ARUNAI ENGINEERING COLLEGE

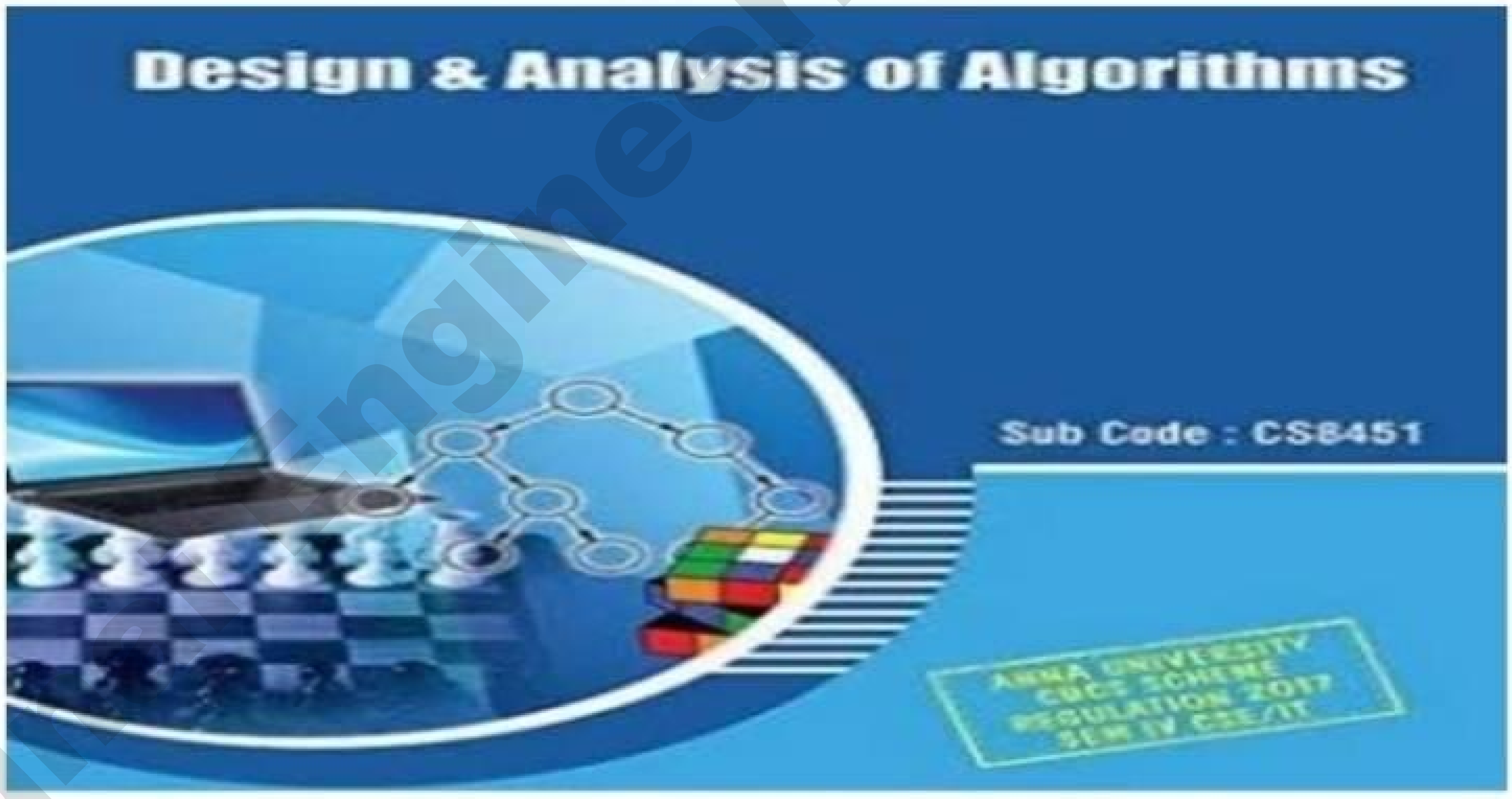
(Affiliated to Anna University)
Velu Nagar, Tiruvannamalai —606603
Phone: 04175-237419/236799/237739
www.arunai.org

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BACHELOR OF ENGINEERING

Second Year

Fourth Semester



CS8451-Design & Analysis Of Algorithm

Lecture By – Mrs. Karthika .D , AP/CSE

CS8451 DESIGN AND ANALYSIS OF ALGORITHMS L T P C

3 0 0 3

UNIT I INTRODUCTION 9

Notion of an Algorithm – Fundamentals of Algorithmic Problem Solving – Important Problem Types – Fundamentals of the Analysis of Algorithmic Efficiency –Asymptotic Notations and their properties. Analysis Framework – Empirical analysis – Mathematical analysis for Recursive and Non-recursive algorithms – Visualization

UNIT II BRUTE FORCE AND DIVIDE-AND-CONQUER 9

Brute Force – Computing an – String Matching – Closest-Pair and Convex-Hull Problems – Exhaustive Search – Travelling Salesman Problem – Knapsack Problem – Assignment problem. Divide and Conquer Methodology – Binary Search – Merge sort – Quick sort – Heap Sort – Multiplication of Large Integers – Closest-Pair and Convex – Hull Problems.

UNIT III DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE 9

Dynamic programming – Principle of optimality – Coin changing problem, Computing a Binomial Coefficient – Floyd‘s algorithm – Multi stage graph – Optimal Binary Search Trees – Knapsack Problem and Memory functions. Greedy Technique – Container loading problem – Prim‘s algorithm and Kruskal's Algorithm – 0/1 Knapsack problem, Optimal Merge pattern – Huffman Trees.

UNIT IV ITERATIVE IMPROVEMENT 9

The Simplex Method – The Maximum-Flow Problem – Maximum Matching in Bipartite Graphs, Stable marriage Problem.

UNIT V COPING WITH THE LIMITATIONS OF ALGORITHM POWER 9

Lower – Bound Arguments – P, NP NP- Complete and NP Hard Problems. Backtracking – n-Queen problem – Hamiltonian Circuit Problem – Subset Sum Problem. Branch and Bound – LIFO Search and FIFO search – Assignment problem – Knapsack Problem – Travelling Salesman Problem – Approximation Algorithms for NP-Hard Problems – Travelling Salesman problem – Knapsack problem.

UNIT I

UNIT-I

Introduction

Notion of an Algorithm - Fundamentals of Algorithmic problem Solving - Important problem types - Fundamentals of the Analysis of Algorithm Efficiency - Analysis framework - Asymptotic Notations and its properties - Mathematical analysis for Recursive and non-Recursive algorithms - visualization.

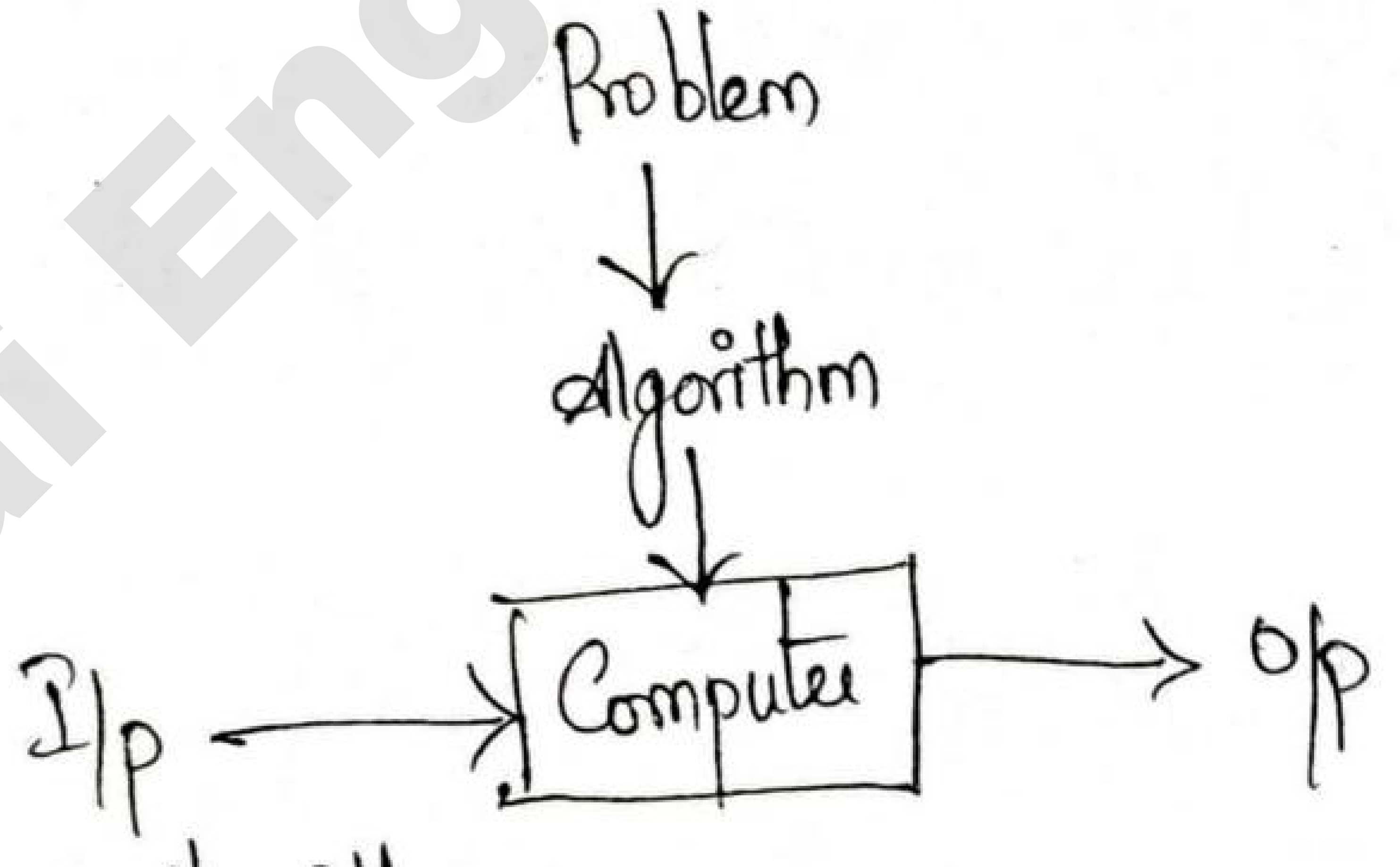
Notion of an Algorithm:

-> Algorithm is a step by step procedure to solve a problem.

-> Algorithm can be specified in a natural language or a pseudocode

-> It is a finite set of instructions that if followed, accomplishes a solution for a problem in a finite amount of time.

Algorithm + data structures -> program.



characteristics of an Algorithm

* Definiteness - Each instruction is clear & complete

* Finiteness - algorithm terminates after a finite number of steps.

* Efficiency - Every instruction must be very basic.

* Unambiguous.

Fundamentals of Algorithm problem Solving:-

→ Algorithms are procedural Solutions to problems. They are not answers but rather Specific instructions for getting answers.

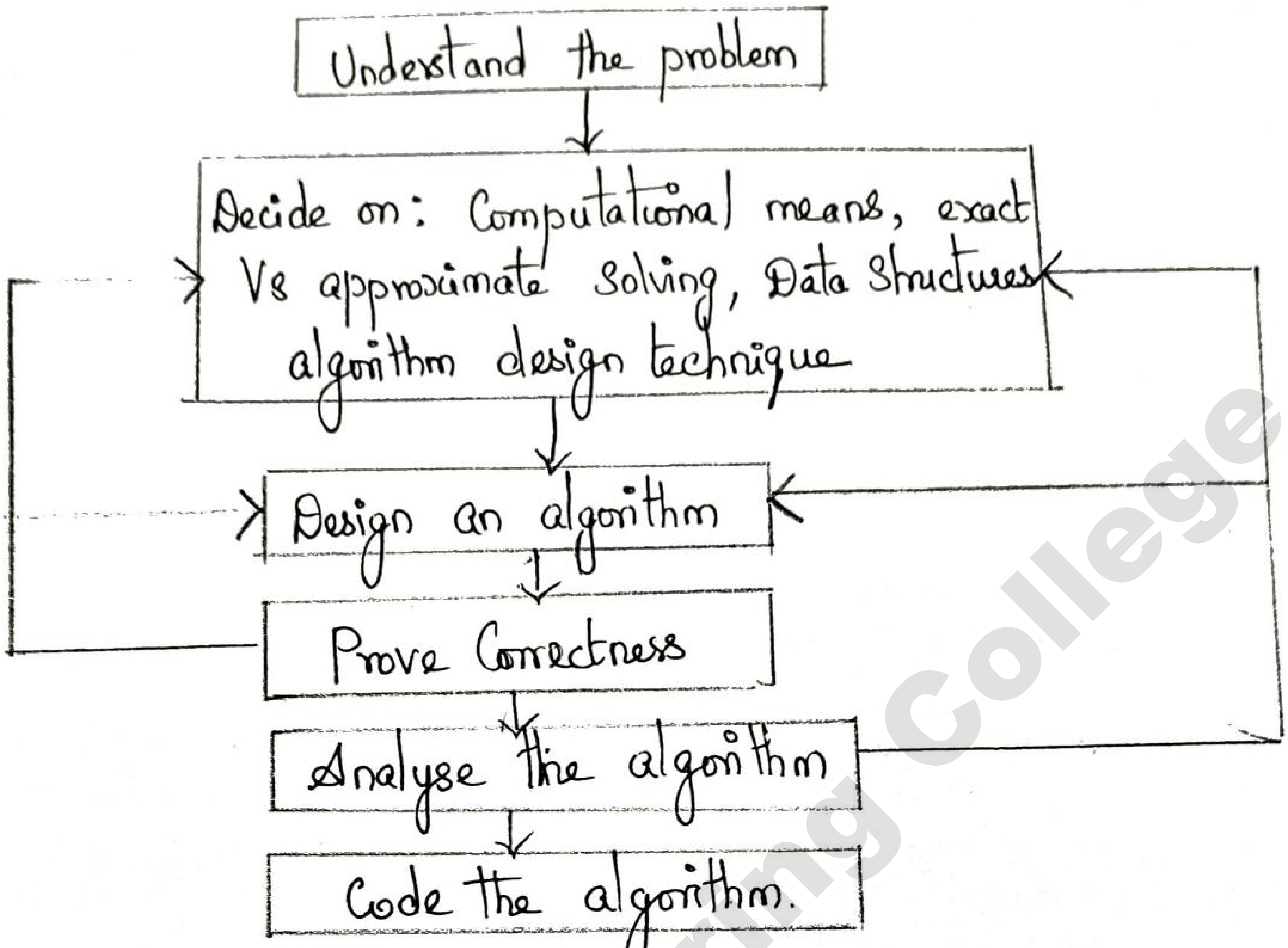
The Various steps involved in the designing & analysis of an algorithm are

1. Understanding the problem
2. Ascertaining the Capabilities of a Computational device
3. choosing between exact and appropriate problem Solving.
4. Deciding on appropriate data Structures.
5. Algorithm design techniques.
6. Methods of Specifying an algorithm.
7. Proving an algorithm's Correctness.
8. Analyzing an algorithm.
9. Coding an algorithm.

Following are the Criteria Used to analyze the algorithm:-

1. Correctness
2. Amount of Work done
3. Amount of Space Used.
4. Simplicity
5. clarity
6. optimality.

Algorithm Design & Analysis process



1. Understanding the problem:-

→ Defining the problem statement, initially we must Concentrate on what must be done rather than how to do it.

→ Determining the overall goals, but it should be in a clear manner.

→ An i/p to an algorithm Specifies the instances of the problem the algorithm solves, So we need to specify the range of instances the algorithm needs to handle.

→ Do Some examples and then think about Special Cases, if required again ask Questions?

2. Ascertaining the capability of Computational device

-> Majority of algorithm are destined to be Programmed for Computers closely resembling Von Neumann machine, which can be assumed as a generic one-processor, Randon-access-machine (RAM)

-> In the RAM model, instructions are executed one after another, one operation at a time (ie) no Concurrent or parallel operation.

-> Algorithm that take advantage of Computers that Can execute operations Concurrently or parallely are called Parallel algorithms

-> We mainly focus on RAM model Computers.

-> To measure the efficiency of algorithm, Computer resources such as memory, Communication bandwidth & speed are considered.

3. Choosing between Exact & approximate problem solving:-

-> An algorithm that Can Solve the problem exactly is called an Exact algorithm.

-> The algorithm that Can Solve the problem approximately is called approximation algorithm.

Problems that Cannot be Solved exactly

* Extracting Square roots
* Solving non-linear equations
* Evaluate definite integrals.

* Algorithm for Solving a problem Exactly is not acceptable because it can be slow due to its intrinsic complexity of that Problem.

eg,.. Travelling Salesman problem which find shortest tour through n cities.

4. Deciding on appropriate data structures:-

→ In object oriented programming, data structures remain Cruicially important for both design & analysis of algorithm.

Algorithm + datastructures = program.
/
organizing related data items.

5. Algorithm Design Techniques

→ It is a general approach to solve problems algorithmically that is applicable to a variety of problems from different areas of Computing.

Benefits

→ provides guidance for designing algorithms for new problems. (ie) problems which has no known Satisfied algorithm.

→ It is Used to classify the algorithm based on design data.

→ Serves as a natural way to both categorize & Study algorithms.

6. Methods of Specifying an Algorithm

→ Two common methods used for Specifying an Algorithm 1. Euclid's algorithm 2. Pseudo code. as well as to fill the gap between program & Algorithm.

1. Euclid's Method

- * Specified by Simple English statement
- * Step by step format.

2. Pseudocode

- * A mixture of a natural lang. and programming language Constructs
- * more precise than a natural language.
- * for Simplicity, declaration of Variables is omitted
- * for, if & while Statements are Used to Show the Scope of the Variables. // – Used for Comments ← – Used for assigned operation

3. Flowchart.

- * Method of expressing an algorithm by diagram representation.
- * very Simple.

example:

```
// input : Array A of n integers
// output : Maximum element of array A.
current Max = A[0]
    for i=1 to n-1 do
    if A[i] > current Max then
        CurrentMax = A[i]
    return CurrentMax.
```

9. Coding an algorithm

* The transition from an algorithm to the program should be done Correctly & efficiently.

* Correctness - done by proving it mathematically

* Validity - done by testing.

→ Implementation of algorithm is done as Computer Programs. It is improved by Code optimization or Code tuning technique.

→ Code tuning refers to modifying the implementation of a Specific design rather than modifying the design itself.

Important problem Types:-

- * Sorting
- * Searching
- * String Processing
- * Graph problems
- * Combinatorial Problems
- * Geometric problems
- * Numerical problems.

1) Sorting :-

→ In this, we rearrange the items of a given list in ascending order. There must be a relation of total ordering. We Usually Sort a list of numbers, Characters from an alphabet, character strings & records.

5

To Sort a list of number or records, we need a piece of information, called key to guide Sorting. The Sorted list helps in Searching.

→ few Sorting algorithms that Sort an arbitiary array of size n using $(n \log_2 n)$ Comparisons,

2 propeties of any Sorting algorithm should posses.

(i) stable (ii) In place.

Stable

→ If the relative order of any 2 equal elements in the i/p is preserved.

(i.e) if the i/p list Contains 2 equal elements in position i & j when $i<j$, then in the Sorted list they have to be in positions i & j respectively. Such that $i<j$.

eg/.. if we have a list of Students Sorted alphabetically and we want to Sort it according to Student GPA, then a stable algorithm will yield a list in which students with Same GPA will be still Sorted alphabetically.

In place:-

→ An algorithm is said to be in place if it does not require extra memory.

2) Searching:-

→ It deals with finding a given value, called a Search key in the given set.

→ In Searching algorithms, Searching has to be considered in Conjunction with addition + deletion of an item in the data set.

→ So, data structures and algorithms should be chosen to strike a balance among the requirements of each operation.

3) string matching

→ Applications dealing with non-numeric data has increased and it leads to string matching at handling algorithms.

→ It is particularly important in searching for a given word or pattern.

→ eg. In bio informatics, DNA alphabet $\Sigma = \{A, C, G, T\}$ finding pattern in DNA Sequence.

4) Graph problem :-

→ Graph is a collection of points called vertices, Connected by line Segments called edges

→ Study of graphs is called graph theory

→ Graphs are used for modelling a wide variety of real life applications, like transportation, Communication networks, project Scheduling & games.

→ Basic graph algorithm.

* Traversal algorithms * shortest path algorithm.

* Topological Sorting.

→ Data Structures used for graph.

* List structures : Adjacency List

* Matrix Structures : Adjacency matrix.

→ Most widely known graph problems

- * Travelling Salesman problem (TSP)
- * Graph coloring (Practical applications).
  - → event Scheduling
  - → Sudoku
  - → puzzle game

5) Combinatorial Problems:-

→ most difficult problems in Computing from both the theoretical & the practical standpoints.

→ Branch of pure mathematics concerning the study of discrete objects.

→ As problem size grows the Combinatorial objects grow rapidly and reach to a huge value.

eg/.. number of possible orderings of a deck of 52 distinct playing cards? $52! = 8.0658 \times 10^{67}$

→ There is no known algorithms available which can solve these problems in finite amount of time. So, they fall mostly into the category of unsolvable problems.

6) ~~Graph~~ Geometric problems:-

→ Deals with geometric objects such as points, lines & Polygons, also includes Constructing Simple shapes like triangle, Circles

Application of Computation geometry

* Computer graphics, CAD/CAM, robotics (motor planning & visibility Problems), GIS (geometrical location & Search.

classical problem of computational geometry is

* closest pair problem - If given n points, in the space, find Pair of points with the Smallest distance b/w them.

* Convex hull problem

↳ It finds the Smallest Convex polygon that would include all the points of a given Set.

7) Numerical problems:-

→ involves mathematical objects of Continuous nature.

→ deals with Solving equations, Systems of equations, Computing definite integrals, evaluating functions which can be solved only by approximation algorithm.

Fundamentals of Analysis of Algorithm Efficiency:-

* An algorithm when implemented, uses the Computers primary memory and cpu. Analysing the amount of resources (time/space) need for Solving the problem.

The algorithm's efficiency depends on the 2 resources: Running time & memory Space.

→ 2 kinds of efficiency

1. Time efficiency - indicates how fast the algorithm runs. Running time of an algorithm on a particular i/p is the number of primitive operations or steps executed

2. Space efficiency - deals with extra space the algorithm requires. , because of memory hierarchy this is of prime importance.

Measuring an i/p size

→ Time taken by an algorithm grows with the size of the input.

→ Input size is best measured as the number of items in the input, need to choose some parameter 'n' indicating the algorithm's i/p size.

eg/.. 1) it will be the array size n for problems like sorting, searching, finding largest & smallest elements etc..

2) for evaluating a polynomial $p(x) = a_n x^n + \dots + a_0$ of degree n, the input size n will either be polynomial degree or the number of its co-efficients which is larger by 1 than its degree.

3) Product of 2 n by n matrix - i/p size is either the matrix order n or the number of elements N in the matrices being multiplied

4) if the i/p to an algorithm is a graph, the i/p size can be measured by the number of vertices & edges of the graph by adjacency linked list or adjacency matrix.

5) i/p size metric is also influenced by the operations of the algorithm.

Scientist prefer measuring size by the number b of bits in the n's binary representation.

$$b = \lfloor \log_2 n \rfloor + 1.$$

Orders of growth

→ Running times on small i/p's does not really distinguishes efficient algorithms from inefficient ones. The difference in algorithmic efficiencies become clear & important, only for large values of n, because of function's order of growth.

eg.

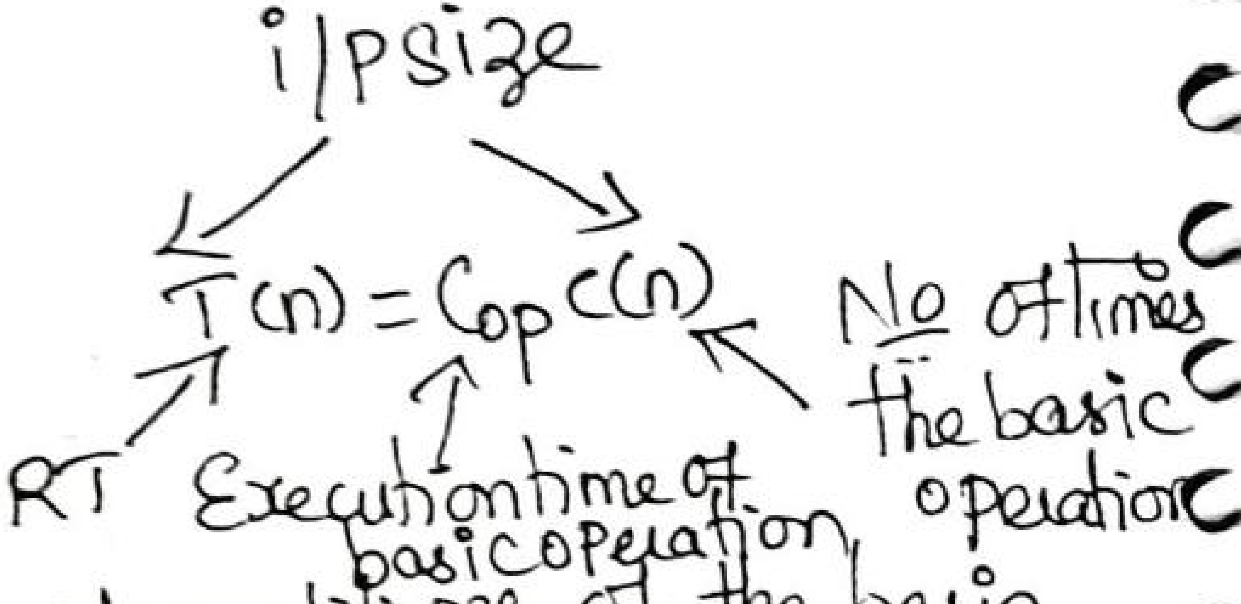
n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	3.3×10^1	10^2	10^3	2^{10}	3.6×10^6.

Algorithms with time Complexity n & $100n$ are called linear time algorithm.

" " " " n^2 & $0.01n^2$ are called Quadratic time Complexity algorithm.

Approaches in Analysis of Algorithms

1. Theoretical Analysis
2. Empirical Analysis.



Theoretical Analysis

(i) Time efficiency is analyzed by determining the no. of repetitions of the basic operation as a function of i/p size

(ii) Basic operation: the operation that contributes most towards the running time of the algorithm. [May 2018]

(iii) $T(n) \approx C_{op}\, C(n)$.

Empirical Analysis

(i) Select a specific (typical) sample of inputs

(ii) Use physical unit of time or count actual number of basic operations executions

iii) Analyze the empirical data.

Worst - Case, Best - case and Average - case Efficiencies

The basic operation in Sequential Search is not done the same number of times for all instances of size n. So this algorithm does not have an every-case time complexity, because the number of times the basic operation is done depends not only on the i/p size, but also on the i/p values.

for eg,.. in the Sequential Search if k is the 1st element in the array, the basic operation is done once, whereas if k is not in the array, it is done n times leads to different case efficiencies, such as worst, best and average case efficiency.

Worst - case efficiency

(i) Considers the maximum number of times the basic operation is executed.

(ii) $W(n)$ or $C_{worst}(n)$ is defined as the maximum number of times the algorithm will ever do its basic operation for an i/p size of n.

$W(n)$ is called the worst Case time complexity analysis.

(iii) If $T(n)$, every case time Complexity exists, then clearly $W(n) = T(n)$ whereas if $T(n)$ does not exist, we have to find $W(n)$.

Best - case efficiency

$B(n)$ or $C_{best}(n)$ is called the best case time Complexity of the algorithm and best-case time Complexity analysis.

It is defined as the minimum number of times the algorithm will ever do its basic operation for an input of size n.

Average case efficiency

→ $A(n)$ or $C_{avg}(n)$ is defined as the average or expected value of the number of times the algorithm does the basic operation for an input size of n.

→ called as average-case time complexity of the algorithm & the determination is called average - case time complexity analysis

→ To compute $A(n)$, we need to assign probabilities to all possible inputs of size n. for example, in sequential search, we will assume that if k is in the array, it is equally likely to be in any of the array slot.

```
Algorithm SequentialSearch (A[0...n-1], k)
    // Searches for a given value in a given array by sequential
       Search.
    // Input : An array A[0...n-1] & a search key k.
    // output : Index of the 1st element of A that matches k or
                -1 if there are no matching elements.

    i ← 0
    while i < n and A[i] ≠ k do
        i ← i+1
    if i < n return i else return -1
```

Basic operation : Comparison of an item in the array with k.

Input size : n, number of items in the array.

Worst-case: The basic operation is done at most n times, which is the case when k is the last item in the array or if k is not in the array

$$W(n) \text{ or } C_{worst}(n) = n.$$

Best case:- Because $n \geq 1$, there must be at least 1 pass through the loop. If key = A[1], there will be 1 pass through the loop regardless of size n.

$$\therefore B(n) \text{ or } C_{best}(n) = 1.$$

Average case:-

first analyse the case where it is known that k is in S, where the items in S are all distinct, & where we have no reason to believe that k is more likely to be in 1 array slot than it is to be in another. for $1 \leq i \leq n$, the probability that k is in the i^{th} array slot is $1/n$.

$$A(n) \text{ or } C_{avg}(n) = \sum_{i=1}^{n} (i \times 1/n)$$

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

$$= 1/n \times \sum (1+2+\dots+n)$$

$$= 1/\cancel{n} \times \frac{\cancel{n}(n+1)}{2}$$

$$= \frac{n+1}{2}.$$

Next, we analyse the case where k may not be in the array. To analyze this case, we must assign some probability p to the event that k is in the array.

If k is in the array, we will again assume that it is equally likely to be in any of the slots from 1 to n in the array. The Probability that k is in the i^{st} slot is the P/n and the probability that it is not in the array is 1-P.

The algorithm Undergoes i passes through the loop if k is found in the i^{th} slot and n passes through the loop, if k is not in the array.

$$\therefore A(n) = C_{avg}(n) = \sum_{i=1}^{n} i \times P/n + n(1-P)$$

$$= \frac{P}{n} \times \sum_{i=1}^{n} i + n(1-P).$$

$$= \frac{P}{\cancel{n}} * \frac{\cancel{n}(n+1)}{2} + n(1-P)$$

$$C_{avg}(n) = \frac{P(n+1)}{2} + n(1-P).$$

Explain briefly Big-oh Notation, Omega Notation & Theta Notations [Apr/May'17]
(or)
what are the rules of manipulate Big-oh Expressions & about the typical growth rate of algorithms [Nov/Dec'17]
Asymptotic Notation & its properties [May 2018]

→ The order of growth of the running time of an algorithms, gives a simple characterization of algorithms efficiency.

→ allows to compare relative performance of alternative algorithms.

for example, when i/p size is large enough, merge sort, whose worst-case running time $O(n \log n)$ beats insertion sort, whose worst case running time is $O(n^2)$.

→ Asymptotic notations are methods to estimate and represent efficiency of an algorithm using simple formula.

→ Useful for separating algorithms that do drastically different amount of work for large inputs.

Assume 2 algorithms for a problem with $2Cn$, $4Cn$ as total number of operations, which one run faster?

To compare or classify functions that ignore constant factors and small inputs, such a classification is called as Asymptotic growth rate or asymptotic order.

It is concerned with how the running time of an algorithm increases with the size of the input, if input increases from small value to large values.

→ Complexity is Usually represented in O, Θ, Ω or in $f(n)$ notation

→ These are the methods to measure efficiency of the algorithm.

1. Big-oh notation (O)
2. Big-Omega notation (Ω)
3. Theta notation (Θ)
4. Little-oh notation (o)
5. Little-omega notation (ω)

Big-oh Notation (O)

→ method of representing the upper bound of algorithm's running time.

→ Used to define the worst-case Complexity & give longest amount of time taken by the algorithm to complete.

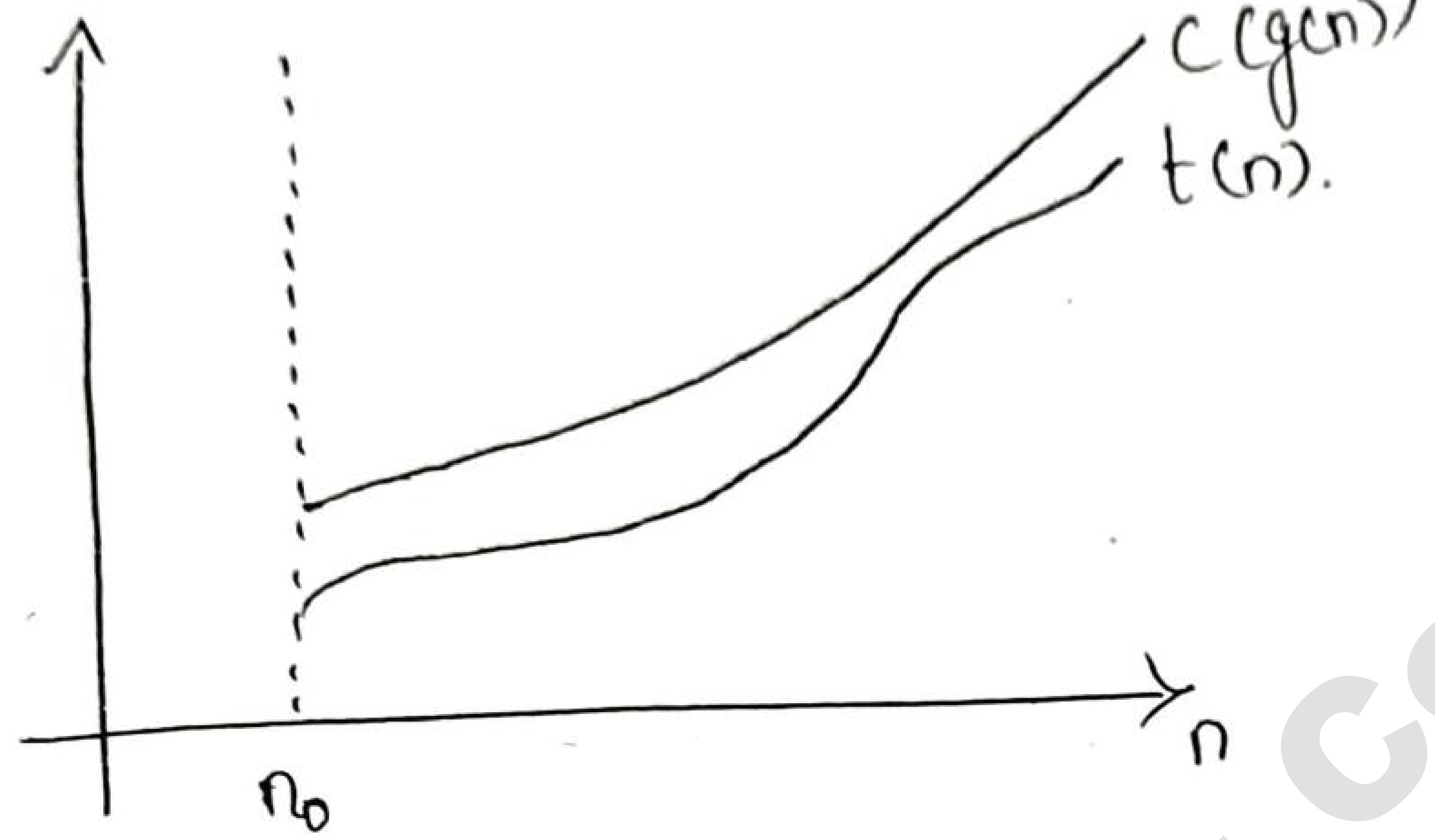
→ Concerned with large values of n.

→ $O(g(n))$ is the set of all functions with a smaller or Same order of growth as $g(n)$

Definition : A function $t(n)$ is said to be in $O(g(n))$, denoted as $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some Constant multiple of $g(n)$, for all large n

(i.e) if there exist some positive Constant c and some non-negative integer n_0 Such that $t(n) \leq c g(n) \ \forall \ n \geq n_0$.

$O(g(n))$: class of functions $t(n)$ that grow no faster than $g(n)$. Big-oh puts asymptotic upper bound on a function.



Eg:- Consider function $f(n)=2n+2$ and $g(n)=n^2$ Then we have to find some constant c, so that $f(n) \leq c * g(n)$. As $f(n)=2n+2$ and $g(n)=n^2$ then we find c for n=1 then

$f(n)=2n+2$
$=2\times1+2=4$

$g(n)=n^2$
$=1$

(i.e) $f(n)$ is not less than $g(n)$

if $n=2$ $f(n)=2\times2+2$
$=4+2=6$

$g(n)=n^2$
$=2^2=4$

(i.e) $f(n) \not< g(n)$

if $n=3$ $f(n)=2\times3+2$
$=8$

$g(n)=3^2=9$

$f(n)<g(n)$. is true.

Hence we conclude for $n>2$, we obtain $f(n)<g(n)$.

$3\log n+8$
$5n+7$
$5n^2+2n$
$2\log n$
$4n^2$
$6n^2+9$

Represents the classes of linear, logarithmic, quadratic functions that belong to $O(n^2)$.

Big - Omega Notation

→ Used to describe the best case running time of algorithms & Concerned with large values of n.

Definition:- A function t(n) is said to be in $\Omega(g(n))$, denoted as $t(n) \in \Omega(g(n))$, if t(n) is bounded below by some positive Constant multiple of g(n) for all large n.

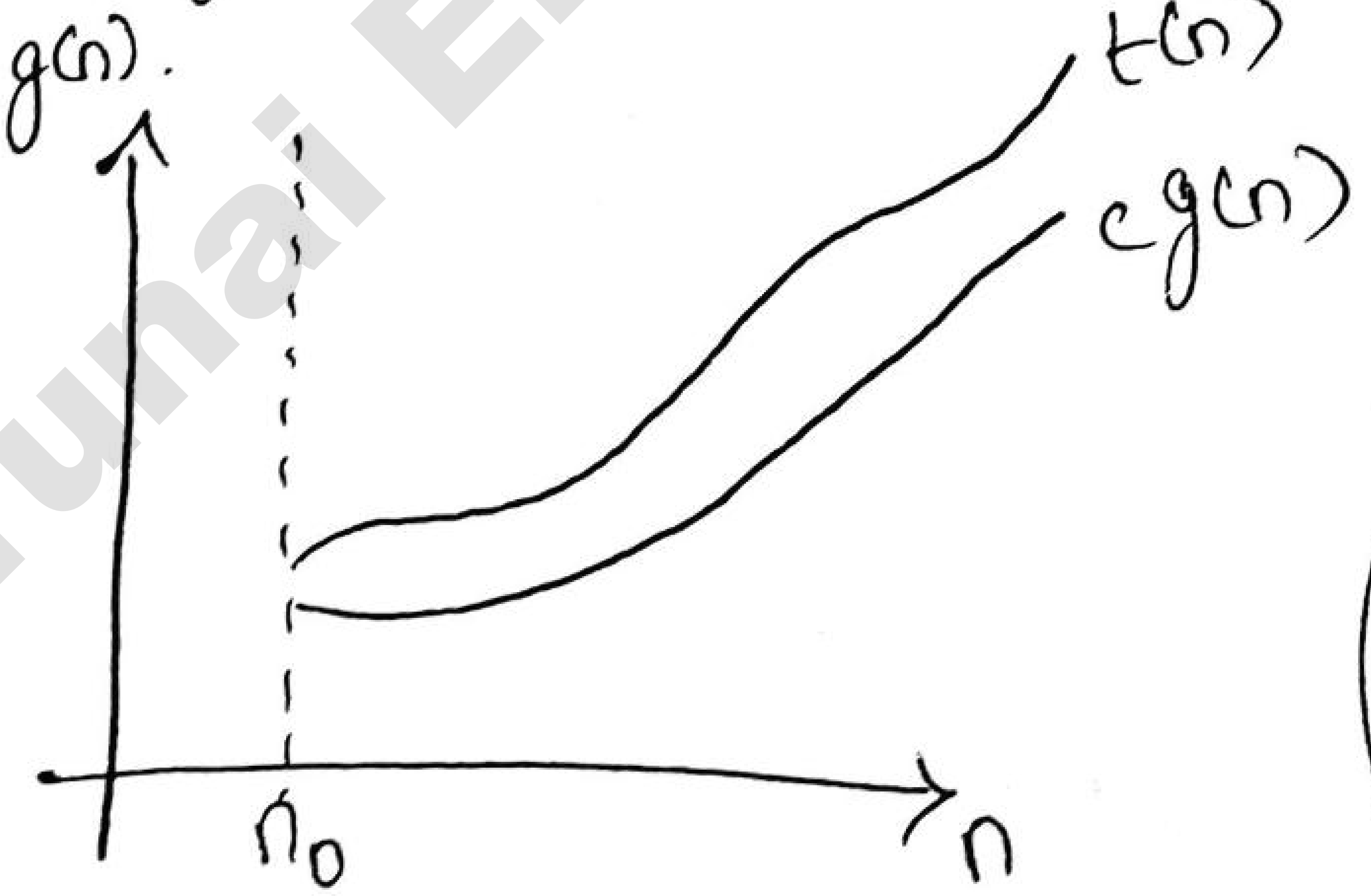
(i.e) there exist Some positive Constant c & Some non-negative integer n_0, Such that

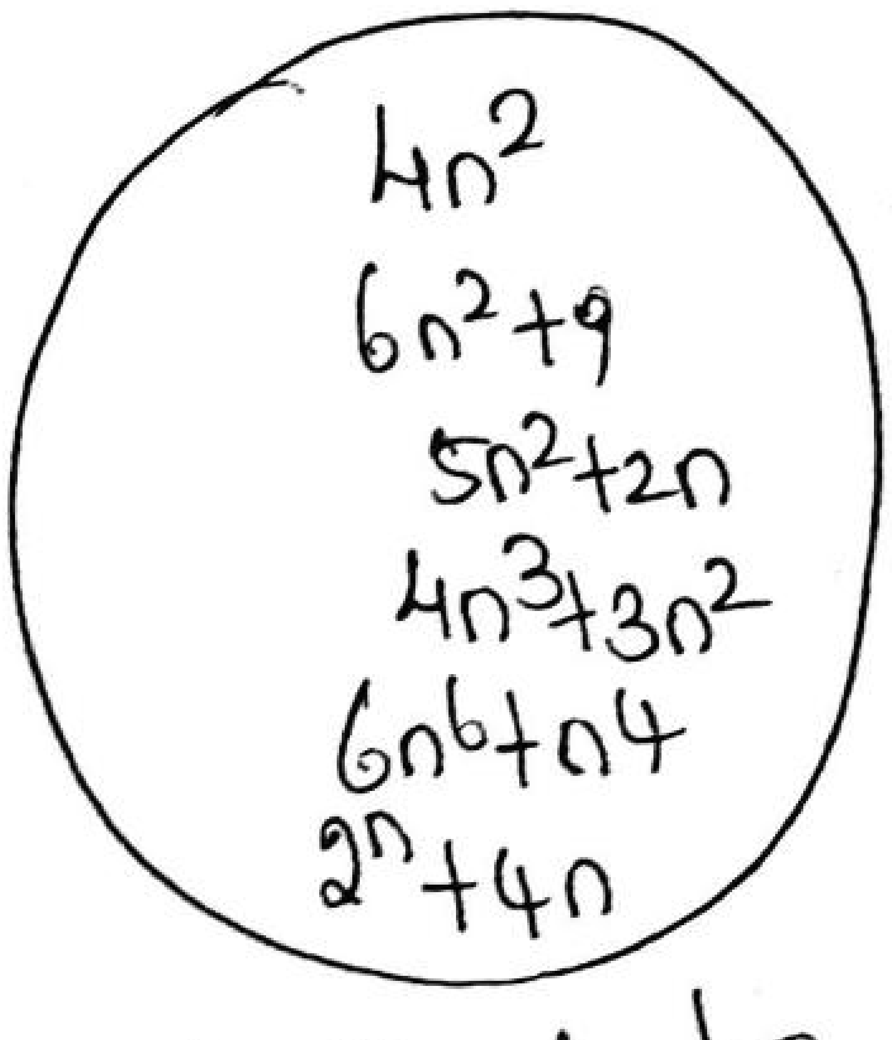
$$t(n) \geq c\,g(n) \;\forall\; n \geq n_0 .$$

→ It represent the lower bound of the resources required to solve a Problem.

→ $\Omega(g(n))$ Stands for the set of all functions with a larger or Same order of growth as g(n).

→ $\Omega(g(n))$: class of functions t(n) that grow atleast as fast as g(n).





Represents the functions that belongs to $\Omega(n^2)$.

eg. Consider $f(n)=2n^2+5$ and $g(n)=7n$

Then if $n=0$

$f(n)=2(0)^2+5=5$ $\quad$ $g(n)=7(0)=0$

$f(n)>g(n)$

if $n=1$

$f(n)=2+5=7$

$g(n)=7(1)=7$ $\quad$ $f(n)=g(n)$

if $n=2$

$f(n)=4+5=9$ $\quad$ $f(n)\leq g(n)$

$g(n)=7(2)=14$

if $n=3$

$f(n)=2(3^2)+5=23$ $\quad$ $f(n)>g(n)$.

$g(n)=7(3)=21$

$\therefore 2n^2+5 \in \Omega(n)$

|||ly any $n^3 \in \Omega(n^2)$.

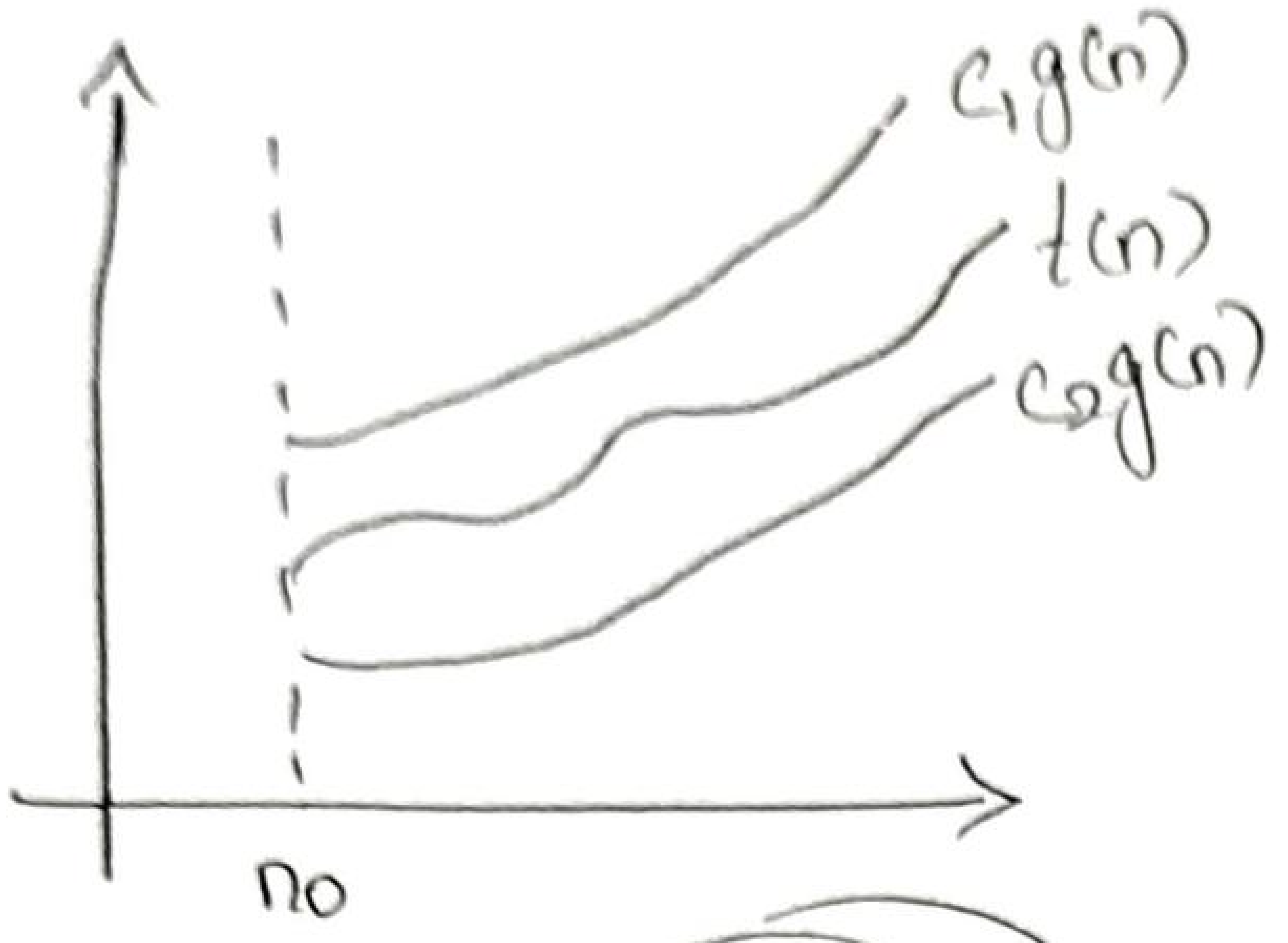
Theta Notation (Θ)

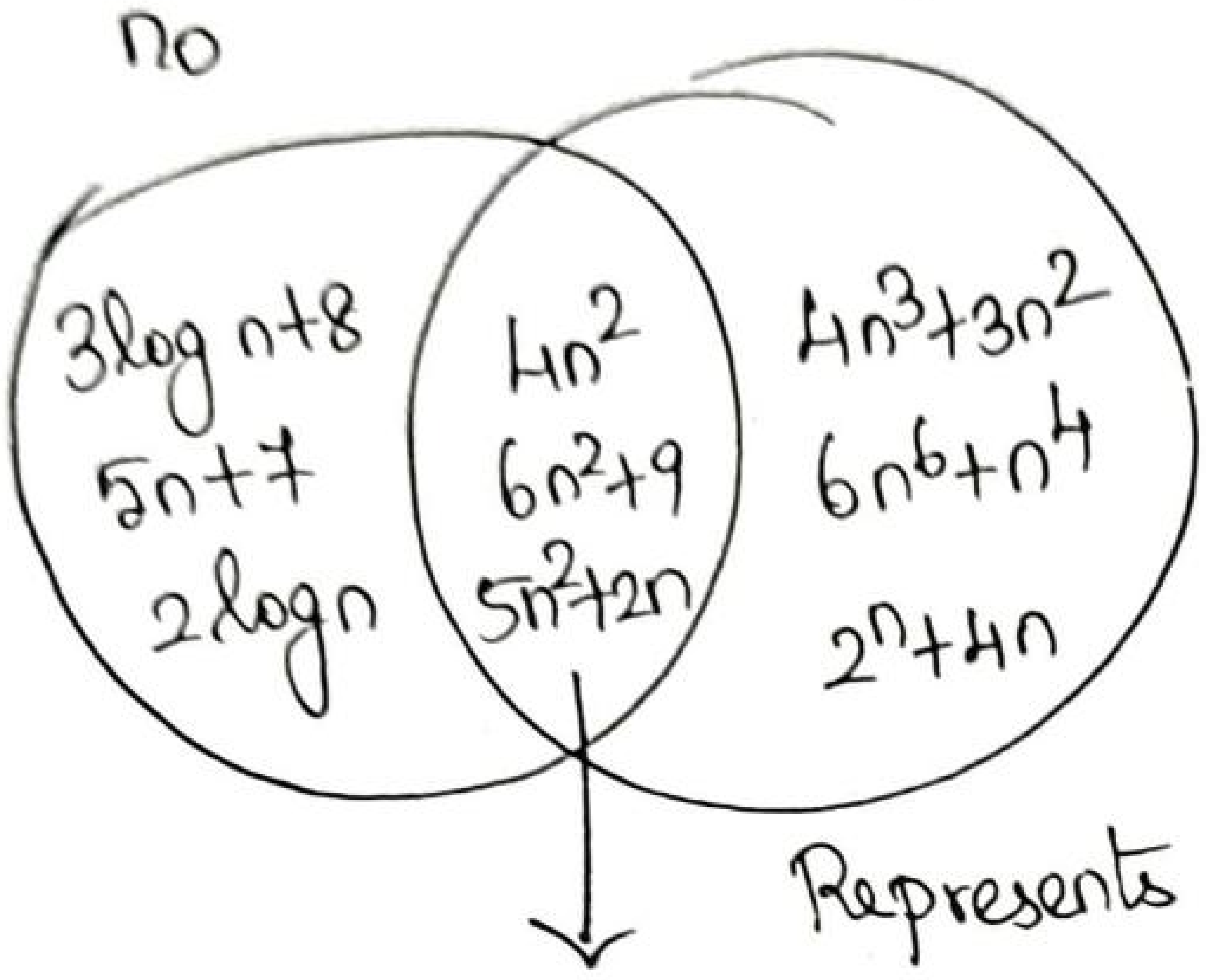
A function $t(n)$ is said to be in $\Theta(g(n))$, denoted by $t(n) \in \Theta g(n)$, if $t(n)$ is bounded both above & below by some positive constant multiples of $g(n)$ for all large n.

(i.e) if there exist some positive constant c_1 & c_2 & some non-negative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n). \quad \forall n > n_0.$$

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$





Represents the functions that belongs to $\Theta(n^2)$.

Little-oh notation :- (o)

→ used to describe worst case analysis of algorithms & concerned with small values of n.

→ Definition :- A function $t(n)$ is said to be in $o(g(n))$, denoted $t(n) \in o(g(n))$, if there exists some positive constant c & some non-negative integer such that

$$t(n) \leq c * g(n).$$

$$\rightarrow \lim_{n \to \infty} \frac{t(n)}{g(n)} = 0.$$

Little-omega Notation :- (ω)

→ notation used to describe the best-case analysis of algorithms & concerned with small values of n.

→ function $t(n) = \omega(g(n))$ iff $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$ or $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$

Propeties of Big-oh, Ω & Θ :-

(i) If there are 2 functions $f_1(n)$ and $f_2(n)$ Such that $f_1(n)=O(g_1(n))$ and $f_2(n)=O(g_2(n))$ then $f_1(n)+f_2(n)=\max(O(g_1(n)), O(g_2(n)))$

(ii) if there are 2 functions $f_1(n)$ and $f_2(n)$ Such that $f_1(n)=O(g_1(n))$ and $f_2(n)=O(g_2(n))$ then $f_1(n) * f_2(n)=O(g_1(n)) * g_2(n))$.

(iii) If there exists a function f_1 Such that $f_1=f_2 * c$ where c is the Constant then, f_1 and f_2 are equivalent. That means $O(f_1+f_2)=O(f_1)=O(f_2)$

(iv) If $f(n)\in O(g(n))$ and $g(n)\in O(h(n))$ then $f(n)\in O(h(n))$ that is O is transitive.

(v) In a polynomial the highest power term dominates other terms.

(vi) Any Constant value leads to O(1) time Complexity.

(vii) if $\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$ then $f(n)\in O(g(n))$ but $f(n)\notin\Theta(g(n))$.

L' Hospital rule

Let t and g be differential functions with derivatives t' & g' respectively Such that

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

then

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\lim_{n\to\infty}\frac{t'(n)}{g'(n)}.$$

Division with infinity & Zero

$\frac{0}{\text{any no}}=0$ $\quad$ $\frac{\text{any no}}{\infty}=0$

$\frac{\text{any no}}{0}=\infty$ $\quad$ $\frac{\infty}{\text{any no}}=\infty$

$\frac{0}{\infty}=0$, $\frac{\infty}{0}=\infty$.

Basic Asymptotic Efficiency classes

Name of Efficiency class	order of growth	Description	Examples
Constant	1	i/p size grows then we get larger running time	Scanning array elements
Logarithmic	logn	When we get logarithmic RT then it is sure that the algorithm does not consider all its i/p rather the problem is divided into smaller parts on each iteration	Binary search.
Linear	n	RT of algorithm depends on the i/p size n	Sequential search
n log n	nlogn	Some instance of i/p considered for the list of size n.	Sorting using merge & Quick sort
Quadratic	n^2	if algm. has 2 nested loops then this type of efficiency occurs	Scanning matrix elements
Cubic	n^3	3 nested loops.	matrix multiplication.
Exponential	2^n	algorithm has very faster rate of growth.	Generating all subsets of n elements
factorial	n!	algorithm is computing all the permutations	Generating all

Briefly Explain the mathematical analysis of recursive and non-recursive
(or) algorithm [APr/May 2017].

14

General plan for Analyzing Efficiency of Non-recursive Algorithms:-

1) Decide on a parameter (or parameters) indicating an input's size.

2). Identify the algorithm's basic operation (As a rule, it is located in its inner-most loop).

3) check whether the number of times the basic operation executed depends only on the size of an i/p. If it also depends on some additional property, the worst-case, best-case & average-case are to be done.

4) Set up a sum expressing the number of times the algorithm's basic operation is executed.

5) Using standard formulas & rules of sum manipulation, either find a closed form formula for the count or at least establish in orders of growth.

Example: Consider a problem of finding the value of the largest element in a list of n numbers.

```
Algorithm MaxElement(A[0...n-1])
    // Determines the value of the largest element in a given array
    // Input: An array A[0...n-1] of real numbers.
    // output: The value of the largest number in A.
maxval ← A[0]
    for i ← 1 to n-1 do
        if A[i] > maxval
            maxval ← A[i]
return maxval.
```

Analysis:

- Input size - Number of elements in the array = n
- Basic operation - The operation executed in the innermost loop - Contains 2 operations.

  * Comparison
  * Assignment.

In this example, Comparison is chosen as the basic operation because assignment depends on comparison.

→ No additional properties - implies there is no best, worst & average case efficiencies.

$$c(n) = \sum_{i=1}^{n-1} 1 = (n-1) - 1 + 1 = n-1 \in \Theta(n)$$

$$(\because \sum_{i=1}^{n} 1 = u - l + 1)$$

2) Algorithm that finds the number of binary digits in the binary representation of a positive decimal integer:-

```
Algorithm Binary(n)
    // input : A positive decimal integer n
    // output : number of binary digits in n's binary representation
    Count ← 1
    while n > 1 do
        Count ← Count + 1
        n ← ⌊n/2⌋
    return count
```

Analysis

- Input size is equivalent to the number of elements in the array = n
- Basic operation - Comparison
- No additional property

* efficiency $\log_2 n+1$ - it implies execution of comparison is done half of the time only.

Discuss the steps in Mathematical analysis for recursive algorithms. Do the Same for finding the factorial of a number. [Nov/Dec 2017].

General plan for Analyzing Efficiency of Recursive Algorithms [May 15 2018]

1. Decide on a parameter (or parameters) indicating an input size.

2. Identify the algorithm's basic operation.

3. Check whether the number of times the basic operation is executed depends only on the input size. If it also depends on some additional property, the worst, average & best case, have to be done.

4) Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

5) Solve the recurrence or at least ascertain the order of growth of its solution.

eg:. factorial Computation

Tower of Hanoi.

finding the number of binary digits in the binary representation of a positive decimal integer.

Fibbonacci Series generation

1) factorial Computation: $F(n)=n!$.

$n! = 1.2. \ldots (n-1) * n = (n-1)! * n$ for $n \geq 1$

The recurrence relation is

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

As per initial Condition $n-i=0$ $\therefore n=i$ ..

$m(n) = m(0) + n$

$= O(n)$.

2) Tower of Hanoi

The problem "Towers of Hanoi" is a classic example of recursive function. The problem have n disks of different sizes & 3 Pegs. Initially all the disks are on the first peg in order of size, the largest on the bottom & the smallest on top.

Goal :- To move all the disks to the third peg, using the second one as an auxillary. only 1 disk can be moved at a time & placing a larger disk on top of smaller one is forbidden.

Solution is stated as

1. Move top n-1 disks from A to B using C as auxillary.
2. Move the remaining disk from A to C.
3. Move the n-1 disks from B to C using A as auxillary.

Mathematical Analysis

1) i/p size is n (i.e) total number of disks.

2) Basic operation is moving disks from 1 peg to another.

(i.e) when $n>1$, then to move those disks from peg A to peg C using Peg B, we first recursively $n-1$ disks from peg A to Peg B using auxillary Peg C. Then we move the largest disk directly from peg A to Peg C. and finally move $n-1$ disks from peg B to peg C. (Peg A - auxillary)

If $n=1$ then we simply move the disk from peg A to peg C.

Step 3: Moves of disks are denoted by $M(n)$. $M(n)$ depends on the number of disks n. The recurrence relation can then be set up

$M(1)=1$

if n>1, then we need 2 recursive calls plus 1 move. Hence

$$M(n) = \underbrace{M(n-1)}_{\text{To move (n-1) disks from peg A to B}} + \underbrace{1}_{\text{To move largest disk from peg A to C}} + M(n-1) \longleftarrow \text{To move (n-1) disks from peg B to C.}$$

$$\therefore \boxed{M(n) = 2M(n-1)+1}$$

Solve the recurrence $M(n)=2M(n-1)+1$

$$M(n) = 2M(n-1)+1 \rightarrow ①$$

Put $n=n-1$ in ① we get

$$M(n-1) = 2M(n-2)+1 \rightarrow ②$$

Subs ② in ① we get

$$M(n) = 2[(2M(n-2))+1]+1$$

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

$$\vdots$$

$$=2^i(M(n-i))+2^{i-1}+1$$

This can also be written as

$$M(n)=2^i(M(n-i))+2^i-1 \longrightarrow ③$$

Subs $n=n-i$ in ③ we get

$\therefore$ by initial condition

$n=1$ then

$n-i=1$

$i=n-1$

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

$$=2 \cdot 2^{n-1}-1$$

$$M(n) = 2^n-1 //$$

The time complexity of ToH as $O(2^n)$.

Important Summation formulas & Rules

1. $\sum_{i=1}^{n} 1 = 1+1+ \dots +1 = n \in O(n)$ $(\because u-l+1)$

2. $\sum_{i=1}^{n} i = 1+2+ \dots +n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 \in O(n^2)$

3. $\sum_{i=1} i^k = 1+2^k+3^k+ \dots +n^k = \frac{n^{k+1}}{k+1} \in \Theta(n^{k+1})$

4. $\sum_{i=1}^{n} a^i = 1+a+a^2+ \dots +a^n = \frac{a^{n+1}-1}{a-1} \in \Theta(a^n)$.

5. $\sum_{i=1}^{n} (a_i \pm b_i) = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i$.

6. $\sum_{i=1}^{n} i2^i = 1*2+2*2^2+ \dots n2^n = (n-1)2^{n+1}+2$

7. $\sum_{i=1}^{n} \frac{1}{i} = 1+\frac{1}{2}+ \dots +\frac{1}{n} \approx \ln n+\gamma$

8. $\sum_{i=1}^{n} \lg i = n \lg n$.

Rules

1. $\sum_{i=1}^{u} c a_i = c \sum_{i=1}^{u} a_i$
2. $\sum_{i=1}^{u} (a_i \pm b_i) = \sum_{i=1}^{u} a_i \pm \sum_{i=1}^{u} b_i$
3. $\sum_{i=1}^{u} a_i = \sum_{i=1}^{m} a_i + \sum_{i=m+1}^{u} a_i$ where $l \le m < u$

Properties of Logarithms

1. $\log x^y = y \log x$
2. $\log xy = \log x + \log y$
3. $\log \frac{x}{y} = \log x - \log y$
4. $\log_a x = \log_a b \log_b x$.
5. $a^{\log_b x} = x^{\log_b a}$
6. $\log_a 1 = 0$
7. $\log_a a = 1$.

Recurrence Relation:- or Recurrence Equation

The recurrence equation is an equation that defines a sequence recursively. It is normally in following form

$T(n) = T(n-1) + n$ for $n > 0$ → ①

$T(0) = 0$ → ②

Here ① is called recurrence relation & equation ② is called initial condition. The recurrence equation can have infinite number of sequences. The general solution to the recursive function specifies some formulla.

⟹ The recurrence relation can be solved by following methods..

1. Substitution
   - Backward
   - Forward

2. Master's Method.

⟹ Substitution Method

→ It is a kind of method in which a guess for the solution is made.

Forward Substitution Method — This method makes use of an initial condition in the initial term & value for the next term is generated. This process is continued until some formula is guessed (very difficult to guess the pattern.)

Backward Substitution Method —

In this method, backward values are substituted recursively in order to derive some formula.

Eg/.. $T(n) = T(n-1) + n$.

FSM

Given $T(0)=0$

$T(1) = T(1-1) + 1$
$= T(0)+1$
$= 1$

$T(2) = T(2-1) + 2$
$= T(1) + 2$
$= 3$

$T(3) = T(3-1) + 3$
$= 3 + 3$
$= 6$

⋮

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

$\therefore T(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + n/2$.

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

BSM

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

Put $n = n-1$ in ① we get

$T(n-1) = T(n-2) + n-1 \rightarrow ②$

Put ② in ① we get

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

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

⋮

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

if $k = n$.

$T(n) = T(0) + 1 + 2 + \ldots n$.
$= 0 + \frac{n(n+1)}{2}$.

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

Masters' Method

We can also solve recurrence relation using a formula denoted by master's method. $T(n) = aT(n/b) + F(n)$ where $n \geq d$ & d is some constant. Then the master's theorem can be stated for efficiency analysis as

if $F(n)$ is $O(n^d)$ where $d \geq 0$ is the RR, then

1. $T(n) = O(n^d)$ if $a < b^d$
2. $T(n) = O(n^d \log n)$ if $a = b$
3. $T(n) = O(n^{\log_b a})$ if $a > b^d$.

Solve the following recurrence relations:- [Nov/Dec '16]

1) $x(n) = x(n-1) + 5$ for $n>1$, $x(1)=0$

2) $x(n) = 3x(n-1)$ for $n>1$, $x(1)=4$

3) $x(n) = x(n-1) + n$ for $n>0$ $x(0)=0$

4) $x(n) = x(n/2) + n$ for $n>1$ $x(1)=1$ (Solve for $n=2^k$)

5) $x(n) = x(n/3)+1$ for $n>1$, $x(1)=1$ (Solve for $n=3^k$)

1) $x(n) = x(n-1) + 5 \rightarrow$ ① Backward Substitution.

Put $n=n-1$ in ① we get

$x(n-1) = x(n-2) + 5 \rightarrow$ ②

Substitute ② in ① we get

$x(n) = x(n-2) + 5 + 5 \rightarrow$ ③

$x(n-2) = x(n-3) + 5 \rightarrow$ ④

Again Substitute ④ in ③ we get

$x(n) = x(n-3) + 10 + 5$

$x(n) = x(n-3) + 3 \times 5$

when we generalize we get

$x(n) = x(n-i) + i \times 5$ (for $1 \leq i \leq n-1$)

Initial Condition $n-i=1$

$\therefore i = n-1$

$\therefore x(n) = x(n-(n-1)) + (n-1) \times 5$

$= x(1) + 5n - 5$

$= 5(n-1)$ is the Solution //.

2) $x(n) = 3x(n-1)$ for $n>1$, $x(1)=4$ (forward Substitution)

Solution:-

$x(1) = 4$

$x(2) = 3x(2-1) = 3x(1) = 3\times4 = 12$

$x(3) = 3x(3-1) = 3*(x(2)) = 3\times12 = 36$

$x(4) = 3x(4-1) = 3*(x(3)) = 3\times36 = 108$

$x(5) = 3x(5-1) = 3*(x(4) = 3\times108 = 324$

we generalize

Such that

$x(n) = 3^{n-1} * 4$

$x(n) = 4(3^{(n-1)})$ is the solution.

3) $x(n) = x(n-1)+n$ for $n>0$, $x(0)=0$

Solution:-

$x(n) = x(n-1)+n \rightarrow ①$

Put $n=n-1$

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

$\therefore x(n) = x(n-2)+n-1+n$

$= x(n-2)+2n-1 \rightarrow ②$

Put $n=n-2$ then

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

Sub in ② we get

$x(n) = x(n-3)+n-2+n-1+n$

for $1\le i\le n$

$x(n) = x(n-i)+(n-i+1)+(n-i+2)+ \dots +n.$

$i=n$ because initial condition is $x(0)=0$ $\therefore n-i=0$

$i=n$ //

$\therefore x(n) = x(0)+1+2+\dots+n.$

$= 0+(1+2+\dots+n)$

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

4) $x(n) = x(n/2) + n$ for $n > 1$, $x(1) = 1$

Soln let $n = 2^k$

$x(n) = x(2^k) = x(\frac{2^k}{2}) + 2^k$

$= x(2^{k-1}) + 2^k \rightarrow ①$

Use in $x(n)$ $n = 2^{k-1}$ we get

$x(2^{k-1}) = x(2^{k-2}) + 2^{k-1} \rightarrow ②$

Subs ② in ①

$x(2^k) = x(2^{k-2}) + 2^{k-1} + 2^k$

$= x(2^{k-3}) + 2^{k-2} - 2^{k-1} + 2^k$

$= x(2^{k-i}) + 2^{k-i+1} + 2^{k-i+2} + \dots + 2^k$

$i = k$ initial condition $x(2^0) = (1)$

$x(1) = 1$

$= x(2^0) + 2^1 + 2^2 + \dots 2^k$

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

$= 2^k \cdot 2 - 1 = 2n - 1$ // is the solution.

5) $x(n) = x(n/3) + 1$ for $n > 1$ $x(1) = 1$

let $n = 3^k$

Solution :- $x(3^k) = x(3^{(k-1)}) + 1$

$x(3^{k-1}) = x(3^{k-2})+1$

$x(3^k) = x(3^{k-2})+1+1 \quad \forall \ 1 \le i \le n$

$x(3^k) = x(3^{k-i})+i$

$i-k=0 \quad i=k$ as per initial condition

$x(3^k) = x(3^0)+k$

$= x(1)+k$

$= 1+k \qquad \because 3^k = n$

$x(3^k) = 1+\log_3 n \qquad k = \log_3 n.$

$\therefore x(n) = 1+\log_3 n$

is the solution

The general form of homogeneous linear recurrence relation is $a_0 t(n) + a_1 t(n-1) + \dots + a_k t(n-k) = 0$

eg.. $1t(n) - 3t(n-1) = 0$

⇒ Solve $t(n) - 3t(n-1) - 4t(n-2) = 0$ for $n > 1$

$n(0)=0, \ t(1)=1$

Solution

$t(n) - 3t(n-1) - 4t(n-2) = 0$

let $t(n) = r^n$

$\Rightarrow r^n - 3r^{n-1} - 4r^{n-2} = 0$

$r^{n-2}(r^2 - 3r - 4) = 0$

Roots of r are $r = 4$ and $r = -1$

$\therefore t(n) = c_1 4^n + c_2 (-1)^n.$

$t(0) = c_1 4^0 + c_2(-1)^0 = 0$

$\Rightarrow c_1 + c_2 = 0 \rightarrow ①$

$t(1) = 1$

$= c_1 4^1 + c_2(-1)^1$

$\Rightarrow 4c_1 - c_2 = 1 \rightarrow ②$

solving ① & ② we get

$5c_1 = 1$

$c_1 = 1/5$

$\therefore c_2 = -1/5$

$\therefore t(n) = \frac{1}{5} \times 4^n + (-1/5)(-1)^n$ //

UNIT II

UNIT - II

Brute Force & Divide and Conquer

Brute-force - closest pair and Convex hull problems - String Matching Exhaustive Search - TSP - knapsack problem - Assignment problem - Computing a^n. Divide & Conquer Methodology - Merge Sort - Quick Sort - Binary Search - Multiplication of large numbers - Strassen's matrix multiplication - Closest pair & Convex hull problems.

→ Brute force is the Simplest technique of design strategies

→ It is a straight forward approach to solve a problem based on Problem statement.

eg.. 1) Computing a^n (given number a and a non-negative integer n)

(i.e) by exponentiation

$$a^n = \underbrace{a * a * a \ldots * a}_{n \text{ times}}$$

2) Computing n!

3) Multiplying 2 Matrices

4) Searching a key from a list

5) Compute gcd.

6) Computing Sum of n numbers

7) finding the largest element in a list.

Analysis of brute force design strategy

→ for Sorting, Searching & string matching problems, the brute force strategy results an algorithm with no limitation on instance size.

→ If the problem belongs to inefficient in general, a brute force algorithm can solve small size instances of a problem.

→ Useful in theoretical or educational purpose.

Convex hull problem

A shape or set is Convex if for any 2 points that are part of the shape, the whole connecting line segment is also a part of the shape.

What is Convex hull?

* Let S be a set of points in the plane.

* Intuition:- Imagine the points of S as being pegs; the Convex hull of S is the shape of a rubber-band stretched around the pegs.

* formal definition:- The Convex hull of S is the Smallest Convex polygon that Contains all the points of S.

* Convex hull:- The Convex hull of a set of n points in the plane is the Smallest Convex polygon that Contains all of them either inside or on its boundary.

* A polygon is said to be convex if:
 - P is non-intersecting &
 - for any 2 points p & q on the boundary of P,

Segment pq lies entirely inside P.

Applications

→ Problems in Computational geometry
→ CAD / computer Animation.
→ Computing accessibility maps produced from Satellite images by GIS.
→ used for detecting outliers by some statistical techniques
→ Solving many optimization problems.

Extreme point of a Convex set

→ It is a point of this set that is not in/or a middle Point of any line Segment with end points in the set.

eg/.. extreme points of a Δ^{le} are its 3 vertices.
Extreme points of a O^{le} are all points of its Circumference.

Analyses

There are $n \cdot \frac{n-1}{2}$ pairs of distinct points.

for each of these pairs. find the sign of $ax+by-c$ for each of the other $n-2$ points.

$$\text{No/.. of checks} = n\cdot\left(\frac{n-1}{2}\right)(n-2) = \frac{n^3}{2} - \frac{3n^2}{2} + n$$

∴ Running time $= O(n^3)$.

Exhaustive Search

→ A brute force Solution to a problem involving Search of an element with a Special property, usually among combinatorial objects Such as permutations, Combinations or Subsets of a set is termed as Exhaustive Search

Strategy

1. Construct a way of listing all potential solutions to the Problem in a Systematic manner.
   - * all Solutions are eventually listed.
   - * no Solution is repeated.
2. Evaluate solutions 1 by 1, perhaps disqualifying infeasible ones & keeping track of the best one found so far.
3. when Search ends, announce the winner.

egs/. TSP – Travelling Salesman problem
0/1 knapsack problem
Assignment problem.

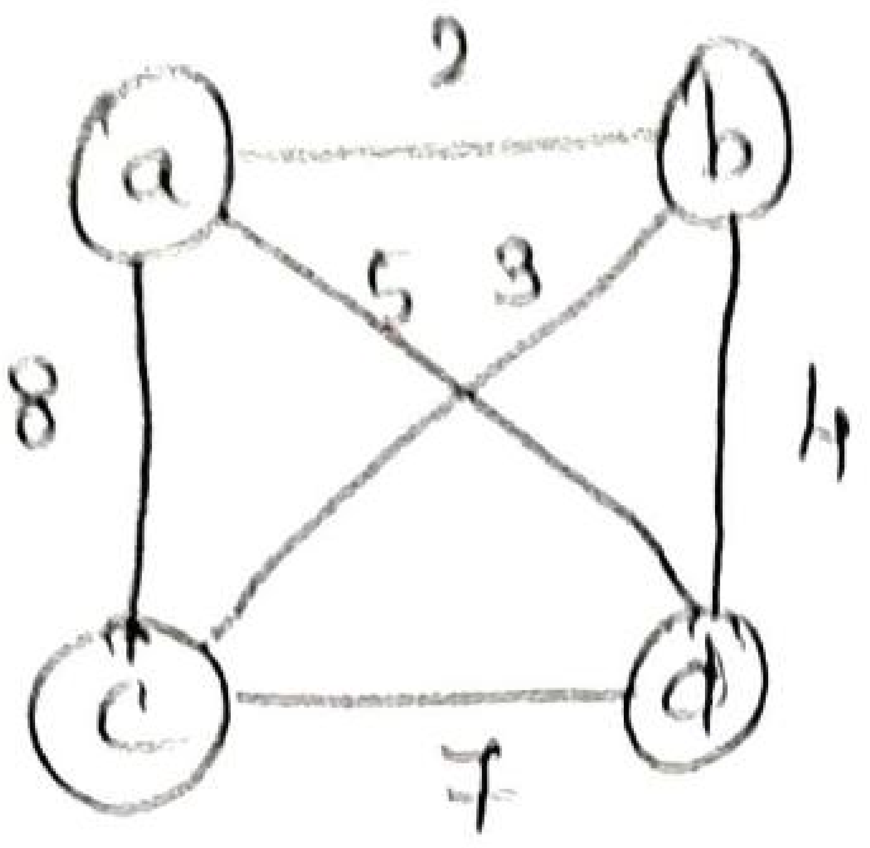
Travelling Saleman problem

→ Given n cities with known distances b/w each pair, find the shortest tour that passes through all cities exactly once before returning to the starting city. (or)

find Shortest Hamiltonian circuit in a weighted

Connected graph

Q1. Find the tour using exhaustive Search for the following graph.



Solution (with a as start point)

Tour	Cost
a-b-c-d-a	2+3+7+5=17
a-b-d-c-a	2+4+7+8=21
a-c-b-d-a	8+3+4+5=20
a-c-d-b-a	8+7+4+2=21
a-d-b-c-a	5+4+3+8=21
a-d-c-b-a	5+7+3+2=17

Efficiency – $(n-1)/2$

0/1 knapsack problem

Statement: Given a knapsack with maximum Capacity W, and a setsof Consisting of n items. Each item i has some weight w_i & benefite value v_i. The knapsack has to be packed to achieve maximum total value. It is mathematically defined as

$$\max \sum_{i \in T} v_i \text{ Subject to } \sum_{i \in T} w_i \le W$$

Eg: find the most valuable Subset of items that fit into the Knapsack

Item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Solution: let's first Solve this problem with a straight forward algorithm:-

Subset	Total weight	Total value
ϕ	0	$0
(1)	2	$20
(2)	5	$30
(3)	10	$50
(4)	5	$10
(1,2)	7	$50
(1,3)	12	$70
(1,4)	7	$30
(2,3)	15	$80
(2,4)	10	$40
(3,4)	15	$60
(1,2,3)	17	$100 not feasible
(1,3,4)	17	$80 not feasible
(2,3,4)	20	$90 not feasible
(1,2,4)	12	$60
(1,2,3,4)	22	$110 not feasible.

efficiency ∵ there are n items, there are 2^n possible Combinations

Thus Running time ∴ $O(2^n)$

Assignment problem

Statement: There are n people who need to be assigned to n jobs, 1 person per job. The cost of assigning person i to job j is $C[i,j]$. Find an assignment that minimizes the total cost.

eg1.	Job 0	Job1	Job2	Job3
0	9	2	7	8
1	6	4	3	7
2	5	8	1	8
3	7	6	9	4

Solution

1. Generate all legitimate assignments, Compute their costs & Select the cheapest one.
2. pose the problem as one about a cost matrix : Cycle cover in a graph.

Assignment colm wise	Cost
1,2,3,4	9+4+1+4 = 18
1,2,4,3	9+4+8+9 = 30
1,3,2,4	9+3+8+4 = 24
1,3,4,2	9+3+8+6 = 26
1,4,2,3	9+7+8+9 = 33
1,4,3,2	9+7+1+6 = 23

for a particular instance, the optimal assignment can be found by exploiting the Specific features of number given: 2,1,3,4 = 2+6+1+4 = 13//

Exhausive Search Applications

→ run in a realistic amount of time only on very small instances

→ In Some Cases, there are much better alternatives

- Euler circuits
- Shortest paths
- MST - Minimum Spanning Tree
- Assignment problem.

→ In many cases, Exhaustive Search or its variation is the only known way to get Exact solution.

Strength & weakness of brute force Strategy

Strength

- Wide applicability
- Simplicity
- Yields reasonable algorithms for Some important problems

Weakness

→ Rarely yields efficient algorithms

→ Some brute force algorithms are unacceptably slow.

→ Not very much Constructive.

[APR/MAY 2017]

eg.. <u>Brute force algorithm for String matching</u>

find whether the given string follows the specified pattern & return 0 or 1 accordingly

eg..

1. pattern: "abba" input : "redblueredblue" should return 1
2. Pattern : "aaaa" input : "asdasdasdasd" Should return 1
3. Pattern : "aabb" input : "xyzabcxyzabc" should return 0.

<u>Solution</u>:: The Brute force approach of string matching algorithm is very simple & straight forward. According to this approach, each character of pattern is compared with each corresponding character of text.

1. pattern: "<u>abab</u>" i/p text : "red blue redblue" return 1

a) if we map 'r' of string "red" with 'a' of pattern &

b) if we map 'b' of string "blue" with 'b' of pattern

then the algorithm will return 1.

```
// array t[] contains "redblue red blue"
// array p[] contains "abab"
// n represents length of text t[]
  int i,j,flag=1;
    i=0;
    j=0;
    While (j<n)
    { if((t[j]=='r')&&(p[i]=='a'))
       { i=i+1;
         i=i+3;
```

```
else if((t[j]=='b')&&
    (p[i]=='b'))
    {
      i=i+1;
      j=j+4;
    }
   else
   { flag=0;
      i++;
```

2) Consider Pattern : "aaaa" & i/p text : "asdasdasdasd" returns 1

idea 1) map 'a' of pattern to "asd" string.

3) Consider pattern : "aabb" & i/p text : "xyzabcxyzabc"

idea : 1) map 'a' of pattern to 'xyz' string & map. 'b' of pattern to 'abc' string which returns 0 for the pattern considered.

Time Complexity for exhaustive Search is always O(n!)

What is divide and Conquer Strategy and Explain the binary Search with Suitable example problem? [APR/MAY '17] 6

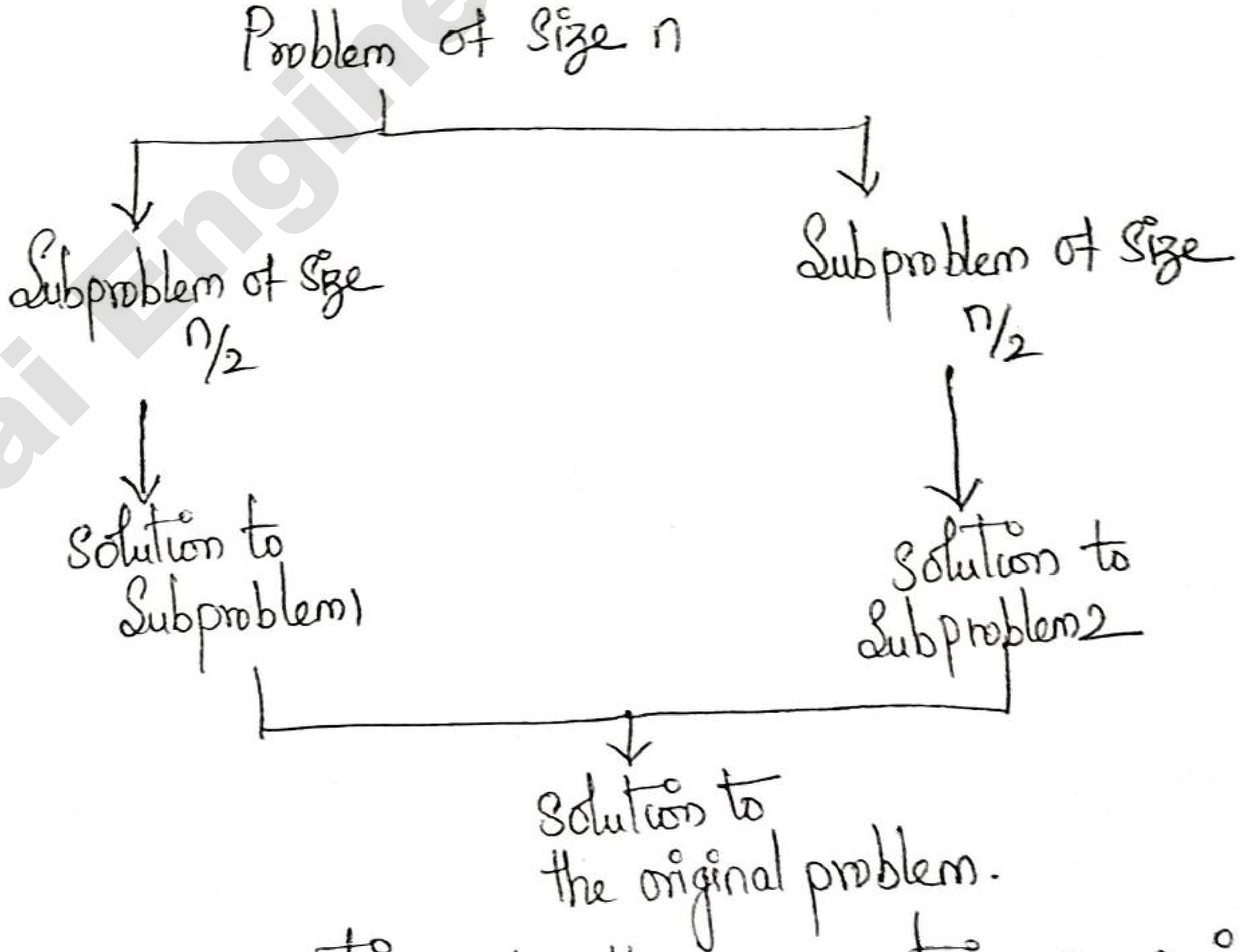
Divide and Conquer Strategy

→ In this method, the given problem is divided into Smaller instances of the Same problem then solve (conquer) the Smaller instances recursively and finally Combine the solutions to obtain the solution for the original input.

The recurrence equation that describes the amount of work done by this strategy

$$T(n) = D(n) + \sum_{i=1}^{K} T(\text{size}(I_i)) + C(n) .$$

where $T(n) = B(n)$ – No. of steps done by directly solve.
$D(n)$ – No. of steps done by divide
$C(n)$ – No. of steps done by Combine.



∴ The recurrence equation for the running time T(n) is

$$T(n) = aT(n/b) + f(n)$$

a, b = Constants where $a \geq 1$ & $b > 1$
n = Power of b

$f(n)$ – a function that accounts for the time spent on dividing the problem into smaller ones & combining their solutions.

Binary Search

→ It is an efficient method, while searching the elements using this method the most essential thing is that the elements in the array should be sorted one.

→ An element which is to be searched from the list of elements stored in array A [0...n−1] is called <u>key</u> element.

→ Let A[m] be the mid element of array A. Then there are three conditions that needs to be tested while searching the array using this method:-

1. If key = A[m] then desired element is present in the list
2. otherwise if key < A[m] then search left sublist.
3. otherwise if key > A[m] then search right sublist.

Q1. Consider a list of elements stored in array A

0	1	2	3	4	5	6
10	20	30	40	50	60	70

$$\text{mid\_term} = A[m] = \frac{0+6}{2} = 3$$

$$\therefore A[m] = A[3] = 40$$

Element to be searched is 60 then

$$A[M] = \text{search key}$$
$$40 \neq 60$$

7

$\therefore$ Search the element from the right Sublist

(i.e)

4	5	6
50	60	70

$\therefore m = \frac{4+6}{2} = 10/2 = 5$

$\therefore A[M] = A[5] = 60.$

is $A[M] = key$

$60 = 60.$

Yes (i.e) The number is present in the list.

The worst case efficiency is that the algorithm compares all the array elements for searching the desired element.

In this method, one Comparison is made and based on this Comparison, array is divided each time in $n/2$ Sublists.

Hence the $C_{worst}(n) = C_W(n/2) + 1$ for $n > 1$ $\rightarrow$ ①

Also $C_W(1) = 1$ $\rightarrow$ ②

Assume $n = 2^k$ then equation ① becomes

as $C_W(2^k) = C_W(2^k/2) + 1$

$\Rightarrow C_W(2^k) = C_W(2^{k-1}) + 1$ $\rightarrow$ ③

Substitute $k = k-1$ in ③ we get

$C_W(2^{k-1}) = C_W(2^{k-2}) + 1$ $\rightarrow$ ④

Substitute ④ in ③ we get

$C_W(2^k) = C_W(2^{k-2}) + 1 + 1$

$= C_W(2^{k-2}) + 2$

Then $C_W(2^k) = C_W(2^{k-3}) + 3$

$\vdots$

$$C_W(2^k) = C_W(2^{k-k}) + k$$

$$= C_W(2^0) + k$$

$$C_W(2^k) = C_W(1) + k$$

$$C_W(1) = 1$$

$$\therefore C_W(2^k) = 1 + k.$$

$\because n = 2^k$

$k = \log_2 n.$

$$C_W(2^k) = 1 + \log_2 n$$

$$\boxed{\therefore C_W(n) = 1 + \log_2 n.}$$

(taking logarithm (base 2) on both sides)

$\log_2 n = \log_2 2^k$

$\log_2 n = k \log_2 2.$

$\log_2(n) = k(1)$

Average case Analysis

The key Compaisons on average case is slightly Smaller than that in the worst Case.

$$C^{yes}_{avg}(n) = \log_2 n - 1$$

$$\& \; C^{no}_{avg}(n) = \log_2 n + 1$$

That is $\boxed{C_{avg}(n) = \log_2 n.}$

Best case: $\boxed{C_{best}(n) = 1.}$

Advantages of binary Search

→ It is an optimal Searching algorithm.
→ Searching is very efficient

Disadvantages:-

→ Requires the list to be sorted.

Applications:-

- * Solving non linear equations with unknown elements.
- * Efficient Search method for record Searching in database.

Explain the Working of Merge Sort Algorithm with an example [Nov/Dec '17] [May 2018]

Merge Sort:-

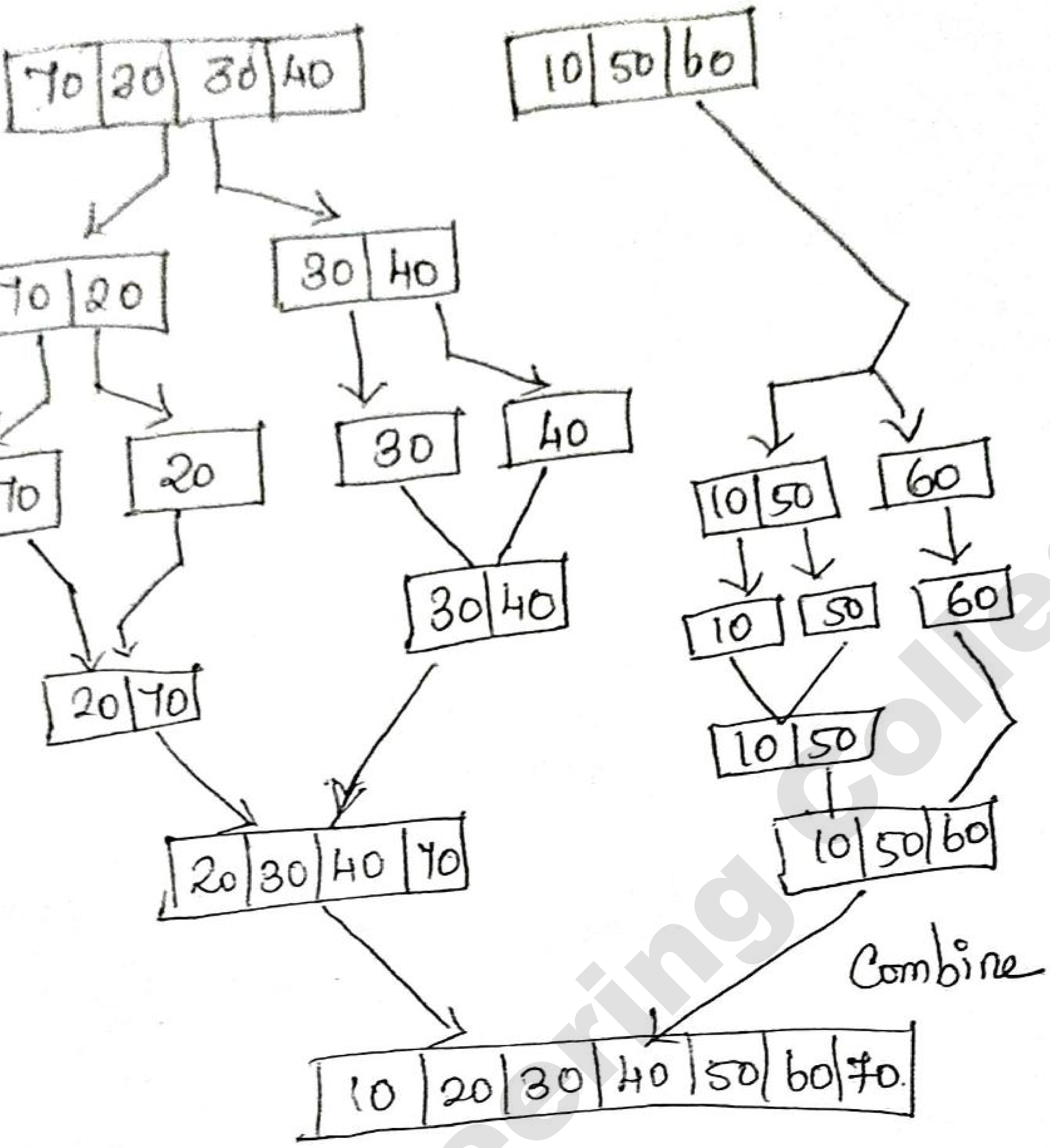
Strategy:-

1. Divides the array into 2 equal halves and Sorts the halves Separately (using recursion), then it merges the sorted halves.

2. Parameters Used for merge are: the array name E, and the first, mid & last indexes of the Subranges it is to merge;
(i.e) 2 sorted Subranges are E[first] ... E[mid], E[mid+1] .... E[last] (after recursion). and final Sorted array is E[First] .... E[last].

eg/.. Consider the elements as

70, 20, 30, 40, 10, 50, 60.
↑ low · ↑ mid · ↑ high.



Analysis:-

In merge sort, algorithm, the two recursive calls are made. Each recursive call focuses on $n/2$ elements of the list. After 2 recursive calls one call is made to combine two sublists (i.e) to merge all n elements. Hence we can write recurrence relation as

$$T(n) = T(n/2) + T[n/2] + Cn \longrightarrow \text{Time taken for Combining 2 Sublists.}$$

$T(n/2)$ ↓ Time taken by left Sublist to get Sorted

$T[n/2]$ ↓ Time taken by right Sublist to get Sorted

where $n > 1$ $(T(1)) = 0$.

Time Complexity :-

Using Substitution Method:-

$T(n) = 2T(n/2) + cn$ for $n>1$

$T(1) = 0$

Substitute $n = 2^k$.

$T(n) = 2T(n/2) + cn$ becomes

$\Rightarrow T(2^k) = 2T(2^k/2) + c.2^k$

$= 2T(2^{k-1}) + c.2^k \rightarrow$ ②

Substitute $k = k-1$ in ② we get

$T(2^{k-1}) = 2T(2^{k-2}) + c.2^{k-1} \rightarrow$ ③

Now ③ in ②

$T(2^k) = 2[(2T(2^{k-2}) + c.2^{k-1}] + c.2^k$

$= 2^2.T[2^{k-2}] + 2.2^{k-1}.c + c.2^k$

$= 2^2.T[2^{k-2}] + \cancel{2}.c.\frac{2^k}{\cancel{2}} + c.2^k$

$T(2^k) = 2^2.T[2^{k-2}] + 2c.2^k$

$= 2^3.T[2^{k-3}] + 3.c.2^k$

$= 2^4.T[2^{k-4}] + 4.c.2^k$

$\vdots$

$= 2^k.T[2^{k-k}] + k.c.2^k$

$= 2^k.T[2^0] + k.c.2^k$

$= 2^k T[1] + k \cdot c \cdot 2^k$

$= 0 + k \cdot c \cdot 2^k$

$\therefore T_W(2^k) = k \cdot c \cdot 2^k$

$= \log_2 n \cdot c \cdot n \qquad \because \log_2 n = k$

$T_W(n) = n \log_2 n.$

$\therefore$ Best case, worst case & ~~Average case of~~ Time Complexity for merge sort is $(n \log_2 n)$

Write the algorithm for Quick Sort and write the time complexity with example [APR/MAY '17].

Quick Sort

→ Uses divide and Conquer strategy. In this method division is carried out dynamically.

→ 3 steps of Quick Sort are

1) Divide : Split the array into 2 Sub arrays that each element in the left Sub array is less than or equal the middle element & each element in the right Sub array is greater than the middle element. Splitting is based on the pivot element.

2) Conquer: Recursively sort the 2 Sub arrays

3) Combine : Combine all the Sorted elements in a group to form a list of Sorted elements.

→ Merge sort – division of array is based on the positions of array elements

→ Quick Sort – division is based on the actual value of the element.

$$\underbrace{A[0] \ldots A[m-1]}_{\text{Elements} < A[m]},\ \underbrace{A[m]}_{\text{Pivot}},\ \underbrace{A[m+1] \ldots A[n-1]}_{\text{elements} > \text{pivot}}$$

<u>example</u>:- Consider the following elements

50, 30, 10, 90, 80, 20, 40, 70 → High

(1) ↓ Low   i/pivot.   ↓ j.

let i/pivot = 50

(2) if A[i] ≤ pivot after incrementing i, Continue incrementing i until an element pointed by i satisfies A[i] ≥ A[Low].

(3) j indicates high or last value. if A[j] > pivot, decrement j, this Continues until the element pointed by j < A[low]

(4) if A[j] < A[LOW], Swap A[i] and A[j].

(5) Repeat the process of step 2, 3, 4

(6) if A[j] < A[LOW] and j crosses i, That is j < i, we Swap A[LOW] and A[j].

Logic Explanation with example

⇒ 50 30 10 90 80 20 40 70
Pivot/i (under 50) j (under 70)

⇒ 50 30 10 90 80 20 40 70
i (under 30) j (under 70)
∴ increment i.

⇒ 50 30 10 90 80 20 40 70
i (under 90) j (under 70)
i = 90 > pivot ∴
Stop increasing i

⇒ 50 30 10 90 80 20 40 70
i (under 90) j (under 40)
j = 70 > pivot
decrement j

⇒ 50 30 10 40 80 20 90 70
i (under 40) j (under 90)
∴ 40 < pivot
Stop decrement j

⇒ 50 30 10 40 80 20 90 70
i (under 80) j (under 20)
∴ j = 90 > pivot
i = 40 < pivot

⇒ 50 30 10 40 20 80 90 70
j (under 20) i (under 80)
Swap.

⇒ 20 30 10 40 50 80 90 70
j (under 50) i (under 80)
∴ j crossed i.

New pivot = 20 80 90 70.
⇒ 20 30 10 40 50
i (under 30) j (under 40)

⇒ 20 30 10 40 50 80 90 70
i (under 30) j (under 10)

Low
⇒ 20 10 30 40 50 80 90 70
Pivot i j

Repeat before steps

⇒ 20 10 30 40 50 80 90 70
i,j (under 30)

⇒ 20 10 30 40 50 80 90 70
j (under 10) i (under 30)

$$\Rightarrow \underset{\longleftrightarrow}{10} \quad \underset{\underset{\text{Pivot}}{|}}{20} \quad \underbrace{30 \quad 40} \quad \boxed{50}\boxed{80 \quad 90 \quad 70}$$

go to right sublist of 1st step

Pivot (over 80)
⇒ 10 20 30 40 50 80 90 70
i (under 90) j (under 70)

⇒ 10 20 30 40 50 80 70 90
p (under 80) i (under 70) j (under 90)

⇒ 10 20 30 40 50 80 70 90
p (under 80) i,j (under 90)

⇒ 10 20 30 40 50 80 70 90
p (under 80) j (under 70) i (under 90)

$$\Rightarrow 10 \quad 20 \quad 30 \quad 40 \quad 50 \quad 70 \xleftarrow{} \underset{\underset{p}{|}}{80} \xrightarrow{} 90.$$

Hence the lis is sorted.

Analysis

Best case (split in the middle)

Complexity: The RR for QS for obtaining best case time is

$$C(n) = C(n/2) + C(n/2) + n.$$

and $C(1) = 0$

$$\therefore C(n) = 2C(n/2) + n.$$

$n = 2^k$ $\quad C(2^k) = 2C(2^k/2) + 2^k \rightarrow$

$$= 2C(2^{k-1}) + 2^k \Rightarrow ①$$

$$\Rightarrow C(2^{k-1}) = 2C(2^{k-2}) + 2^{k-1} \rightarrow ②$$

Substitute ② in ① we get

$$C(2^k) = 2\left(2C(2^{k-2}) + 2^{k-1}\right) + 2^k.$$

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

$$= 2^2 C(2^{k-2}) + \cancel{2} \cdot \frac{2^k}{\cancel{2}} + 2^k$$

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

$$\Downarrow$$

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

$$\vdots$$

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

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

$$= 2^k \cdot 0 + k \cdot 2^k$$

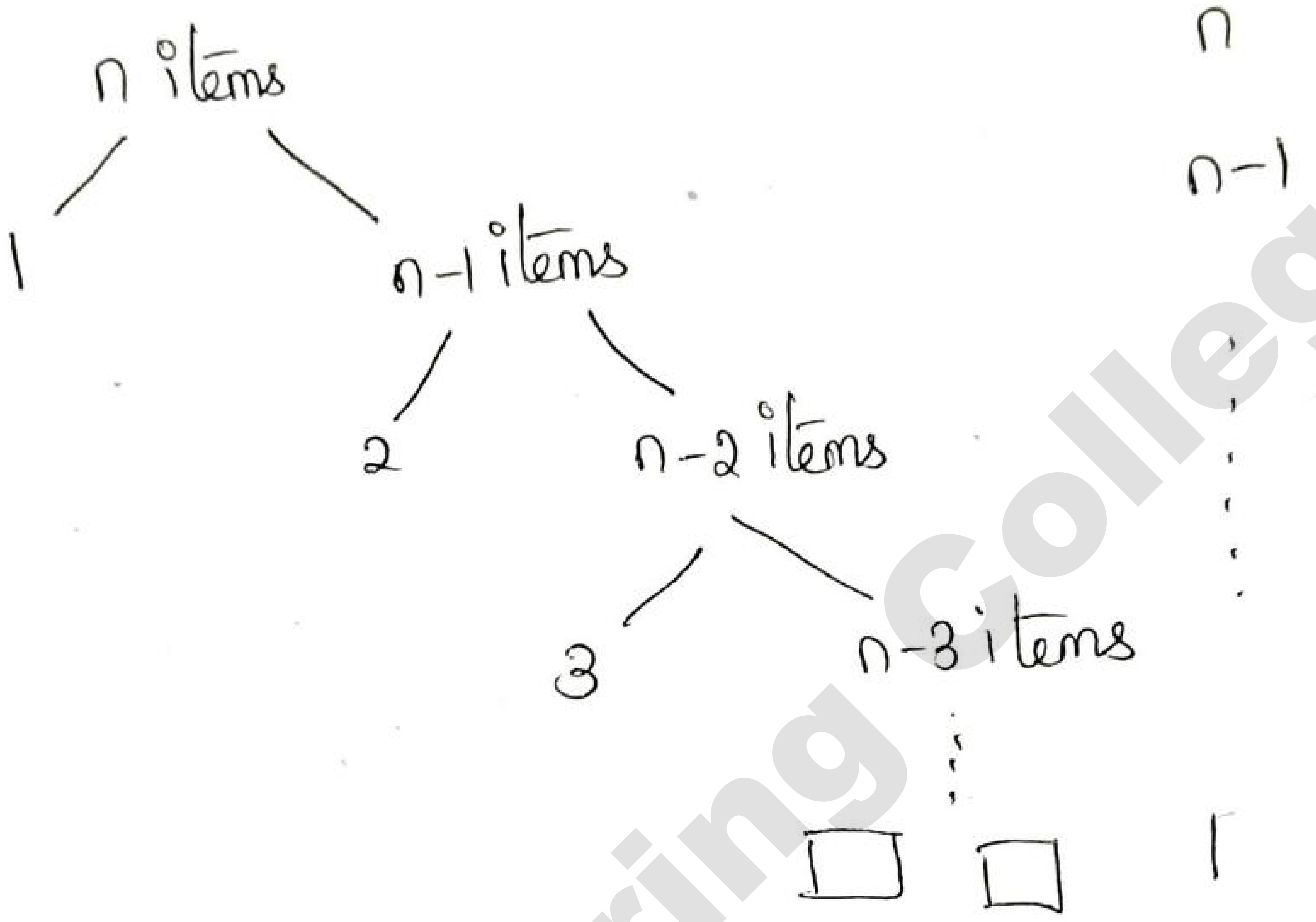
$$\therefore C(2^k) = k \cdot 2^k$$

$$\therefore C(n) = n \log_2 n.$$

$$\boxed{\therefore C_{best} = n \log_2 n.}$$

Worst Case (Sorted array)

Worst case for Quick Sort occurs when the pivot is minimum or maximum of all the elements in the list.



we can write it as

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

$$\Rightarrow C(n) = n + (n-1) + (n-2) + \cdots + 2 + 1$$

∴ we know that

$$1+2+3+\cdots+n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n.$$

$$\boxed{\therefore C_{worst} = n^2/2}$$

Average Case (random array)

Here we assume either (i) the array to be partitioned is randomly ordered or (ii) the pivot element is selected from a random position in the array.

The probability that the pivot element is the k^{th} largest element of the array is $1/n$

In the recurrence ,

$$C(n) = n + C(k-1) + C(n-k),\ C(0) = C(1) = 0 .$$

all values of k are equally likely. we must average over all k.

$$C(n) = (1/n)\sum_{k=1}^{n} (n + C(k-1) + C(n-k))$$

$$= n + (1/n)\sum_{k=1}^{n} C(k-1) + (1/n)\sum_{k=1}^{n} C(n-k)$$

Note :- $\sum_{k=1}^{n} C(k-1) = \sum_{i=0}^{n-1} C(i)$ by Substituting $i = k-1$

$\sum_{k=1}^{n} C(n-k) = \sum_{i=0}^{n-1} C(i)$ by Substituting $i = n-k$.

So our recurrence becomes

$$C(n) = n + 2/n \sum_{i=0}^{n-1} C(i) \quad \text{or}$$

$$n\,C(n) = n^2 + 2\sum_{i=0}^{n-1} C(i). \quad \to ①$$

replace $n = n-1$ we get

$$(n-1)\,C(n-1) = (n-1)^2 + 2\sum_{i=0}^{n-2} C(i) \to ②$$

Subtracting ② from ① we get

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

$$\Rightarrow n\,C(n) = (n+1)\,C(n-1) + 2n - 1$$

Dividing by $n(n+1)$ gives

$$C(n)/(n+1) = [C(n-1)/n + (2n-1)/n(n+1)]$$

$$\Rightarrow C(n)/(n+1) = C(n-1)/n + 2/n.$$

$$C_{avg} \doteqdot 1.39\, n \lg_2(n).$$

Multiplication of large integers :-

Applications :-

- * High precision Computing
- * PC 64 bit Computation
- * weight of Neutrino 100 decimal digits
- * To represent Square root of a number.
- * pi (π) for million digits
- * RSA primes with 1000's of bit long.

The efficiency of grade-school multiplication algorithm is n^2 1digit multiplications.

Our algorithm is based on using divide and Conquer to split an n-digit integer into 2 integers of approximately $n/2$ digits.

eg/.. $567,832 = 567 \times 10^3 + 832$

In general

$$\underset{n\text{ digits}}{u} = \underset{(n/2)\text{ digits}}{x} \times 10^m + \underset{(n/2)\text{ digits}}{y}$$

where $m = n/2$

if we have 2 n-digit numbers

$$u = x \times 10^m + y$$

$$v = w \times 10^m + z$$

their product is given by

$$uv = (x \times 10^m + y)(w \times 10^m + z)$$

$$= xw \times 10^{2m} + (xz + wy) \times 10^m + yz$$

eg/..

$$567{,}832 \times 9{,}423{,}723 = (567 \times 10^3 + 832)(9423 \times 10^3 + 723)$$

$$= 567 \times 9423 \times 10^6 + (567 \times 723 + 9423 \times 832) \times 10^3 + 832 \times 723$$

These smaller integers can then be multiplied by dividing them into yet smaller integers. This division process is continued until a threshold value is reached, at which time the multiplication can be done in the standard way.

Analysis

This algorithm involves 4 times of $n/2$ multiplications. Hence RR Equation is

$T(n) = 4T(n/2) + cn$ for $n > s$, n is a power of 2.

$W(s) = 0$.

The efficiency is $\Theta(n^2)$

In the above algorithm, we need to perform 4 multiplications, An improvement to the above algorithm can be done, which performs only 3 multiplications.

(i.e) $r = (x+y)(w+z) = xw + (xz + yw) + yz$

then

$xz + yw = r - xw - yz$

$r = (x+y)(w+z)$, xw and yz.

Analysis

$M(n) = 3M(n/2)$ for $n>1$, $M(1)=1$

Solving it by backward Substitutions for $n=2^k$ yields

$$M(2^k) = 3M(2^{k-1})$$
$$= 3[3M(2^{k-2})]$$
$$= 3^2 M(2^{k-2})$$
$$\vdots$$
$$= 3^k M(2^0)$$
$$= 3^k$$

$k = \log_2 n$

$\therefore M(n) = 3^{\log_2 n} = n^{\log_2 3} = n^{1.58}$.

eg/..

Let

$a = a_1 a_0$ and $b = b_1 b_0$

$a = a_1 a_0$ implies that $a = a_1 10^{n/2} + a_0$

$b = b_1 b_0$ implies that $b = b_1 10^{n/2} + b_0$

$$c = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$
$$= (a_1 * b_1) 10^n + (a_1 * b_0 + (a_0 * b_1)) 10^{n/2} + (a_0 * b_0$$
$$= c_2 10^n + c_1 10^{n/2} + c_0.$$

where

$c_2 = a_1 * b_1$ is the product of their 1st halves

$c_0 = a_0 * b_0$ is the product of their 2nd halves

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the Sum of the a's halves & the Sum of the b's halves (−) the Sum of c_2 & c_0.

eg/.. $2101 * 1130$

$C_2 = 21 * 11$

$C_0 = 01 * 30$

$C_1 = (21+01) * (11+30) - (C_2 + C_0) = 22 * 41 - (21 * 1) - (01 * 3$

for $(21 * 11)$

$C_2 = (2 * 1) = 2$

$C_0 = (1 * 1) = 1$

$C_1 = (2+1) * (1+1) - (2+1) = 3 * 2 - 3 = 6 - 3 = 3$

So, $21 * 11 = 2 \cdot 10^2 + 3 \cdot 10^1 + 1 = 231$

for $(01 * 30)$

$C_2 = (0 * 3) = 0$

$C_0 = (1 * 0) = 0$

$C_1 = (0+1) * (3 * 0) - (0+0) = 1 * 3 - 0 = 3$

So, $01 * 30 = 0 \cdot 10^2 + 3 \cdot 10^1 + 0 = 30$

for $(22 * 41)$

$C_2 = 2 * 4 = 8$

$C_0 = 2 * 1 = 2$

$C_1 = (2+2) * (4+1) - (8+2) = 4 * 5 - 10 = 10$

So, $22 * 41 = 8 \cdot 10^2 + 10 \cdot 10^1 + 2 = 902$

Hence

$2101 * 1130$

$= 231 \cdot 10^4 + (902 - 231 - 30)\ 10^2 + 30$

$= 2,374,130.$ //

```
2310000
  64100
     30
-------
2,374,130
```

Strassen's Matrix Multiplication [Apr/May 2018]

General Matrix Multiplication

The time Complexity of its number of multiplications for matrix multiplication is by $T(n)=n^3$ where n is the number of rows & columns in the matrices. we can also analyze the number of additions.

The time Complexity including number of additions is given by $T(n) = n^3 - n^2$.

1. The divide & Conquer approach can reduce the number of multiplications.
2. Suppose we want the product C of 2 2×2 matrices A & B (i.e)

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

where we are going to use D&C method we get

$$\begin{array}{cc} & \xleftrightarrow{n/2} \\ n/2 \updownarrow & \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] \end{array} = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]$$

where

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,n/2} \\ a_{21} & a_{22} & \cdots & a_{2,n/2} \\ \vdots & \vdots & & \\ a_{n/2,1} & \cdots & \cdots & a_{n/2,n/2} \end{bmatrix}$$ is a $n/2 * n/2$ matrix.

The Complexity analysis is as follows:

1. Input size : Matrix $n*n$
2. Basic operation : Multiplication / Addition
3. Number of Multiplication is

$$C_{11} = (A_{11} * B_{11}) + (A_{12} * B_{21})$$
$$C_{12} = (A_{11} * B_{12}) + (A_{12} * B_{22})$$
$$C_{21} = (A_{21} * B_{11}) + (A_{22} * B_{21})$$
$$C_{22} = (A_{21} * B_{12}) + (A_{22} * B_{22})$$

So, totally 8 times of $n/2 * n/2$ multiplications & 4 times of $n^2/4$ additions are performed.

$$T(n) = 8T(n/2) + 4T(n^2/4)$$

Algorithm:-

```
{
for i=1 to n do
  for j=1 to n do
    c[i,j]=0;
      for k:=1 to n do
        c[i,j]= c[i,j]+ A[i,k] x B[k,j];
}
```

So complexity is $\Theta(n^3)$.

Strassen showed that 2×2 matrix multiplication can be accomplished in 7 multiplication & 18 additions / subtractions.

The divide & Conquer approach can be used for implementing Strassen's matrix multiplication.

- * Divide - Divide matrices into Sub-matrices: $A_0, A_1 \ldots A_n$ etc..
- * Conquer - Use a group of matrix multiply equations
- * Combine - Recursively multiply Sub-matrices and get the final result of multiplication after performing required additions or Subtractions.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$S_1 = (A_{11} + A_{22}]\ (B_{11} * B_{22}]$$
$$S_2 = (A_{21} + A_{22}) \times B_{11}$$
$$S_3 = A_{11} \times (B_{12} - B_{22})$$
$$S_4 = A_{22} \times (B_{21} - B_{11})$$
$$S_5 = (A_{11} + A_{12}) \times B_{22}$$
$$S_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$$
$$S_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$
$$C_{11} = S_1 + S_4 - S_5 + S_7$$
$$C_{12} = S_3 + S_5.$$
$$C_{21} = S_2 + S_4$$
$$C_{22} = S_1 + S_3 - S_2 + S_6.$$

Now, we will Compare the actual / traditional matrix multiplication Procedure with Strassen's procedure.

$$C_{11} = S_1 + S_4 - S_5 + S_7$$

$$= (A_{11}+A_{22})(B_{11}+B_{22}) + A_{22}\times(B_{21}-B_{11}) - (A_{11}+A_{12}) * B_{22} + (A_{12}-A_{22})\times(B_{21}+B_{22})$$

$$= A_{11}B_{11} + \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + \cancel{A_{22}B_{22}} + \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{11}} - \cancel{A_{11}B_{22}} + A_{12}B_{21} - \cancel{A_{12}B_{22}} + \cancel{A_{12}B_{22}} - \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{22}}$$

$$\boxed{C_{11} = A_{11}B_{11} + A_{12}B_{21}}$$

The recurrence equation for Strassen's matrix multiplication is

$$T(n) = 7T(n/2) + 18(n/2)^2 \text{ for } n>1,\ n \text{ is a power of } 2$$

$$T(1) = 0$$

Solving the equation by backward Substitution method

we get

$$T(2^k) = 7T(2^k/2)$$

$$= 7T(2^{k-1})$$

$$= 7^2 T(2^{k-2})$$

$$\vdots$$

$$= 7^k T(2^{k-k})$$

$$= 7^k$$

$$\boxed{T(n) = 7^{\log_2 n} = n^{\log_2 7} = n^{2.8}}$$

Example:-

Multiply the following Matrix

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

Using Divide and Conquer strassen's technique

$$A_{00} = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} \quad A_{01} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \quad A_{10} = \begin{bmatrix} 0 & 1 \\ 5 & 0 \end{bmatrix}$$

$$A_{11} = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}$$

$$B_{00} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} \quad B_{01} = \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix} \quad B_{10} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}$$

$$M_1 = (A_{00} + A_{11})(B_{00} + B_{11}) = \begin{bmatrix} 4 & 0 \\ 6 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 7 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix}$$

$$M_2 = (A_{10} + A_{11})(B_{00}) = \begin{bmatrix} 3 & 1 \\ 7 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix}$$

$$M_3 = A_{00}(B_{01} - B_{11}) = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ -5 & 4 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix}$$

$$M_4 = A_{11}(B_{10} - B_{00}) = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} = \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix}$$

$$M_5 = (A_{00} + A_{01})B_{11} = \begin{bmatrix} 3 & 1 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix}$$

$$M_6 = (A_{10} - A_{00})(B_{00} + B_{01}) = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 2 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix}$$

$$M_7 = (A_{01} - A_{11})(B_{10} + B_{11}) = \begin{bmatrix} -1 & 1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix}$$

$$C_{00} = M_1 + M_4 - M_5 + M_7$$
$$= \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix} - \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}$$

$$C_{01} = M_3 + M_5$$
$$= \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} + \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix} = \begin{bmatrix} 7 & 3 \\ 1 & 9 \end{bmatrix}$$

$$C_{10} = M_2 + M_4$$
$$= \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 1 \\ 5 & 8 \end{bmatrix}$$

$$C_{11} = M_1 + M_3 - M_2 + M_6$$
$$= \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix}$$
$$= \begin{bmatrix} 3 & 7 \\ 7 & 7 \end{bmatrix}$$

$$C = \begin{bmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 9 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 7 & 7 \end{bmatrix}$$

Convex hull problem

Let S be a set of points in the plane

Definition :- The Convex hull of S is the Smallest convex polygon that Contains all the points of S.

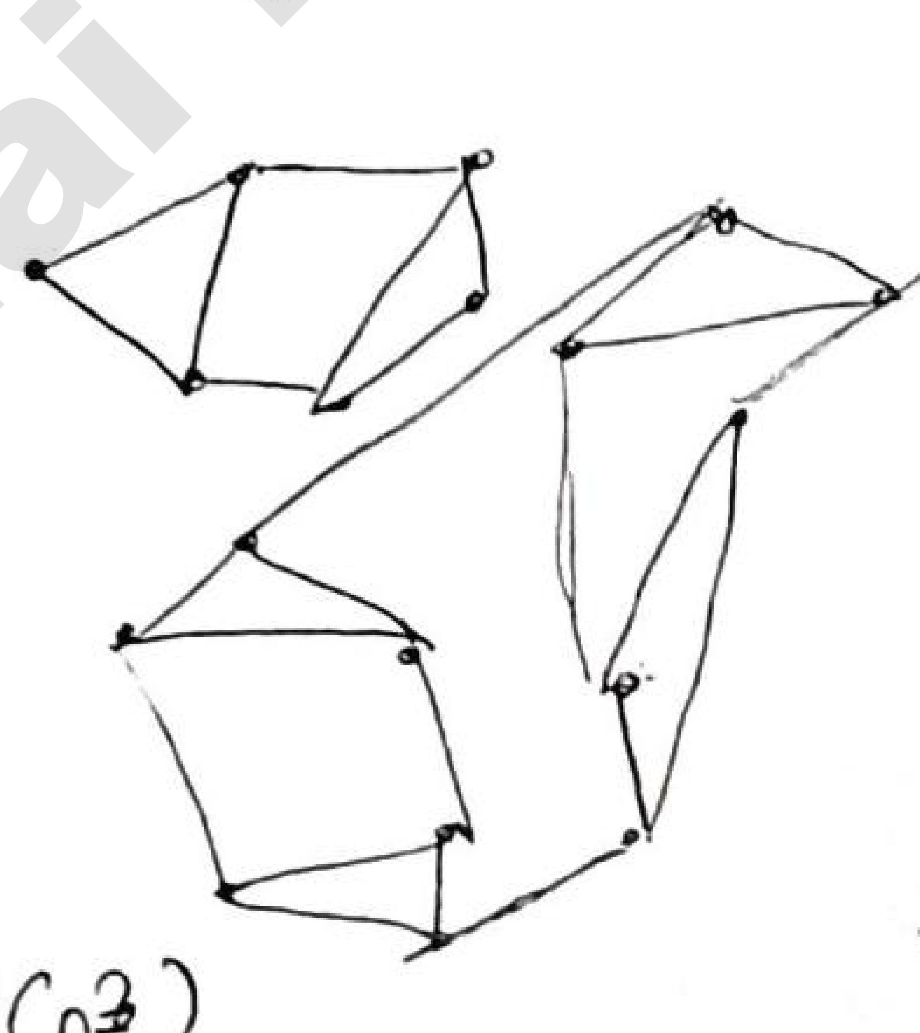
Algorithm:-

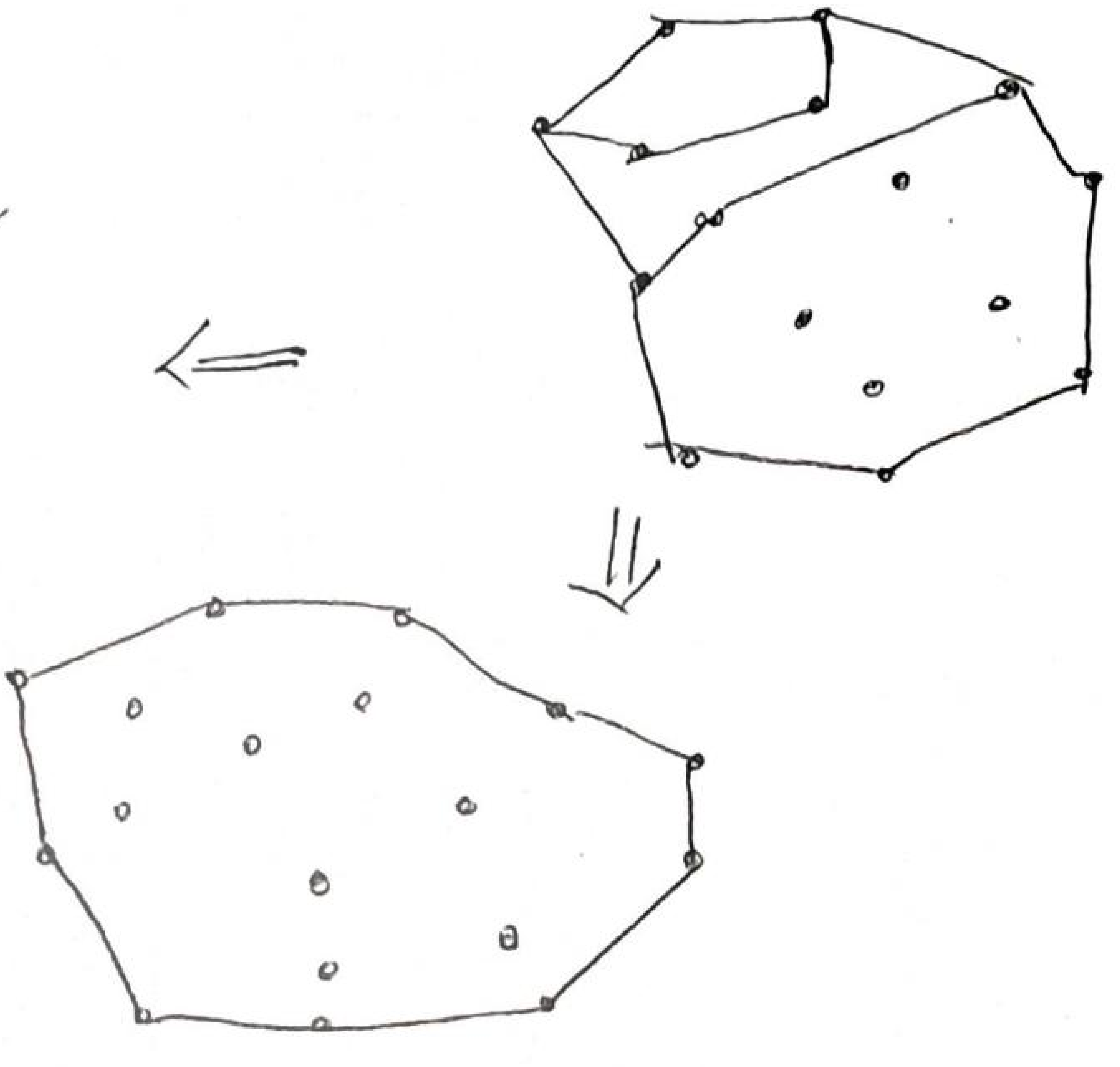
1. Assume points are Sorted by x-coordinate values. If Sorted by Quick Sort, it is called quick hull.

Identify extreme points P_1 and P_2

2. Compute upper hull recursively:
   - → find point P_{max} that is farthest away from line $P1\ P_2$
   - → Compute the upper hull of the points to the left of line P_1P_{max}
   - → Compute the upper hull of the points to the left of line $P_{max}P_2$
3. Compute lower hull in a Similar manner.
4. Merge the 2 Convex hulls.

eg/..





$C_W = \Theta(n^3)$

$C_{avg} = n(\log n)$.

Explain the Brute force method to find the 2 closest Points in a set of n Points in k-dimensional space [Nov/Dec '17]

CLOSEST PAIR PROBLEM:-

→ It is to find the 2 closest points in a set of n Points.

→ Using brute-force algorithm we get

for closest pair problem $- O(n^2)$ time.

for Convex hull problem $- O(n^3)$ time

Applications

→ Used in Computational geometry that deals with proximity of Points in the plane or higher-dimensional Spaces.

→ Air traffic Control

→ post offices.

→ DNA Sequences...

One-dimensional case of closest pair problem

⇒ It can be solved in $O(n \log n)$ via Sorting $(x_1, x_2 \ldots x_5)$ and finding the shortest distance b/w 2 consecutive points.

Two dimensional Case of closest pair problem

If $2 \le n \le 3$, the problem can be solved by brute force algorithm. If $n>3$, we can divide the points into 2 subsets P_l & P_2 of $n/2$ & $n/2$ points, Let d_l and d_r be the Smallest distances b/w Pairs of points in P_l & P_r respectively

$$\text{let } d = \min\{d_1, d_2\}.$$

Note that d is not necessarily the smallest distance b/w all the point pairs, because points of a closest pair can lie on the opposite sides of the separating line.

Algorithm

Input : A set S of n planar points

Output : distance b/w two closest points

Step 1 : Divide the points given into 2 subsets S_1 and S_2 by a vertical line $x = c$ so that half the points lie to the left or on the line and half the points lie to the right or on the line.

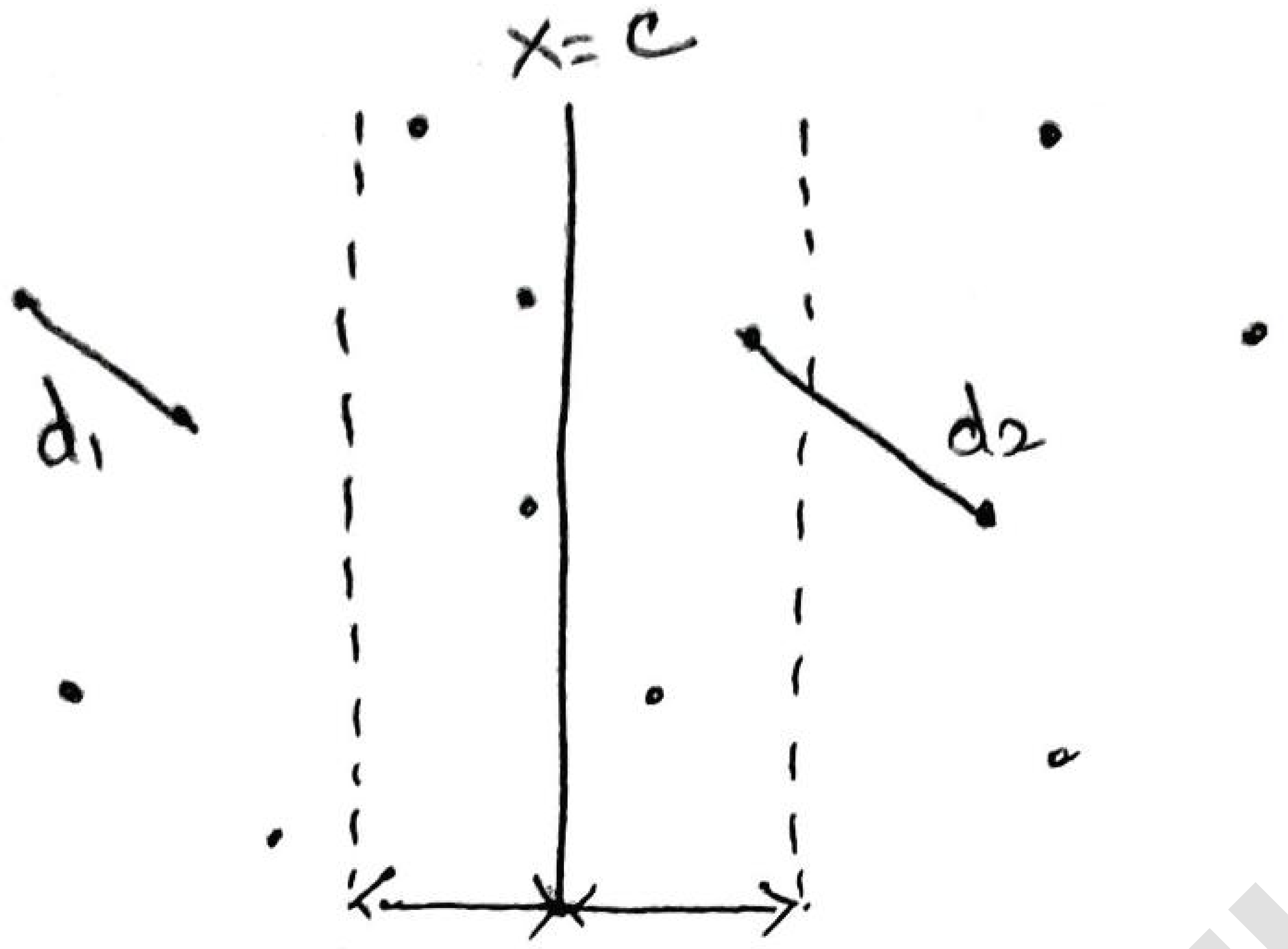
Step 2 : find recursively the closest pair of the left & right subsets.

Step 3 : Set $d = \min\{d_1, d_2\}$

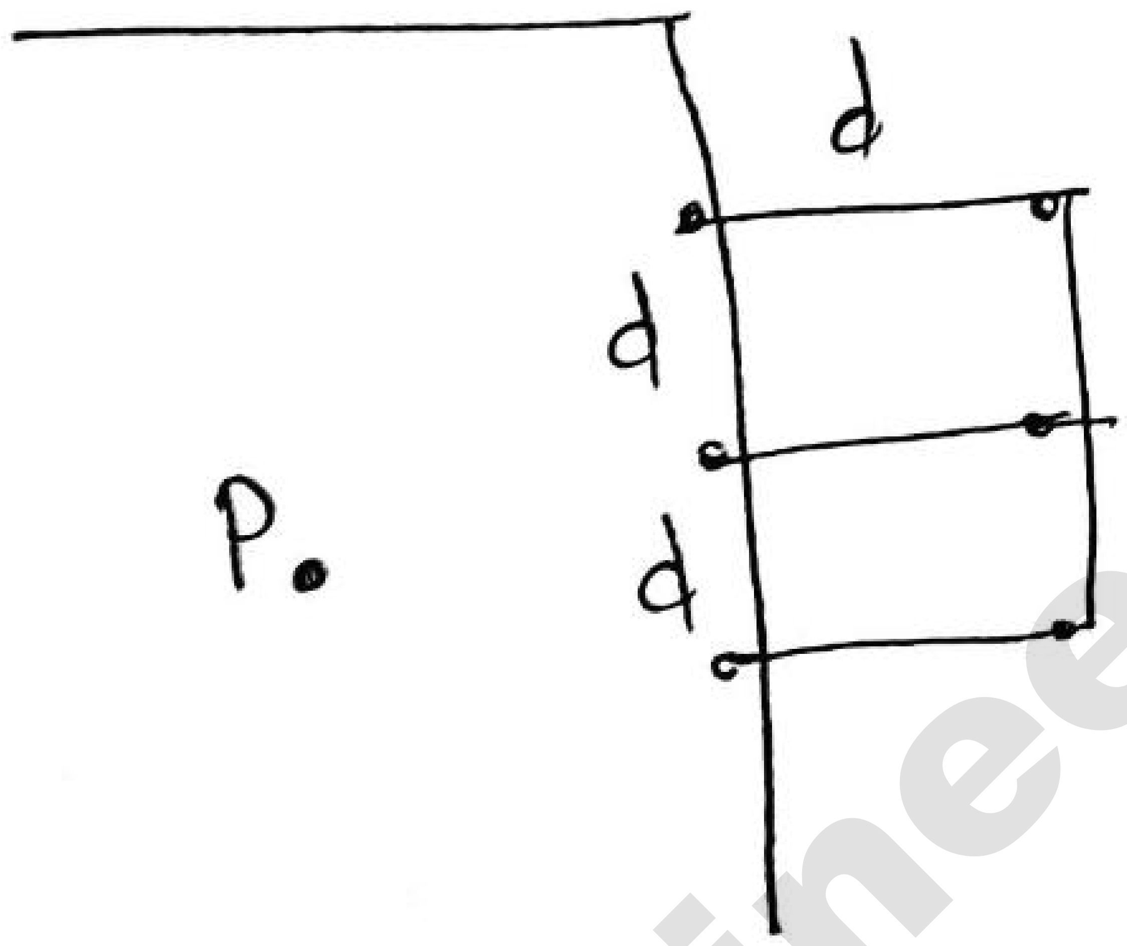
So that we can limit our attention to the points in the symmetric vertical strip of width 2d as possible closest pair. Let C1 & C2 be the subsets of points in the left subset S1 and the right subset S2, that lie in this vertical strip.

→ The points in C1 & C2 are stored in increasing order of their y coordinates, which is maintained by merging during the execution of the next step.

Step 4 : for every point $P(x,y)$ in C1, we inspect points in C2 that may be closer to P than d. There can be no more than 6 such points.



Worst Case Scenario



Analysis

Running time of the algorithm is described by

$T(n) = 2T(n/2) + M(n)$ where $M(n) \in O(n)$

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

Brute force:-

Computing a^n:-

* Computing a^n ($a>0$, n a non-negative integer) based on the definition of exponentiation

$$a^n = a * a * a \, . \, . \, . \, * a$$

* Brute force algorithm requires $n-1$ multiplications

* Recursive algorithm for the same problem, based on the observation that

$$a^n = a^{n/2} * a^{n/2}$$

requires $\Theta(\log(n))$ operations.

String Matching - Brute force

Concept:-

Given a string of n characters called the text and a string of m characters ($m \leq n$) called the pattern, find a substring of the text that matches the pattern.

* To put it more precisely, we want to find i - the index of the leftmost character of the first matching substring in the text - such that

$$\begin{array}{cccccccccl} t_0 & \ldots & t_i & \ldots & t_{i+j} & \ldots & t_{i+m-1} & \ldots & t_{n-1} & \text{text } T \\ & & \updownarrow & & \updownarrow & & \updownarrow & & & \\ & & p_0 & \ldots & p_j & \ldots & p_{m-1} & & & \text{Pattern } P. \end{array}$$

* If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

Working of brute-force algorithm

→ Align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m-pairs of the characters match then the algorithm can stop or a mismatching pair is encountered.

→ In the latter case, shift the pattern one position to the right & resume the character comparisons, starting again with the first character of the pattern & its counterpart in the text.

→ The last position in the text that can still be a beginning of a matching substring is n-m provided the text positions are indexed from 0 to n-1.

→ Beyond that position, there are not enough characters to match the entire pattern; hence the algorithm need not make any comparisons there.

```
BruteForceStringMatch (T[0...n-1], P[0...m-1])
  // Implements brute-force string matching
  // Input : An array T[0...n-1] of n characters representing a text
    and an array P[0...m-1] of m characters representing a pattern
  // output : Index of the first character in the text that starts a
    matching Substring or -1 if the Search is unSuccessful.
      for i ← 0 to n-m do
        j ← 0
        while j < m and P[j] = T[i+j] do
        j ← j+1
        if j = m return i
        return -1
```

✓ The wost Case is much worse : the algorithm may have to make all the m Comparisons before shifting the pattern , and this can happen for each of the n-m +1 tries. Thus, in the wost case, the algorithm makes m(n-m +1) character Comparisons, which Puts it in the O(nm) class.

```
NOBODY_NOTICED_HIM
NOT
 NOT
  NOT
   NOT
    NOT
```

✓ for a typical word search in a natural language text, should expect that most shifts would happen after very few comparisons.

✓ Average-case efficiency should be considerably better than the worst-case efficiency.

✓ Indeed it is: for searching in random texts, it has been shown to be linear, i.e., Θ(n). There are several more sophisticated and more efficient algorithms for string searching.

eg:- Rabin-karp algorithm, Knuth Morris algorithm (KMP).

✓ Also known as substring search or pattern matching algorithm.

UNIT

III

UNIT-III

0/1 knapsack problem

Statement:-

It states that given n items of known weights $w_1, w_2 \ldots w_n$ and the values $v_1 \ldots v_n$ and a knapsack of capacity W. Find the most valuable subset of items that fit into the knapsack.

In dynamic programming we need to obtain the solution by solving the smaller subinstances.

Strategy

Let $V[i, j]$ be the values of an optimal solution to the instance (i.e) the value of the most valuable subset of the 1st i items that fit into the knapsack of capacity j. We can divide the subsets into those that do not include the ith item & those that do.

2 : Those that do not include the ith item.

1. Among the subsets that do not include the ith item, the value of an optimal subset is, by definition : $V[i-1, j]$

2. Among the subsets that do not include the ith item, an optimal subset is made up of this item and an optimal subset of the 1st i-1 items that fit into the knapsack of capacity $j-w_1$. The value of such an optimal subset is $v_i + V(i-1, j-w_1)$.

Thus the value of an optimal solution among all feasible subsets of the 1st i items is the maximum of these 2 values.

Of course, if the ith item does not fit into the knapsack, the value of an optimal subset selected from the 1st i items is the same as the value of an optimal subset selected from the 1st i-1 items.

$$\therefore V[i,j] = \begin{cases} \max\{V[i-1,j], v_i + V[i-1, j-w_i]\} & \text{if } j-w_i \geq 0 \\ V[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

The initial conditions are

$V[0,j]=0$ for $j \geq 0$ and value $V[i,0]=0$ for $i \geq 0$

Our goal is to find $v(n,w)$, the maximal value of a subset of n items which fit into w.

[APR/MAY '17].

eg/.. Given knapsack capacity = 5

Items	weight	Value
1	4	10
2	3	20
3	2	15
4	5	25

Solve the above using dynamic programming.

Solution :-

1. Define initial condition & its corresponding table.

$V[0,j]=0$ for $j \geq 0$

$V[i,0]=0$ for $i \geq 0$

Capacity j

	i	0	1	2	3	4	5
0(0)	0	0	0	0	0	0	0
4(10)	1	0	0	12	12	10	10
3(20)	2	0	0	0	20	20	20
2(15)	3	0	0	15	20	20	35
5(25)	4	0	0	15	20	20	35

2. The recurrence relation to fill the remaining entries for $i, j > 0$, Compute the entry in the i^{th} row & j^{th} column.

$$V[i,j] = \begin{cases} \max\{V[i-1,j], v_i + V[i-1, j-w_i]\} & \text{if } j-w_i \geq 0 \\ V[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

Table can be filled either row by row (or) Column by column

V(n, w) is the goal & optimal Subset of the knapsack Problem.

Row 1 : $w_1 = 2$ $V_1 = 12$ $w_i = 4$, $v_i = 10$

$V[1,1] = j - w_i = 1-2 = -1 < 0$

$= V[i-1, j]$

$= V[0,1]$

$= 0$

$V[1,2] = 2-2 = 0$

$= \max\{V(0,2), 12 + V[0,0]\}$

$= \max\{0, 12\}$

$= 12.$

$V[1,3] = 3-2 = 1 \geq 0$

$= \max\{V[0,3], \cancel{v_i} V[0,1]\}$

$= \max\{0, 12\}$

$= 12$

$V[1,4] = \max\{0,12\}$

$= 12$

$V[1,5] = \max\{0,12\}$

$= 12.$

Row 2 : $w_i = 3$ $v_i = 20$

$V[2,1] = 1-3 = -2 < 0$

$= 0$

$V[2,2] = -1 < 0 = 0$

$V[2,3] = \max(V(1,0), 20 + V(1,0))$

$= \max(0, 20)$

$= 20$

$V[2,4] = 1 > 0$

$= \max\{V(i-1, j), v_i + V(i-1, j-w_i)\}$

$= \max\{V(1,4), 20 + V(1,1)\}$

$= \max\{10, 20\}$

$= 20$

$V[2,5] = 2 > 0$

$= \max\{V(1,5), 20 + V(1,2)\}$

$= \max\{10, 20\}$

$= 20$

Row 3 : $w_i = 2$ $v_i = 15$

$V[3,1] = -1 < 0$

$= V[i-1, j]$

$= V[2,1]$

$= 0$

$V[3,2] = 0 \geq 0$

$= \max[V(2,2), 15 + V(2,0)]$

$= 15$

$V[3,3] = 1 \geq 0$

$= \max(V(2,3), 15 + V(2,1))$

$= \max(20, 15 + 0)$

$= 20$

$V[3,4] = 2 \geq 0$

$= \max(V(2,4), 15 + V(2,2))$

$= \max(20, 0)$

$= 20$

$V[3,5] = 3 \geq 0$

$= \max(V(2,5), 15 + V(2,3))$

$= \max(20, 15 + 20)$

$= 35$

Row 4:- $w_i = 5$ $v_i = 25$

$V[4,1] = -4 \leq 0$
$= V[i-1, j]$
$= V[3,1]$
$= 0$

$V[4,2] = -3 \leq 0$
$= V[3,2]$
$= 15$

$V[4,3] = -2 \leq 0$
$= V[3,3]$
$= 20$

$V[4,4] = -1 \leq 0$
$= V[3,4]$
$= 20$

$V[4,5] = 0$
$= \max(V(3,5), 25 + V(3,0))$
$= \max(35, 25)$
$= 35$ //

To find the subset of items for the profit 35 is

$\Rightarrow V[3,5] \neq V[2,5]$
So item 3 is included.
Remaining Capacity = 3

$\Rightarrow V[2,3] \neq V[1,3]$ therefore item 2 is included.
Remaining Capacity $= 3 - 3 = 0$.

Solution is = {0, 1, 1, 0}

The final solution is
{item 3, item 2} = {0, 3, 2, 0} = {20, 15} = 35 //

Complexity:-

Time & Space efficiency of this algorithm are both in $O(nW)$. The time needed to find the composition of an optimal solution is $O(n+W)$.

[Apr/May 2018]

Memory functions

Need of memory function

The solution of knapsack problem is computed in a top-down manner. Generally, dynamic programming follows overlapping subproblems with bottom approach. But the values which are not required in the computation of subset is also computed. To overcome that, the merits of topdown & bottomup are combined together in memory function.

Approach

Problem is solved in topdown manner, but the table is filled as a kind of bottom up dynamic programming approach.

Algorithm

```
Algorithm MFknapsack(i,j)
//uses a global variables input arrays w[1..n], v[1...n] and
//Table v[0...n, 0...w] whose entries are initialized
//with -1's except for row 0 & column 0 initialized with 0's.
if v[i,j]<0
        if j < weights[i]
                value ← MFknapsack(i-1,j)
        else
                value ← max(MFknapsack(i-1,j), values[i] +
                                MFknapsack(i-1,j-weights[i]))
        v[i,j] ← value
        return v[i,j].
```

Advantages:-

* Number of computations is reduced. For the given problem it is reduced to 11 from 20.

* For larger instances, it is more efficient.

Greedy Technique

Problems that have n inputs and require us to obtain a Subset that Satisfies Some Constraints (minimize total distance (or) cost and maximize the profit).

Any Subset that Satisfies these Constraints is called a feasible Solution.

A feasible Solution that either maximizes or minimizes a given objective function is called an optimal Solution

It is easy to derive feasible Solution but not necessarily an optimal solution.

The greedy method Suggests, an algorithm that works in stages Considering 1 input at a time. At each stage a decision is made whether a Particular input is an optimal Solution. If the inclusion of next input into the partially Constructed optimal Solution will result in an infeasible Solution, then this input is not added else it is added to partial Solution.

The Selection procedure need to find which input to be added is an optimization measure named as objective function

∴ The greedy method selects an i/p at each stage which derives a feasible Solution then it is added to the optimal Solution until the Problem terminates with a Condition.

If the greedy strategy fails to derive an optimal solution then they are considered under the Np class.

(or)

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.

On each step, the choice made must be

* feasible – Should satisfy the problem's constraints.

* Locally optimal – It has to be the best local choice among all feasible choices available on that step.

* Irrevocable – once made, it cannot be changed on subsequent steps of the algorithm.

→ Can be effectively applied for graph optimization problems.

eg/.. Prim's algorithm – To find MST in an undirected graph

Dijkstra's algorithm – To find single source shortest paths in an undirected & directed graph.

Kruskal's algorithm – To find MST for G or a minimum spanning tree collection if G is not connected.

→ finding shortest path.
→ Job Scheduling deadlines
→ optimal storage on tapes.

Above algorithms can be implemented using a priority Queue to select best current choice from a set of candidate edges.

→ also used in construction of Huffman trees (i.e) an application of Huffman code – an important data compression method to encode characters.

Explain the working of prim's Algorithm [Nov/Dec '17]

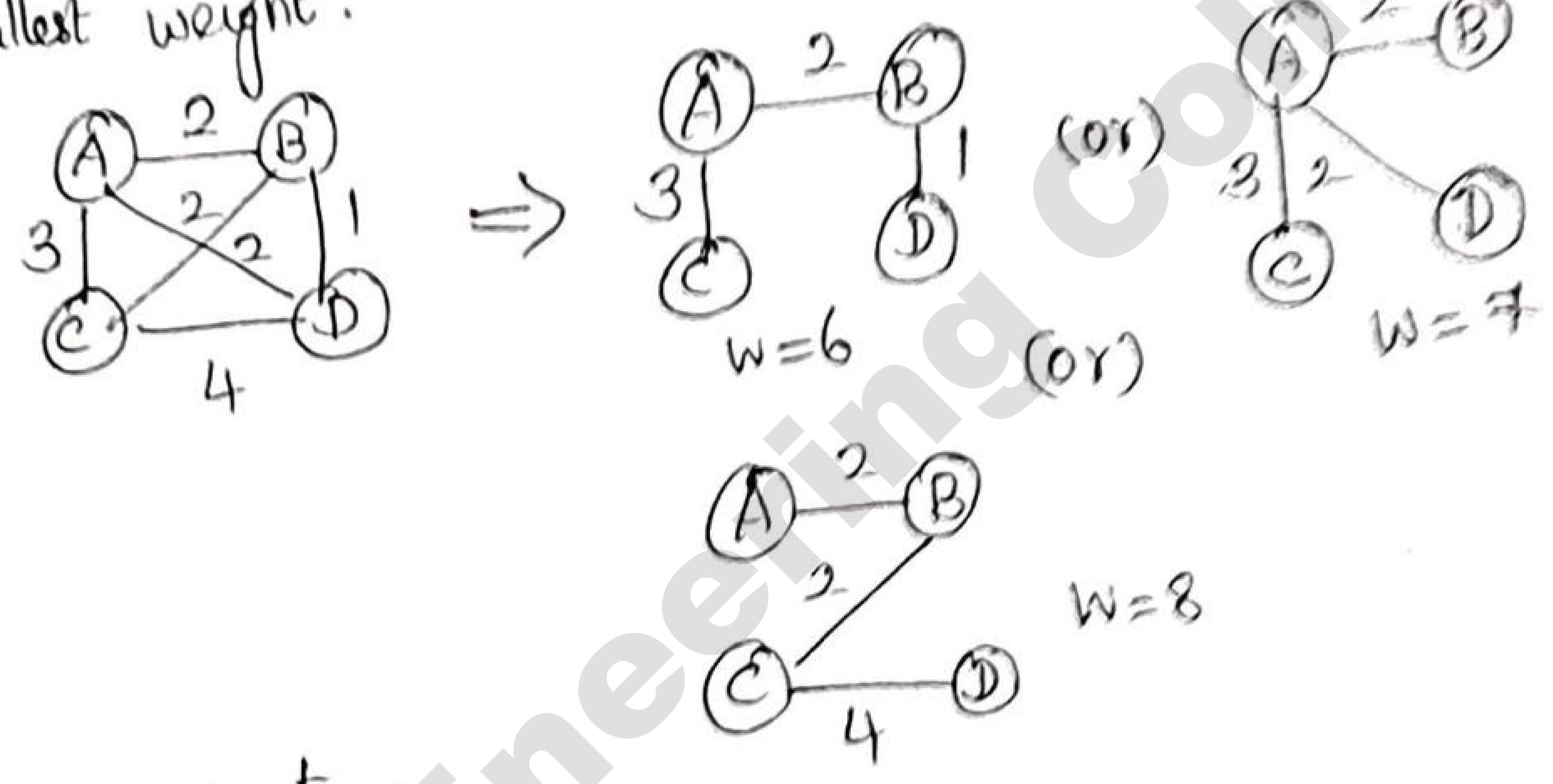
PRIM'S Algorithm [Apr/May 2018]

A Spanning tree for a Connected graph G=(V,E) is a Subgraph t=(V,E') which is basically a tree & it Contains all the vertices of G Containing no circuit.

MST:-

A MST of a weighted Connected graph G is a Spanning tree with minimum or Smallest weight.

eg/..



Applications of Spanning tree:-

→ Important in designing efficient routing algorithms

→ wide applications in many areas such as Network design.

Basic principle (or) strategy

1. Prim's algorithm Selects a starting Vertex from the given graph & classifies the start vertex Under "tree vertices"

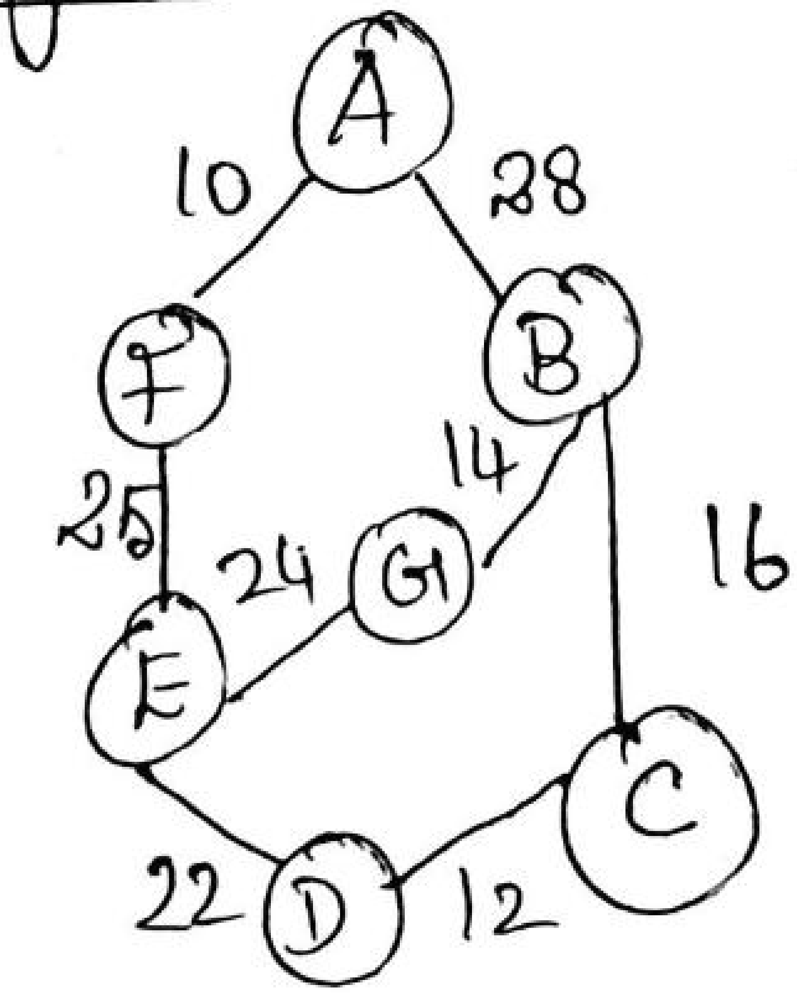
2. The nodes adjacent to tree vertices are identified and classified Under "fringe vertices" & remaining vertices are classified as "Unseen vertices".

3. The Selection of a new vertex from the "fringe" is depend on the weight of the edge (minimum). The process continues until the "fringe" is empty. After this new inclusion again the "fringe vertices" and "Unseen vertices" are reclassified.

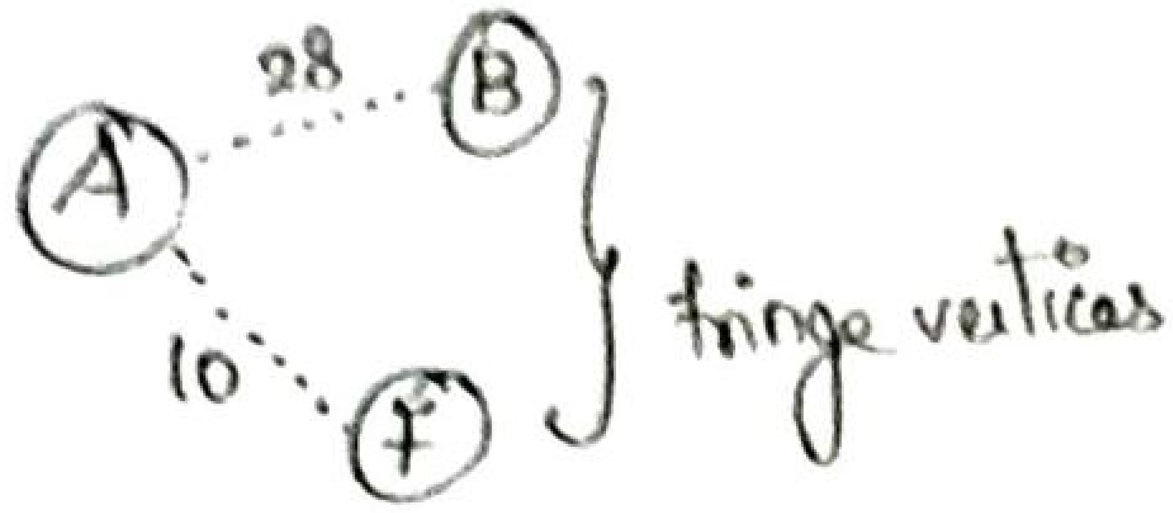
Algorithm:-

```
PrimMST (G, n)
    Initialise all vertices as unseen.
    Select an arbitrary vertex s to start the tree, reclassify the
    tree. Reclassify all vertices adjacent to s as fringe.
  While there are fringe vertices:
        Select an edge of minimum weight b/w a tree vertex t &
        a fringe vertex v;
        Reclassify v as tree; add edge 'tv' to the tree.
        Reclassify all unseen vertices adjacent to v as fringe.
```

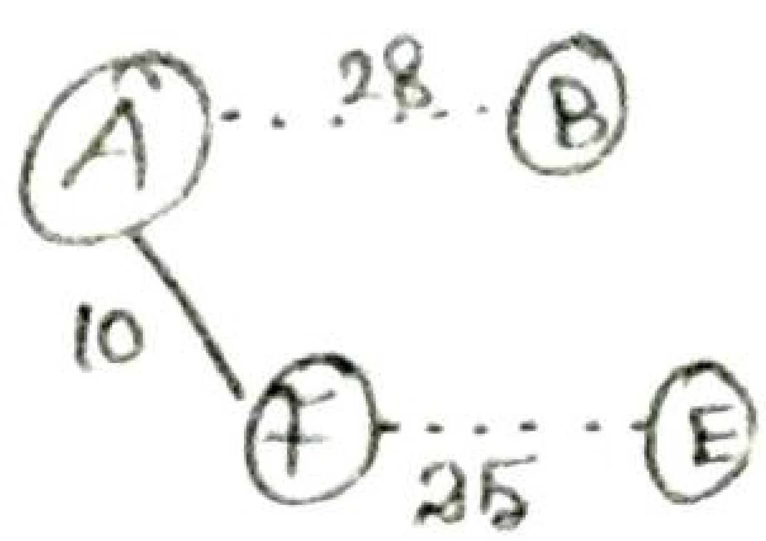
Eg/.. A weighted graph



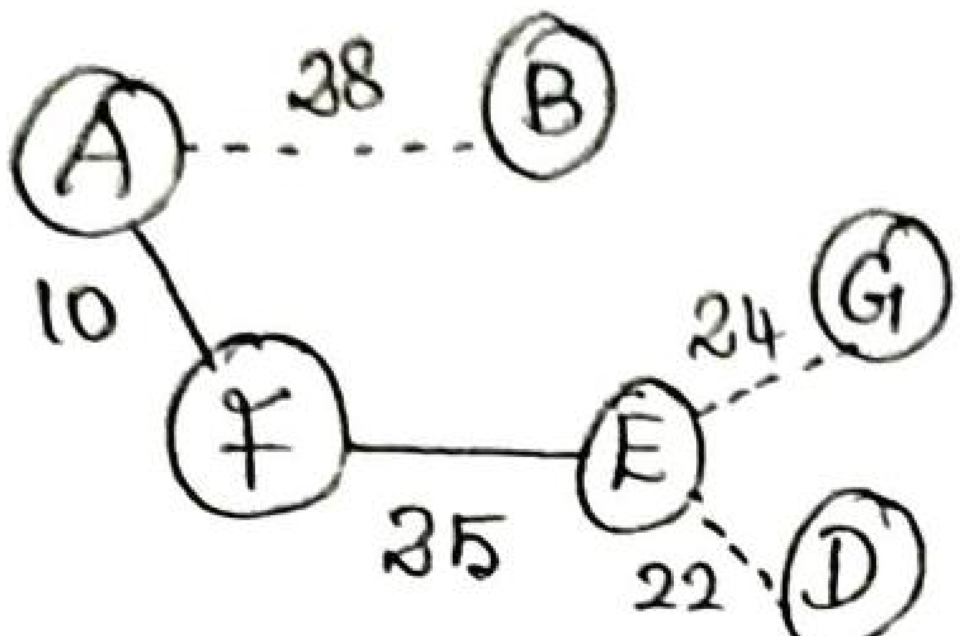
Step 1 :



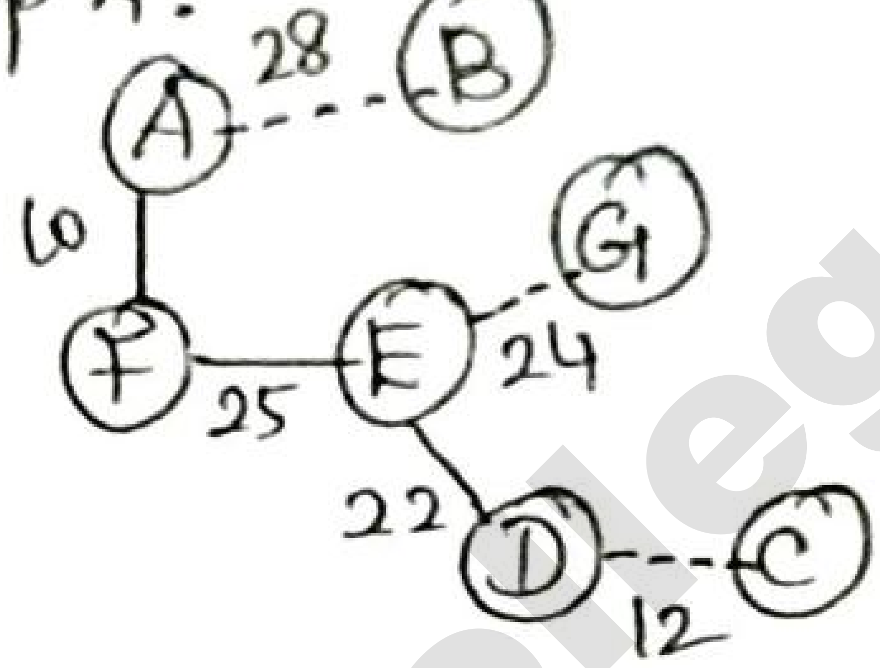
Step 2 :



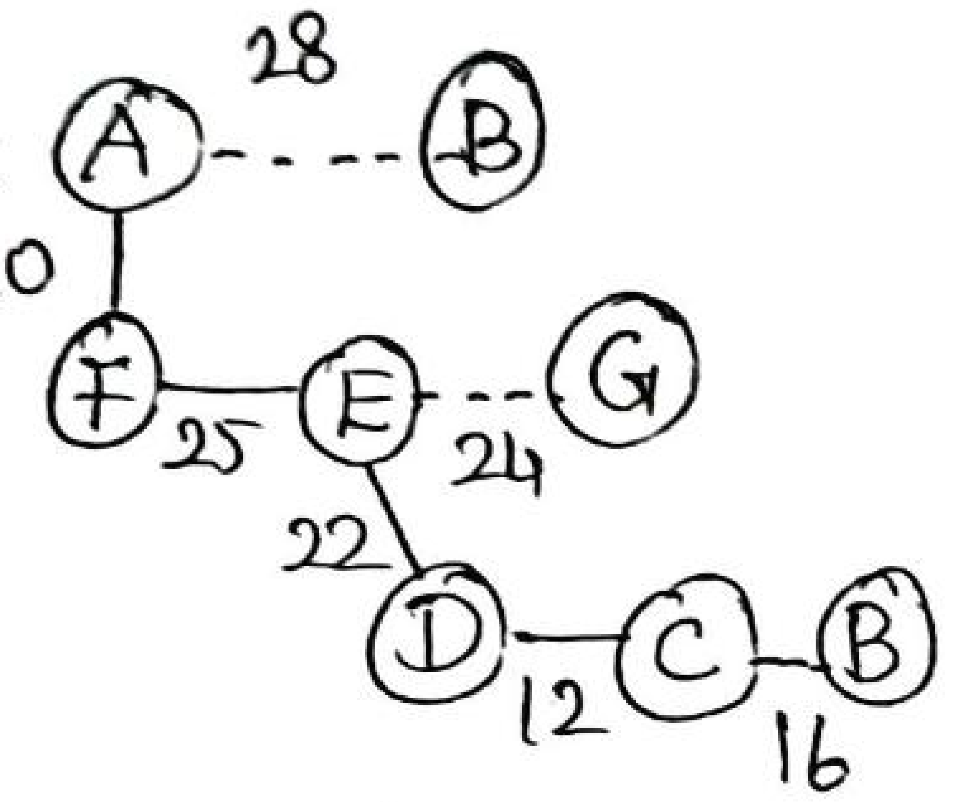
Step 3 :



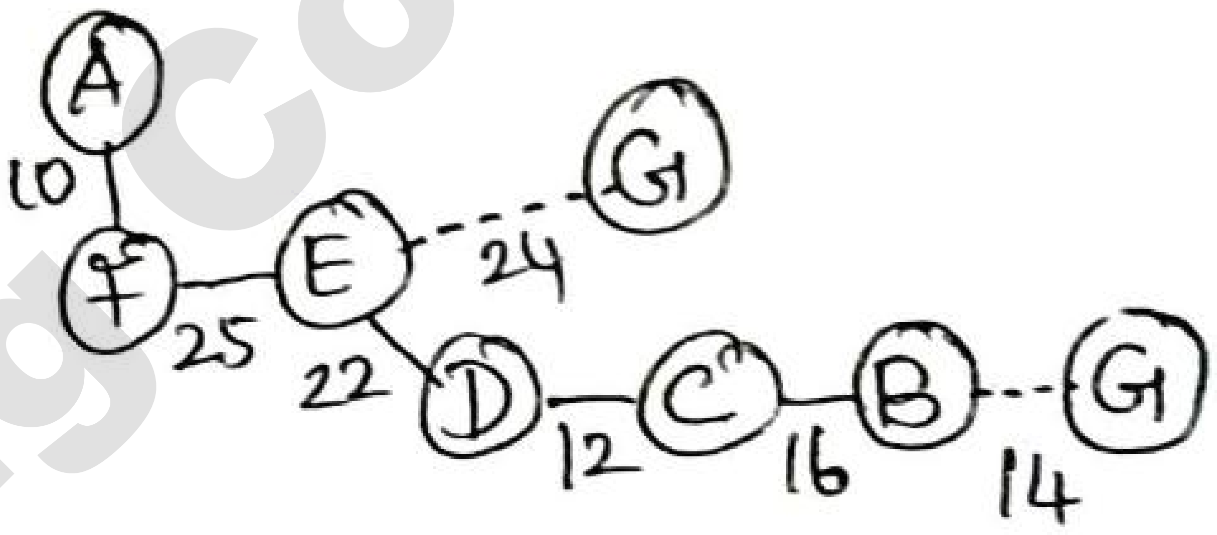
Step 4 :



Step 5 :



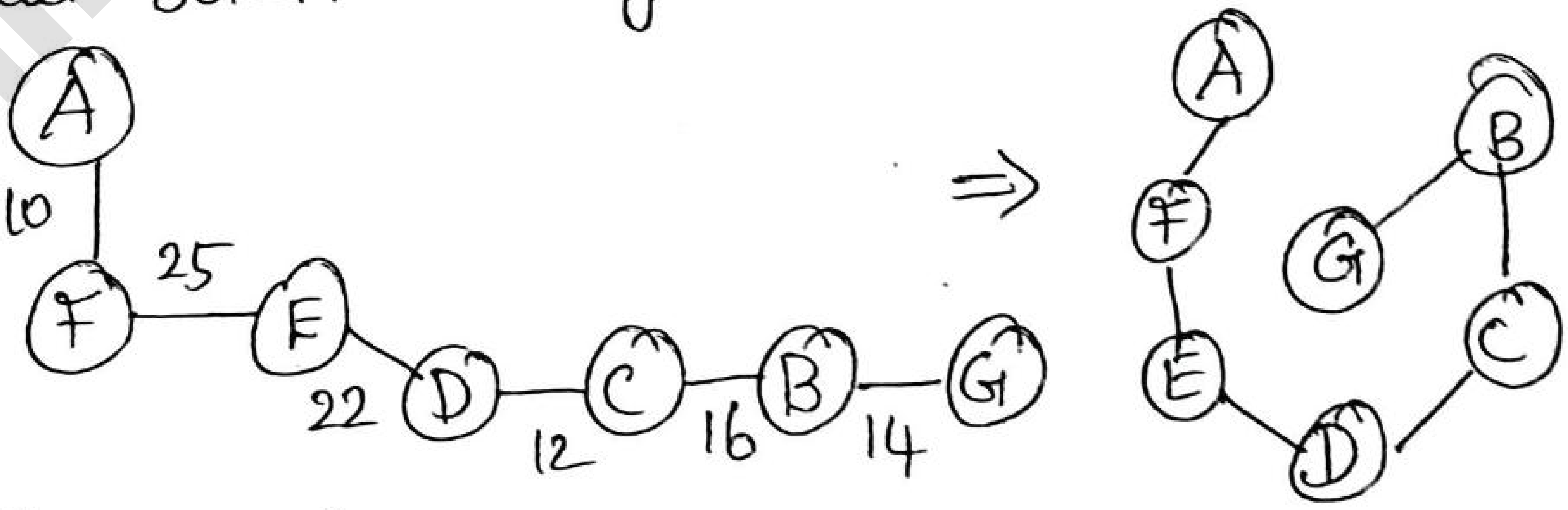
Step 6 :



from step 5 diagram it is clear that node B is existing for 2 times in the fringe vertices (i.e) from A & C. So, select only 1 node for fringe depending on the weightage. So, edge CB=16 is selected & the edge with maximum weightage AB=28 is discarded.

Step 7: As there are 2 paths for G, BG=14 and EG=24.

Consider BG=14 So we get



So the weight of MST is 99//

Analysis

Space Usage:-

3n locations are used to construct a MST using Prim's algorithm. (i.e) a) n locations for priority Queue b) n locations for maintaining the status array c) n locations for o/p array.

Time Complexity:-

The algorithm spends most of the time in selecting the edge with minimum length. Hence the basic operation of this algorithm is to find the edge with minimum path length.

$$T(n) = \sum_{k=1}^{n-1} \left(\sum_{i=0}^{n-1} 1 + \sum_{j=0}^{n-1} \right) \longrightarrow \text{Time taken by for loop of } j.$$

Time taken by for loop of k

Time taken by for loop of i

$$T(n) = \sum_{k=1}^{n-1} \left[((n-1)+0+1) + ((n-1)+0+1) \right]$$

$$= \sum_{k=1}^{n-1} 2n$$

$$= 2n \sum_{k=1}^{n-1} 1$$

$$= 2n \sum_{k=1}^{n-1} 1 = 2n\left((n-1)-1+1\right) = 2n(n-1)$$

$$= 2n^2 - 2n.$$

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

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

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

If the prim's algorithm is implemented using binary heap with the creation of graph using adjacency list then $T(n) = O(E \log_2 V)$ where E stands fo Total no/. of edge & V stands for

Explain the Dijkstra's shortest path algorithm and its efficiency ? [Nov/Dec '19]

Dijkstra's Algorithm

→ This is a single source shortest path algorithm, used for a different types of applications.

a) To find a minimum weight path between two specified vertices.

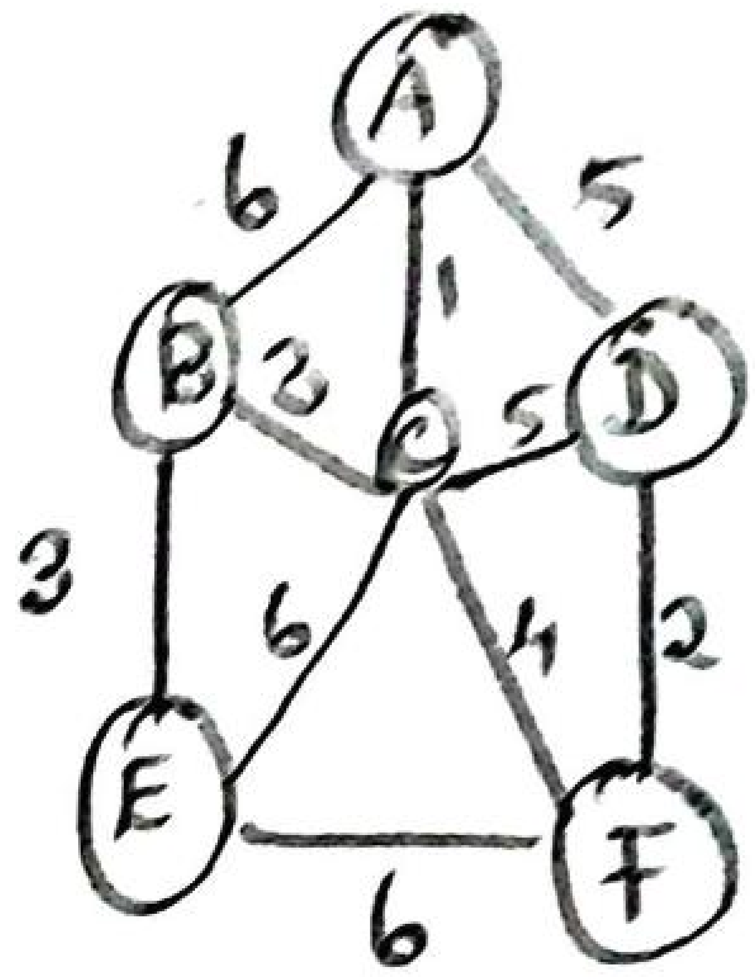
b) To find a minimum weight paths between s and every vertex reachable from s.

When weight is interpreted as distance, a minimum weight path is called as shortest path. This algorithm requires that edge weights be non-negative, for a given weighted graph $G=(V,E,W)$ and a source vertex s. The problem is to find a shortest path from s to each vertex v.

Strategy or Basic principle

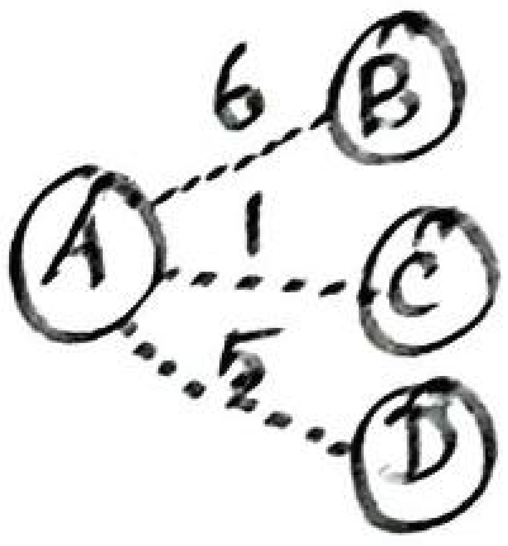
1. Selects a starting vertex from the given graph & classifies the start vertex under "tree vertices".
2. Selection of new vertex from the "fringe" is depends on the total minimum weight of the edge from the start vertex s to current vertex z.
3. Nodes adjacent to tree vertices are identified and classified under "fringe vertices" and the remaining vertices are classified as "unseen vertices".

eg/..



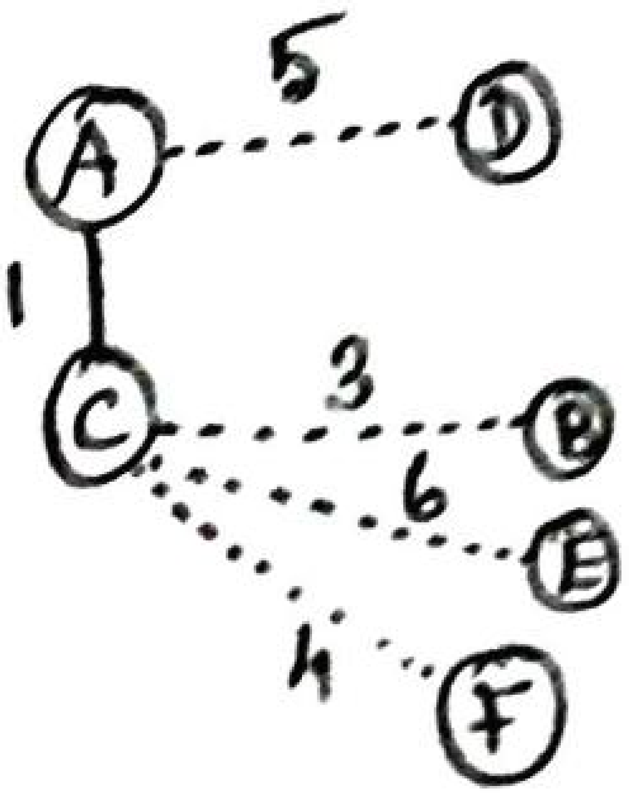
G = (V,E,W)

Step 1:



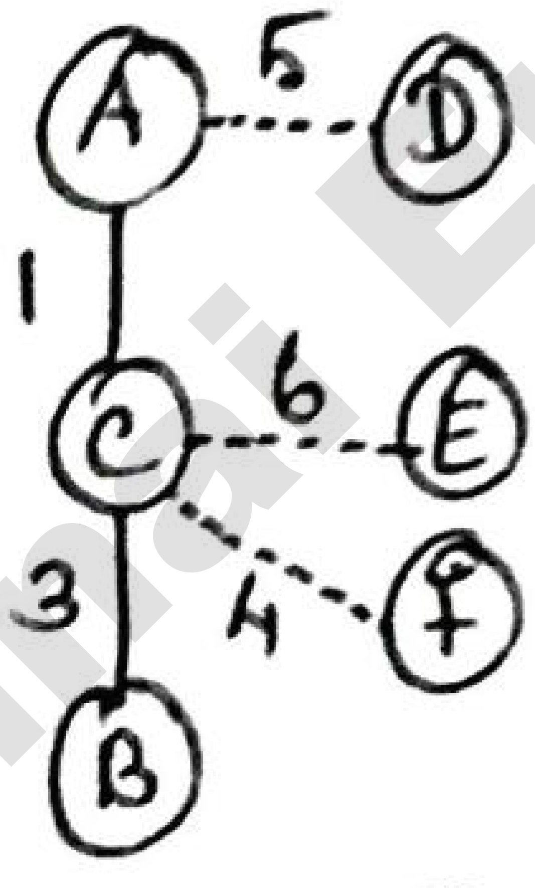
d(A,A)+w(AB)=6
d(A,A)+w(AC)=1
d(A,A)+w(AD)=1
Select AC next.

Step 2:

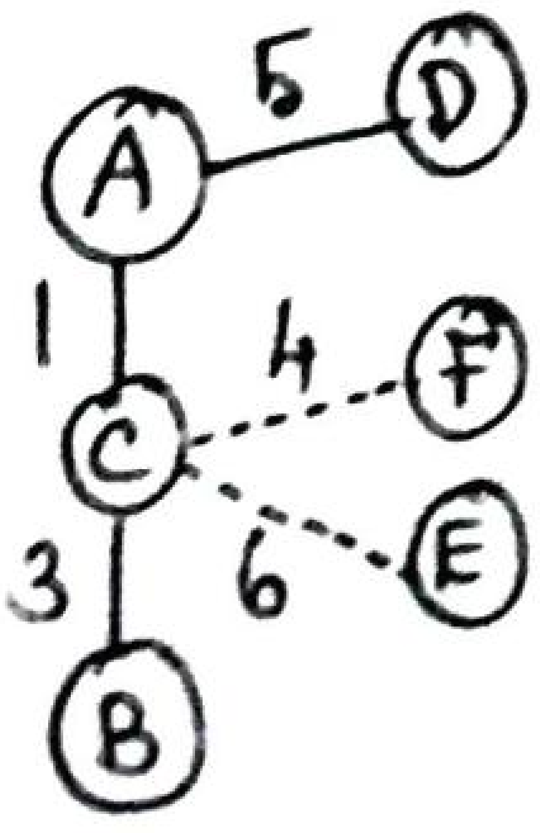


d(A,A)+w(AD)=5
d(A,C)+w(CB)=4
d(A,C)+w(CE)=7
d(A,C)+w(CF)=5
Select CB next.

Step 3:

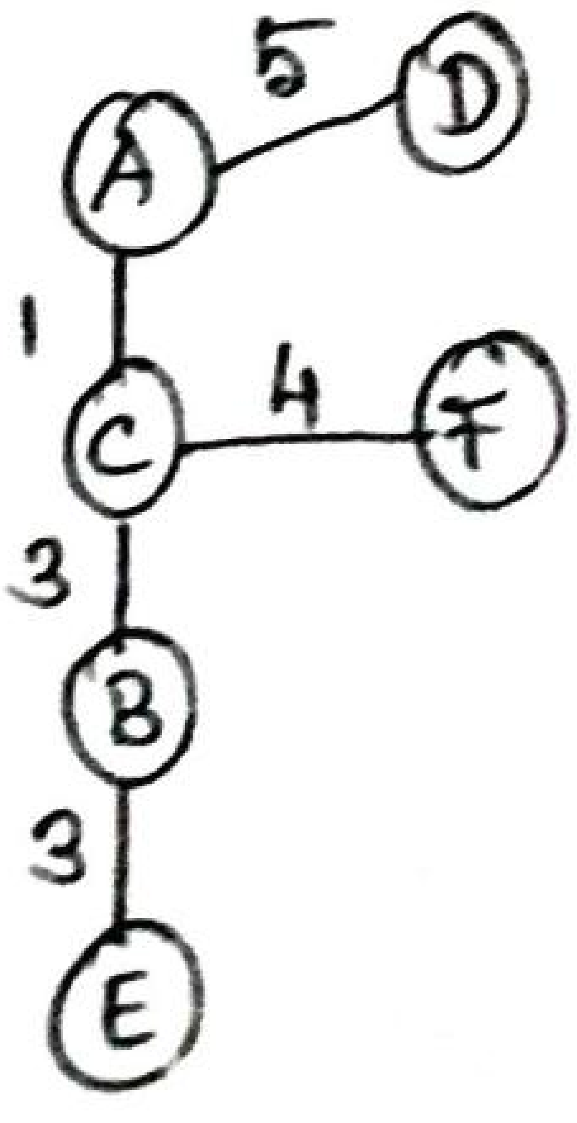


Step 4:



⇒

Step 5:



Algorithm

```
dijkstraSP(G,n)
    Initialise all vertices as Unseen
    Start the tree with the specified Source vertex s; reclassify it as tree.
     define $d(s,s)=0$
     Reclassify all vertices adjacent to s as fringe.
     While there are fringe vertices
                Select an edge between a tree vertex t and a.
                fringe vertex v such that $(d(s,t)+w(tv))$ is minimum;
                Reclassify v as tree; add edge tv to the tree;
                define $d(s,v)=(d(s,t)+w(tv))$
                Reclassify all unseen vertices adjacent to v as fringe.
```

Analysis:-

The worst case Complexity of Dijkstra's algorithm is $O(n^2)$ (for n nodes & m edges).

It a significant number of vertices are expected to be Unreachable, it might be more efficient to test for reachability as a preprocessing step, eliminate unreachable vertices an renumber the remaining vertices as 1 . . . n. The total cost would be in $O(m+n^2)$ rather than $O(n^2)$.

Apply Kruskal's Algorithm to find a MST for the following graph. [Nov/Dec '16]

Kruskal's Algorithm

Strategy

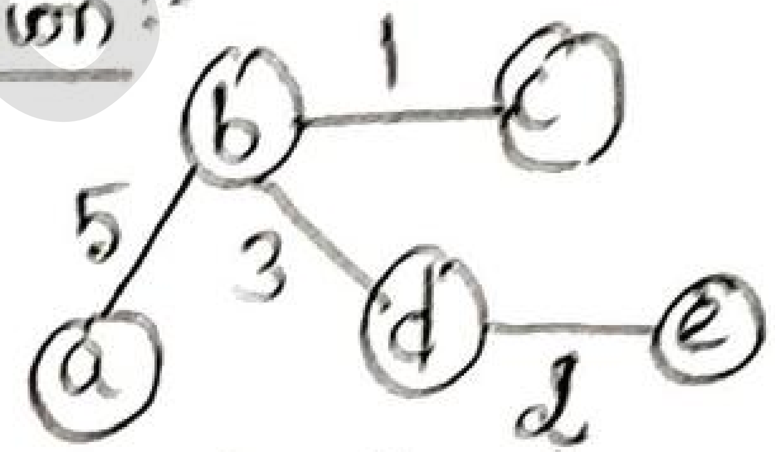
1. To Construct MST, edges of the graph are selected in non decreasing order of cost (or) distance. (discards an edge if it forms a cycle).
2. Set of t edges so far selected for the spanning tree will form a forest, but not necessarily 1 tree.
3. Terminates when all edges have been processed.

Algorithm Kruskal MST (G,n)

```
R=E; remaining edges.
F=ϕ; // foreset edges
 While (R is not empty)
     Remove the lightest (shortest) edge vw, from R;
     if (vw does not make a cycle in F)
         Add vw to F;
 return F;
```

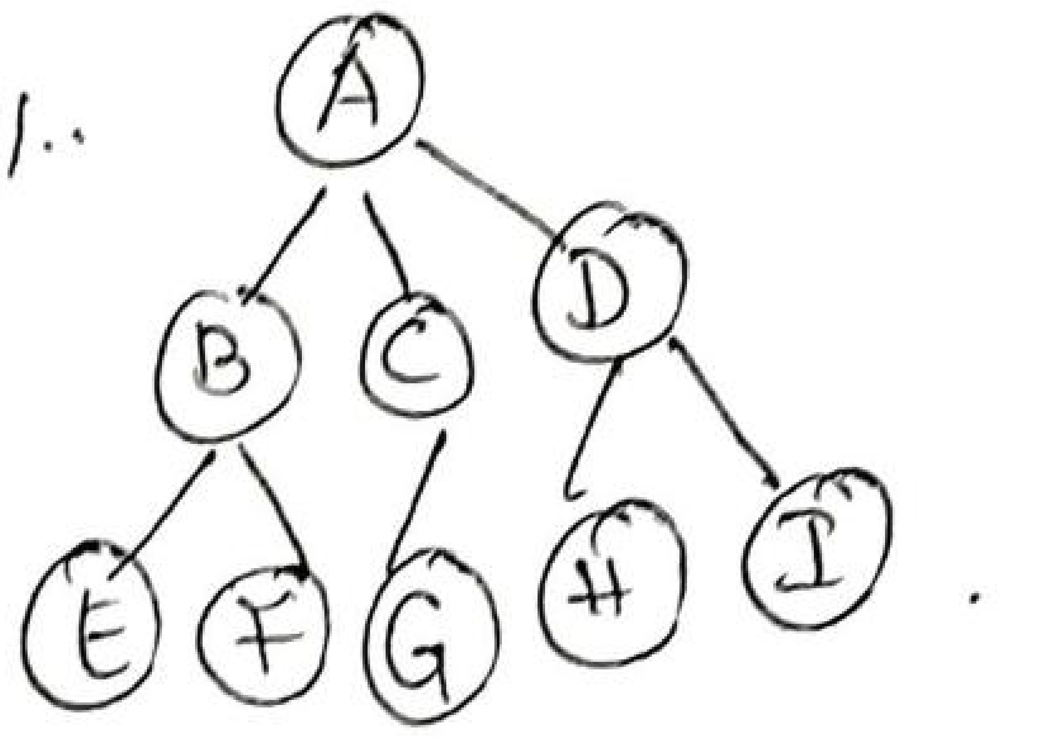
eg/.. **Solution**:

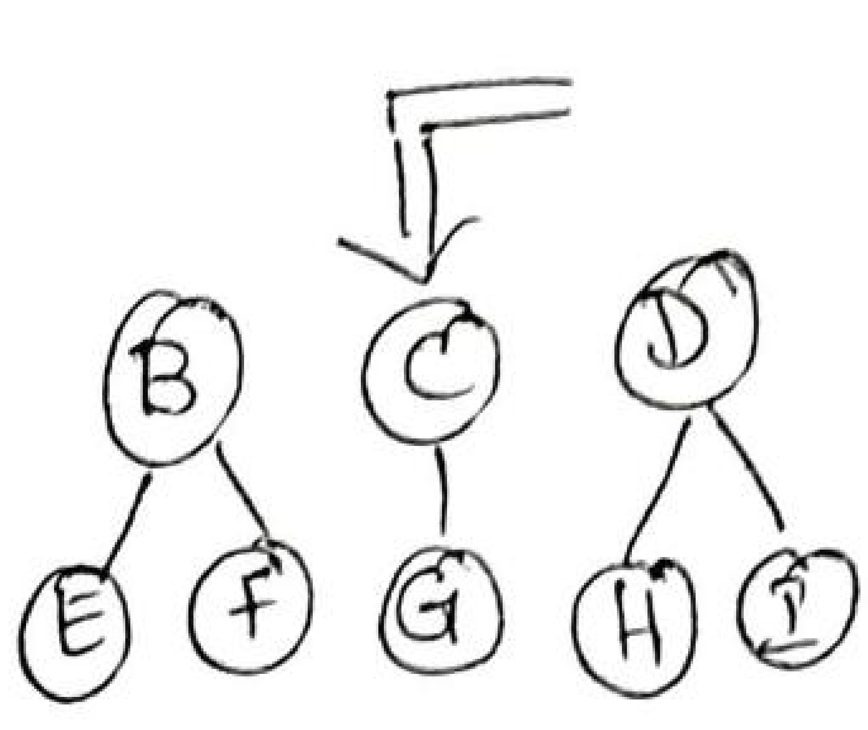
MST



Cost = 11 units.

Forest:- A forest is a set of n≥0 disjoint trees. The notion of a forest very close to that of a tree because if we remove the root of a tree, we get a forest. eg/..





The MST collection is the output when the given graph G=(V,E,W) is not Connected.

Time Complexity is O(m log m) better for Sparse trees / graphs.

APR/MAY 17

Huffman Trees:-

→ They are Constructed for encoding a given text of characters, each character is associated with Some bit Sequence. This bit Sequence is named as code word.

→ Encoding can be classified based on the number of bits used for each character in the text into 2 types

* Fixed length encoding - Each character is associated with a bit String of Some fixed length
eg.. ASCII - 7 bits for a character.

* Variable length encoding - each character is associated with a bit string of different length.

(i) Shorter length codeword for more frequent characters and longer length Codeword for less frequent characters eg.. Telegraph code.

(ii) Using the property of prefix code - no codeword is a Prefix of another character's code word. This property is used to identify the number of bits required to encode the ith character of a text.

How to generate binary Prefix Code:-

Associate the character of a text to be encoded with leaves of a binary tree, in which all the left edges are labelled by 0 & all the right edges by 1.

→ To assign shorter bit strings to high frequency characters and longer bit strings to low frequency character strings? This task is done by the greedy algorithm given below:.

Huffman's Algorithm:.

Step 1: Initialize n one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's weight. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves)

Step 2: Repeat the following operation until a single tree is obtained. find 2 trees with the smallest weight (ties can be broken arbitrarily). Make them the left & right subtree of a new tree & record the sum of their weights in the root of the new tree as its weight

A tree constructed by this algorithm is called a Huffman tree. It defines the character in the form of strings, based on their frequencies in a given text is known as Huffman Code

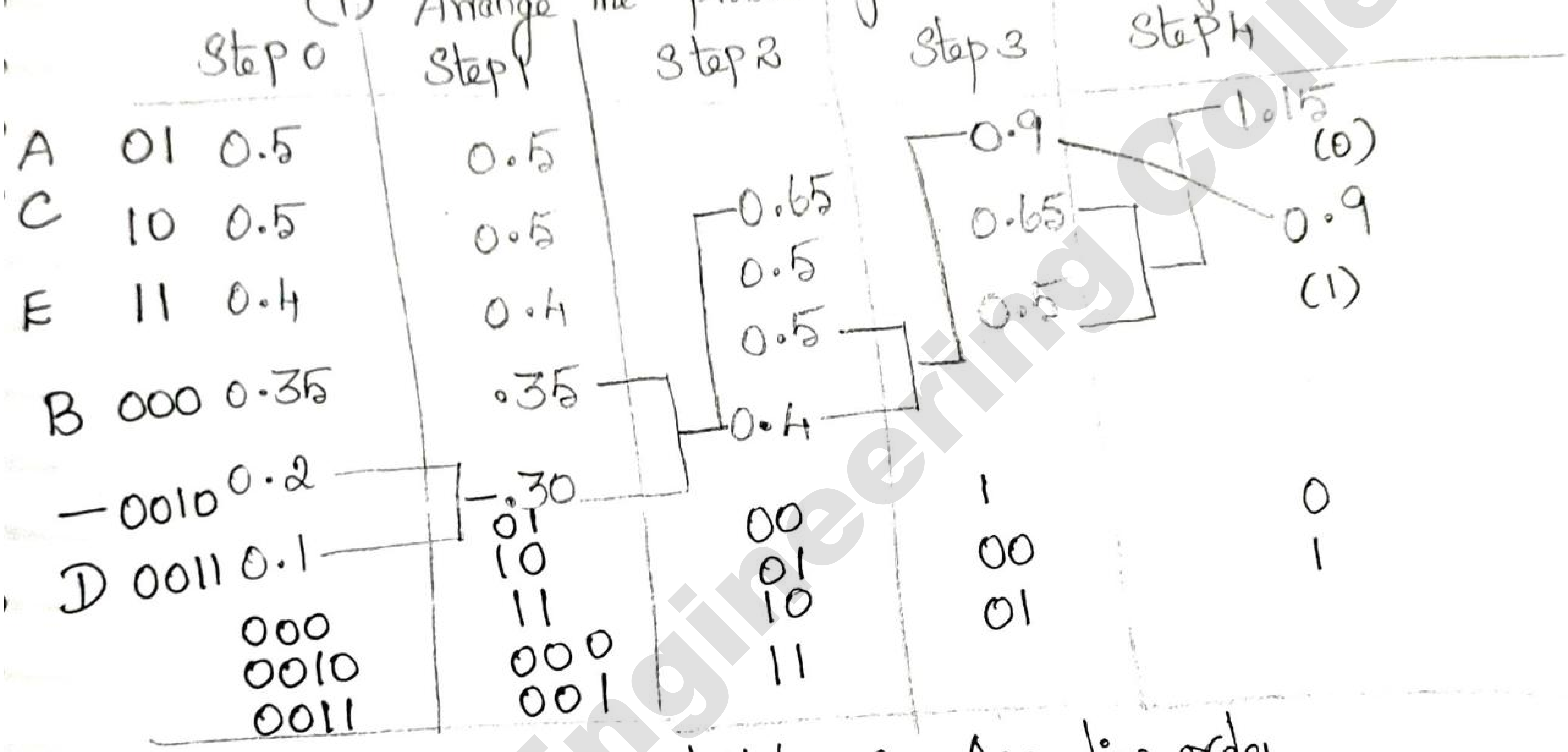
AR/May '17

Eg1. Construct the Huffman's tree for the following data & obtain the Huffman's code:-

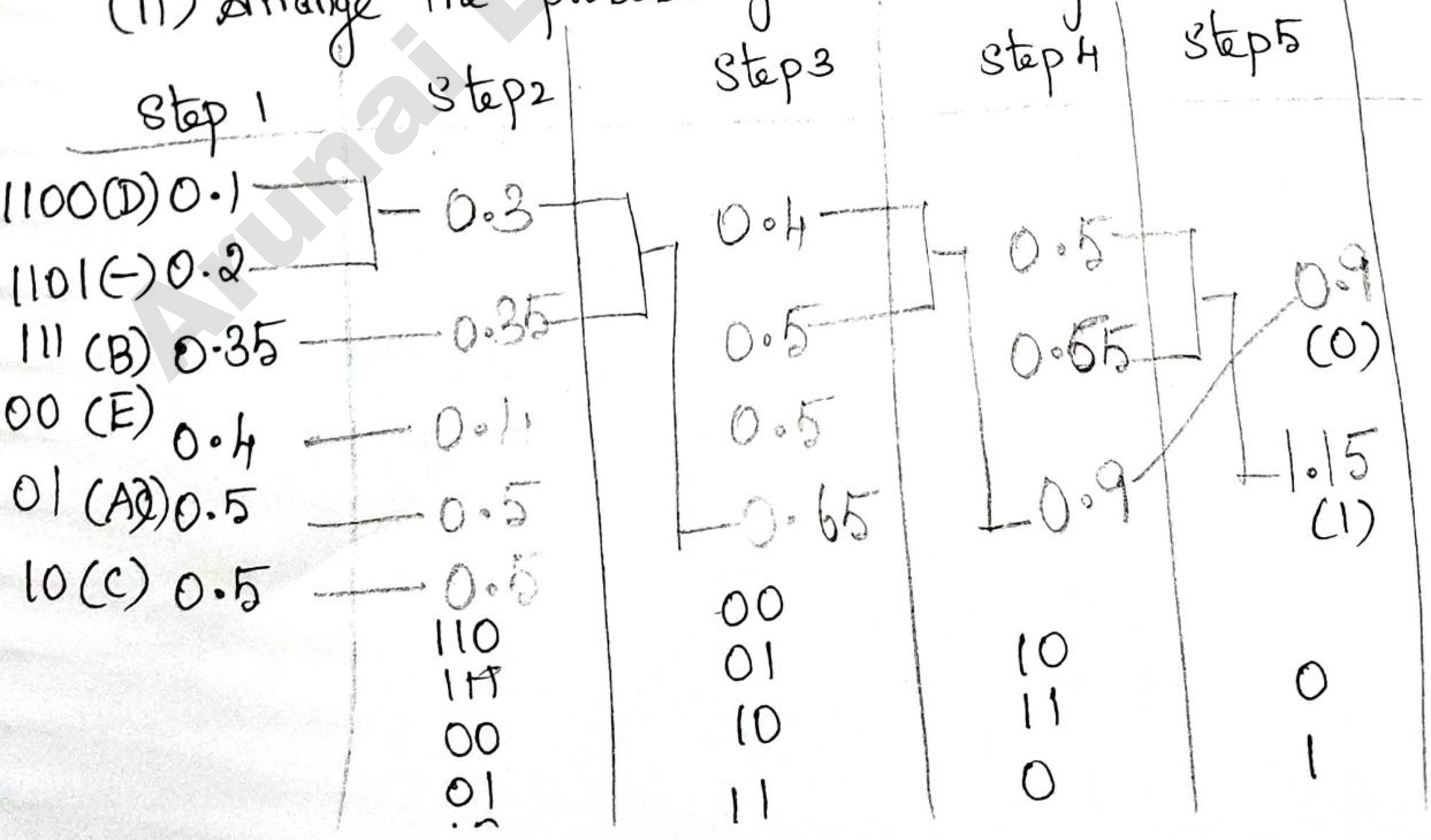
Character	A	B	C	D	E	—
Probability	0.5	0.35	0.5	0.1	0.4	0.2

Solution:-

(i) Arrange the probability in decreasing order



(ii) Arrange the probability in Ascending order



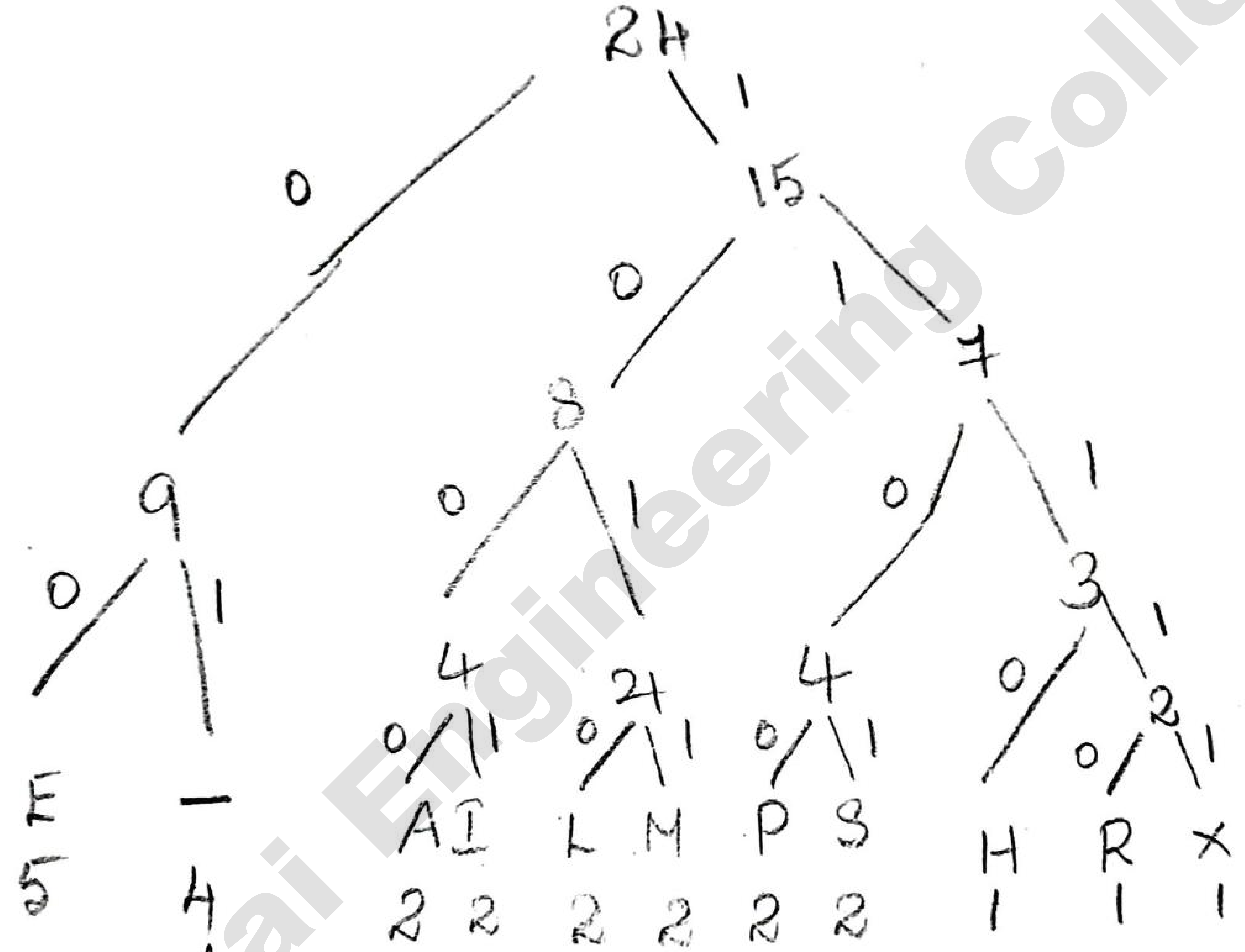
Nov/Dec '2017

eg.. find the Codes for the alphabets below according to the given frequency:-

—	A	E	H	I	L	M	P	R	S	X
4	2	5	1	2	2	2	2	1	2	1

Step 1 : Arrange the frequency in ascending order:-

H	R	X	A	I	L	M	P	S	—	E
1	1	1	2	2	2	2	2	2	4	5



	freq	code
—	4	01
A	2	1000
E	5	00
H	1	1110
I	2	1001
L	2	1010
M	2	1011

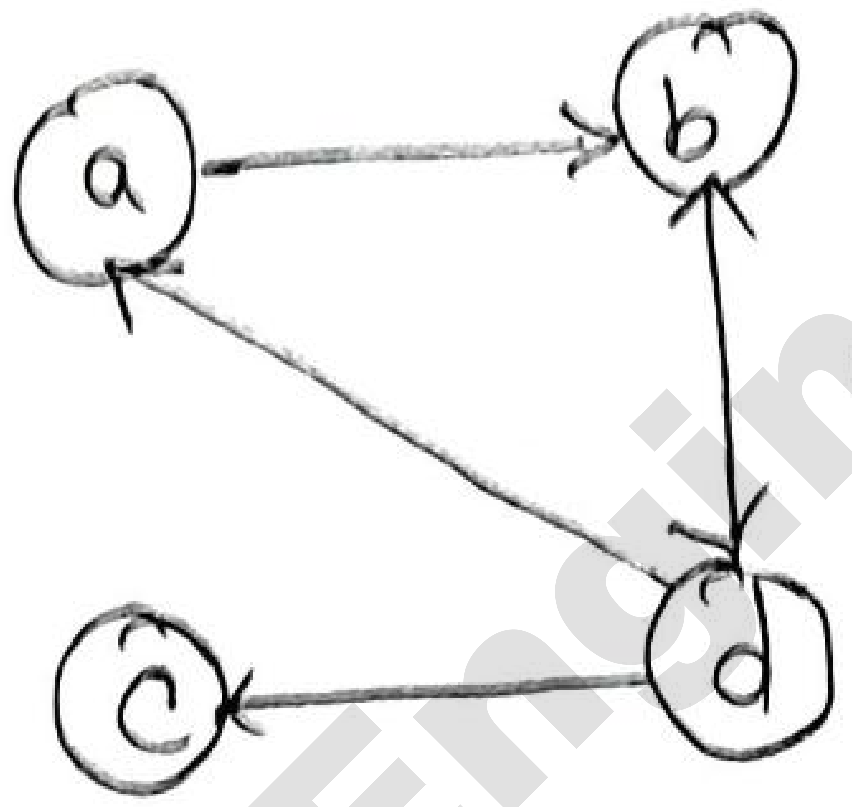
	freq	Code
P	2	1100
R	1	11110
S	2	1101
X	1	11111

Warshall's and Floyd's Algorithm [APR|May 2018]

Warshall's algorithm is used for computing the transitive closure of a directed graph

The transitive closure of a directed graph with n vertices Can be defined as the n by n boolean matrix $T = \{t_{ij}\}$ in which the elements in the i^{th} row $(1 \le i \le n)$ and j^{th} column $(1 \le j \le n)$ is 1 if there exists a non-trivial directed path (a directed path of a positive length) from the i^{th} vertex to the j^{th} vertex, otherwise t^{ij} is 0.

eg/..



$$\Longrightarrow \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

Adjacency matrix

Transitive closure

$$\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

This method of traversing the Same digraph Several times, so better algorithm could be warshall's algorithm.

⇒ Warshall's algorithm Constructs the transitive closure of a given digraph with n vertices through a Series of n×n boolean matrices

$$R^{(0)}, R^{(1)} \ldots R^{(k-1)}, R^{(k)}, \ldots R^{(n)}$$

Each of these matrices provides certain information about directed paths in the digraph.

⇒ The element $r_{ij}^{(k)}$ in the i^{th} row and j^{th} column of matrix $R(k) = (k = 0, 1, \ldots n)$ is equal to 1 iff there exists a directed Path (of positive length) from i^{th} vertex to the j^{th} vertex with each intermediate vertex if any numbered not higher than k.

Series starts with $R^{(0)}$, which does not allow any intermediate vertices in the path. Hence $R^{(0)}$ = adjacency matrix of digraph.

Rules for warshall's Algorithm

1. if an element r_{ij} is 1 in R^{k-1} it remains 1 in R^k to 1

2. If an element r_{ij} is 0 in R^{k-1} it has to be changed in R^k, iff the element in its row i & column k and the element in its column j & row k are both 1's in R^{k-1}.

Algorithm Warshall (A[1..n, 1...n])

// Implements Warshall's algorithm for Computing the transitive closure

// Input : The adjacency matrix A of a digraph with n vertices.

// output : The transitive closure of the digraph.

```
$R^{(0)} \leftarrow A$
  for k ← 1 to n do
     for i ← 1 to n do
        for j ← 1 to n do
           $R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j]$ or $R^{(k-1)}[k,j]$.

  return $R^{(n)}$.
```

Time Complexity :-

⇒ Its time efficiency is $\Theta(n^3)$

⇒ Warshall algorithm Can be Speed up for Some i/p by restructuring its innermost loop.

⇒ Another way to make the algorithm run faster is to treat matrix row as bit strings and employ the bitwise or operation available in most modern Computer languages

Space Efficiency

⇒ Separate matrices is used for recording intermediate results of the algorithm but infact it is Unnecessary.

Floyd's Algorithm

→ Used for all pairs shortest path problems

→ Used to find the distances from each vertex to all other vertices in a given weighted connected graph (directed or undirected)

→ The length of shortest path in an n×n matrix D called the distance matrix.

→ Properties of Floyd's

* applicable to both undirected and directed weighted graphs.

* they do not contain a cycle of a negative length.

* Computes the distance matrix of a weighted graph with n vertices through a series of n×n matrices.

$D(0), \ldots D(k-1), D(k) \ldots D(n)$.

```
Algorithm Floyd (W[1...n, 1...n])
Input : The weight matrix W of a graph
output : The distance matrix of the shortest paths length.
    D ← W // is not necessary if W can be overwritten
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i,j] ← min {D[i,j], D[i,k] + D[k,j]}
return D.
```

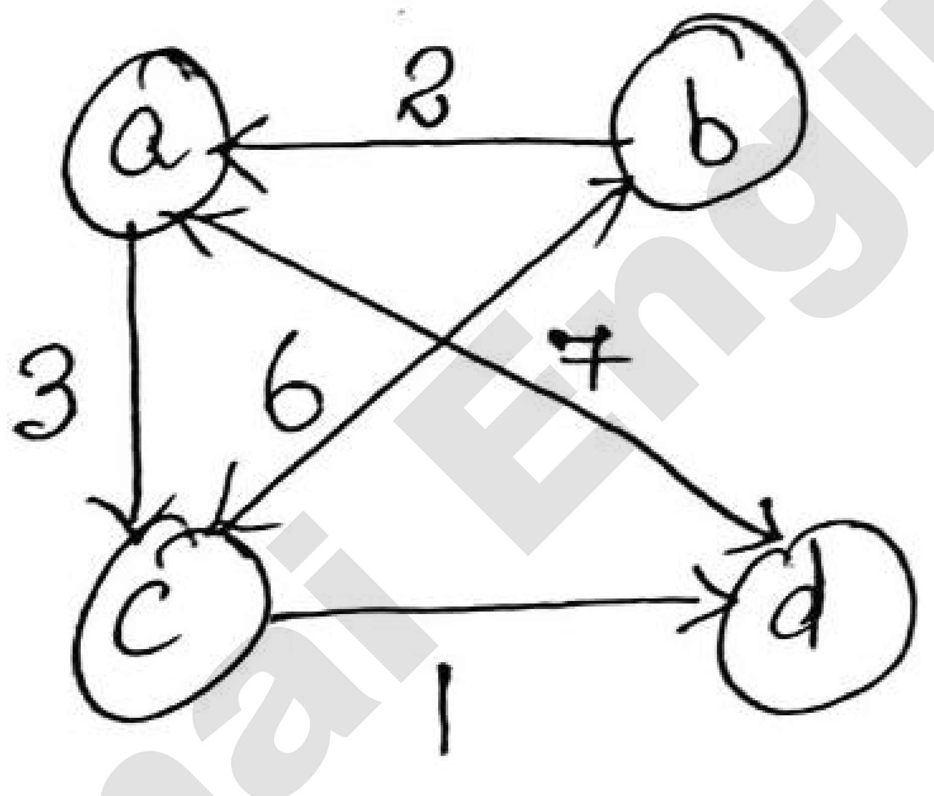
$$D^{(0)} = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 6 & 0 & 1 \\ 7 & \infty & \infty & 0 \end{bmatrix}$$

$D^{(0)}$ - which does not allow any intermediate vertices in the path.

$D^{(n)}$ - Contains the lengths of the shortest path among all Paths that can use all n vertices as intermediate.

⟹ $d_{ij}^{(k)}$ in the i^{th} row and the j^{th} coloumn of matrix $D^{(k)}$ $(k=0,1 \ldots n)$ is equal to the length of the shortest Path among all paths from the i vertex to the j vertex with each intermediate vertex, if any numbered not higher than k.

eg/..



$$D^{(0)} = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 9 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

Optimal Binary Search Tree (OBST).

→ It is a Special kind of advanced tree

→ It focus on how to reduce the Cost of the Search of BST.

→ Dynamic programming be used for Constructing an OBST for a given Set of Keys.

→ If probabilities of Searching for elements of a set are known, it is natural have an optimal binary Search tree for which the average number of Comparison in a Search is the Smallest possible.

Algorithm optimal BST (P[1...n])

Input : An array P[1...n] of Search probabilities for a sorted list of n keys

output : Average number of Comparison in Successful Searches in the optimal BST & table R of Subtrees roots in the optimal BST.

```
for i ← 1 to n do
        C[i, i-1] ← 0
        C[i, i] ← P[i]
          R[i, i] ← i
        C(n+1, n) ← 0
            for d ← 0 to n-d do
```

Solution Using Dynamic Programming

$$OBST(i,j) = \min_{i \leq r \leq j} \left\{ \begin{array}{l} OBST(i,r-1) + P_r + OBT(r+1,j) \\ + \sum_{k=1}^{r-1} P_k + \sum_{s=r+1}^{j} P_s . \end{array} \right.$$

$$= \min_{i \leq r \leq j} \left\{ \begin{array}{l} OBST(i,r-1) + OBST(r+1,j) \\ + \sum_{k=i}^{j} P_k . \end{array} \right.$$

The Solution for the optimal Solution for BST Construction from node i to node j will be the cost of optimal BST Construction for node i to r-1 and node r+1 to j and overlapping Subproblems.

$$OBST(i,j) = \min_{i \leq r \leq j} \left\{ \begin{array}{l} OBST(i,r-1) + P_r + \\ OBST(r+1,j) + \sum_{k=i}^{j} P_k . \end{array} \right.$$

```
j ← i+d
    minval ← ∞
        for k←i to j do
            if c[i,k-1]+c[k+1,j] < minval
    minval ← c[i,k-1] + c(k+1,j)
                Kmin ← k
                R[i,j] ← k
                Sum ← p[i];
                for S←i+1 to j do
                    Sum ← sum+P[S]
                    c[i,j] ← minval + Sum
                return V[1,n],R
```

To Construct optimal Binary Search tree for a given numbers $W_1, W_2, W_3 \cdots W_n$ and fixed probabilities $P_1, P_2 \cdots P_n$ of their Occurrance.

→ Arrange these numbers in a BST that minimizes access time. In a BST, the number of Comparison needed to access an element at depth d is (d+1)

→ To Search a number, it needs d+1 Searches. So the Cost of Search will be $C = P_S \times (d+1)$.

Dynamic Programming :-

Multi-stage graphs :-

The multi-stage graph is to find a minimum cost from Source 's' to sink 't' (ie) destination

Problem description :-

✓ A multistage graph $G=(V,E)$ is a directed graph in which the vertices are partitioned into $k>=2$ disjoint sets V_i $1<=i<=k$

✓ if (u,v) is an Edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some $1 \le i \le k$

✓ The sets V_1 and V_k are such that $|V_1| = |V_k| = 1$

✓ The vertex 's' is the Source & 't' is the Sink.

✓ Let $c(i,j)$ be the cost of edge(i,j)

✓ The cost of a path from 's' to 't' is the sum of the cost of the edges on the path.

✓ Each set V_i defines a stage in the graph.

✓ Every path from s to t starts in stage 1 goes to state 2 ... and Soon and eventually terminates at stage k.

Procedure for multistage problems

* find path from s to t, stage by stage.

* Every s to t path is the result of a sequence of $k-2$ decisions.

* The i^{th} decision involves determining which vertex in V_{i+1},

is to be on the path.

* $P(i,j)$ be a minimum cost path from vertex j in V_i to vertex t.

* $\text{Cost}(i,j)$ be the cost of the path.

* find cost of path using the formula

$$\boxed{\begin{array}{l} \text{cost}(i,j) = \min\{c(j,l) + \text{cost}(i+1,l)\} \\ \qquad\qquad l \in V_{i+1} \\ \qquad\qquad (j,l) \in E \end{array}}$$

* $\text{Cost}(k-1,j) = \text{if } c(j,t) \in E$

* $\text{Cost}(k-1,j) = \infty \text{ if } c(j,t) \notin E$

* Shortest distance between 'S' & 't' using following formula $\text{cost}(1,S)$ by first computing

$$\boxed{\begin{array}{l} \text{Cost}(k-2,j) \ \forall j \in V_{k-2} \\ \text{Cost}(k-3,j) \ \forall j \in V_{k-3} \\ \text{Cost}(1,S) \end{array}}$$

Algorithm for multistage graph using forward Approach:-

```
Void FGraph (Graph G, int k, int n, int p[])
  {// the input is a k-stage graph G = (V,E) with n vertices.
    // Indexed in order of stages
    // E is a set of edges and c[i,j] is the cost of [i,j]
    // P [1:k] in a minimum cost path vertex
{
    Cost [n] = 0.0; // cost of vertex n is 0
    for (j = n-1; j >= 1; j--)
```

```
{ // compute Cost[j]
  // let r be a vertex such that (j,r) is an edge of G &
  // c[j,r] + Cost[r] is minimum
     Cost[j] = c[j,r] + cost[r];
     d[j] = r;
}
// find a minimum Cost path
     P[1] = 1;
     P[k] = n;
   for (j=2 ; j<=k-1; j+1)
        P[j] = d[p[j-1]];
  }
}
```

Multistage graph Using backward approach:-

- * Let $b_p(i,j)$ a minimum Cost path from Vertex S to a j in V_i
- * $b_{cost}(i,j)$ be the cost of $b_p(i,j)$
- * Shortest path from Source 's' to Sink 't' using backward approach

$$bcost(i,j) = \min_{\substack{l \in V_{i-1} \\ (l,j) \in E}} \{bcost((i-1), l) + c(l, i)\}$$

- * $bcost(2,j) = c(1,j)$ if $(1,j) \in E$

  $bcost(2,j) = \infty$ if $(1,j) \notin E$

Algorithm for Multistage graph using backward approach

```
Void BGraph (Graph G, int k, int n, int P[])
  {
   bcost [1] = 0.0;
     // Cost of Vertex 1 is zero
    for (j=2 ; j<n ; j++)
      { // Compute bcost [j]
        // let r be such that (r,j) is an edge
        // of G and bcost [r] + c[r,j] is minimum
    bcost [j] = bcost [r] + c [r,j];
    d[j] =r;
    }
     // find a minimum Cost path
        P[1]=1;
        P[k]=n;
      for (j=k-1 ; j>2 ; j--)
          P[j]= d [p[j+1]];
   }
```

Complexity of Multistage graph for both forward and backward approach:

Time Complexity:-

finding the minimum Cost for each & every stage ⇒ $\Theta(|V|+|E|)$

Shortest path from Source 's' to Sink 't' ⇒ $\Theta(k)$.

Space Complexity:-

Storage Space for Cost array, cost[] = n location
Storage space for minimum cost path p[] = n location
Storage space for decision array d[] = n location
Storage Space for stage k = 1
Storage space for variable 'n' = 1
Control Variable j = 1

∴ Total storage space = 3n+3 = 3 (n+1)

Coin-changing problem

Goal: To make change for an amount using least number of Coins from the available denominations.

Maintain 3 types of array values.

(i) array d[ ] — represent the denominations array

(ii) C[p] → minimum number of Coins required to make change for an amount p using given denomination Coins.
where $0 <= p <= A$.
The array Contains (A+1) elements

(iii) Create an array S[ ] having A+1 elements
Such that S[p] will Contain the index of the first coin in an optimal Solution for making change of an amount p.
where $0 <= p <= A$

So, $A = 6$
$n = 3$
$1 <= i <= n$
$0 <= p <= A$.

To solve this problem we will use the following formula

$$C[p] = \begin{cases} 0 & \text{if } p = 0 \\ \min_{i : d_i \le p} \{1 + C[p - d_i]\} & \text{if } p > 0 \end{cases}$$

Where C[p] denotes the minimum number of Coins required to make change for an amount p using given denominations Coins d[i] where Selected denomination is not greater than amount p.

Formula:-

if $d[i] <= p$ then

if $1+c[p-d[i]] < min$ then

$min = 1+c[p-d[i]]$

$Coin = i$

	1	2	3
$d[i]$	1	2	5

No. of min. Coi

	0	1	2	3	4	5	6
$C[p]$	0	1	1	2	2	1	②

	0	1	2	3	4	5	6
$S[p]$	0	1	2	1	2	3	1

Case 1:

when $p=1$, $min=\infty$ $i=1$ $Coin=0$

(i) is $d[i] <= P$

$d[1] <= 1$

$1 <= 1$

Yes

then check $1+c[1-d[1]] < min$

$\Rightarrow 1+c[1-1] < \infty$

$\Rightarrow 1+0 < \infty$

$\Rightarrow 1 < \infty$

Yes

$\therefore$ Set $min=1$, $Coin=1$ and increment i

$\therefore$ we get

$P=1$, $min=1$, $i=2$, $Coin=1$

Again check

is $d[2] <= P$

$2 <= 1$

No. So just increment i

we get now

for $P=1$ $min=1$, $i=3$ $Coin=1$

Again check is $d[3] <= P$

$5 <= 1$

No

So, we have checked all the denominations for the amount $p=1$

$\therefore$ Set $C[1] = min = 1$

$S[1] = Coin = 1$

$|||^{larly}$ do for amount $p=2$.

Case 2:

$P=2$ $min=\infty$ $i=1$ $Coin=1$

is $d[1] <= 2$ $\Rightarrow$ is $1+c[2-d[1]] < m$

$1 <= 2$ $\quad$ $1+c[2-1]] < m$

Yes

$1+1 < \infty$

$2 < \infty$

Yes

$\therefore$ Set $min=2$

$Coin=2$

$i=2$

$\therefore$ $P=2$ $min=\cancel{\infty}2$ $i=2$ $Coin=1$

is $d[2] <= 2$ $\Rightarrow$ is $1+c[2-2] < 2$

$2 <= 2$ $\quad$ $1+c[0] < 2$

Yes

$1+0 < 2$

Yes

$\therefore$ Set $min=1$

$Coin=2$

$i=3$

$P=2$ $min=1$ $i=3$ $Coin=2$

is $d[3] <= 2$

$5 <= 2$ $\quad$ $\therefore$ Set $C[2] = min = 1$

No $\quad$ $S[2] = Coin = 2$

III^{ly} Calculate for p=3, 4, 5 & 6 and set C[p] and S[p] values.

Procedure

```
Step 1: Set a = A
Step 2: if a>0 then
            Print d[S[a]]
        else Stop
Step 3: Set a = a - d[S[a]]
        Repeat step 2
```

eg/.. A=6

is 6>0 then print d[S[a]]

Yes.

d[S[6]]

=d[1] = 1

Now Set a = 6-1

= 5

again check 5>0

Yes then print d[S[5]]

d[3]

Print 5

Now set a = 5-5 = 0.

∴ Coins are 1,5

So, To make change of amount Rs 6 the Shopkeeper will need 2 minimum coins & the coins will be Rs 1 & Rs 5.

Greedy Technique:-

Container Loading Problem:-

A larger ship is to be loaded with cargo. The cargo is containerized and all containers are the same size. Different containers may have different weights. w_i be the weight of the i^{th} container $\boxed{1 \le i \le n}$. The cargo capacity is C. Based on the greedy concept load the ship with the maximum number of containers.

Procedure for Container Loading:-

- The ship may be loaded in stages.
- At each stage we need to select a container to load.
- Select the container with least weight.
- check each time before the loading of container

  C[i].weight <= capacity
- The process is repeated until it reaches cargo capacity 'C'.

Feasible Solution:-

✓ Every set of x_i's (container) that satisfies the constraint $\sum_{i=1}^{n} w_i x_i < C$ then x_i is assigned to value 1. otherwise it is assigned to ϕ.

$$\boxed{\sum_{i=1}^{n} w_i x_i <= C, \ x_i \in \{0,1\}, \ 1 \le i \le n}$$

where w_i = weight of container 'i'

x_i - Value of the container is assigned 0 or 1

C - Cargo capacity   n - No.. of containers in cargo.

Optimal Solution

✓ Load the ship with the maximum number of Containers

✓ Every feasible solution that maximizes the $\sum_{i=1}^{n} x_i$ function is an optimal Solution.

Algorithm for Container Loading:-

```
Void Container_loading (Container *c, int Capacity, int no/. of
                             Containers, int *x)
  // C- Capacity of Containers
  {
   // Sort the Container in ascending order of their weights
  Sort (c, no/.. of Containers);
      n= number of Containers;
   //initialize the Variable x
      for (i=1; i<=n; i++)
        x[i]=0;
         i=1;
     // Select the Container in order of their weight
     while (i<=n && c[i] <= Capacity)
     { //c[i] weight of the Container i
        x[i]=1; // the Container is loaded.
     Capacity = Capacity - c[i]; //remaining Capacity of the cargo
      i++;
     }
  }
```

Complexity of Container Loading:- [Space Complexity]

Space for weight of n containers stored in an array c[] = n locations

Space for indication of Container loading stored in an array x[] = n location

Capacity of Cargo c = 1

Number of Containers 'n' = 1

Control variable 'i' = 1

∴ Total space required = 2n+3

Time Complexity:-

(1) Sorting the Containers increasing order of their weight (using merge sort) = $O(\log n)$

(2) Algorithm for loading of Container = $O(n)$

∴ Time Complexity = $O(n \log n)$.

Example

The cargo contains 8 containers, $n=8$, weight of the containers $\{w_1, w_2, w_3 \ldots w_8\} = \{100, 200, 50, 90, 150, 50, 20, 80\}$ & capacity of the cargo is 400. Find the optimal solution for loading the containers.

(1) Arrange the containers in ascending order of their weights $\{7, 3, 6, 8, 4, 1, 5, 2\} = \{20, 50, 50, 80, 90, 100, 150, 200\}$

(2) In stage 1, the Container 7 is Selected, whose weight is 20.

check the Constraint $\sum w_i <= C$

(i.e) $20 <= 400$ True

So the Container 7 is loaded $\therefore x_7$ is assigned to 1.

Value of Solution Set $= \{0,0,0,0,0,0,1,0\}$

(3) In stage 2, the Container 3 is Selected whose weight is 50

$\therefore$ by checking the Constraint we get $\sum w_i <= C$

(i.e) $50+20 <= 400$

$70 <= 400$ True.

$\therefore x_3$ is Set to 1.

$\therefore$ Value of Solution Set $= \{0,0,1,0,0,0,1,0\}$

[illegible]

(4) stage 3, Container 6 is loaded & Solution Set $= \{0,0,1,0,0,1,1,0\}$

[illegible]

(5) stage 4, Container 8 is loaded & solution set $= \{0,0,1,0,0,1,1,1\}$

$290 < 400$

(6) stage 5, Container 4 is loaded & Solution Set $= \{0,0,1,1,0,1,1,1\}$

(7) Stage 6, Container 1 is loaded & Solution Set $= \{1,0,1,1,0,1,1,1\}$

$390 < 400$

(8) Stage 7, Container 5 is Selected, whose weight is 150

Check the Constraint $\sum w_i <= C$ (i.e) $390+150 <= 400$

$540 <= 400$

false, So Container 5 is not loaded.

$\therefore$ optimal Solution $\sum x_i = 6$ //

UNIT

IV

UNIT-IV

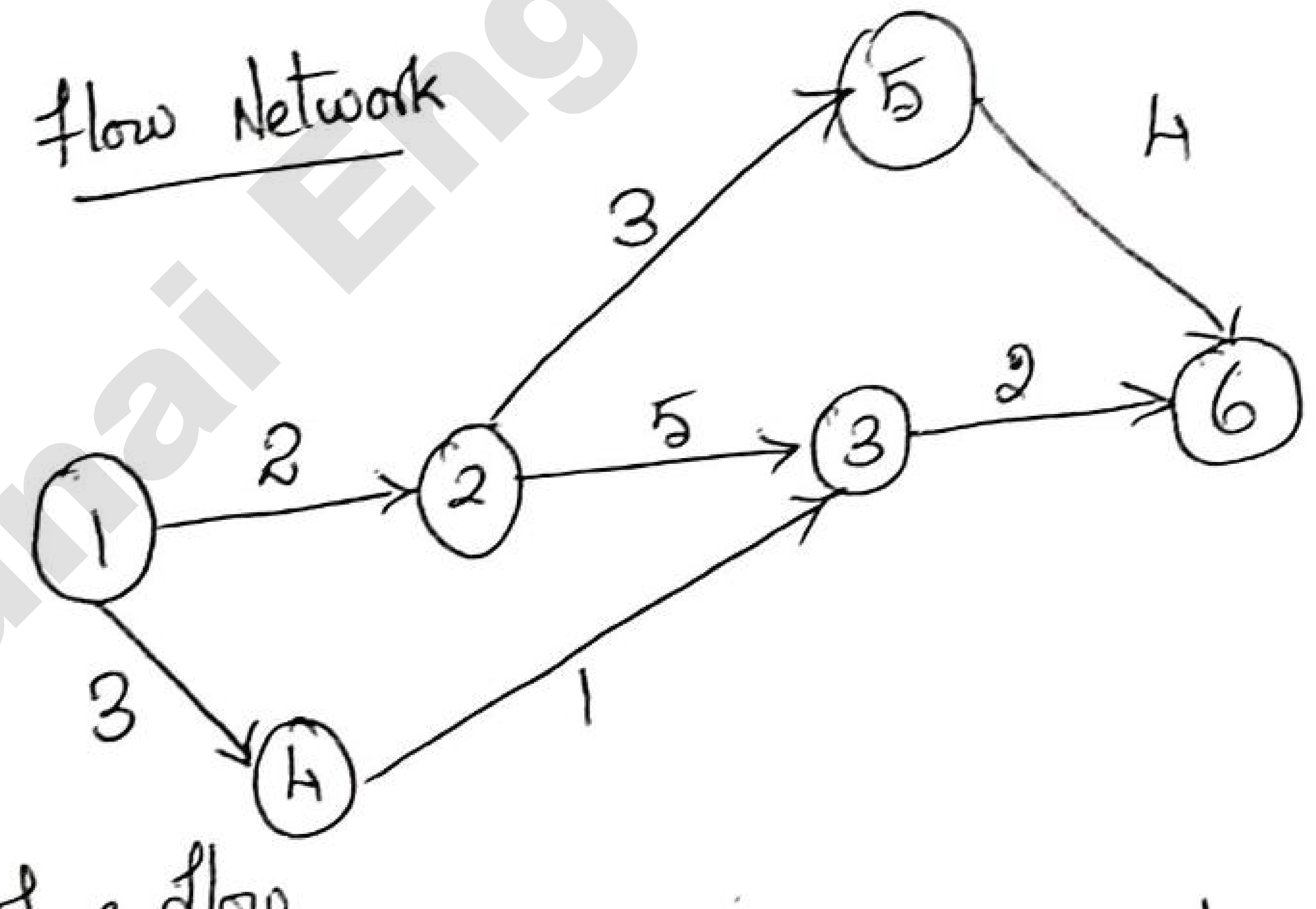
Maximum flow Problem:-

Problem Statement: problem of maximizing the flow of a material through a transportation network (eg,. Pipeline System, Communications or transportation networks)

formally represented by a Connected weighted digraph with n vertices numbered from 1 to n with the following Properties

- * Contains exactly 1 vertex with no entering edges, called the Source
- * Contains exactly 1 vertex with no leaving edges, called the Sink
- * has positive integer weight U_{ij} on each directed edge (i, j), called the edge capacity, indicating the upper bound on the amount of the material that can be sent from i to j through this edge.

Example of flow Network



Definition of a flow

A flow is an assignment of real numbers x_{ij} to edges (i, j) of a given network that satisfy the following

* The total amount of material entering an intermediate vertex must be equal to the total amount of the material leaving vertex

* flow - Conservation requirements

$\sum x_{ji} = \sum x_{ij}$ for (i=2,3, . . n-1)

$j:(j,i) \in E$ $\quad$ $j:(i,j) \in E$

* Capacity Constraints

$0 \le x_{ij} \le u_{ij}$ for every edge $(i,j) \in E$

Since no material can be lost or added to by going through intermediate vertices of the network, the total material amount leaving the Source must end up at the Sink.

$\sum x_{ij} = \sum x_{jn}$ | $j:(1,j) \in E$ $\quad$ $j:(j,n) \in E$

The value of the flow is defined as the total outflow from the Source (= the total inflow into the Sink).

The maximum flow problem is to find a flow of the largest value for a given n/w.

Maximum - flow problem as LP problem

Maximize $v = \sum x_{1j}$ $\quad$ $j:(1,j) \in E$

Subject to

$\sum x_{ji} - \sum x_{ij} = 0$ for $i = 2,3, . . n-1$

$j:(j,i) \in E$ $\quad$ $j:(i,j) \in E$

$0 \le x_{ij} \le u_{ij}$ for every edge $(i,j) \in E$.

An augmenting path is a simple path from Source to Sink which do not include any cycles and that pass only through Positive weighted edges.

A residual network graph indicates how much more flow is allowed in each edge in the network graph. If there are no augmenting paths possible from S to T, then the flow is maximum.

Algorithm:- Ford - Fulkerson (Graph G, Node S, Node T)

```
Initialise flow in all edges to 0
While (there exists an augmenting path (P) b/w S and T in residual network graph);
    Augment flow between S to T along the path P.
    Update residual network graph.
    return.
```

Finding a flow - augmenting Path

To find a flow-augmenting path for a flow x, Consider paths from Source to Sink in the Underlying undirected graph in which any 2 Consecutive vertices i, j are either:

- Connected by a directed edge (i to j) with some + Unused Capacity $r_{ij} = U_{ij} - x_{ij}$. - known as forward edge

- Connected by a directed edge (j to i) with positive flow x_{ji}
  - known as backward edge (←)

If a flow augmenting path is found, the current flow can be increased by r units by increasing x_{ij} by r on each forward edge & decreasing x_{ji} by r on each backward edge.

where $r = \min(r_{ij}$ on all forward edges, x_{ji} on all backward edges).

Disadvantage

→ The augmenting path method doesn't prescibe a specific way for generating flow augmenting paths.

→ selecting a bad sequence of augmenting paths could impact the method's efficiency.

Time Efficiency

* The number of augmenting paths needed by the shortest augmenting - path algorithms never exceeds $nm/2$, where n and m are the number of vertices & edges respectively.

* Running time is based on selection of augmenting paths. Maximum running time is $O(|E|+E)$

* ∵ the time required to find shortest augmenting path by BFS is $O(n+m) = O(m)$. for

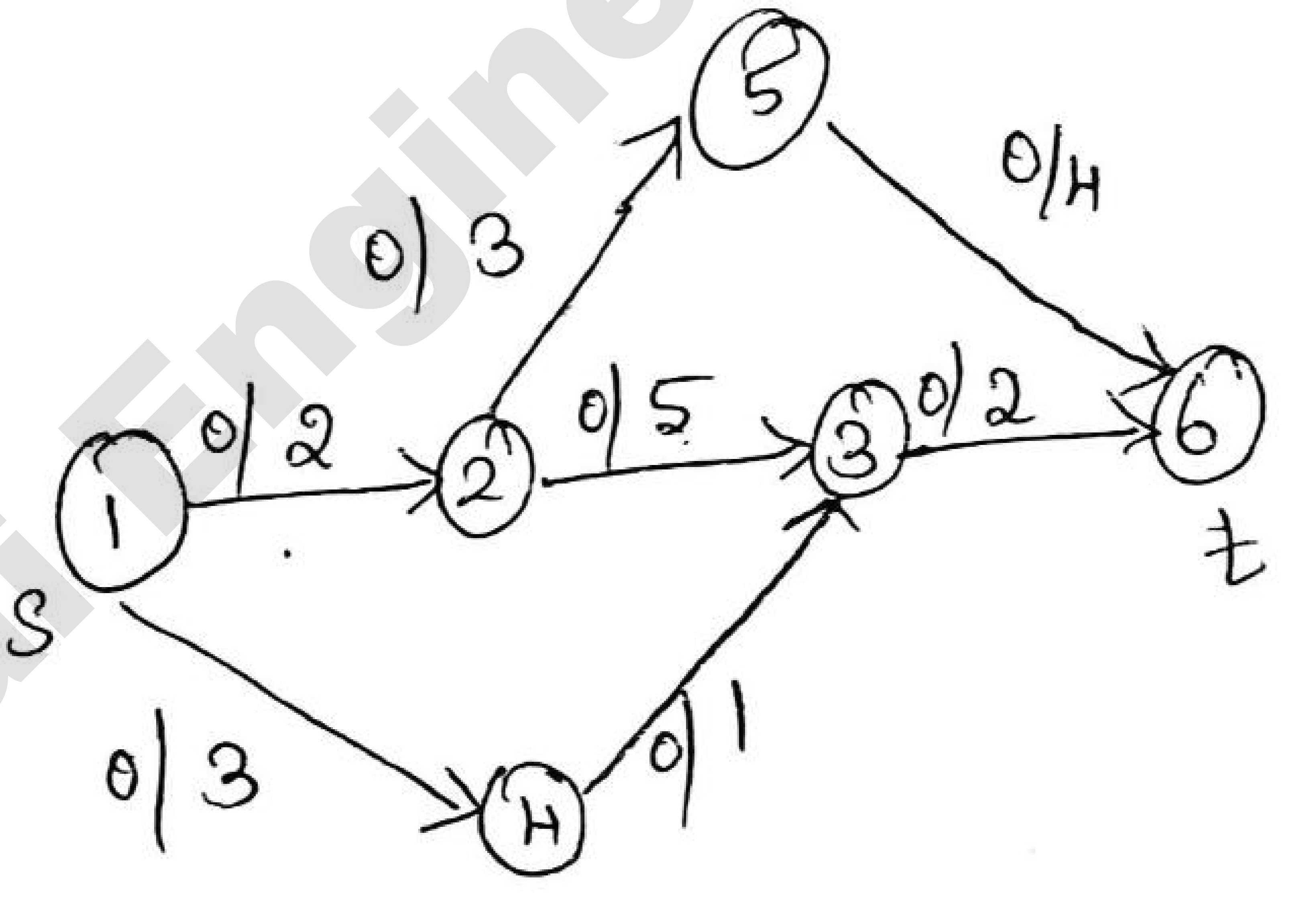
networks represented by their adjacency lists, the time efficiency of the shortest -augmenting path algorithm is in $O(nm^2)$.

Steps

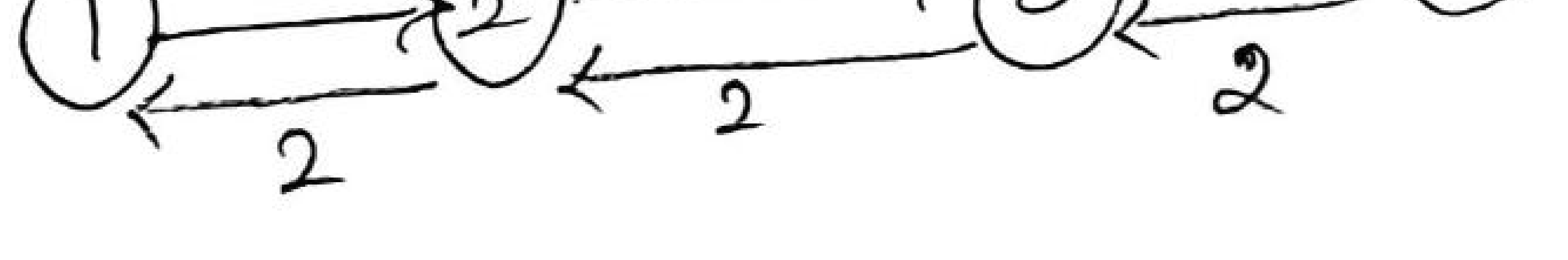
1. Initialize with flow zero on every edge
2. Create residual graph (G_f)
3. find an augmented path in G_f
4. Update flows in G.
5. Repeat steps 2 to 4 until no augmented path is found.

eg1:- Problem

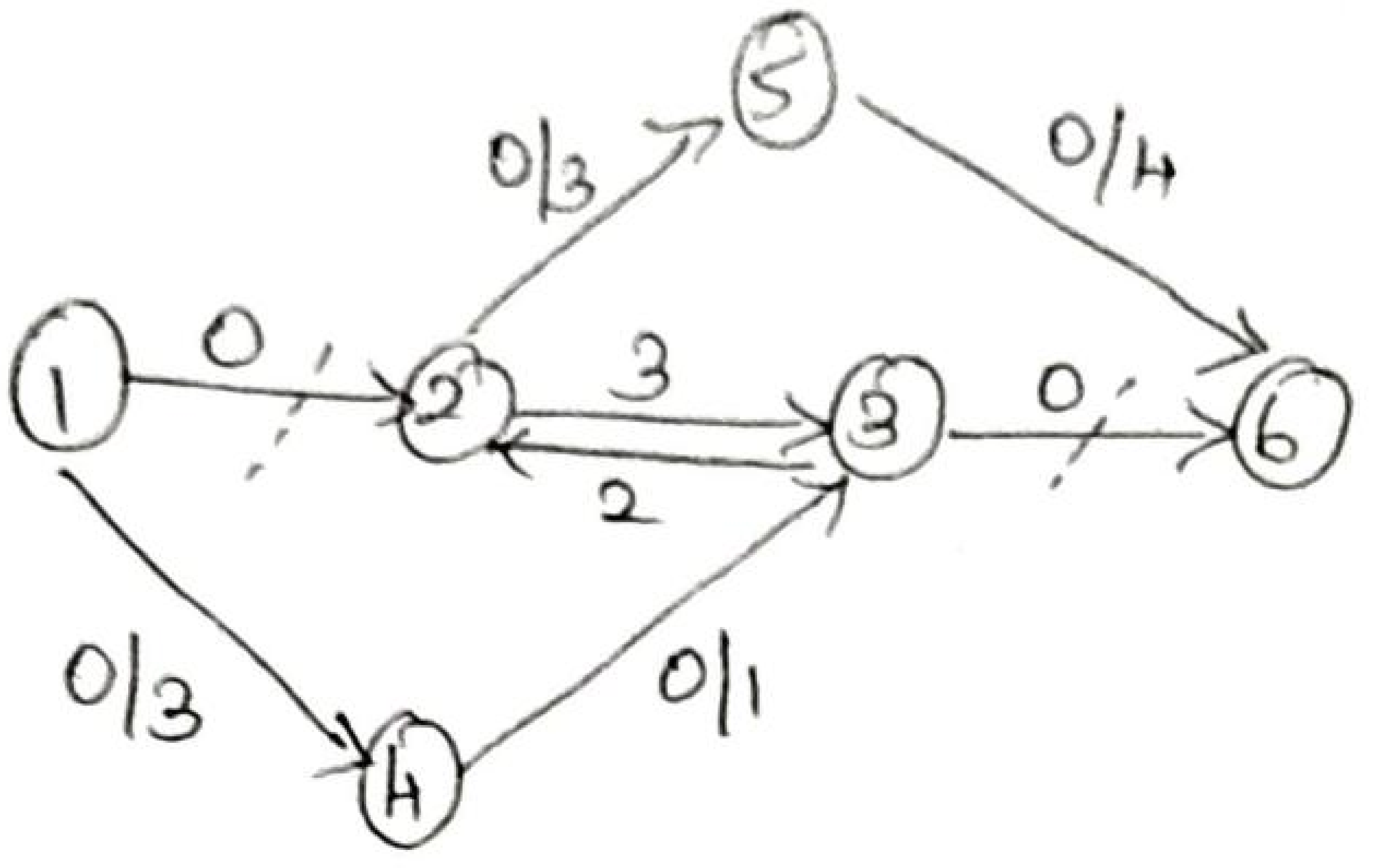
Step 1:



Step 2: Consider the augmented path 1-2-3-6.

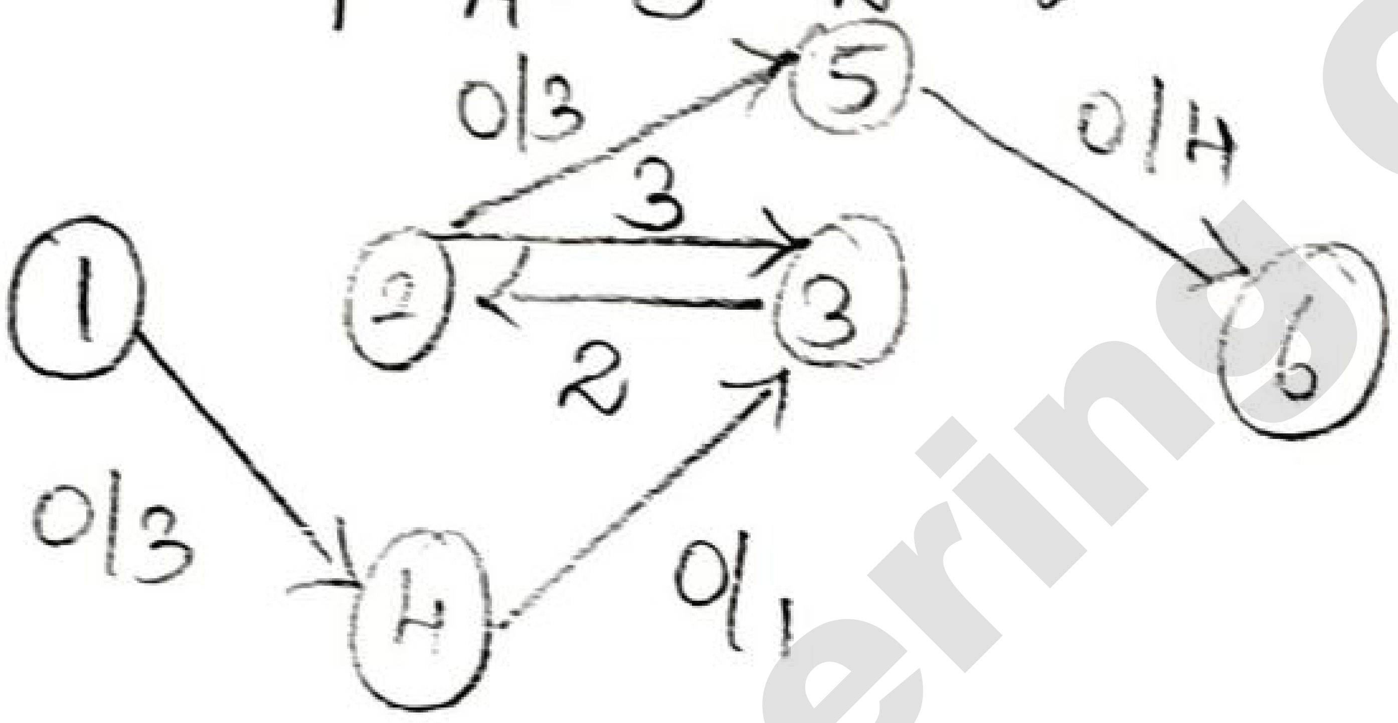


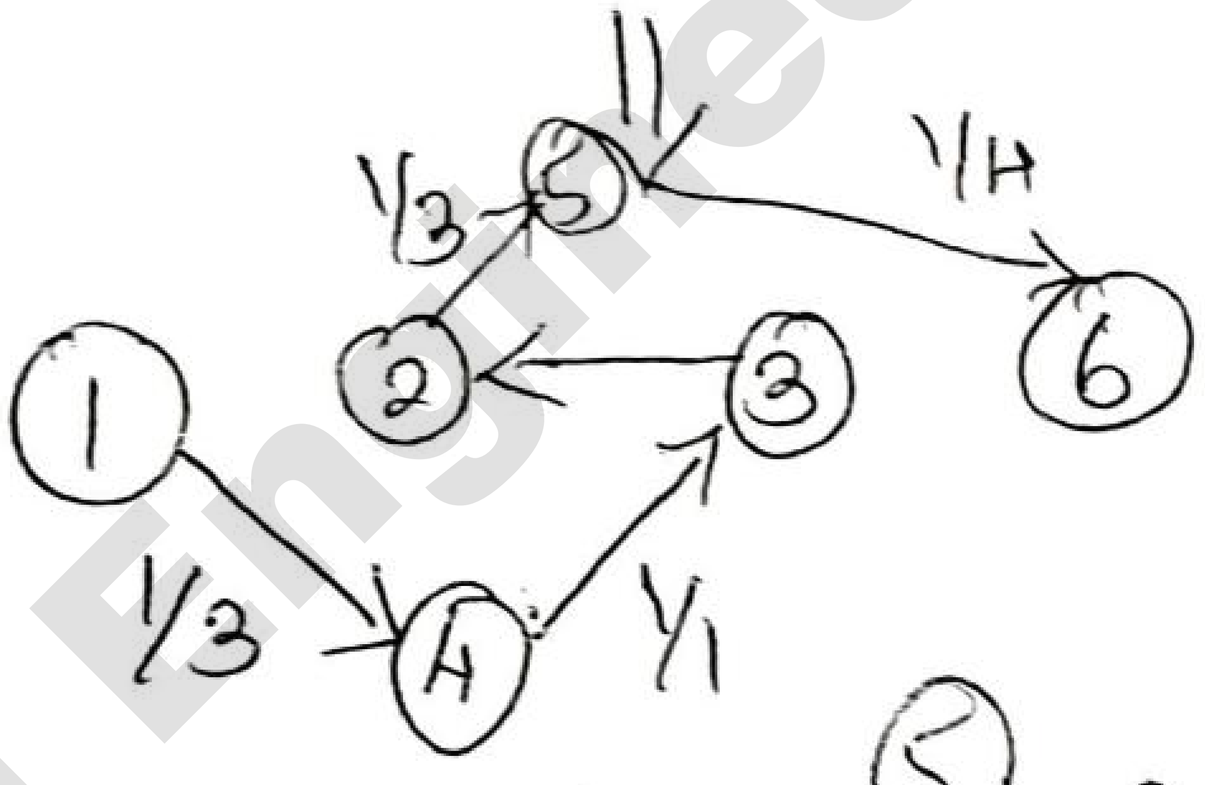
Step 3:

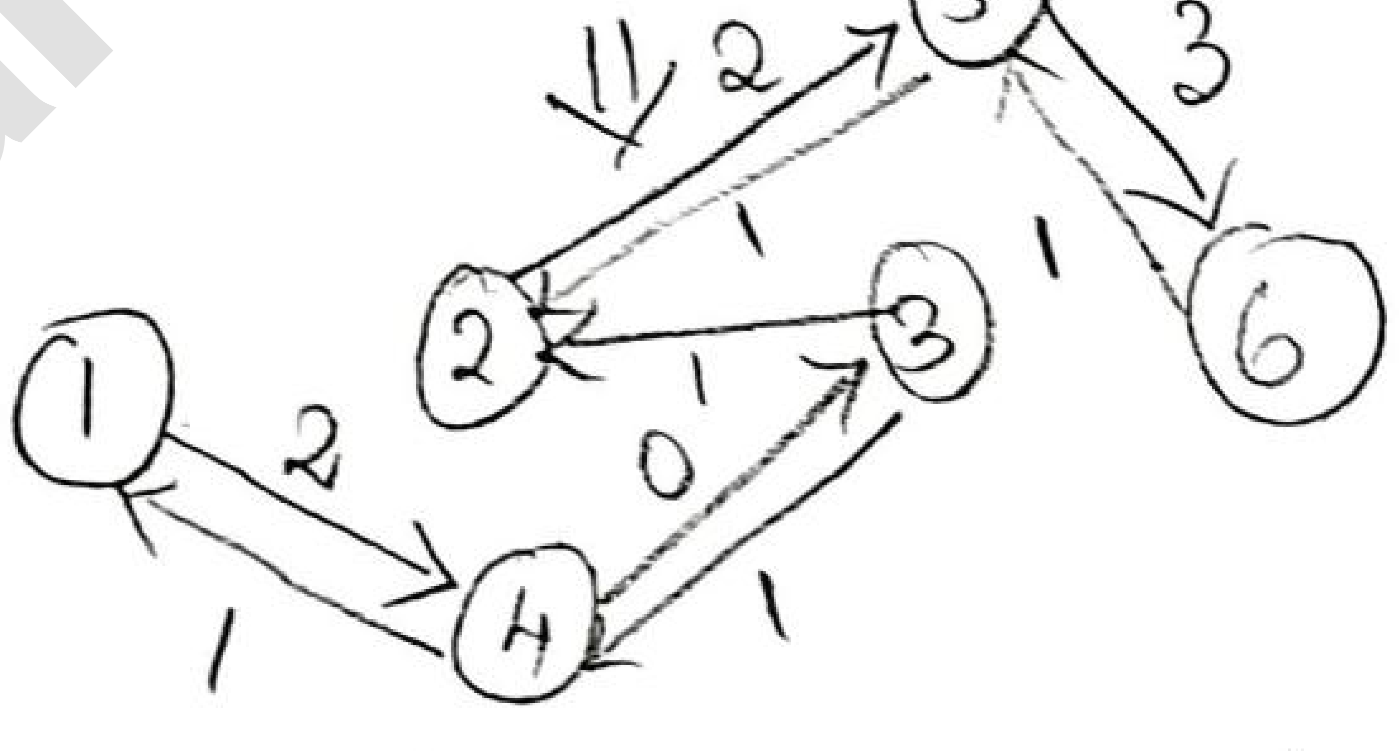


Step 4: Consider another augmented path

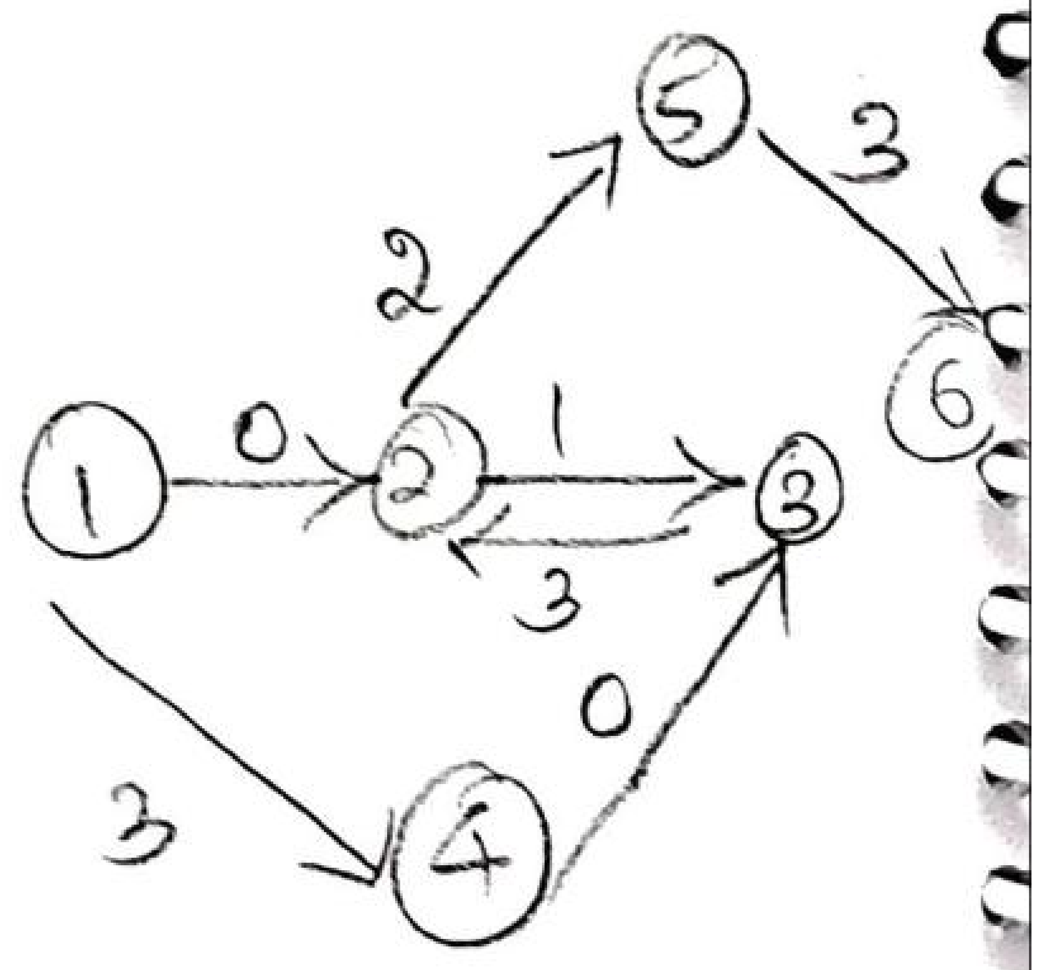
1-4-3-2-5-6.







=>



No augmented path So calculate maximum flow So Maximum flow = 2+1 = 3//

Definition of a cut

Let x be a set of vertices in a network that includes its source but does not include its sink and let x, the complement of x, be the rest of the vertices including the sink. The cut included by this partition of the vertices is the set of all the edges with a tail in x and a head in x.

Capacity of a cut is defined as the sum of capacities of the edges that compose the cut.

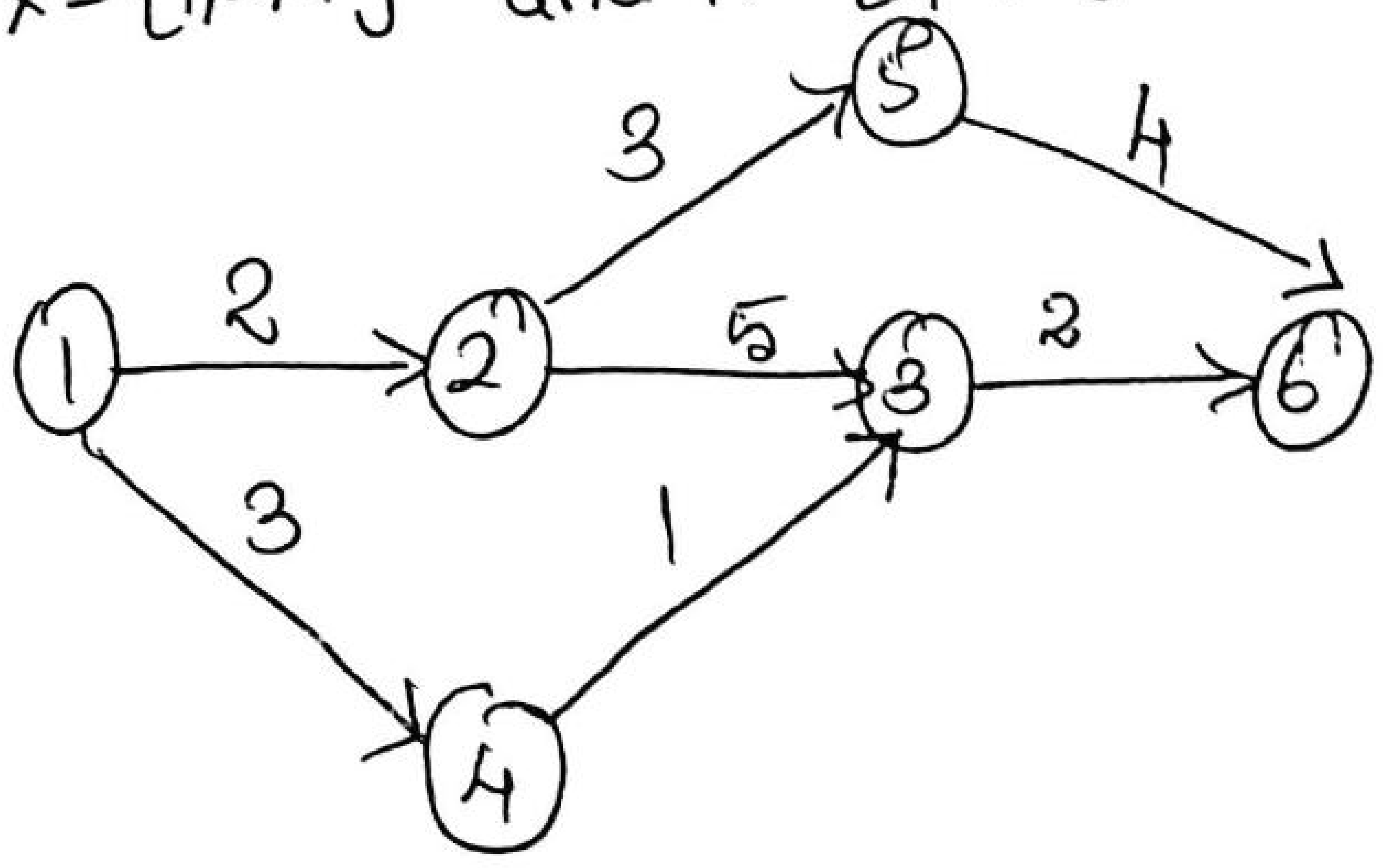
* The cut and its capacity is denoted by C(x,x) and

ç(x,x)

* Note that if all the edges of a cut were deleted from the network, there would be no directed path from source to sink.

* Minimum cut is a cut of the smallest capacity in a given network.

eg/.. if x={1} and x={2,3,4,5,6}, C(x,x) ={(1,2) (1,4)} ç=5
if x={1,2,3,4,5} and x={6} C(x,x)= {(3,6),(5,6)},
ç=6
if x={1,2,4} and x={3,5,6} C(x,x)={ (2,3), (2,5), (4,3) } ç=9



State the Max-flow Min-cut theorem :- (Nov/Dec 2016)
May/June 2016)

Max-flow Min Cut theorem

* The value of maximum flow in a network is equal to the capacity of its minimum cut.

* The shortest augmenting path algorithm yields both a maximum flow and minimum cut.

◇ maximum flow is the final flow produced by the algorithm

◇ minimum cut is formed by all the edges from the labeled vertices to unlabeled vertices on the last iteration of the algorithm.

All the edges from the labeled to unlabeled vertices are full (i.e) their flow amounts are equal to the edge capacities, while all the edges from the unlabeled to labeled vertices, if any have 0 flow amounts on them.

Statement of Theorem

It states that the maximum flow through the network from a given source to a given sink is exactly the sum of the edge weights that, if removed, would totally disconnect the source from sink.

Applications: Network connectivity, availability, reliability. (computer science)
Bipartite matching (Mathematics)
Scheduling.

Algorithm :-

Input : A set of n men & Set of n women along with ranking of women by each men & ranking of the men by each Women

O/p :- A stable Marriage matching.

Step 0 : Start with all the men & Women by being free.

Step 1 : While there are free men, arbitrarily select one of them and do the following

Proposal – The Selected free man m proposes to w, the next woman on his preference list (who is the highest ranked woman who has not rejected him before).

Response – If w is free, she accepts the proposal to be matched with m. If she is not free, she compares m with her Current mate. If she prefers m to him, she accepts m's proposal, making her former mate free. otherwise she simply rejects m's proposal leaving m free.

Step 2 : Returns the set of n matched pairs.

Properties of stable Marriage problem or Gale shapley Algorithm :-

1. Matching between all the men & Women can be found
2. Matching Could be stable.

Example:

Consider the preference list for the men & Women & find the Solution for stable matching.

Men's Preference list

Frank	Kate	Mary	Rhea ③	Jill
Dennis	Mary ②	Jill ⑤	Rhea	Kate
Mac	Kate ②	Rhea	Jill	Mary
Charlie	Rhea ③	Mary ④	Kate	Jill

Women's Preference List

Rhea	Frank	Mac	Dennis ③	Charlie
Mary	Mac	Charlie	Dennis	Frank
Kate	Dennis	Mac	Charlie	Frank
Jill	Charlie	Dennis	Frank	Mac

Step 1: Prepare Ranking Matrix Using the Preference list.

	Rhea	Mary	Kate	Jill
Frank	1,3	4,2	4,1	3,4
Dennis	3,3	3,1	1,4	2,2
Mac	2,2	1,4	2,1	4,3
Charlie	4,1	2,2	3,3	1,4

Step 2 :

Each men proposes to women In his preference list so that

✓ frank proposes Kate But she rejects

✓ Dennis proposes Mary she accepts

✓ Mac proposes Kate she accepts

✓ Charlie proposes Rhea she accepts.

So, frank does not have matching pair

After many iterations we get

frank → Rhea
Dennis – Jill
Mac – Kate
charlie- Mary.

Since all the people had found the stable matching, the algorithm Terminates.

Time Complexity

There are n men ($m_1, m_2 .. m_n$) & n women ($w_1, w_2, .. W_n$)

At each step in the algorithm from the starting to the finishing stage a man will make a propose to woman.

→ Each new proposal involves a new pair

→ So, there can be at most n^2 pairs.

→ ∴ Worst case Time efficiency can be $n^2 \Rightarrow O(n^2)$.

Summarize Simplex Method / Describe in detail the simplex algorithm method (or) List the steps in simplex Method & give the efficiency of the Same. [APR/MAY '17]

[Apr/May 2018] [Nov/Dec 2017] [Nov/Dec 2016].

Simplex Method

→ Algorithm for solving the linear programming problem is called Simplex method.

→ Linear programming Problem is of the general form

maximize (or) minimize $c_1x_1 + c_nx_n$

Subject to $a_{i1}x_n + \dots + a_{in}x_n \le$ (or $\ge$ or $=$) b_i

for $i = 1, \dots n$.

$x_1 \ge 0 \quad \dots x_n \ge 0$ – non-negativity Constraints

→ Linear programming (LP) problem is to optimize a linear function of several variables subject to linear Constraints:

maximize (or) minimize $c_1x_1 + \dots + c_nx_n$

→ Any point (x, y) that satisfies all the Constraints of the Problem is called feasible solution.

→ Variables u and v transforming inequality Constraints into equality Constraints are called slack variables

→ A basic solution for which all variables are non negative is called basic feasible solution

Applications of Linear programming

* Airline Crew Scheduling
* Transportation & Communication network planning
* Oil exploration & refining
* Industrial production optimization

The function $z = c_1x_1 + \dots + c_nx_n$ is called the objective function.

Extreme point Theorem

Any LP problem with a non empty bounded feasible region has an optimal solution. An optimal solution can always be found at an extreme point of the problem's feasible region.

3 possible outcomes in solving an LP problem

* A finite optimal solution, which may not be unique.

* Unbounded : The objective function of maximization (minimization) LP problem is unbounded from above (below) on its feasible region.

* Infeasible : There are no points satisfying all the constraints (i.e) the constraints are contradictory.

Summarize or Outline of the Simplex method

Step 0 : Initialization – present a given LP problem in standard form and set up initial tableau.

Step 1 : [optimality test] – If all the entries in the objective row are non-negative – stop : the tableau represents an optimal solution.

Step 2 : [Find entering variable] – Select (the most) negative entry in the objective row. Mark its column to indicate the entering variable & pivot column.

Step 3 : [Find departing variable] – for each positive entry in the pivot column, calculate the θ-ratio by dividing the row's entry in the rightmost Column by its entry in the pivot column. (If there are no positive entries in the pivot column - Stop : the Problem is Unbounded). Find the row with the Smallest E-ratio, mark this row to indicate the departing variable & the Pivot row.

Step 4 : [Form the next tableau]

Divide all the entries in the pivot row by its entry in the pivot Column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the Pivot Column of the row in question. Replace the label of the Pivot row by the variable's name of the pivot column & go back to Step 1.

Notes on Simplex method

* finding an initial basic feasible Solution may Pose a problem.
* Theoretical possibility of cycling.
* Typical number of iterations is between m and 3m, where m is the number of equality Constraints in the standard form.

Number of operations per iteration is O(nm).

* Worst case efficiency is exponential.

eg/.. Maximize $6x_1+5x_2$
Subject to $x_1+x_2 \le 5$
$3x_1+2x_2 \le 12$
$x_1, x_2 \ge 0$ using tabular form.

Solution :-

Convert inequalities to equalities by adding slack variables

$\therefore$ Maximize $6x_1+5x_2+0S_1+0S_2$.

$x_1+x_2+S_1=5$
$3x_1+2x_2+S_2=12$
$x_1, x_2, S_1, S_2 \ge 0$

Iteration 1:

obj Basis	6 x_1	5 x_2	0 S_1	0 S_2	RHS
0 S_1	1	1	1	0	5
0 S_2	3	2	0	1	12
$C_j - Z_j$	6	5	0	0	0

Compute $C_j - Z_j$:

$C_1 - Z_1 = 6 - [(S_1 * x_1) + (S_2 * x_1)] = 6$
$C_2 - Z_2 = 5 - [(S_1 * x_2) + (S_2 * x_2)] = 5$
$C_3 - Z_2 = 0 - [(S_1 * S_1) + (S_2 * S_1)] = 0$
$C_4 - Z_4 = 0 - [(S_1 * S_2) + (S_2 * S_2)] = 0$
$C_5 - Z_5 = [(S_1 * 5) + (S_2 * 12)] = 0$.

Identify the largest possible $C_j - Z_j$ value in iteration 1.

→ Here $C_j - Z_j$ for $x_1 = 6$

∴ x_1 enters the basis ↑

→ Find the leaving variable either S_1 or S_2. Create another Column θ.

$$\theta = \frac{\text{RHS}}{\text{Corresponding element of entering column}} \quad = \frac{5}{1} = 5$$

$$= \frac{12}{S_2} = \frac{12}{3} = 4.$$

min(5,4)=4 ∴ S_2 leaves the basis →

Entering variable = x_1
leaving variable = S_2

→ Choose the pivot row & pivot Element.

Pivot row = S_2 = Row Corresponding to leaving variable.

Pivot Element = 3 = Intersection of entering column & leaving row.

→ Perform row operation - Divide the pivot row by Pivot Element

	x_1	x_2	S_1	S_2		
S_1	(1-1)	(1-2/3)	(1-0)	(0-1/3)	(5-4)=1	3 →
x_1	1	2/3 ↑	0	1/3	4	6
$C_j - Z_j$	0	1	0	-2	24	

After Iteration 2

Pivot Element = $1/3$

Pivot row = S1

Entering Variable = x_2

Leaving Variable = S1

Perform Row operation = Pivot row / pivot Element.

	X1	X2	S1	S2	RHS
X2	$0/\tfrac{1}{3}$	$\tfrac{1}{3}\big/\tfrac{1}{3}$	$1/\tfrac{1}{3}$	$-\tfrac{1}{3}\big/\tfrac{1}{3}$	$1/\tfrac{1}{3}$
X1					

⇓

	X_1	X_2	S_1	S_2	
X_2	0	1	3	-1	3
X1	$1-\tfrac{2}{3}(0)$	$\tfrac{2}{3}-\tfrac{2}{3}(1)$	$0-\tfrac{2}{3}(3)$	$\tfrac{1}{3}-\tfrac{2}{3}(-1)$	$4-\tfrac{2}{3}(3)$

⇓

	X_1	X_2	S_1	S_2	
X_2	0	1	3	-1	3
X_1	1	0	-2	1	2
C_j-Z_j	0	0	-3	-1	27

Since C_j-Z_j has no positive values, algorithm terminates here ∴ The best solution is

$x_1 = 2$
$x_2 = 3$
$Z = 27$ //

```
Algorithm :- MaximumBipartiteMatching (G)
    // finds a maximum matching in a bipartite graph by a BFS -
       like traversal
    // Input : A bipartite graph $G = V, U, E$
    // output : A maximum - Cardinality Matching M in the input graph.
  Initialize set M of edges with Some valid matching (eg., the
  empty set) Initialize Queue Q with all the free vertices in V
  (in any order)
   while not Empty (Q) do
      $w \leftarrow$ front (Q); Dequeue (Q)
       if $w \in V$
     for every vertex u adjacent to w do
     if u is free
        // augment
         $M \leftarrow M \cup (w,u)$
          $v \leftarrow w$
        while v is labeled do
          $u \leftarrow$ vertex indicated by v's label ; $M \leftarrow M - (v,u)$
           $v \leftarrow$ vertex indicated by u's label ; $M \leftarrow M \cup (v,u)$
        remove all vertex labels
        reinitialize Q with all free vertices in V
        break // exit the for loop
     else // u is matched
```

```
if $(w, u) \notin M$ and u is unlabeled
  label u with w
  Enqueue (Q, u)
  else // $w \in U$ (and matched)
    label the mate v of w with w
    Enqueue (Q, v)
  return M // current Matching is maximum.
```

UNIT V

UNIT-V

3

Give the methods for establishing Lower Bounds:- [Nov/Dec '17]

→ Lower Bound is used to know the best possible efficiency any algorithm solving the problem.

→ It tells us how much improvement to achieve the best for the algorithm.

Methods for Establishing Lower Bounds:-

- * Trivial Lower bounds
- * Information-theoretic arguments (Decision trees)
- * Adversary arguments
- * Problem reduction.

Trivial Lower Bounds

→ Simplest method of obtaining a lower bound class.

→ It is based on Counting the number of items in the problem's input that must be processed and the number of output items that need to be produced.

→ Since any algorithms must be atleast 'read' all the items it needs to process and 'write' all its outputs, Such a Count yields a trivial lower bound.

Examples:-

* finding max element * Sorting * Element Uniqueness.

Conclusions:

* may and may not be useful.

* be careful in deciding how many elements must be processed

Decision trees

A Convenient model of algorithms involving Comparisons in which

→ Internal nodes represent Comparisons

→ leaves represent outcomes.

* Any Comparison-based Sorting algorithm can be represented by a decision tree.

* Number of leaves (outcomes) $\geq n!$

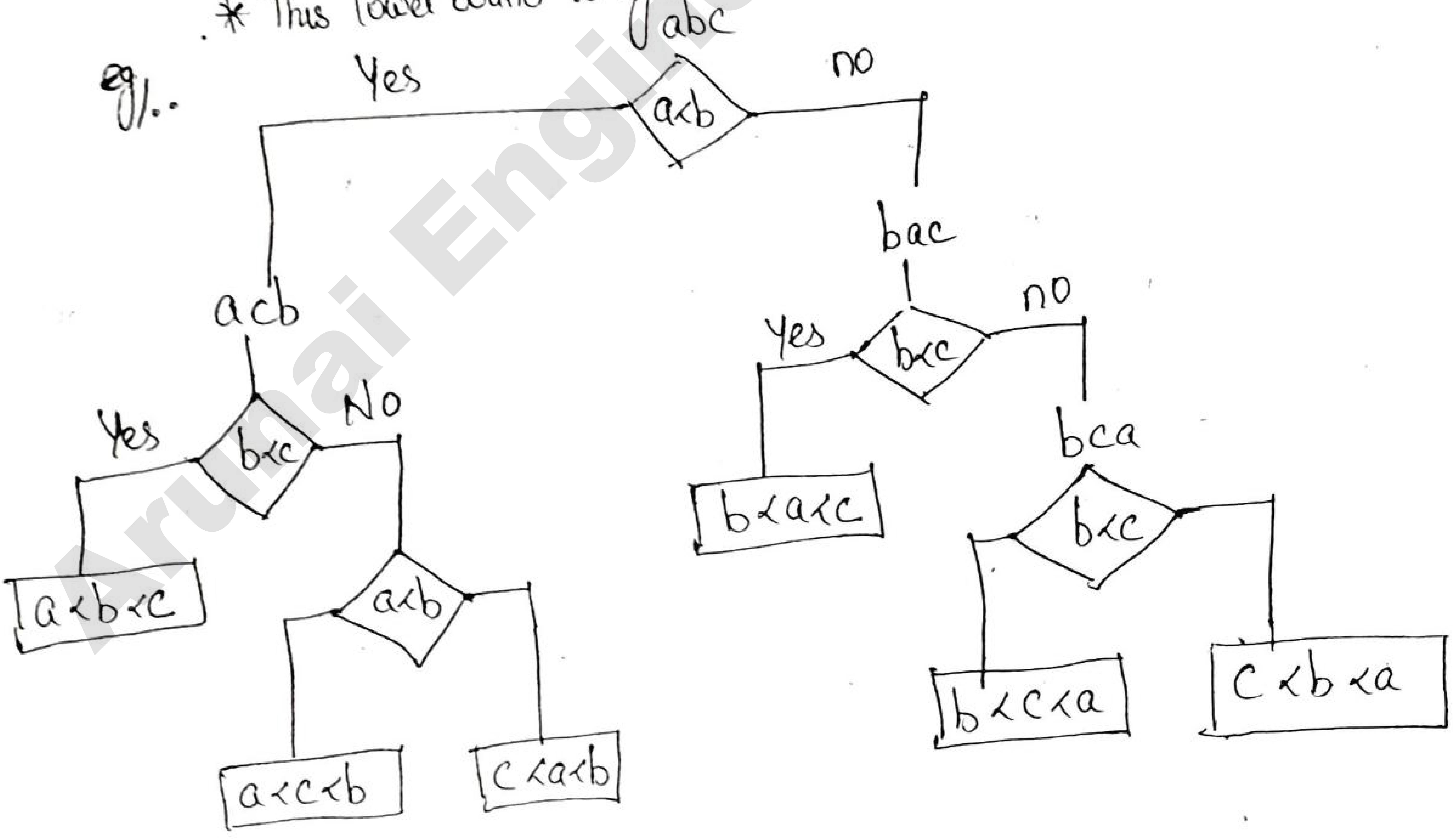
* Height of binary tree with $n!$ leaves $e \geq e\log_2 n!$

* Minimum number of Comparisons in the worst case $\geq e\log_2 n!$ for any Comparison-based Sorting algorithm.

* $e\log_2 n! >> n\log_2 n$.

* This lower bound is tight.

eg/..



Adversary Arguments

It is a method of proving a lower bound by playing role of adversary that makes algorithm work the hardest by adjusting input.

eg1:- "Guessing" a number between 1 and n with yes/no questions.

Adversary: puts the number in a larger of the 2 subsets generated by last question.

eg2: Merging 2 sorted lists of size n.

$a_1 < a_2 < \dots < a_n$ and $b_1 < b_2 < \dots < b_n$.

Adversary: $a_i < b_j$ if $i < j$.

O/P: $b_1 < a_1 < b_2 < a_2 < \dots < b_n < a_n$ requires $2n-1$

Comparisons of adjacent elements.

Problem Reduction

Idea: If problem P is atleast as hard as problem Q, then a lower bound for Q is also a lower bound for P.

Hence, find problem Q with a known lower bound that can be reduced to problem P in question. Then any algorithm that solves P will also solve Q.

eg1:- least Common multiple $(m,n) = (m*n)/gcd(m,n)$

eg2: P is finding MST for n points in Cartesian plane Q is element Uniqueness problem (known to be $\Theta(n \log n)$).

Branch & Bound

* It is an improvement of backtracking algorithm
* To find optimal solution
* Similar to backtracking in which a state space tree is used to a problem
* Computes a number (bound) at a node to determine whether node is promising

If bound is no better than the value of the best solution found so far, The node is non-promising, otherwise it is promising.

Knapsack problem:-

Problem statement : If we are given with n objects or items & a knapsack (or) a bag in which subset of item is to be placed. Each item has a known weight w_i. The knapsack has a capacity W. Then the profit that is earned is v_i.

The objective is to obtain filling of knapsack with maximum profit earned. But it should not exceed W of the knapsack.

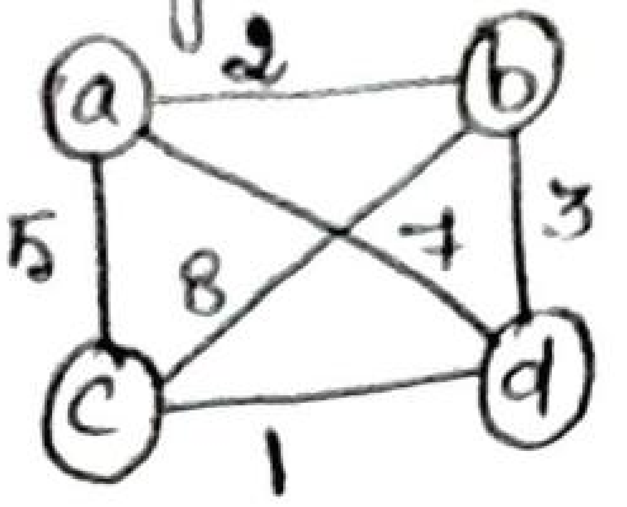
⟹ Order the items of a given instance in descending order by their values to weight ratio's. $v_1/w_1 \geq v_2/w_2 > \dots > v_n/w_n$

we compute upperbound of the tree

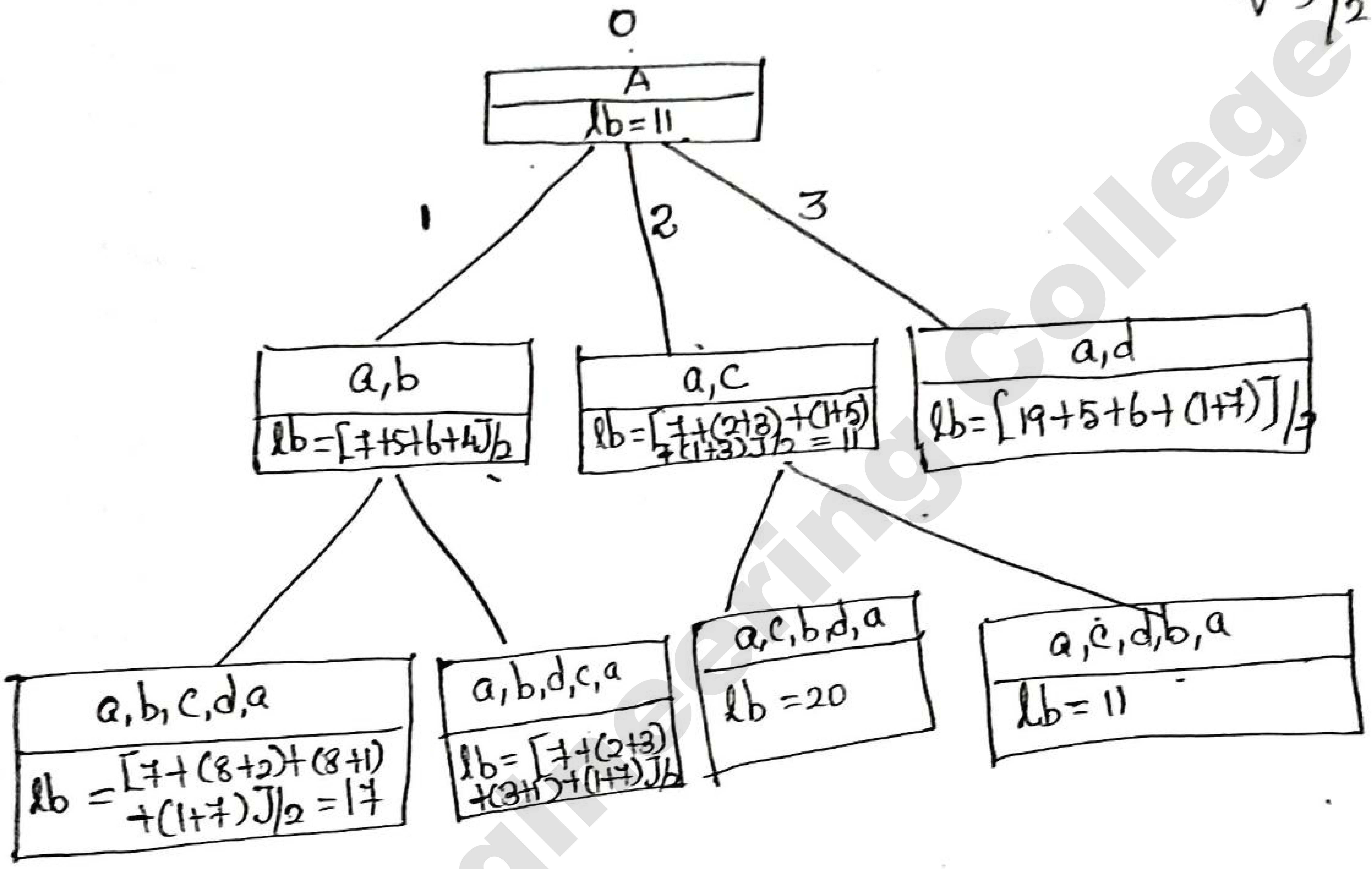
$$Ub = v + (W - w)(v_{i+1}/w_{i+1})$$

APR/MAY 2017 Part - C (UNIT-V)

① Apply Branch and Bound algorithm to solve the TSP for below:-



$LB = \sum_{v \in V}$ (Sum of costs of the two least cost edges adjacent to v) / 2



∴ The solution is 11

a,b → d → c → a

or

a → c → d → b → a.

② Find the optimal solution Using Branch and Bound for the following Assignment problem:- [NOV/DEC 2017].

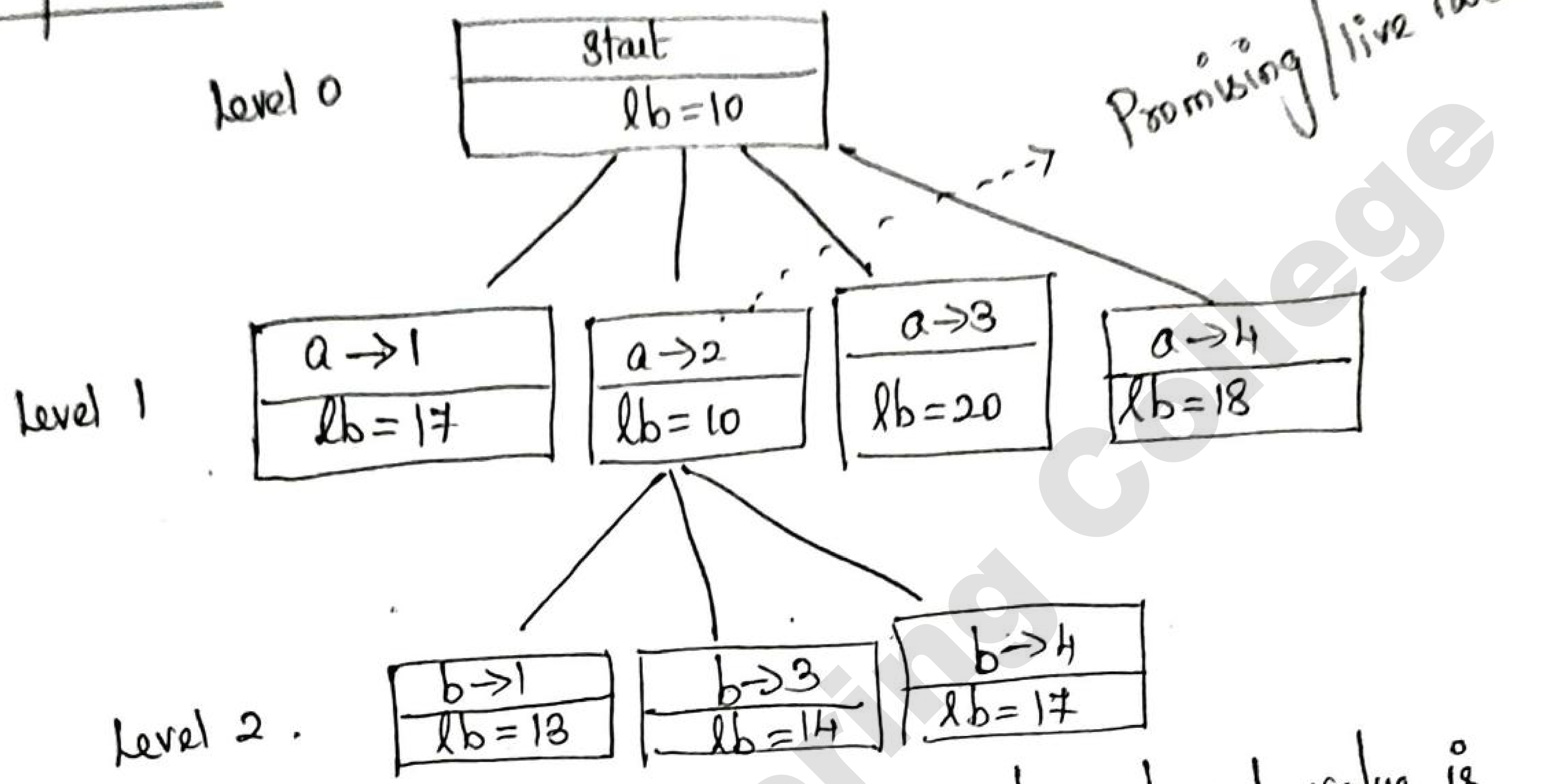
	Job1	Job2	Job3	Job4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Low bound:-

Row wise = 2+3+1+4 = 10

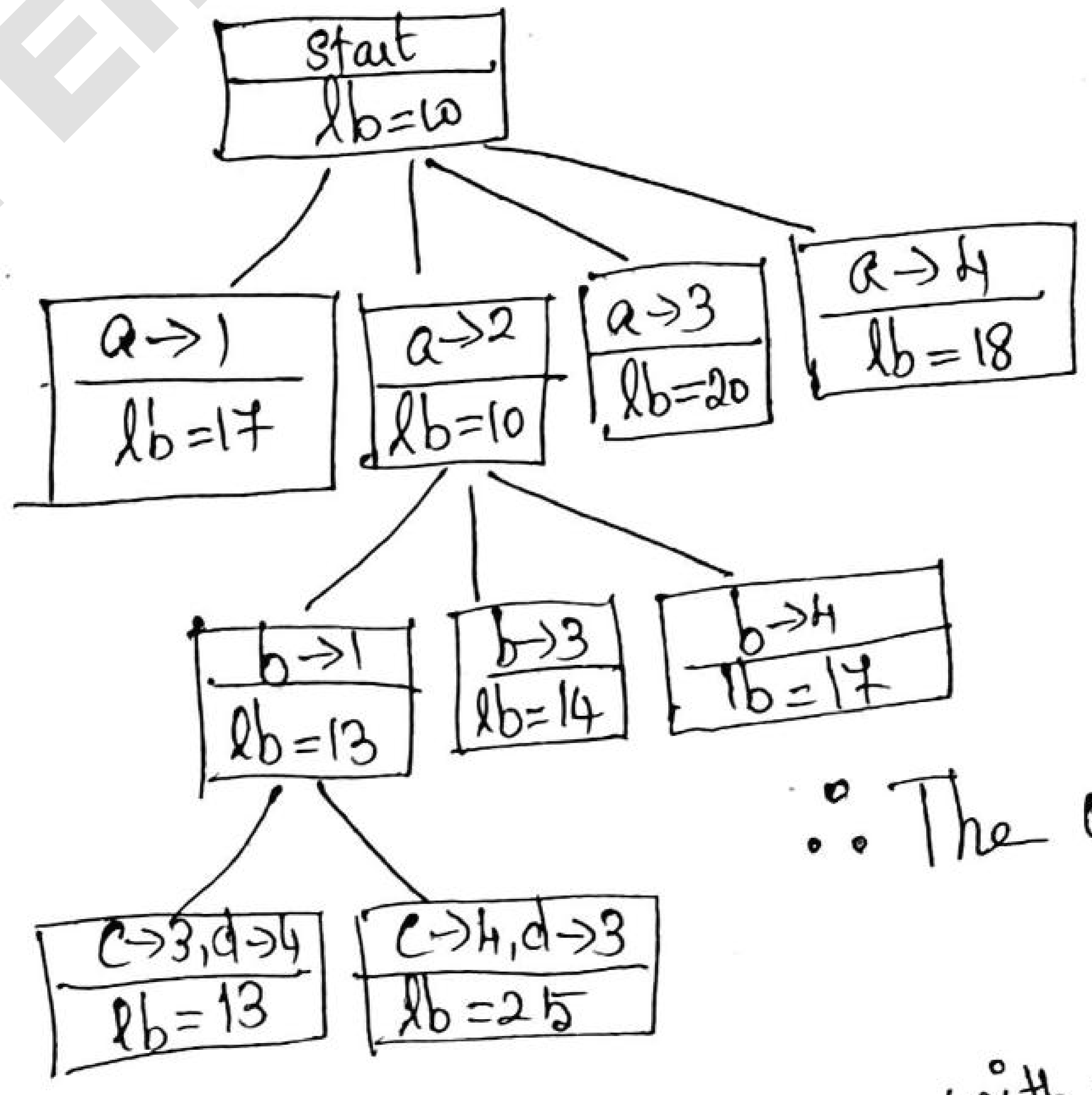
Column wise = 5+2+1+4 = 12

State space tree



From level 1, the node with minimum lower bound value is selected (i.e) node 2 with lb=10 is selected & expanded.

If Level 2. b->1 with lb=13 is expanded we get.

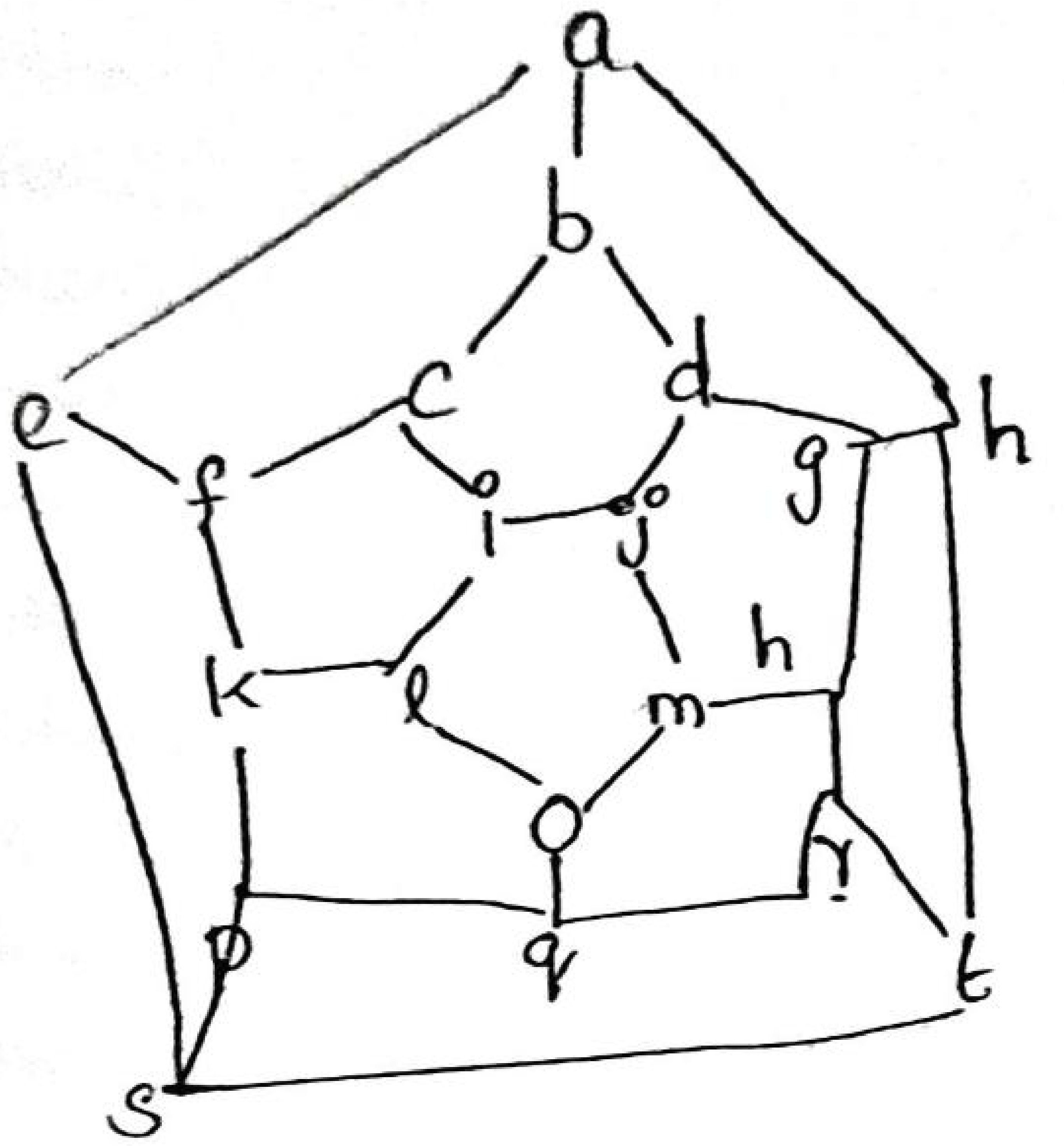


∴ The optimal solution is
a=2, b=6, c=1 &
d=4
with total cost of 13.

UNIT-V

[Nov/Dec '17]

Find a Hamiltonian Circuit or disprove its existence in the graph given below:



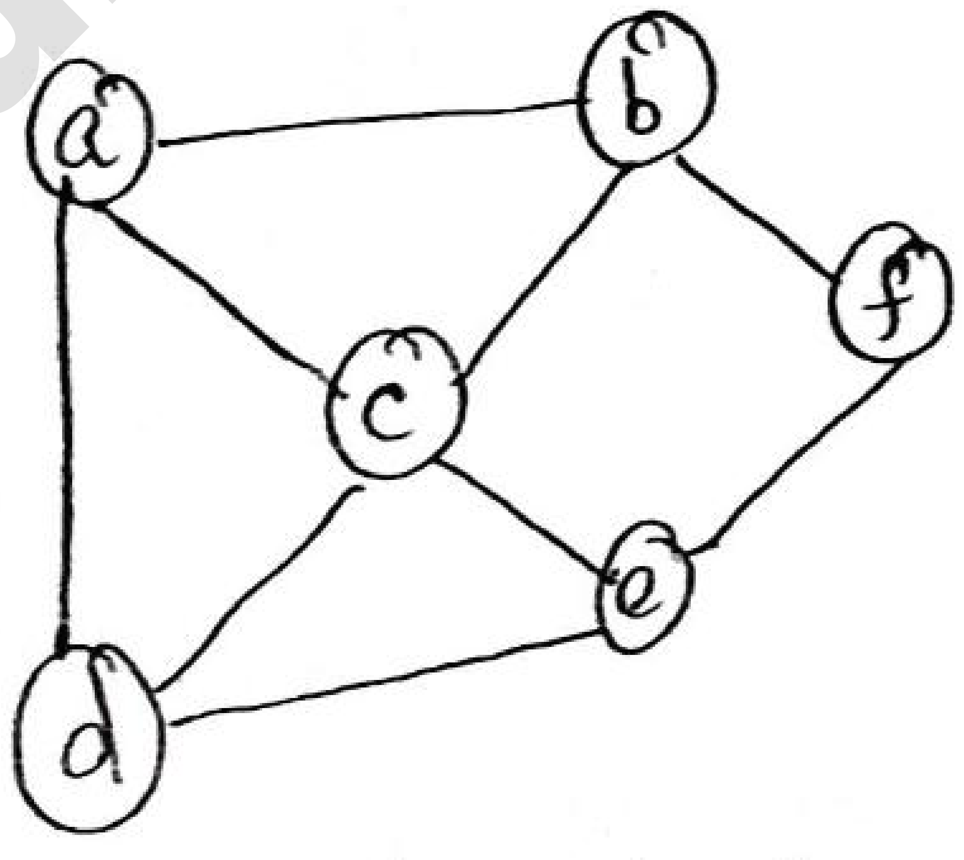
Solution:- There exist a Hamiltonian Circuit for the above given graph as a-b-d-j-m-n-g-h-t-r-q-o-l-i-c-f-k-p-s-e-a

Def of Hamiltonian Circuit

It is a path that starts and ends at the same vertex and passes through all the other vertices exactions without repetition of vertices.

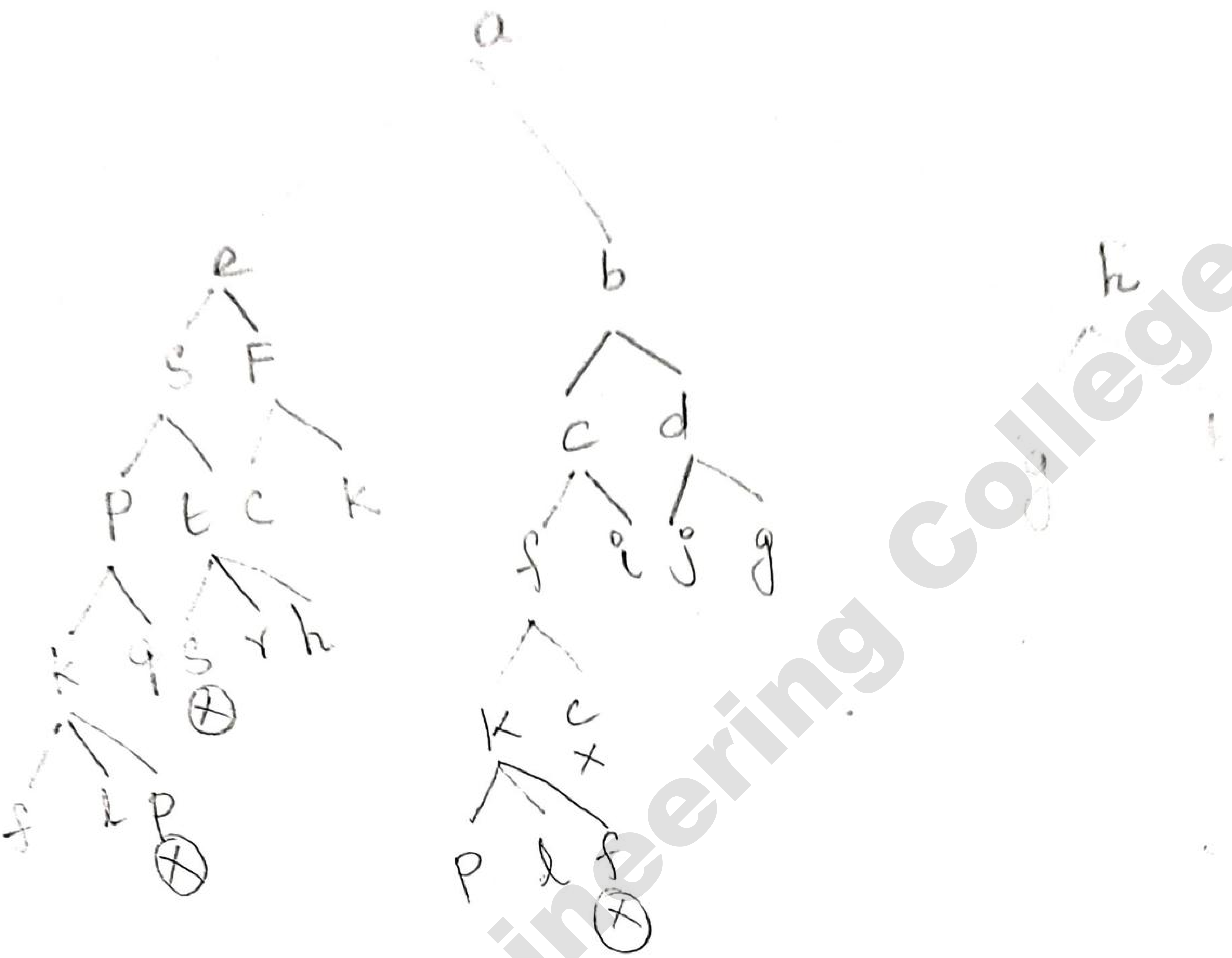
Consider a graph G

eg/..



Problem of solving / finding Hamiltonian Cycle is solved by a backtracking approach.

We make Vertex 'a' the root of the state space tree. From Vertex a we have 3 ways, So we resolve the tie using alphabet



we select vertex b. from b, the algorithm proceeds to c, then to d, then to e and finally f, which provides a deadend. So the algorithm backtracks from f to e, then to d & then to c, which Provides alternative to pursue. going from c to e eventually proves Useless & algorithm has to backtrack from e to c & then to b. from there it goes to the vertices f,e,c & d from which it legitimately return to 'a' yielding the circuit. a,b,f,e,c,d,a

Note: No/. of nodes in state space tree is:

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

Approximation Algorithms for NP-Hard problems [May 2018].

Simplest approximation algorithms for the TSP are based on the greedy technique

Nearest-neighbor algorithm

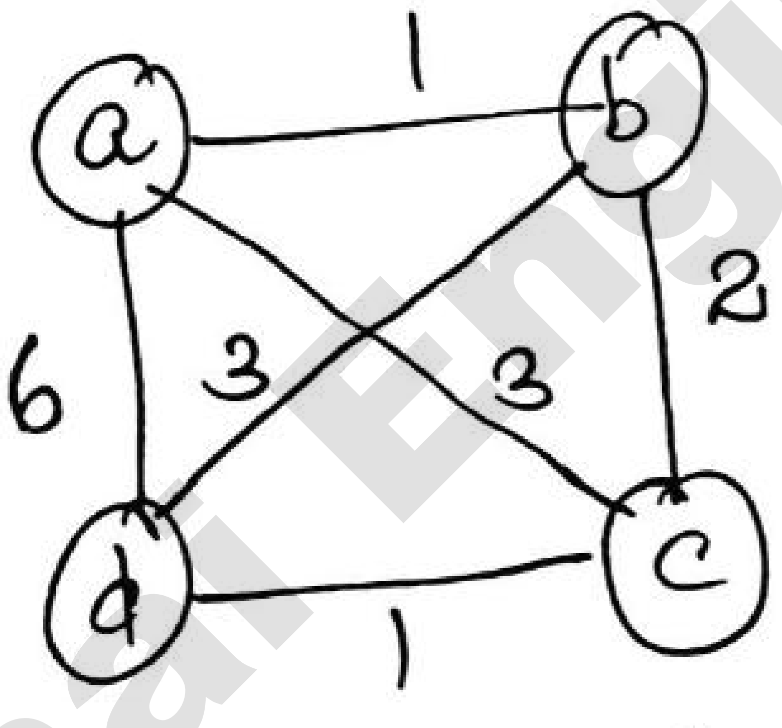
The following well-known greedy algorithm is based on the nearest-neighbor heuristic: always go next to the nearest Unvisited city.

Step 1: Choose an arbitrary city as the start

Step 2: Repeat the following operation until all the cities have been visited: go to the Unvisited city nearest the one visited last (ties can be broken arbitrarily)

Step 3: Return to the starting city.

eg/..



with 'a' as the starting Vertex, the nearest-neighbor algorithm yields the tour (Hamiltonian Circuit)

a-b-c-d-a of length 10.

The optimal Solution, as can be easily checked by exhaustive Search, is the tour a-b-d-c-a of length 8.

The accuracy ratio of this approximation is

$$r(S_a) = \frac{f(S_a)}{f(S^*)} = \frac{10}{8} = 1.25$$

✓ Unfortunately, except for its simplicity, not many good things can be said about the nearest-neighbor algorithm.

✓ In particular, nothing can be said in general about the accuracy of solutions obtained by this algorithm because it can force us to traverse a very long edge on the last leg of the tour.

Multifragment-heuristic algorithm

Another greedy algorithm for the TSP considers it as the problem of finding a minimum-weight collection of edges in a given complete weighted graph so that all the vertices have degree 2.

An application of the greedy technique to this problem leads to the following algorithm:

Step 1: Sort the edges in increasing order of their weights. Initialize the set of tour edges to be constructed to the empty set.

Step 2: Repeat this step n times, where n is the number of cities in the instance being solved: add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise, skip the edge.

Step 3: Return the set of tour edges.

✓ As an example, applying the algorithm to the graph in given figure yields {(a,b), (c,d), (b,c), (a,d)}. This set of edges forms the same tour as the one produced by the nearest-neighbor algorithm.

✓ In general, the multi-fragment-heuristic algorithm tends to produce significantly better tours than the nearest-neighbor algorithm. But the performance ratio of the multifragment-heuristic algorithm is also unbounded.

✓ There is very important subset of instances, called Euclidean, for which we can make a nontrivial assertion about the accuracy of both the nearest-neighbor and multifragment-heuristic algorithms.

✓ These are the instances in which intercity distances satisfy the following natural conditions:

* Triangle inequality $d[i,j] \le d[i,k] + d[k,j]$ for any triple of cities i, j & k (the distance between cities i & j

Cannot exceed the length of a 2-leg path from i to some intermediate city k to j)

* Symmetry $d[i,j] = d[j,i]$ for any pair of cities i & j (distance from i to j is the same as the distance from j to i).

✓ A substantial majority of practical applications of TSP are its Euclidean instances.

✓ They include in particular geometric ones, where cities correspond to points in the plane & distances are computed by the standard Euclidean formula.

✓ Although the performance ratio's of the nearest neighbor and multi fragment heuristic algorithms remain unbounded for Euclidean instances, their accuracy ratio satisfy the following inequality for any such instance with $n \geq 2$ cities:

$$\frac{f(s_a)}{f(s^*)} \leq \frac{1}{2}(\lceil \log_2 n \rceil + 1)$$

where $f(s_a)$ and $f(s^*)$ are the lengths of the heuristic tour & shortest tour respectively.