SIDDARTH INSTITUTE OF ENGINEERING &TECHNOLOGY:: PUTTUR
(AUTONOMOUS)
Siddharth Nagar, Narayanavanam Road – 517583
QUESTION BANK (DESCRIPTIVE)

Subject with Code: Design and Analysis of Algorithms (**20CS0523**)
Course & Branch: B. Tech– CCC
Year & Sem: II B. Tech & II- Sem **Regulation:** R20

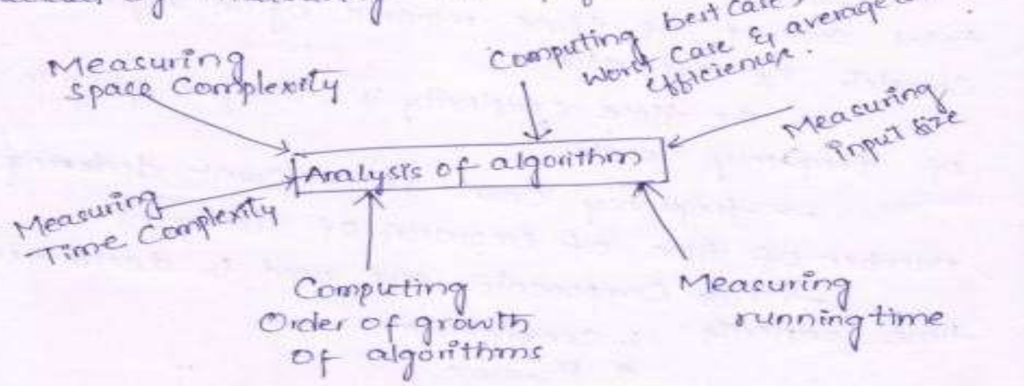
UNIT –I
INTRODUCTION, DISJOINT SETS

1	**a)**	**What do you mean by algorithm? List some of the properties of it.**	**[L1][CO1]**	**[04M]**
		• An algorithm can be defined as a finite set of steps, which has to be followed while carrying out a particular problem. It is nothing but a process of executing actions step by step. **Characteristics of Algorithms:** ○ **Input:** It should externally supply zero or more quantities. ○ **Output:** It results in at least one quantity. ○ **Definiteness:** Each instruction should be clear and ambiguous. ○ **Finiteness:** An algorithm should terminate after executing a finite number of steps. ○ **Effectiveness:** Every instruction should be fundamental to be carried out, in principle, by a person using only pen and paper. ○ **Feasible:** It must be feasible enough to produce each instruction. ○ **Flexibility:** It must be flexible enough to carry out desired changes with no efforts. ○ **Efficient:** The term efficiency is measured in terms of time and space required by an algorithm to implement. Thus, an algorithm must ensure that it takes little time and less memory space meeting the acceptable limit of development time. ○ **Independent:** An algorithm must be language independent, which means that it should mainly focus on the input and the procedure required to derive the output instead of depending upon the language.		
	b)	**Classify the rules of Pseudo code for Expressing Algorithms.**	**[L2][CO1]**	**[08M]**

	1. Algorithm is a procedure consisting of heading and body. The heading consists of keyword algorithm and name of the algorithm and parameter list The syntax is,
	Algorithm name $(P_1, P_2, \ldots P_n)$
	This keyword is written first — Here write the name of the Algorithm — Write parameters (if any)
	2. Then in the heading section we should write following things
	// problem Description:
	// Input:
	// Output:
	3. Then body of an algorithm is written in which various programming constructs like if, for, while (or) some assignment statements may be written.
	4. The compound statements should be enclosed with in { and } brackets.
	5. Single line comments are written using // as beginning of comment.
	6. The identifier should begin by letter and not by digit. An Identifier can be a combination of Alphanumeric string.
	7. Using Assignment operator ← an assignment statement can be given
	Variable ← Expression.
	8. There are other types of operators such as boolean operators such as true (or) false. Logical operations such as and, or, not. And relational operators such as <, >, <= >= & so on.
	9. The array indices are stored with in square brackets [ ]. The index of array usually start at zero. The multidimensional array can also used in algorithm
	10. The inputting and outputting can done using read and write for eg:
	Write ("this message will be displayed on Console");		

2	**Simplify steps involved in performance analysis with example.**	**[L2][CO1]**	**[12M]**

The efficiency of an algorithm can be decided by meacuring the performance of an algorithm.



1. Space Complexity:

The Space Complexity Can be defined as amount of memory required by an algorithm to run.

→ To Compute the space complexity We use two factors

1. Constant / fixed part
2. Instant characteristics / variable part

→ The Space requirement S(p) can be given as.

$S(p) = C + S_p$

S(p) → Space requirement

p → be the algorithm name.

c → Constant

S_p → Variable part

2. Time Complexity:

→ The time complexity of an algorithm is the total amount of time required by an algorithm to Complete its execution

→ The time complexity is therefore given in terms of frequency Count.

→ frequency Count is a Count denoting number of time of execution of statement.

→ Two Components are used to determines the time Complexity. 1. Compiletime.
2. Runtime.

3. Measuring an input size:

→ If the input size is larger then usually algorithm runs for a longer time

→ Hence we can compute the efficiency of an algorithm as a function to which input size is passed as a parameter.

→ To implement an algorithm we require prior knowledge of input size.

4. Measuring Running Time:

The time complexity is measured in terms of a unit called frequency count. The time which is measured for analyzing an algorithm is generally running time.

From an algorithm:

→ First identify the important operation of an algorithm. This operation is called the basic operation. → Basic operation is not difficult to identify from an algorithm

→ Generally the basic operation which is more time consuming is a basic operation in the algorithm.

5. Order of growth:

→ Order of growth provide the behaviour of an algorithm.

→ It gives a simple characterization of the algorithm efficiency.

→ It allows us to compare the relative performance of alternative algorithms.

→ It is done based on the primitive operations like arithmetic, relational, logical operations.

→ Order of growth is defined when the input size of an algorithm increases then the computational time of an algorithm also increases ie, input is directly propotional to computational time.

→ Order of growth follows the relation below

$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < \ldots < O(2^n) < O(n!)$$

→ The order of growth two bounds are used for analyzing the functions.

1. upper bound
2. lower bound

* upper bound defined the after the O(n) function growth ($O(n\log n), O(n^2), O(n^3), O(2^n), O(n!)$)

* Lower bound defines the below the O(n) function growth.

($O(1), O(\log n)$)

6. Computing best case, Worst Case, Average Case 5 efficiency:-

1. Bestcase: It is the minimum number of steps that can be executed for a given parameter.

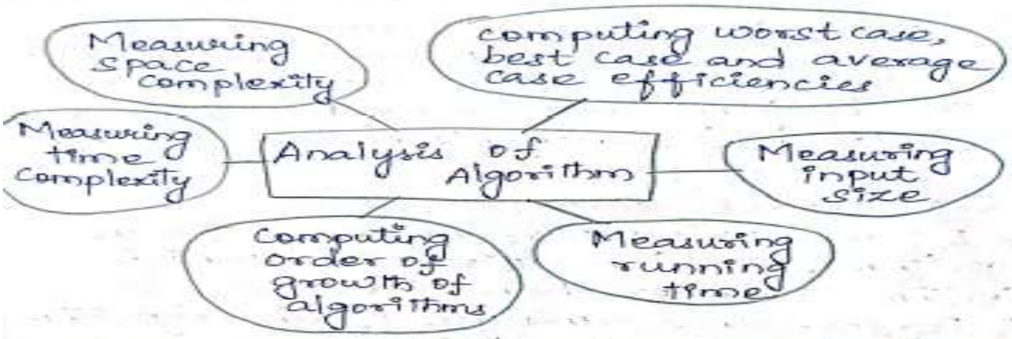
2. Worst case: It is the maximum number of steps that can be executed for a given parameter.

3. Average Case: It is the Average number of steps that can be executed for a given parameter.

3	**a) Explain space complexity and time complexity in detail with example.**	**[L2][CO1]**	**[08M]**

Space complexity:-

→ The space complexity of an algorithm is the amount of memory it needs to run to completion.



→ To compute the space complexity we use two factors

1. constant / fixed part
2. Instance characteristics / Variable Part.

→ The space requirement S(P) can given as

$$S(P) = C + S_p$$

S(P) → space requirement
P → be the algorithm name.
C → constant i.e fixed part
S_p → Variable part.

Fixed Part:-

→ It denotes the space of inputs and outputs.

→ Includes space needed for storing instructions, variables, constants and structured variables [arrays, struct].

Variable part:-

→ This variable part whose space requirement depends on particular problem instance.

→ Includes space needed for stack and for structured variables that are dynamically allocated during run time.

→ Typically include

1. space needed by referenced variable.
2. Recursion stack space and so on.

Rules:-

→ size of variable like n, a, b assigns -1 word.

→ Array values it assigns -n words.

→ In loop variable assignment operators assigns - 1 word.

Example 1 :-

Consider the following algorithm & calculate space complexity.

1) Write an algorithm for sum of n elements in the array & calculate space complexity.

Solution:-

Algorithm sum(a, n)

// problem Description : The algorithm for sum of n elements in the array.

// Input :- An array a & n is total number of elements in the array.

// output :- Returns the value to the sum's' and stored in that variable & display output.

```
{
    s ← 0
    for i ← 1 to n do
    s ← s + a[i]
    return s;
}
```

The space requirement for the above algorithm is

$S(P) = C + S_P$

$S(\text{sum}) = C + S_p$

∴ $s \leftarrow 0$ is assigning the value so 1 word.

∴ $i = 1$ assignment operator so 1 word.

∴ n variable so 1 word.

∴ array a value a[i] so n word.

There fore

$\boxed{S(P) = (n+3)}$ words.

For Recursive algorithm:-

Eg:- sum of n elements.

Sol:- Algorithm sum(a, n)

// problem Description : This is a recursive algorithm.

// which computes addition of all the element in an array a[ ].

// Input : a[i] is Integer type, total number of elements in an array.

// output : Returns addition of 'n' elements of an array.

return sum(a, n-1) + a[n]

∴ The space requirement is

$\boxed{S(P) = 3(n+1)}$

→ The internal stack used for recursion includes space for formal parameters, local variables & return address.

→ The space required by each call to function 'sum' requires atleast three words.

space for n values | space for return address | pointer to a[ ]

→ The depth of recursion in n+1(n times call to function & one return call. The recursion stack space will be 3(n+1). words.

TIME COMPLEXITY:-

→ The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

→ The time complexity is therefore given in terms of frequency count.

→ Frequency count is a count denoting number of times execution of statement.

→ Two components are used to determine the time complexity.

1. compile time (or) constant time complexity.
2. Run time.

1. compile time:-

→ It does not depends on instance characteristics.

→ once compiled program will be run special times without recompilation.

constant time complexity:-

→ If any program requires fixed amount of time for all input values then its time complexity is said to be constant time complexity.

Eg:-
```
int sum(int a, int b)
{
    return a+b;
}
```

2. Run time:-

→ It dependent on particular problem instance [dependent on input and output].

∴ Time complexity calculated in the form of

$$T(P) = C + T_P$$

where

$T(P)$ = Time complexity of algorithm P.
C = compile time
T_P = Run time.

How to calculate Time complexity:-

1. Calculate time complexity will become easy by counting the number of steps each statement in the program executes.
2. A program step is a segment of a program that is independent of instance characteristics & execution on problem statements.
3. Identify the executable statements in the program and count the number of steps in the program assigned.

Rules:-

comments : 0 steps
For Assignment statement : 1 step
condition statement : 1 step
Loop condition for 'n' times: (n+1) steps
Body of loop : n steps.

→ Based on this rules, we are going to calculate the time complexity for any algorithm.

Example:- calculate the time complexity for sum of n elements in the array.

sol:- statement number : Number of lines.
statement : program of sum of n elements written in Pseudo code.

→ S/E : Status of execution.

→ Frequency count : The time complexity is measured in the terms of a unit called frequency count. The time which is measured for analyzing an algorithm is generally running time.

→ Total steps : Multiple both S/E & frequency & place it in total steps table.

→ Finally add all the total steps for the algorithm & you will get time complexity for an algorithm:

statement no	statement	S/E	frequency	Total steps
1.	Algorithm sum(a,n)	0	—	0
2.	{	0	—	0
3.	$S \leftarrow 0$	1	1	1
4.	for $i \leftarrow 1$ to n do	1	(n+1)	(n+1)
5.	$S \leftarrow S + a[i]$	1	n	n
6.	return S	1	1	1
7.	}	0	—	0

Total steps for algorithm $1+(n+1)+n+1$

$S \leftarrow 0$: Assignment operator so 1 step

for $i \leftarrow 1$ to n do : Loop statement iteration is performing so (n+1) steps.

s ← s+a[i] : body of the loop 'n' times

return s : for return statement ← 1 step

∴ Time complexity for the algorithm is

T(P) = (2n+3) steps.

Eg 2 :- calculate Time complexity for two matrix for the algorithm.

stmt no	Statement	s/E	frequency	Total steps
1.	Algorithm Add (a,b,c,m,n)	0	0	0
2.	{	0	0	0
3.	for i←1 to m do	1	(m+1)	(m+1)
4.	for j←1 to n do	1	m(n+1)	(mn+m)
5.	c[i][j]←a[i][j] +b[i][j]	1	mn	mn
6.	}	0		

Total steps for algorithm: (m+1)+(mn+m) +mn

∴ Time complexity for the algorithm is

T(P) = (m+1)+(mn+m)+mn

= m+1+mn+m+m

T(P) = 2m+2mn+1 steps

b) Illustrate an algorithm for Finding sum of natural number **[L2][CO1]** **[04M]**

Algorithm Sum (1,n)

//problem description : This algorithm is for finding the sum of given n numbers.

//Input : 1 to n numbers.

// Output: The sum of n numbers.

result ← 0

for i←1 to n do i← i+1

result ← result +i

return result.

4 **What is asymptotic notation? Explain different types of notations with examples.** **[L2][CO1]** **[12M]**

- **Asymptotic notations** are used to write fastest and slowest possible running time for an algorithm. These are also referred to as 'best case' and 'worst case' scenarios respectively.
- "In asymptotic notations, we derive the complexity concerning the size of the input. (Example in terms of n)"
- "These notations are important because without expanding the cost of running the algorithm, we can estimate the complexity of the algorithms."

Why is Asymptotic Notation Important?

1. They give simple characteristics of an algorithm's efficiency.

2. They allow the comparisons of the performances of various algorithms.

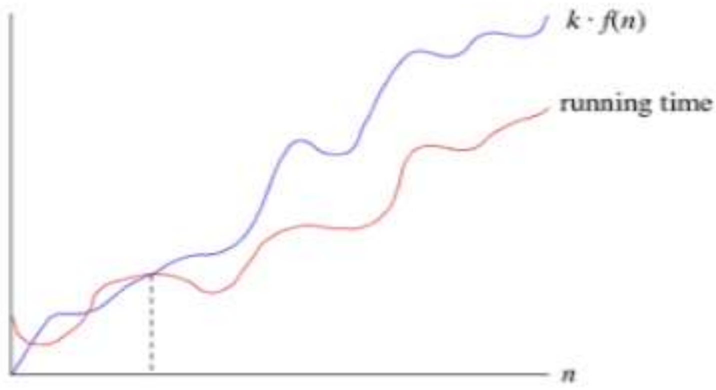
Asymptotic Notations:

- Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm:

1. Big-oh notation: Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time. The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if exist positive constant c and such that

f (n) ⩽ k.g (n)f(n)⩽k.g(n) for n>n0n>n0 in all case

Hence, function g (n) is an upper bound for function f (n), as g (n) grows faster than f (n)



ASYMPTOTIC UPPER BOUND

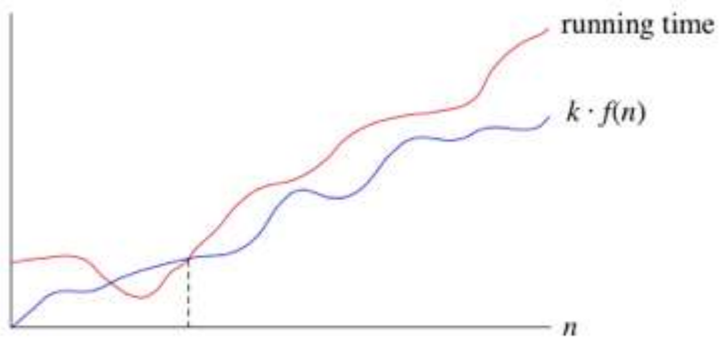
For Example:

1. 3n+2=O(n) as 3n+2≤4n for all n≥2
2. 3n+3=O(n) as 3n+3≤4n for all n≥3

Hence, the complexity of **f(n)** can be represented as O (g (n))

2. Omega () Notation: The function f (n) = Ω (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n_0 such that

$$F(n) \geq k^* g(n) \text{ for all } n, n \geq n_0$$



ASYMPTOTIC LOWER BOUND

For Example:

f (n) $=8n^2+2n-3 \geq 8n^2-3$

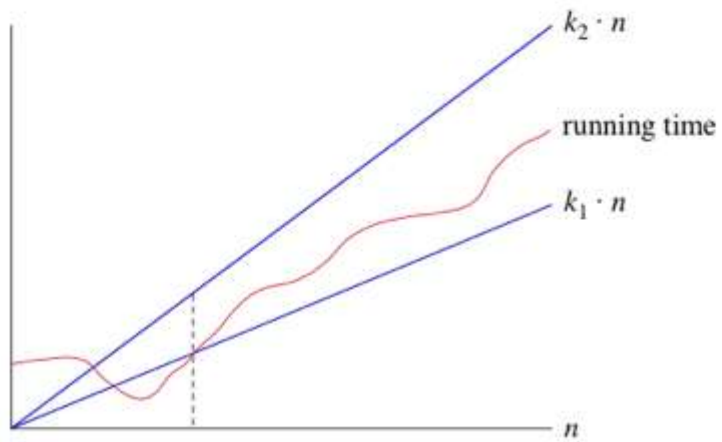
$=7n^2+(n^2-3) \geq 7n^2$ (g(n))

Thus, $k_1=7$

Hence, the complexity of **f (n)** can be represented as **Ω** (g (n))

3. Theta (θ): The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant k_1, k_2 and k_0 such that

$k_1 * g(n) \leq f(n) \leq k_2\, g(n)$ for all n, $n \geq n_0$



ASYMPTOTIC TIGHT BOUND

For Example:

$3n+2= \theta(n)$ as $3n+2 \geq 3n$ and $3n+2 \leq 4n$, for n

$k_1=3, k_2=4$, and $n_0=2$

Hence, the complexity of f (n) can be represented as θ (g(n)).

The Theta Notation is more precise than both the big-oh and Omega notation. The function f (n) = θ (g (n)) if g(n) is both an upper and lower bound.

5	**Discuss briefly with suitable example about Big 'O' notation and Theta notation**	**[L3][CO1]**	**[12M]**

Big Oh Notation:

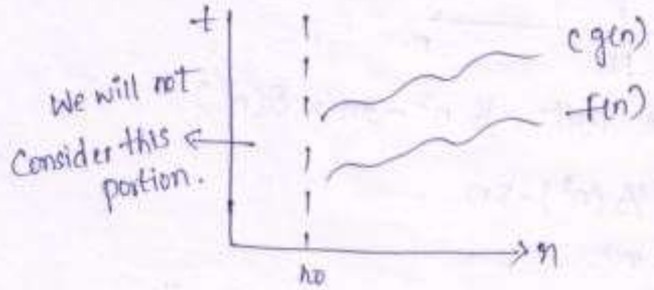
* Big Oh is the formal method of expressing the upper bound of an algorithms running time.

* It is the measure of the largest amount of time.

* the upper bound of $f(n)$ indicates that the functions $f(n)$ will be the worst case

Definition: The function $f(n) = O(g(n))$ such that $\exists$ two. with the Constant that positive Constants 'c' & 'no'

$$f(n) \leq c \cdot g(n) \quad \forall\, n \geq n_0$$



9

Example: Give $f(n) = 3n+2$, then PT $f(n) = O(n)$

$f(n) = 3n+2$

here

$f(n) \leq c \cdot g(n) \quad \forall\, n > n_0.$

$g(n) = n$

there always $n_0 = 1,\ n \geq 1$

$3n+2 \leq c(n)$

$3(1)+2 \leq c(1) \quad \therefore c = 5$

$5 \leq 5$

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

Theta Notation:

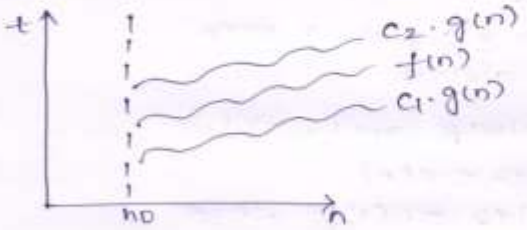
→ Theta Notation is denoted by using 'θ'

→ for some functions the lower bound and upper bound may be same ie Big oh and omega will have the same functions.

→ For example, find the maximum (or) minimum element in the given array, then complexity time for that min (or) max element is $O(n)$ and $\Omega(n)$.

→ If the functions having the same time complexity for lower and upper bounds and this notation is used it is called θ Notation.

Definition: the function $f(n) = \theta g(n)$ if there exists three positive constants c_1, c_2 & no with the

		Constants that $c_1 g(n) \le f(n) \le c_2 g(n) \quad \forall\ n \ge n_0$  Example: prove that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ $f(n) = \frac{1}{2}n^2 - 3n$ $g(n) = n^2$ $c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n) \quad \forall\ n \ge n_0.$ $c_1 \cdot n^2 \le \frac{1}{2}n^2 - 3n \le c_2 \cdot n^2$ $n=1 \quad c_1 = c_2 = -5/2$ $-5/2(1)^2 \le \frac{1}{2}(1)^2 - 3(1) \le -5/2(1)^2$ $-5/2 \le -5/2 \le -5/2$ $\therefore f(n) = \Theta(g(n))$ $\boxed{f(n) = \Theta(n^2)}$		
6	a)	**Solve the given function If f(n)= 5n^2+ 6n+ 4 then prove that f(n) is 0(n^2).**	**[L3][CO1]**	**[04M]**
		$f(n) = 5n^2 + 6n + 4$ $g(n) = n^2$ $f(n) \le c \cdot g(n) \quad \forall\ n \ge n_0.$ $5n^2 + 6n + 4 \le c \cdot n^2$ $c = 15,\ n = 1$ $5(1)^2 + 6(1) + 4 \le 15(1)^2$ $5 + 6 + 4 \le 15$ $15 \le 15$ $f(n) = O(g(n))$ $\boxed{f(n) = O(n^2)}$		
	b)	**Explain two types of recurrences in detail with suitable example.**	**[L2][CO1]**	**[08M]**

Suitable example.

The recurrence equation is an equation that defines a sequence recursively.

$$T(n) = T(n-1) + n \text{ for } n>0 \quad —①$$

$$T(0) = 0 \quad —②$$

Here eq① is called recurrence relation

eq② is called Initial Condition.

The Recurrence relation can be solved by following methods.

1. Substitution method.
2. For Master's Method.

1. The Substitution method is a kind of method consists of two steps.

a. Guess the solution.

b. Use the mathematical Induction to find the boundary condition and shows that the guess is correct.

There are 2 types

(i) Forward Substitution.

(ii) Backward Substitution.

(i) This method makes use of an initial condition in the initial term and value of the next term is generated.

* This process is continued until some formula is guessed

Eg: Consider a recurrence relation $T(n) = T(n-1)+n$ with initial condition $T(0)=0$.

Sol: Let

$$T(n) = T(n-1) + n \quad —①$$

$$T(0) = 0 \rightarrow \text{Initial condition.}$$

$$n=1 \Rightarrow T(1) = T(0) + 1$$

$$T(1) = 0+1 = 1 \quad —②$$

If $n=2$ then

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

If $n=3$ then

$$T(3) = T(3-1)+3$$
$$= T(2)+3$$
$$T(3) = 3+3 = 6 \quad \text{\& so on}$$

$\therefore$ The formula generated

$$T(n) = 1+3+6+\cdots$$

By observing above generated equations

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

$\therefore$ We can also denote $T(n)$ in terms of big oh notation as follows

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

(ii) Backward Substitution Method:

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

Eg: Consider a recurrence relation $T(n) = T(n-1)+n$ with initial condition $T(0)=0$

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

By Backward substitution

Let $n = n-1$

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

Substitute eq ② in eq ① we get

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

let $n = n-2$

$$T(n-2) = T(n-2-1)+(n-2)$$
$$= T(n-3)+(n-2) \quad \text{—④}$$

Sub eq ④ in ③

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

If $k=n$.

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

Again we can denote $T(n)$ in terms of big-oh notation as

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

2. Master's Method:

* Consider the following recurrence relation

$T(n) = aT(n/b) + f(n)$ where $n \geq d$ & d is some constant

* Then the Master's theorem can be stated for efficiency analysis as-

If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ in the recurrence then

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

Example:

1. $T(n) = 4T(n/2) + n$ & Solve by using Master Method

Sol: $T(n) = 4T(n/2) + n$.

The recurrence relation is

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

Here $a = 4$, $b = 2$ $f(n) = n$

Hence $d = 1$

Case 1:

$a < b^d$

$4 < 2^1$

$4 < 2$ (false

Case 2:

$a = b$

$4 = 2$

false

Case 3:

$a > b^d$

$4 > 2^1$

$4 > 2$ (True)

∴ Case 3 applied for the equation

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

$= \Theta(n^{\log_2^4})$

$= \Theta(n^{\log_2^{2^2}})$

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

∴ Hence Time Complexity is $\Theta(n^2)$

7	a)	Apply the Master's theorem to Solve the following Recurrence relations i) T(n) = 4T(n/2) + n ii) T(n) = 2T(n/2) + n1og n	[L3][CO1]	[06M]

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

$\therefore$ The recurrence relation is

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

Here $a = 4,\ b = 2,\ f(n) = n$

Hence $d = 1$

Case 1:

$a < b^d$

$4 < 2^1$

False

Case 2:

$a = b^d$

$4 \neq 2^1$

False

Case 3:

$a > b^d$

$4 > 2^1$

True

$\therefore$ Case 3 applied for the equation.

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

$= \theta(n^{\log_2^4})$

13

$= \theta(n^{\log_2 2^2})$

$= \theta(n^{2\log_2^2})$

$= \theta(n^{2(1)})$

$T(n) = \theta(n^2)$

Hence time complexity is $\theta(n^2)$

(ii) $T(n) = 2T(n/2) + n\log n$

$\therefore$ The recurrence relation is

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

Here $a = 2,\ b = 2,\ f(n) = n\log n$

Hence $d = 1$

Case 1:

$a < b^d$

$2 < 2^1$

False

Case 2:

$a = b^d$

$2 = 2^1$

~~False~~ True

Case 3:

$a > b^d$

$2 > 2^1$

~~True~~ False

$T(n) = \theta(n^d \log n)$

$= \theta(n^1 \log n)$

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

Hence Time Complexity is $\theta(n\log n)$

	b)	What is iterative substitution method? Apply the Iterative substitution method to Solve the following Recurrence relations. $T(n) = 2T(n/2) + n$	[L3][CO1]	[06M]

(b) What is iterative substitution method? Apply the Iterative substitution method to solve the following recurrence relations.

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

Iteration method to expand the recurrence and express it as a summation of terms of n and initial condition.

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

$T(1) = 1 \longrightarrow$ base condition.

$\rightarrow T(n) = 2T(n/2) + n \longrightarrow (1)$

$\boxed{n = n/2}$

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

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

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

sub (2) in (1)

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

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

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

$\boxed{n = n/4}$

Sub in eq (1)

14

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

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

sub (3) in (2)

$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$

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

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

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

$\vdots$

i^{th} term

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

$= 2^i T(1) + in$

$\frac{n}{2^i} = 1 \qquad n = 2^i$

take $\log_2$ both sides

$\log_2 n = i \log_2 2.$

$i = \frac{\log n}{\log 2} = \log_2 n$

$T(n) = 2^i T(1) + i.n$

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

$T(n) = n + n\log n \qquad \therefore T(n) = O(n\log n)$

8	**Demonstrate Towers of Hanoi with algorithm and example.**	**[L3][CO1]**	**[12M]**

8. Demonstrate Towers Of Hanoi with algorithm and example. (12M)

The tower of Hanoi is very well known recursive problem also known as Tower of Lucas
The problem is based on 3 pegs (Source, auxiliary, & destination) and n disks.

Tower of Hanoi is the problem of shifting all n disks from source peg to destination peg using auxiliary peg with the following constraints.

→ Only one disk can be moved at a time.
→ A larger disk cannot be placed on a smaller disk.
→ The initial and final Configuration of the disks are shown in following steps below.

step-01: move n-1 disks from Source to auxiliary.
step-02: move nth disk from source to destination
step-03: move n-1 disks from auxilaary to destination

Algorithm:

```
Start
 Procedure TOH (disk, Source, dest, aux)
 if disk==1 then
     move disk from Source to dest
 else TOH (disk-1, aux, dest, Source)
     moveDisk (Source to dest)
     TOH (disk-1, aux, dest, source)
     ENDIF
     END Procedure.
 STOP
```

Complexity Analysis Of Towers Of Hanoi:

* Moving n-1 disks from Source to aux means the first peg to the second peg. This can be done in $T(n-1)$ steps

* Moving the nth disk from Source to dest means a larger disk from the first peg to the third peg will require 1 step.

* Moving n-1 disks from aux to dest means the Second peg to the third peg will require again $T(n-1)$ steps.

So total time taken $T(n) = T(n-1) + 1 + T(n-1)$

Our equation will be.

$$T(n) = 2T(n-1)+1 \rightarrow ①$$

after putting n=n-1 in eq ① It will become

$$T(n-1) = 2T(n-2)+1 - ②$$

after putting n=n-2 in eq ①. It will become

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

Put the value of T(n-2) in the equation ② with help of eq ③

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

Put the value of T(n-1) in equation ① with help of eq ④

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

After Generalization

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

$k=3$

$$T(n) = 2^K T(n-K) + 2^{K-1} + 2^{K-2} + \cdots$$

From our base Condition $T(1)=1$

$$n-k=1$$

$$K = n+1$$

Now put $K=n-1$ in above equation

$$T(n) = 2^{(n-1)} T(n-(n-1) + 2^{K-1} + 2^{K-2} + \cdots + 2^2 + 2^1 + 2^0$$

$$= 2^{(K)(1)} + 2^{(K-1)} + 2^{(K-2)} + \cdots + 2^2 + 2^1 + 2^0$$

Time Complexity:

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

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

9	a)	**Define disjoint set. Explain any four types of disjoint sets operations with Examples.**	[L2][CO1]	[06M]

Operations with example.

A pair of sets which does not have any common element are called disjoint sets.

→ Consider a set $S=\{1,2,3,4 \ldots 10\}$ these elements can be partitioned into three disjoint sets

$$A=\{1,2,3,6\} \quad B=\{4,5,7\} \quad C=\{8,9,10\}$$

→ Each of the set can be represented as a tree hence these threes corresponding to the sets are

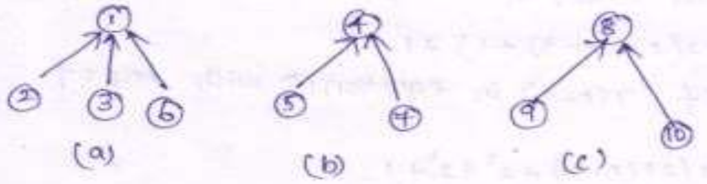


fig: Representation of sets A, B, C

→ These sets can be represented by storing every element of the set in the same array.

→ The ith element of this array represents the tree node that contains element i. This array elements gives the parent pointer of the corresponding tree node.

16

i	1	2	3	4	5	6	7	8	9	10
Parent	-1	1	1	-1	4	1	4	-1	8	8

Operations:

1. Member(a, S):

Determine whether a is a member of S, if so print "Yes" otherwise print "No".

Eg: $S=\{1,2,3\}$ $a=2$

member (2, S), it means $2 \in S$ or not

∴ The function print "yes"

2. INSERT (a, S)

Replaces set S by $S \cup \{a\}$

Eg: INSERT (4, S) $S=\{1,2,3\}$

$S = S \cup \{4\} = \{1,2,3,4\}$

3. DELETE (a, S)

Replaces set S by $S-\{a\}$

Eg: $S=S-\{a\}$

$a=4$, $S=\{1,2,3,4\}$

$S=S-\{4\}$

$=\{1,2,3,4\}-\{4\}$

$=\{1,2,3\}$

4. UNION (S_1, S_2, S_3)

calculates $S_3 = S_1 \cup S_2$

We will assume that, S_1 & S_2 and disjoint

eg: $S_1=\{1,2\}$ $S_2=\{4,6\}$

$S_3=S_1 \cup S_2 = \{1,2\} \cup \{4,6\}$

$= \{1,2,4,6\}$

5. FIND(a): Prints the name of the set of which a is currently a member

$S_1=\{1,2\}$, $S_2=\{4,6\}$ $a=4$

FIND(4) this returns S_2

$\therefore$ Since $4 \in S_2$

b)	**Explain the weighted union algorithm for union algorithm with example.**	**[L2][CO1]**	**[06M]**

If the number of nodes in tree 'i' is less than the number in tree j then make j the parent of i otherwise make 'i' the parent of 'j'.

* Both the arguments of UNION must be roots.

Algorithms

Algorithm weighted UNION(i,j)

// Union sets with roots i & j i≠j using the weighted rule p[i] = -count[i], p[j] = = count[j]

```
{
    temp = p[i]+p[j]
    if (p[i] > p[j]) then
    {
        // i has fewer nodes
        p[i] = j;
        p[j] = temp;
    }
```

else

{ $P[j]$ has fewer (or) equal nodes.

$P[j] = i;$

$P[i] = temp;$

}

}

∴ Initially array Representation

i	1	2	3	4	...	n
p	-1	-1	-1	-1		-1

Eg: ① ② ③ ... ⓝ are the disjoint sets and apply Union Operation using weighting rule for these sets.

Sol: step1: perform UNION(1,2)

→ By tracing algorithm, we plot the tree, for UNION(1,2)

→ $P[i] > P[j]$ ∴ $temp = P[i] + P[j]$

$P[1] > P[2]$ $i = 1 \quad j = 2$

$-1 > -1$ $temp = P[1] + P[2] = -1 + -1$

· Condition False $= -2$

→ Goto else part

$P[j] = i \quad , P[2] = 1$

$P[i] = temp \quad P[1] = -2$

By tabular Representation.

temp	i	j
-1	1	2
-2		

Tree representation

Array representation of UNION(1,2)

i	1	2	3	4	--	n
P	-2	+1	-1	-1		-1

Step2: perform UNION(1,3)

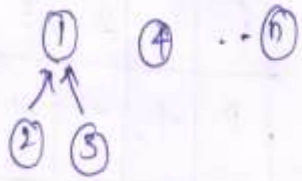
→ By tracing out

→ temp = P[1] + P[3]

= -2 + -1

temp = -3

By ~~array~~ Tree representation of UNION(1,3)



→ P[i] > P[j]

P[1] > P[3]

-2 > -1

Condition False

→ Goto else part

P[j] = i P[3] = 1

P[i] = temp P[1] = -3

By array Representation of sets

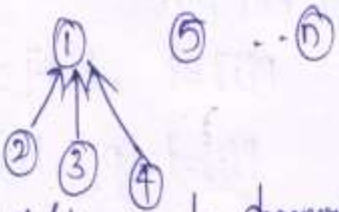
i	1	2	3	4	n
P	-3	+1	+1	-1	-1

Step3: Perform UNION (1,4).

Array Representation

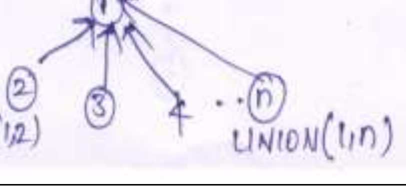
i	1	2	3	4	.. n
P	-4	1	1	1	-1

Tree Representation



∴ While performing this union weighting rule degenerated tree will not appear & Complexity also reduced.

∴ It will perform upto UNION (1,n) Operations



10	a)	Explain the collapsing rule for Find algorithm with example.	[L2][CO1]	[06M]

10(a) Explain the Collapsing rule for Find algorithm with example. 18

→ Replace the search path to the root. this is Called a collapsing find operation.

→ In this function

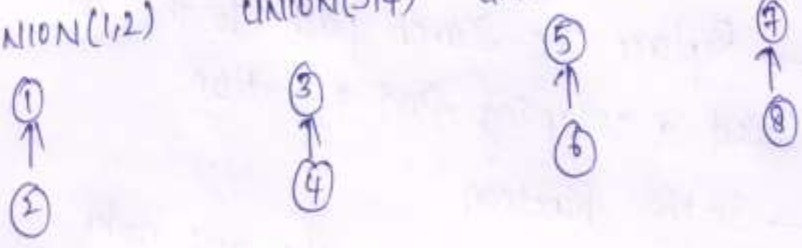
1. Determines the root nodes
2. Second pass is made up the chain from the initial node to the root.

Algorithm:

```
Algorithm collapsing Find(i)
// Find the root of the containing elements 'i' use
// the collapsing rule to collapse all nodes from
// i to the root
{ r:=i
  while (P[r] > 0) do
    r=P[r] // Find the root
  while (i≠r) do //collapsing
  { s=P[i]
    P[i]=r
    i=s
  }
  return s
}
```

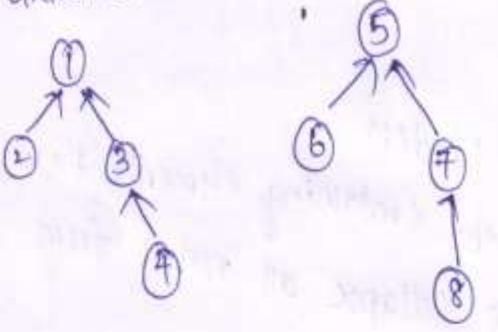
Eg: 1 2 3 4 5 6 7 8 apply union first & perform find using collapsing rule.

Sol: UNION(1,2) UNION(3,4) UNION(5,6) UNION(7,8)



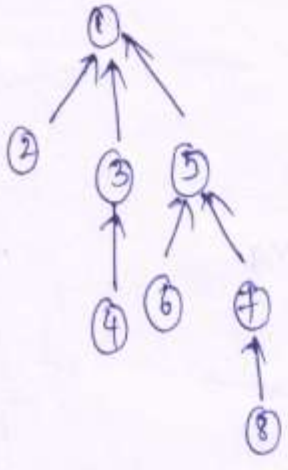
∴ It is of height -2

UNION(1,3) UNION(5,7)



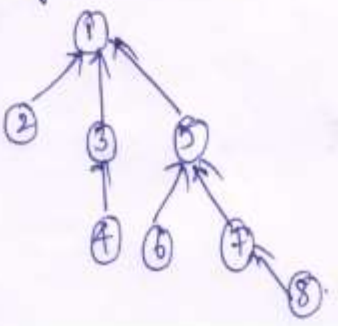
∴ It is of height -3

UNION(1,5)



Height of initial nodes 1 2 3 4 5 6.

-Apply collapsing find algorithms for set



step 1: Perform find (8)

Array Representation

i	1	2	3	4	5	6	7	8
P	-3	1	1	3	1	5	5	7

1) $r=i \rightarrow r=8$

   while $(P[8]>0)$

   $7>0$ (True)

$\rightarrow r=P[r] \rightarrow r=P[8]=7$

2) Traverse the element root until it become false

   Now $r=7$

   $\rightarrow P[7]>0$

   $5>0$ True

   $\rightarrow r=P[7]$

   $\boxed{r=5}$

3) Again check condition

   $\rightarrow r=5$

   $P[5]>0$ $1>0$ True

   $\rightarrow r=P[5]=1$

4) Again check condition.

   $\rightarrow r=1$

   $P[1]>0 \Rightarrow -3>0$ (False)

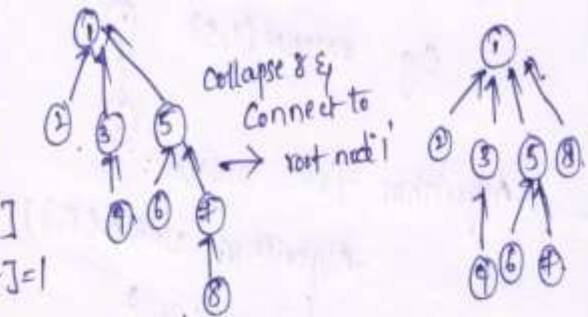
∴ There fore my root element is '1'

→ Then problem collapsing rule in algorithm

1) while $(i \neq r)$ do

   $\rightarrow (8 \neq 1)$ False

2) Enter into the loop



$S=P[i] \Rightarrow S=P[8]$

$P[i]=r \Rightarrow P[8]=1$

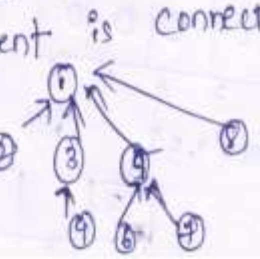
$i=S \Rightarrow i=7$ // next finding element

step 2: Next Operation FIND(7) and performs continously until it reach to Find(5)

∴ Therefore FIND(7) = 1

∴ 7 element is connected to root node 1 after

Collapsion



⇒ collapsed '7' element is Connected to root node '1'

b)	Determine steps of Union and Find algorithms with example.	[L5][CO1]	[06M]

b) Determine ... example

UNION[i,j] : Union[i,j] , it means the elements of set i and elements of set j are combined.

* If we want to represent UNION operation in the form of a tree.

* Then UNION(i,j) i is the parent , j is the child

Eg: UNION (1,3)

UNION(2,5)

Algorithm for UNION:

Algorithm Union(i,j)
{ P[i] = j;
}

Eg: step1: Initially parent array contains zeros

1	2	3	4	5	6
0	0	0	0	0	0

step2: perform UNION (1,3) i=1 j=3 P[1]=3

1	2	3	4	5	6
3					

step3: perform UNION(2,5)

i=2 , j=5 P[2]=5

20

	1	2	3	4	5	6
P	3	5				

Step4 : perform UNION(1,2)

i=1 j=2

P[1]=2

1	2	3	4	5	6
2					

Time Complexity:

Since the time taken for UNION is constant. in time O(n) all the (n-1) UNIONS can be processed

FIND Operation:

→ Find(i) implies that it finds the root node of ith node in other words it returns the name of the set 'i'

eg: Union(1,3)

find(1) = 1 since its parent is '1'

find(3)=1

Algorithm FIND(i):

Algorithm FIND(i)
{ while (P[i]>0) do
  i:=P[i];
  return i;
}

Eg: consider the tree sets and perform find operation

Sol: Initially array representation is

P	0	1	1	0	2	0
	1	2	3	4	5	6

Step1: perform FIND(5)

i=5 while P[5] ≥ 0 do (True)

i = P[i] => i: = P[5] = 2 return 2

∴ Find (5) = 2 [since it is root node of parent of 5]

Step2: i=2 while P[2] ≥ 0 (T)

i: = P[i] = i: = P[1] i = 1

return P

Step3: i=1

while P[1] ≥ 0 T

i = P[i] => i = P[1]

i=0

return = 0

finally 1 is root node of 5

$\boxed{\text{FIND}(5) = 1}$

Course Code: 20CS0523 **R20**

UNIT –II

BASIC TRAVERSAL AND SEARCH TECHNIQUES, DIVIDE AND CONQUER

1	**Explain techniques of binary trees with suitable example.**	[L2][CO2]	[12M]

Binary Tree:-

→ A binary tree is a finite collection of elements or it can be said it is made up of nodes. Where each node contains the left pointer, right pointer and a data element.

→ The root pointer points to the topmost node in the tree. When the binary tree is not empty, so it will have a root element and the remaining elements are partitioned into two binary trees which are called the left pointer and right pointer of a tree.

Traversing in the Binary Tree:-

→ Tree traversal is the process of visiting each node in the tree exactly once. Visiting each node in a graph should be done in a systematic manner.

→ If search result in a visit to all the vertices, it is called a traversal. There are basically three traversal techniques for a binary tree that are,

1. Preorder traversal
2. postorder traversal
3. Inorder Traversal.

1. Preorder Traversal:-

→ To traverse a binary tree in preorder, following operations are carried out:

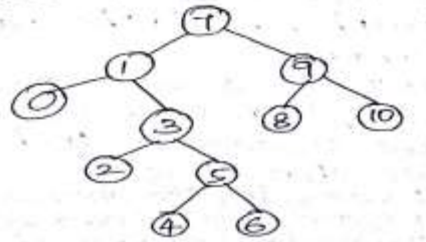
1. Visit the root
2. Traverse the left subtree of root.
3. Traverse the right subtree of root.

→ Preorder traversal is also known as NLR traversal.

Algorithm:-

```
Algorithm preorder (t)
/* lchild, data, and rchild */
{
    If t != 0 then
    {
        visit(t);
        Preorder(t→lchild);
        preorder(t→rchild);
    }
}
```

Example:-



The preorder traversal is

7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10

2) Inorder traversal:-

→ To traverse a binary tree in inorder traversal, following operations are carried out:

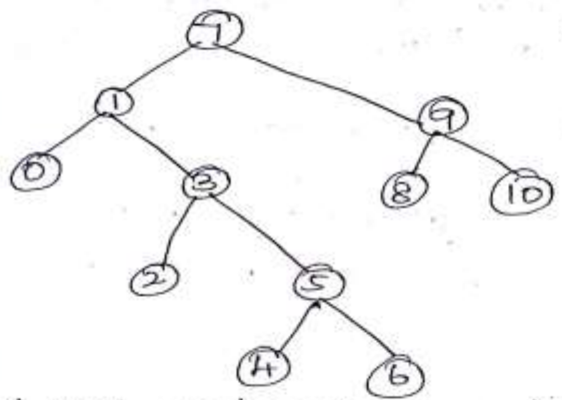
1. Traverse the left most sub tree.
2. Visit the root.
3. Traverse the right most sub tree.

→ Inorder traversal is also known as LNR traversal

Algorithm:-

```
Algorithm inorder(t)
{
    if t!=0 then
    {
        Inorder(t→lchild);
        Visit (t);
        Inorder (t→rchild);
    }
}
```

Example:-



The inorder traversal is :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

3) post order traversal:-

→ To traverse a binary tree in postorder traversal, following operations are carried out:

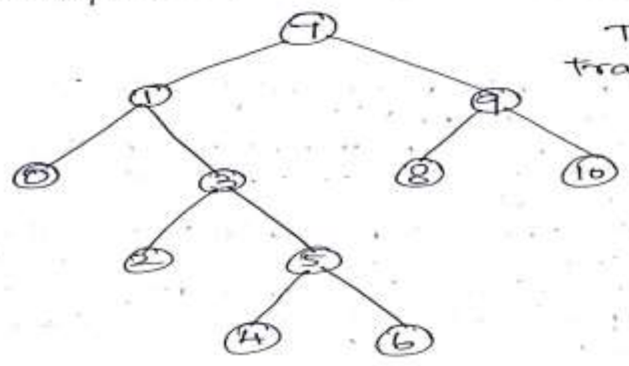
1. Traverse the left subtree of root.
2. Traverse the right subtree of root.
3. Visit the root.

→ postorder traversal is also known as LRN traversal.

Algorithm:-

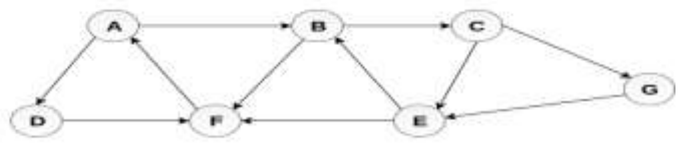
```
Algorithm postorder(t)
{
    if t!=0 then
    {
        postorder(t→lchild)
        postorder(t→rchild)
```

visit(t);
}
}
Example:-



The postorder traversal is
0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7

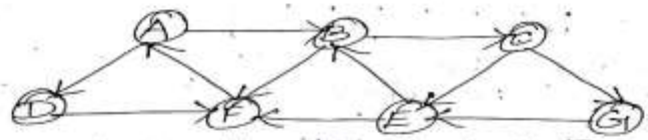
2 Elaborate BFS algorithm and trace out minimum path for BFS for the following example. **[L6][CO2]** **[12M]**



→ Breadth first search is a graph traversal algorithm that starts Traversing the graph from root node and explores all the neighboring nodes.

→ Then it selects the nearest node and explores all the unexplored nodes.
→ The algorithm follows the same process for each of the nearest node until it finds the goal.
Steps for BFS:-
step 1 :- Mention all adjacency list of each node in graph 'G'.
step 2 :- Enqueue the starting node in the source vertex 's' and set it in waiting state.
step 3 :- Repeat the steps and place the adjacency vertex of soure vertex in the queue 'Q' and dequeued waiting state of source vertex in 'Q'.
step 4 :- Repeat the steps 2 and 3 until Queue is empty.
step 5 :- Exit.
Algorithm for BFS:-
Algorithm BFS(G, S)
{
Let Q be queue
Q. enqueue(S)
// Inserting S in queue until all its neighbour vertices are marked.
mark S as visited

while (Q is not empty)
// Remove that vertex from queue, whose neighbor will be visited now.
V = Q.dequeue()
// processing all the neighbours of 'V' for all neighbours W of 'V' in graph 'G'
if W is not visited.
Q.enqueue (W)
mark 'W' as visited.
}



step 1:- Select all adjacency vertices of node and place it in adjacency list.

A → B, D G → E F → A
B → C, F D → F
C → E, G E → B, F

step 2:- The algorithm uses two queues namely Queue 1 and Queue 2. Queue 1 holds the all nodes that are to be processed. While Queue 2 holds all the nodes that are processed and deleted from Queue 1.
Lets start examine the graph from node 'A'.

Step 3:- Select the vertex 'A' from the graph.
Add 'A' to the Queue 1 and NULL to Queue 2.

Queue 1	A					
Queue 2	NULL					

step 4:- Delete the node 'A' (i.e dequeue) from Queue 1 and place it in Queue 2) Insert all its neighbours into Queue 1.

Queue 1	B	D				
Queue 2	A					

step 5:- Delete the node 'B' from Queue 1 and insert all neighbours of B. Insert node 'B' into Queue 2.

Queue 1	D	C	F			
Queue 2	A	B				

step 6:- Delete the node 'D' from Queue 1 and insert all neighbours of D. Insert 'D' into Queue 2.

Queue 1	C	F				
Queue 2	A	B	D			

since 'F' is only neighbour of 'D' and it is already inserted, we will not insert it again.

(5)

Step 7:- Delete the node 'C' from Queue 1 and insert all its neighbour nodes. Add node 'C' into Queue 2.

Queue 1 | F | E | G | | | | |

Queue 2 | A | B | D | C | | | |

Step 8:- Delete the node 'F' from Queue 1 and insert all its neighbours. If already in Queue 1, then we won't add it again. Add node 'F' to Queue 2

Queue 1 | E | G | | | | | |

Queue 2 | A | B | D | C | F | |

Step 9:- Delete node 'E' from Queue 1 and its neighbour nodes already visited. Then add node 'E' to Queue 2.

Queue 1 | G | | | | | | |

Queue 2 | A | B | D | C | F | E | |

Step 10:- Delete node 'G' from Queue 1. Queue 1 is empty and no more nodes in graph to be traversed. Add 'G' to Queue 2.

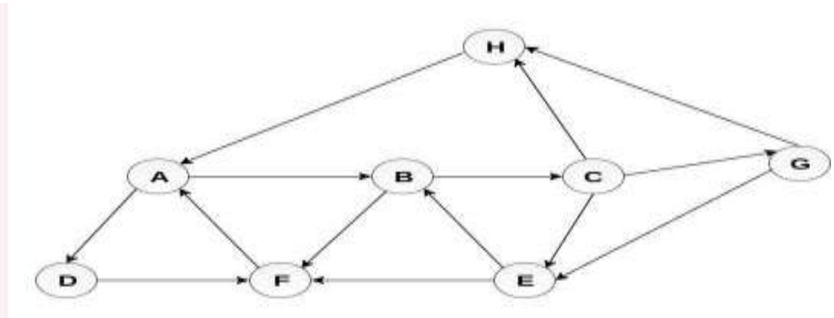
Queue 1 | | | | | | | |

Queue 2 | A | B | D | C | F | E | G |

∴ BFS Traversal order is
A → B → D → C → F → E → G

∴ Time complexity of BFS is O(V+E)

3 **Explain DFS algorithm and trace out minimum path for DFS for the following example.** **[L5][CO2]** **[12M]**



→ DFS algorithm is a recursive algorithm that uses the idea of back tracking.
→ Here, backtracking means, when you are moving forward and there are no more nodes along the current path, you have to move backwards on the same path to find the traverse.
→ DFS uses stack datastructure.

Steps for DFS:-
Step 1:- Pick a starting node and push all its neighbour nodes into stack.
Step 2:- Pop a node from stack to select the next node to visit and push all its adjacent nodes into stack.
Step 3:- Repeat this process until stack is empty.

Algorithm for DFS :-

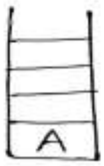
Algorithm DFS (G, s)
{
let s be stack
s.push(s)
mark s as visited.
while (s is not empty)
V = s.top()
s.pop()
if W is not visited
s.push(w)
mark 'w' as visited
}

step 1 :- Adjacency lists for graph

A → B, D
B → C, F
C → E, G, H
D → F
E → F
G → E, H
F → B, F
H → A

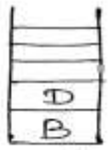
step 2 :- select the source vertex 'H' from the graph, push 'H' onto stack.

Step 3 :- pop the top element of stack i.e, 'H' and push all the neighbours of 'H into stack that are in read state.



Print : H

Step 4 :- pop the top element of stack, i.e, 'A' Print it and push all neighbours of A into stack.



Print : A

Step 5 :- pop the top element of stack 'D', Print it and push all neighbours of 'D'.

Print : D

Step 6 :- pop the top element of stack 'F', Print it and push all neighbours of 'F' into stack.

Print : F

<table>
<tr><td></td><td>step 7 :- pop the top element of stack, i.e 'B' and push the neighbours.


Print: B

step 8 :- pop the top element in stack 'c' and push all neighbours.

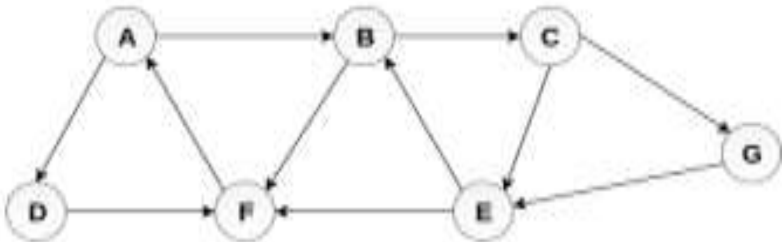

Print : C

step 9 :- pop the top element i.e 'G' and push all its neighbours and print 'G'.


Print : G

step 10 :- pop the top element i.e 'E' and push all its neighbours and prin 'E'.

Print : E

→ The traversing order is
H → A → D → F → B → C → G → E
→ Time complexity of DFS is O(V+E)</td><td></td><td></td></tr>
<tr><td>4</td><td>What is connected component and spanning tree? Draw the spanning tree for the following graph using DFS algorithm

</td><td>[L2][CO2]</td><td>[12M]</td></tr>
</table>

Connected Components:-

→ A connected component is a subgraph in which any two vertices are connected to each other by paths and which is connected to no additional vertices of super graph.

Spanning Trees:-

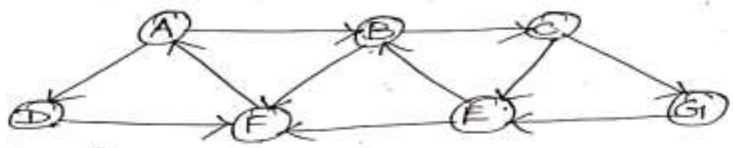
→ A spanning tree can be defined as the sub graph of an undirected graph. It includes all the vertices along with the least possible number of edges.

→ If any vertex is missed, it is not a spanning tree.

→ A spanning tree is a subset of graph that doesnot have cycles and it also cannot be disconnected.

→ A spanning tree consists of n-1 edges where n is the number of vertices.

Draw the spanning tree for the following graph using DFS algorithm.



steps for DFS:-

step 1:- pick a starting node and push all its adjacent nodes into stack.

step 2:- pop a node from stack to select the next node to visit and push all its adjacent nodes into stack.

step 3:- Repeat this process until stack is empty.

step 1:- Adjacency list for graph

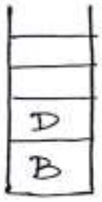
A → B, D D → F G → E

B → C, F E → B, F

C → E, G F → A

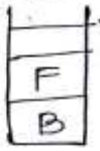
step 2:- select the source vertex 'A' from the graph, push 'A' onto stack.

Step 3:- pop the top element of stack i.e 'A' and push all the neighbours of 'A' into stack that are in read state.



Print : A

Step 4:- pop the top element of the stack i.e 'D' print it and push all neighbours of D into stack.



print : D

Step 5:- pop the top element of the stack i.e 'F' print it and push all neighbours of F into stack.

print : F

[∵ neighbour of 'F' is A already printed, so no need of push into stack].

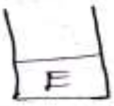
Step 6:- pop the top element of stack i.e 'B', print it and push all neighbours of 'B' into stack.

print : B

Step 7:- pop the top element of stack i.e 'C', print it and push all the neighbours of 'C' into stack.

print : C

Step 8:- pop the top element of stack i.e 'G', print it and push all neighbours of 'G' into stack.



print : G

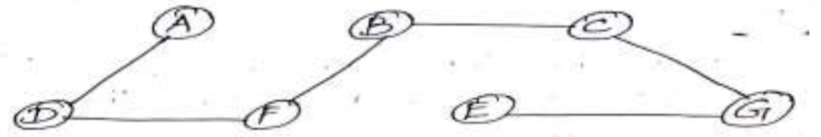
Step 9:- pop the top element of stack i.e 'E', print it and push all neighbours of 'E' onto stack.

print : E

empty

∴ Hence, the stack now becomes empty and all the nodes of graph have been traversed.

→ The spanning tree for the above Graph is



→ The number of vertices : n = 7
→ The number of edges : (n − 1) = 6

5 a) **Compare between BFS and DFS techniques.** [L4][CO2] [04M]

S.NO	BFS	DFS
1.	BFS stands for Breadth first search.	DFS stands for Depth first search.
2.	BFS uses Queue data structure for finding the shortest path.	DFS uses stack data structure for finding the shortest path.
3.	It works on the concept of FIFO.	It works on the concept of LIFO.
4.	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
5.	Here, siblings are visited before the children.	Here, children are visited before the siblings.
6.	In BFS, there is no concept of Back tracking.	In DFS, algorithm is a recursive algorithm that uses the idea of backtracking.
7.	Time complexity of BFS = O(V+E) where V is vertices and E is edges.	Time complexity of DFS is also O(V+E) where V is vertices and E is edges.
8.	BFS is slow as compared to DFS	DFS is fast as compared to BFS.

5	b)	What is divide and conquer strategy? Write briefly about general method and its algorithm	[L3][CO2]	[08M]

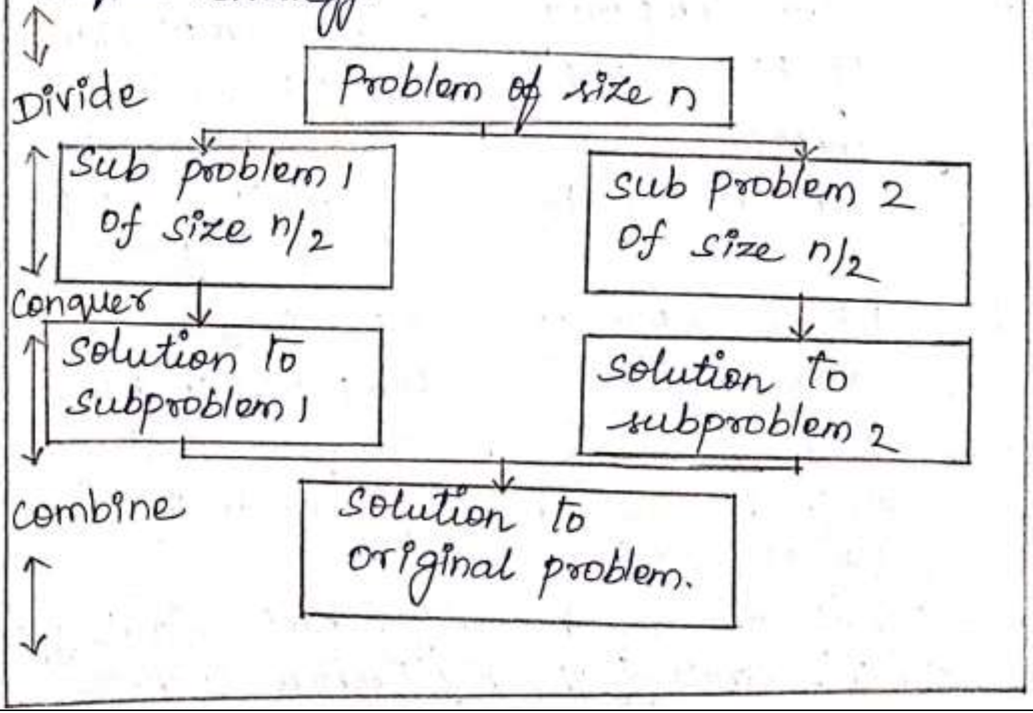
→ It is one of the algorithmic strategy.
→ In this strategy the big Problem is broke down into smaller sub problems and solution to these sub problems is obtained.

→ In divide and conquer method, a given problem is,

1. Divide :- Divided into smaller sub problems.
2. Conquer :- These subproblems are solved independently.
3. combine :- combining all the solutions of sub problems into a solution of the whole.

→ If the subproblems are large enough then divide and conquer is reapplied.

→ The generated sub problems are usually of same type as the original problem. Hence recursive algorithms are used in divide and conquer strategy.



Control abstraction of Divide and conquer:-

→ A control abstraction for divide and conquer is as given below, using control abstraction a flow of control of a procedure is given.

→ Given a problem 'p' of size $n = 2^K$.

→ Algorithm for DAC(P)

- If 'n' is so small, solve it.
- else divide 'p' into two subproblems P_1 & P_2.
- DAC(P_1);
- DAC(P_2); // solve each subproblem recursively.
- combine solutions of subproblems P_1 & P_2.

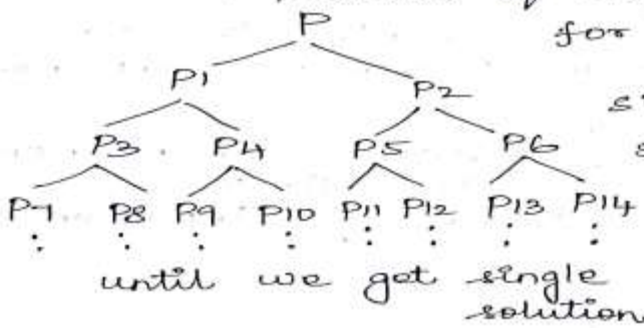
Algorithm:-

```
Algorithm DAC(P)
{
   if small(P) then    //return solution
   return S(P).              of P.
 else
 {
    // If large
    K = Divide(P)  // obtain P1, P2, ..., Pk
                         subproblems.
   //combine the subproblem by using DAC
   return combine(DAC(P1), DAC(P2),
                  ... DAC(PK)).
  }
}
```

Example:-

Tree of recursive calls:-

Problem of size n



for problem p'n' size $n = 2^K$

size of $n/2 = 2^{K-1}$

size of $n/4 = 2^{K-2}$

$n/8 = 2^{K-3}$

⋮

2^0

∴ The height of the tree is reduced from 2^K to 2^{K-1} ... to 2^0.

→ Therefore the computing time of above algorithm procedure of divide and conquer is given by recurrence relation

$$T(n) = \begin{cases} g(n) & \text{If } n \text{ is small} \\ T(n_1) + T(n_2) + \dots T(n_r) + f(n) & \text{If } n \text{ is large.} \end{cases}$$

6	**What is divide and conquer strategy? Explain the working strategy of Binary Search and find element 60 from the below set by using the above technique: {10, 20, 30, 40, 50, 60, and 70}. Analyze time complexity for binary search.**	[L2][CO2]	[12M]

Divide and conquer :-

→ It is one of the algorithmic strategy.

→ In this strategy the big problem is broke down into smaller problems and solution to these subproblems is obtained.

Working strategy of Binary search.

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

→ An element which is to be searched from the list of elements stored in the array A[0... n-1] is called KEY element.

→ Let A[m] be the mid element of array A.

→ Therefore three conditions that needs to be tested while searching the array using this method.

1. If KEY = A[m] then desired elements is Present in the list.
2. If KEY < A[m] then search the left sub list of mid element.
3. If KEY > A[m] then search the right sub list of mid element.

∴ This can be represented as

A[0] ... A[m-1] .. A[m] .. A[m+1] .. A[n-1]

Search here if KEY < A[m] — KEY? — Search here if KEY > A[m]

Consider a list of elements sorted in array 'A' as

10	20	30	40	50	60	70

The searching element is '60'.

→ For searching, we need to test three conditions.

Case 1 :-

→ Search the element is present in the middle location. To test that element, we apply the formula.

$$m = \frac{(low + high)}{2}$$

∴ condition is A[m] == KEY.

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

low ↑ ... ↑ hight

$$m = \frac{0+6}{2} = \frac{6}{2} = 3$$

$\boxed{m=3}$ mid location is '3'.

∴ A[m] == KEY
A[3] == 60
40 == 60

→ Element is not present in the middle location.

→ check for second condition

case 2:-

→ If KEY < A[m] then search left sub list.

60 < 40

→ The element is not present in the left sub list.

case 3:-

KEY > A[m]
60 > 40

∴ search for right sub list.

50	60	70
4	5	6

low ↑ ... ↑ high

→ Find out the middle and check if the element is present in the list (or) not.

$$m = \frac{low+high}{2} = \frac{4+6}{2} = \frac{10}{2} = 5$$

$\boxed{m=5}$

50	60	70

KEY = A[m]
60 = A[5]
60 = 60

→ Therefore element '60' is present in the location A[5].

→ It is searched in right sub list of the middle element.

Algorithm for Binary search:-

```
Algorithm Binsearch (A[0...n-1], KEY)
{
    low ← 0
    high ← n-1
    while (low < high) do
    {
      m ← (low + high)/2
      if (KEY = A[m]) then
      return m;
```

```
else if(KEY < A[m])
  high ← m-1
else
  low ← m+1
}
return -1
}
```

Time complexity analysis of Binary Search:-

1. For best case, If searching element is in the middle portion.
   Time complexity is O(1) [only 1 comparison occurs].
2. For Average case, In this case the searching element is not in first (or) last position.
   Time complexity is O(log n).
3. For worst case:-
   In this case the searching element is in the first (or) last position.
   Time complexity is O(log n).

7	**Summarize an algorithm for quick sort. Provide a complete analysis of quick sort for given set of numbers 12, 33, 23, 43, 44, 55, 64, 77and 76.**	**[L2][CO2]**	**[12M]**

→ Quick sort is an divide and conquer algorithm.

steps:-

1. Select a pivot element. pivot means target. Any element can be Taken as pivot.
2. Partition operation :- partition means divide.
   → partition operation divide the array into '2' subarray [left & right subarray].
   → Partition operation places the point element at proper position.
   → That means all element before pivot is smaller and all element after pivot is greater.
3. Recursively, apply quicksort to sub arrays [left subarray & Right sub array].

Rules:-

1. The value of i is incremented till a[i] ≤ pivot.
2. The value of j is decremented till a[j] > pivot, this process is repeated until i<j.
3. If a[i] > pivot and a[j] < pivot and also if i<j then swap a[i] & a[j].
4. If i>j then swap a[j] and a[pivot]

∴ once the correct location for pivot is found, then partition array into left sub array contains all elements less than pivot & right sub array contains all the elements greater than the pivot.

Algorithm for Quick sort:-

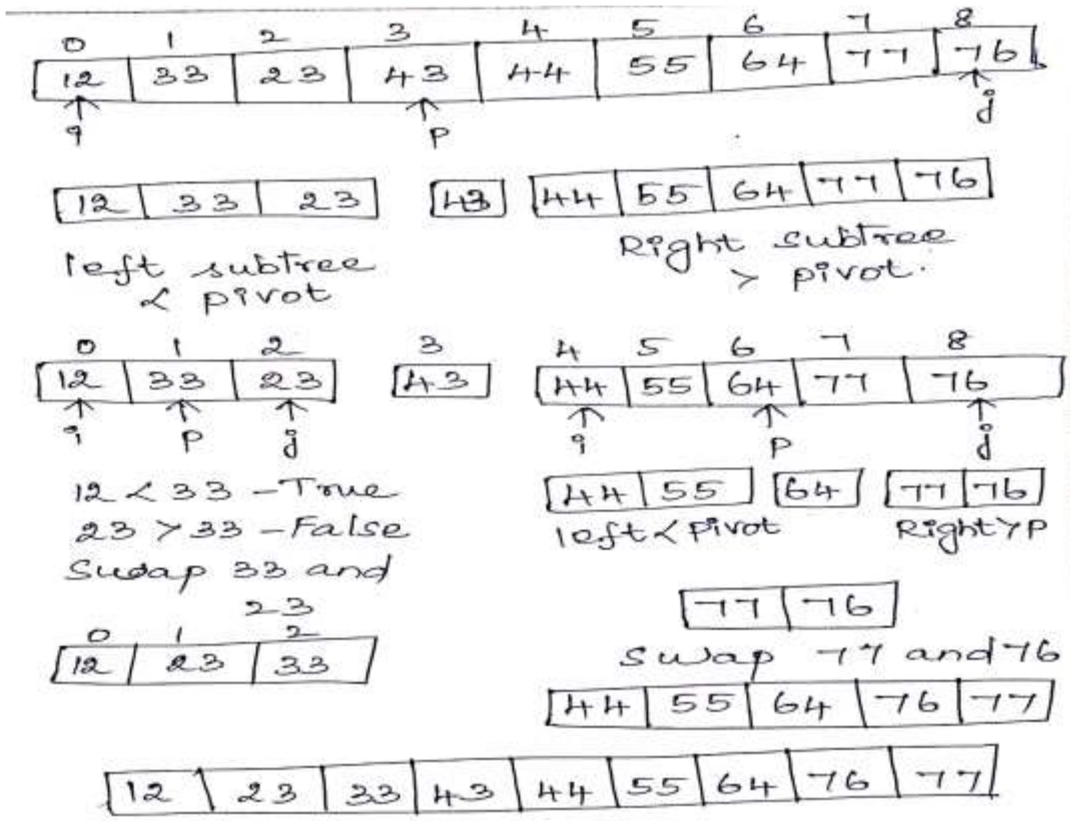
```
Algorithm qsort(A[ ], left, right)
{
  if(left < right) then
  {
      pivot ← left
      i = left + 1
      j = right - 1
    while(i<j)
    {
 while(A[pivot] >= A[i])
      i = i+1
  while(A[pivot] < A[j])
      j = j-1
    if (i<j)
    {
      temp ← A[i]
      A[i] ← A[j]
      A[j] ← temp
    }
 }
  temp = A[pivot]
  A[pivot] ← A[j]
    A[j] ← temp
  qsort(A, left, j-1)
  qsort(A, j+1, right)
 }
}
```

12, 33, 23, 43, 44, 55, 64, 77 and 76

i = 12

j = 76

pivot = 43



8	**Analyze the working strategy of merge sort and illustrate the process of merge sort algorithm for the given data: 43, 32, 22, 78, 63, 57, 91 and 13.**	**[L4][CO2]**	**[12M]**

→ The merge sort is a sorting algorithm that uses the divide and conquer strategy.

→ In this method division is dynamically carried out.

→ Merge sort on an input array with n elements consists of three steps.

Divide :- Partition array into two sublists S_1 & S_2 with $n/2$ elements each.

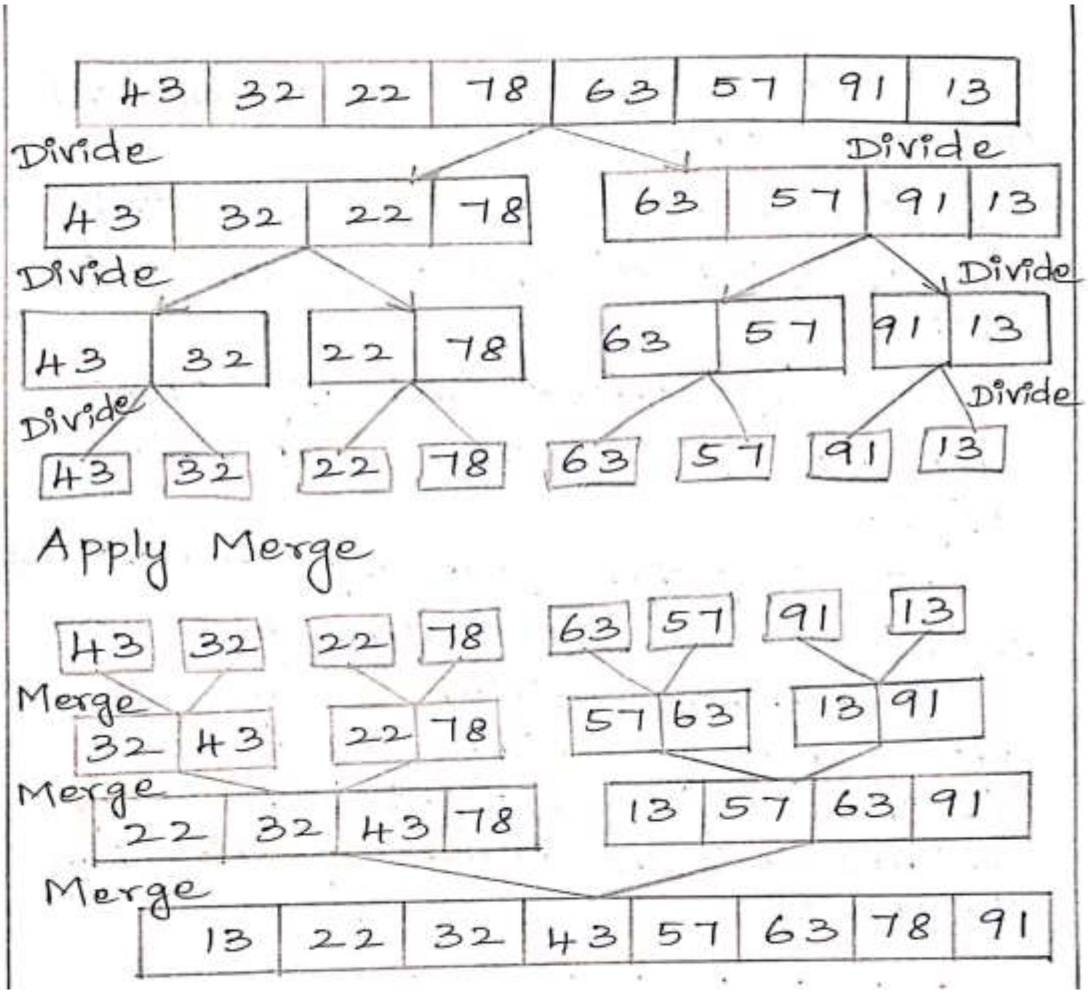
Conquer :- Then sort sublists S_1 & sublist S_2.

Combine :- Merge S_1 & S_2 into a unique sorted group.

Algorithm :-

```
Algorithm divide (low, high)
{
    low ← 0
    high ← n-1
    if (low < high) then
    {
```

```
 mid <- (low+high)/2
 divide (low, mid)
 divide (mid+1, high)
 merge (low, mid, high)
 }
}
Algorithm merge (low, mid, high)
{
 while (i<=mid and j<=high) do
 {
  if (A[i] < A[j]) then
  {
   B[k] = A[i]
   k++
   i++
  }
 else
  {
   B[k] = A[j]
   k++
   j++
  }
 }
}
```



9	a)	Sort the records with the following index values in the ascending order using quick sort algorithm. 9, 7, 5, 11, 12, 2, 14, 3, 10, 6.	[L2][CO2]	[6M]

Rules:-

1. The value of i is incremented till a[i] ≤ pivot.
2. The value of j is decremented till a[j] > pivot, this process is repeated until i < j.
3. If a[i] > pivot and a[j] < pivot and also if i < j then swap a[i] & a[j].
4. If i > j then swap a[j] and a[pivot]

∴ once the correct location for pivot is found, then partition array into left subarray contains all elements less than pivot & right sub array contains all the elements greater than the pivot.

9	7	5	11	12	2	14	3	10	6

9	7	5	11	12	2	14	3	10	6
↑ p	↑ i								↑ j

i < pivot < j

(i) 7 < pivot – T   (j) 6 < pivot – F
5 < pivot – T   Swap 6 and 11
11 > pivot – F

9	7	5	6	12	2	14	3	10	11
↑			↑						↑
p			i						j

(i) 6 < pivot – T   (j) 11 > pivot – T
12 > pivot – F   10 > pivot – T
3 < pivot – F
Swap 12 and 3

9	7	5	6	3	2	14	12	10	11
↑				↑			↑		
p				i			j		

(i) 3 < pivot – T   (j) 12 > pivot – T
2 < pivot – T   14 > pivot – T
14 > pivot – F   2 < pivot – F

Swap 2 and 9

2	7	5	6	3	9	14	12	10	11
left < p					↑ p	~~lef~~ Right > p			

2	7	5	6	3	9	14	12	10	11
↑	right > p					↑	↑		↑
p						p	i		j

2 | 7 | 5 | 6 | 3 | 9 | 14 | 12 | 10 | 11

p i j p i j

i) 5<p-T, 6<p-T, 3<p-T

j) 3>p-F

Swap 7 and 3

i) 12<14 -T, 10<14 -T, 11<14 -T

j) 11>14 -F

swap 11 & 14

2 | 3 | 5 | 6 | 7 | 9 | 11 | 12 | 10 | 14

left < pivot p

2 | 3 | 5 | 6 | 7 | 9 | 11 | 12 | 10 | 14

p i j

i) 12>p-F

j) 10<p-F

swap 10 and 12

2 | 3 | 5 | 6 | 7 | 9 | 11 | 10 | 12 | 14

p i j

i) 10<p-T, 12>p-F

j) 12>pivot -T, 10<p-F

swap 10 & 11

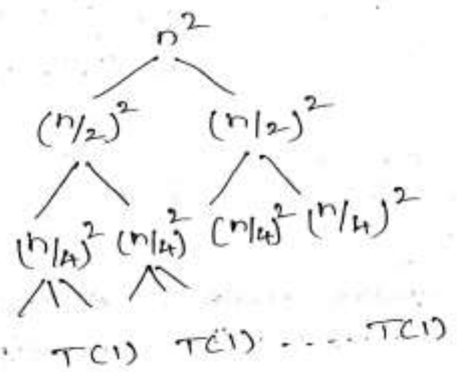
2 | 3 | 5 | 6 | 7 | 9 | 10 | 11 | 12 | 14

2	3	5	6	7	9	10	11	12	14

9	b)	Analyze the time complexity of merge sort using recurrence relation	[L2][CO2]	[6M]

Let $T(n)$ denote the worst case running time of mergesort on an array of n elements. We have

$$T(n) = c_1 + T(n/2) + T(n/2) + c_2 n$$
$$= 2T(n/2) + (c_1 + c_2 n)$$
$$T(n) = 2T(n/2) + \theta(n)$$
$$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 2T(n/2) + \theta(n) & \text{if } n>1 \end{cases}$$



n^2

$8(n/2)^2 = 2n^2$

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

$8^{\log n} T(1)$

$$T(n) = 8T(n/2) + n^2$$
$$T(n) = n^2 + 8T(n/2)$$
$$= n^2 + 8\left(8T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^2\right)$$
$$= n^2 + 8^2 T\left(\frac{n}{2^2}\right) + 8\left(\frac{n^2}{4}\right)$$
$$= n^2 + 2n^2 + 8^2 T\left(\frac{n}{2^2}\right)$$
$$= n^2 + 2n^2 + 8^2\left(8T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2\right)$$
$$= n^2 + 2n^2 + 8^3 T\left(\frac{n}{2^3}\right) + 8^2\left(\frac{n^2}{4^2}\right))$$
$$= n^2 + 2n^2 + 2^2 n^2 + 8^3 T\left(\frac{n}{2^3}\right)$$
$$\vdots$$
$$= n^2 + 2n^2 + 2^2 n^2 + 2^3 n^2 + 2^4 n^2 + \cdots + 2^{\log n - 1} n^2 + 8^{\log n}$$
$$T(n) = n^2 \cdot \theta(n) + n^3$$
$$T(n) = \theta(n^3)$$

$\left[\because \sum_{k=0}^{\log n - 1} 2^k n^2 + 8^{\log n}\right.$

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

$(2^3)^{\log n} = (2^{\log n})^3$

$= n^3.$

10	**Explain the Strassen's algorithm for matrix multiplication and analyze time complexity.**	[L5][CO2]	[12M]

Problem statement:-

→ Given two square matrices A & B of size $n \times n$ each, find their multiplication matrix.

→ Here two matrix $A \times B$ & result stored in 'c' matrix.

$$C = A \times B$$

Example:-

$$\underset{\substack{n\times n \\ 2\times 2}}{\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}} = \underset{\substack{n\times n \\ 2\times 2}}{\begin{bmatrix} A_{11} & A_{12} \\ A_{22} & A_{23} \end{bmatrix}} \underset{\substack{n\times n \\ 2\times 2}}{\begin{bmatrix} B_{11} & B_{12} \\ B_{22} & B_{23} \end{bmatrix}}$$

→ Here these 2x2 matrix is divided into size $n \times n$.

→ $A_{11}, A_{12}, A_{22}, A_{23}$ are sub matrices of A and size $n/2 \times n/2$

→ $B_{11}, B_{12}, B_{22}, B_{23}$ are sub matrices of B & size of $n/2 \times n/2$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} (A_{11}B_{11})+(A_{12}B_{22}) & (A_{11}B_{12})+(A_{12}B_{23}) \\ (A_{22}B_{11})+(A_{23}B_{22}) & (A_{22}B_{12})+(A_{23}B_{23}) \end{bmatrix}$$

→ Therefore four formula are derived to do 2x2 multiplication.

$$C_{11} = (A_{11} * B_{11}) + (A_{12} * B_{22})$$
$$C_{12} = (A_{11} * B_{12}) + (A_{12} * B_{23})$$
$$C_{21} = (A_{22} * B_{11}) + (A_{23} * B_{22})$$
$$C_{22} = (A_{22} * B_{12}) + (A_{23} * B_{23})$$

→ It performs

Multiplication : 8

Addition : 4

→ Addition of 2 matrices takes $O(n^2)$ time.

→ The Recurrence relation for Naive method of normal matrix multiplication is

$$T(n) = \begin{cases} 1 & n < 2 \\ 8T(n/2) + n^2 & n > 2 \end{cases}$$

→ Recurrence relation is

$$\boxed{T(n) = 8T(n/2) + n^2}$$

By master method calculate time complexity :-

$a = 8$
$b = 2$
$f(n) = n^2$

$T(n) = O(n^{\log_b^a})$ if $a > b$
$= O(n^{\log_2^8})$
$= O(n^{\log_2^{2^3}})$
$= O(n^3)$

∴ Time complexity for 2x2 matrix multiplication is $O(n^3)$

Algorithm for naive method :-

```
Algorithm mult (A[], B[], C[])
{
    for i <- 0 to n do step 1
    {
    for j <- 0 to n do step 1
    {
        C[i, j] <- 0
      for k <- 0 to n do step 1
      {
        C[i][j] = C[i][j] + A[i][k] *
                        B[k][i]
      }
    }
}
```

$$\left[\begin{array}{cc|cc} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ \hline C_{31} & C_{32} & C_{33} & C_{34} \\ C_{41} & C_{42} & C_{43} & C_{44} \end{array}\right] = \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right] \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right]$$

∴ $a_{11}, a_{12}, a_{21}, a_{22}$ are of sub matrices of A of size $n/2 \times n/2$

∴ 4x4 matrix is reduced to 2x2 matrix

∴ Big problem is reduced to smaller problem.

$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$ is assume to C_{11} $\begin{bmatrix} C_{13} & C_{14} \\ C_{23} & C_{24} \end{bmatrix}$ is assumed as C_{12}

$\begin{bmatrix} C_{31} & C_{32} \\ C_{41} & C_{42} \end{bmatrix}$ is assumed as C_{21} $\begin{bmatrix} C_{33} & C_{34} \\ C_{43} & C_{44} \end{bmatrix}$ is assumed as C_{22}

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

∴ 2x2 matrices formula is applied here.

→ The Recurrence relation for matrix multiplication using DAC is

$$T(n) = \begin{cases} 1 & n < 2 \\ 8T(n/2) + O(n^2) & n > 2 \end{cases}$$

∴ Multiplication = 8, Addition Performed = 4

→ Divide and conquer is applied when it is large problem and it breaks into smaller problem until we get the solution to original problem.

→ Following is simple divide and conquer method to multiply 4×4 matrices.

steps:-

1. Divide matrices A & B into 4 sub matrices of size ($n/2 \times n/2$).
2. Calculate following values recursively.

Example:-

Assume 4×4 for the following matrices & apply divide and conquer approach.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$n \times n \Rightarrow 4 \times 4$ matrix.

$$C = \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & C_{34} \\ C_{41} & C_{42} & C_{43} & C_{44} \end{bmatrix}$$

$C = A \times B$

→ This method is similar to the previous DAC method in the sense that this method also divide matrices to sub matrices of size $n/2 \times n/2$.

→ It reduces the number of recursive calls to '7'.

By using formula we can reduce the time complexity by using multiplication:-

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = (A_{21} + A_{22})B_{11}$

$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = (A_{11} + A_{12})B_{22}$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{21} + B_{22})$

$\therefore C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} (P+S-T+V) & (R+T) \\ (Q+S) & (P+R-Q+U) \end{bmatrix}$$

→ Total of multiplication is done in strassen's matrix multiplication = 7

→ Total no. of additions & subtraction done in strassen's matrix multiplication is = 18

Recurrence relation for strassen's matrix multiplication:-

$$T(n) = 7T(n/2) + 18n^2 \quad n \geq 2$$

Total time taken to compute addition & multiplication is $= O(n^2)$

$\therefore T(n) = 7T(n/2) + kn^2$ Assume $a = 18$ (or) $k = 18$

By using master Method:-

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

$a > b \Rightarrow 7 > 2$ then

$T(n) = O(n^{\log_b^a}) = O(n^{\log_2^7})$

$T(n) = O(n^{\log_2^7}) = O(n^{2.81})$ $\quad \log_2^7 = 2.81$

$\therefore$ Time complexity reduced to $O(n^3)$ to $O(n^{2.81})$

$\therefore$ strassen's matrix multiplication Time complexity is

$$O(n^{\log_2^7}) \simeq O(n^{2.81})$$

Example:- If matrices $A = \begin{bmatrix} 9 & 4 & 6 & 7 \\ 7 & 8 & 1 & 4 \\ 4 & 3 & 2 & 6 \\ 5 & 3 & 0 & 2 \end{bmatrix}$ $B = \begin{bmatrix} 7 & 6 & 2 & 1 \\ 3 & 9 & 0 & 3 \\ 2 & 5 & 2 & 9 \\ 3 & 2 & 4 & 7 \end{bmatrix}$

Implement strassen's matrix multiplication on A & B.

$$A = \begin{bmatrix} 9 & 4 & 6 & 7 \\ 7 & 8 & 1 & 4 \\ 4 & 3 & 2 & 6 \\ 5 & 3 & 0 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 7 & 6 & 2 & 1 \\ 3 & 9 & 0 & 3 \\ 2 & 5 & 2 & 9 \\ 3 & 2 & 4 & 7 \end{bmatrix}$$

$$C = \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & C_{34} \\ C_{41} & C_{42} & C_{43} & C_{44} \end{bmatrix}$$

→ Divide the given 4x4 matrices into sub matrices of size 2x2.

$$A = \left[\begin{array}{cc|cc} 9 & 4 & 6 & 7 \\ 7 & 8 & 1 & 4 \\ \hline 4 & 3 & 2 & 6 \\ 5 & 3 & 0 & 2 \end{array}\right]$$

$A_{11} = \begin{bmatrix} 9 & 4 \\ 7 & 8 \end{bmatrix}$ $A_{12} = \begin{bmatrix} 6 & 7 \\ 1 & 4 \end{bmatrix}$ $A_{21} = \begin{bmatrix} 4 & 3 \\ 5 & 3 \end{bmatrix}$ $A_{22} = \begin{bmatrix} 2 & 6 \\ 0 & 2 \end{bmatrix}$

$$B = \left[\begin{array}{cc|cc} 7 & 6 & 2 & 1 \\ 3 & 9 & 0 & 3 \\ \hline 2 & 5 & 2 & 9 \\ 3 & 2 & 4 & 7 \end{array}\right]$$

$B_{11} = \begin{bmatrix} 7 & 6 \\ 3 & 9 \end{bmatrix}$ $B_{12} = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$ $B_{21} = \begin{bmatrix} 2 & 5 \\ 3 & 2 \end{bmatrix}$ $B_{22} = \begin{bmatrix} 2 & 9 \\ 4 & 7 \end{bmatrix}$

→ The sub-matrices of size 2x2 can be computed as follows.

$A_{11} = \begin{bmatrix} 9 & 4 \\ 7 & 8 \end{bmatrix}$ $B_{11} = \begin{bmatrix} 7 & 6 \\ 3 & 9 \end{bmatrix}$

$C = A_{11} \times B_{11}$

	$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$ $P_1 = (A_{11}+A_{22})(B_{11}+B_{22})$ $= (9+8)(7+9)$ $= (17)(16)$ $\boxed{P = 272}$ $Q = (A_{21}+A_{22})B_{11}$ $= (7+8)7$ $= (15)7$ $\boxed{Q = 105}$ $R = A_{11}(B_{12}-B_{22})$ $= 9(6-9)$ $= 9(-3)$ $\boxed{R = -27}$ $S = A_{22}(B_{21}-B_{11}) \Rightarrow 8(3-7) = 8(-4)$ $\boxed{S = -32}$ $T = (A_{11}+A_{12})B_{22}$ $= (9+4)9$ $= (13)9$ $\boxed{T = 117}$ $U = (A_{21}-A_{11})(B_{11}+B_{12})$ $= (7-9)(7+6)$ $= (-2)(13)$ $\boxed{U = -26}$ $V = (A_{12}-A_{22})(B_{21}+B_{22})$ $= (4-8)(3+9)$ $= (-4)(12)$ $\boxed{V = -48}$ $\therefore C_{11} = P+S-T+V$ $= 272-32-117-48$ $= 272-197$ $\boxed{C_{11} = 75}$ $C_{12} = R+T = -27+117 = 90$ $C_{21} = Q+S$ $= 105-32$ $= 73$ $C_{22} = P+R-Q+U$ $= 272-27-105+(-26)$ $= 272-27-105-26$ $= 114$ $\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 75 & 90 \\ 73 & 114 \end{bmatrix}$ $\therefore C_{11} = \begin{bmatrix} 75 & 90 \\ 73 & 114 \end{bmatrix}$ In the similar way, compute the remaining 2x2 matrices. $A_{12} = \begin{bmatrix} 6 & 7 \\ 1 & 4 \end{bmatrix}$ $B_{12} = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$		

find $C_{12} = \begin{bmatrix} C_{13} & C_{14} \\ C_{23} & C_{24} \end{bmatrix}$

$P = (A_{11}+A_{22})(B_{11}+B_{22})$
$= (6+4)(2+3)$
$= (10)(5)$
$\boxed{P = 50}$

$Q = (A_{21}+A_{22})B_{11}$
$= (1+4)2$
$= (5)2$
$\boxed{Q = 10}$

$R = A_{11}(B_{12}-B_{22})$
$= 6(1-3)$
$= 6(-2)$
$\boxed{R = -12}$

$S = A_{22}(B_{21}-B_{11})$
$= 4(0-2)$
$= 4(-2)$
$\boxed{S = -8}$

$T = (A_{11}+A_{12})B_{22}$
$= (6+7)3$
$= (13)(3)$
$\boxed{T = 39}$

$U = (A_{21}-A_{11})(B_{11}+B_{12})$
$= (1-6)(2+1)$
$= (-5)(3)$
$\boxed{U = -15}$

$V = (A_{12}-A_{22})(B_{21}+B_{22})$
$V = (7-4)(0+3)$
$= (3)(3)$
$\boxed{V = 9}$

$\therefore C_{13} = P+S-T+V = 50-8-39+9 = 12$
$C_{14} = R+T = -12+39 = 27$
$C_{23} = Q+S = 10-8 = 2$
$C_{24} = P+R-Q+U = 50-12-10-15 = 13$

$$\begin{bmatrix} C_{13} & C_{14} \\ C_{23} & C_{24} \end{bmatrix} = \begin{bmatrix} 12 & 27 \\ 2 & 13 \end{bmatrix}$$

Similarly do for another matrix :-

$A_{21} = \begin{bmatrix} 4 & 3 \\ 5 & 3 \end{bmatrix}$ $B_{21} = \begin{bmatrix} 2 & 5 \\ 3 & 2 \end{bmatrix}$ find $C = \begin{bmatrix} C_{31} & C_{32} \\ C_{41} & C_{42} \end{bmatrix}$

$P = (4+3)(2+2) = (7)(4) = 28$ $\boxed{P = 28}$
$Q = (5+3)(2) = (8)(2) = 16$ $\boxed{Q = 16}$
$R = 4(5-2) = 4(3) = 12$ $\boxed{R = 12}$
$S = 3(3-2) = 3(1) = 3$ $\boxed{S = 3}$
$T = (4+3)2 = (7)(2) = 14$ $\boxed{T = 14}$
$U = (5-\cancel{4})(2+5) = (1)(7) = 7$ $\boxed{U = 7}$
$V = (3-3)(3+2) = 0(5) = 0$ $\boxed{V = 0}$

$C_{31} = 17$ $C_{41} = 16+3 = 19$
$C_{32} = 26$ $C_{42} = 31$

$$\therefore \begin{bmatrix} C_{31} & C_{32} \\ C_{41} & C_{42} \end{bmatrix} = \begin{bmatrix} 17 & 26 \\ 19 & 31 \end{bmatrix}$$

Similarly do it for another matrix :-

$A_{22} = \begin{bmatrix} 2 & 6 \\ 0 & 2 \end{bmatrix}$ $B_{22} = \begin{bmatrix} 2 & 9 \\ 4 & 7 \end{bmatrix}$ $C = \begin{bmatrix} C_{33} & C_{34} \\ C_{43} & C_{44} \end{bmatrix}$

$P = (2+2)(2+7)$
$= (4)(9)$
$\boxed{P = 36}$

$Q = (0+2)2$
$\boxed{Q = 4}$

$R = 2(9-7)$
$= 2(2)$
$\boxed{R = 4}$

$S = 2(4-2)$
$= 2(2)$
$\boxed{S = 4}$

$T = (2+6)7$
$= (8)7$
$\boxed{T = 56}$

$U = (0-2)(2+9)$
$= (-2)(11)$
$\boxed{U = -22}$

$$V = (6-2)(4+7)$$
$$= (4)(11)$$
$$\boxed{V = 44}$$

$$\therefore C_{33} = 28 \qquad C_{43} = 8$$
$$C_{34} = 60 \qquad C_{44} = 14$$

$$\therefore \begin{bmatrix} C_{33} & C_{34} \\ C_{43} & C_{44} \end{bmatrix} = \begin{bmatrix} 28 & 60 \\ 8 & 14 \end{bmatrix}$$

Thus the product of the matrices.

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & C_{34} \\ C_{41} & C_{42} & C_{43} & C_{44} \end{bmatrix} = \begin{bmatrix} 9 & 4 & 6 & 7 \\ 7 & 8 & 1 & 4 \\ 4 & 3 & 2 & 6 \\ 5 & 3 & 0 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 6 & 2 & 1 \\ 3 & 9 & 0 & 3 \\ 2 & 5 & 2 & 9 \\ 3 & 2 & 4 & 7 \end{bmatrix}$$

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & C_{34} \\ C_{41} & C_{42} & C_{43} & C_{44} \end{bmatrix} = \begin{bmatrix} 45 & 90 & 12 & 27 \\ 73 & 114 & 2 & 13 \\ 17 & 26 & 28 & 60 \\ 19 & 31 & 8 & 14 \end{bmatrix}$$

UNIT –III
GREEDY METHOD, DYNAMIC PROGRAMMING

1	**Explain in detail about general method of greedy method with algorithm and list the few applications of greedy method.** General method : → The greedy method is the most possible straight forw- and design Technique, used to determine a feasible solution that may or may not be optimal. Feasible solution :- → most problems have n inputs and its solution contain-s a subset of inputs that satisfies a given constraint (condition) → Any subset that satisfies the constraint is called feasible solution optimal solution :- → To find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called optimal solution → the greedy method suggests that an algorithm works in stages, considering one input at a time. → At each stage a decision is made regarding whether a particular input is in an optimal solution → Greedy algorithms neither postpone nor revise ~~the~~ decisions (i.e., no backtracking) Example : kruskals algorithm . minimal spanning tree that select an edge from a sorted list, check, decide on & never visit it again.	[L2][CO3]	[12M]

(2)

Algorithm for Greedy method :

```
Algorithm Greedy (a,n)
// a[1:n] contains the n inputs
{
  solution ::= 0
  for i=1 to n do
  {
     x:= Select (a);
     if feasible (solution, x) then
     solution := union (solution, x);
  }
   return solution ;
}
```

Selection :- Function that selects an input from a[] and removes it. ~~the~~ selected inputs value is assigned to x.

Feasible : Boolean-valued function that determines whether x can be included into the solution vector

union :- Function that combines x with solution and updates ~~the~~ shortest p objectives functions.

Topic 2 : Applications :

1. Job sequencing with dead-lines
2. 0/1 Knapsack problem
3. minimum cost spanning Tree
4. Single source shortest path problem.

Topic 3 : Job sequencing with dead-lines.

-> There is set of n jobs. for any job i, is a integer dead lines $d_i > 0$ and profit $p_i > 0$, the profit p_i is earned if and if the job completed by its deadlines

	③ → To complete a job one had to process the job on a machine for one unit of time. only one machine is available for processing jobs → A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its dead lines → The value of feasible solution J is the sum of the profits of the job in J, i.e, $\sum_{i \in j} P_i$ → An optimal solution is a feasible solution with maximum value. → The problem involves identification of a subset of job which can be completed by its dead line. → Therefore the problem suits the subset methodology and can be solved by the greedy method. Example: Obtain the optimal sequence for following jobs $(P_1, P_2, P_3, P_4) = (\overset{J_1}{100}, \overset{J_2}{10}, \overset{J_3}{15}, \overset{J_4}{27})$ $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$ $n = 4$ Feasible solution | processing sequence. | value. j_1, j_2 (1, 2) | (2, 1) | 100+10 = 110 (1, 3) | (1,3) (or) (3,1) | 100+15 = 115 (1, 4) | (4, 1) | 27+100 = 127 (2, 3) | (2, 3) | 10+15 = 25 (3, 4) | (4, 3) | 15+27 = 42		
2	**Elaborate job sequencing with deadlines by using greedy method where given the jobs, their deadlines and associated profits as shown below. Calculate maximum earned profit.**	[L6][CO3]	[12M]

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Job Sequencing With Deadlines-

The sequencing of jobs on a single processor with deadline constraints is called as Job Sequencing with Deadlines

- You are given a set of jobs.
- Each job has a defined deadline and some profit associated with it.
- The profit of a job is given only when that job is completed within its deadline.
- Only one processor is available for processing all the jobs.
- Processor takes one unit of time to complete a job.

Greedy Algorithm-

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.

The greedy algorithm described below always gives an optimal solution to the job sequencing problem-

Step-01:

Sort all the given jobs in decreasing order of their profit.

Step-02:

Check the value of maximum deadline.

Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline

Step-03:

Pick up the jobs one by one.

Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

Given the jobs, their deadlines and associated profits as shown-

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Solution-

Step-01:

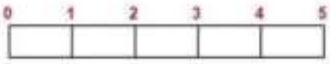
Sort all the given jobs in decreasing order of their profit-

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

Value of maximum deadline = 5.

So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



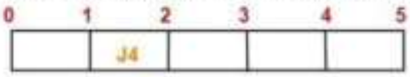
Gantt Chart

Now,

- We take each job one by one in the order they appear in Step-01.
- We place the job on Gantt chart as far as possible from 0.

Step-03:

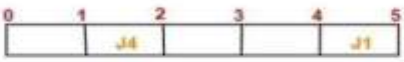
- We take job J4.
- Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-



Step-04:

We take job J1.

- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



Step-05:

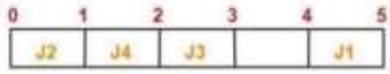
We take job J3.

- Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-

Step-06:

We take job J2.

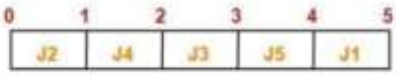
- Since its deadline is 3, so we place it in the first empty cell before deadline 3.
- Since the second and third cells are already filled, so we place job J2 in the first cell as-



Step-07:

Now, we take job J5.

- Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-



Now,

- The only job left is job J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 can not be completed.

Part-01:

The optimal schedule is-

J2 , J4 , J3 , J5 , J1

This is the required order in which the jobs must be completed in order to obtain the maximum profit.

Part-02:

All the jobs are not completed in optimal schedule.

- This is because job J6 could not be completed within its deadline.

Part-03:

Maximum earned profit

= Sum of profit of all the jobs in optimal schedule

= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1

= 180 + 300 + 190 + 120 + 200

= 990 units

3	**Construct an optimal solution for Knapsack problem, where n=7,M=15 and (p1,p2,p3,p4,p5,p6,p7) = (10,5,15,7,6,18,3) and (w1,w2,w3,w4,w5,w6,w7) = (2,3,5,7,1,4,1) by using Greedy strategy.**	[L3][CO3]	[12M]

Knapsack problem: (5)

Given n item of known weights w_i and values $v_{i=1,2,\dots n}$ and a knapsack of capacity w. find most valuable subset of the item that fit the knapsack it is convenient to order the item of a given instance in descending order by their value to weight ratios. then the first item gives the best payoff for weight unit and the last one gives worst pay off for weight unit.

Constrain. $\Sigma x_i w_i \le M$

max $\Sigma x_i p_i$

1) compare the value to weight ratio
2) Sort the item in non increasing order of the ratio
3) If the current item on the list fits into the knapsack, place it in knapsack, otherwise proceed to next.

Algorithm:

```
for i = 1 to n
do x[i] = 0
weight = 0
for i=1 to n
if weight + w[i] ≤ w then
   x[i] = 1
   weight = weight + w[i]
else
   x[i] = (w - weight) / w[i]
   weight = w
   break
return x.
```

Example:

(6)

object	1	2	3	4	5	6	7
Profit	10	5	15	7	6	18	3
weight	2	3	5	7	1	4	1

No. of objects (n) = 7

weight of bag (w) = 15

Solution

objects (x)	1	2	3	4	5	6	7
profit (P)	10	5	15	7	6	18	3
weight (w)	2	3	5	7	1	4	1
P/w	5	1.6	3	1	6	4.5	3
objects (x_i)	x_1	x_2	x_3	x_4	x_5	x_6	x_7
	1	2/3	1	0	1	1	1

capacity of bag m = 15

bag capacity = 15 kg

15 - 1 = 14
14 - 2 = 12
12 - 4 = 8
8 - 5 = 3
3 - 1 = 2
2 - 2 = 0

$$\Sigma x_i p_i = 1 * 10 + 2/3 * 5 + 1 * 15 + 0 * 7 + 1 * 6 + 1 * 18 + 1 * 3$$
$$= 10 + 3.33 + 15 + 0 + 6 + 18 + 3$$
$$= 55.33$$

$$\Sigma x_i w_i = 1 * 2 + 2/3 * 3 + 1 * 5 + 0 * 7 + 1 * 1 + 1 * 4 + 1 * 1$$
$$= 2 + 2 + 5 + 0 + 1 + 4 + 1 = 15$$

4	a)	**Simplify the algorithm for Knapsack problem and analyze time complexity.**	[L4][CO3]	[6M]

Knapsack Problem Using Greedy Method: The selection of some things, each with profit and weight values, to be packed into one or more knapsacks with capacity is the fundamental idea behind all families of knapsack problems. The knapsack problem had two versions that are as follows:

1. **Fractional Knapsack Problem**
2. **0 /1 Knapsack Problem**

The fractional **Knapsack problem using the Greedy Method** is an efficient method to solve it, where you need to sort the items according to their ratio of value/weight.

In a fractional knapsack, we can break items to maximize the knapsack's total value. This problem in which we can break an item is also called the **Fractional knapsack problem**.

Here, we will see Knapsack Problem using Greedy method in detail, along with its algorithm and examples

In this method, the Knapsack's filling is done so that the maximum capacity of the knapsack is utilized so that maximum profit can be earned from it. The knapsack problem using the Greedy Method is referred to as:

Given a list of n objects, say $\{I_1, I_2, \ldots\ldots, I_n)$ and a knapsack (or bag).
The capacity of the knapsack is M.

Each object I_j has a weight w_j and a profit of p_j
If a fraction x_j (where $x \in \{0\ldots., 1\})$ of an object I_j is placed into a knapsack, then a profit of $p_j x_j$ is earned.

The problem (or Objective) is to fill the knapsack (up to its maximum capacity M), maximizing the total profit earned.
Mathematically:

$$\text{Maximize (the profit)} = \sum_{j=1} p_j x_j$$

$$= \sum_{j=1}^{n} w_j x_j \leq M \text{ and } x_j \in \{0, \ldots., 1\}, 1 \leq j \leq n$$

Note that the value of x_j will be any value between 0 and 1 (inclusive). If any object I_j is completely placed into a knapsack, its value is 1 ($x_j = 1$).

If we do not pick (or select) that object to fill into a knapsack, its value is 0 ($x_j = 0$). Otherwise, if we take a fraction of any object, then its value will be any value between 0 and 1

Knapsack Problem Using Greedy Method Pseudocode

A pseudo-code for solving knapsack problems using the greedy method is;
greedy fractional-knapsack (P[1...n], W[1...n], X[1..n], M)
/*P[1...n] and W[1...n] contain the profit and weight of the n-objects ordered such that X[1...n] is a solution set and M is the capacity of knapsack*/
{

For j ← 1 to n do
X[j]← 0
profit ← 0 // Total profit of item filled in the knapsack
weight ← 0 // Total weight of items packed in knapsacks
j ← 1
While (Weight < M) // M is the knapsack capacity

Time complexity of the fractional knapsack problem using greedy method

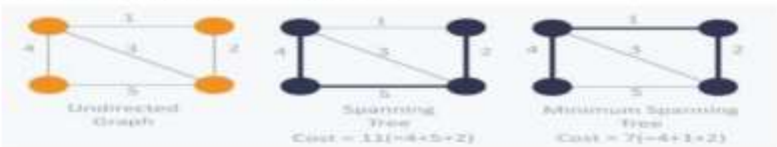
Sorting of n items (or objects) in decreasing order of the ratio Pj/Wj takes O (n log n) time. Since this is the lower bound for any comparison-based sorting algorithm. Therefore, the total time including sort is **O(n log n).**

b) What is minimum cost spanning tree and write the algorithm of pseudo code for kruskals algorithm [L3][CO3] [6M]

A **spanning tree** is defined as a tree-like subgraph of a connected, undirected graph that includes all the vertices of the graph. Or, to say in Layman's words, it is a subset of the edges of the graph that forms a tree (**acyclic**) where every node of the graph is a part of the tree.

The minimum spanning tree has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all possible spanning trees. Like a spanning tree, there can also be many possible MSTs for a graph

*A **minimum spanning tree (MST)** is defined as a **spanning tree** that has the minimum weight among all the possible spanning trees.*



Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

If the graph is not linked, then it finds a Minimum Spanning Tree.

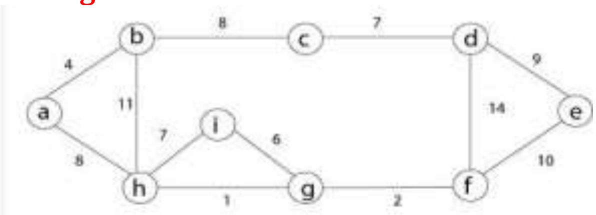
Steps for finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

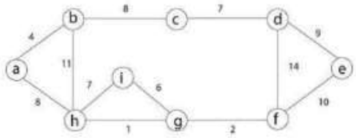
Alg:

MST- KRUSKAL (G, w)

1. A ← Ø
2. for each vertex v ∈ V [G]
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u, v) ∈ E, taken in non decreasing order by weight
6. do if FIND-SET (μ) ≠ if FIND-SET (v)
7. then A ← A ∪ {(u, v)}
8. UNION (u, v)
9. return A

	Analysis: O (E log E) = O (E log V). **Explain any example.**		
5	**Apply the minimum spanning tree of the following graph using Kruskals algorithm and prims algorithm.**  	[L3][CO3]	[12M]

For Example: Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm



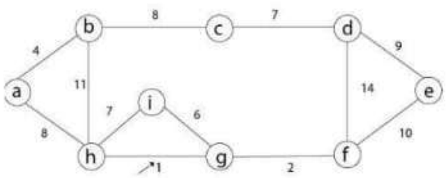
Solution: First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

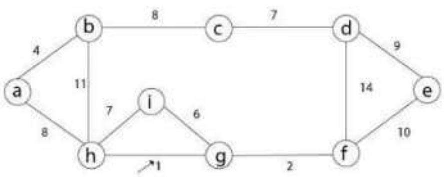
Weight	Source	Destination
1	h	g
2	g	f
4	a	b
6	i	g
7	h	i
7	c	d
8	b	c
8	a	h
9	d	e
10	e	f
11	b	h
14	d	f

Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

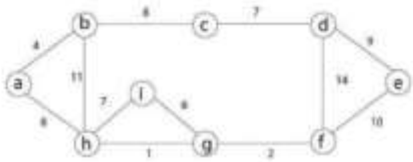
Step1: So, first take (h, g) edge



Step 2: then (g, f) edge.

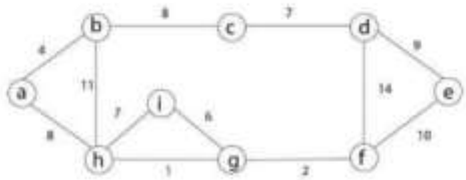


Step 3: then (a, b) and (i, g) edges are considered, and the forest becomes



Step 4: Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.

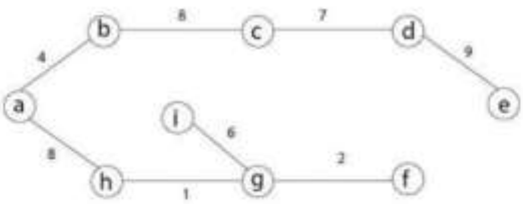


Step 5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

Step 6: After that edge (d, f) and the final spanning tree.

it contains all the 9 vertices and (9 - 1) = 8 edges

e → f, b → h, d → f [cycle will be formed]



6	a)	**Write short notes about general method of dynamic programming.**	[L3][CO3]	[3M]

GENERAL METHOD:-

→ It is a strategy for designing algorithm.
→ Dynamic programmig is also used in optimization problems.
→ Like divide and conquer method, Dynamic programming solves problems by combining the solutions of subproblems.
→ Moreover, Dynamic programming algorithm solves each sub problem just once & then saves its answer in a table, there by avoiding the work of re computing the answer.

Two main properties:-
→ These two main properties suggest that the given problem can be solved using dynamic programming.
1. overlapping subproblems.
2. optimal substructure.

1. overlapping subproblems:-
→ Similar to divide and conquer approach, Dynamic programming also combines solutions to sub problems.

25

→ It is mainly used where the solution of one subproblem is needed repeatedly.
→ The computed solutions are stored in a table, so that these don't have to be re-computed.
→ Hence, this technique is needed where overlapping subproblems exists.
For example:-
→ Binary search does not have overlapping subproblems, whereas recursive program of fibonacci numbers have many overlapping sub problems.

2. optimal substructure (principle of optimality)
→ A given problem has optimal substructure property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub problems.
For example:-
→ The shortest path problem has the following optimal substructure property.

Steps for Dynamic programming:-
→ Dynamic programming design involves four major steps.
1. characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution typically in a bottom-up fashion.
4. Construct an optimal solution from the computed information.

APPLICATIONS:-

→ Various problems that can be solved using dynamic programming are:-

1. All pairs shortest path
2. optimal binary search trees.
3. 0/1 knapsack problem.
4. Travelling salesperson problem.
5. Multi stage Graphs.

Example for dynamic programming - how it will applied.

1) Greedy method → It will always take only one decision.
2) Dynamic programming → It will took all possible sequences and picks best optimal solution.
   → It follows principle of optimality (sequence of solutions).

→ Every stage we take decisions.
→ Dynamic programming adopts tabulization method (or) memorization method.

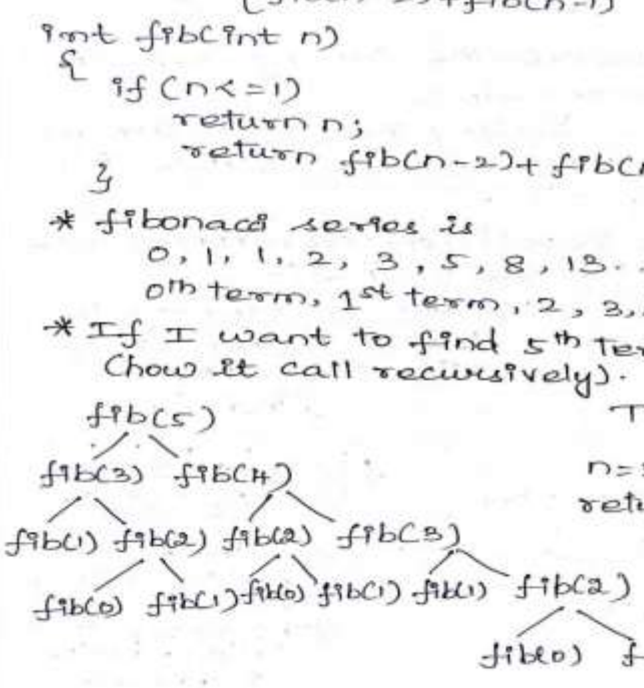
Recursion definition of fibonacci term:-

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

```
int fib(int n)
{
  if (n<=1)
     return n;
     return fib(n-2)+ fib(n-1);
}
```

* fibonacci series is
  0, 1, 1, 2, 3, 5, 8, 13....
  0th term, 1st term, 2, 3, 4, 5, 6, 7.....

* If I want to find 5th term in fibonacci (how it call recursively).



Trace the program
n=5, 5<=1 (false)
return fib(5-2)+ fib(5-1).
fib(3)+fib(4)

(2.8)

Recurrence relation:-

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

Time complexity is $O(2^n)$.

Disadvantage:-

→ If i call this fibonacci function, it will take too much time to compute.

Is there any way to reduce the function time.

→ When you observe the tracing tree, fib(1) is calling so many times.

→ why we are calling these function so many times & why can't we reduce this one.

To reduce the repetition (overlapping), we are moving to tabular method:-

```
int fib(int n)
{
  if(n<=1)
    return n;
  f[0]=0;
  f[1]=1;
  for(int i=2; i<=n; i++)
  {
    f[i]=f[i-2]+f[i-1];
  }
  return f[n];
}
```

Dynamic programming in Tabular method (It uses iterative approach)

find fib(5)

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

→ filling is done on from '0' term onwards.

→ Therefore it is called, bottom-up approach.

b) Build any one application of dynamic programming with an example. [L6][CO1] [9M]

THE TRAVELLING SALES PERSON PROBLEM:-

→ It is one of the algorithm strategy used in dynamic programming.

→ Here the salesman should start ~~at~~ at a Point and travels all the places and comes back to starting point.

→ The main objective of the problem is to minimize the travelling cost.

→ The main requirement is there should be communication between nodes.

Formula for calculating the cost adjacency matrix in dynamic programming is,

$$g(i, S) = \min_{j \in S} \{ C_{ij} + g(j, S - \{j\}) \}$$

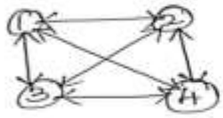
(50)

$g(i,s) \rightarrow$ length of shortest path starting at vertex i, going through all vertices in s & terminating at vertex 1.

$g\{1, v-\{1\}\}$ is the length of an optimal salesperson tour.

Example :-

For the following graph find minimum cost tour for the travelling salesperson problem.



The cost adjacency matrix

$$= \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Let the cost adjacency matrix

$$C_{ij} = \begin{array}{c|c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix} \end{array}$$

$\rightarrow$ Let us start the tour from vertex 1.

formula :-

$$g(i,s) = \min\{C_{ij} + g(j, s-\{j\})\} \rightarrow ①$$

clearly,

$$g(1,\phi) = C_{11} = 0$$
$$g(2,\phi) = C_{21} = 5$$
$$g(3,\phi) = C_{31} = 6$$

(51)

$$g(4,\phi) = C_{41} = 8$$

Using equation 1 we obtain

$$g(1,\{2,3,4\}) = \min\{C_{12} + g(2,\{3,4\}), C_{13} + g(3,\{2,4\}), C_{14} + g(4,\{2,3\})\}$$

$$g(2,\{3,4\}) = \min\{C_{23} + g(3,\{4\}), C_{24} + g(4,\{3\})\}$$

$$g(3,\{4\}) = \min\{C_{34} + g(4,\phi)\}.$$
$$= \min\{12+8\}$$
$$= 20$$

$$g(4,\{3\}) = \min\{C_{43} + g(3,\phi)\}$$
$$= 9+6$$
$$= 15$$

Therefore calculate the value for $g(2,\{3,4\})$

$$g(2,\{3,4\}) = \min\{C_{23} + g(3,\{4\}), C_{24} + g(4,\{3\})\}$$
$$= \min\{9+20, 10+15\}$$
$$= \min\{29, 25\}$$
$$g(2,\{3,4\}) = 25$$

Therefore, $g(3,\{2,4\})$

$$g(3,\{2,4\}) = \min\{C_{32} + g(2,\{4\}), C_{34} + g(4,\{2\})\}.$$

$$g(2,\{4\}) = \min\{C_{24} + g(4,\phi)\}$$

52

$= 10 + 8$

$= 18$

$g(4, \{2\}) = \min\{C_{42} + g(2, \phi)\}$

$= 8 + 5$

$= 13$

$g(3, \{2,4\}) = \min\{13 + 18,\ 12 + 13\}$

$= \min\{31, 25\}$

$= 25$

Therefore $g(4, \{2,3\})$

$g(4, \{2,3\}) = \min\{C_{42} + g(2, \{3\}),\ C_{43} + g(3, \{2\})\}.$

$g(2, \{3\}) = \min\{C_{23} + g(3, \phi)\}$

$= \min\{9 + 6\}$

$= 15$

$g(3, \{2\}) = \min\{C_{32} + g(2, \phi)\}$

$= 13 + 5$

$= 18$

$g(4, \{2,3\}) = \min\{8 + 15,\ 19 + 18\}$

$= \min\{23, 27\}$

$g(3, \{2,4\}) = 23$

$\therefore g(1, \{2,3,4\}) = \min\{C_{12} + g(2, \{3,4\}),\ C_{13} + g(3, \{2,4\}),\ C_{14} + g(4, \{2,3\})\}$

$= \min\{10 + 25,\ 15 + 25,\ 20 + 23\}$

$= \min\{35, 40, 43\}$

$g(1, \{2,3,4\}) = 35$

$\therefore$ The optimal tour for the graph has length $= 35$.

(or)

minimum cost tour for the travelling salesperson problem is 35.

$\therefore$ The best optimal tour path is

$1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$

7	Discuss about Optimal binary search tree with suitable example.	[L2][CO3]	[12M]

OPTIMAL BINARY SEARCH TREE :-

Problem description:-

→ Let $\{a_1, a_2, \ldots a_n\}$ a set of identifiers such that $a_1 < a_2 < a_3$.

→ Let $P(i)$ be the probability with which we can search for a_i & $q(i)$ be the probability of successful searching element x such that $a_i < x < a_{i+1}$ & $0 \leq i \leq n$.

→ In other words.

$P(i) \rightarrow$ probability of successful search.

$q(i) \rightarrow$ probability of unsuccessful search.

→ Also $\sum_{1 \leq i \leq n} P(i) + \sum_{1 \leq i \leq n} q(i)$ Then obtain a tree with minimum cost.

→ Such a tree with optimum cost is called optimal binary search tree.

→ To solve this problem using dynamic programming method we will perform following steps.

Steps for optimal Binary Search Tree:-

Step 1:- Notations used Let,

$$T_{ij} = OBST(a_{i+1}, \ldots, a_j)$$

(50)

C_{ij} denotes the cost (T_{ij})

W_{ij} is the weight of each T_{ij}.

T_{0n} is the final tree obtained.

T_{00} is empty.

$T_{i,i+1}$ is a single-node tree that has element a_{i+1}.

During the computations the root values are computed & r_{ij} stores the root value of T_{ij}.

Step 2:-

→ The OBST can be build using the principle of optimality. consider the process of creating OBST.

→ Let T_{0n} be an OBST for the elements $a_1 < a_2 < \ldots < a_n$ & Let L & R be its left & right subtree. Suppose that the root of T_{0n} is a_k, for some k.

→ Then the elements in the left subtree 'L' are $a_1, a_2, \ldots, a_{k-1}$ & the elements in the right subtree 'R' are $a_{k+1}, a_{k+2}, \ldots, a_n$.

→ The cost of computing the T_{0n} can be given as

$$C(T_{0n}) = C(L) + C(R) + P_1 + P_2 + P_3 + \cdots + P_n + q_0 + q_1 + q_2 + \cdots + q_n$$

i.e

(51)

$C(T_{on}) = C(L) + C(R) + W$

where

$W = P_1 + P_2 + \dots + P_n + q_0 + q_1 + q_2 + \dots + q_n$.

→ If L is not an optimal BST for its elements, then we can find another tree 'L'' for the same elements, with the property $C(L') < C(L)$ (i.e. optimal cost).

∴ Let T' be the tree with root a_k, left subtree L' & right subtree R.

Then,

$$C(T') = C(L') + C(R) + W$$

i.e $C(T') < C(L) + C(R) + W$

i.e $< C(T_{on})$

→ That means, T' is optimal than Ton. This contradicts the fact that Ton is an optimal BST. Therefore L must be an optimal for its elements.

→ In the same manner we can obtain optimal binary search tree by building the optimal subtrees. This ultimately shows that optimal binary search tree follows the principle of optimality.

Step 3:–

→ We will apply following formula for computing each sequence.

(52)

$C(i,j) = \min_{i<k\le j} \{C(i,k-1) + C(k,j)\} + W(i,j)$

$W(i,j) = W[i, j-1] + P[j] + q[j]$;

$r[i,j] = k$

Analysis:–

→ The computation of each c and r can be done using three nested for loops. Here the time complexity turns out to be $O(n^3)$.

Example:–

consider four elements a_1, a_2, a_3 and a_4 with $Q_0 = 1/8$, $Q_1 = 3/16$, $Q_2 = Q_3 = Q_4 = 1/16$ and $P_1 = 1/4$, $P_2 = 1/8$, $P_3 = 1/16$. construct an optimal binary search tree. Solving for $C(0,n)$:

Step 1: –

computing all $C(i,j)$ such that $j - i = 1$; $j = i+1$ and as $0 \le i < 4$; $i = 0,1,2$ and 3; $i < k \le j$

Start with $i=0$; so $j=1$; as $i<k\le j$, so the possible value for $k=1$

$W(0,1) = P(1) + Q(1) + W(0,0) = 4+3+2 = 9$

$C(0,1) = W(0,1) + \min\{C(0,0) + C(1,1)\}$

$= 9 + [0+0] = 9$ $r(0,1) = 1$.

Next with $i=1$; so $j=2$; as $i<k\le j$; so the possible value for $k=2$

$W(1,2) = P(2) + Q(2) + W(1,1) = 2+1+3 = 6$

(33)

$C(1,2) = W(1,2) + \min\{C(1,1) + C(2,2)\}$

$= 6 + [(0+0)] = 6$ $ft(1,2) = 2$

Next with $i=2$; so $j=3$; as $i<k\le j$; so the possible Value for $k=3$

$W(2,3) = P(3) + Q(3) + W(2,2) = 1+1+1 = 3$.

$C(2,3) = W(2,3) + \min\{C(2,2) + C(3,3)\}$

$= 3 + [(0+0)] = 3$ $ft(2,3) = 3$

Next with $i=3$; So $j=4$; as $i<k\le j$; so the possible Value for $k=4$

$W(3,4) = P(4) + Q(4) + W(3,3) = 1+1+1 = 3$

$C(3,4) = W(3,4) + \min\{[C(3,3) + C(4,4)]\}$

$= 3 + [(0+0)] = 3$ $ft(3,4) = 4$

step 2:-

computing all $C(i,j)$ such that $j-i=2$; $j=i+2$ and ~~so~~ as $0\le i<3$; $i=0,1,2$; $i<k\le j$.

start with $i=0$; so $j=2$; as $i<k\le j$, so the possible Values for $k=1$ and 2.

$W(0,2) = P(2) + Q(2) + W(0,1) = 2+1+9 = 12$

$C(0,2) = W(0,2) + \min\{(C(0,0) + C(1,2)),$
$(C(0,1) + C(2,2))\}$.

$= 12 + \min\{(0+6, 9+0)\} = 12 + 6$

$= 18$ $ft(0,2) = 1$

(34)

Next with $i=1$, so $j=3$ as $i<k\leq j$; so the possible value for $k=2$ and 3.

$W(1,3)=p(3)+Q(3)+W(1,2)=1+1+6=8$

$C(1,3)=W(1,3)+\min\{[C(1,1)+C(2,3)],$
$[C(1,2)+C(3,3)]\}$

$=W(1,3)+\min\{(0+3),(6+0)\}$

$=8+3=11$

$ft(1,3)=2$

Next with $i=2$, so $j=4$, as $i<k\leq j$, so the possible value for $k=3$ and 4.

$W(2,4)=p(4)+Q(4)+W(2,3)=1+1+3=5$

$C(2,4)=W(2,4)+\min\{[C(2,2)+C(3,4)],$
$[C(2,3)+C(4,4)]$

$=5+\min\{(0+3),(3+0)\}$

$=5+3=8$

$ft(2,4)=3$

Step 3:-

computing all $C(i,j)$ such that $j-i=3$;
$j=i+3$ and as $0\leq i<2$; $i=0,1$; $i<k\leq j$.

start with $i=0$, so $j=3$ as $i<k\leq j$, so the possible values for $k=1,2$ and 3.

$W(0,3)=p(3)+Q(3)+W(0,2)=1+1+12=14$

$C(0,3)=W(0,3)+\min\{[C(0,0)+C(1,3)],$
$[C(0,1)+C(2,3)],[C(0,2)+C(3,3)]\}$

$=14+\min\{(0+11),(9+3),(18+0)\}$

$= 14 + 11$

$= 25 \quad ft(0,3) = 1$

Start with $i = 1$, so $j = 4$; as $i < k \le j$, so the possible values for $k = 2, 3$ and 4.

$W(1,4) = p(4) + Q(4) + W(1,3) = 1 + 1 + 8 = 10$

$C(1,4) = W(1,4) + \min\{[C(1,1) + C(2,4)], [C(1,2) + C(3,4)], [C(1,3) + C(4,4)]\}$

$= 10 + \min\{(0+8), (6+3), (11+0)\}$

$= 10 + 8 = 18 \quad ft(1,4) = 2$

Step 4:

Computing all $C(i,j)$ such that $j - i = 4$; $j = i + 4$ and as $0 \le i < 1$; $i = 0$; $i < k \le j$. Start with $i = 0$; so $j = 4$; as $i < k \le j$, so the possible values for $k = 1, 2, 3$ and 4.

$W(0,4) = p(4) + Q(4) + W(0,3)$

$= 1 + 1 + 14 = 16$

$C(0,4) = W(0,4) + \min\{[C(0,0) + C(1,4)], [C(0,1) + C(2,4)], [C(0,2) + C[3,4)], [C(0,3) + C[4,4)]\}$

$= 16 + \min\{0 + 18), (9+8), (18+3), 25+0)]$

$= 16 + 17$

$= 33 \quad R(0,4) = 2$

36

Table for recording W(i, j), C(i, j) and R(i, j)

Column / Row	0	1	2	3	4
0	2,0,0	1,0,0	1,0,0	1,0,0	1,0,0
1	9,9,1	6,6,2	3,3,3	3,3,4	
2	12,18,1	8,11,2	5,8,3		
3	14,25,2	11,18,2			
4	16,33,2				

→ From the table we see that C(0,4)=33 is the minimum cost of a binary search tree for (a_1, a_2, a_3, a_4).

→ The root of the tree 'T_{04}' is 'a_2'.

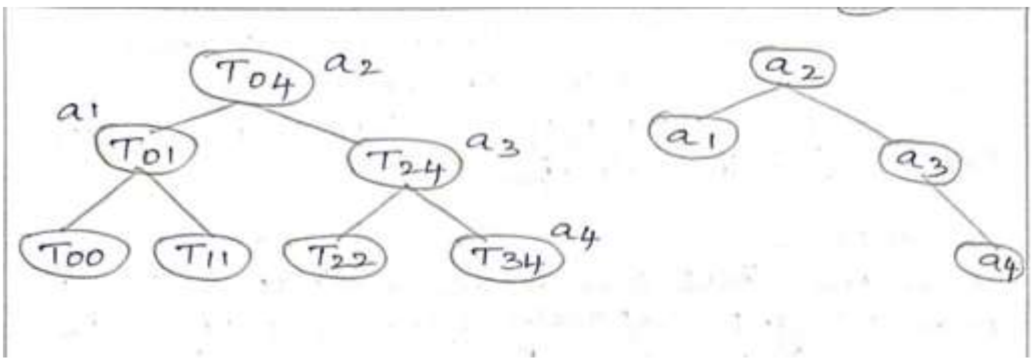
→ Hence the left subtree is 'T_{01}' and right sub tree is T_{24}. The root of 'T_{01}' is 'a_1' and the root of 'T_{24}' is a_3.

→ The left and right sub trees for 'T_{01}' are 'T_{00}' and 'T_{11}' respectively. The root of T_{01} is 'a_1'. The left and right sub trees for T_{24} are T_{22} and T_{34} respectively.

→ The root of T_{24} is 'a_3'.

→ The root of T_{22} is 'null'.

→ The root of T_{34} is 'a_4'



8	**Explain 0/1 knapsack problem by using dynamic programming with an examples.**	[L2][CO3]	[12M]

0/1 KNAPSACK:-

Problem statement:-

→ The profit should be maximum.

Note:-

→ Here fractions are not included.

→ x_i value should be =0/1.

Formula:-

1. max $\sum x_i p_i$ (sum of profits should be maximized).
2. $\sum x_i w_i \leq m$ (weights should be less than (or) equal to the bag capacity)

→ In this problem, we have a knapsack that has a weight limit W.

→ There are items $i_1, i_2, \ldots i_n$ each having weight $w_1, w_2, \ldots w_n$ and some benefit (value (or) profit) associated with it $v_1, v_2, \ldots v_n$.

→ our objective is to maximize the benefit such that the total weight inside the knapsack is at most W.

(38)

→ since this is 0-1 knapsack problem so we can either take an entire item (or) reject it completely. we cannot break an item & fill the knapsack.

Example:-

Assume that we have a knapsack with max weight capacity w=5. our objective is to fill the knapsack with items such that the benefit (value (or) profit) is maximum.

consider the following 4 items & their weights & value.

items(i)	1	2	3	4
Value(val)	100	20	60	40
weight(wt)	3	2	4	1

n=4 & w=5

solution:-

Create a value table V[i,w] where i denotes number of item & w denotes the weight of the items (capacity)

V[i,w]	w=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	0	100	100	100
2	0	0	20	100	100	120
3	0	0	20	100	100	120
4	0	40	40	100	140	140

(39)

→ Rows denote the items.
→ Columns denotes the weight.
→ As there are 4 items i=0, 1, 2, 3, 4
Initially i=0 & place remaining 4 items.
→ The weight limit of the Knapsack is W=5 so we have 6 columns from 0 to 5.

Initially:-
Step 1:-
i=0 , w=0
Formula:-
wt[i] > w
V[i, w] = V[i-1, w]
wt[0] > 0
wt[0] > 0
0 > 0 [false]

Step 2:-
1) i=1, w=0
wt[1] > 0
3 > 0 [True]
V[1, 0] = V[1-1, 0]
= V[0, 0]
$\boxed{V[1,0]=0}$

2) i=1, w=1
wt[1] > 1
3 > 1 [True]
V[1, 1] = V[1-1, 1]
= V[0, 1]
$\boxed{V[1,1]=0}$

3) i=1, w=2
wt[1] > 2
3 > 2 [True]
V[1, 2] = V[1-1, 2]
= V[0, 2]
$\boxed{V[1,2]=0}$

4) i=1, w=3
wt[1] > 3
3 > 3 [false]

$\boxed{V[i, w] = max(V(i-1), w), Val[i] + V[i-1, w-wt[i])}$

(40)

V(1, 3) = max(V(1-1, 3), Val[1] + V(1-1, 3 - wt[1]))
= max(V(0, 3), 100 + V(0, 3-3))
= max (V(0, 3), 100 + V(0,0))
= max (0, 100+0)

$\boxed{V(1, 3) = 100}$

5) i=1, w=4
wt[1] > 4
3 > 4 [False]
V(1, 4) = max(V(1-1, 4), Val[1] + V(1-1, 4-wt(1)))
= max(V(0, 4), Val[1] + V(0, 4-3))
= max (0, 100 + V(0, 1))
= max(0, 100 + 0)
V(1, 4) = max(0, 100)

$\boxed{V(1,4) = 100}$

6) i=1, w=5
wt[1] > 5
3 > 5 [false]
V(1, 5) = max(V(1-1, 5), Val[1] + V(1-1, 5-wt(1)))
= max(V(0, 5), 100 + V(0, 5-3))
= max (0, 100 + V(0, 2))
= max (0, 100 + 0)
$\boxed{V(1,5) = 100}$

(41)

Step 3 :-

1) i=2, w=0

V(2,0)=0

2) i=2, w=1

V(2,1)=0

3) i=2, w=2

V(2,2)=20

4) i=2, w=3

V(2,3)=100

5) i=2, w=4

V(2,4)=100

6) i=2, w=5

V(2,5)=120

Step 4 :-

1) i=3, w=0

V(3,0)=0

2) i=3, w=1

V(3,1)=0

3) i=3, w=2

V(3,2)=20

4) i=3, w=3

V(3,3)=100

5) i=3, w=4

V(3,4)=100

6) i=3, w=5

V(3,5)=120

Step 5 :-

1) i=4, w=0

V(4,0)=0

2) i=4, w=1

V(4,1)=40

3) i=4, w=2

V(4,2)=40

4) i=4, w=3

V(4,3)=100

5) i=4, w=4

V(4,4)=140

6) i=4, w=5

V(4,5)=140

∴ Maximum value earned

∴ Max value

= V(n,w)

= V[4,5]

= 140

(42)

→ Items that were put inside the knap-sack are found using the following rule.

```
set i=n and w=W
while i and w>0 do
  if (V[i,w] != V[i-1,w]) then
    mark the i th item.
    Set w=w-wt[i]
    Set i=i-1
  else
     set i=i-1
  endif
  endwhile.
```

Step 1 :-

V[i,w] != V[i-1,w]

max value = 140 ∵ V(4,5)

i=4 , w=5

V[4,5] != V[4-1,5]

V[4,5] != V[3,5]

140 != 120 [YES]

∴ Mark the 4th items indicated as '1'.

Step 2 :-

set w=w-wt[i]

w=5-wt[4]

=5-1

$\boxed{w=4}$

set i=i-1

i=4-1

$\boxed{i=3}$

(43)

∴ Now $i = 3$, $w = 4$

$v[3,4] \;!= v[3-1,4]$

$= v[2,4]$

$100 \;!= 100$ [NO]

Don't mark the 3^{rd} item, indicated as '0'.

step 3:-

set $i = i - 1$, $i = 3 - 1$, $\boxed{i = 2}$

$v[2,4] \;!= v[2-1,4]$

$= v[1,4]$

$100 \;!= 100$ [NO]

Don't mark the 2^{nd} item indicated as '0'.

step 4:-

Set $i = i - 1$

$i = 2 - 1$

$\boxed{i = 1}$

$v[1,4] \;!= v[1-1,4]$

$v[1,4] \;!= v[0,4]$

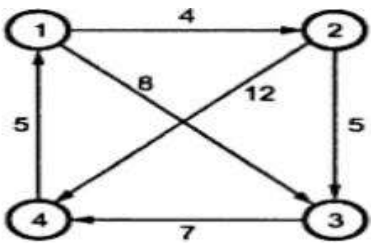
$100 \;!= 0$ [True]

So, Mark the 1^{st} item, indicated as '1'.

Conclusion:-

→ So item we are putting inside the Knapsack are 4 & 1.

9	**Construct an algorithm for All pairs of shortest path and calculate shortest path between all pairs of vertices by using dynamic programming method for the following graph.**	**[L6][CO3]**	**[12M]**



All pair shortest path algorithm

1. In all pair shortest path, when a weighted graph is represented by its weight matrix W then objective is to find the distance between every pair of nodes.
2. We will apply dynamic programming to solve the all pairs shortest path.
3. In all pair shortest path algorithm, we first decomposed the given problem into sub problems.
4. In this principle of optimally is used for solving the problem.
5. It means any sub path of shortest path is a shortest path between the end nodes.

Steps:

i. Let Ai,j be the length of shortest path from node i to node j such that the label for every intermediate node will be ≤ k.
ii. Now, divide the path from i node to j node for every intermediate node, say 'k' then there arises two case

a. Path going from i to j via k. b. Path which is not going via k.

iii. Select only shortest path from two cases.

iv. Using recursive method we compute shortest path.

v. Initially: A0=W[i,j
vi. Next computations: Aki,j=minAk−1i,j,Ak−1i,k,Ak−1k,j

```
Algorithm All_pair(W, A)
{
For i = 1 to n do
For j = 1 to n do
A [i , j] = W [i , j]
For k = 1 to n do
        {
        For i = 1 to n do
                {
                For j = 1 to n do
                {
                A [i , j] = min(A [i, j], A [i, k] + A [k, j])
                }
                }}}
```

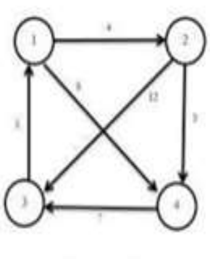
Algorithm:

Analysis of Algorithm:

i. The first double for loop takes O (n2) time.

ii. The nested three for loop takes O (n3) time.

iii. Thus, the whole algorithm takes O (n3) time.



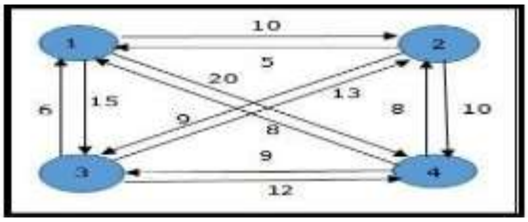
Solution:

$$A^0=\begin{bmatrix} 0 & 4 & 8 & \infty \\ \infty & 0 & 5 & 12 \\ \infty & \infty & 0 & 7 \\ 5 & \infty & \infty & 0 \end{bmatrix} \quad A^1=\begin{bmatrix} 0 & 4 & 8 & \infty \\ \infty & 0 & 5 & 12 \\ 12 & \infty & 0 & 7 \\ 5 & 9 & 13 & 0 \end{bmatrix} \quad A^2=\begin{bmatrix} 0 & 4 & 8 & 16 \\ 17 & 0 & 5 & 12 \\ 12 & \infty & 0 & 7 \\ 5 & 9 & 13 & 0 \end{bmatrix} \quad A^3=\begin{bmatrix} 0 & 4 & 8 & 16 \\ 17 & 0 & 5 & 12 \\ 12 & 16 & 0 & 7 \\ 5 & 9 & 13 & 0 \end{bmatrix}$$

Thus the shortest distances between all pair are obtained.

Course Code: 20CS0523 **R20**

10 **Analyze the minimum cost tour for given problem in travelling sales person Concepts by using dynamic programming.** [L4][CO3] [12M]



THE TRAVELLING SALES PERSON PROBLEM :-

→ It is one of the algorithm strategy used in dynamic programming.

→ Here the salesman should start ~~at~~ at a point and travels all the places and comes back to starting point.

→ The main objective of the problem is to minimize the travelling cost.

→ The main requirement is there should be communication between nodes.

Formula for calculating the cost adjacency matrix in dynamic programming is,

$$g(i,s) = \left\{ \min_{j \in s} \{ c_{ij} + g(j, s-\{j\}) \} \right\}$$

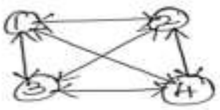
(80)

$g(i,s)$ → length of shortest path starting at vertex i, going through all vertices in s & terminating at vertex 1.

$g\{1, v-\{1\}\}$ is the length of an optimal salesperson tour.

Example :-

For the following graph find minimum cost four for the travelling salesperson problem.



The cost adjacency matrix

$$= \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Let the cost adjacency matrix

$$c_{ij} = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix} \end{array}$$

→ Let us start the tour from vertex 1.

formula :-

$$g(i,s) = \min\{c_{ij} + g(j, s-\{j\})\} \rightarrow ①$$

clearly,

$$g(1,\phi) = c_{11} = 0$$
$$g(2,\phi) = c_{21} = 5$$
$$g(3,\phi) = c_{31} = 6$$

(51)

$g(4, \phi) = C_{41} = 8$

Using equation 1 we obtain

$g(1, \{2,3,4\}) = \min\{C_{12} + g(2, \{3,4\}),$
$C_{13} + g(3, \{2,4\}),$
$C_{14} + g(4, \{2,3\})\}$

$g(2, \{3,4\}) = \min\{C_{23} + g(3, \{4\}),$
$C_{24} + g(4, \{3\})\}$

$g(3, \{4\}) = \min\{C_{34} + g(4, \phi)\}.$
$= \min\{12 + 8\}$
$= 20$

$g(4, \{3\}) = \min\{C_{43} + g(3, \phi)\}$
$= 9 + 6$
$= 15$

Therefore calculate the value for $g(2, \{3,4\})$

$g(2, \{3,4\}) = \min\{C_{23} + g(3, \{4\}), C_{24} + g(4, \{3\})\}$
$= \min\{9 + 20, 10 + 15\}$
$= \min\{29, 25\}$
$g(2, \{3,4\}) = 25$

Therefore, $g(3, \{2,4\})$

$g(3, \{2,4\}) = \min\{C_{32} + g(2, \{4\}),$
$C_{34} + g(4, \{2\})\}.$

$g(2, \{4\}) = \min\{C_{24} + g(4, \phi)\}$

(52)

$= 10 + 8$
$= 18$

$g(4, \{2\}) = \min\{C_{42} + g(2, \phi)\}$
$= 8 + 5$
$= 13$

$g(3, \{2,4\}) = \min\{13 + 18, 12 + 13\}$
$= \min\{31, 25\}$
$= 25$

Therefore $g(4, \{2,3\})$

$g(4, \{2,3\}) = \min\{C_{42} + g(2, \{3\}),$
$C_{43} + g(3, \{2\})\}.$

$g(2, \{3\}) = \min\{C_{23} + g(3, \phi)\}$
$= \min\{9 + 6\}$
$= 15$

$g(3, \{2\}) = \min\{C_{32} + g(2, \phi)\}$
$= 13 + 5$
$= 18$

$g(4, \{2,3\}) = \min\{8 + 15, 19 + 18\}$
$= \min\{23, 27\}$
$g(3, \{2,4\}) = 23$

$\therefore g(1, \{2,3,4\}) = \min\{C_{12} + g(2, \{3,4\}),$
$C_{13} + g(3, \{2,4\}),$
$C_{14} + g(4, \{2,3\})\}$
$= \min\{10 + 25, 15 + 25,$
$20 + 23\}$

	$= \min\{35, 40, 43\}$ $g(1, \{2, 3, 4\}) = 35$ ∴ The optimal tour for the graph has length = 35. (or) minimum cost tour for the travelling Salesperson problem is 35. ∴ The best optimal toue path is $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$		

UNIT –IV
BACKTRACKING, BRANCH AND BOUND

1	**Distinguish in detail 8-queens problem using backtracking with state space tree.**	[L4][CO4]	[12M]

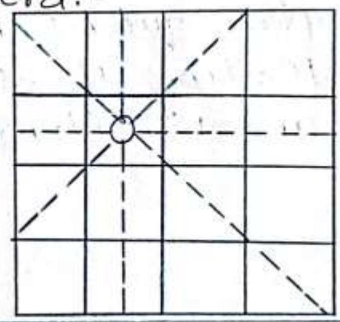
8-QUEEN PROBLEM:-

Problem Statement:-

→ The n queen's problem can be stated as follows.
→ Consider a nxn chessboard on which, we have to place 'n' queens. So that no two queens attack each other by being in the same row (or) in the same column (or) on the same diagonal.

For example:-

Consider 4x4 board:-



The next Queen - if is placed on the paths marked by dotted lines, then they can attack each other.

2 Queen's problem is not solvable:-

→ Because 2-queens can be placed on 2x2 chessboard as,

Q	Q

illegal

Q	
Q	

illegal

Q	
	Q

illegal

Q	Q

illegal

	Q
	Q

illegal

But 4-queen's problem is solvable:-

		Q	
Q			
			Q
	Q		

← No two queens can attack each other.

How to solve n-queen's problem?

Let us take 4-queen's & 4x4 chessboard.

Step 1:- Now we start with empty chessboard.

Step 2:- place queen 1 in the first possible position of its row i.e 1st row and 1st column.

Q			

Step 3:- Then place queen 2 after trying unsuccessful place - 1 (1,2), (2,1), (2,2) at (2,3) we can place it i.e 2nd row & 3rd column.

Q			
		Q	

Step 4:- This is the dead end. Because a 3rd queen cannot be placed in next column. As there is no acceptable position for queen 3. Hence algorithm backtracks and places the 2nd queen at (2,4) position.

8-queens problem:-

(4 6 8 2 7 1 3 5)

			Q				
					Q		
							Q
	Q						
						Q	
Q							
		Q					
				Q			

Solution 1

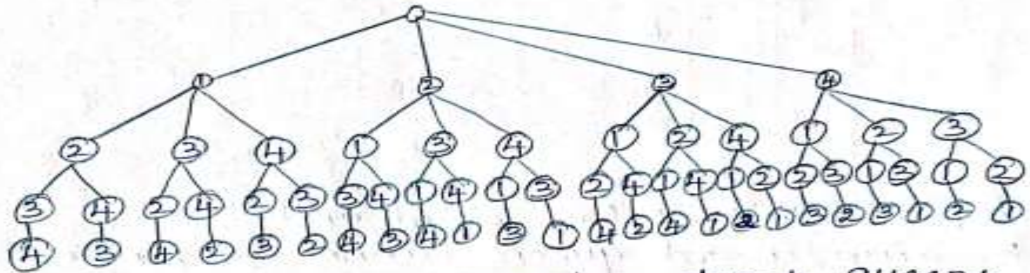
(5 3 1 7 2 8 6 4)

				Q			
		Q					
Q							
						Q	
	Q						
							Q
					Q		
			Q				

Solution 2

→ Formula to check

$$|i-k| = |j-l|$$



sample state space tree for 4-queens problem.

→ solution representation using state space tree for 8-queens problem.

$$1+\sum_{j=0}^{7}\left[\prod_{i=0}^{j}(8-i)\right] = 69821 \text{ (Total nodes)}$$

(To find how many number of nodes in state space tree).

2	Explain sum of subsets by using backtracking with an example.	[L5][CO4]	[12M]

SUM OF SUBSETS:-

Problem statement:-

→ Let $S = \{S_1, \ldots, S_n\}$ be a set of n positive integers, then we have to find the subset whose sum is equal to given positive integer d.

→ It is always convenient to sort the set's elements in ascending order. That is,

$$S_1 \leq S_2 \leq \ldots \leq S_n$$

→ Let us first write an general algorithm for sum of subset problem.

Algorithm:-

→ Let S be a set of elements and d is the expected sum of subsets. Then,

Step 1:- start with a empty set.

Step 2:- Add to the subset, the next element from the list.

Step 3:- If the subset is having sum d then stop with that subset as solution.

Step 4:- If the subset is not feasible (or) if we have reached to end of the set then backtrack through the subset until we find the most suitable value.

Step 5:- If the subset is feasible then repeat step 2.

Step 6:- If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

Example:-

consider a set $S = \{5, 10, 12, 13, 15, 18\}$ and $d = 30$. Solve it for obtaining sum of subset.

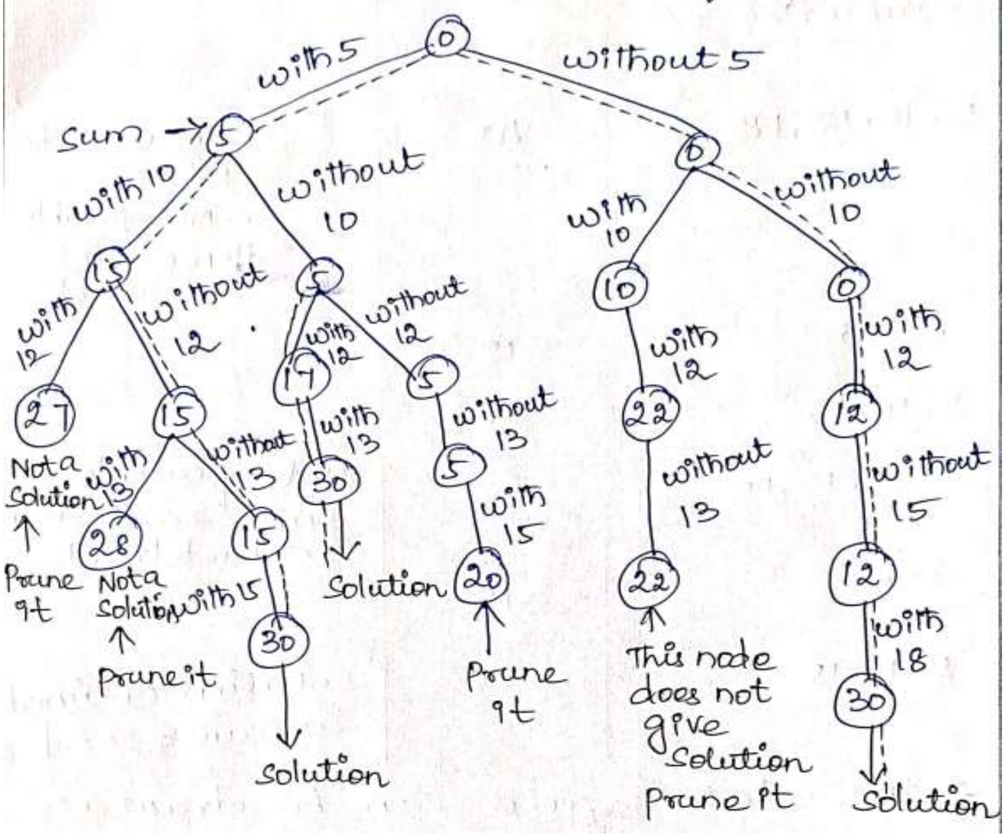
Initially subset { }	sum = 0	—
5	5	Then add next element
5, 10	15 ∴ 15 < 30	Add next element
5, 10, 12	27 ∴ 27 < 30	Add next element
5, 10, 12, 13	40 ∴ 40 > 30	sum exceeds d = 30 Hence backtrack
5, 10, 12, 15	42	Sum exceeds d = 30 ∴ Backtrack applying

Initially subset={ }	sum=0	—
5, 10, 12, 18	45	sum exceeds d. ∴ Not feasible Hence back track.
5, 10	15	
5, 10, 13	28	
5, 10, 13, 15	33	Not feasible Sum exceeds d=30 ∴ backtrack.
5, 10	15	
5, 10, 15	30	Solution obtained as sum = 30 = d

→ The state space can be draw as follows.

{5, 10, 12, 13, 15, 18}

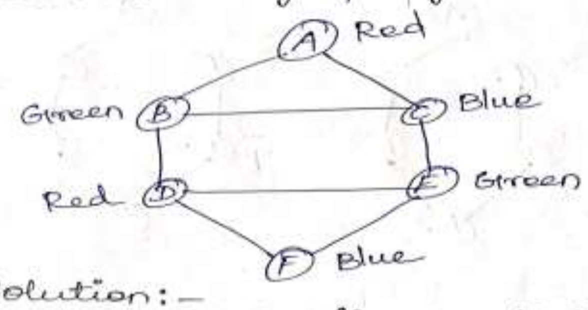
State space tree for sum of subset:-



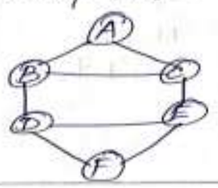
3 a) Recall the graph coloring. Explain in detail about graph coloring with an example. [L5][CO4] [9M]

- Graph Coloring is a problem of coloring each vertex in graph in such a way that no two adjacent vertices have same color and yet m-colors are used.
- This problem is also called as m-coloring problem. If the degree of given graph is d then we can color it with d+1 colors.

For example:-
consider a graph given.



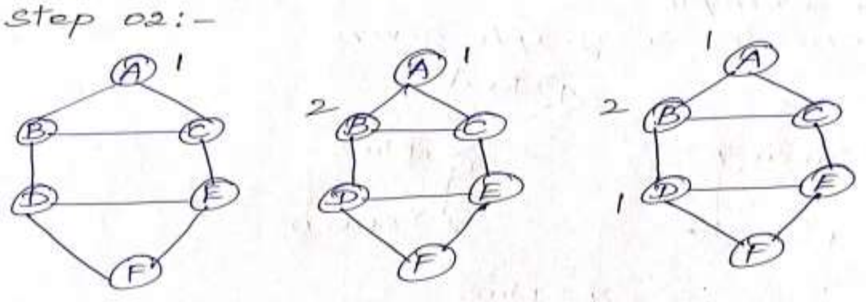
Solution:-
→ As given in figure, we require three colors to color the graph. Hence the chromatic number of given graph is '3'. We can use backtracking technique to solve the graph coloring problem as follows.
Step 01:-

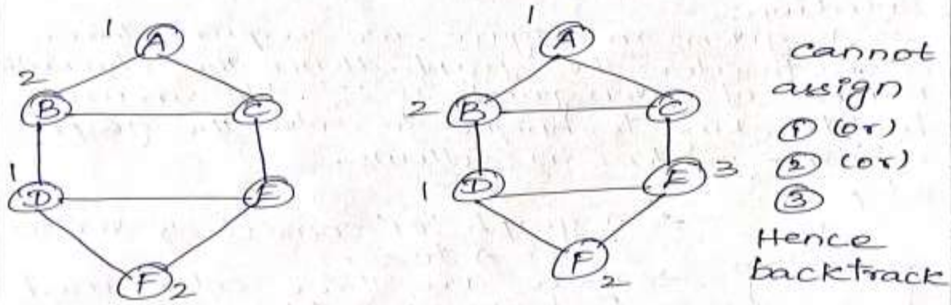


→ A graph 'G' consists of vertices from A to f.
→ There are three colors used Red, Green & Blue.
→ we will number them out.

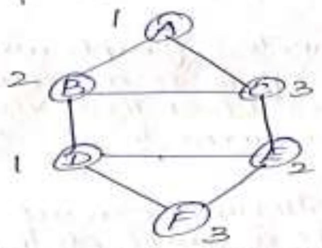
→ That means.
1 indicates → Red
2 indicates → Green
3 indicates → Blue color.

Step 02:-



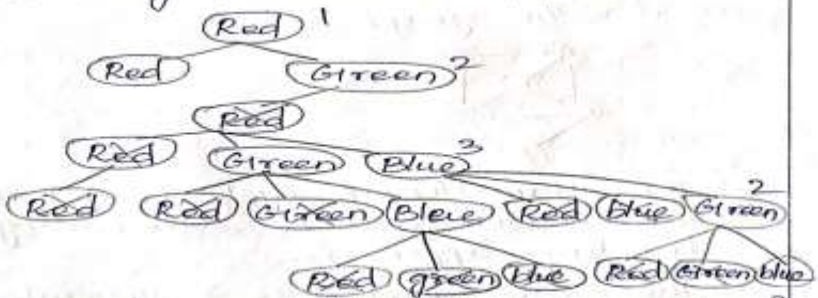


step 03:-



→ Thus the graph coloring problem is solved.

→ The state space tree can be drawn for better understanding of graph coloring technique using backtrack approach.



∴ Here we have assumed, color index Red = 1, Green = 2, Blue = 3.

b) **Discuss about General method of backtracking** **[L3][CO4]** **[3M]**

GENERAL METHOD:-

→ Backtracking is one of the most general technique.

→ In this technique, we search for the set of solutions (or) optimal solution which satisfies some constraints.

→ One way of solving a problem is by exhaustive search, we enumerate all possible solutions and see which one produces the optimum result.

For example: Knapsack problem.

→ We look at every possible subset objects and find out one which has the greatest profit value and at the same time not greater than the weight bound.

→ Backtracking is a variation of exhaustive search, where the search is refined by eliminating certain possibilities.

→ Backtracking is usually faster method than an exhaustive search.

→ In the backtracking method,

1. The desired solution is expressible as an 'n' tuple $(x_1, x_2, \ldots x_n)$ where x_i is chosen from some finite set S_i.
2. The solution maximizes (or) minimizes (or) satisfies a criterion from function $C(x_1, x_2, \ldots, x_n)$.

→ The problem can be categorized into three categories.

1. for Instance:-

   → for a problem p let c be the set of constraints for p. Let D be the set containing all solutions satisfying c then.
2. Finding whether there is any feasible solution? - is the decision problem.
3. What is the best solution? - is the optimization problem.

→ The basic idea of backtracking is to build up a vector, one component at a time & to test whether the vector being formed has any chance of success.

→ The major advantage of this algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be ignored.

→ Backtracking algorithm determines the solution by systematically searching the solution space. (i.e, set of all feasible solutions) for the given problem.

→ Backtracking is a depth first search with some bounding function. All solutions using backtracking are required to satisfy a complex set of all constraints.

→ The constraints may be explicit (or) implicit.

→ Explicit constraints are rules, which restrict each vector element to be chosen from the given set.

→ Implicit constraints are rules, which determine which tuples in the solution space, actually satisfy the criterion function.

4	**Discuss the Hamiltonian cycle algorithm with step by step operation with example.**	[L6][CO4]	[12M]

Definition:-

Let $G=(V,E)$ be a connected graph with n Vertices. A Hamiltonian cycle is a round trip path along n-edges of G that visits every vertex once and returns to its starting position.

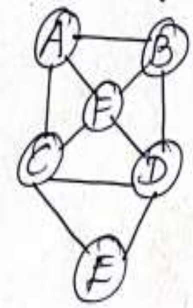
→ It is called the Hamiltonian circuit.

→ Hamiltonian circuit is a graph cycle (i.e, closed loop) through a graph that visits each node exactly once.

→ A graph possessing a Hamiltonian cycle is said to be Hamiltonian graph.

For example:-

Consider the graph G.



→ The Hamiltonian cycle is A-B-D-E-C-F-A.

→ This problem can be solved using back tracking approach.

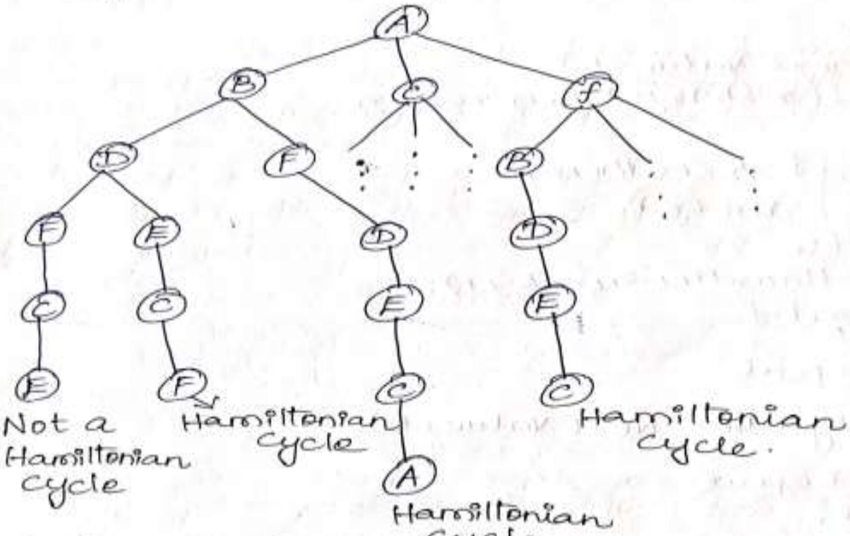
→ The state space tree is generated in order to find all the Hamiltonian cycles in the graph.

→ Only distinct cycles are output of this algorithm.

→ Hamiltonian cycle can be identified as follows.

State space tree for finding Hamiltonian cycle:-

→ In below figure clearly the backtracking approach is adopted.



→ For instance A - B - D - F - C - E ; here we get stuck.

→ Hence we backtrack and from 'D' node another path is choosen.

→ A - B - D - E - C - F - A which is Hamiltonian cycle.

~~Exa~~ Algorithm:-

```
Algorithm Hamiltonian(K)
{
 loop
  next value(K)
  if(x(K)=0) then return
  {
    if K=n then
    print(x)
  Else
    Hamiltonian(K+1);
  Endif
 }
 Repeat
}
Algorithm Next Value(K)
{
 Repeat
 {
   x(K)= (x[K+1])mod (n+1);
   if (x[K]=0) then return
   if (G[x(K-1)], x(K) ≠ 0]) then
   {
   for j=1 to K-1 do
```

if [x(j) = x(k)] then
break
if (j=k) then
if ((k<n) (k=n) and G[x(n), x(1)] ≠0)
then return
}
until false
}

5 Give brief description about the general method of branch and bound. [L2][CO4] [6M]

GENERAL METHOD:-

→ Branch and Bound (B & B) is general algorithm (or systematic method) for finding optimal solution of various optimization problems, especially in discrete and combinatorial optimization.

→ The Branch and Bound is very similar to backtracking in that a state space tree is used to solve a problem.

→ The differences are that the B & B method
1. Does not limit us to any particular way of traversing the tree.
2. It is used only for optimization problem.
3. It is applicable to a wide variety of discrete combinatorial problem.

→ Branch and Bound is rather general optimization technique that applies where the greedy method and dynamic programming fail.

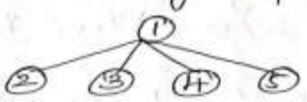
→ It is much slower, indeed, it often leads to exponential time complexities in the worst case.

→ In term B & B refers to all state space search methods in which all children of the "E-node" are generated before any other "live node" can become the "E-node".

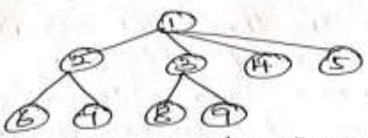
Live node:- Live node is a node that has been generated but whose children have not yet been generated.

E-node:- E-node is a live node whose children are currently being explored.

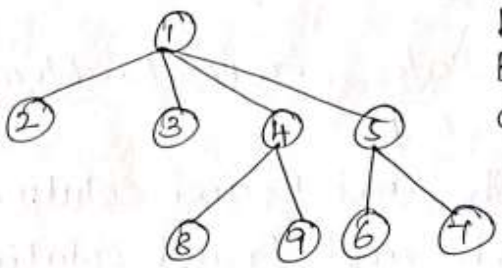
Dead node:- Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already expanded.



Live node: 2, 3, 4 and 5



FIFO Branch & Bound (BFS) children of Enode are inserted in a queue.



LIFO Branch & Bound (D-Search) children of E-node are inserted in a stack.

→ Two graph search strategies, BFS and D-search (DFS) in which the exploration of a new node cannot begin until the node currently being explored is fully explored.

→ Both BFS & D-search (DFS) generalized to B & B strategies.

BFS:- Like state space search will be called FIFO (First In First Out) search as the list of live nodes is "First-in-first-out" list (or queue).

D-search (DFS):- Like state space search will be called LIFO (Last In First out) search as the list of live nodes is a "last-in-first-out" list (or stack).

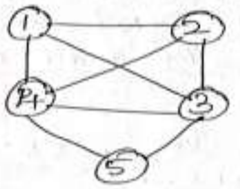
→ In backtracking, bounding function are used to help avoid the generation of sub-trees that do not contain an answer node.

→ We will use 3-types of search strategies in branch and bound.

1) FIFO (First In First Out) search.
2) LIFO (Last In First Out) Search.
3) LC (Least count) search.

6 Find the LC branch and bound solution for the traveling sale person problem whose cost matrix is as follows: [L4][CO4] **[12M]**

$$\begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \end{array}$$



Cost adjacency matrix

$$\begin{array}{c|c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \end{array}$$

Step 01:-

Row Reduction:-

→ Note down the minimum value as per row wise and subtract it

$$\begin{array}{c|c|c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} & \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} & \begin{array}{c} 10 \\ 2 \\ 2 \\ 3 \\ 4 \end{array} \end{array} \Rightarrow \begin{array}{c|c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix} \end{array}$$

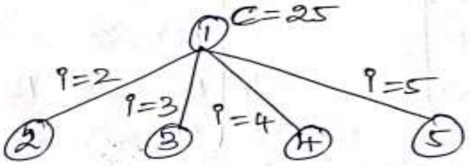
Column Reduction:-

→ Note down the minimum value as per column wise and subtract it.

$$\begin{array}{c|c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix} \\ & \begin{array}{ccccc} 1 & 0 & 3 & 0 & 0 \end{array} \end{array} \Rightarrow \begin{array}{c|c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix} \end{array}$$

∴ Total amount subtracted, $r = 21 + 4 = 25$

State space tree for the 1st node



Step 02:-

→ consider the path (1,2): change all entries of first row and second column of reduced matrix to ∞ & set A(2,1) to ~~do~~ ∞.

$$\begin{array}{c|c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix} \end{array}$$

Row Reduction:-

$$\begin{array}{r} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\ \left[\begin{array}{ccccc}\infty&\infty&\infty&\infty&\infty\\ \infty&\infty&11&2&0\\ 0&\infty&\infty&0&2\\ 15&\infty&12&\infty&0\\ 11&\infty&0&12&\infty\end{array}\right]\end{array}\begin{array}{c} \\ \\ 0\\0\\0\\0\end{array} \Rightarrow \begin{array}{r} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\ \left[\begin{array}{ccccc}\infty&\infty&\infty&\infty&\infty\\ \infty&\infty&11&2&0\\ 0&\infty&\infty&0&2\\ 15&\infty&12&\infty&0\\ 11&\infty&0&12&\infty\end{array}\right]\end{array}$$

Column Reduction:-

$$\begin{array}{r} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ \\ \end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\ \left[\begin{array}{ccccc}\infty&\infty&\infty&\infty&\infty\\ \infty&\infty&11&2&0\\ 0&\infty&\infty&0&2\\ 15&\infty&12&\infty&0\\ 11&\infty&0&12&\infty\end{array}\right]\\ \begin{array}{ccccc}0&&0&0&0\end{array}\end{array} \Rightarrow \begin{array}{r} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\ \left[\begin{array}{ccccc}\infty&\infty&\infty&\infty&\infty\\ \infty&\infty&11&2&0\\ 0&\infty&\infty&0&2\\ 15&\infty&12&\infty&0\\ 11&\infty&0&12&\infty\end{array}\right]\end{array}$$

$\therefore$ Total amout reduced $\hat{r} = 0+0 = 0$

$$\hat{c}(2) = c(1,2) + c(1) + \hat{r}$$
$$= 10 + 25 + 0$$
$$= 35$$

step 03:-

→ Consider the path (1, 3):- change all entries of first row and third column of reduced matrix to ∞ and set A(3, 1) to ∞.

$$\begin{array}{r} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\ \left[\begin{array}{ccccc}\infty&\infty&\infty&\infty&\infty\\ 12&\infty&\infty&2&0\\ \infty&3&\infty&0&2\\ 15&3&\infty&\infty&0\\ 11&0&\infty&12&\infty\end{array}\right]\end{array}$$

Row Reduction:-

$$\begin{array}{r} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\ \left[\begin{array}{ccccc}\infty&\infty&\infty&\infty&\infty\\ 12&\infty&\infty&2&0\\ \infty&3&\infty&0&2\\ 15&3&\infty&\infty&0\\ 11&0&\infty&12&\infty\end{array}\right]\end{array}\begin{array}{c} \\ \\ 0\\0\\0\\0\end{array} \Rightarrow \begin{array}{r} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\ \left[\begin{array}{ccccc}\infty&\infty&\infty&\infty&\infty\\ 12&\infty&\infty&2&0\\ \infty&3&\infty&0&2\\ 15&3&\infty&\infty&0\\ 11&0&\infty&12&\infty\end{array}\right]\end{array}$$

Column Reduction:-

$$\begin{array}{r} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ \\ \end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\ \left[\begin{array}{ccccc}\infty&\infty&\infty&\infty&\infty\\ 12&\infty&\infty&2&0\\ \infty&3&\infty&0&2\\ 15&3&\infty&\infty&0\\ 11&0&\infty&12&\infty\end{array}\right]\\ \begin{array}{ccccc}11&0&&0&0\end{array}\end{array} \Rightarrow \begin{array}{r} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c}\begin{array}{ccccc}1&2&3&4&5\end{array}\\ \left[\begin{array}{ccccc}\infty&\infty&\infty&\infty&\infty\\ 1&\infty&\infty&2&0\\ \infty&3&\infty&0&2\\ 4&3&\infty&\infty&0\\ 0&0&\infty&12&\infty\end{array}\right]\end{array}$$

$\therefore$ Total amount reduced $\hat{r} = 11$

$$\hat{c}(3) = \hat{c}(1) + c(1,3) + \hat{r}$$
$$= 25 + 17 + 11$$
$$= 53$$

Step 04 :-

→ consider the path (1,4) : change all entries of first row and fourth column to '∞' and set $A(4,1)$ to ∞.

1^{st} matrix Reduced :

$$\begin{array}{cc} & \begin{array}{ccccc}1&2&3&4&5\end{array}\\ \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\12&\infty&11&\infty&0\\0&3&\infty&\infty&2\\\infty&3&12&\infty&0\\11&0&0&\infty&\infty\end{bmatrix}\end{array}$$

Row Reduction:-

$$\begin{array}{ccc} & \begin{array}{ccccc}1&2&3&4&5\end{array} & \\ \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\12&\infty&11&\infty&0\\0&3&\infty&\infty&2\\\infty&3&12&\infty&0\\11&0&0&\infty&\infty\end{bmatrix} & \begin{array}{c}\\0\\0\\0\\0\end{array}\end{array} \Rightarrow \begin{array}{cc} & \begin{array}{ccccc}1&2&3&4&5\end{array}\\ \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\12&\infty&11&\infty&0\\0&3&\infty&\infty&2\\\infty&3&12&\infty&0\\11&0&0&\infty&\infty\end{bmatrix}\end{array}$$

Column Reduction:-

$$\begin{array}{cc} & \begin{array}{ccccc}1&2&3&4&5\end{array}\\ \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\12&\infty&11&\infty&0\\0&3&\infty&\infty&2\\\infty&3&12&\infty&0\\11&0&0&\infty&\infty\end{bmatrix}\\ & \begin{array}{ccccc}0&0&0&&0\end{array}\end{array} \Rightarrow \begin{array}{cc} & \begin{array}{ccccc}1&2&3&4&5\end{array}\\ \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\12&\infty&11&\infty&0\\0&3&\infty&\infty&2\\\infty&3&12&\infty&0\\11&0&0&\infty&\infty\end{bmatrix}\end{array}$$

∴ Total subtracted reduction $\hat{r} = 0$

$$\hat{C}(4) = C(1,4) + \hat{C}(1) + \hat{r}$$
$$= 0 + 25 + 0$$
$$= 25$$

Step 05 :-

consider the path (1,5): change all entries of first row and 5^{th} column to ∞ and set $A(5,1)$ to ∞

$$\begin{array}{cc} & \begin{array}{ccccc}1&2&3&4&5\end{array}\\ \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\12&\infty&11&2&\infty\\0&3&\infty&0&\infty\\15&3&12&\infty&\infty\\\infty&0&0&12&\infty\end{bmatrix}\end{array}$$

Row Reduction:-

$$\begin{array}{ccc} & \begin{array}{ccccc}1&2&3&4&5\end{array} & \\ \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\12&\infty&11&2&\infty\\0&3&\infty&0&\infty\\15&3&12&\infty&\infty\\\infty&0&0&12&\infty\end{bmatrix} & \begin{array}{c}\\2\\0\\3\\0\end{array}\end{array} \Rightarrow \begin{array}{cc} & \begin{array}{ccccc}1&2&3&4&5\end{array}\\ \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\10&\infty&9&0&\infty\\0&3&\infty&0&\infty\\12&0&9&\infty&\infty\\\infty&0&0&12&\infty\end{bmatrix}\end{array}$$

Column Reduction:-

$$\begin{array}{cc} & \begin{array}{ccccc}1&2&3&4&5\end{array}\\ \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\10&\infty&9&0&\infty\\0&3&\infty&0&\infty\\12&0&9&\infty&\infty\\\infty&0&0&12&\infty\end{bmatrix}\\ & \begin{array}{ccccc}0&0&0&0&\end{array}\end{array} \Rightarrow \begin{array}{cc} & \begin{array}{ccccc}1&2&3&4&5\end{array}\\ \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\10&\infty&9&0&\infty\\0&3&\infty&0&\infty\\12&0&9&\infty&\infty\\\infty&0&0&12&\infty\end{bmatrix}\end{array}$$

∴ Total subtracted Reduction

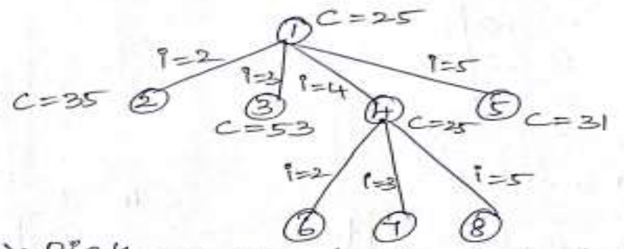
$\hat{r} = 5$

$\hat{c}(5) = c(1,5) + \hat{c}(1) + \hat{r}$

$= 1 + 25 + 5$

$\hat{c}(5) = 31$

State space Tree :-



→ Pick up the least cost from the state space tree and explore the child nodes for the node (4).

Reduced cost matrix for the node 4 is:-

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & 12 & \infty & 11 & \infty & 0 \\ 3 & 0 & 3 & \infty & \infty & 2 \\ 4 & \infty & 3 & 12 & \infty & 0 \\ 5 & 11 & 0 & 0 & \infty & \infty \end{array}$$

Step 06 :-

→ consider the path (4,2): change all entries of 4^{th} row and 2^{nd} column from reduced matrix as ∞ and set (2,1) to ∞.

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & \infty & 11 & \infty & 0 \\ 3 & 0 & \infty & \infty & \infty & 2 \\ 4 & \infty & \infty & \infty & \infty & \infty \\ 5 & 11 & \infty & 0 & \infty & \infty \end{array}$$

Row Reduction:-

$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline 1 & \infty & \infty & \infty & \infty & \infty & \\ 2 & \infty & \infty & 11 & \infty & 0 & 0 \\ 3 & 0 & \infty & \infty & \infty & 2 & 0 \\ 4 & \infty & \infty & \infty & \infty & \infty & \\ 5 & 11 & \infty & 0 & \infty & \infty & 0 \end{array} \Rightarrow \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & \infty & 11 & \infty & 0 \\ 3 & 0 & \infty & \infty & \infty & 2 \\ 4 & \infty & \infty & \infty & \infty & \infty \\ 5 & 11 & \infty & 0 & \infty & \infty \end{array}$$

Column Reduction:-

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & \infty & 11 & \infty & 0 \\ 3 & 0 & \infty & \infty & \infty & 2 \\ 4 & \infty & \infty & \infty & \infty & \infty \\ 5 & 11 & \infty & 0 & \infty & \infty \\ \hline & 0 & & 0 & & 0 \end{array} \Rightarrow \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & \infty & 11 & \infty & 0 \\ 3 & 0 & \infty & \infty & \infty & 2 \\ 4 & \infty & \infty & \infty & \infty & \infty \\ 5 & 11 & \infty & 0 & \infty & \infty \end{array}$$

∴ Total subtracted reduction $\hat{r} = 0$

$\hat{C}(6) = C(4,2) + \hat{C}(4) + \hat{r}$

$= 3 + 25 + 0$

$\hat{C}(6) = 28$

step 07:-

→ consider the path (4,3): change all the entries of 4^{th} row and 3^{rd} column as ∞ and set (3,1) to ∞.

$$\begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} \end{array}$$

Row Reduction:-

$$\begin{array}{c c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} & \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} & \begin{array}{c} \\ 0 \\ 2 \\ \\ 0 \end{array} \end{array} \Rightarrow \begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} \end{array}$$

column Reduction:-

$$\begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ \\ \end{array} & \begin{array}{c} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} \\ \begin{array}{ccccc} 11 & 0 & & & 0 \end{array} \end{array} \end{array} \Rightarrow \begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix} \end{array}$$

∴ Total subtracted reduction

$\hat{r} = 11 + 2 = 13$

$\hat{C}(7) = C(4,3) + \hat{C}(4) = \hat{r}$

$= 12 + 25 + 13$

$= 50$

step 08:-

→ consider the path (4,5): change all entries of 4^{th} row and 5^{th} column to ∞ and set (5,1) to ∞.

$$\begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \end{array}$$

Row Reduction:-

$$\begin{array}{c c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} & \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} & \begin{array}{c} \\ 11 \\ 0 \\ \\ 0 \end{array} \end{array} \Rightarrow \begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \end{array}$$

Column Reduction:-

$$\begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \end{array} \Rightarrow \begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \end{array}$$

∴ Total subtracted reduction

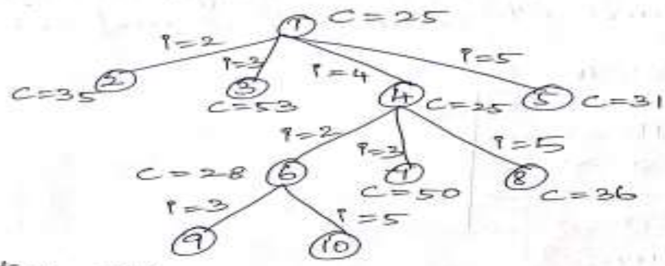
$\hat{r} = 11 + 0 = 11$

$\hat{c}(8) = c(4,5) + \hat{c}(4) + \hat{r}$

$= 0 + 25 + 11$

$\hat{c}(8) = 36$

state space tree:-



step 09:-

→ consider the path (2,3): change all entries of 2nd row and 3rd column to ∞ and set (3,1) to ∞

$$\begin{array}{cc} & \begin{matrix}1&2&3&4&5\end{matrix} \\ \begin{matrix}1\\2\\3\\4\\5\end{matrix} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&2\\ \infty&\infty&\infty&\infty&\infty\\ 11&\infty&\infty&\infty&\infty\end{bmatrix}\end{array}$$

Row Reduction:-

$$\begin{array}{ccc} & \begin{matrix}1&2&3&4&5\end{matrix} & \\ \begin{matrix}1\\2\\3\\4\\5\end{matrix} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&2\\ \infty&\infty&\infty&\infty&\infty\\ 11&\infty&\infty&\infty&\infty\end{bmatrix} & \begin{matrix}\\ \\2\\ \\11\end{matrix}\end{array} \Rightarrow \begin{array}{cc} & \begin{matrix}1&2&3&4&5\end{matrix} \\ \begin{matrix}1\\2\\3\\4\\5\end{matrix} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&0\\ \infty&\infty&\infty&\infty&\infty\\ 0&\infty&\infty&\infty&\infty\end{bmatrix}\end{array}$$

Column Reduction:-

$$\begin{array}{cc} & \begin{matrix}1&2&3&4&5\end{matrix} \\ \begin{matrix}1\\2\\3\\4\\5\end{matrix} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&0\\ \infty&\infty&\infty&\infty&\infty\\ 0&\infty&\infty&\infty&\infty\end{bmatrix} \\ & \begin{matrix}0&&&&0\end{matrix}\end{array} \Rightarrow \begin{array}{cc} & \begin{matrix}1&2&3&4&5\end{matrix} \\ \begin{matrix}1\\2\\3\\4\\5\end{matrix} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&0\\ \infty&\infty&\infty&\infty&\infty\\ 0&\infty&\infty&\infty&\infty\end{bmatrix}\end{array}$$

∴ Total subtracted reduction $\hat{r} = 13 + 0 = 13$

$\hat{c}(9) = c(2,3) + \hat{c}(6) + \hat{r}$

$= 11 + 28 + 13$

$= 52$

Step 10:-

→ Consider the path (2,5): Change all entries of 2nd row and 5th column to ∞ and set (5,1) to ∞.

$$\begin{array}{cc} & \begin{matrix}1&2&3&4&5\end{matrix} \\ \begin{matrix}1\\2\\3\\4\\5\end{matrix} & \begin{bmatrix}\infty&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&\infty\\ 0&\infty&\infty&\infty&\infty\\ \infty&\infty&\infty&\infty&\infty\\ \infty&\infty&0&\infty&\infty\end{bmatrix}\end{array}$$

Row Reduction :-

$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ 1 & \infty & \infty & \infty & \infty & \infty & \\ 2 & \infty & \infty & \infty & \infty & \infty & \\ 3 & 0 & \infty & \infty & \infty & \infty & 0 \\ 4 & \infty & \infty & \infty & \infty & \infty & \\ 5 & \infty & \infty & 0 & \infty & \infty & 0 \end{array} \Rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix} \end{array}$$

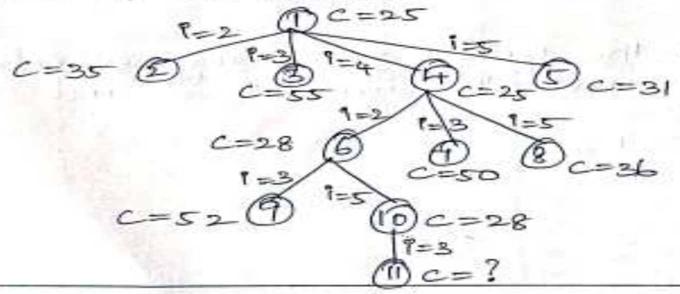
Column Reduction :-

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix} \end{array} \Rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix} \end{array}$$

$\therefore$ Total subtracted reduction $\hat{r} = 0$

$$\hat{c}(10) = c(2,5) + \hat{c}(6) + \hat{r}$$
$$= 0 + 28 + 0$$
$$= 28$$

State space Tree :-



$\therefore$ The minimum cost in the state space tree is $c = 28$ for node 10.

10^{th} matrix $=$

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix} \end{array}$$

Step 11 :-

$\rightarrow$ consider the path $(5,3)$: Change all entries of 5^{th} row and 3^{rd} column to ∞ and set $(3,1)$ to ∞.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix} \end{array}$$

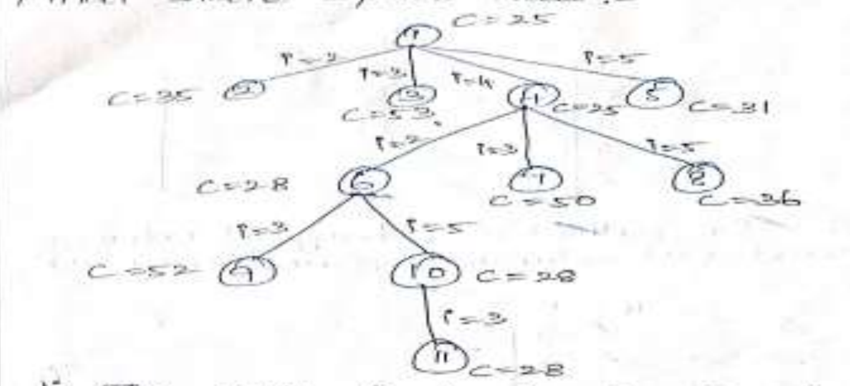
$\therefore$ Row Reduction : '0'

Column Reduction : '0'

$\hat{r} = 0$

$$\hat{c}(11) = c(5,3) + \hat{c}(10) + \hat{r}$$
$$= 0 + 28 + 0$$
$$\hat{c}(11) = 28$$

Final state space Tree :-



∴ The path is 1 – 4 – 2 – 5 – 3 – 1

∴ The minimum tour cost is 28.

7	**Simplify 0/1 knapsack problem and design an algorithm of LC Branch and Bound and find the solution for the knapsack instance of n = 4,(p1, p2, p3, p4) = (10, 10, 12, 18),(w1,w2,w3, w4) =(2, 4,6, 9)and M =15.**	**[L4][CO4]**	**[12M]**

items i	1	2	3	4
Profits	10	10	12	18
Weight	2	4	6	9

Step 1 :- convert the profits to negative.

$(P_1, P_2, P_3, P_4) = (-10, -10, -12, -18)$

Step 2 :-

→ place the first item in bag i.e $W = 2$.

→ calculate the upperbound (u) and cost (c).

$u = -\sum p_i x_i$ (without fraction)

$c = -\sum p_i x_i$ (with fraction)

→ place the second item in bag $w = 4$.

$\therefore 2 + 4 = 6$

→ place the third item in bag i.e $w = 6$

$2 + 4 + 6 = 12$

→ If we place 4^{th} item the capacity exceeds

$\therefore \min z = -P_1 x_1 - P_2 x_2 \cdots - P_n x_n$

for weights

$W_1 x_1 + W_2 x_2 + \cdots + w_n x_n \leq m$

$x_i = 0$ (or) 1

→ Let two functions used $c(x)$ & $u(x)$

$c(x)$ = cost function

$u(x)$ = upper bound function

$c(x) = -\sum p_i x_i$ & $u(x) = -\sum p_i x_i$

$c(x)$ = calculate without fraction.

$u(x)$ = calculate the cost without fraction.

→ Then select the node whose cost is minimum i.e,

$c(x) = \min\{c(\text{lchild}(x)), c(\text{rchild}(x))\}$

→ The problem can be solved by making a sequence of decisions on the variables $x_1, x_2, \ldots x_n$ level wise.

→ A decision on the variable x_i involves determining which of the values 0 (or) 1 is to be assigned to it by defining $c(x)$ recursively.

→ The path from root to the leaf node whose height is maximum is selected & is the solution space for the 0/1 Knapsack problem.

∴ upper bound $u = -(10+10+12)$

$$\boxed{u = -32}$$

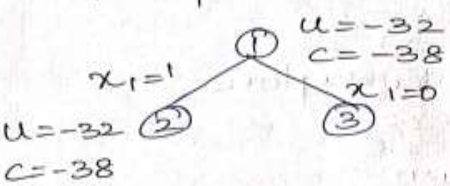
∴ cost $c = -(\sum p_i x_i)$ [with fraction]

$$= -(10+10+12+\cancel{18}^{2}/\cancel{9} \times 3)$$

$$= -(10+10+12+6)$$

$$\boxed{c = -38}$$

State space tree:-



step 3:-

For node 3, $(x_1=0)$

→ If 1st object is not included in the bag, then

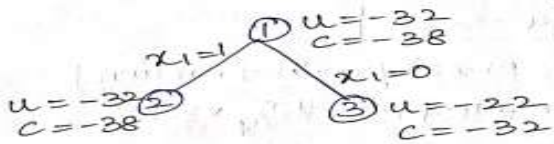
weight (without fraction): $4+6=10$

weight (with fraction): $4+6+5/9$, $4+6+5=15$

$u = -(\sum p_i x_i)$
$= -(10+12)$
$= -22$

$c = -(10+12+18 \times 5/9)$
$= -(10+12+10)$
$= -32$

state space tree



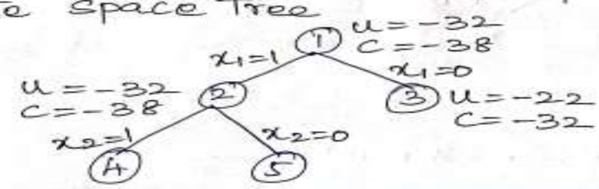
step 4:-

→ Select the minimum cost i.e,

$\min(\hat{c}(2), \hat{c}(3)) = \min\{-38, -32\}$
$= -38$
$= \hat{c}(2)$

∴ Choose the node 2 & explore

state space tree



Step 5 :-

For node 4 ($x_2=1$) :- Included 2nd object

weight (with fraction)	weight (without fraction)
$w = 2+4+6+3/9$	$w=(2,4,6)$
$= 2+4+6+3$	$= 2+4+6$
$\boxed{w=15}$	$\boxed{w=12}$

$c = -\sum p_i x_i$

$= -(10+10+12+\overset{2}{\cancel{18}} \times 3/\cancel{9})$

$= -(10+10+12+6)$

$\boxed{c=-38}$

$u = -\sum p_i x_i$

$= -(10+10+12)$

$\boxed{u=-32}$

For node 4 ($x_2=0$) :- If 2nd object not included

weight (with fraction)	weight (without fraction)
$w=(2,6,7/9)$	$w=(2,6)$

$c = -\sum p_i x_i$

$= -(10+12+\overset{2}{\cancel{18}} \times 7/\cancel{9})$

$= -(10+12+14)$

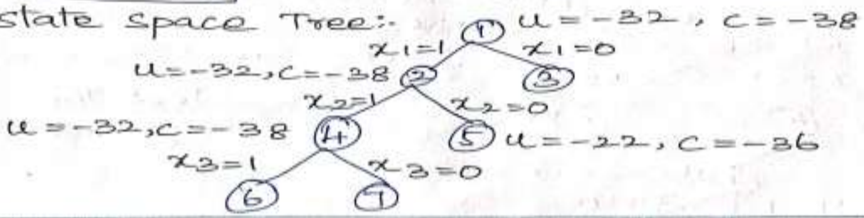
$\boxed{c=-36}$

$u = -\sum p_i x_i$

$= -(10+12)$

$\boxed{u=-22}$

state space Tree:-



∴ Select the minimum cost i.e, $\hat{c}(4)$ and explore the nodes.

Step 6:-

For node 6 ($x_3=1$)

weight (with fraction)	weight (without fraction)
$w=(2,4,6,3)$	$w=(2,4,6)$

$c = -\sum p_i x_i$

$= -(10+10+12+\overset{2}{\cancel{18}} \times 3/\cancel{9})$

$\boxed{c=-38}$

$u = -(10+10+12)$

$\boxed{u=-32}$

For node 7 ($x_3=0$):-

→ If 3rd object is not included.

weight (with fraction)	weight (without fraction)
$w=(2,4,9)$	$w=(2,4,9)$

$c = -\sum p_i x_i$

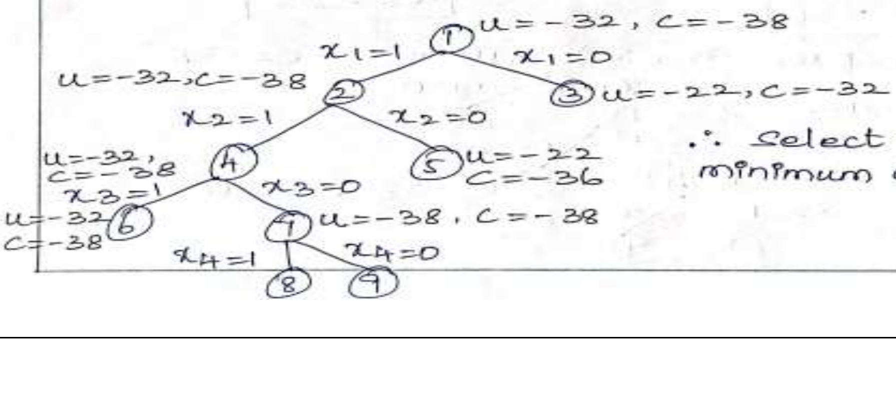
$= -(10+10+18)$

$\boxed{c=-38}$

$u = -\sum p_i x_i$

$= -(10+10+18)$

$\boxed{u=-38}$



∴ Select the minimum cost.

Step 7:-

For node 8 ($x_4 = 1$)

→ If 4^{th} object is included in the bag

$w = (2, 4, 6, \overset{3}{9})$

$C = -\sum p_i x_i$

$= -(10 + 10 + 12 + 18 \times 3/9)$

$\boxed{C = -38}$

$w = (2, 4, 6)$

$u = -(10 + 10 + 12)$

$\boxed{u = -32}$

For node 9 ($x_4 = 0$)

→ If 4^{th} object is not included in the bag.

weight (with fraction)

$w = (2, 4)$

$C = -\sum p_i x_i$

$= -(10 + 10)$

$\boxed{C = -20}$

weight (without fraction)

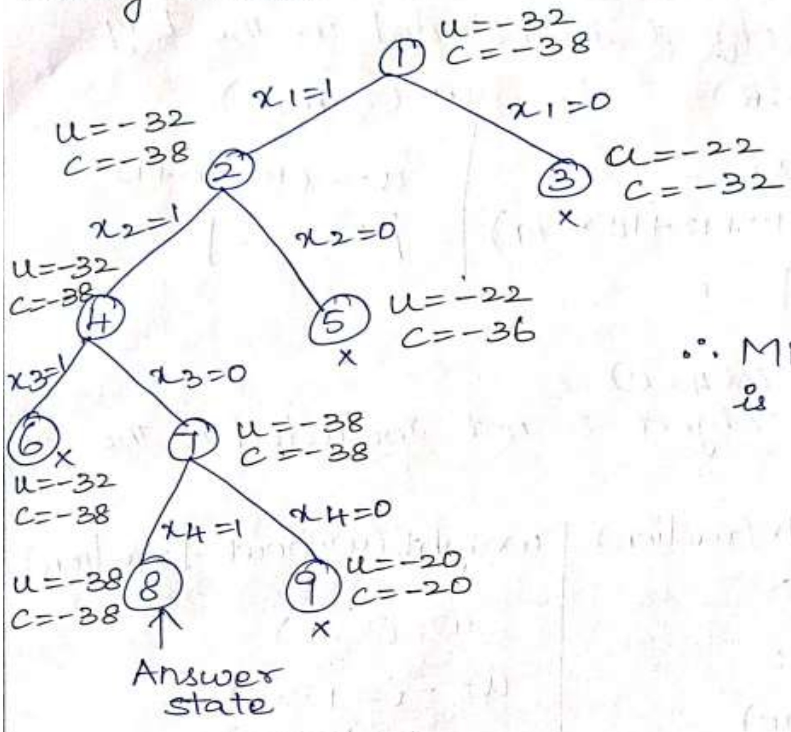
$w = (2, 4)$

$u = -(\sum p_i x_i)$

$= -(10 + 10)$

$\boxed{u = -20}$

Final state space tree for 0/1 Knapsack using LCBB.



∴ Minimum node is $\hat{C}(8)$

∴ The path is 1 - 2 - 4 - 7 - 8

∴ The solution for 0/1 Knapsack is

$x = \{1, 1, 0, 1\}$

∴ Maximum profit $= 10 + 10 + 0 + 18$

$= 38$

∴ weight $= 2 + 4 + 9 = 15$

8	Construct the LC branch and bound search. Consider knapsack instance n=4 with Capacity M=15 such that pi={10,10,12,18}, wi={2,4,6,9} apply FIFO branch and bound technique.	[L6][CO4]	[12M]

→ In FIFO branch and bound approach variable tuple size state space tree is drawn.

→ For each node N, cost function $\hat{c}(\cdot)$ and upper bound $u(\cdot)$ is computed similarly to the previous approach.

→ In LC Search, E-node is selected from two child of current node.

→ In FIFO branch and bound approach, both the children of sibling are inserted in list and most promising node is selected as new E-node.

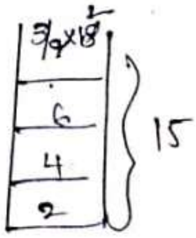
Example: $(P_1, P_2, P_3, P_4) = (10, 10, 12, 18)$

$(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$

<u>Solution</u>: Let us compute $u(1)$ and $\hat{c}(1)$

$u(1) = -(10+10+12) = -32$

$\hat{c}(1) = -(10+10+12+3/9 \times 18) = -38$

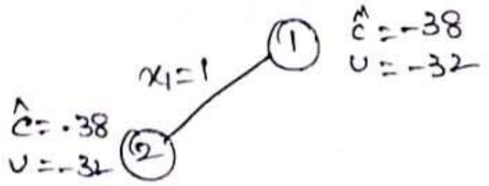


(1) $\hat{c} = -38$, $U = -32$.

Node 2: inclusion of item 1 at node 1

$u(2) = -(10+10+12) = -32$

$\hat{c}(2) = -(10+10+12+3/9 \times 18) = -38$



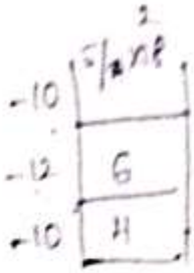
Node 3: Exclusion of item 1 at node 1

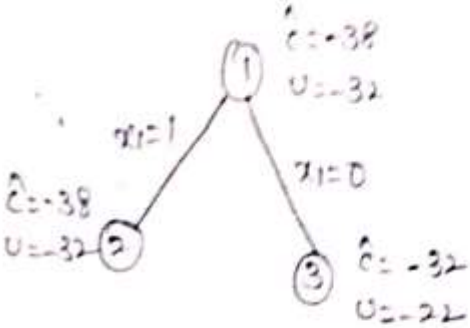
→ we are excluding item 1, including 2 and 3.

→ Item 4 cannot be accommodated in a knapsack along with 2 and 3.

$u(3) = -(10+12) = -22$

$\hat{c}(3) = -(10+12+\frac{5}{9}\times 18) = -32$





→ LC approach, node 2 would be selected as E-Node, as it has minimum(x)

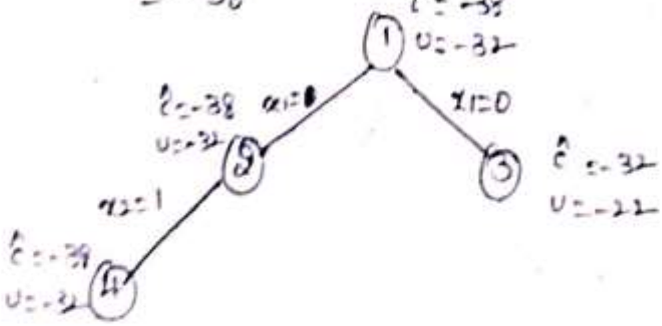
→ But in FIFO approach, all child of node 2 and 3 are expanded and the most promising child becomes E-nodes.

Node 4: Inclusion of item 2 at node 2

$u(4) = -(10+10+12) = -32$

$\hat{c}(4) = -(10+10+12+\frac{5}{9}\times 18)$

$= -38$



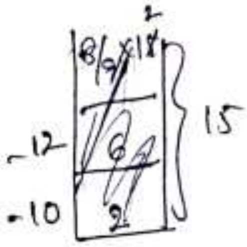
Node 5: Exclusion of item 2 at node 2

→ We are excluding item 1, including 2 and 3.

→ Item 4 cannot be accommodated in a knapsack along with 2 and 3.

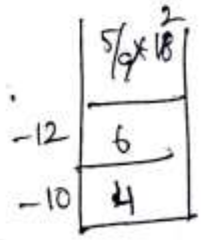
$U(5) = -(10+12) = -22$

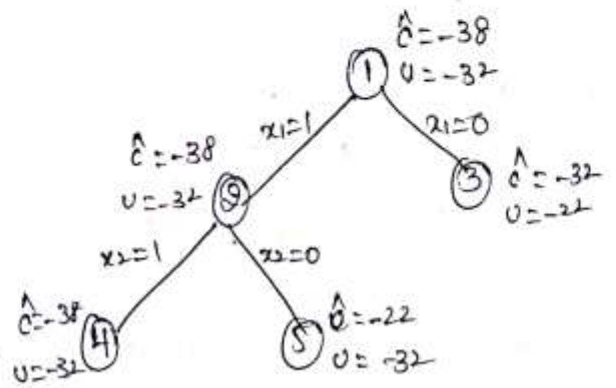
$\hat{c}(5) = -(10+\cancel{12+8/9\times18})$



$\hat{c}(5) = -(10+12+5/9\times18)$

$= -(10+12+10) = -32$

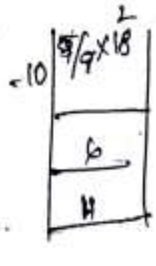


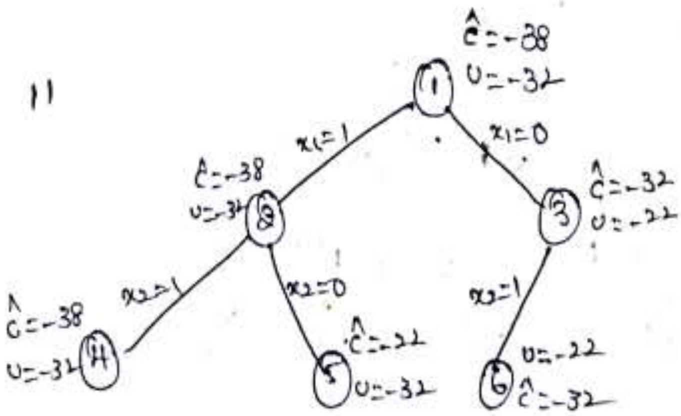


Node 6: Inclusion of item 2 at node 3

$U(6) = -(p_2+p_3) = -(10+12) = -22$

$\hat{c}(6) = -(10+12+10) = -32$





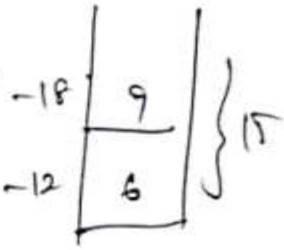
Node 7: Exclusion of item 2 at node 3.

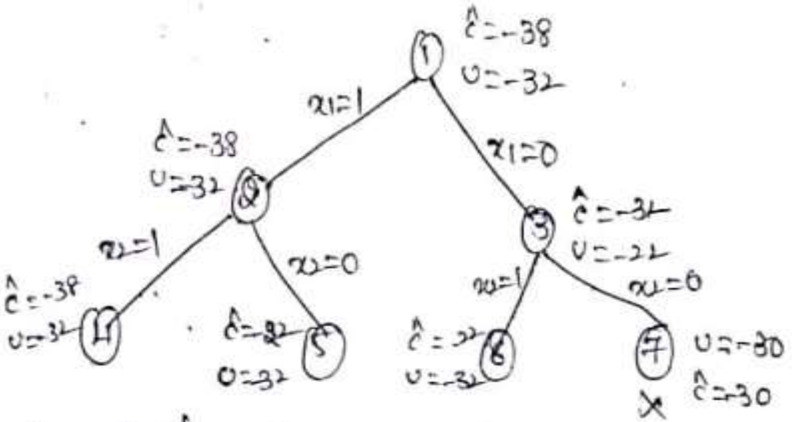
→ We are excluding item 1, including 2 and 3.

→ Item 4 cannot be accommodated in knapsack along with 2 and 3

$u(7) = -(p_3+p_4) = -(12+18) = -30.$

$\hat{c}(7) = -(12+18) = -30$





→ out of node 4, 5, 6 and 7, c(7) > upper so kill node 7

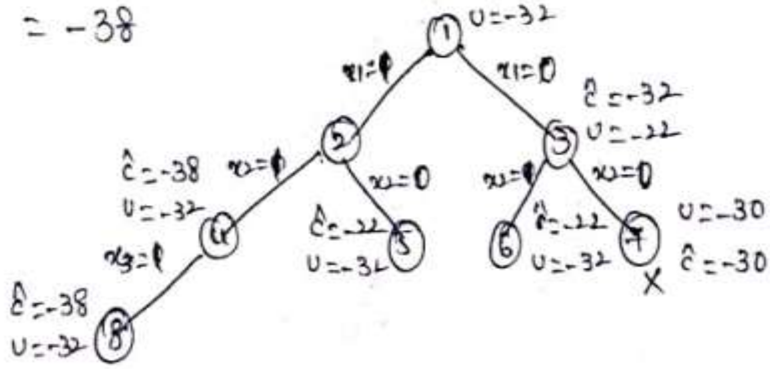
→ out of 4, 5 and 6, node 4 has minimum ĉ( ), so it becomes next E-node.

Node 8: inclusion of item 3 at node 4

$u(8) = -(10+10+12) = -32$

$\hat{c}(8) = -(10+10+12+3/9 \times 18)$

$= -38$



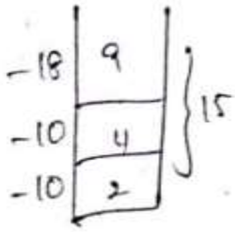
Node 9: Exclusion of item 3 at node 4

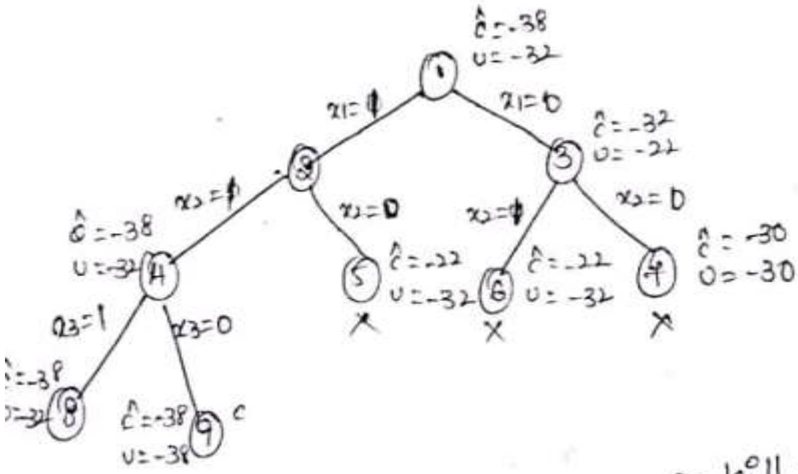
→ We are excluding item 3, including 1 and 2 and 4.

$u(9) = -(p_1+p_2+p_4) = -(10+10+18)$

$= -38$

$\hat{c}(9) = -(10+10+18) = -38$





→ $\hat{c}(5) >$ upper and $\hat{c}(6) >$ upper so kill them

→ If we continue in this way, final state space tree.

Node 10: inclusion of item 4 at node 8

$u(10) = -(10+10+12) = -32$

$\hat{c}(10) = -(10+10+12+\frac{3}{9}\times 18)$

$= -38$

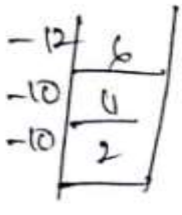
Node 11: Exclusion of item 4 at node 8

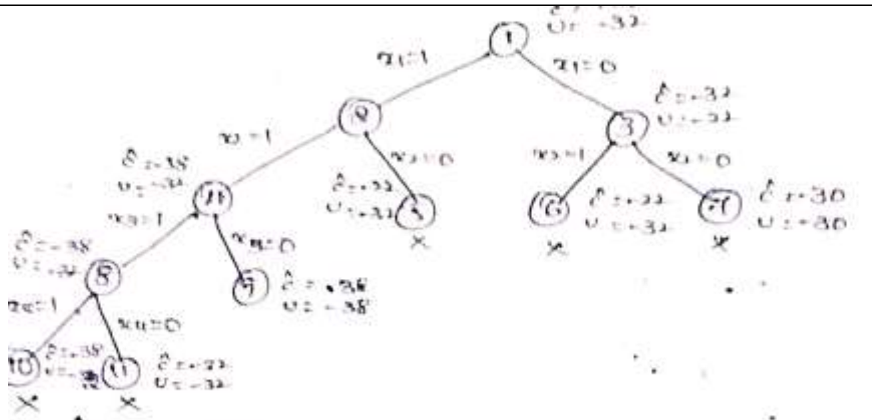
→ We are excluding item 4, including 1 and 2 and 3.

$u(11) = -(p_1+p_2+p_3) = -(10+10+12)$

$= -32$

$\hat{c}(11) = -(10+10+12) = -32$





→ $\hat{c}(10)$ > upper and $\hat{c}(11)$ > upper , so kill them. we will continue with node 9.

Node 12 : Inclusion of item 4 at node 9

$u(12) = -(10+10+18) = -38$

$\hat{c}(12) = -(10+10+18) = -38$

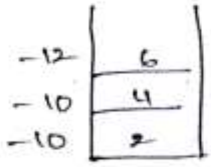
Node 13 : Exclusion of item 4 at node 9

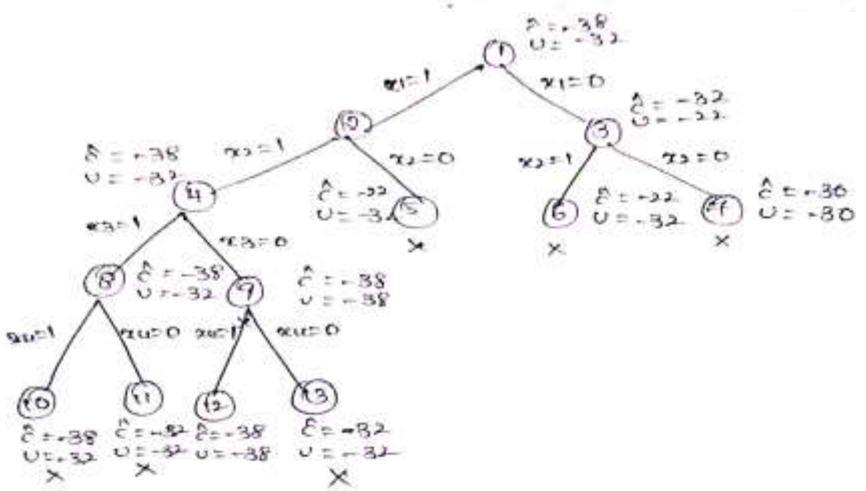
→ we are excluding item 4, including 1, 2, and 3

→ Item 4 cannot be accommodated in knapsack along with 1, 2 and 3

$u(13) = -(P_1+P_2+P_3) = -(10+10+12)$
$= -32$

$\hat{c}(13) = -(10+10+12) = -32$



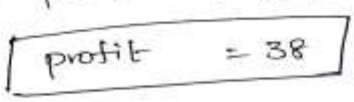


→ $\hat{c}(13)$ > upper then kill them

→ Node 12 has minimum cost function value, so it will be the answer node.

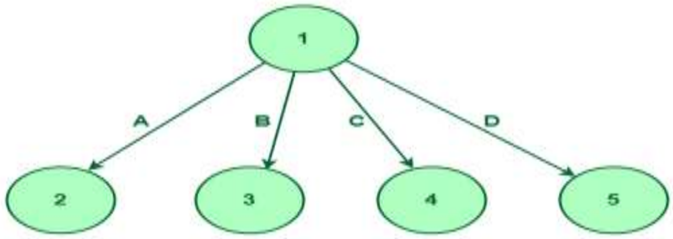
∴ Solution vector $x_i = \{x_1, x_2, x_3, x_4\}$
$= \{1, 1, 0, 1\}$

∴ profit = 10+10+0+18

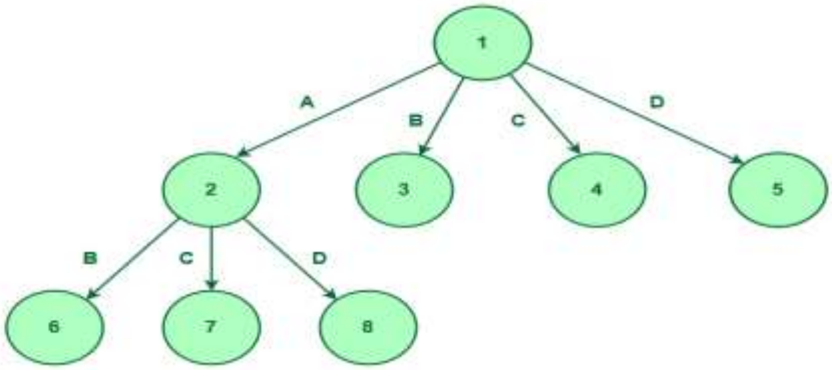


9	a)	**Explain the principles of FIFO branch and bound.**	[L2][CO4]	[6M]

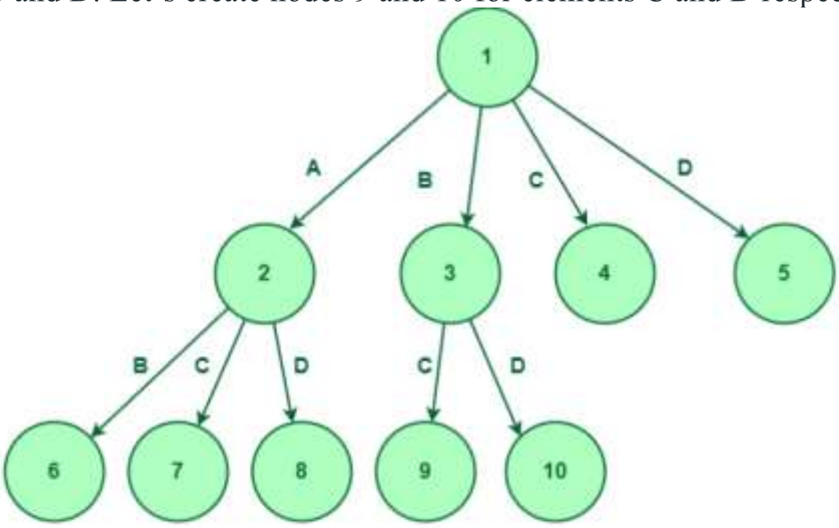
- First-In-First-Out is an approach to the branch and bound problem that uses the queue approach to create a state-space tree. In this case, the breadth-first search is performed, that is, the elements at a certain level are all searched, and then the elements at the next level are searched, starting with the first child of the first node at the previous level.
- For a given set {A, B, C, D}, the state space tree will be constructed as follows :



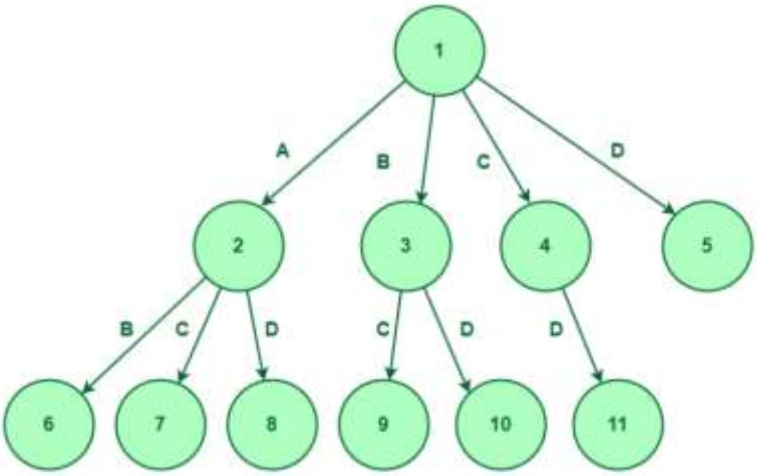
- The above diagram shows that we first consider element A, then element B, then element C and finally we'll consider the last element which is D. We are performing BFS while exploring the nodes.
- So, once the first level is completed. We'll consider the first element, then we can consider either B, C, or D. If we follow the route then it says that we are doing elements A and D so we will not consider elements B and C. If we select the elements A and D only, then it says that we are selecting elements A and D and we are not considering elements B and C.



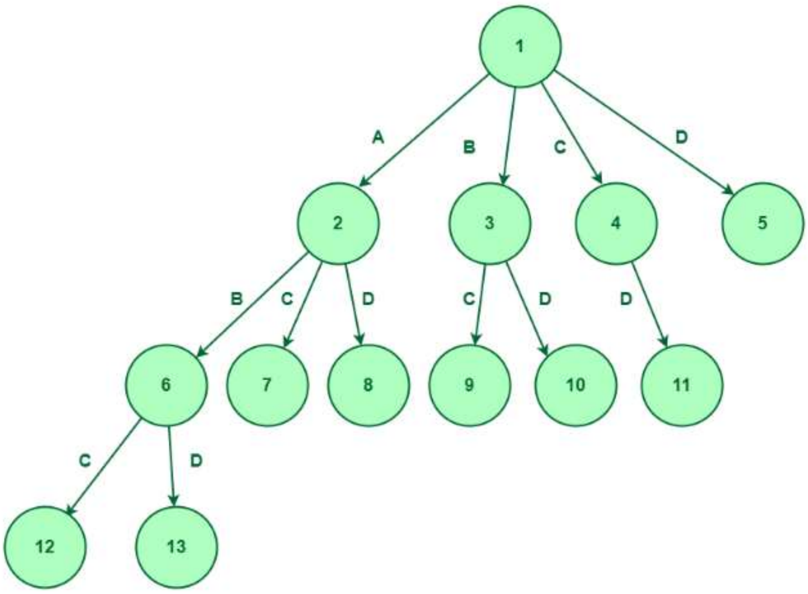
- Now, we will expand node 3, as we have considered element B and not considered element A, so, we have two options to explore that is elements C and D. Let's create nodes 9 and 10 for elements C and D respectively.



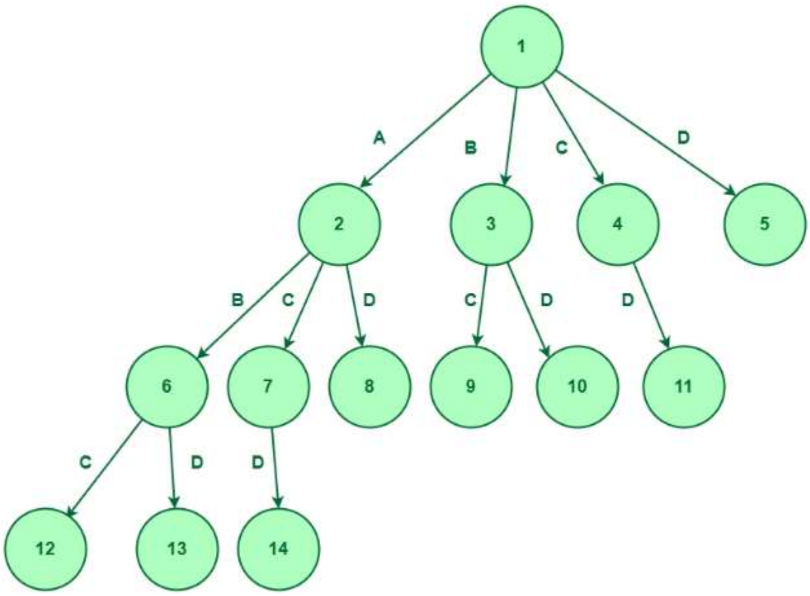
- Now, we will expand node 4 as we have only considered elements C and not considered elements A and B, so, we have only one option to explore which is element D. Let's create node 11 for D.



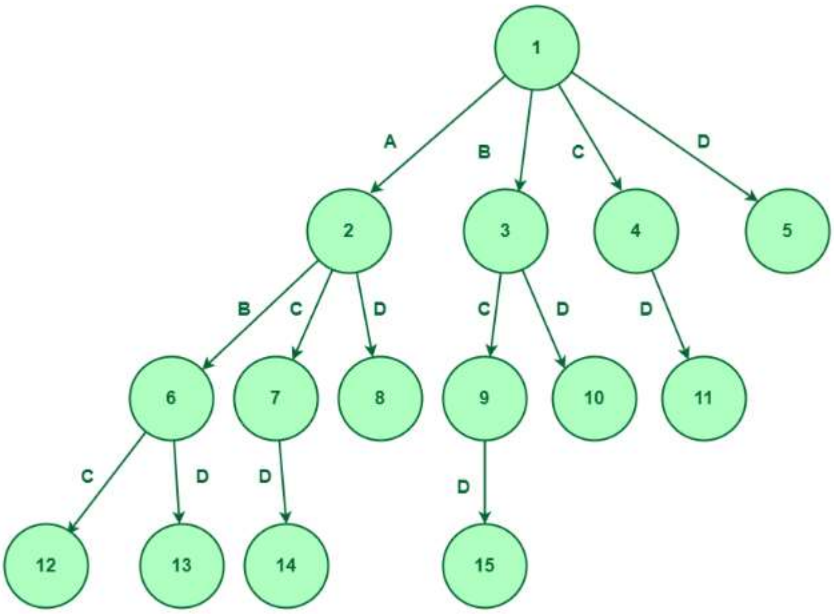
- Till node 5, we have only considered elements D, and not selected elements A, B, and C. So, We have no more elements to explore, Therefore on node 5, there won't be any expansion.
- Now, we will expand node 6 as we have considered elements A and B, so, we have only two option to explore that is element C and D. Let's create node 12 and 13 for C and D respectively.



- Now, we will expand node 7 as we have considered elements A and C and not consider element B, so, we have only one option to explore which is element D. Let's create node 14 for D.

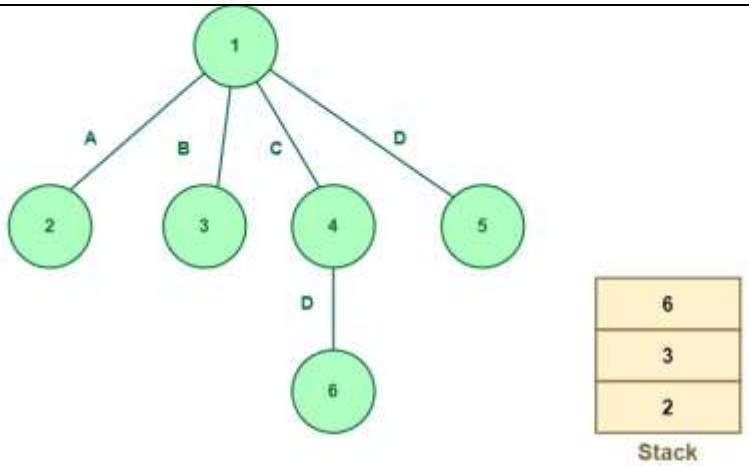


- Till node 8, we have considered elements A and D, and not selected elements B and C, So, We have no more elements to explore, Therefore on node 8, there won't be any expansion.
- Now, we will expand node 9 as we have considered elements B and C and not considered element A, so, we have only one option to explore which is element D. Let's create node 15 for D.

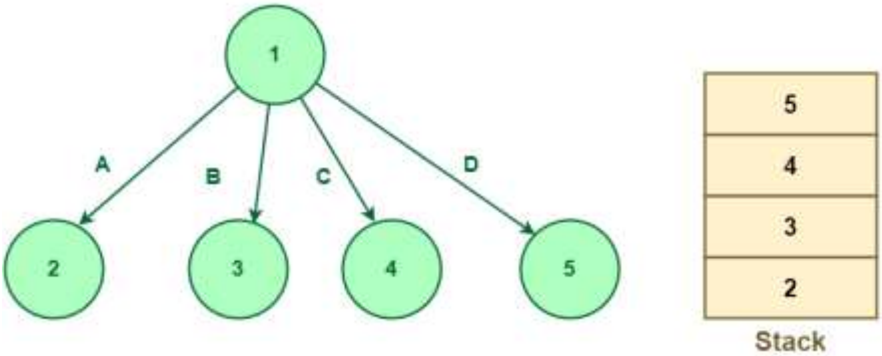


b) Explain the principles of LIFO branch and bound. **[L2][CO4]** **[6M]**

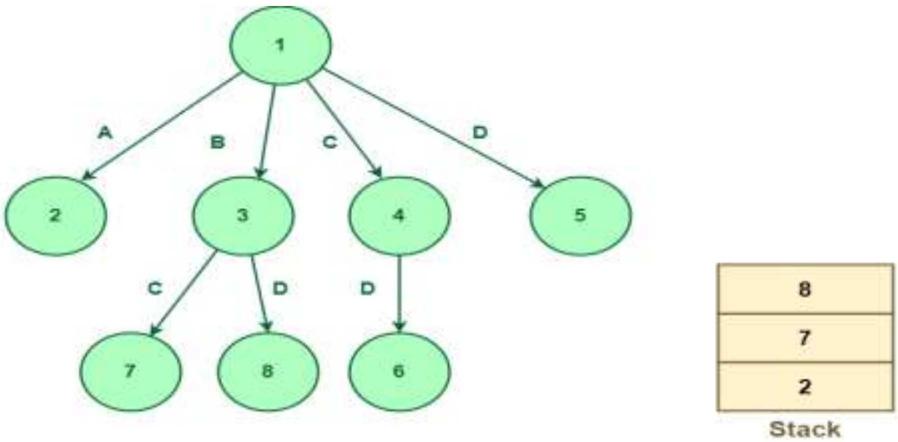
- The Last-In-First-Out approach for this problem uses stack in creating the state space tree. When nodes are added to a state space tree, they are added to a stack. After all nodes of a level have been added, we pop the topmost element from the stack and explore it.
- For a given set {A, B, C, D}, the state space tree will be constructed as follows :



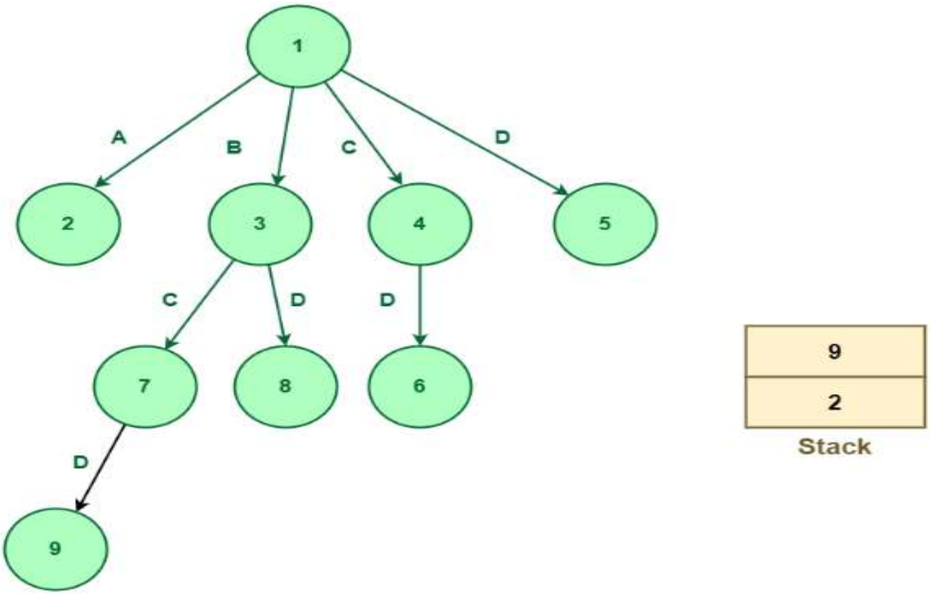
- Now the expansion would be based on the node that appears on the top of the stack. Since node 5 appears on the top of the stack, so we will expand node 5. We will pop out node 5 from the stack. Since node 5 is in the last element, i.e., D so there is no further scope for expansion.
- The next node that appears on the top of the stack is node 4. Pop-out node 4 and expand. On expansion, element D will be considered and node 6 will be added to the stack shown below:



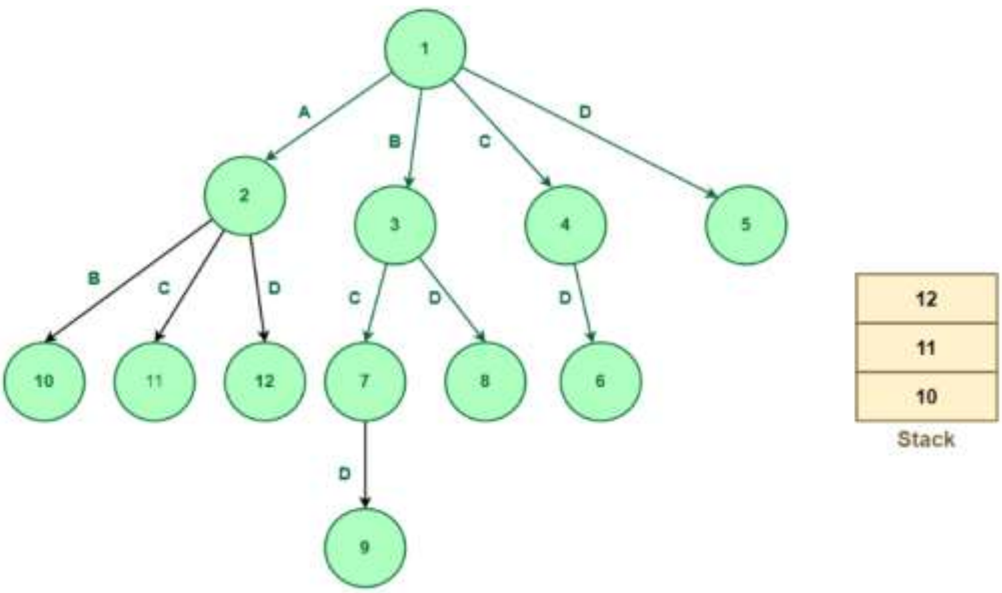
- The next node is 6 which is to be expanded. Pop-out node 6 and expand. Since node 6 is in the last element, i.e., D so there is no further scope for expansion.
- The next node to be expanded is node 3. Since node 3 works on element B so node 3 will be expanded to two nodes, i.e., 7 and 8 working on elements C and D respectively. Nodes 7 and 8 will be pushed into the stack.
- The next node that appears on the top of the stack is node 8. Pop-out node 8 and expand. Since node 8 works on element D so there is no further scope for the expansion.



- The next node that appears on the top of the stack is node 7. Pop-out node 7 and expand. Since node 7 works on element C so node 7 will be further expanded to node 9 which works on element D and node 9 will be pushed into the stack.
- The next node is 6 which is to be expanded. Pop-out node 6 and expand. Since node 6 is in the last element, i.e., D so there is no further scope for expansion.



- The next node that appears on the top of the stack is node 9. Since node 9 works on element D, there is no further scope for expansion.
- The next node that appears on the top of the stack is node 2. Since node 2 works on the element A so it means that node 2 can be further expanded. It can be expanded up to three nodes named 10, 11, 12 working on elements B, C, and D respectively. There new nodes will be pushed into the stack shown as below:



- In the above method, we explored all the nodes using the stack that follows the LIFO principle.

10	**Implement any one branch and bound application with an example.** **Refer 6 or 7 or 8**	**[L3][CO4]**	**[12M]**

UNIT-V
NP-HARD AND NP-COMPLETE PROBLEM

1	**Explain the following**	**[L2][CO5]**	**[12M]**

i. P class:

- P problems are a set of problems that can be solved in polynomial time by deterministic algorithms.
- P is also known as PTIME or DTIME complexity class.
- P problems are a set of all decision problems which can be solved in polynomial time using the deterministic Turing machine.
- They are simple to solve, easy to verify and take computationally acceptable time for solving any instance of the problem. Such problems are also known as *"tractable"*.
- In the worst case, searching an element from the list of size n takes n comparisons. The number of comparisons increases linearly with respect to the input size. So linear search is P problem.
- In practice, most of the problems are P problems. Searching an element in the array (O(n)), inserting an element at the end of a linked list (O(n)), sorting data using selection sort($O(n^2)$), finding the height of the tree ($O(\log_2 n)$), sort data using merge sort($O(n\log_2 n)$), matrix multiplication $O(n^3)$ are few of the examples of P problems.
- An algorithm with $O(2^n)$ complexity takes double the time if it is tested on a problem of size (n + 1). Such problems do not belong to class P.
- It excludes all the problems which cannot be solved in polynomial time. The knapsack problem using the brute force approach cannot be solved in polynomial time. Hence, it is not a P problem.
- There exist many important problems whose solution is not found in polynomial time so far, nor it has been proved that such a solution does not exist. TSP, Graph colouring, partition problem, knapsack etc. are examples of such classes.

Examples of P Problems:

1. Insertion sort
2. Merge sort
3. Linear search
4. Matrix multiplication
5. Finding minimum and maximum elements from the array

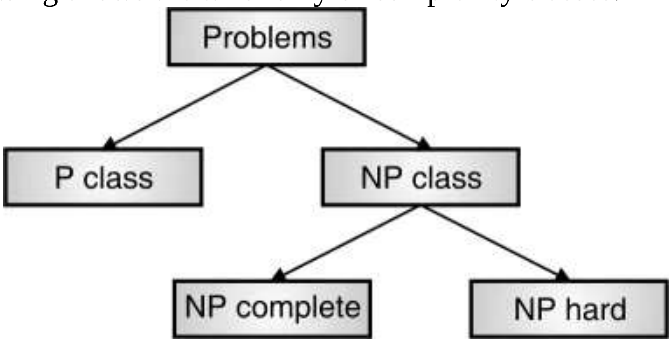
ii. NP class:

- NP is a set of problems which can be solved in nondeterministic polynomial time. NP does not mean non-polynomial, it stands for Non-Deterministic Polynomial-time.
- The non-deterministic algorithm operates in two stages.
- **Nondeterministic (guessing) stage:** For input instance I, some solution string S is generated, which can be thought of as a candidate solution.
- **Deterministic (verification) stage:** I and S are given as input to the deterministic algorithm, which returns "Yes" if S is a solution for input instance I.
- The solution to NP problems cannot be obtained in polynomial time, but given the solution, it can be verified in polynomial time.
- NP includes all problems of P, i.e. $P \subseteq NP$.
- Knapsack problem ($O(2^n)$), Travelling salesman problem ($O(n!)$), Tower of Hanoi ($O(2^n - 1)$), Hamiltonian cycle ($O(n!)$) are examples

of NP problems.

- NP Problems are further classified into NP-complete and NP-hard categories.

The following shows the taxonomy of complexity classes.



- The NP-hard problems are the hardest problem. NP-complete problems are NP-hard, but the converse is not true.
- If NP-hard problems can be solved in polynomial time, then so is NP-complete.

Examples of NP problems

1. Knapsack problem ($O(2^n)$)
2. Travelling salesman problem ($O(n!)$)
3. Tower of Hanoi ($O(2^n – 1)$)
4. Hamiltonian cycle ($O(n!)$)

iii. NP complete:

- Polynomial time reduction implies that one problem is at least as hard as another problem, within the polynomial time factor. If A $\leq_p$ B, implies A is not harder than B by some polynomial factor.
- Decision problem A is called NP-complete if it has the following two properties :
  - It belongs to class NP.
  - Every other problem B in NP can be transformed to A in polynomial time, i.e. For every B ∈ NP, B $\leq_p$ A.
- These two facts prove that NP-complete problems are the harder problems in class NP. They are often referred to as NPC.
- If any NP-complete problem belongs to class P, then P = NP. However, a solution to any NP-complete problem can be verified in polynomial time, but cannot be obtained in polynomial time.

Theorem

- Let A bea NP-complete problem. For some decision problem B ∈ NP, if B $\leq_p$ A then B is also an NP-complete problem.
- NP-complete problems are often solved using randomization algorithms, heuristic approaches or approximation algorithms.

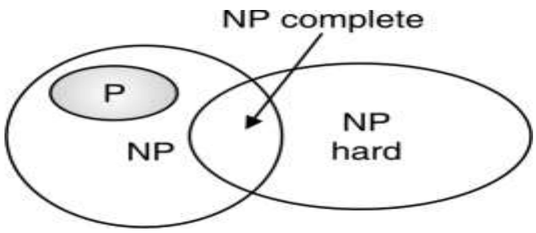
Some of the well-known NP-complete problems are listed here :

1. Boolean satisfiability problem.
2. Knapsack problem.
3. Hamiltonian path problem.
4. Travelling salesman problem.
5. Subset sum problem.
6. Vertex covers the problem.
7. Graph colouring problem.

8. Clique problem.

iv.NP Hard:

- Formally, a decision problem p is called NP-hard, if every problem in NP can be reduced to p in polynomial time.
- NP-hard is a superset of all problems. NPC is in NP-hard, but the converse may not be true.
- NP-hard problems are at least as hard as the hardest problems in NP.
- If we can solve any NP-hard problem in polynomial time, we would be able to solve all the problems in NP in polynomial time.
- NP-hard problems do not have to be in NP. Even they may not be a decision problem.
- The subset subproblem, the travelling salesman problem is NPC and also belongs to NP-hard. There are certain problems which belong to NP-hard but they are not NP-complete.
- A well-known example of the NP-hard problem is the Halting problem.
- The halting problem is stated as, "Given an algorithm and set of inputs, will it run forever ?" The answer to this question is Yes or No, so this is a decision problem.
- There does not exist any known algorithm which can decide the answer for any given input in polynomial time. So halting problem is an NP-hard problem.
- Different mathematicians have given different relationships considering the possibilities of P = NP and P ≠ NP.



- Non-deterministic problem Subset sum problem and travelling salesman problem are NPC and also belong to NP-hard. There are certain problems which belong to NP-hard but they are not NP-complete. A well-known example of an NP-hard problem is the Halting problem.
- The halting problem is stated as, "Given the algorithm and set of inputs, will it run forever?" The answer to this question is Yes or No, so this is a decision problem.
- There does not exist any known algorithm which can decide the answer for any given input in polynomial time. So halting problem is an NP-hard problem.
- Also, the Boolean satisfiability problem, which is in NPC, can be reduced to an instance of the halting problem in polynomial time by transforming it to the description of a Turing machine that tries all truth value assignments. The Turing machine halts when it finds such an assignment, otherwise, it goes into an infinite loop.

v. Non-deterministic problem:

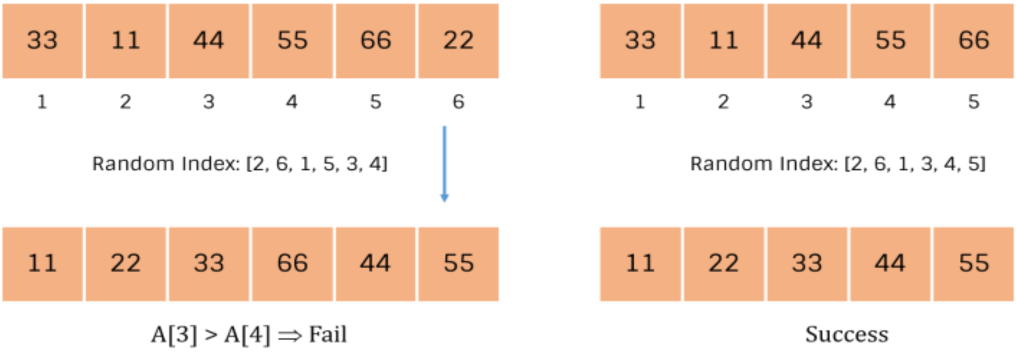
- Algorithm with the property that the result of every operation is uniquely defined is termed as deterministic algorithms.
- Such algorithms agree with the way programs are executed on a computer.
- Algorithms which contain operations whose outcomes are not uniquely defined but are not uniquely defined but are limited to

	specified set of possibilities. Such algorithms are called non-deterministic algorithms.
	• The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition to be defined later.
	• To specify non-deterministic algorithms, there are 3 new functions:
	➢ **Choice(s):** orbitary chooses one of the elements of set S.
	➢ **Failure():** signals an unsuccessful completion.
	➢ **Succuss():** Signals as successful completion.
	Example for non-deterministic algorithm:
	Algorithm Search(x)
	{
	// problem is to search an element x
	//output J, such that A[J]=x; or J=0 if x is not in A
	J:=choice(1,n);
	If(A[J]:=x) then
	{
	Write (J);
	Success();
	}
	Else
	{
	Write(s);
	Failure();
	}
	}		

2	**Construct the non-deterministic algorithms with suitable example.**
	Non-deterministic problem:
	• Algorithm with the property that the result of every operation is uniquely defined is termed as deterministic algorithms.
	• Such algorithms agree with the way programs are executed on a computer.
	• Algorithms which contain operations whose outcomes are not uniquely defined but are not uniquely defined but are limited to specified set of possibilities. Such algorithms are called non-deterministic algorithms.
	• The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition to be defined later.
	• To specify non-deterministic algorithms, there are 3 new functions:
	➢ **Choice(s):** orbitary chooses one of the elements of set S.
	➢ **Failure():** signals an unsuccessful completion.
	➢ **Succuss():** Signals as successful completion.
	Example for non-deterministic algorithm:
	Algorithm Search(x)
	{
	// problem is to search an element x
	//output J, such that A[J]=x; or J=0 if x is not in A
	J:=choice(1,n);
	If(A[J]:=x) then
	{
		[L3][CO5]	[12M]

	Write (J);
	Success();
	}
	Else
	{
	Write(s);
	Failure();
	}
	}
	Non- deterministic knapsack algorithm:
	Algorithm DKP(p,w,n,m,r,x)
	{
	W:=0;
	P:=0;
	For i:=1 to n do
	{
	X[i]:=choice(0,1);
	W:=W+X[i]*W[i];
	P:=P+X[i]*P[i];
	}
	If(W>m) or (P<r)
	Failure();
	Else
	Success();
	}		

3	**Build the non-deterministic sorting algorithm and also analyze its complexity.** ▪ Non Deterministic Sorting Algorithm produces different outputs on every execution. They work in a probabilistic way. The output of the algorithm depends on the sequence of random numbers generated. ▪ Consider A and B are input and output arrays of size n, respectively. The non-deterministic sorting approach selects any random number j between 1 to n and inserts the first element of array A on location j in array B. ▪ The process is repeated a maximum of *n* times. If location B[j] is already occupied then the algorithm fails. Otherwise, the selection of the next position continues. In this way, all elements of input array A are placed in output array B. ▪ After putting all elements in B, the algorithm enters in verification stage. In verification, two adjacent elements are compared in output array B. If any pair of adjacent elements are out of order, it implies array B is not properly sorted and the algorithm fails. ▪ In the below example (left), the size of the array is 6. We generated a random number between 1 to 6, six times. Assume that the sequence of generated random numbers is <2, 6, 1, 5, 3, 4>. Elements from the input array are rearranged based on those index values. ▪ A[1] < A[2], A[2] < A[3], but A[3] > A[4], implies that the array is not sorted and the algorithm will return fail. ▪ On the right side, the generated random index sequence is <2, 6, 1, 3, 4, 5>. Elements from the input array are rearranged. For each element A[I] < A[I + 1], implies this array is sorted and the algorithm returns true. ▪ Thus, the success of the algorithm purely depends on generated index sequence	**[L6][CO5]**	**[12M]**



Algorithm NON_DET_SORT(A, B)

// Description : Sort array A non-deterministically and store in array B
// Input : Array A and B of size n, representing input and output array respectively.
// Output : Success / Failure

```
// Guessing stage
for i ← 1 to n do
  B[i] ← 0
end
for i ← 1 to n do
  j ← select(1…n)
  if B[j] ≠ 0 then
    fail()
  end
  B[j] ← A[i]
end
// Verification stage
for  i ← 1  to n – 1 do
  if B[i + 1] < B[i] then
    fail()
  end
end
B[1…n]

success()
```

- The output of the algorithm is acceptable when all n guesses are true. If we run this algorithm multiple times, each time we may get different outputs.
- The running time of this algorithm is a function of a correct guess. Hence, the non-deterministic algorithm runs in O(f(n)) time, where n is the size of the input.

Complexity of Non-Deterministic Sorting Algorithm:

Non-deterministic Sorting Algorithm No nested loops so, complexity = O(n) Sorting array A[1:n] of positive integers in ascending order Algorithm Nsort(A,n) //sort n positive integers.

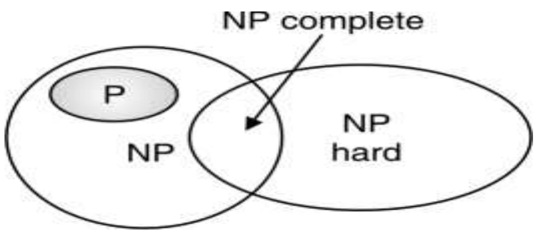
4	**Determine the classes NP-hard and NP-complete problem with example.**	**[L5][CO5]**	**[12M]**

Np- Hard:

- Formally, a decision problem p is called NP-hard, if every problem in NP can be reduced to p in polynomial time.
- NP-hard is a superset of all problems. NPC is in NP-hard, but the

converse may not be true.

- NP-hard problems are at least as hard as the hardest problems in NP.
- If we can solve any NP-hard problem in polynomial time, we would be able to solve all the problems in NP in polynomial time.
- NP-hard problems do not have to be in NP. Even they may not be a decision problem.
- The subset subproblem, the travelling salesman problem is NPC and also belongs to NP-hard. There are certain problems which belong to NP-hard but they are not NP-complete.
- A well-known example of the NP-hard problem is the Halting problem.
- The halting problem is stated as, "Given an algorithm and set of inputs, will it run forever ?" The answer to this question is Yes or No, so this is a decision problem.
- There does not exist any known algorithm which can decide the answer for any given input in polynomial time. So halting problem is an NP-hard problem.
- Different mathematicians have given different relationships considering the possibilities of P = NP and P ≠ NP.



- Non-deterministic problem Subset sum problem and travelling salesman problem are NPC and also belong to NP-hard. There are certain problems which belong to NP-hard but they are not NP-complete. A well-known example of an NP-hard problem is the Halting problem.
- The halting problem is stated as, "Given the algorithm and set of inputs, will it run forever?" The answer to this question is Yes or No, so this is a decision problem.
- There does not exist any known algorithm which can decide the answer for any given input in polynomial time. So halting problem is an NP-hard problem.
- Also, the Boolean satisfiability problem, which is in NPC, can be reduced to an instance of the halting problem in polynomial time by transforming it to the description of a Turing machine that tries all truth value assignments. The Turing machine halts when it finds such an assignment, otherwise, it goes into an infinite loop.

Np-Complete:

- Polynomial time reduction implies that one problem is at least as hard as another problem, within the polynomial time factor. If A $\leq_p$ B, implies A is not harder than B by some polynomial factor.
- Decision problem A is called NP-complete if it has the following two properties :
    - It belongs to class NP.
    - Every other problem B in NP can be transformed to A in polynomial time, i.e. For every B ∈ NP, B $\leq_p$ A.
- These two facts prove that NP-complete problems are the harder problems in class NP. They are often referred to as NPC.

	• If any NP-complete problem belongs to class P, then P = NP. However, a solution to any NP-complete problem can be verified in polynomial time, but cannot be obtained in polynomial time. **Theorem** • Let A be an NP-complete problem. For some decision problem B ∈ NP, if $B \leq_p A$ then B is also an NP-complete problem. • NP-complete problems are often solved using randomization algorithms, heuristic approaches or approximation algorithms. **Some of the well-known NP-complete problems are listed here :** 9. Boolean satisfiability problem. 10. Knapsack problem. 11. Hamiltonian path problem. 12. Travelling salesman problem. 13. Subset sum problem. 14. Vertex covers the problem. 15. Graph colouring problem. 16. Clique problem.		
5	**State and explain cook's theorem.** • Cook's Theorem implies that any NP problem is at most polynomially harder than SAT. This means that if we find a way of solving SAT in polynomial time, we will then be in a position to solve any NP problem in polynomial time. • This would have huge practical repercussions, since many frequently encountered problems which are so far believed to be intractable are NP. • This special property of SAT is called NP-completeness. • A decision problem is NP-complete if it has the property that any NP problem can be converted into it in polynomial time. • SAT was the first NP-complete problem to be recognized as such (the theory of NP-completeness having come into existence with the proof of Cook's Theorem), but it is by no means the only one. • There are now literally thousands of problems, cropping up in many different areas of computing, which have been proved to be NP-complete. • In order to prove that an NP problem is NP-complete, all that is needed is to show that SAT can be converted into it in polynomial time. • The reason for this is that the sequential composition of two polynomial-time algorithms is itself a polynomial-time algorithm, since the sum of two polynomials is itself a polynomial. • Suppose SAT can be converted to problem D in polynomial time. Now take any NP problem D0. • We know we can convert it into SAT in polynomial time, and we know we can convert SAT into D in polynomial time. • The result of these two conversions is a polynomial-time conversion of D0 into D. since D0 was an arbitrary NP problem, it follows that D isNP-complete	[L2][CO5]	[12M]
6	**Illustrate the satisifiability problem and write the algorithm.** • A propositional (or Boolean) variable that may be assigned the value true or false	[L2][CO5]	[12M]

	• If v is a propositional variable, then v the negation of v, has the value false.
	• A literal is a propositional variable or the negation of a propositional variable or a propositional constant (i.e., true or false) or an expression consisting of a Boolean operator and it operands, which is propositional formula.
	• Propositional formula may be represented in several forms, including functional notation(E.G. and (x,y)), operator notation(E.g.,(x^y)) or as an expression tree in which each internal node is a Boolean operator and each leaf is a propositional variable or one of the constants, true or false.
	• If truth values are assigned to the variables, the formula has a truth value that is obtained by applying the rules for the operators.
	• Certain regular form for propositional formulas, called conjunctive normal form turns out to be very useful.
	• A clause is a sequence of literals separated by the boolean OR operator(V).
	• A propositional formula is in Conjunctive Normal Form(CNF), if it consists of a sequence of clauses separated by the boolean AND operator (^).
	• An Example of a propositional formula in CNF is
	(pvqvs)^(p v r)^(r v s)^(p v s v q))
	Where p,q,r and s are propositional variables
	• A truth assignment for set propositional variables is an assignment of one of the values true or false to each propositional variables is assignment of one of the values true or false to each propositional variable in the set, in other words, a boolean valued function on the set.
	• A truth assignment is said to satisfy a formula if it makes the value of the entire formula true.
	• A CNF formula is said to be satisfiable and only if at least one literal in the clause is true.
	• Basically, CNF satisfiability is the satisfiability problem for CNF formulas.
	• If a propositional statement is satisfiable then it is possible to generate polynomial time non-deterministic algorithm.
	• This algorithm can be executed by selecting one of the two possible assignmentsof truth values of (p_1,p_2,p_3,…..p_k) and verify whether the statement S(p_1,p_2,p_3,…p_k) is true for that assignment.
	Following algorithm illustrate the aforementioned concepts:
	Algorithm Eval(E,K)
	{
	For i← to K do
	Pi← choice(false,true);
	If S(p1,p2,p3,..pk) then
	Success();
	Else
	Failure();
	}		

7	**Explain Reduction source problem With example.** Problem reduction is an algorithm design technique that takes a complex	**[L4][CO5]**	**[12M]**

problem and reduces it to a simpler one. The simpler problem is then solved and the solution of the simpler problem is then transformed to the solution of the original problem.
Problem reduction is a powerful technique that can be used to simplify complex problems and make them easier to solve. It can also be used to reduce the time and space complexity of algorithms.

Example:

Let’s understand the technique with the help of the following problem:

Calculate the LCM (Least Common Multiple) of two numbers X and Y.
Approach 1:
To solve the problem one can iterate through the multiples of the bigger element (say X) until that is also a multiple of the other element. This can be written as follows:

- Select the bigger element (say X here).
- Iterate through the multiples of X:
    - If this is also a multiple of Y, return this as the answer.
    - Otherwise, continue the traversal.

Algorithm:

```
Algorithm LCM(X, Y):
   if Y > X:
      swap X and Y
   end if
   for i = 1 to Y:
      if X*i is divisible by Y
         return X*i
      end if
   end for
```

Time Complexity: O(Y) as the loop can iterate for maximum Y times [because X*Y is always divisible by Y]
Auxiliary Space: O(1)
Approach 2 (Problem Reduction): The above method required a linear amount of time and if the value of Y is very big it may not be a feasible solution. This problem can be reduced to another problem which is to “**calculate GCD of X and Y**” and the solution of that can be transformed to get the answer to the given problem as shown below:

- Calculate the GCD of X and Y using Euclid’s algorithm.
- Now we know that GCD * LCM = X*Y. So the LCM can be calculated as **(X*Y/GCD)**.

Algorithm:

```
GCD (X, Y):
   if X = 0:
      return Y
   end if
   return GCD(Y%X, X)

Algorithm LCM(X, Y):
   G = GCD (X, Y)
   LCM = X * Y / G
```

Must Remember points about Problem Reduction:

- Reducing a problem to another one is only practical when the total time taken for transforming and solving the reduced problem is lower than solving the original problem.

- If problem A is reduced to problem B, then the lower bound of B can be higher than the lower bound of A, but it can never be lower than the lower bound of A.

8	**Explain the following:** **(a) decision problem** **(b) clique** **(c) non deterministic machine** **(d) satisfiability**	**[L4][CO5]**	**[12M]**

(a) decision problem:

- Any problem for which the answer is either yes or no is called decision problem.
- The algorithm for decision problem is called decision algorithm.
- Example: Max clique problem, sum of subsets problem.

Also, decision problem is one of the key concepts used to show a problem to be NP-complete.

Example:

The knapsack problem is a decision problem which is to determine the assigned values of A_i to be '0' or '1' such that $1 \leq i \leq n$, $\sum WiAi \leq z$ where $0 \leq pi \leq n$, $0 \leq Wi \leq n$, y is a number

therefore , the input size of knapsack decision problem, q is

$$q = \sum_{1 \le i \le n} ([\log_2 P_i] + [\log_2 W_i]) + 2n + [\log_2 z] + [\log_2 y] + 2$$

(b) clique:

- A clique is a subgraph of a graph such that all the vertices in this subgraph are connected with each other that is the subgraph is a complete graph.
- The Maximal Clique Problem is to find the maximum sized clique of a given graph G, that is a complete graph which is a subgraph of G and contains the maximum number of vertices.
- This is an optimization problem. Correspondingly, the Clique Decision Problem is to find if a clique of size k exists in the given graph or not.

(c) non deterministic machine:

In a Non-Deterministic Turing Machine, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic. The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration.

An input is accepted if there is at least one node of the tree which is an accept configuration, otherwise it is not accepted. If all branches of the computational tree halt on all inputs, the non-deterministic Turing Machine is called a **Decider** and if for some input, all branches are rejected, the input is also rejected.

A non-deterministic Turing machine can be formally defined as a 6-tuple (Q, X, ∑, δ, q_0, B, F) where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- ∑ is the input alphabet

- **δ** is a transition function;
  δ : Q × X → P(Q × X × {Left_shift, Right_shift}).
- **q_0** is the initial state
- **B** is the blank symbol
- **F** is the set of final states

(d) satisfiability:

- The satisfiability is a boolean formula that can be constructed using the following literals and operations. 1.
- A literal is either a variable or its negation of the variable. 2. The literals are connected with operators ∨, ∧, ⇒ ,⇔ 3.
- Parenthesis The satisfiability problem is to determine whether a Boolean formula is true for some assignment of truth values to the variables.
- In general, formulas are expressed in Conjunctive Normal Form (CNF). A Boolean formula is in conjunctive normal form iff it is represented by (xi∨ xj ∨ xk 1) ∧ (xi ∨ x 1 ∨ xk) A Boolean formula is in 3CNF if each clause has exactly 3 distinct literals.
- Example: The non-deterministic algorithm that terminates successfully iff a given formula E(x1,x2,x3) is satisfiable.

9 How to make reduction for 3-sat to clique problem? and Explain 3SAT - Determine whether a boolean formula in 3CNF can be satisfied **[L3][CO5]** **[12M]**

3SAT - Determine whether a boolean formula in 3CNF can be satisfied

Literal: a boolean variable, e.g., x or $\neg x$

Clause: a disjunction of literals, e.g., $x \vee y \vee z$

CNF: conjunctions of disjunctions, e.g., $(a \vee b) \wedge (\neg b \vee c) \wedge (b \vee c \vee d)$

3CNF: CNF where clauses are composed of exactly 3 literals, e.g. $(x \vee x \vee y) \wedge (\neg x \vee \neg y \vee \neg z)$

3SAT = {ϕ | ϕ is a satisfiable 3CNF}

K-Clique - Determine whether a graph has a k-clique

Clique: a subgraph in an undirected graph, where every pair of vertices is connected by an edge

K-clique: a clique containing k vertices

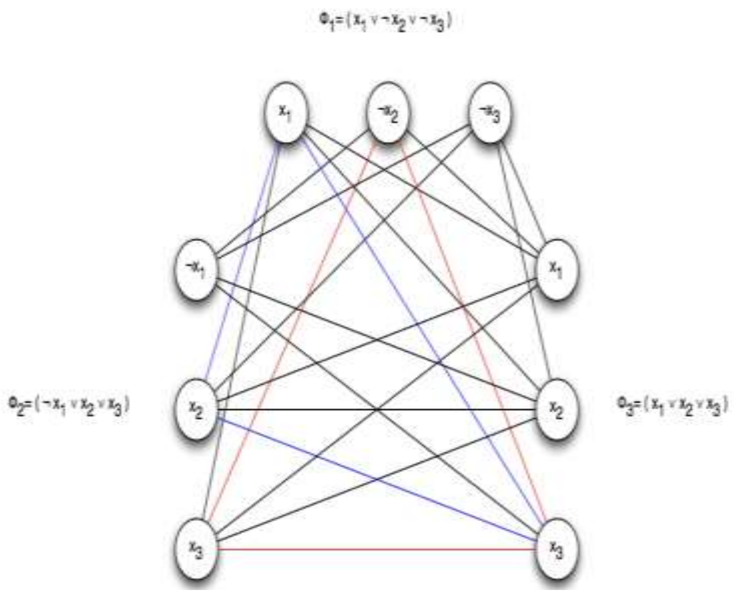
K-Clique = {$\langle G, k \rangle$ | G is an undirected graph with a k-clique}

Reduction of 3SAT to K-Clique

Proof:

- Let $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$
- Reduce this 3CNF into an undirected graph G by grouping the vertices of G into k groups of 3 vertices, each called a triple, $t_1, t_2, \ldots, t_k$
- Each triple corresponds to one of the clauses in ϕ
  Each vertex in a triple corresponds to a literal in the corresponding clause
- Connect all vertices by an edge except the following:
  - vertices in the same triple
  - vertices representing complementary literals, e.g., x and $\neg x$

Example:



10	a)	**Statement the following with examples**	[L4][CO5]	[6M]

a) Optimization problem :

Any problem that involves the identification of an optimal value (maximum or minimum) is called optimization problem. Example: Knapsack problem, travelling salesperson problem. In decision problem, the output statement is implicit and no explicit statements are permitted. The output from a decision problem is uniquely defined by the input parameters and algorithm specification. Many optimization problems can be reduced by decision problems with the property that a decision problem can be solved in polynomial time iff the corresponding optimization problem can be solved in polynomial time. If the decision problem cannot be solved in polynomial time then the optimization problem cannot be solved in polynomial time

b) Decision problem:

- Any problem for which the answer is either yes or no is called decision problem.
- The algorithm for decision problem is called decision algorithm.
- Example: Max clique problem, sum of subsets problem.

Also, decision problem is one of the key concepts used to show a problem to be NP-complete.

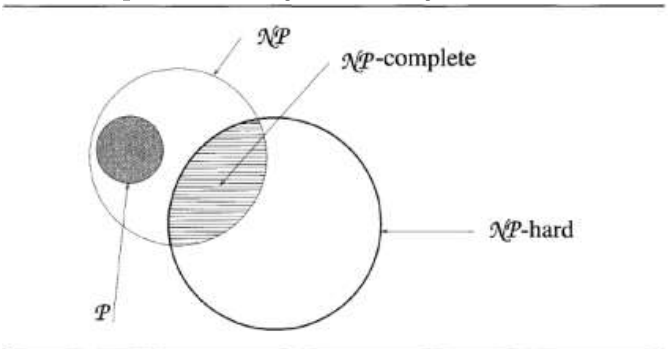
- Example:

The knapsack problem is a decision problem which is to determine the assigned values of A_i to be '0' or '1' such that $1 \leq i \leq n$, $\sum$ WiAi $\leq z$ where $0 \leq pi \leq n$, $0 \leq Wi \leq n$, y is a number therefore , the input size of knapsack decision problem, q is

$$q = \sum_{1 \le i \le n} ([\log_2 P_i] + [\log_2 W_i]) + 2n + [\log_2 z] + [\log_2 y$$

b) **Explain and shows the relationship between P,NP,NP Hard and NP Complete with neat diagram** **[L3][CO5]** **[6M]**

- P, NP, NP-hard, NP-Complete are the sets of all possible Let decision problems that are solvable in polynomial time by using deterministic algorithms, non-deterministic algorithms, NP-Hard and NP-complete respectively.
- Then the relationship between P, NP, NP-hard, NP-Complete can be expressed using Venn diagram as:



Commonly believed relationship among P, NP, NP-complete, and NP-hard problems

- Problem conversion A decision problem D1 can be converted into a decision problem D2 if there is an algorithm which takes as input an arbitrary instance I1 of D1 and delivers as output an instance I2 of D2such that I2 is a positive instance of D2 if and only if I1 is a positive instance of D1.
- If D1 can be converted into D2, and we have an algorithm which solves D2, then we thereby have an algorithm which solves D1.
- To solve an instance I of D1, we first use the conversion algorithm to generate an instance I0 of D2, and then use the algorithm for solving D2 to determine whether or not I0 is a positive instance of D2. If it is, then we know that I is a positive instance of D1, and if it is not, then we know that I is a negative instance of D1.
- Either way, we have solved D1 for that instance. Moreover, in this case, we can say that the computational complexity of D1 is at most the sum of the computational complexities of D2 and the conversion algorithm.
- If the conversion algorithm has polynomial complexity, we say that D1 is at most polynomials harder than D2. It means that the amount of computational work we have to do to solve D1, over and above whatever is required to solve D2, is polynomial in the size of the problem instance.

		• In such a case the conversion algorithm provides us with a feasible way of solving D1, given that we know how to solve D2. • Given a problem X, prove it is in NP-Complete. 1. Prove X is in NP. 2. Select problem Y that is known to be in NP-Complete. 3. • Define a polynomial time reduction from Y to X. 4. Prove that given an instance of Y, Y has a solution iffX has a solution.		