



**N.B.K.R. INSTITUTE OF SCIENCE AND TECHNOLOGY::VIDYANAGAR  
(AUTONOMOUS)**

\*\*\*\*\*

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**II-B.Tech I SEM(R-23)**

**ADVANCED DATA STRUCTURES & ALGORITHM  
ANALYSIS**

**(COMMON TO CSE, IT, AI&DS, AND ALLIED BRANCHES)**

## **UNIT-I**

Introduction to Algorithm Analysis, Space and Time Complexity analysis, Asymptotic Notations.

**AVL Trees** – Creation, Insertion, Deletion operations and Applications

**B-Trees** – Creation, Insertion, Deletion operations and Applications

## **UNIT-II**

**Heap Trees (Priority Queues)** – Min and Max Heaps, Operations and Applications

**Graphs** – Terminology, Representations, Basic Search and Traversals, Connected Components and Bi-connected Components, applications

**Divide and Conquer:** The General Method, Quick Sort, Merge Sort, Strassen's matrix multiplication, Convex Hull

## **UNIT-III**

**Greedy Method:** General Method, Job Sequencing with deadlines, Knapsack Problem, Minimum cost spanning trees, Single Source Shortest Paths

**Dynamic Programming:** General Method, All pairs shortest paths, Single Source Shortest Paths– General Weights (Bellman Ford Algorithm), Optimal Binary Search

Trees, 0/1 Knapsack, String Editing, Travelling Salesperson problem.

#### UNIT-IV

**Backtracking:** General Method, 8-Queens Problem, Sum of Subsets problem, Graph Coloring, **0/1 Knapsack Problem**

**Branch and Bound:** The General Method, 0/1 Knapsack Problem, Travelling Salesperson problem.

#### UNIT-V

**NP Hard and NP Complete Problems:** Basic Concepts, Cook's theorem.

**NP Hard Graph Problems:** Clique Decision Problem (CDP), Chromatic Number Decision Problem (CNDP), **Traveling Salesperson Decision Problem (TSP)**

**NP Hard Scheduling Problems:** Scheduling Identical Processors, Job Shop Scheduling

## UNIT-I

Introduction to Algorithm Analysis, Space and Time Complexity analysis, Asymptotic Notations.

**AVL Trees** – Creation, Insertion, Deletion operations and Applications

**B-Trees** – Creation, Insertion, Deletion operations and Applications

### 1.1 Introduction to Algorithm Analysis

An algorithm is an effective method for finding out the solution for a given problem. It is a sequence of instruction. That conveys the method to address a problem.

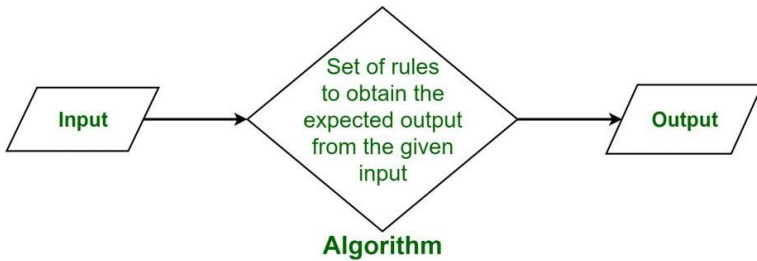
#### 1.1.1 Definition of Algorithm

Algorithm: an algorithm is a step by step procedure to solve a computational problem is called an algorithm.

**or**

A finite set of instruction that specifies a sequence of operation is to be carried out in order to solve a specific problem or class of problems is called an Algorithm.

Therefore Algorithm refers to a sequence of finite steps to solve a particular problem.



**1.1.2 Use of the Algorithms:** Algorithms play a crucial role in various fields and have many applications. Some of the key areas where algorithms are used include:

**a) Computer Science:** Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.

**b) Mathematics:** Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.

**c) Operations Research:** Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.

**d) Artificial Intelligence:** Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as

image recognition, natural language processing, and decision-making.

**e) Data Science:** Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.

These are just a few examples of the many applications of algorithms. The use of algorithms is continually expanding as new technologies and fields emerge, making it a vital component of modern society.

### **1.1.3 What is the need for algorithms?**

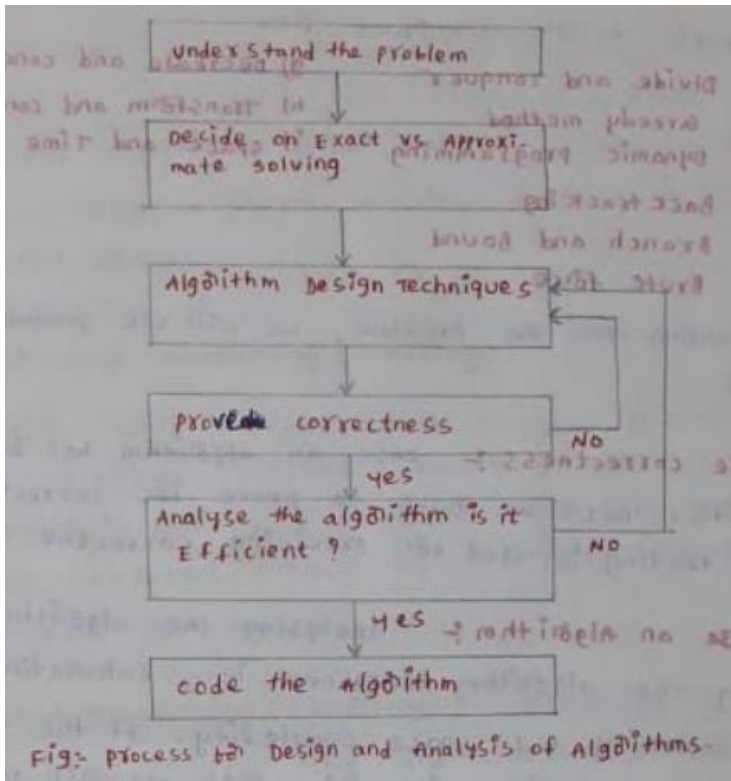
- Algorithms are necessary for solving complex problems efficiently and effectively.
- They help to automate processes and make them more reliable, faster, and easier to perform.
- Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.
- They are used in various fields such as mathematics, computer science, engineering, finance, and many others to optimize processes, analyze data, make predictions, and provide solutions to problems.

### 1.1.4 Properties of an Algorithm

According to D.E. Knuth a pioneer in the computer science discipline an algorithm must have the following properties

1. **INPUT:** The Algorithm should be given **zero** or more input.
2. **OUTPUT:** **At least one** quantity is produced. For each input the algorithm produced value from specific task.
3. **DEFINITENESS:** Each instruction is clear and **unambiguous**.
4. **FINITENESS:** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a **finite number** of steps.
5. **EFFECTIVENESS:** Every instruction must **very basic** so that it can be carried out, in principle, by a person using only pencil & paper.
6. **Correctness.**  
An algorithm must produce the correct output values for all legal input instances of the problem
7. **Generality**  
The algorithm should be applicable to all problems of a similar form
8. **Multiple view**  
Same algorithm may be represented in different ways
9. **Multiple Availability**  
Several algorithms for solving the same problem may exist - with different properties

## 1.1.5 Process for design and analysis of algorithms





1. understand the problem :- The first thing you need to do before designing an algorithm is to understand problem completely. This is a crucial phase, so we should be very careful. If we did any mistake in this step the entire algorithm becomes wrong.

2. Exact vs Approximate solving :- solve the problem exactly if possible, otherwise use approximation methods. Even though some problems are solvable by exact method, but they are not faster when compared to approximation method. so in that situation, we will use approximation method.

3. Algorithm design techniques :- In this we will use different design techniques like

- a) Divide and conquer
- b) Greedy method
- c) Dynamic programming
- d) Backtracking
- e) Branch and Bound
- f) Brute force
- g) Decrease and conquer
- h) Transform and conquer
- i) Space and Time trade-offs

depending upon the problem, we will use suitable design method.

4. **prove correctness** :- once an algorithm has been specified, next we have to prove its correctness. usually testing is used for providing correctness.

5. **Analyze an algorithm** :- Analyzing the algorithm means studying the algorithm's behaviour, i.e. calculating the time complexity and space complexity. If the time complexity of algorithm is more, then we will use one more designing technique such that time complexity should be minimum.

6. **coding an algorithm** :- After completion of all phases successfully, then we will code an algorithm. coding should not depend on any program language. we will use general notation (Pseudo code) and English language statement. ultimately algorithms are implemented as computer programs.

## 1.1.6 Pseudo code for expressing algorithms

The general procedure for writing the pseudo code is presented below.

1. single line comments are written using // as beginning of comment.
2. The beginning and end of block should be indicated by { and } respectively. The compound statements should be enclosed within { and } brackets.

Example :-

```
{  
  stmt1;  
  stmt2;  
  ...  
}
```

3. All statements are separated by ; (semicolon)
4. The identifier should begin by letter and not by digit. An identifier can be a combination of alphanumeric characters, space, time, underscore, and string.

Example :-

```
var x, y;  
x = 10;  
y = 20;
```

5. Assignment of values to the variables is done using the assignment operator := or ←
6. There are two boolean values TRUE and FALSE.  
Logical operators - AND, OR, NOT.  
Relational operators - <, <=, =, ≠, >, >=.

7. The conditional statement if-then or if-then else are written in the following form  
if (condition) then statement  
if (condition) then statement<sub>1</sub> else statement<sub>2</sub>.  
Here condition is a boolean expression and statement<sub>1</sub>, statement<sub>2</sub> are either simple or compound statements.

## 8. Case Statement

```
case  
{  
  : (condition1) : statement1  
  : (condition2) : statement2  
  :  
  : (conditionn) : statementn  
  :  
  : else : (statementn+1)  
}
```

If condition<sub>1</sub> is true, statement<sub>1</sub> is executed and the case statement is exited. If statement<sub>1</sub> is false, condition<sub>2</sub> is evaluated. If condition<sub>2</sub> is true, statement<sub>2</sub> is executed and so on. If none of the conditions are true, statement<sub>(n+1)</sub> is executed and loop is exited. The else clause is optional.

## 9. Looping statements

### ① For loop

The general form for writing for loop is:

```
for (variable = value1 to valuen step do  
{  
  stmt1;  
  stmt2;  
  ...  
  stmtn;  
}
```

Here value<sub>1</sub> is initialization condition and value<sub>n</sub> is a terminating condition. The step indicates the increments or decrements in value for executing the for loop.

## 2. while loop

while statement can be written as

```
while (condition) do
{
    stmt1;
    stmt2;
    :
    stmt n;
}
```

As long as condition is true, the statement gets executed otherwise the loop is exited. The value of condition is evaluated at the beginning of the loop.

10. The break or return can be used within any of the above instruction to exist from the loop.

11. Algorithm is a procedure consisting of heading and body. The heading has the form,

Algorithm Name (parameter list)

where Name is the name of the procedure and parameter list is a listing of the procedure parameters.

The body has one or more simple or compound statements enclosed within braces { and }.

Some Examples

Example 1:- write an algorithm to count the sum of n numbers.

Sol:-

```

algorithm sum(i, n)
{
    sum = 0;
    for i = 1 to n do
        i := i + 1
        sum = sum + i;
}

```

Example 2:- write an algorithm to check whether given number is even or odd.

Sol:-

```

algorithm evenOrOdd(valu)
{
    if (valu % 2 == 0) then
        write "number is even";
    else
        write "number is odd";
}

```

Example 3:- write an algorithm to sorting the elements.

Sol:-

```

algorithm sort(a, n)
{
    // sorting elements in ascending order
    for i = 1 to n do
        for j = i + 1 to n - 1 do
            if (a[i] > a[j]) then
                temp := a[i];
                a[i] := a[j];
                a[j] := temp;
        }
    }
    write "list is sorted";
}

```

Example 4 :- write an algorithm to find factorial of n number.

```
sol:- Algorithm fact(n) ;
      {
        if n := 1 then
          return 1;
        else
          return n * fact(n-1);
      }
```

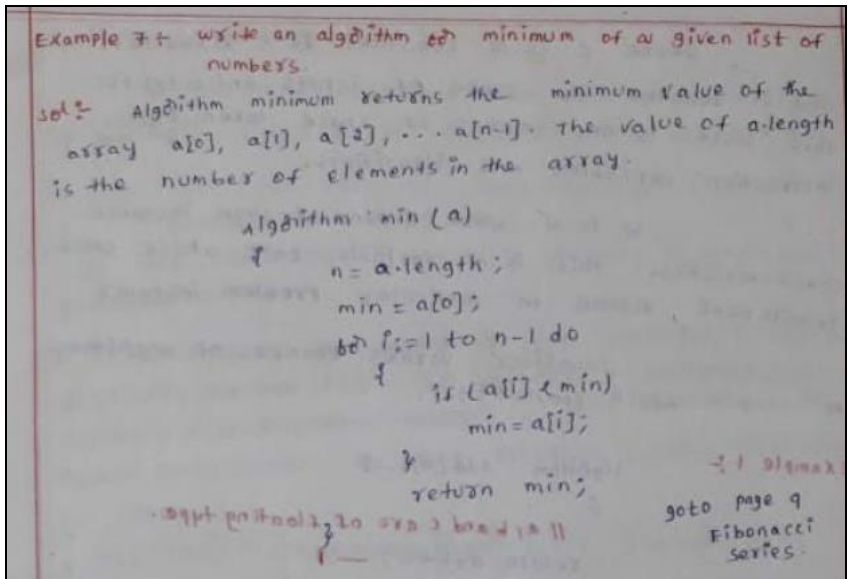
Example 5:- write an algorithm to perform multiplication of two matrices.

```
sol:- Algorithm mul(A, B, n)
      {
        for i := 1 to n do
          for j := 1 to n do
            c[i, j] := 0
            for k := 1 to n do
              c[i, j] := c[i, j] + A[i, k] * B[k, j];
      }
```

Example 6 :- write an algorithm for finding the maximum of a given list of numbers.

sol:- Algorithm maximum returns the maximum value of the array  $a[0], a[1], a[2], \dots, a[n-1]$ . The value of a.length is the number of elements in the array.

```
Algorithm max(a)
  {
    n = a.length;
    max = a[0];
    for i := 1 to n-1 do
      {
        if (a[i] > max)
          max = a[i];
      }
    return max;
  }
```

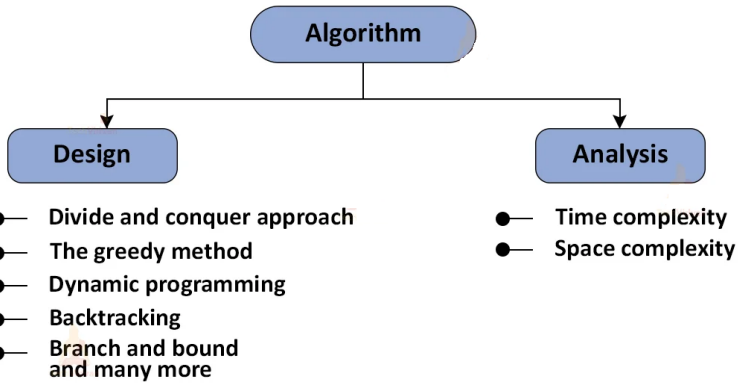


## 1.2 Overview of time and space complexity analysis

**Performance Analysis:** The efficiency of an algorithm can be decided by measuring the performance of algorithm. We can measure the performance of an algorithm by computing amount of time and storage requirement. We can analyse an algorithm by two ways.

1. By checking the correctness of an algorithm.
2. By measuring time and space complexity of an algorithm.





## Time Complexity

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input.

### Example

Statements	s/e	Frequency	Total steps
Algorithm sum(a,n)	0	-	0
{	0	-	0
s:=0;	1	1	1
for i:=1 to n do	1	n+1	n+1
s:=s+a[i]	1	n	n
return s;	1	1	1
}	0	-	0
			<b>2n+3</b>

## Space Complexity

The amount of memory used by a program to execute it is represented by its space complexity. The space requirement  $S(P)$  can be given as

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

### Example

Statements	s/e	Frequency	Total steps
Algorithm sum(a,n)	0	-	0
{	0	-	0
s:=0;	1	1	1
for i:=1 to n do	1	n+1	n+1
s:=s+a[i]	1	n	n
return s;	1	1	1
}	0	-	0
			<b>2n+3</b>

The space requirement for algorithm given in example is  $S(P) = 2n+3$ . Neglect the constants

$\therefore$  space complexity is  $O(n)$ .

Time Complexity	Space Complexity
Calculates time needed	Calculates memory space needed
Counts time for all statements.	Counts memory space for all variables even inputs and outputs
Depends mostly on input data size	Depends mostly on auxiliary variables size
More important for solution optimization	Less important with modern hardwares
Time complexity of merge sort is $\mathcal{O}(n \log n)$	Space complexity of merge sort is $\mathcal{O}(n)$

### 1.3 Asymptotic Notations

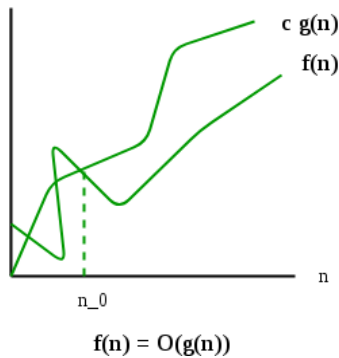
To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity. Using asymptotic notations we can give time complexity as “fastest possible”, “slowest possible” or “average time”. There are mainly three asymptotic notations:

1. Big-O Notation (O-notation)
2. Omega Notation ( $\Omega$ -Notation)
3. Theta Notation ( $\Theta$ -Notation)

#### 1. Big-O Notation (O-notation)

Big-oh notation denoted by ‘O’ is a method of representing the upper bound of algorithms running time. Using big-oh notation we can give longest amount of time taken by the algorithm to complete.

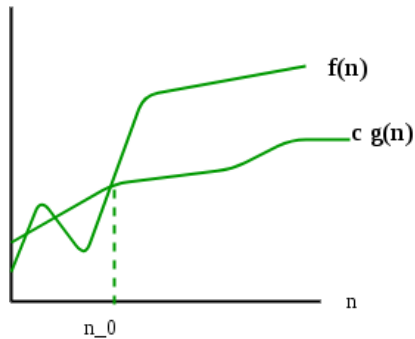
**Definition:** The function  $f(n)=O(g(n))$  iff there exist positive constants  $c$  and  $n_0$  such that  $f(n) \leq cg(n)$  for all  $n \geq n_0$



## 2. Omega Notation ( $\Omega$ -Notation)

Omega notation denoted by  $\Omega$  is a method of representing lower bound of algorithms running time. Using omega notation we can denote shortest amount of time taken by the algorithm to complete.

**Definition:** The function  $f(n)=\Omega (g(n))$  iff there exist positive constants  $c$  and  $n_0$  such that  $f(n) \geq c \cdot g(n)$  for all  $n, n \geq n_0$

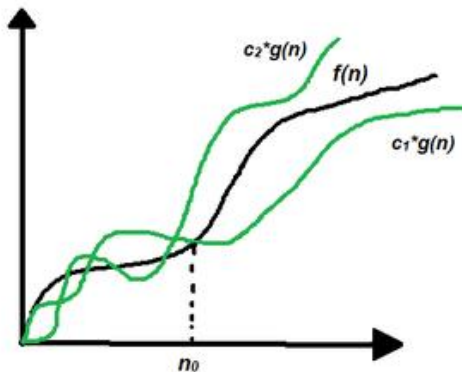


$$f(n) = \Omega(g(n))$$

### 3. Theta Notation ( $\Theta$ -Notation)

The Theta notation denoted as  $\Theta$  is a method of representing running time between upper bound and lower bound.

**Definition:** The function  $f(n) = \Theta(g(n))$  iff there exist positive constants  $c_1$ ,  $c_2$  and  $n_0$  such that  $c_1 g(n) \leq f(n) \leq c_2 g(n)$  for all  $n$ ,  $n \geq n_0$



## Linear Data structures - Time Complexities

Data Structure	Insert	Delete	Access	Search
Array	$O(N)$	$O(N)$	$O(1)$	$O(N)$
String	$O(N)$	$O(N)$	$O(1)$	$O(N)$
Queue	$O(1)$	$O(1)$	$O(N)$	$O(N)$
Stack	$O(1)$	$O(1)$	$O(N)$	$O(N)$
Linked List	$O(1)$ , if inserted on head $O(N)$ , elsewhere	$O(1)$ , if deleted on head $O(N)$ , elsewhere	$O(N)$	$O(N)$

## Searching Algorithms - Time Complexities

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$

## Sorting Algorithms - Time Complexities

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n + k)$
Bucket Sort	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n)$

### 1.3 AVL Trees –

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

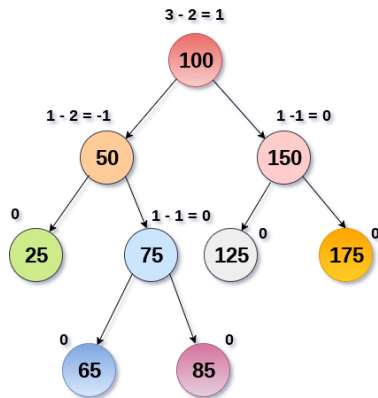
- AVL Trees are Self-Balanced Binary Search Trees.
- In AVL trees, the balancing factor of each node is either 0 or 1 or -1.
- Balance Factor of AVL Tree calculated as = Height of Left Sub-tree - Height of Right Sub-tree

#### Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

#### Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is **1**, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is **-1**, it means that the left sub-tree is one level lower than the right sub-tree.
- If balance factor of any node is **0**, it means that the left sub-tree and right sub-tree contain equal height.



AVL Tree

## Operations on an AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.



## AVL Rotations

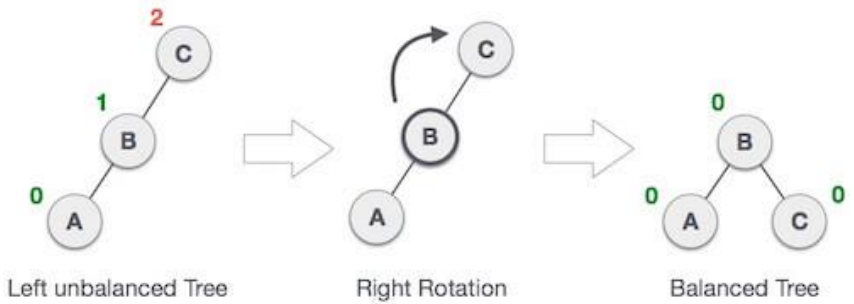
We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

The first two rotations **LL** and **RR** are **single rotations** and the next two rotations **LR** and **RL** are **double rotations**. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

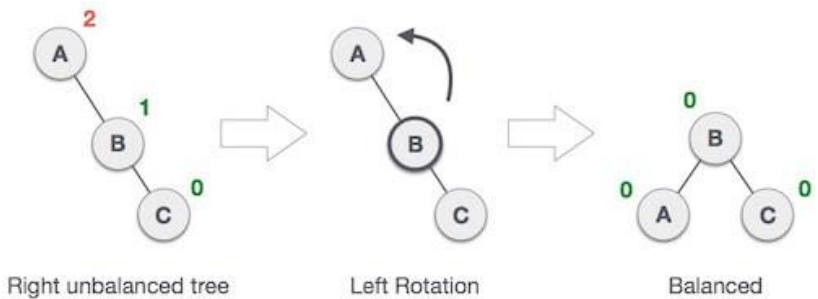
### 1. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



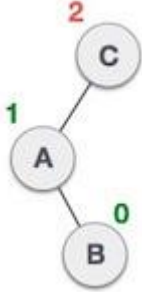
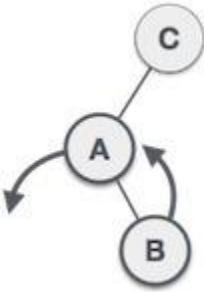
## 2. RR Rotation

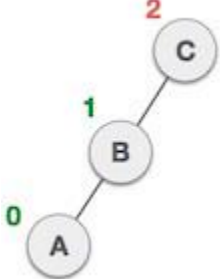
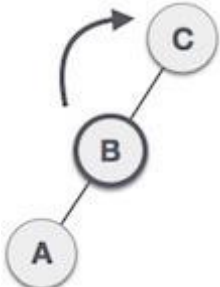
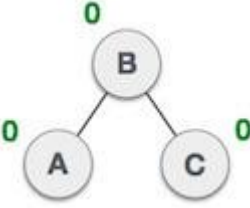
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



### 3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

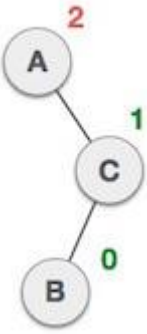
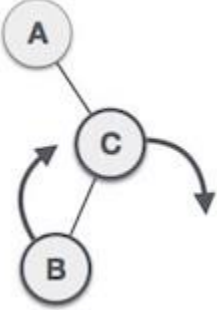
State	Action
	<p>A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C</p>
	<p>As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.</p>

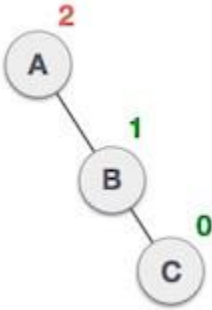
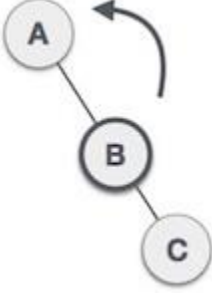
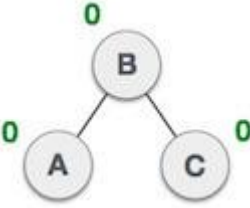
	<p>After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C</p>
	<p>Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.</p>

#### 4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. RL rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on

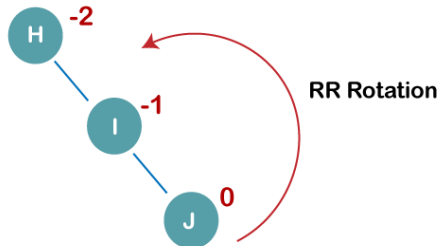
full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	<p>A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A</p>
	<p>As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.</p>

	<p>After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.</p>
	<p>Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.</p>

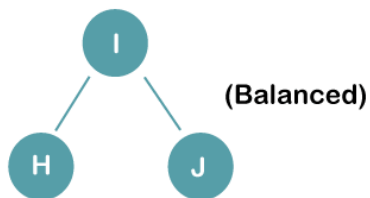
**Q: Construct an AVL tree having the following elements  
H, I, J, B, A, E**

**Sol: 1. Insert H, I, J**

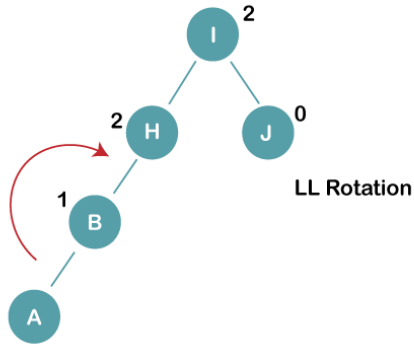


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H. The resultant balance tree is:

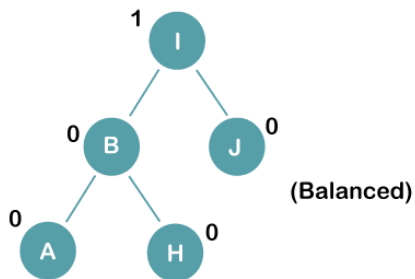
The resultant balance tree is:



## 2. Insert B, A

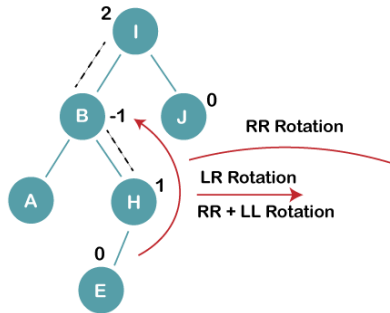


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H. The resultant balance tree is:





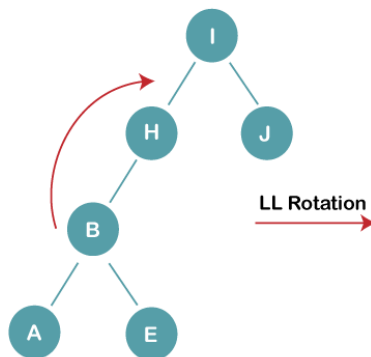
### 3. Insert E



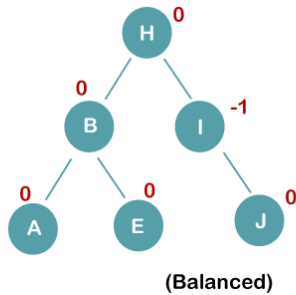
On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

**3 a) We first perform RR rotation on node B**

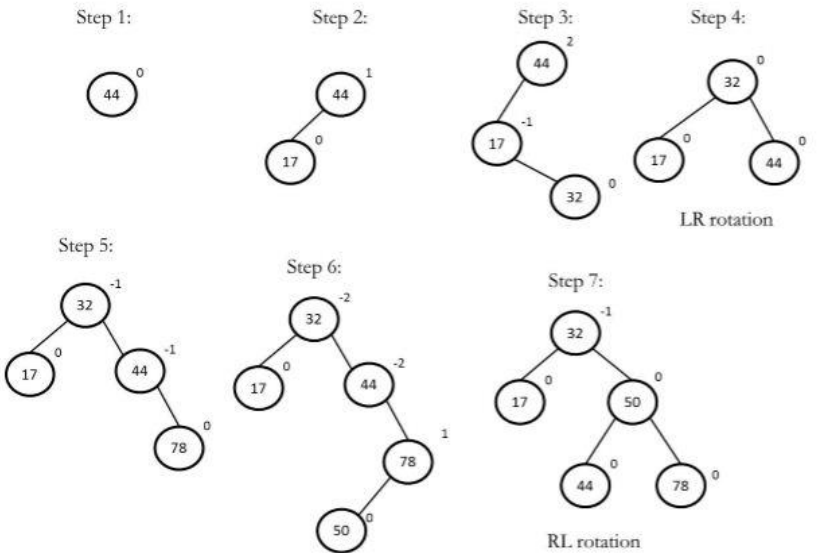
**The resultant tree after RR rotation is:**

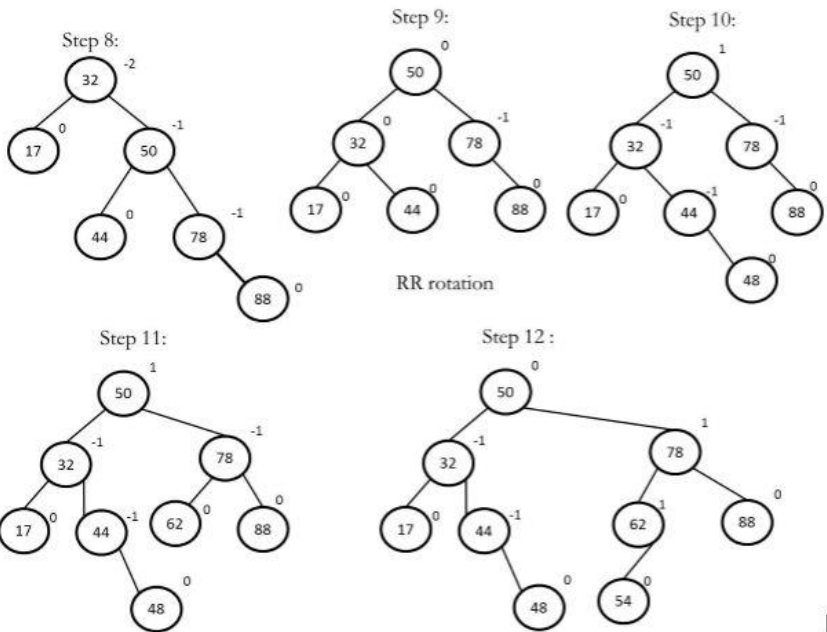


3b) We first perform LL rotation on the node I  
 The resultant balanced tree after LL rotation is:



Construct an AVL tree having the following elements  
 44, 17, 32, 78, 50, 88, 48, 62, 54

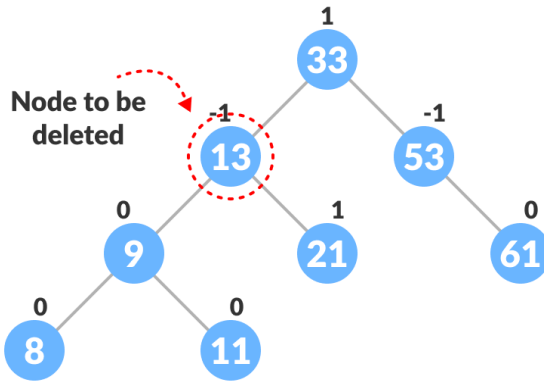




## Delete a node

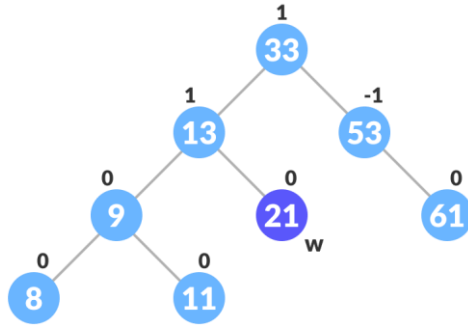
A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.

1. Locate nodeToBeDeleted (recursion is used to find nodeToBeDeleted in the code used below).

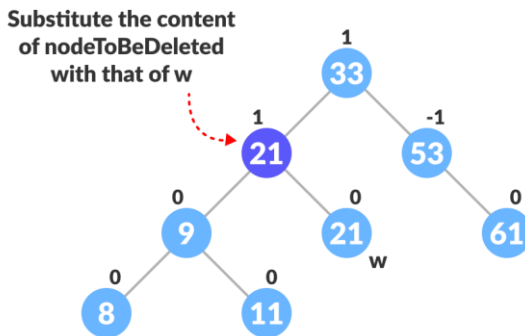


2. There are three cases for deleting a node:

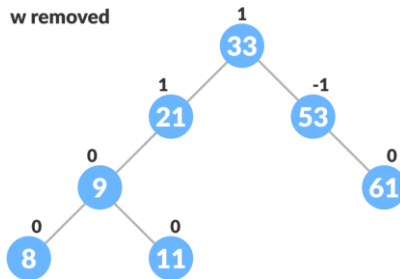
- If `nodeToBeDeleted` is the leaf node (ie. does not have any child), then remove `nodeToBeDeleted`.
- If `nodeToBeDeleted` has one child, then substitute the contents of `nodeToBeDeleted` with that of the child. Remove the child.
- If `nodeToBeDeleted` has two children, find the inorder successor `w` of `nodeToBeDeleted` (ie. node with a minimum value of key in the right subtree).



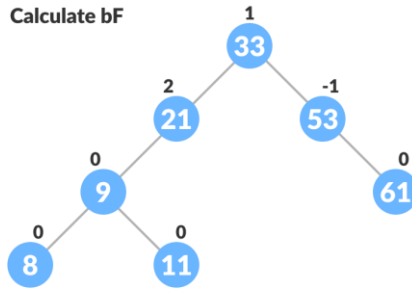
a) Substitute the contents of nodeToBeDeleted with that of w



b) Remove the leaf node w.

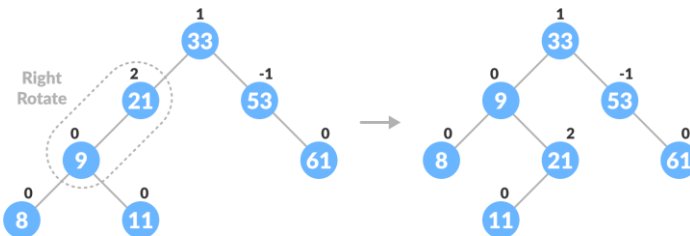


3. Update balanceFactor of the nodes.



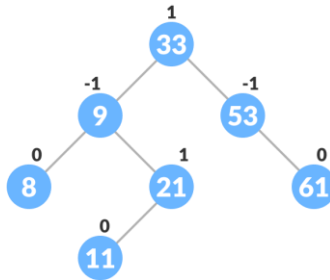
4. Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.

- a) If balanceFactor of currentNode > 1,
- b) If balanceFactor of leftChild >= 0, do right rotation.



- Else do left-right rotation.
- If balanceFactor of currentNode < -1,
- If balanceFactor of rightChild <= 0, do left rotation.
- Else do right-left rotation.

5. The final tree is:



## Applications

### 1. Databases:

AVL trees are used in databases to ensure quick data retrieval. Their self-balancing property makes them ideal for implementing index structures that allow for efficient searching, insertion, and deletion operations.

### 2. File Systems:

File systems often use AVL trees to manage directories and files. The balance of the tree ensures that file operations are performed efficiently, which is critical for maintaining the performance of the file system.

### 3. Memory Management:

AVL trees are used in memory management systems to keep track of free memory blocks. The balanced nature of AVL trees ensures quick allocation and deallocation of memory.

#### **4. Networking:**

In networking, AVL trees can be used to manage routing tables efficiently. The balanced structure helps in maintaining the routing information for quick lookup, insertion, and deletion.

#### **5. Compilers:**

Compilers use AVL trees to implement symbol tables, which store information about variables, functions, and other entities. The efficient searching capability of AVL trees helps in quick symbol resolution during compilation.

#### **6. Geographical Information Systems (GIS):**

AVL trees can be used in GIS for indexing spatial data. The balanced structure ensures efficient query processing for operations such as range searches and nearest neighbor searches.

#### **7. Telecommunication:**

In telecommunications, AVL trees can be used to manage call routing and connection management. The efficient lookup and update operations help in maintaining real-time performance.



## **8. Scheduling Systems:**

AVL trees are used in various scheduling systems to keep track of tasks and resources. The balanced nature ensures that scheduling operations are performed quickly and efficiently.

## **9. Spell Checkers:**

AVL trees can be used in spell checkers to store dictionaries. The efficient searching capability helps in quickly finding words and suggesting corrections.

## **10. Gaming:**

In gaming, AVL trees can be used to manage objects and characters in a game world. The balanced structure helps in efficient collision detection, scene management, and rendering.

## **11. Priority Queues:**

AVL trees can be used to implement priority queues. The balanced nature ensures that operations such as insert, delete, and find-minimum can be performed efficiently.

## **12. Version Control Systems:**

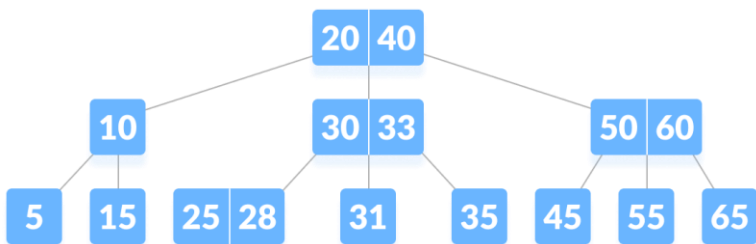
In version control systems, AVL trees can be used to manage versions of files and directories. The efficient

update and retrieval operations help in maintaining and accessing different versions quickly.

### 1.4 B-Trees

B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree.

It is also known as a height-balanced m-way tree.



#### Why do you need a B-tree data structure?

The need for B-tree arose with the rise in the need for lesser time in accessing physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. There was a need for such types of data structures that minimize the disk access.

Other data structures such as a binary search tree, avl tree, red-black tree, etc can store only one key in one node. If you have to store a large number of keys, then the height of such trees becomes very large, and the access time increases.

However, B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses.

### **Time Complexity of B-Tree:**

Sr. No.	Algorithm	Time Complexity
1.	Search	$O(\log n)$
2.	Insert	$O(\log n)$
3.	Delete	$O(\log n)$

### **Basic Operations of B Trees**

The operations supported in B trees are Insertion, deletion and searching with the time complexity of  $O(\log n)$  for every operation.

### **Insertion operation**

The insertion operation for a B Tree is done similar to the Binary Search Tree but the elements are inserted into the same node until the maximum keys are reached. The insertion is done using the following procedure –

**Step 1** – Calculate the maximum  $(m-1)$

and, minimum  $(\lceil m/2 \rceil - 1)$

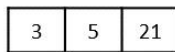
number of keys a node can hold, where  $m$  is denoted by the order of the B Tree.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

- Order  $(m) = 4$
- Maximum Keys  $(m - 1) = 3$
- Minimum Keys  $(\lceil \frac{m}{2} \rceil) - 1 = 1$
- Maximum Children = 4
- Minimum Children  $(\lceil \frac{m}{2} \rceil) = 2$

**Step 2** – The data is inserted into the tree using the binary search insertion and once the keys reach the maximum number, the node is split into half and the median key becomes the internal node while the left and right keys become its children.

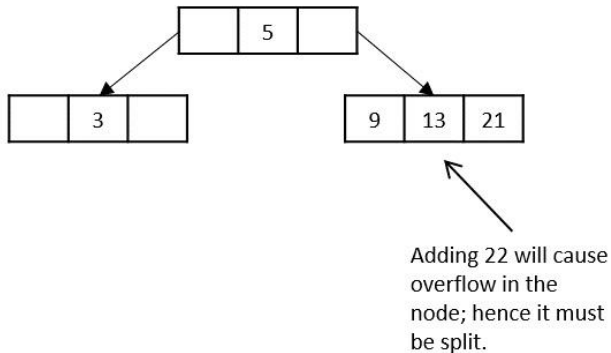
Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree



Adding 9 will cause overflow in the node; hence it must be split.

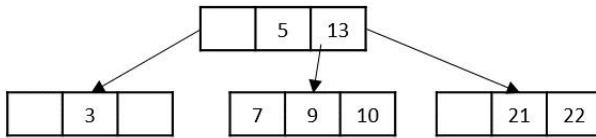
**Step 3** – All the leaf nodes must be on the same level.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree



The keys, 5, 3, 21, 9, 13 are all added into the node according to the binary search property but if we add the key 22, it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued.

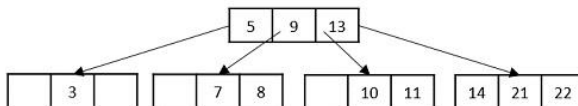
Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree



Adding 11 will cause overflow in the node; hence it must be split.

Another hiccup occurs during the insertion of 11, so the node is split and median is shifted to the parent.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

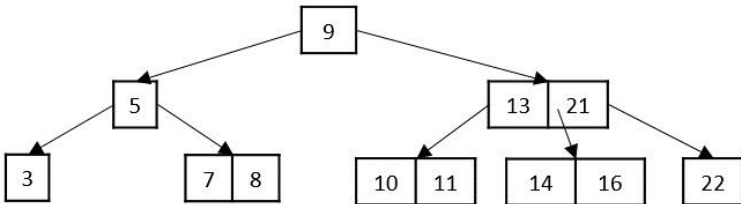


Adding 16 will cause overflow in the node; hence it must be split.

While inserting 16, even if the node is split in two parts, the parent node also overflows as it reached the maximum keys. Hence, the parent node is split first and the median key

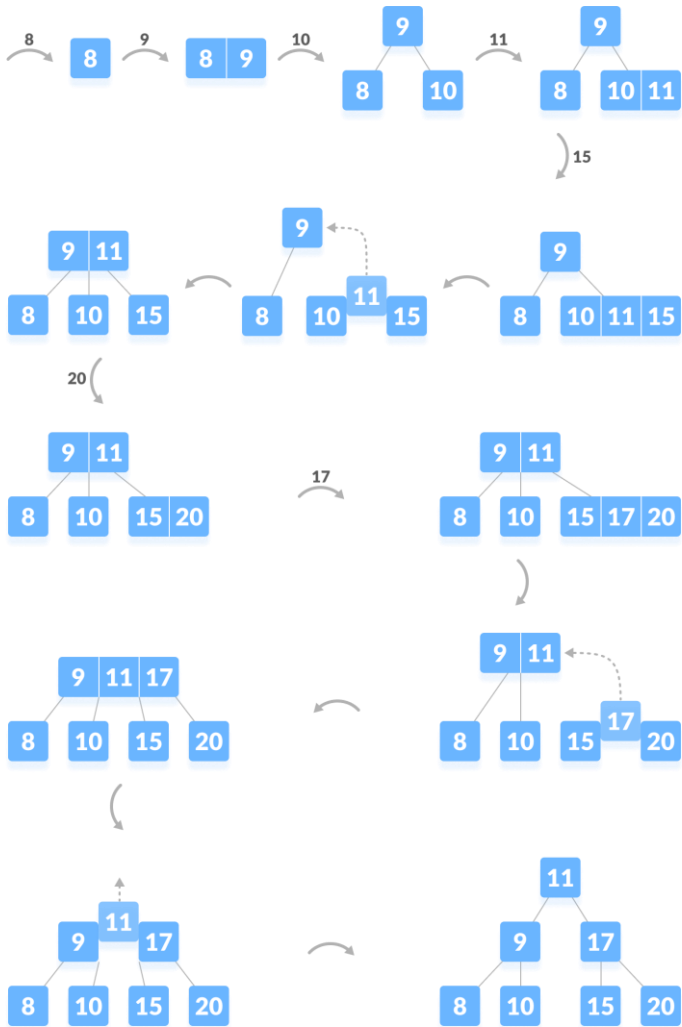
becomes the root. Then, the leaf node is split in half the median of leaf node is shifted to its parent.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree



The final B tree after inserting all the elements is achieved.

**Example 2: The elements to be inserted are 8, 9, 10, 11, 15, 20, 17**



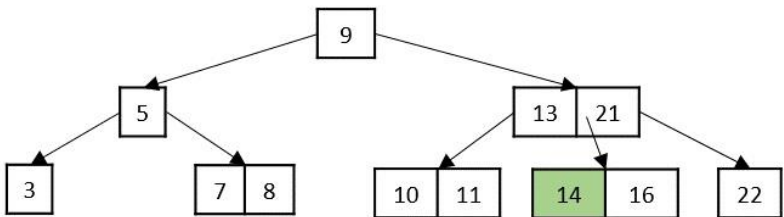


## Deletion operation

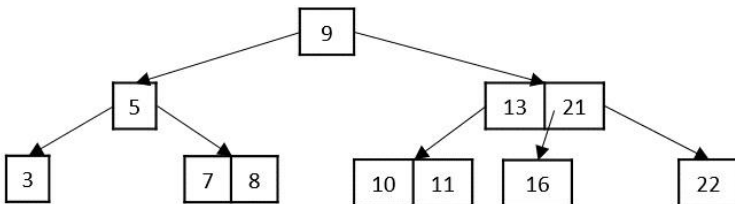
The deletion operation in a B tree is slightly different from the deletion operation of a Binary Search Tree. The procedure to delete a node from a B tree is as follows –

**Case 1** – If the key to be deleted is in a leaf node and the deletion does not violate the minimum key property, just delete the node.

Delete key 14

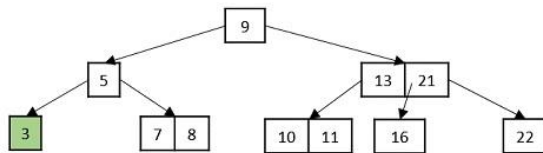


Delete key 14

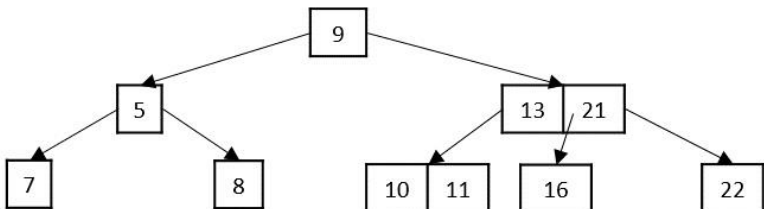


**Case 2** – If the key to be deleted is in a leaf node but the deletion violates the minimum key property, borrow a key from either its left sibling or right sibling. In case if both siblings have exact minimum number of keys, merge the node in either of them.

Delete key 3

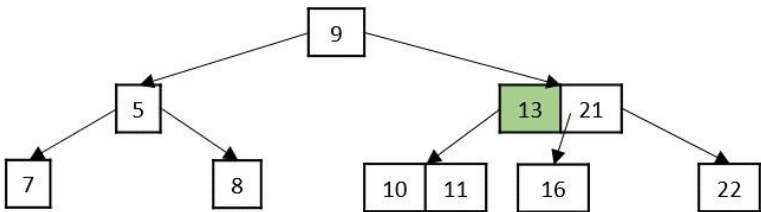


Delete key 3

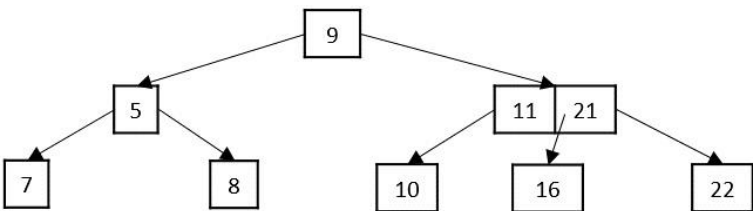


**Case 3** – If the key to be deleted is in an internal node, it is replaced by a key in either left child or right child based on which child has more keys. But if both child nodes have minimum number of keys, they're merged together.

Delete key 13

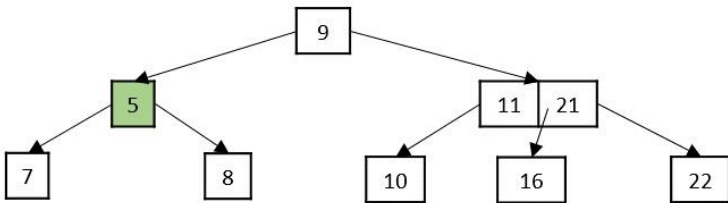


Delete key 13

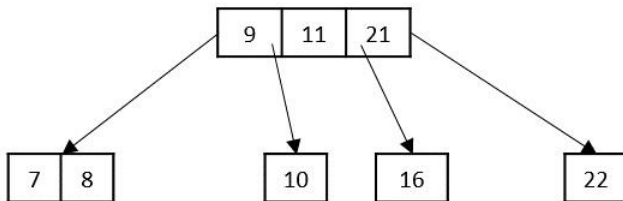


**Case 4** – If the key to be deleted is in an internal node violating the minimum keys property, and both its children and sibling have minimum number of keys, merge the children. Then merge its sibling with its parent.

Delete key 5

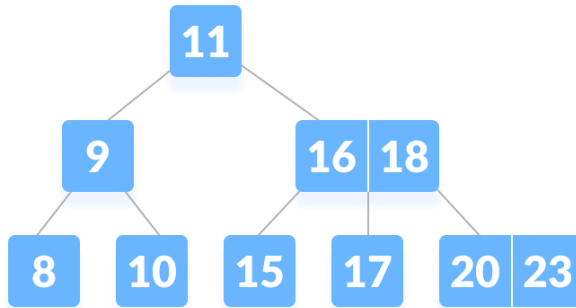


Delete key 5

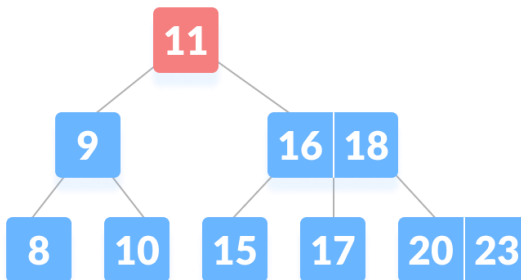


## Searching an element in a B-tree

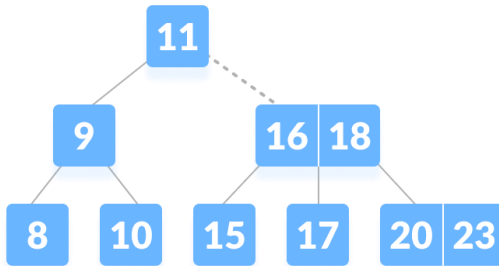
a) Let us search key  $k = 17$  in the tree below of degree 3



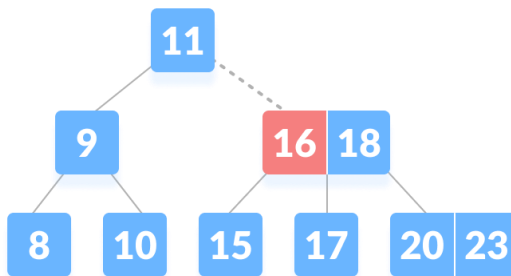
b)  $k$  is not found in the root so, compare it with the root key.



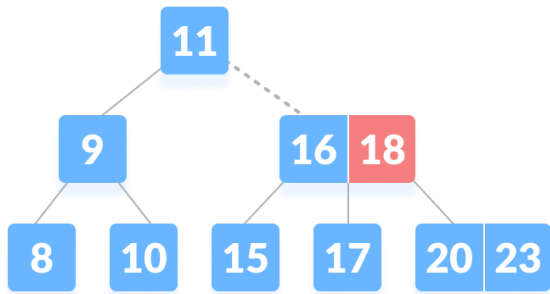
C) Since  $k > 11$ , go to the right child of the root node.



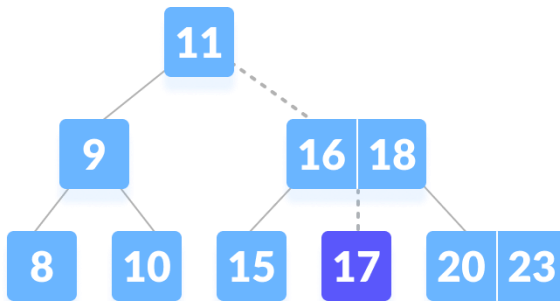
d) Compare  $k$  with 16. Since  $k > 16$ , compare  $k$  with the next key 18.



E) Since  $k < 18$ ,  $k$  lies between 16 and 18. Search in the right child of 16 or the left child of 18.



f) k is found.



### Applications

- Large databases employ it to access information stored on discs.
- Using the B-Tree, finding data in a data set can be done in a great deal less time.

- Multilevel indexing is possible with the indexing feature.
- The B-tree method is also used by the majority of servers.
- In CAD systems, B-Trees are used to catalogue and search geometric data.
- Other applications of B-Trees include encryption, computer networks, and natural language processing.
- Since accessing values stored in a large database that is stored on a disc takes a long time, B trees are used to index the data and provide quick access to the actual data stored on the disks.
- In the worst case, it takes  $O(n)$  running time to search a database with  $n$  key values that is not sorted or indexed. However, if we use B Tree to index this database, it will be searched in  $O(\log n)$  time in worst case.



## UNIT-II

**Heap Trees (Priority Queues)** – Min and Max Heaps, Operations and Applications

**Graphs** – Terminology, Representations, Basic Search and Traversals, Connected Components and Bi-connected Components, applications

**Divide and Conquer:** The General Method, Quick Sort, Merge Sort, Strassen's matrix multiplication, Convex Hull

### **2.1 Heap Trees (Priority Queues)**

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed

on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

### **Characteristics of a Priority queue**

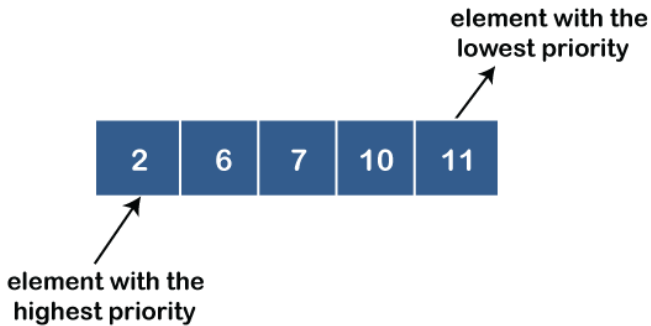
A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

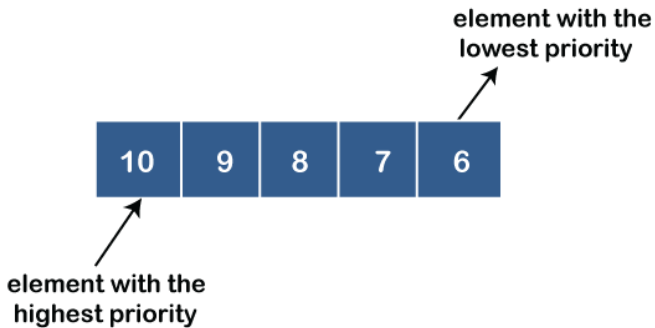
### **Types of Priority Queue**

There are two types of priority queue:

**Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



**Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Priority queue can be implemented using an array, a linked list, a **heap data structure**, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

**A heap is a complete binary tree in which the node can have the utmost two children.**

**What is heap sort?**

Heap sort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list. Heap sort is the **in-place sorting algorithm**.

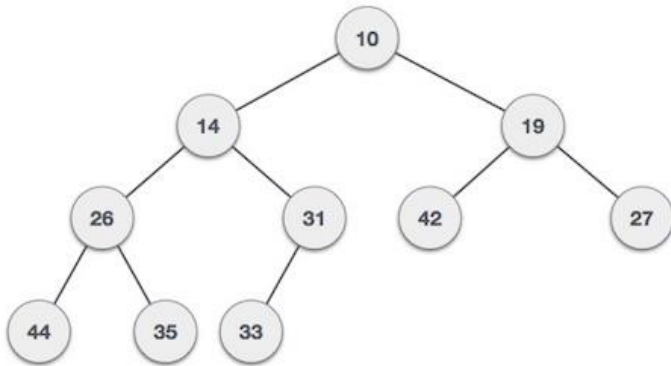
**Types**

There are two types of Heap

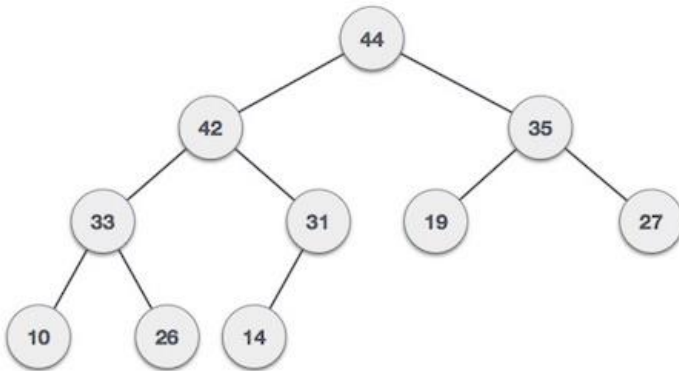
**Max Heap:** The value of each node is greater than its children.

**Min Heap:** The value of each node is Smaller than its children.

**Min-Heap** – The value of the root node is less than or equal to either of its children.



**Max-Heap** – The value of the root node is greater than or equal to either of its children.



### Step by Step Process

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

- **Step 1** - Construct a **Binary Tree** with given list of Elements.

- **Step 2** - Transform the Binary Tree into **Min Heap**.
- **Step 3** - Delete the root element from Min Heap using **Heapify** method.
- **Step 4** - Put the deleted element into the Sorted list.
- **Step 5** - Repeat the same until Min Heap becomes empty.
- **Step 6** - Display the sorted list.

## Max Heap Construction Algorithm

```

Step 1 - Create a new node at the end of heap.
Step 2 - Assign new value to the node.
Step 3 - Compare the value of this child node with its parent.
Step 4 - If value of parent is less than child, then swap them.
Step 5 - Repeat step 3 & 4 until Heap property holds.

```

## Max Heap Deletion Algorithm

```

Step 1 - Remove root node.
Step 2 - Move the last element of last level to root.
Step 3 - Compare the value of this child node with its parent.
Step 4 - If value of parent is less than child, then swap them.
Step 5 - Repeat step 3 & 4 until Heap property holds.

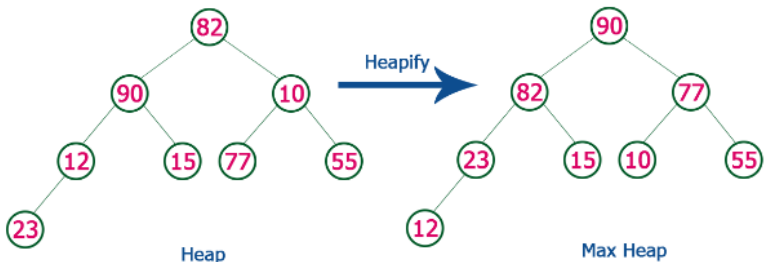
```

## Example

Consider the following list of unsorted numbers which are to be sort using Heap Sort

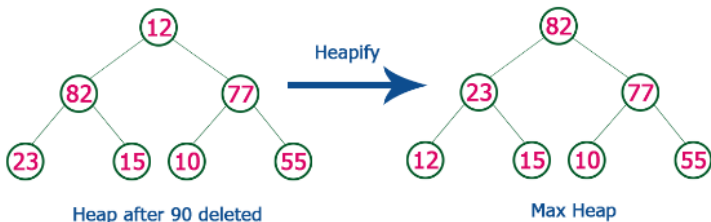
**82, 90, 10, 12, 15, 77, 55, 23**

**Step 1** - Construct a Heap with given list of unsorted numbers and convert to Max Heap



**90, 82, 77, 23, 15, 10, 55, 12**

**Step 2** - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



**12, 82, 77, 23, 15, 10, 55, 90**

**Step 3** - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

**12, 55, 77, 23, 15, 10, 82, 90**

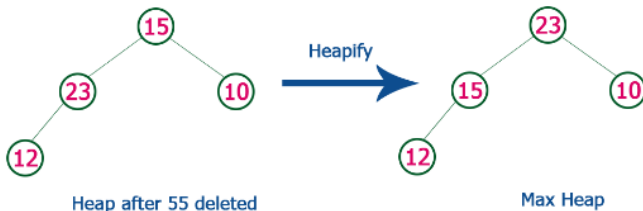
**Step 4** - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 77 with 10.

**12, 55, 10, 23, 15, 77, 82, 90**

**Step 5** - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

**12, 15, 10, 23, 55, 77, 82, 90**



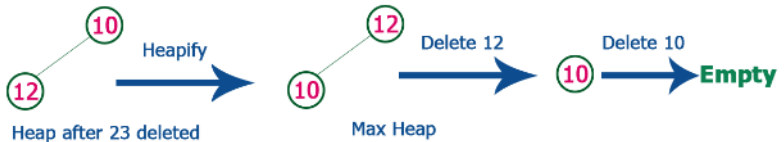
**Step 6** - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 23 with 12.

**12, 15, 10, 23, 55, 77, 82, 90**

**Step 7** - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

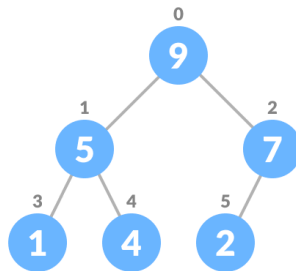
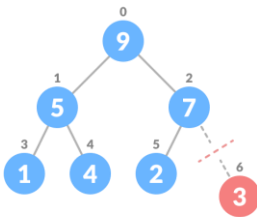
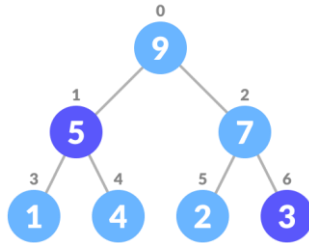
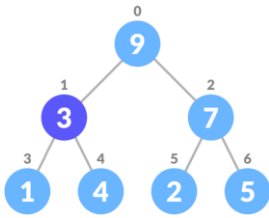
**10, 12, 15, 23, 55, 77, 82, 90**

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

## Delete

Deleting an element from a priority queue (max-heap) is done as follows:

- Select the element to be deleted.
- Swap it with the last element
- Remove the last element.
- Heapify the tree



## Applications

### 1. Systems and Embedded Systems:

Heap sort is often used in systems where memory constraints are tight. Its in-place sorting characteristic, which requires only a constant amount of additional storage space, makes it ideal for embedded systems and real-time applications.

## **2. Operating Systems:**

Heap sort is utilized in operating systems for job scheduling. The algorithm helps in efficiently managing the order of jobs by sorting them based on priority.

## **3. Network Traffic Management:**

In network systems, heap sort is used to manage packet scheduling. Sorting packets based on priority or arrival time ensures efficient handling of network traffic.

## **4. Event Simulation:**

In discrete event simulation, heap sort is used to manage the event queue. The algorithm helps in efficiently sorting and processing events in chronological order.

## **5. Data Stream Processing:**

Heap sort is useful in scenarios where a continuous stream of data needs to be sorted in real-time. The algorithm's efficiency in handling large data streams makes it suitable for such applications.

## **6. Priority Queue Implementation:**

Heap sort is foundational in implementing priority queues. Priority queues are widely used in various applications such as task scheduling, shortest path algorithms (like Dijkstra's), and bandwidth management.

## **7. Selection Algorithms:**

Heap sort is used in selection algorithms to find the k-th smallest (or largest) element in an array. The algorithm's ability to efficiently sort and maintain order is leveraged in these scenarios.

## **8. Graph Algorithms:**

Many graph algorithms, such as Prim's and Dijkstra's for finding minimum spanning trees and shortest paths, respectively, use heap sort or its underlying heap data structure for efficient edge and vertex selection.

## **9. External Sorting:**

Heap sort is useful in external sorting, where data that cannot fit into memory needs to be sorted. It is often used in combination with other algorithms to manage and merge large data sets.

## **10. Database Management:**

In databases, heap sort is used to sort large datasets. Its efficiency in handling large volumes of data makes it suitable for indexing and query optimization.

### **11. Real-Time Systems:**

Heap sort is applied in real-time systems where guarantees on time complexity are crucial. Its predictable performance helps in ensuring that time constraints are met.

### **12. Resource Allocation:**

Heap sort is used in resource allocation problems to efficiently assign resources based on priority or other criteria. This is common in cloud computing and other resource management applications.

### **13. Multimedia Applications:**

In multimedia applications, heap sort is used to manage and sort large collections of data, such as images, audio, and video files, ensuring efficient storage and retrieval.

### **14. E-commerce and Online Marketplaces:**

Heap sort is used to sort product listings based on various criteria such as price, rating, and relevance. This helps in enhancing the user experience by providing sorted and relevant search results.

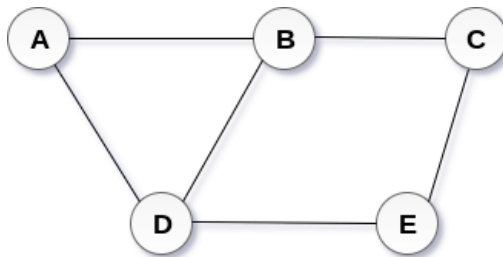
## 2.2 Graphs

A graph is a non-linear kind of data structure made up of nodes or vertices and edges. The edges connect any two nodes in the graph, and the nodes are also known as vertices.

### Definition

A graph  $G$  can be defined as an ordered set  $G(V, E)$  where  $V(G)$  represents the set of vertices and  $E(G)$  represents the set of edges which are used to connect these vertices.

A Graph  $G(V, E)$  with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



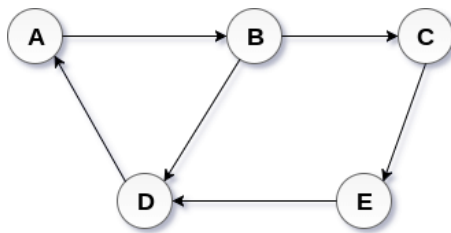
### Undirected Graph

#### Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the

directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node. A directed graph is shown in the following figure.



**Directed Graph**

## 2.3 Graph Terminology

### Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

## Connected Graph

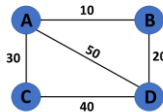
A connected graph is the one in which some path exists between every two vertices  $(u, v)$  in  $V$ . There are no isolated nodes in connected graph.

## Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain  $n(n-1)/2$  edges where  $n$  is the number of nodes in the graph.

## Weighted Graph

A graph  $G = (V, E)$  is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



## Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.



## Adjacent Nodes

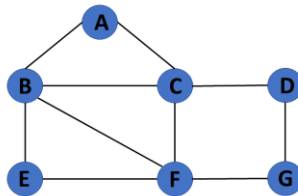
If two nodes  $u$  and  $v$  are connected via an edge  $e$ , then the nodes  $u$  and  $v$  are called as neighbours or adjacent nodes.

## Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

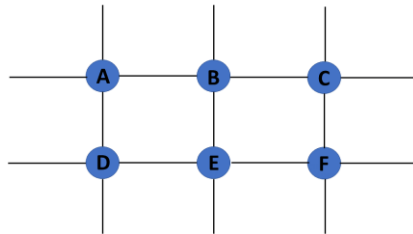
## Finite Graph

The graph  $G=(V, E)$  is called a finite graph if the number of vertices and edges in the graph is limited in number



## Infinite Graph

The graph  $G=(V, E)$  is called a finite graph if the number of vertices and edges in the graph is interminable.



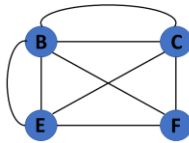
### **Trivial Graph**

A graph  $G = (V, E)$  is trivial if it contains only a single vertex and no edges.



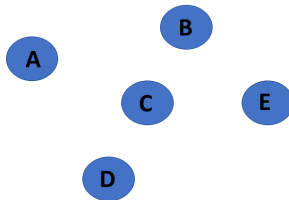
### **Multi Graph**

If there are numerous edges between a pair of vertices in a graph  $G = (V, E)$ , the graph is referred to as a multigraph. There are no self-loops in a Multigraph.



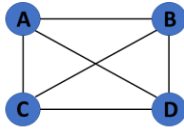
## Null Graph

It's a reworked version of a trivial graph. If several vertices but no edges connect them, a graph  $G = (V, E)$  is a null graph.



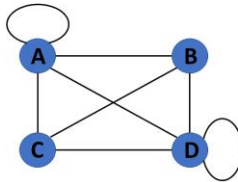
## Complete Graph

If a graph  $G = (V, E)$  is also a simple graph, it is complete. Using the edges, with  $n$  number of vertices must be connected. It's also known as a full graph because each vertex's degree must be  $n-1$ .



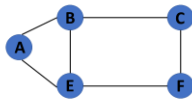
### Pseudo Graph

If a graph  $G = (V, E)$  contains a self-loop besides other edges, it is a pseudograph.



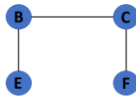
### Cyclic Graph

If a graph contains at least one graph cycle, it is considered to be cyclic.



## Acyclic Graph

When there are no cycles in a graph, it is called an acyclic graph.



## 2.4 Representation of Graphs in Data Structures

Graphs in data structures are used to represent the relationships between objects. Every graph consists of a set of points known as vertices or nodes connected by lines known as edges. The vertices in a network represent entities.

The most frequent graph representations are the two that follow:

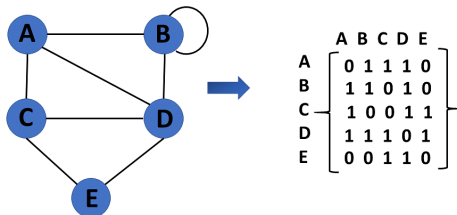
- Adjacency matrix
- Adjacency list

## Adjacency matrix

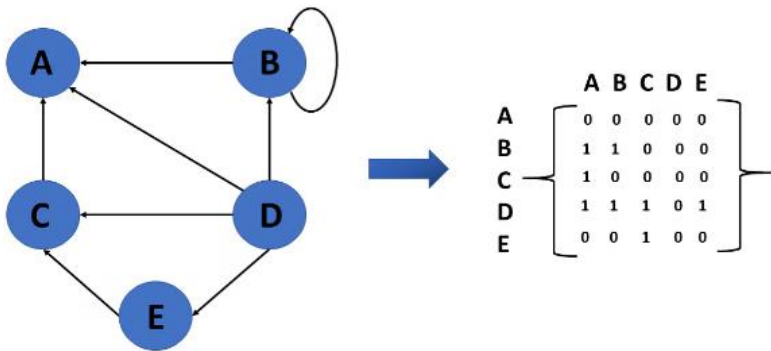
An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's). Let's assume there are  $n$  vertices in the graph. So, create a 2D matrix  $\text{adjMat}[n][n]$  having dimension  $n \times n$ .

- If there is an edge from vertex  $i$  to  $j$ , mark  $\text{adjMat}[i][j]$  as 1.
- If there is no edge from vertex  $i$  to  $j$ , mark  $\text{adjMat}[i][j]$  as 0.

## Undirected Graph Representation

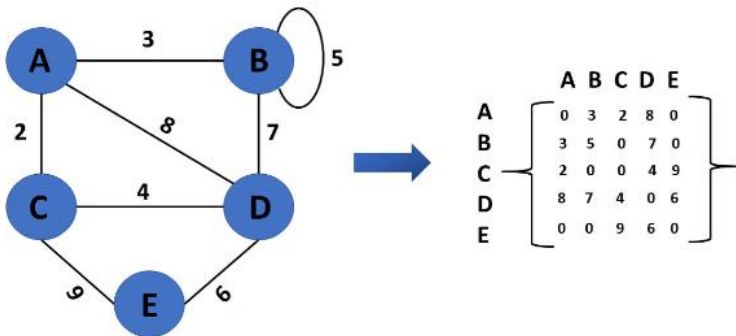


## Directed Graph Representation



## Weighted Undirected Graph Representation

Weight or cost is indicated at the graph's edge, a weighted graph representing these values in the matrix.



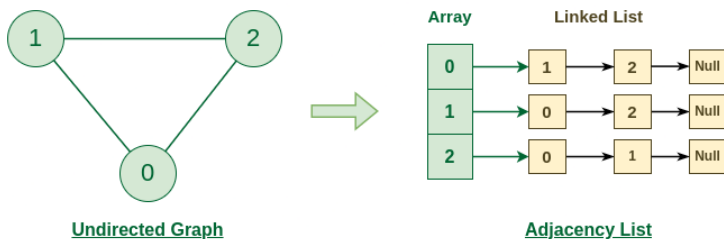
## Adjacency List

- A linked representation is an adjacency list.
- You keep a list of neighbors for each vertex in the graph in this representation. It means that each

vertex in the graph has a list of its neighboring vertices.

### **Representation of Undirected Graph to Adjacency list:**

The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



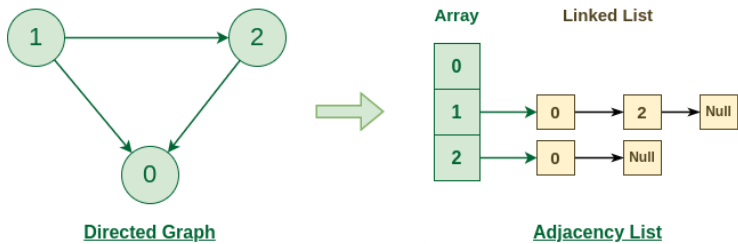
### Graph Representation of Undirected graph to Adjacency List

### **Representation of Directed Graph to Adjacency list:**

The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at



indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



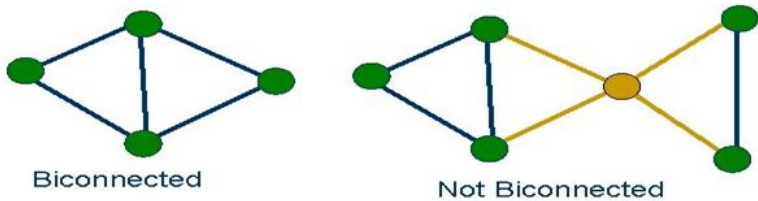
Graph Representation of Directed graph to Adjacency List

## 2.5 Bi-connected components

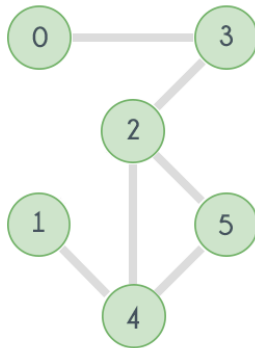
A graph is said to be Biconnected if:

- It is connected, i.e. it is possible to reach every vertex from every other vertex, by a simple path.
- Even after removing any vertex the graph remains connected.

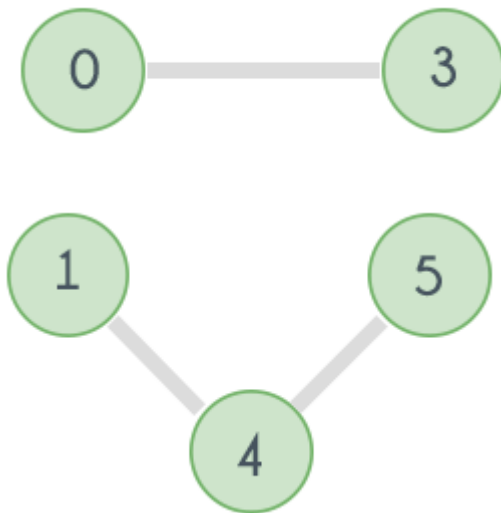
For example, consider the graph in the following figure



Removing any of the vertices does not increase the number of connected components. So the given graph is Biconnected.

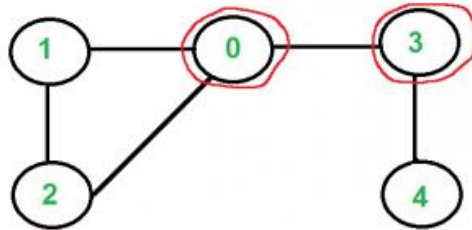


In the above graph if the vertex 2 is removed, then here's how it will look:



## Articulation point or cut point

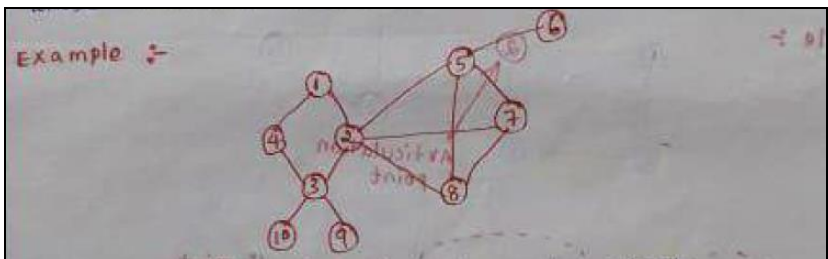
If a point in a graph becomes separated from the entire graph upon removal, it is referred to as an Articulation Point or Cut-Vertex.



Articulation points are 0 and 3

## Construction of Bi-connected graph

- Check the given graph whether it is bi-connected or not.
- If the given graph is not bi-connected then identify all the articulation points.
- If articulation points exist, determine a set of edges whose inclusion makes the graph bi-connected.



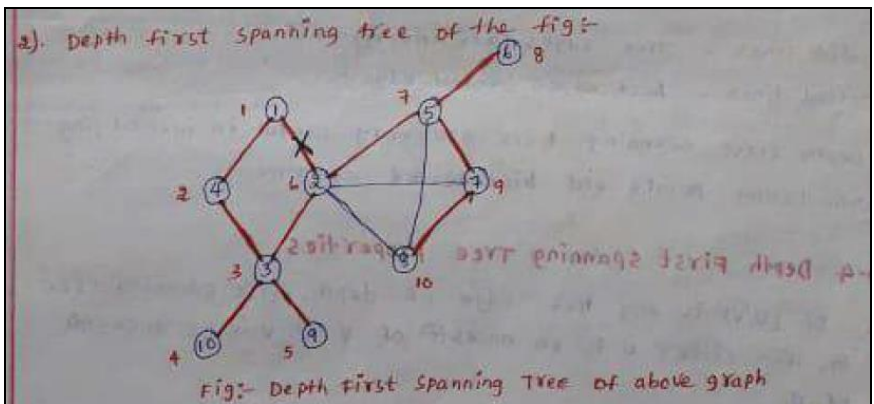
The articulation points are 2, 3, and 5

To transform the graph into bi-connected graph, the new edges are included corresponding to the articulation point.

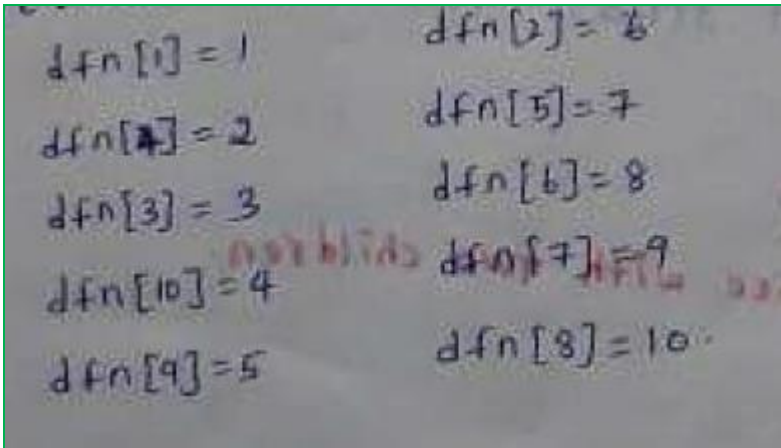
Edges corresponding to the articulation point 3- (4,10)(10,9)

Edges corresponding to the articulation point 2-(1,5)(3,8)

Edges corresponding to the articulation point 5-(6,7)



In the fig: there is a number of outside each vertex, corresponds to the order in which a DFS visits these vertices and are named as depth first numbers(dfns) of the vertex ie



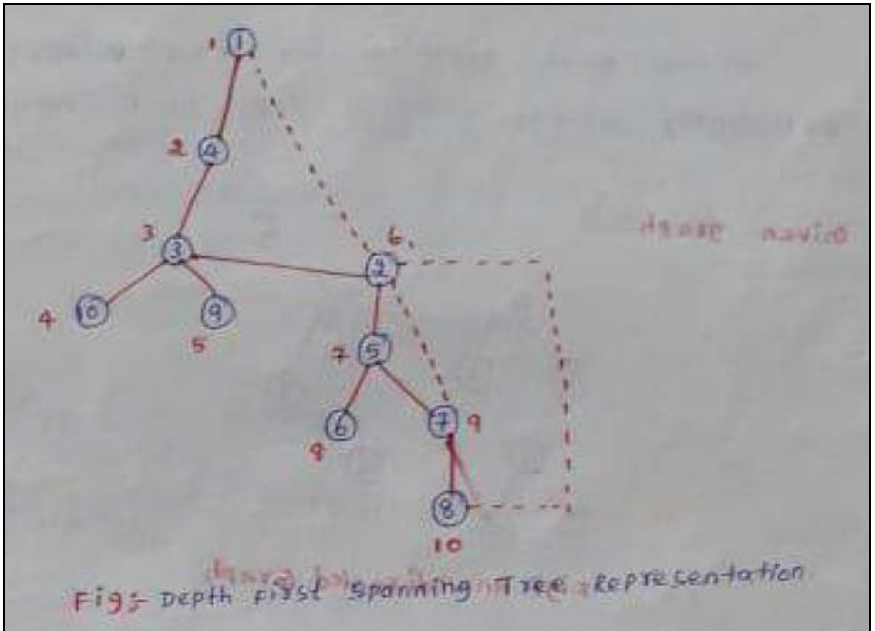
The solid edges of fig, will form a depth first spanning tree.

The depth first spanning tree representation is

**Solid lines:** Tree edges

**-Dotted lines:** Back edges

-Depth first spanning trees are very useful in identifying articulation points and bi-connected components.



## Depth First Spanning tree properties

✓ 2.7.4 Depth First spanning tree properties

- 1) If  $(u, v)$  is any tree edge of depth first spanning tree, then either  $u$  is an ancestor of  $v$  or  $v$  is an ancestor of  $u$ .  
More words to use primitive terms like self

Example :-

Fig:- A tree with  $(u, v)$  Edge

- 2) The root node of a depth first spanning tree is an articulation point iff it has atleast two children.

Example :-

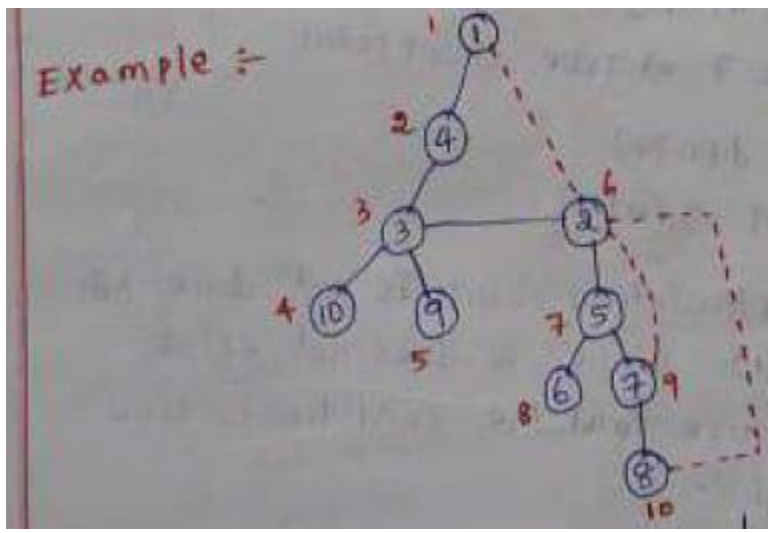
Fig:- A tree with two children.

For each vertex,  $u$ , define  $L[u]$  as follows.

$$L[u] = \min \{ \text{dfn}[u], \min \{ L[w] \mid w \text{ is a child of } u \}, \min \{ \text{dfn}[w] \mid (u,w) \text{ is a backedge} \} \}$$

where  $L[u]$  is the lowest depth first number that can be reached from  $u$  using a path of descendants followed by at most one back edge.

b) If  $u$  is an articulation point iff  $u$  has a child  $w$  such that  $L[w] \geq \text{dfn}[u]$ .



$$L[10] = \min \{ 4, -, - \}$$

$$L[10] = 4$$

$$L[9] = \min \{ 5, -, - \}$$

$$L[9] = 5$$

$$L[8] = \min \{ 8, -, - \}$$

$$L[8] = 8$$

$$L[9] = \min \{ 10, -, 6 \}$$

$$L[9] = 6$$

$$L[7] = \min \{ 9, 6, 6 \}$$

$$L[7] = 6$$

$$L[5] = \min \{ 7, 6, - \}$$

$$L[5] = 6$$

$$L[2] = \min \{ 6, 6, 1 \}$$

$$L[2] = 1$$

$$L[3] = \min \{ 3, 1, - \}$$

$$L[3] = 1$$

$$L[4] = \min \{ 2, 1, - \}$$

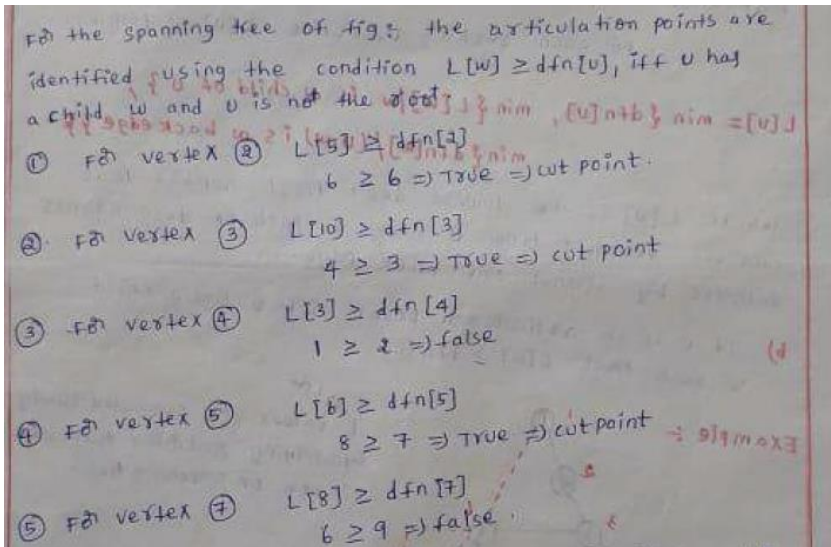
$$L[4] = 1$$

$$L[1] = \min \{ 1, 1, - \}$$

$$L[1] = 1$$

$$(ie) L[1:10] = \{ 1, 1, 1, 1, 6, 8, 6, 6, 5, 4 \}$$





The condition to check articulation point is not done for the vertices 6, 8, 9 and 10. Since  $w$  does not exist in the spanning tree structure and the condition is true for the vertices 2, 3 and 5. Therefore the articulation points are 2, 3 and 5.

## 2.6 Graph Traversal

The process of visiting or updating each vertex in a graph is known as graph traversal.

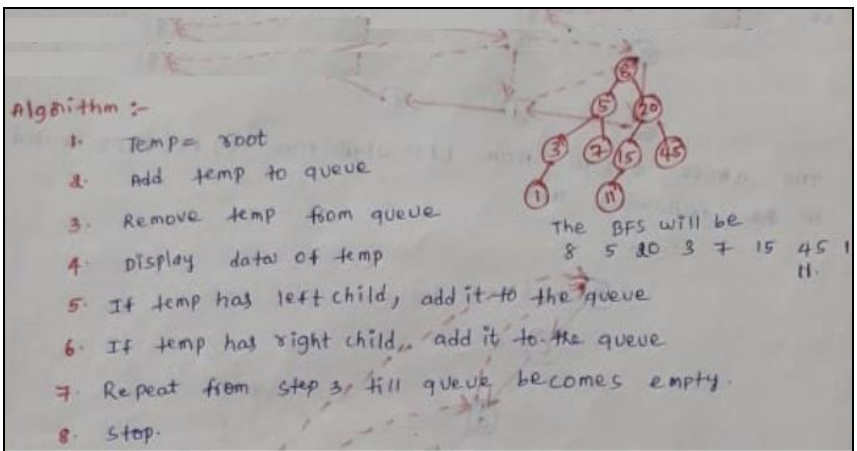
There are two techniques to implement a graph traversal algorithm:

- Breadth-first search
- Depth-first search

### Breadth-first search(BFS)

We traverse the tree level wise from left to right starting from the root. It is implemented using a queue.

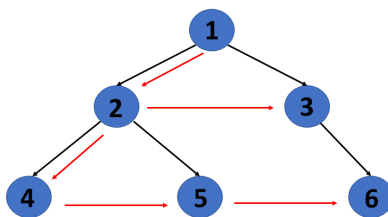
Algorithm



Algorithm :-

1. Temp = root
2. Add temp to queue
3. Remove temp from queue
4. Display data of temp
5. If temp has left child, add it to the queue
6. If temp has right child, add it to the queue
7. Repeat from step 3, till queue becomes empty.
8. Stop.

The BFS will be 8 5 20 3 7 15 45 11.

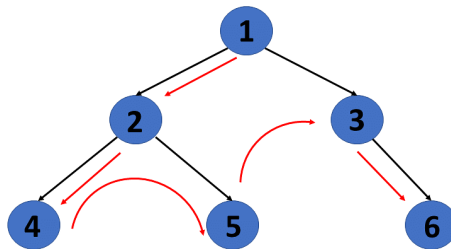
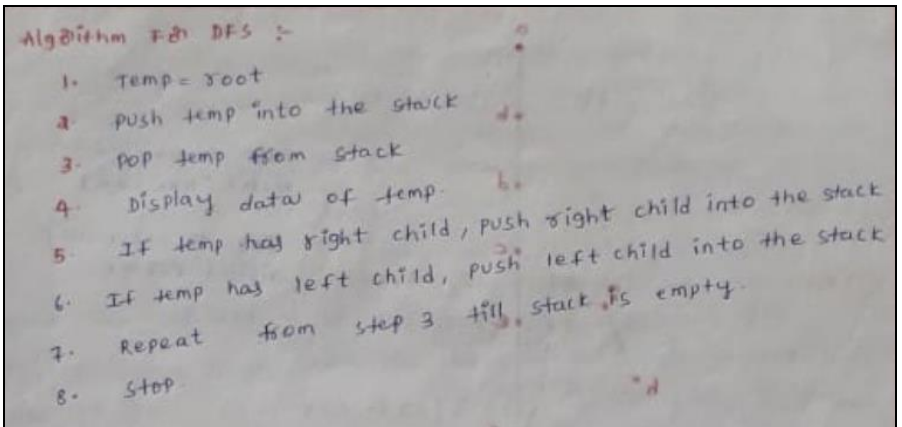


BFS 

1	2	3	4	5	6
---	---	---	---	---	---

## Depth-first search(DFS)

In the DFS traversal method, we start from the root and traverse left as far as we can go once the leftmost end is reached, we go to the right child of the node in the path and move to its left most end.



DFS 

1	2	4	5	3	6
---	---	---	---	---	---

## 2.7 Applications

### 1. Social Networks:

- **Friendship Networks:** Graphs represent users as nodes and friendships as edges, enabling analysis of social relationships, community detection, and influence propagation.
- **Professional Networks:** Platforms like LinkedIn use graphs to model professional connections and recommend potential contacts.

### 2. Transportation and Logistics:

- **Route Planning:** Graphs model transportation networks (roads, railways, flight routes) to find the shortest or fastest routes using algorithms like Dijkstra's or A\*.
- **Supply Chain Management:** Graphs optimize logistics by modeling supply chains, managing inventory, and minimizing transportation costs.

### 3. Telecommunications:

- **Network Design:** Graphs represent communication networks, optimizing the layout of networks to ensure efficient data transfer and minimal latency.

- **Routing:** Graph algorithms find optimal paths for data packets in a network, ensuring efficient and reliable communication.

#### 4. Biology and Bioinformatics:

- **Protein-Protein Interaction Networks:** Graphs model interactions between proteins, helping understand cellular processes and discover potential drug targets.
- **Genetic Networks:** Graphs represent genetic regulatory networks, analyzing gene interactions and expression patterns.

#### 6. Computer Science:

- **Compiler Design:** Graphs represent program structures, optimizing code through control flow graphs and dependency graphs.
- **Web Page Ranking:** Search engines like Google use graph algorithms (e.g., PageRank) to rank web pages based on link structures.

#### 7. Geographical Information Systems (GIS):

- **Map Services:** Graphs model geographical regions, providing route planning, geospatial analysis, and location-based services.

- **Urban Planning:** Graphs assist in designing efficient public transportation systems and optimizing urban infrastructure.

## 8. Recommendation Systems:

- **Collaborative Filtering:** Graphs model user-item interactions, recommending products, movies, or content based on user preferences and behaviors.
- **Content Recommendation:** Platforms like YouTube and Netflix use graphs to suggest relevant content based on user interaction patterns.

## 9. Electrical Engineering:

- **Circuit Design:** Graphs represent electrical circuits, analyzing and optimizing circuit layouts for better performance and reliability.
- **Power Grid Management:** Graphs model power grids, optimizing energy distribution and managing grid stability.

## 10. Finance and Economics:

- **Stock Market Analysis:** Graphs represent relationships between stocks, aiding in market analysis, portfolio optimization, and risk management.

- **Fraud Detection:** Graphs detect fraudulent activities by analyzing transaction patterns and identifying anomalies.

## 11. Artificial Intelligence and Machine Learning:

- **Knowledge Graphs:** Graphs represent relationships between entities, enabling semantic search, question answering, and AI-driven insights.
- **Graph Neural Networks:** Graphs are used in deep learning to model complex relationships and improve prediction accuracy in various applications.

## 12. Linguistics and Natural Language Processing:

- **Syntax Trees:** Graphs represent the syntactic structure of sentences, aiding in language understanding and processing.
- **Semantic Networks:** Graphs model the meaning of words and phrases, improving natural language understanding and generation.

## 13. Gaming and Entertainment:

- **Game Maps:** Graphs model game environments, managing path finding for characters and optimizing game dynamics.

- **Storytelling:** Graphs represent story elements and their relationships, enabling dynamic and interactive storytelling experiences.

## 2.8 Introduction to Divide and conquer

If a problem is given, we divide it into some k number of sub-problems with k size each. If we get the solution of each part then we stop at that point, otherwise we still divide the problem. We solve all individual sub-problems and combine all these solutions of sub-problems which is the required solution of a given problem.

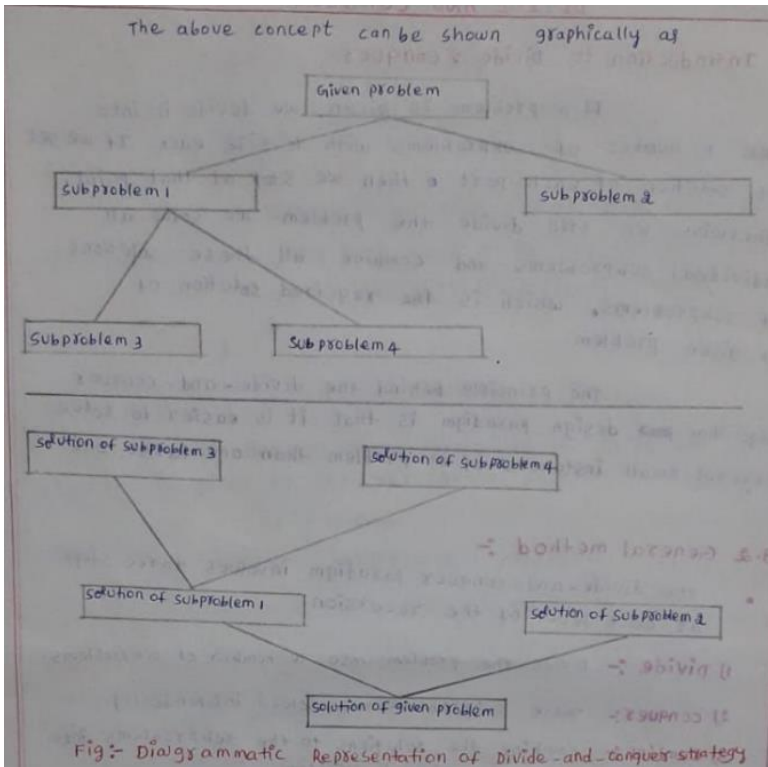
The principle behind the divide and conquer algorithm design paradigm is that it is easier to solve several small instances of a problem than one large one.

### 2.8.1 General method

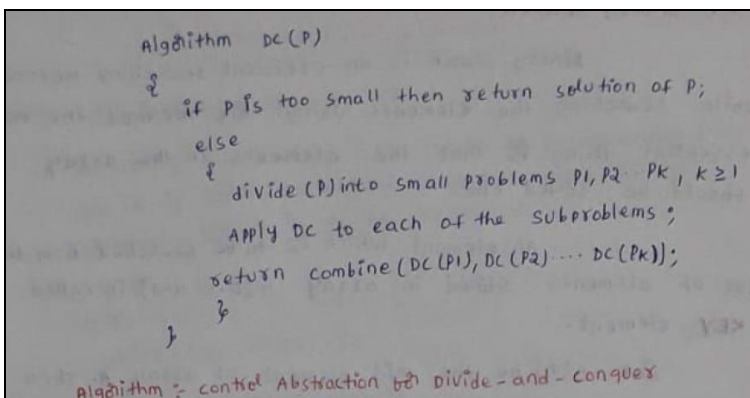
The divide and conquer paradigm involves three steps at each level of the recursion.

1. **Divide:** Divide the problem into a number of sub-problems.
2. **Conquer:** These sub-problems are solved independently.
3. **Combine:** Combine the solutions to the sub-problems into the solution for the original problem.





## Algorithm



## 2.9 Applications of Divide and Conquer

- Quick sort
- Merge sort
- Stassen's matrix multiplication.
- Convex Hull

### 2.9.1 Quick Sort

Quick sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out. The 3steps of quick sort are as follows.

**a) Divide:** Rearrange the elements and split the array into two sub arrays and an element in between. Each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater that middle element.

**b) Conquer:** recursively sort the two sub arrays

**c) Combine:** combine all the sorted elements in a group to form a list of sorted elements.

### 3.3.2.1 Algorithm

The algorithm for quick sort is as given below

Algorithm quicksort (low, high)

{  
// The elements stored in array  $a[1, n]$  are not in sorted  
// initially.  $n$  is total number of elements. low is  
// initially at index 1 and high is at index  $n$  of an  
// array  $a$ . By this recursive algorithm, the array will be  
// sorted in ascending order.

if (low < high) then

{  
// partition is used to divide the array into two  
// sub arrays and  $k$  is the position of partitioning  
// element.

$k = \text{partition}(a, \text{low}, \text{high} + 1);$

// recursively sort the sub arrays

quicksort (low,  $k - 1$ );

quicksort ( $k + 1$ , high);

}

}

algorithm = sorting by partitioning



Elements that are less than pivot



Pivot element



Elements that are greater than pivot

Algorithm partition(a, low, high)

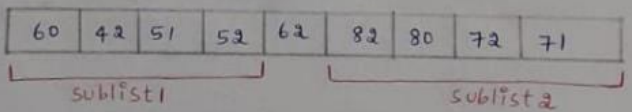
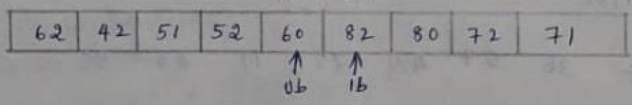
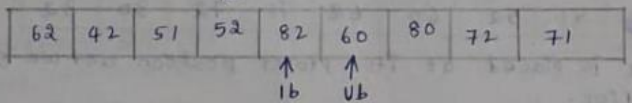
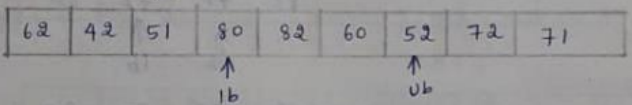
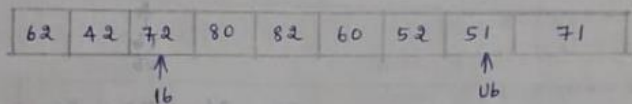
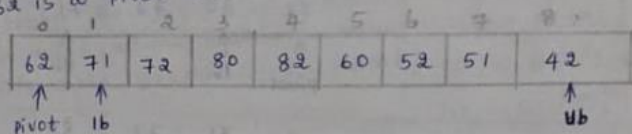
```
{
    // first element of the sub array is assumed to be pivot
    pivot = a[low];
    lb = low;
    ub = high;
    while (lb <= ub) do
    {
        while (a[lb] <= pivot)
            lb = lb + 1;
        while (a[ub] >= pivot)
            ub = ub - 1;
        if (lb < ub) then
            swap(a, low, high);
    }
    a[low] = a[ub];
    a[ub] = pivot;
    return ub;
}
```

1) if  $A[i] < \text{pivot}$  then  $i++$   
2) if  $A[j] > \text{pivot}$  then  $j--$   
3) if  $A[i] < A[j]$  then swap  $i$  &  $j$   
4) while crossing  $j$  &  $i$  then swap  $A[i]$  &  $A[\text{pivot}]$  &  $\text{low}$ .

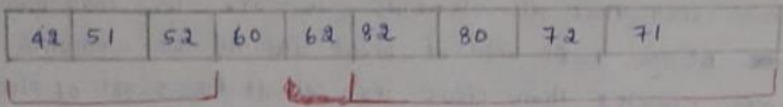
Algorithm: partition the array  $a[\text{low}:\text{high}-1]$  about  $a[\text{low}]$

Example :- considers the list of unsorted elements as  
 62, 71, 72, 80, 82, 60, 52, 51, 42.

sol:- we will consider pivot =  $a[\text{low}]$  where  $\text{low} = 0$ . Hence  
 62 is a pivot element.



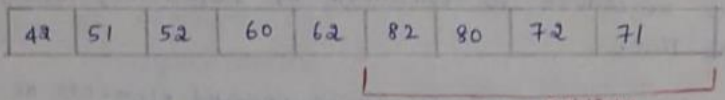
Now we will consider 60 as pivot for sublist1 and repeat above steps to place pivot at proper position. proper position of pivot means all the elements that are less than pivot should be left side of pivot and all the elements that are greater than pivot at right side of pivot.  
 Hence we get.



sublist 1

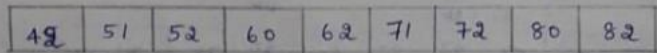
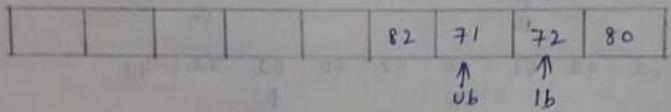
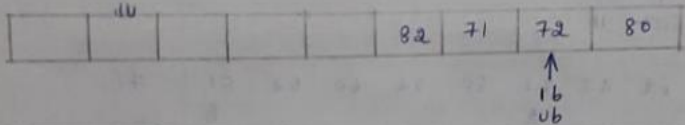
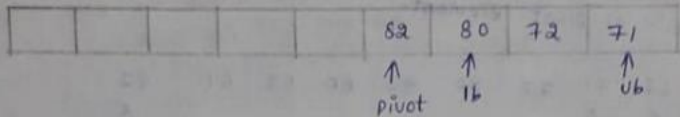
sublist 2

Now from sublist 1, element 42 becomes a pivot element and we will sort sublist 1



sublist 2

Now we will consider sublist 2. In this list the element 82 becomes a pivot element. we will repeat the procedure of partition and we will get



Thus 82 is placed at its proper position and we get a sorted list.

Example:- 36 57 44 25 19 28 89

## 2.9.2 Merge Sort

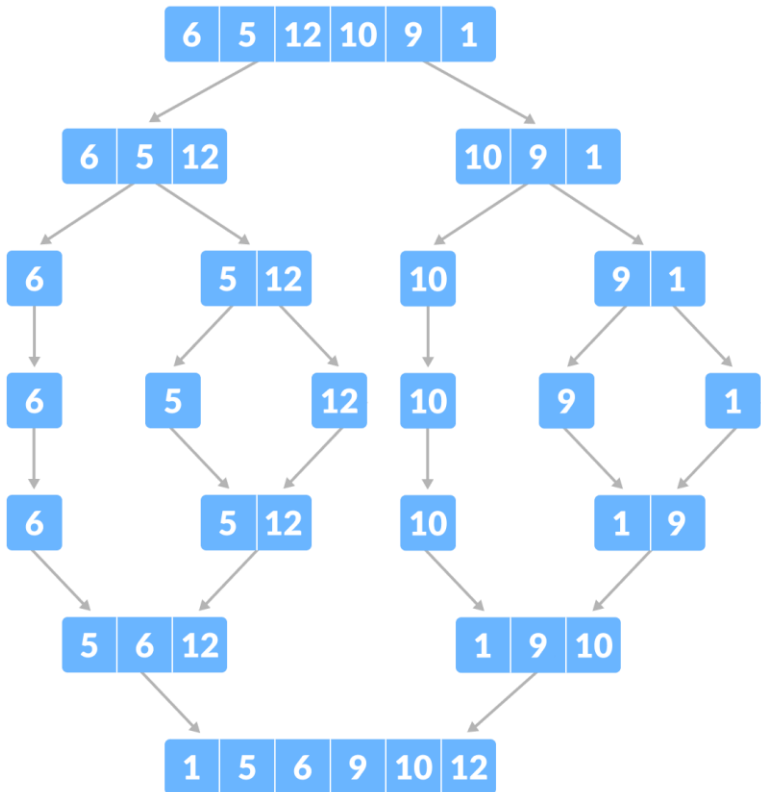
The merge sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out. Merge sort on an input array with  $n$  elements consists of three steps:

- a) **Divide:** Partition array into two sublists  $s_1$  and  $s_2$  with  $n/2$  elements each.
- b) **Conquer:** recursively sort  $s_1$  and  $s_2$ .
- c) **Combine:** merge  $s_1$  and  $s_2$  into a unique sorted group.

```
Algorithm MergeSort(Arr, start, end)
1. If start < end Then true
2.   mid = (start + end)/2
3.   MergeSort(Arr, start, mid)
4.   MergeSort(Arr, mid + 1, end)
5.   Merge(Arr, start, mid, end)
```

```
Algorithm Merge(Arr, start, mid, end)
1. temp = Create array temp of same size Arr
2. i = start, j = mid + 1, k = start
3. While i <= mid and j <= end
4.   If Arr[i] > Arr[j] Then
5.     temp[k++] = Arr[j++]
6.   Else
7.     temp[k++] = Arr[i++]
8. While i <= mid
9.   temp[k++] = Arr[i++]
10. While j <= end
11.  temp[k++] = Arr[j++]
12. Loop from p = start to end
13.  Arr[p] = temp[p]
```

## Example





### 2.9.3 Strassen's matrix multiplication.

suppose we want to multiply two matrices A and B each of size  $N \times N$  i.e.,

$$C = A \times B \quad \text{then}$$
$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

The multiplication gives

$$c_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$
$$c_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$
$$c_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$$
$$c_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

Thus to accomplish  $2 \times 2$  matrix multiplication there are total 8 multiplications and 4 additions.

To accomplish this multiplication we can write the following algorithm for the same.

Algorithm mat-mul(A, B, C, n)

```
for i := 0 to n do
  for j := 0 to n do
    c[i,j] = 0;
    for k := 0 to n do
      c[i,j] = c[i,j] + A[i,k] * B[k,j];
```

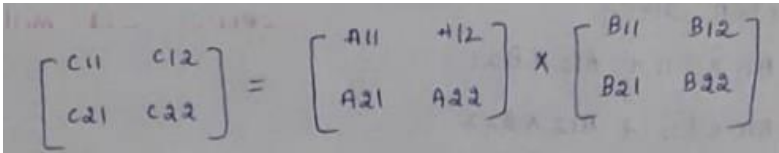
The time complexity of above algorithm turns to be  $O(N \times N \times N) = O(N^3)$

The divide and conquer approach can be used for implementing strassen's matrix multiplication.

**a) Divide:** Divide matrices into sub-matrices  $A_0, A_1, A_2$ ..etc.

**b) Conquer:** use a group of matrix multiply equations.

**c) Combine:** recursively multiply sub-matrices and get the final result of multiplication after performing required additions or subtractions.



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

$$P_1 = A_{11} \cdot (B_{12} - B_{22})$$

$$P_5 + P_4 - P_2 + P_6 = C_{11}$$

$$P_2 = (A_{11} + A_{12}) \cdot B_{22}$$

$$P_1 + P_2 = C_{12}$$

$$P_3 = (A_{21} + A_{22}) \cdot B_{11}$$

$$P_3 + P_4 = C_{21}$$

$$P_4 = A_{22} \cdot (B_{21} - B_{11})$$

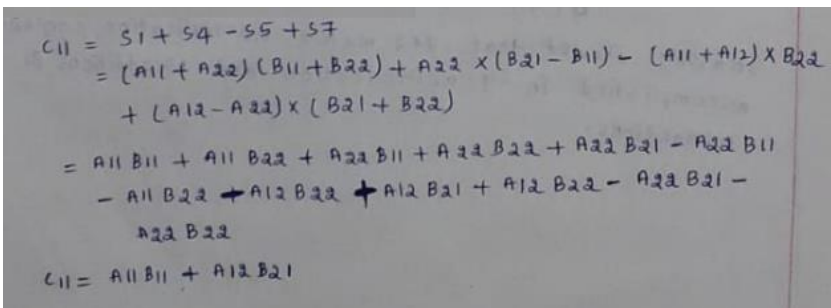
$$P_5 + P_1 - P_3 - P_7 = C_{22}$$

$$P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

$$P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

$$P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$$

Now we will compare the actual our traditional matrix multiplication procedure with strasses's procedure. In strasses's multiplication.

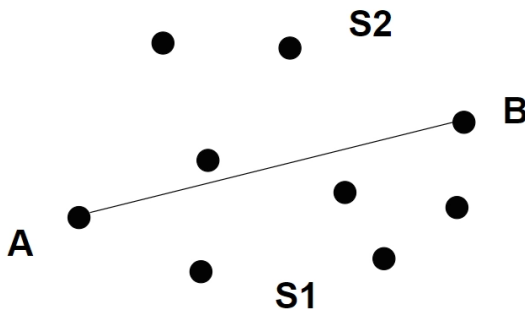


$$\begin{aligned} C_{11} &= S_1 + S_4 - S_5 + S_7 \\ &= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22} \times (B_{21} - B_{11}) - (A_{11} + A_{12}) \times B_{22} \\ &\quad + (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ &= A_{11} B_{11} + A_{11} B_{22} + A_{22} B_{11} + A_{22} B_{22} + A_{22} B_{21} - A_{22} B_{11} \\ &\quad - A_{11} B_{22} - A_{12} B_{22} + A_{12} B_{21} + A_{12} B_{22} - A_{22} B_{21} - \\ &\quad A_{22} B_{22} \\ C_{11} &= A_{11} B_{11} + A_{12} B_{21} \end{aligned}$$

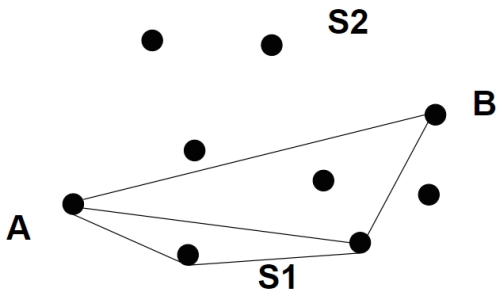
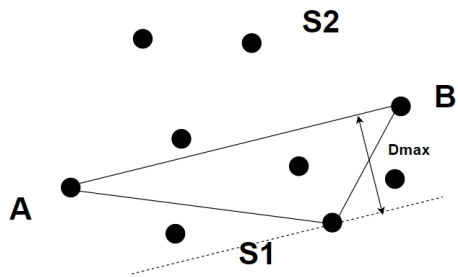
## 2.9.4 Convex Hull

A convex hull is the smallest convex polygon that completely encloses a set of points in a two-dimensional or three-dimensional space. It can be thought of as the "envelope" or "wrapper" of the points. We use the divide and conquer approach to solve this by recursively calling the function for smaller parameters. It is a fundamental concept with applications in various fields such as computer graphics, robotics, and image processing. Convex Hull Problem deals with the problem of finding convex polygon with the minimum number of edges. Here is an illustration of our approach:

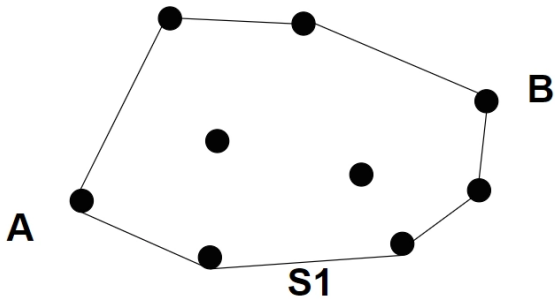
The first step is to find out the farthest two points in the plane:



Then, in the two spaces S1 and S2, we will find out the farthest point:



Finally, Our resultant polygon would look something like this:



## **Importance of Convex Hull:**

Convex hulls are important in computational geometry for several reasons:

- **Collision detection:** Convex hulls can be used to efficiently detect collisions between objects in 2D or 3D space.
- **Image processing:** Convex hulls can be used to extract meaningful shapes from images, such as the outline of an object.
- **Data visualization:** Convex hulls can be used to visualize the distribution of data points in a scatter plot.

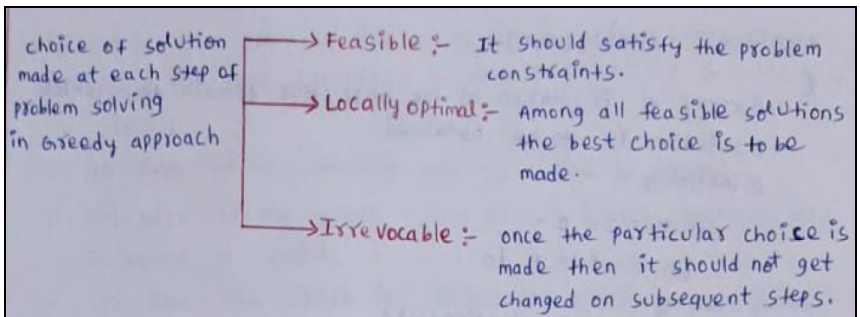
## UNIT-III

**Greedy Method:** General Method, Job Sequencing with deadlines, Knapsack Problem, Minimum cost spanning trees, Single Source Shortest Paths

**Dynamic Programming:** General Method, All pairs shortest paths, Single Source Shortest Paths– General Weights (Bellman Ford Algorithm), Optimal Binary Search Trees, 0/1 Knapsack, String Editing, Travelling Salesperson problem.

### 3.1 Greedy Method

In an algorithmic strategy like greedy, the decision is taken based on the information available. The greedy method is the most straight forward method. It is popular for obtaining the optimized solutions.



Example :- Now we have to find the maximum value for the following problem.

$$Z = 3x + 4y \quad \begin{matrix} 0 \leq x \leq 1 \\ -1 \leq y \leq 1 \end{matrix}$$

Sol :- Assume input set as  $(x, y)$

$(0, -1), (0, 0), (0, 1), (1, -1), (1, 0), (1, 1), (2, 2), (3, 3), (-4, -4)$

In greedy method, we will apply constraints (conditions) for input set then some of the inputs may be eliminated from the input set. The remaining inputs are known as feasible solution. Next we will take one input at a time and check, whether that input leads to optimal solution or not. If it leads to optimal solution, then add that input to optimal solution vector, otherwise discard the input.

After applying conditions to input set  $(2, 2), (3, 3), (-4, -4)$  pairs are removed from input set. Remaining pairs  $(0, -1), (0, 0), (0, 1), (1, -1), (1, 0), (1, 1)$  are called feasible solution.

Among these feasible solution at  $(1, 1)$  the objective function is maximum

$$\begin{aligned} \text{ie; } Z &= 3x + 4y \\ Z &= 3(1) + 4(1) \\ Z &= 7 \end{aligned}$$

So  $(1, 1)$  is an optimal solution for the given objective function.

Note :- The above example is given, in order to understand the Greedy method.

## 3.2 General Method

```

Algorithm greedy(a, n)
{
  // array a is taken as the objective domain from which
  // solution is to be obtained.
  // initially
  solution := 0;
  for i := 1 to n do
  {
    s := selection(a);
    if (feasible (solution, s)) then
    {
      solution := Add (solution, s);
    }
    else
      reject();
  }
  return solution;
}

```

- In the greedy method there are three important activities
  1. A selection of solution from the given input domain is performed.
  2. The feasibility of the solution is performed. And then all the feasible solutions are obtained.
  3. From the set of feasible solutions, the particular solution that minimize or maximizes the given objective function is obtained. Such a solution is called optimal solution.

For an algorithm that uses greedy method works in stages. At each stage only one input is considered at a time. Based on this input it is decided whether particular input gives the optimal solution or not.



### 3.3 Applications of Greedy Method

- 3.3.1 Job Sequencing with deadlines
- 3.3.2 Knapsack Problem
- 3.3.3 Minimum cost spanning trees
- 3.3.4 Single Source Shortest Paths

#### 3.3.1 Job Sequencing with deadlines

Consider that there are  $n$  jobs that are to be executed. At any time  $t=1,2,3,\dots$  only exactly one job is to be executed. The profits  $p_i$  are given. These profits are gained by corresponding jobs. For obtaining feasible solution we should take care that the jobs get completed within their given deadlines.

Let  $n=4$

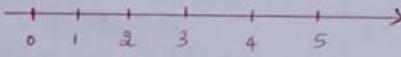
$n$	$p_i$	$d_i$
1	70	2
2	12	1
3	18	2
4	35	1

dead line is 2 so we will take only 2 combinations of values  $\leq 2$

We will follow following rules to obtain the feasible solution.

- Each Job takes one unit of time.
- If job starts before  $\theta$  at its deadline, profit is obtained otherwise no profit.
- Goal is to schedule Jobs to maximize the total profit.
- considers all possible schedules and compute the minimum total time in the system.

consider the Time line as



The feasible solutions are obtained by various permutations and combinations of Jobs.

$n$	$P_i$
1	70
2	12
3	18
4	35
1,3	88
2,1	82
2,3	30
3,1	88
4,1	105 ✓
4,3	53

Finally the feasible sequence is 4,1. This sequence is optimum solution as well.

### Example 2:

optimum solution is 4,1

Example 2: Solve the job sequencing problem given  $n=5$ , profits (1, 5, 20, 15, 10) and deadlines (1, 2, 4, 1, 3) using greedy strategy.

Sol: Since the maximum deadline is 4 units of time the feasible solution set must have  $\leq 4$  jobs.

Now arranging the jobs in the decreasing order of profits.

$(P_1, P_2, P_3, P_4, P_5) = (1, 5, 20, 15, 10)$

Decreasing order  $(P_3, P_4, P_5, P_2, P_1) = (20, 15, 10, 5, 1)$

Similarly  $(d_3, d_4, d_5, d_2, d_1) = (4, 1, 3, 2, 1)$

$\leq 4$

Feasible solution	processing sequence	profit
1	3	20
2	3,4	$20+15=35$
3	3,4,5	$20+15+10=45$
4	3,4,5,2	$20+15+10+5=50$ ✓
5	3,5,2,1	$1+5+10+20=36$

solution 4 is an optimal solution. The jobs must be processed in the order 4,2,5,3 or 4,2,3,5 and the value of the optimal solution is 50.

### 3.3.2 Knapsack Problem

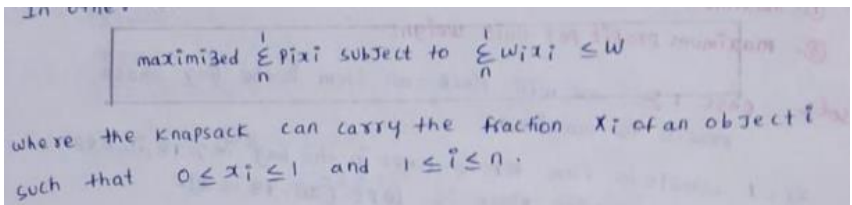
The knapsack problem can be stated as follows. Suppose there are  $n$  objects from  $i=1,2,\dots,n$ . each object  $i$  has some positive weight  $w_i$  and some profit value is associated with

each object which is denoted as  $p_i$ . This knapsack carries at the most weight  $w$ .

While solving above mentioned knapsack problem we have the capacity constraints. When we try to solve this problem using greedy approach our goal is

1. Choose only those objects that give maximum profit.
2. The total weight of selected objects should be  $\leq w$ .

And then we can obtain the set of feasible solutions. In other words,



The image shows a handwritten mathematical formulation of the knapsack problem. It is enclosed in a rectangular box. The text inside the box reads: "maximized  $\sum_{i=1}^n p_i x_i$  subject to  $\sum_{i=1}^n w_i x_i \leq w$ ". Below the box, there is a handwritten note: "where the knapsack can carry the fraction  $x_i$  of an object  $i$  such that  $0 \leq x_i \leq 1$  and  $1 \leq i \leq n$ ".

## Example

Example :- consider the following instance of Knapsack Problem.

$$n=3 \quad M=20 \quad (P_1, P_2, P_3) = (25, 24, 15) \\ (w_1, w_2, w_3) = (18, 15, 10). \text{ Find the optimal solution for}$$

- ①. Maximum profit
- ②. minimum weight
- ③. maximum profit per unit weight

Sol :- case 1 :- we will place an item in the bag whose profit is maximum.

$x_1 = 1$  complete item  $w_1$  is kept in the bag i.e., 18 is kept in bag, only 2 units profit is left ( $20 - 18 = 2$ ).

$$x_2 = \text{left out space / weight of item to be placed} \\ = 2/15$$

when  $w_2$  is placed no space is left in bag. so  $x_3 = 0$ .

$$\sum_{1 \leq i \leq n} P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3 \\ = 25 \times 1 + 24 \times \frac{2}{15} + 15 \times 0 \\ = 25 + 3.2 + 0 \\ = 28.2$$

Case 2 :- we will place an item in bag, whose weight is minimum.

$$x_3 = 1 \quad (20 - 10 = 10)$$

$$x_2 = 10/15 = 2/3$$

$$x_1 = 0$$

$$\sum P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3 \\ = 25 \times 0 + 24 \times \frac{2}{3} + 15 \times 1 \\ = 0 + 16 + 15 \\ = 31$$

Case 3:- we will place an item in the bag, whose profit per unit weight ratio is maximum.

$$\frac{P_1}{W_1} = \frac{25}{18} = 1.4$$

$$\frac{P_2}{W_2} = \frac{24}{15} = 1.6$$

$$\frac{P_3}{W_3} = \frac{15}{10} = 1.5$$

$P_2/W_2$  is maximum so  $x_2 = 1$ . Now remaining weight of bag is  $20 - 15 = 5$ . Next maximum is  $P_3/W_3$ . Now we have to place third item in bag but the space not sufficient.

so we have to place  $1/2$  (ie;  $5/10$ ) of third item.

$\therefore x_3 = 1/2$ . The bag is already filled, so

$$x_1 = 0$$

$$x_2 = 1$$

$$x_3 = \frac{5}{10} = 1/2 = 0.5$$

$$\sum P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$$

$$= 25 \times 0 + 24 \times 1 + 15 \times 0.5$$

$$= 0 + 24 + 7.5$$

$$= 31.5 \checkmark$$

## Example 2:

Example 2:- Find the optimal solution for given instance of knapsack problem.

$$n = 7 \quad M = 15$$

$$(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (10, 5, 15, 7, 6, 18, 3)$$

$$(W_1, W_2, W_3, W_4, W_5, W_6, W_7) = (2, 3, 5, 7, 1, 4, 1)$$

Find the optimal solution for

① maximum profit

② minimum weight

③ maximum profit per unit weight.

Sol: case 1:  $P(18, 15, 10, 7)$

$w(4, 5, 2, 7)$

$$x_1 = 1 \quad (15 - 4 = 11)$$

$$x_2 = 1 \quad (11 - 5 = 6)$$

$$x_3 = 1 \quad (6 - 2 = 4)$$

$x_4 =$  left out space/weight of item to be placed

$$x_4 = 4/7 \quad x_5 = 0 \quad x_6 = 0 \quad x_7 = 0$$

$$\begin{aligned} \sum p_i x_i &= p_1 x_1 + p_2 x_2 + p_3 x_3 + p_4 x_4 \\ &= 18 \times 1 + 15 \times 1 + 10 \times 1 + 7 \times 4/7 \\ &= 18 + 15 + 10 + 4 \\ &= 47 \end{aligned}$$

case 2:  $(1, 1, 2, 3, 4, 4/5)$

$(10, 5, 15, 7, 6, 18, 3)$   
 $(2, 3, 5, 7, 1, 4, 1)$

$$15 - 1 = 14 \quad x_7 = 1$$

$$14 - 1 = 13 \quad x_8 = 1 \quad x_4 = 0$$

$$13 - 2 = 11 \quad x_1 = 1$$

$$11 - 3 = 8 \quad x_2 = 1$$

$$8 - 4 = 4 \quad x_6 = 1$$

$$4 - 5 = 4/5 \quad x_3 = 4/5$$

10	5	15	7	6	18	3
2	3	5	7	1	4	1
1	4/5	1	0	1	1	1

$$\begin{aligned} \sum p_i x_i &= 1 \times 10 + 5 \times 1 + 15 \times \frac{4}{5} + 7 \times 0 + 1 \times 6 + 1 \times 18 + 1 \times 3 \\ &= 10 + 5 + 12 + 0 + 6 + 18 + 3 \\ &= 54 \end{aligned}$$

$x_5 = 0$   
 $x_5 = 1$

Case 3 :-  $\frac{p_1}{w_1} = \frac{10}{2} = 5$   $\frac{p_3}{w_3} = \frac{15}{5} = 3$   $\frac{p_5}{w_5} = \frac{6}{1} = 6$   
 $\frac{p_2}{w_2} = \frac{5}{3} = 1.6$   $\frac{p_4}{w_4} = \frac{7}{7} = 1$   $\frac{p_6}{w_6} = \frac{18}{4} = 4.5$   
 $\frac{p_7}{w_7} = \frac{3}{1} = 3$

$E_{\text{profit}} = p_5x_5 + p_1x_1 + p_6x_6 + p_3x_3 + p_7x_7 + p_2x_2$   
 $= 6 \times 1 + 10 \times 1 + 18 \times 1 + 15 \times 1 + 3 \times 1 + 5 \times \frac{2}{3}$   
 $= 6 + 10 + 18 + 15 + 3 + 3.3$   
 $= 55.3$

A greedy strategy is used to find the optimum solution. In general, we arrange the objects based on  $p/w$  values in decreasing order and then select the objects one by one and add the object into the knapsack if it fits.

Algorithm for Knapsack Problem

```

Algorithm Knapsack(w, n)
{
  // p[i] contains the profits of i items such that 1 ≤ i ≤ n
  // w[i] contains weights of i items
  // x[i] is the solution vector.
  // w is the total size of Knapsack.
  for i := 1 to n do
  {
    if (w[i] < w)
    {
      x[i] := 1.0;
      w = w - w[i];
    }
  }
  if (i < n) then
  x[i] := w/w[i];
}

```



### 3.3.3 Minimum Cost Spanning Tree

A spanning tree of a graph is any tree that includes every vertex in the graph.

**Definition :-** A spanning tree of a graph  $G$  is a subgraph of  $G$ , that is a tree and contains all the vertices of  $G$  containing no circuit

- An edge of a spanning tree is called a branch.
- An edge in the graph that is not in the spanning tree is called a chord.
- It spans the graph i.e.; it includes every vertex of the graph.
- It is a minimum, i.e., the total weight of all the edges is as low as possible.

**Example :-**

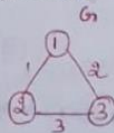
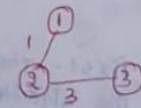
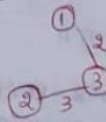


Fig:- weighted Graph

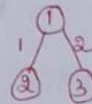
If the graph consists of  $n$  vertices then the possible spanning trees are  $n^{n-2}$ . For above example  $n=3$  i.e.,  $3^{3-2} = 3^1 = 3$  spanning trees. These spanning trees can be shown in fig:-



cost =  $1+3=4$   
(a)



cost =  $2+3=5$   
(b)



cost =  $1+2=3$   
(c)

Fig:- spanning trees

among these spanning trees (c) has minimum cost, so it is called a minimum spanning tree. For this graph minimum spanning tree is



fig:- minimum cost spanning trees

A tree which includes all vertices of  $G$  with minimum cost is called minimum spanning tree.

spanning trees are important because of following reasons

1. spanning trees are very important in designing efficient routing algorithms.
2. spanning trees have wide applications in many areas, such as network design.

The time complexity of this algorithm is clearly  $O(n)$ , where  $n$  is the number of edges in the graph.

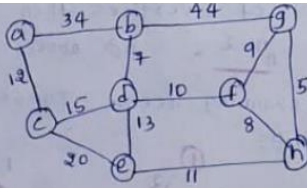
There are two important algorithms to obtain minimum spanning tree (MST) and those are -

1. Prim's Algorithm
2. Kruskal's Algorithm

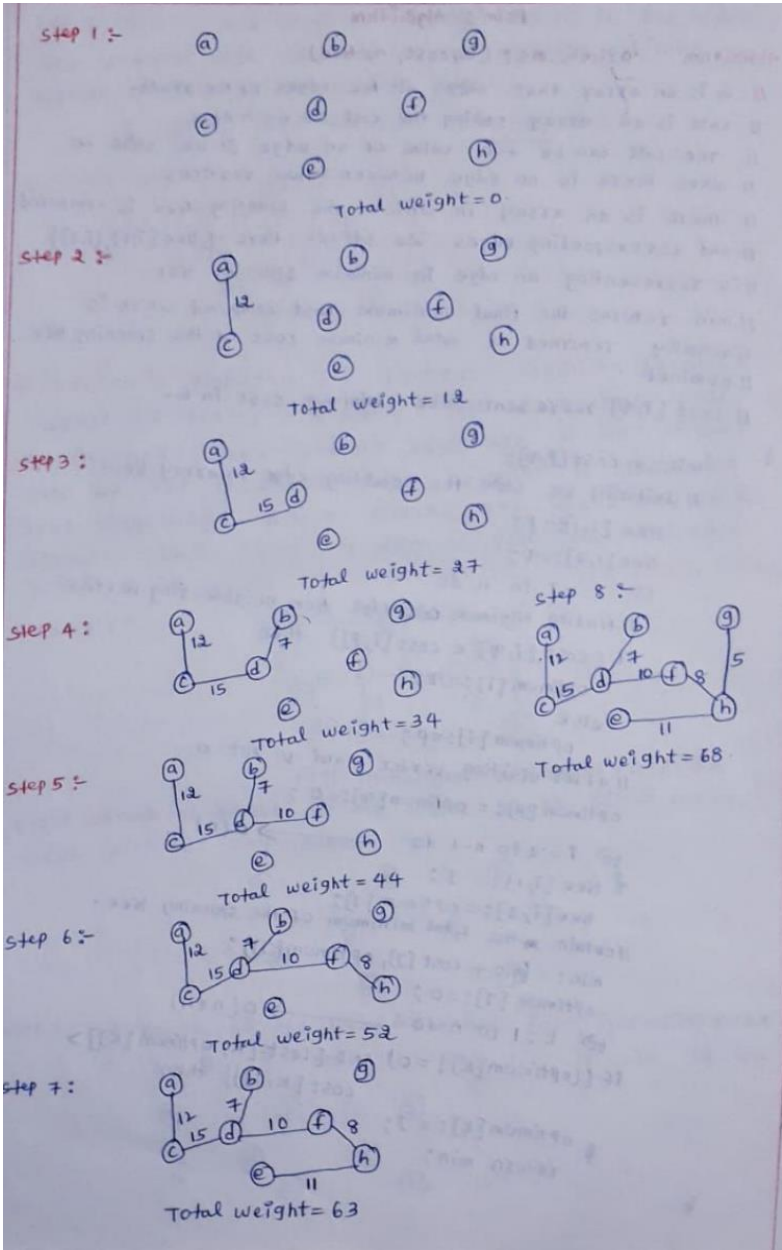
## 1. Prim's algorithm

Let us understand the prim's algorithm with the help of some example.

Example :-



Now we will consider all the vertices first. Then we will select an edge with minimum weight. The algorithm proceeds by selecting adjacent edges with minimum weight. care should be taken for not forming circuit.



## 2. Kruskal's Algorithm

2. **Kruskal's Algorithm** :- Kruskal's algorithm creates a forest of trees, initially the forest consists of  $n$  single node trees and no edges are shown. At each step we add cheapest edge so that it joins two trees together. Always checks are there any cycle formed. If it were to form a cycle then the edge is not to be added.

**Example :-**

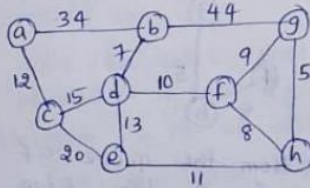
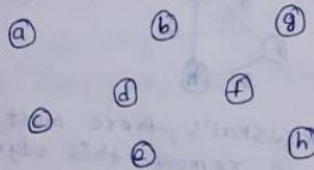


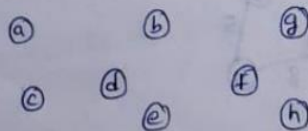
Fig:- Initial graph of Kruskal's algorithm.

First create a forest for the above graph. Forest means nodes without any edges.

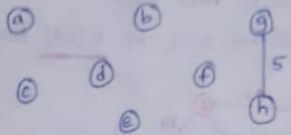


Step 1:- Arrange all the weighted values into increasing order

5 7 8 9 10 11 12 13 15 20 34 44

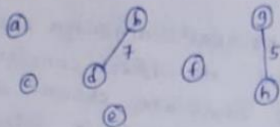


STEP 2:- pick the smallest weight from the sorted weights that is 5, draw edge between g and h.



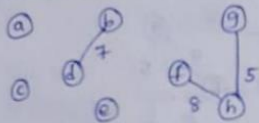
STEP 3:- Remove 5 from the queue of sorted weights. Next minimum is 7, so draw edge between b and d.

7	8	9	10	11	12	13	15	20	34	44
---	---	---	----	----	----	----	----	----	----	----



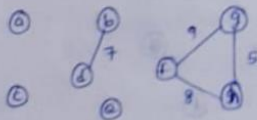
STEP 4:- Remove 7 from the queue of sorted weights. Next minimum is 8, draw edge between h and f.

8	9	10	11	12	13	15	20	34	44
---	---	----	----	----	----	----	----	----	----



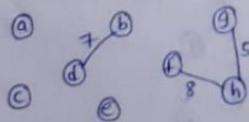
STEP 5:- Remove 8 from the queue of sorted weights. Next minimum is 9, draw edge between g and f.

9	10	11	12	13	15	20	34	44
---	----	----	----	----	----	----	----	----



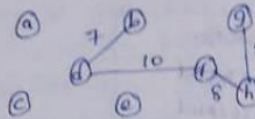
STEP 6:- According to Kruskal's, there must be no cycle in the minimum spanning tree, so remove this edge from the above diagram. Remove 9 from the queue.

10	11	12	13	15	20	34	44
----	----	----	----	----	----	----	----



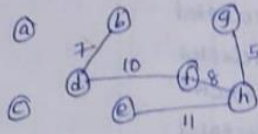
Step 7: Next minimum value is 10, so draw edge between d and f.

10 11 12 13 15 20 34 44



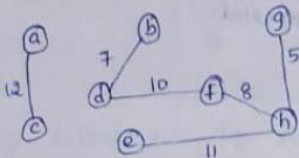
Step 8: Remove 10 from the queue. Next minimum value is 11 so draw edge between e and h

11 12 13 15 20 34 44



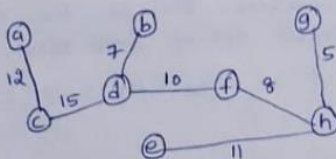
Step 9: Remove 11 from the queue. Next minimum value is 12 so draw edge between a and c.

12 13 15 20 34 44



Step 10: Remove 12 from the queue. Next minimum value is 13, so draw edge between e and d (cycle) so Remove the edge, next minimum is 15 so draw edge between c and d.

15 20 34 44



All the vertices are connected, Hence completed Kruskal's minimum spanning tree.

actions of kruskal's algorithms are shown in the following table.

Edge	weight	Action
(a, b)	34	Rejected
(a, c)	12	Accepted
(b, g)	44	Rejected
(c, d)	15	Accepted
(c, e)	20	Rejected
(d, b)	7	Accepted
(d, f)	10	Accepted
(e, d)	13	Rejected
(e, h)	11	Accepted
(f, g)	9	Rejected
(f, h)	8	Accepted
(h, g)	5	Accepted.



### 3.3.4 Single source shortest path problem

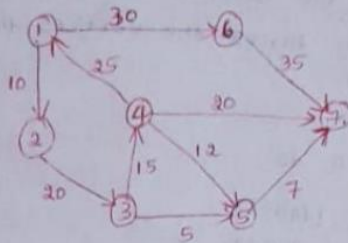
Many times, graph is used to represent the distance between two cities. Everybody is often interested in moving from one city to other as quickly as possible. The single source shortest path is based on this interest.

In a single source shortest path problem the shortest distance from a single vertex called source is obtained.

Let  $G=(V,E)$  be a graph, then in single source shortest path the shortest paths from vertex  $V_0$  to all remaining vertex is determined. The vertex  $V_0$  is then called as source and the last vertex is called destination.

- It is assumed that all the distance are +ve.

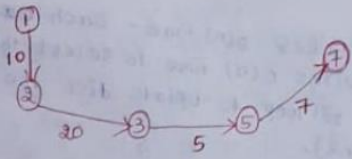
Example 1:- Consider <sup>a)</sup> graph  $G$  as given below.



sol:-

	1	2	3	4	5	6	7
1	$\infty$	(10)	$\infty$	$\infty$	$\infty$	30	$\infty$
2	$\infty$	10	(30)	$\infty$	$\infty$	30	$\infty$
3	$\infty$	10	30	45	35	(30)	$\infty$
6	$\infty$	10	30	45	(35)	30	65
5	$\infty$	10	30	45	35	30	(42) ✓
7	$\infty$	10	30	(45)	35	30	42
4	70	10	30	45	<del>35</del>	30	<del>42</del> 42

1, 2, 3, 5, 7 = (42) ✓



EX-2

### 3.4 Difference between Divide and Conquer and Greedy Method

Divide - and - conquer	Greedy Approach
<ol style="list-style-type: none"><li>1. Divide - and - conquer approach is the result oriented approach.</li><li>2. The time taken by this algorithm is efficient when compared to greedy approach.</li><li>3. This approach does not depend on constraints to solve a specific problem.</li><li>4. This approach is not efficient for larger problems.</li></ol>	<ol style="list-style-type: none"><li>1. By Greedy method, there are some chances of getting an optimal solution to a specific problem.</li><li>2. The time taken by this algorithm is not that much efficient when compared to divide - and - conquer approach.</li><li>3. This approach cannot make further move, if the subset chosen does not satisfy the specified constraints.</li><li>4. This approach is applicable and as well as efficient for a wide variety of problems.</li></ol>
<ol style="list-style-type: none"><li>5. As the problem is divided into a large number of subproblems, the space requirement is very much high.</li><li>6. This approach is not applicable to problems which are not divisible.</li></ol> <p>EX:- KNAPSACK problem.</p>	<ol style="list-style-type: none"><li>5. Space requirement is less when compared to the divide - and - conquer approach.</li><li>6. This problem is rectified in the greedy method.</li></ol>

### **3.5 Dynamic Programming:**

#### **Introduction to Dynamic programming method**

The drawback of greedy method is, we will make one decision at a time. This can be overcome in dynamic programming. In this, we will make more than one decision at a time.

In order to obtain the optimal solution to the given problem all possible decision sequences are being generated. And using principle of optimality the optimal sequence becomes the final solution.

#### **General Method**

Dynamic programming is typically applied to optimization problems. For a given problem we may get any number of solutions. From all those solutions we seek for the optimum solution. And such an optimal solution becomes the solution to the given problem.(Minimum value or maximum value solution).

## Difference between Divide and Conquer and Dynamic Programming

<b>Divide and conquer</b>	<b>Dynamic Programming</b>
The problem is divided into small sub-problems. These sub-problems are solved independently. Finally all the solutions of sub-problems are collected together to get the solution to the given problem.	In dynamic programming, we will make more than one decision at a time, in-order to obtain the optimal solution.
In this duplicate solutions may be obtained.	In this duplicate solutions are avoided totally.
Divide and conquer is less efficient than dynamic programming.	It is efficient than divide and conquer.
The divide and conquer uses top down approach of problem solving(recursive method)	It uses bottom up approach of problem solving(Iterative method)
Divide and conquer splits its	It splits its input at every

input in the middle.	possible points, rather than at a particular point. After it determines which split point is optimal.
----------------------	--

## Difference between Greedy programming and Dynamic programming

Greedy method	Dynamic programming
1. It is used to obtaining optimum solution.	1. It is also used for obtaining optimum solution.
2. In Greedy method a set of feasible solutions pick up the optimum solution.	2. There is no special set of feasible solutions in this method.
3. In Greedy method the optimum selection is without revising previously generated solutions.	3. Dynamic programming considers all possible sequences in order to obtain the optimum solution.
4. In this method there is no as such guarantee of getting optimum solution.	4. It is guarantee of getting optimum solution. <i>→ bottom up approach</i>

## Principle of optimality

It states that “in optimal sequence of decisions or choices, each subsequence must also be optimal”.

When it is not possible to apply the principle of optimality it is almost impossible to obtain the solution using dynamic programming approach.

### **3.6 Applications of Dynamic programming**

3.6.1 All pairs shortest paths

3.6.2 Single source shortest paths

3.6.3 Optimal Binary Search Tree(OBST)

3.6.4 0/1 Knapsack problem

3.6.5 Travelling sales person problem.

3.6.6 String Editing

#### **3.6.1 All pairs shortest paths**

When a weighted graph, represented by its weight matrix  $W$  then objective is to find the distance between every pair of nodes. We will apply dynamic programming to solve the all pairs shortest path.

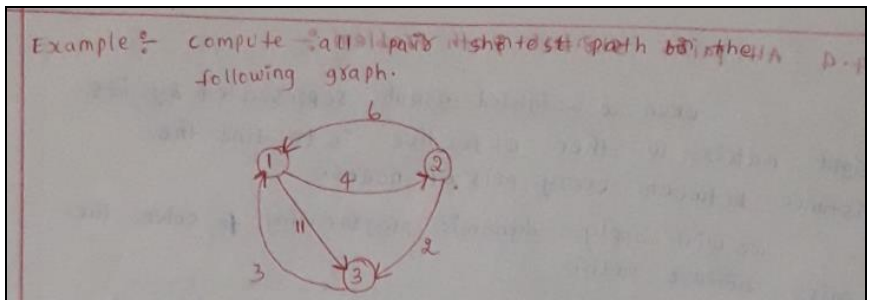
Step 1:- we will decompose the given problem into sub problem.  
 Let  $A^k_{(i,j)}$  be the length of shortest path from node  $i$  to node  $j$  such that the label on every intermediate node will be  $\leq k$ .  
 we will compute  $A^k$  for  $k=1 \dots n$  for  $n$  nodes.

Step 2:- For solving all pairs shortest path, the principle of optimality is used.

Step 3:- Initially

1.  $A^k_{(i,j)} = W_{(i,j)}$  if  $k=0$ .
2. Next computations:
 
$$A^k_{(i,j)} = \min \left\{ A^{k-1}_{(i,j)}, A^{k-1}_{(i,k)} + A^{k-1}_{(k,j)} \right\} \quad \text{where } 1 \leq k \leq n$$

## Example



**Sol:**



Sol: cost of adjacency matrix

$$A^0_{(i,j)} = w_{(i,j)}$$
$$A^0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{matrix}$$

Step 1: For  $k=1$   $i=1$   $j=1$

$$A^1_{(1,1)} = \min \{ A^0_{(1,1)}, A^0_{(1,1)} + A^0_{(1,1)} \}$$
$$= \min \{ 0, 0+0 \}$$

$$A^1_{(1,1)} = 0$$

$k=1$   $i=1$   $j=2$

$$A^1_{(1,2)} = \min \{ A^0_{(1,2)}, A^0_{(1,1)} + A^0_{(1,2)} \}$$
$$= \min \{ 4, 0+4 \}$$

$$A^1_{(1,2)} = 4$$

$k=1$   $i=1$   $j=3$

$$A^1_{(1,3)} = \min \{ A^0_{(1,3)}, A^0_{(1,1)} + A^0_{(1,3)} \}$$
$$= \min \{ 11, 0+11 \}$$

$$A^1_{(1,3)} = 11$$

$$i=2 \quad j=1 \quad k=1$$

$$A^1(2,1) = \min \{ A^0(2,1), A^0(2,1) + A^0(1,1) \}$$
$$= \min \{ 6, 6+0 \}$$

$$A^1(2,1) = 6$$

$$i=2 \quad j=2 \quad k=1$$

$$A^1(2,2) = \min \{ A^0(2,2), A^0(2,1) + A^0(1,2) \}$$
$$= \min \{ 0, 6+4 \}$$
$$= 0$$

$$i=2 \quad j=3 \quad k=1$$

$$A^1(2,3) = \min \{ A^0(2,3), A^0(2,1) + A^0(1,3) \}$$
$$= \min \{ 2, 6+11 \}$$

$$A^1(2,3) = 2$$

$$i=3 \quad j=1 \quad k=1$$

$$A^1(3,1) = \min \{ A^0(3,1), A^0(3,1) + A^0(1,1) \}$$
$$= \min \{ 3, 3+0 \}$$

$$A^1(3,1) = 3$$

$$i=3 \quad j=2 \quad k=1$$

$$A^1(3,2) = \min \{ A^0(3,2), A^0(3,1) + A^0(1,2) \}$$

$$= \min \{ \infty, 3+4 \}$$

$$= 7$$

~~$$A^1(3,2) = 7$$~~

$$i=3 \quad j=3 \quad k=1$$

$$A^1(3,3) = \min \{ A^0(3,3), A^0(3,1) + A^0(1,3) \}$$

$$= \min \{ 0, 3+11 \}$$

$$A^1(3,3) = 0$$

$$\therefore A^1(1,1) = \begin{matrix} & \begin{matrix} 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{matrix}$$

step 2 :- For  $k=2$  i.e. going from  $i$  to  $j$  through vertex 2.

$$i=1 \quad j=1 \quad k=2$$

$$A^2(1,1) = \min \{ A^1(1,1), A^1(1,2) + A^1(2,1) \}$$

$$= \min \{ 0, 4+6 \}$$

$$A^2(1,1) = 0$$

$$\begin{aligned}
 & i=1 \quad j=2 \quad k=2 \\
 A^2(1,2) &= \min \{ A^1(1,2), A^1(1,2) + A^1(2,2) \} \\
 &= \min \{ 4, 4+0 \} \\
 A^2(1,2) &= 4 \\
 & i=1 \quad j=3 \quad k=2 \\
 A^2(1,3) &= \min \{ A^1(1,3), A^1(1,2) + A^1(2,3) \} \\
 &= \min \{ 11, 4+2 \} \\
 A^2(1,3) &= 6 \\
 & i=2 \quad j=1 \quad k=2 \\
 A^2(2,1) &= \min \{ A^1(2,1), A^1(2,2) + A^1(2,1) \} \\
 &= \min \{ 6, 0+6 \} \\
 A^2(2,1) &= 6 \\
 & i=2 \quad j=2 \quad k=2 \\
 A^2(2,2) &= \min \{ A^1(2,2), A^1(2,2) + A^1(2,2) \} \\
 &= \min \{ 0, 0+0 \} \\
 A^2(2,2) &= 0 \\
 & i=2 \quad j=3 \quad k=2 \\
 A^2(2,3) &= \min \{ A^1(2,3), A^1(2,2) + A^1(2,3) \} \\
 &= \min \{ 2, 0+2 \} \\
 &= 2 \\
 & i=3 \quad j=1 \quad k=2 \qquad i=3 \quad j=2 \quad k=2 \qquad i=3 \quad j=3 \quad k=2 \\
 A^2(3,1) &= 3 \qquad A^2(3,2) = 7 \qquad A^2(3,3) = 0 \\
 \therefore A^2(i,j) &= \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}
 \end{aligned}$$

Step 3 :- For  $k=3$  i.e., going from  $i$  to  $j$  through vertex 3.

$$\begin{aligned}
 A^3(1,1) &= 0 & A^3(1,2) &= 4 & A^3(1,3) &= 6 \\
 A^3(2,1) &= 5 & A^3(2,2) &= 0 & A^3(2,3) &= 2 \\
 A^3(3,1) &= 3 & A^3(3,2) &= 7 & A^3(3,3) &= 0
 \end{aligned}$$

$$\therefore A^3(i,j) = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

This matrix gives the all pairs shortest path solution.

Time complexity of the algorithm :-

Algorithm All pairs shortest path [W, A, n]

// W is weighted array matrix, n is the number of vertices

// A is the cost of shortest path from vertex i to j

```
{ for i := 1 to n do } O(n^2)
```

```
  for j := 1 to n do
```

```
    A[i, j] := W[i, j]
```

```
  for k := 1 to n do
```

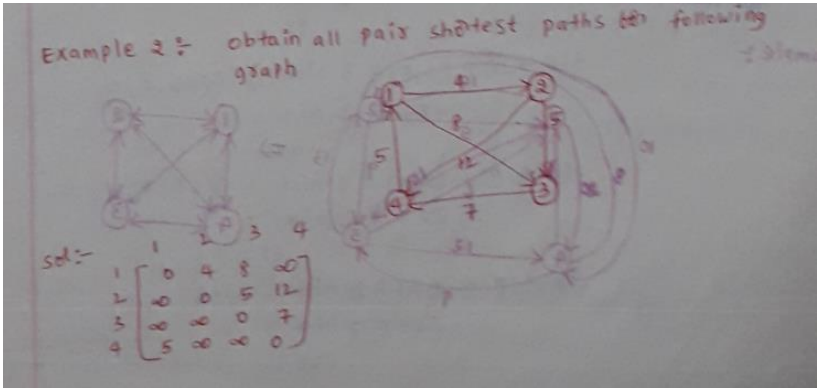
```
    for i := 1 to n do
```

```
      for j := 1 to n do
```

```
        A[i, j] = min{ A[i, j], A[i, k] + A[k, j] }
```

Time complexity for this method is  $O(n^3)$

## Example 2:



### 3.6.2 Single source shortest paths

The single source shortest path algorithm (for arbitrary weight positive or negative) is also known Bellman-Ford algorithm is used to find minimum distance from source vertex to any other vertex.

The main difference between this algorithm with Dijkstra's algorithm is, in Dijkstra's algorithm we cannot handle the negative weight, but here we can handle it easily.

#### Bellman-ford algorithm Steps

1. Initialize all distance values as  $\infty$  except source (0)

2. Repeat (V-1) times:

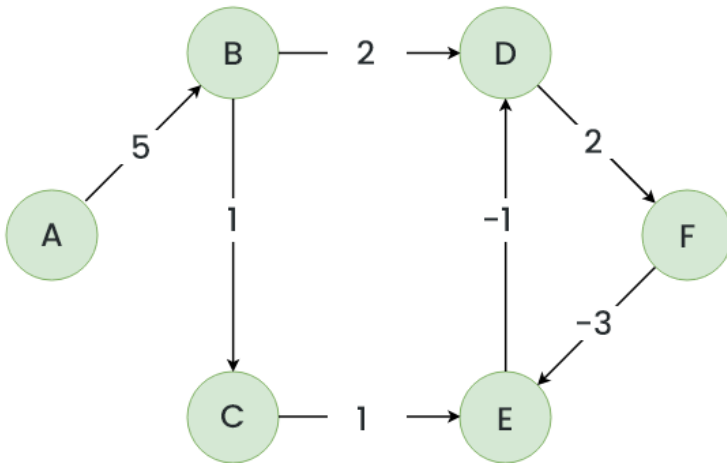
if  $d[u] + \text{cost}(uv) < d[v]$  then update  $d[v]$

Else skip

3. Relax all vertices once more. If you find any new shortest distance value then we have  $-ve$  edge with cycle else we don't.

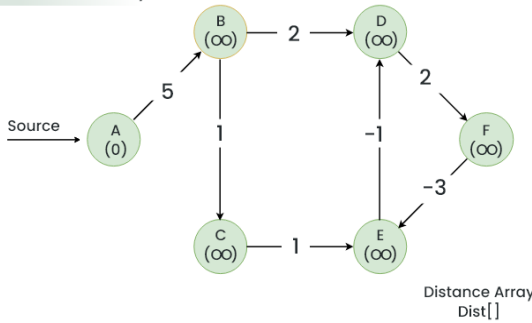
### **Working of Bellman-Ford Algorithm to Detect the Negative cycle in the graph:**

Let's suppose we have a graph which is given below and we want to find whether there exists a negative cycle or not using Bellman-Ford.



Step 1: Initialize a distance array  $Dist[]$  to store the shortest distance for each vertex from the source vertex. Initially distance of source will be 0 and Distance of other vertices will be INFINITY.

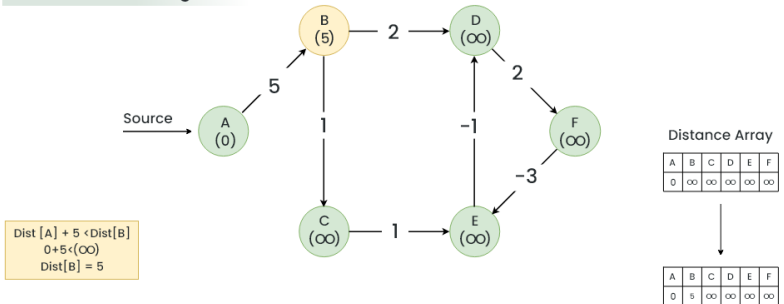
### Initialize The Distance Array



### Step 2: Start relaxing the edges, during 1st Relaxation:

- Current Distance of B > (Distance of A) + (Weight of A to B) i.e. Infinity > 0 + 5
- Therefore, Dist[B] = 5

### 1st Relaxation Of Edges



$$\begin{aligned} \text{Dist}[A] + 5 &< \text{Dist}[B] \\ 0 + 5 &< (\infty) \\ \text{Dist}[B] &= 5 \end{aligned}$$



Step 3: During 2nd Relaxation:

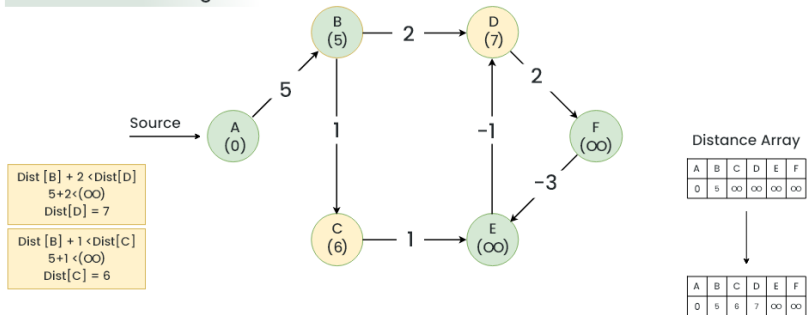
Current Distance of D > (Distance of B) + (Weight of B to D) i.e. Infinity > 5 + 2

$$\text{Dist}[D] = 7$$

Current Distance of C > (Distance of B) + (Weight of B to C) i.e. Infinity > 5 + 1

$$\text{Dist}[C] = 6$$

2nd Relaxation Of Edges



Step 4: During 3rd Relaxation:

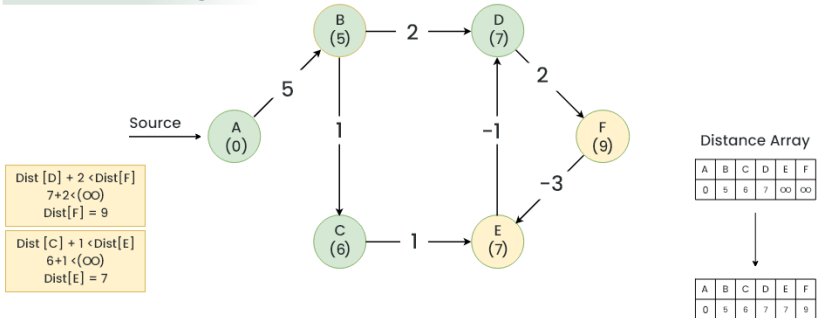
Current Distance of F > (Distance of D) + (Weight of D to F) i.e. Infinity > 7 + 2

$$\text{Dist}[F] = 9$$

Current Distance of E > (Distance of C) + (Weight of C to E) i.e. Infinity > 6 + 1

$$\text{Dist}[E] = 7$$

### 3rd Relaxation Of Edges



Step 5: During 4th Relaxation:

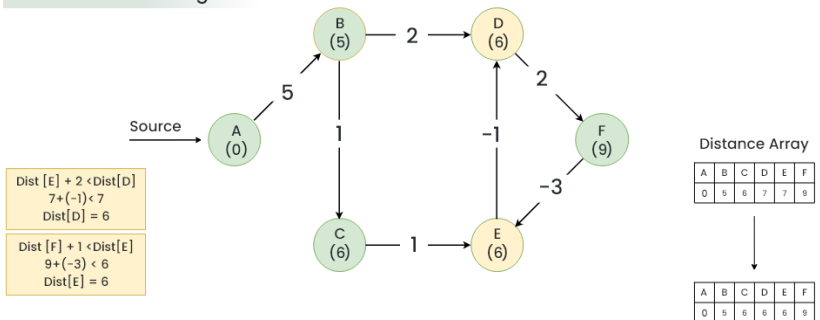
Current Distance of D > (Distance of E) + (Weight of E to D) i.e.  $7 > 7 + (-1)$

$$\text{Dist}[D] = 6$$

Current Distance of E > (Distance of F) + (Weight of F to E) i.e.  $7 > 9 + (-3)$

$$\text{Dist}[E] = 6$$

### 4th Relaxation Of Edges



Step 6: During 5th Relaxation:

Current Distance of F > (Distance of D) + (Weight of D to F) i.e.  $9 > 6 + 2$

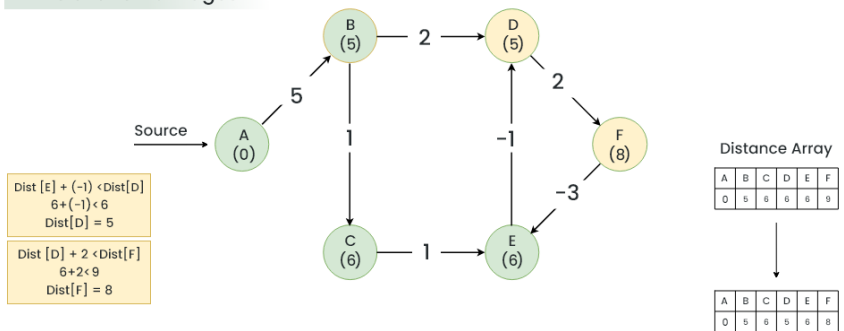
$\text{Dist}[F] = 8$

Current Distance of D > (Distance of E) + (Weight of E to D) i.e.  $6 > 6 + (-1)$

$\text{Dist}[D] = 5$

Since the graph has 6 vertices, So during the 5th relaxation the shortest distance for all the vertices should have been calculated.

5th Relaxation Of Edges



Step 7: Now the final relaxation i.e. the 6th relaxation should indicate the presence of negative cycle if there is any changes in the distance array of 5th relaxation.

During the 6th relaxation, following changes can be seen:

Current Distance of E > (Distance of F) + (Weight of F to E)

i.e.  $6 > 8 + (-3)$

$\text{Dist}[E]=5$

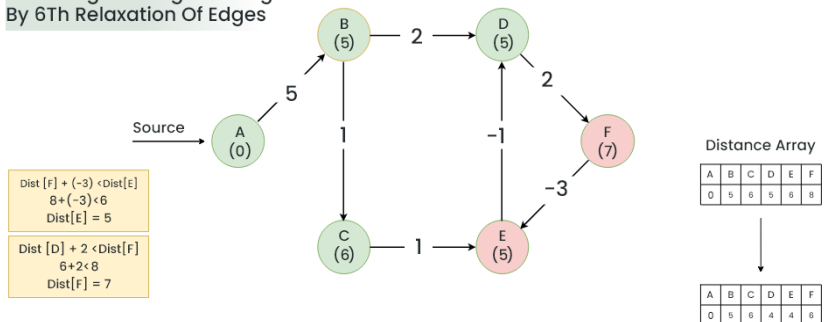
Current Distance of F > (Distance of D ) + (Weight of D to F)

i.e.  $8 > 5 + 2$

$\text{Dist}[F]=7$

Since, we observe changes in the Distance array Hence, we can conclude the presence of a negative cycle in the graph.

Detecting The Negative Edge By 6Th Relaxation Of Edges



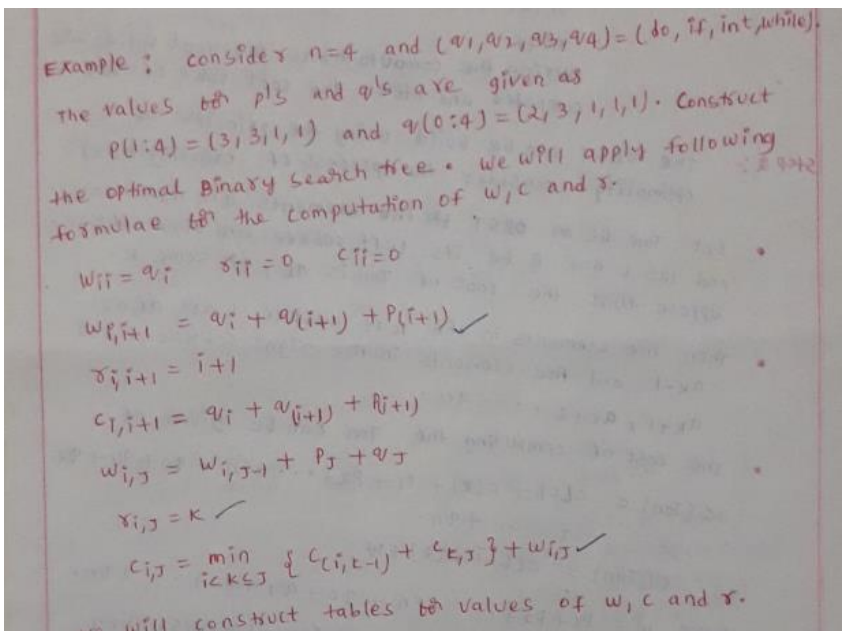
**Result:** A negative cycle (D->F->E) exists in the graph.

### 3.6.3 Optimal Binary Search Tree (OBST)

Suppose we are searching a word from a dictionary, and for every required word, we are looking up in the dictionary then it becomes time consuming process.

To perform this lookup more efficiently we can build the binary search tree of common words as key elements. Again we can make this binary search tree efficiently by arranging frequently words nearer to the root and less frequently words away from the root. Such a binary search tree makes our task more simplified as well as efficient. This type of binary search tree is called optimal binary search tree(OBST).

### Example



	$j$				
	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{01} = 3$ $c_{01} = 0$ $r_{01} = 0$	$w_{02} = 1$ $c_{02} = 0$ $r_{02} = 0$	$w_{03} = 1$ $c_{03} = 0$ $r_{03} = 0$	$w_{04} = 1$ $c_{04} = 0$ $r_{04} = 0$
1	$w_{10} = 8$ $c_{10} = 8$ $r_{10} = 1$	$w_{11} = 7$ $c_{11} = 7$ $r_{11} = 2$	$w_{12} = 3$ $c_{12} = 3$ $r_{12} = 3$	$w_{13} = 3$ $c_{13} = 3$ $r_{13} = 4$	
2	$w_{20} = 12$ $c_{20} = 19$ $r_{20} = 1$	$w_{21} = 9$ $c_{21} = 12$ $r_{21} = 2$	$w_{22} = 6$ $c_{22} = 8$ $r_{22} = 3$		
3	$w_{30} = 14$ $c_{30} = 21$ $r_{30} = 2$	$w_{31} = 11$ $c_{31} = 19$ $r_{31} = 2$			
4	$w_{40} = 16$ $c_{40} = 35$ $r_{40} = 2$				

$i \downarrow$

$j-i$

Let

$\frac{i=0}{w_{ii} = q_i}$	$\frac{i=1}{w_{ii} = q_1}$	$\frac{i=2}{w_{ii} = q_2}$	$\frac{i=3}{w_{ii} = q_3}$	$\frac{i=4}{w_{ii} = q_4}$
$w_{00} = q_0$	$w_{11} = 3$	$w_{22} = 1$	$w_{33} = 1$	$w_{44} = 1$
$w_{00} = 2$				

$\frac{i=0}{c_{ii} = 0}$	$\frac{i=1}{c_{ii} = 0}$	$\frac{i=2}{c_{ii} = 0}$	$\frac{i=3}{c_{ii} = 0}$	$\frac{i=4}{c_{ii} = 0}$
$c_{00} = 0$				

$\frac{i=0}{\gamma_{00} = 0}$	$\frac{i=1}{\gamma_{11} = 0}$	$\frac{i=2}{\gamma_{22} = 0}$	$\frac{i=3}{\gamma_{33} = 0}$	$\frac{i=4}{\gamma_{44} = 0}$
-------------------------------	-------------------------------	-------------------------------	-------------------------------	-------------------------------

$i=0 \quad j-i=1 \quad k=1$

$$w_{(i,i+1)} = q_i + q_{(i+1)} + p_{(i+1)}$$

$$w_{(0,1)} = q_0 + q_1 + p_1$$

$$= 2 + 3 + 3$$

$$= 8$$

$$c_{(i,j)} = \min \{ c_{(i,k-1)} + c_{(k,j)} \} + (w_{(i,j)})$$

$$= \min \{ c_{(0,0)} + c_{(1,1)} \} + w_{(0,1)}$$

$$= \min \{ 0 + 0 \} + 8$$

$$c_{(0,1)} = 8$$

$$\gamma_{(i,j)} = k$$

$$\gamma_{(0,1)} = 1$$
  

$i=1 \quad j-i=2 \quad k=2$

$$w_{(i,i+1)} = q_i + q_{(i+1)} + p_{(i+1)}$$

$$w_{(1,2)} = q_1 + q_2 + p_2$$

$$= 3 + 1 + 3$$

$$w_{(1,2)} = 7$$

$$c_{(i,j)} = w_{(i,j)} + \min \{ c_{(i,k-1)} + c_{(k,j)} \}$$

$$c_{(1,2)} = w_{(1,2)} + \min \{ c_{(1,1)} + c_{(2,2)} \}$$

$$c_{(1,2)} = 7 + \min \{ 0 + 0 \}$$

$$c_{(1,2)} = 7$$

$$c_{(1,2)} = 7$$

$$\gamma_{ij} = k$$

$$\gamma_{12} = 2$$

$i < k \leq j$

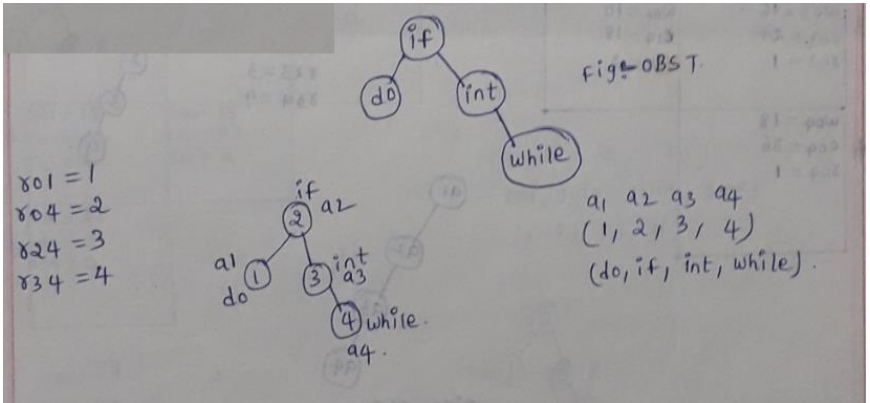
<p><math>i=2 \quad j-i=3 \quad k=3</math></p> $w_{(2,3)} = q_2 + q_3 + p_3$ $= 1 + 1 + 1$ $w_{(2,3)} = 3$ $c_{(2,3)} = w_{(2,3)} + \min \{ c_{(2,2)} + c_{(3,3)} \}$ $= 3 + \min \{ 0 + 0 \}$ $c_{(2,3)} = 3$ $\gamma_{ij} = k$ $\gamma_{23} = 3$	<p><math>i=3 \quad j-i=4 \quad k=4</math></p> $w_{(3,4)} = q_3 + q_4 + p_4$ $= 1 + 1 + 1$ $w_{(3,4)} = 3$ $c_{(3,4)} = w_{(3,4)} + \min \{ c_{(3,3)} + c_{(4,4)} \}$ $c_{(3,4)} = 3 + \min \{ 0 + 0 \}$ $c_{(3,4)} = 3$ $\gamma_{ij} = k$ $\gamma_{34} = 4$
---	---

$i=0 \quad j-i=2 \quad k=1$ $w(i,j) = w(i,j-1) + p_j + q_j$ $w(0,2) = w(0,1) + p_2 + q_2$ $= 8 + 3 + 1$ $= 12$ $c(0,2) = w(0,2) + \min\{c(0,0) + c(1,2)\}$ $= 12 + \min\{0 + 7\}$ $= 12 + 7$ $= 19$ $\delta(i,j) = k$ $\delta(0,2) = 1$	$i=0, \quad j-i=3 \quad k=2$ $w(0,3) = w(0,2) + p_3 + q_3$ $= 12 + 1 + 1$ $= 14$ $c(0,3) = w(0,3) + \min\{c(0,0) + c(1,3)\}$ $= 14 + \min\{0 + 12\}$ $= 14 + 12$ $c(0,3) = 26$ $\delta(0,3) = k$ $\delta(0,3) = 2$
---	--

$i=1 \quad j-i=4 \quad k=2$ $w(1,4) = w(1,3) + p_4 + q_4$ $= 9 + 1 + 1$ $= 11$ $c(1,4) = w(1,4) + \min\{c(1,1) + c(2,4)\}$ $= 11 + \min\{0 + 8\}$ $= 11 + 8$ $= 19$ $\delta(i,j) = k$ $\delta(1,4) = 2$	$i=0 \quad j-i=4 \quad k=2$ $w(0,4) = w(0,3) + p_4 + q_4$ $= 14 + 1 + 1$ $= 16$ $c(0,4) = w(0,4) + \min\{c(0,0) + c(1,4)\}$ $= 16 + \min\{0 + 19\}$ $= 16 + 19$ $= 35$ $\delta(i,j) = k$ $\delta(0,4) = 2$
---	--

The tree for  $T_{04}$  has a root,  $\delta_{04}$ . The value of  $\delta_{04} = 2$ .  
 From  $(a_1, a_2, a_3, a_4) = (do, if, int, while)$ ,  $a_2$  becomes the root node.  
 $\delta_{ij} = k$   
 $\delta_{04} = 2$





### 3.6. 4 0/1 Knapsack problem

**Problem Description :-** If we are given  $n$  objects and a knapsack or a bag in which the object  $i$  that has weight  $w_i$  is to be placed. The knapsack has a capacity  $W$ . Then the profit that can be earned  $P_i x_i$ . The objective is to obtain filling of knapsack with maximum profit earned.

$$\text{maximized } \sum_{i=1}^n P_i x_i \text{ subject to constraint } \sum_{i=1}^n w_i x_i \leq W$$

where  $1 \leq i \leq n$  and  $n$  is total number of objects. And  $x_i = 0$  or  $1$ .

**Step 1 :-** The notations used are  
 Let,  $f_i(y_i)$  be the value of optimal solution.  
 Then  $s^i$  is a pair  $(P, w)$  where  $P = f(y_i)$  and  $w = y_j$   
 Initially  $s^0 = \{(0, 0)\}$   
 we can compute  $s^{i+1}$  from  $s^i$   
 these computations of  $s^i$  are basically the sequence  
 of decisions made to obtaining the optimal solutions.

**Step 2 :-** we can generate the sequence of decisions  
 in order to obtain the optimum solution to  
 solving the knapsack problem.

**Step 3 :-** The formula that is used while solving 0/1 knapsack  
 is Let  $f_i(y_i)$  be the value of optimal solution. Then  
 $f_i(y) = \max \{f_{i-1}(y), f_{i-1}(y - w_i) + P_i\}$   
 Initially compute  
 $s^0 = \{(0, 0)\}$   
 $s^1 = \{(P_1, w_1) | (P - P_1, w - w_1) \in s^0\}$   
 $s^{i+1}$  can be computed by merging  $s^i$  and  $s_i$

## Purging Rule

**Purging Rule :-** If  $s^{i+1}$  contains  $(P_j, w_j)$  and  $(P_k, w_k)$   
 these two pairs such that  $P_j \leq P_k$  and  $w_j \geq w_k$  then  
 $(P_j, w_j)$  can be eliminated.

This purging rule is also called as  
 dominance rule. In purging rule basically the dominated  
 tuples gets purged. In short, remove the pair with  
less profit and more weight.

## Example

Example :- consider knapsack instance  $M=8$  and  $n=4$   
 Let  $P_i$  and  $w_i$  are as shown below.

$i$	$P_i$	$w_i$
1	1	2
2	2	3
3	5	4
4	6	5

Sol:- Initially  $S^0 = \{(0,0)\}$   
 $S_1 = \{(1,2)\}$   $\rightarrow S_1 = \{\text{select next } (P,w) \text{ and add it to } S^0\}$   
 $\quad \quad \quad = \{(1,2) (1+0, 2+0)\}$   
 $\quad \quad \quad = \{(1,2)\}$   $\therefore$  repetition of  $(1,2)$  is avoided.

$S^1 = \{\text{merge } S^0 \text{ and } S_1\}$   
 $S^1 = \{(0,0) (1,2)\}$   
 $S'_1 = \{\text{select next } (P,w) \text{ and add it to } S^1\}$   
 $S'_1 = \{(2,3), (2+0, 3+0), (3,5)\}$   
 $S^2 = \{(2,3), (3,5)\}$   $\therefore$  repetition of  $(2,3)$  is avoided.

$S^2 = \{\text{merge } S^1, S'_1\}$   
 $\quad = \{(0,0) (1,2) (2,3) (3,5)\}$   
 $S^2_1 = \{\text{select next } (P,w) \text{ and add it to } S^2\}$   
 $S^2_1 = \{(5,4) (6,6) (7,7) (8,9)\}$   
 $S^3 = \{\text{merge } S^2, S^2_1\}$   
 $S^3 = \{(0,0), (1,2), (2,3), (5,4), (6,6), (7,7), (8,9)\}$

Remove  
less profit and more  
weight  $(P_j, w_j)$   $(P_k, w_k)$   
 $(3, 5)$   $(5, 4)$   
↑ ↑  
so eliminate  $(P_j, w_j)$ .

pruning RULE  
 $(P_j, w_j)$  and  $(P_k, w_k)$   
 ~~$(P_j, w_j)$~~   $P_j \leq P_k$  and  $w_j \geq w_k$   
 $(P_j, w_j)$   $(P_k, w_k)$   
 $(3, 5)$   $(5, 4)$   
 $3 \leq 5$  and  $5 \geq 4$  TRUE  
hence we will eliminate  
 $(P_j, w_j)$  ie;  $(3, 5)$ .

$S^3 = \{ \text{select next } (p, w) \text{ and add it to } S^3 \}$   
 $S^3 = \{ (6, 5), (7, 7), (8, 8), (11, 9), (12, 11), (13, 12), (14, 14) \}$   
 $S^4 = \{ \text{merge } S^3 \text{ and } S^3 \}$   
 $S^4 = \{ (0, 0), (1, 2), (2, 3), (5, 4), (6, 6), (7, 7), (8, 9), (6, 5), (8, 8), (11, 9), (12, 11), (13, 12), (14, 14) \}$

↓ ↓  
 $(8, 9)$   $(6, 5)$   
 $(P_j, w_j)$   $(P_k, w_k)$   
so False.  
we can not remove  
 ~~$(P_j, w_j)$~~ .

$(P_i, w_i) \in S^n$  and  $(P_j, w_j) \notin S^{n-1}$   
TRUE  
 $x_n = 1$   
else,  
 $x_n = 0$ .

Now we are interested in  $M=8$ .

we get  $(8,8)$  is in  $S^4$

$(p_i, w_i) \in S^n$  and  $(p_i, w_i) \notin S^{n-1}$   
 $(8,8) \in S^4$  and  $(8,8) \notin S^3 \rightarrow \text{True}$

$$x_4 = 1$$

$(8,8) - (6,5)$

$(8-6)$  and  $(8-5)$

$(2,3) \in S^3$  and  $(2,3) \notin S^2$

False

$$x_3 = 0$$

$(2,3)$  and  $(5,4)$

$(3,1) \in S^2$  and  $(3,1) \notin S^1$

False

$$x_2 = 0$$

$(3,1) - (2,3)$

$(3-2)$  and  $(1-3)$

$(1,2) \in S^1$  and  $(1,2) \notin S^0$

True

$$x_1 = 1$$

$$\therefore x_1 = 1, x_2 = 0, x_3 = 0 \text{ and } x_4 = 1$$

maximum profit is

$$\begin{aligned} \sum p_i x_i &= p_1 x_1 + p_2 x_2 + p_3 x_3 + p_4 x_4 \\ &= 1 \times 1 + 2 \times 0 + 5 \times 0 + 6 \times 1 \\ &= 1 + 0 + 0 + 6 \\ &= 7. \end{aligned}$$

Example 2:- consider the knapsack problem the instance  $n=4$ .  $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$  and  $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$  and  $m=21$  then we can generate sequence of decisions.

### 3.6.5 Travelling sales person problem.

Let  $G$  be directed graph denoted by  $(V, E)$  where  $V$  denotes set of vertices and  $E$  denotes set of edges. The edges are given along with their cost  $C_{ij}$ . The cost  $C_{ij} > 0$  for all  $i$  and  $j$ . if there is no edge between  $i$  and  $j$  then  $C_{ij} = \infty$

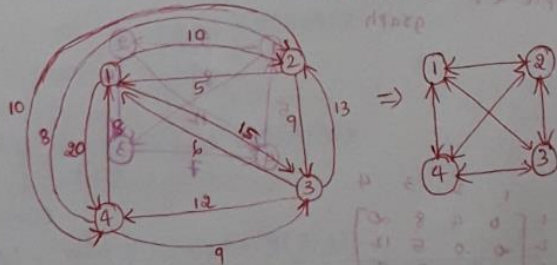
A tour for the graph should be such that all the vertices should be visited only once and cost of the tour is sum of cost of edges on the tour. The travelling sales person problem is to find the tour of minimum cost. The following formula can be used to obtain the optimum cost tour.

1.  $g(i, \emptyset) = c_{i,1}, 1 \leq i \leq n$
2.  $g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$

$g(i, S)$  means  $i$  is starting node and the nodes in  $S$  are to be traversed.

$\min_{j \in S}$  is considered as the intermediate node  $g(j, S - \{j\})$  means  $j$  is already traversed. so next we have to traverse  $S - \{j\}$  with  $j$  as starting point.

Example :-



Sol:  $g(1, \{2, 3, 4\}) = ?$

step 1:-

	1	2	3	4
1	0	10	15	20
2	5	0	10	
3	6	13	0	12
4	8	8	9	0

For  $S = \emptyset$

$g(1, \emptyset) = c_{1,1} = 0$

$g(2, \emptyset) = c_{2,1} = 5$

$g(3, \emptyset) = c_{3,1} = 6$

$g(4, \emptyset) = c_{4,1} = 8$



Step 2:-

$$F8) |S| = 1$$

$$g(2, \{3\}) = \min \{ c(2,3) + g(3, \emptyset) \}$$
$$= \min \{ 9 + 6 \}$$

$$g(2, \{3\}) = 15$$

$$g(2, \{4\}) = \min \{ c(2,4) + g(4, \emptyset) \}$$
$$= \min \{ 10 + 8 \}$$

$$g(2, \{4\}) = 18$$

$$g(3, \{2\}) = \min \{ c(3,2) + g(2, \emptyset) \}$$
$$= \min \{ 13 + 5 \}$$
$$= 18$$

$$g(3, \{4\}) = \min \{ c(3,4) + g(4, \emptyset) \}$$
$$= \min \{ 12 + 8 \}$$
$$= 20$$

$$g(4, \{2\}) = \min \{ c(4,2) + g(2, \emptyset) \}$$
$$= \min \{ 8 + 5 \}$$
$$= 13$$

$$g(4, \{3\}) = \min \{ c(4,3) + g(3, \emptyset) \}$$
$$= \min \{ 9 + 6 \}$$
$$= 15$$

Step 3 :-

For  $|S|=2$

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

$$g(2, \{3, 4\}) = 25$$

$$g(3, \{2, 4\}) = \min \{ c(3,2) + g(2, \{4\}), c(3,4) + g(4, \{2\}) \}$$
$$= \min \{ 13 + 18, 12 + 13 \}$$
$$= \min \{ 31, 25 \}$$

$$g(3, \{2, 4\}) = 25$$

$$g(4, \{2, 3\}) = \min \{ c(4,2) + g(2, \{3\}), c(4,3) + g(3, \{2\}) \}$$
$$= \min \{ 8 + 15, 9 + 18 \}$$
$$= \min \{ 23, 27 \}$$

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

Step 4 :- For  $|S|=3$

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

$$= \min \{ c(1,2) + g(2, \{3, 4\}), c(1,3) + g(3, \{2, 4\}), c(1,4) + g(4, \{2, 3\}) \}$$

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

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

$$= 35$$

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

∴ The optimal tour is  $1 - 2 - 4 - 3 - 1$   
 $10 + 10 + 9 + 6$

∴ The optimal tour cost is 35

### 3.6.6 String Editing

There are two strings given. The first string is the source string and the second string is the target string. In this program, we have to find how many possible edits are needed to convert first string to the second string.

The edit of strings can be either Insert some elements, delete something from the first string or modify something to convert into the second string.

Given two strings str1 and str2 and below operations that can be performed on str1. Find the minimum

number of edits (operations) required to convert 'str1' into 'str2'.

- Insert
- Remove
- Replace

All of the above operations are of equal cost.

Minimum Edit Distance

	NULL	a	b	c	f	g
NULL						
a						
d						
c						
e						
g						

insert →      ↓ Remove

1. If  $r \neq c$

Replace	Remove
insert	$\min(\text{Replace}, \text{Remove}, \text{insert}) + 1$

2. If  $r = c$   
Just copy the diagonal element.


**Sol:**

Convert a,d,c,e,g string into a,b,c,f,g.

**Let fill first row**

Convert NULL to NULL it requires 0 operations.

Convert NULL to a it requires 1 operation (ie., insert a)

Convert NULL to b it requires 2 operation (ie., insert a, b)

Convert NULL to c it requires 3 operation (ie., insert a,b, c)

Convert NULL to a it requires 4 operation (ie., insert a,b,c, f)

Convert NULL to a it requires 5 operation (ie., insert a,b,c,f, g)

### **Let's fill first column**

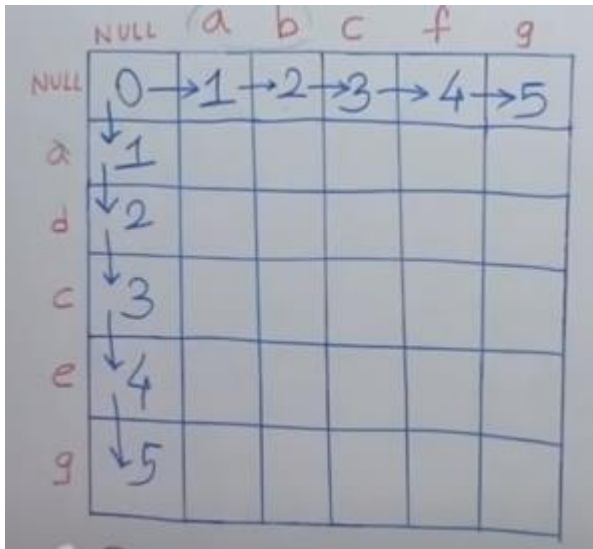
If we have convert a to NULL it requires 1 operations (ie., remove a)

If we have convert a to NULL it requires 2 operations (ie., remove a, d )

If we have convert a to NULL it requires 3 operations (ie., remove a, d, c )

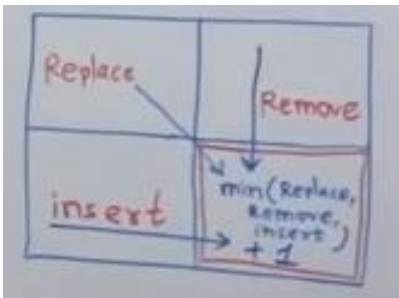
If we have convert a to NULL it requires 4 operations (ie., remove a, d, c ,e )

If we have convert a to NULL it requires 5 operations (ie., remove a, d, c ,e,g )

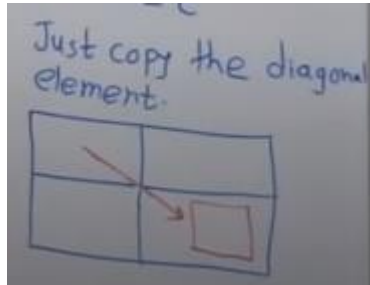


Now fill internal cell use conditions

1. If  $x \neq c$



2. If  $x == c$



	NULL	a	b	c	f	g
NULL	0 → 1 → 2 → 3 → 4 → 5					
a	↓ 1	0 → 1 → 2 → 3 → 4				
d	↓ 2	↓ 1	1 → 2 → 3 → 4			
c	↓ 3	↓ 2	↓ 1	1 → 2 → 3		
e	↓ 4	↓ 3	↓ 2	↓ 1	2 → 3	
g	↓ 5	↓ 4	↓ 3	↓ 2	↓ 1	2

adceg  
 ↓ ↓  
 abcfg

## UNIT-IV

**Backtracking:** General Method, 8-Queens Problem, Sum of Subsets problem, Graph Coloring, 0/1 Knapsack Problem

**Branch and Bound:** The General Method, 0/1 Knapsack Problem, Travelling Salesperson problem.

### **4.1 Introduction**

In case of greedy and dynamic programming techniques, we will use brute force approach. It means, we will evaluate all possible solutions, among which we select one solution as optimal solution.

In backtracking technique, we will get same optimal solution with a less number of steps. So by using backtracking technique, we will solve problems in an efficient way, when compared to other methods like greedy method and dynamic programming.

In this we will use bounding functions(criterion function), implicit and explicit constraints.

**Explicit constraints:** These are rules which restrict each  $x_i$  to take on values only from a given set.

Example: 1. In knapsack problem, the explicit constraints are  $x_i = 0$  or  $1$   $0 \leq x_i \leq 1$



2. In 4-queens problem, 4-queens can be placed in 4X4 chessboard in 44 ways.

**Implicit constraints:** These are rules which determine which of the tuples in the solution space satisfy the criterion function.

**Example:** In 4-queens problem, the implicit constraints are no two queens can be on the same column, same row and same diagonal.

## **Terminology in Backtracking**

### **1. Criterion Function**

It is a function  $p(x_1, x_2, \dots, x_n)$  which needs to be maximized or minimized for a given problem.

### **2. Solution space**

All the tuples that satisfy the explicit constraints define a possible solution space for a particular instance 'I' of the problem.

### **3. Problem state**

Each node in the tree organization defines a problem state.

### **4. Solution states**

These are those problem states  $s$  for which the path from the root to  $S$  defines a tuple in the solution space.

## **5. State space tree**

If we represent solution space in the form of a tree then the tree is referred as the state space tree.

## **6. Answer state**

These solution states  $s$  for which the path from the root to  $s$  defines a tuple which is a member of the set of solutions

## **7. Live node**

A node which has been generated and all of whose children have not yet been generated is live node.

## **8. E-node**

The live nodes whose children are currently being generated is called the E-node (node being expanded).

## **9. Dead node**

It is generated node that is either not to be expanded further.

## **4.2 General Method**

In the backtracking method

- a) The desired solution is expressible as an  $n$  tuple  $(x_1, x_2, \dots, x_n)$  where  $x_i$  is chosen from some finite set  $s_i$ .
- b) The solution maximizes or minimizes or stratifies a criterion function

- The basic idea of backtracking is to build up a vector, one component at a time and to test whether the vector being formed has any chance of success.
- Backtracking algorithm determines the solution by systematically searching the solution space for the given problem.

### **4.3 Applications of Backtracking**

4.3.1 8-Queens Problem

4.3.2 Sum of Subsets problem

4.3.3 Graph Coloring

4.3.4 0/1 Knapsack Problem

#### **4.3.1 8-Queens Problem**

##### **1. n-Queens problem(4,8 Queens problem)**

Consider an  $n \times n$  chessboard. Let there are  $n$  queens. These  $n$  queens are to be placed on the  $n \times n$  chessboard so that no two queens are on the same column, same row or same diagonal.

- AlgorithmNQueens(k,n)
- //Using backtracking,this procedure prints all possible //placements of n queens on an n\*n //chessboard so that they are non attacking

```

{
  for(i=1 to n ) do
  {
    if Place(k,i) then
    {
      x[k]=i
      If(k=n) then write (x[1:n]);
      else Nqueens(k+1,n);
    }
  }
}

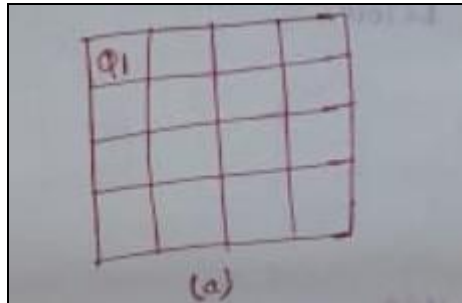
```

### 4-queens problem

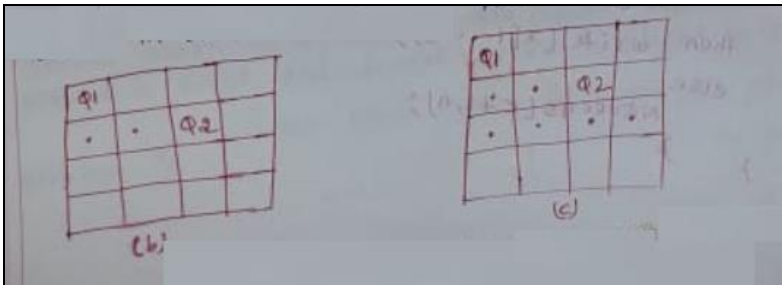
Consider a 4X4 chessboard. Let there are 4-queens. The objective is to place there 4 queens on 4X4 chessboard in such away that no two queens should be placed in the same row, same column or diagonal position.

- The explicit constraints are 4-queens are to be placed on 4X4 chessboard in  $4^4$  ways
- The implicit constraints are no two queens are in the same row or column or diagonal.

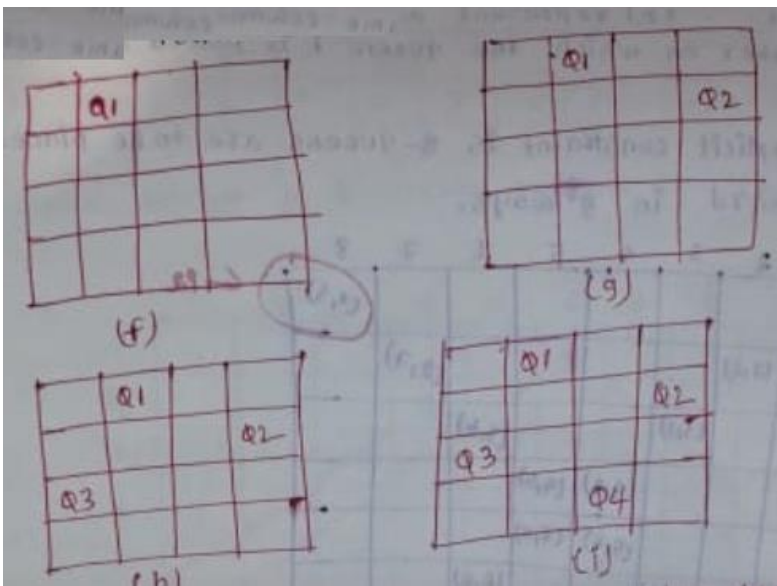
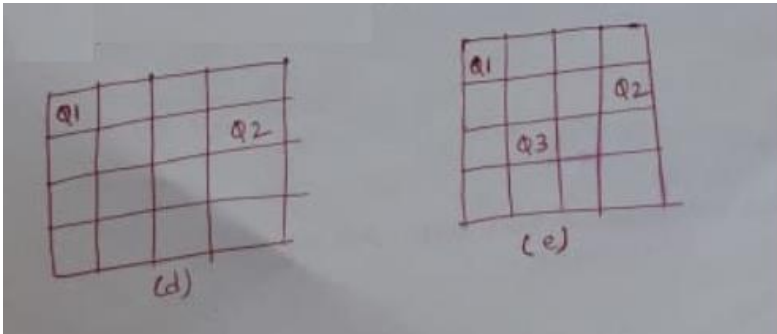
Let  $\{x_1, x_2, x_3, x_4\}$  be the solution vector where  $x_i$ , column number on which the queens  $i$  is placed. First queen  $Q_1$  is placed in first row and first column.



The second queen should not be placed in first row and second column. It should be placed in second row and in second, third or fourth column. If we placed in second column, both will be in same diagonal, so place it in third column.



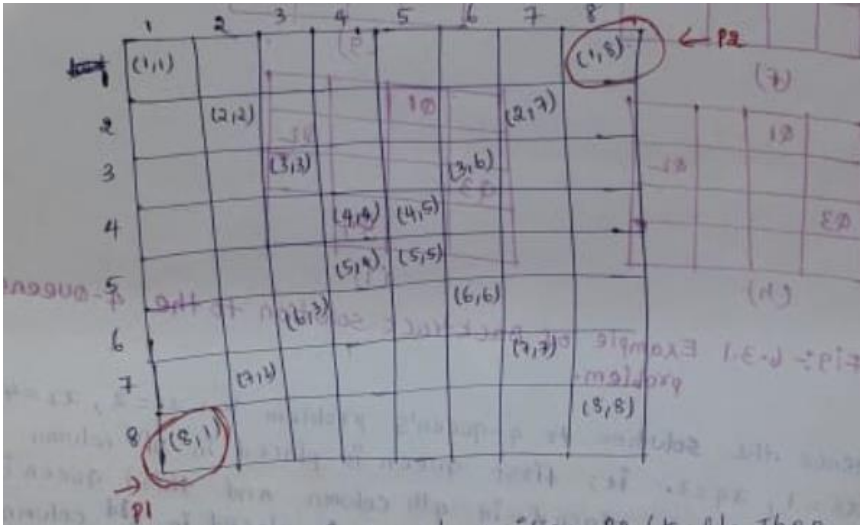
We are unable to place  $Q_3$  in third row so go back to  $Q_2$  and place it somewhere else.



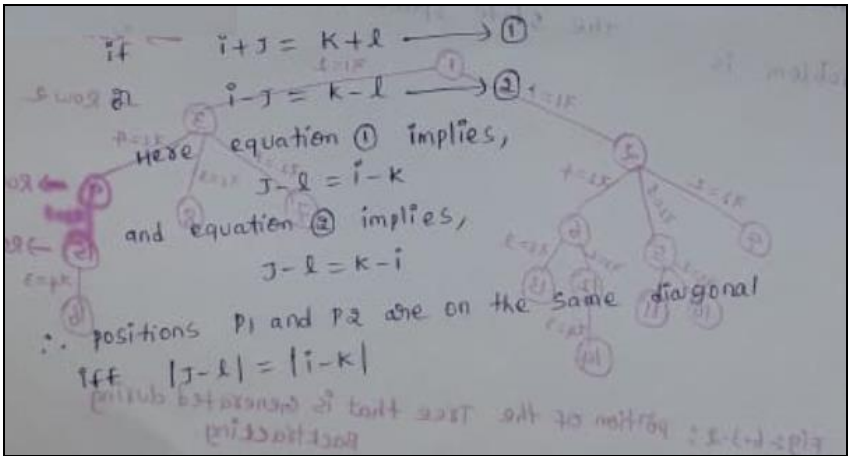
## 8-Queens problem

Consider a 8X8 chessboard. Let there are 8 queens. The objective is to place these 8 queens on the board so that no two queens are on the same row or same column or same diagonal.

The explicit constraint is 8-queens are to be placed in 8X8 chessboard in  $8^8$  ways.



Let position  $P1=(i, j)$  and position  $P2=(k, l)$ . Then  $P1$  and  $P2$  are on the same diagonal.



Ex<sup>n</sup> instance  $\therefore P_1 = (8, 1)$  and  $P_2 = (1, 8)$

$\therefore i = 8, j = 1$  and  $k = 1, l = 8$

$$i+j = k+l$$

$$8+1 = 1+8 = 9$$

$$j-l = k-i$$

$$1-8 = 1-8$$

Hence  $P_1, P_2$  i.e.  $(8, 1)$  and  $(1, 8)$  are on the same diagonal.

**Example1:** for the following feasible sequence solve 8-queens problem. (6,4,7,1) using back tracking.

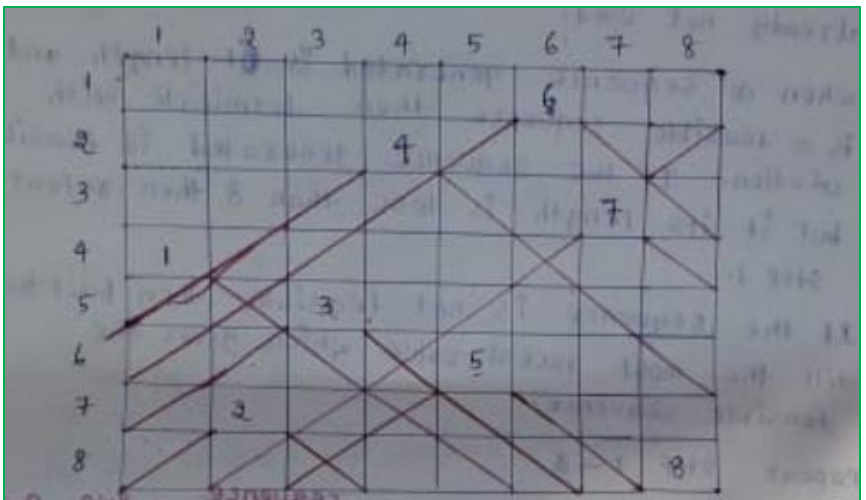
**Sol:**



Sol:- Here the given sequence represents column position and each queen is placed row by row on given column. i.e; place 1<sup>st</sup> queens at (1,6)  
 2<sup>nd</sup> queen at (2,4)  
 3<sup>rd</sup> queen at (3,7)  
 4<sup>th</sup> queen at (4,1)

Queen positions								Action
1	2	3	4	5	6	7	8	
6	4	7	1					start
6	4	7	1	2				Not feasible $5-4=2-1$
6	4	7	1	3				
6	4	7	1	3	2			Not feasible $6-5=3-2$
6	4	7	1	3	5			
6	4	7	1	3	5	2		
6	4	7	1	3	5	2	8	list ends, feasible sequence

Hence, solution to 8-queens problem is (6,4,7,1,3,5,2,8).



**Time complexity:** The solution space tree of 8-queens problem contain  $8^8$  tuple. After imposing implicit constraint, the size of solution space is reduced to  $8!$  Tuples. Hence the time complexity is  $O(8!)$ . For n-queens problem, it is  $O(n!)$

### 4.3.2 Sum of Subsets problem

- Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K.
- We are considering the set contains non-negative values.
- It is assumed that the input set is unique (no duplicates are presented).

#### Example

Consider a set  $S = \{5, 10, 12, 13, 15, 18\}$  an  $N=30$

Subset	Sum=0	Action initially set is empty
5	5	
5,10	15	
5,10,12	27	
5,10,12,13	40	Sum exceeds

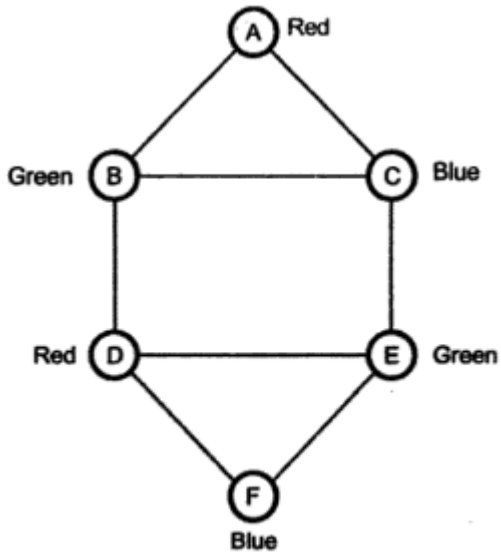
		N=30 hence backtrack
5,10,12,13		Not feasible
5,10		List ends. Backtrack
5,10,13	28	
5,10,13,15	33	Not feasible. Backtrack
5,10	15	

### 4.3.3 Graph Coloring

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 m-coloring problem. If the degree of given graph is d then we can color it with  $d + 1$  colors. The least number of colors needed to color the graph is called its chromatic number.

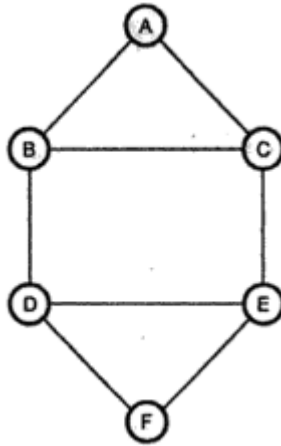
For example : Consider a graph given in Fig. below. As given in Fig. we require three colors to color the graph. Hence the chromatic number of given graph is 3. We can

use backtracking technique to solve the graph coloring problem as follows –



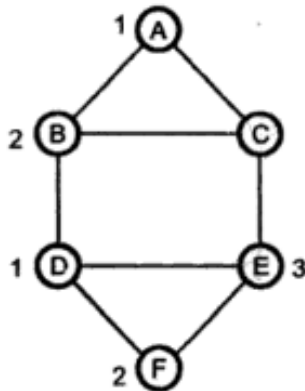
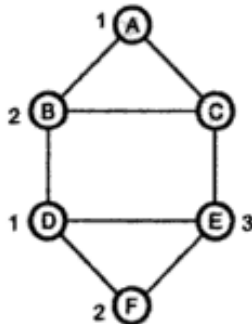
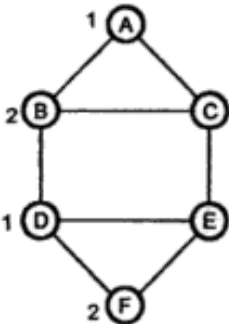
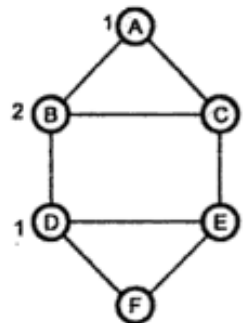
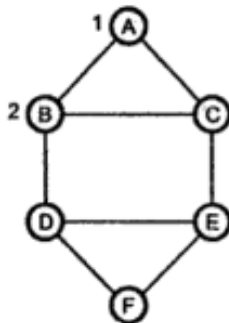
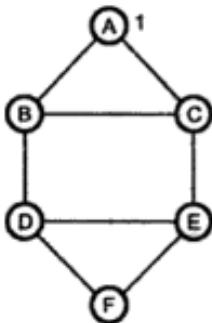
**Graph and its coloring**

**Step1:**



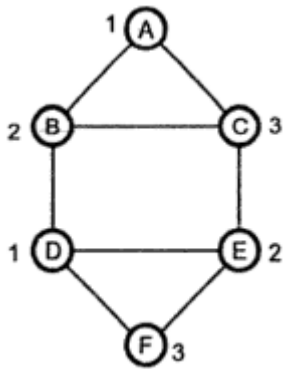
A graph G consists of vertices from A to F. There are colors used Red, Green and Blue. We will number them out. That means 1 indicates Red, 2 indicates Green and 3 indicates Blue color.

**Step 2:**

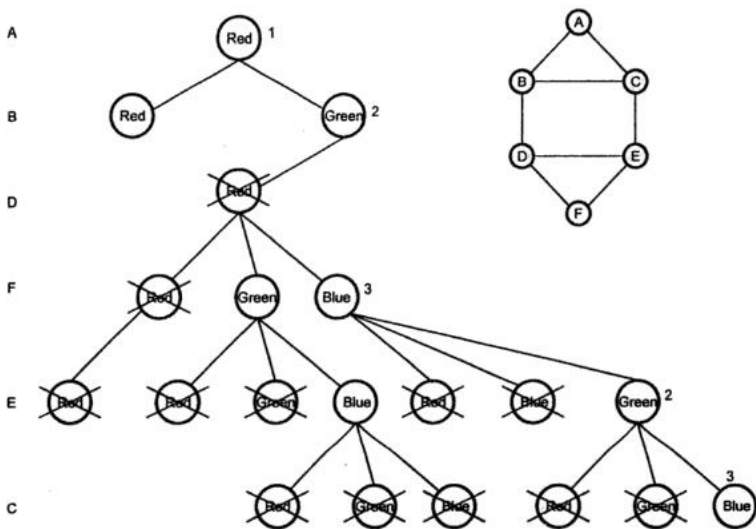


Stuck Here !!  
 Cannot assign  
 1 or 2 or 3.  
 Hence backtrack

### Step 3:



Thus the graph coloring problem is solved. The state-space tree can be drawn for better understanding of graph coloring technique using backtracking approach-



Here we have assumed, color index Red=1, Green=2 and Blue=3.

#### **4.3.4 0/1 Knapsack Problem**



#### **4.4 Branch and Bound: Introduction**

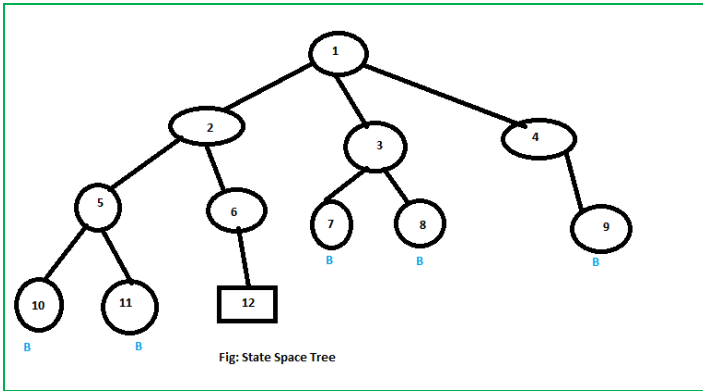
The back tracking algorithm is effective for decision problems, but it is not designed for optimization problems. This drawback is rectified in case of branch and bound technique.

The essential difference between back tracking and branch and bound is, if we get a solution then we will terminate the search procedure in back tracking, where as in branch and bound , we will continue the process(Search) until we get an optimal solution.

Branch and bound technique is applicable for only minimization problems.

We will use Three types of search strategies in branch and bound.

- 1.FIFO search
2. LIFO search
- 3.LC(Least Cost) search



## 4.5 General Methods

### FIFO Branch and Bound search

Assume the node is an answer node. In FIFO search, first we will take E-node as node 1. next we generate the children of node 1.. We will place all these live nodes in a queue.

This can be shown in fig:

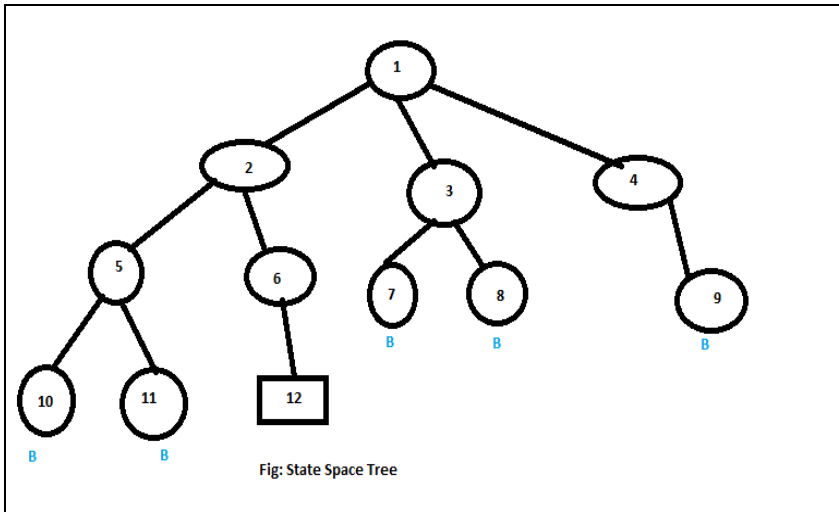


Now, we will delete an element from queue i.e., node 2 next generate children of node 2 and place in the queue.



Next, delete an element from queue and take it as E-node generate the children of node 3. 7 , 8 are children of 3 and

these live nodes are killed by bounding functions. So we will not include in the queue.



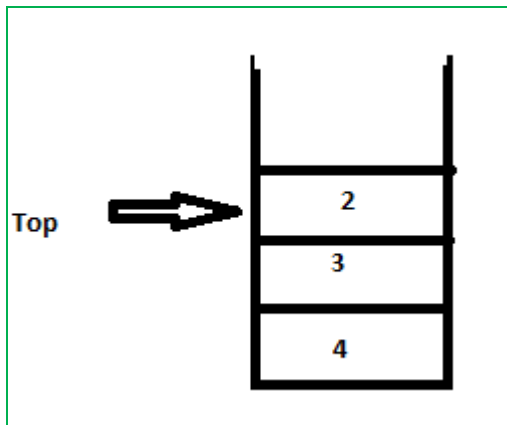
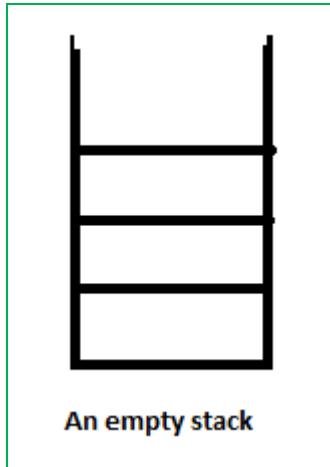
Again delete 4 from the queue. 9 is the children of 4 and this is live node are killed by bounding function.

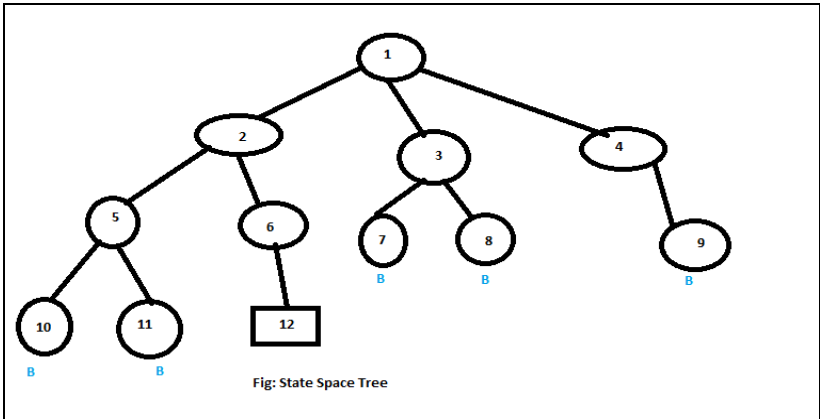


Next, delete 5 from the queue. 10 and 11 are generated and killed by boundary function. Last node in queue is 5. the child of node 6 is 12 and it satisfies the condition of the problem, which is the answer node, so search terminates.

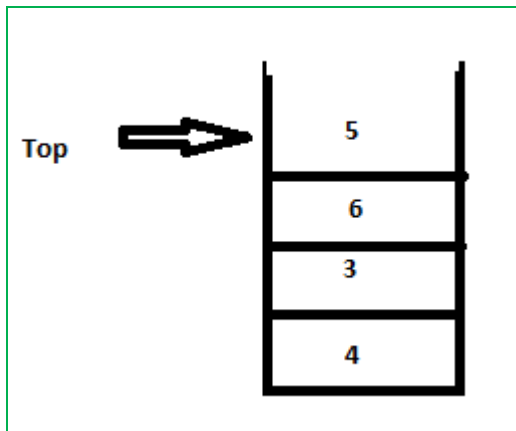
## 2. LIFO Branch and Bound search

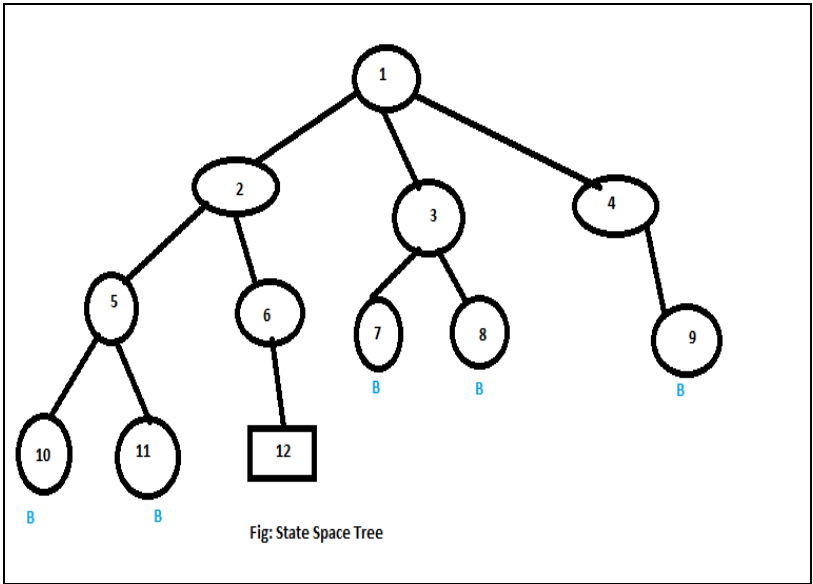
For this we will use a data structure called stack. Initially stack is empty. Generate children of node 1 and place these live nodes into stack. This can be shown in fig:



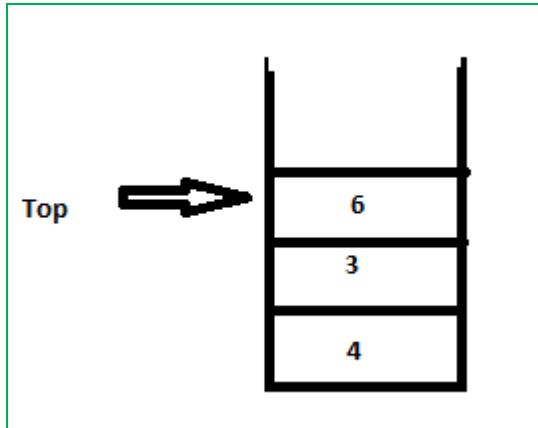


Remove element from stack and generate the children of it, place those nodes into stack. 2 is removed from stack. The children of 2 are 5, 6. the content of stack is





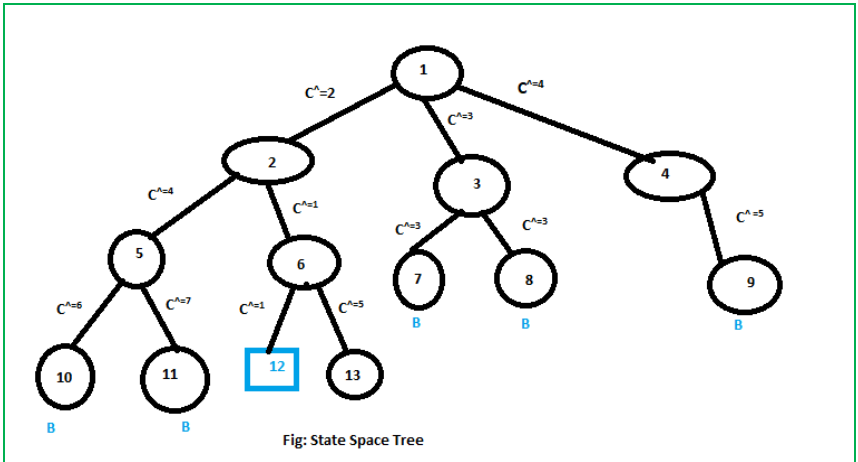
Again remove an element from stack ie; node 5 is removed and nodes generated by 5 are 10,11 which are killed by bounding function. So we will not place 10,11 into stack. This can be shown in fig:



Delete an element from stack ie; node 6. generate child of node 6 ie; node 12 which is the answer node, so search process terminates.

### **3. Least cost(LC) Branch and Bound search**

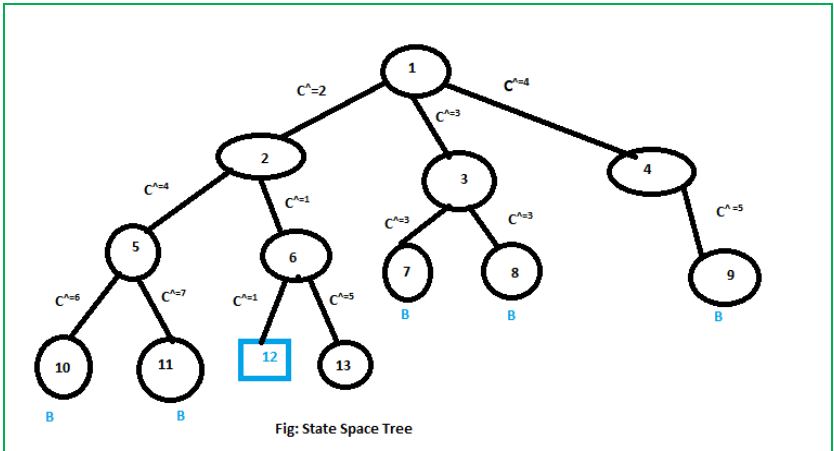
In this we will use ranking function or cost function, which is denoted by  $C^{\wedge}(x)$ . Which generates the children of E-node, among these live nodes, we select a node which has minimum cost. By using ranking function we will calculate the cost of each node.



Initially we will take node 1 as E-node. Generate children of node 1, the children are 2,3,4. by using ranking function we will calculate the cost of 2,3,4 node is  $C^=2$ ,  $C^=3$ , AND  $C^=4$  respectively.

Now we will select a node which has minimum cost ie; node 2.





For node 2, the children are 5 , 6 . The cost of the node 5 is  $C^A=4$  and node 6 is  $C^A=1$  so select node 6.

For the node 6 the children are 12 and 13. now we will select a node which has minimum cost ie; node 12. node 12 is the answer node. So, we terminate search process.

## 4.6 Applications

### 4.6.1 Travelling Salesperson Problem

### 4.6.2 0/1 Knapsack problem

#### 4.6.1 Travelling Salesperson Problem

If there are  $n$  cities and cost of travelling from one city to other city is given. A salesman has to start from any one of the city and has to visit all the cities exactly once and has to

return to the starting place with shortest distance or minimum cost.

If there are  $n$  cities and cost of travelling from one city to other city is given. A sales man has to start from any one of the city and has to visit all the cities exactly once and has to return to the starting place with shortest distance or minimum cost.

Let  $G=(V,E)$  be a directed graph defining an instance of the travelling sales person problem. Let  $C_{ij}$  be the cost of the edge  $(i, j)$  and  $c_{ij} = \infty$ .

### **Reduced Cost Matrix**

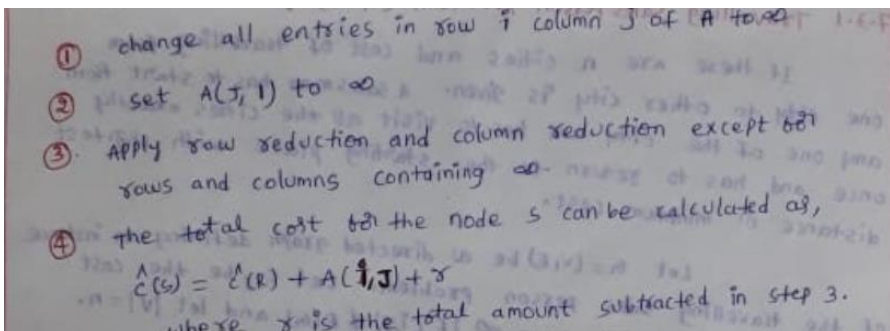
A row or column is said to be reduced if it contains at-least one zero and all remaining entries are non-negative. A matrix is reduced iff every row and column is reduced.

a) If a constant  $t$  is chosen to be minimum entry in row  $i$  or column  $j$  then subtracting it from all entries in row  $i$  (column  $j$ ) will reduce a zero into a row  $i$ (column  $j$ ).

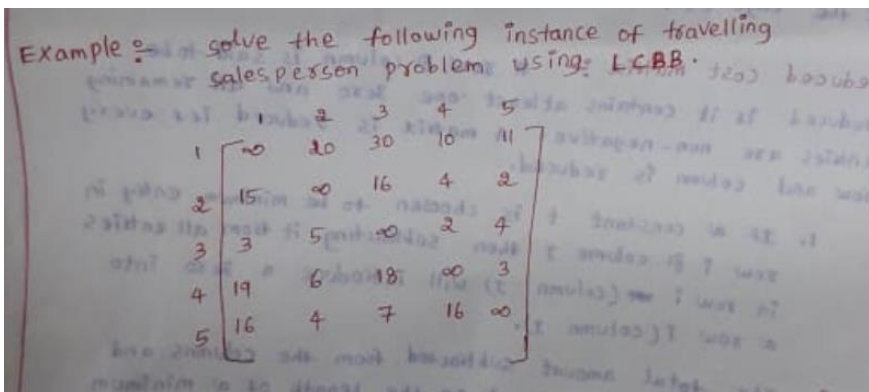
b) The total amount subtracted from the columns and rows is lower bound on the length of a minimum cost tour and can be used as the  $C^*(x)$ . value for the root of state space tree.

With every node in the state space tree, we associate a reduced cost matrix.

Let  $A$  be the reduced cost matrix for node  $R$ . Let  $S$  be the child of  $R$  such that the edge  $(R,s)$  corresponds to including edge  $(i, j)$  in the tour. If  $S$  is not a leaf node then the reduced cost matrix for node  $S$  can be obtained as follows.



## Example



Sol:-

Row Reduction

$$\begin{bmatrix} 10 & 20 & 30 & 10 & 11 \\ 15 & 0 & 14 & 4 & 2 \\ 3 & 5 & 0 & 2 & 4 \\ 19 & 6 & 15 & 0 & 3 \\ 16 & 9 & 7 & 16 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 10 & 20 & 30 & 10 & 11 \\ 13 & 0 & 14 & 2 & 0 \\ 1 & 3 & 0 & 0 & 2 \\ 16 & 3 & 15 & 0 & 0 \\ 12 & 0 & 3 & 12 & 0 \end{bmatrix}$$

Column Reduction

$$\begin{bmatrix} 10 & 20 & 30 & 10 & 11 \\ 13 & 0 & 14 & 2 & 0 \\ 1 & 3 & 0 & 0 & 2 \\ 16 & 3 & 15 & 0 & 0 \\ 12 & 0 & 3 & 12 & 0 \end{bmatrix} \Rightarrow A = \begin{bmatrix} 10 & 17 & 0 & 1 \\ 12 & 0 & 11 & 2 & 10 \\ 0 & 3 & 0 & 0 & 2 \\ 15 & 3 & 12 & 0 & 0 \\ 11 & 0 & 0 & 12 & 0 \end{bmatrix}$$

1 - 3 - - - = 4

Total amount subtracted  
 $x = 21 + 4 = 25$

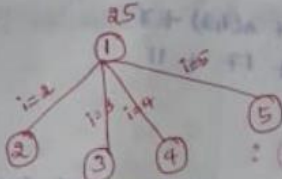


Fig: 7.3.1 part of a state space tree

consider the path (1, 2):

change all entries of first row and second column of reduced matrix to  $\infty$  and set (2,1) to  $\infty$ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

Apply row, column reduction then  $\gamma=0$ .

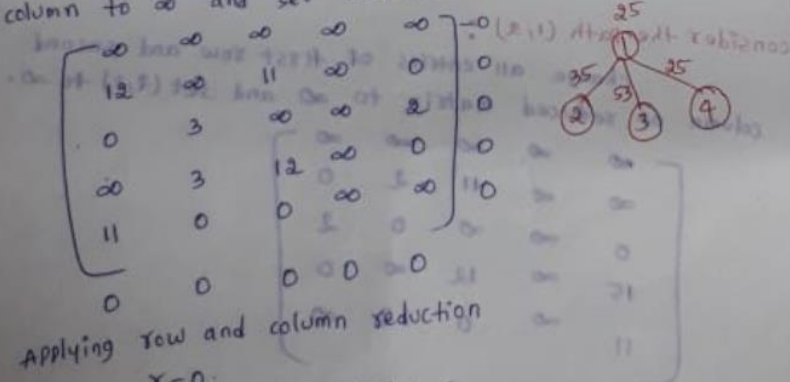
$\hat{C}(2) = \hat{C}(1) + A(2,1) + \gamma$   
 $\hat{C}(2) = 25 + 10 + 0$   
 $= 35$

consider path (1,3):  
 change all entries of first row and third column of reduced matrix to  $\infty$  and set (3,1) to 0.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 & 2 \\ \infty & 3 & \infty & 0 & 2 & 0 \\ 15 & 3 & \infty & \infty & 0 & 0 \\ 11 & 0 & 6 & 12 & \infty & 0 \end{bmatrix} \xrightarrow{\substack{-0 \\ -0 \\ -0 \\ -0 \\ -11}} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & \infty & 0 & 2 & \infty \\ 4 & 3 & \infty & \infty & 0 & 0 \\ 0 & 0 & \infty & 12 & \infty & 0 \end{bmatrix}$$

Applying row and column reduction  
 $\gamma = 11$   
 $\hat{C}(3) = \hat{C}(1) + A(1,3) + \gamma$   
 $= 25 + 17 + 11$   
 $= 53$

consider path (1,4):  
 change all entries of first row and fourth  
 column to  $\infty$  and set  $A(4,1)$  to  $\infty$ .



$$\hat{C}(4) = \hat{C}(1) + A(4,4) + \delta$$

$$= 25 + 0 + 0$$

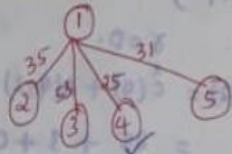
$$= 25$$

consider the path (1,5)

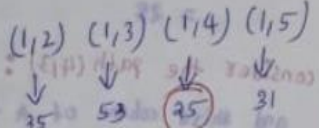
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 12 & 0 & 11 & 2 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 15 & 3 & 12 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 10 & 11 & 9 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 12 & 0 & 9 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 \end{bmatrix}$$

$$\begin{aligned} \hat{c}(5) &= \hat{c}(1) + A(1,5) + x \\ &= 25 + 1 + 5 \\ &= 31 \end{aligned}$$



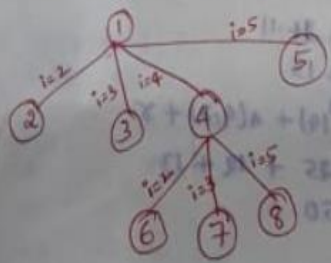
since the minimum cost is 25  
select node 4.





The matrix obtained for path (1,4) is considered as reduced cost matrix

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$



Consider the path (4,2): change all entries of fourth row  
 and second column reduced matrix to  $\infty$  and set  $A(2,1)$  to  $\infty$ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Applying row, column reduction

$\delta = 0$

$$\hat{C}(2) = \hat{C}(4) + A(4,2) + \delta$$

$$= 25 + 3 + 0$$

(2,1)  $\hat{C}(1) = 28$

consider the path (4,3) : change all entries of fourth row and third column of A to  $\infty$  and set (3,1) to  $\infty$ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \\ 11 & - & - & - & - \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \\ \infty & - & - & - & - \end{bmatrix}$$

applying row and column reduction

$$\gamma = 2 + 11 = 13$$

$$\begin{aligned} \hat{C}(3) &= \hat{C}(4) + A(4,3) + \gamma \\ &= 25 + 12 + 13 \\ &= 50 \end{aligned}$$

consider the path (4,5) : change all entries of fourth row and fifth column of A to  $\infty$  and A(5,1) to  $\infty$ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & \infty \\ \infty & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & 0 & 0 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & \infty \\ \infty & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & 0 & 0 \\ \infty & 0 & 0 & 0 & 0 \end{bmatrix}$$

Applying row and column reduction : (1,5)  $\rightarrow$  11

$\gamma = 11 + 0$   
 $\gamma = 11$

$$C(5) = C(4) + A(4,5) + \gamma$$

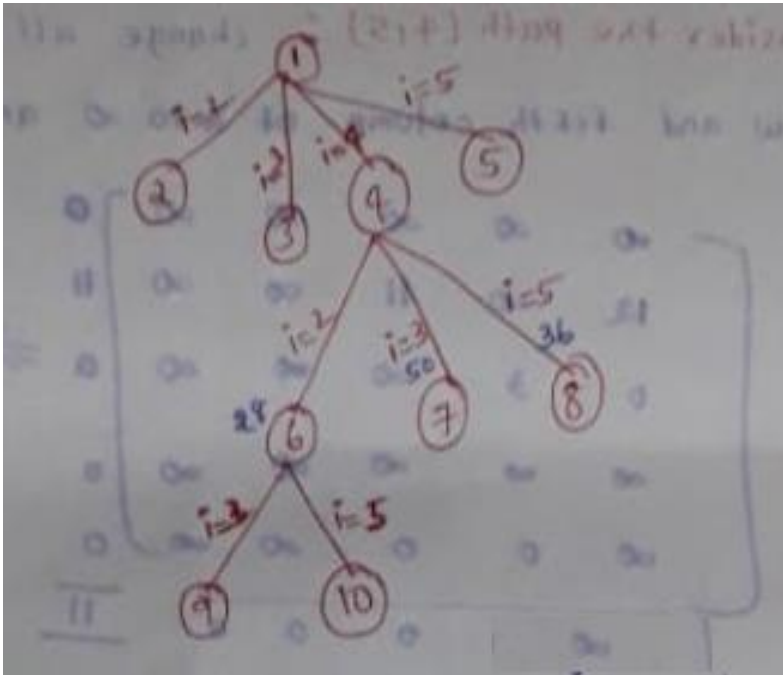
$$= 25 + 0 + 11 = 36$$

Since the minimum cost is 28 select node 2.

$(4,2)$   $(4,3)$   $(4,5)$   
 $\downarrow$   $\downarrow$   $\downarrow$   
 $28$   $50$   $36$

The matrix obtained for path (4,2) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$



consider the path (2/3) : change all entries of <sup>second</sup> row and third column to  $\infty$  and set  $A(3,1)$  to  $\infty$

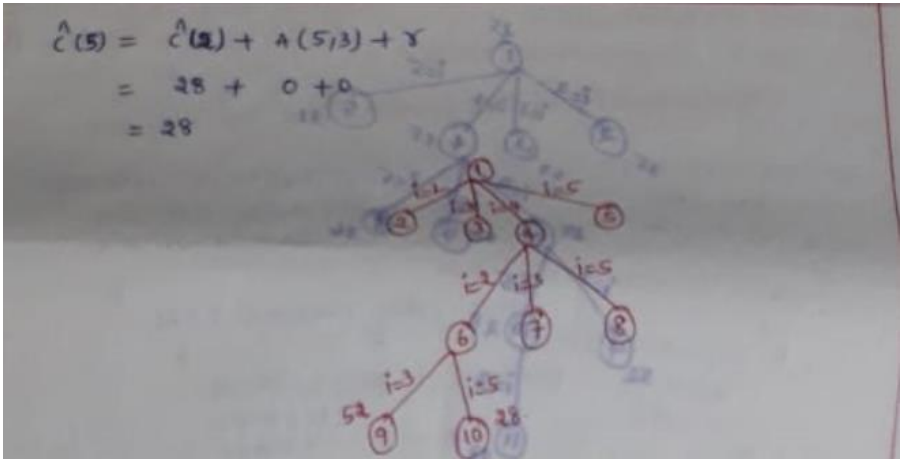


$\delta = 28 + 11$   
 $\delta = 13$   
 $C^1(3) = C^1(2) + A(2,3) + \delta$   
 $= 28 + 11 + 13$   
 $= 52$

consider the path (2/5) : change all entries of second row and fifth column to  $\infty$  and set  $A(5,1)$  to  $\infty$ .



$\delta = 0$



since the minimum cost is 28, select node 5.  
 The matrix obtained for path (2,5) is considered as reduced cost matrix.

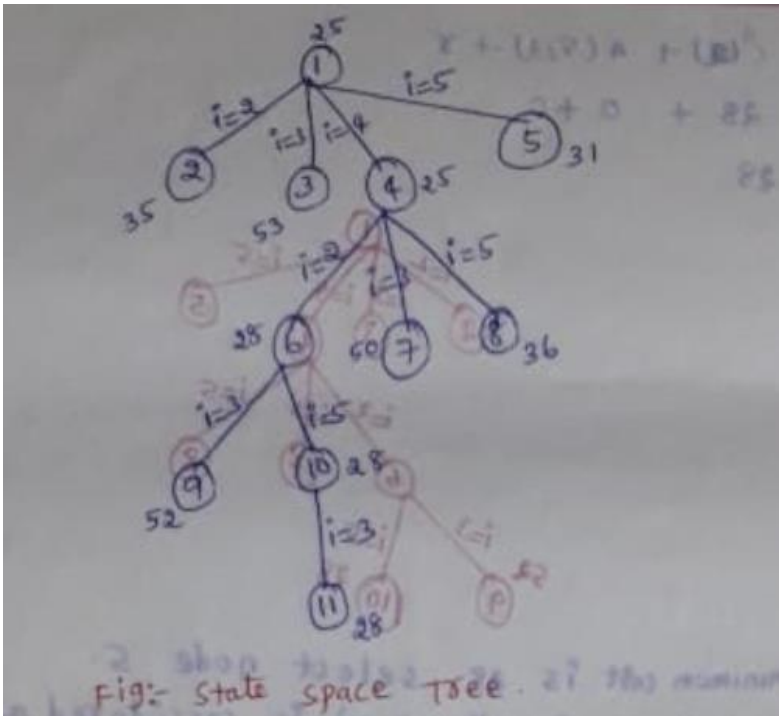
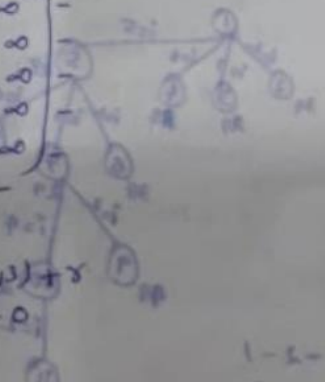
$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$2 + 2 + 2 + 2 + 0 = 10$  minimum  
 $28 \in$

consider the path (5,3) : changes all entries in fifth row and third column to  $\infty$  and  $A(3,1)$  to  $\infty$ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

$$\begin{aligned} \gamma &= 0 \\ C(3) &= C(5) + A(5,3) + \gamma \\ &= 28 + 0 + 0 \\ &= 28 \end{aligned}$$

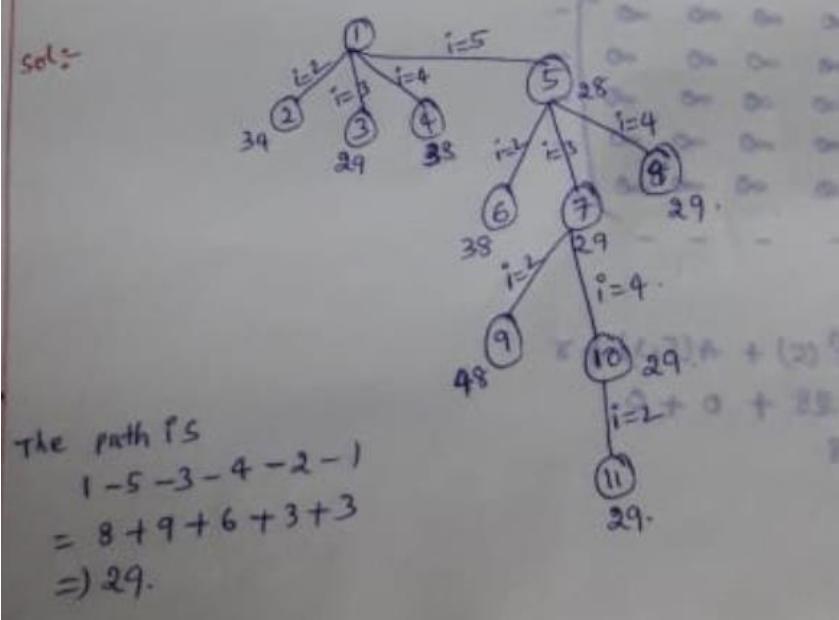




The path is 1-4-2-5-3-1  
 minimum cost = 10 + 6 + 2 + 7 + 3  
 ⇒ 28.

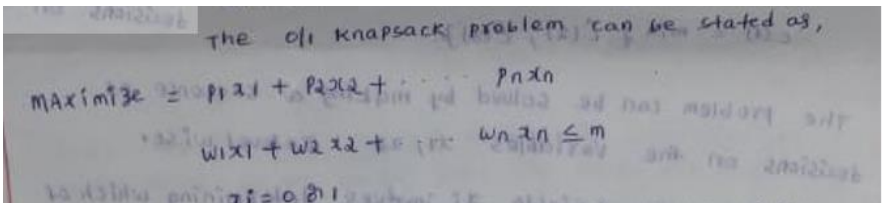
Example 2:- solve the following instance of TSP using LCBB

∞	7	3	12	8
3	∞	6	14	9
5	8	∞	6	18
9	3	5	∞	11
18	14	9	8	∞



#### 4.6.2 0/1 Knapsack problem

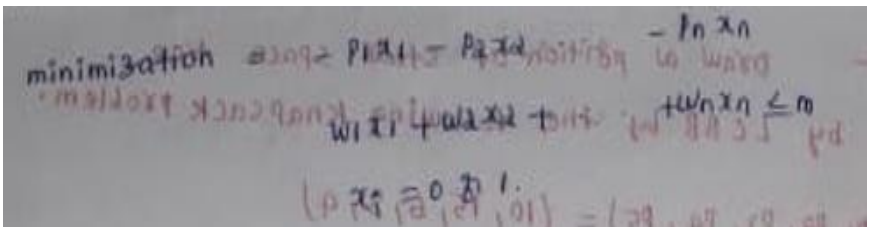
The 0/1 knapsack problem states that, there are  $n$  objects given and capacity of knapsack is  $M$ . Then select some objects to fill the knapsack in such way that it should not exceed the capacity of knapsack and maximum profit can be earned.



The 0/1 knapsack problem can be stated as,

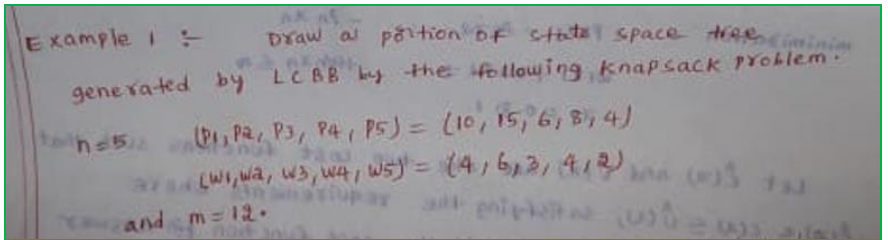
$$\text{Maximize } z = p_1x_1 + p_2x_2 + \dots + p_nx_n$$
$$w_1x_1 + w_2x_2 + \dots + w_nx_n \leq m$$
$$x_i = 0 \text{ or } 1$$

The branch and bound technique is used to find solution to the knapsack problem. But we can not directly apply the branch and bound technique to the knapsack problem because the branch and bound deals only the minimization problems. We modify the knapsack problem to the minimization problem. The modified problem is



minimization

$$z = p_1x_1 + p_2x_2 + \dots + p_nx_n$$
$$w_1x_1 + w_2x_2 + \dots + w_nx_n \leq m$$
$$x_i = 0 \text{ or } 1$$



1. Fractions are not allowed in calculation of upper bound.

2. Lower bound fractions are allowed

3.  $X_1=1$  means we should place first item.

$x_2=1, x_3=1, x_4=1, x_5=1$

$X_1=0$  can not consider

$X_2=0, x_3=0, x_4=0, x_5=0$

4. Both lower bound are same, then select the minimum of upper bound.

5. If both upper bounds are same then select  $x_1=0$  or  $x_2=0$  or  $x_3=0$  or  $x_4=0$  or  $x_5=0$

convert the profits to negative

$(P_1, P_2, P_3, P_4, P_5) = (-10, -15, -6, -8, -4)$

calculate the lower bound and upper bound for each node.

place the first item in bag i.e. 4 remaining weight is

$12 - 4 = 8$

place the second item in bag i.e. 6 remaining weight is

$8 - 6 = 2$

since fractions are not allowed in calculation of upper bound, so we cannot place the third and fourth items. place fifth item.

$\therefore$  profit learned =  $-10 - 15 - 4$   
 $= -29$  (upper bound)

To calculate the lower bound, place third item in a bag since fractions are allowed.

$\therefore$  Lower bound =  $-10 - 15 - \frac{3}{4} \times 6$

$\therefore \hat{U}(1) = -29$

$\hat{C}(1) = -29$

For node 2 ( $x_1=1$ ) means we should place 1st item in the bag.

$\hat{U}(2) = -10 - 15 - \frac{3}{4} \times 6 = -29$

$\hat{C}(2) = -10 - 15 - \frac{3}{4} \times 6 = -29$

For node 3 ( $x_1=0$ )

$\hat{U}(3) = -15 - 6 - 4 = -25$

$\hat{C}(3) = -15 - 6 - \frac{3}{4} \times 8 = -27$

select the minimum of lower bounds i.e.

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

$= \min \{ -29, -27 \}$

$= -29$

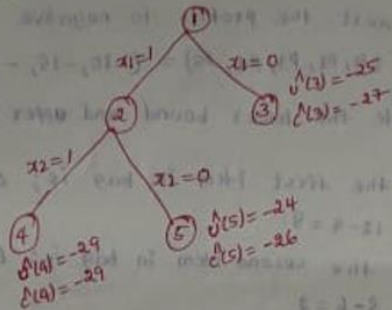
$= \hat{C}(2)$

$\therefore$  choose node 2.

First object is selected i.e.  $x_1=1$

(Detailed handwritten notes and diagrams are present in the image, including a tree structure and various calculations.)

consider the second variable  
to take the decision at  
second level.



F.O node 4 ( $x_2=1$ ):

$$\hat{\Delta}(4) = -10 - 15 - 4$$

$$\hat{C}(4) = -10 - 15 - \frac{2}{3} \times 6 = -29$$

F.O node 5 ( $x_2=0$ ):

$$\hat{\Delta}(5) = -10 - 6 - 8 = -24$$

$$\hat{C}(5) = -10 - 6 - 8 - \left(\frac{1}{2}\right) \times 4 = -26$$

$$\therefore \min \{ \hat{C}(4), \hat{C}(5) \} = \min \{ -29, -26 \} = -29$$

$\therefore$  Node 4 is selected.

$\therefore$  Second object is selected i.e.;

$x_2=1$

$PS = (1) \hat{u}$   
 $PS = (1) \hat{v}$

**6th node 6 ( $x_3=1$ )**

$\hat{u}(6) = -10 - 15 - 4 = -29$   
 $\hat{v}(6) = -10 - 15 - \frac{2}{3} \times 6 = -29$

**7th node 7 ( $x_3=0$ )**

$\hat{u}(7) = -10 - 15 - 4 = -29$   
 $\hat{v}(7) = -10 - 15 - \frac{2}{3} \times 0 = -29$

since the lower bound are same, select the minimum of upper bounds.  
 $\therefore \min \{ \hat{u}(6), \hat{u}(7) \}$   
 $\therefore \min \{ -29, -29 \}$   
 $= -29$   
 $\therefore$  Node 7 is selected.  
 $\therefore$  Third object is not selected i.e.,  $x_3=0$ .

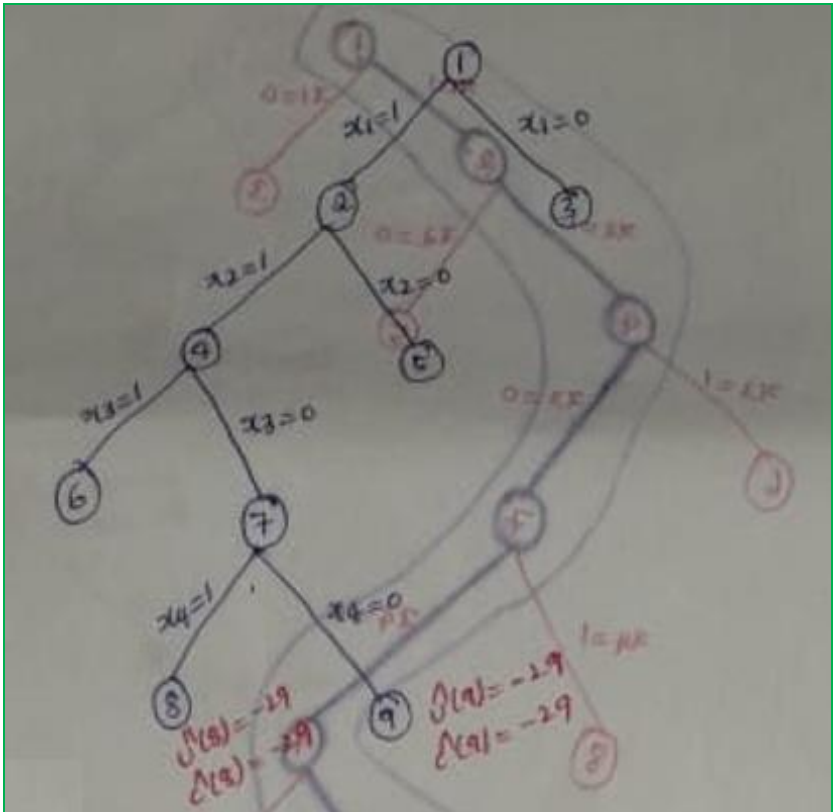
**8th node 8 ( $x_4=1$ )**

$\hat{u}(8) = -10 - 15 - 4 = -29$   
 $\hat{v}(8) = -10 - 15 - \left(\frac{2}{4}\right) \times 8 = -29$

**9th node 9 ( $x_4=0$ )**

$\hat{u}(9) = -10 - 15 - 4 = -29$   
 $\hat{v}(9) = -10 - 15 - \frac{2}{3} \times 0 = -29$

$\therefore \min \{ \hat{u}(8), \hat{u}(9) \}$   
 $= \min \{ -29, -29 \}$   
 $= -29$   
 $\therefore$  Node 9 is selected.  
 $\therefore$  Fourth object is not selected i.e.,  $x_4=0$ .





$\hat{J}(10) = -10 - 15 - 4$   
 $= -29$

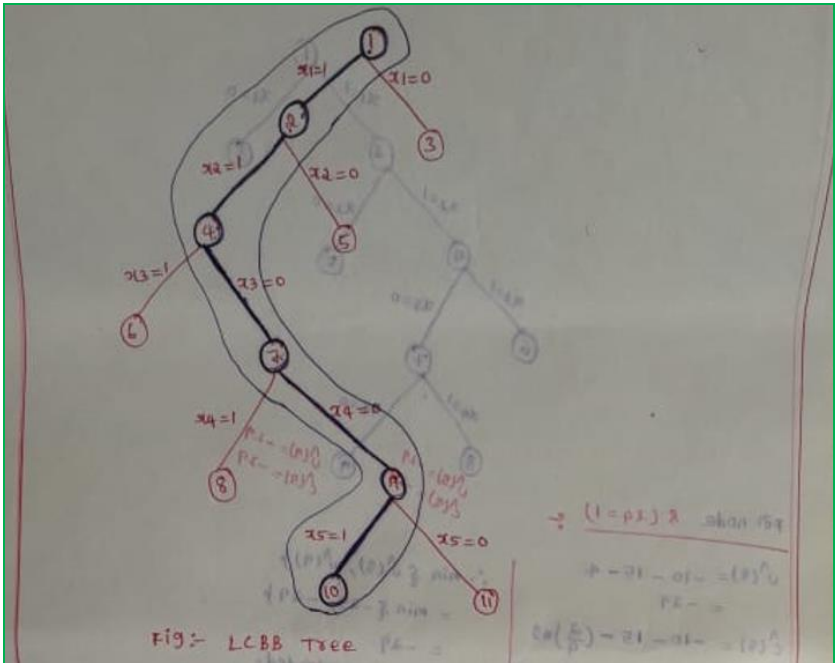
$\hat{C}(10) = -10 - 15 - \left(\frac{2}{3}\right) \times 6$   
 $= -10 - 15 - 4$   
 $= -29$

**for node 11 ( $x_5 = 0$ ) :-**  
 $\hat{J}(11) = -10 - 15$   
 $= -25$

$\hat{C}(11) = -10 - 15 - \frac{2}{3} \times 6$   
 $= -29$

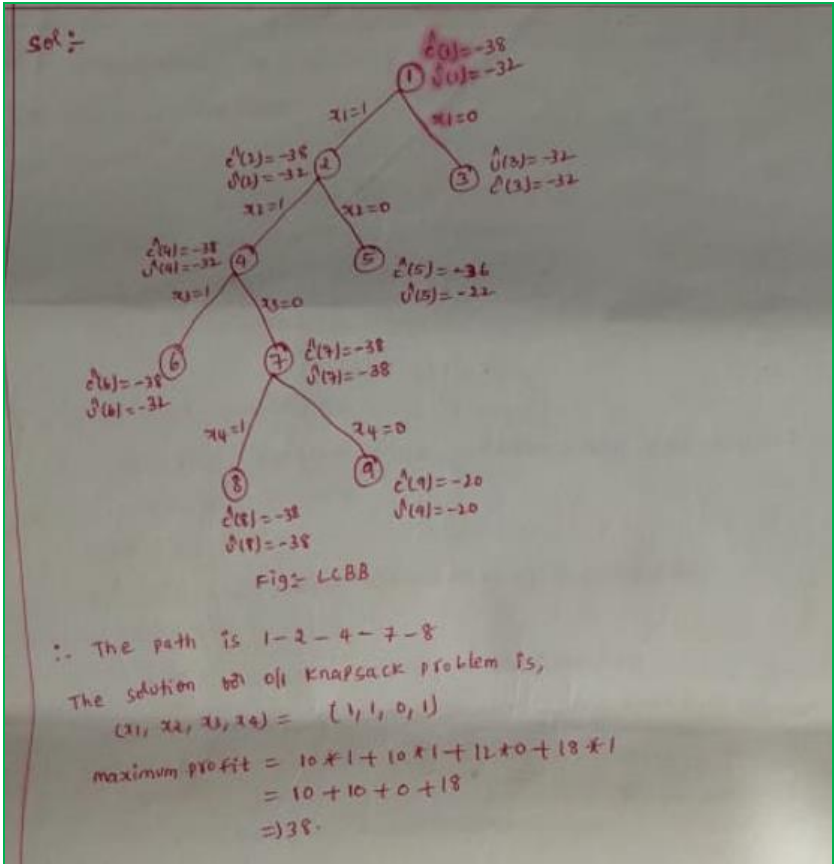
A search tree diagram is shown with nodes 1 through 11. Node 1 is the root with  $x_1 = 0$ . Node 2 is reached by  $x_1 = 1$  and node 3 by  $x_1 = 0$ . Node 4 is reached from node 2 by  $x_2 = 1$  and node 5 by  $x_2 = 0$ . Node 6 is reached from node 4 by  $x_3 = 1$  and node 7 by  $x_3 = 0$ . Node 8 is reached from node 6 by  $x_4 = 1$  and node 9 by  $x_4 = 0$ . Node 10 is reached from node 8 by  $x_5 = 1$  and node 11 by  $x_5 = 0$ .

$\min \{ \hat{J}(10), \hat{J}(11) \}$   
 $= \min \{ -29, -25 \}$   
 $= -29$   
 $\therefore$  select the node is 10.  
 $\therefore$  fifth object is selected  $x_5 = 1$ .



∴ The path is 1-2-4-7-9-10  
 The solution to knapsack problem is  
 $(x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 0, 1)$   
 Maximum profit  $= 10 \times 1 + 15 \times 1 + 6 \times 0 + 8 \times 0 + 4 \times 1$   
 $= 10 + 15 + 0 + 0 + 4$   
 $= 29$

Example 2:- Draw the portion of state space tree generated by LCBB for the knapsack instance  $\eta = 4$ .  
 $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$   
 $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$   
 $n = 15$ .



## UNIT-V

**NP Hard and NP Complete Problems:** Basic Concepts, Cook's theorem.

**NP Hard Graph Problems:** Clique Decision Problem (CDP), Chromatic Number Decision Problem (CNDP), Traveling Salesperson Decision Problem (TSP)

**NP Hard Scheduling Problems:** Scheduling Identical Processors, Job Shop Scheduling

### **5.1 NP Hard and NP Complete Problems:**

The problems are classified into two groups

#### **Polynomial Time**

1. The first group consists of the problems that can be solved in polynomial time by using deterministic algorithm

#### **Example:**

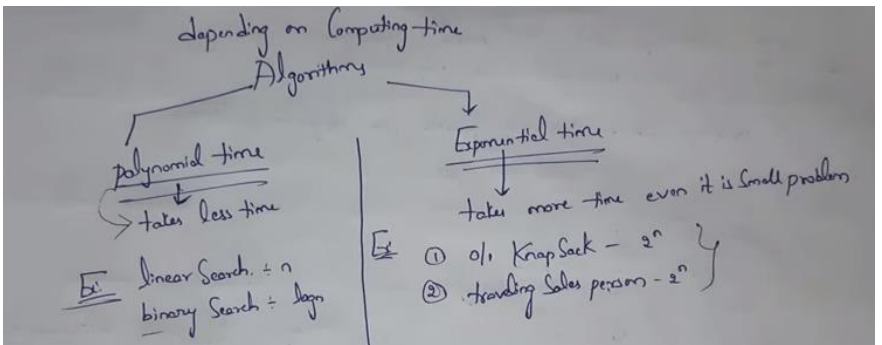
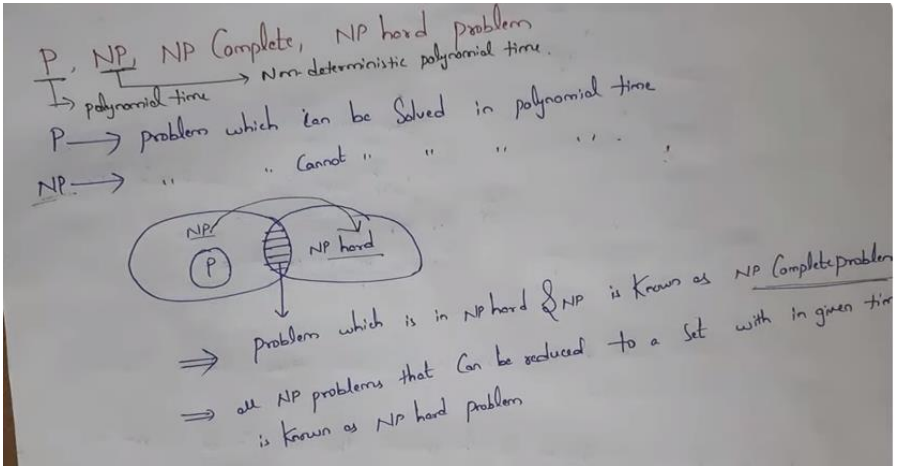
Searching of an element from an array – $O(\log n)$ .

Sorting of given  $n$  elements – $O(n \log n)$

All pairs shortest path problem- $O(n^3)$

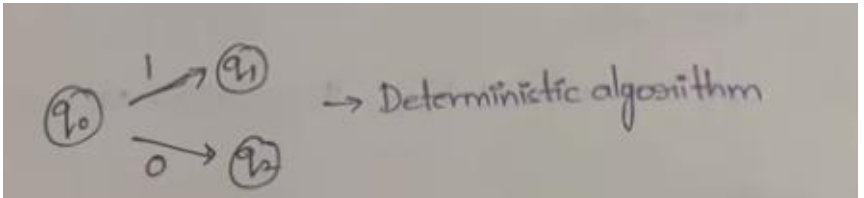
#### **Non-Polynomial Time**

2. The second group consists of the problems that can be solved in polynomial time by using non-deterministic algorithm.



## Deterministic Algorithm

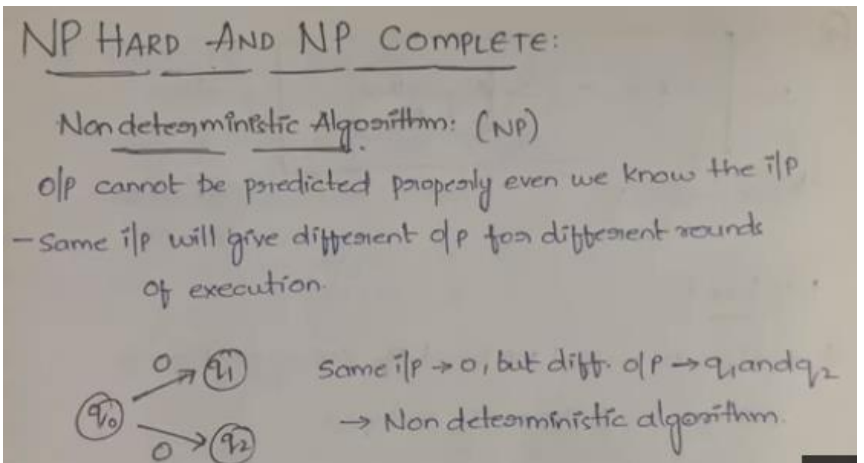
A deterministic algorithm is one where, given a particular input, the algorithm will always produce the same output and follow the same sequence of states.



## Non-Deterministic Algorithm

A non-deterministic algorithm is one where the same input can lead to multiple possible outcomes.

### Non deterministic Algorithm (NP):



- algorithm will take more than one path.
- $\therefore$  We cannot determine next step of execution.
- only approximate solutions, no accurate solutions.
- Ex: Monte Carlo Problem, Genetic Algorithm etc.

## Reduction or Reducible

Reduction: → Yes/No

Consider we have two decision Problems -  $P_1$  and  $P_2$

$P_1$  → Input ( $I_1$ )

→ Algorithm ( $A_1$ ) - unknown

$P_2$  → Input ( $I_2$ )

→ Algorithm ( $A_2$ ) - known

If  $P_1$  can be solved with help of  $A_2$ , then we convert  
 i/p  $I_1$  into  $I_2$  and find solution for  $P_2$   
 →  $P_1$  is reducible to  $P_2$

Problems which don't ~~to~~ be solved in polynomial time can be further classified into two types

NP Complete:  
 A problem that is NP Complete has the property that it can be solved in polynomial time if and only if all other NP Complete problems can be solved in polynomial time.

NP hard: If an NP hard problem can be solved in polynomial time, then all NP Complete problems can be solved in polynomial time.

## NP-Hard

NP-hard:

A problem is called NP hard if every problem in NP can be polynomially reduced

$A, B, C, D \rightarrow$  NP problems

if all these can be reduced to x, it is called NP hard

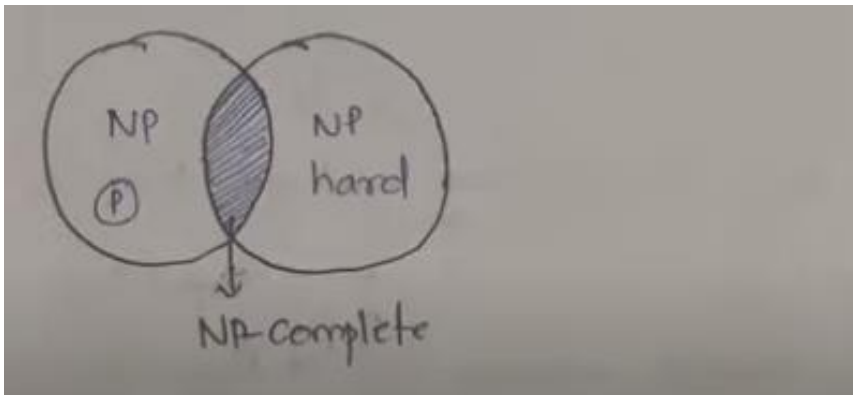
## NP-Complete

NP-Complete:

A problem is called NP complete if the problem P is

- P is in NP ✓

- P is in NP hard ✓





### **P-Class problems (Deterministic algorithms)**

Problems that can be solved in polynomial time are called P-class problems. Where 'p' stand for polynomial time.

Example:

1. Searching of key element among n number of elements-  $O(n)$
2. Sorting of 'n' elements-  $O(n^2)$
3. Addition of two matrices-  $O(n^2)$
4. Multiplication of matrices-  $O(n^3)$
5. All pairs shortest path problem-  $O(n^3)$

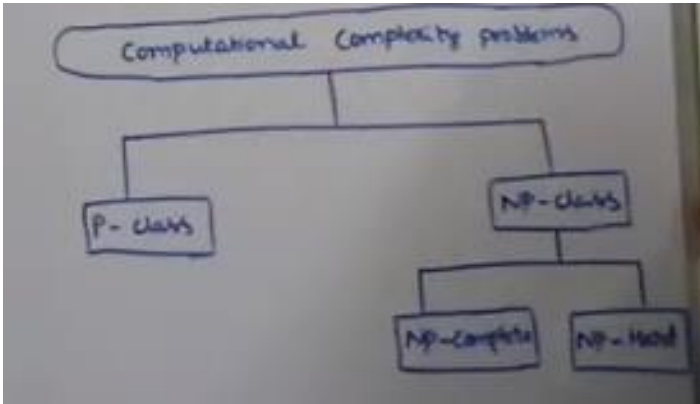
These problems can be solved by using deterministic algorithms.

### **NP-Class Problems (Non deterministic algorithm)**

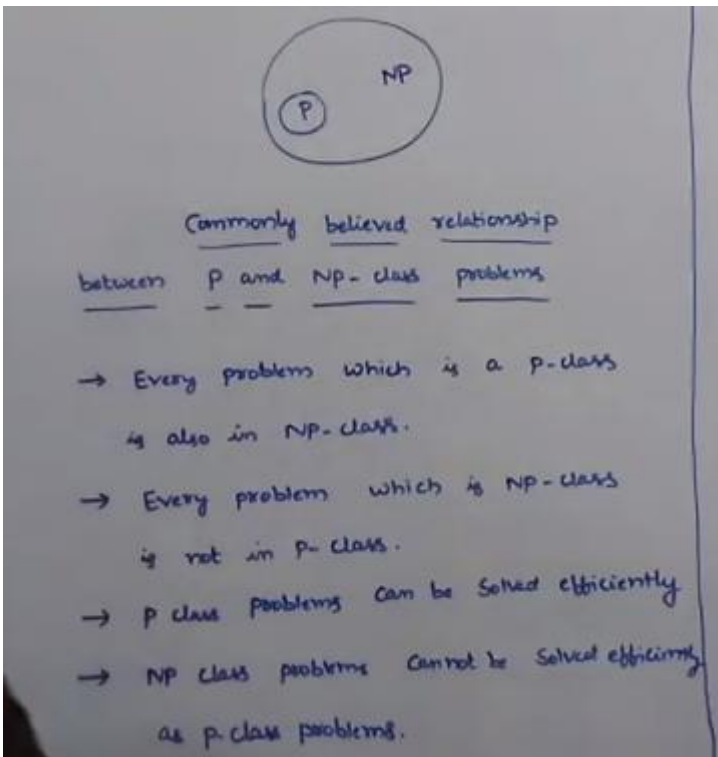
Problems that can be solved in polynomial time by using Non-deterministic algorithms are called NP-class problems. Where 'NP' stand for Non-Polynomial time.

Example:

1. Travelling salesperson problem
2. 0/1 knapsack problem
3. Graph coloring problem
4. Hamiltonian cycle problem



## Relationship between P and NP-class problem



## **5.2 Cook's theorem.**

Cook's theorem states that the problems of satisfiability and determining if  $P=NP$  are equivalent. It proves this by showing that if satisfiability is in  $P$  (can be solved in polynomial time), then any problem in  $NP$  (can be solved by a non-deterministic Turing machine in polynomial time) can also be solved in  $P$ , meaning  $P=NP$ . Conversely, if  $P=NP$ , then satisfiability must also be in  $P$ .

### **Statement**

The satisfiability problem (SAT) is  $NP$  complete.

In computational complexity theory, the Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is  $NP$ -complete. That is, it is in  $NP$ , and any problem in  $NP$  can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.

### **Boolean Satisfiability Problem**

Boolean Satisfiability or simply SAT is the problem of determining if a Boolean formula is satisfiable or unsatisfiable.

**Satisfiable:** If the Boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the formula is satisfiable.

**Unsatisfiable :** If it is not possible to assign such values, then we say that the formula is unsatisfiable.

**Examples:**

- $F = A \wedge \bar{B}$ , is satisfiable, because A = TRUE and B = FALSE makes F = TRUE.
- $G = A \wedge \bar{A}$ , is unsatisfiable, because:

A	$\bar{A}$	G
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE

**Note:** Boolean satisfiability problem is NP-complete

## Cooks Theorem

- Stephen Cook introduced the notion of NP-Complete Problems.
  - This makes the problem “P = NP ?” much more interesting to study.
  - L is **NP-Complete** if  $L \in NP$  then for all other  $L' \in NP, L' \propto L$
  - L is **NP-Hard** if for all other  $L' \in NP, L' \propto L$
  - If an NP-complete problem can be solved in polynomial time then all problems in NP can be solved in polynomial time.
  - If a problem in NP cannot be solved in polynomial time then all problems in NP-complete cannot be solved in polynomial time.
  - Note that an NP-complete problem is one of those hardest problems in NP.
- Lemma :  
 If L1 and L2 belong to NP,  
 L1 is NP-complete, and  $L1 \propto L2$   
 then L2 is NP-complete.  
*i.e.  $L1, L2 \in NP$  and for all other  $L' \in NP, L' \propto L1$  and  $L1 \propto L2 \rightarrow L' \propto L2$*

# Cooks Theorem and Satisfiability

- SATISFIABILITY is NP-Complete. (The first NP-Complete problem)
  - Instance : Given a set of variables, U, and a collection of clauses, C, over U.
  - Question : Is there a truth assignment for U that satisfies all clauses in C?
- CNF-Satisfiability (CNF – Conjunctive Normal Form)
  - because the expression is in (the product of sums).
- Example:

“ $\neg x_i$ ” = “not  $x_i$ ”    “OR” = “logical or”    “AND” = “logical and”     $U = \{x_1, x_2\}$

$$\begin{aligned} C_1 &= \{(x_1, \neg x_2), (\neg x_1, x_2)\} \\ &= (x_1 \text{ OR } \neg x_2) \text{ AND } (\neg x_1 \text{ OR } x_2) \\ &\quad \text{if } x_1 = x_2 = \text{True} \rightarrow C_1 = \text{True} \end{aligned}$$

$$C_2 = (x_1, x_2) (x_1, \neg x_2) (\neg x_1) \rightarrow \text{not satisfiable}$$

## 5.3 NP Hard Graph Problems:

### 5.3.1 Clique Decision Problem (CDP)

### 5.3.2 Chromatic Number Decision Problem (CNDP)

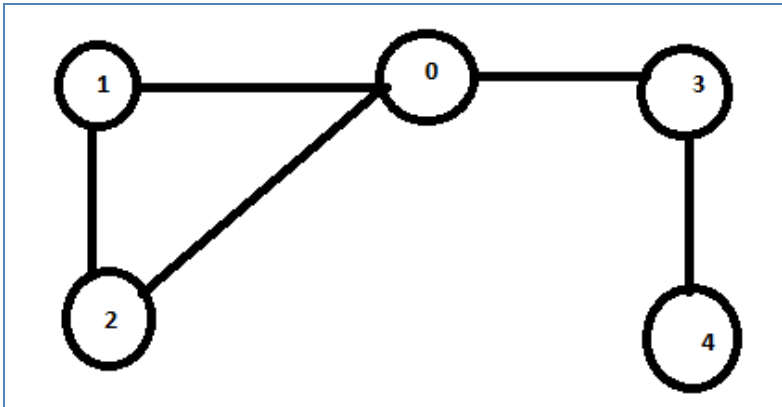
### 5.3.3 Traveling Salesperson Decision Problem (TSP).

#### 5.3.1 Clique Decision Problem (CDP)

A clique is a sub-graph of graph such that all vertices in sub-graph are completely connected with each other.

The max-clique problem is the computational problem of finding maximum clique of the graph.

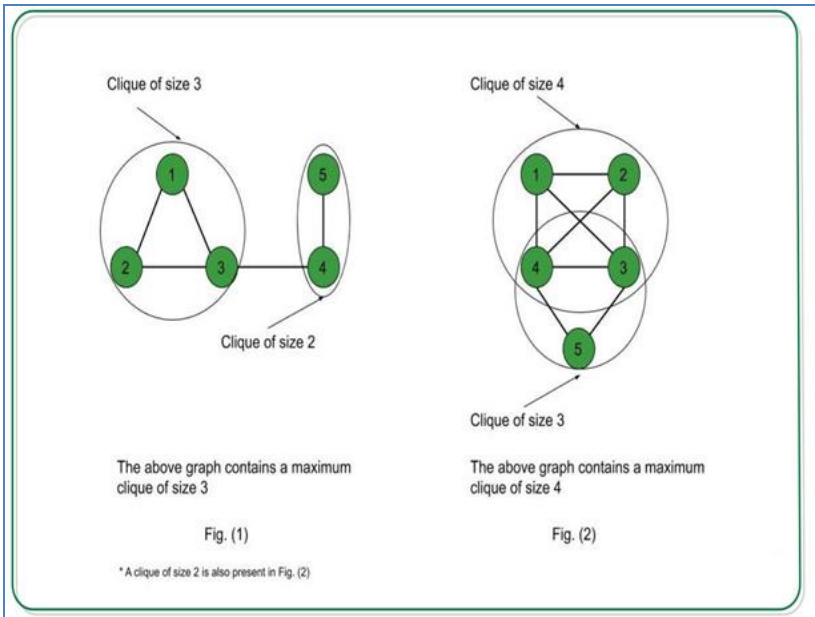
#### Example



This graph can be divided into two cliques

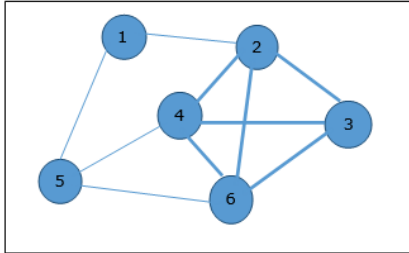
One clique contains  $\{0,1,2\}$

Other clique contains  $\{3,4\}$



## Example

Take a look at the following graph. Here, the sub-graph containing vertices 2, 3, 4 and 6 forms a complete graph. Hence, this sub-graph is a **clique**. As this is the maximum complete sub-graph of the provided graph, it's a **4-Clique**.



### 5.3.2 Chromatic Number Decision Problem (CNDP)

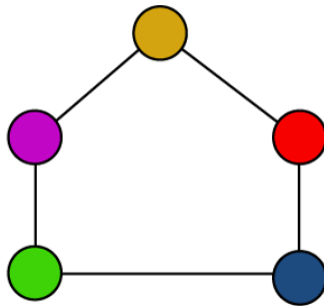
#### Graph Coloring Algorithm

- There exists **no efficient algorithm** for coloring a graph with minimum number of colors.
- Graph Coloring is a **NP complete** problem.
- However, a following **greedy algorithm** is known for finding the **chromatic number** of any given graph.
- Graph coloring can be described as a **process of assigning colors to the vertices of a graph**.
- In this, **the same color should not be used** to fill the two adjacent vertices.
- We can also call graph coloring as **Vertex Coloring**.
- In graph coloring, we have to take care that a graph must not contain any edge whose end vertices are colored by the same color.
- This type of graph is known as the **Properly colored graph**.

Graph coloring can be described as a process of assigning colors to the vertices of a graph. In this, the same color should not be used to fill the two adjacent vertices. We can also call graph coloring as Vertex Coloring. In graph coloring, we have to take care that a graph must not contain any edge whose end vertices are colored by the same color. This type of graph is known as the properly colored graph.

### **Example of Graph coloring**

In this graph, we are showing the properly colored graph, which is described as follows:



The above graph contains some points, which are described as follows:

- The same color cannot be used to color the two adjacent vertices.
- Hence, we can call it as a properly colored graph.



## Applications of Graph coloring

There are various applications of graph coloring. Some of their important applications are described as follows:

- Assignment
- Map coloring
- Scheduling the tasks
- Sudoku
- Prepare time table
- Conflict resolution

## Chromatic Number decision problem

### Chromatic Number

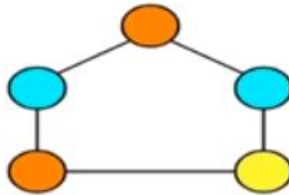
- Chromatic Number is the **minimum number of colors** required to properly color any graph.

OR

- Chromatic Number is the minimum number of colors required to color any graph such that no two adjacent vertices of it are assigned the same color.

## Chromatic Number Example-

- Consider the following graph-



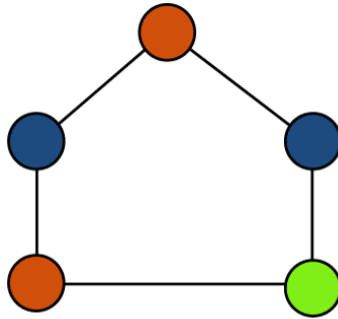
**Chromatic Number = 3**

In this graph,

- No two adjacent vertices are colored with the same color.
- Minimum number of colors required to properly color the vertices = 3.
- Therefore, Chromatic number of this graph = 3.
- We can not properly color this graph with less than 3 colors.

### **Example of Chromatic number:**

To understand the chromatic number, we will consider a graph, which is described as follows:



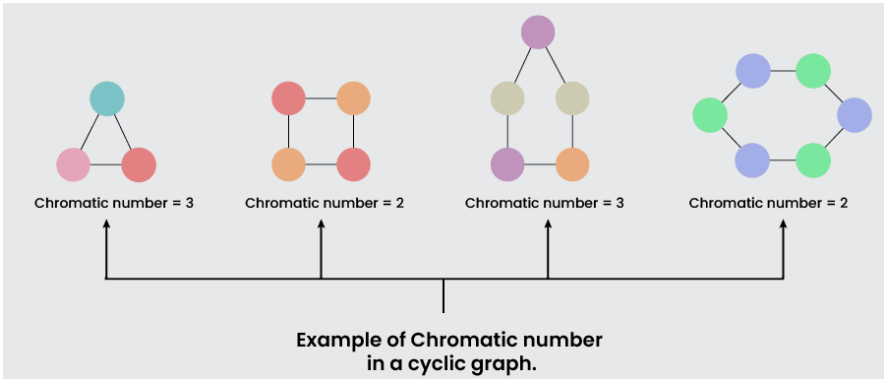
The above graph contains some points, which are described as follows:

The same color is not used to color the two adjacent vertices.

The minimum number of colors of this graph is 3, which is needed to properly color the vertices.

Hence, in this graph, the chromatic number = 3

If we want to properly color this graph, in this case, we are required at least 3 colors.



### 5.3.3 Traveling Salesperson Decision Problem (TSP).

The NP-hard Traveling Salesman Problem (TSP) asks to find the shortest route that visits all vertices in the graph. To be precise, the TSP is the shortest tour that visits all vertices and returns back to the start.

## **5.4 NP Hard Scheduling Problems:**

5.4.1 Scheduling Identical Processors

5.4.2 Job Shop Scheduling

### **5.4.1 Scheduling Identical Processors**

### **5.4.2 Job Shop Scheduling**

Job Shop Scheduling Problem (JSSP), which aims to schedule several jobs over some machines in which each job has a unique machine route, is one of the NP-hard optimization problems researched over decades for finding optimal sequences over machines.

[https://optimization.cbe.cornell.edu/index.php?title=Job\\_shop\\_scheduling](https://optimization.cbe.cornell.edu/index.php?title=Job_shop_scheduling)