UNIT I INTRODUCTION 9

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.

Notion of an algorithm

An algorithm is a sequence of unambigous instructions for solving a problem. i.e for obtaining a required output for any legitimate input in a finite amount of time.

It is a step by step procedure with the input to solve the problem in a finite amount of time to obtain the required output.

The notion of algorithm illustrates some important points:

* The non-ambiguity requirement for each step of an algorithm cannot be compromised.
* The range of inputs for which an algorithm works has to be specified carefully.
* The same algorithm can be represented in several different ways.
* There may exist several algorithms for solving the same problem.
* Algorithms for the same problem can be based on very different ideas and can solve the problem with different speeds.

characteristics of an algorithm:

Input: Zero/more quantities are externally supplied.

Output: Atleast one quantity is produced

Definiteness: Each instruction is clear and unambigous [contains only one meaning]

Finiteness: If the instructions of an algorithm is traced, then for all cases, the algorithm must terminate after a finite number of steps.

Efficiency: Every instruction must be very basic and runs in short time.

Example:

The Greatest Common Divisor (GCD) of two non-negative integers m and n (not-both-zero), denoted as gcd(m,n) is defined as the largest integer that divides both m and n evenly. i.e with a remainder of zero.

Euclid's algorithm is based on applying repeatedly the equality gcd(m, n) = gcd(n, m mod n), where m mod n is the remainder of the division of m by n, until m mod n is equal to 0. Since gcd(m, 0) = m, the last value of m is also the greatest common divisor of the initial m and n.

gcd(60, 24) can be computed as follows:

gcd(60, 24) = gcd(24, 12) = gcd(12, 0) = 12

Euclid's algorithm for computing gcd(m, n) in simple steps

Step 1: If n = 0, return the value of m as the answer and stop, otherwise proceed to step 2.

Step 2: Divide m by n and assign the value of the remainder to r.

Step 3: Assign the value of n to m and the value of r to n. Goto step 1

Euclid's algorithm for computing gcd(m, n) expressed in pseudocode

ALGORITHM Euclid_gcd(m, n)

// Computes gcd(m, n) by Euclid's algorithm

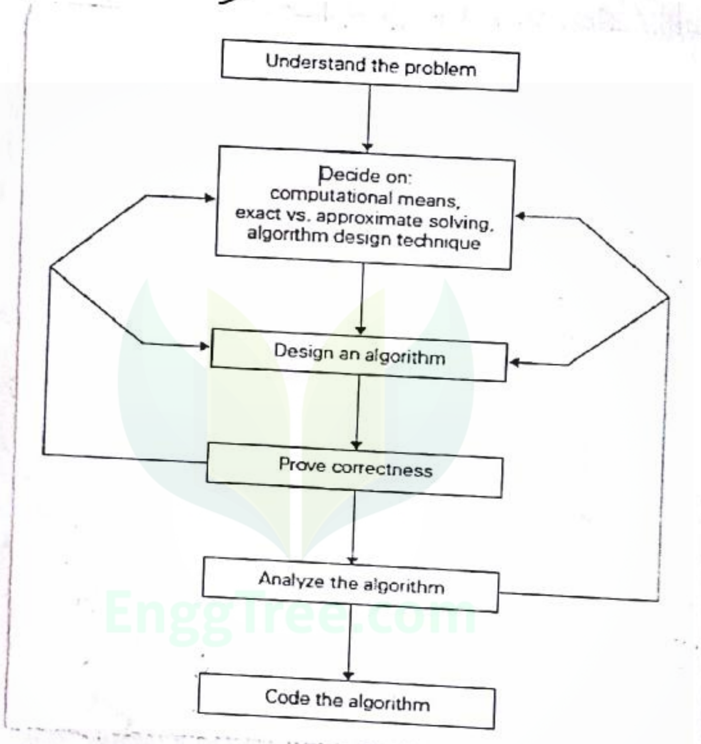
// Input: Two non-negative numbers, not-both-zero integers m and n

// Output: Greatest common divisor of m and n

```
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

Fundamentals of algorithmic problem solving

A sequence of steps involved in designing and analyzing an algorithm is shown in the figure:



(i) Understanding the problem

* This is the first step in designing of algorithm.
* Read the problem's description carefully to understand the problem statement completely.
* Ask doubt clary clarifying the doubts about the problem.
* Identify the problem types and use existing algorithm to find solution.
* Input (instance) to the problem and range of input get fixed.

(ii) Decision making

The decision making is done on the following:

(a) Ascertaining the capabilities of the computational device

* In random-access machine (RAM), instructions are executed one after another. (The central assumption is that, one operation at a time). Accordingly, algorithms designed to be executed on such machines are called sequential algorithms.

* In some new computers, operations are executed concurrently, ie in parallel. Algorithms which makes use of this capability are called parallel algorithms.

* choice of computational devices like processor and memory is mainly based on space and time efficiency.

(b) choosing between exact and approximate problem solving

* The next principal decision is to choose between solving the problem exactly or solving it approximately.

* An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.

* If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm.

i.e, it produces approximate answer.
Eg. extracting square roots, solving non-linear equations and evaluating definite integrals.

(c) Algorithm Design Techniques

* An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

* Algorithms + Data structures = programs

* Though algorithms and data structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing the algorithm.

* Algorithmic strategy / technique / paradigm are a general approach by which many problems can be solved algorithmically.
Eg. Brute Force, Divide and conquer, Dynamic programming, Greedy technique and so on.

(iii) Methods of specifying an algorithm

There are three ways to specify an algorithm. They are :

a) Natural language
b) Pseudocode
c) Flowchart.

a) Natural language

It is a very simple and easy to specify an algorithm using natural language.

Eg: Algorithm to perform addition of two numbers.

Step 1: Read the first number, say a.
Step 2: Read the second number, say b.
Step 3: Add the above two numbers and store the result in c.
Step 4: Display the result from c.

b) Pseudocode

* It is a mixture of a natural language and programming language constructs.
* It is more precise than natural language.

Eg: ALGORITHM Sum(a, b)

// Problem Description: This algorithm performs addition of two numbers

// Input: Two integers a and b

// Output: Addition of two integers

c ← a+b

return c

c) Flowchart

Flowchart is a graphical representation of an algorithm. It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

(iv) Proving an algorithm's correctness

* once an algorithm has been specified, then its correctness must be proved.

* An algorithm must yeild a required result for every legitimate input in a finite amount of time.

* A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

* The notion of correctness for approximation algorithm is less straightforward than it is for exact algorithm. The error produced by the algorithm should not exceed a predefined limit.

(v) Analysing an algorithm

* For an algorithm the most important is efficiency. There are two kinds of algorithm efficiencies available. They are:

* Time efficiency – indicating how fast the algorithm runs, and

* Space efficiency – indicating how much extra memory it uses.

* The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.

(vi) Coding an algorithm

* The coding/implementation of an algorithm is done by a suitable programming language like C, C++, Java.

* The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The power of algorithm should not be reduced by inefficient implementation.

* Standard tricks like computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.

* Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude. But once an algorithm is selected, a 10-50% speed up may be worth an effort.

* It is very essential to write an optimized code (efficient code) to reduce the burden of compiler.

Important Problem types

The most important problem types are:

i) sorting
ii) searching
iii) string processing
iv) graph problems
v) Combinatorial problems
vi) Geometric problems
vii) numerical problems

Sorting

1) The sorting problem is to rearrange the items of a given list in nondecreasing (ascending) order.
2) Sorting can be done on numbers, characters, strings or records.
3) To sort student records in alphabetical order by names or by student number or by student grade-point average can be done using special information called key.
4) A sorting algorithm is stable if it preserves the relative order of any two equal elements in its input.

Searching

1. The searching problem deals with finding a given value called a search key, in a given set.
Eg. ordinary linear search and fast binary search.

String Processing

1. A string is a sequence of characters from an alphabet.
2. Searching for a given word in a text is called

string matching.

Graph problems

1. A graph is a collection of points called vertices, some of which are connected by line segments called edges.
2. Some of the graph problems are graph problems are graph traversal, shortest path algorithm, topological sort, travelling salesman problem, graph-coloring problem and so on.

Combinatorial problems

1. These are problems that ask explicitly or implicitly to find a combinatorial object such as permutation, combination or a subset that satisfies certain constraints.
2. In practical, the combinatorial problems are the most difficult problems in computing.

Geometric problems

1. Geometric algorithms deal with geometric objects such as points, lines and polygons
2. Geometric algorithms are used in computer graphics, robotics and tomography.
3. The closest-pair problem and the convex-hull problem comes under this category.

Numerical Problems

1. Numerical problems are problems that involve mathematical equations, systems of equations, computing definite integrals, evaluating functions, and so on.
2. The majority of such mathematical problems can be solved only approximately.

Fundamentals of the analysis of algorithm efficiency

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can be analyzed by the following ways.

a. Analysis framework
b. Asymptotic notation and its properties
c. Mathematical analysis for recursive algorithms.
d. Mathematical analysis for non-recursive algorithms.

Analysis framework

There are two kinds of efficiencies to analyze the efficiency of any algorithm. They are:

* Time efficiency, indicating how fast the algorithm runs, and
* Space efficiency, indicating how much extra memory it uses

The algorithm analysis framework consists of:

* Measuring an input's size
* Units for measuring run time.

* orders of growth
* Worst-case, best-case and average-case efficiencies

Measuring an input's size

* An algorithm's efficiency is defined as a function of some parameter 'n' indicating the algorithm's input size. In most cases, selecting such a parameter is straightforward.
* For the problem of evaluating a polynomial $p(x) = a_n x^n + \dots + a_0$ of degree n, the size of the parameter will be polynomial's degree (or) the number of its coefficients
* In computing the product of two $n \times n$ matrices, the choice of a parameter indicating an input size does matter.
* Consider a spell-checking algorithm, if the algorithm examines individual characters of its input, then the size is measured by number of characters.
* The input size by the number b of bits in the n's binary representation is $b = \lfloor \log_2 n \rfloor + 1$.

Units for measuring running time

Some standard unit of time measurement such as a second or millisecond can be used to measure the running time of a program after implementing the algorithm.

Drawbacks

- Dependence on the speed of a particular computer.
- Dependence on the quality of a program implementing the algorithm.
- The compiler used in generating the machine code
- The difficulty of clocking the actual running time of the program.

So, we need to measure an algorithm's efficiency that does not depend on these factors.

One possible approach is to count the number of times each of the algorithm's operation is executed.

The most important operation (+, -, *, /) of the algorithm are called basic operations. Computing the number of times the basic operation is executed is easy. The total running time of the program is determined by basic operations count.

Orders of growth

* A difference between run times of small inputs will not reveal efficient algorithms from inefficient ones.

* For example, finding GCD of two small numbers using Euclid's algorithm is not immediately clear about its efficiency. But the efficiency of it can be determined when comes to larger numbers.

* For large values of n, it is the function's order of growth that counts the most. The order of growth for different 'n' values using few functions are given in the table.

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Worst-case, Best-case and Average-case efficiencies

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] and a search key K.

//Output: The index of the first 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

The running time of this algorithm can be different for the same list size n.

Worst-case efficiency

* In the worst case, there is no matching element (or) the first matching element can be found as last in the list. So 'n' comparisons to be made.
* For the input of size n, the running time is

$C_{worst}(n) = n$.

Best-case efficiency

* In the best case, the matching of element can be found at first comparison on the list.
* If we search the first element in list of size n, (i.e. first element = search key), then the running time is $C_{best}(n) = 1$

Average-case efficiency

* The average case efficiency lies between best and worst case.
* To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n.
* The standard assumptions are that
  - The probability of successful search is equal to p $(0 \le p \le 1)$ and
  - The probability of the first match occurring in the i^{th} position of the list is the same for every i.

$$C_{avg}(n) = \left[1 \cdot p/n + 2 \cdot p/n + \cdots i \cdot p/n + \cdots + n \cdot p/n\right] + n \cdot (1-p)$$

$$= p/n\left[1+2+\cdots+i+\cdots+n\right] + n(1-p)$$

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

Asymptotic notations and its properties

i/p	$A_1 = n$ steps	$A_2 = 5n$ steps	$A_3 = n^2$ steps
$n=1$	1 nsec	5 nsec	1 nsec
$n=100$	100 nsec	500 nsec	10^4 nsec
$n=10^9$	1 sec	5 sec	10^{18} nsec $= 10^9$ sec $\simeq$ 31 years
$n=10^{20}$	3170 years	15000 years	3170×10^{11} years

1 step takes $\Rightarrow$ 1 unit of computer time $\Rightarrow 1\times10^{-9}$ sec $\Rightarrow$ 1 ns (ie $1/10^9$ sec)

More or Less same values $\left\{ \begin{array}{l} A_1 \\ A_2 \end{array} \right.$ | $\left. A_3 \right\}$ larger value than A_1 & A_2

A_1 & A_2 belong to same class of time complexities, whereas A_3 differs as other class of time complexity.

So considering the above scenario, we need different notations to say A_1 and A_2 are approximately equal and A_1 & A_2 are lesser than A_3. The notation which is needed to differentiate the time complexities of different algorithms are called asymptotic notations.

The basic asymptotic notations are

1. Bigoh O notation
2. Big omega Ω notation
3. Big theta Θ notation

Big oh notation (O)

Let f(n) and g(n) be two functions,

Definition: A function f(n) is said to be in O(g(n)), denoted as $f(n) \in O(g(n))$, if and only if $f(n) \leq c \cdot g(n)$ for some $c > 0$ after $n \geq n_0 \geq 0$.

Proof: Assume $f(n) = n$ & $g(n) = 5n$

$$f(n) = O\,g(n)$$
$$f(n) \leq c \cdot g(n)$$
$$n \leq c \cdot (5n)$$

If $C = 1$, then

$5n \geq n$ [for any n value ≥ 0]

For $C = 1$ and $n \geq 0$, $f(n) \leq c \cdot g(n)$ (ie) $n \leq 5n$

Hence proved.

Eg.2: $f(n) = 2n+10$ $g(n) = n$

$$f(n) = O(g(n))$$
$$f(n) \leq c \cdot g(n) \text{ for } c > 0,\ n \geq n_0 \geq 0$$

$2n+10 \leq c \cdot (n)$ [To prove, choose $c = 3$

$$2n+10 \leq 3n$$

~~3n − 2n ≤ 10~~ $10 \leq 3n - 2n$

$10 \leq n$ (ie) $n \geq 10$

So for $c = 3$ and $n \geq \underset{n_0}{10} \geq 0$, $f(n) \leq c \cdot g(n)$

Hence proved.

Big Omega (Ω)

Definition: A function $f(n)$ is said to be in $\Omega(g(n))$ denoted as $f(n) = \Omega(g(n))$, if and only if $f(n) \geq c \cdot g(n)$ for some $c > 0$ after $n \geq n_0 \geq 0$.

Proof: $f(n) = 5n+10$ $\quad g(n) = 2n$

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

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

$$5n+10 \geq c \cdot 2n$$

If $c = 1$ ~~then~~

$5n+10 \geq 2n$ [for any n value ≥ 0]

$\cancel{10 \geq 3n}$

$\cancel{n = 3}$

Hence for $c=1$ and $n \geq 0$, $f(n) \geq c \cdot g(n)$

Hence proved.

Eg 2: $f(n) = n^2$ $\quad g(n) = n^2+n$

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

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

$$n^2 \geq c \cdot (n^2+n)$$

Choose the value of c, such that $n^2 \geq c \cdot (n^2+n)$

Hence $c = 1/2$

$$n^2 \geq 1/2 (n^2+n)$$

$$\cancel{1/2 n^2} + 1/2 n^2 \geq \cancel{1/2 n^2} + n/2$$

$$\cancel{1/2} n^2 \geq \cancel{1/2} n$$

$$n^2 \geq n$$

Hence for $c = 1/2$ and $n \geq 0$ $f(n) \geq c \cdot g(n)$

Hence proved.

Θ notation

Definition: A function $f(n)$ is said to be in $\Theta(g(n))$, denoted as $f(n) = \Theta(g(n))$, if and only if

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$, for some $c_1 > 0$ and $c_2 > 0$ after $n \geq n_0 \geq 0$.

Condition 1: $f(n) \geq c_1 \cdot g(n) \Rightarrow f(n) = \Omega(g(n))$

Condition 2: $f(n) \leq c_2 \cdot g(n) \Rightarrow f(n) = O(g(n))$

Proof: $f(n) = n^2 + n \quad g(n) = 5n^2$

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

Condition 2:

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

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

$n^2 + n \leq c \cdot 5n^2$

If $c = 1$

$n^2 + n \leq 5n^2$

$n \leq 4n^2$ (or)

$4n^2 \geq n$

for $c = 1$ & $n \geq \boxed{0}$ [n_0]

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

Hence proved

Condition 2:

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

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

$n^2 + n \geq c \cdot (5n^2)$

Carefully choose $c > 0$ [$c = 1/5$]

$n^2 + n \geq \frac{1}{\cancel{5}} \cdot \cancel{5}n^2$

$\cancel{n^2} + n \geq \cancel{n^2}$

$n \geq 1$

Hence for $c = 1/5$ & $n \geq 0$

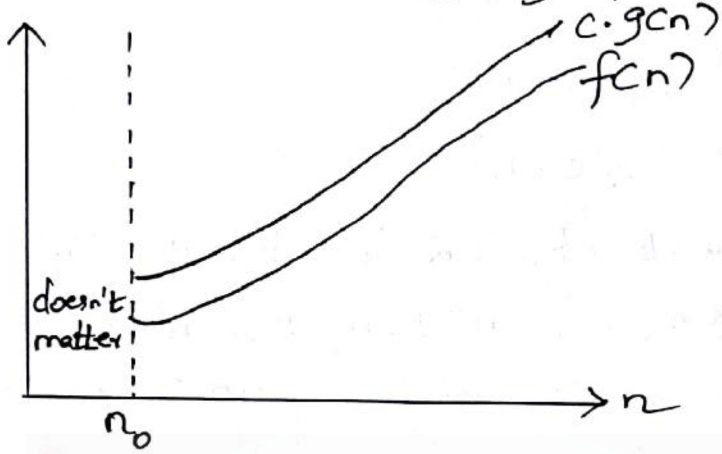
$f(n) \geq c \cdot g(n)$

Hence proved

Diagramatic notation of O, Ω and Θ

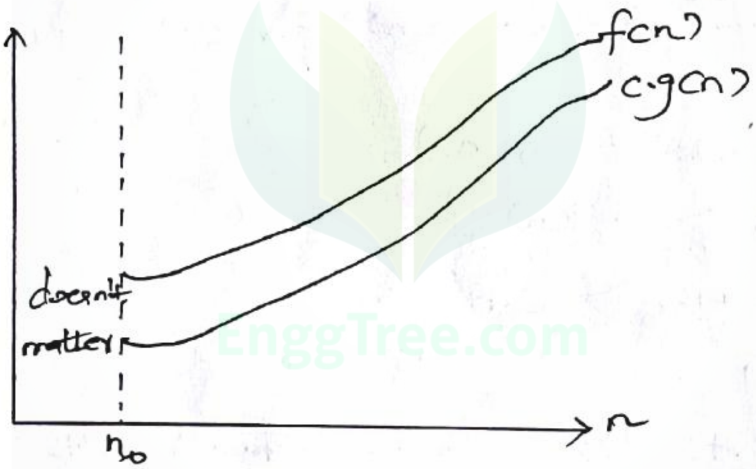
Big oh notation: $f(n) = O(g(n))$

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



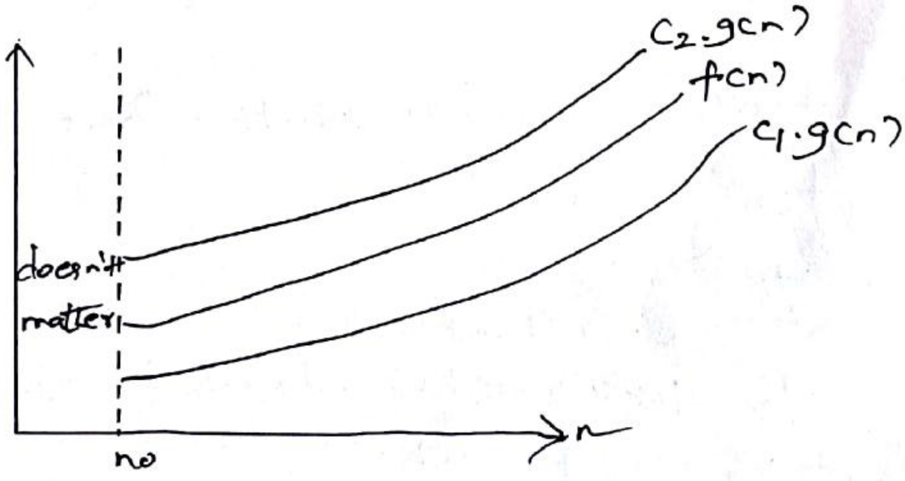
Big omega notation: $f(n) = \Omega(g(n))$

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



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

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



Example

$f(n) = n^2 + 5n + 100 \quad g(n) = n^2$

To prove : $f(n) = O(g(n))$ and also $f(n) = \Omega(g(n))$

Proof 1 : $f(n) = O(g(n))$

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

$n^2 + 5n + 100 \leq c \cdot n^2$

n^2 is utmost n^2, $5n$ is utmost $5n^2$ and 100 is utmost $100n^2$, so $n^2 + 5n^2 + 100n^2 \Rightarrow 105n^2$

So choose the value of $c > 105$ (i.e) $c = 106$

So, $n^2 + 5n + 100 \leq 106n^2$, R.H.S will be definitely greater than LHS when you prefer $c = 106$ & greater values.

Hence $f(n) \leq c \cdot (g(n))$ for $c=1$ & $n \geq 0$

Proof 2 : $f(n) = \Omega(g(n))$

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

$n^2 + 5n + 100 \geq c \cdot n^2$

when $c = 1$

$n^2 + 5n + 100 \geq n^2$ which shows LHS is directly greater than RHS for $c > 0$ and $n \geq 0$.

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

Example

$f(n) = n^2 \quad g(n) = n^2 + 5n + 100$, Prove $f(n) = \Omega(g(n))$

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

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

$n^2 \geq c \cdot (n^2 + 5n + 100)$

Take $c = 1/2$, RHS will be reduced by $1/2$, i.e

$n^2 \geq \frac{1}{2} n^2 + \frac{5}{2} n + 50$

$n^2 \geq \frac{1}{2} n^2 + 2.5n + 50$

$\frac{1}{2}n^2 \geq 2.5n + 50$

$\times 2 \Rightarrow n^2 \geq 5n + 100$

For $n = 0$ $n^2 \geq 5n + 100$ will be wrong

$n = 1$ ×

$n = 10$ ×

But for $n = 20$

$(20)^2 \geq 5(20) + 100$

$400 \geq 100 + 100 \Rightarrow 400 \geq 200$

Hence $f(n) \geq c \cdot g(n)$, if and only if $c > 0$ ie $(c = 1/2)$ and $n \geq \underset{(n_0)}{20} \geq 0$.

Mathematical analysis of Non-recursive algorithm

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on input's size
2. Identify the algorithm's basic operation
3. Check whether the no. of times the basic operation is executed depends on the size of an input.
4. Set up a sum expressing the no. of times the algorithm's basic operation is executed.
5. Manipulate the sum using standard rules and formulas.

Two basic rules of sum manipulation

R_1 $\sum_{i=l}^{u} c a_i = c \sum_{i=l}^{u} a_i$

$R_2 \quad \sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$

Important summations

$S_1 \quad \sum_{i=l}^{u} 1 = u - l + 1$

1. $\sum_{i=1}^{n} 1 = \underset{\downarrow u}{n} - \underset{l}{1} + 1 \Rightarrow n$

2. $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2} n^2$

3. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3} n^3$

4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1} n^{k+1}$

5. $\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a-1}$ where $(a \neq 1)$

$\sum_{i=0}^{k} 2^i = 1 + 2 + 2^2 + \cdots 2^k = \frac{2^{k+1} - 1}{2-1} = \boxed{2^{k+1} - 1}$

6. $\sum_{i=1}^{n} i 2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n 2^n = (n-1) 2^{n+1} + 2$

Example 1

To find the largest element in a list of 'n' numbers.

Algorithm:

//Input : An array A[0...n-1]

//output : Returns the value of maxval

Maxval ← A[0]

for i=1 to n-1 do

if A[i] > Maxval

Maxval ← A[i]

return Maxval

Analysis:

1) I/p size

the no. of elements used in this array is n. ie, i/p size is 'n'.

2) Basic operation

There are two operations in the loop's body. The comparison A[i] > Maxval and the assignment Maxval ← A[i]. Since the comparison is executed on each repitition of the loop, we should consider the comparison to be the algorithms basic operation.

3) whatever the input is, the comparison is made for n-1 times. so, it depends on our i/p size 'n'.

4) C(n) denotes, the no. of times this comparison is executed and try to find a formula expressing it as a function of size 'n'.

The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable 'i', within the bounds between 1 and n-1.

$$\therefore C(n) = \sum_{i=1}^{n-1} 1 \quad [u-l+1]$$

$$= \sum_{i=1}^{n-1} n-\cancel{1}-1+\cancel{1} \Rightarrow \sum_{i=1}^{n-1} n-1 \in O(n)$$

Example 2

To check whether all the elements in a given array are distinct. [Element uniqueness problem]

Algorithm:

// Input : An array [0 ... n-1]
// Output : If the elements are distinct, returns "true" otherwise returns "false"

```
for i <- 0 to n-2 do
    for j <- i+1 to n-1 do
        If A[i] = A[j] return false
return true
```

Analysis:

1. I/p size → n [i/p size is equal to no. of elements in array]
2. Basic operation → Comparison, since the innermost loop contains a single operation, we should consider it as the algorithm's basic operation.
3. In this algorithm, the no. of element comparisons will depend not only on 'n', but also on whether there are any equal elements in the array and if there are, which array position they occupy. ie, the efficiency depends on i/p size and ordering of data. So, we'll discuss the problem to the worst case
4. Since one comparison is made for each repetition of the innermost loop and this is also repeated for each value of the outer loop.

$$\therefore C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} \left[(n-1) - (i+1) + 1\right] = \sum_{i=0}^{n-2} \left[n - 1 - i - \cancel{1} + \cancel{1}\right]$$

Based on R_2, $\displaystyle\sum_{i=0}^{n-2} n-1 - \sum_{i=0}^{n-2} i$

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

[Substituting the lower & higher limits, we get $\underset{i}{0} + 1 + 2 + \cdots + \underset{u}{(n-2)}$

we know, $1 + 2 + \cdots + n = \dfrac{n(n+1)}{2}$

Substituting n by $n-2$ in above eqn

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

$$\therefore (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)(n-2-0+1) - \left[\frac{(n-2)(n-1)}{2}\right]$$

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

$$= (n-1)\left[(n-1) - \frac{(n-2)}{2}\right]$$

$$= (n-1)\left[\frac{2n - \cancel{2} - n + \cancel{2}}{2}\right] \Rightarrow \frac{(n-1)(n)}{2}$$

$$\frac{n(n-1)}{2} \approx \frac{1}{2}n^2 \in O(n^2)$$

In general, this algorithm needs to compare all $\frac{n(n-1)}{2}$ distinct pairs of its 'n' elements in the worst case.

Example 3: Matrix multiplication

Algorithm :

```
//Input : 2 n by n matrices A & B
//output : Matrix C = AB
for i <- 0 to n-1 do
    for j <- 0 to n-1 do
        C[i,j] = 0
    for k <- 0 to n-1 do
    c[i,j] = c[i,j] + A[i,k] * B[k,j]
return C
```

Analysis

1. Input size → n by matrix order
2. Basic operation → Addition & multiplication

$$T(n) = A(n) + M(n)$$

Here $\underset{\text{No. of addition}}{A(n)} = \underset{\text{No. of multiplication}}{M(n)}$

No. of multiplication

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

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

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$$

$$= \sum_{i=0}^{n-1} n \sum_{j=0}^{n-1} 1 \Rightarrow \sum_{i=0}^{n-1} n(n-1-0+1)$$

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

$$n^2 \sum_{i=0}^{n-1} 1 \Rightarrow n^2(n-1-0+1) \Rightarrow n^3$$

$$M(n) = n^3$$

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

$$T(n) = 2n^3 = O(n^3).$$

Example 4 :

Find the number of binary digits in the binary representation of a +ve decimal integer.

Algorithm:

```
//Input : integer n
// output : No. of binary digits in n's binary
                                  representation
Binary(n)
Count ← 1
while n>1 do
    Count ← Count + 1
    n ← ⌊n/2⌋
return Count
```

Analysis :

Basic operation : Comparison

The number of times the comparison n>1 will be executed is larger than the no. of repetitions of the loop, by exactly one.

The loop get stopped when 'n' becomes 1.

Eg $n=8$, Count =

(1) $8>1$, T, count = 2, n = 4
(2) $4>1$, T, count = 3, n = 2
(3) $2>1$, T, Count = 4, n = 1
(4) $0>1$, F return 4

So 4 comparisons and 3 repetitions of operation.

In general:

$$\frac{8}{2} = \frac{4}{2} = \frac{2}{2} = 1$$

$$\therefore \frac{8}{2^3} = 1$$

$$\frac{n}{2^k} = 1$$

$$n = 2^K$$

Take log on both sides

$$\log n = \log_2 2^K$$

$$\log n = K \log_2 2 \qquad [\because \log_n n = 1]$$

$$\boxed{\log n = K}$$

TABLE 2.2 Basic asymptotic efficiency classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Mathematical analysis of recursive algorithm:

General plan for analysing the efficiency of recursive algorithm:

1) Decide an i/p size
2) Identify the basic operation
3) Check whether $C(n) \in n$ (i/p size) or not.
4) Set up a recurrence relation. with an initial Condition, for the no. of times the basic operation is executed.
5) Solve the recurrence.

Example:

To find the factorial function $F(n) = n!$

Algorithm:

//Input: A non-negative integer n
//output: Value of n!

if n=0 return 1
else return $F(n-1) * n$

$n! = 1 \cdots (n-1) \cdot n$

Analysis:

Basic operation $\rightarrow$ multiplication, $M(n)$

No. of multiplications = $\underbrace{M(n-1)}_{\text{for computing } F(n-1)} \underbrace{+1}_{\text{for multiplying } F(n-1) \text{ by } n}$, $n>0$

$\therefore \boxed{M(n) = M(n-1)+1, \ n>0} \rightarrow$ Recurrence relation

Initial Condition:

From the algorithm, when n=0, the algorithm performs no multiplication.

Hence $M(n) = M(n-1)+1$
$M(0) = 0$

$M(n) = M(n-1)+1$ —— (1)

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

$M(n-2) = M(n-3)+1$ —— (3)

sub (3) in (2)

$M(n-1) = M(n-3)+1+1$
$= M(n-3)+2$ —— (4)

Sub (4) in (1)

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

$$\boxed{M(n) = M(n-3)+3}$$

In general, $\boxed{M(n) = M(n-i)+i}$

Sub $i=n$ in above eqn.

$$M(n) = M(n-n)+n$$
$$= M(0)+n$$

$\because M(0)=0$

$$M(n)=n$$

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

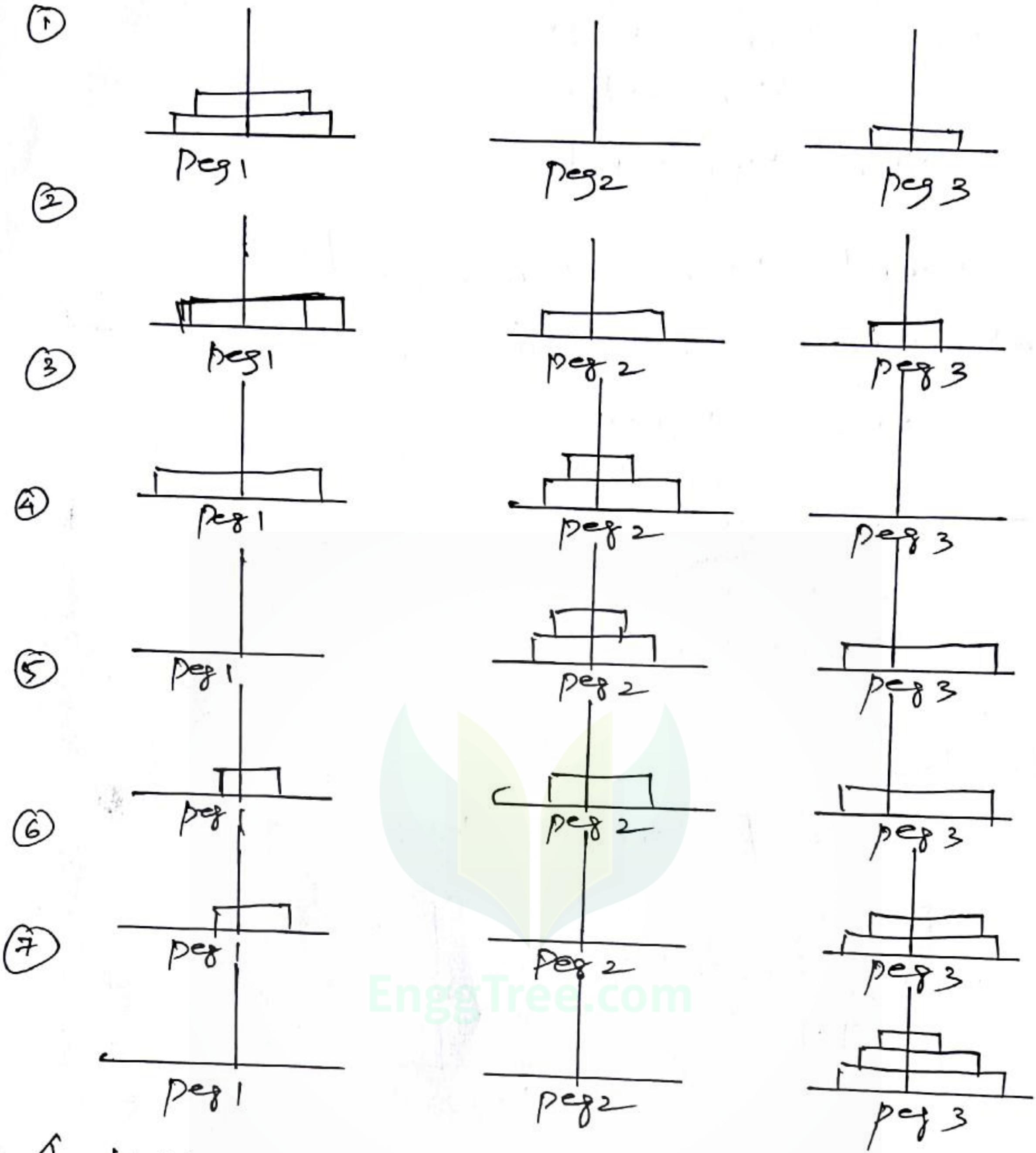
Example 2 :

Tower of Hanoi puzzle:

Condition :

1. Move n disks from peg1 to peg3 using peg2 as intermediate.
2. Only one disk can be moved at a time.
3. Smaller disks can be placed on larger one and not vice versa.

Assume $n = 3$



Analysis:

No. of moves $= 2^n - 1$

$$M(n) = 2^3 - 1$$
$$= 7$$

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

$$M(n) = 2M(n-1) + 1, \text{ for } n > 1$$

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

$$M(1) = 1.$$

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

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

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

$M(n-2) = 2M(n-3)+1$ —— (3)

Sub (3) in (2)

$M(n-1) = 2[2M(n-3)+1]+1$

$M(n-1) = 2^2 M(n-3)+2+1$ —— (4)

Sub (4) in (1)

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

$= 2^2[2M(n-3)+1]+2+1$

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

In general,

$M(n) = 2^i . M(n-i) + 2^{i-1} + 2^{i-2} + 2^{i-3} + \cdots 2^1 + 2^0$

$$\boxed{\sum_{i=0}^{n-1} 2^i = 2^0+2^1+2^2+\cdots+2^{n-2}+2^{n-1} \Rightarrow 2^n-1}$$

$\therefore M(n) = 2^i[M(n-i)] + 2^i - 1$ —— (5)

Initial Condition,

$M(1) = 1$

To get 1, $n-i=1$

$\therefore i = n-1$

Substitute $i=n-1$ in eqn (5)

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

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

$= 2[2^{n-1}] - 1$

$= 2^{n-1+1} - 1 \Rightarrow 2^n - 1$

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

Example 3

Recursive algorithm to find no. of binary digits in a binary representation of a decimal number

Algorithm:

// input : n

// output : no. of bits in n's

if n=1 return 1

else return BinRec($\lfloor n/2 \rfloor$)+1 → floor function

Analysis

1) i/p size → n

2) Basic operation → addition

3) Setting up a recurrence equation based on the no. of additions made in the algorithm.

(ie) $A(n) = A(\lfloor n/2 \rfloor)+1$ —— ①

4) Finding initial condition:

when n=1, No addition operation is performed

(ie) $A(1) = 0$ —— ②

In floor function, Backward substitution can't be applied.

Based on smoothness rule, we assume the variable 'n' takes the value as 2^k, which gives correct answer for all values of n.

$$① \longrightarrow A(2^k) = A\left(\frac{2^k}{2}\right)+1$$

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

$$② \longrightarrow A(2^0) = 0$$

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

$$\cancel{A(2^{k-1}) = [A(2^{k-1-1})+1] + 1}$$

$$\cancel{= A(2^{k-2}) + 2}$$

$$\cancel{A(2^{k-2}) = [A(2^{k-3-1})+1}$$

$$= [A(2^{k-2})+1] + 1 = A(2^{k-2}) + 2$$

$$= [A(2^{k-3})+1] + 2 = A(2^{k-3}) + 3$$

In general,

$$A(2^k) = A(2^{k-i}) + i$$

$$\vdots$$

$$A(2^{k-k}) + k$$

Thus $A(2^k) = A(2^0) + k$

$$= A(1) + k$$

$$\boxed{A(2^k) = k} \quad \because A(1) = 0$$

we know that from previous assumption

$$n = 2^k$$

$$k = \log_2 n$$

Substituting the value of 2^k and k in above eqn

$$A(n) = \log_2 n$$

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

Brute force

It is a straightforward approach of solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

In short, the brute force method is 'just do it approach'.

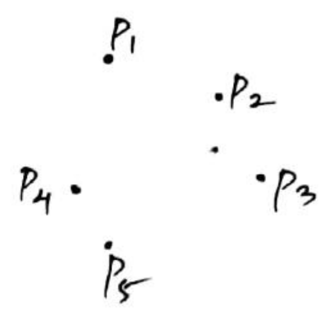
Example 1

closest pair to find the closest points in a set of points, Cartesian co-ordinates, the distance between two points.

Distance between two points,

$$d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Compute the distance between each pair of distinct points and find a pair with the smallest distance [closest pair]



(P_4, P_5) is the closest pair

Algorithm

BruteForceClosestPair (P)

//Input: A list 'P' of 'n' (n ≥ 2) points where $P = P_1(x_1, y_1) \cdots P_n(x_n, y_n)$

//output: Index1 and Index2 of the closest pair of points

```
dmin ← ∞
for i ← 1 to n-1 do
  for j ← i+1 to n do
    d ← sqrt((xi - xj)^2 + (yi - yj)^2)
    if d < dmin
      dmin ← d;
      index1 ← i;
      index2 ← j;
```

Analysis:

Basic operation → Computing the Euclidean distance between two points.

Square roots are irrational numbers and can be found only approximately. So it can be avoided by squaring the square root function.

So if we replace,

$d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ by

$dsqr \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2$, then the

Basic Operation → Squaring operation.

$C(n)$ = No. of times the squaring is done

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

$$= \sum_{i=1}^{n-1} 2[n - i \cancel{-1} \cancel{+1}]$$

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

$$= 2\left[\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i\right]$$

$$= 2\left[n[n-1-1+1] - [1+2+3+\cdots+n-1]\right]$$

$$= 2\left[n(n-1) - \frac{(n-1)(n-1+1)}{2}\right]$$

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

$$= 2\left[n(n-1) - \frac{(n-1)n}{2}\right]$$

$$= 2(n-1)\left[n - \frac{n}{2}\right] \Rightarrow 2(n-1)\left[\frac{2n-n}{2}\right]$$

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

$$C(n) = n^2 - n$$

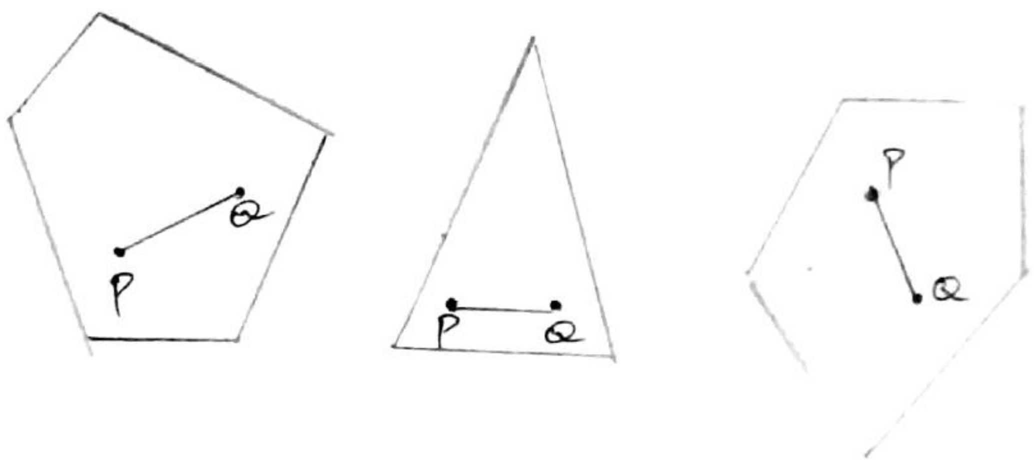
$$\underline{C(n) = O(n^2)}$$

Convex-Hull Problem

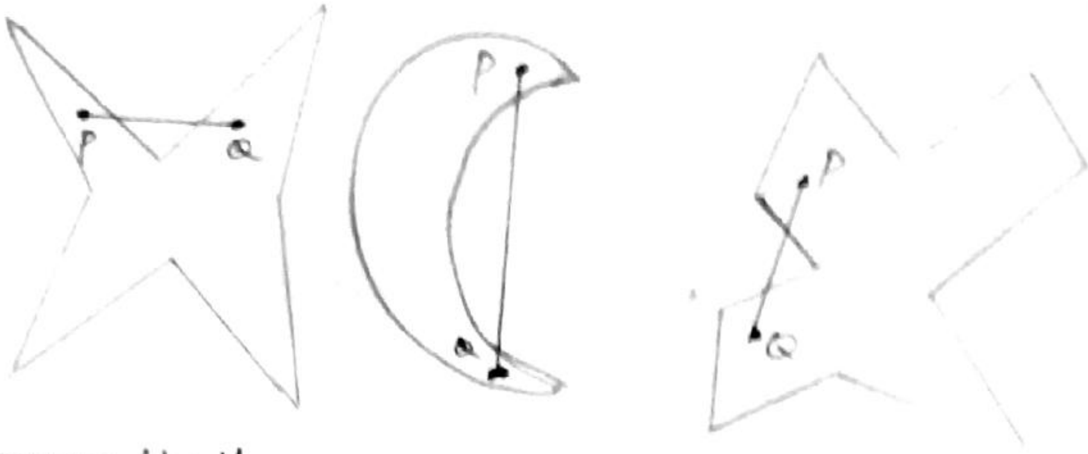
Convex set

A set of points (finite or infinite) in the plane is called convex, if for any two points P and Q in the set, the entire line segment with the endpoints at P and Q belongs to the set.

Convex

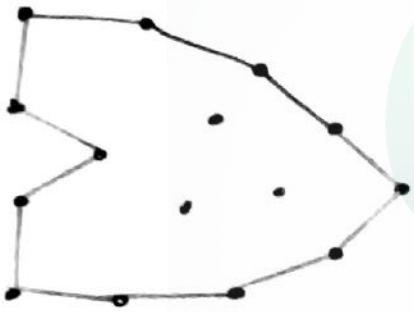


Non-Convex



Convex Hull

Convex hull of a set of 'n' points in the region is the smallest convex polygon that contains all of them either inside or on its boundary.



All the points should be within the surface. No point should be outside a polygon. Join the boundary points.

The convex hull of a set 'S' of 'n' points is the smallest convex set containing S.

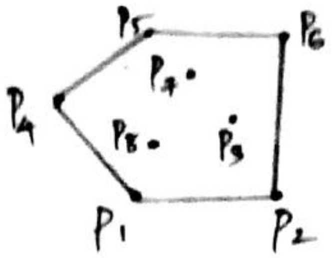
(i) $S = \{P_1, P_2\}$

Extreme points $= P_1, P_2$.

(ii) $S = \{P_1, P_2, P_3\}$

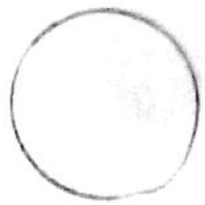
$E.P = P_1, P_2, P_3$

(iii) $S = \{P_1, P_2, \cdots P_8\}$



$E.P = P_1, P_2, P_4, P_5, P_6$

(iv)



All the boundary points are extreme points.

Extreme points

Extreme point is a line segment making up a boundary of a convex hull.

Theorem

The convex hull of any set 'S' of 'n' points [not all on the same line] is a convex polygon with the boundary points [Vertices] at some of the points of 'S'.

If all points do lie on the same line, the polygon degenerates to a line segment with 2 end-points of S.

The convex-hull problem

The convex-hull problem is to construct a convex-hull for a given set of 'n' points and to find the extreme points that will serve as the vertices of the polygon.

Theorem - 2

A line segment connecting 2 points P_1 and P_2 is a part of the convex-hull's boundary, if and only if all other points in the set lie on the same side of the line drawn through these points.

g.

=> For all point above the line,

$ax+by>c$

=> For all points below the line,

$ax+by<c$

=> For all points on the line,

$ax+by=c$

Example 3 : Exhaustive Search

It is a brute force approach to combinational problems (permutations, combinations, subset).

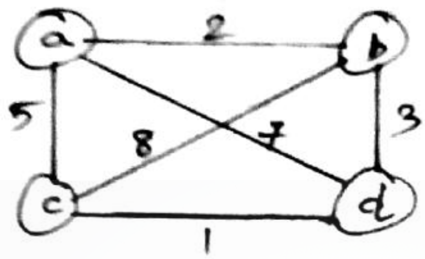
Eg : Travelling Salesman Problem

The problem asks to find the shortest tour through a given set of 'n' cities that visits each city exactly once before returning to the city where it started.

The problem can be modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances.

Then the problem can be stated as the problem of finding the shortest Hamiltonian Circuit of the graph.

The Hamiltonian circuit can also be defined as a sequence of n+1 adjacent vertices $v_{i0}, v_{i1}, \ldots, v_{in-1}, v_{i0}$, where the first vertex of the sequence is the same as the last one while all the other n-1 vertices are distinct.



n = 4

(n-1)! possibilities

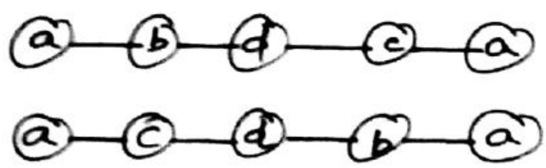
=> 3! = 6 possibilities

To find the shortest tour for a given set of 'n' cities that visits each city exactly once.

Tour	Length	
a-b-c-d-a	l=18	
a-b-d-c-a	l=11	optimal
a-c-b-d-a	l=23	
a-c-d-b-a	l=11	optimal
a-d-c-b-a	l=18	
a-d-b-c-a	l=23	

Minimum cost = 11

Shortest tour :



Example 4 : Knapsack Problem

Given 'n' items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots v_n$ and a knapsack of capacity W, find the most valuable subset of items that fit into the knapsack.

The exhaustive search approach for this problem leads to generating all the subsets of the set of n' items given, computing the total weight of each subset to identify feasible subsets (ie, the ones with the total weight not exceeding the knapsack's capacity), and finding a subset of the largest value among them.

Item	Weight	Value
1	7	42
2	3	12
3	4	40
4	5	25

The no. of subsets of an n-element set is 2^n.

$\Rightarrow \Omega(2^n)$

W = 10

Subset	Total weight	Total Value
ϕ	0	0
{1}	7	42
{2}	3	12
{3}	4	40
{4}	5	25
{1, 2}	10	54

{1,3}	11	82 Not feasible
{1,4}	12	67 Not feasible
{2,3}	7	52
{2,4}	8	37
{3,4}	9	65
{1,2,3}	14	Not feasible
{1,2,4}	15	Not feasible
{2,3,4}	12	Not feasible
{1,3,4}	16	Not feasible
{1,2,3,4}	19	Not feasible

The most valuable subset = {3,4}

Value = 65

Example 5: Assignment Problem

There are 'n' people who need to be assigned to execute 'n' jobs, one person per job. The cost that would take if the i^{th} person is assigned to the j^{th} job is a known quantity $C[i,j]$ for each pair $i,j = 1,2,\ldots n$. The problem is to find the assignment with the minimum total cost.

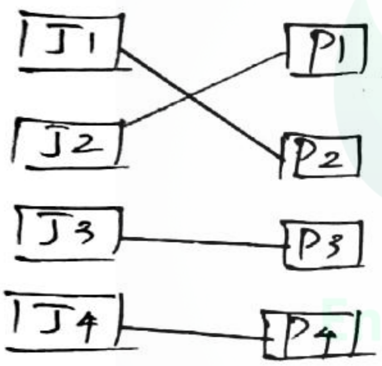
The table entries represent the assignment costs C[i, j]:

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

Find the minimum assignment cost

Person	J_1	J_2	J_3	J_4	Cost
P_1	1	2	3	4	9+4+1+4 = 18
	1	2	4	3	9+4+9+8 = 30
	1	3	2	4	9+8+3+4 = 24
	1	3	4	2	9+8+9+7 = 33
	1	4	2	3	9+6+3+8 = 26
	1	4	3	2	9+6+1+7 = 23
P_2	2	1	3	4	6+2+1+4 = 13 optimal
	2	1	4	3	6+2+9+8 = 25
	2	3	1	4	6+8+7+4 = 25
	2	3	4	1	6+8+9+8 = 31
	2	4	1	3	6+6+7+8 = 27
	2	4	3	1	6+6+1+8 = 21

P_3	3	1	2	4	5+2+3+4 = 14
	3	1	4	2	5+2+9+7 = 23
	3	2	1	4	5+4+7+4 = 20
	3	2	4	1	5+4+9+8 = 26
	3	4	1	2	5+4+7+7 = 23
	3	4	2	1	5+4+3+8 = 20
P_4	4	1	2	3	7+2+3+8 = 20
	4	1	3	2	7+2+1+7 = 17
	4	2	1	3	7+4+7+8 = 26
	4	2	3	1	7+4+1+8 = 20
	4	3	1	2	7+8+7+7 = 29
	4	3	2	1	7+8+3+8 = 26

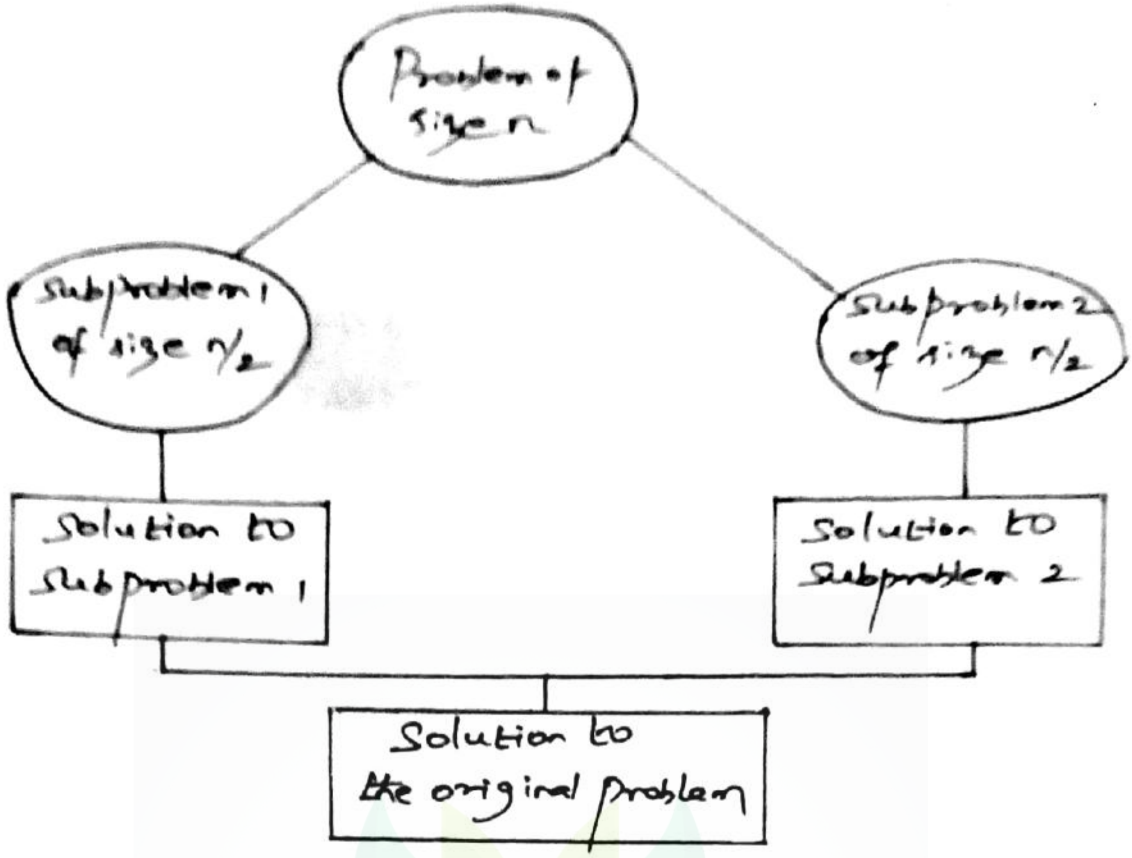


Minimum total cost = 13

DIVIDE AND CONQUER

It works according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.



'n' instances are divided into two instances of size $n/2$

In general,

'n' instances are divided into 'b' instances of size n/b
with a of them needed to be solved.

a & b are constants : $a \geq 1$ & $b \geq 1$

$\boxed{T(n) = aT(n/b) + f(n)}$ → Recurrence relation for divide and conquer

$T(n/b)$ → Time taken for solving each instance

$f(n)$ → Time taken for dividing the problem into smaller ones and combining the solutions.

Example 1: Mergesort

It sorts a given array A[0 --- n-1] by dividing into two halves, A[0 --- ⌈n/2⌉-1] and A[⌈n/2⌉ --- n-1] sorting each of them recursively and then merging the two sorted smaller arrays into a single sorted one.

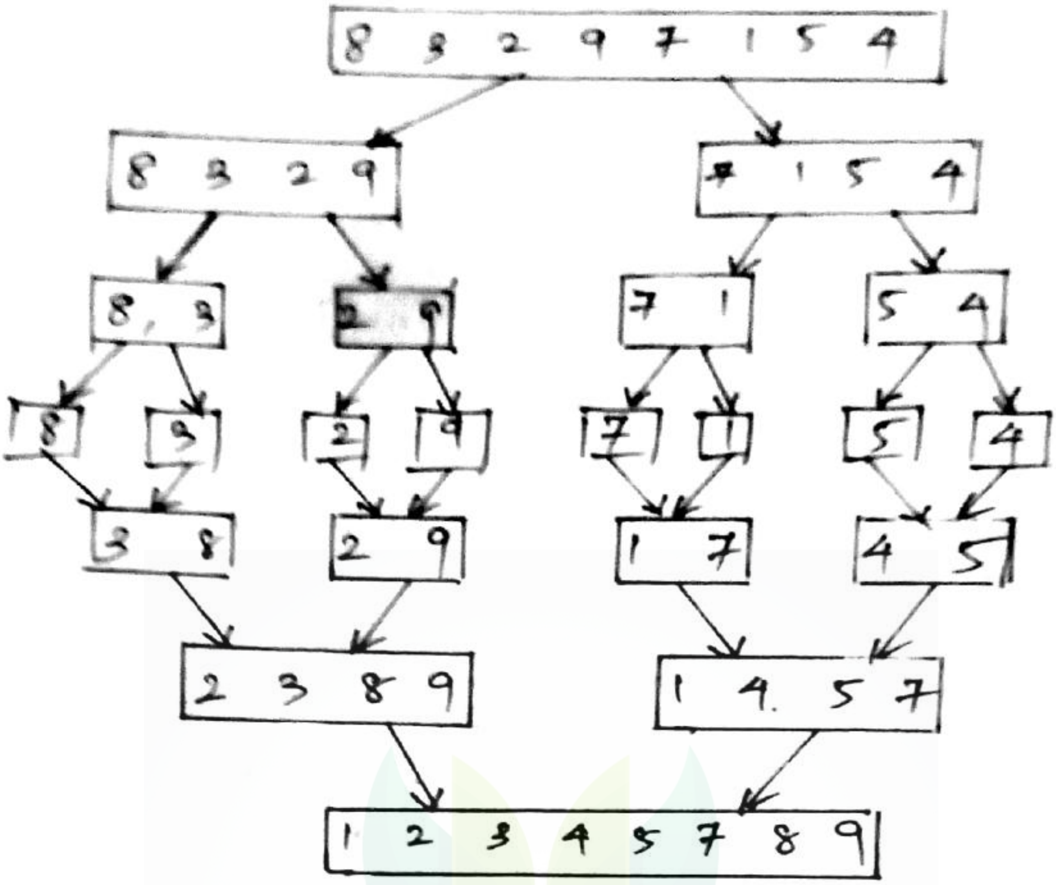
Algorithm: MergeSort (A[0 --- n-1])
//Sorts array A[0 --- n-1] by recursive mergesort
//Input: An array A[0 --- n-1] of orderable elements.
//Output: Array[0 --- n-1] sorted in non-decreasing order.

```
If n > 1
  Copy A[0 --- ⌊n/2⌋-1] to B[0 --- ⌊n/2⌋-1]
  Copy A[⌊n/2⌋ --- n-1] to C[0 --- ⌈n/2⌉-1]
  Mergesort (B[0 --- ⌊n/2⌋-1])
  Mergesort (C[0 --- ⌈n/2⌉-1])
  Merge (B, C, A)
```

Algorithm: Merge(B[0 -- p-1], C[0 -- q-1], A[0 -- p+q-1])
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0 -- p-1] and C[0 -- q-1] both sorted
//Output: Sorted array A[0 -- p+q-1] of the elements of B and C

```
i ← 0 ; j ← 0 ; k ← 0
while i < p and j < q do
   if B[i] ≤ C[j]
      A[k] ← B[i] ; i ← i+1
   else A[k] ← C[j] ; j ← j+1
      k ← k+1
if i = p
   Copy C[j ... q-1] to A[k ... p+q-1]
else copy B[i -- p-1] to A[k -- p+q-1]
```

Example: 8, 3, 2, 9, 7, 1, 5, 4



Divide algorithm

A[0...n-1] =>	A[0]							A[7] A[n-1]	
	8	3	2	9	7	1	5	4	n = 8
	0	1	2	3	4	5	6	7	

$n > 1 \Rightarrow 8 > 1$ T

Copy $A[0 \ldots \lfloor n/2 \rfloor - 1]$ to $B[0 \ldots \lfloor n/2 \rfloor - 1]$

$A[0 \ldots 3]$ to $B[0 \ldots 3]$ => B[8 3 2 9]

Copy $A[\lfloor n/2 \rfloor \ldots n-1]$ to $C[0 \ldots \lceil n/2 \rceil - 1]$

$A[4 \ldots 7]$ to $C[0 \ldots 3]$ => C[7 1 5 4]

Solve & Combine algorithm

B

0	1	2	3
2	3	8	9

p=4

C

0	1	2	3
1	4	5	7

q=4

Initial Condition

i=0, j=0, k=0

while $i < p$ and $j < q$

$0 < 4$ & $0 < 4$ T

1) $B[i] \leq C[j]$

$2 \leq 1$ F $j = 1$

A

1							
0	1	2	3	4	5	6	7

$k = 1$

2) $B[i] \leq C[j]$

$2 \leq 4$ T $i = 1$

A

1	2	

$k = 2$

3) $B[i] \leq C[j]$

$3 \leq 4$ T $i = 2$

A

1	2	3	

$k = 3$

4) $B[i] \leq C[j]$

$8 \leq 4$ F $j = 2$

A

1	2	3	4	

$k = 4$

5) $B[i] \leq C[j]$

$8 \leq 5$ F $j = 3$

A

1	2	3	4	5	

$k = 5$

6) $B[i] \leq C[j]$

$8 \leq 7$ T $j = 4$

A

1	2	3	4	5	7	
0	1	2	3	4	5	

$k = 6$

$j < q$ F

$4 < 4$

if $i = p$ $2 = 4$ F

Copy $B[i..p-1]$ to $A[k..p+q-1]$

$B[2..3]$ to $A[6..7]$

= B

8	9

to A

1	2	3	4	5	7	8	9

Analysis for recurrence relation for the no. of key comparisons.

$$C(n) = 2C(n/2) + C_{merge}(n), \; n > 0, \; C(1) = 0$$

$C_{merge}(n)$ = no. of key comparisons performed during the merging stage

$C_{merge}(n) = n-1$ Comparisons

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

According to master theorem

$$\boxed{C(n/2) = n\log n}$$

$\therefore C(n) = 2n\log n + n - 1$

$$\therefore \boxed{C(n) = O(n\log n)}$$

Example 2: Quicksort

It is another important sorting algorithm based on the divide and conquer approach. It divides its input's elements according to their value in the array. Specifically it rearranges elements of a given array $A[0 \ldots n-1]$ to achieve its partition, in a situation where all the elements before some position 's' are smaller than or equal to $A[S]$ and all the elements after position s are greater than or equal to $A[S]$:

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are } \geq A[s]}$$

Algorithm: Quicksort($A[l \ldots r]$)

// Sorts a subarray by quicksort

// Input: A subarray $A[l \ldots r]$ of $A[0 \ldots n-1]$ defined by its left and right indices l and r

// Output: Subarray $A[l \ldots r]$ sorted in non-decreasing order.

s ← Partition (A[l..r]) // s is a split position
Quicksort (A[l..s-1])
Quicksort (A [s+1..r])

Algorithm: Partition (A[l..r])
// Partition a subarray by using its first element as a pivot
// Input: A subarray A[l..r] of A[0..n-1], defined by its left and right indices l and r (l<r)
// Output: A partition of A[l..r], with the split position returned as this function's value.

```
p ← A[l]
i ← l; j ← r+1.
repeat
    repeat i ← i+1 until A[i] ≥ p
    repeat j ← j-1 until A[j] ≤ p
    swap (A[i], A[j])
until i ≥ j
swap(A[i], A[j]) // undo last swap when i ≥ j
swap (A[l], A[j])
return j
```

Quicksort:

Steps:

1. Take the first element as pivot element
2. Second = i
3. last = j
4. From left, Check i > pivot, stop
5. From right, Check j < pivot, stop
6. If Index[i] < Index[j], then swap A[i] & A[j]
7. Increment i and decrement j

8. Repeat the above steps
9. If i crosses j, swap A[j] & pivot
10. This is the permanent position of the pivot in the sorted array.
11. Split the array, before and after pivot.
12. Before the pivot, do steps 1 to 11 for the array
13. After the pivot, do steps 1 to 11 for the array

Example:

5 3 1 9 8 2 4 7

Step 1:

	0	1	2	3	4	5	6	7
A	5	3	1	9	8	2	4	7
	↓ pivot	↓ i						↓ j

From left,

Check till $\boxed{i > pivot}$

$3<5, 1<5, \underline{9>5}$: stop i

Current position of $i = 9$

From right,

Check till $\boxed{j < pivot}$

$7>5, \underline{4<5}$: stop j

Current position of $j = 4$

	0	1	2	3	4	5	6	7
A	5	3	1	9	8	2	4	7
	↓ Pivot			↓ i			↓ j	

If index[i] < index[j], swap A[i] & A[j]

$3<6$, $\therefore A[i]=4 \quad A[j]=9$

	0	1	2	3	4	5	6	7
A	5	3	1	4	8	2	9	7
	pivot			i			j	

Increment i and decrement j

5	3	1	4	8	2	9	7
pivot				i	j		

Step 2 :

from left 8>5, stop i
from right 2<5, stop j
Index [i] < Index [j]
swap A[i] & A[j]
Increment i and decrement j

5	3	1	4	2	8	9	7
pivot				j	i		

i crosses j,
swap A[j] and pivot

0	1	2	3	4	5	6	7
2	3	1	4	$\boxed{5}$	8	9	7
				p			

Divide the array into 2, before and after pivot

0	1	2	3
2	3	1	4
p	i		j

→ 1st array

From left, i > pivot
3 > 2 stop i
From left, j < pivot
4 > 2, 1 < 2 stop j

Current $i=3$, $j=1$

0	1	2	3
2	3	1	4
	i	j	

index [i] < index [j]

1 < 2

swap A [i] and A [j]

0	1	2	3
2	1	3	4
	i	j	

Increment i and decrement j

0	1	2	3
2	1	3	4
	j	i	

i crosses j

∴ swap A[j] and pivot

0	1	2	3
1	$\boxed{2}$	3	4

This is the permanent position of the pivot element in the sorted array.

Before the pivot, there is only one element.

∴ No need of sorting.

After the pivot, there are 2 elements

2	3
3	4
p	i=j

→

$p=3$, $i=j=4$

From left, $i>p$, $4>3$ stop i

From right, $j<p$ $4>3$

index[i] < index [j], swap A[i] & A[j]

Since $i=j$, no swapping

2	3
3	4

After the pivot, we have

5	6	7
8	9	7
p	i	j

→ 2^{nd} array

From left, $i > p$, $9 > 8$ stop i

From right, $j < p$, $7 < 8$ stop j

Index [i] < Index [j]

$6 < 7$

swap A[i] and A[j]

5	6	7
8	7	9
p	i	j

Increment i and decrement j

5	6	7
8	7	9
p	j	i

i crosses j

∴ swap pivot and A [j]

5	6	7
$\boxed{7}$	8	9

After the pivot element, we have

6	7
8	9
	i=j

$p = 8 \quad i = j = 9$

From left, $i > p$ $9 > 8$ stop i

From right, $j < p$ $9 > 8$

Index [i] < Index [j], swap A[i] & A[j]

Since i=j, no swapping

Finally, the sorted array contains

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

Analysis :

Best case :

$$C_{best}(n) = 2C_{best}(n/2) + n$$

According to master theorem,

$$C_{best}(n/2) = n \log n$$

$$C_{best}(n) = 2n\log n + n$$

$$\boxed{C_{best}(n) = O(n \log n)}$$

Worst Case :

$$C_{worst}(n) = n^2$$

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

Average case :

$$\boxed{C_{avg}(n) = 1.38 n \log n}$$

Example 3: Binary Search

It is an efficient algorithm for searching in a sorted array. It works by comparing a search key 'K' with the array's middle element A[m]. If they match, the algorithm stops, otherwise the same operation is repeated recursively for the first half of the array if K < A[m], and for the second half if K > A[m]

$$\underbrace{A[0] \cdots A[m-1]}_{\text{search here if } K < A[m]} \quad \overset{K}{A[m]} \quad \underbrace{A[m+1] \cdots A[n-1]}_{\text{search here if } K > A[m]}$$

Algorithm:

BinarySearch (A[0 .. n-1], K)

$l = 0, r = n-1$

while $l \leq r$ do

$m = \lfloor (l+r)/2 \rfloor$

if $K = A[m]$ return m

else if $K < A[m]$ $r = m-1$

else $l = m+1$

return -1

Example:

0	1	2	3	4	5	$\boxed{6}$	7	8	9	10	11	12
3	14	27	31	39	42	$\boxed{55}$	70	74	81	85	93	98

$K = 70, n = 13$

All elements should be in sorted order

l = left, r = right

$l = 0, r = 12$

$l < r$, $0 < 12$ T

$m = \lfloor (l+r)/2 \rfloor = \lfloor (0+12)/2 \rfloor = 6$

$K = A[m]$, $70 \neq 55$

$\therefore K < A[m]$, $70 < 55$ F

$\therefore$ K is greater $K > A[m]$

then $l = m+1$

$l = 6+1 = 7$

7	8	$\boxed{9}$	10	11	12
70	74	$\boxed{81}$	85	93	98
l					r

Now, $l = 7$, $r = 12$

$l < r$, $7 < 12$, T

$\therefore m = \lfloor (l+r)/2 \rfloor = \lfloor 19/2 \rfloor = \lfloor 9.5 \rfloor = 9$

$m = 9$

$K = A[m]$ $70 \neq 81$

$K < A[m]$, $70 < 81$, T

$\therefore l = 7$, $r = m - 1$
$= 9 - 1 \Rightarrow r = 8$

7	8
70	74
l	r

Now, $l = 7$, $r = 8$

$l < r$, $7 < 8$, T

$\therefore m = \lfloor (l+r)/2 \rfloor = \lfloor 15/2 \rfloor = \lfloor 7.5 \rfloor = 7$

$m = 7$

$K = A[m]$, $70 = 70$

return 7

$\therefore$ 7 is the location of the search element.

Analysis:

Best case:

The no. of key comparisons performed on 'n' i/p size is only one.

$C_{best}(n) = 1$

$$\boxed{C_{best}(n) = O(1)}$$

(i.e) Key appears in the first middle.

Worst case:

If the 'k' appears as last middle (or) key element is not present,

$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1$

Substitute $n = 2^k$ in above eq.

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

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

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

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

$C_w(2^k) = C_w(2^{k-i})+i$

Put $i=k$ in above eqn

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

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

we know $n=2^k$ $\therefore$ $k=\log_2 n$

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

$$\boxed{C_w(n) = O\log n}$$

Average case :

The no. of key comparisons is slightly smaller than the worst case,

$C_{avg}(n) \approx \log n$

$$\boxed{C_{avg}(n) = O\log n}$$

Example 4: Multiplication of Large integers

Method 1:

$n_1 = 23$, $n_2 = 14$

$23 = 2\times10^1 + 3\times10^0$

$14 = 1\times10^1 + 4\times10^0$

$23 * 14 = (2\times10^1+3\times10^0) * (1\times10^1+4\times10^0)$

$= (2\times1)10^2 + (2\times4)10^1 + (3\times1)10^1 + (3\times4)10^0$

$= (2\times1)10^2 + (2\times4+3\times1)10^1 + (3\times4)10^0$

$= 200+110+12$

$= 322$

No. of multiplications $= 4$

Method 2 :

Brute Force method

$$\begin{array}{r} 23 \times 14 \\ \hline 92 \\ 23 \\ \hline 322 \\ \hline \end{array}$$

No. of multiplications = 4

n = no. of digits = 2

No. of multiplications = $n^2 = 2^2 = 4$

Method 3 :

Divide and Conquer method

For any pair of 2 digit integers ($n=2$)

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

$C = a * b = c_2 10^2 + c_1 10^1 + c_0$

where

$c_2 = a_1 * b_1$ (product of their first digits)

$c_0 = a_0 * b_0$ (product of their second digits)

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$

Given :

$a = \underset{a_1}{2}\underset{a_0}{3}$ $\quad b = \underset{b_1}{1}\underset{b_0}{4}$

$c_2 = a_1 * b_1 \Rightarrow 2 * 1 = 2$

$c_0 = a_0 * b_0 \Rightarrow 3 * 4 = 12$

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$

$= (2+3) * (1+4) - (2+12)$

$= 5 * 5 - 2 - 12$

$= 25 - 14$

$c_1 = 11$

$c_2 = 2$, $c_0 = 12$, $c_1 = 11$

$c = c_2 10^2 + c_1 10 + c_0$

$= 2 * 10^2 + 11 \times 10^1 + 12$

$= 200 + 110 + 12$

$\boxed{c = 322}$

For multiplying 2 two digit integers, we have

$c = c_2 10^2 + c_1 10^1 + c_0$

For multiplying 2 'n' digit integers,

$$\boxed{c = c_2 10^n + c_1 10^{n/2} + c_0}$$

Analysis:

Recurrence equation:

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

Substitute $n = 2^k$ in above eqn

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

$= 3[3M(2^{k-2})]$

$= 3^2 M(2^{k-2})$

$= 3^2 [3M(2^{k-3})]$

$= 3^3 M(2^{k-3})$

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

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

$M(2^{k-2}) = 3M(2^{k-3})$

$M(2^k) = 3^i M(2^{k-i})$

Sub $i = k$ in above eqn

$M(2^k) = 3^k M(2^0)$

$= 3^k$ $\quad [n = 2^k, \ k = \log_2 n]$

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

$= n^{\log_2 3} \approx n^{1.585}$ $\quad [a^{\log_b c} = c^{\log_b a}]$

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

Example 5: Strassen's Matrix Multiplication

The principal concept behind this method is to find the product C of two 2-by-2 matrices A and B with just seven multiplications, when compared to the eight multiplications required by the brute-force algorithm.

Example: Find C=AB where A and B are n-dimensional matrices

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Method 1: Brute force method

$$C = \begin{bmatrix} 1*5+2*7 & 1*6+2*8 \\ 3*5+4*7 & 3*6+4*8 \end{bmatrix}$$

$$C = \begin{bmatrix} 5+14 & 6+16 \\ 15+28 & 18+32 \end{bmatrix}$$

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

No. of multiplications = $8 = n^3$

Method 2: Divide & conquer method

It is accomplished by using the following formula:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1+m_4-m_5+m_7 & m_3+m_5 \\ m_2+m_4 & m_1+m_3-m_2+m_6 \end{bmatrix}$$

where

$$m_1 = (a_{00}+a_{11}) * (b_{00}+b_{11})$$

$$m_2 = (a_{10}+a_{11}) * b_{00}$$

$$m_3 = a_{00} * (b_{01}-b_{11})$$

$m_4 = a_{11} * (b_{10} - b_{00})$

$m_5 = (a_{00} + a_{01}) * b_{11}$

$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$

$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

$m_1 = (1+4) * (5+8) = 5 * 13 = 65$

$m_2 = (3+4) * 5 = 7 * 5 = 35$

$m_3 = 1 * (6-8) = -2$

$m_4 = 4 * (7-5) = 8$

$m_5 = (1+2) * 8 = 24$

$m_6 = (2-0) * (5+6) = 2 * 11 = 22$

$m_7 = (2-4) * (6+8) = -2 * 14 = -28$

$$C = \begin{bmatrix} 65+8-24-28 & -2+24 \\ 35+8 & 65+(-2)-35+22 \end{bmatrix}$$

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

No. of additions / subtractions = 18

No. of multiplications = 7

Analysis:

$M(n)$ = No. of multiplications performed by Strassen's algorithm

For multiplying two n-by-n matrices (where n is a power of 2), we get the following recurrence relation:

$M(n) = 7M(n/2)$ for $n > 1$, $M(1) = 1$

Substitute $n = 2^k$

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

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

$M(2^{k-2}) = 7M(2^{k-3})$

$$M(2^k) = 7M(2^{k-1})$$
$$= 7[7M(2^{k-2})]$$
$$= 7^2 M(2^{k-2})$$

$$= 7^2[7M(2^{k-3})]$$
$$= \underline{7^3 M(2^{k-3})}$$

In general,

$$M(2^k) = 7^i M(2^{k-i})$$

sub $i=k$ in above eqn

$$M(2^k) = 7^k M(2^0)$$
$$M(2^k) = 7^k M(1)$$

$n = 2^k$, $k = \log_2 n$, $M(1) = 1$

$\therefore M(n) = 7^{\log_2 n}$ $\quad \boxed{a^{\log_b c} = c^{\log_b a}}$

$\therefore M(n) = n^{\log_2 7}$

$$\underline{\approx n^{2.807}}$$

Which is <u>smaller than n^3 (Brute-force)</u>.

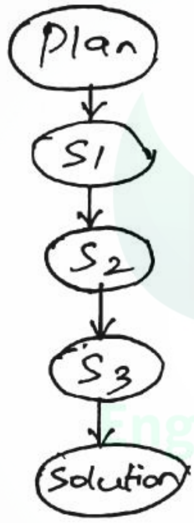
$\therefore$ <u>Divide and conquer is efficient than brute force.</u>

Prepared by Verified by HOD Approved by

30/1/18

Dynamic Programming

It is a technique for solving problems with overlapping subproblems. It suggest solving each of the smaller subproblems only once and recording the results in a table from which we can then obtain a solution to the original problem.



$S_1 \rightarrow$ output is input to S_2

Example 1 : Computing a Binomial Co-efficient

It is a standard example of applying dynamic programming to a non optimization problem.

Binomial Co-efficient, denoted as $C(n,k)$ or $\binom{n}{k}$ is the number of Combinations (subsets) of k elements from an n-element set $(0 \leq k \leq n)$. The name "binomial coefficients" comes from the binomial formula:

$$(a+b)^n = C(n,0)a^n + \cdots + C(n,k)a^{n-k}b^k + \cdots + C(n,n)b^n$$

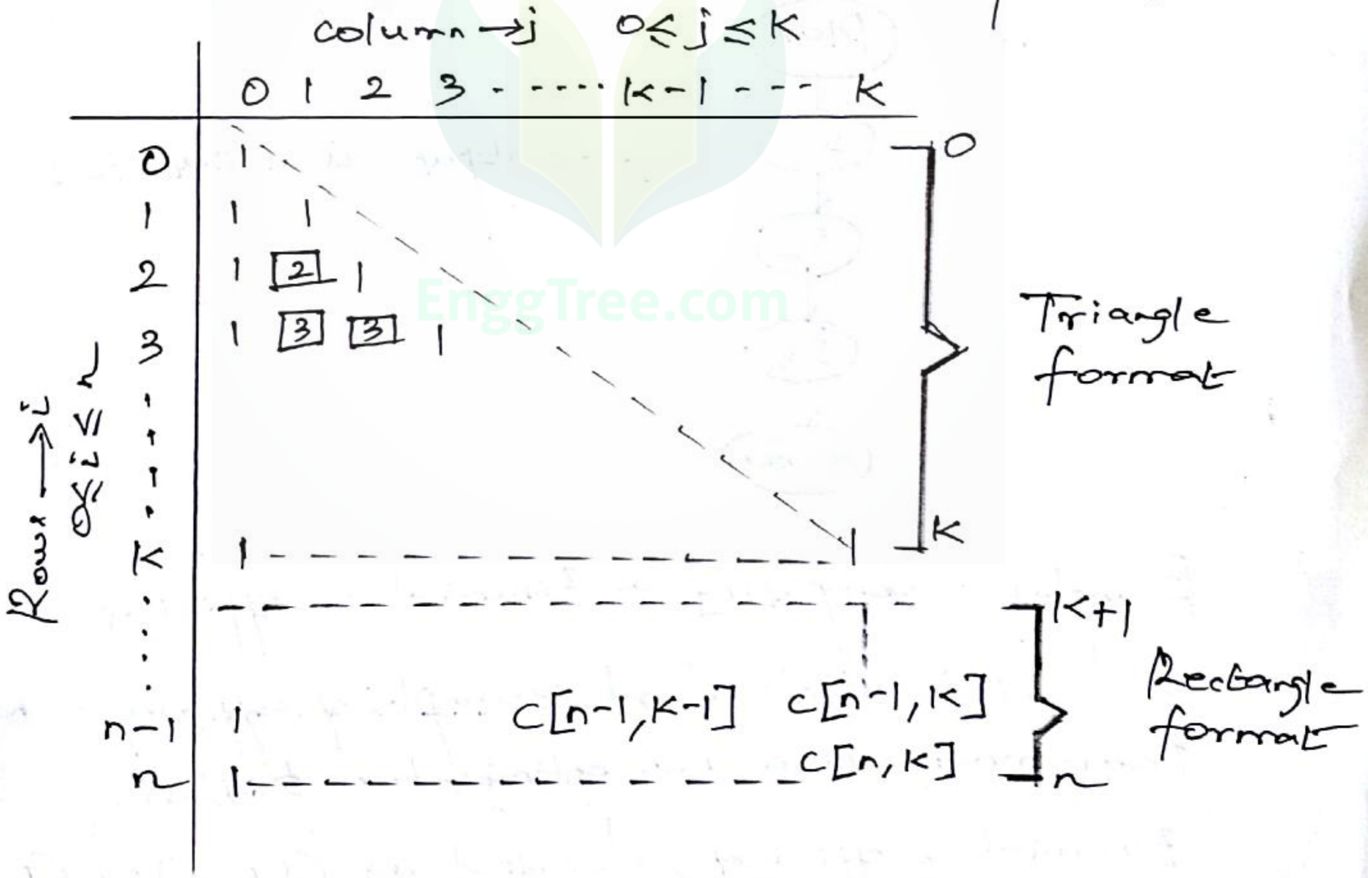
The two properties of binomial coefficients are:

1. $C(n,k) = C(n-1,k-1) + C(n-1,k)$, for $n > k > 0$
2. $C(n,0) = C(n,n) = 1$

Task:

The problem of constructing $C(n,k)$ in terms of the smaller and overlapping problems of computing $C(n-1,k-1)$ and $C(n-1,k)$ by dynamic programming technique.

⇒ Record the values of binomial coefficients in a table of $n+1$ rows and $k+1$ columns numbered from 0 to n and from 0 to k respectively.



$C(0,0) = C(1,1) = C(2,2) = C(n,n) = 1$

$C(1,0) = C(2,0) = C(3,0) = C(n,0) = 1$

$C(\overset{n}{2},\overset{k}{1}) = C(2-1, 1-1) + C(2-1,1)$

$= C(1,0) + C(1,1) \Rightarrow 1+1 = 2$

$$c(\overset{n}{3},\overset{k}{1}) = c(3-1,1-1)+c(3-1,1)$$
$$= c(2,0)+c(2,1)$$
$$= 1+2$$
$$= 3$$

$$c(\overset{n}{3},\overset{k}{2}) = c(3-1,2-1)+c(3-1,2)$$
$$= c(2,1)+c(2,2)$$
$$= 2+1$$
$$= 3$$

Algorithm: Binomial (n, k)

```
for i ← 0 to n do
  for j ← 0 to min(i,k) do
    if j=0 (or) j=i
    c[i,j] ← 1
 else c[i,j] ← c[i-1,j-1]+c[i-1,j]
 return c[n,k]
```

Analysis:

$A(n,k)$ = No. of additions performed by algorithm in computing $c(n,k)$

$$A(n,k) = \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n} \sum_{j=1}^{k} 1$$

[∵ first $k+1$ rows of the table form a triangle and remaining $n-k$ rows form a rectangle, $A(n,k)$ is split into above two parts]

$$A(n,k) = \sum_{i=1}^{k} [i-1\cancel{-1}\cancel{+1}] + \sum_{i=k+1}^{n} [k\cancel{-1}\cancel{+1}]$$

$$= \sum_{i=1}^{k} [i-1] + \sum_{i=k+1}^{n} k$$

$$= \sum_{i=1}^{k} i - \sum_{i=1}^{k} 1 + \sum_{i=k+1}^{n} k$$

$$= \sum_{i=1}^{k} i - \sum_{i=1}^{k} 1 + k[n-k\cancel{+1}+\cancel{1}]$$

$$= \sum_{i=1}^{k} -[k\cancel{-1}+\cancel{1}] + k(n-k)$$

$$= 1+2+3+\cdots k - (k) + k(n-k)$$

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

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

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

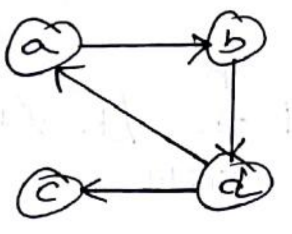
$$\therefore A(n,k) \in \Theta(nk)$$

Example2 : Warshall's Algorithm :

It is used to compute the transitive closure of a directed graph.

The adjacency matrix $A=[a_{ij}]$ of a directed graph is the boolean matrix that has 1 in its i^{th} row and j^{th} column, if and only if there is a directed edge from the i^{th} vertex to the j^{th} vertex.

Diagraph



DIAGRAPH – Directed graph

Adjacency Matrix (A) =

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ & a & b & c & d \\ \hline 1\,a & & & & \\ 2\,b & & & & \\ 3\,c & & & & \\ 4\,d & & & & \end{array} \quad \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

n = 4 nodes

Value of self node = 0

value of directed vertex = 1

Algorithm : Warshall(A[1...n, 1...n])

//Input: Adjacency Matrix A [$R^0 = A$]

//output: Transitive closure R

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 ($R^{(k-1)}[i,k]$ and $R^{(k-1)}[k,j]$)

return $R^{(n)}$

Task:

Find R^n

n = 4 nodes

∴ To find R^4

$R^0 \rightarrow R^1 \rightarrow R^2 \rightarrow R^3 \rightarrow R^4$

Step 1:

Find R^1, k=1,

$R^1[1,1] = R^{(1-1)}[1,1] + (R^{(1-1)}[1,1] * R^{(1-1)}[1,1])$

$= R^0[1,1] + (R^0[1,1] * R^0[1,1])$

$= 0 + (0*0) = 0+0 = 0$

$R^1[1,2] = 1 + (0*1) = 1+0 = 1$

$R^1[1,3] = 0 + (0*0) = 0+0 = 0$

$R^1[1,4] = 0 + (0*0) = 0+0 = 0$

$R^1[2,1] = 0+(0*0) = 0$

$R^1[2,2] = 0+(0*1) = 0$

$R^1[2,3] = 0+(0*0) = 0$

$R^1[2,4] = 1+(0*0) = 0$

$R^1[3,1] = 0+(0*0) = 0$

$R^1[3,2] = 0+(0*1) = 0$

$R^1[3,3] = 0+(0*0) = 0$

$R^1[3,4] = 0+(0*0) = 0$

$R^1[4,1] = 1+(1*0) = 1$

$R^1[4,2] = 0+(1*1) = 1$

$R^1[4,3] = 1+(1*0) = 1$

$R^1[4,4] = 0+(1*0) = 0$

$$R^1 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Step 2: Find R^2, $k=2$, Take i/p as R^1

Following the above steps, we get

$$R^2 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Step 3: Find R^3, $k=3$, Take i/p as R^2

following the above steps, we get

$$R^3 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Step 4: Find R^4, $k=4$, Take i/p as R^3

Following the above steps, we get

$$R^4 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Hence the transition matrix R^4 is found.

The values of R^1, R^2, R^3 and R^4 can be found using short cut method and checked against the computational values of R^1, R^2, R^3 and R^4.

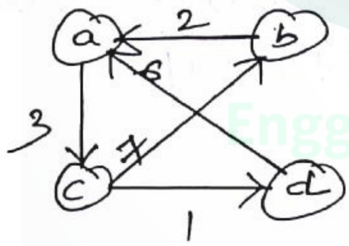
Time complexity of Warshall's algorithm is

$\boxed{O(n^3)}$ (for 3 loops)

Example 3: Floyd's algorithm for the all-pairs shortest-paths problem

Given a weighted connected graph (undirected or directed), Floyd's algorithm is used to find the distances (the length of the shortest paths) from each vertex to all the vertices.

Eg:



weighted diagraph

Input: weighted diagraph / weighted matrix

Output: Distance matrix of shortest path

There are 3 values

$\infty \rightarrow$ No direct path

$0 \rightarrow$ self node

$d_{ij} \rightarrow$ distance from vertex i to j

Distance matrix,

$$D^0 = W = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Algorithm: Floyd (W[1..n, 1..n])

$D^{(0)} \leftarrow W$

for $k \leftarrow 1$ to n do

for $i \leftarrow 1$ to n do

for $j \leftarrow 1$ to n do

$D^{(k)}[i,j] = \min(D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j])$

return D

Task:

Find D^4

$n = 4$ nodes

$\therefore$ To find D^4

$D^0 \rightarrow D^1 \rightarrow D^2 \rightarrow D^3 \rightarrow D^4$

$$D_0 = \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \\ a & b & c & d \end{array} \\ \begin{array}{c} 1\,a \\ 2\,b \\ 3\,c \\ 4\,d \end{array} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Step 1:

Find D^1, $k=1$

$D^1[1,1] = \min(D^{(1-1)}[1,1],\ D^{(1-1)}[1,1] + D^{(1-1)}[1,1])$

$= \min(D^0[1,1],\ D^0[1,1] + D^0[1,1])$

$= \min(0, 0+0)$

$= \min(0,0)$

$= 0$

$D^1[1,2] = \min(\infty, 0+\infty) = \infty$

$D^1[1,3] = \min(3, 0+3) = 3$

$D^1[1,4] = \min(\infty, 0+\infty) = \infty$

$D^1[2,1] = \min(2, 2+0) = 2$

$D^1[2,2] = \min(0, 2+\infty) = 0$

$D^1[2,3] = \min(\infty, 2+3) = 5$

$D^1[2,4] = \min(\infty, 2+\infty) = \infty$

$D^1[3,1] = \min(\infty, \infty+3) = \infty$

$D^1[3,2] = \min(7, \infty+\infty) = 7$

$D^1[3,3] = \min(0, \infty+3) = 0$

$D^1[3,4] = \min(1, \infty+\infty) = 1$

$D^1(4,1) = \min(6, 6+0) = 6$

$D^1(4,2) = \min(\infty, 6+\infty) = \infty$

$D^1(4,3) = \min(\infty, 6+3) = 9$

$D^1(4,4) = \min(0, 6+\infty) = 0$

$$D^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & \infty & 3 & \infty \\ 2 & 2 & 0 & 5 & \infty \\ 3 & \infty & 7 & 0 & 1 \\ 4 & 6 & \infty & 9 & 0 \end{array}$$

Step 2: Find D^2, $k=2$, Take i/p as D^1

Following the above steps, we get

$$D^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & \infty & 3 & \infty \\ 2 & 2 & 0 & 5 & \infty \\ 3 & 9 & 7 & 0 & 1 \\ 4 & 6 & \infty & 9 & 0 \end{array}$$

Step 3: Find D^3, $k=3$, Take i/p as D^2

Following the above steps, we get

$$D^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 10 & 3 & 4 \\ 2 & 2 & 0 & 5 & 6 \\ 3 & 9 & 7 & 0 & 1 \\ 4 & 6 & 16 & 9 & 0 \end{array}$$

Step 4: Find D^4, $k=4$, Take i/p as D^3

Following the above steps, we get

$$D^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 10 & 3 & 4 \\ 2 & 2 & 0 & 5 & 6 \\ 3 & 7 & 7 & 0 & 1 \\ 4 & 6 & 16 & 9 & 0 \end{array}$$

The above values can be compared against the values found using short cut method for checking.

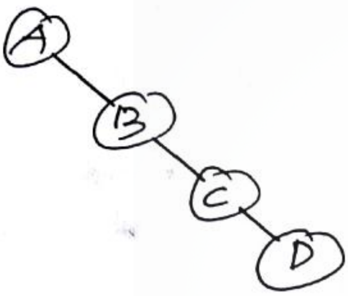
Time Complexity of Floyd's algorithm $= O(n^3)$

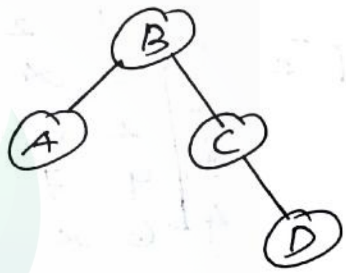
Example 4: Optimal Binary Search Tree (OBST)

1. Minimizing the average no. of key comparisons in a search operation.
2. Binary tree has maximum 2 child nodes.
3. Binary search tree is used to perform search operation in binary tree.

   optimal → reduced no. of comparisons

Example

ⓐ BST 

ⓑ OBST 

$\boxed{Key = D}$

Property : $\boxed{\text{left} < \text{root} < \text{right}}$

No. of comparisons = 4

No. of comparisons = 3

ⓑ is optimal than ⓐ

Construct optimal binary search tree (OBST) for the following key sets

Key	A	B	C	D
Probability	0.1	0.2	0.4	0.3

Total Probability = 1

	0	1	2	3	4
1	0	$0.1^{(1)}$	$0.4^{(2)}$	$1.1^{(3)}$	$1.7^{(3)}$
2		0	$0.2^{(2)}$	$0.8^{(3)}$	$1.4^{(3)}$
3			0	$0.4^{(3)}$	$1.0^{(3)}$
4				0	$0.4^{(4)}$
5					0

Case 1:

$d=1$ [Take the key individually]

No. of nodes	1	2	3	4
Nodes	A	B	C	D
Frequency	0.1	0.2	0.4	0.3

Node 1 as root

$C(1,1) = 0.1^{(1)}$

Node 2 as root

$C(2,2) = 0.2^{(2)}$

Node 3 as root

$C(3,3) = 0.4^{(3)}$

Node 4 as root

$C(4,4) = 0.3^{(4)}$

Case 2:

$d=2$ [Take two values at a time]

1	2	
A	B	[1,2]
0.1	0.2	

$0.1+0.2+\min\begin{Bmatrix} 0.2 \leftarrow 1 \\ 0.1 \leftarrow 2 \end{Bmatrix}$

$0.1+0.2+0.1^{(2)} = 0.4^{(2)}$

2	3	
B	C	[2,3]
0.2	0.4	

$0.2+0.4+\min\begin{Bmatrix} 0.4 \leftarrow 2 \\ 0.2 \leftarrow 3 \end{Bmatrix}$

$0.2+0.4+0.2^{(3)}$

$= 0.8^{(3)}$

3 4 $[3,4]$

C D

0.4 0.3

$$0.4+0.3+\min\begin{Bmatrix}0.3 \leftarrow 3\\ 0.4 \leftarrow 4\end{Bmatrix}$$

$$=0.4+0.3+0.3^{(3)}$$

$$=1.0^{(3)}$$

Case 3:

$l=3$ [Take three values at a time]

1 2 3 $[1,3]$

A B C

0.1 0.2 0.4

$$0.1+0.2+0.4+\min\begin{Bmatrix}0.8 \leftarrow 1 & [2,3]\\ 0.1+0.4 \leftarrow 2 & \\ 0.4 \leftarrow 3 & [1,2]\end{Bmatrix}$$

$$0.7+0.4^{(3)}$$

$$=1.1^{(3)}$$

2 3 4 $[2,4]$

B C D

0.2 0.4 0.3

$$0.2+0.4+0.3+\min\begin{Bmatrix}1.0 \leftarrow 2 & [3,4]\\ 0.2+0.3 \leftarrow 3 & \\ 0.8 \leftarrow 4 & [2,3]\end{Bmatrix}$$

$$=0.2+0.4+0.3+0.5^{(3)}$$

$$1.4^{(3)}$$

Case 4: [Take all the 4 values]

$l=4$ $[1,4]$

1 2 3 4

A B C D

0.1 0.2 0.4 0.3

12

$$0.1+0.2+0.4+0.3+\min \begin{cases} 1.4 \leftarrow 1 & [2,4] \\ 0.1+1.0 \leftarrow 2 & 1,[3,4] \\ 0.4+0.3 \leftarrow 3 & [1,2],4 \\ 1.1 \leftarrow 4 & [1,3] \end{cases}$$

$= 1.0 + 0.7$

$= 1.7^{(3)}$

The above table can be displayed as 2 different tables

Main table (C)

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.4
5					0

Root table (R)

	0	1	2	3	4
1	0	1	2	3	3
2		0	2	3	3
3			0	3	3
4				0	4
5					0

Algorithm: OptimalBST (p[1...n])

```
for i ← 1 to n do
  C[i, i-1] ← 0   // First diagonal = 0
  C[i, i] ← p[i]  // Before the first diagonal (ie) Prob. value
  R[i, i] ← i     // Root table → C(1,1) = 1
C[n+1, n] ← 0     // C[5,4] = 0
for d ← 1 to n-1 do   // diagonal count  n=4, d=n-1=3
 for i ← 1 to n-d do  // row i = 1 to n-d => 4-3=1
                                            4-2=2
                                            4-1=3
   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] ← kmin
   sum ← p[i]
```

for $s \leftarrow i+1$ to j do

sum $\leftarrow$ sum + $p[s]$

$c[i,j] \leftarrow$ minval + sum

return $C[1,n]$, R

$d=1$

$c[i,j] = c[i,k-1] + c[k+1,j] + p(s)$

k is i to j

$c[1,2]$

k value : i to j = 1 to 2 ; k = 1, 2

Take k=1

$p(s) = p(1) + p(2) = 0.1 + 0.2 = 0.3$

$c[1,2] = c[1,1-1] + c[1+1,2] + 0.3$

$= c[1,0] + c[2,2] + 0.3$

$= 0 + 0.2 + 0.3$

$= 0.5$

Take k=2

$c[1,2] = c[1,1] + c[3,2] + 0.3$

$= 0.1 + 0 + 0.3$

$= 0.4$

$\therefore$ $c[1,2]$ is minimum for $k=2$ [k = root node]

$c[2,3]$

k value = 2, 3

$p(s) = p(2) + p(3)$

$0.2 + 0.4 = 0.6$

Take k=2

$c[2,3] = c[2,1] + c[3,3] + 0.6$

$= 0 + 0.3 + 0.6$

$= 0.9$

Take $K=3$

$C[2,3] = C[2,2] + C[4,3] + 0.6$
$= 0.2 + 0 + 0.6$
$C[2,3] = 0.8$

$C[2,3]$ is minimum for $K=3$

$C[3,4]$

$K = 3, 4$

$P(S) = P(3) + P(4)$
$= 0.4 + 0.3$
$= 0.7$

Take $K=3$

$C[3,4] = C[3,2] + C[4,4] + P(S)$
$= 0 + 0.3 + 0.7$
$= 1$

Take $K=4$

$C[3,4] = C[3,3] + C[5,4] + P(S)$
$= 0.4 + 0 + 0.7$
$= 1.1$

$C[3,4]$ is minimum for $K=3$

$d=2$

$C[1,3]$

$K = 1, 2, 3$

$P(S) = P(1) + P(2) + P(3) = 0.1 + 0.2 + 0.4$
$= 0.7$

Take $K=1$

$C[1,3] = C[1,0] + C[2,3] + P(S)$
$= 0 + 0.8 + 0.7$
$= 1.5$

Take $K=2$

$C[1,3] = C[1,1] + C[3,3] + P(S)$
$= 0.1 + 0.4 + 0.7$
$= 1.2$

Take $k=3$

$C[1,3] = C[1,2] + C[4,3] + p(s)$

$= 0.4 + 0 + 0.7$

$= 1.2$

$\therefore$ $C[1,3]$ is minimum for $k=3$

$C[2,4]$

$k = 2, 3, 4$

$p(s) = p(2) + p(3) + p(4) = 0.2 + 0.4 + 0.3$

$= 0.9$

Take $k=2$

$C[2,4] = C[2,1] + C[3,4] + p(s)$

$= 0 + 1 + 0.9$

$= 1.9$

Take $k=3$

$C[2,4] = C[2,2] + C[4,4] + p(s)$

$= 0.2 + 0.3 + 0.9$

$= 1.4$

Take $k=4$

$C[2,4] = C[2,3] + C[5,4] + p(s)$

$= 0.8 + 0 + 0.9$

$= 1.7$

$C[2,4]$ is minimum for $k=3$

$d=3$

$C[1,4]$:

$k = 1, 2, 3, 4$

$p(s) = 0.1 + 0.2 + 0.4 + 0.3 = 1$

$p(s) = 1$

Take $k=1$

$C[1,4] = C[1,0] + C[2,4] + p(s)$

$= 0 + 1.4 + 1$

$= 2.4$

Take $k=2$

$$C[1,4] = C[1,1] + C[3,4] + P(s)$$
$$= 0.1 + 1 + 1$$
$$= 2.1$$

Take $k=3$

$$C[1,4] = C[1,2] + C[4,4] + P(s)$$
$$= 0.4 + 0.3 + 1$$
$$= 1.7$$

$C[1,4]$ is minimum for $k=3$

Analysis:

Basic operation :- Comparison

Space efficiency is $O(n^2)$ quadratic

Time efficiency is $O(n^3)$ cubic

Knapsack problem:

Given n items of known weights $w_1, \dots w_n$ and values $v_1, \dots v_n$ and knapsack capacity W find the most valuable subset of the items that fit into the knapsack.

W=5

Item	Weight w	Value/Profit v/P
1	2	12
2	1	10
3	3	20
4	2	15

Find the optimal solution for knapsack problem

Soln:

n=4 ⇒ ~~5×5 matrix~~

P matrix j→0

n+1 omit

i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22 =	30	32
n 4	0	10	15	25	30	37

Steps:

I. define the initial condition row $i=0$ to n,
column $j=0$ to $n+1$

$P(0,j)=0$, $j\geq 0$

$P(i,0)=0$, $i\geq 0$.

II. Calculate $P(i,j)$

$P(i,j)=\max\{P(i-1,j), P_i+P(i-1, j-w_i)\}$,

if $j-w_i\geq 0$.

$P(i,j)=P(i-1,j)$, if $j-w_i<0$.

Row: $i=1$. $w_1=2$, $P_1=12$.

$P(1,1)$:

$P(1,1)\Rightarrow j-w_i = j-w_1 = 1-2=-1<0$.

$\therefore P(1,1)=P(0,1)=0$.

$P(1,2)$:

$P(1,2)\Rightarrow j-w_i = 2-2=0\geq 0$

$P(1,2)=\max\{P(0,2), P_1+P(0,0)\}$

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

$=12$.

$P(1,3)$:

$P(1,3)\Rightarrow j-w_i = 3-2=1\geq 0$

$P(1,2)=\max\{P(0,2), P_1+P(0,1)\}$

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

$=12$.

$P(1,4)$:

$j-w_i = 4-2=2\geq 0$

$P(1,4)=\max\{P(0,4)+P_1+P(0,2)\}$

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

$= 12.$

$P(1,5)$:

$j - w_i = 5 - 2 = 3 \geq 0$

$P(i,j) = \max\{P(0,5), P_1 + P_0(0,3)\}$

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

$= 12.$

<u>Row, $i=2$:</u> $W_2 = 1$, $P_2 = 10$.

$P(2,1)$:

$j - w_i = 1 - 1 = 0$

$P(2,1) = \cancel{P(1,1) = 0}$

$= \max\{P(1,1), P_2 + P(1,0)\}$

$= \max\{0, 10 + 0\}$

$= 10.$

$P(2,2)$:

$j - w_i = 2 - 1 = 1 \geq 0$

$P(2,2) = \max\{P(1,2), P_2 + P(1,1)\}$

$= \max\{12, 10 + 0\}$

$= 12$

$P(2,3)$:

$j - w_i = 3 - 1 = 2 \geq 0$

$P(2,3) = \max\{P(1,3), P_2 + P(1,2)\}$

$= \max\{12, 10 + 12\}$

$= 22$

$P(2,4)$:

$j - w_i = 4 - 2 = 2 \geq 0$.

$P(2,4) = \max\{P(1,4), P_2 + P(1,2)\}$

$= \max\{12, 10 + 12\}$

$= 22$.

$P(2,5)$:

$j - w_i = 5 - 2 = 3 \geq 0$.

$P(2,5) = \max\{P(1,5), P_2 + P(1,3)\}$

$= \max\{12, 10 + 12\}$

$= 22$

Row $i = 3$: $w_3 = 3$, $P_3 = 20$.

$P(3,1)$: $j - w_i = 1 - 3 = -2 < 0$.

$P(3,1) = P(2,1) = 10$.

$P(3,2)$: $j - w_i = 2 - 3 = -1 < 0$.

$P(3,2) = P(2,2) = 12$

$P(3,3)$: $j - w_i = 3 - 3 = 0$.

$P(3,3) = \max\{P(2,3), P_3 + P(2,0)\}$

$= \max\{22, 20 + 0\}$

$= 22$

$P(3,4)$: $j - w_i = 4 - 3 = 1$.

$P(3,4) = \max\{P(2,4), P_3 + P(2,1)\}$

$= 30$.

Step: 1 :

compare P(4,5) and P(3,5)

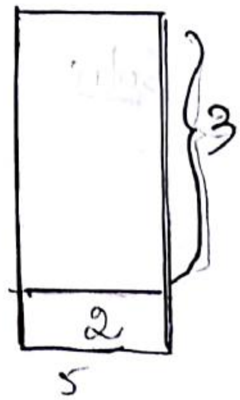
$$P(4,5) \neq P(3,5)$$

Not equal means item 4 is included in the solution set, then delete 4th row and 4th column.

$$\therefore S = \{4\}$$

$$W_1 = 5-2 = 3.$$

We can still add only 3.



Step: 2 :

compare P(3,3) with P(2,3)

$$P(3,3) = P(2,3)$$

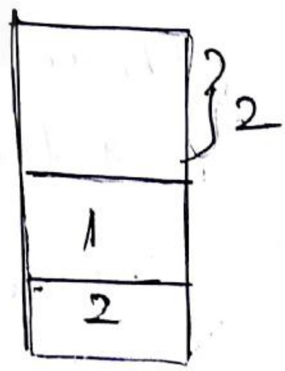
Equal means item 3 is not included in solution set then delete only row

Step: 3 :

compare (2,3) & (1,3)

$$(2,3) \neq (1,3)$$

Item 2 can be included in the solution set.



$$S = \{4, 2\}$$

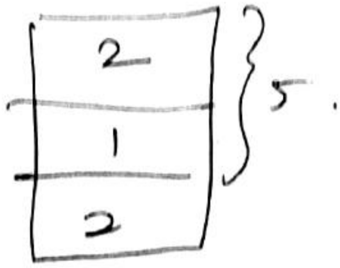
$$W_1 = 5-(2+1) = 2.$$

Step:4:

compare (1,2) & (0,2)

$P(1,2) \neq P(0,2)$.

∴ add item 1 to the subset.



$S = \{4,2,1\}$

$W_1 = 5-(2+1+2) = 0$.

Soln:

Feasible subset $S = \{1,2,4\}$

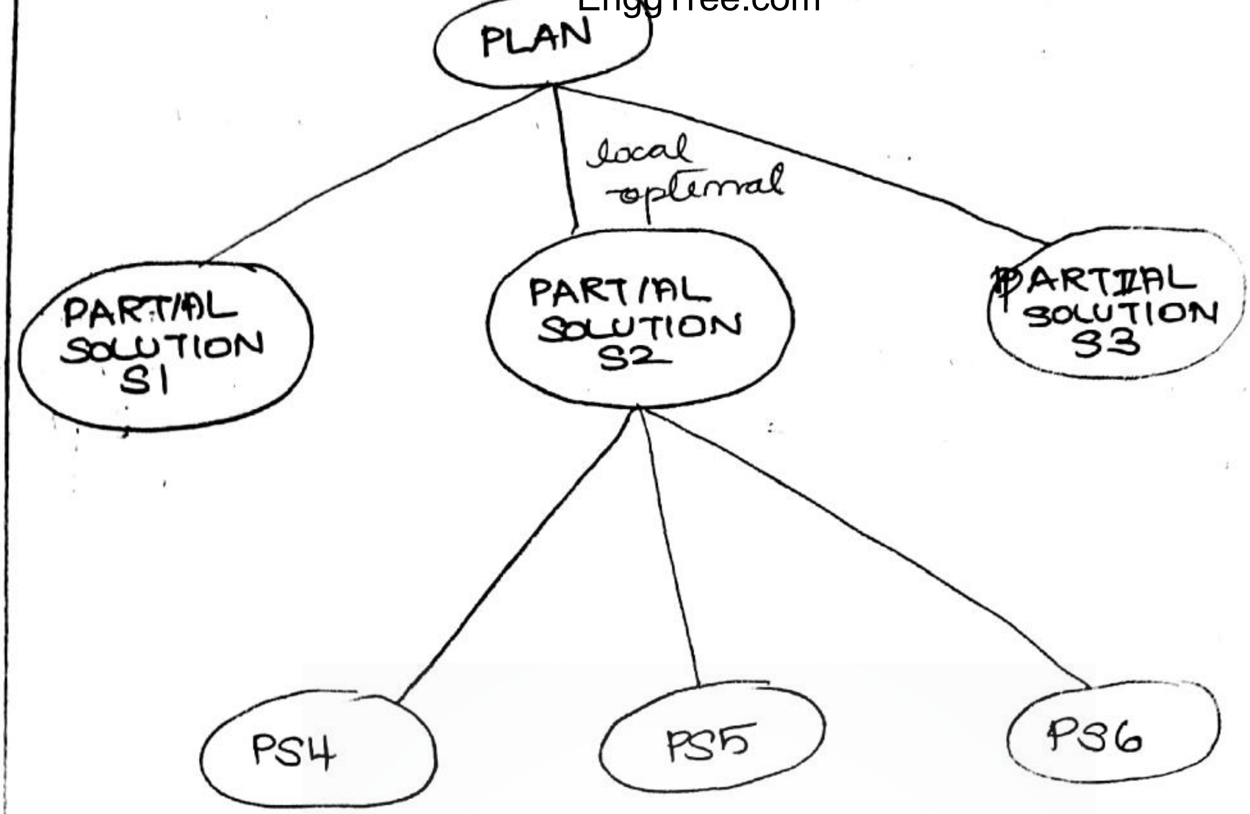
Optimal value / solution $= 12+10+15$

$= 37$, (last value in the matrix)

GREEDY TECHNIQUE

It suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete to the problem is reached. On each step, the choice made must be

* feasible (ie) it has to satisfy the problem's constraints
* locally optimal (ie) it has to be the best local choice among all feasible choices available on that step.
* irrevocable (ie) once made, it cannot be changed on subsequent steps of the algorithm.



1. PRIMS ALGORITHM:

To construct minimum spanning tree

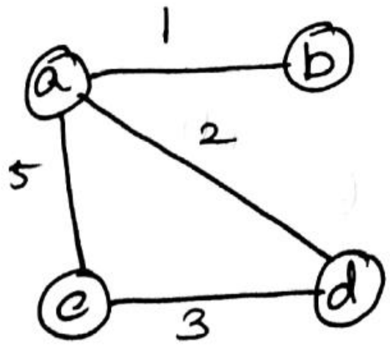
SPANNING TREE:

A spanning tree of a weighted connected graph is its ~~spanning tree~~ connected acyclic sub-graph that contains all the vertices of a graph.

MINIMUM SPANNING TREE:

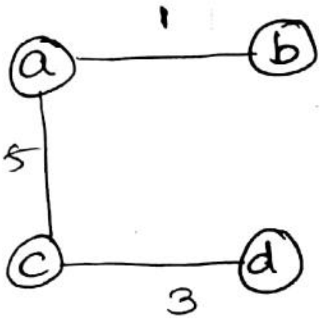
A minimum spanning tree of a weighted connected graph is the spanning tree of the smallest weight where the weight of a tree is defined as a sum of the weights of all its edges.

EXAMPLE:



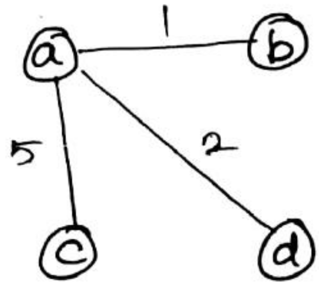
SPANNING TREES POSSIBLE:

ST1:



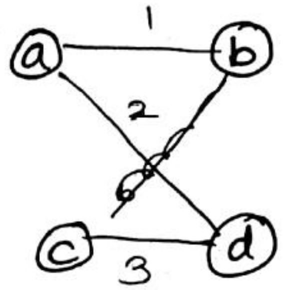
W(ST1) = 9

ST2:



W(ST2) = 8

ST3:

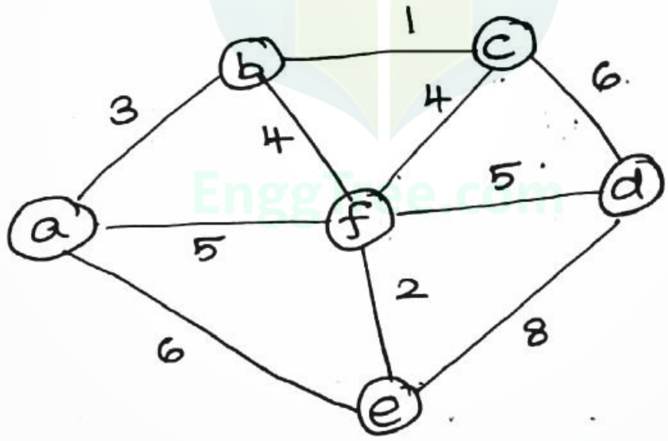


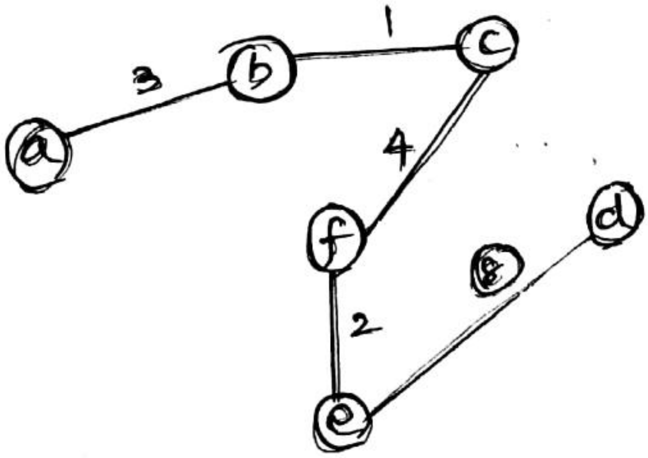
W(ST3) = 6

MINIMUM SPANNING TREE:

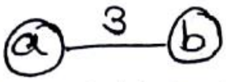
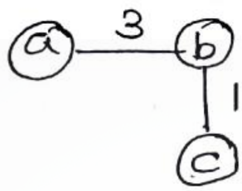
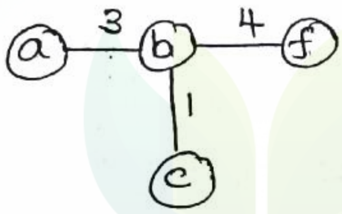
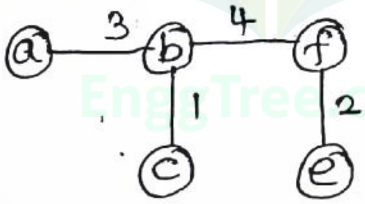
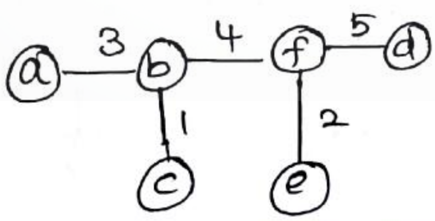
ST3 is the minimum spanning tree which has the least weight W(ST3) = 6.

1. construct minimum spanning tree using Prims Algorithm.





b(a,3) ⇒ to 'b' from 'a' distance 3.

TREE VERTICES	MINIMUM SPANNING TREE	REMAINING VERTICES.
a(–,–)	a	b(a,3) min, e(a,6) f(a,5), c(–,∞), d(–,∞)
b(a,3)	 use neighbours for both a and b & take min	c(b,1) min, f(b,4), e(a,6) d(–,∞)
c(b,1)		f(b,4), d(c,6), e(a,6)
f(b,4)		d(f,5), e(f,2)
e(f,2)		d(f,5)
d(f,5)		–

Weight of minimum spanning tree:

W = 3+4+5+1+2

$\boxed{W = 15.}$

ALGORITHM:

prim(G)

$V_T = \{v_0\}$

$E_T = \phi$ (null)

for $i = 1$ to $|V| - 1$ do // $|V|$ ~~ab~~ means no. of vertex

find the minimum weighted edge,

$e^* = (v^*, u^*)$

among all the edges (v, u) → union operation.

$V_T = V_T \cup \{u^*\}$ // $V_T \cup \{u^*\}$

$E_T = E_T \cup \{e^*\}$

return E_T → spanning tree.

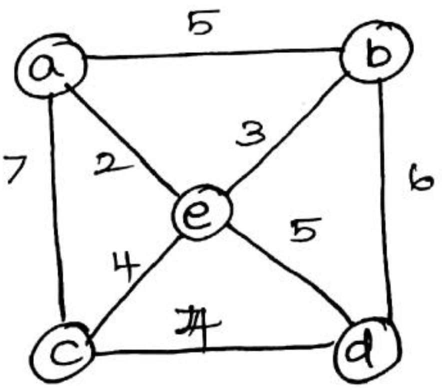
ANALYSIS:

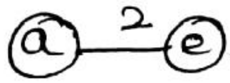
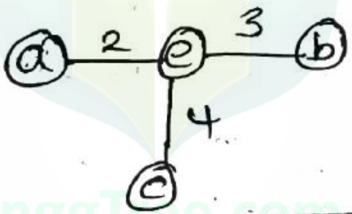
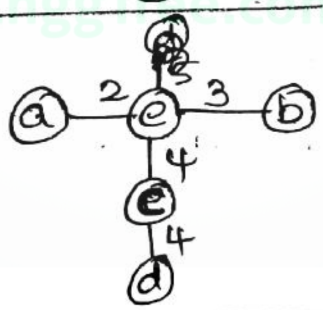
If a graph is represented by weight matrix and the priority queue, time efficiency is $O(|V|^2)$

If a graph is represented by linked list and the priority queue, time efficiency is,

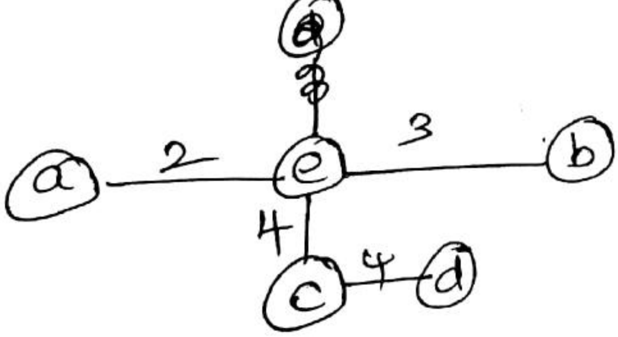
$T(n) = O(|E| \cdot \log |V|)$

2. Apply prims Algorithm to the following graph,



TREE VERTICES	MINIMUM SPANNING TREE	REMAINING VERTICES
a(-,-)	a	b(a,5), c(a,7) d(-,∞), e(a,2)
e(a,2)		b(e,3) c(e,4) d(e,5)
b(e,3)		c(e,4) d(e,5)
c(e,4)		d(c,4)
d(e,5)		—

Minimum Spanning Tree:

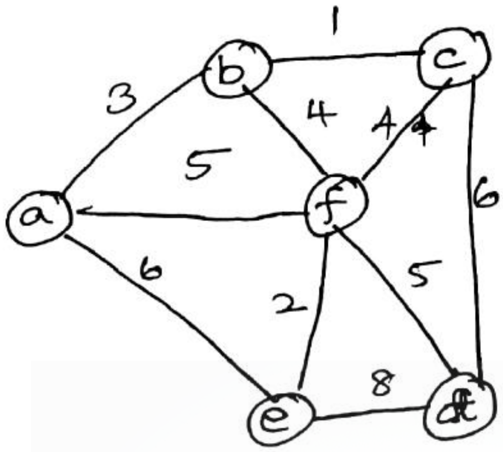


Weight of MSP = 2+4+3+4 = 13

2) KRUSKAL'S ALGORITHM:

to find minimum spanning tree.

Apply Kruskals Algorithm for the following graph to find the minimum spanning tree.



Step:1: Write the edges with weight.

ab	bc	cd	de	ea	af	bf	cf	df	ef
3	1	6	8	6	5	4	4	5	2

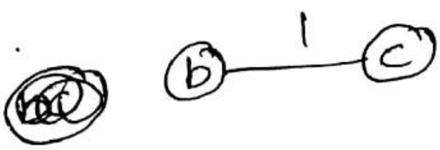
Step:2: Sort the edges in asscending order of weight.

bc	ef	ab	bf	cf	af	df	cd	ea	de
1	2	3	4	4	5	5	6	6	8

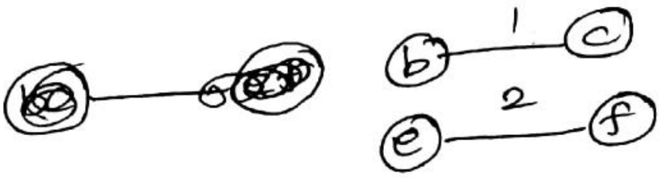
Step:3:

select the edges and add is the spanning tree that it does not form a cycle.

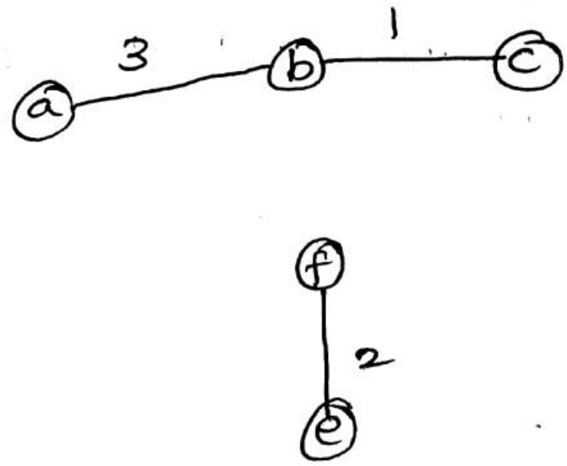
select bc



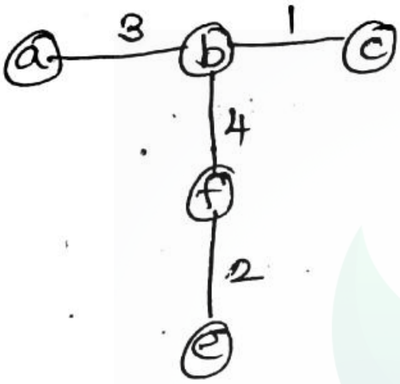
select ef



select ab



select bf



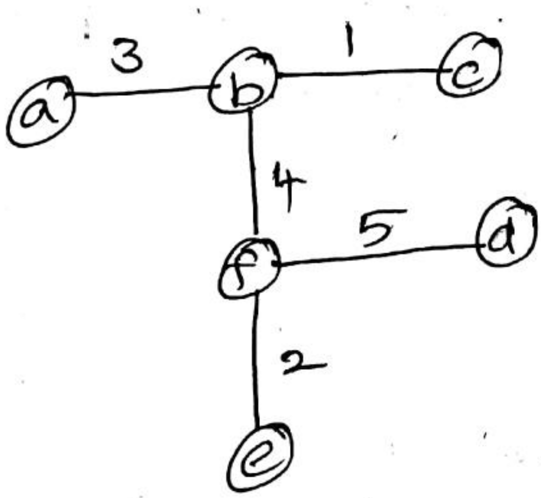
select cf.

cannot be added. It forms cycle.

select af

cannot be added. It forms cycle.

select df.



select cd.

cd forms a cycle, We cannot add cd.

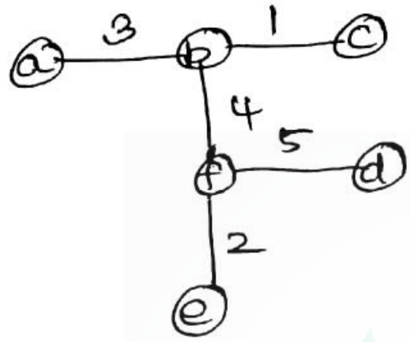
select ae.

It forms cycle. We cannot add. ae.

select de.

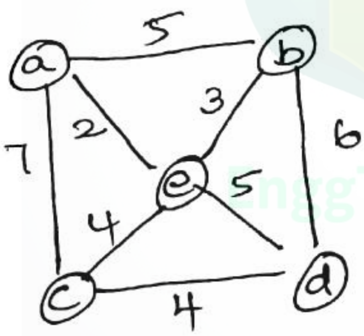
It forms cycle.

Minimium spanning Tree.



Cost = 3+1+4+5+2 = 15.

2)



Step:1:

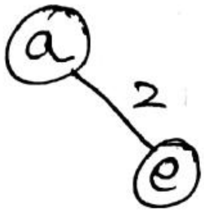
ab	bd	dc	ca	ae	be	de	ce
5	6	4	7	2	3	5	4

Step:2:

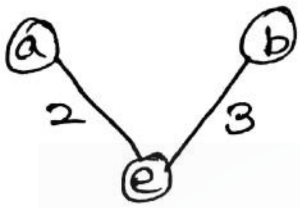
ae	be	ce	dc	ab	de	bd	ca
2	3	4	4	5	5	6	7

Step: 3:

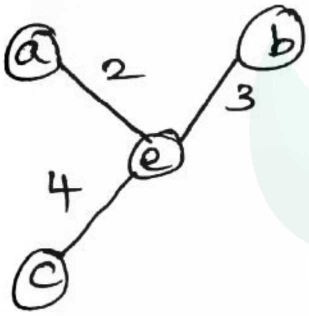
① select ae.



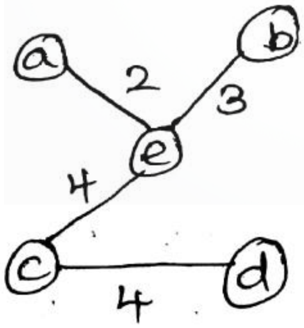
② select be



③ select ce



④ select dc



⑤ select ab – cycle

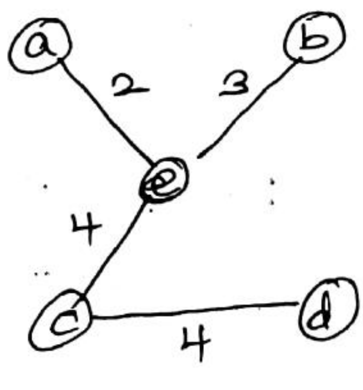
⑥ select de – cycle

⑦ select bd – cycle

⑧ select ca = cycle.

Weight = 2+3+4+4=13.

Minimum Spanning tree:



ALGORITHM:

Kruskal(G)

sort edges E in increasing order of the edge weight.

$E_T = \phi$ no. of vertex

k=0, encounter=0 → total no. of vertices.

while encounter < |V|-1

k=k+1

if $E_T \cup \{e_{ik}\}$ is acyclic

$E_T = E_T \cup e_{ik}$

encounter = encounter+1;

return E_T

Analysis:

The time efficiency of Kruskal algorithm is,

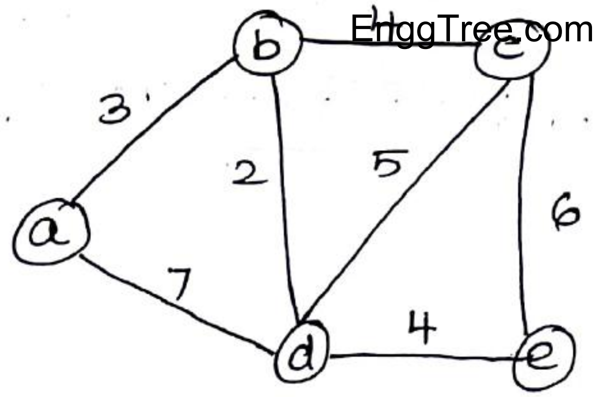
$t(n) = O(|E| \cdot \log|E|)$

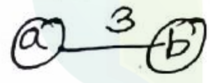
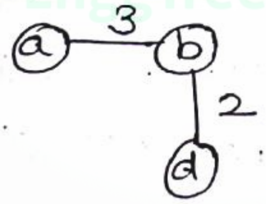
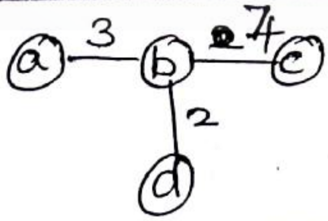
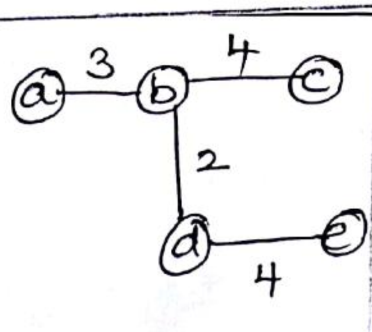
3. DIJKSTRA'S ALGORITHM:

single source shortest path problem

For a given vertex called source in a weighted connected graph, find the shortest path to all its other vertices.

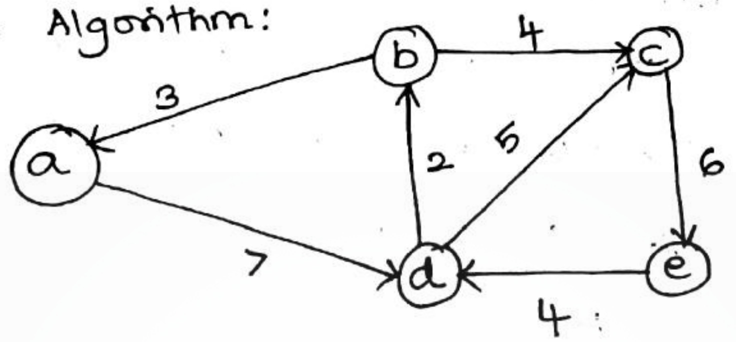
1. Apply dijkstras Algorithm to find the shortest path from source node A.



TREE VERTICES	SHORTEST PATH	VERTICES.
a(−,0)	a	b(a,3) d(a,7) c(−,∞) e(−,∞)
b(a,3)		c(b,7) d(b,5) e(−,∞)
d(b,5)		c(b,7) e(d,9)
c(b,7)		e(d,9)
e(d,9)		—

Shortest path	distance
a→b	3
a→c	7
a→d	5
a→e	9

Apply Dijstra's Algorithm:



S.P	DIST
a→b	9
a→c	12
a→d	7
a→e	18

TREE VERTICES	SHORTEST PATH	VERTICES.
a(−,0)	a	b(−,∞), d(a,7), c(−,∞), e(−,∞)
d(a,7)	$a \xrightarrow{7} d$	b(d,9), c(d,12) e(−,∞)
b(d,9)	$a \xrightarrow{7} d \xrightarrow{2} b$	c(d,12) e(c,18)
c(d,12)	$a \xrightarrow{7} d \xrightarrow{2} b$, $d \xrightarrow{5} c$	e(c,18)
e(c,18)	$a \xrightarrow{7} d \xrightarrow{2} b$, $d \xrightarrow{5} c \xrightarrow{6} e$	—

4. Huffmann Trees (Semester purpose)

Text Replaced by code word.

Text containing n character assigning to each of the text characters. some sequence of bits called code word.

eg. AB → 0110 111

Fixed-length encoding:

A bit string or code word of same length m is assigned to each character of text.

eg: A B → 010 111

m = 3.

Encoding:

Text → codeword.

Decoding:

code word → text.

Variable length encoding:

Assigning code word of different length to different characters of text.

eg. A → 0110 B → 111

m = 4 m = 3.

Condition:

No code word is a prefix of a code word of another character.

eg. A B → 11 × 110

1. consider the five character alphabet {A, B, C, D, –} with the following occurance of probabilities,

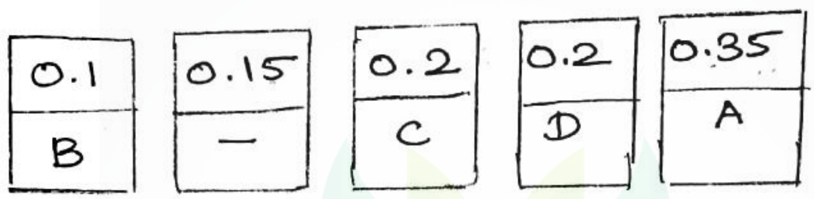
Character	A	B	C	D	–
Probability	0.35	0.1	0.2	0.2	0.15

Encode DAD & Decode 10011011011101

Huffman tree:

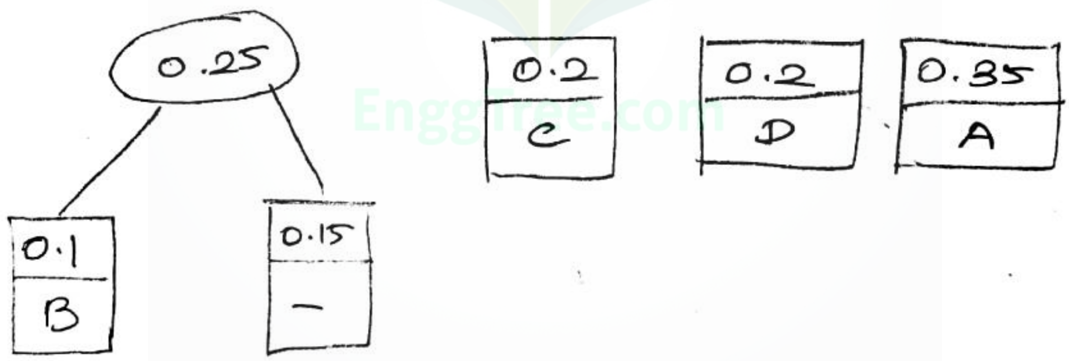
Step:1:

Sort the characters is ascending order of probability.



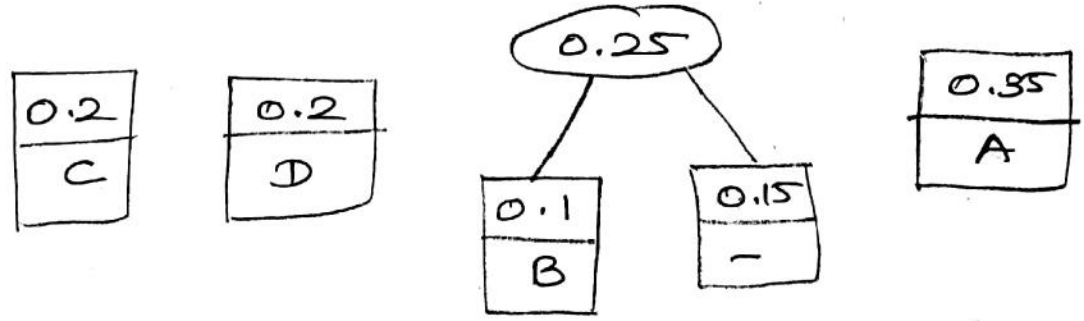
Step:2:

Combine the first 2 minimum characters.



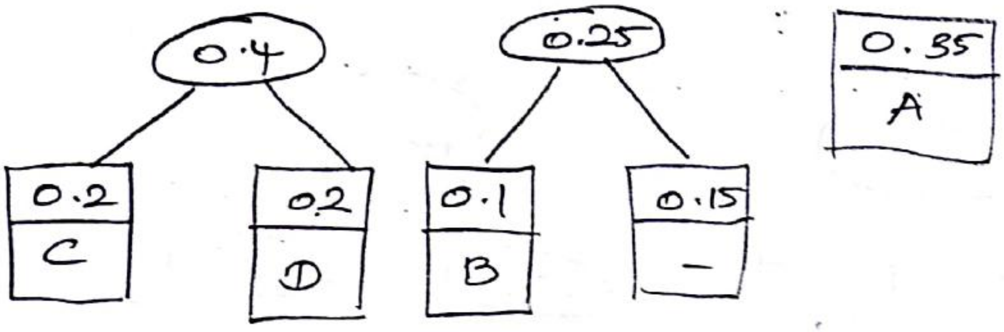
Step:3:

Again sort.

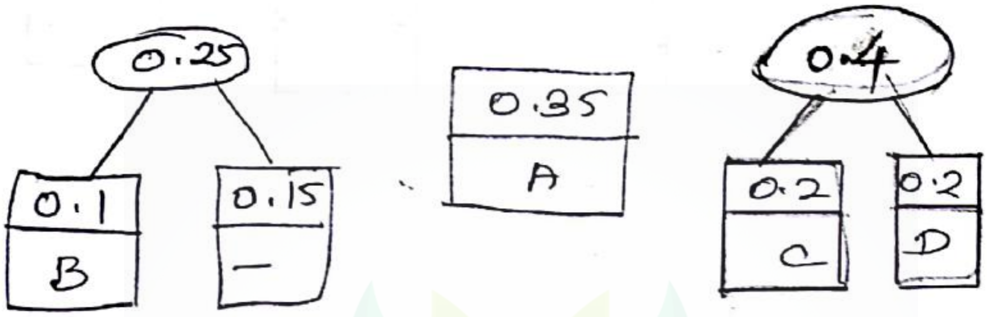


Step: 4:

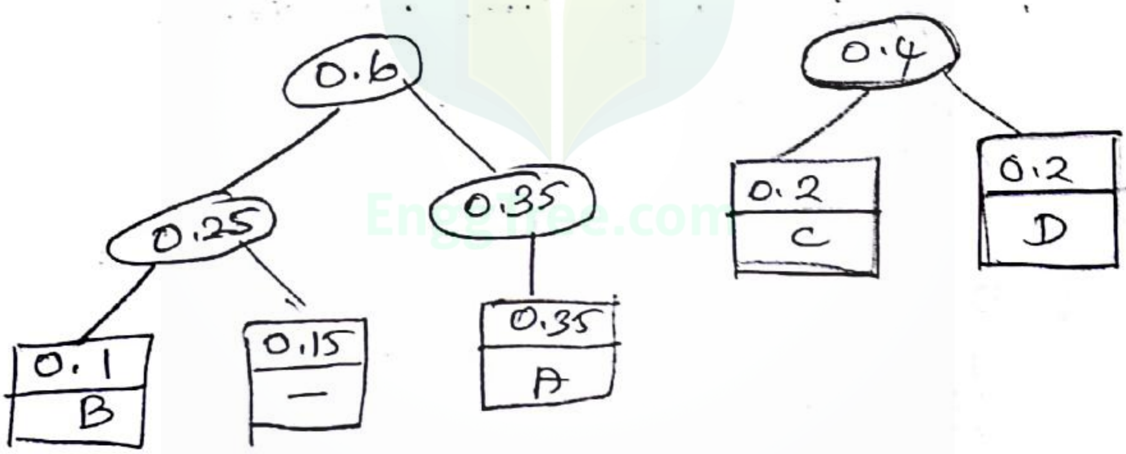
combine.



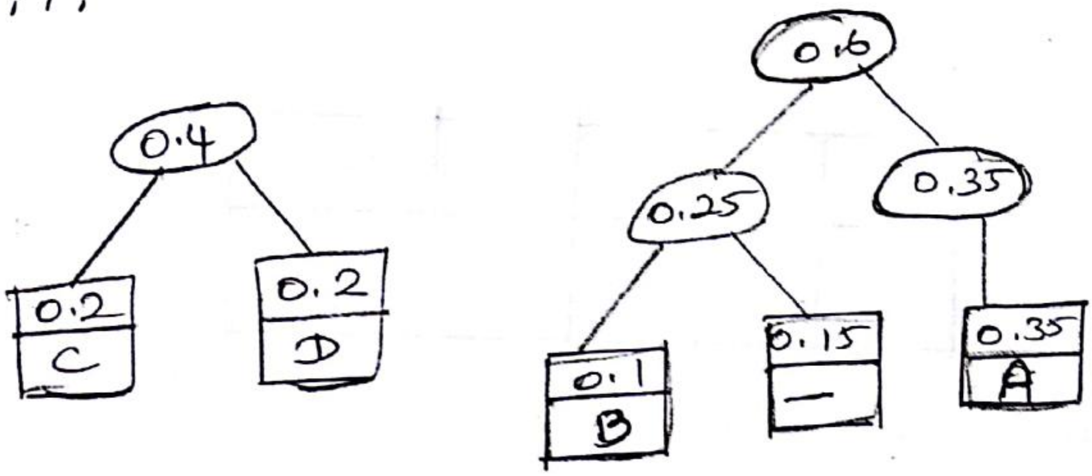
Step: 5:



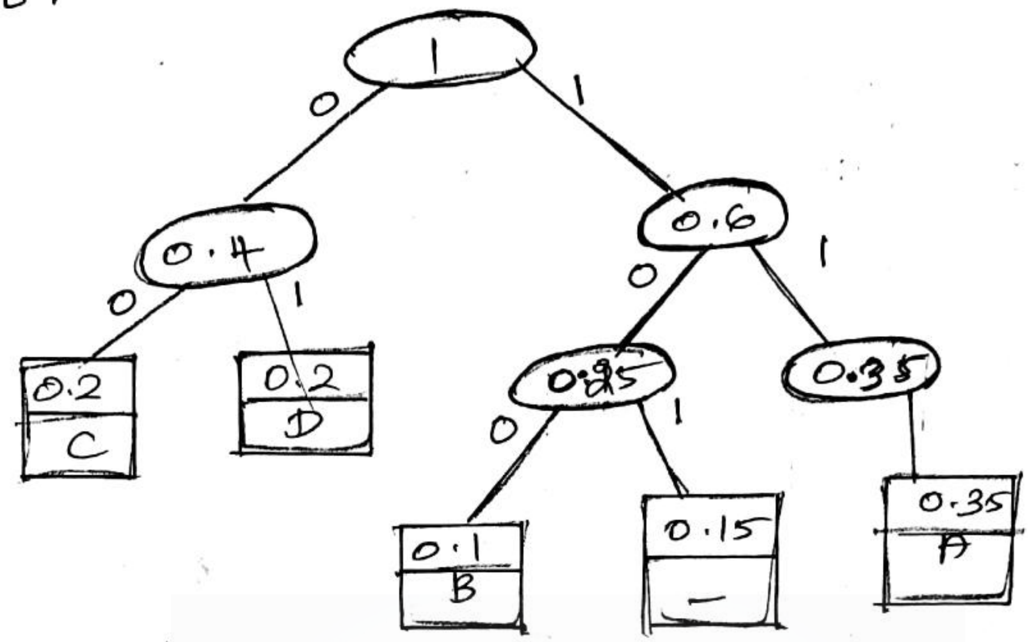
Step: 6:



Step: 7:



Step: 8 :



Code word

A → 11

B → 100

C → 00

D → 01

ø — → 101

see from root to leaf node & compute code word.

Encoding :

DAD — 011101

Decoding :- 100110110111101

BAD_AD

2. Construct a Huffman tree.

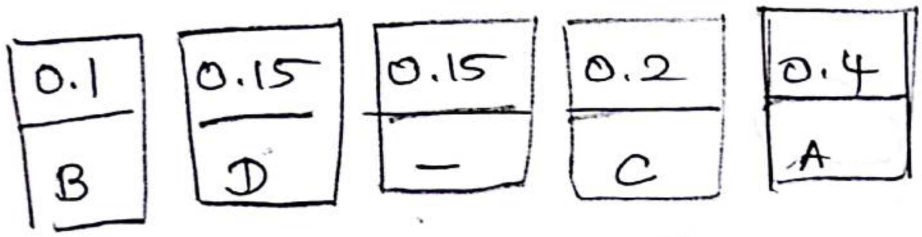
A	B	C	D	—
0.4	0.1	0.2	0.15	0.15

Encode ABACABAD

Decode 100010111001010

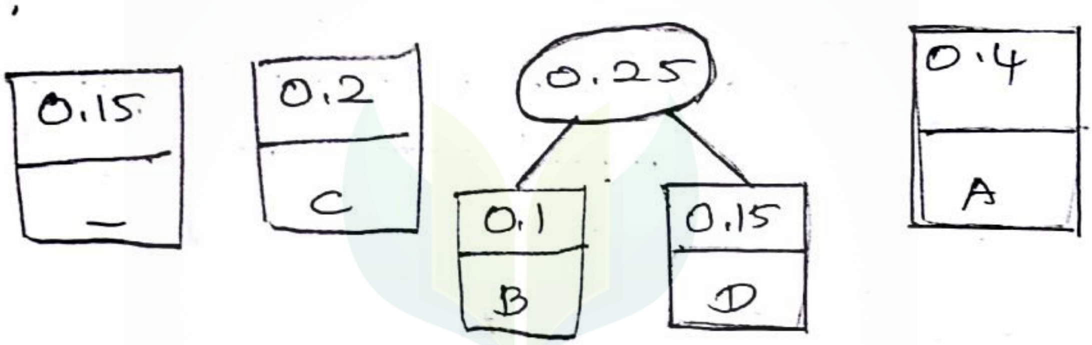
40

Step: 1:

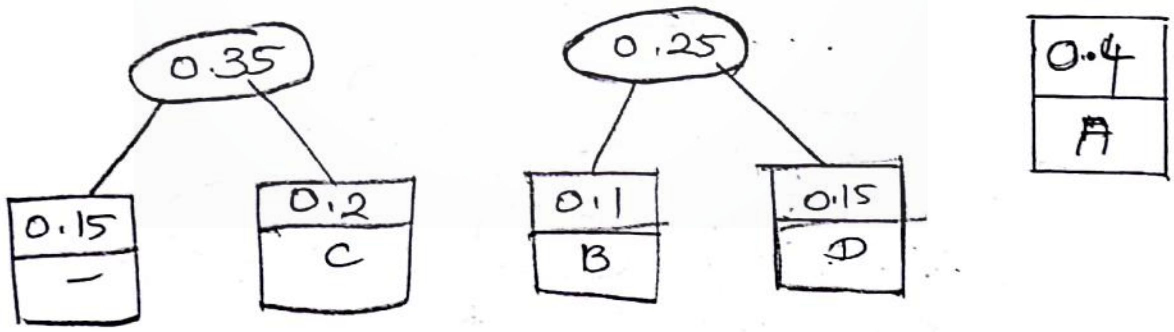


Step: 2:

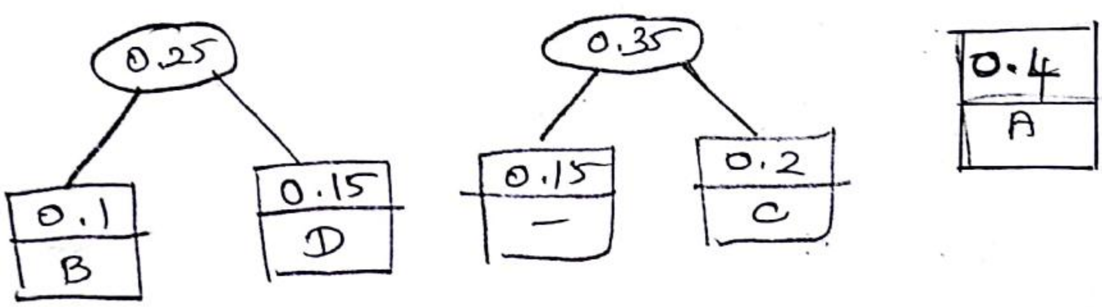
Step: 3:



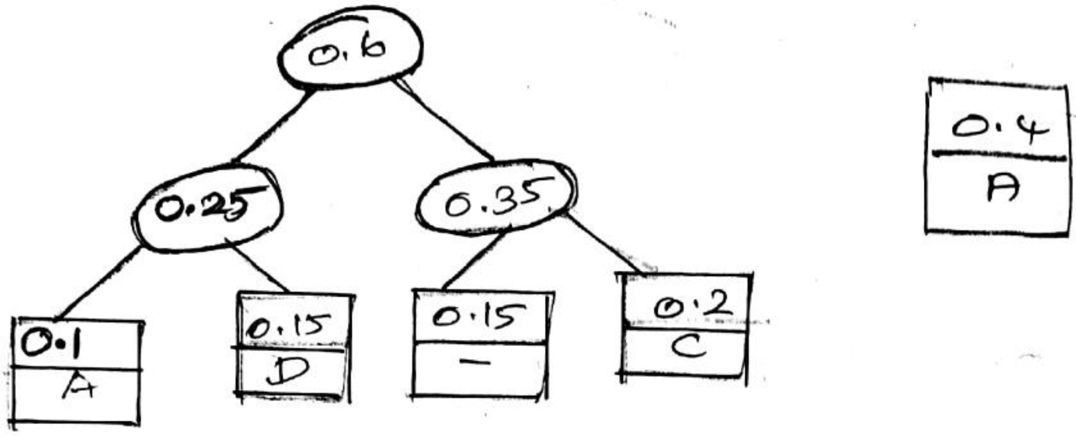
Step: 4:



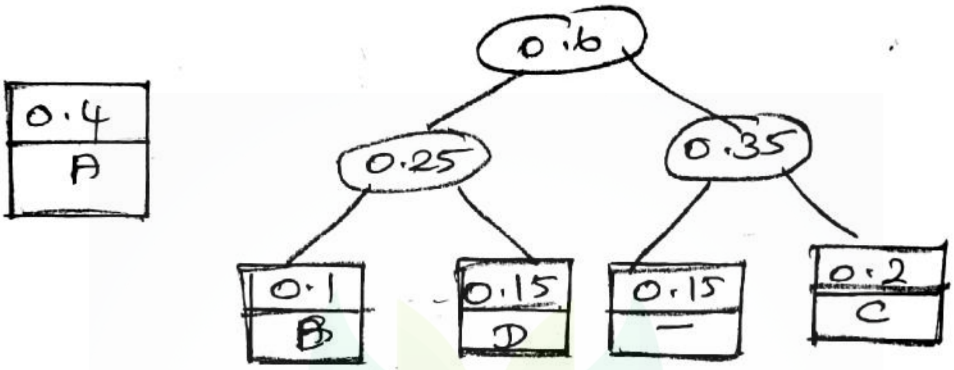
Step: 5:



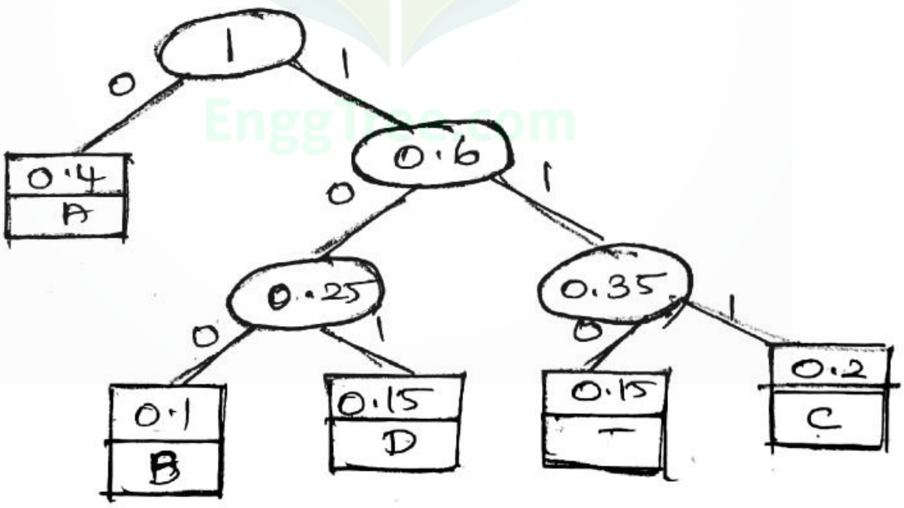
Step: 6:



Step: 7:



Step: 8:



A → 0

B → 100

C → 111

D → 101

— → 110

Encoding:-

ABACABAD

0100011101000101.

Decoding:- 1000101111001010

BAD—ADA

6/2/18 6/2/18 7/2/18

42

UNIT IV ITERATIVE IMPROVEMENT

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

Example 1 Simplex method

Solve the following LPP using the simplex method.
Linear Programming Problems

Maximize $z = 12x_1 + 16x_2$ [objective function]

Subject to $10x_1 + 20x_2 \leq 120$ [Constraint]

$8x_1 + 8x_2 \leq 80$ [Constraint]

x_1 & $x_2 \geq 0$

Solution

Max $z = 12x_1 + 16x_2$

Add slag variables to the constraints

Max $z = 12x_1 + 16x_2 + 0s_1 + 0s_2$

subject to $10x_1 + 20x_2 + s_1 = 120$ —— (1)

$8x_1 + 8x_2 + s_2 = 80$ —— (2)

x_1, x_2, s_1 & $s_2 \geq 0$

Initial simplex Table

CB_i	c_j	12	16	0	0	Solution	Ratio
	Basic variable	x_1	x_2	s_1	s_2		
0	s_1	10	20	1	0	120	$120/20 = 6$
0	s_2	8	8	0	1	80	$80/8 = 10$
	z_j	0	0	0	0	0	
	$c_j - z_j$	12	16	0	0		

$C_j \rightarrow$ Co-efficient of objective function

Max $z = \underline{12} x_1 + \underline{16} x_2 + \underline{0} S_1 + \underline{0} S_2$

Basic Variables $\rightarrow x_1, x_2, S_1$ & S_2

$CB_i \rightarrow$ Co-efficient of basic Variable (ie slag Variable added to obj fn). ∴ 0 for S_1 and 0 for S_2

Formula to find $\boxed{Z_j = \sum_{i=1}^{2} (CB_i)(a_{ij})}$

<u>Simple way to find Z_j</u>

$(CB_1 * x_1) + (CB_2 * S_2)$, $CB_i * x_2$, $CB_i * S_1$, $CB_i \times S_2$, $CB_i *$ Soln

$(0 \times 10) + (0 \times 8) \Rightarrow 0 + 0 = 0$

$(0 \times 20) + (0 \times 8) \Rightarrow 0 + 0 = 0$

$(0 \times 1) + (0 \times 0) \Rightarrow 0 + 0 = 0$

$(0 \times 0) + (0 \times 1) \Rightarrow 0 + 0 = 0$

$(0 \times 120) + (0 \times 80) \Rightarrow 0 + 0 = 0$

Write the above Z_j values in the Z_j row of simplex table

Now to compare C_j and Z_j using formula $\underline{C_j - Z_j}$

$12 - 0 = 12$, $16 - 0 = 16$, $0 - 0 = 0$, $0 - 0 = 0$

Update the above $C_j - Z_j$ Values in the table

<u>Optimality Condition:</u>

<u>For maximizing:</u>

all $C_j - Z_j \leq 0$

<u>For minimizing</u>

all $C_j - Z_j \geq 0$

Our objective is to maximize, for maximizing $C_j - Z_j$ should be negative or zero. But from our simplex table, $C_j - Z_j$ is positive

To reach Optimality, proceed further.

1. Select the maximum value in $C_j - Z_j$
   ∴ 16 is the max value
   Hence 20 8 is the key column.

2. Find the ratio between Solution and key Column
   $120/20 = \boxed{6}$
   $80/8 = \boxed{10}$ & fill the values in the Ratio Column

3. To find the key row, take the minimum value of Ratio, ∴ 6 is the min ratio.
   Hence 10 20 1 0 120 is the key row

4. The intersection point of key row and key column is called key element.
   Hence 20 is the key element.

5. Here x_2 is the entering variable and S_1 is the leaving variable.

6. Proceed with the 1st iteration to achieve optimality.

Iteration 1

CB_i	C_j	12	16	0	0	Solution	Ratio
	B.V	x_1	x_2	S_1	S_2		
16	x_2	1/2	1	1/20	0	6	$\frac{6}{1/2} = 12$
0	S_2	4	0	-2/5	1	32	$\frac{32}{4} = 8$
	Z_j	8	16	4/5	0		
	$C_j - Z_j$	4	0	-4/5	0		

In the previous table we found, S_1 is the leaving variable and x_1 is the entering variable. So in Iteration 1 table, instead of S_1 write x_2

To find the new value, divide all the old values by the key element in the 1st row

$10/20 = 1/2$

$20/20 = 1$

$1/20 = 1/20$

$0/20 = 0$

$120/20 = 6$

update the new values in i1 table

To find the new values for S_2, apply a simple formula

New Value = Old Value − $\dfrac{\text{Corr. key column value} * \text{Corr. key row value}}{\text{key element}}$

$$8 - \frac{8 * 10}{20} \Rightarrow 8 - \frac{80}{20} \Rightarrow 8 - 4 \Rightarrow 4$$

$$8 - \frac{8 * 20}{20} \Rightarrow 8 - \frac{160}{20} \Rightarrow 8 - 8 \Rightarrow 0$$

$$0 - \frac{8 * 1}{20} \Rightarrow 0 - \frac{8}{20} \Rightarrow 0 - 2/5 = -2/5$$

$$1 - \frac{8 * 0}{20} \Rightarrow 1 - 0/20 \Rightarrow 1$$

$$80 - \frac{8 * 120}{20} \Rightarrow 80 - \frac{960}{20} \Rightarrow 80 - 48 \Rightarrow 32$$

update the above new values for 2^{nd} row in I_1 table.

To find Z_j

$(16 \times 1/2) + (0 \times 4) \Rightarrow 8 + 0 = 8$

$(16 \times 1) + (0 \times 0) \Rightarrow 16 + 0 = 16$

$(16 \times 1/20) + (0 \times -2/5) \Rightarrow 4/5 + 0 = 4/5$

$(16 \times 0) + (0 \times 1) \Rightarrow 0 + 0 = 0$, update in I_1

Then find $C_j - Z_j$

$12 - 8 = 4$

$16 - 16 = 0$

$0 - 4/5 = -4/5$

$0 - 0 = 0$, update in I_1

Optimality Condition for maximizing:

all $C_j - Z_j \leq 0$

But we have one positive value as 4, Hence Proceed with Iteration 2

1. Select the max value in $C_j - Z_j$

   4 is the max

   Hence $\begin{pmatrix} 1/2 \\ 4 \end{pmatrix}$ is the key column

2. Find the ratio between Solution & key column

   $\frac{6}{1/2} = \boxed{12}$

   $32/4 = \boxed{8}$, fill the Values in Ratio Column

3. Take the min value of ratio to find the key row.

   8 is the minimum,

   Hence (4 0 -2/5 1 32) is the key row

4. Intersection point of key row and key Column is 4 and is the key element.

5. Here, S_2 (the row Variable is leaving Variable) and x_1 (the column Variable is entering Variable

6. Proceed with 2nd iteration to achieve optimality.

Iteration 2

C_{Bi}	C_j / B.V	12 / x_1	16 / x_2	0 / S_1	0 / S_2	Solution
16	x_2	0	1	1/10	-1/8	2
12	x_1	1	0	-1/10	1/4	8
	Z_j	12	16	2/5	1	128
	$C_j - Z_j$	0	0	-2/5	-1	

In I_1, S_2 is leaving variable and x_1 is entering variable and hence in I_2 replace S_2 with x_1.

To find new value, divide all the old values by key element in the 2nd row

$4/4 = 1$

$0/4 = 0$

$\frac{-2/5}{4} = -2/20 = -1/10$

$1/4 = 1/4$

$32/4 = 8$, update the new values in I_2

To find the new values for x_2 (ie first row), apply the formula

$$n.V = o.V - \frac{C.K.C.V * C.K.r.V}{K.E}$$

$$1/2 - \left(\frac{1/2 * 4}{4}\right) = 0$$

$$1 - \left(\frac{1/2 * 0}{4}\right) = 1$$

$$1/20 - \left(\frac{1/2 * -2/5}{4}\right) = 1/10$$

$$0 - \left(\frac{1/2 * 1}{4}\right) = -1/8$$

$$6 - \left(\frac{1/2 * 32}{4}\right) = 2$$, update in I_2

To find Z_j

$(16\times0)+(12\times1) \Rightarrow 0+12=12$

$(16\times1)+(12\times0) \Rightarrow 16+0=16$

$(16\times\frac{1}{10})+(12\times-\frac{1}{10}) \Rightarrow \frac{2}{5}$.

$(16\times-\frac{1}{8})+(12\times\frac{1}{4}) \Rightarrow 1$

$(16\times2)+(12\times8) \Rightarrow 128$, update in I_2

$C_j - Z_j$

$12-12=0$

$16-16=0$

$0-\frac{2}{5}=-\frac{2}{5}$

$0-1=-1$, update in I_2

Condition of optimality for max is all $C_j - Z_j \le 0$

From I_2, we have all $C_j - Z_j \le 0$ $(0, 0, -\frac{2}{5}, -1)$

Take x_1 and x_2 values from solution column

$x_1 = 8$

$x_2 = 2$

$Z = 128$

For confirming the above values, substitute the values in objective function equation

$12x_1 + 16x_2 = Z$

$12(8)+16(2)=128$

$96+32=128$

$128=128$

LHS = RHS [Hence proved]

Example 2 Ford-Fulkerson Algorithm for maximum flow problem

Problem Important Problem of maximizing the flow of material through a transportation n/w [pipeline system, Commn. system, electrical distribution system, etc]

Given a graph which represents a flow network where every edge has a capacity. Also given two vertices source s and sink t in the graph. Find out the maximum possible flow from s to t with the following constraints.

a) Flow on an edge doesn't exceed the given capacity of the edge

b) In-flow is equal to out-flow for every vertex except s and t.

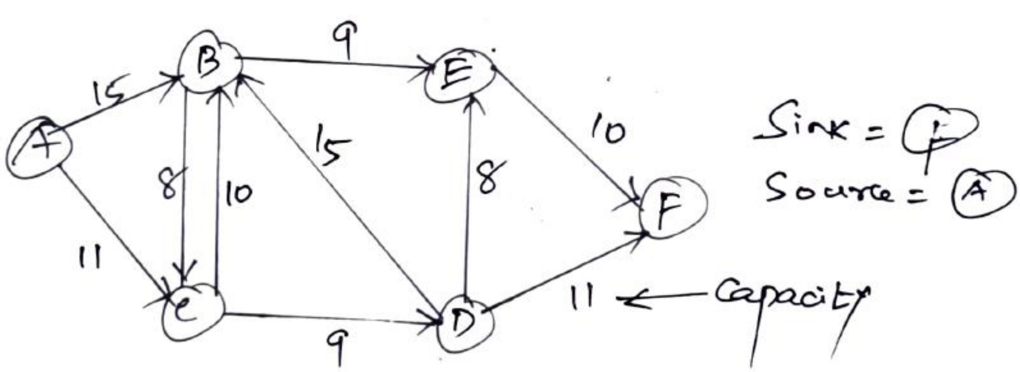
Algorithm

Ford-Fulkerson algorithm

1) Start with a initial flow as 0.

2) While there is an augmenting path from source to sink, Add this path flow to flow

3) Return flow



Terminologies:

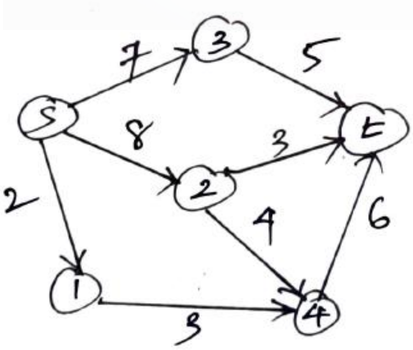
* Residual graph: It's a graph which indicates additional possible flow. If there is such path from source to sink, then there is a possibility to add flow.

* Residual capacity: It's original capacity of the edge minus flow.

* Minimal cut: Also known as bottle neck capacity, which decides maximum possible flow from source to sink through an augmented path.

* Augmented path: Augmenting path can be done in two ways.
  1) Non-full forward edges
  2) Non-empty backward edges.

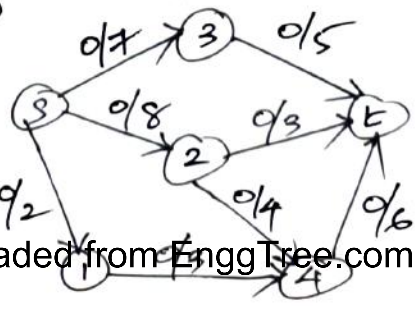
Apply Ford-Fulkerson method to find maximum flow and minimum cut.



Step 1

Initially Zero flow

Flow = 0



Step 2

Augumenting path

S → 1 → 4 → t

Edges	Total capacity	Current usage	Residual capacity
S → 1	2	0	2
1 → 4	3	0	3
4 → t	6	0	6

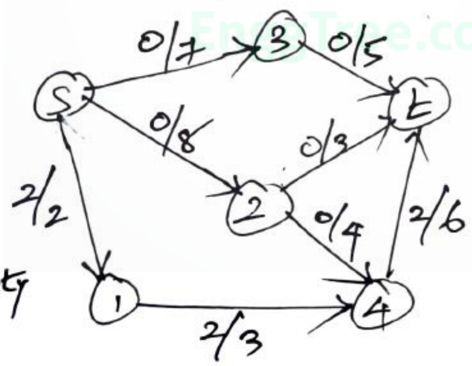
Find minimum residual capacity in the augumenting path.

Min(2, 3, 6) = 2

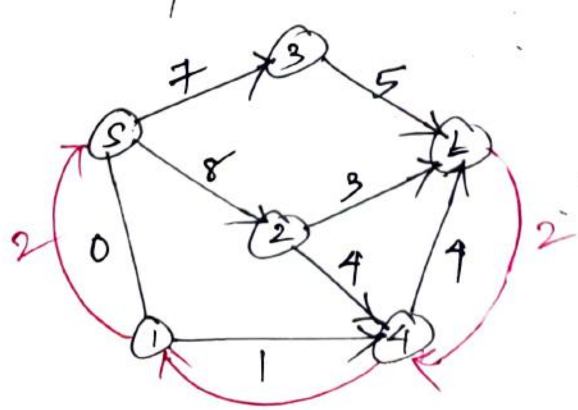
∴ $\boxed{flow_1 = 2}$

Augumenting graph:

Current usage / Total capacity



Residual graph:



Forward edge → Residual value

Backward edge → Current flow

Step 3 :

Augumenting path

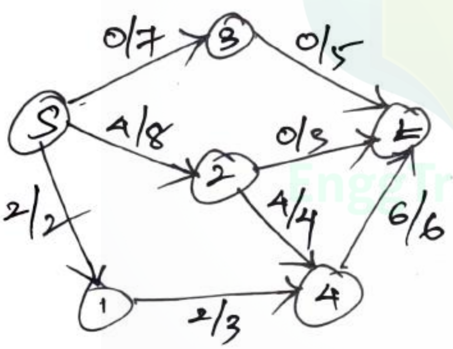
$S \rightarrow 2 \rightarrow 4 \rightarrow t$

Edges	T.capacity	C.used	Res.capacity
$s \rightarrow 2$	8	0	8
$2 \rightarrow 4$	4	0	4
$4 \rightarrow t$	6	2	4

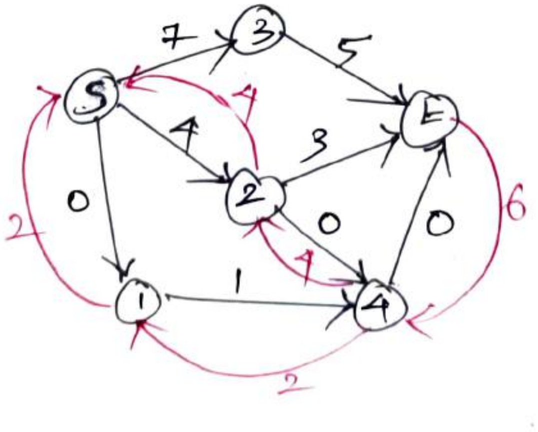
min(8, 4, 4)

$Flow_2 = 4$

Augumenting graph



Residual graph



Step 4:

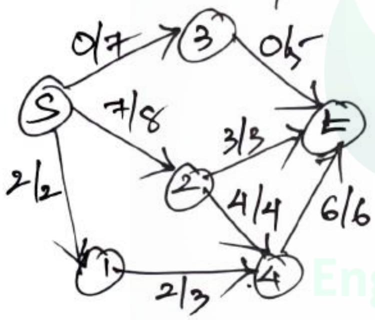
Augumented path:

Edges	T. capacity	C. used	Res. Capacity
S→2	8	4	4
2→t	3	0	3

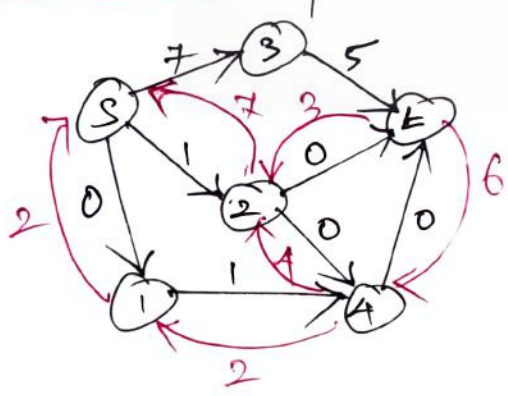
Min (4, 3)

Flow$_3$ = 3

Augumenting graph:



Residual graph:



Step 5

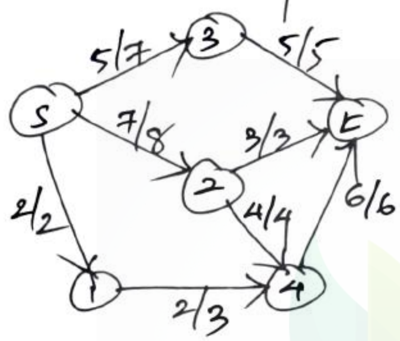
Augumented path

Edges	T. capacity	C. used	R. capacity
S→3	7	0	7
3→t	5	0	5

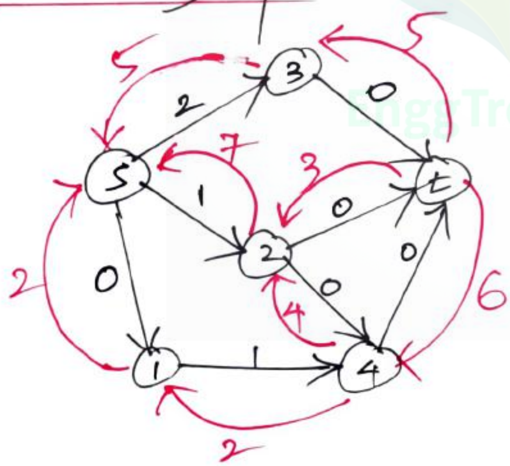
Min (7, 5)

$Flow_4 = 5$

Augumenting graph :



Residual graph



Maximum flow = Sum of flows in all augumented path

Max. flow = 2+4+3+5 = 14

Example 3 Stable Marriage Problem

Consider a set $Y = \{m_1, m_2, \dots m_n\}$ of n men and set $X = \{w_1, w_2, \dots w_n\}$ of n women.

=> Each man has a preference list of ordering the moment as potential marriage partners with no ties allowed.

=> Similarly, each woman has a preference list of men with no ties allowed.

=> Matching pair (m, w) whose members are selected from those 2 sets, based on the preference.

=> Ranking matrix n by n, a cell in row m and column w contains 2 rankings.

=> 1^{st} position in row m = m's preference list

=> 2^{nd} position in row m = w's preference list

Men's preference

	1st	2nd	3rd
Bob	Lea	Ann	Sue
Jim	Lea	Sue	Ann
Tom	Sue	Lea	Ann

Women's preference

	1st	2nd	3rd
Ann	Jim	Tom	Bob
Lea	Tom	Bob	Jim
Sue	Jim	Tom	Bob

Ranking Matrix

	Ann	Lea	Sue
Bob	(2,3)	(1,2)	(3,3)
Jim	(3,1)	(1,3)	(2,1)
Tom	(3,2)	(2,1)	(1,2)

Stable marriage algorithm

//Input : A set of n men and set of n women.
Preference list and Ranking list

//output : Stable marriage matching

Step 0 :
Start with all men and women being free

Step 1 :
while there are free men, initially select one of them and do the following

(i) Proposal – The selected free man 'm' proposed w, the next woman on his preference list. (the highest ranked woman and who has not rejected him before)

(ii) Response – If w is free, she accepts the proposal to be matched with m.
If she is not free, she compares m with the current mate.
If she prefers m, she accepts m's proposal, making that former mate free, otherwise she simply rejects m's proposal leaving m free.

Step 2 :

Accept – first offer
Reject – Worse than current offer
Accept – Better than current offer

Ranking matrix

Free men Bob, Jim, Tom		Ann	Lea	Sue
	Bob	2,3	$\boxed{1,2}$	3,3
	Jim	3,1	1,3	2,1
	Tom	3,2	2,1	1,2

Bob proposed to Lea, Lea accepted

Free men Jim, Tom		Ann	Lea	Sue
	Bob	2,3	$\boxed{1,2}$	3,3
	Jim	3,1	1,3	$\boxed{2,1}$
	Tom	3,2	2,1	1,2

_ → rejected $\square$ → accepted

Jim proposed to Lea, Lea rejected because Lea's current preference = 2 and new proposal = 3
∴ 2 is better than 3.

Jim proposed to Sue, Sue accepted.

Free men Tom		Ann	Lea	Sue
	Bob	2,3	1,2	3,3
	Jim	3,1	1,3	$\boxed{2,1}$
	Tom	3,2	$\boxed{2,1}$	1,2

Tom proposed to Sue, Sue Rejected, because Sue's current preference = 1 and new proposal = 2
Tom proposed to Lea, Lea compares Bob with Tom, Tom has 1st preference. So she rejected Bob and accepts Tom.

		Ann	Lea	Sue
Free man Bob	Bob	$\boxed{2,3}$	1,2	3,3
	Jim	3,1	1,3	$\boxed{2,1}$
	Tom	3,2	$\boxed{2,1}$	1,2

Bob proposed to Lea, she rejected and Bob proposed to Ann, she accepted.

Stable Matching

(Bob, Ann)
(Jim, Sue)
(Tom, Lea)

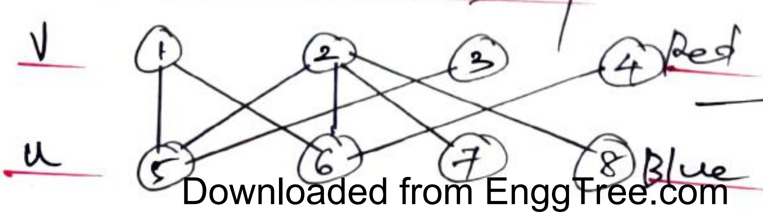
Example 4 Maximum matching in Bipartite graphs

Bipartite graph

It is a graph, where all the vertices can be partitioned into two disjoint sets V and U, not necessary of the same size, so that every edge connects a vertex in one of these sets to a vertex in the other set.

In other words, a graph is bipartite, if its vertices can be coloured in two colours, so that every edge has its vertices coloured in different colours. Such graphs can also be called 2-Colourable graph.

Eg



→ Bipartite graph

Matching

A matching in a graph is a subset of its edges with the property that no two edges share a same vertex.

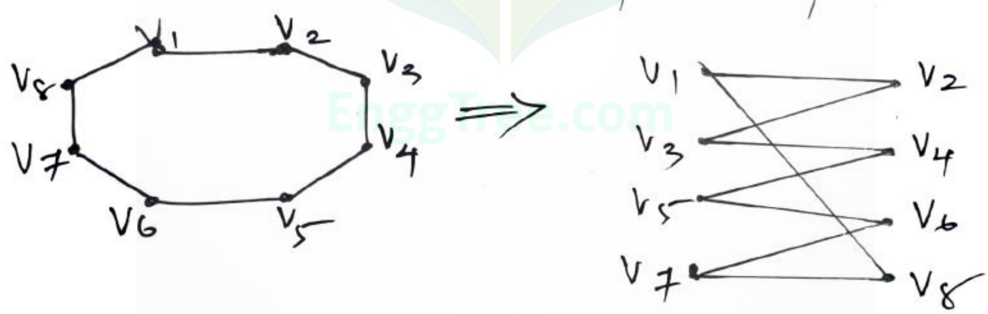
Maximum matching (or) maximum cardinality matching

It is a matching with the largest number of edges.

Perfect match

A matching that matches all the vertices of a graph.

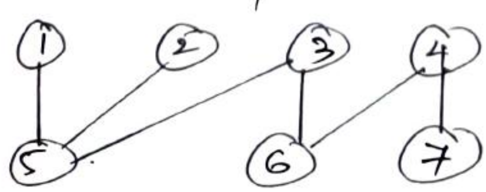
1) Find whether the given graph is Bipartite or not.



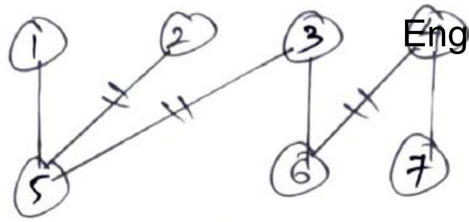
The above graph is bipartite.

A graph is bipartite, if and only if it does not have a cycle of an odd length.

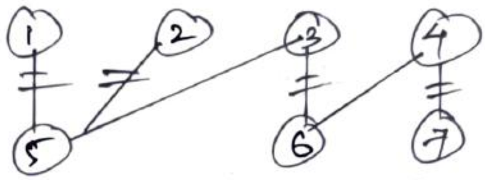
2) Apply the maximum-matching algorithm to the following bipartite graph.



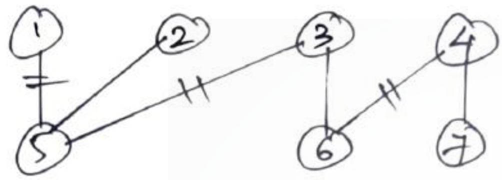
Max matching pair $= \lfloor n/2 \rfloor$
$= \lfloor 7/2 \rfloor = \lfloor 3.5 \rfloor$
$= 3$



Matching pair = {(1,5) (3,6) (4,7)}



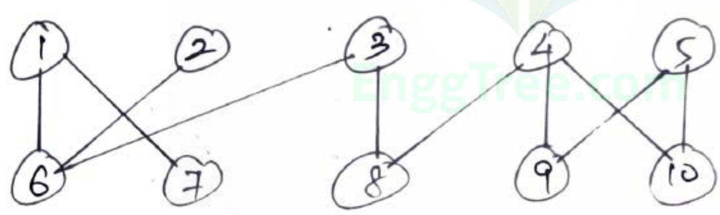
Matching pair = {(3,5) (4,6)}

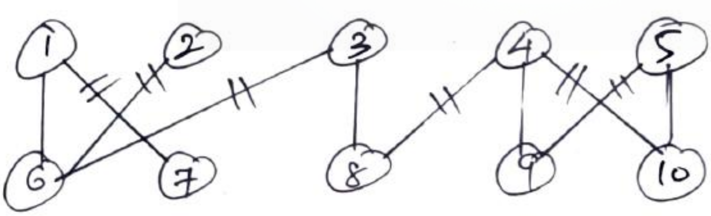


Matching pair = {(2,5), (3,6) (4,7)}

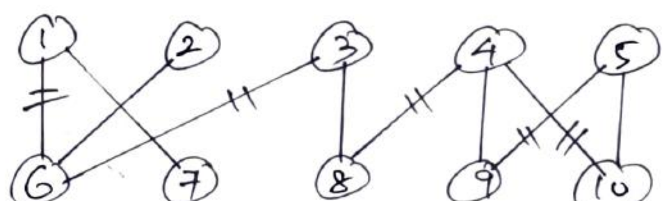
Maximum matching = 3

3.

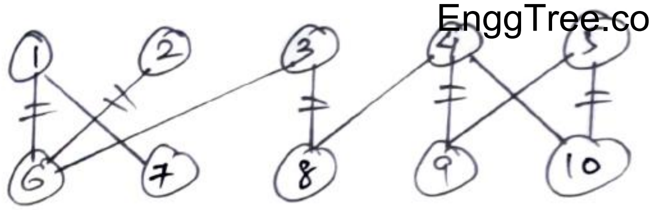




Matching pair = {(1,6) (3,8) (5,10) (4,9)}



Matching pair = {(2,6) (3,8) (1,7) (4,9) (5,10)}



matching pair = { (1,7) (3,6) (4,8) (4,10) (5,9) }

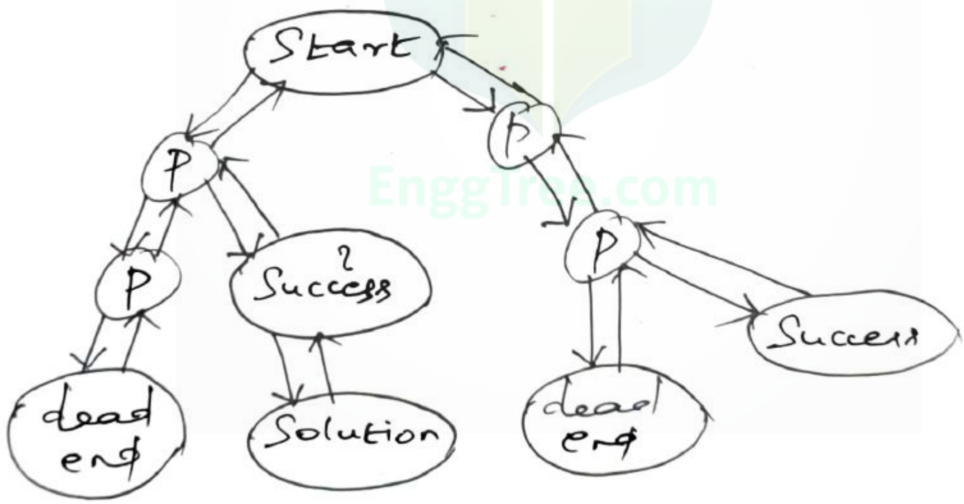
Max matching = 5

$\lfloor n/2 \rfloor = \lfloor 10/2 \rfloor = 5$

UNIT V

Backtracking

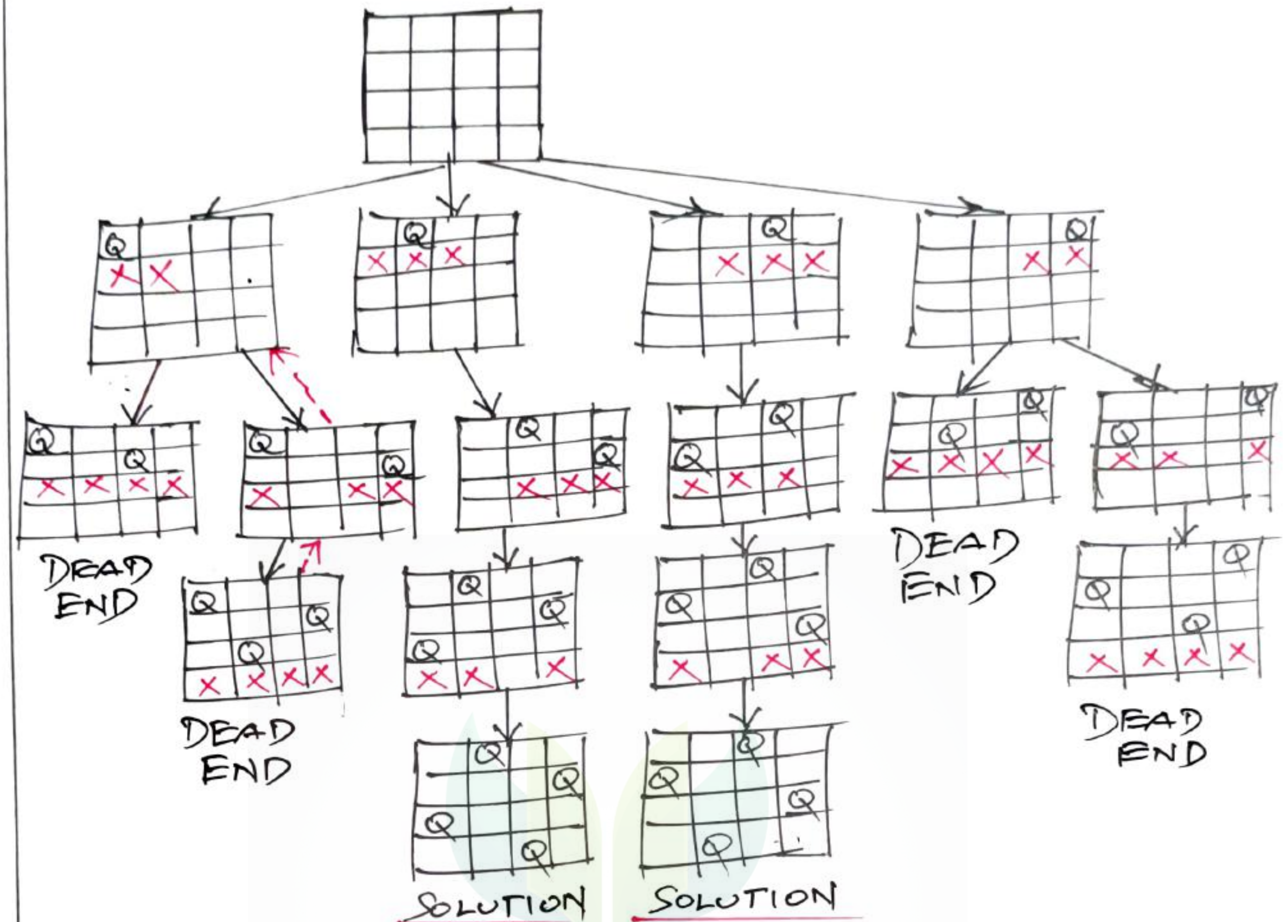
It is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.



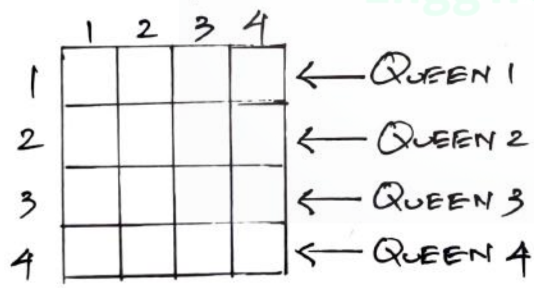
Example: n-Queens problem

The problem is to place n queens on an n by n chessboard, so that no 2 queens attack each other by being in the same row or in the same column or in the same diagonal.

State Space Tree



Conditions



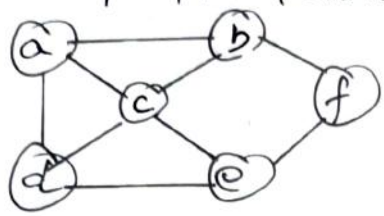
Queen 1 should be placed in only row 1
Queen 2 should be placed in only row 2
Queen 3 should be placed in only row 3
Queen 4 should be placed in only row 4

From the above state space tree

the optimal solution is (2, 4, 1, 3) & (3, 1, 4, 2)

Example 2 Hamiltonian Circuit Problem

Find the Hamiltonian cycles for the following graph



Make vertex a as the root of the state space tree using the alphabet order to break the three-way tie among the vertices adjacent to a, vertex b is selected. From b, the algorithm proceeds to c and then to d and e and finally to f, which proves to be deadend. So the algorithm backtracks from f to e, then to d and then to c. Illy the algorithm checks every possible tour to find the solution.

Adjacency nodes:

Write the adjacent nodes in ascending order

a = {b, c, d}

b = {a, c, f}

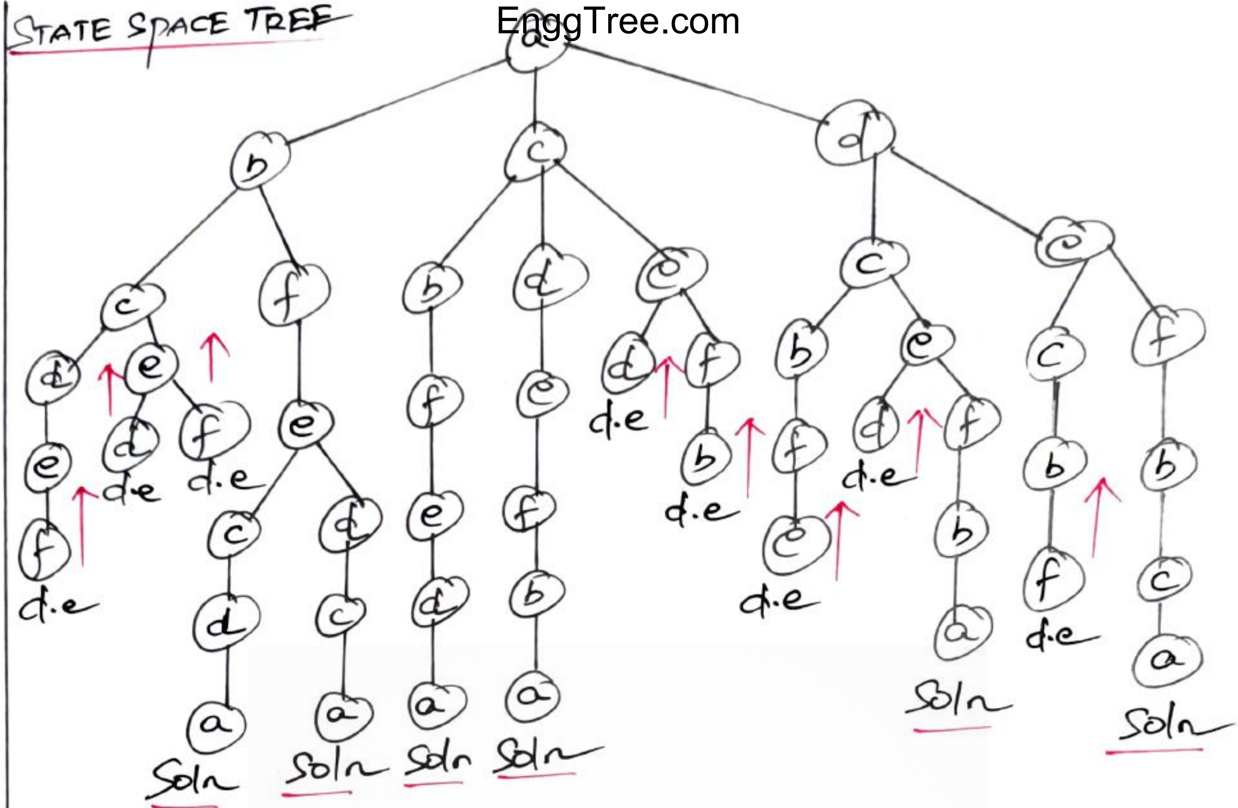
c = {a, b, d, e}

d = {a, c, e}

e = {c, d, f}

f = {b, e}

STATE SPACE TREE



Hamiltonian cycles

a – b – f – e – c – d – a

a – b – f – e – d – c – a

a – c – b – f – e – d – a

a – c – d – e – f – b – a

a – d – c – e – f – b – a

a – d – e – f – b – c – a

Example 3 — Subset sum problem

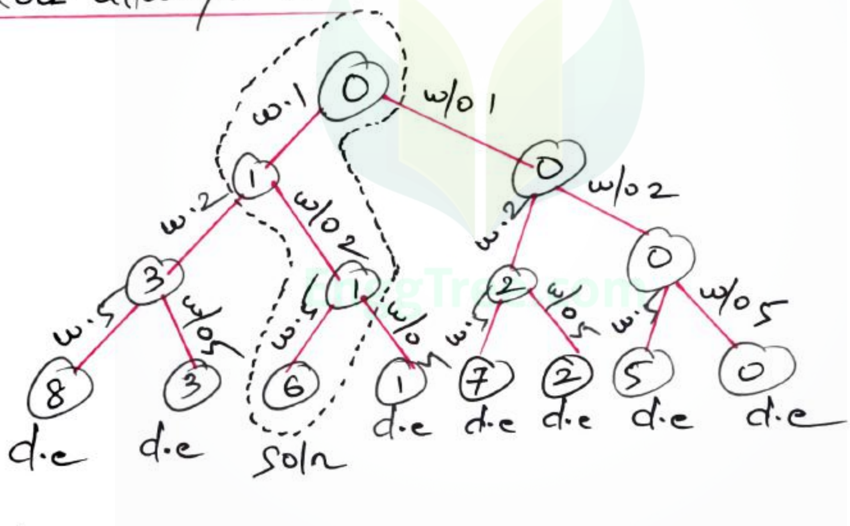
Find the subset of a given set $s = \{e_1, e_2, \ldots e_n\}$ of n positive integers where sum of subset is equal to the given integer d, which is a positive integer.

1. $S = \{1, 2, 5\}$
   $d = 6$

Apply backtracking concept and find the subset of $s = \{1, 2, 5\}$, $d = 6$

State space tree :

Root allways 0



$s' = \{1, 5\}$

Branch and Bound Technique

* It is an enhancement of backtracking
* Applicable to optimization problems
* An optimization problem seeks to minimize (or) maximize some objective function, usually subject to some constraints.

In optimization problem, there are 2 types of solutions.

1) Feasible solution: is a point in the problem's search space that satisfies all problem's constraints.
2) Optimal solution: is a feasible solution with the best value of the objective function.

Example 4 Assignment Problem

It is minimization problem, each person is assigned to job, with minimum cost with total assignment cost should be reduced.

Cost Matrix

$$\begin{array}{c|cccc} & J_1 & J_2 & J_3 & J_4 \\ \hline P_1 & 9 & \boxed{2} & 7 & 8 \\ P_2 & 6 & 4 & \boxed{3} & 7 \\ P_3 & 5 & 8 & \boxed{1} & 8 \\ P_4 & 7 & 6 & 9 & \boxed{4} \end{array}$$

Step 1: Construct root with lower-bound value.

lower bound (lb) = sum of min. values in each row.

$lb = 2+3+1+4 = 10$

Step 2:

Assign person 1 to all jobs and find the lower bound.

$P_1 \rightarrow J_1$

$lb = 9+3+1+4 = 17$

$P_1 \rightarrow J_2$

$lb = 3+2+1+4 = 10$

$P_1 \rightarrow J_3$

$lb = 4+5+7+4 = 20$

$P_1 \rightarrow J_4$

$lb = 8+3+1+6 = 18$

$P_1 \rightarrow J_2$ is small

Hence $\boxed{P_1 \rightarrow J_2}$ is fixed

Step 3:

Assign P_2 to all jobs and find lb

$P_1 \rightarrow J_2$ is fixed

$P_2 \rightarrow J_1$

$lb = 2+6+1+4 = 13$

$P_2 \rightarrow J_3$

$lb = 2+3+5+4 = 14$

$P_2 \rightarrow J_4$

$lb = 2+7+1+7 = 17$

min $lb = P_2 \rightarrow J_1$

Hence $\boxed{P_2 \rightarrow J_1}$ is fixed

Step 4:

Assign P_3 to all jobs and find lb

$P_1 \rightarrow J_2$ and $P_2 \rightarrow J_1$ is fixed

$P_3 \rightarrow J_3$

$lb = 2+6+1+4 = 13$

$P_3 \rightarrow J_4$

$lb = 2+6+8+9 = 25$

min $lb = P_3 \rightarrow J_3$

Hence $\boxed{P_3 \rightarrow J_3}$ is fixed

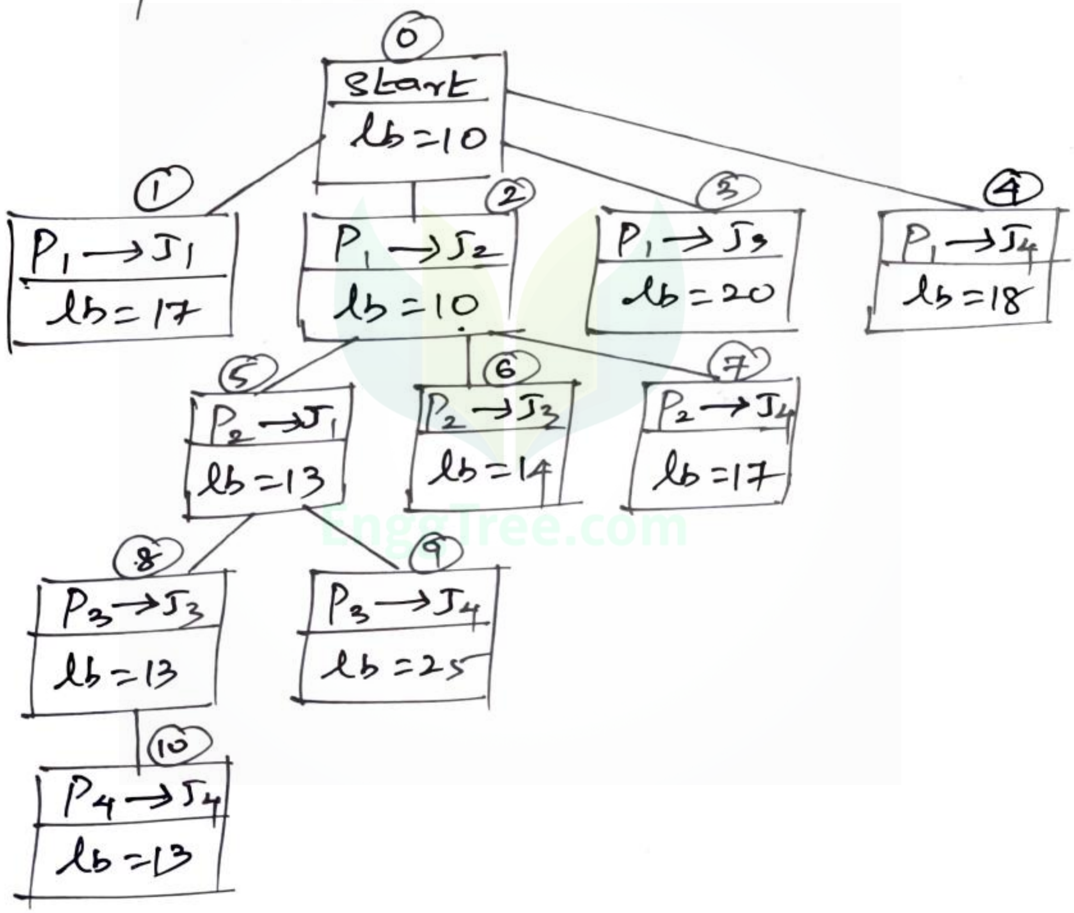
Step 5

Assign $\boxed{P_4 \rightarrow J_4}$ with $P_1 \rightarrow J_2$, $P_2 \rightarrow J_1$ & $P_3 \rightarrow J_3$ fixed

$P_4 \rightarrow J_4$

$lb = 2+6+1+4 = 13$

STATE SPACE TREE



Optimal Solution

$P_1 \rightarrow J_2$

$P_2 \rightarrow J_1$

$P_3 \rightarrow J_3$

$P_4 \rightarrow J_4$

Minimum assignment cost $= 2+6+1+4$

$= 13$

Example 5 Knap-sack Problem

Item	weight	Value
1	4	40
2	7	42
3	5	25
4	3	12

Knapsack quantity = 10
$W = 10$

Step 1

Find the value-weight ratio (v/w)

Item	v/w
1	$40/4 = 10$
2	$42/7 = 6$
3	$25/5 = 5$
4	$12/3 = 4$

Step 2

Sort the item in descending order of v/w.
It is already in sorted order.

Step 3

Node 0

Initially no items have been selected

So $w = 0$, $v = 0$, $W = 10$

upperbound $ub = v + (W - w)\dfrac{v_{i+1}}{w_{i+1}}$ where i = items

$v_i \rightarrow$ Value of item1

$w_i \Rightarrow$ weight of item1

$ub = 0 + (10 - 0).10$

$\boxed{ub = 100}$

Step 4:

① $v = 40, w = 4$

$ub = 40 + (10-4) \cdot 6$

$= 40 + 6 \cdot 6$

$= 76$

② $v = 0, w = 0$

$ub = 0 + (10-0) \cdot 6$

$= 60$

node ① is max, expand node ①

Step 5:

At node ③ $11 > w$, ∴ we cannot proceed

④ $v = 40, w = 4$

$ub = 40 + (10-4) \cdot 5$

$= 40 + 30$

$= 70$

Step 6:

⑤ $v = 65, w = 9$

$ub = 65 + (10-9) \cdot 4$

$= 65 + 4$

$= 69$

⑥ $v = 40, w = 4$

$ub = 40 + (10-4) \cdot 4$

$= 40 + 24$

$= 64$

Node ⑤ is max, ∴ expand node ⑤

(7) $w = 9+3 = 12 \geq W$

$\therefore$ we cannot proceed

(8) $v = 65, w = 9$

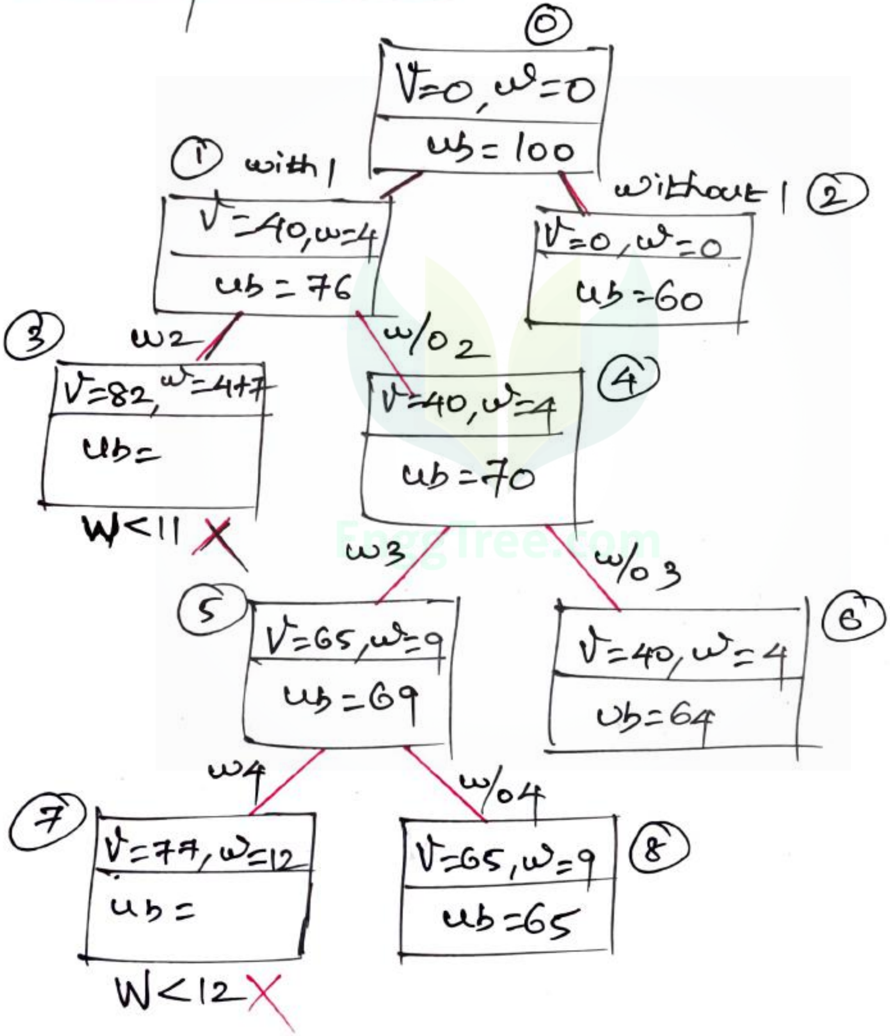
$ub = 65 + (10-9) \cdot 0$

$ub = 65$

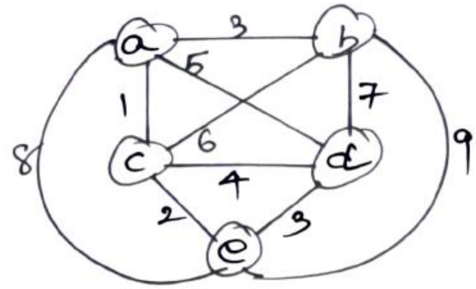
Hence most valuable subset = {1, 3}

Value = 65

STATE SPACE TREE



Example 6 Travelling Salesman Problem



Computing lower bound,

$lb = \lceil$ sum of costs of 2 least cost edges adjacent to $v / 2 \rceil$

$$lb = \lceil \overset{a}{(1+3)} + \overset{b}{(3+6)} + \overset{c}{(1+2)} + \overset{d}{(3+4)} + \overset{e}{(2+3)} / 2 \rceil$$

$$= \lceil (4+9+3+7+5)/2 \rceil$$

$$= \lceil 28/2 \rceil = 14$$

Conditions

1. Consider the tour start at a. Graph is undirected, we can generate only tours in which b is visited before c.
2. After visiting $n-1 = 4$ cities, a tour has no choice but to visit the remaining unvisited city and return to starting city.

At node 0

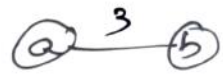
Obtain lower bound

$a = ac + ab = 1+3 = 4$
$b = ba + bc = 3+6 = 9$
$c = ca + ce = 1+2 = 3$
$d = dc + de = 4+3 = 7$
$e = ec + ed = 2+3 = 5$

$lb = \lceil (4+9+3+7+5)/2 \rceil = \lceil 28/2 \rceil = 14$

At node 1

Consider a-b and choose each vertex with one edge incident with a-b and other edge as minimum.



$a = ab + ac = 3 + 1 = 4$
$b = ab + bc = 3 + 6 = 9$
$c = ac + ce = 1 + 2 = 3$
$d = dc + de = 4 + 3 = 7$
$e = ec + ed = 2 + 3 = 5$

$lb = \lceil (4 + 9 + 3 + 7 + 5)/2 \rceil$
$= \lceil 28/2 \rceil = 14$

At node 2

B is to be visited before c. Hence a-c cannot be considered.

At node 3

Consider a-d

$a = ad + ac = 5 + 1 = 6$
$b = ba + bc = 3 + 6 = 9$
$c = ca + ce = 1 + 2 = 3$
$d = ad + de = 5 + 3 = 8$
$e = ed + ec = 3 + 2 = 5$

$lb = \lceil (6 + 9 + 3 + 8 + 5)/2 \rceil$
$= \lceil 31/2 \rceil = \lceil 15.5 \rceil = 16$

At node 4

Consider a-e

$a = ae + ac = 8 + 1 = 9$
$b = ba + bc = 3 + 6 = 9$
$c = ca + ce = 1 + 2 = 3$
$d = de + dc = 3 + 4 = 7$

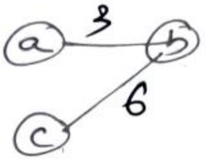
$e = ae + ce = 8+2 = 10$

$lb = \lceil (9+9+3+7+10)/2 \rceil$

$= \lceil 38/2 \rceil = 19$

At node 5

consider a-b b-c



$a = ab + ac = 3+1 = 4$

$b = ab + bc = 3+6 = 9$

$c = ac + bc = 1+6 = 7$

$d = dc + de = 4+3 = 7$

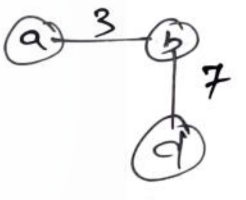
$e = ec + ed = 2+3 = 5$

$lb = \lceil (4+9+7+7+5)/2 \rceil$

$= \lceil 32/2 \rceil = 16$

At node 6

Consider a-b b-d



$a = ab + ac = 3+1 = 4$

$b = ba + bd = 3+7 = 10$

$c = ca + ce = 1+2 = 3$

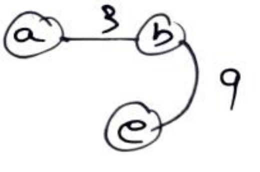
$d = bd + de = 7+3 = 10$ $e = ed + ec = 2+3 = 5$

$lb = \lceil (4+10+3+10+5)/2 \rceil$

$= \lceil 32/2 \rceil = 16$

At node 7

consider a-b b-e



$a = ab + ac = 3+1 = 4$

$b = ab + be = 3+9 = 12$

$c = ca + ce = 1+2 = 3$

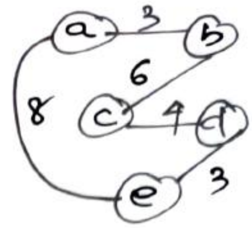
$d = de + dc = 3+4 = 7$

$e = be + ec = 9+2 = 11$

$lb = \lceil (4+12+3+7+11)/2 \rceil$

$= \lceil 37/2 \rceil = \lceil 18.5 \rceil = 19$

At node 8

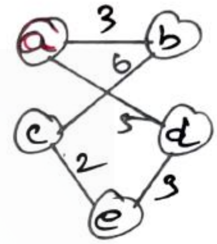


a b c d with e a

$a = 3+8 = 11$
$b = 3+6 = 9$
$c = 6+4 = 10$
$d = 4+3 = 7$
$e = 3+8 = 11$

$lb = \lceil (11+9+10+7+11)/2 \rceil = \lceil 48/2 \rceil = 24$

At node 9

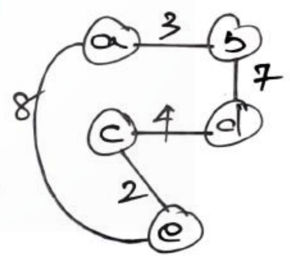


a b c e with d a

$a = 3+5 = 8$
$b = 3+6 = 9$
$c = 6+2 = 8$
$d = 5+3 = 8$
$e = 2+3 = 5$

$lb = \lceil (8+9+8+8+5)/2 \rceil = \lceil 38/2 \rceil = 19$

At node 10

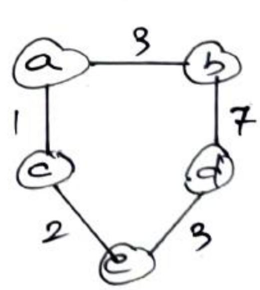


a b d c with e a

$a \quad 3+8 = 11$
$b = 3+7 = 10$
$c = 4+2 = 6$
$d = 7+4 = 11$
$e = 2+8 = 10$

$lb = \lceil (11+10+6+11+10)/2 \rceil = \lceil 48/2 \rceil = 24$

At node 11



a b d e with c a

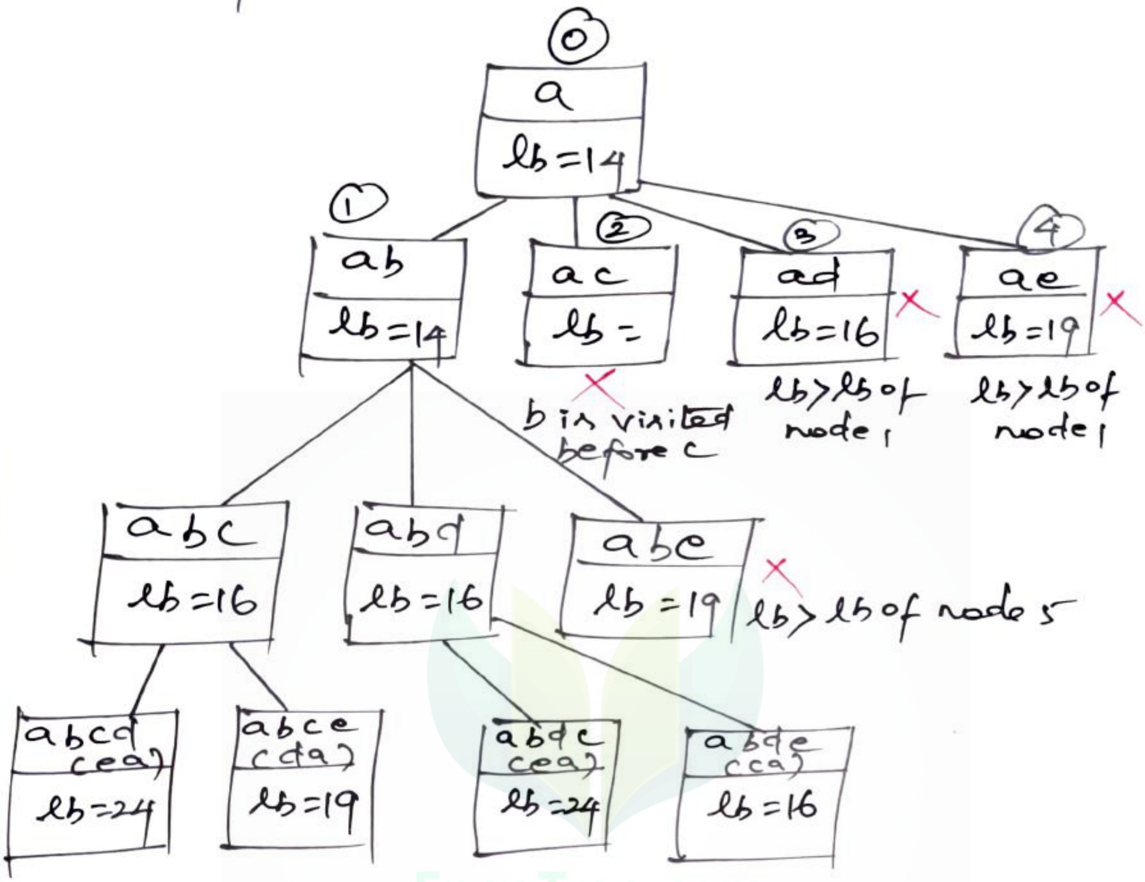
$a = 3+1 = 4$
$b = 3+7 = 10$
$c = 1+2 = 3$
$d = 7+3 = 10$
$e = 2+3 = 5$

$lb = \lceil (4+10+3+10+5)/2 \rceil = \lceil 32/2 \rceil = 16$

optimal tour is $a \rightarrow b \rightarrow d \rightarrow e \rightarrow c \rightarrow a$

optimal cost is 16

STATE SPACE TREE



Polynomial time

An algorithm is said to be solvable in polynomial time, if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k, where n is the complexity of the i/p.

Class P problems

P – polynomial time solving

problems which can be solved in polynomial time, which take time like $O(n)$, $O(n^2)$, $O(n^3)$...

Eg: Finding max element in an array

class NP problems

NP – non deterministic polynomial time solving problems which cannot be solved in polynomial time

Eg: Travelling Salesman Problem, Subset sum problem

But NP problems are checkable in polynomial time. It means that the NP problem can be checked for correctness in polynomial time.

Take two problems A and B, both are NP problems

Reducibility – If one instance of a problem A is converted into problem B, then it means that A is reducible to B.

NP-hard : If A is reducible to B, it means that B is atleast as hard as A.

NP-Complete : The group of problems which are both in NP and NP-hard are called NP-Complete problems.

Approximation algorithm for NP-hard problems

It is the branch of algorithm which deals with special problems. Approximation comes from the fact that, the algorithm gives approximate solution and not the best solution.

If approximation algorithm is used, the output is just an approximation of the actual optimal solution.

Accuracy of the appr. alg. can be calculated by

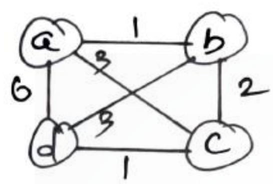
$$r_e(S_a) = \frac{f(S_a)}{f(S^*)}$$

where, $S_a \rightarrow$ approximation solution

$S^* \rightarrow$ exact solution

Approximation algorithm for Travelling Salesman Problem

Nearest neighbour algorithm



Step 1: Choose an arbitrary city as the start.

Step 2: choose the edge with the smallest cost and use that as the first edge in your circuit.

Step 3: Go to the unvisited city nearest to the one visited last with the smallest cost.

Step 4: Repeat the above until all the vertices are visited.

Step 5: Return to the starting vertex.

From the given graph,

Nna yields the tour

$S_a = a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{1} d \xrightarrow{6} a$

$= 1+2+1+6 = 10$

Optimal solution by exhaustive search is

$S^* = a \xrightarrow{1} b \xrightarrow{3} d \xrightarrow{1} c \xrightarrow{3} a$

$= 1+3+1+3 = 8$

$$r_e(S_a) = \frac{f(S_a)}{f(s^*)} = 10/8 = 1.25$$

(ie) The tour Sa is 25% longer than optimal tour s^*.

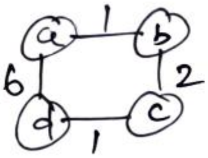
Multi-fragment heuristic algorithm

Step 1 : Start 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 until a tour of length n is obtained, where n is the no. of cities.
Add the next edge to the set of tour edge, 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

$\boxed{\begin{array}{l} a-b=1 \\ d-c=1 \\ b-c=2 \\ b-d=3 \\ a-c=3 \\ a-d=6 \end{array}}$



The algorithm yields
$\{(a,b)\ (b,c)\ (c,d)\ (d,a)\}$

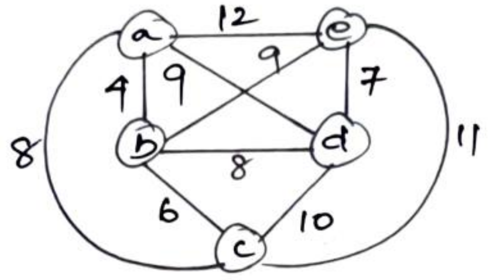
The majority of practical applications of the TSP are its Euclidean instances.

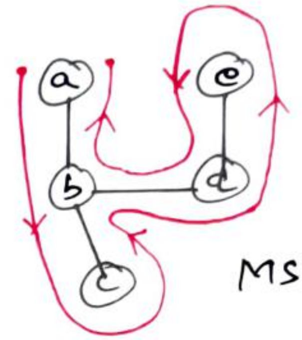
The performance ratio of nearest-neighbour and multifragment-heuristic algorithm is based on Euclidean instances.

Accuracy ratio $\frac{f(S_a)}{f(s^*)} \leq \frac{1}{2}\left(\lceil \log_2 n \rceil + 1\right).$

Minimum - spanning tree based algorithm

Twice - around - the - tree algorithm





MST

Step 1: construct a minimum spanning tree of the graph

Step 2: walk around the spanning tree starting at an arbitrary vertex, by recording the vertices covered in the walk.

Step 3: From the obtained list, eliminate the repeated nodes, except the node from which the walk is started. The circuit formed by the remaining node is the Hamiltonian Circuit

MST is made up of edges

(a, b) (b, d) (b, c) (d, e)

Twice - around - the - tree walk gives

a - b - c - b - d - e - d - b - a

Eliminate the second b, second d and third b which yeilds the Hamiltonian Circuit.

a - b - c - d - e - a

4 + 6 + 10 + 7 + 12 = 39

length of tour = 39