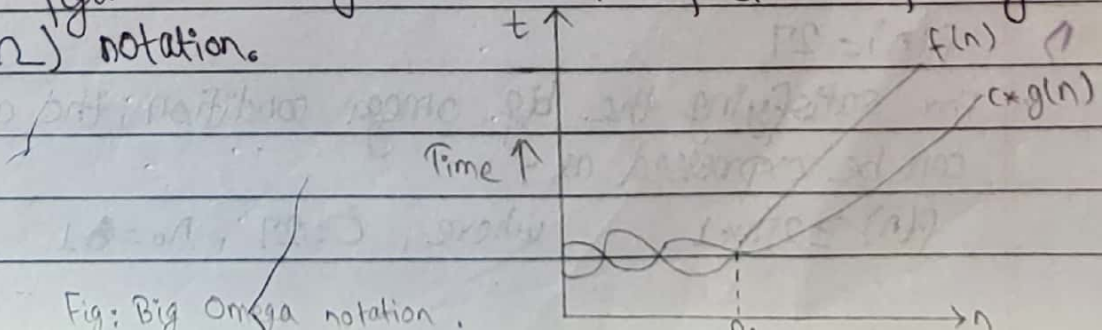


Fig: Big Omega notation.

Some properties of Big Omega (Ω) notation:

1) Transitivity: $f(n) = \Omega(g(n))$ & $g(n) = \Omega(h(n))$
$\Rightarrow f(n) = \Omega(h(n))$

2) Reflexivity: $f(n) = \Omega(f(n))$.

Examples:

1) $f(n) = 3n^2 + 4n + 7$
$g(n) = n^2$
Prove that $f(n) = \Omega(g(n))$.

$\Rightarrow$

Proof: To prove $f(n) = \Omega(g(n))$, we need to show that,
$0 \leq c * g(n) \leq f(n) \quad \forall n \geq n_0, c > 0$.
Let us choose c and n_0 values as 1 and 1 respectively.
$0 \leq c * n^2 \leq 3n^2 + 4n + 7 \quad \forall n \geq 1, c = 1$
$0 \leq 1 * n^2 \leq 3n^2 + 4n + 7 \quad \forall n \geq 1, c > 0$.
The above inequality is true.
Hence, $f(n) = \Omega(g(n))$
i.e. $3n^2 + 4n + 7 = \Omega(n^2)$.

proved.

Example 2: Find the big omega notation for following function.

1) $f(n) = 27$

→ For satisfying the big omega condition, the above function can be expressed as:
$f(n) \geq 27 * 1$ where, $c = 27$, $n_0 = 1$.

$f(n) \geq C \times g(n)$
$4n \geq 2n$

So, $f(n) = \Omega(1)$.
i.e. $27 = \Omega(1)$.

ii) $f(n) = 3n + 5$
→ $3n \leq 3n + 5$
So, $C = 3$ & $n_0 = 1$.
Thus, $f(n) = \Omega(n)$.
i.e. $3n + 5 = \Omega(n)$.

iii) $f(n) = 27n^2 + 10n$
→ $27n^2 \leq 27n^2 + 10n$
So, $C = 27$ & $n_0 = 1$
Thus, $f(n) = \Omega(n^2)$.
i.e. $27n^2 + 10n = \Omega(n^2)$.

iv) $f(n) = 3^n + 6n^2 + 3n$
→ $3^n \leq 3^n + 6n^2 + 3n$
So, $C = 1$, $n_0 = 1$
Thus, $f(n) = \Omega(3^n)$
So, $3^n + 6n^2 + 3n = \Omega(3^n)$.

2068
2070

Big Theta (Θ) Notation:

When we need asymptotically tight bound then we use big theta notation. A function $f(n) = \Theta(g(n))$ (read as $f(n)$ is big theta of $g(n)$) if and only if there exists three positive constants c_1, c_2 and n_0 such that

$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \quad \forall n \geq n_0$$

This relation says that $g(n)$ is tight bound of $f(n)$. The following figure shows geometrical interpretation of Big theta (Θ) notation.

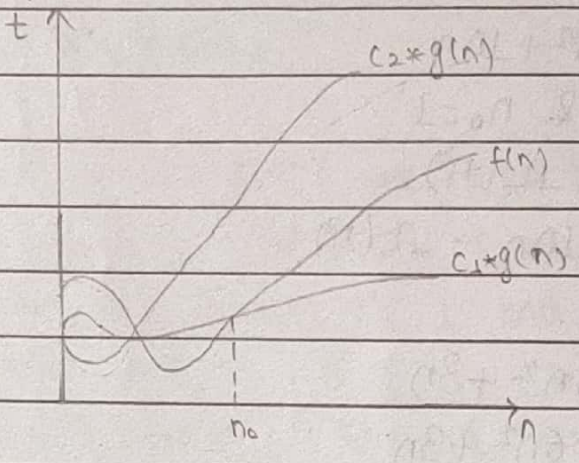


Fig: Big theta notation

Some properties of Big theta notation:

i) Transitivity: $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$
$\Rightarrow f(n) = \Theta(h(n))$.

ii) Reflexivity: $f(n) = \Theta(f(n))$

Example 1:

$f(n) = 3n^2 + 4n + 7$

$g(n) = n^2$, then prove that $f(n) = \Theta(g(n))$.

Proof: To prove $f(n) = \Theta(g(n))$, we need to show that,

$0 \leq C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n) \quad \forall n \geq n_0$ and $C > 0$.

i.e. $0 \leq C_1 \cdot n^2 \leq 3n^2 + 4n + 7 \leq C_2 \cdot n^2 \quad \forall n \geq n_0$ & $C > 0$.

Let us choose C_1, C_2 and n_0 values as 3, 14 and 1 respectively then,

$0 \leq 3 * n^2 \leq 3n^2 + 4n + 7 \leq 14 * n^2 \quad \forall n \geq 1, C_1 = 3, C_2 = 14$.

This inequality is true,

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

i.e. $3n^2 + 4n + 7 = \Theta(n^2)$.

Example 2:

Find the big theta notation for the following functions:

i) $f(n) = 16$

∴ For satisfying the big theta notation, the above function can be expressed as:

$0 \leq C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n) \quad \forall n \geq n_0$ and $C > 0$.

$0 \leq 16 \times 1 \leq f(n) \leq 16 \times 1 \quad \forall n \geq n_0$ & $C > 0$.

$f(n) =$

ii) $f(n) = 3n+5$

→ $3n \leq 3n+5 \quad \forall n \geq 1, C_1 = 3.$

also,

for $n \geq 5$

$3n+5 \leq 3n+n \leq 4n \quad \forall n \geq 5, C_2 = 4.$

So,

$3n \leq 3n+5 \leq 4n$

i.e. $3 * g(n) \leq f(n) \leq 4 * g(n)$ where, $C_1 = 3, C_2 = 4, n_0 = 5.$

Hence,

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

i.e. $3n+5 = \theta(n).$

iii) $f(n) = 2n^3 + n^2 + 2n$

→ $2n \leq 2n^3+n^2+2n \quad \forall n \geq 1, C_1 = 2.$

also,

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

$2n^3+n^2 \leq 2n^3+n^2+2n \leq 4n^3+n \geq 1.$

So,

$2n \leq 2n^3+n^2+2n \leq 2\frac{1}{3}n^3$

$n_0 = 2$

linear eq^n = O(n)

Analysis of sample algorithm using RAM model:

```
int sum(int n)                    constant:      time
{
   int total = 0;                    C1            1
   for(i=1; i<=n; i++)               C2           n+1
     total+ = i*i;                   C3            n
   return total;                     C4            1
}
```

Time complexity = $C_1*1 + C_2*(n+1) + C_3*n + C_4*1$

$= C_1 + C_2n + C_2 + C_3n + C_4$

$= (C_2+C_3)n + C_1 + C_2 + C_4$

$= O(n)$

Another way of analysis:

total = 0	⇒ 1 step
i = 1	⇒ 1 step
i<=n	⇒ n+1 step
i++	⇒ n step
total += i*i	⇒ n step
return total	⇒ 1 step

Total step = 3n+4 step

$T(n) = O(n)$

Some theorems (without proof):

Theorem 1:
if $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ and $a_m > 0$ then $f(n) = O(n^m)$.

Theorem 2:
if $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ and $a_m > 0$ then $f(n) = \Omega(n^m)$.

Theorem 3:
if $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ and $a_m > 0$ then $f(n) = \Theta(n^m)$.

Theorem 4:
For any two function $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$.

Some examples of illustrating theorems:

a) Applying theorem 1:

i) $f(n) = 2n + 3$ here, $m = 1$, thus $f(n) = O(n^m) = O(n)$.
ii) $f(n) = 27n^2 + 16n + 25$ here, $m = 2$, thus $f(n) = O(n^m) = O(n^2)$.
iii) $f(n) = 10n^2 + 7$ here, $m = 2$, thus $f(n) = O(n^m) = O(n^2)$.

b) Applying theorem 2:

i) $f(n) = 2n+3$ here $m=1$, thus $f(n) = \Omega(n^m) = \Omega(n)$.
ii) $f(n) = 27n^2+16n+25$ here $m=2$, thus $f(n) = \Omega(n^m) = \Omega(n^2)$.
iii) $f(n) = 10n^2+7$ here $m=2$, thus $f(n) = \Omega(n^m) = \Omega(n^2)$.

c) Applying theorem 3:

i) $f(n) = 2n+3$ here $m=1$, thus $f(n) = \Theta(n^m) = \Theta(n)$.
ii) $f(n) = 27n^2+16n+25$ here $m=2$, thus $f(n) = \Theta(n^m) = \Theta(n^2)$.
iii) $f(n) = 10n^2+7$ here $m=2$, thus $f(n) = \Theta(n^m) = \Theta(n^2)$.

d) Applying theorem 4:

$f(n) = an^2+bn+c = O(n^2)$ for any constant a, b and c.
$f(n) = an^2+bn+c = \Omega(n^2)$ for any constant a, b and c.
so,
$f(n) = an^2+bn+c = \Theta(n^2)$ for any constant a, b and c.

Analysis of more algorithms using RAM model:

1)

```
                          constant      time
i=0                       C1            1
while(i<n)                C2            n+1
{
  j=0                     C3            n
  while(j<n)              C4            n(n+1)
  { x=x+y;                C5            n*n
    j++;                  C6            n*n
  }
  i++;                    C7            n
}
```

Total running time of above code fragment = $C_1 + C_2(n+1) + C_3 + C_4(n(n+1)) + C_5n^2 + C_6n^2 + C_7n$

$= C_1 + C_2n + C_2 + C_3n + C_4n^2 + C_4n + C_5n^2 + C_6n^2 + C_7n$

$= (C_4 + C_5 + C_6)n^2 + (C_2 + C_3 + C_4 + C_4)n + C_1 + C_2$.

Another way of analyzing algorithm:

1)

i = 0	⇒ 1 step
i < n	⇒ n+1 step
j = 0	⇒ 1step * n time = n step
j < n	⇒ n+1 step * n time = n(n+1) step
x = x+y	⇒ nstep * n time = n^2 step
j++	⇒ nstep * n time = n^2 step
i++	⇒ 1step * n time = n step

Total step = $3n^2 + 4n + 2$

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

II)

	constant:	time:
n = read input	C_1	1
sum = 0	C_2	1
i = 1	C_3	1
while (i < n)	C_4	n-1.
{ num = read input number	C_5	1*(n-1)
sum = sum + num	C_6	1*(n-1)
i = i+1	C_7	1*(n-1)
} avg = sum/n;	C_8	1
print avg;	C_9	1

Time complexity = $C_1+C_2+C_3+C_4(n-1)+C_5(n-1)+C_6(n-1)+C_7(n-1)+C_8+C_9$

$= C_1+C_2+C_3+C_4n-C_4+C_5n-C_5+C_6n-C_6+C_7n-C_7+C_8+C_9$

$= (C_4+C_5+C_6+C_7)n+C_1+C_2+C_3-C_4-C_5-C_6-C_7+C_8+C_9$

$= O(n)$

(iii)

```
function()
{
  int a;
  a=5;
  a++;                                  } O(1)
  for(i=0; i<n; i++)
  {                                     } O(n)
    // constant
  }
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)                 } O(n^2)
    {
      // constant
    }
  }
}.
```

The running time of code segment $T(n) = O(1)+O(n)+O(n^2)$

$= O(n^2)$.

Time complexity analysis of code segment (general rule):

Rule 1: Running time = Σ Running time of all statements

Rule 2: Running time = Σ Running time of code fragments

- If fragment contains simple statements then running time of fragment is O(1).

eg:

```
int a;
a=5;
a++;
```

O(1)

- If fragment have simple loops as

```
for(i=0; i<n; i++)
{
  // simple statement
}
```

O(n)

- If fragment have nested loop as

```
for (i=0; i<n; i++)
{
   for(j=0; j<n; j++)
   {
      // simple statement
   }
}
```

$O(n^2)$

Rule 3: If there is a conditional statement, pick the complexity of condition which is worst.

Example:

```
function()
{
    if (condition)
    {
        for (i=0; i<n; i++)
        {
            // simple statement
        }
    }
    else
    {
        for (i=0; i<n; i++)
        {
            for (j=0; j<n; j++)
            {
                // simple statement
            }
        }
    }
}
```

O(n) — for the if-branch

O(n²) — for the else-branch

The running time of code segment $T(n) = O(n^2)$.

Example:

		constant:	time:
1)	`for (i ← 1 to k)`	C_1	$k+1$
	`{`		
	`for (i ← 1 to k²)`	C_2	$k(k^2+1)$
	`{`		
	`for (i ← 1 to k³)`	C_3	$k(k^2)(k^3+1)$
	`{`		
	`P ← P*P;`	C_4	$k \cdot k^2 \cdot k^3$
	`}`		
	`}`		
	`}`		

Time complexity $T(k) = C_1(k+1) + C_2(k(k^2+1)) + C_3(k)(k^2)($
$+ C_4 k \cdot k^2 \cdot k^3$
$= C_1k + C_1 + C_2k^3 + C_2k + C_3k^6 + C_3k^3 +$
C_4k^6
$= O(k^6).$

		constant:	time:
2)	`for (i ← 2 to k-1)`	C_1	$k-1$
	`{`		
	`for (j ← 3 to k-2)`	C_2	$(k-2)(k-3)$
	`{`		
	`A ← A+2;`	C_3	$(k-2)(k-4)$
	`}`		
	`}`		

$\therefore T(k) = O(k^2).$

Imp: Q: Why we usually concentrate on finding the worst case running time?

- The worst case running time is the longest running time for any input of size 'n'. We usually concentrate on finding only the worst case running time because of the following reason:

i) The worst case running time of an algorithm gives us an upper bound on the running time for any input, knowing it provides a guarantee that algorithm will never take any longer time.

ii) For some algorithm the worst case occur, for example: when searching, the worst case often occurs when the item being search for is not present and search for absent items may be frequent.

Recurrence :-

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. To solve a recurrence relation means to obtain a function defined on the natural numbers that satisfies the recurrence. Recurrence relation in the computer algorithms can model the complexity of the running time, particularly divide and conquer algorithm.

For example the worst case running time $T(n)$ of merge sort procedure is described by the recurrence

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

Solving the Recurrence relations:

Following are the four techniques that are used to solve the recurrence relations:

i) Substitution Method
ii) Iteration Method
iii) Recursion Tree
iv) Masters Method.

i) Iteration Method:

In this method, the given recurrence relation is expanded until the recursion termination point and try to bound it using known series.

2070

Example

1) $T(n) = 2T(n/2) + 1$ when $n > 1$
$= 1$ $n = 1$

Sol:

Here, $T(n) = 2T(n/2) + 1$

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

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

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

$= 2^2\left[2T\left(\frac{n}{8}\right) + 1\right] + 2 + 1 \quad \left\{\because T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 1\right\}$

$= 2^3 T\left(\frac{n}{8}\right) + 4 + 2 + 1$

Expanding in similar way;

$T(n) = 2^k T\left(\frac{n}{2^k}\right) + 2^{k-1} + 2^{k-2} + \ldots + 2^0$

Let $2^k = n$ then $k = \log n$

$T(n) = 2^k T(1) + 2^{k-1} + 2^{k-2} + \ldots + 2^0$

$T(n) = 2^k + 2^{k-1} + 2^{k-2} + \ldots + 2^0$

$= \sum_{i=0}^{k} 2^i$

$= \frac{2^{k+1} - 1}{2 - 1}$

$= 2^{k+1} - 1$

$= 2^k \cdot 2 - 1$

$= 2n - 1$

$= O(n)$.

2) $T(n) = 2T(n/2) + n$ when $n > 1$

$= 1$ $n = 1$

Sol:

Here, $T(n) = 2T(n/2) + n$

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

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

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

$= 2^3\left[2T(n/8) + n/4\right] + 2n$

$= 2^3 T(n/8) + n + 2n$

$= 2^3 T(n/8) + 3n$

Expanding in similar way,

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

let $2^k = n$ then $k = \log n$

$T(n) = 2^k T(1) + kn$

$= 2^k + kn$

$= n + \log n . n$

$= O(n \log n)$

3) $T(n) = T(n-1) + n^4$ when $n > 1$

$= 1$ $n = 1$

Sol:

Here, $T(n) = T(n-1) + n^4$

$= T(n-2) + (n-1)^4 + n^4$

$= T(n-3) + (n-2)^4 + (n-1)^4 + n^4$

$= T(n-4) + (n-3)^4 + (n-2)^4 + (n-1)^4 + n^4$

$\vdots$

$= 1^4 + 2^4 + \dots (n-3)^4 + (n-2)^4 + (n-1)^4 + n^4$

$= \sum_{i=1}^{n} i^4$

11) Substitution Method:

The substitution method for solving recurrences comprises two steps:

a) Guess the form of the solution
b) Use mathematical induction to find constants and show that the solution works.

How to guess:

- If recurrence is similar to one we are familiar then guessing a similar solution is reasonable.
- Draw a recursion tree to generate good guesses.
- Applying master theorem to generate good guess.
- Do a few steps of iteration method and then guess.

Making a good guess:

- If a recurrence is similar to one we have seen before then guessing a similar solution is reasonable. For example;

$$T(n) = 2T(\lceil n/2 \rceil + 17) + n$$

which looks difficult because of added 17 in the argument to T, but this additional term cannot substantially affect the solution to the recurrence because when n is large, the difference between $T\left(\left\lceil \frac{n}{2} \right\rceil\right)$ and $T\left(\left\lceil \frac{n}{2} \right\rceil + 17\right)$ is not large and both cut n nearly evenly in half. Hence, we make the guess that $T(n) = O(n \log n)$.

- Another way to make a good guess is to prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainly.
- There are situation when we can correctly guess at an asymptotic bound on the solution of a recurrence but somehow math doesn't seem to work out in the induction. When we hit such a situation revising the guess by substituting a lower order term often permits the math to go through.
- Changing variable ~~that~~ is i.e. sometimes a little algebric manipulation can make an unknown recurrence similar to one we have seen before.

Example 1: Solve the recurrence $T(n) = 2T(n/2) + n$ by substitution method.

Sol:

1. Guess the solution
   Let $T(n) = O(n \log n)$
   i.e. $T(n) = c \cdot n \log n$

2. Now we use mathematical induction.
   i) Best case :- Here our guess does not hold for $n=1$ because
   $T(1) \leq c \cdot 1 \log 1$
   i.e. $T(1) \leq 0$ which is contradiction with $T(1) = 1$.
   Now for $n=2$
   $T(2) \leq c \cdot 2 \log 2$
   $2T\left(\frac{2}{2}\right) + 2 \leq c \cdot 2 \cdot 1$
   $2 \cdot 1 + 2 \leq c \cdot 2$
   $4 \leq c \cdot 2$

so, $T(n) \leq c.n\log n$ is true for $n=2$.

ii) Inductive step : Now assume it is true for $n=n/2$
i.e. $T\left(\frac{n}{2}\right) \leq c.\frac{n}{2}\log n/2$ is true.

Now we have to show that it is true for $n=n$
i.e. $T(n) \leq c.n\log n$

We know,

$$T(n) \leq 2T(n/2)+n$$
$$\leq 2\left[c.n/2\log n/2\right]+n$$
$$\leq \left[c.n(\log n-\log 2)\right]+n$$
$$\leq cn\log n - cn\log 2 + n$$
$$\leq cn\log n + n - cn$$
$$\leq cn\log n \quad \forall\ c\geq 1$$

Thus, $T(n) = O(n\log n)$.

Example 2:
$T(n) = T(n/2)+1$
We have to show that it is asymptotically bound by $O(\log n)$

soln:

For $T(n) = O(\log n)$
We have to show that for some constant c.
$T(n) \leq c.\log n$.
Put this in the given recurrence equation
$T(n) \leq c.\log\left(\frac{n}{2}\right)+1$

$\leq c \log\left(\frac{n}{2}\right) + 1 = c \log n - c \log_2 2 + 1$

$\leq c \log n$ for $c \geq 1$.

Thus, $T(n) = O(\log n)$.

P.T.O.

Example 3:

Consider the following recurrence

$$T(n) = 2T(\sqrt{n}) + \log n$$

Solve it by changing variable.

soln:

Here,

$$T(n) = 2T(\sqrt{n}) + \log n$$

Suppose, $m = \log_2 n \Rightarrow n = 2^m$

$n^{1/2} = 2^{m/2} \Rightarrow \sqrt{n} = 2^{m/2}$

Putting the values, we get,

$$T(2^m) = 2T(2^{m/2}) + m$$

Again consider,

$$S(m) = T(2^m)$$

we have,

$$S(m) = 2S(m/2) + m$$

We know that, this recurrence has the solution

$$S(m) = O(m \log m).$$

Substituting the values of m, we get

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

Example 4:

Solve the recurrence.

$$T(n) = 2T(\sqrt{n}) + 1$$

By making the change of variable.

Soln:

Here,

$$T(n) = 2T(\sqrt{n}) + 1$$

Suppose, $m = \log_2 n \Rightarrow n = 2^m$

$n^{1/2} = 2^{m/2} \Rightarrow \sqrt{n} = 2^{m/2}$

Putting the values, we get,

$$T(2^m) = 2T(2^{m/2}) + 1$$

Again consider,

$$S(m) = T(2^m)$$

we have,

$$S(m) = 2S(m/2) + 1$$

We know that, this recurrence has the solution

$$S(m) = O(\log m)$$

Substituting the values of m we get,

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

iii) Recursion tree:

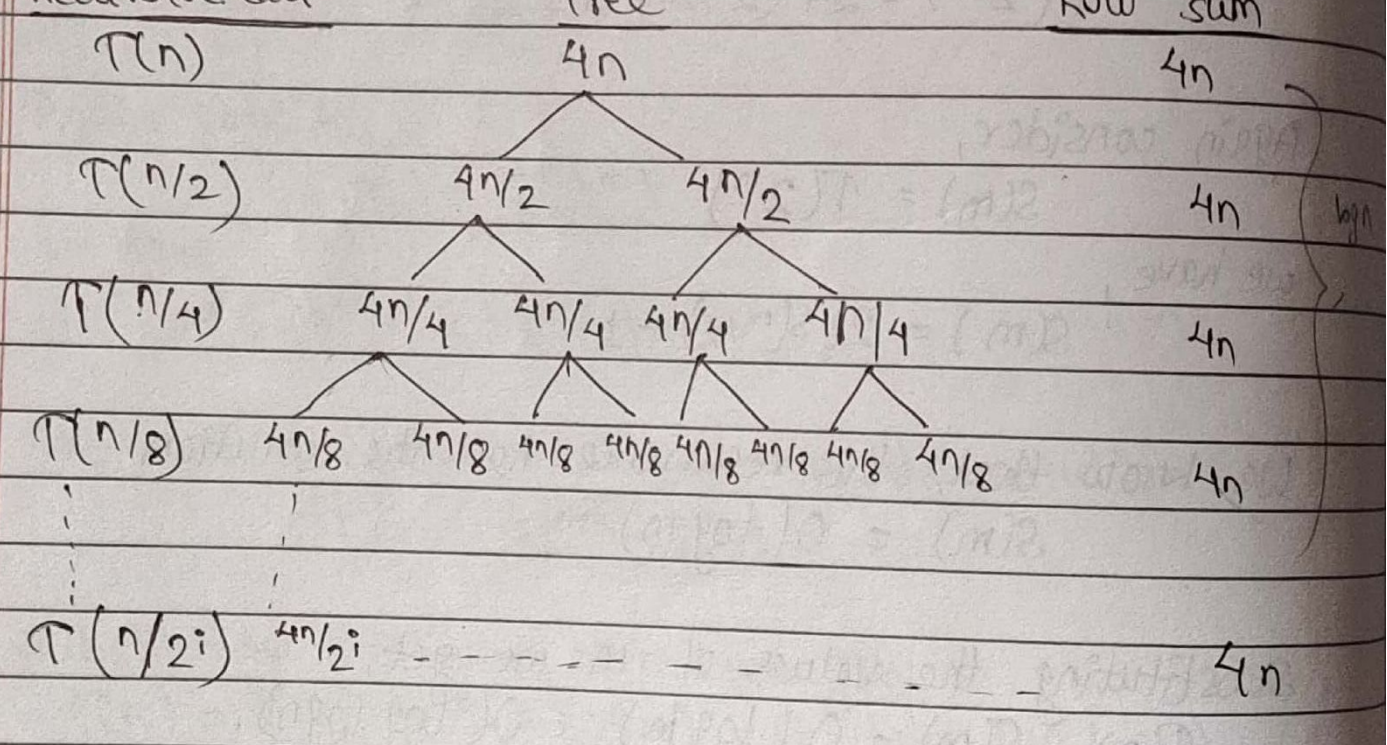
Recursion tree method is a pictorial representation of an iterative method, which is in the form of tree, where at each level nodes are expanded. It is useful when divide and conquer algorithm is used.

Example 1:

Obtain asymptotic bound using recursion tree method for the following recurrence relation.

$T(n) = 2T(n/2) + 4n$

Soln:



Cost at each level $= 4n$

For simplicity, let us assume that, $2^i = n \Rightarrow i = \log n$

Summing the cost at each level

Total cost $(T(n)) = \sum_{i=0}^{\log n} 4n = 4n \sum_{i=0}^{\log n} 1$

$= 4n(\underbrace{1+1+1+\ldots+1}_{\{\log n + 1 \text{ times}\}})$

$= 4n(\log n + 1)$

$= 4n\log n + 4n$

$= O(n\log n)$.

2067

Example 2:

Obtain the asymptotic bound of following recurrence using recursion tree method.

$$T(n) = T(n/2) + 1.$$

Soln:

Recursive call	Tree	Row sum
$T(n)$	1	1
	↓	
$T(n/2)$	1	1
	↓	
$T(n/4)$	1	1
⋮	⋮	⋮
$T(n/2^i)$	1	1

$\log n$

cost at each level = 1

For simplicity, let us assume that, $2^i = n \Rightarrow i = \log n$.

Summing the cost at each level,

Total cost = $1+1+1+\ldots$ Upto $\log n+1$ terms. $= \sum_{i=0}^{\log n} 1 = \log n + 1$.

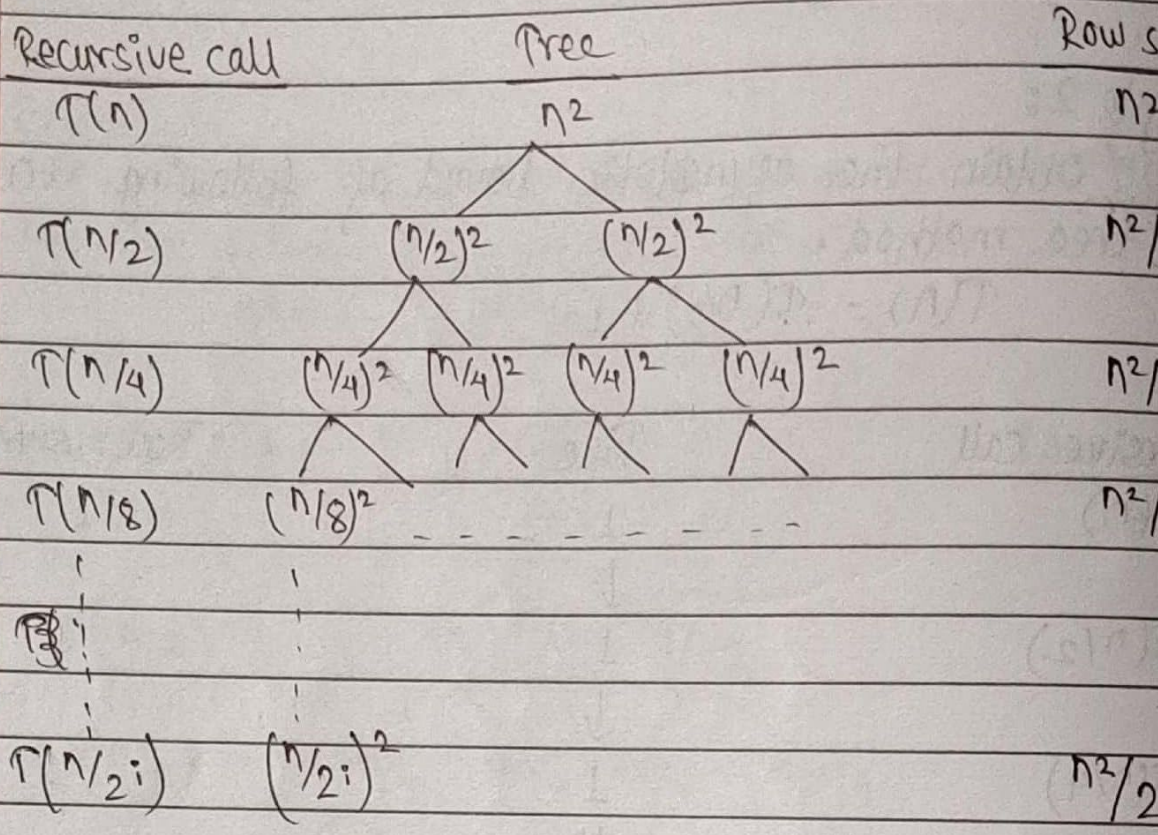
$\Rightarrow$ Complexity = $O(\log n)$.

Example 3:

Obtain the asymptotic bound of following recurrence using recursion tree method.

$T(n) = 2T(n/2) + n^2$.

Soln:



For simplicity, let us assume that $2^i = n \Rightarrow i = \log n$.

Total cost $T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \dots$ $\log n$ times

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right) = n^2 \left(\frac{1}{1 - 1/2}\right)$$

$$= 2n^2$$

$\therefore T(n) = O(n^2)$.

Example 4:

Solve the following recurrence by using the recursion tree method.

$$T(n) = 8T(n/2) + n^2.$$

Soln:

Recursive call	Tree	Row sum
$T(n)$	n^2	n^2
$T(n/2)$	$(n/2)^2$ 8 times $(n/2)^2$	$8n^2/4 = 2n^2$
$T(n/4)$	$(n/4)^2$ $(n/4)^2$	$64n^2/16 = 4n^2 = 2^2n^2$
⋮		
$T(n/2^i)$	$(n/2^i)^2$ - - - - - -	$2^i n^2$

Example 5:

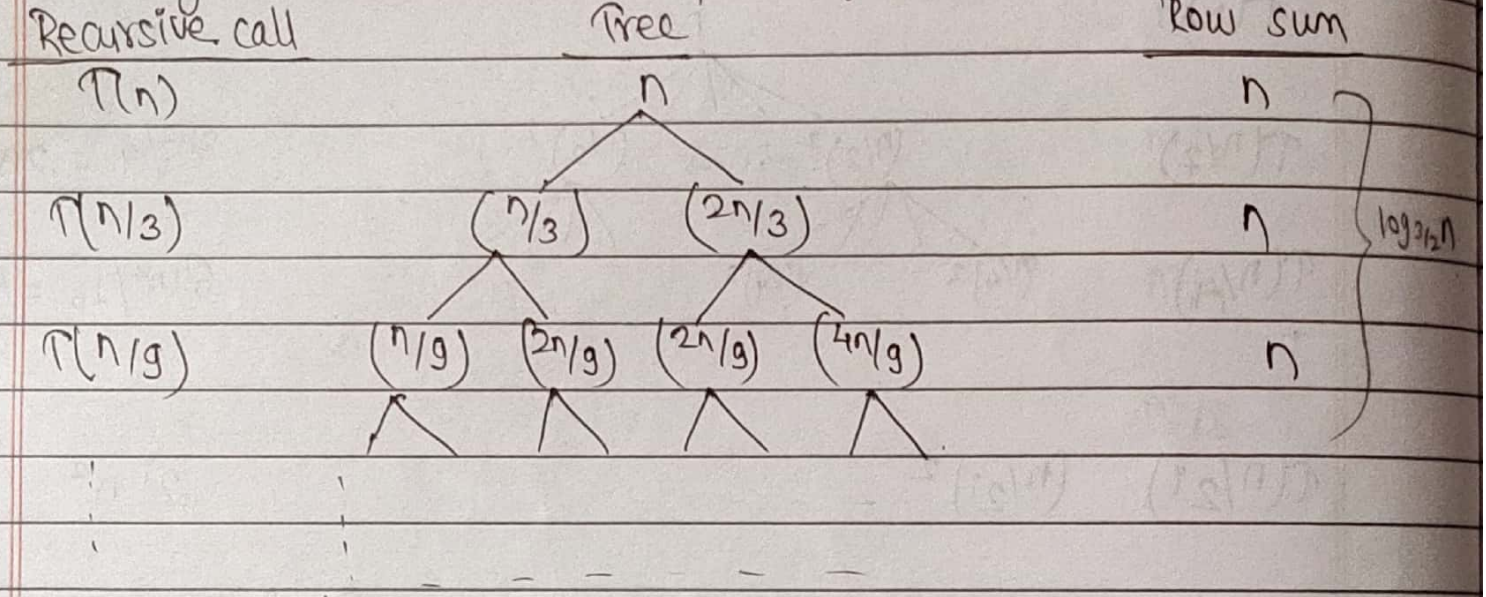
Consider the following recurrence.

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

Obtain the asymptotic bound using recursion tree method.

Sol:

The given recurrence has following recursion tree.



Cost at each level = n

The longest path from the root to a leaf is:

$n \rightarrow \frac{2}{3}n \rightarrow \left(\frac{2}{3}\right)^2 n \rightarrow 1$

Since $\left(\frac{2}{3}\right)^i n = 1$ then $i = \log_{3/2} n$

Thus the height of the tree is $\log_{3/2} n$.

$\therefore$ Total cost $T(n) = n + n + n + \ldots + \log_{3/2} n$ times

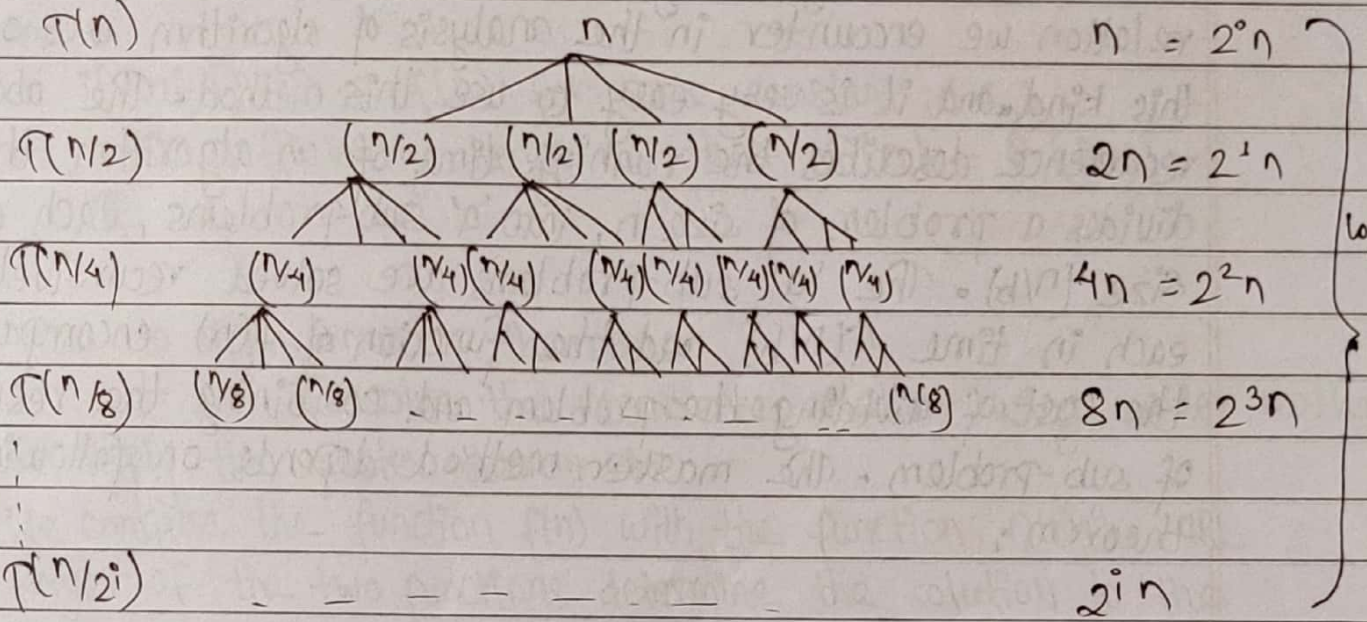
$= O(n \log n)$.

Example 6:

$$T(n) = 4T(n/2) + n$$

Soln:

Recursive call | Tree | Row sum



$T(n) = aT(n/b) + f(n)$ $b \rightarrow$ size $a \geq 1, b > 1$

2071(2)

iv) Masters Method:-

The master method is used to solve the recurrences of the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. This method is very handy because most of the recurrence relation we encounter in the analysis of algorithm are of this kind and it is very easy to use this method. The above recurrence describes the running time of an algorithm that divides a problem of size n, into 'a' sub-problems, each of size (n/b). The 'a' sub-problems are solved recursively each in time $T(n/b)$ and the function $f(n)$ encompasses the cost of dividing the problem and combining the result of sub-problem. The master method depends on following theorem:

Master Theorem:

Let $a \geq 1$ and $b > 1$ be constants, Let $f(n)$ be a function, and let $T(n)$ be defined on non-negative integers by the recurrence.

$$T(n) = a\,T(n/b) + f(n)$$

where,

n/b is either $\lfloor n/b \rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$. Then $T(n)$ has following asymptotic bound.

Case 1: $f(n) < n^{\log_b a}$

if $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$.

then, $T(n) = \Theta\left(n^{\log_b a}\right)$.

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

if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

Case 3: $f(n) > n^{\log_b a}$

if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$.

and,

if $af(n/b) \leq cf(n)$ for some constant $c < 1$.

and, all sufficiently large n, then $T(n) = \Theta(f(n))$

How to apply master theorem:-

To solve the recurrence relation using master method we apply the master theorem as:

i) We compare the function $f(n)$ with the function $n^{\log_b a}$. The larger of the two functions determine the solution to the recurrence

ii) If as in case 1, the function $n^{\log_b a}$ is larger, then the solution of recurrence relation is $T(n) = \Theta(n^{\log_b a})$.

iii) If as in case 3, the function $f(n)$ is larger, then the solution is $T(n) = \Theta(f(n))$.

iv) If as in case 2, the two functions are the same size, then the solution is $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$.

Example 1: $T(n) = 9T(n/3) + n$

Soln:

Here,

$a = 9, b = 3, f(n) = n$

Then,

$n^{\log_b a} = n^{\log_3 9} = n^2$

Now, $f(n) < n^{\log_b a}$ i.e. $n < n^2$

Then, from case 1 of master theorem.

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

Example 2: $T(n) = T(2n/3) + 1$

Soln:

Here,

$a = 1, b = 3/2, f(n) = 1$

Then,

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Now,

$f(n) = n^{\log_b a}$ i.e. $1 = 1$.

Then,

from case 2 of master theorem,

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

$= \Theta(\log n)$.

2069

Example 3: $T(n) = 3T(n/4) + n\log n$

Soln:

Here,

$a = 3$, $b = 4$, $f(n) = n\log n$

Now,

$n^{\log_b a} = n^{\log_4 3} = n^{0.8}$

Then,

$f(n) = \Omega(n^{\log_4 3 + \epsilon})$ where $\epsilon = 0.2$.

Hence, case 3 is applied. Now, for regularity condition i.e.

$3(n/4)\log(n/4) \leq 3/4\, n\log n$ for $c = 3/4$

$\because af(n/b) \leq c \cdot f(n)$

$3f(n/4) \leq c \cdot f(n)$.

$3(n/4)\log(n/4) \leq 3/4\, n\log n$.

Therefore, the solution is $T(n) = \Theta(n\log n)$.

Example 4: $T(n) = 16T(n/4) + n^3$

Soln:

Here,

$a = 16$, $b = 4$, $f(n) = n^3$

Now,

$n^{\log_b a} = n^{\log_4 16} = n^2$

Then,

$f(n) \geq n^{\log_b a}$ i.e. $n^3 > n^2$

Then, i.e. $f(n) = \Omega(n^{2+\epsilon})$ for $\epsilon = 1$.

From case 3 of master theorem,

$T(n) = \Theta(n^3)$.

Now, regularity condition i.e.

$af(n/b) = 16(n/4)^3 = \frac{1}{4}n^3 \leq \frac{1}{4}n^3 = c \cdot f(n)$ for $c = \frac{1}{4}$.

Hence, case 3 is applied, thus,

$T(n) = \Theta(n^3)$

Example 5: $T(n) = 2T(n/2) + n\log n$

Sol:

Comparing the given recurrence with $T(n) = aT(n/b) + f(n)$

then,

$a = 2, b = 2, f(n) = n\log n$ solution cannot be found.

It is not polynomially

Now,

$n^{\log_b a} = n^{\log_2 2} = n^1 = n$

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

Hence, case 2 is applied, then the solution is,

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

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

Example 6: $T(n) = 2T(n/2) + \theta(n)$

Sol:

By master theorem,

$a = 2, b = 2, f(n) = \theta(n)$

Now,

$n^{\log_b a} = n^{\log_2 2} = n$

or, $f(n) = \theta(n^{\log_b a}) = \theta(n)$

Hence, case 2 is applied, then the solution is

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

Example 7 : $T(n) = 8T(n/2) + \theta(n^2)$.

Sol:

By master theorem,

$a = 8, \quad b = 2, \quad f(n) = \theta(n^2)$

Now,

$n^{\log_b a} = n^{\log_2 8} = n^3$

or, $f(n) = O(n^{\log_b a - \epsilon})$

$= O(n^{3-\epsilon})$ for $\epsilon = 1$.

Hence, case 1 is applied, then the solution is,

$T(n) = \theta(n^3)$.

Example 8: $T(n) = 7T(n/2) + \theta(n^2)$.

Sol:

By master theorem,

$a = 7, \quad b = 2, \quad f(n) = \theta(n^2)$

Now,

$n^{\log_b a} = n^{\log_2 7} = n^{2.8}$

or, $f(n) = \theta(n^2)$

Case 1 is applied.

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

$= O(n^{2.8 - 0.8})$ for $\epsilon = 0.8$

$\therefore T(n) = \theta(n^{\log_2 7})$ //

Note:

It is important to note that the three cases of master theorem do not cover all the possibilities for $f(n)$. There is a gap between case 1 and case 2 when $f(n) < n^{\log_b a}$ but not polynomially smaller. Similarly, there is a gap between case 2 and case 3 when $f(n) > n^{\log_b a}$ but not polynomially larger. If the function $f(n)$ falls into one of these gaps or if the regularity condition in case 3 fails to hold, master theorem cannot be used to solve the recurrence.

Program → algorithm + data structure.
integer data type - Primitive

logical representation of data

1.2) DATA STRUCTURE BRIEF OVERVIEW:

Data structure is a way of organizing all data items and establishing the relationship among those data items. In another words, it is a mathematical or logical model of a particular data. Thus, a data structure is the portion of memory allocated for a model in which the required data can be arranged in proper fashion. Algorithm and its associated data structure form a program.

Program = Algorithm + Data Structure

Data structure can be classified into two types:

1) Primitive Data Structure
2) Non primitive Data Structure

1) Primitive Data Structure:

These are the basic data structure that are directly operated upon by machine level instruction i.e. fundamental data types such int, float, double, etc are known as primitive data structure.

2) Non-primitive Data Structure:

These are the most sophisticated data structures and are derived from the primitive data structures. The non-primitive data structure emphasize on structuring of homogeneous or heterogeneous data items. Examples of non-primitive data structures are array, list, etc. There are two types of non-primitive data structures i.e. linear

data structure and non-linear data structure.

a) Linear data structure:
2068

A data structure is said to be linear if its element form a sequence. For example: stack, queue, list, array, etc.

b) Non-linear or hierarchical data structure:

A data structure is said to be non-linear if its element donot form a sequence. For example: tree and graph.

Classification of data structure based on capacity:-

Data structure can also be classified based on its capacity as:

i) Static Data structure
ii) Dynamic Data structure

i) Static Data Structure:-

A static data structure is one whose capacity is fixed at creation time. An array is the example of static data structure.

ii) Dynamic Data structure:-

A dynamic data structure is one whose capacity is variable so it can be expanded or contracted at any time. The linked list, binary tree, etc are the example of dynamic data structure.

Representation of data structure:

Any data structure can be represented in two ways, they are:

i) Sequential Representation
ii) Linked Representation

i) Sequential Representation (Array):-

A sequential representation maintains data in contiguous memory location which takes less time to retrieve data but lead to more complexity during insertion and deletion operation.

ii) Linked Representation:-

Linked representation maintains the list by means of a link between the adjacent element which need not be stored in contiguous memory location. During the insertion and deletion operation link will be created or removed which takes less time compared to sequential representation.

Linear Data Structure:-

A data structure is said to be linear if its element form a sequence or linear list. The two common linear data structure are:

i) List
ii) Array

1) List :-

List is a simple general purpose data structure. Most fundamental representation of list is through an array and linked list. List requires linear space to collect and store the element where linearity is proportional to number of items. For example :- to store 'n' items in an array requires nxd space where d is the size of an element.

static list

11) Array :-

An array is a data structure used to store homogeneous elements at contiguous location. This means that array can store either all integers, all floating point numbers, all characters, or any other complex data type but all of the same type. Size of an array must be provided before storing the data. An array can store multiple values which can be referred by single name. One dimensional array can be declared as follows :

```
data_type array_name [size];
eg: int number [10];
```

Accessing array elements :

Individual elements of the array can be accessed using the following syntax :

```
array_name [index];
eg: int num [100];
    num [5] = 90;
```

Insertion in an array:

For inserting an element in the array, the logic is straight forward. Inserting an element at the end of the array can be easily done if the memory space allocated for the array is large enough to accomodate the additional element. For inserting an element at required position, element must be shifted one location to accomodated new element.

1	5	6	7	22	90		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Suppose we want to insert 20 in array at location 4. then first shift element from index 4 to end

1	5	6	7		22	90	

Now insert the element at index 4.

1	5	6	7	20	22	90	

array after insertion

Algorithm to insert an element in an array:

```
insert (A, length, position, num)
{
  set i = length
  repeat for i = length down to position
     set A [i+1] = A [i]
  set A[position] = num;
  reset length = length + 1;
}
```

Deletion of an element in an array:

For the deletion of an element from the array, the logic is straight forward. Deleting an element at the end of array present no difficulties, but deleting an element from some other position of the array would require to shift all element to fill the space emptied by deletion of the element.

For example:

10	7	20	13	66	90	5	10
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

original array.

Suppose we want to delete element a[5] = 90 then,

10	7	20	13	66		5	10

Now, shift all elements after the deleted element to accomodate emptied position.

10	7	20	13	66	5	10

after deleting an element from the array

Algorithm:

```
Delete (A, position, n)
{
    item = A[position]
    set i = position upto n
        set A[i] = A[i+1];
        set n = n-1;
}
```

In summary, let the size of array be n then;

i) access time : O(1). This is possible because elements are stored in contiguous location.

ii) search time : O(n). For sequential search.

iii) Insertion time : O(n). The worst case occurs when insertion happens at beginning of an array.

iv) Deletion time : O(n). The worst case occurs when deletion happens at beginning of an array that requires shifting all elements.

v) Traversing : O(n)

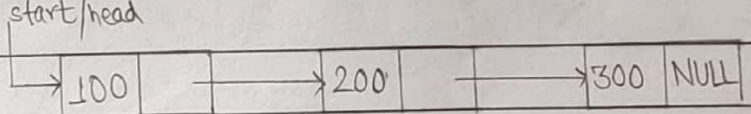
dynamic list

Linked List:

A linked list is a linear collection of data elements called nodes where each node has two parts:

i) Info : The actual value to be stored and processed in the list, it is also called data field.

ii) Link : The link field contains the address of the next data item in the linked list. It is called next field or pointer field. For example:



[Fig:- Singly linked list of integer value]

The node in the linked list are not stored contiguously in the memory. Memory for each node can be allocated dynamically whenever need arises. The size of the linked list can grow and

shrink dynamically. The last node in the linked list contains Null pointer to indicated end of the list. If the head point is NULL then list is empty.

1) Singly Linked List:-

If a node has link only to its successor in the list then such a list is called singly linked list. The problem with this list is that we cannot access the predecessor of node from current node. This can be overcomed in doubly linked list.

For example:

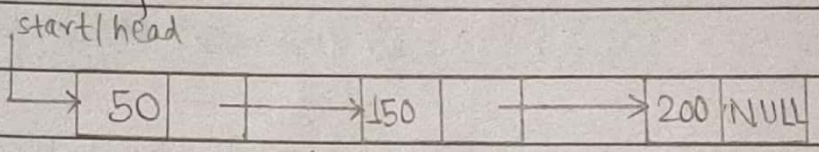


Fig:- Singly linked list of integer value.

Representation of singly linked list:

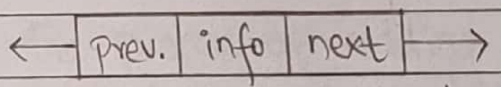
Suppose we want to store list of integer number then the singly linked list can be represented in the memory with following declaration.

```
struct node
{
    int info;
    struct node *next;
};
node *head = NULL;
```

Insertion and deletion in singly linked list can be done in O(1) time if pointer to the node is given, otherwise O(n) time. Search and traversing can be done in O(n) time. Accessing time of element is also O(n).

ii) Doubly Linked List:-

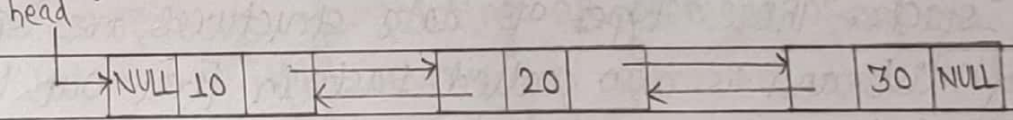
A doubly linked list is one in which all nodes are linked together by multiple number of links which helps in accessing both the successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node in doubly linked list has two link fields. These are used to point the successor node and the predecessor node. Figure below shows the structure of a node in doubly linked list.



[Fig:- A node of a doubly linked list]

The left link point to the predecessor node and right link point to successor node.

For example:



C Representation of doubly linked list:

```
struct node
{
    int info
    struct node *next;
    struct node *prev;
}
node *head = NULL;
```

Insertion and deletion can be done in O(1) time if pointer to the node is given, otherwise O(n). Searching and traversing can be done in O(n) time. Accessing time of an element is also O(n) time.

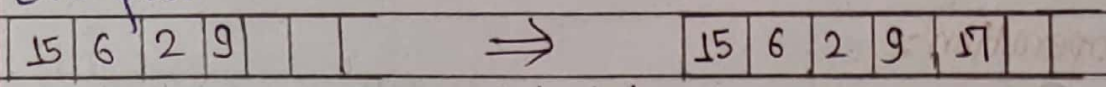
non primitive

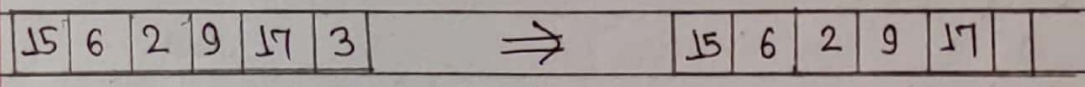
Stack:-

A stack is a non-primitive linear data structure in which addition of new data item and deletion of already existing data item is done from only one end known as top of stack. These types of data structures are special case of list and is also called Last-In-First-Out (LIFO) type of list as last added element will be the first to be removed from the stack. Stack are either represented in array or in singly linked list and in both cases insertion/deletion time is O(1) but search time is O(n).
The following figure shows stack of at most n-element with an array S[1, 2, ..., n] where, S[1] is element

in bottom of stack and S[top] is the element at the top.

Example:





Operations on Stack:

1) **Push:-**

The process of adding the new element in the top of stack is called push operation. The top of stack is increment after every push operation.

2) **Pop:-**

The process of deleting an element from the top of stack is called pop. After every pop operation, the stack top is decremented by 1.

3) **IsFull:-**

This operation is used to check whether the stack is full or not i.e. overflow.

4) IsEmpty:-

This operation is used to check whether the stack is empty or not i.e. stack underflow.

5) Top operation:-

This operation is used to return the current item at the top of stack.

6) Create Empty stack:-

This operation is used to create an empty stack.

Implementation of stack:

1) IsEmpty operation:-

- Returns true if and only if stack is empty: Complexity O(1).

```
Boolean IsEmpty (S)
{
   if (stop == 0)
      return true;
   else
      return false;
}
```

2) IsFull Operation:-

- Returns true if the stack is full: Complexity O(1).

```
Boolean IsFull(S)
{
  if (stop == MAX)
     return true;
  else
     return false;
}
```

3) Push Operation:-

- Insert an item at the top of stack: Complexity O(1).

```
Push (S,x)
{
  if (IsFull(S))
  {
     print "stack overflow";
  }
  else
     {
        stop = stop +1;
        S[S.top] = x;
     }
}
```

4) Pop operation :-

- This deletes the top of stack : Complexity O(1).

```
Pop(s)
{
    if (IsEmpty(s))
    {
      print "stack underflow";
    }
    else
      {
        s.top = s.top - 1;
        return s[s.top+1];
      }
}
```

5) Create an empty stack :-

- Make the empty stack : Complexity O(1).

```
void clear(s)
{
   s.top = 0;
}
```

6) To find the top of stack :-

```
void top(s)
{ return (s.top);
}
```

Queue:-

A queue is a non-primitive linear data structure in which new elements are added at one end called rear and the existing elements are deleted from the other end called front end. The queue is the first-In-First-Out (FIFO) list as it always the first item to be put into the queue is the first item to be removed. The front and rear are also called head and tail respectively. Queue can be implemented by using array or linked list. In array implementation of queue, an array is used to store the data element and in linked list implementation, linked list with pointer to front and rear node are used.

Operation on Queue:

The operation that are performed in a queue are as follows:

1) MakeEmpty(q):-
   To make 'q' as a empty queue. Complexity O(1).

2) Enqueue(q):-
   To insert an item x at the rear of the queue q. Complexity O(1).

3) Dequeue(q):-
   To delete an item from the front of the queue q. Complexity O(1).

4) IsFull(q):-
To check whether the queue q is full. Complexity O(1).

5) IsEmpty(q):-
To check whether the queue q is empty. Complexity O(1).

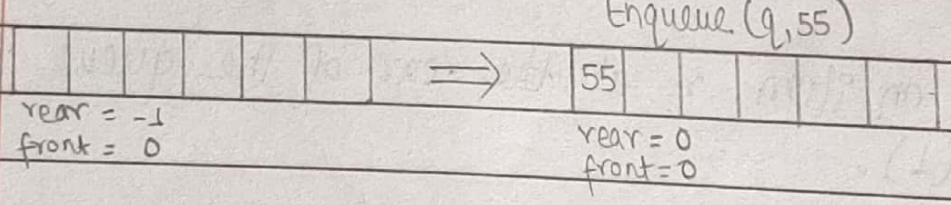
6) Traverse(q):-
To read entire queue i.e. display the content of the queue. Complexity O(n).

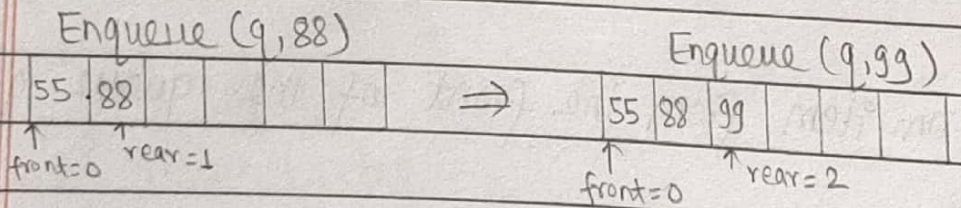
Types of Queue:-

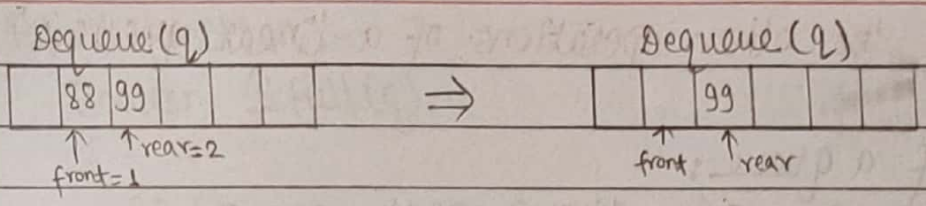
1) Linear Queue:-

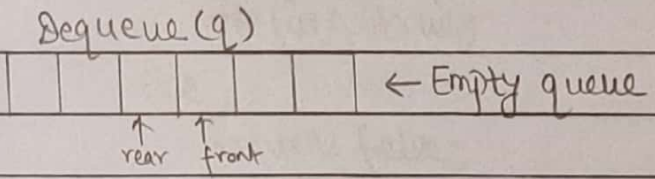
It is the linear array implementation. To enqueue an item, we increase the rear by 1 and put item in that position and to dequeue an item, we take it from the front position and then increase the front by 1.

Example:









Algorithm for enqueue and dequeue in linear queue:

Let queue is of size MAX.

1) Insertion (enqueue)

```
set front = 0, rear = -1
if rear >= MAX-1
    print "Queue overflow" and return
else
    set rear = rear + 1
        queue [rear] = item;
```

2) Deletion (Dequeue)

```
if rear < front
    print "Queue is empty" and return
else
    item = queue [front];
    front = front + 1;
    return item;
stop.
```

Implementing the other operations of a linear queue:-

1) Declaration of a queue:-

```
#define MAXQUEUE    //size of the queue
struct queue
{
   int front;
   int rear;
   int item [MAXQUEUE];
};
```

2) MakeEmpty Queue:-

```
void MakeEmpty (q)
{
   q.rear = -1;
   q.front = 0;
}
```

3) Is Empty Function:-

```
Boolean IsEmpty (q)
{
   if (q.rear < q.front)
      return true;
   else
      return false;
}
```

4) IsFull function:-

```
Boolean IsFull(q)
{
    if (q.rear == MAXQUEUE-1)
        return true;
    else
        return false;
}
```

2) Circular Queue:-

A circular queue is one in which the insertion of new element is done at very first position of the queue if last location of the queue is full. A circular queue overcomes the problem of unutilized space in linear queue implementation of an array.

Non-linear (or Hierarchical) Data structure:

When the data elements are organized in some arbitrary fashion without any sequence such data structures are called non-linear data structure.
Examples of non-linear data structures are tree and graph.

Tree Data Structure:-

A tree is a non-linear data structure which organizes data in hierarchical structure. A tree is defined as a finite set of one or more data items (nodes) such that there is a

special data item called the root of the tree and its remaining data items are partitioned into a number of mutually exclusive subset each of which itself is a tree called subtree.
For example:

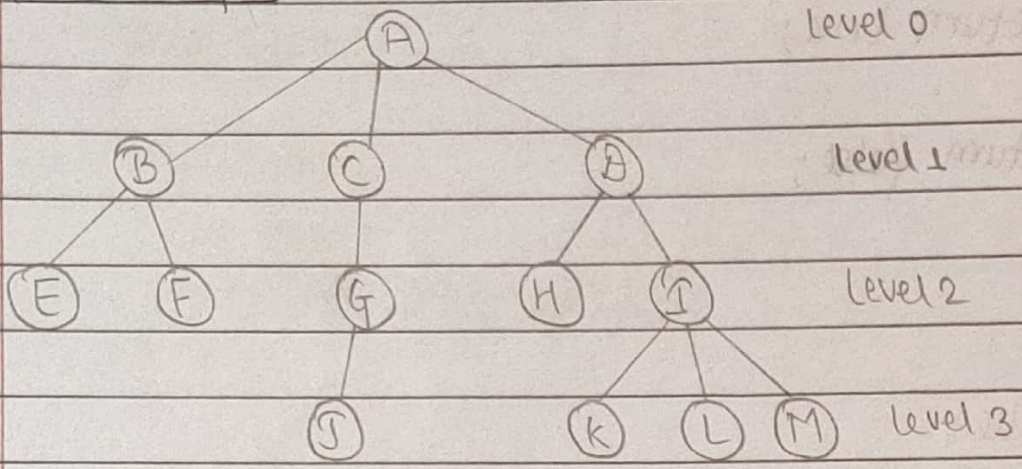


Fig : Tree

Tree Terminology :-

i) **Root:-**
It is the first data item in hierarchical arrangement. In above figure, A is the root.

ii) **Node:-**
Each data item in a tree is called a node.

iii) **Degree of Node:-**
The degree of a node is the number of children of that node. In above figure, degree of A is 3.

iv) **Degree of tree:-**
It is the maximum degree of nodes in a given tree.

The degree of above tree is 3.

v) Terminal node or leaf node:-

The node with degree 0 is called terminal node or leaf node. i.e. the node with no child. In above figure; E, F, G, H, K, L, M are terminal nodes.

vi) Non-terminal or internal nodes:-

Any node except root where degree is not zero is called non-terminal node or internal node.

vii) Siblings:-

The children nodes of same parents are called siblings.

viii) Depth of node:-

Depth of a node is a path-length from root node to that node. The root node has the depth 0.

ix) Height of node:-

Number of edge in the longest path from that node to a leaf. Height of a leaf node is 0.

x) Height of tree:-

Height of the tree is the height of the root node.

xi) Ancestor and Descendent:-

If there is a path from A to another node B then A is ancestor of B and B is the decendent of A.

xiii) Level of node :-

The level of node is 0 if it is root otherwise it is 1 more than its parents.

Binary Tree :-

A binary tree is a finite set of data items in which each node in the tree has at most two children.

Example:

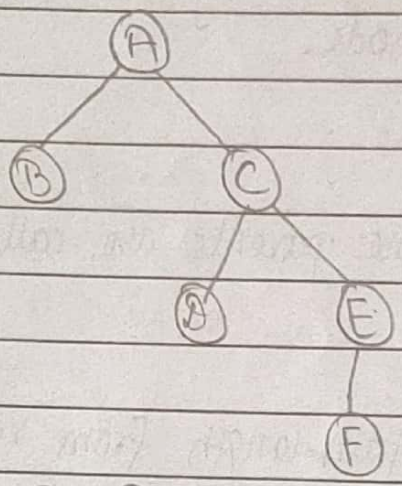


Fig: Binary Tree

Types of Binary Tree:

1) Strictly Binary Tree :-

If every non-leaf node in a binary tree consists of non-empty left subtree and right subtree then such a tree is called strictly binary tree. In another word, a binary tree in which each non-leaf node must have both left and right child is called strictly binary tree.

Example:

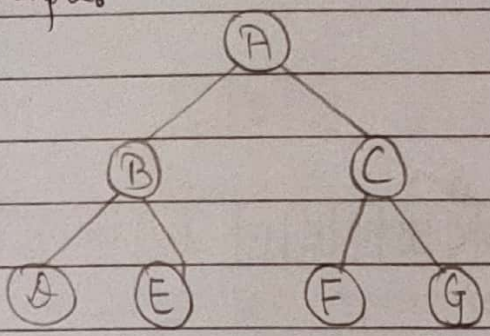


Fig: Strictly Binary Tree

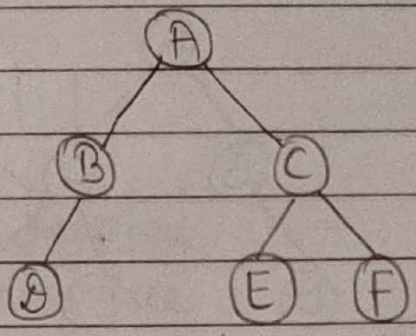


Fig: Non-strictly binary tree.

2) Complete Binary Tree:-

A complete binary tree of depth d is strictly binary tree all of whose leaves are of level d.

For example:

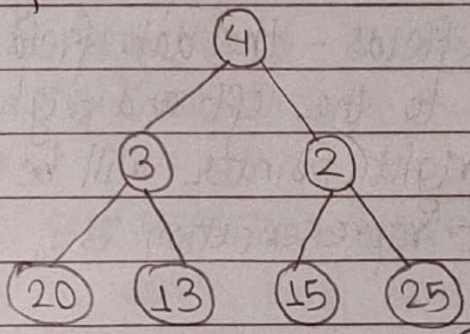


Fig:- Complete binary Tree

Representation of binary tree:

1) Array Representation:-

An array is used to store the node of binary tree. A root node is always in index 0 then in successive memory location, the left child and right child are stored. To store the element of binary tree using array, define an array of size $2^{d+1} - 1$ i.e. the total number of nodes in complete binary tree.

Example:

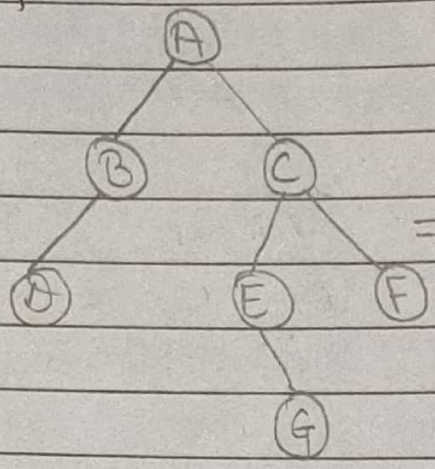


Fig: Binary tree

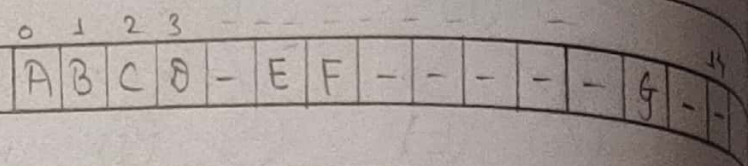


Fig:- Array representation of given binary tree

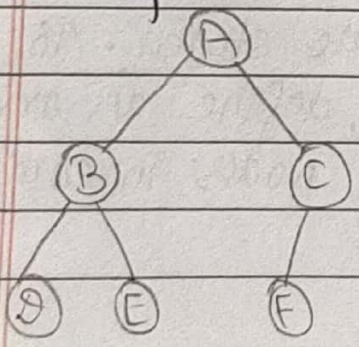
2) Linked List Representation:-

In linked list representation of binary tree, each node in the binary tree has 3 fields - the data field and two pointer fields that point to the left and right subtree. For a leaf node, left and right pointer will be Null.

Declaration of linked list representation is:

```
struct btree
{
    int info;
    struct btree *left;
    struct btree *right;
};
struct btree *root = NULL;
```

Example:



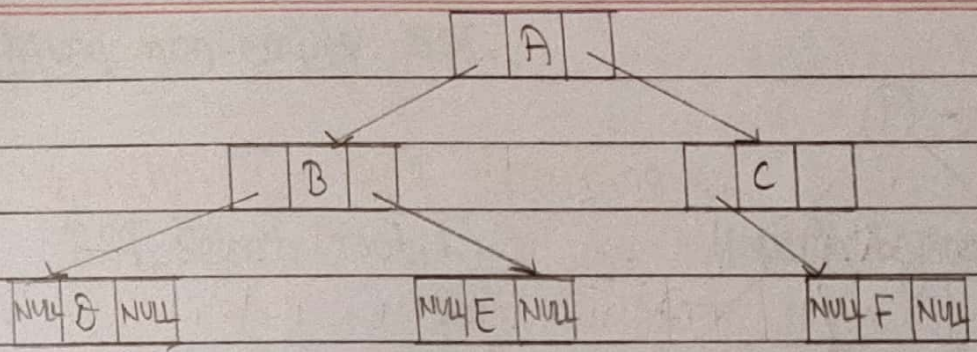


Fig: Linked list representation of binary tree

Binary Search Tree (BST):-

A binary search tree is a binary tree with special property called BST property which is given as follows:

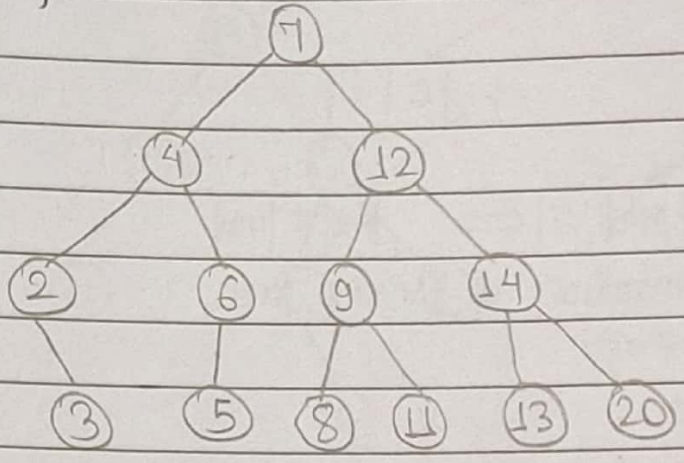
For each node x and y if y belongs to left sub-tree of x then key at y is less than key at x and if y belongs to right sub-tree of x then key at y is greater than key at x.

We can represent binary search tree by linked data structure in which each node has following attributes:

1) P, left, right : which are the pointer to the parent, left child and right child respectively.

11) key ; which is the value stored at a node.

Example:



Operations on BST:-

Following operations can be done in BST:

1) Search (T, k):-
Search for the key k in the tree T, if k is found in some node of the tree then return true, otherwise return false.

2) Insert (T, k):-
Insert a new node with value k in the tree T such that property of BST is maintained.

3) Delete (T, k):-
Delete the node with value k in the tree T such that the property of BST is maintained.

4) Find_min (T), find_max (T):-
Find the maximum and minimum element from the

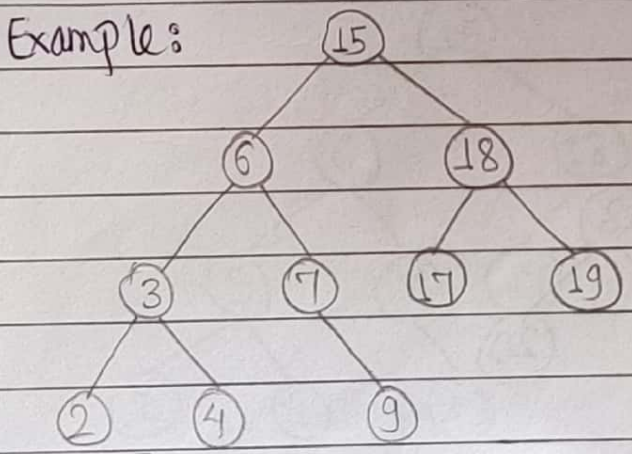
given non-empty BST.

Algorithm for BST Searching :-

```
BST_Search (root, k)          //search begins at root
{
   if (root == null)
      return null;
   else if (root == k)
      return root;
   else if (k < root)
      BST_Search (root.left, k);
   else
      BST_Search (root.right, k);
}
```

The complexity of the algorithm is O(h) where h is depth, i.e. height of tree.

Example:



k = 13

15 → 6 → 7 → 9 → null ; path unsuccessful

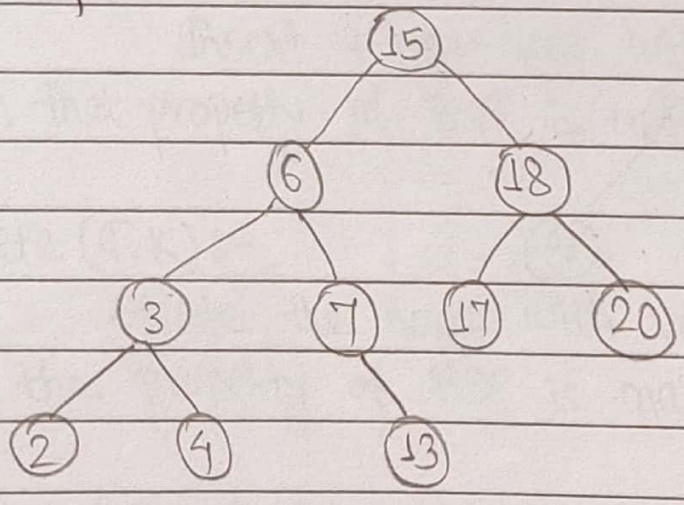
Algorithm for finding max and min in a BST :-

i) Finding Maximum:

We can find the max element by following the right child pointer of root until we encounter NULL.

```
find_max (root)
{
    if (root == null)
        return NULL;
    else
      {
        while (root.right ≠ null)
            root = root.right;
        return root;
      }
}
```

Example:



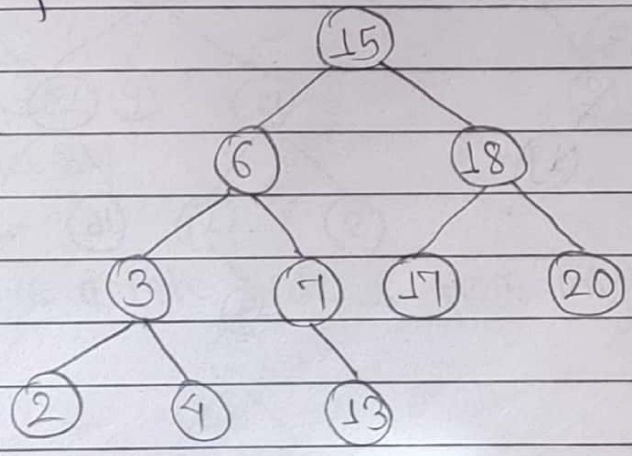
Maximum = 20

ii) Finding Minimum:

We can find the minimum element in BST by following the left child pointer of root until we encounter null.

```
find_min(root)
{
    if (root==null)
        return NULL;
    else
    {
        while (root.left ≠ null)
            root = root.left
        return root;
    }
}
```

Example:



Minimum = 2.

Algorithm for insertion of a node in BST :-

Let x be the key value of a node to be inserted in BST

i) If there is empty tree then create a new node n, set data item of this node as x and make this node as root node.

ii) Otherwise, compare x with value stored in root node.

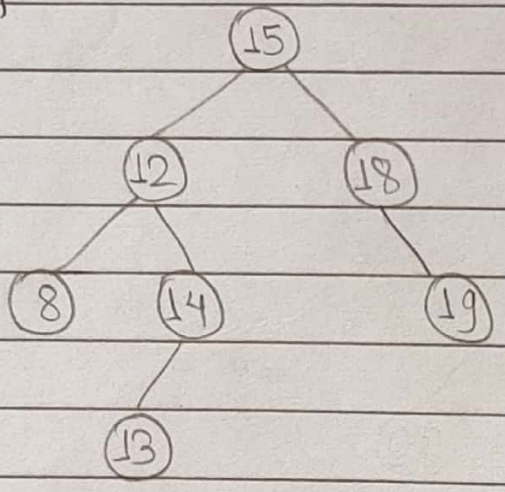
iii) if (x == root.key) then
Just stop because this value is already exist.

iv) if (x < root.key) then
Insert x into left subtree of root node recursively.

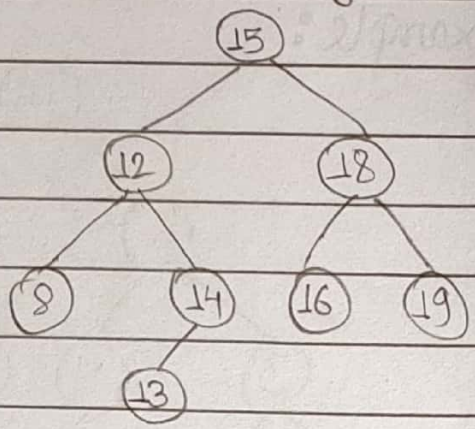
v) else
Insert x into the right subtree of root node recursively.

vi) Stop.

Example :



To insert key = 16

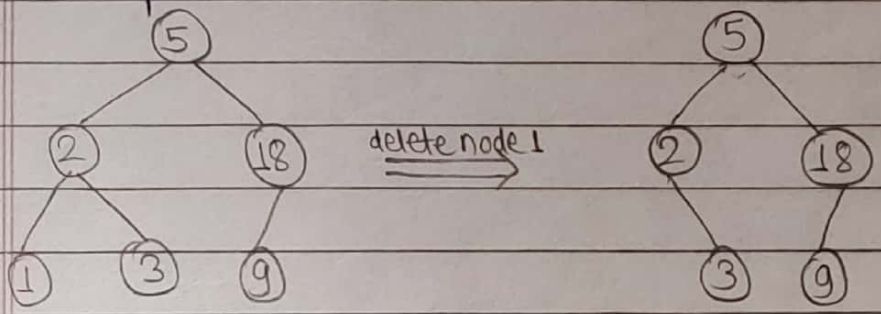


Exam ✓✓

Algorithm for deletion of a node in BST :-

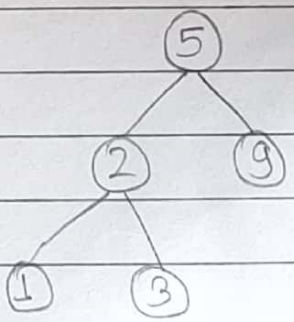
1. Start
2. If a node to be deleted is a leaf node at the left side then simply delete the node and set null pointer to its parents left pointer.
3. If a node to be deleted is a leaf node at right side then simply delete the node and set null pointer to its parent's right pointer.
4. If a node to be deleted has one child then connect its child pointer with its parents pointer and delete it.
5. If a node to be deleted has two children then replace node to be deleted with its inorder successor.
6. End

Example:



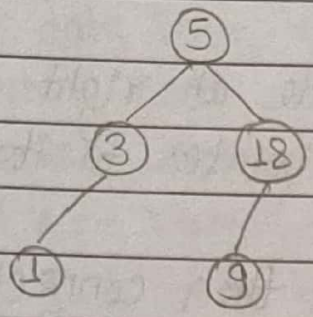
Original BST

⇒ If we delete node 18 from original BST.



⇒ If we want to delete 2 from original BST.

- the inorder traversing : 1 2 3 5 9 18
- Here inorder successor of 2 is 3
- So,



UNIT 2:

Sorting:-

Sorting is the process of arranging the item in the list in some order i.e. either ascending or descending order.

- Let a[n] be an array of n elements $a_0, a_1, \ldots, a_{n-1}$ in memory.
- The sorting of array 'a' means arranging the elements of 'a' either in increasing or decreasing order.

i.e. $a_0 \leq a_1 \leq a_2 \leq \ldots \leq a_{n-1}$

Example:

Consider the list of values

7 9 3 6 8 7 18

After sorting these values

3 6 7 7 8 9 18

Types of Sorting:-

There are two basic categories of sorting method on the basis of memory use.

i) Internal Sorting
ii) External Sorting

1) Internal sorting:-

In internal sorting, all the data to sort are stored in memory at all times while sorting. Internal sorting is applied when entire collection of data to be sorted

is small enough so that sorting takes place within the main memory.

ii) External Sorting:-

In external sorting, data is stored outside memory and only loaded into memory in small chunks. External sorting is usually applied in a data which cannot fit in main memory.

use one array to sort

Inplace Sorting algorithm:

Inplace sorting algorithm is one that use no additional array of storage. For example; bubble sort, insertion sort, selection sort.

Stable Sorting algorithm:

A sorting algorithm is stable if two elements that are equal remains in same relative position after sorting is completed. For example; merge sort.

Applications of Sorting:

Sorting has many applications. Some of them are as follows:

i) Searching
ii) Closest pair finding
iii) Median and selection problems

Types of internal sorting

Types of internal sorting are:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort.

1. Bubble Sort :-

The basic idea of this sort is to pass through array sequentially several times. Each pass consists of comparing each element in array with its successor (i.e. a[i] with a[i+1]) and interchanging the two elements if they are not in the proper order. Bubble sort gets its name because smaller elements bubble towards the top of the list.

Algorithm:

```
Bubble_Sort(A,n)
{
  for(i=0; i<n-1; i++)
  {
    for(j=0; j<n-i-1; j++)
    {
      if (A[j]>A[j+1])
      {
        temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
      }
    }
  }
```

Example:

1) Sort the following data using bubble sort.

A[8] = {25, 57, 48, 37, 12, 92, 86, 33}

⇒

Original array:- 25 57 48 37 12 92 86 33

Pass 1 : 25 48 37 12 57 86 33 [92]

Pass 2 : 25 37 12 48 57 33 [86] 92

Pass 3 : 25 12 37 48 33 [57] 86 92

Pass 4 : 12 25 37 33 [48] 57 86 92

Pass 5 : 12 25 33 [37] 48 57 86 92

Pass 6 : 12 25 [33] 37 48 57 86 92

Pass 7 : 12 [25] 33 37 48 57 86 92

2)

10 52 33 7 2 9 32 40 25

⇒

Original array:- 10 52 33 7 2 9 32 40 25

Pass 1 : 10 33 7 2 9 32 40 25 [52]

Pass 2 : 10 7 2 9 32 33 25 [40] 52

Pass 3 : 7 2 9 10 32 25 [33] 40 52

Pass 4 : 2 7 9 10 25 [32] 33 40 52

Pass 5 : 2 7 9 10 [25] 32 33 40 52

Pass 6 : 2 7 9 [10] 25 32 33 40 52

Pass 7 : 2 7 [9] 10 25 32 33 40 52

Pass 8 : 2 [7] 9 10 25 32 33 40 52.

Analysis of Bubble Sort:

Time Complexity:-

For $i = 0$, inner loop executes $(n-1)$ times

For $i = 1$, inner loop executes $(n-2)$ times.

⋮ and so on.

So, Time complexity $= (n-1) + (n-2) + \ldots + 1$

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

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

Space Complexity:
Space complexity = O(n).

2. Selection Sort:-

The selection sort algorithm sorts a list by selecting successive elements in order and placing into their proper sorted positions. Let A[n] be a linear array of n elements then the selection sort works as follows:

Pass 1 : Find the location loc of the smallest in the list of n elements A[0], A[1], ... A[n-1] and then interchange A[loc] and A[0].

Pass 2: Find the location loc of the smallest in the sublist through 1 to n-1, then interchange A[1] and A[loc] such that A[0], A[1] is sorted since A[0] ≤ A[1]. and so on.

Algorithm:

```
Selection_sort (A, n)
{
  for (i=0 ; i<n ; i++)
   {
     min = A[i];
     loc = i;
     for (j= i+1; j<n ; j++)
       {
         if (A[j]<min)
           {
```

```
            min = A[j];
            loc = j;
        }
    }
    if (loc != i)
    {
        swap(A[i], A[loc]);
    }
  }
}
```

Example: (1) Suppose we have a list which contains the following elements.

16 15 2 13 6

⇒

Initially: 16 15 2 13 6

Pass 1 : 16 15 2 13 6 loc = 2
(16: min, 2: loc)

Pass 2 : 2 15 16 13 6 loc = 4
(15: min, 6: loc)

Pass 3 : 2 6 16 13 15 loc = 3
(16: min, 13: loc)

Pass 4 : 2 6 13 16 15 loc = 4
(16: min, 15: loc)

Pass 5 : 2 6 13 15 16
(16: min)

2) A[8] = {25, 57, 48, 37, 12, 92, 86, 33}

⇒

Initially :	25	57	48	37	12	92	86	33	
Pass 1 :	25 (min)	57	48	37	12 (loc)	92	86	33	loc = 4
Pass 2 :	12	57 (min)	48	37	25 (loc)	92	86	33	loc = 4
Pass 3 :	12	25	48 (min)	37	57	92	86	33 (loc)	loc = 7
Pass 4 :	12	25	33	37 (min)	57	92	86	48	i = loc = 3
Pass 5 :	12	25	33	37	57 (min)	92	86	48 (loc)	loc = 7
Pass 6 :	12	25	33	37	48	92 (min)	86	57 (loc)	loc = 7
Pass 7 :	12	25	33	37	48	57	86 (min)	92	i = loc = 6
Pass 8 :	12	25	33	37	48	57	86	92 (min)	

Analysis of Selection Sort:

Time Complexity :-

When i=0, the inner loop executes n-1 times.

When i=1, the inner loop executes n-2 times.

and so on.

So, the total time complexity,

$$T(n) = (n-1)+(n-2)+(n-3)+\ldots+2+1$$

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

Space Complexity:
Space complexity = O(n).

3. Insertion Sort :-

Insertion Sort is one that sorts a set of values by inserting values into an existing sorted file. Suppose an array A with n elements A[0], A[1], ..., A[n-1] is in memory. The insertion sort algorithm scan A from A[0] to A[n-1] inserting each element A[k] into its proper position in the previously sorting sub-array A[0], A[1], ..., A[k-1]. i.e.

Pass 1: A[1] is inserted either before or after A[0]. so that A[0], A[1] is sorted.

Pass 2: A[2] is inserted into its proper place in A[0], A[1] that is before A[0], between A[0] & A[1] or after A[1], so that: A[0], A[1], A[2] is sorted.

- - - -
: : : :

Pass N: A[n-1] is inserted into its proper place in A[0], A[1], ... A[n-2], so that: A[0], A[1], ... A[n-1], is sorted.

Algorithm:-

Let A[n] be an array of size n. The following function sorts the elements of the array using the insertion sort.

```
Insertion_sort(A,n)
{
  for (i=1; i<n ; i++)
   {
     key = A[i];
     j = i-1;
     while (j>=0 && key<A[j])
     {
        A[j+1] = A[j]
        j=j-1;
     }
     A[j+1] = key
   }
}.
```

Example:-

1) Consider the following array with 6 elements.

5 2 4 6 1 3

⇒

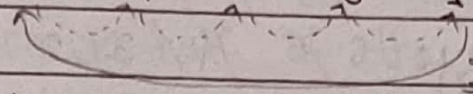
Original array: 5 2 4 6 1 3

Pass 1: 5 2 4 6 1 3 key=2

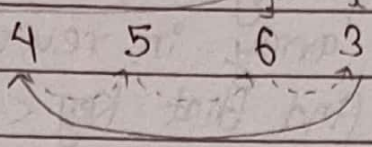
Pass 2: 2 5 4 6 1 3 key=4

Pass 3 : 2 4 5 6 1 3 key = 6

Pass 4 : 2 4 5 6 1 3 key = 1



Pass 5 : 1 2 4 5 6 3 key = 3



Pass 6 : 1 2 3 4 5 6

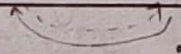
2) Consider the following array with 8 elements.
A[8] = {25, 57, 48, 37, 12, 92, 86, 33}

⇒

Original array : 25 57 48 37 12 92 86 33

Pass 1 : 25 57 48 37 12 92 86 33 key = 57.

Pass 2 : 25 57 48 37 12 92 86 33 key = 48



Pass 3 : 25 48 57 37 12 92 86 33 key = 37

Pass 4 : 25 37 48 57 12 92 86 33 key = 12

Pass 5 : 12 25 37 48 57 92 86 33 key = 92

Pass 6 : 12 25 37 48 57 92 86 33 key = 86

Pass 7 : 12 25 37 48 57 86 92 33 key = 33

Pass 8 : 12 25 33 37 48 57 86 92

Analysis of insertion Sort:-

Time complexity:

a) Worst case complexity: 6 5 4 3 2 1
Elements of array in reverse order. In worst case we can always find that Key < A[j] in while loop test

so,
For i=1, inner loop executes 1 time
For i=2, inner loop executes 2 times
For i=3, inner loop executes 3 times
⋮
For i=n-1, inner loop executes n-1 times.

Thus,
Total time complexity = $1+2+3+\ldots+(n-1)$
$= \frac{n(n-1)}{2}$
$= O(n^2)$.

b) Best Case Complexity: 1 2 3 4 5 6
Array elements are already sorted. So,
For i=1, inner loop executes 1 time
For i=2, inner loop executes 1 time
For i=3, inner loop executes 1 time
⋮
For i=n-1, inner loop executes 1 time

Thus,
Total time complexity = 1+1+1+.....+1 (n-1 times)
$= (n-1)$
$= O(n)$.

c) Average Case Complexity:

On average case, half of the elements in A are less than Ai and half of the elements are greater. On average we check half of the sub-array so, total time complexity = O(n).

Space Complexity:

Since no extra space besides three variables is needed for sorting so, space complexity = O(n).

2.1) Divide & Conquer:-

2068 2071

Concept of Divide & Conquer approach:-

Divide & conquer paradigm is an important problem solving technique that makes the use of recursion. Divide and conquer paradigm breaks the problem into several sub-problems that are similar to original problem but smaller in size, solves sub-problems to create the solution of original problem. The divide and conquer paradigm involves 3 steps at each level of recursion.

i) Divide the problem into number of sub-problems that are smaller instances of same problem.
ii) Conquer sub-problems by solving them recursively.
iii) Combine the solutions to the sub-problems into a solution for the original problem.

Applications:

i) The divide and conquer paradigm is the basis of efficient algorithm for all kinds of problem such as sorting (merge sort and quick sort), searching (binary search), etc.

ii) The application of divide & conquer approach is in the problem domain involving the sorting, searching, closest pair finding, multiplying large numbers and other problems (that can recursively define the similar sub-problems.)

iii) An algorithm developed using divide and conquer paradigm are called divide and conquer algorithms and analysis of such algorithms needs recursion since divide and conquer algorithm use recurrences.

Pictorial representation of divide and conquer paradigm is:

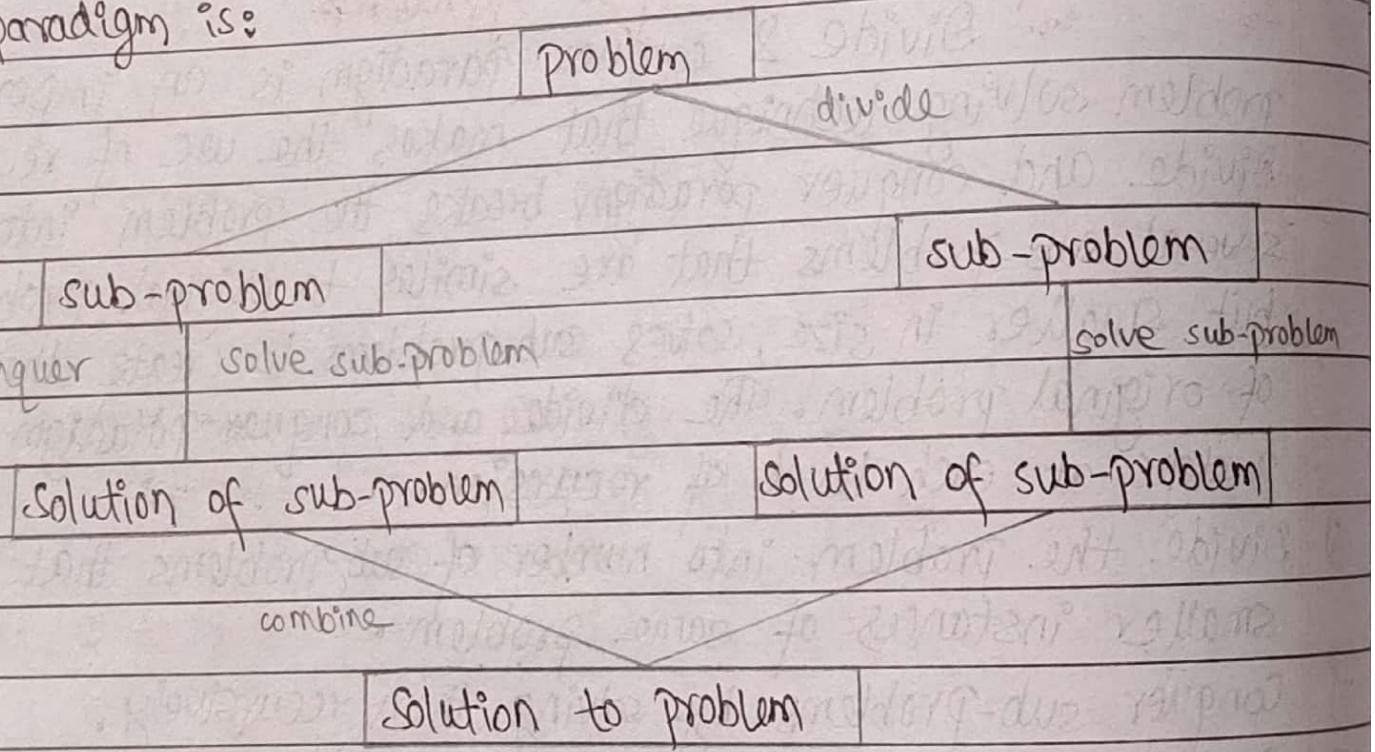


Fig:- Pictorial representation of divide and conquer paradi

Merge Sort :-

Merge Sort is an efficient sorting algorithm based on the divide and conquer paradigm. In this algorithm, an input array of size n is divided recursively into two halves until the size of sub-list equal to 1. After the complete division process, the algorithm merge two adjacent list into single list recursively. More precisely, to sort an array A[n], merge sort works as follows:

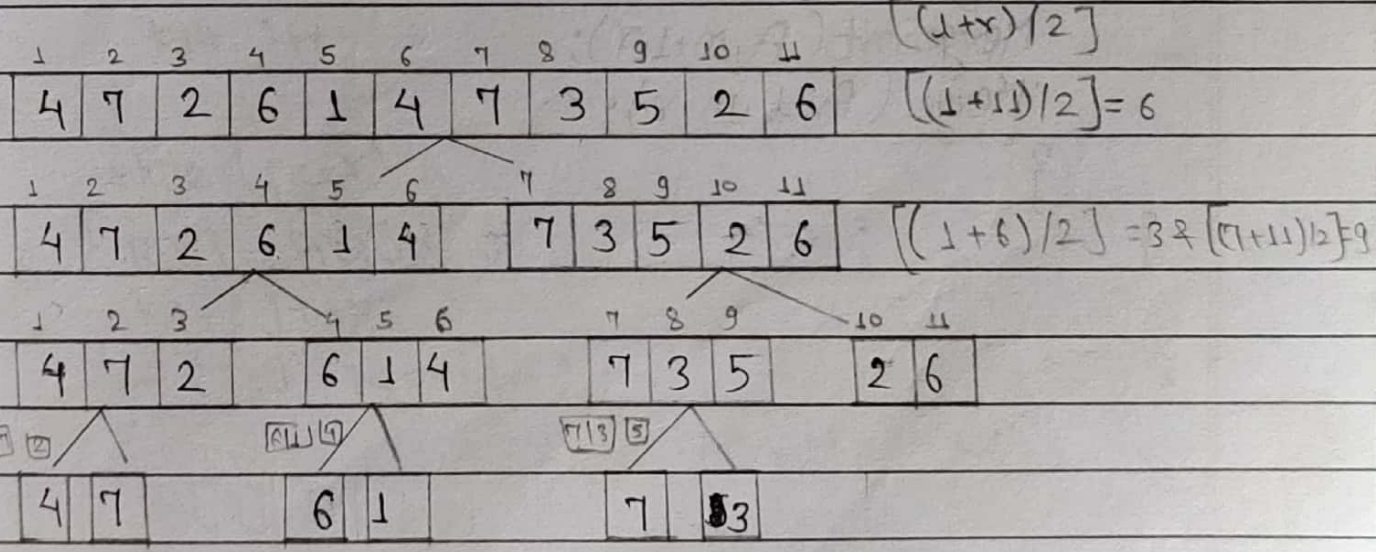
i) Divide :- Divide n elements sequence to be sorted into two sub-sequences of size n/2 element each.

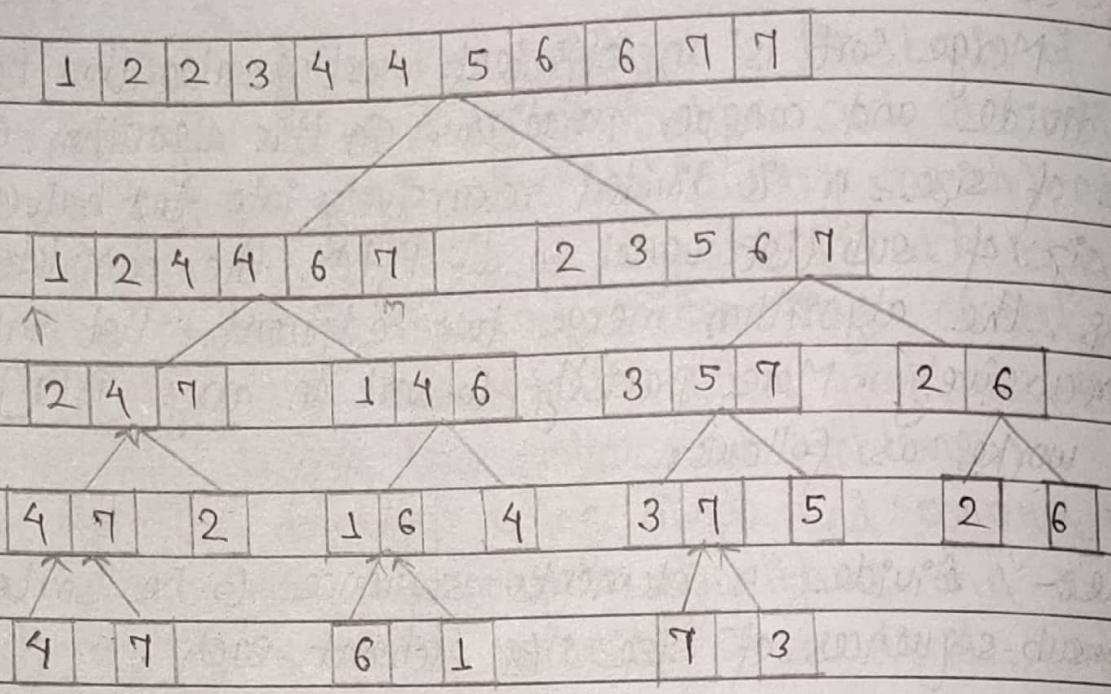
ii) Conquer :- Sort the sub-sequences recursively using merge sort.

iii) Combine :- Merge the two ~~odd~~ sorted subsequences to produce sorted answer.

Example:

A[] = {4, 7, 2, 6, 1, 4, 7, 3, 5, 2, 6}





Algorithm:

```
MergeSort (A,l,r)
{
  if (l<r)
  {                                  {divide}
    m = ⌊(l+r)/2⌋
    MergeSort (A,l,m);
    MergeSort(A,m+1,r);
    Merge (A,l,m,r);
  }
}
```

Algorithm for Merge Operation:

```
Merge (A, l, m, r)
{
  B[], i=l, j=m+1, k=[illegible]l
  while (i<=m && j<=r)
  {
    if (A[i] <= A[j])
    {
      B[k] = A[i];
      i++, k++;
    }
    else
    {
      B[k] = A[j];
      j++, k++;
    }
  }
  while (i<=m)
  {
    B[k] = A[i]
    k++, i++;
  }
  while (j<=r)
  {
    B[k] = A[j];
    k++, j++;
  }
  for (i=l; i<=r; i++)
  {
    A[i] = B[i];
  }
}
```

Conquer & Merge

Analysis of Merge sort:-

a) Time complexity:

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

$T(n) = 2T(n/2) + O(n)$; $a=2$, $b=2$

$n^{\log_b a} = n^{\log_2 2} = n = O(n)$

$\therefore T(n) = \theta(n^{\log_b a} \cdot \log n)$

OR

Let number of items in the input array is 'n' and T(n) be the running time of merge sort on 'n' number of input then recurrence relation of merge sort is

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

Now, by using Master's theorem,

$a = 2, b = 2$

$f(n) = O(n)$

Now,

$n^{\log_b a} = n^{\log_2 2} = n = O(n)$

Then, by using case2 of Master's theorem.

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

$= \theta(n \log n)$

b) Space complexity:

It uses one extra array and some variables during sorting so, space complexity $= 2n + c$

$= O(n)$.

2068

Quick Sort :-

Quick sort is an in-place sorting technique based on divide and conquer approach. The main idea of a quick sort is recursively partitioning the array about an element called pivot element. An array A[] is said to be partitioned about an element A[i] if all elements to the left of A[i] are less than or equal to A[i] and all elements to the right of A[i] are greater than A[i].

For example:-

9 7 13 15 20 18 19 70

This array is partitioned about 15 but not partitioned about 20. The partitioning step of quick sort rearrange the pivot element into its proper sorted position in that array so the recursive partitioning will sort the whole array. More precisely, given an array A[p....r] to sort, the three step of divide and conquer process for quick sort is:

I) **Divide**:- Partition the array A[p....r] into a array A[p....q-1] and A[q+1....r] such that element of A[p....q-1] are less than or equal to A[q] and elements of A[q+1....r] are greater than A[q]. We need to find index q.

II) **Conquer**:- Sort the 2 sub array A[p....q-1] and A[q+1....r] recursively by calling quick sort.

III) **Combine**:- Because subarray are sorted in-place, no additional work ~~is~~ to combine them.

20/62
20/10

Algorithm :-

```
QuickSort (A,p,r)
{
    if (p<r)
    {
     pivot = Partition (A,p,r);
     QuickSort (A,p, pivot-1)
     QuickSort(A, pivot+1, r)
    }
}

Partition (A,p,r)
{
   i=p , j=r;
  pivot = A[p];
  while (i<j)
  {
      while (A[i] <=pivot)
          i++;
      while (A[j] > pivot)
          j--;
       if (i<j)
          swap (A[i], A[j]);
  }
   A[p] = A[j]
  A[j] = pivot
  return j;
}
```

Example:

A[] = {5, 3, 2, 6, 4, 1, 3, 7}

```
5  3  2  6  4  1  3  7        pivot = 5
i                 j

5  3  2  6  4  1  3  7        swap (A[i], A[j])
      i           j

5  3  2  3  4  1  6  7
      i           j

5  3  2  3  4  1  6  7        exchange (A[i], pivot)
               j  i

1  3  2  3  4  (5) 6  7
```

One partition is completed.

Now, Apply the quick-sort on left half.

```
1  3  2  3  4                 pivot = 1
i           j

1  3  2  3  4                 exchange (A[i], pivot)
j  i

(1) 3  2  3  4
```

Again,

```
3  2  3  4                    pivot = 3
i           j

3  2  3  4
      j  i

3  2  (3) 4
```

Again,

3 2 pivot = 3
i j

3 2 exchange(A[1], pivot)
ij

(2) 3

Again, quicksort on right half

6 7 pivot = 6
i j

6 7 exchange(A[1], pivot)
j i

(6) 7

69

Analysis of Quick Sort:

The running time of quick sort depends on whether the partioning is balanced, which inturns depends on which elements are used for pivoting. The time complexity of partition operation is $\Theta(n)$ since for loop on the algorithm of partion runs n time.

a) Worst Case Complexity:-

The worst case complexity for Quick sort occurs when partitioning procedure produce one sub-problem with n-1 element and one with 0 element. This case happens when array is sorted or reversed sorted. Therefore recurrence relation for this case is:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$

After solving this using substitution method,
$$T(n) = \Theta(n^2)$$

Therefore, the worst case complexity of Quick Sort is $\Theta(n^2)$.

b) Best Case Complexity:-

The best case for Quick Sort occurs when partitioning procedure produce two sub-problems of equal size. Therefore, recurrence relation for this case is:

$$T(n) = T(n/2) + T(n/2) + \Theta(n)$$
$$= 2T(n/2) + \Theta(n)$$

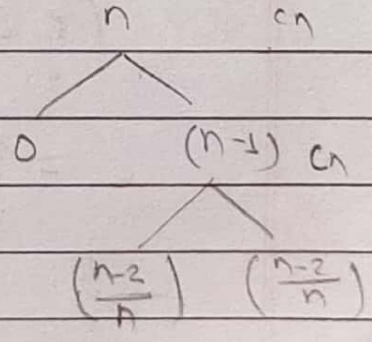
Solving this using master's theorem,
$$T(n) = \Theta(n \log n).$$

Therefore, the best case complexity of Quick Sort is $\Theta(n \log n)$.

c) Average Case Complexity:-

Suppose partitioning alternates between balanced, unbalanced, balanced, unbalanced, balanced,..... then the recurrence relation is:

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



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

$$= 2T(n/2) + \Theta(n)$$

$$= \Theta(n \log n)$$

Therefore, average case complexity for Quick Sort is $\Theta(n \log n)$.

d) Case between worst case and best case :-

Suppose the partitioning algorithm always produces a 9:1 split of array therefore, the recurrence relation for this case is:

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n)$$

Solving this recurrence using recursion tree method,

$$T(n) = \Theta(n \log n).$$

Soln:

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n)$$

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

$$T(n) = (\underbrace{cn + cn + \dots}_{\log_{10/9} n})$$

$$= cn \underbrace{(1 + 1 + \dots + 1)}_{\log_{10/9} n}$$

$$= cn (\log_{10/9} n)$$

$$= n \log_{10/9} n$$

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

2067
2068

Randomized Algorithm:-

The algorithm is called randomized algorithm if its behaviour is determined not only by its input but also by the value produced by random number generator. The beauty of randomized algorithm is that no particular input can produce worst case behaviour of an algorithm. The worst case only depends on random number generated.

2067

Randomized Quick Sort:-

The idea of randomized Quick sort is to partition the input array around a random element i.e. instead of always using A[p] as the pivot, we select a randomly choosen element from array A[p.....r]. We do so by exchanging element A[p] with an element choosen at random.

Algorithm:-

```
Randomized_quicksort(A,p,r)
{
  if (p<r)
    {
      q=RandomPartition(A,p,r)
        Randomized_quicksort(A,p,q-1)
        Randomized_quicksort(A,q+1,r)
    }
}
```

```
RandomPartition (A,p,r)
{
  i = Random(p,r)   // generating a number between p & r.
  exchange ( A[i], A[p]);
  Partition (A,p,r)
}
```

Algorithm for partitioning the array :-

```
Partition (A,p,r)
{
  i=p;
  j=r;
  pivot = A[p];
  while (i<j)
   {
     while (A[i] ≤ pivot)
       i++;
     while (A[j] > pivot)
       j--;
     if (i<j)
        exchange (A[i], A[j])
   }
  A[p] = A[j]
  A[j] = pivot
  return j;
}
```

Example:

A[] = {1, 7, 9, 10, 15, 20, 25}

1	7	9	10	15	20	25
p						r

Here, p<r, so random partition called.

Let random number selected is i=4, then swap(A[i], A[p])

15	7	9	10	1	20	25	pivot = 15
p, i						r, j	

15	7	9	10	1	20	25
p				j	i	r

1	7	9	10	[15]	20	25

One partition is completed.

Now,

Applying the randomized quick sort in left side.

1	7	9	10
p			r

Here, p<r so random partition called.

Let random number selected is i=2 then swap(A[i], A[p])

9	7	1	10	pivot = 9
p, i			r, j	

9	7	1	10
		j	i

1	7	[9]	10

2nd partition is completed.

Applying the randomized quick sort in left side,

1 7

Here, p<r. so random partition called.

Let random number selected is i=1 then swap(A[i], A[p])

7 1 pivot = 7

p,i r,j

7 1 exchange(A[i], A[p])

i,j

1 7

Again, applying randomized quick sort on right side.

20 25

Here, p<r. so random partition called.

Let random number selected is i=1 then swap(A[i], A[p])

25 20 pivot = 25

p,i r,j

25 20 exchange(A[i], A[p])

i,j

20 25

2069(6)

2072 ✓

How Randomized Quick sort works efficiently even in worst case?

The quick sort algorithm select first element in the array as pivot at each step so, worst case happen when input array is already sorted or reverse sorted. At such a case, quick sort partition the array with one partition contains no elements and other partition contains all elements except pivot element, leading to the complexity of $O(n^2)$.

But randomized quick sort algorithm select a pivot randomly from the array at each step so, running time is independent of input order and no specific

input is responsible for the worst case. Since, randomized quick sort choose pivot randomly. there will be high probability of balanced partitioning so randomized algorithm work efficiently even in worst case i.e. when array is already sorted or reverse sorted.

Analysis of Randomized Quicksort :-

a) Worst case complexity :-

Let $T(n)$ be the worst case running time as input of size n, then we have the recurrence relation.

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + O(n)$$

After solving this using substitution method

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

∴ Worst case complexity of quick sort is $O(n^2)$.

Extra:

For clarity:

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

We need to show $T(n) \le cn^2$

Let $T(k) \le ck^2$ for all $k < n$

∴ $T(q) = cq^2$

similarly $T(n-q-1) = c(n-q-1)^2$.

Now,

$$T(n) = \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + O(n)$$

$$= c \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + O(n)$$

Now, this expression $q^2 + (n-q-1)^2$ is maximum over the range $0 \le q \le n-1$ at the end points i.e. 0 or $n-1$, so,

$$T(n) = c(n-1)^2 + O(n)$$
$$= cn^2 - 2cn + c + O(n)$$
$$\le cn^2$$
$$= O(n^2).$$

b) Average case complexity (Expected running time of randomized quick sort) :-

Let $T(n)$ is the expected running time of algorithm. Since, each split occurs with probability $1/n$.

Now, we can give the recurrence relation for algorithm as:

$$T(n) = \frac{1}{n} \sum_{k=0}^{n-1} (T(k) + T(n-k-1)) + O(n)$$

For some $k = 0, 1, 2, \ldots, (n-1)$, $T(k)$ and $T(n-k-1)$ is repeated two times, then

$$T(n) = \frac{2}{n} \sum_{k=0}^{n-1} T(k) + O(n)$$

After solving this recurrence,

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

Therefore, expected complexity of randomized quick sort is $O(n \log n)$.

For clarity:

$$T(n) = \frac{2}{n} \sum_{k=0}^{n-1} T(k) + O(n)$$

Multiply both side by n.

$$n\,T(n) = 2 \sum_{k=0}^{n-1} T(k) + O(n^2) \quad \text{---} \quad (1)$$

Now,

$$(n-1)\,T(n-1) = 2 \sum_{k=0}^{n-2} T(k) + O((n-1)^2) \quad \text{---} \quad (2)$$

Subtracting eq^n (2) from (1)

$$n\,T(n) - (n-1)\,T(n-1) = 2T(n-1) + O(n^2) - O((n-1)^2)$$

$$n\,T(n) - (n-1)\,T(n-1) = 2T(n-1) + c(2n-1)$$

$$n\,T(n) = (n+1)\,T(n-1) + c(2n-1)$$

Dividing both side by $n(n+1)$.

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

$$\frac{T(n)}{(n+1)} \approx \frac{T(n-1)}{n} + \frac{2n}{n(n+1)}$$

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

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

$$= \frac{T(n-3)}{(n-2)} + \frac{2}{(n-1)} + \frac{2}{n} + \frac{2}{(n+1)}$$

$$= \frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \cdots \frac{2}{n+1}$$

$$= 2\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n+1}\right)$$

Heap :-

A heap is a nearly complete binary tree or complete binary tree with following two properties:

i) Structural property
ii) Heap order property

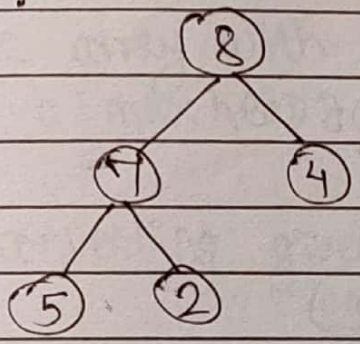
i) Structural property :-

All levels are full, except possibly the last one, which is filled from left to right.

ii) Heap order property :- (≤, ≥)

There is some ordering relation, between any node with its parent.

Example:



(almost complete)

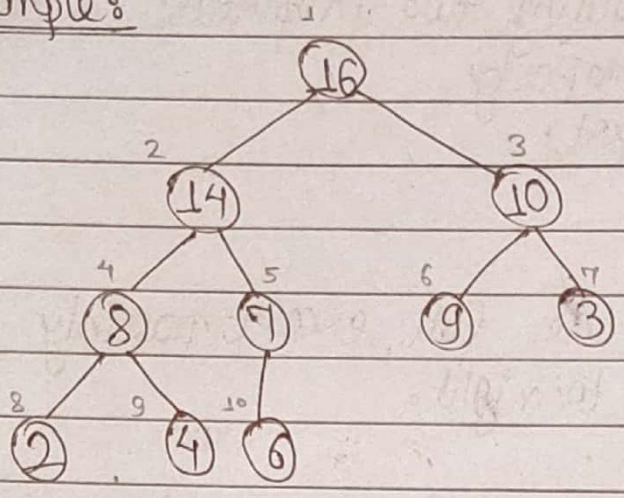
Array Representation of Heap:

A heap can be represented as an array A with the following properties:

- Root of the tree is A[1].
- Left child of A[i] = 2i
- Right child of A[i] = 2i + 1
- Parent of A[i] = A[⌊i/2⌋]
- Heapsize ≤ length(A).

- The element from A $[\lfloor n/2 \rfloor + 1 \ldots n]$ are leaves

Example:



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	6

Types of Heap:-

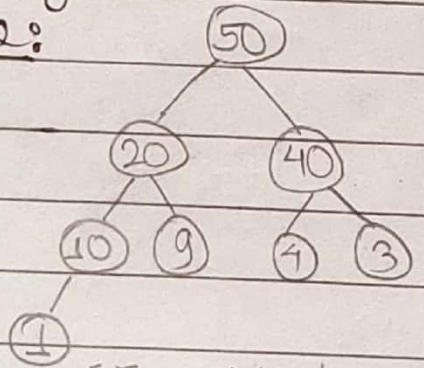
There are two kinds of heap:

1) Max heap
2) Min heap

1) Max heap:-

A heap is called max heap if values stored in any node is greater than or equal to its child node. The largest element in max heap is stored at the root.

Example:

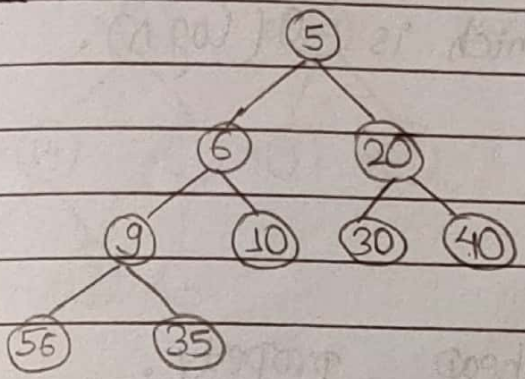


[Fig:- Maxheap]

2) Min heap :-

A heap is called min heap if value stored in any node is less than or equal to its child node. The smallest element in min heap is at root.

Example:



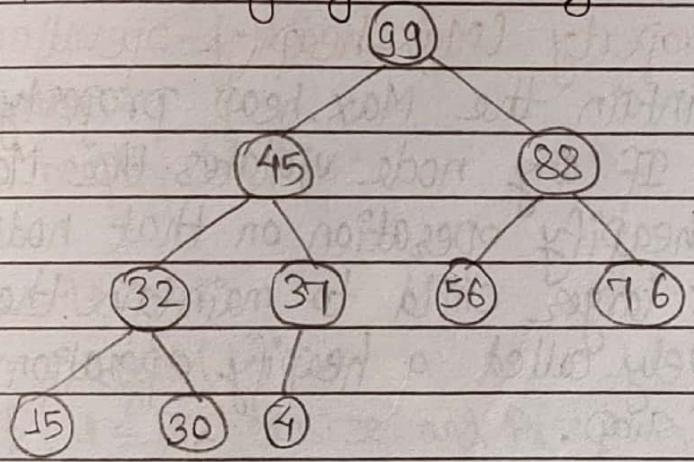
[Fig:- Min heap]

2072

Q? Is the array with values <99, 45, 88, 32, 37, 56, 76, 15, 30, 4> is a maxheap?

Sol:

Now, converting given array into binary tree



This is a maxheap because it satisfies maxheap property i.e. all nodes have the value greater than or equal to the child node.

Height of the Heap :-

Height of the node in a heap to be the number of edges on the largest downward path from node to leaf. Height of the heap to be the height of root. Since heap of n element is based on complete binary tree, its height is $\lfloor \log n \rfloor$ which is $O(\log n)$.

Operations on Heap :

1) Max Heapify :-
   Maintain the Max heap property.

2) Build Max Heap :-
   Create a Max heap from unsorted input array.

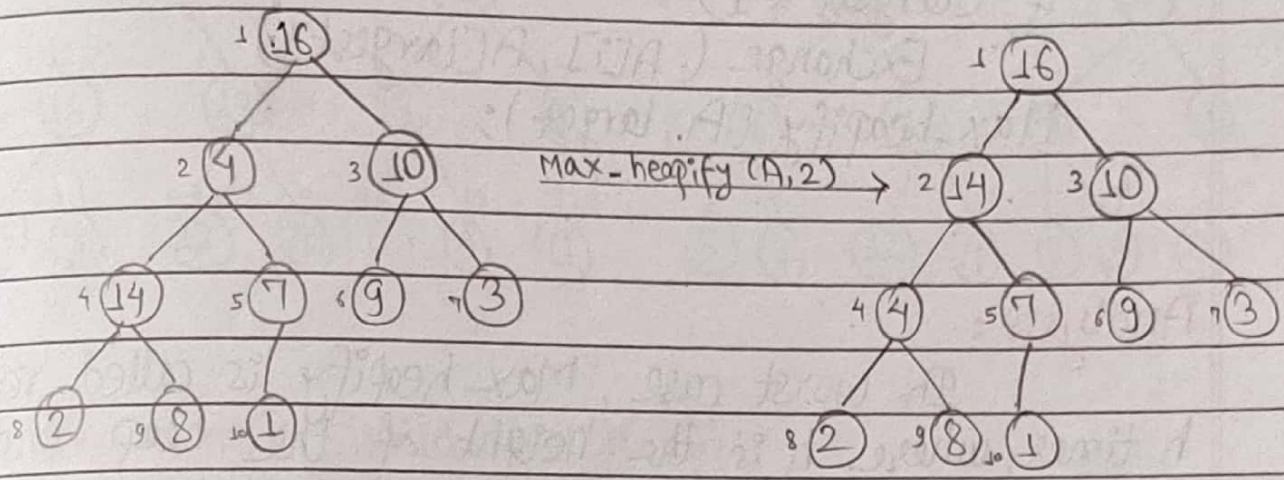
3) Heap Sort :-
   Sort the input array in-place.

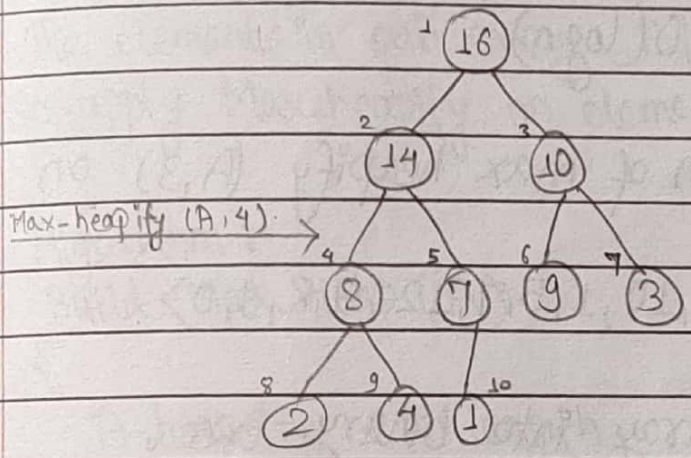
Maintaining Heap Property (Max_heapify operation) :-

In order to maintain the Max_heap property, max_heapify operation is used. If any node violates the Max heap property then Max_heapify operation on that node swap this node with its larger child to maintain the Max heap property and recursively called a heapify operation on largest child after swap.

Example:-

A[] = { 16, 4, 10, 14, 7, 9, 3, 2, 8, 1 }





Algorithm:

```
Max_heapify (A,i)
{
  l = Left(i)
  r = Right(i)
  if ( l ≤ A.heapsize 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], A[largest])
Max_heapify (A, largest);
}.
```

Analysis:

In worst case, Max_heapify is called recursively h times, where h is the height of the heap and since each to the heapify take constant time, so the total time complexity,

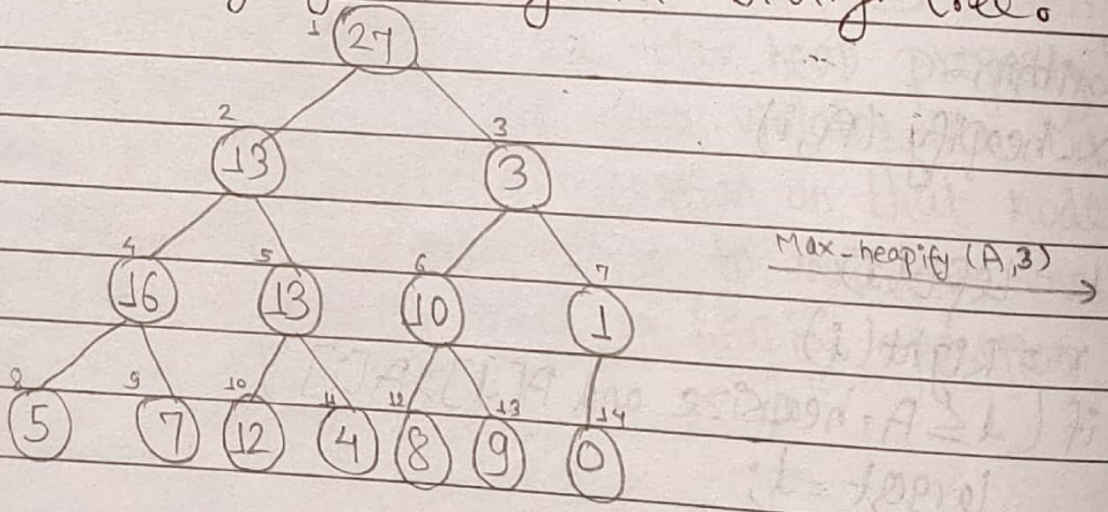
$$T(n) = O(h) = O(\log n)$$

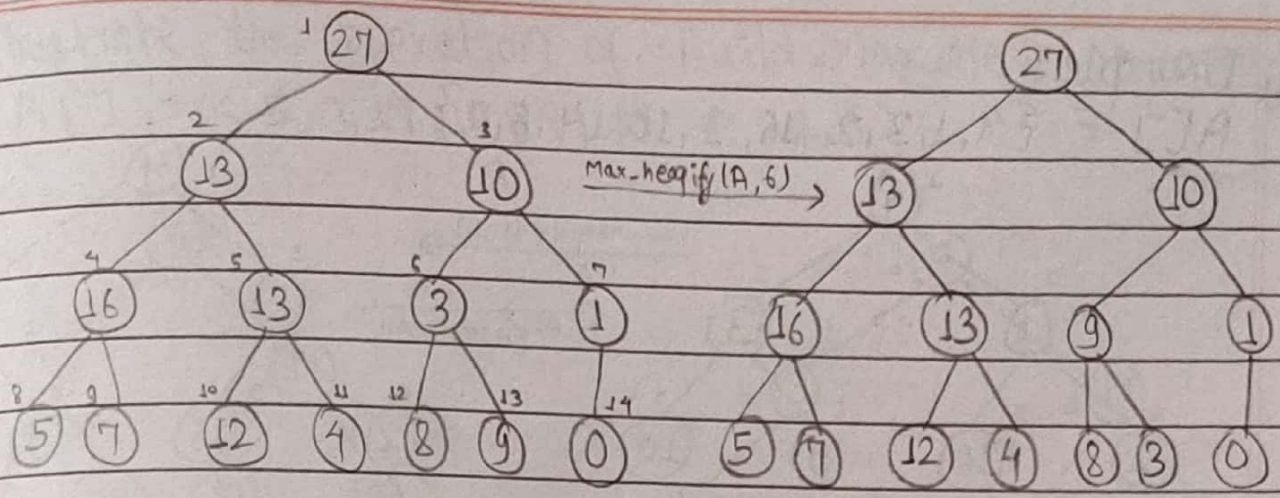
Q: Illustrate the operation of max heapify (A,3) on the array

$A[\,] = \langle 27, 13, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$

Sol:

Now converting given array into binary tree.





Building a heap (Build_Max_Heap):-

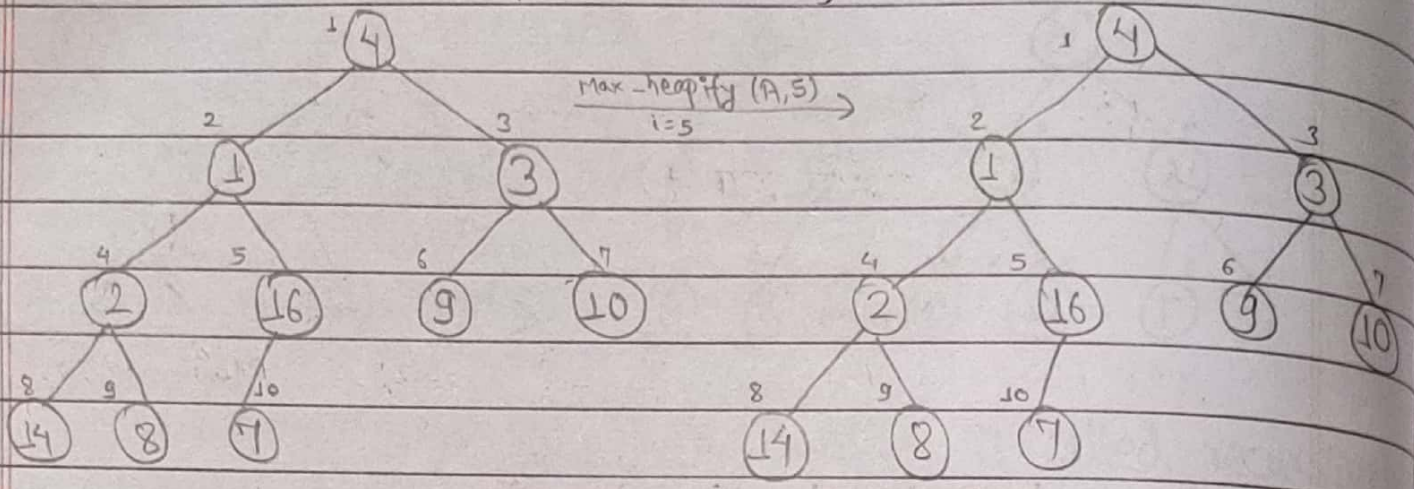
- We can use the Max_heapify operation in bottom up manner to convert an array A[1...n] into max heap.
- The element in sub array A[⌊n/2⌋+1 ... n] are leaves.
- So apply Max_heapify on element between ⌊n/2⌋ to 1.

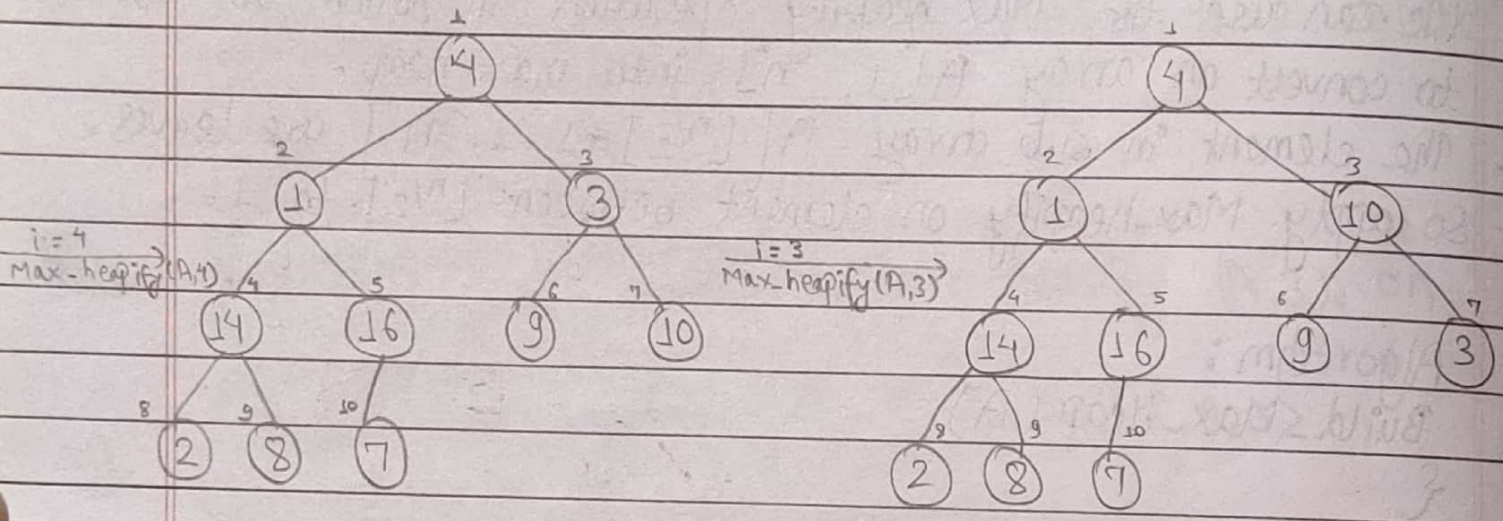
Algorithm:

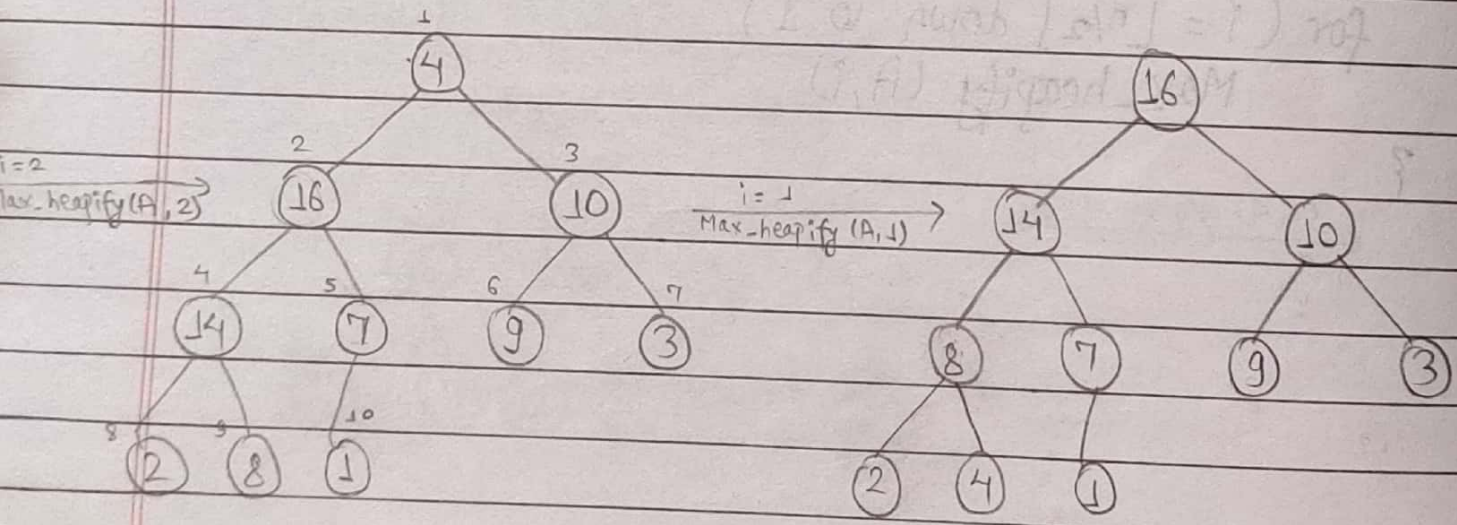
```
Build_Max_Heap (A)
{
   A.heapsize = A.length
   for (i = ⌊n/2⌋ down to 1)
        Max_heapify (A, i)
}
```

Example:

A[] = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}

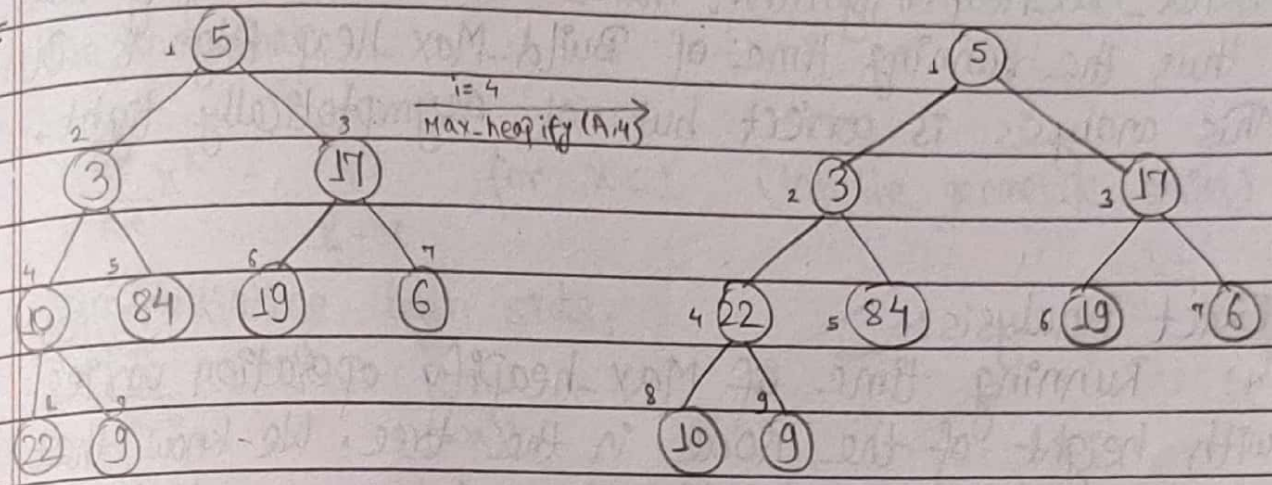


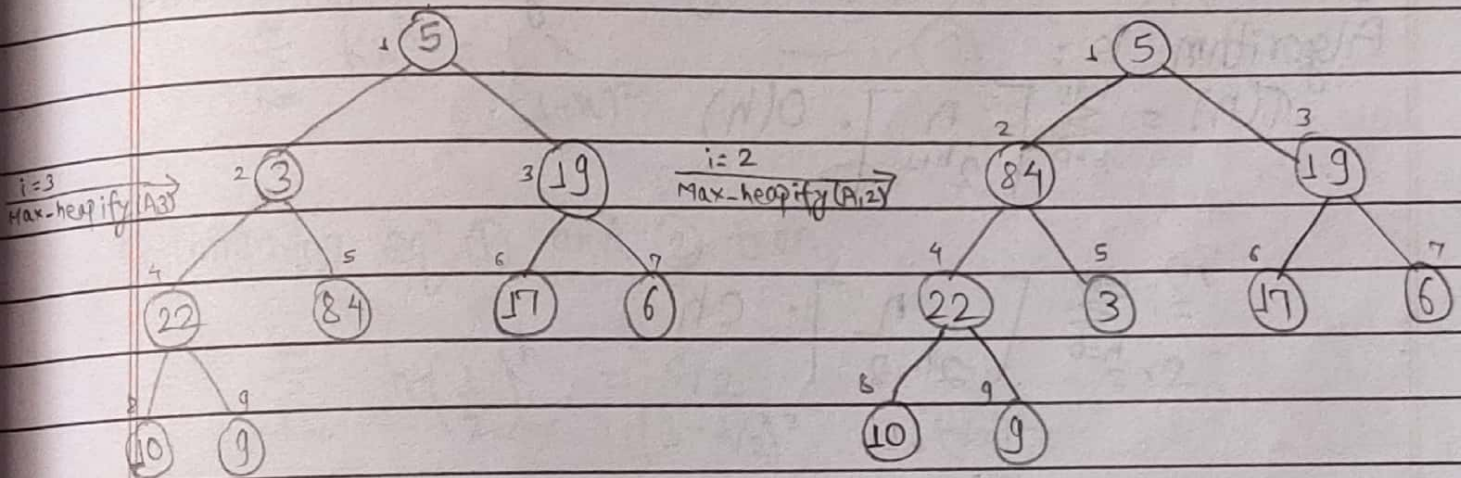


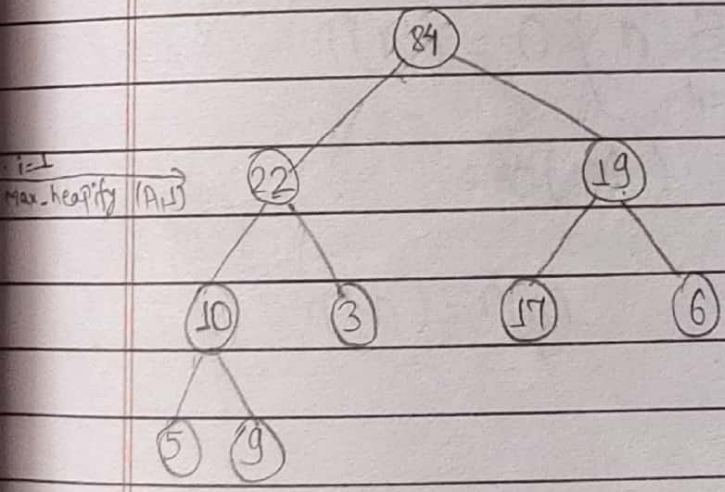


Q. Illustrate the operation of Build-Max-Heap on array A[] = <5, 3, 17, 10, 84, 19, 6, 22, 9>

Soln:







Analysis of Build_Max_Heap Algorithm:

- Each call to Max_heapify takes $O(\log n)$ time.
- Build_Max_Heap Algorithm makes $O(n)$ Max_heapify calls, thus the running time of Build_Max_Heap is $O(n \log n)$.
- This analysis is correct but not asymptotically tight.

Exact Analysis:-

Running time of Max_heapify operation varies with height of the node in the tree. We know that n element heap has height at most $\log n$ and height h is $O(h)$, so total time required by Build_Max_Heap Algorithm is:

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

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

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

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

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

$$= O\left(n \sum_{h=0}^{\infty} h\left(1/2\right)^h\right) \quad \text{——— (1)}$$

We know that,

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad \text{for } x<1 \quad \text{(infinite geometric series)}$$

Differentiating both sides,

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

Multiplying both side by x,

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

Comparing eq^n (1) and (2) then,

$$\sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{1/2}{(1-1/2)^2} = \frac{1/2}{(1/2)^2} = 2$$

Again,

$$T(n) = O\left(n \sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h\right)$$

$$= O(2h)$$

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

2068

Heap Sort Algorithm:-

To sort the given input array of n elements using heap sort it follows the following steps:

- Build a Max_heapify from array.
- Swap the root (the maximum element with the last element).
- Discard this last node by decreasing heap size.
- Perform the Max_heapify operation on the new root node.
- Repeat the process until one node remaining.

Algorithm:

```
Heapsort (A)
{
  Build_Max_Heap (A)
  for (i = A.length down to 2)
  {
    Exchange (A[1], A[i]);
    A.heapsize = A.heapsize - 1
    Max_heapify (A, 1);
  }
}
```

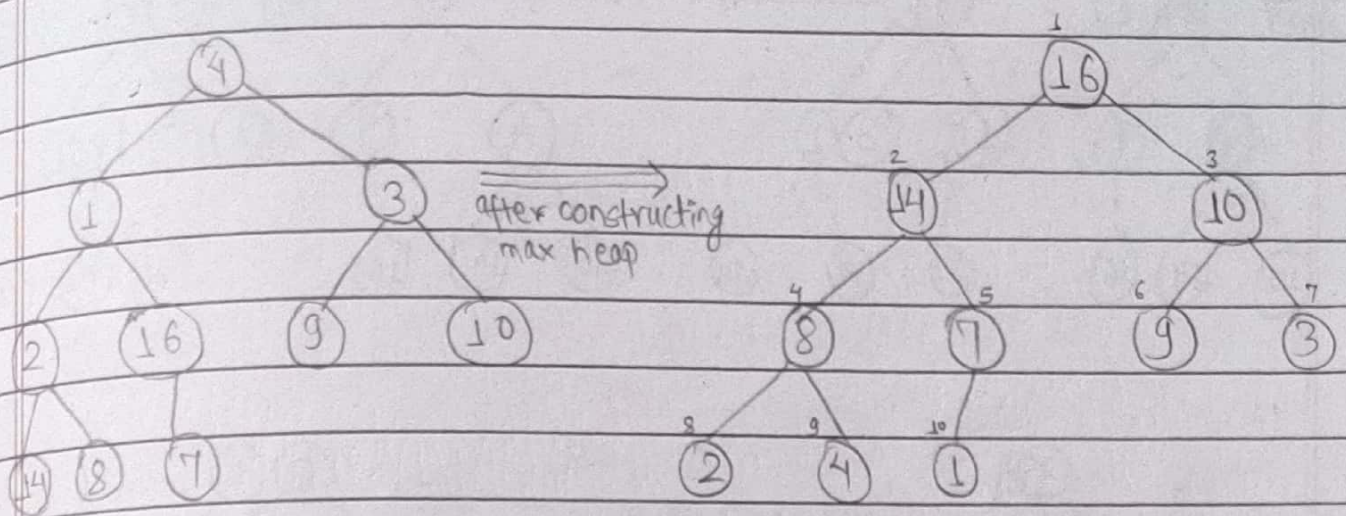
Example:

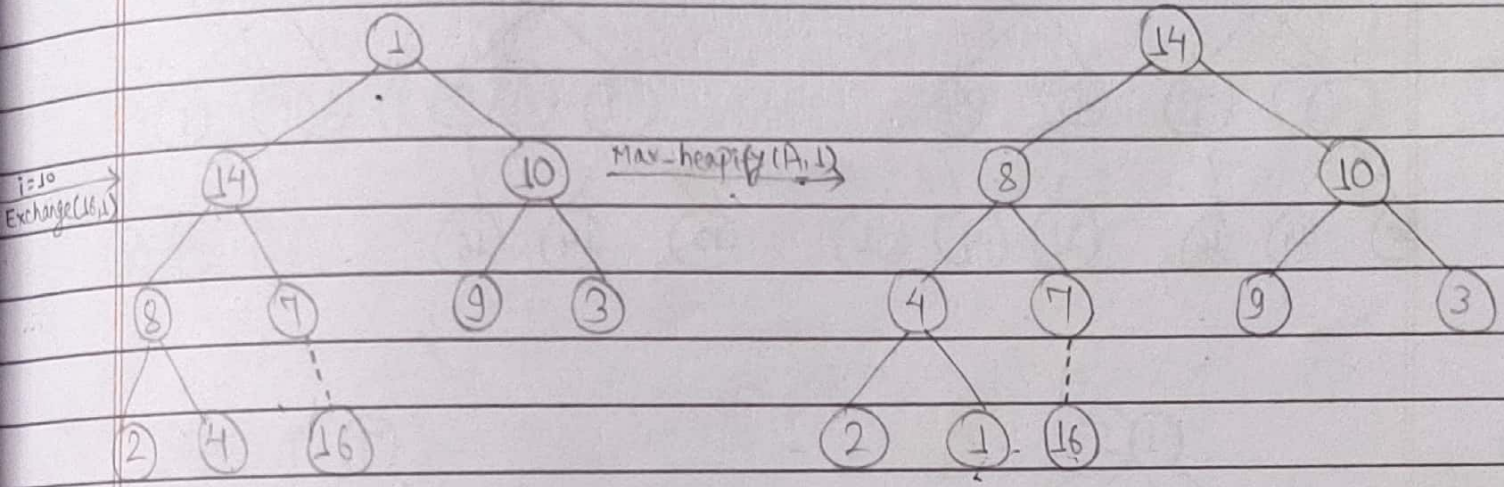
Sort the following elements using heap sort.

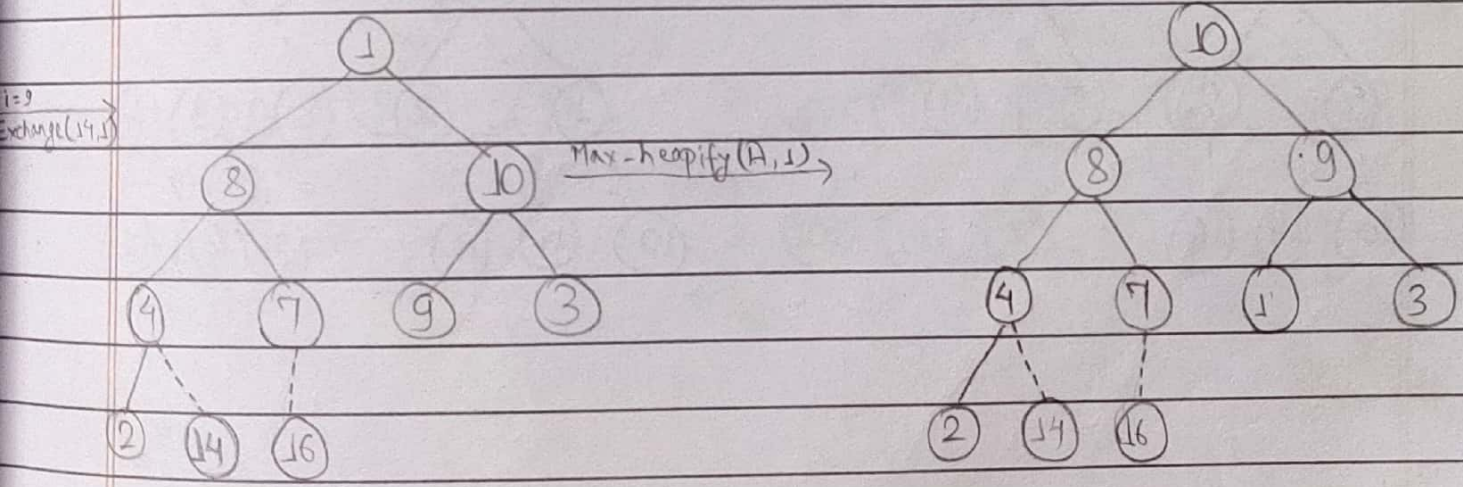
A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}

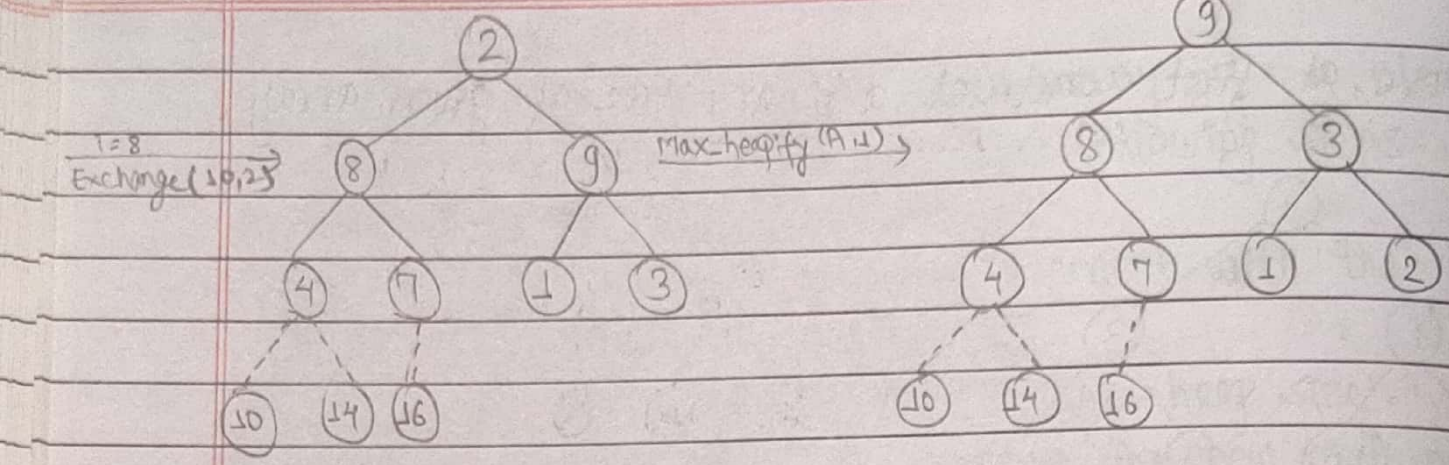
Sol^n:

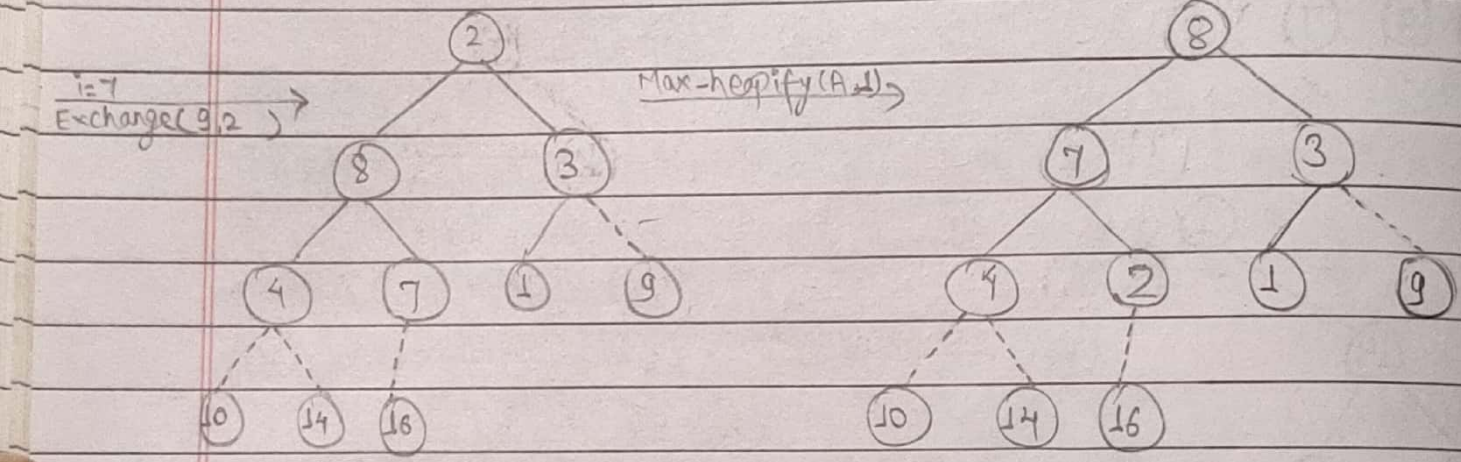
Here, at first construct a binary tree of given array.

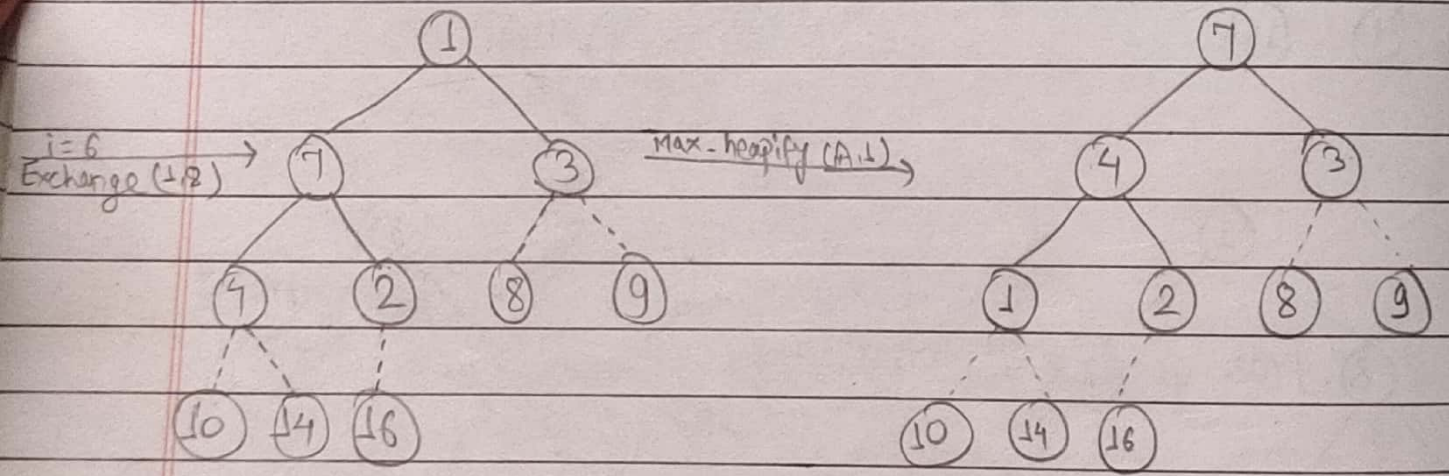


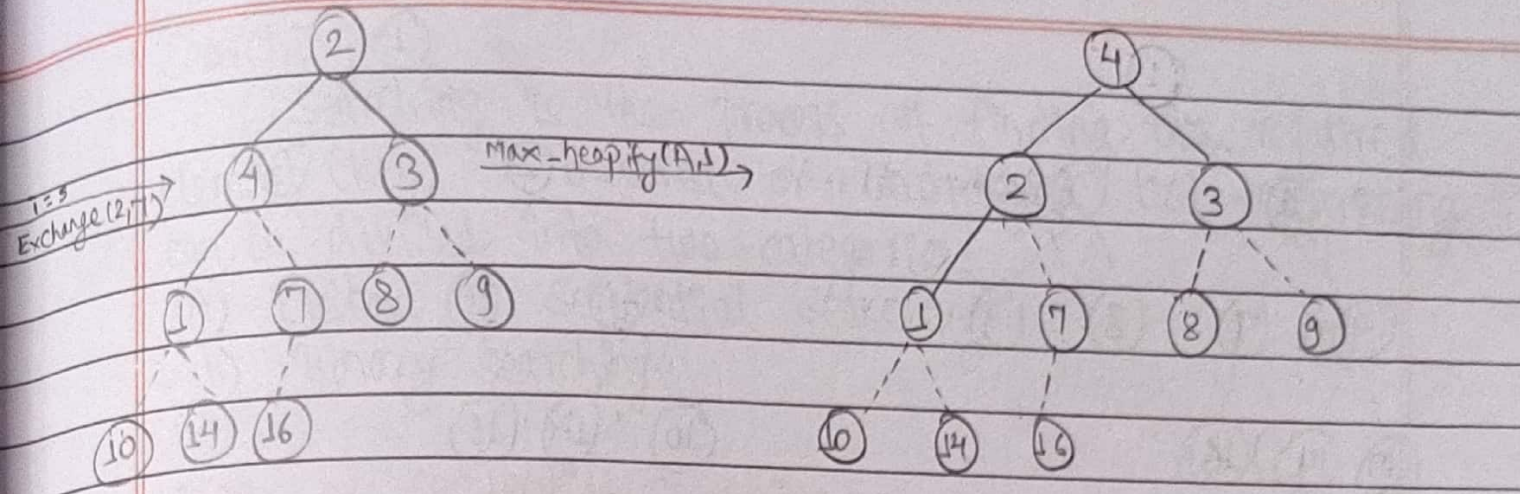


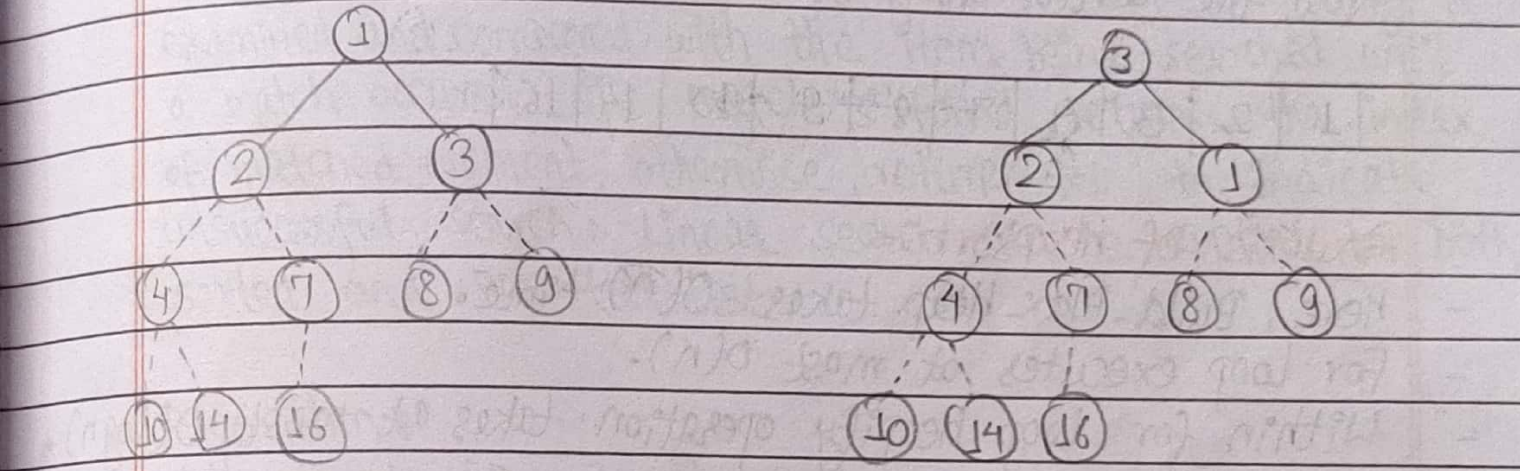


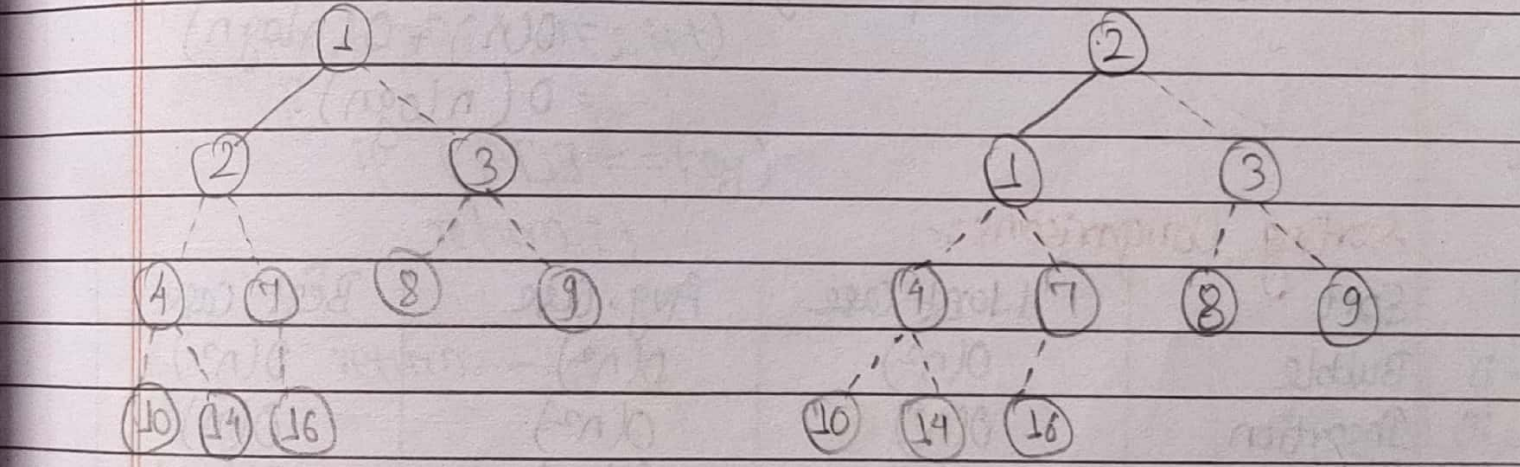


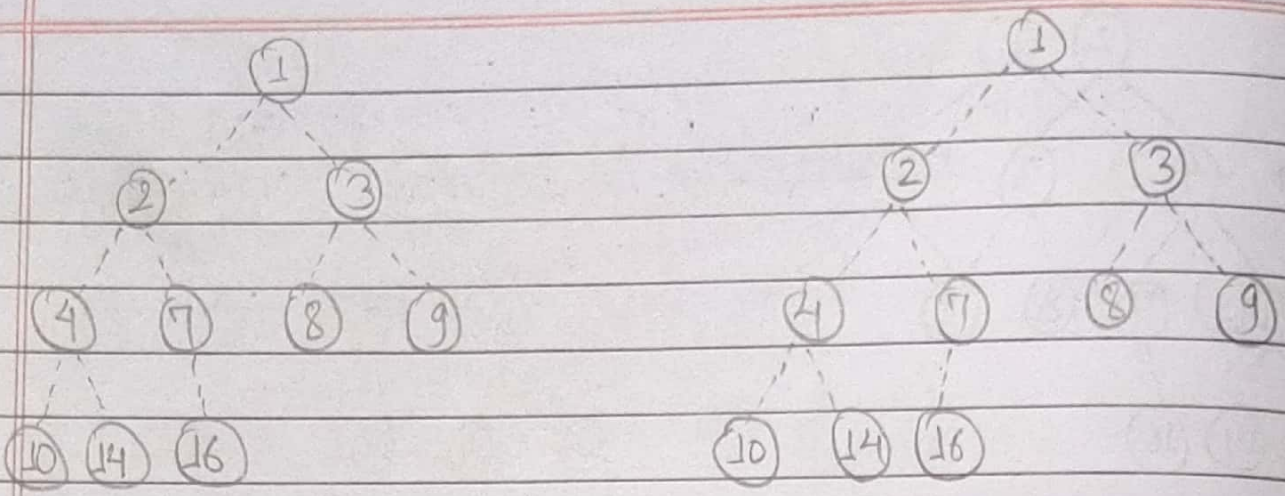












Now, the sorted array is:

1	2	3	4	7	8	9	10	14	16

Analysis of Heapsort:-

- Here, Build_Max_Heap takes $O(n)$ time.
- For loop executes at most $O(n)$.
- Within for loop heapify operation takes at most $O(\log n)$ time.
- Thus, total time complexity $T(n) = O(n) + O(n) * O(\log n)$
  $= O(n) + O(n \log n)$
  $= O(n \log n)$.

Sorting Comparisons:-

	Sort	Worst case	Avg. case	Best case
i)	Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$
ii)	Insertion	$O(n^2)$	$O(n^2)$	$O(n)$
iii)	Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
iv)	Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
v)	Quick	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
vi)	Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Searching :-

Searching is the process of finding the required element (key) in ordered or unordered list. Searching can be divided into two categories:

i) Linear or Sequential Searching
ii) Binary searching

i) Linear or Sequential Searching :-

In linear search, each item of the array is examined and compared with the item being searched until a match occurs. If match is found, returns the index of matched element, otherwise, return -1 to indicate unsuccessful search. Linear search can be applied to both sorted and unsorted list.

Algorithm :

```
LinearSearch (A, n, key)
{
   for (i=0; i<n; i++)
   {
      if (A[i] == key)
         return i;
   }
   return -1;
}
```

Analysis:
Since the loop in the algorithm executes at most n times
so, time complexity $T(n) = O(n)$

Example:-
Let input array
A[] = { 4, 5, 10, 12, 16, 24, 33, 42 }
key = 33

i = 0
A[0] == key, no, so increment i.

i = 1
A[1] == key, no, so increment i.

i = 2
A[2] == key, no, so increment i.

i = 3
A[3] == key, no, so increment i.

i = 4
A[4] == key, no, so increment i

i = 5
A[5] == key, no, so increment i.

i=6
A[6] == key. Yes, return 6;

2071(6)

ii) Binary Search :-

Binary Search is an extremely efficient algorithm based on divide and conquer technique. This search technique searches the given item (key) in minimum possible comparison. This algorithm is useful to search an item in ordered list. The general idea of binary search algorithm is:

a, Find the mid position of array A.
b, Compare the value being searched for i.e. key with A[mid].
c, There are three case:
   i) If key == A[mid] then search is successful and return the index mid.
   ii) If key < A[mid] then search process is repeated in first half of the array.
   iii) If key > A[mid] then search process is repeated in second half of the array.

Algorithm of Binary Search:

```
Binary_Search (A, low, high, key)
{
    if (low == high)        // only one element
    {
        if (key == A[low])
            return low;
        else
            return -1;
    }
```

```
else
  {
    mid = (low+high)/2 ;   // integer division
    if (key == A[mid])
        return mid;
    else if (key < A[mid])
        return BinarySearch (A, low, mid-1, key)
    else
        return Binarysearch(A, mid+1, high, key)
  }
}
```

Analysis of Binary Search Algorithm:

From the above algorithm, the running time of binary search algorithm can be expressed by following recurrence relation:

$$T(n) = 1 \quad \text{when } n=1$$
$$= T(n/2)+1 \quad \text{when } n>1$$

Solving this recurrence relation using iterative method $T(n) = O(\log n)$.

Therefore, running time of binary search is $O(\log n)$.

- The best case, output is obtained if key is at middle so best case complexity for successful search is $O(1)$.
- In worst case, the output is at the end of the array

So running time at worst case is $O(\log n)$.

- Similarly in average case, it also have the running time $O(\log n)$.

- For unsuccessful search; best, worst and average case complexity is $O(\log n)$.

Example:

Let an input array

A[] = {2, 5, 7, 9, 18, 45, 53, 59, 67, 72, 88, 95, 101, 104}

For key = 88,

low	high	mid	
0	13	6	key > A[mid]
7	13	10	key == A[mid]

termination condition return (mid) 10

For key = 2,

low	high	mid	
0	13	6	key < A[mid]
0	5	2	key < A[mid]
0	1	1/2	key < A[mid]

For key = 103,

low	high	mid	
0	13	6	key > A[mid]
7	13	10	key > A[mid]
11	13	12	key > A[mid]
13	13		termination condition because low == high, key ≠ A[low], so return -1.

Maximum and Minimum finding:

Maximum and Minimum finding is the problem of selecting maximum and minimum item in a set of n elements. This problem can be solved by two methods:

i) Iterative method
ii) Divide and conquer method

i) Iterative Algorithm for max and min finding:-

```
MinMax (A, n)
{
    Max = Min = A[0];
    for (i=0; i<n; i++)
    {
        if (A[i] > Max)
            Max = A[i];
        else if (A[i] < Min)
            Min = A[i]
    }
}
```

Analysis:

- The time complexity of this algorithm is
  T(n) = number of comparison required.
- The best case occurs when the elements are in increasing order, at that time (n-1) comparisons are needed for finding the min and max item.

- The worst case occurs when the elements are in decreasing order, at that time, 2(n-1) comparisons are needed.
- For average case A[i] > Max is about half of the time so the number of comparison is 3n/2.

Example:

Let input array is

A[] = {5, 40, 7, 23, 16, 90, 51, 1}

5	40	7	23	16	90	51	1

Max = Min = 5

For i = 1:

A[1] > Max, Yes

Max = A[1] = 40

For i = 2:

A[2] > Max, No

A[2] < Min, No

For i = 3:

A[3] > Max, No

A[3] < Min, No

For i = 4:

A[4] > Max, No

A[4] < Min, No

For i=5:
A[5] > Max, yes
Max = A[5] = 90

For i=6:
A[6] > Max, No
A[6] < Min, No

For i=7:
A[7] > Max, No
A[7] < Min, yes
Min = A[7] = 1

Hence, Min = 1
Max = 90

ii) Divide and Conquer algorithm for Max and Min finding:

The main idea behind this algorithm is: If the number of element is one or two then max and min are obtained trivially, otherwise, we split the problem into approximately equal part and solve recursively.

Algorithm:

```
MinMax(l, r)
{
    if (l == r)
        Max = Min = A[l]
```

```
else if (l == r-1)
 {
    if (A[l] < A[r])
     {
       Max = A[r];
       Min = A[l];
     }
    else
     {
       Max = A[l];
       Min = A[r];
     }
 }
else
 {
    mid = (l+r)/2        // integer division
    {min, max} = MinMax (l, mid);
    {min1, max1} = MinMax (mid+1, r);
     if (max1 > max)
        Max = max1;
     if (min1 < min)
        Min = min1;
 }
}
```

Analysis:

The recurrence relation for this algorithm in terms of number of comparisons is

$$T(n) = 2T(n/2) + 2 \quad \text{if } n > 2$$
$$= 1 \quad \text{if } n \leq 2$$

Now, solving this recurrence relation using master's theorem.

Here,

$$a = 2, b = 2, f(n) = 2 = O(1)$$

then,

$$n^{\log_b a} = n^{\log_2 2} = n$$

Hence, by case 1 of Master's theorem,

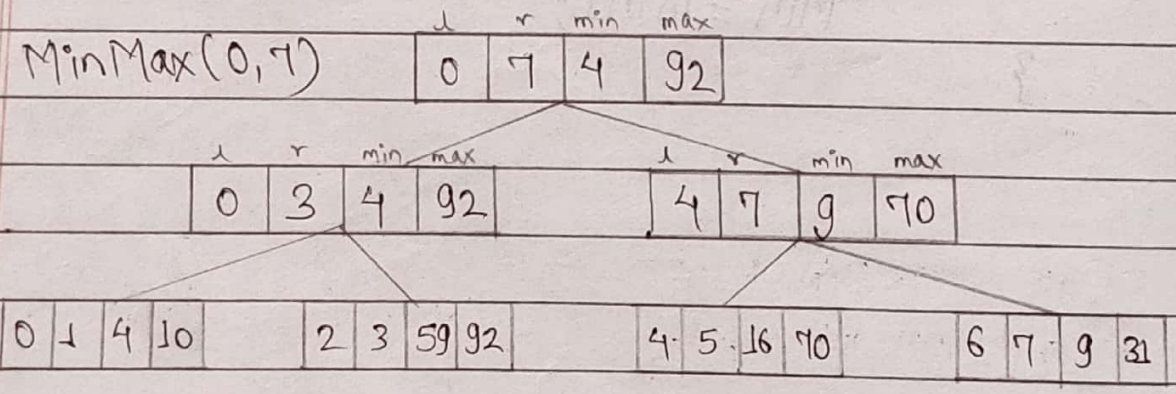
$$T(n) = \Theta(n)$$

Therefore, the time complexity of this divide and conquer algorithm for finding Max and Min is $\Theta(n)$.

Example:

Let input array

A[] = {54, 10, 59, 92, 16, 70, 9, 31}
0 1 2 3 4 5 6 7



Median and Order Statistics:

unsorted list lai sort garisakeri kun place ma kun value aaucha

Order Statistics:-

The i^{th} order statistics of a set of n elements is the i^{th} smallest element. Thus, the first order statistics ($i=1$) indicates the minimum element of set and n^{th} order statistics ($i=n$) indicates the maximum element of set. The median is referred as half-way point of the set of n elements when n is odd, the median is unique occurring at $i = \frac{n+1}{2}$ and when n is even then there are two medians occurring at $i = n/2$, $i = n/2 + 1$. Thus, regardless of parity of n, the median is given by i^{th} order statistics where $i = \left\lceil \frac{n+1}{2} \right\rceil$ for higher median

$i = \left\lfloor \frac{n+1}{2} \right\rfloor$ for lower median

Selection Problem:-

The problem of selecting i^{th} order statistics from the set of n distinct elements is called selection problem. We can formally specify selection problem as follows:

Input: A set of n distinct and an integer 'i' with $1 \le i \le n$.

Output: The element $x \in A$ that is larger than exactly i-1 other element of A.

We can solve the selection problem in $O(n \log n)$ time in very straight forward way by first sorting the element and then simply pick up the i^{th} element from the array in constant time. The question is now how to construct linear time algorithm for this problem.

Non-linear General Selection Algorithm:

We can construct a simple, inefficient general algorithm for finding k^{th} smallest element in a list that requires non-linear time. To accomplish this, we simply find the most extreme value and move into the beginning until we reach our desired index.

Algorithm:

```
Select (A,k)    // A is an array from which we find kth smallest value.
{
  for (i=1; i<=k; i++)
  {
    min_index = i;
    min_value = A[i];
    for (j=i+1; j<=n; j++)
    {
      if (A[j] < min_value)
      {
        min_index = j;
        min_value = A[j];
        swap (A[i], A[min_index])
      }
    }
  }
  return A[k];
}
```

Example:

Let A = {2,3,1,8,10}. Find the 3rd smallest element i.e. 3rd order statistics.

	1	2	3	4	5
A	2	3	1	8	10
	i	j			

Select (A,3)

For i=1

min_index = 1

min_value = 2

j=2

A[j] < min_value, No

j=3

A[j] < min_value, Yes

min_index = 3

min_value = 1

swap(A[i], A[min_index])

1	3	2	8	10

j=4

A[j] < min_value, No

j=5

A[j] < min_value, No

For i=2

min_index = 2

min_value = 3

j = 3

A[j] < min_value, Yes

min_index = 3

min_value = 2

j=5
A[j] < min_value, No

For i=3

Analysis:-
When i=1, inner loop executes n-1 times
When i=2, inner loop executes n-2 times
When i=k, inner loop executes n-k times

So, total time complexity T(n) = (n-1)+(n-2)+.....+(n-k)
T(n) = O(kn)
= O(n²) {where k=n}.

Thus, the time complexity of this algorithm is O(n²).

Median and Order statistics:

Selecting in expected Linear time (Randomized selection):-

The general selection problem is solved in expected linear time by divide and conquer approach with randomized partition. The main idea for this algorithm is to partition the element list as in quick sort which recursively processes both sides of partition but here we work on only one side of partition. The following algorithm randomized select returns i^{th} smallest element in array $A[p.....r]$ in expected linear time.

Algorithm:-

```
Randomized_select(A,p,r,i)
{
   if (p==r)
      return A[p]
   q = Randomized_partition(A,p,r)
   k = q-p+1        //Rank of pivot
   if (k==i)
      return A[k];
   else if (i<k)
      return Randomized_Select(A,p, q-1, i)
   else
      return Randomized_select(A, q+1, r, i-k)
}
```

Example:

Let us consider an array

A[] = {5, 15, 8, 3, 7, 1, 9, 10, 6, 7, 12, 14, 2, 16, 19}. Find 9th order

p=1														r=15
5	15	8	3	7	1	9	10	6	7	12	14	2	16	19

Randomized_select (A, 1, 15, 9)

p=1			q=4											r=15
3	1	2	(5)	15	8	7	9	10	6	7	12	14	16	19

k = 4 - 1 + 1

= 4 , i > k (i.e. 9 > 4)

Randomized_select (A, 5, 15, 5)

p=5								q=13		r=15
8	7	9	10	6	7	12	14	(15)	16	19

k = 13 - 5 + 1 = 9 , i < k (i.e. 5 < 9)

Randomized_select (A, 5, 12, 5)

p=5			q=8				r=12
7	6	7	(8)	9	10	12	14

k = 8 - 5 + 1

= 4 , i > k (i.e. 5 > 4)

Randomized_select (A, 9, 12, 1)

p=9			r=12
(9)	10	12	14

q=9

k = 9 - 9 + 1 = 1 , i = k

return A[1], termination condition.

Analysis:

Worst Case:- The worst case occurs when randomized partition algorithm partition the array into $0:(n-1)$ splits. This happen if every time pivot choosen by random number generator is largest remaining element. Since the randomized partition takes $O(n)$ time, thus the recurrence relation for worst case is: $T(n) = T(n-1) + O(n)$

Solving this recurrence relation, we get

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

Hence, the running time of this algorithm for worst case is $O(n^2)$.

Expected Running time:-

Let $T(n)$ be the expected running time of algorithm on n elements.

Since, each split occurs with probability $1/n$.

The recurrence relation for expected running time of algorithm is:

$$T(n) = \frac{1}{n} \sum_{k=1}^{n} T(\text{Max}(k-1, n-k))$$

Solving this recurrence relation, we get,

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

Hence, expected running time of this algorithm is $O(n)$.

How can you devise an algorithm that guarantee the selection of i^{th} order statistics in linear time? Write the algorithm of it and analyze it.

Selecting in worst case linear time :-

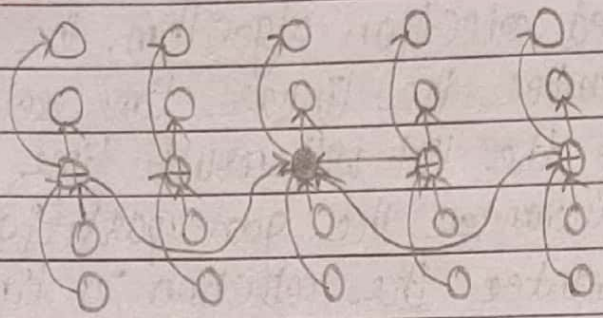
In randomized selection algorithm, the algorithm does not always guarantee the linear time selection since worst partitioning of the list will result time complexity $O(n^2)$. If some algorithm guarantee the good partition at each step then it will guarantee the selection in linear time. To guarantee the selection in linear time we design the algorithm that guarantee the good partition of input array by using median of median from small groups of elements in the list and use that median for partitioning the input array. The following is the algorithm that select i^{th} smallest element of input array having n elements in worst case linear time.

Algorithm:

select (A, i) // select i^{th} smallest element from array A.

1. Divide n elements of input array into $\lceil n/5 \rceil$ group of 5 elements each and at most one group contains remaining n mod 5 elements.
2. Find the median of $\lceil n/5 \rceil$ groups.
3. Use select recursively to find median x of $\lceil n/5 \rceil$ medians found in step 2.
4. Partition the input array around of median of median x. Let k = rank(x).
5. if i == k then return x.
   if (i<k) then use select recursively to find i^{th} smallest element in first partition.
   if (i>k) then use select recursively to find $(i-k)^{th}$ smallest element in second partition.

Analysis:



[Fig:- Analysis of select algorithm]

Here in circle plus (⊕) indicates group of medians, black circle represents median of median and arrow from larger to smaller. From the above figure at least half of the group median greater than x. Thus, half of the $\lceil n/5 \rceil$ group contains 3 elements that are greater than x except group that has lower than 5 elements and group containing x so number of elements greater than x is at least

$$3\left(\frac{1}{2}\left(\left\lceil \frac{n}{5} \right\rceil - 2\right)\right) = \frac{3n}{10} - 6 \text{ elements}$$

Similarly, no. of elements less than x is at least $\frac{3n}{10} - 6$ elements. Thus, in worst case step 5 calls select recursively on largest partition.

$$n - \left(\frac{3n}{10} - 6\right)$$

$$= \frac{7n}{10} + 6 \text{ elements.}$$

Now,

Step 1 takes $O(1)$

Step 2 takes $O(n)$

Step 3 takes $T\left(\left\lceil \frac{n}{5} \right\rceil\right)$

Step 4 takes $O(n)$

Step 5 takes $T\left(\frac{7n}{10} + 6\right)$

Hence the recurrence relation for this algorithm is:

$$T(n) = T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n)$$

Solving this using substitution method.

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

i.e. $T(n) \leq cn$

Assume our guess is true for $k < n$.

Now,

$$T(n) \leq c\left\lceil \frac{n}{5} \right\rceil + c\left(\frac{7n}{10} + 6\right) + O(n)$$

$$\geq c\frac{n}{5} + c\frac{7n}{10} + 6c + O(n)$$

$$= \frac{2cn + 7cn}{10} + 6c + O(n)$$

$$= \frac{9cn}{10} + 6c + O(n)$$

$$= cn - \frac{cn}{10} + 6c + O(n)$$

$$= cn - \left(\frac{cn}{10} - 6c - O(n)\right)$$

$\leq cn$ (when $\frac{cn}{10} - 6c - o(n)$ positive)

Hence, $T(n) = O(n)$.

Matrix Multiplication:-

Given two $A = (a_{ij})$ and $B = (b_{ij})$ are $n \times n$ square matrices. Our aim is to find product of A and B as C that is also $n \times n$ matrix. We can find this matrix $C = A \cdot B$ by using following relation

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

The following algorithm is the straight-forward algorithm for multiplying $n \times n$ matrices A and B and returning their $n \times n$ product matrix C.

Algorithm:

```
Matrix_multiply (A,B,n)  // multiply n×n matrix A & B
{
  for (i=1; i≤n ; i++)
  {
    for (j=1; j≤n ; j++)
    {
      cij = 0;
      for (k=1 ; k≤n ; k++)
      {
        cij = cij + aik · bkj
      }
    }
  }
```

Analysis:

There are three nested loops exactly running n times. Hence, matrix multiply algorithm takes $O(n^3)$.

Divide and Conquer approach for Matrix Multiplication :-

For Divide and conquer algorithm to compute the matrix product $C = A \cdot B$ we assume that n is exact power of 2 in each $n \times n$ matrix. Divide the $n \times n$ square matrix into four matrices of size $n/2 \times n/2$. We can write the equation $C = A \cdot B$ in divide and conquer approach as

C = A · B

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where,

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

Example:

Let $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}_{2\times2}$ $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}_{2\times2}$

Divide A and B into 4 matrices of size $n/2 \times n/2$.

Sol:

$A = \begin{matrix} A_{11} & A_{12} \\ (1) & (2) \\ A_{21} & A_{22} \\ (3) & (4) \end{matrix}$ $B = \begin{matrix} B_{11} & B_{12} \\ (5) & (6) \\ B_{21} & B_{22} \\ (7) & (8) \end{matrix}$

Now,

$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} = 1 \times 5 + 2 \times 7 = 19$

$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22} = 1 \times 6 + 2 \times 8 = 22$

$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21} = 3 \times 5 + 4 \times 7 = 43$

$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22} = 3 \times 6 + 4 \times 8 = 50$

Hence,

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Algorithm:

```
Matrix_multiply (A,B,n)   /* multiply n×n matrices A and
{                            B by using Divide and Conquer
                             approach */
  if (n==1)
    return A*B;
  else
    {
      Partition A,B,C as A11 ... Ann, B11 ... Bnn, C11 ... Cnn
      C11 = Matrix_multiply (A11, B11, n/2)+
                Matrix_multiply (A12, B21, n/2);
      C12 = Matrix_multiply ( A11, B12, n/2) +
                Matrix_multiply ( A12, B22, n/2);
      C21 = Matrix_multiply (A21, B11, n/2) +
                Matrix_multiply (A22, B21, n/2);
      C22 = Matrix_multiply (A21, B12, n/2) +
                Matrix_multiply (A22, B22, n/2);
    }
  return C
}
```

Analysis:

Let $T(n)$ be the time to multiply two $n \times n$ matrices using above algorithm then recurrence relation for this algorithm is

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ = O(1) + 8T(n/2) + O(n^2) & n \geq 2 \end{cases}$$

$O(1)$ — divide; $8T(n/2)$ — recursive call; $O(n^2)$ — addition

This recurrence can be written as:

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ = 8 \cdot T(n/2) + O(n^2) & n \geq 2 \end{cases}$$

Now, by using Master theorem,

$a=8$, $b=2$, $f(n) = O(n^2)$

Now,

$n^{\log_b a}$ i.e. $n^{\log_2 8} = n^3$

Using case 3 of Master theorem

$$T(n) = \theta(n^3)$$

Thus, simple divide and conquer approach is no faster than straight-forward multiplying algorithm.

VII

Strassen's Matrix Multiplication:

Strassen was the first person to give the better algorithm for matrix multiplication. The idea behind this algorithm is divide the problem into 4 sub-problems of dimension $n/2 \times n/2$ and solve it recursively. In this method, instead of performing eight recursive multiplication of $n/2 \times n/2$ matrices, it performs only seven. We can write the equation, $Z = X \cdot Y$ in Strassen's matrix multiplication as:

$$\begin{bmatrix} P_6+P_5+P_4-P_2 & P_1+P_2 \\ P_3+P_4 & P_1+P_5-P_3-P_7 \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

X Y

where,

$P_1 = A*(F-H)$

$P_2 = H*(A+B)$

$P_3 = E*(C+D)$

$P_4 = D*(G-E)$

$P_5 = (A+D)*(E+H)$

$P_6 = (B-D)*(G+H)$

$P_7 = (A-C)*(E+F)$

Algorithm:

```
Strassen (X, Y, n)          /*Strassen's method to multiply two
{                            matrices */
  if (n==1)
     return x*y
  else
     {
     Partition A,B,C,D,E,F,G,H, Z11, Z12, Z21, Z22;
     P1 = strassen (A, F-H, n/2);
     P2 = strassen (H, A+B, n/2);
     P3 = Strassen ( E, C+D , n/2);
     P4 = strassen (D, G-E, n/2);
     P5 = Strassen (A+D, E+H, n/2);
     P6 = Strassen (B-D, G+H, n/2);
     P7 = Strassen (A-C, E+F, n/2);
     Z11 = P6+P5+P4-P2
     Z12 = P1+P2
     Z21 = P3+P4
     Z22 = P1+P5-P3-P7
     }
     return Z;
  }
```

Analysis:

There are seven recursive call in problem of size $n/2$ and addition requires $O(n^2)$ so recurrence relation of this algorithm is;

$$T(n) = 1 \quad \text{if } n=1$$
$$= 7T(n/2) + O(n^2) \quad \text{if } n>1$$

Now, by using master theorem.

$a = 7, b = 2, f(n) = O(n^2)$

then,

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$$

Hence, by using case 1 of Master theorem.

$$T(n) = \theta(n^{2.81})$$

Thus, Strassen's method is asymptotically faster than simple divide and conquer matrix multiplication method.

Example:

Use strassen's method to compute the matrix product of following matrices.

$$X = \begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix} \qquad Y = \begin{bmatrix} 6 & 8 \\ 4 & 2 \end{bmatrix}$$

Soln:

Here, at first divide the matrix X and Y into four matrices as:

$$X = \begin{matrix} A & B \\ (1) & (3) \\ C & D \\ (7) & (5) \end{matrix} \qquad Y = \begin{matrix} (E) & F \\ (6) & (8) \\ G & H \\ (4) & (2) \end{matrix}$$

Now,

$P_1 = A * (F - H) = 1 * (8 - 2) = 6$

$P_2 = H * (A + B) = 2 * (1 + 3) = 8$

$P_3 = E * (C + D) = 6 * (7 + 5) = 72$

$P_4 = D*(G-E) = 5*(4-6) = -10$

$P_5 = (A+D)*(E+H) = (1+5)*(6+2) = 48$

$P_6 = (B-D)*(G+H) = (3-5)*(4+2) = -12$

$P_7 = (A-C)*(E+F) = (1-7)*(6+8) = -84$

$Z_{11} = P_6 + P_5 + P_4 - P_2 = -12+48-10-8 = 18$

$Z_{12} = P_1 + P_2 = 6+8 = 14$

$Z_{21} = P_3 + P_4 = 72 - 10 = 62$

$Z_{22} = P_1 + P_5 - P_3 - P_7 = 6+48-72+84 = 66.$

Hence,

$$Z = \begin{bmatrix} 18 & 14 \\ 62 & 66 \end{bmatrix}$$

bottom-up, overlapped sub-problems

2.2 Dynamic Programming

2069 (2 marks)

Dynamic Programming is a technique for solving problem with overlapping sub-problems. Like divide and conquer approach, dynamic programming approach solves the problem by combining the solution to the sub-problem. This is the bottom-up problem solving technique in which each sub-problem is solved only once and solution to these sub-problems are stored in the table thereby, avoiding the work of recomputing the solution each time it solves each sub-problem. Dynamic programming approach is typically used for solving optimization problem. The basic elements that characterize dynamic programming algorithm are:

1) Substructure:

Decompose the problem into smaller sub-problem.

11) Table structure:

Store the answer to the sub-problem in the table. This is because sub-problem solutions are reused many times.

111) Bottom-up computation:

Combine the solution on solution to larger sub-proble

Dynamic programming is not applicable to all optimizatio problem. It works when a problem has following characteristics:

1) Optimal substructure: The problem exhibits optimal sub-structu

if the optimal solution to the problems contains with its optimal solution to sub-problem.

ii) Overlapping subproblem: The given problem has overlapping subproblems if an algorithm calculates the solution to the sub-problem more than once inorder to use its solution for problem of size greater than the solved sub-problem.

Development of Dynamic Programming Algorithm:

When we develop a dynamic programming algorithm, we follow the sequence of four steps:

i) Characterize the structure of optimal solution
ii) Recursively define the value of optimal solution
iii) Compute the value of an optimal solution in bottom-up fashion
iv) Construct an optimal solution from computed information.

Difference between Dynamic Programming and Divide and Conquer approach:

Divide and conquer algorithm	Dynamic Programming
i) Divide and conquer algorithm splits a problem into seperate sub-problems, solve the sub-problems and combine the results for a solution to the original problem. Example: Quick Sort, Merge Sort, Binary Search, etc.	i) Dynamic programming splits a problem into sub-problems, some of which are common solves the sub-problems & combines the results for a solution to the original problem. Example: Matrix Chain Multiplication, Longest common sub-sequence.

ii)	Divide and Conquer algorithm thought of as top-down algorithm.	ii) Dynamic programming can be thought of as bottom-up.
iii)	In divide-and-conquer, sub-problems are independent.	iii) In dynamic programming, sub-problems are not independent.
iv)	Divide and conquer solutions are simple as compared to dynamic programming.	iv) Dynamic programming solutions can often be quite complex and tricky.
v)	Divide and conquer can be used for any kind of problem.	v) Dynamic programming is generally used for optimization problems.
vi)	Only one decision sequence is ever generated.	vi) Many decision sequence may be generated.

Fibonacci numbers:-

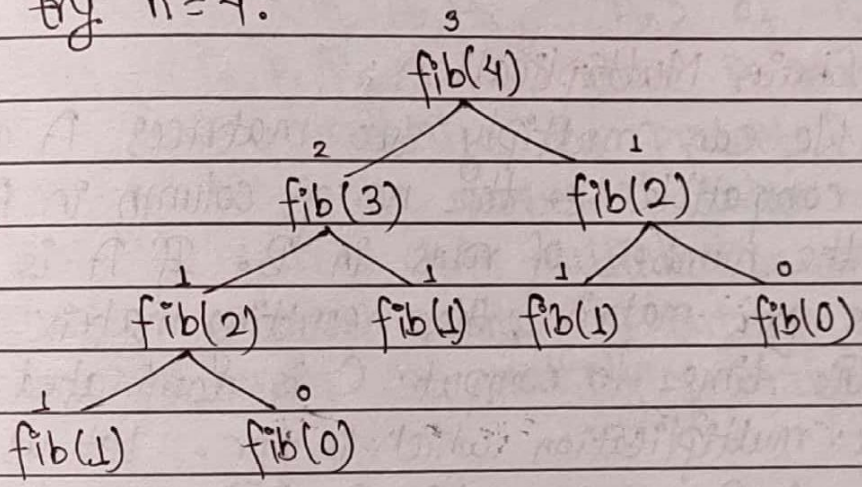
- Fibonacci number are those numbers that are obtained by summing-up two previous fibonacci numbers.
- So recursive definition of fibonacci number is:

$$fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ fib(n-1) + fib(n-2) & \end{cases}$$

Recursive algorithm for finding n^{th} fibonacci number :-

```
fib(n)
{
    if (n==0)
        return 0;
    else if (n==1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

- In the recursive version of an algorithm for finding fibonacci number we can notice that there are many redundancies in calculating fibonacci number of particular number.
- Let us try n=4.



- In above tree we saw that calculation of fib is repeated so we use the dynamic programming approach to eliminate the redundancy as:

Algorithm (using dynamic programming) :-

```
Dynamic(n)
{
    A[0] = 0
    A[1] = 1
    for (i=2; i≤n; i++)
        A[i] = A[i-2] + A[i-1]
    return A[n];
}
```

Analysis :-
The running time of this algorithm is $O(n)$.

Matrix Chain Multiplication :-

We can multiply two matrices A and B only if they are compatible i.e. the no. of column in A must be equal to the number of rows in B. If A is $p \times q$ matrix and B is $q \times r$ matrix, the resulting matrix C is $p \times r$ matrix. The time to compute C is dominated by number of scalar multiplication which is pqr. Let us consider the chain of 3 matrices $\langle A_1, A_2, A_3 \rangle$ with dimension 10×100, 100×5 & 5×50 respectively and we want to compute $A_1 \cdot A_2 \cdot A_3$. We compute $A_1 \cdot A_2 \cdot A_3$ in two ways:

i) $A_1 \cdot (A_2 \cdot A_3)$
ii) $(A_1 \cdot A_2) \cdot A_3$

If we compute the product according to parenthesization $A_1 \cdot (A_2 \cdot A_3)$ then total number of scalar multiplication required are
$(100 \times 5 \times 50) + (10 \times 100 \times 50)$
$= 75000$

If we compute the product according to parenthesization $(A_1 \cdot A_2) \cdot A_3$ then total number of scalar multiplication required are $(10 \times 100 \times 5) + (10 \times 5 \times 50)$
$= 7500$

Thus, computing the product according to second parenthesization is 10 times faster.

Matrix Chain Multiplication Problem :-

- Given a chain $\langle A_1, A_2, A_3, \ldots, A_n \rangle$ of n matrices and dimensions $\langle P_0, P_1, P_2, \ldots, P_n \rangle$ where A_i has dimension $P_{i-1} \times P_i$, fully parenthesize the product $A_1, A_2, \ldots A_n$ in a way that minimize the number of scalar multiplication.

Note that, in matrix chain multiplication problem we are not actually multiplying matrices but our goal is to determine an order for multiplying matrix that has lower cost.

Dynamic Programming approach for matrix chain multiplication :-

- Let $A_{i \ldots j}$ denotes the matrix that results from evaluating the product $A_i \cdot A_{i+1} \cdot A_{i+2} \ldots A_j$.
- if $i<j$, then to parenthesize the product $A_i \cdot A_{i+1} \cdot A_{i+2} \ldots A_j$ we must split the product between A_k and A_{k+1} for some k such that $i \le k < j$
- Hence in this way total cost for computing $A_i \ldots A_j$ is sum of cost of computing $A_{i \ldots k}$, cost of computing $A_{k+1} \ldots A_j$ and cost of multiplying $A_{i \ldots k}$ and $A_{k+1 \ldots j}$
- Let $m[i,j]$ denotes the minimum cost needed to compute the matrix $A_{i \ldots j}$ then we have the recursive formula,

$$m[i,j] = \begin{cases} 0 & i=j \\ \min\limits_{i \le k < j} \left(m[i,k] + m[k+1,j] + P_{i-1} * P_k * P_j \right) \end{cases}$$

$s[i,j]$ = value of k which minimize the cost.

Example:

Given a chain of matrices $\langle A_1, A_2, A_3, A_4 \rangle$ and sequence of dimensions $p = \langle 3,4,5,2,3 \rangle$, find an optimal parenthesization of matrix chain product.

Soln:

Given chain of matrices $\langle A_1 \cdot A_2 \cdot A_3 \cdot A_4 \rangle$ and sequence of dimensions $\langle 3,4,5,2,3 \rangle$

Now compute the table m[i,j] and s[i,j] as m table (cost of multiplication)

	1	2	3	4
1	0	60	64	82
2		0	40	64
3			0	30
4				0

m-table

	2	3	4
1	1	1	3
2		2	3
3			3

s-table (point of parenthesization)

Fill $m[i,j] = 0$ for $i = j$

For $m[1,2]$, $k = 1$

$$m[1,2] = \min_{1 \le i < 2} \{ m[1,1] + m[2,2] + 3*4*5 \}$$

$$= 0 + 0 + 60$$

$$= 60$$

For $m[2,3]$, $k = 2$

$$m[2,3] = m[2,2] + m[3,3] + 4*5*2$$

$$= 0 + 0 + 40$$

$$= 40$$

For $m[3,4]$, $k = 3$

$$m[3,4] = m[3,3] + m[4,4] + 5*2*3$$

$$= 0 + 0 + 30$$

For chain of 3 matrices:

*. For $m[1,3]$, $k = 1, 2$

$$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + p_0 * p_1 * p_3 \\ \quad 0 + 40 + 3*4*2 = 64 \\ m[1,2] + m[3,3] + p_0 * p_2 * p_3 \\ \quad 60 + 0 + 3*5*2 = 90 \end{cases}$$

* For $m[2,4]$, $k = 2, 3$

$$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + P_1 * P_2 * P_4 \\ \quad 0 + 30 + 4*5*3 = 90 \\ m[2,3] + m[4,4] + P_1 * P_3 * P_4 \\ \quad 40 + 0 + 4*2*3 = 64 \end{cases}$$

Chain consists of 4 matrices:-

For $m[1,4]$, $k = 1, 2, 3$

$$m[1,4] = \min \begin{cases} m[1,1] + m[2,4] + P_0 * P_1 * P_4 \\ \quad 0 + 64 + 3*4*3 = 100 \\ m[1,2] + m[3,4] + P_0 * P_2 * P_4 \\ \quad 60 + 30 + 3*5*3 = 135 \\ m[1,3] + m[4,4] + P_0 * P_3 * P_4 \\ \quad 64 + 0 + 3*2*3 = 82 \end{cases}$$

Thus, from m-table number of multiplication needed for given chain $m[1,4] = 82$. Hence, optimal parenthesization is:

$$A_1 \cdot A_2 \cdot A_3 \cdot A_4 \Rightarrow (A_1 \cdot A_2 \cdot A_3) \cdot A_4$$
$$\Rightarrow ((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$$

Algorithm for Matrix Chain Multiplication problem:

1) Algorithm (Computing optimal cost):-

Input: Array p of dimensions $\{P_0, P_1, P_2, \ldots, P_n\}$
Output: m table and s table, m table holds the cost and s table is used to construct the optimal solution.

```
Matrix_Chain_Multiplication(p)
{
  n = p.length - 1;
  Let m[1...n, 1...n] and s[1...n-1, 2...n]
  for (i=1; i<=n; i++)
  {
    m[i,i] = 0    // chain consist of only one matrix.
  }
  for (l=2; l<=n; l++)
  {
    for(i=1; i<=n-l+1; i++)
    {
      j = i+l-1
      m[i,j] = ∞
      for (k=i; k<=j-1; k++)
      {
        q = m[i,k] + m[k+1,j] + P_{i-1} * P_k * P_j
        if ( q < m[i,j])
        {
          m[i,j] = q;
          s[i,j] = k;
        }
      }
  } return m and s;
}
```

Analysis :-

The running time of this algorithm is $O(n^3)$ due to the three nested loop since each loop index l, i, and k takes n-1 steps.

2) Algorithm (Constructing Optimal solution) :-

The s table returned by matrix_chain_multiplication algorithm is used to construct an optimal solution.

```
Print_optimal_solution (s, i, j)
{
  if (i == j)
     print "Ai"
  else
     {
      print "(";
      Print_optimal_solution (s, i, s[i, j]);
      Print_Optimal_Solution (s, s[i, j] + 1, j);
      print ")";
     }
}
```

5, 1, 2.
5, 2, 6

Example:

Given a chain of matrices < $A_1 A_2 A_3 A_4 A_5 A_6$ >
and, dimension P = < 2, 5, 3, 5, 2, 3, 5 >
Find the optimal parenthesization of matrix chain product.

Sol:

Here,

Matrix Chain = < $A_1 A_2 A_3 A_4 A_5 A_6$ >
Dimension P = < 2, 5, 3, 5, 2, 3, 5 >
(P_0 P_1 P_2 P_3 P_4 P_5 P_6)

Now,

compute table m and S as:

	1	2	3	4	5	6
1	0	30	60	72	84	114
2		0	75	60	90	140
3			0	30	48	90
4				0	30	80
5					0	30
6						0

m table

	2	3	4	5	6
1	1	2	2	4	5
2		2	2	4	4
3			3	4	4
4				4	4
5					5

s table.

Fill m [i, j] = 0 for i == j

⇒ Chain consists of two matrices.

For m [1,3] , k = 1, 2

$$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + P_0 * P_1 * P_3 \\ \quad 0 \quad\quad 75 \quad + 2*5*5 = 125 \\ m[1,2] + m[3,3] + P_0 * P_2 * P_3 \\ \quad 30 \quad\quad 0 \quad + 2*3*5 = 60 \end{cases}$$

= 60

$m[i,k] + m[k+1,j] + p_{i-1} * p_k * p_j$

For $m[2,4]$, $k = 2,3$

$$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + p_1 * p_2 * p_4 \\ \quad 0 \quad + \quad 30 \quad + 5*3*2 = 60 \\ m[2,3] + m[4,4] + p_1 * p_3 * p_4 \\ \quad 75 \quad + \quad 0 \quad + \quad 5*5*2 = 125 \end{cases}$$

For $m[3,5]$, $k = 3,4$

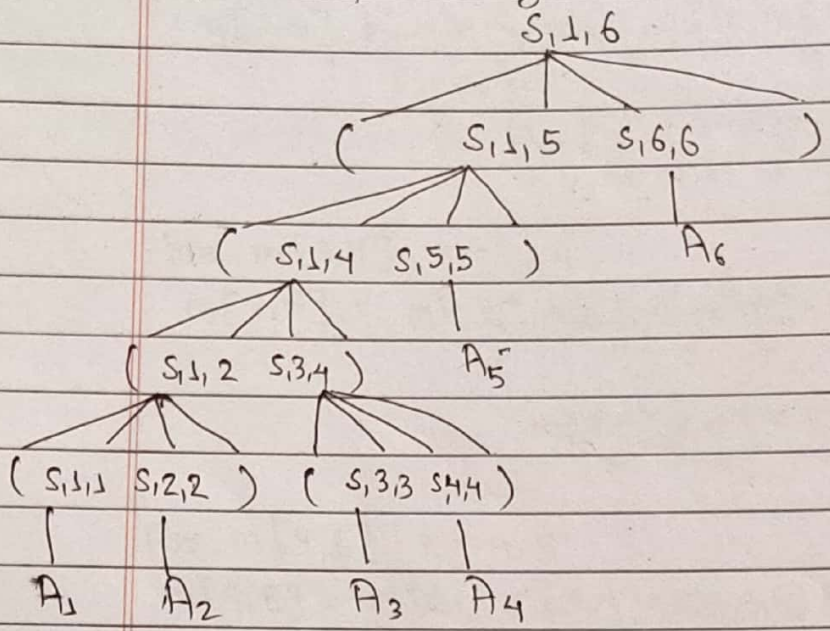
$$m[3,5] = \min \begin{cases} m[3,3] + m[4,5] + p_2 * p_3 * p_5 \\ \quad 0 \quad + \quad 30 \quad + 3*5*3 = 75 \\ m[3,4] + m[5,5] + p_2 * p_4 * p_5 \\ \quad 30 \quad + \quad 0 \quad + \quad 3*2*3 = 48 \end{cases}$$

For $m[4,6]$, $k = 4,5$

$$m[4,6] = \min \begin{cases} m[4,4] + m[5,6] + p_3 * p_4 * p_6 \\ \quad 0 \quad + \quad 30 \quad + 5*2*5 = 80 \\ m[4,5] + m[6,6] + p_3 * p_5 * p_6 \\ \quad 30 \quad + \quad 0 \quad + 5*3*5 = 105 \end{cases}$$

Thus, from above table the minimum number of multiplication needed to multiply the given chain is $m[1,6] = 114$.

Optimal parenthesization is:



$(((((A_1 \cdot A_2)(A_3 \cdot A_4))A_5)A_6)$

2ta given sequence bata banne sabai subsequence maddhye ko longest subsequence find out garne technique

Longest Common Subsequence Problem:-

Subsequence:-

A subsequence of the given sequence is just the given sequence with some elements left out. Formally given two sequences $X = \langle x_1, x_2, \ldots x_m \rangle$ and $Z = \langle z_1, z_2, \ldots$ we say that z is sub-sequence of X if there exist a strictly increasing sequence $\langle i_1, i_2, \ldots i_k \rangle$ of indices of X on a sequence z.

- For example:-

$X = \langle \overset{1}{A}\overset{2}{B}\overset{3}{C}\overset{4}{B}\overset{5}{D}\overset{6}{A}\overset{7}{B} \rangle$ then

$Z = \langle \underset{2}{B}\underset{3}{C}\underset{5}{D}\underset{7}{B} \rangle$ is a sub-sequence of X with indices $\langle 2,3,5,7 \rangle$.

2 5 7 6

$Z = \langle BOBA \rangle$ is not sub-sequence of X.

Common subsequence :-

- Given two sequences X and Y, we say that a sequence Z is common subsequence of X and Y if Z is the common subsequence of both X and Y.
- Example :-

$X = \langle ABCBOAB \rangle$

$Y = \langle BO\ CABA \rangle$

Some common subsequences are :

$\langle BCB \rangle$, $\langle BCBA \rangle$, $\langle CAB \rangle$, $\langle BCAB \rangle$ etc.

Longest Common Subsequence (LCS) :-

Given two sequences X and Y, the longest common subsequence of X and Y is the longest sequence Z that is subsequence of both X and Y.

Example :-

$X = \langle abbaab \rangle$

$Y = \langle aabaabb \rangle$

$Z = \langle abaab \rangle$

Longest Common Subsequence Problem :-

Given two subsequence $X = \langle x_1 x_2 \ldots x_m \rangle$ and $Y = \langle y_1 y_2 \ldots y_m \rangle$ then longest common subsequence problem is to find a maximum length common subsequence of X and Y.

Dynamic Programming approach for longest common subsequence Problem:-

- Given a sequence $X = \langle x_1 x_2 \ldots x_m \rangle$, then $X_i = \langle x_1 x_2 \ldots x_i \rangle$ is called i^{th} prefix of X and X_0 is empty sequence.
- Let $C[i,j]$ be the length of an LCS of the sequence X_i and Y_j then we have recursive formula as:

$$C[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \text{ (i.e. either of the subsequence is empty)} \\ C[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i == x_j \text{ (i.e. if last character match)} \\ \max(C[i,j-1], C[i-1,j]) & \text{if } i,j>0 \text{ and } x_i \neq x_j \text{ (i.e. last characters do not match)} \end{cases}$$

Similar ex. 2067, 2068, 2070 (6)

Example:-

Find the longest common subsequence of the sequence.

$X = \langle ABCBDAB \rangle$

$Y = \langle BDCABA \rangle$

Soln:

To compute LCS of given sequence, we need to compute the C table, and b as below:

if $x_i = x_j$; diagonal + 1

if $x_i \neq x_j$, max(left, top).

			B	D	C	A	B	A
		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
A	1	0	0↑	0↑	0↑	↖1	←1	↖1
B	2	0	↖1	←1	←1	1↑	↖2	←2
C	3	0	1↑	1↑	↖2	←2	2↑	2↑
B	4	0	↖1	1↑	2↑	2↑	↖3	←3
D	5	0	1↑	↖2	2↑	2↑	3↑	3↑
A	6	0	1↑	2↑	2↑	↖3	3↑	↖4
B	7	0	↖1	2↑	2↑	3↑	↖4	4↑

↑

Length of LCS = 4

LCS = BCBA

Algorithm (Computing the length of LCS)

input: two sequences $X = \langle x_1 x_2 \ldots x_m \rangle$ and $y = \langle y_1 y_2 \ldots y_n \rangle$

output: two table b and c, c[m,n] contain the length of LCS of X and Y. b table is used computation of LCS.

```
LCS_Length (X,Y)
{
   m = x.length;
   n = y.length;
   Let c [0...m, 0...n] and b[1...m, 1...n]
```

RS

```
for (i=0; i≤m; i++)
    c[i,0] = 0;
for (j=0; j≤n; j++)
    c[0,j] = 0;
for (i=1; i≤m; i++)
{
    for (j=1; j≤n; j++)
    {
        if (xi == yj)
        {
            c[i,j] = c[i-1,j-1] + 1
            b[i,j] = "↖"
        }
        else if (c[i-1,j] ≥ c[i,j-1])
        {
            c[i,j] = c[i-1, j]
            b[i,j] = "↑"
        }
        else
        {
            c[i,j] = c[i, j-1];
            b[i,j] = "←"
        }
    }
}
return c and b
}
```

RS

Analysis:-

The running time of above algorithm is O(m.n). Due to two nested loop, first runs m time and second runs n time.

Space complexity: O(m.n)

Algorithm for constructing LCS:-

The b table constructed by LCS_length is need to construct LCS of sequence X and Y.

```
Print_LCS (b,X,Y,i,j)
{
  if (i==0 or j==0)
    return;
  if (b[i,j] == "↖")
  {
    Print_LCS (b,X,Y,i-1,j-1)
    Print "Xi";
  }
  else if (b[i,j] == "↑")
    Print_LCS (b,X,Y,i-1,j);
  else
    Print_LCS(b,X,Y,i,j-1);
}
```

RS

Analysis:-
This algorithm takes $O(m+n)$ since it decrement at least one of i and j in each recursive call.

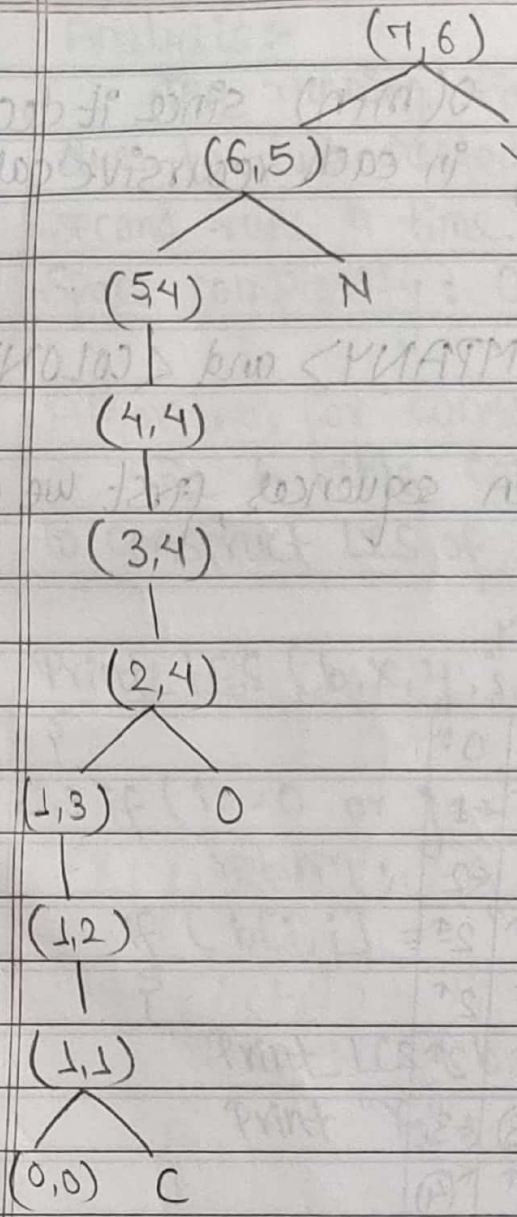
Example:
Find the LCS of <COMPANY> and <COLONY>.

Sol:
To find the LCS of given sequences, first we construct b and c table as:

		j→	C	O	L	O	N	Y
i		0	1	2	3	4	5	6
↓	0	0	0	0	0	0	0	0
C	1	0	①	←1	←1	←1	←1	←1
O	2	0	1↑	↖2	←2	②	←2	←2
M	3	0	1↑	2↑	2↑	2↑	2↑	2↑
P	4	0	1↑	2↑	2↑	2↑	2↑	2↑
A	5	0	1↑	2↑	2↑	2↑	2↑	2↑
N	6	0	1↑	2↑	2↑	2↑	③	←3
Y	7	0	1↑	2↑	2↑	2↑	3↑	④

Therefore, length of LCS = $C[7,6] = 4$

To find the LCS we begin at $b[7,6]$ and trace through the table by following the arrow. Whenever "↖" obtained, it indicates $x_i == y_j$ is an element of LCS. So LCS in this case is <CONY>


(7,6)
(6,5)
Y
(5,4)
N
(4,4)
(3,4)
(2,4)
(1,3)
O
(1,2)
(1,1)
(0,0)
C

Example:-

Determine the LCS of :-

a) <XMJYAVZ> and <MZJWXY>

b) <10010101> and <010110110>

a) Sol?

To find the LCS of given sequences, first we construct b and c table as:

		j→	M	Z	J	W	X	Y
i		0	1	2	3	4	5	6
↓	0	0	0	0	0	0	0	0
X	1	0	0↑	0↑	0↑	0↑	↖1	←1
M	2	0	↖1	←1	←1	←1	↖2	←2
J	3	0	1↑	↖2	↖2			
Y	4	0						
A	5	0						
V	6	0						
Z	7	0						

RS

2071

0/1 knapsack Problem:-

A thief has a bag or knapsack that contain maximum weight w of his loot. There are n-items and each item has some weight w_i and value v_i. An amount of item can be put in a bag is either 0 or 1 (i.e. either rejected or accepted). Here, the problem is to collect the items in a knapsack that maximize the total profit earned. We can formally state this problem as:

maximize $\sum_{i=1}^{n} x_i v_i$ using the constant $\sum_{i=1}^{n} x_i w_i \leq w$

Simple approach for 0/1 knapsack problem:-

One method for solution of 0/1 knapsack problem is to find all possible combination and select the combination that fit into the knapsack and maximize the profit. There are 2^n possible combination of items as there are n items so using this straight forward approach, the complexity of choosing one combination of items from 2^n different combination requires $O(2^n)$.

RS

Date
Page

Dynamic Programming approach for 0/1 knapsack Problem:-

Let us assume that c[i,w] is the maximum profit for item 1 to i and maximum capacity of knapsack w. Then we can define the recursive formula for computing c[i,w] as:

$$c[i,w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ c[i-1,w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w-w_i],\ c[i-1,w]) & \text{if } i>0 \text{ and } w \geq w_i \end{cases}$$

This relation expresses that:

i) If we have no item i.e. $i=0$ or maximum capacity of knapsack is 0 i.e. $w=0$, then maximum profit is 0.

ii) If the weight of item being added exceeds the capacity of knapsack then value of profit is equal to value of profit upto previous addition.

iii) If the capacity of knapsack is greater than or equal to weight of item being added then profit is maximum between value upto previous addition ite

RS

or value after addition of last item.

Example:

Given a knapsack with capacity

$W = 5$ kg

$n = \{1, 2, 3, 4\}$

$w_i = \{2, 3, 4, 5\}$

$v_i = \{3, 4, 5, 6\}$

Find the maximum profit obtained by this item set and find items are to be added in knapsack using 0/1 knapsack algorithm.

Soln:

Let us construct the value table $C[i, w]$

		items(i) ↓ → knapsack capacity(w)	0	1	2	3	4	5
v_i	w_i	0	0	0	0	0	0	0
3	2	1	0	0	3	3	3	3
4	3	2	0	0	3	4	4	7
5	4	3	0	0	3	4	5	7
6	5	4	0	0	3	4	5	7

Hence, total maximum profit from given item set is 7 and item to be added into knapsack are 1, 2.

RS

Q: There are 7 items, where,

$V_i = \{2, 3, 3, 4, 4, 5, 7\}$

$W_i = \{3, 5, 7, 4, 3, 9, 2\}$

$W = 15$

Find the maximum profit and item to be added into knapsack.

Sol:

Let us consider the value table $C[i, w]$

items (i) ↓ knapsack capacity (w) →

V_i	W_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	3	1	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2
3	5	(2)	0	0	0	2	2	3	3	3	5	5	5	5	5	5	5
3	7	3	0	0	0	2	2	3	3	3	5	5	5	5	6	6	6	8
4	4	(4)	0	0	0	2	4	4	4	6	6	7	7	7	9	9	9	9
4	3	(5)	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
5	9	6	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
7	2	(7)	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	(18)

$I = \{I_2, I_4, I_5, I_7\}$

RS

Algorithm for 0/1 Knapsack Problem:-

Algorithm (computing the value table)

```
Dyna_knapsack (W, n, v, w)
{
    for (w=0; w≤W; w++)
    {
        c[0,w] = 0;
    }
    for (i=1; i≤n; i++)
    {
        c[i,0] = 0;
    }
    for (i=1; i≤n; i++)
    {
        for (w=1; w≤W; w++)
        {
            if (wt[i] ≤ w)
            {
                if ((v[i] + c[i-1, w-wt[i]]) > c[i-1,w])
                {
                    c[i,w] = v[i] + c[i-1, w-wt[i]]
                }
                else
                {
                    c[i,w] = c[i-1,w];
                }
            }
            else c[i,w] = c[i-1,w]
        }
    }
}
```

Algorithm (item to be added in knapsack):
This algorithm uses C-table for determining the item to be added in knapsack for achieving max profit.

```
start from C[n,W]
    i=n, w=W
    while (i>0 && w>0)
    {
      if (C[i,w] ≠ C[i-1,w])
      {
        mark ith item
        i ← i-1;
        w ← w - wt[i];
      }
      else
        i ← i-1;
    }
```

Analysis:-
The complexity of first algorithm is O(nW)
The complexity of second algorithm is O(n).
So, overall complexity of 0/1 knapsack is O(nW).

RS

optimal sol^n findout garne technique tha nahuda samma use gardainau

2.3) Greedy Paradigm:- 2067

Fractional knapsack Problem:-

Statement:

A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and weight of i^{th} item is w_i and value v_i. Any amount of item can be put into the bag i.e. x fraction of item can be collected where $0 \leq x \leq 1$. Here, the problem is to collect the item in a knapsack that maximize the total profit earned. We can formally state that the problem as maximize $\sum_{i=1} x_i v_i$ using the constant $\sum_{i=1} x_i w_i \leq W$, where w_i is the weight of i^{th} item and $0 \leq x_i \leq 1$.

Greedy approach for fractional knapsack Problem:-

The greedy approach for fraction knapsack problem is:

i) Calculate the value per weight of each item i.e. v_i/w_i

ii) Sort the item with respect to v_i/w_i in decreasing order.

RS

iii) Take as much as item with highest v_i/w_i as we can, if item is finished move on next item that has highest v_i/w_i and continue this until knapsack is full.

Example:-

Assume knapsack with maximum capacity $W = 16$ and there are 6 items with their weight and value as:

Weight = {6, 10, 3, 5, 1, 3}

value = {6, 2, 1, 8, 3, 5}

What is the maximum profit and item to be collected in knapsack using greedy approach of fractional knapsack problem.

Sol:

Now calculate value per-weight.

item	weight(w_i)	value(v_i)	v_i/w_i
I_1	6	6	1.00
I_2	10	2	0.20
I_3	3	1	0.33
I_4	5	8	1.60
I_5	1	3	3.00
I_6	3	5	1.67

Sort the item in decreasing order w.r.t V_i/W_i.

Item	weight (W_i)	value (V_i)	V_i/W_i
I_5	1	3	3.00
I_6	3	5	1.67
I_4	5	8	1.60
I_1	6	6	1.00
I_3	3	1	0.33
I_2	10	2	0.20

Now, our objective is to fill the knapsack with item to get max. profit without crossing the weight limit 16.

Now, draw the table to represent knapsack.

item taken	weight (W_i)	value (V_i)	total weight in knapsack	Remaining weight	Profit Earned
I_5	1	3	1	15	3
I_6	3	5	4	12	8
I_4	5	8	9	7	16
I_1	6	6	15	1	22
I_3	1	0.33	16	0	22.33

∴ Total maximum profit = 22.33 &

Item to be collected in bag = $< I_5\ I_6\ I_4\ I_1\ I_3 >$

$= <1\ 1\ 1\ 1\ 1/3>$

Date
Page

complexity: O(n)

Algorithm :-

- Let v[1...n] and w[1...n] contains the value and weight respectively of n objects sorted in decreasing order of v[i]/w[i].
- Let [W] be the capacity of knapsack and x[1...n] is the solution vector that includes fractional amount of items.
- Then, the greedy fractional knapsack algorithm is:

```
GreedyKnapsack (W, n)
{
  for (i=0; i<=n; i++)
    x[i] = 0;
  tempw = W    // Remaining capacity
  for (i=1; i<=n; i++)
  {
    if (w[i] > tempw)
      break;
    else
      {
        x[i] = 1.0;
        tempw = tempw - w[i];
      }
  }
  if (i<=n)
    x[i] = tempw / w[i];
  return(x);
}
```

RS

Analysis:-

The above algorithm just contains the single loop hence running time of above algorithm is $\theta(n)$.

However, our requirement is that $v[1...n]$ and $w[1...n]$ are sorted so sorting takes $O(n \log n)$. Thus, overall complexity including sorting becomes $O(n \log n)$.

used for generating prefix code

Huffman Algorithm and Huffman Code:-

2081

Huffman algorithm is the greedy algorithm that constructs an optimal prefix code called Huffman code for each character in the message or file to compress the size of the file. Huffman codes are used to compress data by representing each alphabet by unique binary code in an optimal way. The Huffman coding is the variable length coding system that assigns smaller code for more frequently ~~used~~ characters and longer code for less frequently used characters in order to reduce size of file being compressed.

2070(2)
2071(2)

Prefix Code:-

Prefix code is a binary code given to each character in a file such that no any code is prefix of another code.

RS

Eg: a = 0 b = 101 c = 100 d = 111 e = 1101 f = 1100

Let us consider a scenario a file with following data:

XXXXXXYYYYZZ. Here f(X) = 6, f(Y) = 4, f(Z) = 2.

If we use fixed length code for representing such a file, we need two bits to represent 3 characters like X = 00 Y = 01 Z = 10

Using this method of fixed length code, total number of bits required to represent a file is 2 x 12 = 24. If above data were represented using Huffman coding then more frequently occurring characters would be represented by smaller bits like X = 0 Y = 10 Z = 11.

Then using this variable length prefix code, total bit required is: 6 x 1 + 4 x 2 + 2 x 2 = 18.

RS

Constructing the Huffman Code

To generate the Huffman code, we create a binary tree called Huffman tree. The process of building a Huffman tree is as follows:

i) Maintains a priority queue in ascending order of frequency representing a tree of single node for each character.

ii) At each step extract two min node, delete from the queue, merge two node into a single tree and insert it into a priority queue until all of the partial Huffman tree were combined into one.

Example:-

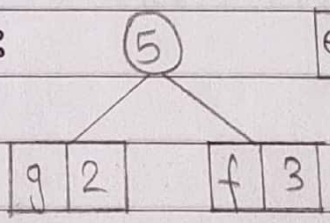
Let us consider a file or a message contains 100 characters with 7 distinct characters {a,b,c,d,e,f,g}. The frequency of each character are {40,20,15,12,8,3,2} respectively. Obtain the optional prefix code for each character using Huffman algorithm.

Soln:

Initial Priority Queue.

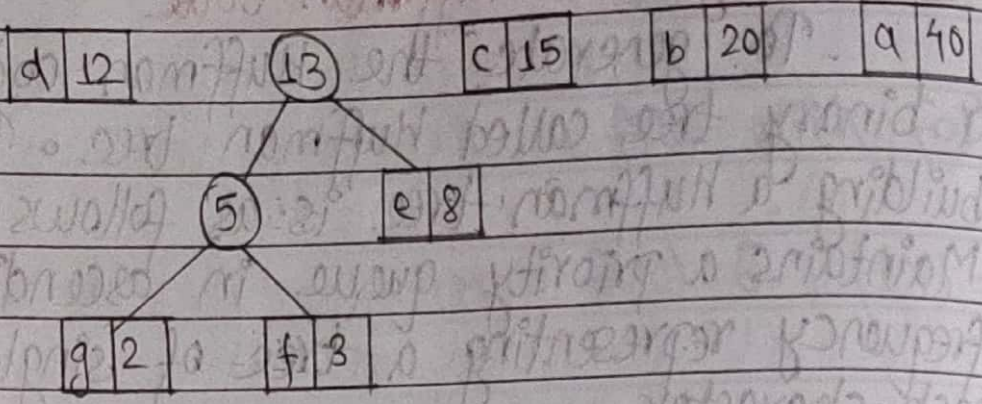
g	2	f	3	e	8	d	12	c	15	b	20	a	40

Step 1:

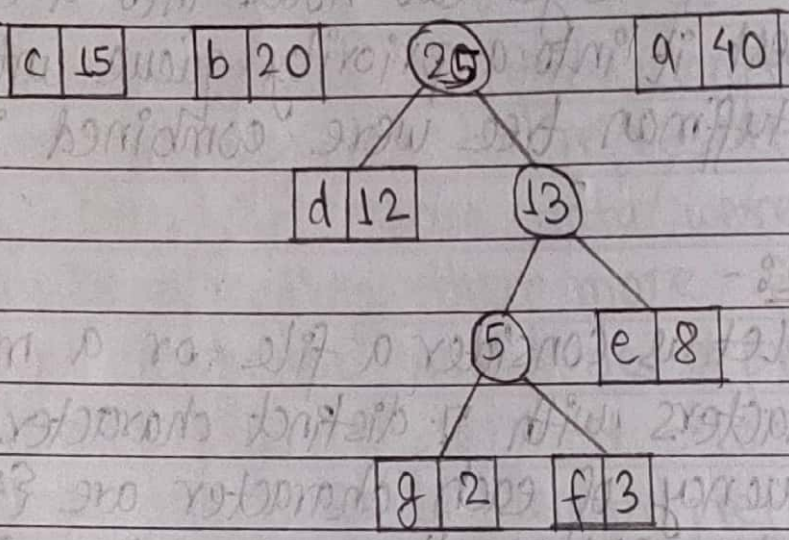


e	8	d	12	c	15	b	20	a	40

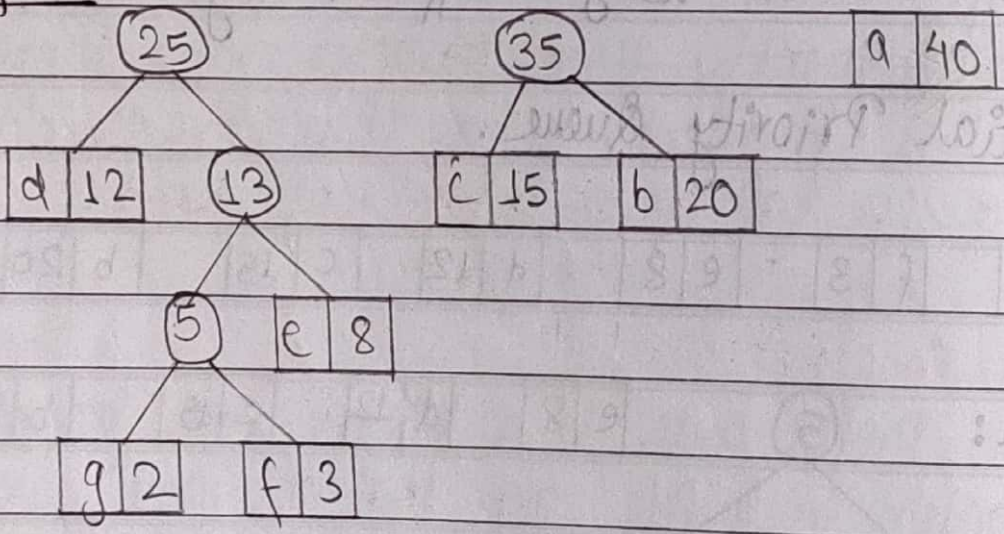
Step 2:



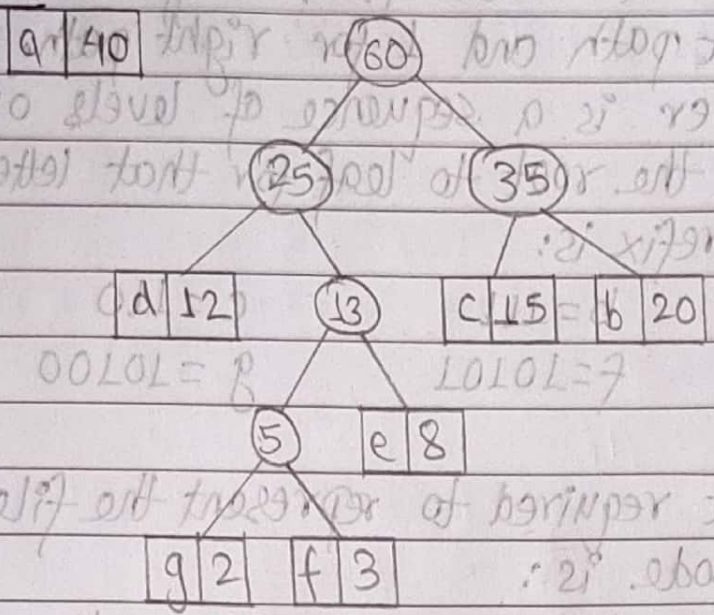
Step 3:



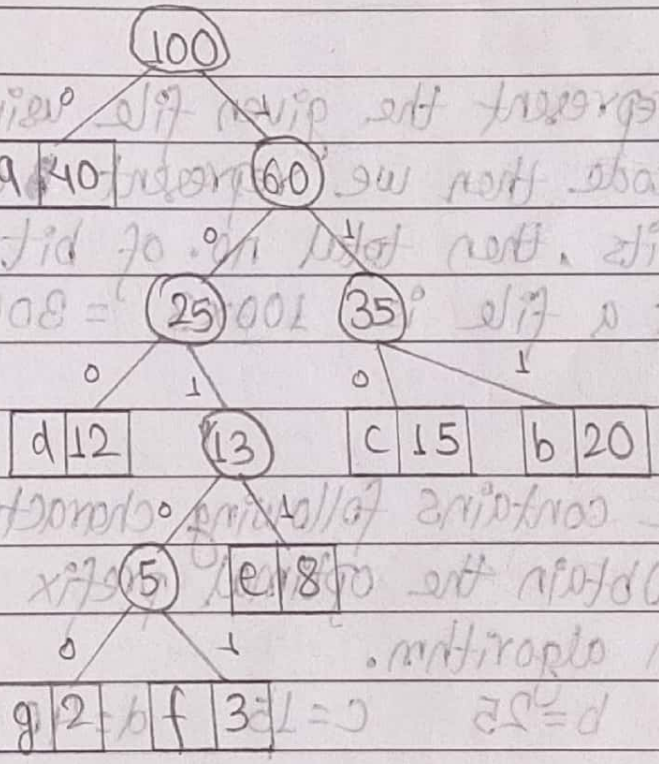
Step 4:



Step 5:


a 40
60
25
35
d 12
13
c 15
b 20
5
e 8
g 2
f 3

Step 6:


100
a 40
60
25
35
0
1
0
1
d 12
13
c 15
b 20
0
1
5
e 8
0
1
g 2
f 3

Date
Page

To create a code from Huffman tree, assign 0 for left path and 1 for right path. The code for a letter is a sequence of levels on the edge, containing the root to leaf for that letter. So, optimal prefix is:

a = 0 b = 111 c = 110 d = 100
e = 1011 f = 10101 g = 10100

Total bit required to represent the file using Huffman code is:

$= 40\times1 + 20\times3 + 15\times3 + 12\times3 + 8\times4 + 3\times5 + 2\times5$
$= 238$

If we represent the given file using the fixed length code then we represent each character by 3-bits, then total no. of bit required to represent a file is $100\times3 = 300$.

A file contains following characters with frequency below. Obtain the optimal prefix code using Huffman algorithm.

a = 50 b = 25 c = 15 d = 40 e = 75

RS

2067 W

Huffman Algorithm :-

```
Huffman (C)        // C is the distinct character set.
{
    n = |C|        // no. of distinct characters
    Q = C          (* maintaining the priority queue in
                      ascending order of frequency *)
    for (i=1; i≤n-1; i++)
    {
        Allocate new node z
        x = Extract_min(Q);
        y = Extract_min(Q);
        z.left = x;
        z.right = y;
        f(z) = f(x) + f(y)
        insert(Q, z)
    }
    return Extract_min(Q);
}
```

Analysis:-

We can use build heap to create a priority queue that takes $O(n)$ time.
The heap executes $(n-1)$ times and each heap operation takes $O(\log n)$. Thus total running time complexity of heap is $O(n \log n)$. Thus, running time of Huffman algorithm is

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

RS

Job Sequencing with Deadline :-

Problem Statement:-

The job sequencing problem is stated as follows:

There are n jobs to be processed on a machine, each job i has deadline $d_i > 0$ and profit ≥ 0. The profit is earned if and only if the job is completed by its deadline. The job is complete if it is processed on a machine for one unit of time. There is only one machine available for processing a job and only one job is processed at a time on a machine. Our objective is to find feasible subset of job such that profit is maximized.

Feasible Solution:-

The feasible solution for this problem is select J of jobs such that each job in this subset can be completed within deadline.

Optimal Solution:-

The optimal solution for this problem is a feasible solution with maximum profit.

Example:

Let $n = \{P_1, P_2, P_3, P_4\} = \{100, 10, 15, 27\}$ & deadline $d_i = \{d_1, d_2, d_3, d_4\} = \{2, 1, 2, 1\}$ of jobs, find the optimal solution for this problem.

Sol:

Now feasible solution and their profits are:

	feasible solution	processing sequence	profit
1	$\{I_1\}$	$\{I_1\}$	100
2	$\{I_2\}$	$\{I_2\}$	10
3	$\{I_3\}$	$\{I_3\}$	15
4	$\{I_4\}$	$\{I_4\}$	27
5	$\{I_1, I_2\}$	$\{I_2, I_1\}$	110
6	$\{I_1, I_3\}$	$\{I_1, I_3\}$ or $\{I_3, I_1\}$	115
7	$\{I_1, I_4\}$	$\{I_4, I_1\}$	127
8	$\{I_2, I_3\}$	$\{I_3, I_2\}$	25
9	$\{I_3, I_4\}$	$\{I_4, I_3\}$	42

Here, the solution 7 is optimal solution and maximum profit is 127. So, the feasible subset of job that achieve maximum profit is $\{I_1, I_4\}$ and processing sequence is $\{I_4, I_1\}$.

The straight forward way of finding the solution of job sequence problem takes $O(2^n)$ because there are 2^n combination of job and optimal solution is 1 of the sequence from 2^n combination.

RS

Greedy Method for Job Sequencing with Deadline:

The objective for job sequencing problem with deadline is to maximize the profit. Using this method, the greedy strategy is to sort the job with respect to profit in decreasing order and the job are selected from the order in solution if they are feasible. So, greedy algorithm for job sequencing problem can be expressed as below:

2069

Algorithm:-

input: n jobs with deadline d[] and profit P[]

output: Feasible subset of jobs that maximize the profit (i.e. optimal solution)

```
Greedy_jobsequence (n,d,P)
{
  - sort the job with decreasing order of profit.
  - J = Job[1];
  - Assign time slot for Job[1] = [r-1, r]
    where r is the deadline of Job[1].
  - Profit = P[1],
  for (i=2; i<=n; i++)
  {
      if (J U {Job[i]} is feasible)
          J = J U Job[i];
```

- Assign time slot for Job[i] = [r-1, r] if it is free, otherwise check the slot until first
- Profit = Profit + P[i];
}
}
return J and profit
}

Analysis:-

For sorting the job time required = O(n log n).
For loop executes (n-1) times and inside loop the assignment of slot to select job takes n steps so time complexity of loop is $O(n^2)$.

Hence, total time complexity of this greedy algorithm is $O(n \log n) + O(n^2)$
$= O(n^2)$.

Example:-

n=4 Jobs = {1, 2, 3, 4}
d_i = {2, 1, 2, 1}
P_i = {100, 10, 15, 27}

Find the feasible subset of jobs that maximize the profit and find the maximum profit also.

Sol:

Sort the jobs in according to decreasing order of their profits.

Jobs	deadline (di)	Profit (P_i)
J_1	2	100
J_4	1	27
J_3	2	15
J_2	1	10

initially $J = \phi$ and profit = 0.

⇒ Consider the job 1 and add to set J. so, $J = \{J_1\}$
time slot = [1,2]
profit = 100

⇒ Consider the job4 (i=2)
{1,4} is feasible, add Job4 to set J
$J = \{J_1, J_4\}$
time slot = [0,1]
profit = profit + p[i] = 100 + 27 = 127

⇒ Consider the job 3 (i=3)
$\{J_1, J_4, J_3\}$ is not feasible so discard it.

⇒ Consider the job 2 (i=4)
$\{J_1, J_4, J_2\}$ is not feasible so discard it.

Hence the feasible subset of job that maximize the profit is $\{J_1, J_4\}$.

Maximum profit = 127
Processing sequence = {J4, J1}.

2069

Example:-

Consider the following jobs with profit and deadline. Find the job sequence that maximize the profit.

Jobs	P_i	d_i
1	20	2
2	10	1
3	5	3
4	15	2
5	1	3

RS

Date
Page

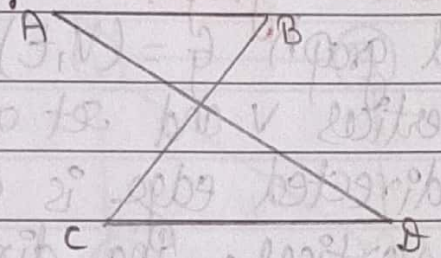
UNIT : 3

3.1) Graph Algorithms:

Graph:-

Graph is a collection of vertices or nodes connected by the collection of edges. Formally, graph is the pair $G = (V, E)$ where V denotes non-empty set of vertices and E denotes edges connecting two vertices.

Example:



where,

$V = \{A, B, C, D\}$

$E = \{(A,B), (A,D), (B,C), (C,D)\}$

Applications of graphs:-

There are number of applications of graph. Some of them are as follows:

i) Graphs are used to model the geographic map of cities in which city can be represented by nodes or vertices and road connecting such cities are represented by an edge.

RS

ii) Graphs are used to model computer networks in which each node is a machine and link between them represent the edge.

iii) Graph with weight assigned to their edge can be used to solve problem such as finding the shortest path between two cities.

iv) Using the graph as a model, we can determine whether it is possible to walk down all the street in the city without going down a street twice.

Directed and Undirected graph:

A directed graph $G = (V, E)$ consists of non-empty set of vertices V and set of directed edges E. Each directed edge is associated with an ordered pair of vertices. The directed edge associated with the order pair (U, V) is said to start at U and end at V.

An undirected graph $G = (V, E)$ consists of non-empty set of vertices V and set of undirected edges E. Each undirected edge is associated with an unordered pair of vertices i.e. in an undirected graph we consider (U, V) and (V, U) be the same edge.

RS

Example:

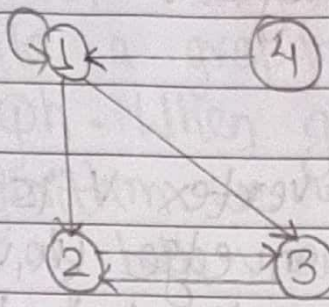


Fig: directed graph

$V = \{1,2,3,4\}$

$E = \{(1,1), (1,2), (1,3), (2,3), (3,2), (4,1)\}$

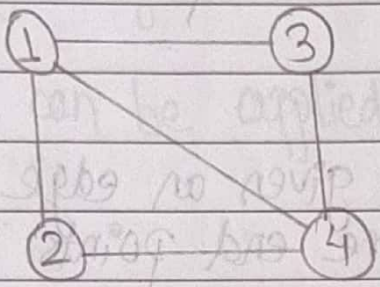


Fig: undirected graph

$V = \{1,2,3,4\}$

$E = \{(1,2), (1,3), (1,4), (2,4), (3,4)\}$

RS

Date
Page

Some Graph Terminologies :-

We say that vertex V is adjacent to vertex u if there is an edge (u,v) in the graph G. when graph is undirected, the adjacency relation is symmetric, if graph is directed, the adjacency relation is not necessarily symmetric.

- In directed graph, given an edge e = (u,v) we say that u is the origin of e and v is the destination of e i.e. e is incident from u and is incident to v.

- In undirected graph, given an edge e = (u,v) we say that u and v are end points of edge i.e. the edge e is incident on both u and v.

- In directed graph number of edges coming out of the vertex is called out-degree of that vertex and number of edges coming in, is called the in-degree.

- The degree of vertex in an undirected graph is the number of edge incident on it.

- The degree of vertex in directed graph is its in-degree plus out-degre.

RS

- A graph with relatively few edges is called sparse graph while a graph with many edges is called dense graph. When giving the running time of graph algorithm we will usually express it as a function of both v and e.

Graph representation :-

Generally, we represent the graph in two ways :

i) Adjacency matrix
ii) Adjacency list

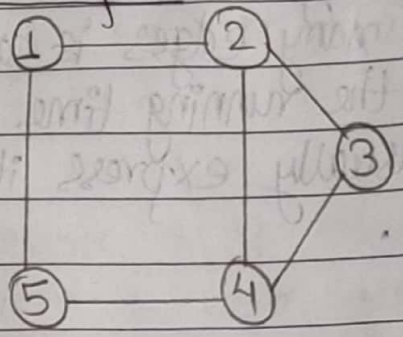
Both way can be applied to represent any kind of graph.

i) Adjacency matrix representation :

Let $G = (V, E)$ be a simple graph with n vertices $v_1, v_2, \ldots, v_n$. The adjacency matrix $A(G)$ with respect to given ordered list of vertices is $n \times n$ matrix $A = a_{ij}$ such that

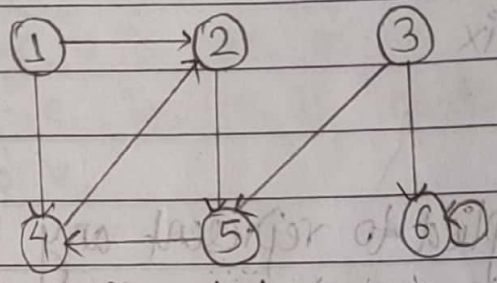
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \text{ (edges)} \\ 0 & \text{otherwise} \end{cases}$$

Example:



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	0
3	0	1	0	1	0
4	0	1	1	0	1
5	1	0	0	1	0

Fig: Undirected graph



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

Fig: Directed graph.

Advantages:-

i) Edge insertion and deletion of adjacency matrix takes O(1) time.

ii) It is very easy to check whether there is edge from vertex u to vertex v.

iii) Easier to implement.

Date
Page

Disadvantages:-

i) The adjacency matrix of the graph requires $\Theta(V^2)$ space independent of the number of edges in the traph.

2) Adjacency List Representation:-

The adjacency list representation of the graph $G = (V, E)$ consists of an array adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list adj[u] contains all the vertices adjacent to u in G.

Example:

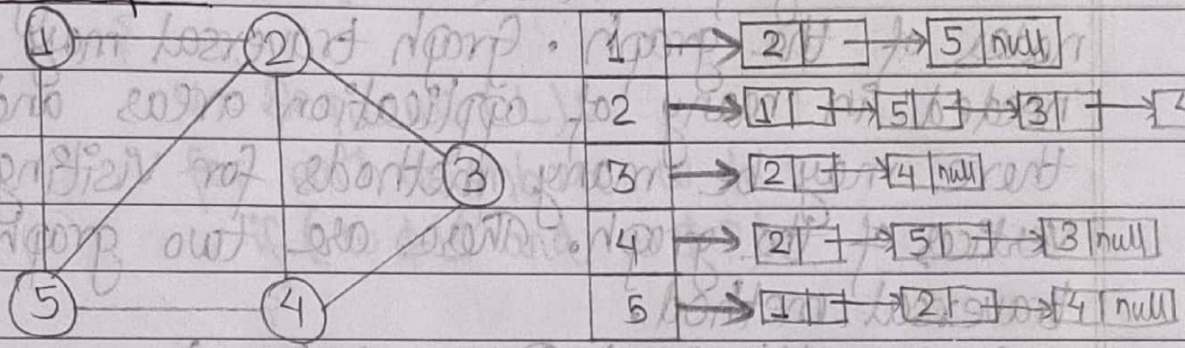


Fig:- Undirected graph

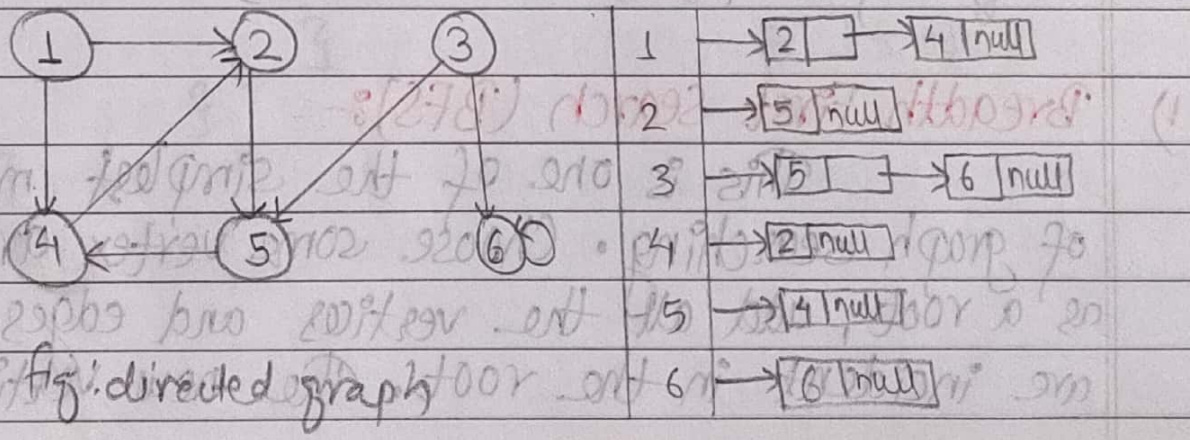


Fig:- directed graph

RS

Advantages:

i) It saves the space i.e. it require $O(|V|+|E|)$
ii) To find the degree of vertex adjacency list is good.

Disadvantages:

i) Searching for some degree (i,j) requires $O(d)$ times where d is the degree of vertex.
ii) It takes more time to determine whether an edge (u,v) a given is present or not.

Graph Traversal :-

A graph traversal means visiting all the nodes of the graph. Graph traversal may be needed in many of application areas and there may be many methods for visiting the vertices of the graph. There are two graph traversal method.

1) Breadth First Traversal (BFS)
2) Depth First Traversal (DFS)

1) Breadth First Search (BFS):-

This is one of the simplest method of graph searching. Choose some vertex arbitrarily as a root, add all the vertices and edges that are incident in the root. The new vertices

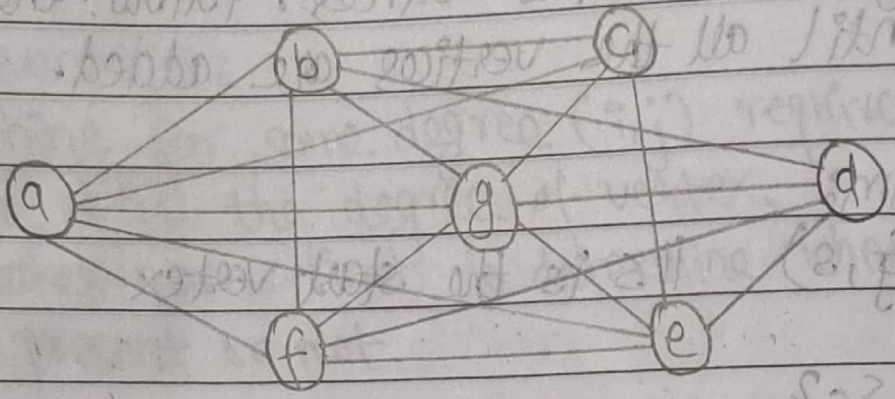
RS

added will become the vertices at the level 1 of BFS tree. From the set of added vertices of level 1, find other vertices such that they are connected by edges at level 1 vertices. Follow the above step until all the vertices are added.

Algorithm:

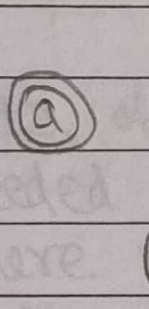
```
BFS (G,s)   // s is the start vertex
{
   T = {s}
   L = ϕ    // an empty queue
   Enqueue (L,s)
   while (L != ϕ)
    {
      v = Dequeue (L);
      for each neighbour w to v
        if (w ∉ L and w ∉ T)
        {
          Enqueue (L,w);
          T = T ∪ {w}   //put edge (v,w) also
        }
    }
}
```

Example: Use the breadth first search to find a BFS tree of the following graph.



Sol:

Step1:

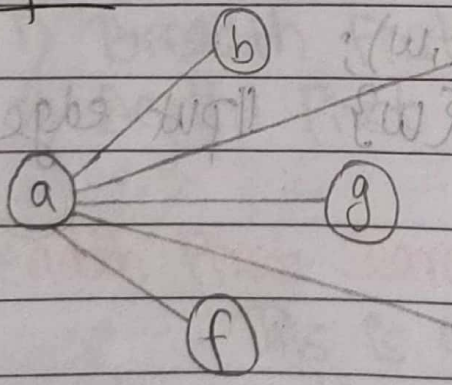


T = {a}

L = {a}

v = a

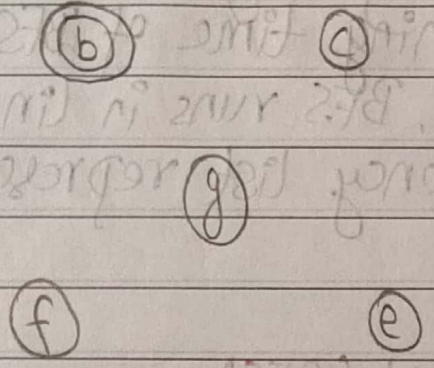
Step2:



T = {a, b, c, e, f, g}

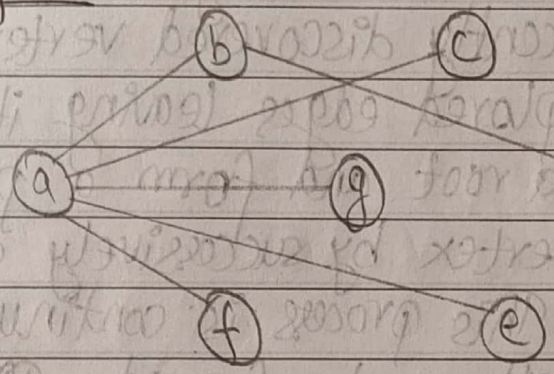
L = {b, c, e, f, g}

Step 3:



T = {a, b, c, e, f, g}
L = {c, e, f, g}
V = b

Step 4:



T = {a, b, c, d, e, f, g}
L = {c, e, f, g}

Analysis:-

From the above algorithm, all the vertices are put once in the queue and they are accessed. The operations of enqueuing and dequeuing takes O(1) so, the total time devoted to queue operation O(V). Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once. Since the sum of the length of all the adjacency list is θ(E), the total time

RS

spent in scanning adjacency list is $O(E)$. Thus, the total running time of BFS is $O(|V|+|E|)$ or, $O(V+E)$. Thus, BFS runs in linear time in size of the adjacency list representation of G.

2) Depth First Search Traversal (DFS):-

In depth first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it i.e. choose a vertex as a root and form a path by starting at a root vertex by successively adding vertices and edges. This process is continued until no possible path can be formed. If the path contains all the vertices then the tree consisting this path is DFS tree, otherwise, we must add other edges and vertices. For this move back from the last vertex that is met in the previous path and find whether it is possible to find new path starting from the vertex just met. If there is such a path continue the process above. The whole process is continued until all the vertices are met. This method of search is also called back tracking

2072(3)

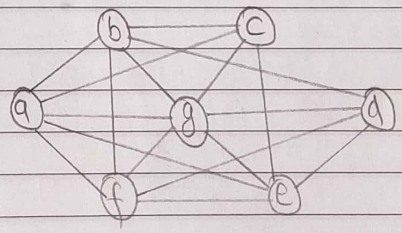
Algorithm:

```
DFS (G, s)
{
   T = {s};
   Traverse (s);
}
   Traverse (v)
   {
     for each w adjacent to v and not yet in T
     {
        T = T ∪ {w}  // put edge {v, w} also
        Traverse (w)
     }
   }
```

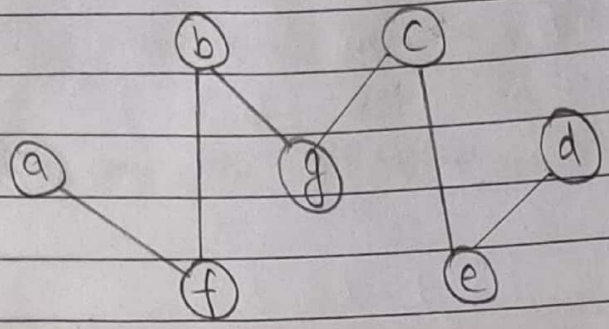
2072(5)

Example:

Use the depth first search to find the DFS tree of the following graph.



Sol:



T = {a, f, b, g, c, e, d}

Analysis:

From aggregate analysis we can write the complexity of O(V+E) because traverse function is invoked V times maximum and for loop executes O(E) time in total.

2067

Minimum Spanning Tree (MST):-

A tree is defined to be an undirected, acyclic and connected graph. Given a graph G, a spanning tree of a graph G is a sub-graph of G i.e. tree and contains all the vertices of G. A minimum spanning tree is a spanning tree but has a weight associated with edges and the total weight of the tree is minimum. Formally, given an undirected graph $G=(V,E)$, a sub-graph $T=(V,E')$ is a spanning tree of G if and only if T is a tree. The minimum spanning tree is the spanning tree of connected weighted graph such that total sum of weights of edges is minimum among all the spanning tree of that graph.

Applications of MST:-

i) In designing network.
ii) Finding the airline routes

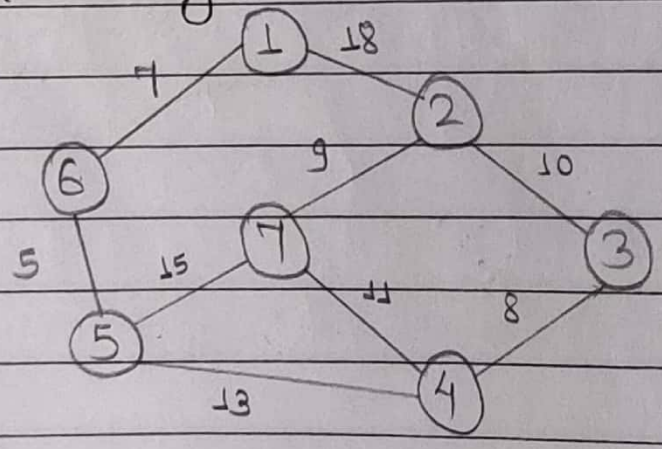
1) Kruskal's Algorithm :-

It is an algorithm in graph theory that find the minimum spanning tree for connected weighted graph. The main idea of algorithm is:

a) Create a forest F (a set of trees); where each vertex in the graph is separate tree.
b) Create a set S containing all the edges in non-decreasing order of weight.
c) While S is non-empty and F is not yet spanning
d) Remove an edge with minimum weight from S.
e) If that edge connects two different trees, then add it to the forest, containing two trees into a single tree (i.e. it doesnot creates a cycle).
f) Otherwise discard that edge.

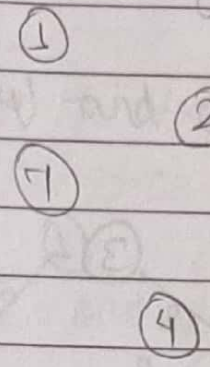
Example:

Find the MST of following graph using kruskal's algorithm.



Sol^n:

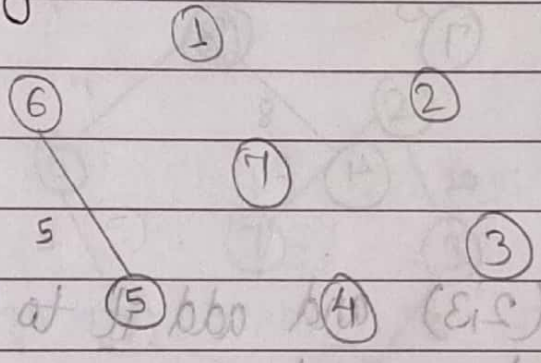
Initial forest



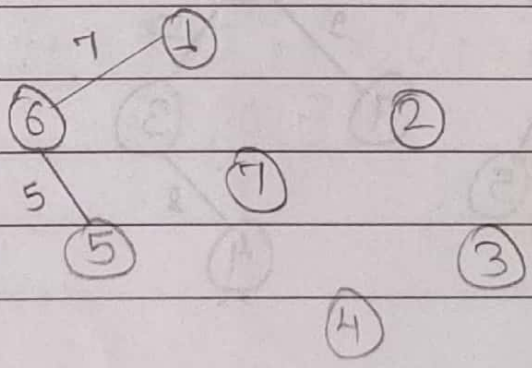
Sorted edge list is:

edges	(6,5)	(6,1)	(4,3)	(7,2)	(2,3)	(7,4)	(5,4)	(5,7)	(1,2)
weight	5	7	8	9	10	11	13	15	18

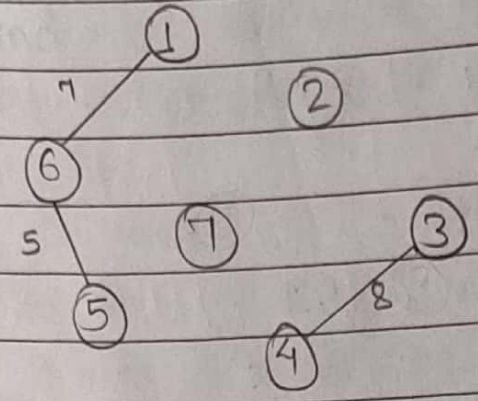
⇒ Select edge (6,5) and add it to T since it doesnot create a cycle.



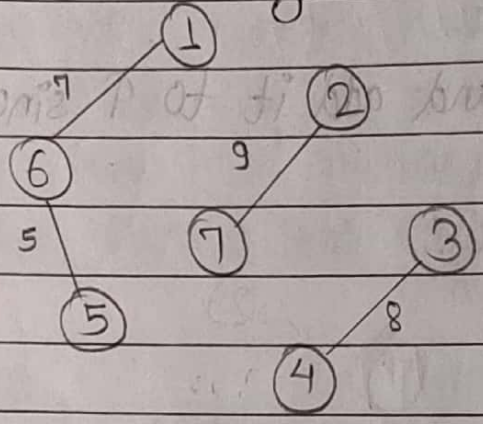
⇒ Select edge (6,1) and add it to T since it doesnot create a cycle.



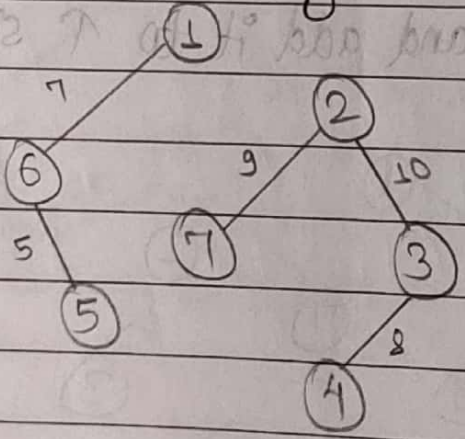
⇒ Select edge (4,3) and add it to T since it doesnot create a cycle.



⇒ Select edge (7,2) and add it to T since it doesnot create a cycle.

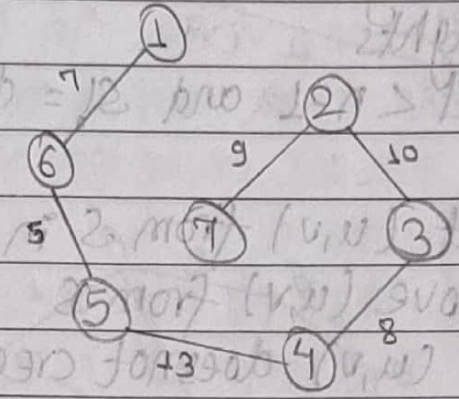


⇒ Select edge (2,3) and add it to T since it doesnot create a cycle.

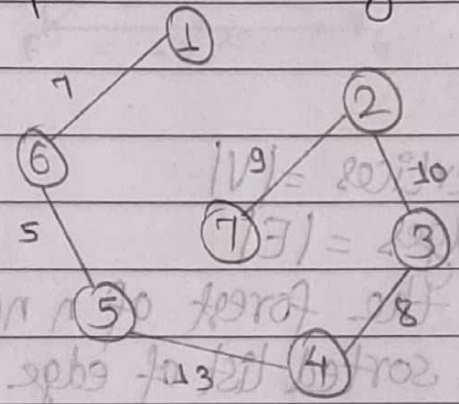


⇒ Select edge (7,4) and discard it, since it creates a cycle.

⇒ Select edge (5,4) and add it to T since it doesnot create a cycle.



Hence, the final MST using kruskal's algorithm is:



∴ Weight of MST = 52

Algorithm:

```
Kruskal_MST(G)
{
  T = {U}.  // forest of n nodes
  S = set of edges sorted in non-decreasing order
      of weights
  While ( |T| < n-1 and S != ϕ )
  {
    * select (u,v) from S in order
    * remove (u,v) from S
    *. if ( (u,v) does not create a cycle in T )
          T = T U {(u,v)}
  }
}
```

Analysis:

- Number of vertices = |V|
- Number of edges = |E|
- Maintaining the forest of n node requires O(V)
- Maintaining the sorted list of edge requires O(ElogE)
- Execution of while loop requires O(E) times thus the total complexity

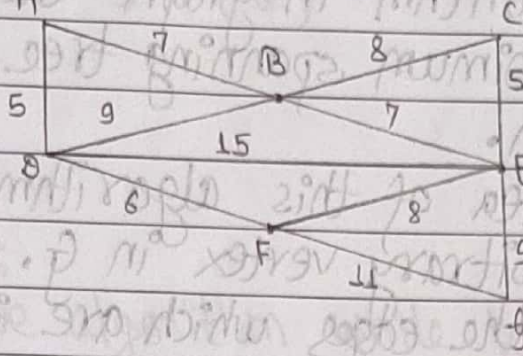
$$T(n) = O(V) + O(E\log E) + O(E)$$
$$= O(E\log E).$$

HW

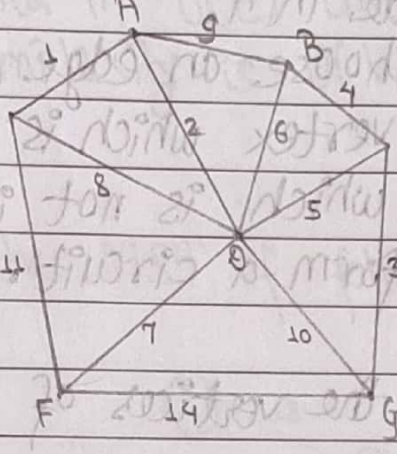
226

Date
Page

a)



b)



2069/2070

11) Prim's Algorithm:-

- It is the algorithm in graph theory that finds the minimum spanning tree for connected weighted graph.
- The main idea of this algorithm is:

a) Take any arbitrary vertex in G.
b) Arrange all the edges which are incident on choosen vertex, choose an edge of minimum weight and include in T.
c) At each stage, choose an edge of smallest weight joining a vertex which is already included in T and vertex which is not included yet so that it does not form a circuit. Then include it in T.
d) Repeat until all the vertices of G are included with (n-1) edges.

2069/2070

Algorithm:

```
Prim_MST(G)
{
    T = ϕ          // T is set of edge in G
    S = {s}        // s is randomly choosen start vertex
    while (S != V)
    {
        e = (u, v) an edge of minimum weight
            incident on vertex in S and not
            forming a circuit where u ∈ S and v ∈ V
```

$T = T \cup \{u, v\}$

$S = S \cup \{v\}$

}

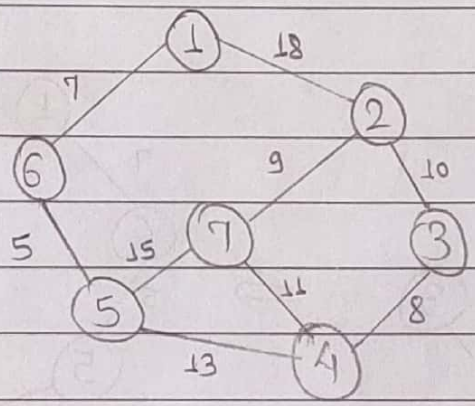
}

Analysis:

- In above algorithm while loop executes $O(V)$ times.
- The edge of minimum weight incident on vertex can be found in $O(E)$ times.
- So total complexity is $O(V \cdot E)$.

Example:

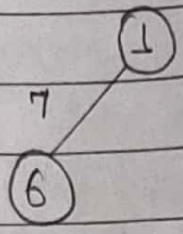
Find the MST of following graph using Prim's algorithm.



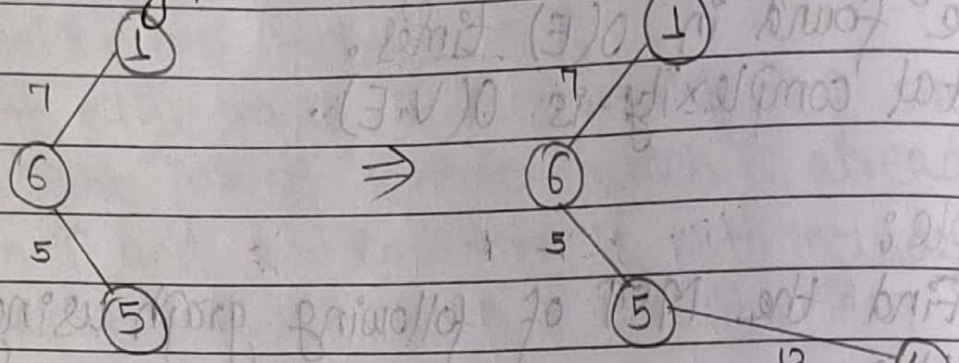
Sol:

Let us choose vertex 1 as an initial vertex.

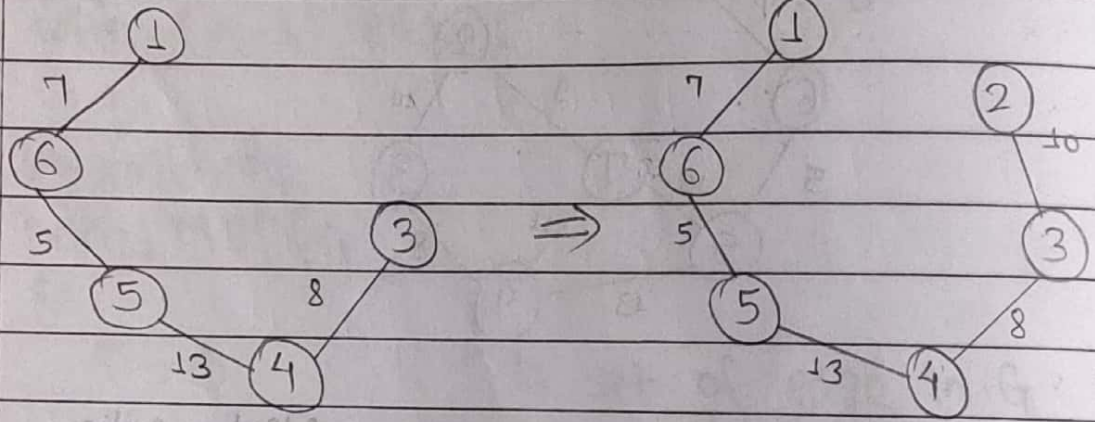
⇒ Now choose an edge incident on vertex 1 with minimum weight i.e. (1,6) and does not make cycle.



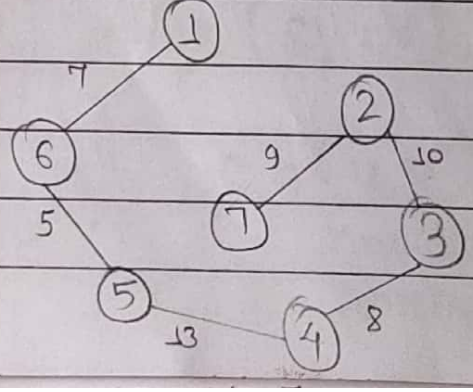
⇒ Similarly,



⇒



⇒

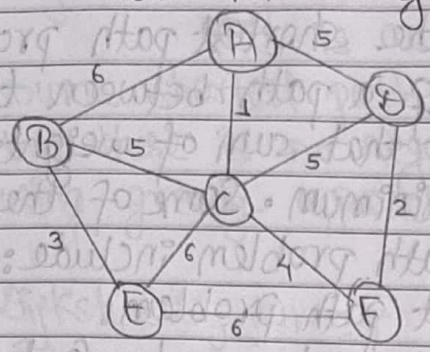


Exam?

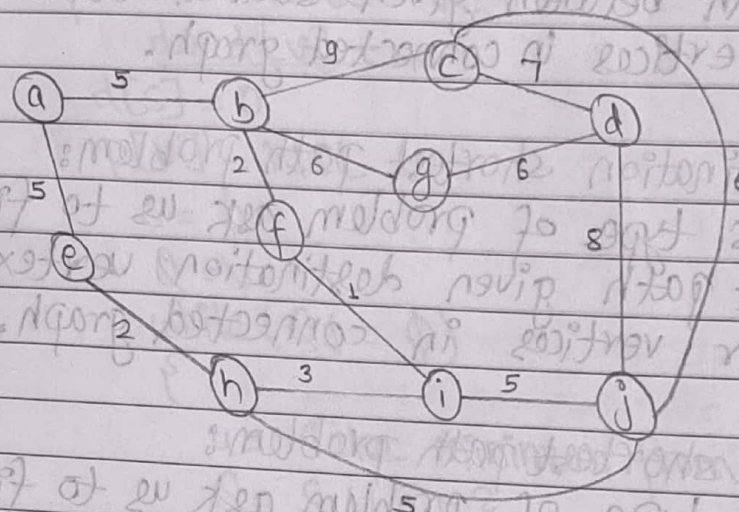
Trace the Prim's algorithm for following graph.

a)

2069



b)



a)

Soln:

Let

2068/23 ✓✓

Shortest Path Problem :-

In graph theory, the shortest path problem is the problem of finding a path between two vertices in a graph such that sum of weights of its consistent edge is minimum. Some of the variations of shortest path problem include:

I) Single source shortest path problem:
This type of problem ask us to find the shortest path between given source vertex to all other vertices in connected graph.

II) Single destination shortest path problem:
This type of problem ask us to find the shortest path given destination vertex from all other vertices in connected graph.

III) Single pair shortest path problem:
This type of problem ask us to find the shortest path from given source to given destination vertex.

IV) All pair shortest path problem:
This type of problem ask us to find the shortest path from all vertices to all other vertices in the graph.

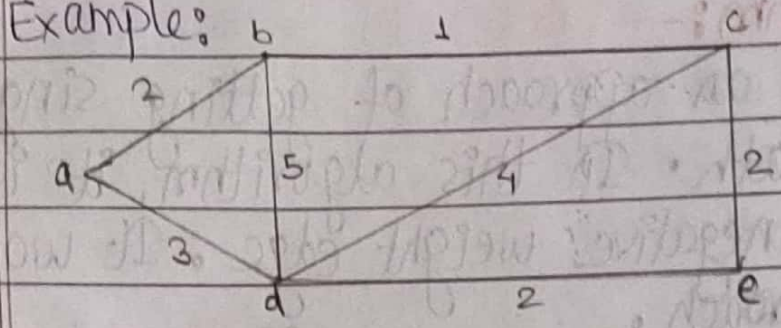
2068
2070

Dijkstra Algorithm:-

This is an approach of getting single source shortest path. In this algorithm, it is assumed that there is no negative weight edge. It works using greedy approach.

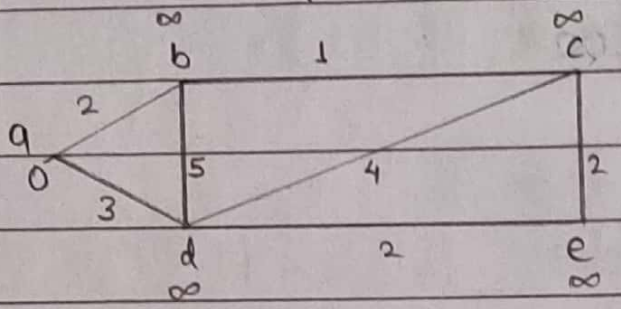
```
Dijkstra(G, w, s)
{
  for each vertex v∈V
      do d[v] = ∞
  d[s] = 0
  P[v] = NIL
  S = ϕ
  Q = V
  while(Q != ϕ)
  {
    u = Take minimum from Q and delete
    S = S ∪ {u}
    for each vertex v adjacent to u
        do if (d[v] > d[u] + w(u,v))
            then do d[v] = d[u] + w(u,v)
        P[v] = u
  }
}
```

Example:

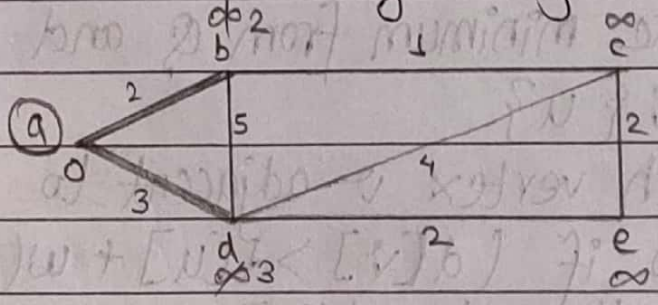


Sol:

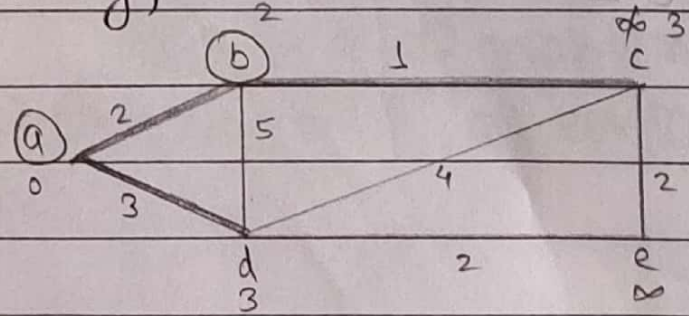
Initially, set the vertex 'a' as source i.e. $d[a] = 0$ and set the values of other vertices ∞ i.e. $d[a] = 0$, and $d[v] = \infty$

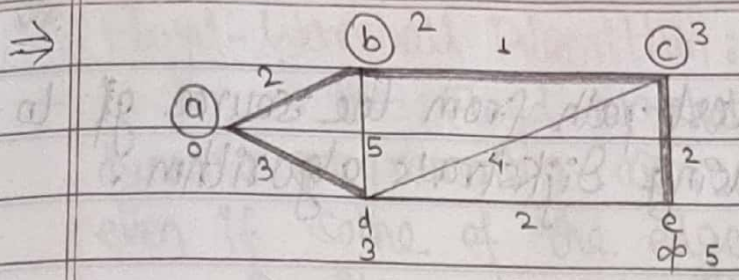


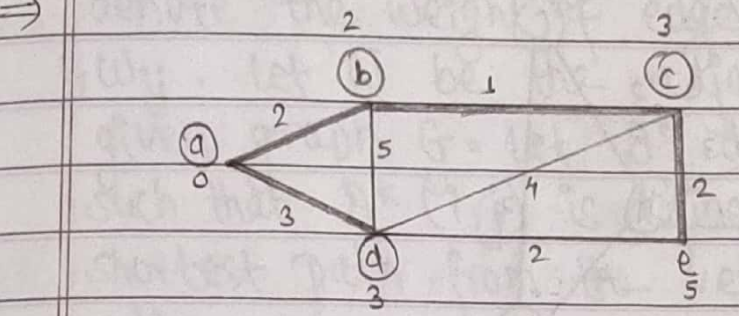
⇒ Select the vertex with minimum $d[v]$ i.e. vertex 'a' and relax all edges adjacent to 'a'.

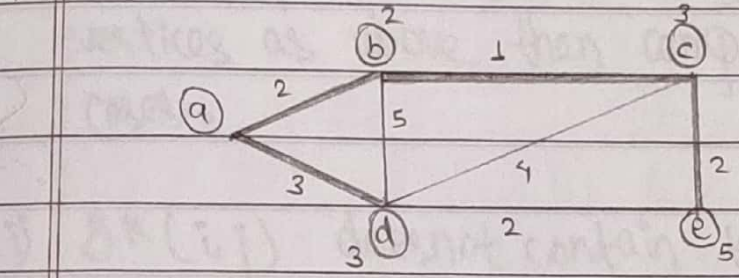


Similarly,









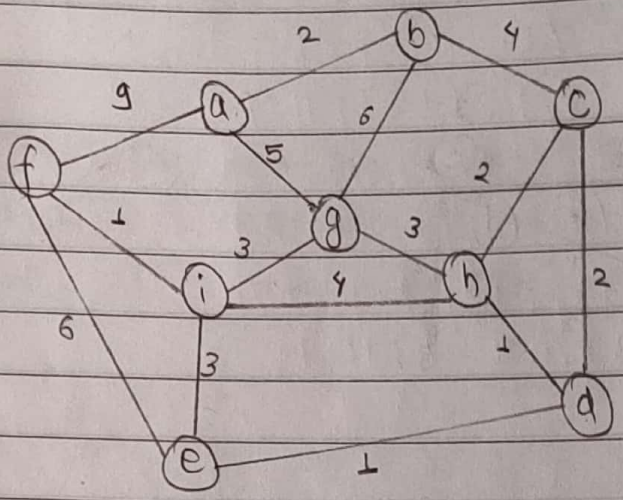
Analysis:-

In the above algorithm, the first loop block takes $O(V)$ time. Initialization of priority queue takes $O(V)$ times. The while loop executes for $O(V)$ time where for each execution the block inside the loop takes $O(V)$ time. Hence, total running time is: $O(V) + O(V) \cdot O(V)$

$= O(V^2)$.

Example:-

Find the shortest path from the source g to all other vertices using Dijkstra's algorithm.



Sol^n:

Floyd-Warshall Algorithm :-

The Floyd-Warshall Algorithm is based on dynamic programming approach. This algorithm works even if some of the edges have negative weights.

Consider a weighted graph $G = (V, E)$ and denote the weight of edge connecting vertices i and j by w_{ij}. Let W be the adjacency matrix for the given graph G. Let D^k denote an $n \times n$ matrix such that $D^k(i,j)$ is defined as the weight of the shortest path from the vertex i to j using only vertices $1, 2, \ldots, k$ as intermediate vertices in the path. If we consider shortest path with intermediate vertices as above then computing path contains two cases.

i) $D^k(i,j)$ doesnot contain k as an intermediate vertex.
ii) $D^k(i,j)$ contains k as an intermediate vertex.

Then we have the following relations:

$\Rightarrow D^k(i,j) = D^{k-1}(i,j)$ when k is not an intermediate vertex.

$\Rightarrow D^k(i,j) = D^{k-1}(i,k) + D^{k-1}(k,j)$ when k is an intermediate vertex.

So, from above relation

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^{k-1}(k,j)\}$$

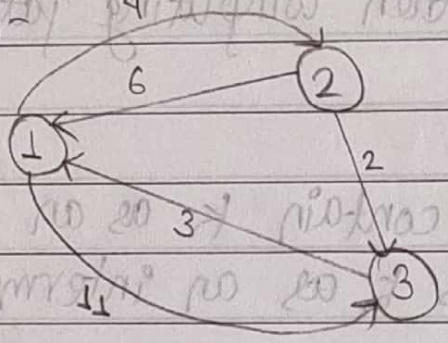
2069 (4)

Algorithm :-

```
FloydWarshall (W, D, n)   // W is adjacency matrix of graph G.
{
  for (i=1; i≤n; i++)
    for (j=1; j≤n; j++)
      D[i][j] = W[i][j]   // initially D[][] is D0.
  for (k=1; k≤n; k++)
    for (i=1; i≤n; i++)
      for (j=1; j≤n; j++)
        D[i][j] = min{D[i][j], D[i][k]+D[k][j]}
}
```

2069(4)

Example:-



Sol:

The adjacency matrix of given graph is :

$$W = D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{array}$$

Now, create D^1 from D^0, since diagonal $D^k(i,i)$ is not changed, so we not consider diagonal here.

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & - \\ 3 & - & 0 \end{bmatrix} \end{array}$$

$D^1(1,2) = \min\{D^0(1,2), D^0(1,1) + D^0(1,2)\}$
$= \min\{4, 0+4\}$
$= 4$

$D^1(1,3) = \min\{D^0(1,3), D^0(1,1) + D^0(1,3)\}$
$= \min\{11, 0+11\}$
$= 11$

$D^1(2,1) = \min\{D^0(2,1), D^0(2,1) + D^0(1,1)\}$
$= \min\{6, 6+0\}$
$= 6$

$D^1(2,3) = \min\{D^0(2,3), D^0(2,1) + D^0(1,3)\}$
$= \min\{2, 6+11\}$
$= 2$

$D^1(3,2) = \min\{D^0(3,2), D^0(3,1) + D^0(1,2)\}$
$= \min\{\infty, 3+4\}$
$= 7$

Hence,

$$D^1 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 11 \\ 2 & 6 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{array}$$

Now,

$$D^2 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & - \\ 2 & 6 & 0 & 2 \\ 3 & - & 7 & 0 \end{array}$$

$D^2(1,3) = \min\{D^1(1,3), D^1(1,2) + D^1(2,3)\}$
$= \min\{11, 4+2\}$
$= 6$

$D^2(3,1) = \min\{D^1(3,1), D^1(3,2) + D^1(2,1)\}$
$= \min\{3, 7+6\}$
$= 3$

Hence,

$$D^2 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 6 \\ 2 & 6 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{array}$$

Now, for D^3,

$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

$$D^3(1,2) = \min\{D^2(1,2), D^2(1,3) + D^2(3,2)\}$$
$$= \min\{4, 6+7\}$$
$$= 4$$

$$D^3(2,1) = \min\{D^2(2,1), D^2(2,3) + D^2(3,1)\}$$
$$= \min\{6, 2+3\}$$
$$= 5$$

Hence,

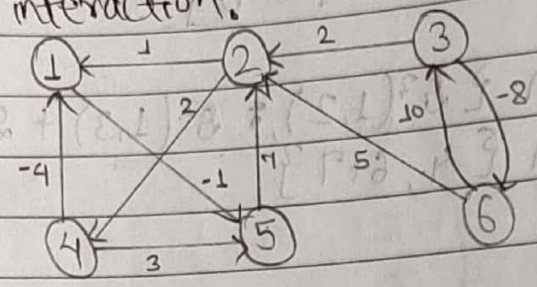
$$D_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

Analysis:-

Since, there are 3 nested loops so the running time of above algorithm is $O(n^3)$ where n is the number of vertices in the graph.

Example:

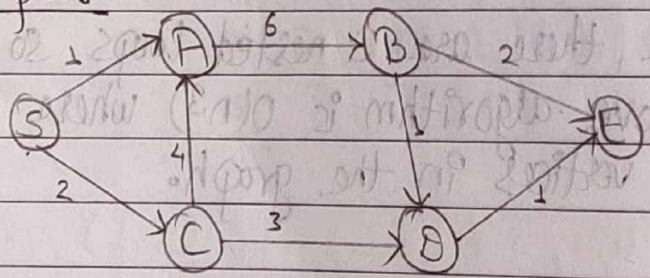
Apply Floyd-Warshall algorithm for constructing shortest path. Show the matrix D^k that results each interaction.



Directed Acyclic Graph (DAG):-

DAG is a directed graph G=(V,E) without a cycle. DAG can be used to find the shortest path from a given source node to all other nodes. To find shortest path by using DAG, first of all sort the vertices of graph topologically and then relax the vertices in topological order.

Example:



sol:

Step 1

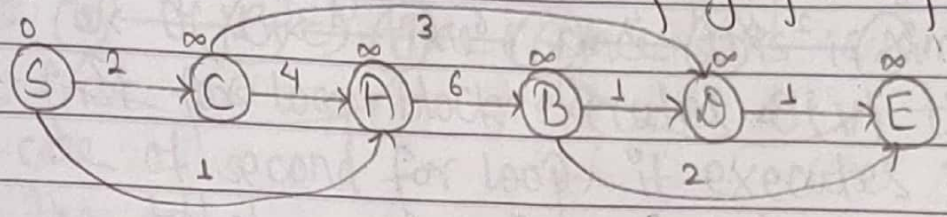
Step 2

Step 3

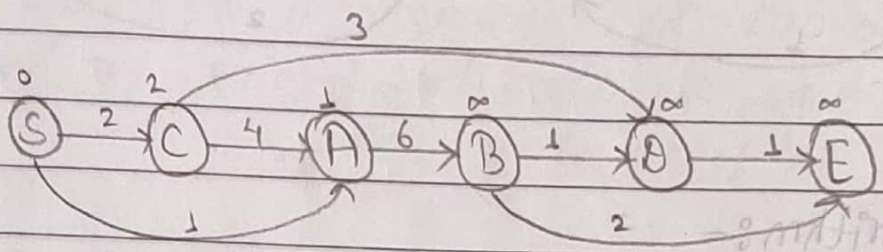
Step

Sol:

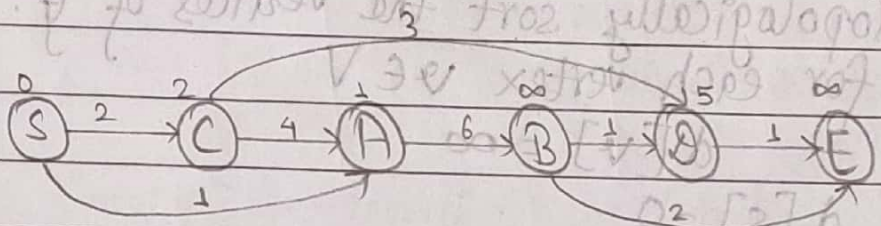
Step 1: Sort the vertices of graph topologically.



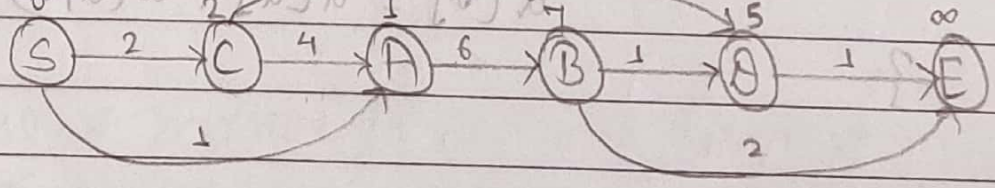
Step 2: Relax from S.



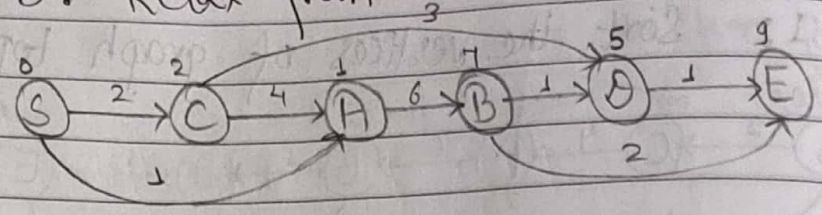
Step 3: Relax from C



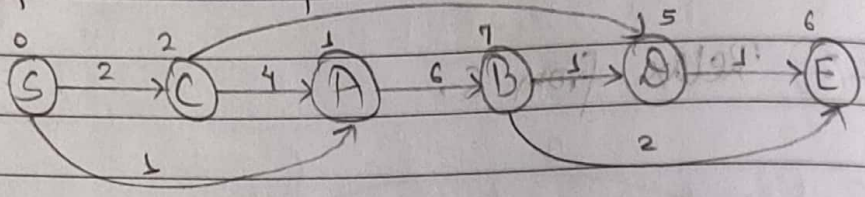
Step 4: Relax from A



Step 5: Relax from B



Step 6: Relax from D



Algorithm:-

```
DagSP(G, w, s)
{
  Topologically sort the vertices of G.
  for each vertex v ∈ V
      d[v] = ∞
  d[s] = 0
  for each vertex u taken in topologically sorted
    for each vertex v adjacent u
      if (d[v] > d[u] + w(u,v))
          d[v] = d[u] + w(u,v)
}
```

Analysis:

In above algorithm the topological sort can be done in $O(V+E)$ time (since this is similar to DFS). The first for loop block executes $O(V)$ time. In case of second for loop it executes in $O(V^2)$ time. So, the total running time is $O(V^2)$.

Aggregate analysis gives us the running time $O(V+E)$.

3.2. Geometric Algorithm:

Computational Geometry:-

The field of computational geometry deals with the study of geometric problems. The geometric problems may be decomposition of polygon into triangles, finding the convex hull, determining the intersection of the line segment and so on. The computational geometry deals with the study of algorithms to solve the problem stated in terms of geometry.

Application Domain of Computational Geometry:

The application domain of computational geometry includes:

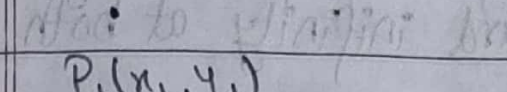
i) Computer Graphics
ii) Robotics
iii) GIS (Geographical Information System)
iv) Pattern Recognition System

Some Definition:

Point:-

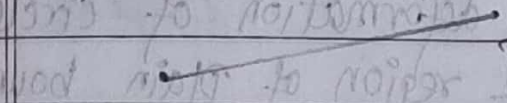
A point is a pair of numbers in 2D and triplet in 3D, specially numbers are integers. For example:- $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ are two

points as shown below.



Line segment:-

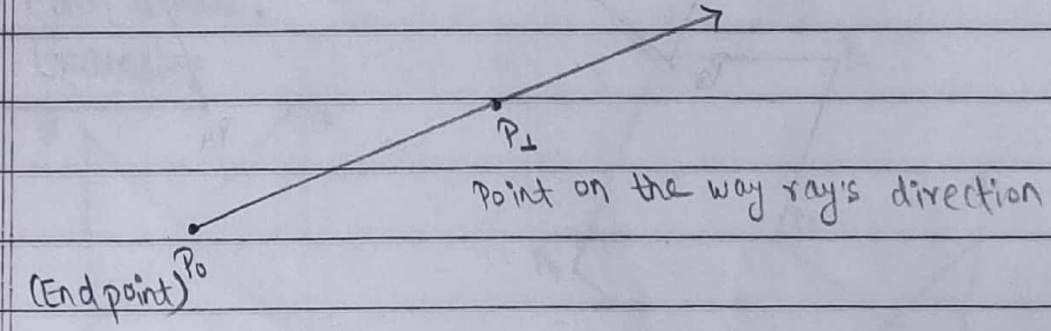
A line segment is the pair of points where each point represent the end point of line segment For example:



Here, $S(P_1, P_2)$ is a line segment.

Ray:-

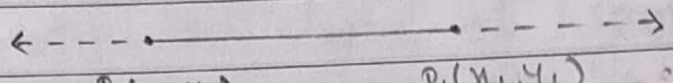
A ray is an infinite one-dimensional subset of a line determined by two points says P_0 and P_1 where one point is denoted as the end point. Thus, a ray consists of a bounded point and is extended to infinitely along a line segment.



Date
Page

Line:-

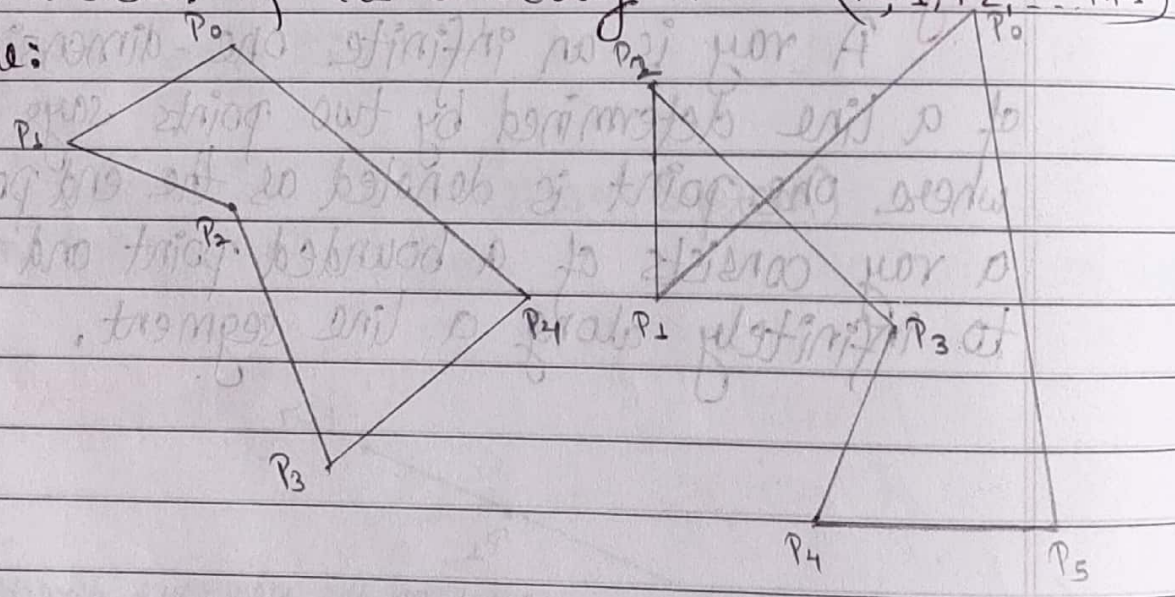
Line is represented by pair of points say P_0 and P_1 which extends toward infinity at both direction.



Polygon:-

A polygon is a homeomorphic image of a circle i.e. it is a certain deformation of circle. It is also defined as the region of plain bounded by a finite collection of line segment, forming a simple closed curve. The polygon P is represented by its vertices usually in counter clockwise order of traversal of its boundary i.e. $P = (P_0, P_1, P_2, \ldots P_{n-1})$

Example:



RS

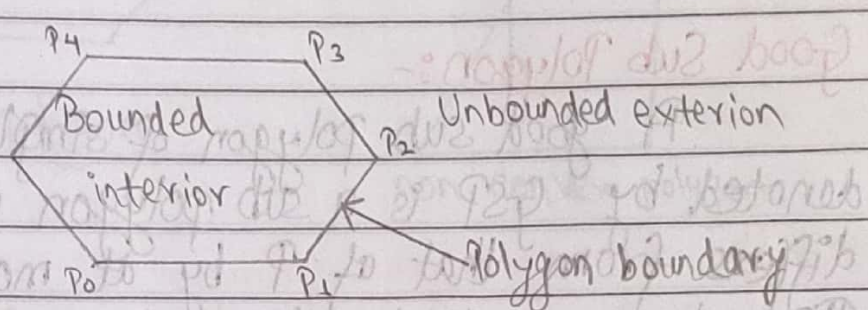
Date
Page

Simple polygon:-

A simple polygon is a polygon with no two non-consecutive edges intersecting. Every simple polygon partition the plain into 3 connected component. i.e.

i) Unbounded connected exterior
ii) Bounded connected interior
iii) Polygon boundary itself.

Example:

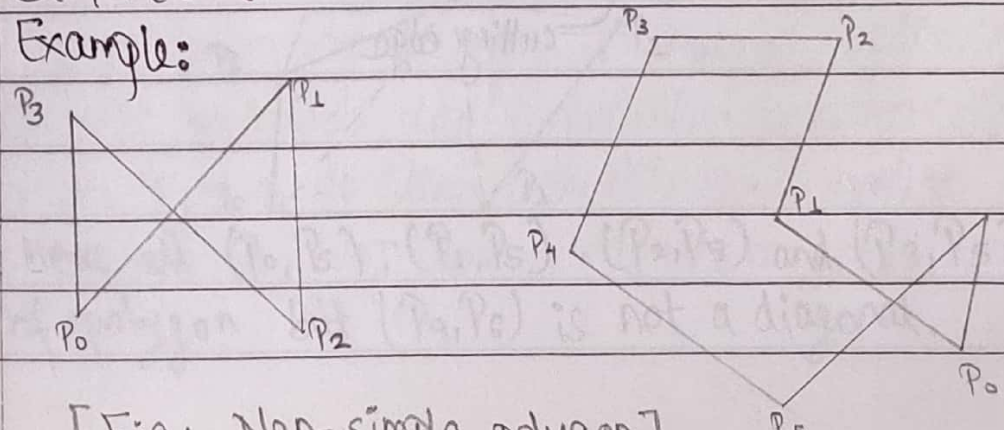


[Fig:- Simple polygon]

Non-simple polygon:-

A polygon is non-simple if there is no single interior region i.e. non-adjacent edges intersect each other.

Example:

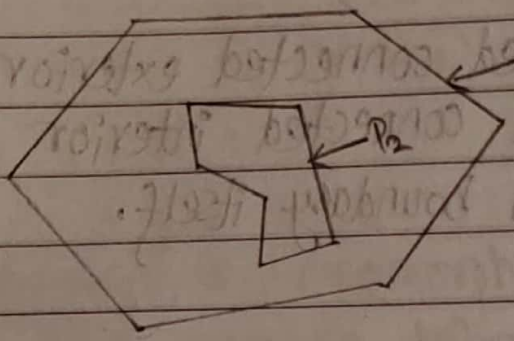


[Fig:- Non-simple polygon]

RS

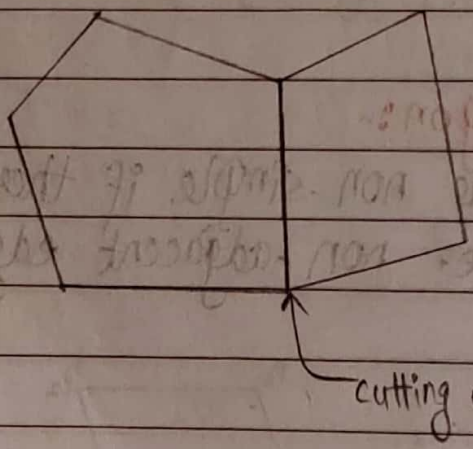
Polygon with hole:-

A simple polygon with another simple polygon in its interior is called a simple polygon with hole.



Good Sub Polygon:-

A good sub polygon of simple polygon P denoted by GSP is a sub polygon whose boundary differs from that of P by at most one edge. This edge is called cutting edge.

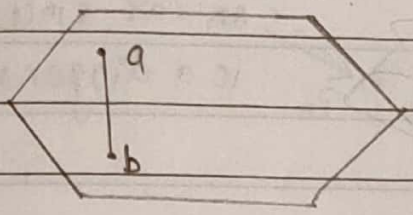


RS

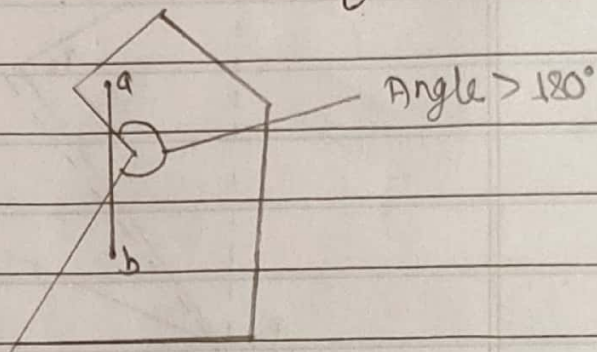
Date
Page

Convex Polygon:-

A simple polygon is said to be convex if and only if any line segment connecting two interior points of the polygon lies completely inside the polygon otherwise it is non-convex or concave polygon. In the convex polygon, all the interior polygo angles are less than or equal to 180°



Convex polygon

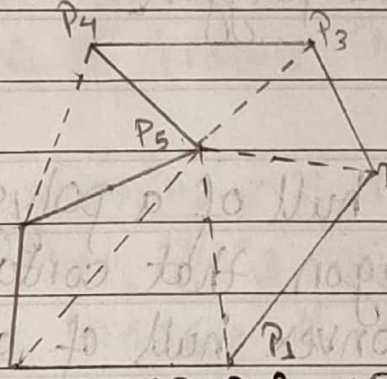


non-convex polygon

Diagonal of a simple polygon:-

A diagonal of a simple polygon is a line segment connecting two non-adjacent vertices and lies completely inside the polygon

Example:

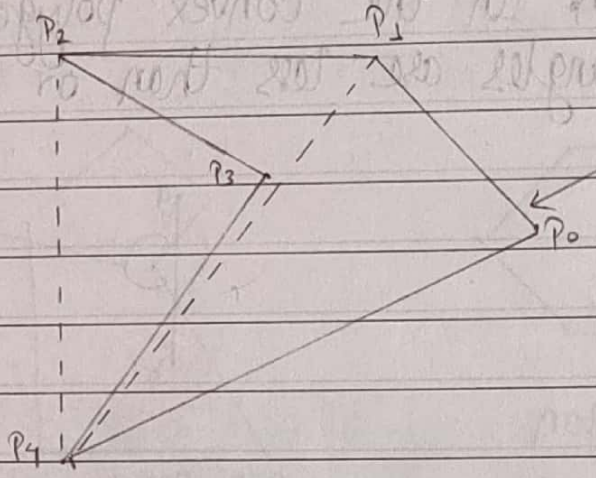


Here all (P_0, P_5), (P_1, P_5), (P_2, P_5) and (P_3, P_5) are diagonals of polygon but (P_4, P_6) is not a diagonal.

RS

Ear of Simple Polygon:-

Three consecutive vertices P_i, P_{i+1}, P_{i+2} of a polygon form the ear of polygon if (P_i, P_{i+2}) is a diagonal. If P_i, P_{i+1}, P_{i+2} is a ear of polygon then P_{i+1} is the tip of ear.



Mouth of a simple polygon:-

Three consecutive vertices P_i, P_{i+1}, P_{i+2} of a polygon form a mouth of the polygon if (P_i, P_{i+2}) represents the line segment lies completely outside of the polygon. In the above figure (P_2, P_3, P_4) is the mouth of the polygon.

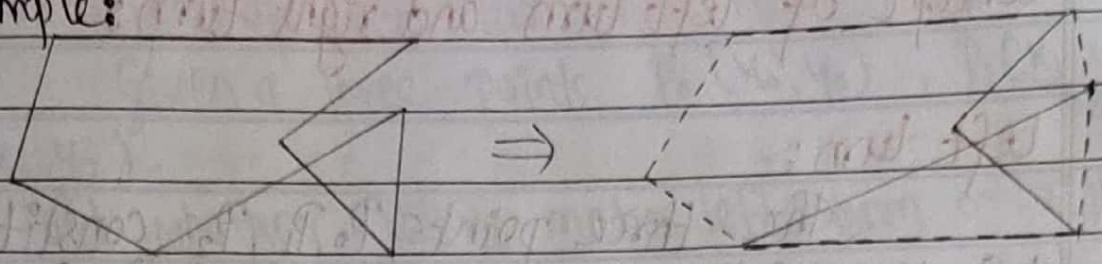
2070(2)
2071(2)

Convex Hull:-

The convex hull of a polygon P is the smallest convex polygon that contains P. Similarly, we can define the convex hull of set of points R as the smallest convex polygon containing R.

RS

Example:



Applications of Convex Hull :-

There are wide range of application areas where it comes to use of convex hull such as:

i) In pattern recognition: an unknown shape may be represented by its convex hull which is then matched to database of known shape.

ii) In motion planning: if robot is approximate by its convex hull then it is easier to plan collision free path on the landscape of obstacles.

RS

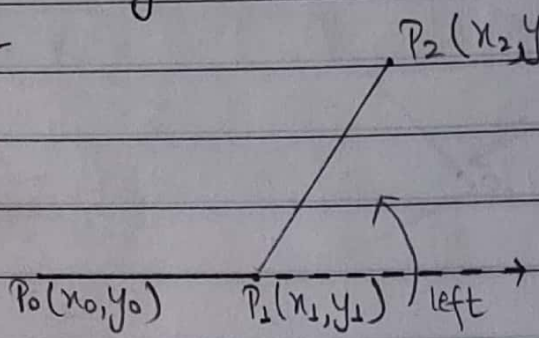
2068(2)

Concept of left turn and right turn:

Left turn:-

The three points P_0, P_1, P_2 constitute the left turn if the line segment (P_1, P_2) lies to the left of line segment (P_0, P_1).

Example:-

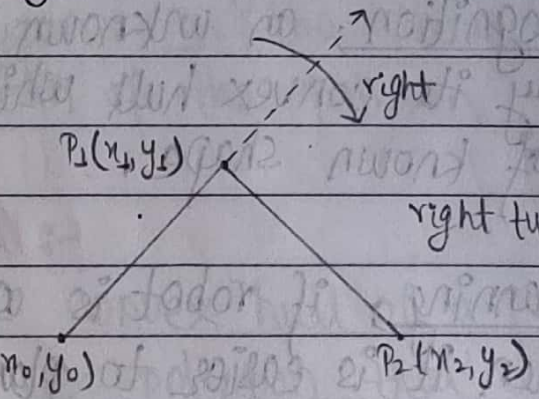


left turn $P_0 P_1 P_2$

Right turn:-

Similarly, P_0, P_1, P_2 constitute the right turn if the line segment (P_1, P_2) lies to the right of the line segment (P_0, P_1).

Example:-



RS

Computation of left turn and right turn:

- Given a three points $P_0(x_0, y_0)$, $P_1(x_1, y_1)$ $P_2(x_2, y_2)$.
- To find whether $P_0P_1P_2$ make left turn or right turn we check whether the vector P_0P_1 in clockwise or counter clockwise with respect to vector P_0P_2.
- For this we calculate a cross product of $(P_1 - P_0) \times (P_2 - P_0)$ as:

$$(P_1 - P_0) \times (P_2 - P_0) = \begin{vmatrix} x_1 - x_0 & y_1 - y_0 \\ x_2 - x_0 & y_2 - y_0 \end{vmatrix}$$

$$= (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

This can be represented as:

$$(P_1 - P_0) \times (P_2 - P_0) = \begin{vmatrix} 1 & 1 & 1 \\ x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \end{vmatrix}$$

But we know geometrical interpretation of cross product gives area of parallelogram having vector as adjacent sides:

Thus area of triangle $\Delta P_0P_1P_2$ is:

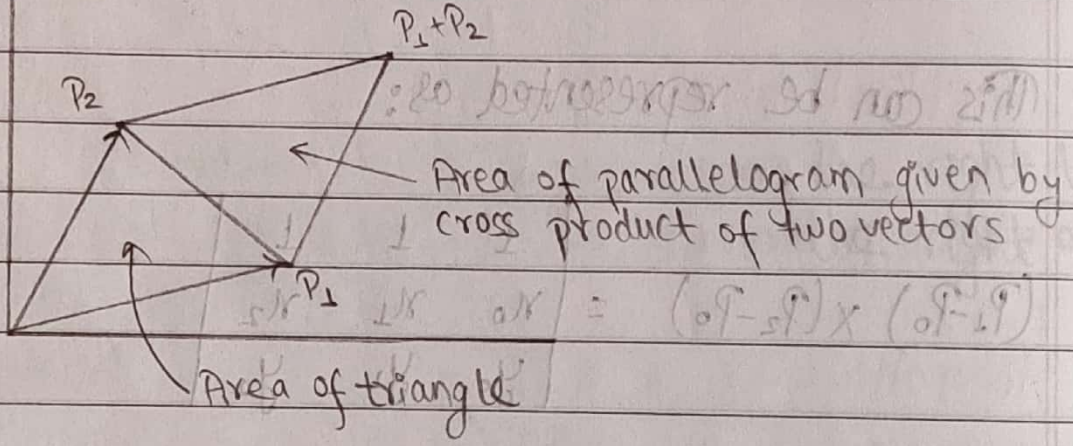
$$\Delta P_0P_1P_2 = 1/2 \begin{vmatrix} 1 & 1 & 1 \\ x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \end{vmatrix}$$

RS

$$= \frac{1}{2}(x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

- If $\Delta P_0P_1P_2 = 0$ then $P_0P_1P_2$ are collinear.
- If $\Delta P_0P_1P_2 > 0$ then $P_0P_1P_2$ makes left turn (i.e. P_0P_1 is clockwise with respect to P_0P_2).
- If $\Delta P_0P_1P_2 < 0$ then $P_0P_1P_2$ makes right turn (i.e. P_0P_1 is counter clockwise with respect to P_0P_2)

For clearity, the geometric interpretation is:



2068

Detecting the intersection of two line segments:-

There are two approach for detecting the intersection of two line segments:

First Approach:-

- The first approach uses the coordinate geometry method for detection of intersection between two

RS

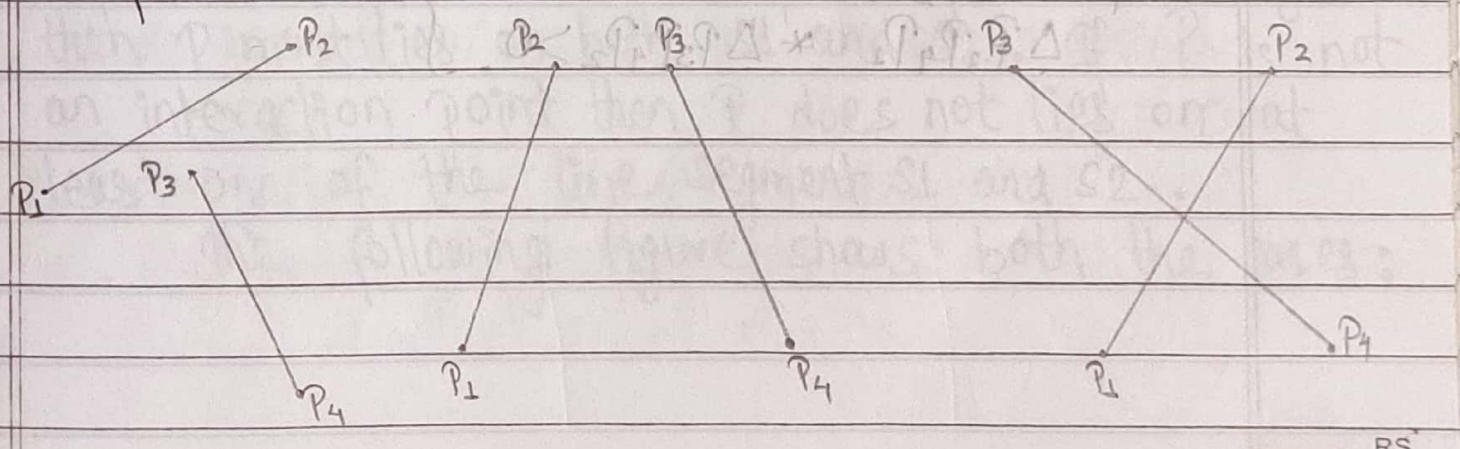
line segment.

- The steps are as follows:
- Given a two line segments l_1 and l_2, find the equation of the line through l_1 and l_2. Let the equations are L_1 and L_2.
- Solve the equation L_1 and L_2 to find the point of intersection of segment l_1 and l_2.
- Let $P(x,y)$ be the point then check the point of interaction (x,y) on both segment l_1 and l_2.
- If (x,y) lies on both l_1 and l_2 then the segment l_1 and l_2 intersect otherwise not.

This process requires much more computation to find the equation and solve.

Second Approach (using the idea of left turn & right turn)

- The method of left turn and right turn is an efficient method for detecting the segment intersection.
- This method does not have any division operation, so fast to detect the intersection.



RS

- Given a two segments $l_1(P_1, P_2)$ and $l_2(P_3, P_4)$ the step of this method are:
  - *. find the turn $P_1P_2P_3$, $P_1P_2P_4$
  - *. find the turn $P_3P_4P_1$, $P_3P_4P_2$

  if ($P_1P_2P_3$ and $P_1P_2P_4$ are opposite turn) and ($P_3P_4P_1$ and $P_3P_4P_2$ are opposite turn) then,

  $l_1(P_1, P_2)$ and $l_2(P_3, P_4)$ intersect

  otherwise

  They do not intersect.

In terms of Area of triangle:

$l_1(P_1, P_2)$ and $l_2(P_3, P_4)$ intersect

if $\Delta P_1P_2P_3 * \Delta P_1P_2P_4 < 0$ and $\Delta P_3P_4P_1 * \Delta P_3P_4P_2 < 0$ } opposite turn

$l_1(P_1, P_2)$ and $l_2(P_3, P_4)$ do not intersect

if $\Delta P_1P_2P_3 * \Delta P_1P_2P_4 > 0$, $\Delta P_3P_4P_1 * \Delta P_3P_4P_2 > 0$ } same turn.

Date
Page

Computing the point of intersection between two line segment:

- We can apply the coordinate geometry method for finding the point of intersection between two line segments.
- Let S_1 and S_2 be any two line segments. The following steps are used to calculate point of intersection between two line segments.

i) Determine the equation of line through the line segment S_1 and S_2. Let equations are $L_1 = (m_1x + c_1)$ and $L_2 = (m_2x + c_2)$ respectively, where $m_1 = (y_2 - y_1)/(x_2 - x_1)$, (x_1, y_1) and (x_2, y_2) are two end point of line segment S_1. Similarly we can find m_2 for L_2 also.

ii) Solve the equations of line L_1 and L_2. Let the value obtained by solving be $P = (x, y)$.

iii) If P is the interaction point of two line segment then P must lies on both S_1 and S_2. If P is not an interaction point then P does not lies on at least one of the line segment S_1 and S_2.

The following figure shows both the cases:

RS

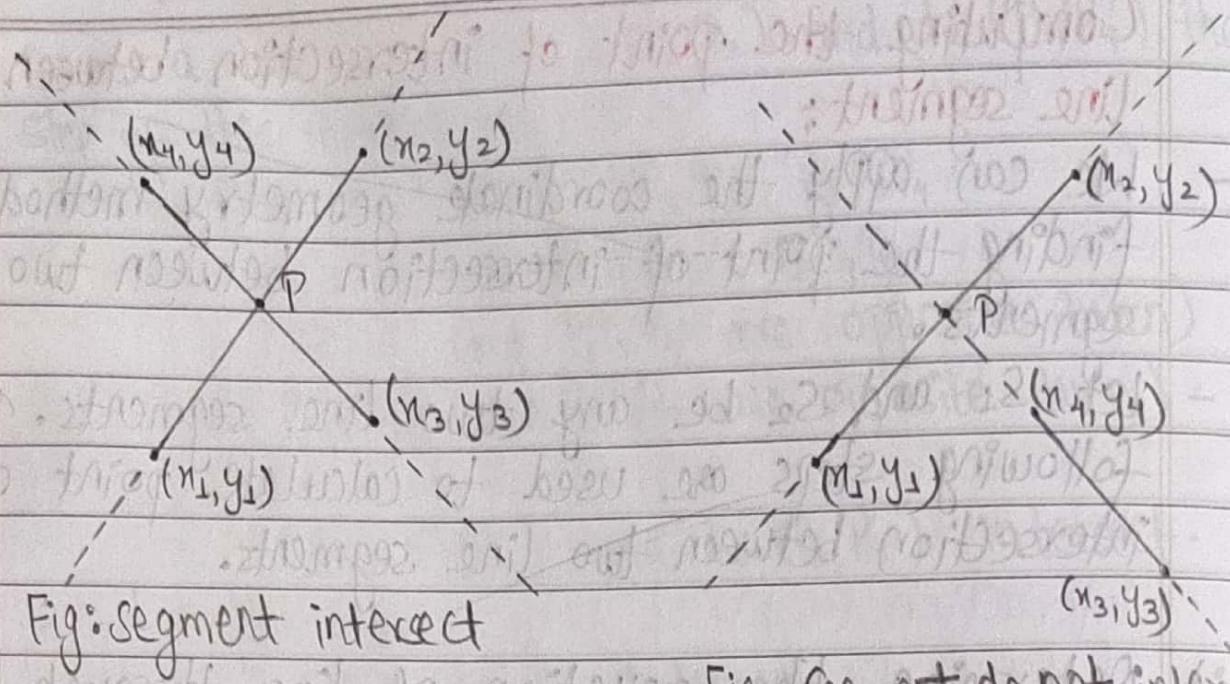


Fig: segment intersect

Fig: Segment do not intersect

Convex Hull Computation problem:

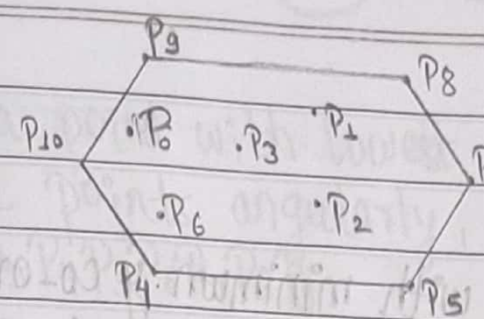
- As we know, the convex hull of point set Q is a convex polygon.
- A natural way to represent a polygon is by listing its vertices in counter-clockwise order, starting with an arbitrary one.
- So convex hull computation problem is given a set $Q = \{P_0, P_1, \ldots, P_{n-1}\}$ of points in the plane, compute a list that contains those points from Q that are vertices of convex hull listed in counter clockwise order.

i.e. input: set of points $\{P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}\}$

output: representation of convex hull.

$\{P_4, P_5, P_7, P_8, P_9, P_{10}\}$.

RS



Graham's scan algorithm for convex hull:-

- The Graham's scan algorithm solves the convex hull problem by maintaing a stack 'S' of candidate points.
- It pushes each point of input set Q onto the stack once and it eventually pops from the stack each point that is not the vertex of convex hull.
- When algorithm terminates, stack S contains exactly the vertices of convex hull, in counter clockwise order of their appearance on the boundary.
- The algorithm start by picking a point P_0 as the point with lowest y co-ordinate (picking the left most point in the case of tie).
- The sorts the remaining point of input set Q by polar angle relative to P_0.

Algorithm:

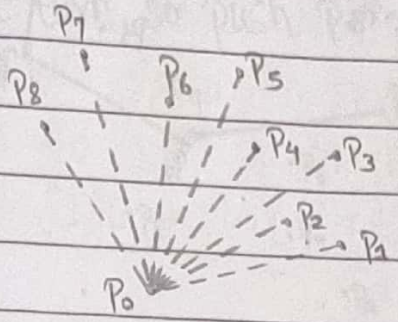
```
Graham_scan(Q)
{
  - P0 = Point in Q with minimum y co-ordinate value.
  - Angularly sort the other point with respect to P0
    Let (P1, P2, ... Pm) be sorted order.
  - Let S be empty stack.
    Push (P0, S)
    Push (P1, S)
    Push (P2, S)
    for (i=3; i<=m; i++)
    {
      while (next to top(S), top(S), Pi makes
              non-left turn)
        pop(S)
      Push (Pi, S)
    }
    return S;
}
```

Example:

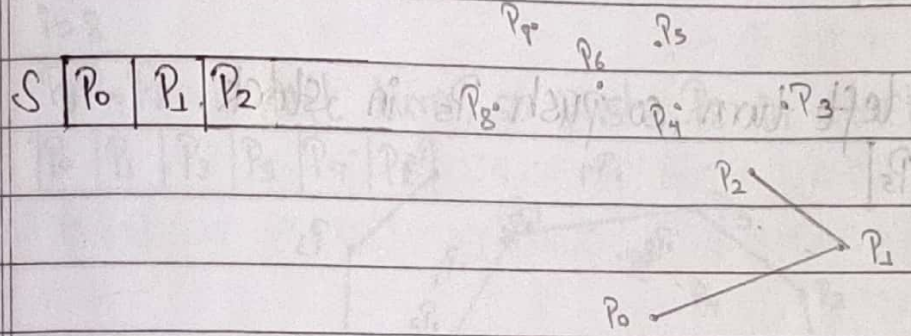
Find the convex hull of the set of points given below using Graham's scan algorithm.

Soln:

Select a point with lowest y coordinate (i.e. P_0) and sort the points angularly, let sorted order is $P_0P_1P_2P_3P_4P_5P_6P_7P_8P_9$

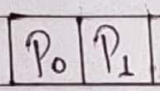


⇒ Push $P_0P_1P_2$ on the stack 'S'.

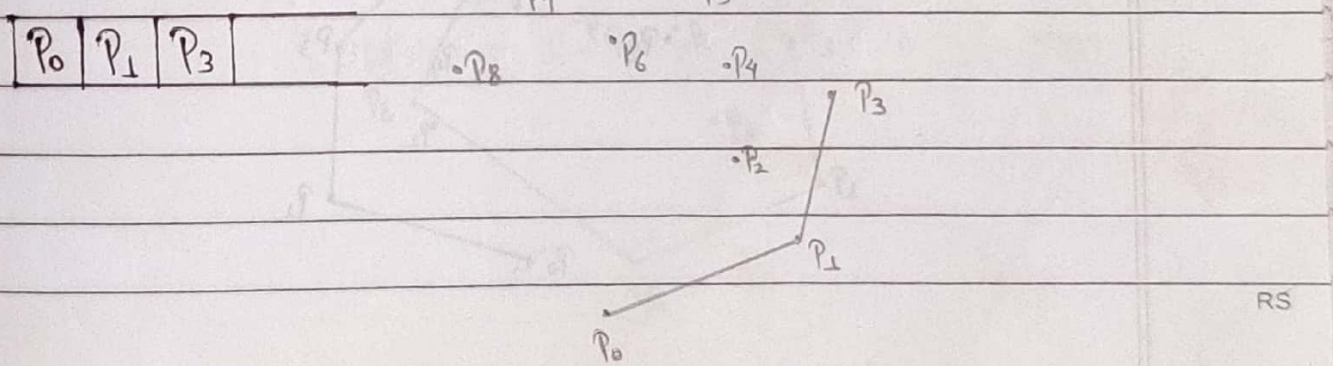


i=3

$P_1P_2P_3$ is right turn, so pop(S) i.e. pop P_2.



$P_0P_1P_3$ is left turn, so push P_3 in stack.



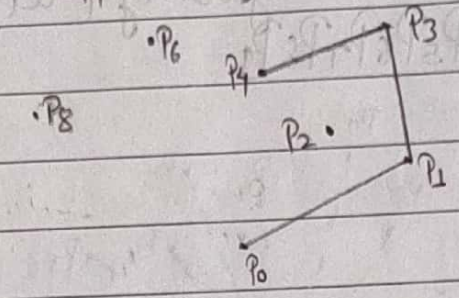
RS

Date
Page

i=4

$P_1P_3P_4$ is left turn, so push P_4 in stack.

P_0	P_1	P_3	P_4					



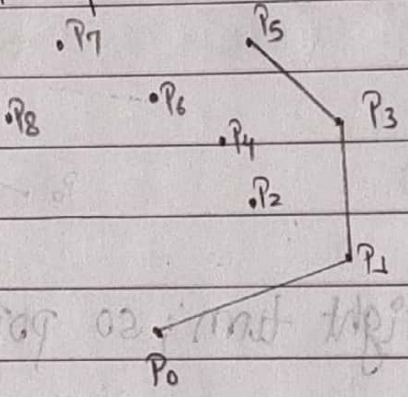
i=5

$P_3P_4P_5$ is right turn, so pop(s) i.e. pop P_4.

P_0	P_1	P_3						

$P_1P_3P_5$ is left turn, so push P_5 in stack.

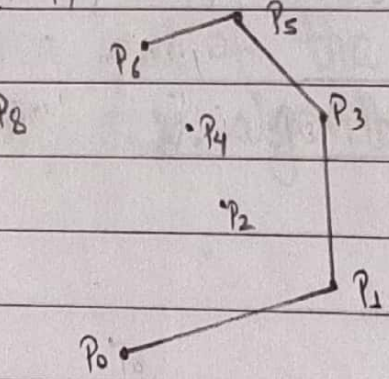
P_0	P_1	P_3	P_5					



i=6

$P_3P_5P_6$ is left turn, so push P_6 in stack.

P_0	P_1	P_3	P_5	P_6				



RS

Date
Page

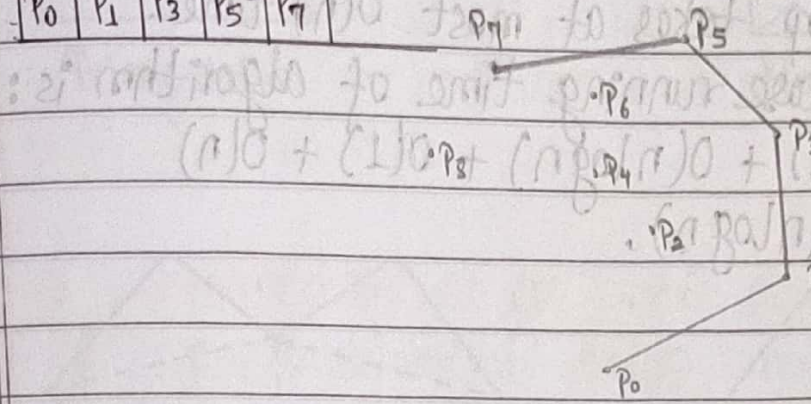
i=7

$P_5P_6P_7$ is right turn, so pop(S) i.e. Pop P_6.

P_0	P_1	P_3	P_5

$P_3P_5P_7$ is left turn, so push P_7 in stack.

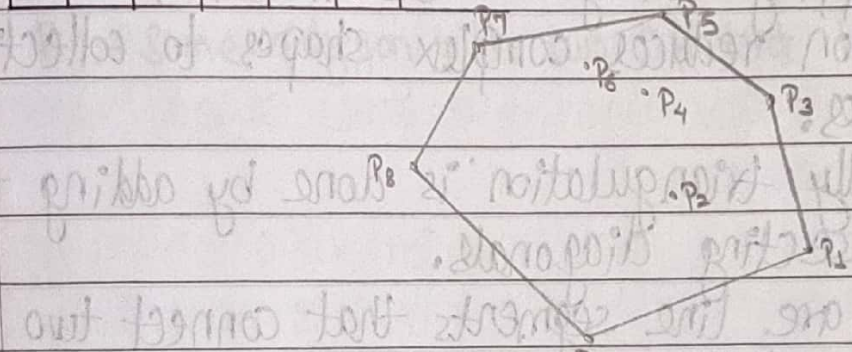
P_0	P_1	P_3	P_5	P_7



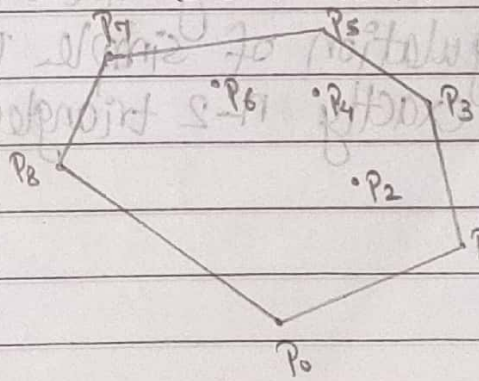
i=8

$P_5P_7P_8$ is left turn, so push P_8 in stack.

P_0	P_1	P_3	P_5	P_7	P_8



Hence, final convex hull is: $P_0P_1P_3P_5P_7P_8$.



RS

Analysis:

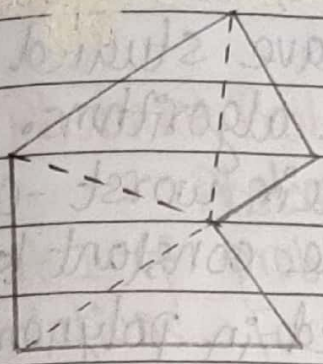
- Let $n = |Q|$
- Finding minimum y-coordinate point takes $O(n)$ time.
- Sorting the point angularly takes $O(n \log n)$ time.
- Push and Pop operation takes constant time i.e. $O(1)$.
- The while loop takes at most $O(n)$ time.
- Hence, worst case running time of algorithm is:

$$T(n) = O(n) + O(n \log n) + O(1) + O(n)$$
$$= O(n \log n).$$

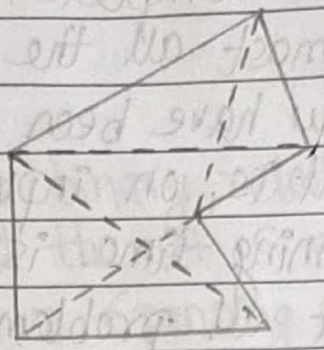
Polygon Triangulation:-

- Triangulation of a polygon P is its partition into non-overlapping triangle whose union is P.
- Triangulation reduces complex shapes to collection of simpler ones.
- Geometrically triangulation is done by adding the non-intersecting diagonals.
- Diagonals are line segments that connect two vertices of polygon and lies interior of P.
- Triangulation are usually not unique.
- Any triangulation of simple polygon with n-vertices consists of exactly $n-2$ triangles.

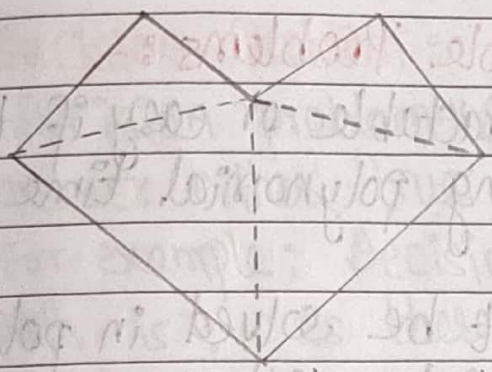
RS



Triangulation



Non-triangulation



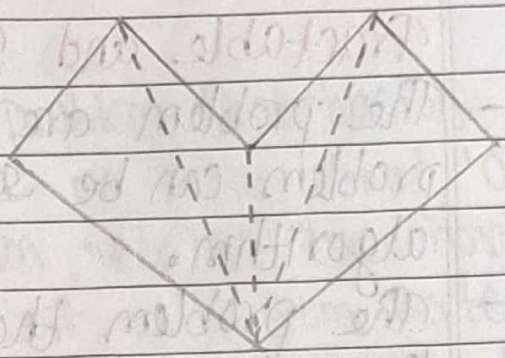


Fig:- Same polygon with two different triangulation

RS

3.3) NP Completeness

- Almost all the algorithm we have studied upto now have been polynomial-time algorithms.
- That is, on input of size n, their worst-case running time is $O(n^k)$ for some constant k.
- But all problems can't be solved in polynomial time such as Halting problem.

Tractable and Intractable Problems :-

- The problem are called tractable or easy if the problem can be solved using polynomial time algorithm.
- The problem that cannot be solved in polynomial time but requires super polynomial time or exponential time algorithm are called interactable or hard problems.
- There are many problems for which no algorithm with running time better than exponential time is known, some of them are travelling salesman problem, hamilton cycle and circuit satisfiability problem.

RS

Problems:-

a) Abstract Problems:-

Abstract problem A is binary relation on set I of problem instances, and the set S of problem solutions. For example; Minimum Spanning tree of a graph G can be viewed as a pair of the given graph G and MST graph T.

b) Decision Problem:-

- Decision problem 'D' is a problem that has an answer either 'true', 'yes', '1' or 'false', 'no', '0'.
- For example; Decision Problem related to shortest path is: "Is there a shortest path from vertex u to vertex v with at most k edges. In this situation the answer is either yes or no.

c) Optimization problem:-

- We encounter many problems where there are many feasible solution and our aim is to find the feasible solution with the best value. This kind of problem is called optimization problem.
- Optimization problem requires some value to be maximized or minimized.
- For example; given a graph G and the vertices u and v find the shortest path from u to v with minimum number of edges.

RS

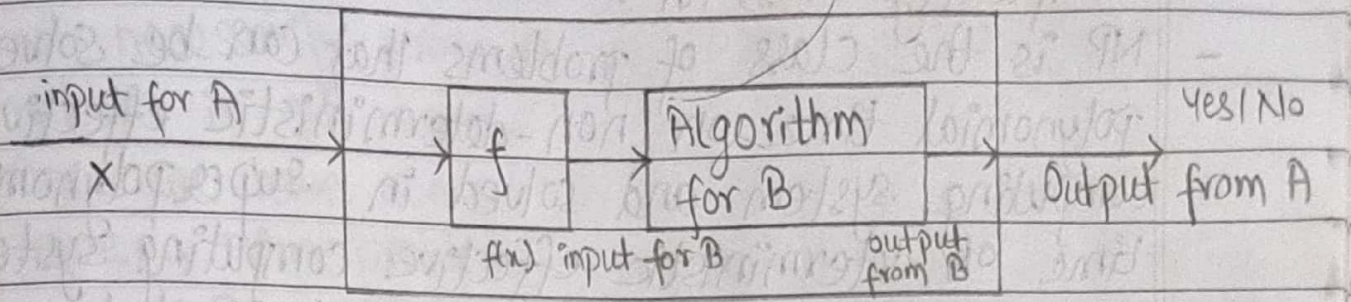
Date
Page

d) Encoding :-

- Encoding of a set S is a function f from S to the set of binary strings.
- With the help of encoding, we define concrete problem as a problem with problem instances as a set of binary strings.

Polynomial Time Reduction :-

- Given two problems A and B, a polynomial time reduction from A to B is a polynomial time function f that transforms the instances of A into instances of B such that the output of algorithm for problem A on input instance x must be same as the output of algorithm for problem B on input instance of f(x) as shown in figure below
- If there is a polynomial time computable function f such that it is possible to reduce A to B then it is denoted by $A \leq_p B$ i.e. problem A is polynomial time reducible to B.
- We call such function 'f' is reduction function and algorithm that computes 'f' is called reduction algorithm.



[Fig:- Reduction function]

2067
2068
2069
2070

Complexity Classes P, NP and NP complete :-

Complexity Class P:-

- The class P consists of those problems that are solvable in polynomial time.
- More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.
- In another word, P is the class of problem that can be solved in polynomial time on deterministic effective computing system.
- All computing machine that exists in real world are deterministic so P is the class of problem that can be computed in polynomial time on real computer.
- Example: sorting problem, searching problem, etc.

RS

Date
Page

Complexity class NP :-

- NP is the class of problems that can be solved in polynomial time on non-deterministic effective computing system and solved in super polynomial time on deterministic effective computing system.
- All computing machine that exist in real world are deterministic, so NP is the class of problem that can be computed in super polynomial time on real computer.
- Examples of problems in class NP are: Satisfiability problem (SAT), Hamilton Cycle Problem, etc.
- Any problem in P is also in NP since if there is a polynomial time algorithm for the problem then we can get the verification algorithm.
- Thus, $P \subseteq NP$ but it is unknown whether $P = NP$.
- Most researcher believe that P and NP are not same class.

Complexity class NP-complete :-

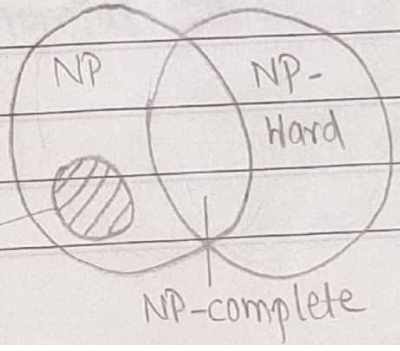
- NP complete problems are those problems that are hardest problems in class NP.
- We define some problem A is NP complete if
  i) $A \in NP$ (i.e. problem A is in the class NP)
  ii) $B \leq_P A$ for every $B \in NP$ (i.e. for any problem B in NP, there exist a polynomial time reduction of B to A).

RS

- If the problem satisfies the property (ii) but not necessarily property (i) then we say that problem is NP hard.
- Once we have some NP complete problems, we can prove a new problem to be NP-complete by reducing known NP complete problem to it using polynomial time reduction.

- Some properties of NP complete problems are:
i, No polynomial time algorithm has been found for any one of them.
ii, It is not established that polynomial time algorithm for these problem do not exists.
iii, If a polynomial time algorithm is found for one of them, then there will be polynomial time algorithm for all of them.

- The examples of NP complete problems are travelling salesman problem, circuit satisfiability problem, vertex cover problem, etc.
- The commonly believed relationship among P, NP and NP complete is as shown in figure below:



RS

Date
Page

Cook's Theorem:-

SAT is NP-complete.

Proof:-

To show SAT is NP-complete, we have to show that

i) SAT $\in$ NP
ii) SAT is NP-Hard

i) SAT $\in$ NP :-

- Circuit Satisfiability problem (SAT) is the question "given a boolean combinational circuit, is it satisfiable ?" i.e. does the circuit has assignment sequence of truth value that produces the output of the circuit as 1 ?
- Given a circuit satisfiability problem, take a circuit x and certificate y with the set of values that produce output 1, we can verify that whether given circuit satisfies the circuit in polynomial time.
- So, we can say that satisfiability problem is NP.

RS

ii) SAT is NP-Hard :-

- Take a problem $V \in NP$, let A be the algorithm that verifies V in polynomial time.
- We can program A on computer and therefore there exists logical circuit whose input wires corresponds to the bits of the input x and y of A and which outputs 1 precisely when A(x, y) returns yes.
- For any instance of x of V, let A_x be the circuit obtained from A by setting the x-input wires values according to specific string x. The construction of A_x from x is our reduction function.

2071(3)

Approximation Algorithm :-

- The algorithm that returns near-optimal solution are called approximation algorithm.
- An approximation algorithm is a way of dealing with NP completeness for optimization problem.
- This technique does not guarantee the best solution.
- The goal of approximation algorithm is to come as close as possible to the optimum value in at most polynomial time.

RS

Vertex Cover problem :-

- The vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G then either $u \in V'$ or $v \in V'$ or both.
- The problem here is to find the vertex cover of minimum size in a given graph G.
- Vertex cover problem is NP complete so we do not know how to find optimal vertex cover in graph G. in polynomial time so we can use the approximation algorithm that can effectively find the vertex cover that is near optimal.
- The following approximation algorithm takes as input an undirected graph G and returns a vertex cover which size is guaranteed to be no more than twice the size of an optimal vertex cover.

2068
2071 (6)

Algorithm:

```
Approx_vertex_cover(G)
{
    C = ϕ      // C is vertex cover of G.
    E' = E     // E' is edge list
    while (E' ≠ ϕ)
    {
        choose an arbitrary edge (u,v) of E'.
        C = C ∪ {u,v}
```

Remove from E' every edges incident on either 'u' or 'v'.

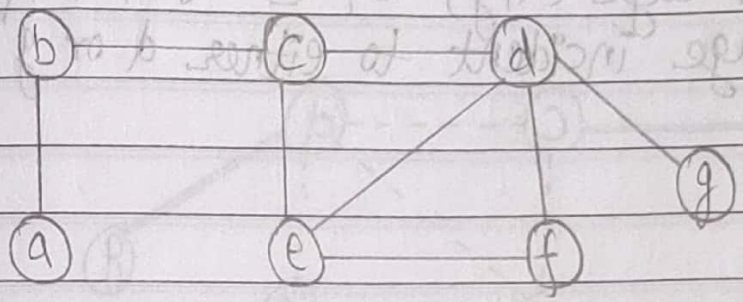
}

return C

}.

Example:-

Find the vertex cover of following graph.



Soln:

Initially,

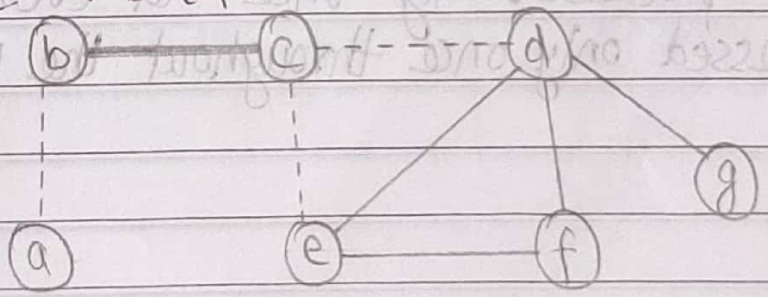
$C = \phi$

$E' = \{(a,b), (b,c), (c,e), (d,e), (c,d), (d,f), (d,g), (e,f)\}$

Now,

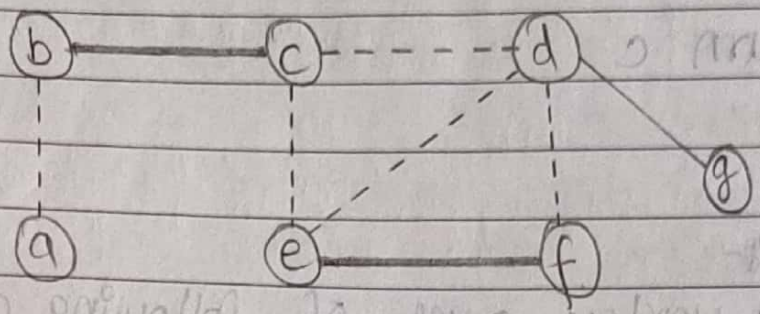
⇒ Choose an edge (b,c) so, $C = \{b,c\}$

delete edges from E' which are incident on either b or c or both.

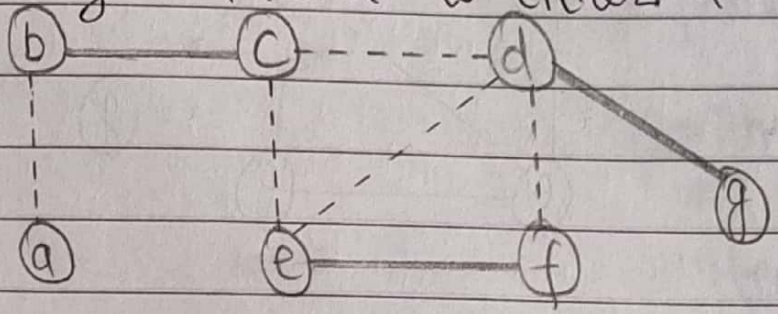


RS

⇒ Choose an edge (e,f) so, C = {b,c,e,f}
Delete an edge incident to either e or f or both.



⇒ Choose an edge (d,g) so, C = {b,c,e,f,d,g}
Delete edge incident to either d or g or both.



Hence, the vertex cover of given graph is:
C = {b,c,e,f,d,g}.

Analysis:-
This algorithm takes O(V+E) time since each edge is processed only once and every vertex is processed only once throughout the whole operation.

RS