

## Unit-I

**INTRODUCTION** - Algorithm definition, Pseudo code Specifications, Performance Analysis-Space Complexity, Time Complexity, Asymptotic Notations-Big-Oh, Omega, Theta, little-oh, Recurrences- Iteration Method, Master's Method. Disjoint set Operations' and algorithms-Find, Union.

**DIVIDE AND CONQUER** - General Method, Binary Search, Finding Maximum and Minimum, Merge Sort, Quick sort, Strassen's Matrix Multiplication.

### **Recurrences- Iteration method**

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs.

To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

We solve recurrence relations using the iteration method. In this method, we keep substituting the smaller terms again and again until we reach the base condition. Thus the base term can be replaced by its value, and we get the value of the expression.

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

**Example1:** Consider the Recurrence

$$\begin{aligned} T(n) &= 1 && \text{if } n=1 \\ &= 2T(n-1) && \text{if } n>1 \end{aligned}$$

**Solution:**

$$\begin{aligned} T(n) &= 2T(n-1) \\ &= 2[2T(n-2)] = 2^2T(n-2) \\ &= 4[2T(n-3)] = 2^3T(n-3) \\ &= 8[2T(n-4)] = 2^4T(n-4) \quad (\text{Eq.1}) \end{aligned}$$

Repeat the procedure for i times

$$T(n) = 2^i T(n-i)$$

Put  $n-i=1$  or  $i= n-1$  in (Eq.1)

$$\begin{aligned} T(n) &= 2^{n-1} T(1) \\ &= 2^{n-1} \cdot 1 \quad \{T(1)=1 \text{ .....given}\} \\ &= 2^{n-1} \end{aligned}$$

**Masters Method:**

The Master Method is used for solving the following types of recurrence

$T(n) = a T\left(\frac{n}{b}\right) + f(n)$  with  $a \geq 1$  and  $b \geq 1$  be constant &  $f(n)$  be a function and  $\frac{n}{b}$  can be interpreted as

Let  $T(n)$  is defined on non-negative integers by the recurrence.

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

In the function to the analysis of a recursive algorithm, the constants and function take on the following significance:

- $n$  is the size of the problem.
- $a$  is the number of subproblems in the recursion.
- $n/b$  is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- $f(n)$  is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.
- It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \epsilon}) \\ \Theta(n^{\log_b a} \log n) & f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \epsilon}) \text{ AND } \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \begin{matrix} \epsilon > 0 \\ c < 1 \end{matrix}$$

$$T(n) = 8 T\left(\frac{n}{2}\right) + 1000n^2 \quad \text{apply master theorem on it.}$$

**Solution:**

$$\text{Compare } T(n) = 8 T\left(\frac{n}{2}\right) + 1000n^2 \quad \text{with}$$

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \text{ with } a \geq 1 \text{ and } b > 1$$

$$a = 8, b = 2, f(n) = 1000 n^2, \log_b a = \log_2 8 = 3$$

$$\text{Put all the values in: } f(n) = O(n^{\log_b a - \epsilon})$$

$$1000 n^2 = O(n^{3-\epsilon})$$

$$\text{If we choose } \epsilon = 1, \text{ we get: } 1000 n^2 = O(n^{3-1}) = O(n^2)$$

-----

### Disjoint set operations:

Disjoint sets in mathematics are two sets that don't have any element in common. Sets can contain any number of elements, and those elements can be of any type.

We can have a set of cars, a set of integers, a set of colors, etc. Sets also have various operations that can be performed on them, such as union, intersection, difference, etc.

A disjoint set data structure is a data structure that stores a list of disjoint sets. In other words, this data structure divides a set into multiple subsets - such that no 2 subsets contain any common element. They are also called union-find data structures or merge-find sets.

Ex:

If the initial set is [1,2,3,4,5,6,7,8].

A Disjoint Set Data Structure might partition it as - [(1,2,4), (3,5), (6,8),(7)].

This contains all of the elements of the original set, and no 2 subsets have any element in common.

The following partitions would be invalid:

[(1,2,3),(3,4,5),(6,8),(7)] - Invalid because 3 occurs in 2 subsets.

[(1,2,3),(5,6),(7,8)] -invalid as 4 is missing.

### Representative Member

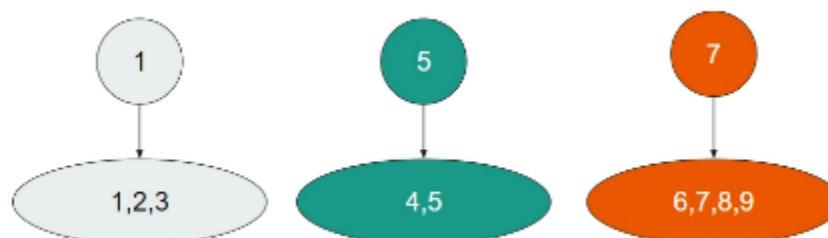
Each subset in a Disjoint Set Data Structure has a **representative member**. More commonly, we call this the **parent**. This is simply the member *responsible for identifying* the subset.

Ex:

Suppose this is the partitioning of sets in our Disjoint Set Data structure.



We wish to identify each subset, so we assign each subset a value by which we can identify it. The easiest way to do this is to choose a **member** of the subset and make it the **representative member**. This representative member will become the **parent** of all values in the subset.



Here, we have assigned representative members to each subset. Thus, if we wish to know which subset **3** is a part of, we can simply say - **3 is a part of the subset** with representative member 1. More simply, we can say that **the parent of 3 is 1**.

Here, 1 is its own parent.

## Disjoint Set Operations in DAA

There are 2 major disjoint set operations in DAA:

1. **Union/Merge**
2. **Find**

### Union/Merge:

This is used to merge 2 subsets into a single subset.

Ex:

$s1 = \{1, 2, 3, 4\}$

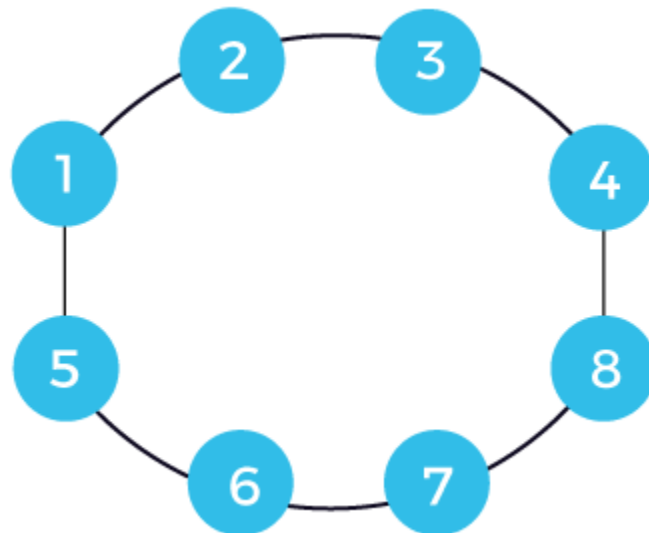
$s2 = \{5, 6, 7, 8\}$

Suppose we want to perform the union operation on these two sets. First, we have to check whether the elements on which we are performing the union operation belong to different or same sets.

If they belong to the different sets, then we can perform the union operation; otherwise, not.

Ex:

We want to perform the union operation between 4 and 8. Since 4 and 8 belong to different sets, so we apply the union operation. Once the union operation is performed, the edge will be added between the 4 and 8 shown as below



$s1 \cup s2 = \{1, 2, 3, 4, 5, 6, 7, 8\}$

Suppose we add one more edge between 1 and 5. Now the final set can be represented as:

$s3 = \{1, 2, 3, 4, 5, 6, 7, 8\}$

### Find:

This is used to find which subset a particular value belongs to.

### Find

The find operation helps us find the parent of a node. As we saw above, the direct parent of a node might not be its actual (logical) parent. E.g., the logical parent of Vrindavan should be Delhi in the above example. But its direct parent is Agra.

So, how do we find the actual parent?

The find operation helps us to find the actual parent.

In pseudo code, the find operation looks like this:

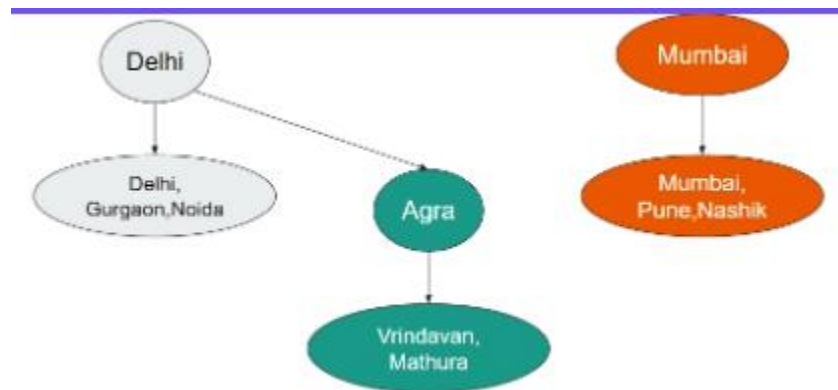
```
find(node):
```

```
if (parent(node)==node) return node;
```

```
else return find(parent(node));
```

We don't return the direct parent of the node. We keep going up the tree of parents until we find a node that is its own parent.

Ex:



Suppose we have to find the parent of Mathura.

1. Check the direct parent of Mathura. It's Agra. Is Agra == Mathura?  
The answer is false. So, now we call the find operation on Agra.
2. Check the direct parent of Agra. It's Delhi. Is Delhi ==Agra?  
The answer is false. So now we call the find operation on Delhi.
3. Check the direct parent of Delhi. It's Delhi. Is Delhi==Delhi.  
The answer is true! So, our final answer for the parent of Mathura is Delhi.

This way, we keep going "up" the tree until we reach a root element. A root element is a node with its own parent - in our case, Delhi.

## Divide and Conquer:

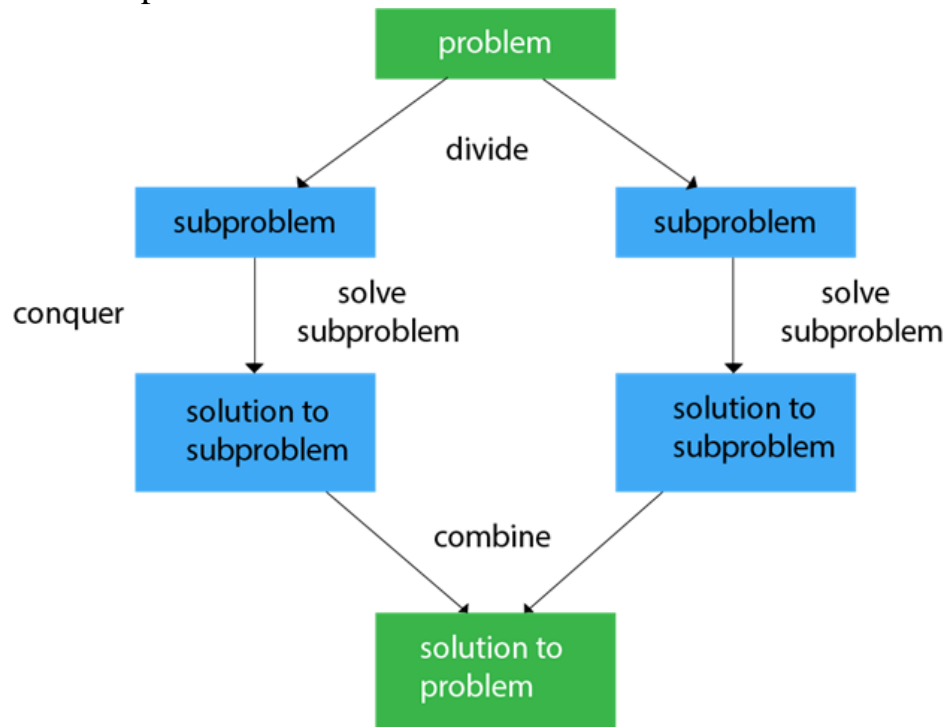
### General Method:-

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of sub problems.
2. **Conquer:** Solve every sub problem individually, recursively.

3. **Combine:** Put together the solutions of the sub problems to get the solution to the whole problem.



### Pseudo code Representation of Divide and conquer rule for problem “P”

```

Algorithm DAndC(P)
{
  if small(P) then return S(P)
  else{
    divide P into smaller instances P1,P2,P3...Pk;
    apply DAndC to each of these subprograms; // means DAndC(P1),
    DAndC(P2).....DAndC(Pk)
    return combine(DAndC(P1), DAndC(P2)..... DAndC(Pk));
  }
}

```

**//P □ Problem**

**//Here small(P) □ Boolean value function. If it is true, then the function S is**

**//invoked**

### Time Complexity of DAnd C algorithm:

$$\begin{aligned}
 T(n) &= T(1) && \text{if } n=1 \\
 &= aT(n/b)+f(n) && \text{if } n>1
 \end{aligned}$$

a,b are constants.

This is called the **general divide and-conquer recurrence**.

### Example for GENERAL METHOD:

As an example, let us consider the problem of computing the sum of n numbers

$a_0, \dots, a_{n-1}$ .

If  $n > 1$ , we can divide the problem into two instances of the same problem. They are sum of the first  $\lfloor n/2 \rfloor$  numbers

Compute the sum of the 1<sup>st</sup>  $\lfloor n/2 \rfloor$  numbers, and then compute the sum of another  $n/2$  numbers. Combine the answers of two  $n/2$  numbers sum.

i.e.,

$$a_0 + \dots + a_{n-1} = (a_0 + \dots + a_{n/2}) + (a_{n/2} + \dots + a_{n-1})$$

Assuming that size  $n$  is a power of  $b$ , to simplify our analysis, we get the following recurrence for the running time  $T(n)$ .

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

This is called the general **divide and-conquer recurrence**.

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

### Advantages of DAndC:

- The time spent on executing the problem using DAndC is smaller than other method. This technique is ideally suited for parallel computation.
- This approach provides an efficient algorithm in computer science.

### Binary Search:

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to  $O(\log N)$ .

Binary Search										
	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
23 > 16 take 2 <sup>nd</sup> half	L=0	1	2	3	M=4	5	6	7	8	H=9
	2	5	8	12	16	23	38	56	72	91
23 < 56 take 1 <sup>st</sup> half	0	1	2	3	4	L=5	6	M=7	8	H=9
	2	5	8	12	16	23	38	56	72	91
Found 23, Return 5	0	1	2	3	4	L=5, M=5	H=6	7	8	9
	2	5	8	12	16	23	38	56	72	91

## Binary Search Algorithm:

*In this algorithm,*

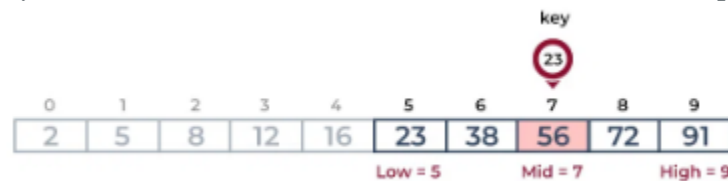
- *Divide the search space into two halves by finding the middle index “mid”.*
- *Compare the middle element of the search space with the key.*
- *If the key is found at middle element, the process is terminated.*
- *If the key is not found at middle element, choose which half will be used as the next search space.*
  - *If the key is smaller than the middle element, then the left side is used for next search.*
  - *If the key is larger than the middle element, then the right side is used for next search.*
- *This process is continued until the key is found or the total search space is exhausted.*

*Consider an array  $arr[] = \{2, 5, 8, 12, 16, 23, 38, 56, 72, 91\}$ , and the target = 23. First Step: Calculate the mid and compare the mid element with the key. If the key is less than mid element, move to left and if it is greater than the mid then move search space to the right.*

- *Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.*



- *Key is less than the current mid 56. The search space moves to the left.*



**Second Step:** *If the key matches the value of the mid element, the element is found and stop search.*





#### Advantages of Binary Search:

- Binary search is faster than linear search, especially for large arrays.
- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

#### Complexity

- Time Complexity:
  - Best Case:  $O(1)$
  - Average Case:  $O(\log N)$
  - Worst Case:  $O(\log N)$
- Auxiliary Space:  $O(1)$ , If the recursive call stack is considered then the auxiliary space will be  $O(\log N)$ .

```

Algorithm BinSrch( $a, i, l, x$ )
// Given an array  $a[i : l]$  of elements in nondecreasing
// order,  $1 \leq i \leq l$ , determine whether  $x$  is present, and
// if so, return  $j$  such that  $x = a[j]$ ; else return 0.
{
    if ( $l = i$ ) then // If Small( $P$ )
    {
        if ( $x = a[i]$ ) then return  $i$ ;
        else return 0;
    }
    else
    { // Reduce  $P$  into a smaller subproblem.
         $mid := \lfloor (i + l) / 2 \rfloor$ ;
        if ( $x = a[mid]$ ) then return  $mid$ ;
        else if ( $x < a[mid]$ ) then
            return BinSrch( $a, i, mid - 1, x$ );
        else return BinSrch( $a, mid + 1, l, x$ );
    }
}

```

-----

### Finding Maximum and Minimum:

Let us consider another simple problem that can be solved by the divide and-Conquer technique. The problem is to find the maximum and minimum items in a set of  $n$  elements.

In analyzing the time complexity of this algorithm, we once again concentrate on the number of element comparisons. The justification for this is that, the frequency count for other operations in this algorithm is of the same order as that for element comparisons. More importantly, when the elements in  $a[1:n]$  are polynomials, vectors, very large numbers, or strings of characters, the cost of an element comparison is much higher than the cost of the other operations. Hence the time is determined mainly by the total cost of the element comparisons.

### Algorithm:

```

Algorithm StraightMaxMin( $a, n, max, min$ )
// Set  $max$  to the maximum and  $min$  to the minimum of  $a[1 : n]$ .
{
     $max := min := a[1]$ ;
    for  $i := 2$  to  $n$  do
    {
        if ( $a[i] > max$ ) then  $max := a[i]$ ;
        if ( $a[i] < min$ ) then  $min := a[i]$ ;
    }
}

```

-----

### Merge sort:

- Given a sequence of  $n$  elements (also called keys)  $a[1], a[2], \dots, a[n/2]$  and  $a[n/2+1], \dots, a[n]$ .
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of  $n$  elements.
- By using divide and conquer strategy in which the splitting is into two equal sized sets and the combining is the operation is the merging of two sorted sets into one.
- Merge sort algorithm describes this process very succinctly using recursion and a function Merge Algorithm which merges two sorted sets.
- Before executing Mergeset, the  $n$  elements should be placed in  $a[1:n]$ .
- Then Mergesort(1,n) causes the keys to be rearranged into nondecreasing order in  $a$ .

Ex:

Consider the array of ten elements  $a[1:10] = (310, 285, 179,$

$652, 351, 423, 861, 254, 450, 520)$ .

Algorithm Merge Sort begins by splitting  $a[ ]$  into two subarrays each of size five ( $a[1:5]$  and  $a[6:10]$ ). The elements in  $a[1:5]$  are then split into two subarrays of size three ( $a[1:3]$ ) and two ( $a[4:5]$ ). Then the items in  $a[1:3]$  are split into subarrays of size two ( $a[1:2]$ ) and one ( $a[3:3]$ ). The two values in  $a[1:2]$  are split a final time into one-element subarrays, and now the merging begins.

A record of the subarrays is implicitly maintained by the recursive mechanism.

Pictorially the file can now be viewed as

$(310 | 285 | 179 | 652, 351 | 423, 861, 254, 450, 520)$

Where vertical bars indicate the boundaries of subarrays. Elements  $a[1]$  and

$a[2]$  are merged to yield

$(285, 310 | 179 | 652, 351 | 423, 861, 254, 450, 520)$

Then  $a[3]$  is merged with  $a[1:2]$  and

$(179, 285, 310 | 652, 351 | 423, 861, 254, 450, 520)$

is produced.

Next, elements  $a[4]$  and  $a[5]$  are merged:

$(179, 285, 310 | 351, 652 | 423, 861, 254, 450, 520)$

and then  $a[1:3]$  and  $a[4:5]$ :

$(179, 285, 310, 351, 652 | 423, 861, 254, 450, 520)$

At this point the algorithm has returned to the first invocation of MergeSort and is about to process the second recursive call. Repeated recursive calls are invoked producing the following subarrays:

(179,285,310,351,652| 423 | 861| 254 | 450,520)

Elements a[6] and a[7] are merged. Then a[8] is merged with a[6:7]:

(179,285,310,351,652| 254, 423,861| 450,520)

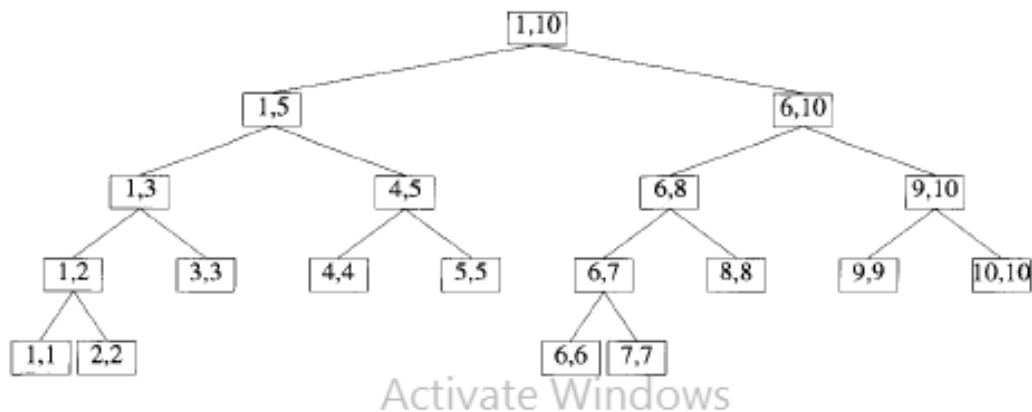
Next a[9] and a[10] are merged, and then a[8] and a[9:10]:

(179,285,310,351,652| 254, 423,450,520,861)

At this point there are two sorted subarrays and the final merge produces the fully sorted result

(179,254, 285,310,351,423,450,520,652,861)

The below figure represents the tree of calls if MergeSort(1,10)



Algorithm MergeSort(low, high)

// a[low :high] is a global array to be sorted.

// Small(P) is true if there is only one element

// to sort. In this case the list is already sorted.

if (low < high) then // If there are more than one element

{

// Divide P into subproblems.

// Find where to split the set.

mid := (low + high) / 2; // Solve the subproblems.

MergeSort(low, mid);

MergeSort(mid + 1, high);

// Combine the solutions.

Merge(low, mid, high);

}

```

}
Algorithm Merge(low,mid,high)
// a[low :high] is a global array containing two sorted
// subsets [low :mid] and in a[mid+ 1:high]. The goal
// is to merge these two sets into a single set residing
// in a[low :high]. b[ ] is an auxiliary global array.
{
h :=low; i :=low; j :=mid+ 1;
while ((h <mid) and (j < high)) do
{
if (a[h] <a[j])then
{
b[i] :=a[h];h:=h + 1;
}else
{
b[i] :=a[j];  j:=j+ 1;
}
i :=i + 1;
}
if (h >mid) then
for k :=j to high do
{
b[i] :=a[k]; i :=i + 1;
}
else
{
for k :=h to mid do
{
b[i] :=a[k]; i :=i + 1;
}
for k :=low to high do
a[k] :=b[k];
}
}

```

If the time for the merging operation is proportional to  $n$ , then the computing time for merge sort is described by the recurrence relation

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When  $n$  is a power of 2,  $n = 2^k$ , we can solve this equation by successive substitutions:

$$\begin{aligned}
 T(n) &= 2(2T(n/4) + cn/2) + cn \\
 &= 4T(n/4) + 2cn \\
 &= 4(2T(n/8) + cn/4) + 2cn \\
 &\vdots \\
 &= 2^k T(1) + kcn \\
 &= an + cn \log n
 \end{aligned}$$

It is easy to see that if  $2^k < n \leq 2^{k+1}$ , then  $T(n) \leq T(2^{k+1})$ . Therefore

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

The time complexity of the merge sort is

Best Case :  $O(n \log n)$

Average Case :  $O(n \log n)$

Worst Case :  $O(n \log n)$

### Quick Sort:

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

#### Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

**Step 1** – Choose the highest index value has pivot

**Step 2** – Take two variables to point left and right of the list excluding pivot

**Step 3** – left points to the low index

**Step 4** – right points to the high

**Step 5** – while value at left is less than pivot move right

**Step 6** – while value at right is greater than pivot move left

**Step 7** – if both step 5 and step 6 does not match swap left and right

**Step 8** – if  $\text{left} \geq \text{right}$ , the point where they met is new pivot.

#### **Quick Sort Pseudocode:**

To get more into it, let see the pseudocode for quick sort algorithm.

```

procedure quickSort(left, right)

    if right-left <= 0
        return
    else
        pivot = A[right]
        partition = partitionFunc(left, right, pivot)
        quickSort(left,partition-1)
        quickSort(partition+1,right)
    end if
end procedure

```

### **Complexity:**

Worst case complexity-  $O(n^2)$ .  
 Average case complexity-  $O(n \log n)$ .  
 Best case complexity-  $O(n \log n)$ .

-----

### **Strassen's Matrix Multiplication:**

**Strassen** in 1969 gave an overview on how we can find the multiplication of two **2\*2 dimension matrices by the brute-force algorithm**. But by using the divide and conquer technique the overall complexity for the multiplication of two matrices has been reduced. This happens by decreasing the total number of multiplications performed at the expense of a slight increase in the number of additions.

**Strassen** has used some formulas for multiplying the two 2\*2 dimension matrices where the number of multiplications is seven, additions and subtractions are is eighteen, and in brute force algorithm, there is eight number of multiplications and four addition.

When the order **n** of matrix reaches infinity, the utility of Strassen's formula is shown by its asymptotic superiority. For example, let us consider two matrices **A** and **B** of **n\*n** dimension, where **n** is a power of two. It can be observed that we can have four sub matrices of order **n/2 \* n/2** from **A**, **B**, and their product **C** where **C** is the resultant matrix of **A** and **B**.

## **The procedure of Strassen's matrix multiplication**

Here is the procedure:

1. Divide a matrix of the order of  $2 \times 2$  recursively until we get the matrix of order  $2 \times 2$ .
2. To carry out the multiplication of the  $2 \times 2$  matrix, use the previous set of formulas.
3. Subtraction is also performed within these eight multiplications and four additions.
4. To find the final product or final matrix combine the result of two matrixes.

## **Formulas for Strassen's matrix multiplication.**

Following are the formulae that are to be used for matrix multiplication.

1.  $D1 = (a11 + a22) * (b11 + b22)$
2.  $D2 = (a21 + a22)*b11$
3.  $D3 = (b12 - b22)*a11$
4.  $D4 = (b21 - b11)*a22$
5.  $D5 = (a11 + a12)*b22$
6.  $D6 = (a21 - a11) * (b11 + b12)$
7.  $D7 = (a12 - a22) * (b21 + b22)$

$$C00= d1 + d4 - d5 + d7$$

$$C01 = d3 + d5$$

$$C10 = d2 + d4$$

$$C11 = d1 + d3 - d2 - d6$$

Here, C00, C01, C10, and C11 are the elements of the  $2 \times 2$  matrix.

-----



## **UNIT-II**

### **Greedy Method:**

The greedy method is one of the strategies like Divide and conquer used to solve the problems.

This method is used for solving optimization problems. An optimization problem is a problem that demands either maximum or minimum results.

The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique.

The main function of this approach is that the decision is taken on the basis of the currently available information.

Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

This technique is basically used to determine the feasible solution that may or may not be optimal.

The feasible solution is a subset that satisfies the given criteria.

The optimal solution is the solution which is the best and the most favorable solution in the subset.

In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

#### **Applications of Greedy Algorithm**

- It is used in finding the shortest path.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in job sequencing with a deadline.
- This algorithm is also used to solve the fractional knapsack problem.

### **Algorithm:**

The function Select selects an input from  $a[]$  and removes it. The selected inputs value is assigned to 'x'. Feasible is a Boolean-valued function that determines whether 'x' can be included into the solution vector. The function Union combines 'x' with the solution and updates the objective function.

```

Algorithm Greedy( $a, n$ )
//  $a[1 : n]$  contains the  $n$  inputs.
{
     $solution := \emptyset$ ; // Initialize the solution.
    for  $i := 1$  to  $n$  do
    {
         $x := \text{Select}(a)$ ;
        if  $\text{Feasible}(solution, x)$  then
             $solution := \text{Union}(solution, x)$ ;
    }
    return  $solution$ ;
}

```

### Knapsack Problem:

Knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits. We have to put some items in the knapsack in such a way total value produces a maximum profit.

Ex:

The weight of the container is 20 kg. We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

There are two types of knapsack problems:

- 0/1 knapsack problem
- Fractional knapsack problem

### 0/1 knapsack problem

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack.

Ex:

We have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

## Fractional knapsack problem

The fractional knapsack problem means that we can divide the item.

Ex:

We have an item of 3 kg then we can pick the item of 2 kg and leave the item of 1 kg. The fractional knapsack problem is solved by the Greedy approach.

Let us try to apply the greedy method to solve the knapsack problem. We are given  $n$  objects and a knapsack or bag. Object  $i$  has a weight  $w_i$  and the Knapsack has a capacity  $m$ . If a fraction  $x_i$ ,  $0 < x_i < 1$ , of object  $i$  is placed In to the knapsack, then a profit of  $p_i x_i$  is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is  $m$ , we require the total weight of all chosen objects to be at most  $m$ . Formally, the problem can be stated as

$$\text{maximize } \sum_{1 \leq i \leq n} p_i x_i$$

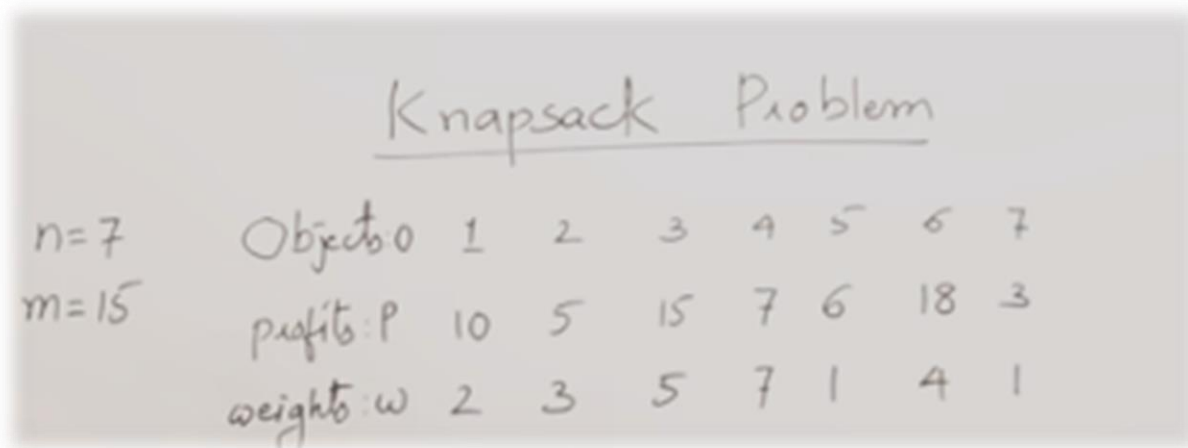
$$\text{subject to } \sum_{1 \leq i \leq n} w_i x_i \leq m$$

$$\text{and } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n$$

The profits and weights are positive numbers. A feasible solution(or filling) is any set  $(x_1, x_2, \dots, x_n)$  satisfying the second equation and third above. An optimal solution is a feasible solution for which first equation is maximized.

Ex:

## Solving Knapsack problem with General Method



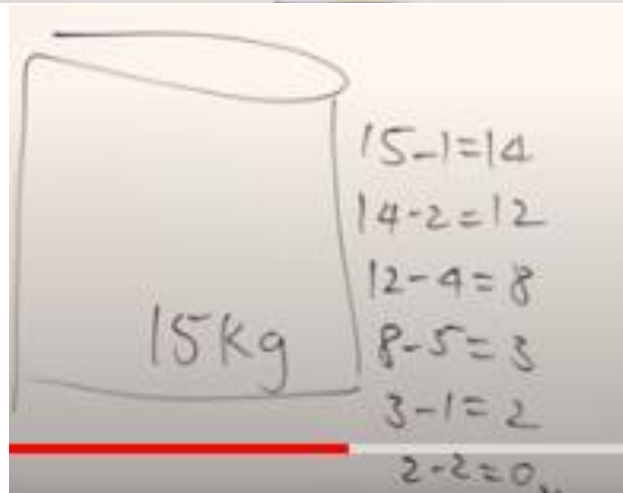
A handwritten table titled "Knapsack Problem" showing the data for a knapsack problem with 7 objects. The table lists the number of objects (n=7) and the knapsack capacity (m=15). It then lists the objects (0 to 7) with their respective profits (P) and weights (w).

	Object	0	1	2	3	4	5	6	7
n=7									
m=15									
	profits: P	10	5	15	7	6	18	3	
	weights: w	2	3	5	7	1	4	1	

## Knapsack Problem

$n=7$   
 $m=15$

Objecto	0	1	2	3	4	5	6	7
profits P		10	5	15	7	6	18	3
weights w		2	3	5	7	1	4	1
$\frac{P}{w}$		5	1.3	3	1	6	4.5	3
$x$		(1	$\frac{2}{3}$	1	0	1	1	1)
		$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$



## Knapsack Problem

Objecto	0	1	2	3	4	5	6	7	<u>Constraint</u>
profits P		10	5	15	7	6	18	3	$\sum x_i w_i \leq m$
weights w		2	3	5	7	1	4	1	
$\frac{P}{w}$		5	1.3	3	1	6	4.5	3	<u>Objective</u>
$x$		(1	$\frac{2}{3}$	1	0	1	1	1)	$\max \sum x_i p_i$
		$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	

$$\sum x_i w_i = 1 \times 2 + \frac{2}{3} \times 3 + 1 \times 5 + 0 \times 7 + 1 \times 1 + 1 \times 4 + 1 \times 1$$

$$2 + 2 + 5 + 0 + 1 + 4 + 1 = 15$$

$$\sum x_i p_i = 1 \times 10 + \frac{2}{3} \times 5 + 1 \times 15 + 1 \times 6 + 1 \times 18 + 1 \times 3$$

$$= 10 + 2 \times 1.3 + 15 + 6 + 18 + 3 = 54.6$$

-----

### **Job sequencing with deadlines:**

Job scheduling algorithm is applied to schedule the jobs on a single processor to maximize the profits.

The greedy approach of the job scheduling algorithm states that, “Given ‘n’ number of jobs with a starting time and ending time, they need to be scheduled in such a way that maximum profit is received within the maximum deadline”.

#### **Job Scheduling Algorithm**

Set of jobs with deadlines and profits are taken as an input with the job scheduling algorithm and scheduled subset of jobs with maximum profit are obtained as the final output.

#### **Algorithm**

- Find the maximum deadline value from the input set of jobs.
- Once, the deadline is decided, arrange the jobs in descending order of their profits.
- Selects the jobs with highest profits, their time periods not exceeding the maximum deadline.
- The selected sets of jobs are the output.

Ex:

### **Problem-**

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

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

Answer the following questions-

1. Write the optimal schedule that gives maximum profit.
2. Are all the jobs completed in the optimal schedule?

3. What is the maximum earned profit

**Step-01:**

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

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

**Step-02:**

Value of maximum deadline = 5.

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



**Gantt Chart**

Now,

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

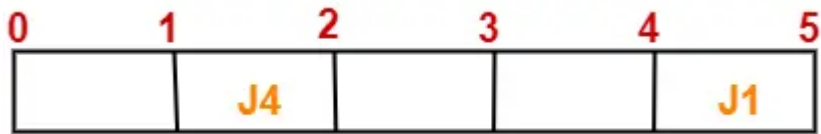
**Step-03:**

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



**Step-04:**

- We take job J1.
- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



**Step-05:**

- We take job J3.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-



**Step-06:**

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

0	1	2	3	4	5
J2	J4	J3		J1	

**Step-07:**

- Now, we take job J5.
- Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-

0	1	2	3	4	5
J2	J4	J3	J5	J1	

Now,

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

Now, the given questions may be answered as-

**Part-01:**

The optimal schedule is-

**J2, J4, J3, J5, J1**

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



## **Part-02:**

- All the jobs are not completed in optimal schedule.
- This is because job J6 could not be completed within its deadline.

## **Part-03:**

Maximum earned profit

= Sum of profit of all the jobs in optimal schedule

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

= 180 + 300 + 190 + 120 + 200

= 990 units

-----

## **Minimum cost spanning Trees:**

A **spanning tree** is a subset of an undirected Graph that has all the vertices connected by minimum number of edges.

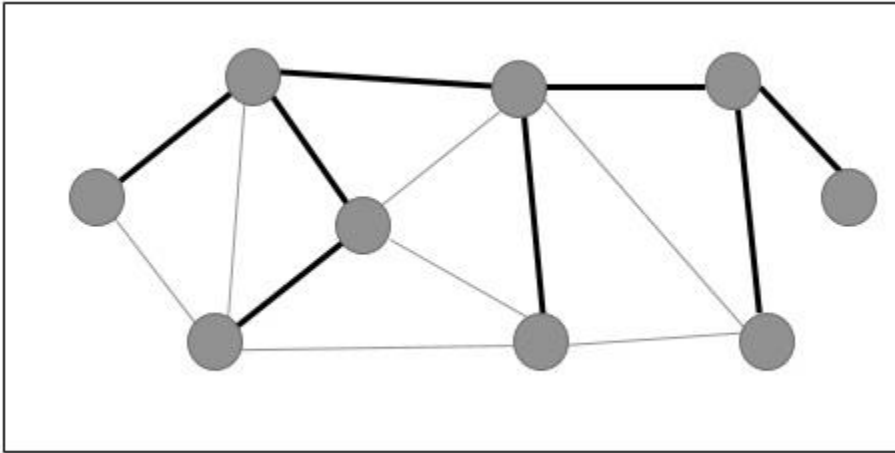
If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.

Properties

- A spanning tree does not have any cycle.
- Any vertex can be reached from any other vertex.

Example

In the following graph, the highlighted edges form a spanning tree.

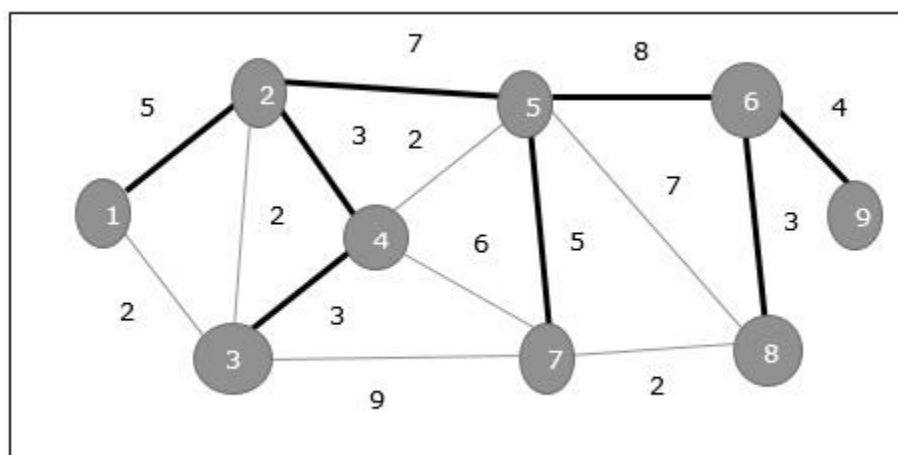


### Minimum Spanning Tree

A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used. Hence, we will discuss Prim's algorithm in this chapter.

As we have discussed, one graph may have more than one spanning tree. If there are  $n$  number of vertices, the spanning tree should have  $n-1$  number of edges. In this context, if each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.

Moreover, if there exist any duplicate weighted edges, the graph may have multiple minimum spanning tree.

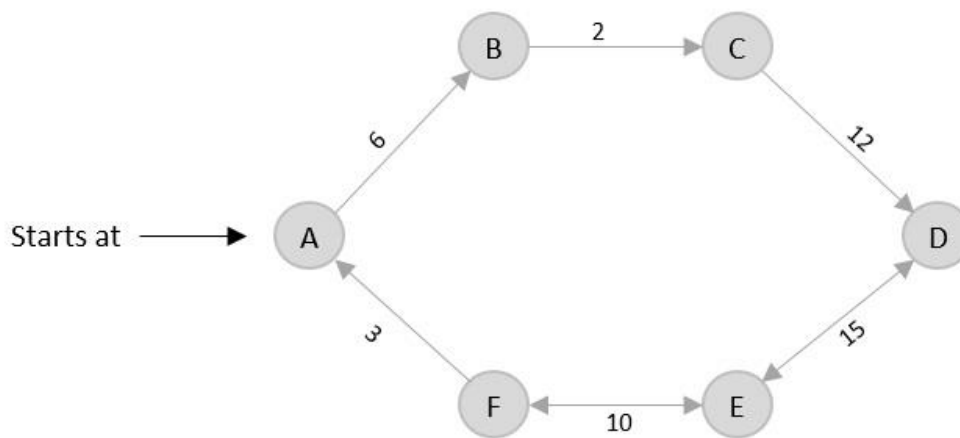


In the above graph, we have shown a spanning tree though it's not the minimum spanning tree. The cost of this spanning tree is  $(5 + 7 + 3 + 3 + 5 + 8 + 3 + 4) = 38$ .

## Prim's Algorithm:

Prim's minimal spanning tree algorithm is one of the efficient methods to find the minimum spanning tree of a graph. A minimum spanning tree is a subgraph that connects all the vertices present in the main graph with the least possible edges and minimum cost (sum of the weights assigned to each edge).

The algorithm, similar to any shortest path algorithm, begins from a vertex that is set as a root and walks through all the vertices in the graph by determining the least cost adjacent edges.



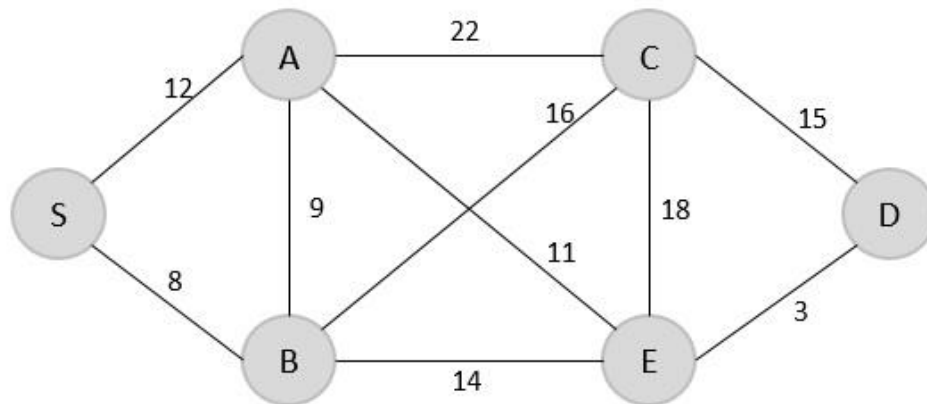
To execute the prim's algorithm, the inputs taken by the algorithm are the graph  $G$   $\{V, E\}$ , where  $V$  is the set of vertices and  $E$  is the set of edges, and the source vertex  $S$ . A minimum spanning tree of graph  $G$  is obtained as an output.

### Algorithm

- Declare an array *visited*[] to store the visited vertices and firstly, add the arbitrary root, say  $S$ , to the visited array.
- Check whether the adjacent vertices of the last visited vertex are present in the *visited*[] array or not.
- If the vertices are not in the *visited*[] array, compare the cost of edges and add the least cost edge to the output spanning tree.
- The adjacent unvisited vertex with the least cost edge is added into the *visited*[] array and the least cost edge is added to the minimum spanning tree output.
- Steps 2 and 4 are repeated for all the unvisited vertices in the graph to obtain the full minimum spanning tree output for the given graph.
- Calculate the cost of the minimum spanning tree obtained.

### Examples

- Find the minimum spanning tree using prim's method (greedy approach) for the graph given below with S as the arbitrary root.



### Step 1

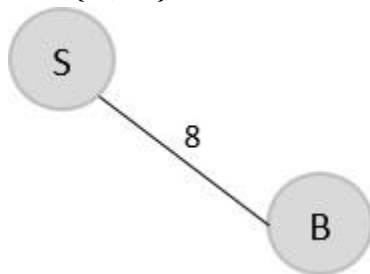
Create a visited array to store all the visited vertices into it.

$V = \{ \}$

The arbitrary root is mentioned to be S, so among all the edges that are connected to S we need to find the least cost edge.

$S \rightarrow B = 8$

$V = \{S, B\}$



### Step 2

Since B is the last visited, check for the least cost edge that is connected to the vertex B.

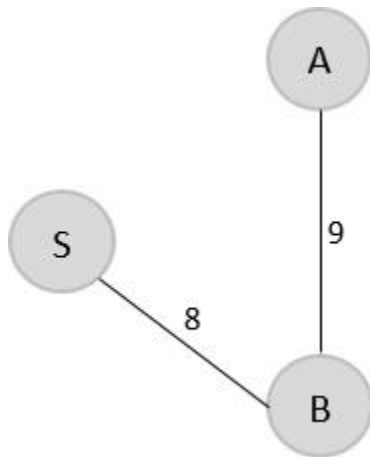
$B \rightarrow A = 9$

$B \rightarrow C = 16$

$B \rightarrow E = 14$

Hence,  $B \rightarrow A$  is the edge added to the spanning tree.

$V = \{S, B, A\}$



### Step 3

Since A is the last visited, check for the least cost edge that is connected to the vertex A.

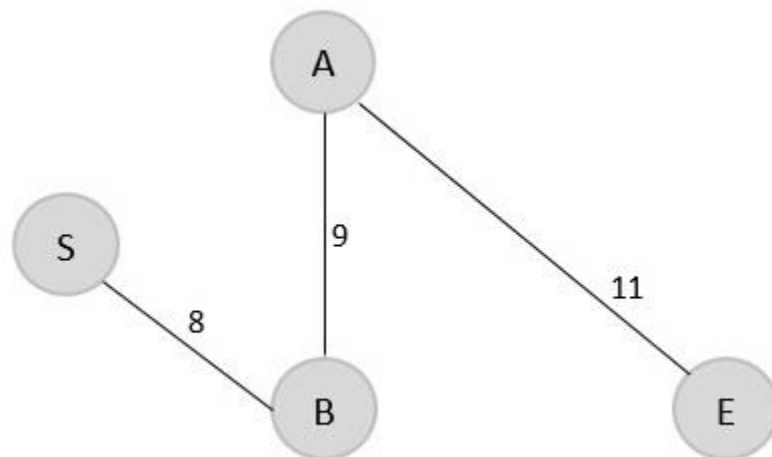
$$A \rightarrow C = 22$$

$$A \rightarrow B = 9$$

$$A \rightarrow E = 11$$

But  $A \rightarrow B$  is already in the spanning tree, check for the next least cost edge. Hence,  $A \rightarrow E$  is added to the spanning tree.

$$V = \{S, B, A, E\}$$



### Step 4

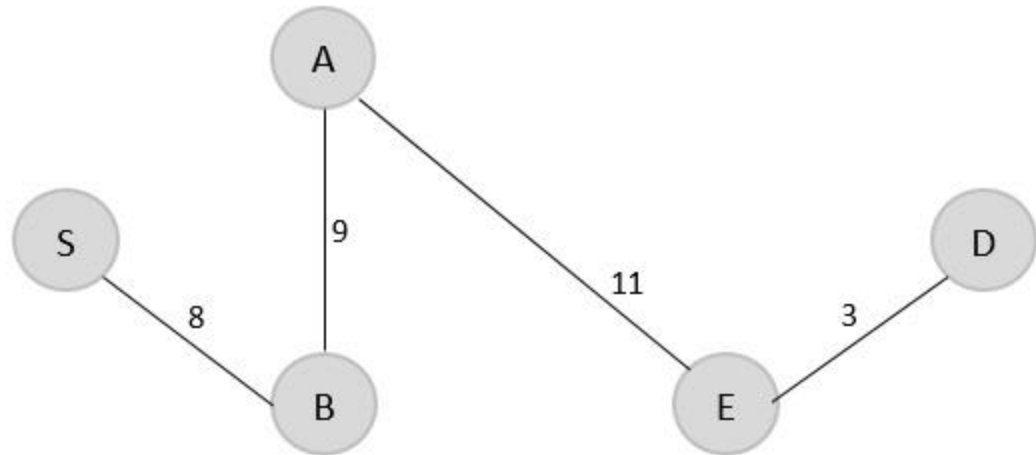
Since E is the last visited, check for the least cost edge that is connected to the vertex E.

$$E \rightarrow C = 18$$

$$E \rightarrow D = 3$$

Therefore,  $E \rightarrow D$  is added to the spanning tree.

$$V = \{S, B, A, E, D\}$$



### Step 5

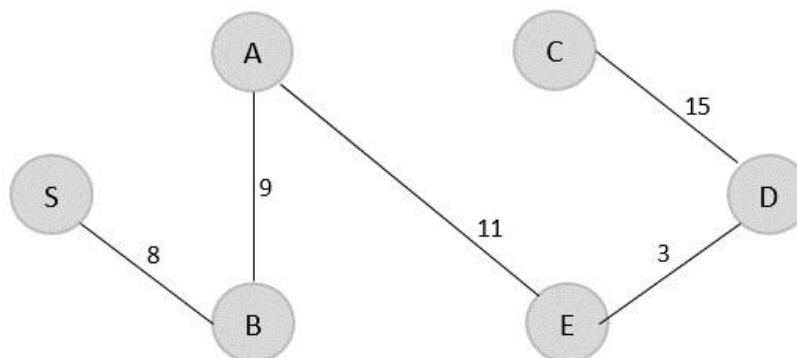
Since D is the last visited, check for the least cost edge that is connected to the vertex D.

$D \rightarrow C = 15$

$E \rightarrow D = 3$

Therefore,  $D \rightarrow C$  is added to the spanning tree.

$V = \{S, B, A, E, D, C\}$



The minimum spanning tree is obtained with the minimum cost = 46

### Kruskal Algorithm:

The Kruskal Algorithm is used to find the minimum cost of a spanning tree. A spanning tree is a connected graph using all the vertices in which there are no loops. In other words, we can say that there is a path from any vertex to any other vertex but no loops.

The minimum spanning tree is a spanning tree that has the smallest total edge weight. The Kruskal algorithm is an algorithm that takes the graph as input and finds the edges from the graph, which forms a tree that includes every vertex of a graph.

### Working of Kruskal Algorithm

The working of the Kruskal algorithm starts from the edges, which has the lowest weight and keeps adding the edges until we reach the goal.

#### **The following are the steps used to implement the Kruskal algorithm:**

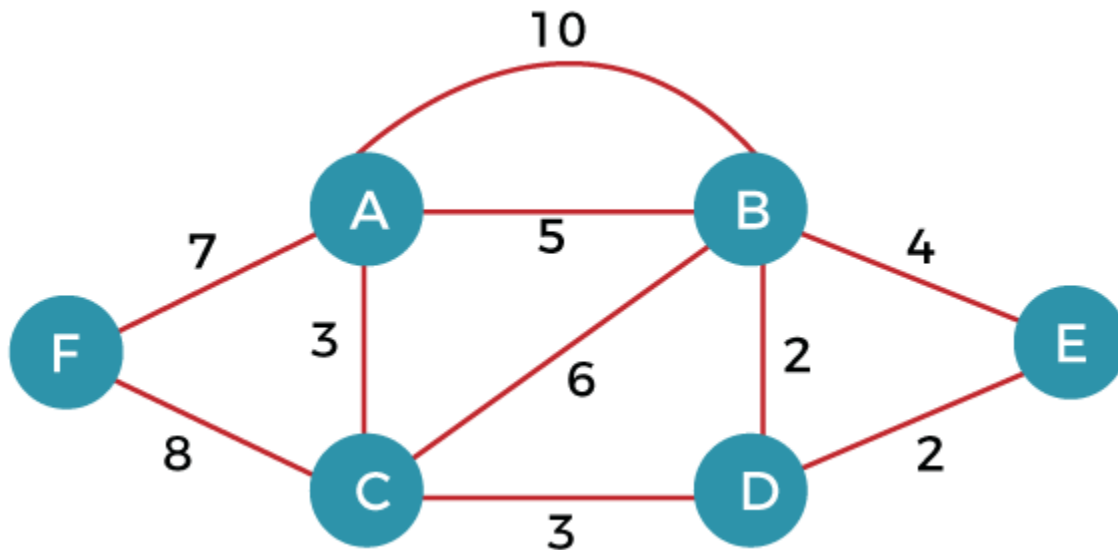
- First, sort the edges in the ascending order of their edge weights.
- Consider the edge which is having the lowest weight and add it in the spanning tree. If adding any edge in a spanning tree creates a cycle then reject that edge.
- Keep adding the edges until we reach the end vertex.

#### **Algorithm**

1. Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.
2. Step 2: Create a set E that contains all the edges of the graph.
3. Step 3: Repeat Steps 4 and 5 **while** E is NOT EMPTY and F is not spanning
4. Step 4: Remove an edge from E with minimum weight
5. Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F
6. (**for** combining two trees into one tree).
7. ELSE
8. Discard the edge
9. Step 6: END

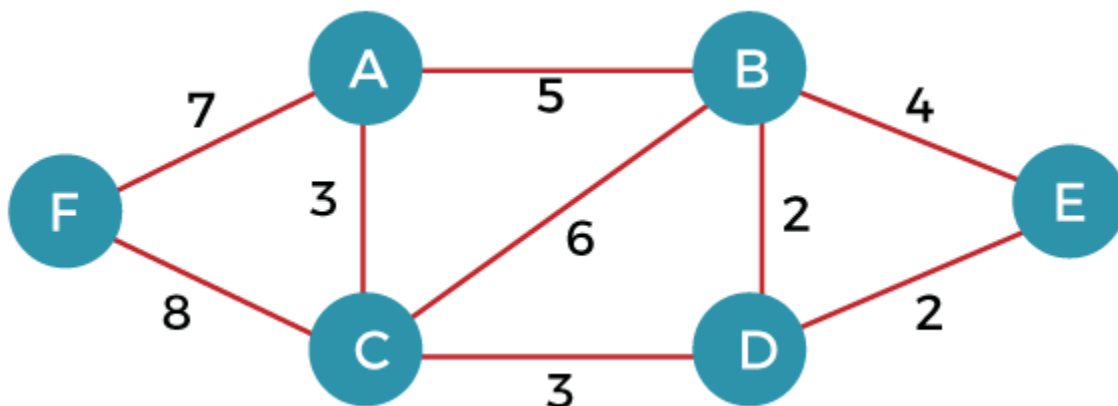
**Ex:**

**Consider the below graph.**



Now we have to calculate the minimum spanning tree of the above graph. To find the minimum cost spanning tree, the first step is to remove all the loops and the parallel edges from the graph. Here, parallel edges mean that there exists more than one edge between the two vertices. For example, in the above graph, there exists two edges between the vertices A and B having weights 5 and 10 respectively, so these edges are the parallel edges. We have to remove any of the edges.

In case of parallel edges, the edges with the highest weight will be removed. In the above graph, if we consider the vertices A and B then the edge with a weight 10 is more than the edge with weight 5 so we discard the edge having weight 10. So, we remove the edge having weight 10 as shown as below:



Once we remove the parallel edge, we will arrange the edges according to the increasing order of their edge weights. As we can observe in the above graph, the minimum edge is 2. There are two edges having weight equals to 2 which can be written as:

$BD = 2$

$DE = 2$



The next minimum edge weight is 3. There are two edges with a weight 3 so it can be written as:

$$AC = 3$$

$$CD = 3$$

The next edge with a minimum weight is 4. There is only one edge having weight 4 and it can be written as:

$$BE = 4$$

The next edge with a minimum weight is 5. There is only one edge having weight 5 and it can be written as:

$$AB = 5$$

The next edge with a minimum weight is 6. There is only one edge having weight 6 and it can be written as:

$$CB = 6$$

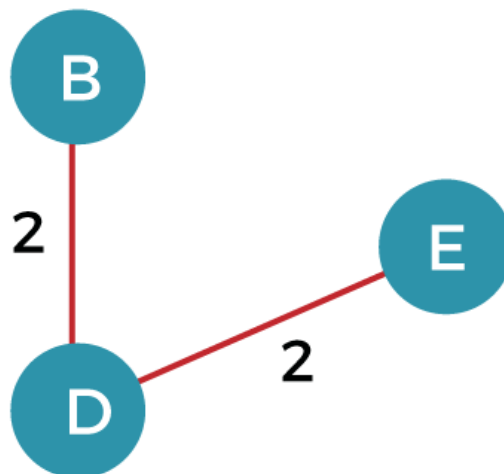
The next edge with a minimum weight is 7. There is only one edge having weight 7 and it can be written as:

$$AF = 7$$

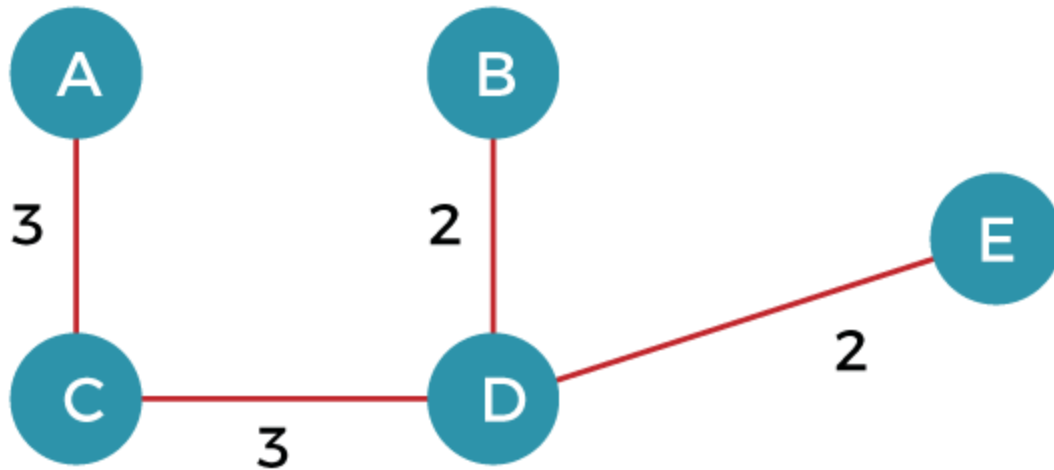
The next edge with a minimum weight is 8. There is only one edge having weight 8 and it can be written as:

$$FC = 8$$

Once the edge weights are written in the increasing order, the third step is to connect the vertices according to their weights. The edge weight 2 is minimum, so we connect the vertices with a edge weight 2 as shown as below:



The next edge is AC having weight 3 shown as below:



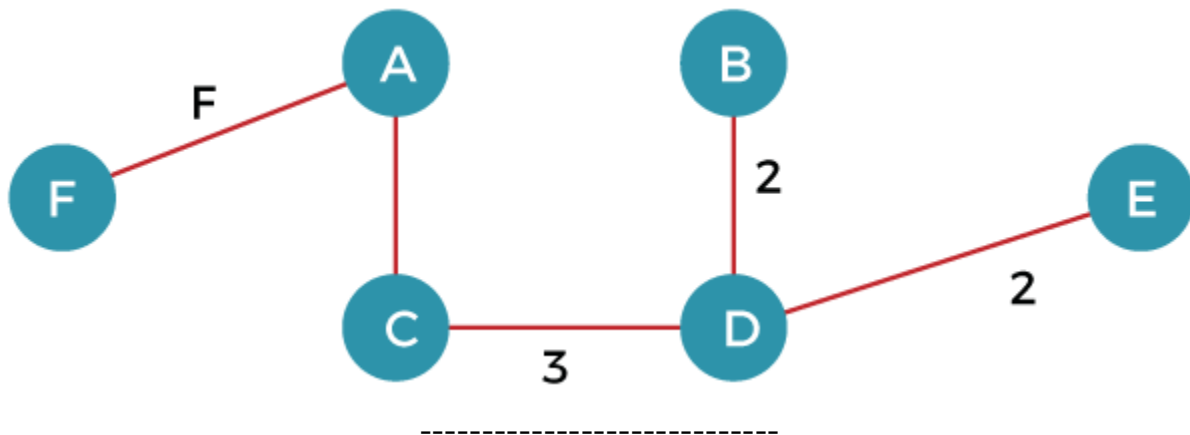
The next edge is CD. If we connect the vertices C and D then it does not form any cycle shown as below:

The next edge is BE. But we cannot connect the vertices B and E as it forms a cycle.

The next edge is AB. But we cannot connect the vertices A and B as it forms a cycle.

The next edge is BC. But we cannot connect the vertices B and C as it forms a cycle.

The next edge is AF. If we connect the vertices A and F then it does not form any cycle shown as below:



### **Optimal Storage on Tapes:**

Given  $n$  programs stored on a computer tape and length of each program is  $l_i$  where  $i = 1, 2, \dots, n$  find the order in which the programs should be stored in the tape for which the Mean Retrieval Time (MRT given as  $M$ ) is minimized.

The greedy algorithm finds the MRT as following:

Algorithm MRT\_SINGLE\_TAPE(L)

// Description: Find storage order of n programs to such that mean retrieval time is minimum

//Input: L is array of program length sorted in ascending order

// Output: Minimum Mean Retrieval Time

```
Tj <— 0
for i <— 1 to n do
  for j <- 1 to i do
    Tj <- Tj + L[j]
  end
end
```

Example:

Input : n = 3

L[] = { 5, 3, 10 }

Output : Order should be { 3, 5, 10 } with MRT = 29/3

A magnetic tape provides only sequential access of data. In an audio tape/cassette, unlike a CD, a fifth song from the tape can't be just directly played. The length of the first four songs must be traversed to play the fifth song. So in order to access certain data, head of the tape should be positioned accordingly.

Now suppose there are 4 songs in a tape of audio lengths 5, 7, 3 and 2 mins respectively. In order to play the fourth song, we need to traverse an audio length of  $5 + 7 + 3 = 15$  mins and then position the tape head.

Retrieval time of the data is the time taken to retrieve/access that data in its entirety. Hence retrieval time of the fourth song is  $15 + 2 = 17$  mins. Now, considering that all programs in a magnetic tape are retrieved equally often and the tape head points to the front of the tape every time, a new term can be defined called the Mean Retrieval Time (MRT).

The sequential access of data in a tape has some limitations. Order must be defined in which the data/programs in a tape are stored so that least MRT can be obtained. Hence the order of storing becomes very important to reduce the data retrieval/access time.

Ex:

Suppose there are 3 programs of lengths 2, 5 and 4 respectively. So there are total  $3! = 6$  possible orders of storage.

	Order	Total Retrieval Time	Mean Retrieval Time
1	1 2 3	$2 + (2 + 5) + (2 + 5 + 4) = 20$	

2	1 3 2	$2 + (2 + 4) + (2 + 4 + 5) = 19$	19/3
3	2 1 3	$5 + (5 + 2) + (5 + 2 + 4) = 23$	23/3
4	2 3 1	$5 + (5 + 4) + (5 + 4 + 2) = 25$	25/3
5	3 1 2	$4 + (4 + 2) + (4 + 2 + 5) = 21$	21/3
6	3 2 1	$4 + (4 + 5) + (4 + 5 + 2) = 24$	24/3

In above example, the first program's length is added 'n' times, the second 'n-1' times...and so on till the last program is added only once. So, careful analysis suggests that in order to minimize the MRT, programs having greater lengths should be put towards the end so that the summation is reduced. Or, the lengths of the programs should be sorted in increasing order. That's the Greedy Algorithm in use – at each step we make the immediate choice of putting the program having the least time first, in order to build up the ultimate optimized solution to the problem piece by piece.

-----

### **Optimal merge pattern:**

Given n number of sorted files, the task is to find the minimum computations done to reach the Optimal Merge Pattern.

When two or more sorted files are to be merged altogether to form a single file, the minimum computations are done to reach this file are known as **Optimal Merge Pattern**.

If more than 2 files need to be merged then it can be done in pairs.

Ex:

If need to merge 4 files A, B, C, D. First Merge A with B to get X1, merge X1 with C to get X2, merge X2 with D to get X3 as the output file.

If we have two files of sizes m and n, the total computation time will be m+n. Here, we use the greedy strategy by merging the two smallest size files among all the files present.

**Ex:**

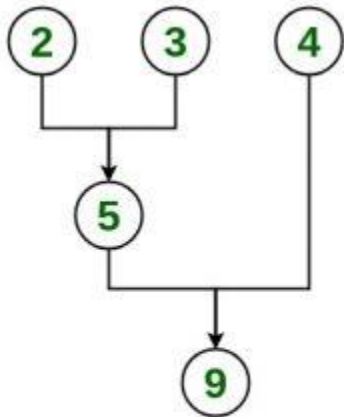
Given 3 files with sizes 2, 3, 4 units. Find an optimal way to combine these files

**Input:**  $n = 3$ ,  $size = \{2, 3, 4\}$

**Output:** 14

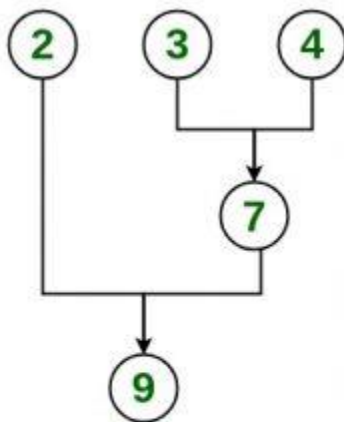
**Explanation:** There are different ways to combine these files:

**Method 1:** Optimal method



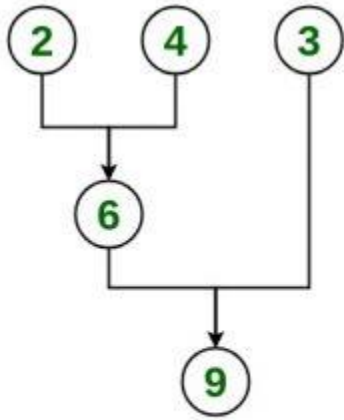
$$\text{Cost} = 5 + 9 = 14$$

**Method 2:**



$$\text{Cost} = 7 + 9 = 16$$

**Method 3:**

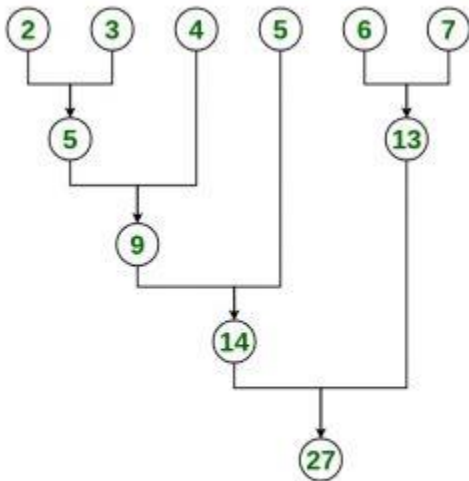


$$\text{Cost} = 6 + 9 = 15$$

**Input:**  $n = 6$ ,  $\text{size} = \{2, 3, 4, 5, 6, 7\}$

**Output:** 68

**Explanation:** Optimal way to combine these files



$$\text{Cost} = 5 + 9 + 13 + 14 + 27 = 68$$

**Input:**  $n = 5$ ,  $\text{size} = \{5, 10, 20, 30, 30\}$

**Output:** 205

-----

### Single source shortest Path:

In a shortest- paths problem, we are given a weighted, directed graphs  $G = (V, E)$ , with weight function  $w: E \rightarrow \mathbb{R}$  mapping edges to real-valued weights. The weight of path  $p = (v_0, v_1, \dots, v_k)$  is the total of the weights of its constituent edges:

$$w(P) = \sum_{i=1}^k w(v_{i-1}v_i)$$

We define the shortest - path weight from  $u$  to  $v$  by  $\delta(u,v) = \min (w(p): u \rightarrow v)$ , if there is a path from  $u$  to  $v$ , and  $\delta(u,v) = \infty$ , otherwise.

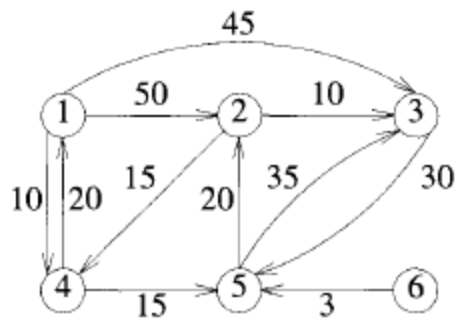
The shortest path from vertex  $s$  to vertex  $t$  is then defined as any path  $p$  with weight  $w(p) = \delta(s,t)$ .

The breadth-first- search algorithm is the shortest path algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight.

In a Single Source Shortest Paths Problem, we are given a Graph  $G = (V, E)$ , we want to find the shortest path from a given source vertex  $s \in V$  to every vertex  $v \in V$ .

There are some variants of the shortest path problem.

- **Single- destination shortest - paths problem:** Find the shortest path to a given destination vertex  $t$  from every vertex  $v$ . By shift the direction of each edge in the graph, we can shorten this problem to a single - source problem.
- **Single - pair shortest - path problem:** Find the shortest path from  $u$  to  $v$  for given vertices  $u$  and  $v$ . If we determine the single - source problem with source vertex  $u$ , we clarify this problem also. Furthermore, no algorithms for this problem are known that run asymptotically faster than the best single - source algorithms in the worst case.
- **All - pairs shortest - paths problem:** Find the shortest path from  $u$  to  $v$  for every pair of vertices  $u$  and  $v$ . Running a single - source algorithm once from each vertex can clarify this problem; but it can generally be solved faster, and its structure is of interest in the own right.



(a) Graph

Path	Length
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

(b) Shortest paths from 1

---

### Greedy algorithm to generate shortest paths:

**Algorithm** ShortestPaths( $v, cost, dist, n$ )  
 //  $dist[j]$ ,  $1 \leq j \leq n$ , is set to the length of the shortest  
 // path from vertex  $v$  to vertex  $j$  in a digraph  $G$  with  $n$   
 // vertices.  $dist[v]$  is set to zero.  $G$  is represented by its  
 // cost adjacency matrix  $cost[1 : n, 1 : n]$ .  
 {  
   for  $i := 1$  to  $n$  do  
   { // Initialize  $S$ .  
      $S[i] := \text{false}$ ;  $dist[i] := cost[v, i]$ ;  
   }  
    $S[v] := \text{true}$ ;  $dist[v] := 0.0$ ; // Put  $v$  in  $S$ .  
   for  $num := 2$  to  $n - 1$  do  
   {  
     // Determine  $n - 1$  paths from  $v$ .  
     Choose  $u$  from among those vertices not  
     in  $S$  such that  $dist[u]$  is minimum;  
      $S[u] := \text{true}$ ; // Put  $u$  in  $S$ .  
     for (each  $w$  adjacent to  $u$  with  $S[w] = \text{false}$ ) do  
       // Update distances.  
       if ( $dist[w] > dist[u] + cost[u, w]$ ) then  
          $dist[w] := dist[u] + cost[u, w]$ ;  
   }  
 }

-----



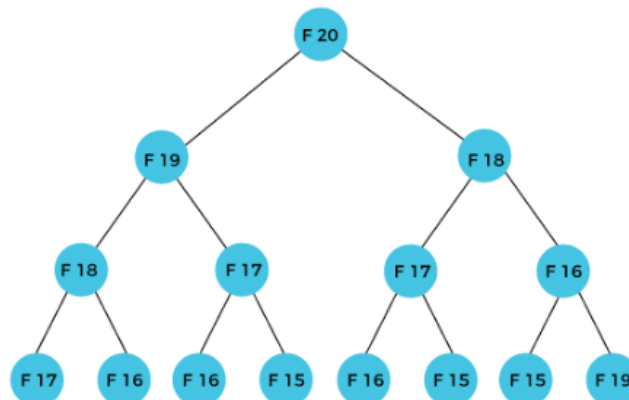
### General Method:

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each sub problem just once, and then storing their solutions to avoid repetitive computations.

**0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,.....**

$$\mathbf{F(n) = F(n-1) + F(n-2),}$$

The  $F(20)$  term will be calculated using the  $n$ th formula of the Fibonacci series. The below figure shows that how  $F(20)$  is calculated.



The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (memoization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.

-----

### **All pairs shortest path:**

It aims to figure out the shortest path from each vertex  $v$  to every other  $u$ . Storing all the paths explicitly can be very memory expensive indeed, as we need one spanning tree for each vertex. This is often impractical regarding memory consumption, so these are generally considered as all pairs-shortest distance problems, which aim to find just the distance from each to each node to another. We usually want the output in tabular form: the entry in  $u$ 's row and  $v$ 's column should be the weight of the shortest path from  $u$  to  $v$ .

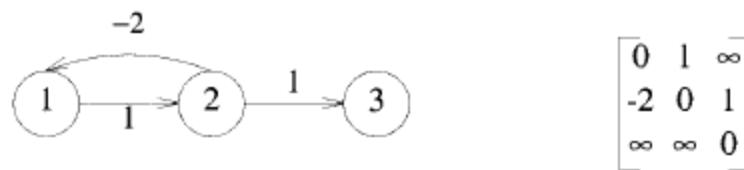
Three approaches for improvement:

Algorithm	Cost
Matrix Multiplication	$O(V^3 \log V)$
Floyd-Warshall	$O(V^3)$

Unlike the single-source algorithms, which assume an adjacency list representation of the graph, most of the algorithm uses an adjacency matrix representation. (Johnson's Algorithm for sparse graphs uses adjacency lists.) The input is a  $n \times n$  matrix  $W$  representing the edge weights of an  $n$ -vertex directed graph  $G = (V, E)$ . That is,  $W = (w_{ij})$ , where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i, j) & \text{if } i \neq j \text{ and } (i, j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E. \end{cases}$$

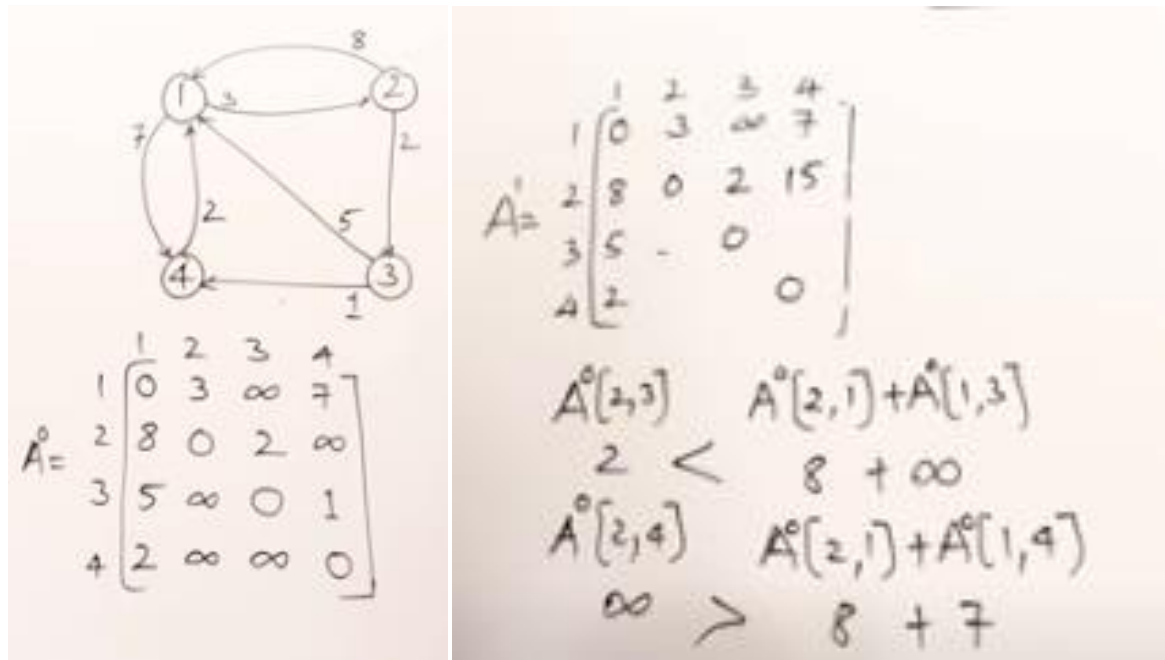
Ex:



Algorithm:

**Algorithm AllPaths**(*cost*, *A*, *n*)  
 // *cost*[1 : *n*, 1 : *n*] is the cost adjacency matrix of a graph with  
 // *n* vertices; *A*[*i*, *j*] is the cost of a shortest path from vertex  
 // *i* to vertex *j*. *cost*[*i*, *i*] = 0.0, for  $1 \leq i \leq n$ .  
 {  
   for *i* := 1 to *n* do  
     for *j* := 1 to *n* do  
       *A*[*i*, *j*] := *cost*[*i*, *j*]; // Copy *cost* into *A*.  
     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*]);  
 }

Ex:



$$A^1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 8 & \infty & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 3 & 8 & 0 & 1 \\ 4 & 8 & 0 & 1 \end{bmatrix}$$

$$A^1[1,3] \quad A^1[1,2] + A^1[2,3]$$

$$\infty > 3 + 2$$

$$A^1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{bmatrix}$$

$$A^2[1,2] \quad A^2[1,3] + A^2[3,2]$$

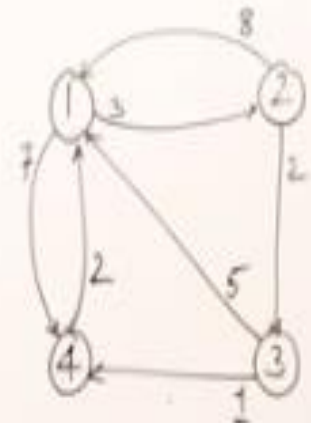
$$3 < 5 + 8$$

$$A^1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{bmatrix}$$

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

$$A^4 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$



$$A^0 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & 8 & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix}$$

$$A^k[i,j] = \min \left\{ \underbrace{A^{k-1}[i,j]}_{\text{direct edge}}, \underbrace{A^{k-1}[i,k] + A^{k-1}[k,j]}_{\text{path through k}} \right\}$$

## Matrix chain Multiplication:

It is a Method under Dynamic Programming in which previous output is taken as input for next.

Here, Chain means one matrix's column is equal to the second matrix's row [always].

In general:

If  $A = [a_{ij}]$  is a  $p \times q$  matrix

$B = [b_{ij}]$  is a  $q \times r$  matrix

$C = [c_{ij}]$  is a  $p \times r$  matrix

Then

$$AB = C \text{ if } c_{ij} = \sum_{k=1}^q a_{ik} b_{kj}$$

Given following matrices  $\{A_1, A_2, A_3, \dots, A_n\}$  and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications

$A_1 \times A_2 \times A_3 \times \dots \times A_n$

Matrix Multiplication operation is **associative** in nature rather commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to **parenthesize** the above multiplication depending upon our need.

In general, for  $1 \leq i \leq p$  and  $1 \leq j \leq r$

$$C[i, j] = \sum_{k=1}^q A[i, k]B[k, j]$$

It can be observed that the total entries in matrix 'C' is 'pr' as the matrix is of dimension  $p \times r$ . Also each entry takes  $O(q)$  times to compute, thus the total time to compute all possible entries for the matrix 'C' which is a multiplication of 'A' and 'B' is proportional to the product of the dimension  $pqr$ .

It is also noticed that we can save the number of operations by reordering the parenthesis.

**Example1:** Let us have 3 matrices,  $A_1, A_2, A_3$  of order  $(10 \times 100)$ ,  $(100 \times 5)$  and  $(5 \times 50)$  respectively.

Three Matrices can be multiplied in two ways:

1.  **$A_1, (A_2, A_3)$ :** First multiplying  $(A_2$  and  $A_3)$  then multiplying and resultant with  $A_1$ .
2.  **$(A_1, A_2), A_3$ :** First multiplying  $(A_1$  and  $A_2)$  then multiplying and resultant with  $A_3$ .

No of Scalar multiplication in Case 1 will be:

1.  $(100 \times 5 \times 50) + (10 \times 100 \times 50) = 25000 + 50000 = 75000$

No of Scalar multiplication in Case 2 will be:

1.  $(100 \times 10 \times 5) + (10 \times 5 \times 50) = 5000 + 2500 = 7500$

To find the best possible way to calculate the product, we could simply parenthesis the expression in every possible fashion and count each time how many scalar multiplication are required.

Matrix Chain Multiplication Problem can be stated as "find the optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplication is minimized".

-----

### **Optimal Binary search Trees:**

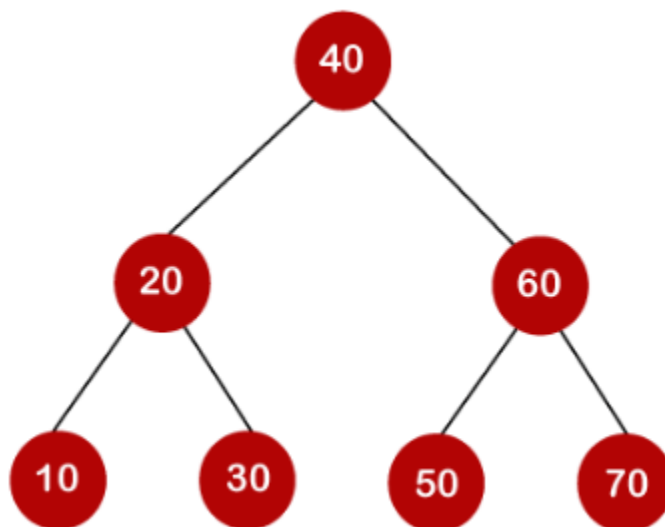
In binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node.

We know the key values of each node in the tree, and we also know the frequencies of each node in terms of searching means how much time is required to search a node. The frequency and key-value determine the overall cost of searching a node. The cost of searching is a very important factor in various applications. The overall cost of searching a node should be less. The time required to search a node in BST is more than the balanced binary search tree as a balanced binary search tree contains a lesser number of levels than the BST. There is one way that can reduce the cost of a binary search tree is known as an **optimal binary search tree**.

Ex:

Consider the key elements 10, 20, 30, 40, 50, 60

The binary search tree for the above key elements can be represented as follows.



In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node. The maximum time required to search a node is equal to the minimum height of the tree, equal to  $\log n$ .

Ex:

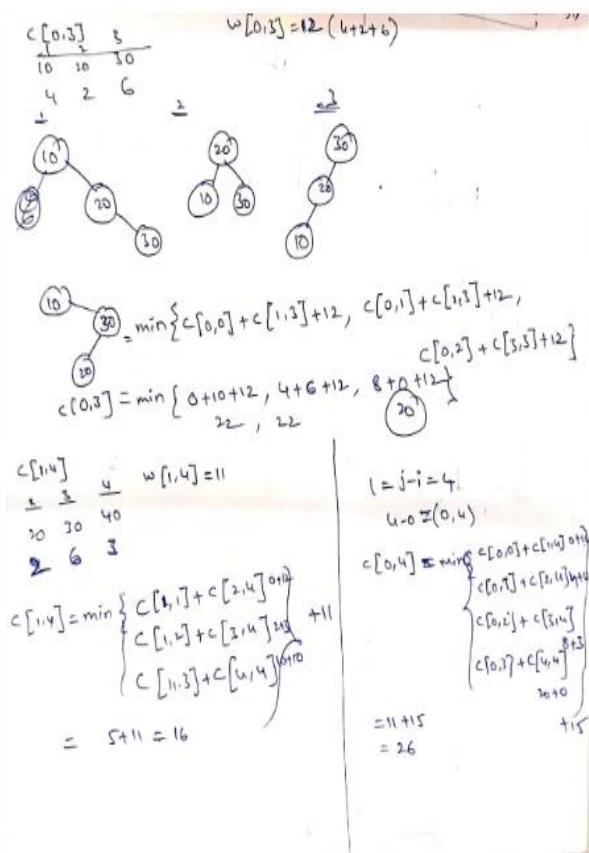
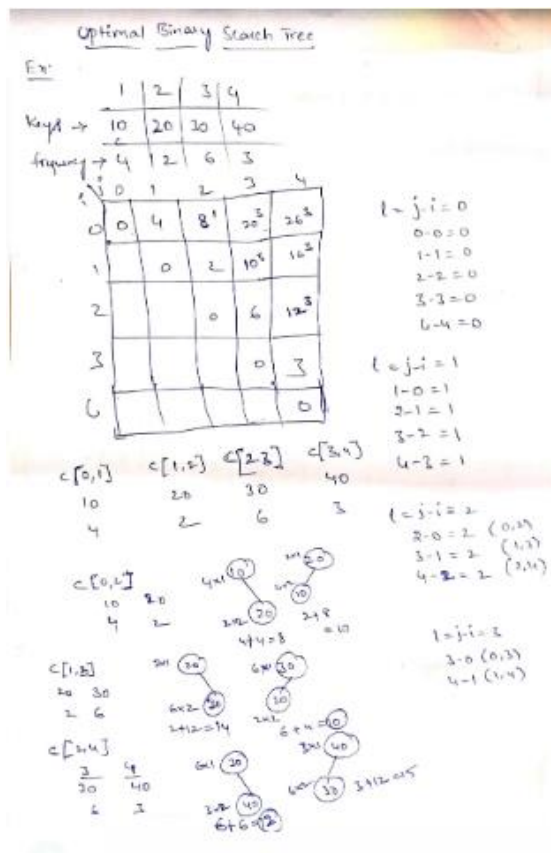
Consider 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.

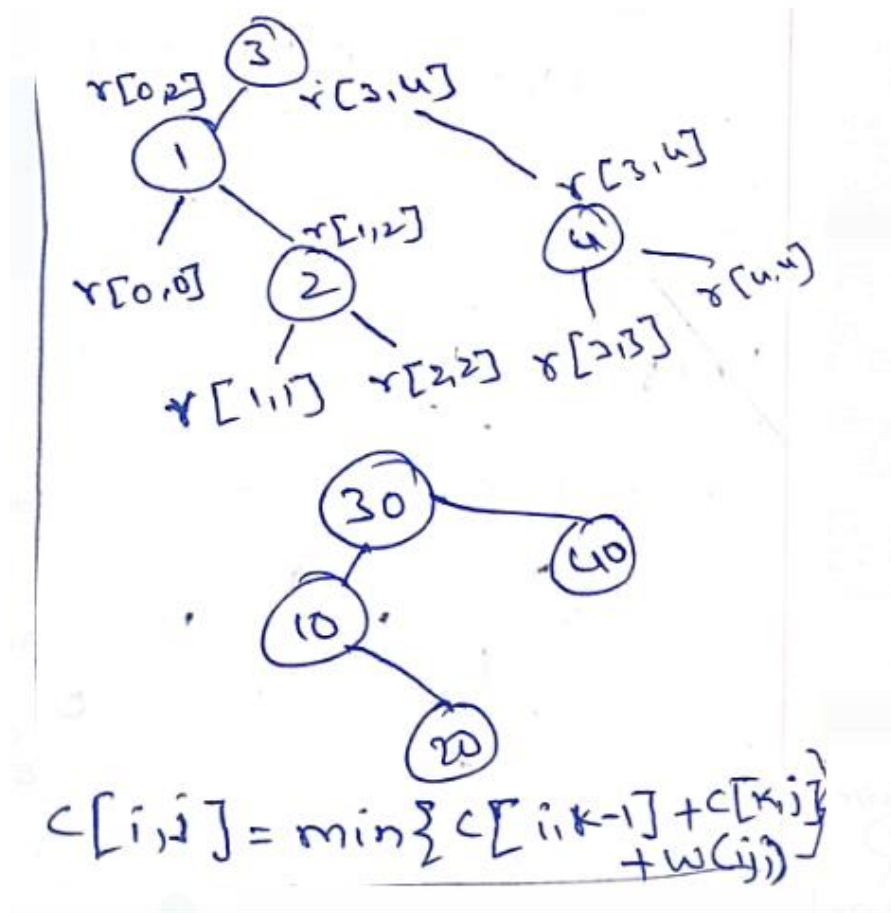
The Formula for calculating the number of trees:

$$\frac{2^n C_n}{n+1}$$

When we use the above formula, then it is found that total 5 number of trees can be created.

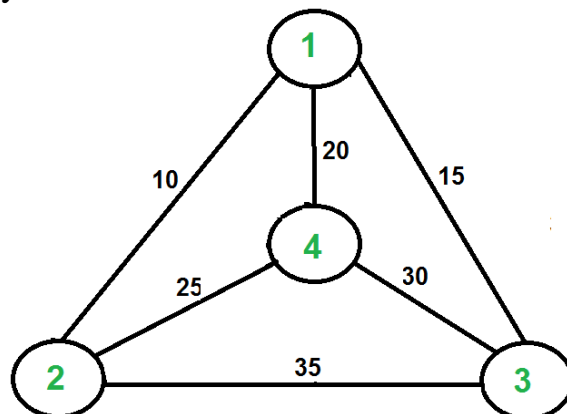
Ex:





### The Travelling salesman problem:

Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.





For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is  $10+25+30+15$  which is 80. The problem is a famous NP-hard problem. There is no polynomial-time known solution for this problem. The following are different solutions for the traveling salesman problem.

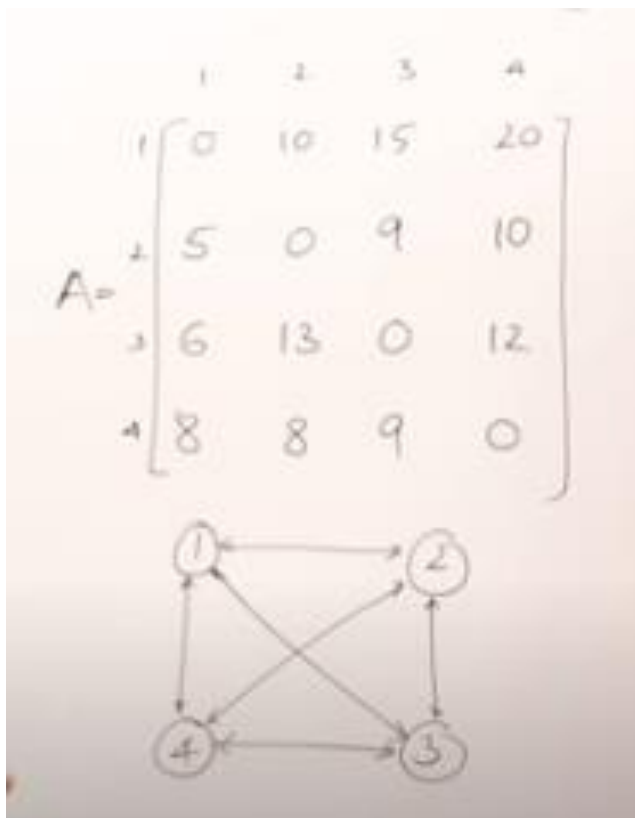
### Naive Solution:

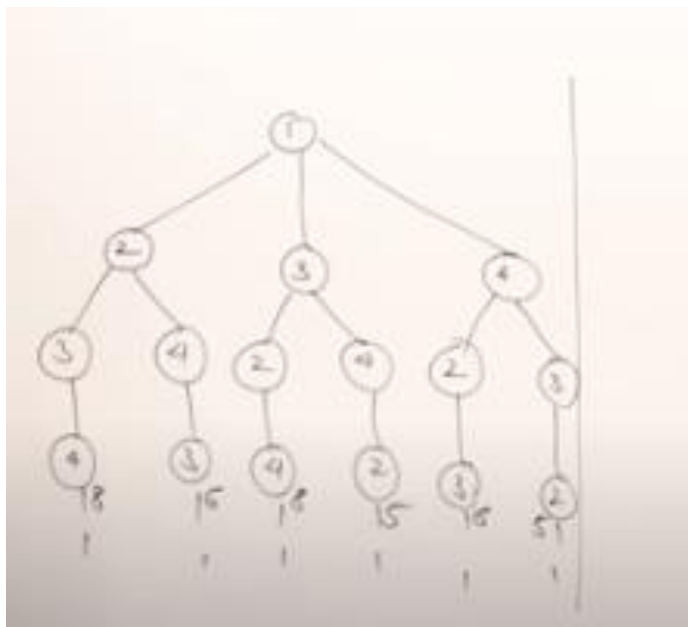
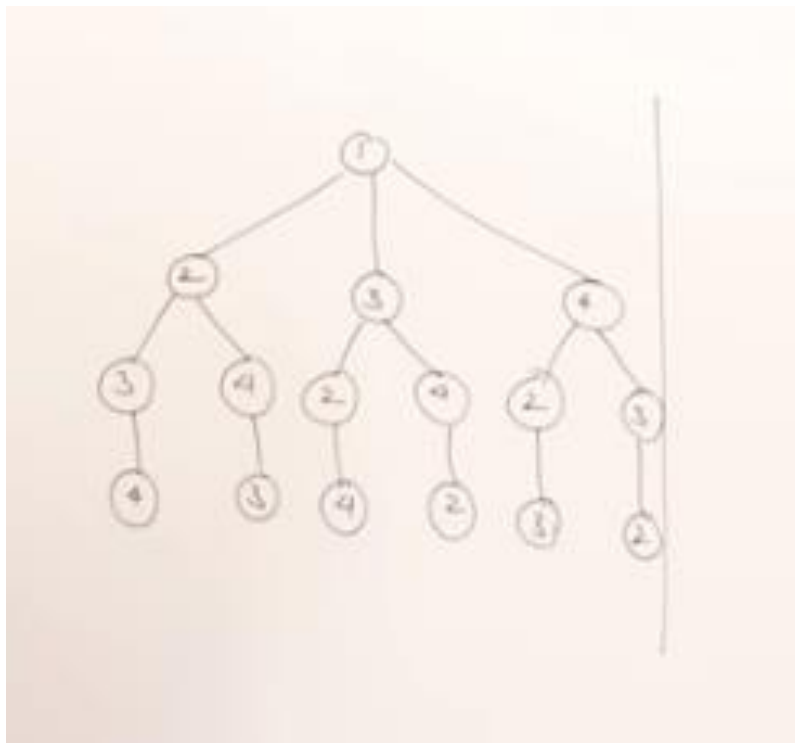
- 1) Consider city 1 as the starting and ending point.
- 2) Generate all  $(n-1)!$  Permutations of cities.
- 3) Calculate the cost of every permutation and keep track of the minimum cost permutation.
- 4) Return the permutation with minimum cost.

Time Complexity:  $\Theta(n!)$

Ex:

The brute force solution to the Traveling Salesman Problem (TSP) is to generate all possible city permutations and calculate each route's length. The shortest route is then returned as the solution to the TSP. The time complexity of this solution is  $O(n!)$ , where  $n$  is the number of cities.





$$g(i, \{2, 3, 4\}) = \min_{k \in \{2, 3, 4\}} \{C_{ik} + g(k, \{2, 3, 4\} - \{k\})\}$$

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

Traveling Salesperson Problem

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

$$= 35 \quad \frac{10 + 25}{35}, \frac{15 + 25}{35}, \frac{20 + 25}{35}$$



$$C_{ij}$$

$$g(2, \emptyset) = 5$$

$$g(3, \emptyset) = 6$$

$$g(4, \emptyset) = 8$$

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

$$g(2, \{3, 4\}) = 8$$

$$g(3, \{2\}) = 5$$

$$g(3, \{2, 4\}) = 8$$

$$g(4, \{2, 3\}) = 5$$

$$g(4, \{3\}) = 6$$

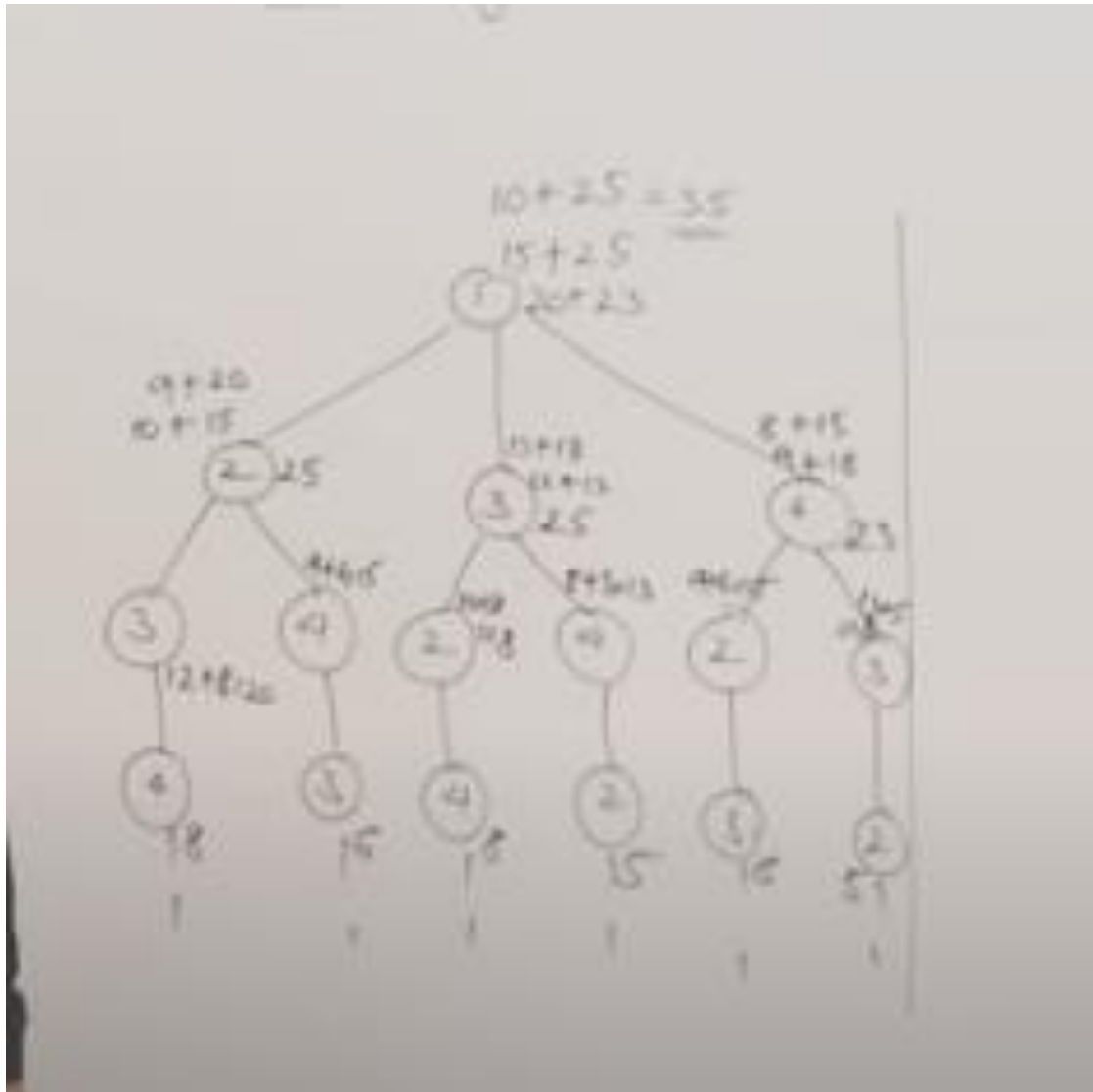
$$g(2, \{3, 4\}) = 25$$

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

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

$$A = \begin{bmatrix} 0 \\ 5 \\ 6 \\ 8 \end{bmatrix}$$





### Reliability Design:

*The reliability design problem is the designing of a system composed of several devices connected in series or parallel. **Reliability** means the probability to get the success of the device.*

Let's say, we have to set up a system consisting of  $D_1, D_2, D_3, \dots$ , and  $D_n$  devices, each device has some costs  $C_1, C_2, C_3, \dots, C_n$ . Each device has a reliability of 0.9 then the entire system has reliability which is equal to the **product** of the reliabilities of all devices i.e.,  $\pi r_i = (0.9)^4$ .

## String Editing:

Given two strings: Source String and Destination String.

- The Source String will never modify.
- It is required to convert the destination string into source string using three possible operations: INSERT/DELETE/UPDATE.
- The cost of operation: INSERT=1, DELETE=1, UPDATE=2.
- The objective is to perform string editing in minimum cost.
- The process of translation/mapping is carried out phase wise.

String Editing: Cost of conversion

- Let the source string be Y [y1... yj]
- Let the destination string be X [x1...xi]
- Cost of transformation: depends on character at Xi and Yj

Insert/Delete/Update operation

- (i) If no string in Y and character present in X, to make Y=X: Perform?
- (ii) If string is in Y and no string is present in X, to make Y=X: Perform?
- (iii) If string contents present in Y and X, then cost of transformation depends upon

$$cost(i, j) = \begin{cases} 0 & i = j = 0 \\ cost(i-1, 0) + D(x_i) & j = 0, i > 0 \\ cost(0, j-1) + I(y_j) & i = 0, j > 0 \\ cost'(i, j) & i > 0, j > 0 \end{cases}$$

$$\text{where } cost'(i, j) = \min \left\{ \begin{array}{l} cost(i-1, j) + D(x_i), \\ cost(i-1, j-1) + C(x_i, y_j), \\ cost(i, j-1) + I(y_j) \end{array} \right\}$$

-----

## Unit-IV

### Graphs:

#### Breadth First Search:

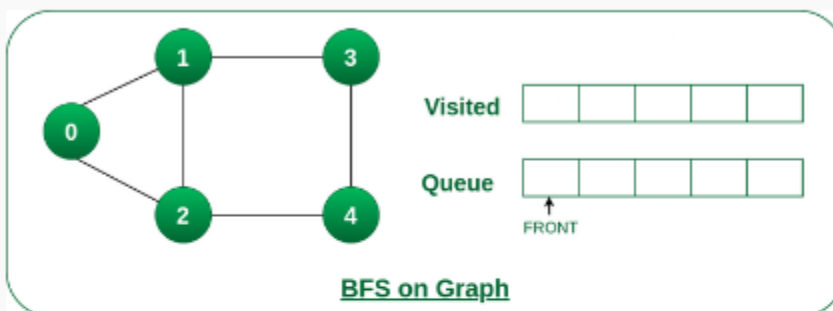
The Breadth First Search (BFS) algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.

Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.

To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

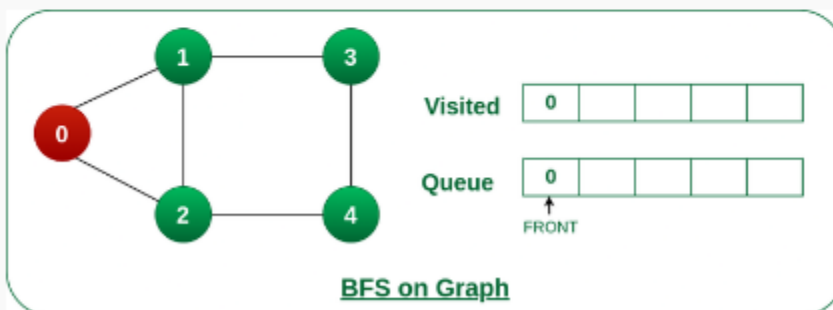
Ex:

*Step1: Initially queue and visited arrays are empty.*



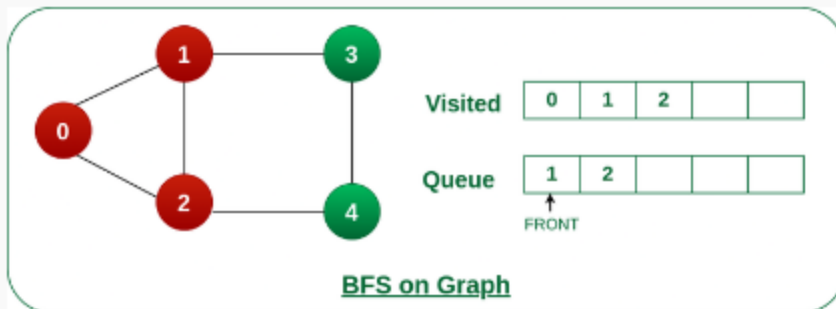
*Queue and visited arrays are empty initially.*

*Step2: Push node 0 into queue and mark it visited.*



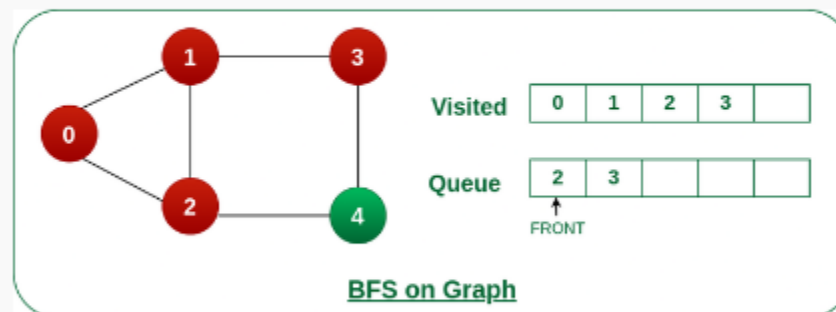
*Push node 0 into queue and mark it visited.*

**Step 3:** Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.



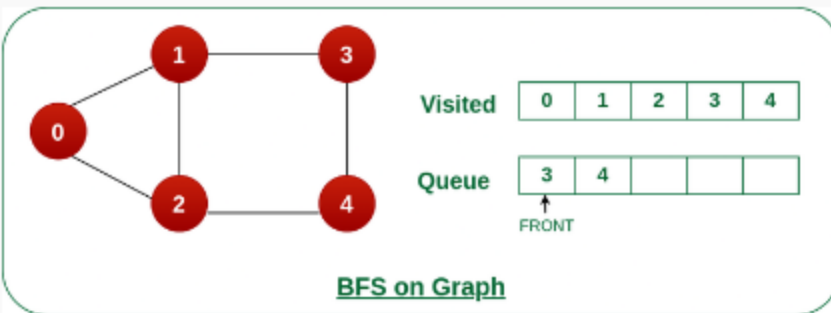
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

**Step 4:** Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



Remove node 1 from the front of queue and visited the unvisited neighbours and push

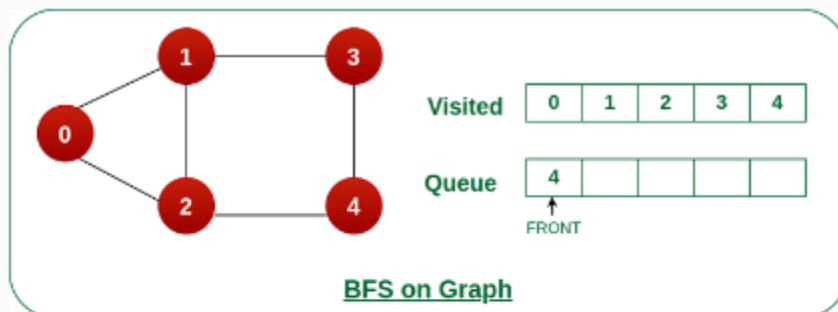
**Step 5:** Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

**Step 6:** Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

**Steps 7:** Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



*Remove node 4 from the front of queue and visit the unvisited neighbours  
and push them into queue.*

*Now, Queue becomes empty, So, terminate these  
process of iteration.*

-----

### **Depth First Search:**

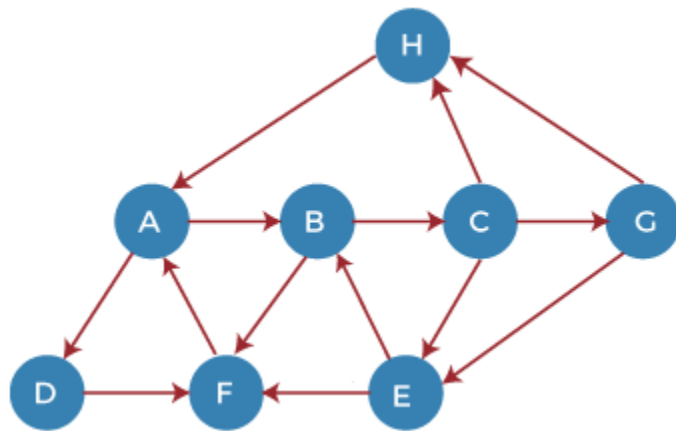
It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Ex:



### Adjacency Lists

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

Now, let's start examining the graph starting from Node H.

**Step 1** - First, push H onto the stack.

STACK: H

**Step 2** - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

Print: H | STACK: A

**Step 3** - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

Print: A

STACK: B, D

**Step 4** - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

Print: D

STACK: B, F

**Step 5** - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

Print: F

STACK: B

**Step 6** - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

Print: B

STACK: C

**Step 7** - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

Print: C

STACK: E, G

**Step 8** - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

Print: G

STACK: E

**Step 9** - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

Print: E

STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

### Complexity of Depth-first search algorithm

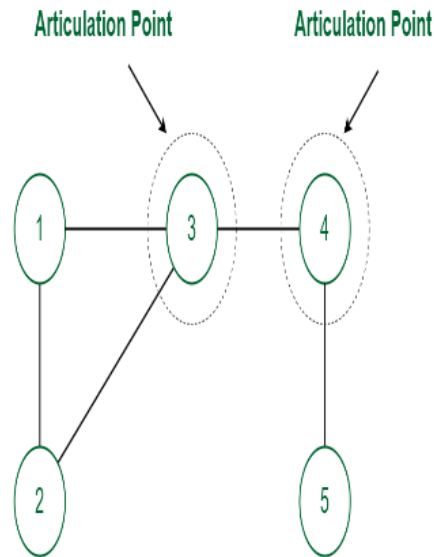
The time complexity of the DFS algorithm is  $O(V+E)$ , where V is the number of vertices and E is the number of edges in the graph.

The space complexity of the DFS algorithm is  $O(V)$ .

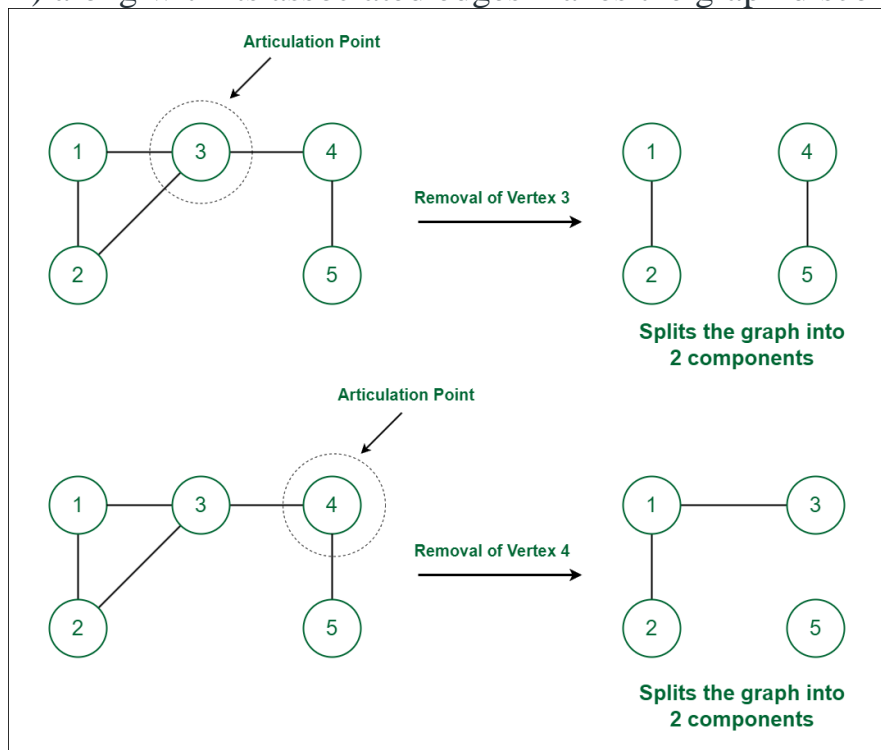
---

### **Articulation point:**

A vertex **v** is an **articulation point** (also called cut vertex) if removing **v** increases the number of connected components.



In the above graph vertex 3 and 4 are Articulation Points since the removal of vertex 3 (or 4) along with its associated edges makes the graph disconnected.

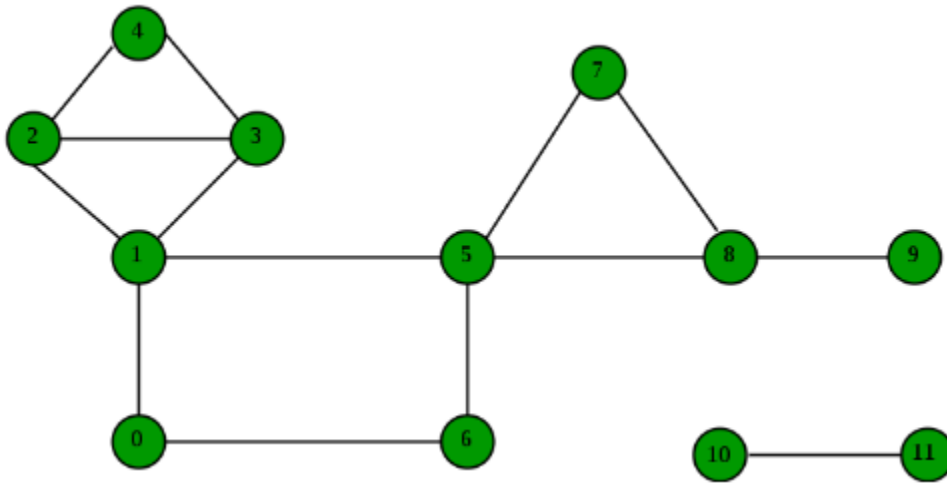


-----

### Connected and disconnected components:

If there is no Articulation Point in graph, then graph is biconnected and so there will be one biconnected component which is the graph itself.

Ex:



In above graph, following are the biconnected components:

- 4-2 3-4 3-1 2-3 1-2
- 8-9
- 8-5 7-8 5-7
- 6-0 5-6 1-5 0-1
- 10-11

-----

### Back Tracking:

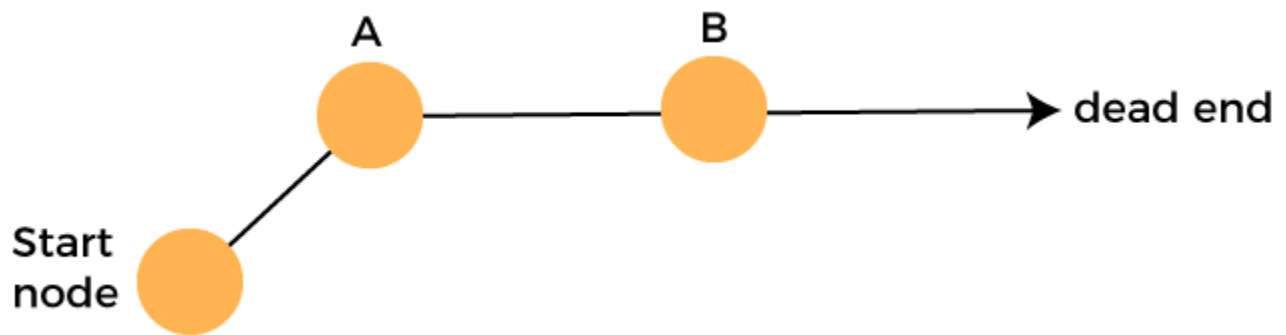
Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

- A piece of sufficient information is not available to make the best choice, so we use the backtracking strategy to try out all the possible solutions.
- Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In this case, we need to use the backtracking strategy.

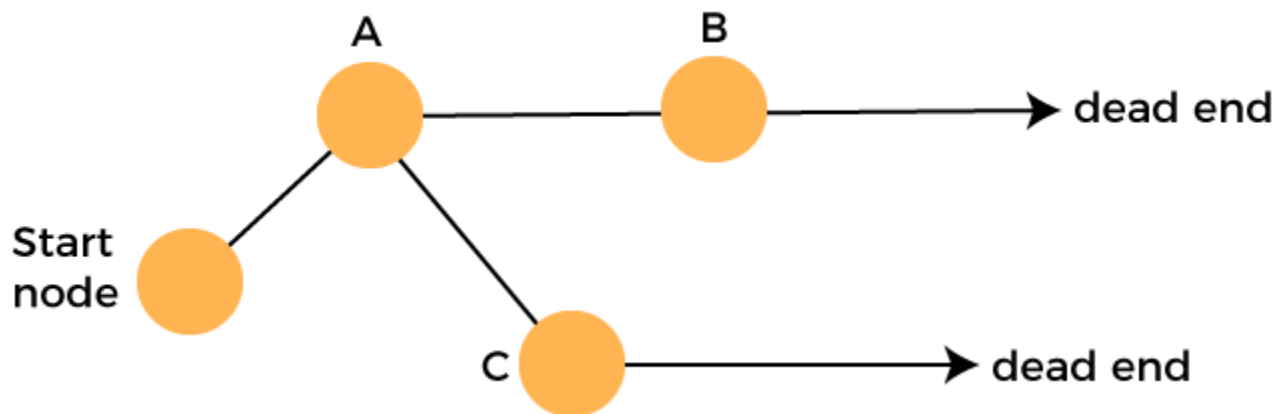
Backtracking is a systematic method of trying out various sequences of decisions until you find out that works.

Ex:

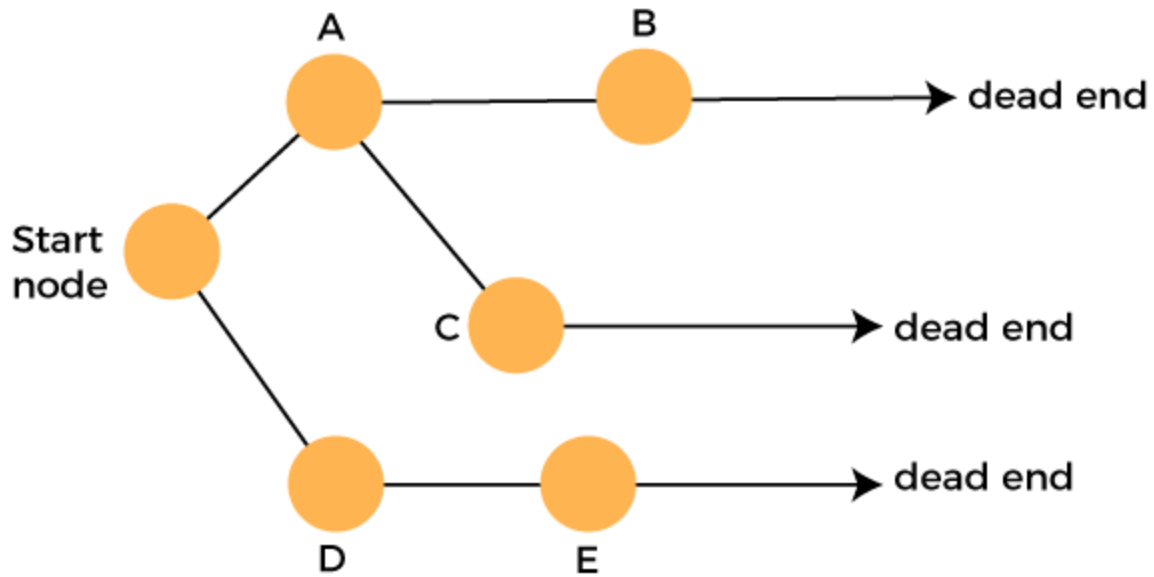
We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.



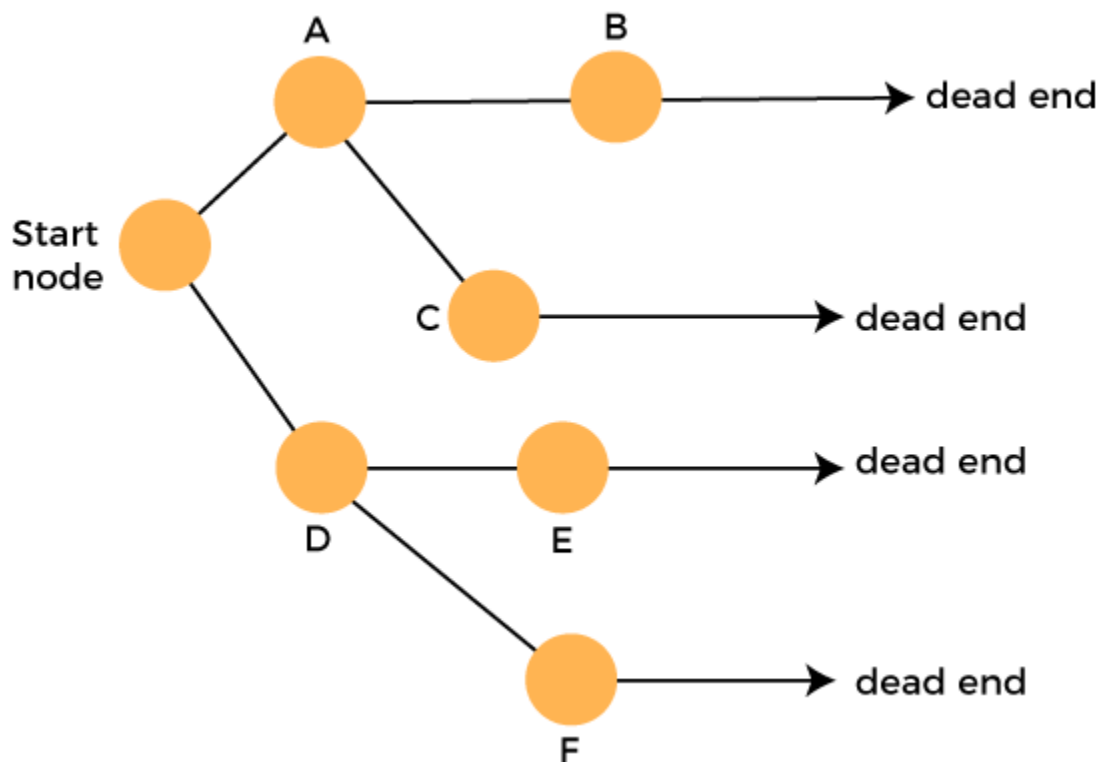
Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead-end, so again backtrack from node C to node A. We move from node A to the starting node.



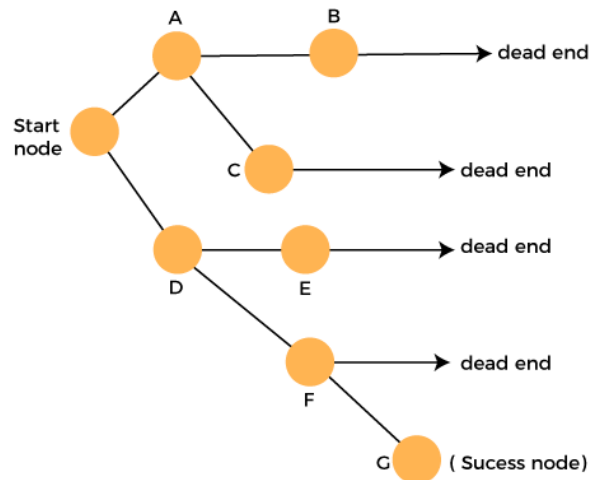
Now we will check any other path exists from the starting node. So, we move from start node to the node D. Since it is not a feasible solution so we move from node D to node E. The node E is also not a feasible solution. It is a dead end so we backtrack from node E to node D.



Suppose another path exists from node D to node F. So, we move from node D to node F. Since it is not a feasible solution and it's a dead-end, we check for another path from node F.



Suppose there is another path exists from the node F to node G so move from node F to node G. The node G is a success node.



### **Applications of Backtracking:**

- N Queens Problem
- Sum of subsets problem
- Graph coloring
- Hamiltonian cycles.

### **Terminology:**

**Problem state** is each node in the depth-first search tree

**State space** is the set of all paths from root node to other nodes

**Solution states** are the problem states  $s$  for which the path from the root node to  $s$

**Answer states** are that solution states  $s$  for which the path from root node to  $s$  defines a tuple that is a member of the set of solutions

**State space tree** is the tree organization of the solution space

**Live node** is a generated node for which all of the children have not been generated yet.

**E-node** is a live node whose children are currently being generated or explored

**Dead node** is a generated node that is not to be expanded any further

-----



### n-Queens Problem:

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for  $n = 1$ , the problem has a trivial solution, and no solution exists for  $n = 2$  and  $n = 3$ . So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.

	1	2	3	4
1				
2				
3				
4				

4x4 chessboard

Since, we have to place 4 queens such as  $q_1$   $q_2$   $q_3$  and  $q_4$  on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen  $q_1$  in the very first acceptable position (1, 1). Next, we put queen  $q_2$  so that both these queens do not attack each other.

We find that if we place  $q_2$  in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for  $q_2$  in column 3, i.e. (2, 3) but then no position is left for placing queen ' $q_3$ ' safely.

So we backtrack one step and place the queen ' $q_2$ ' in (2, 4), the next best possible solution. Then we obtain the position for placing ' $q_3$ ' which is (3, 2).

But later this position also leads to a dead end, and no place is found where 'q<sub>4</sub>' can be placed safely.

Then we have to backtrack till 'q<sub>1</sub>' and place it to (1, 2) and then all other queens are placed safely by moving q<sub>2</sub> to (2, 4), q<sub>3</sub> to (3, 1) and q<sub>4</sub> to (4, 3).

That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem.

For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.

	1	2	3	4
1			q <sub>1</sub>	
2	q <sub>2</sub>			
3				q <sub>3</sub>
4		q <sub>4</sub>		

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x<sub>1</sub>, x<sub>2</sub>, x<sub>3</sub>, x<sub>4</sub>) where x<sub>i</sub> represents the column on which queen "q<sub>i</sub>" is placed.

One possible solution for 8 queens problem is shown in fig:

	1	2	3	4	5	6	7	8
1				q <sub>1</sub>				
2						q <sub>2</sub>		
3								q <sub>3</sub>
4		q <sub>4</sub>						
5							q <sub>5</sub>	
6	q <sub>6</sub>							
7			q <sub>7</sub>					
8					q <sub>8</sub>			

1. Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l).
3. Then they are on same diagonal only **if**  $(i - j) = k - l$  or  $i + j = k + l$ .
4. The first equation implies that  $j - l = i - k$ .
5. The second equation implies that  $j - l = k - i$ .
6. Therefore, two queens lie on the duplicate diagonal **if** and only **if**  $|j-l|=|i-k|$

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs  $x_1, x_2, \dots, x_{k-1}$  and whether there is no other queen on the same diagonal.

-----

### Sum of Subsets:

Subset sum problem is the problem of finding a subset such that the sum of elements equals a given number. The backtracking approach generates all permutations in the worst case but in general, performs better than the recursive approach towards subset sum problem.

A subset A of n positive integers and a value sum (d) is given, find whether or not there exists any subset of the given set, the sum of whose elements is equal to the given value of sum.

### **Steps:**

1. Start with an empty set
2. Add the next element from the list to the set
3. If the subset is having sum M, then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set, then backtrack through the subset until we find the most suitable value.
5. If the subset is feasible (sum of subset < d) then go to step 2.
6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

Ex:

Input: set[] = {4, 16, 5, 23, 12}, sum = 9

Output = true

Subset {4, 5} has the sum equal to 9.

Ex:

Input:  $\text{set}[] = \{2, 3, 5, 6, 8, 10\}$ ,  $\text{sum} = 10$

Output = true

There are three possible subsets that have the sum equal to 10.

Subset1:  $\{5, 2, 3\}$

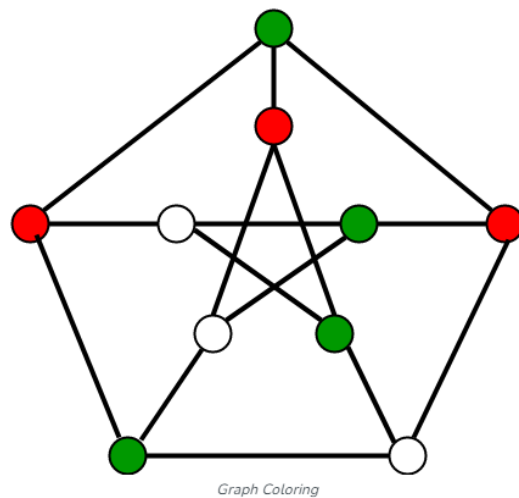
Subset2:  $\{2, 8\}$

Subset3:  $\{10\}$

-----

### Graph Coloring:

**Graph coloring** refers to the problem of **coloring vertices** of a graph in such a way that **no two adjacent** vertices have the **same color**. This is also called the **vertex coloring** problem. If coloring is done using at most  $m$  colors, it is called  $m$ -coloring.



The minimum number of colors needed to color a graph is called its chromatic number.

Ex:

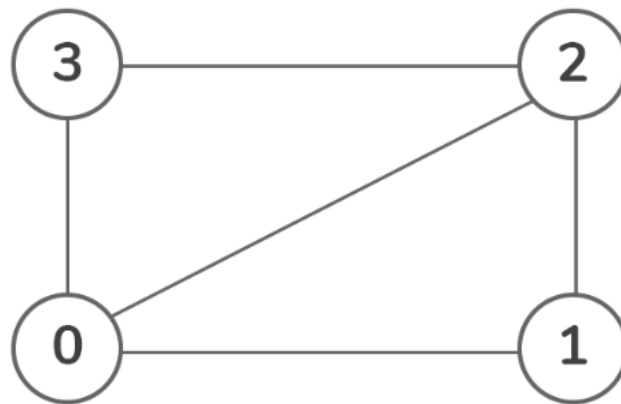
The following can be colored a minimum of 2 colors.

Graph coloring problem is both, a **decision problem** as well as an **optimization problem**.

- A decision problem is stated as, “With given  $M$  colors and graph  $G$ , whether a such color scheme is possible or not?”.
- The optimization problem is stated as, “Given  $M$  colors and graph  $G$ , find the minimum number of colors required for graph coloring.”

- A graph represented in 2D array format of size  $V * V$  where  $V$  is the number of vertices in graph and the 2D array is the adjacency matrix representation and value  $graph[i][j]$  is 1 if there is a direct edge from  $i$  to  $j$ , otherwise the value is 0.
- An integer  $m$  that denotes the maximum number of colors which can be used in graph coloring.
- Consider the input as shown in the image below:

**Ex:**



**The above graph can be represented as follows:**

```

graph[4][4] = {
    { 0, 1, 1, 1 },
    { 1, 0, 1, 0 },
    { 1, 1, 0, 1 },
    { 1, 0, 1, 0 },
};
  
```

**Consider  $m = 3$ .**

- **Return array color of size  $V$  that has numbers from 1 to  $m$ . Note that  $color[i]$  represents the color assigned to the  $i$ th vertex.**
- **Return false if the graph cannot be colored with  $m$  colors.**

**Naive Approach:**

- The brute force approach would be to generate all possible combinations (or configurations) of colors.
- After generating a configuration, check if the adjacent vertices have the same colour or not. If the conditions are met, add the combination to the result and break the loop.

- Since each node can be colored by using any of the  $m$  colors, the total number of possible color configurations are  $m^V$ . The complexity is exponential which is very huge.

### Using Backtracking:

- By using the backtracking method, the main idea is to assign colors one by one to different vertices right from the first vertex (vertex 0).
- Before color assignment, check if the adjacent vertices have same or different color by considering already assigned colors to the adjacent vertices.
  - If the color assignment does not violate any constraints, then we mark that color as part of the result. If color assignment is not possible then backtrack and return false.

**Time Complexity:**  $O(m^V)$ . There is a total of  $O(m^V)$  combinations of colors. The upper bound time complexity remains the same but the average time taken will be less.

**Auxiliary Space:**  $O(V)$ . The recursive Stack of the graph coloring function will require  $O(V)$  space.

-----

### Hamiltonian Cycles:

**Hamiltonian Cycle or Circuit** in a graph  $G$  is a cycle that visits every vertex of  $G$  exactly once and returns to the starting vertex.

- If graph contains a Hamiltonian cycle, it is called **Hamiltonian graph** otherwise it is **non-Hamiltonian**.
- Finding a Hamiltonian Cycle in a graph is a well-known NP-complete problem, which means that there's no known efficient algorithm to solve it for all types of graphs. However, it can be solved for small or specific types of graphs.

The Hamiltonian Cycle problem has practical applications in various fields, such as **logistics, network design, and computer science**.

**Hamiltonian Path** in a graph  $G$  is a path that visits every vertex of  $G$  exactly once and **Hamiltonian Path** doesn't have to return to the starting vertex. It's an open path.

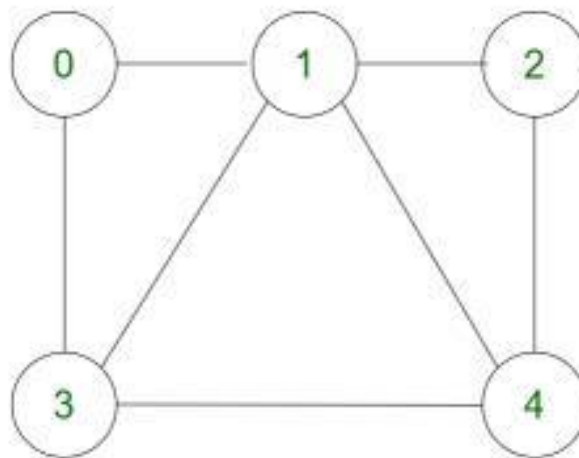
- Similar to the **Hamiltonian Cycle** problem, finding a **Hamiltonian Path** in a general graph is also NP-complete and can be challenging. However, it is often a more easier problem than finding a Hamiltonian Cycle.

- Hamiltonian Paths have applications in various fields, such as **finding optimal routes in transportation networks, circuit design, and graph theory research.**

**Ex:**

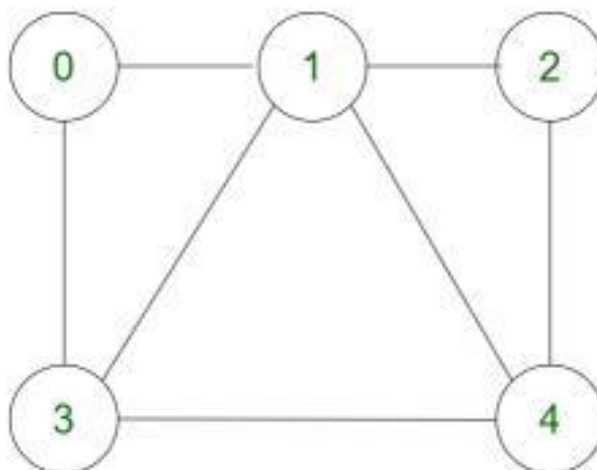
**Problems Statement:** Given an undirected graph, the task is to determine whether the graph contains a Hamiltonian cycle or not. If it contains, then prints the path.

**Input:**  $graph[][] = \{\{0, 1, 0, 1, 0\}, \{1, 0, 1, 1, 1\}, \{0, 1, 0, 0, 1\}, \{1, 1, 0, 0, 1\}, \{0, 1, 1, 1, 0\}\}$



**Output:**  $\{0, 1, 2, 4, 3, 0\}$ .

**Input:**  $graph[][] = \{\{0, 1, 0, 1, 0\}, \{1, 0, 1, 1, 1\}, \{0, 1, 0, 0, 1\}, \{1, 1, 0, 0, 0\}, \{0, 1, 1, 0, 0\}\}$



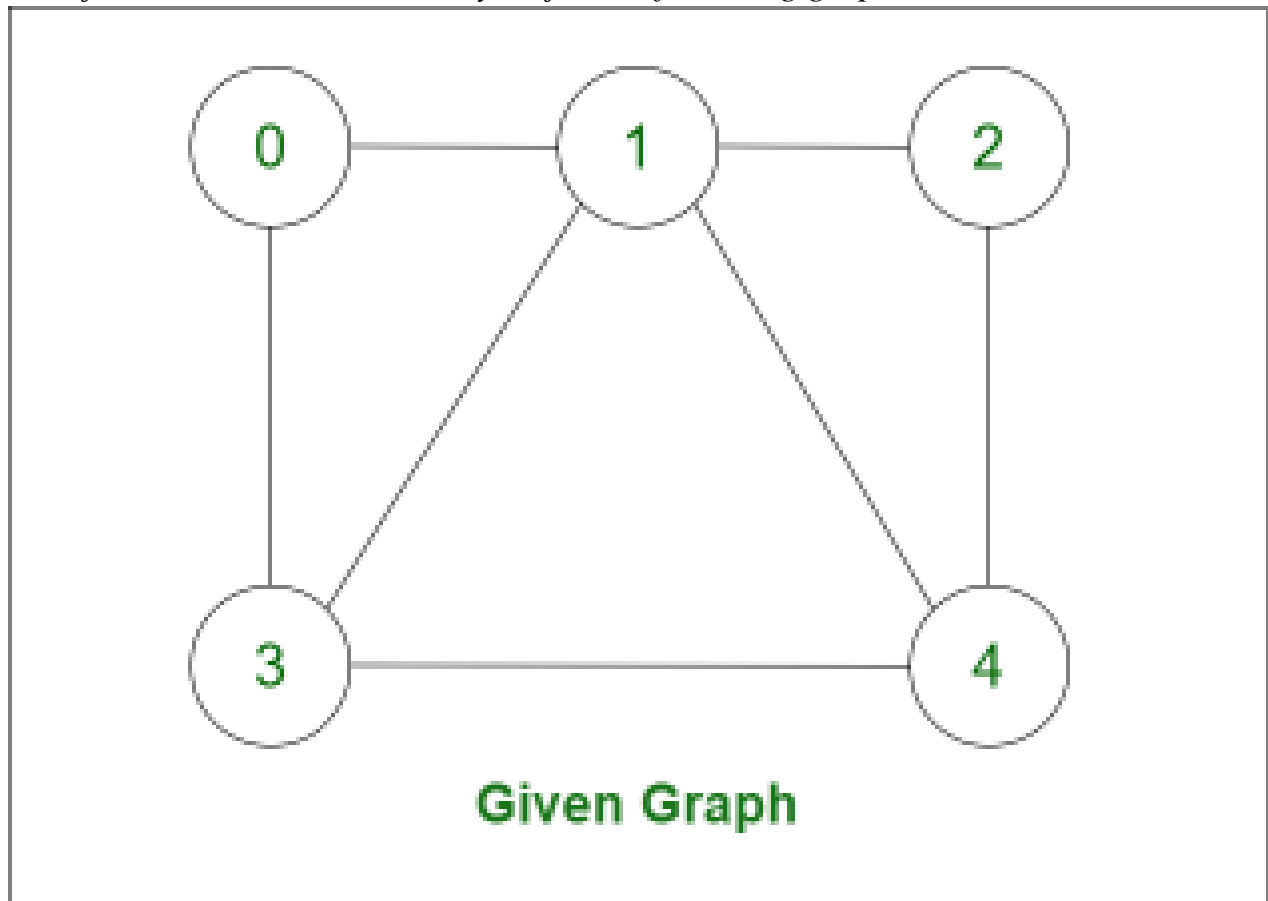
**Output:** *Solution does not exist*

## Hamiltonian Cycle using Backtracking Algorithm:

Create an empty path array and add vertex **0** to it. Add other vertices, starting from the vertex **1**. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return **false**.

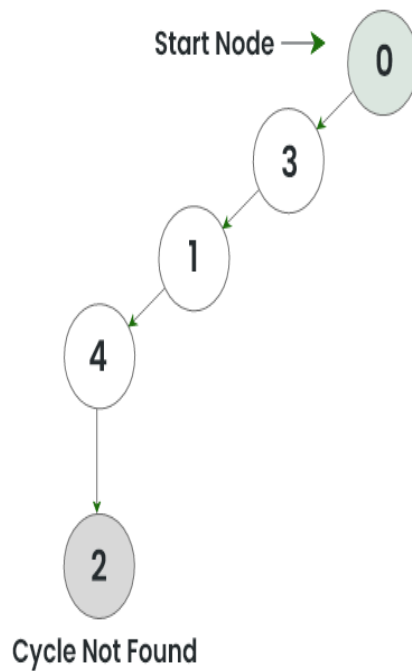
Ex:

*Let's find out the Hamiltonian cycle for the following graph:*



- *Start with the node 0 .*
- *Apply DFS for finding the Hamiltonian path.*
- *When base case reach (i.e. **total no of node traversed == V (total vertex)**):*
  - *Check weather current node is a neighbour of starting node.*
  - *As node 2 and node 0 are not neighbours of each other so return from it.*



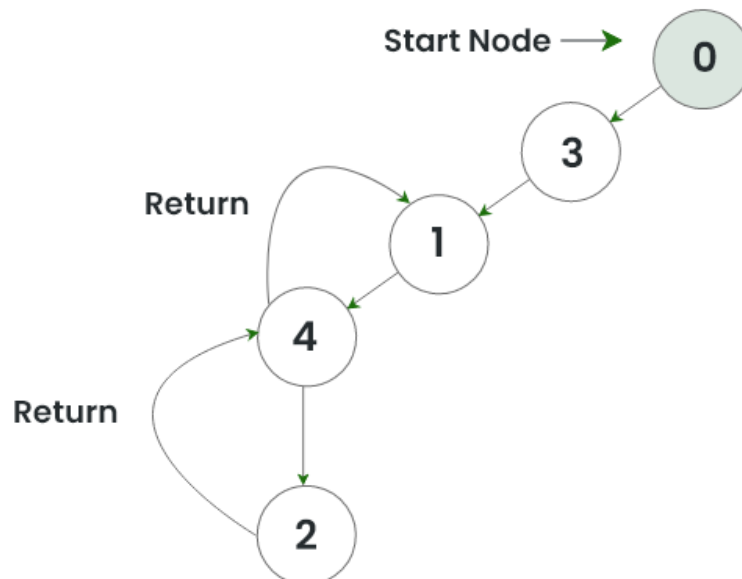


Hamiltonian Cycle



Starting from start node 0 calling DFS

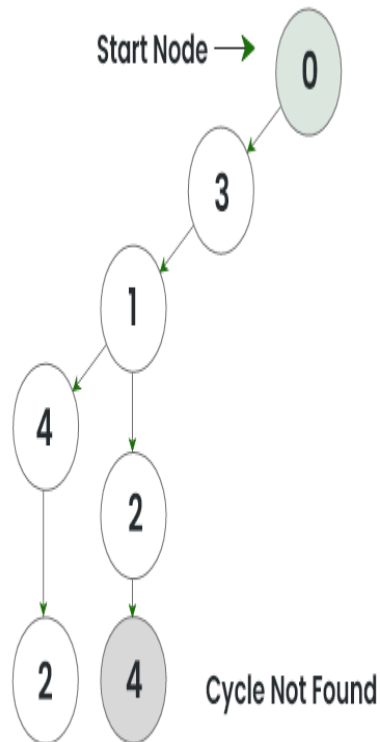
- As cycle is not found in path {0, 3, 1, 4, 2}. So, return from node 2, node 4.



Hamiltonian Cycle

- Now, explore another option for node 1 (i.e node 2)
- When it hits the base condition again check for Hamiltonian cycle

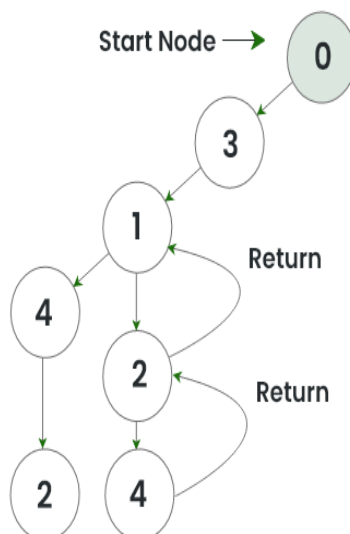
- As node 4 is not the neighbour of node 0, again cycle is not found then return.



## Hamiltonian Cycle



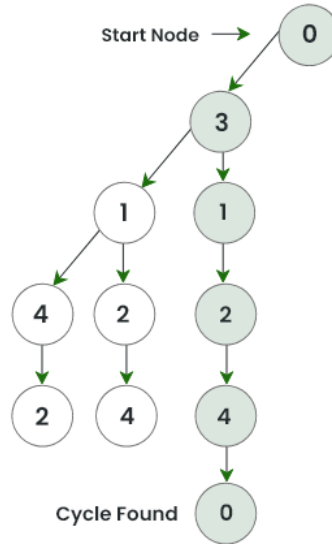
- Return from node 4, node 2, node 1.



## Hamiltonian Cycle



- Now, explore other options for node 3.



## Hamiltonian Cycle



Found the Hamiltonian Cycle

- In the Hamiltonian path  $\{0,3,4,2,1,0\}$  we get cycle as node 1 is the neighbour of node 0.
- So print this cyclic path .
- This is our Hamiltonian cycle.

-----

## Unit-V

### Branch and Bound

#### General Method

Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree. It is used for solving the optimization problems and minimization problems. If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem.

**Ex:**

$P = \{10, 5, 8, 3\}$

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

The above are jobs, problems and problems given. We can write the solutions in two ways which are given below:

Suppose we want to perform the jobs  $j_1$  and  $j_2$  then the solution can be represented in two ways:

The first way of representing the solutions is the subsets of jobs.

$S_1 = \{j_1, j_4\}$

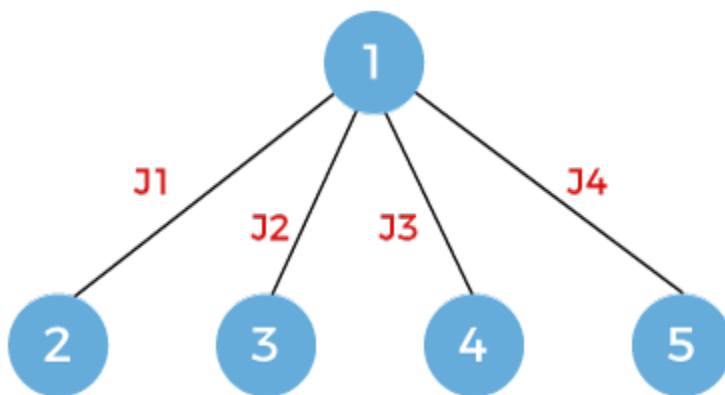
The second way of representing the solution is that first job is done, second and third jobs are not done, and fourth job is done.

$S_2 = \{1, 0, 0, 1\}$

The solution  $s_1$  is the variable-size solution while the solution  $s_2$  is the fixed-size solution.

**First, we will see the subset method where we will see the variable size.**

**First method:**

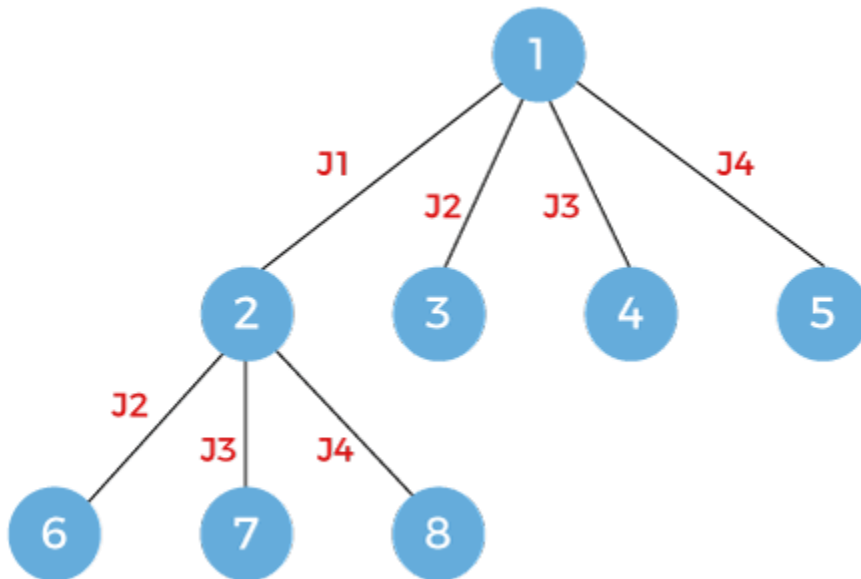


In this case, we first consider the first job, then second job, then third job and finally we consider the last job.

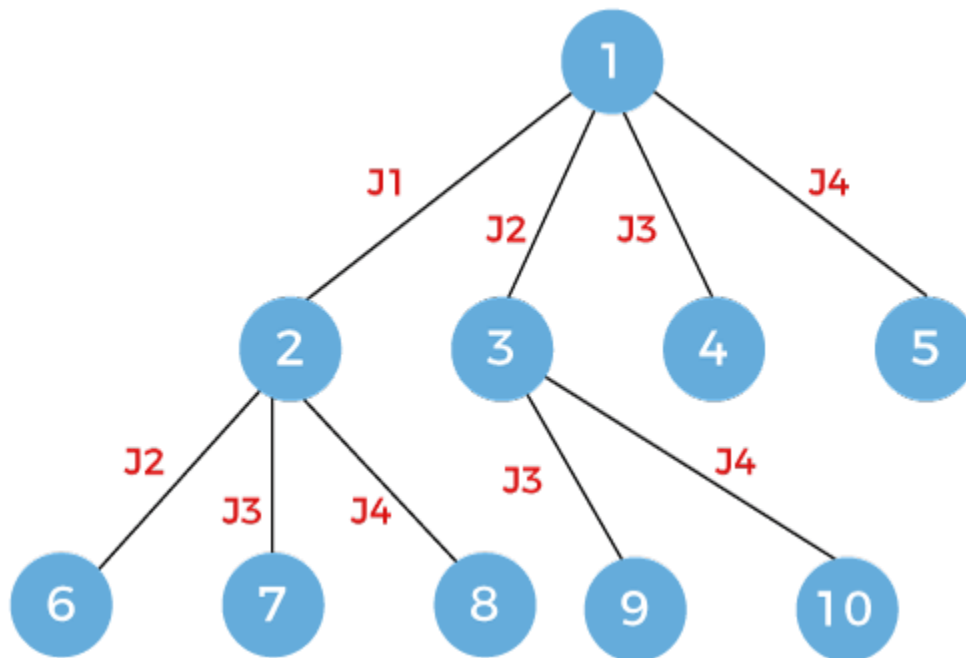
As we can observe in the above figure that the breadth first search is performed but not the depth first search. Here we move breadth wise for exploring the solutions.

In backtracking, we go depth-wise whereas in branch and bound, we go breadth wise.

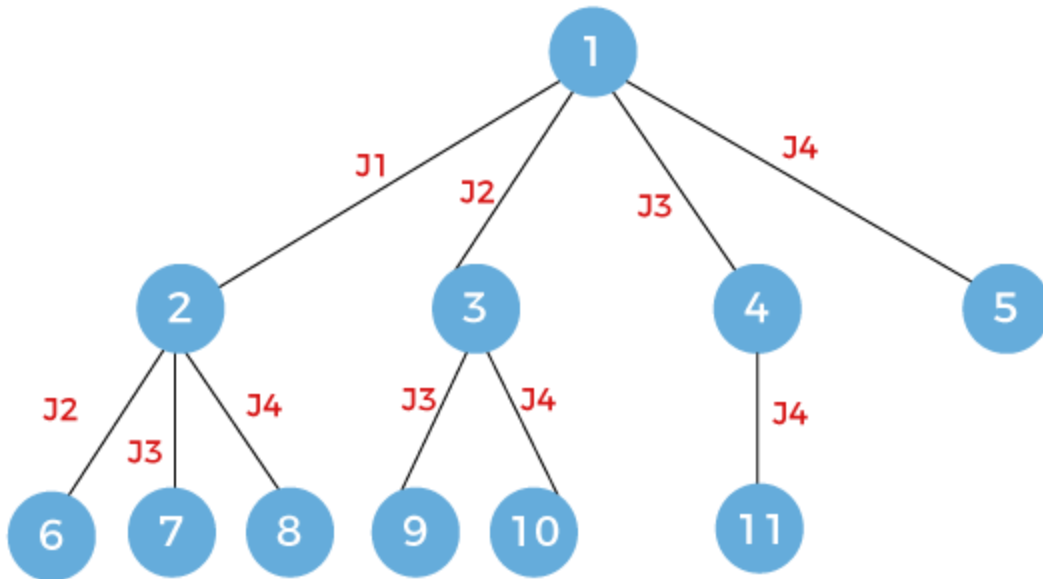
Now one level is completed. Once I take first job, then we can consider either j2, j3 or j4. If we follow the route then it says that we are doing jobs j1 and j4 so we will not consider jobs j2 and j3.



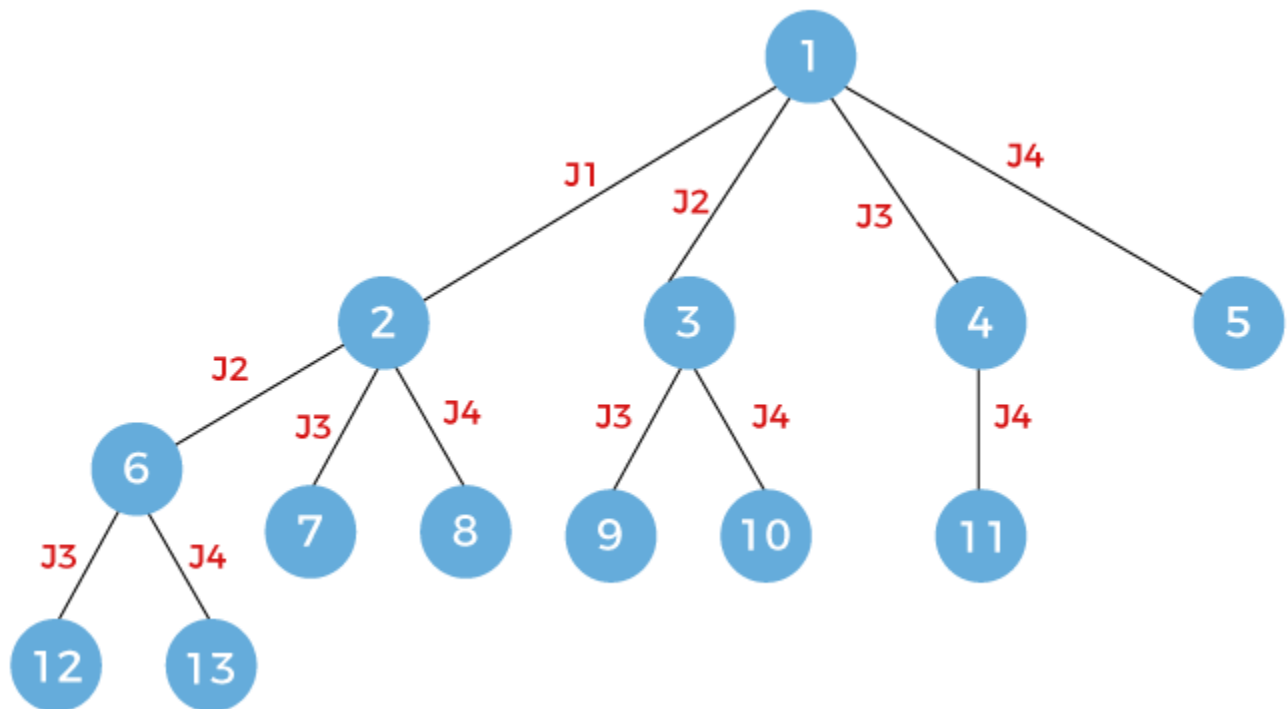
Now we will consider the node 3. In this case, we are doing job j2 so we can consider either job j3 or j4. Here, we have discarded the job j1.



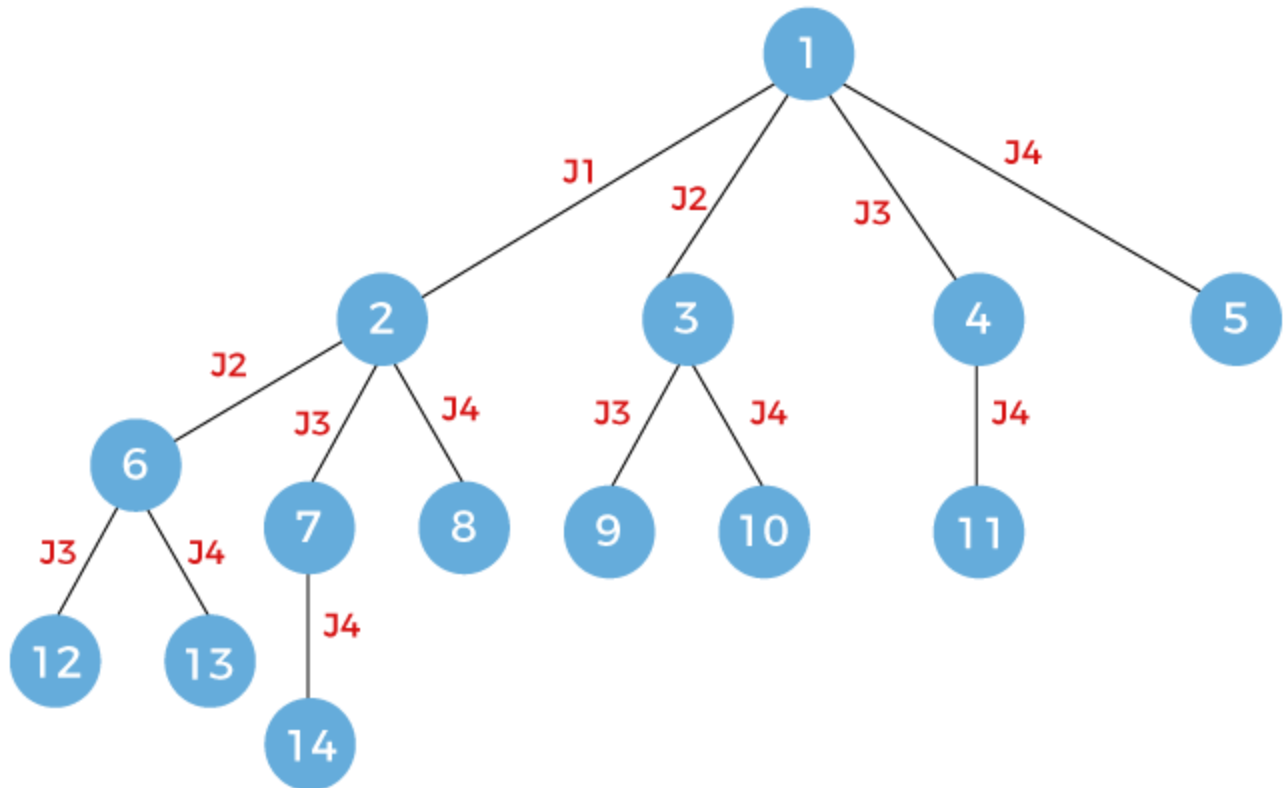
Now we will expand the node 4. Since here we are doing job j3 so we will consider only job j4.



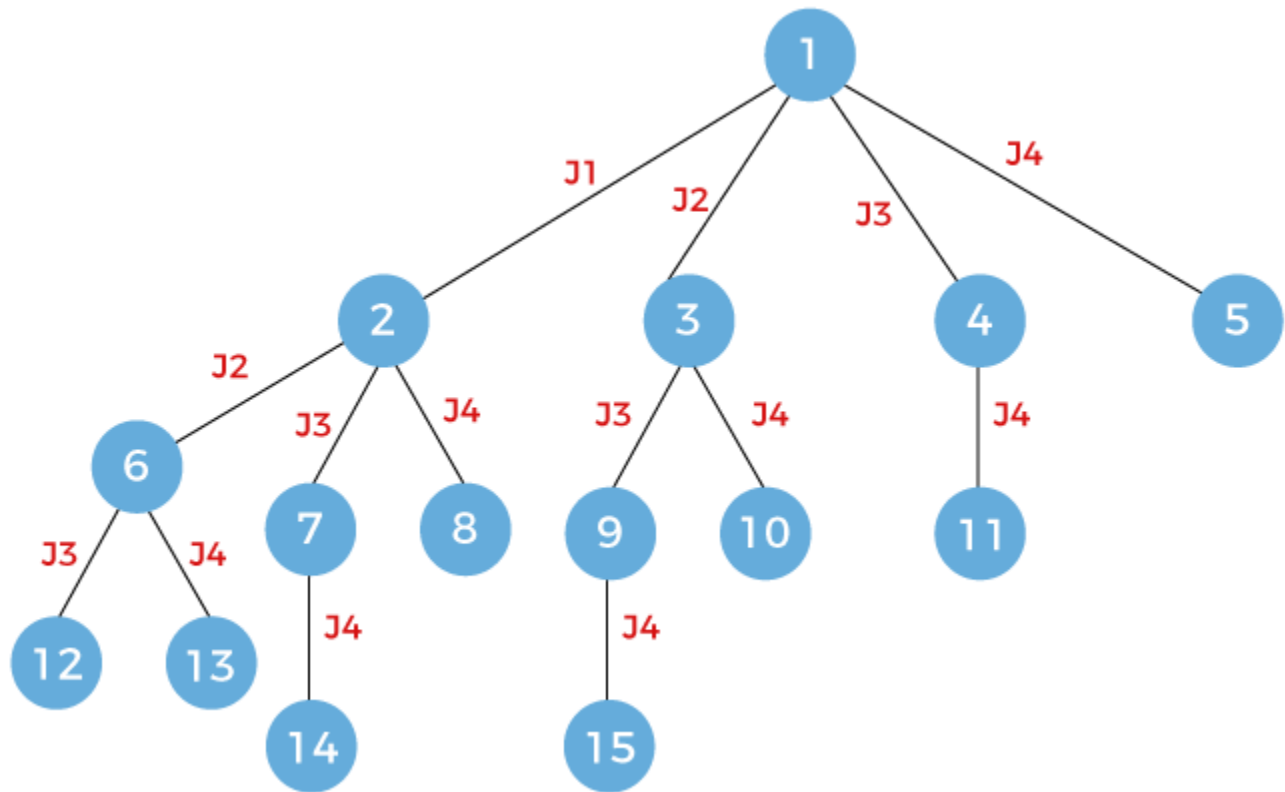
Now we will expand node 6, and here we will consider the jobs j3 and j4.



Now we will expand node 7 and here we will consider job j4.



Now we will expand node 9, and here we will consider job j4.



The last node, i.e., node 12 which is left to be expanded. Here, we consider job j4.  
The above is the state space tree for the solution  $s1 = \{j1, j4\}$

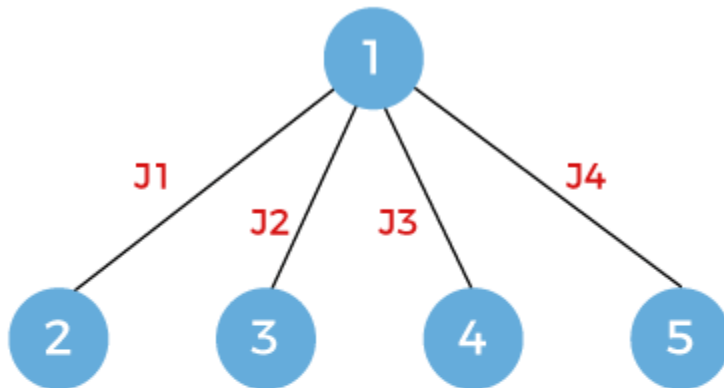
### Second method:

We will see another way to solve the problem to achieve the solution s1.

First, we consider the node 1 shown as below:

Now, we will expand the node 1. After expansion, the state space tree would be appeared as:

On each expansion, the node will be pushed into the stack shown as below:

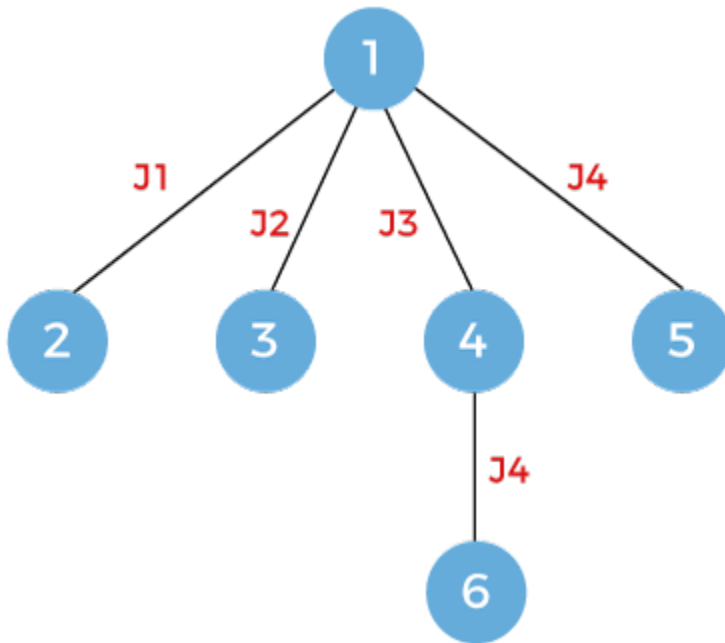


Now the expansion would be based on the node that appears on the top of the stack. Since the node 5 appears on the top of the stack, so we will expand the node 5. We will pop out the node 5 from the stack. Since the node 5 is in the last job, i.e., j4 so there is no further scope of expansion.

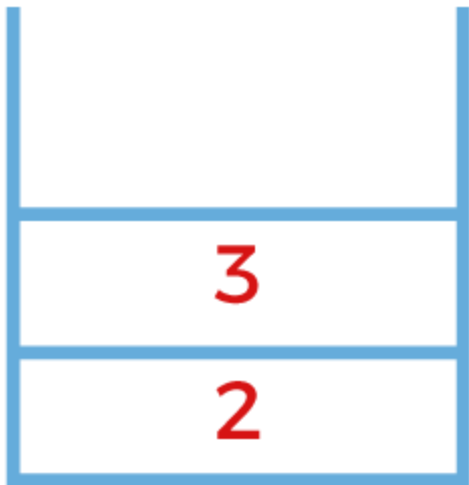




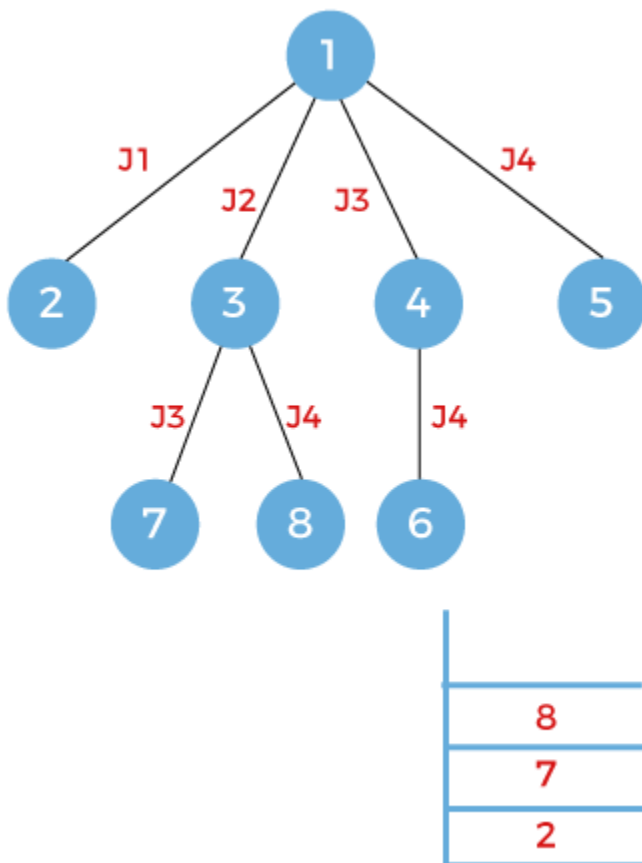
The next node that appears on the top of the stack is node 4. Pop out the node 4 and expand. On expansion, job j4 will be considered and node 6 will be added into the stack shown as below:



The next node is 6 which is to be expanded. Pop out the node 6 and expand. Since the node 6 is in the last job, i.e., j4 so there is no further scope of expansion.



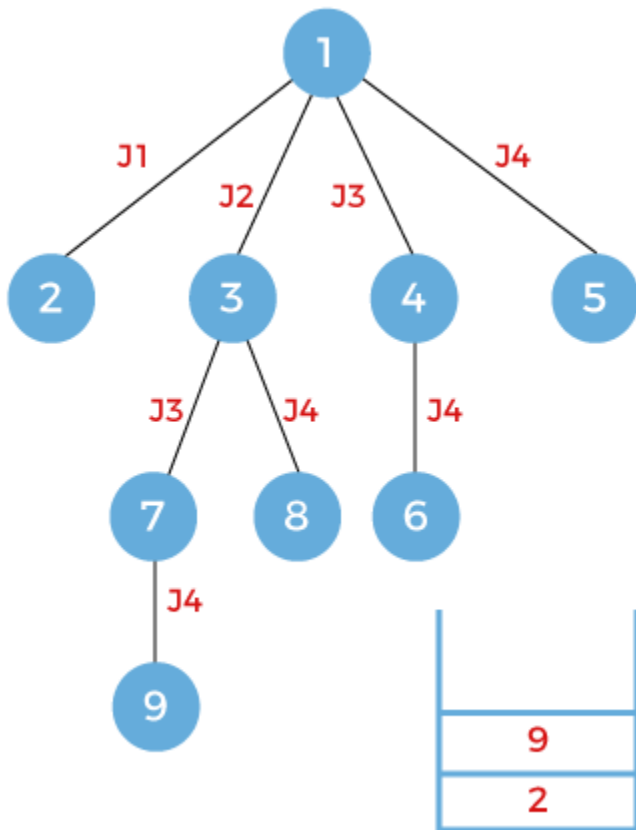
The next node to be expanded is node 3. Since the node 3 works on the job j2 so node 3 will be expanded to two nodes, i.e., 7 and 8 working on jobs 3 and 4 respectively. The nodes 7 and 8 will be pushed into the stack shown as below:



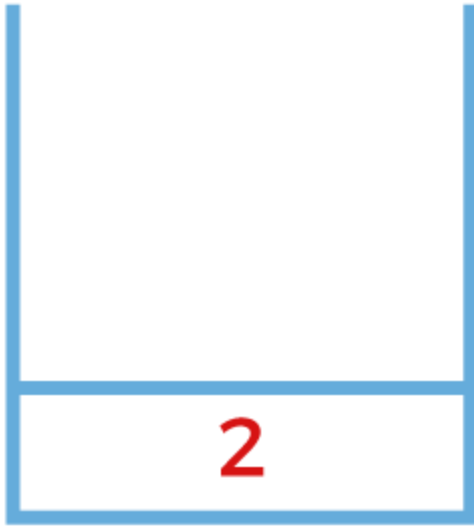
The next node that appears on the top of the stack is node 8. Pop out the node 8 and expand. Since the node 8 works on the job j4 so there is no further scope for the expansion.



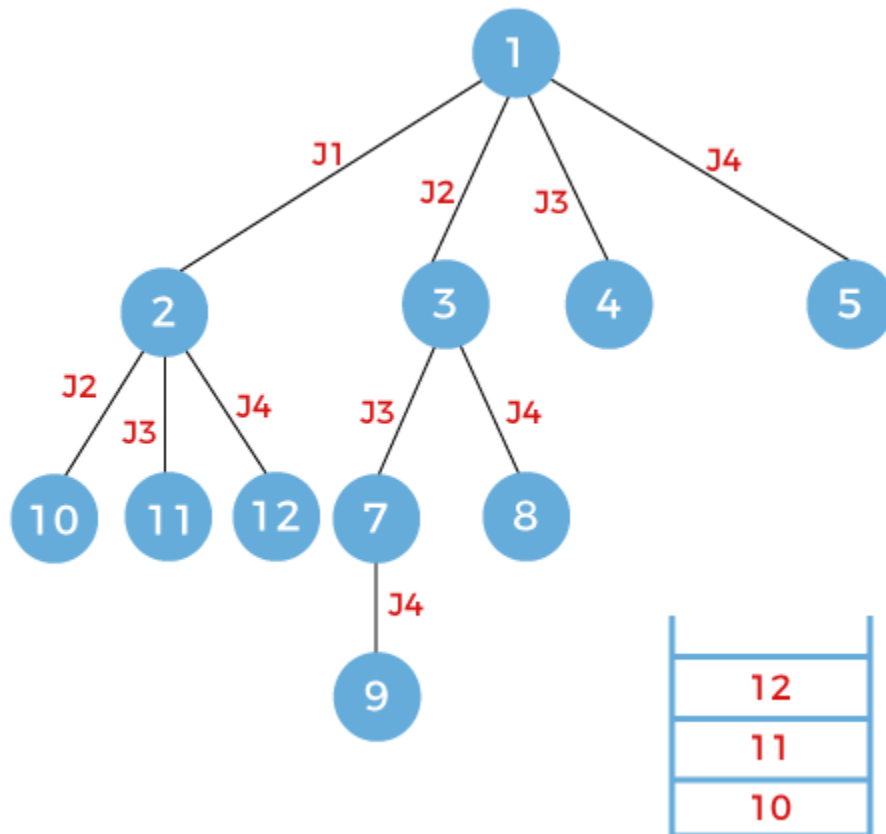
The next node that appears on the top of the stack is node 7. Pop out the node 7 and expand. Since the node 7 works on the job j3 so node 7 will be further expanded to node 9 that works on the job j4 as shown as below and the node 9 will be pushed into the stack.



The next node that appears on the top of the stack is node 9. Since the node 9 works on the job 4 so there is no further scope for the expansion.



The next node that appears on the top of the stack is node 2. Since the node 2 works on the job j1 so it means that the node 2 can be further expanded. It can be expanded upto three nodes named as 10, 11, 12 working on jobs j2, j3, and j4 respectively. There newly nodes will be pushed into the stack shown as below:



In the above method, we explored all the nodes using the stack that follows the LIFO principle.

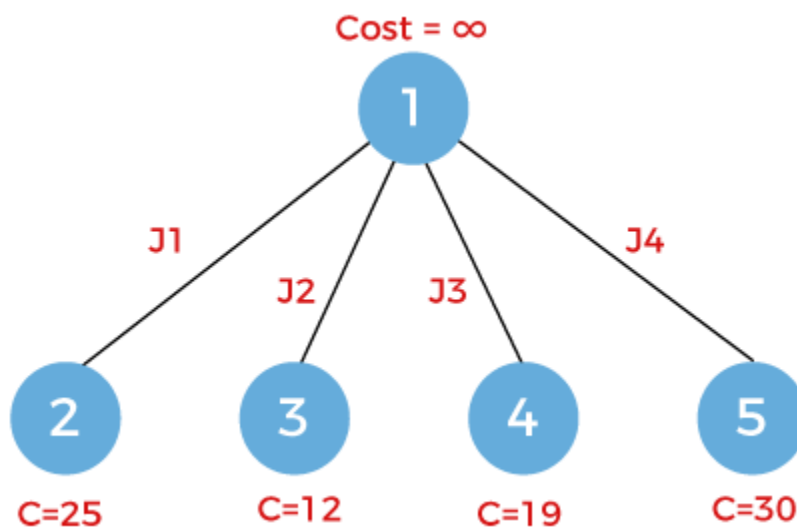
-----

### Least Cost Branch and Bound:

There is one more method that can be used to find the solution and that method is Least cost branch and bound. In this technique, nodes are explored based on the cost of the node. The cost of the node can be defined using the problem and with the help of the given problem, we can define the cost function. Once the cost function is defined, we can define the cost of the node.

**Let's first consider the node 1 having cost infinity shown as below:**

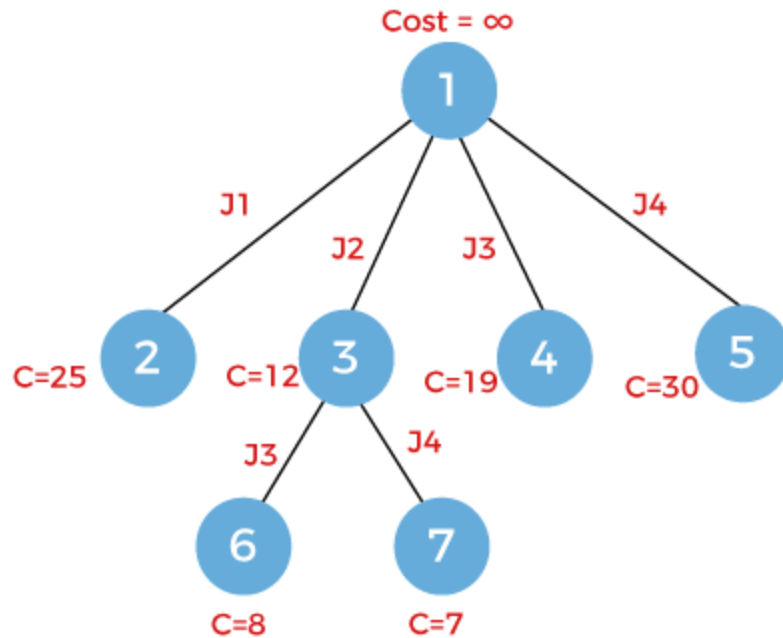
Now we will expand the node 1. The node 1 will be expanded into four nodes named as 2, 3, 4 and 5 shown as below:



**Let's assume that cost of the nodes 2, 3, 4, and 5 are 25, 12, 19 and 30 respectively.**

Since it is the least cost branch n bound, so we will explore the node which is having the least cost. In the above figure, we can observe that the node with a minimum cost is node 3. So, we will explore the node 3 having cost 12.

Since the node 3 works on the job j2 so it will be expanded into two nodes named as 6 and 7 shown as below:

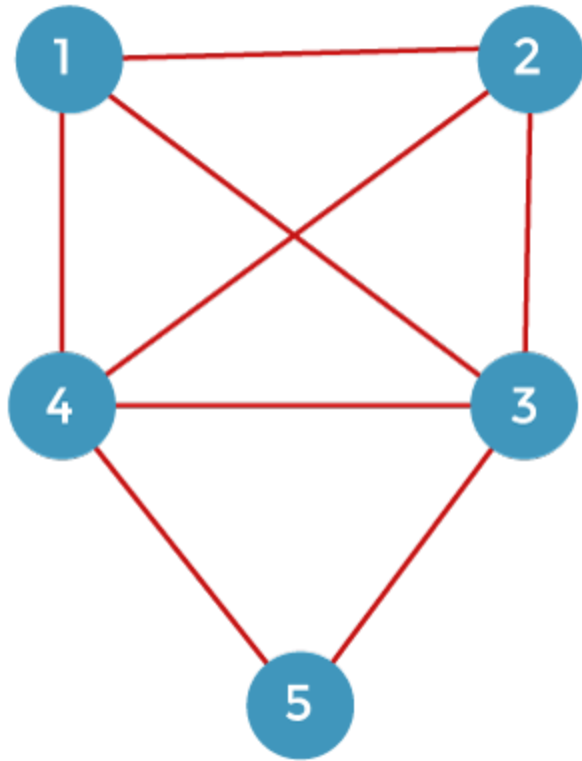


The node 6 works on job j3 while the node 7 works on job j4. The cost of the node 6 is 8 and the cost of the node 7 is 7. Now we have to select the node which is having the minimum cost. The node 7 has the minimum cost so we will explore the node 7. Since the node 7 already works on the job j4 so there is no further scope for the expansion.

-----

### **Travelling sales person problem:**

Given the vertices, the problem here is that we have to travel each vertex exactly once and reach back to the starting point. Consider the below graph:



As we can observe in the above graph that there are 5 vertices given in the graph. We have to find the shortest path that goes through all the vertices once and returns back to the starting vertex. We mainly consider the starting vertex as 1, then traverse through the vertices 2, 3, 4, and 5, and finally return to vertex 1.

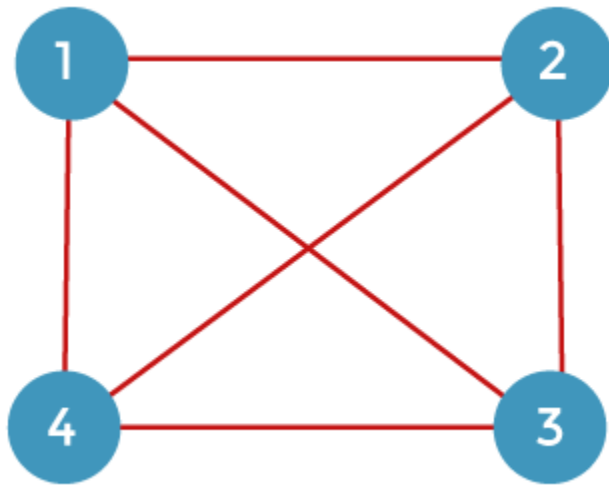
**The adjacent matrix of the problem is given below:**

	1	2	3	4	5
1	$\infty$	20	30	10	11
2	15	$\infty$	30	10	11
3	3	5	$\infty$	2	4
4	19	6	18	$\infty$	3
5	16	4	7	16	$\infty$

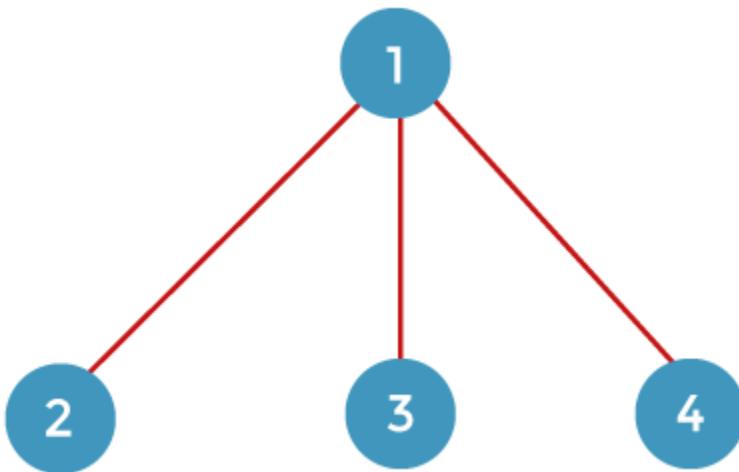
Now we look at how this problem can be solved using the branch n bound.

**Let's first understand the approach then we solve the above problem.**

The graph is given below, which has four vertices:

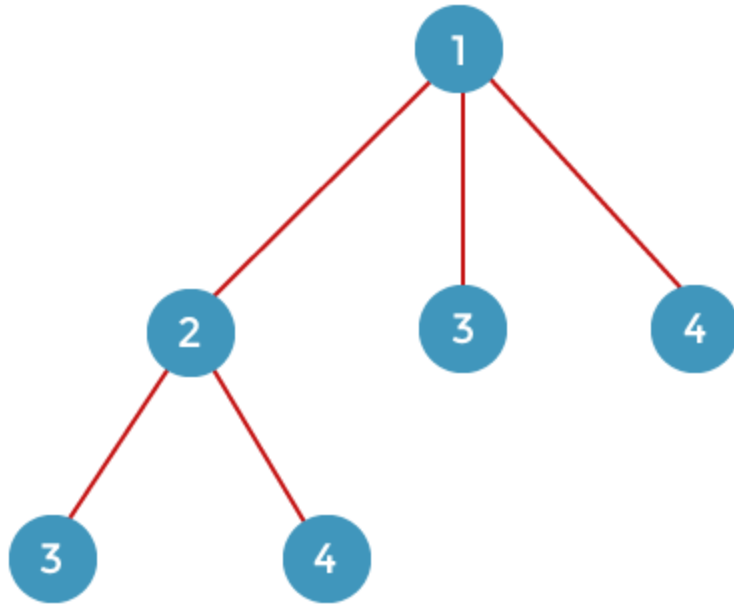


Suppose we start travelling from vertex 1 and return back to vertex 1. There are various ways to travel through all the vertices and returns to vertex 1. We require some tools that can be used to minimize the overall cost. To solve this problem, we make a state space tree. From the starting vertex 1, we can go to either vertices 2, 3, or 4, as shown in the below diagram.

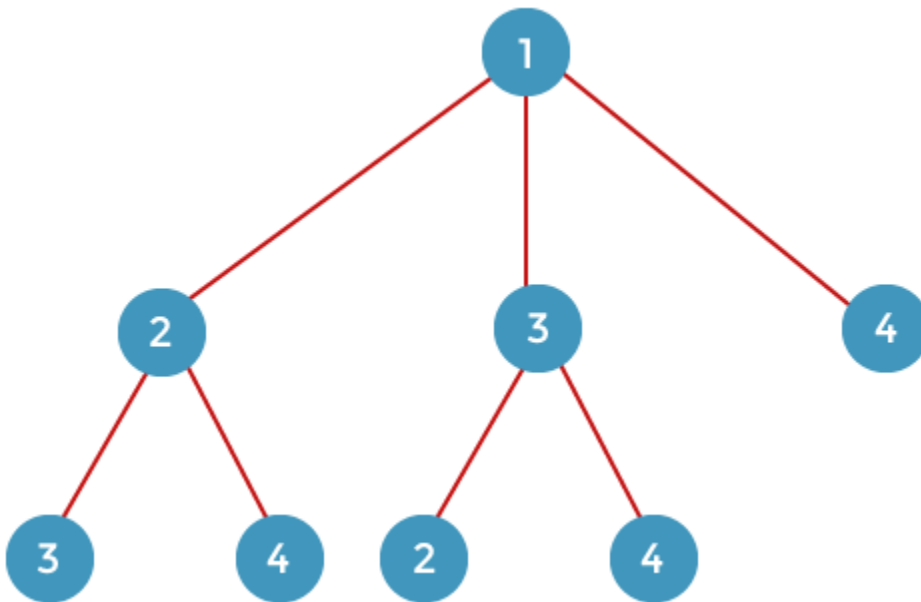


From vertex 2, we can go either to vertex 3 or 4. If we consider vertex 3, we move to the remaining vertex, i.e., 4. If we consider the vertex 4 shown in the below diagram:

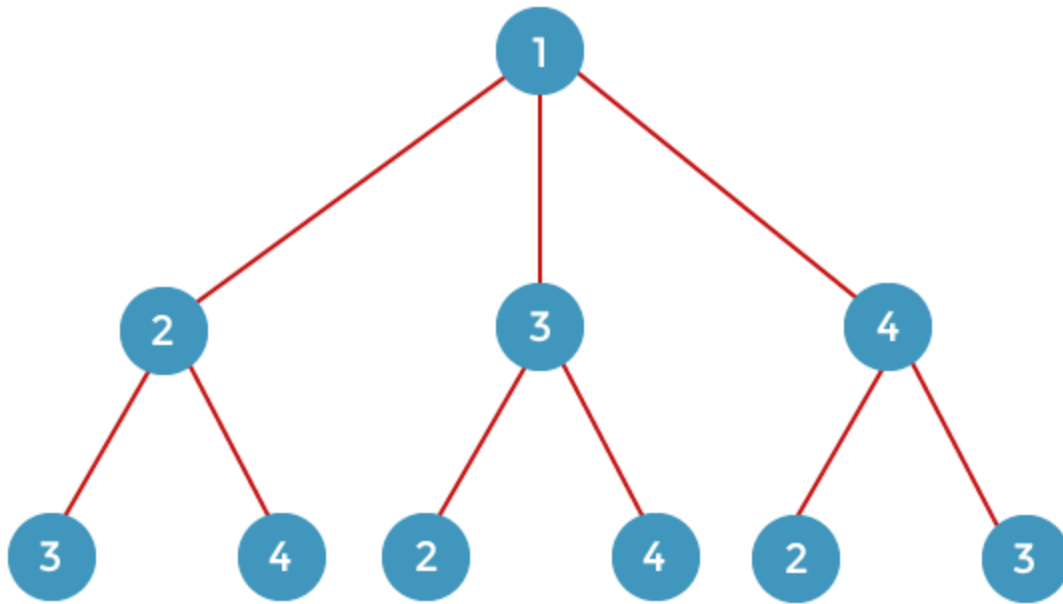




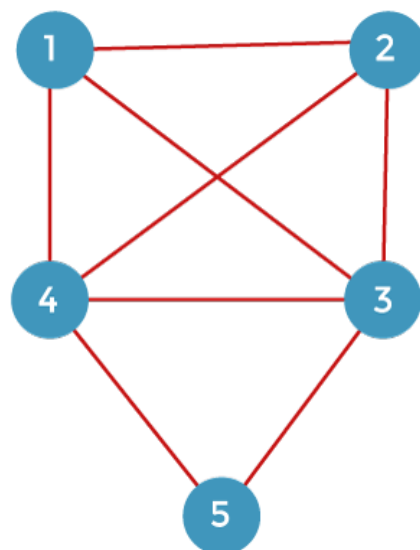
From vertex 3, we can go to the remaining vertices, i.e., 2 or 4. If we consider the vertex 2, then we move to remaining vertex 4, and if we consider the vertex 4 then we move to the remaining vertex, i.e., 3 shown in the below diagram:



From vertex 4, we can go to the remaining vertices, i.e., 2 or 3. If we consider vertex 2, then we move to the remaining vertex, i.e., 3, and if we consider the vertex 3, then we move to the remaining vertex, i.e., 2 shown in the below diagram:



The above is the complete state space tree. The state space tree shows all the possibilities. Backtracking and branch n bound both use the state space tree, but their approach to solve the problem is different. Branch n bound is a better approach than backtracking as it is more efficient. In order to solve the problem using branch n bound, we use a level order. First, we will observe in which order, the nodes are generated. While creating the node, we will calculate the cost of the node simultaneously. If we find the cost of any node greater than the upper bound, we will remove that node. So, in this case, we will generate only useful nodes but not all the nodes.



Ex:

	1	2	3	4	5
1	$\infty$	20	30	10	11
2	15	$\infty$	30	10	11
3	3	5	$\infty$	2	4
4	19	6	18	$\infty$	3
5	16	4	7	16	$\infty$

As we can observe in the above adjacent matrix that 10 is the minimum value in the first row, 2 is the minimum value in the second row, 2 is the minimum value in the third row, 3 is the minimum value in the third row, 3 is the minimum value in the fourth row, and 4 is the minimum value in the fifth row.

Now, we will reduce the matrix. We will subtract the minimum value with all the elements of a row. First, we evaluate the first row. Let's assume two variables, i.e., i and j, where 'i' represents the rows, and 'j' represents the columns.

When  $i = 0, j = 0$

$$M[0][0] = \infty - 10 = \infty$$

When  $i = 0, j = 1$

$$M[0][1] = 20 - 10 = 10$$

When  $i = 0, j = 2$

$$M[0][2] = 30 - 10 = 20$$

When  $i = 0, j = 3$

$$M[0][3] = 10 - 10 = 0$$

When  $i = 0, j = 4$

$$M[0][4] = 11 - 10 = 1$$

The matrix is shown below after the evaluation of the first row:

	0	1	2	3	4
0	$\infty$	10	20	0	1
1	13	$\infty$	14	2	0
2	3	5	$\infty$	2	4
3	19	6	18	$\infty$	3
4	16	4	7	16	$\infty$

Consider the second row.

When  $i = 1, j = 0$

$$M[1][0] = 15 - 2 = 13$$

When  $i = 1, j = 1$

$$M[1][1] = \infty - 2 = \infty$$

When  $i = 1, j = 2$

$$M[1][2] = 16 - 2 = 14$$

When  $i = 1, j = 3$

$$M[1][3] = 4 - 2 = 2$$

When  $i = 1, j = 4$

$$M[1][4] = 2 - 2 = 0$$

The matrix is shown below after the evaluation of the second row:

	0	1	2	3	4
0	$\infty$	10	20	0	1
1	13	$\infty$	14	2	0
2	1	3	$\infty$	0	2
3	19	6	18	$\infty$	3
4	16	4	7	16	$\infty$

Consider the third row:

When  $i = 2, j = 0$

$$M[2][0] = 3 - 2 = 1$$

When  $i = 2, j = 1$

$$M[2][1] = 5 - 2 = 3$$

When  $i = 2, j = 2$

$$M[2][2] = \infty - 2 = \infty$$

When  $i = 2, j = 3$

$$M[2][3] = 2 - 2 = 0$$

When  $i = 2, j = 4$

$$M[2][4] = 4 - 2 = 2$$

The matrix is shown below after the evaluation of the third row:

Consider the fourth row:

When  $i = 3, j = 0$

$$M[3][0] = 19 - 3 = 16$$

When  $i = 3, j = 1$

$$M[3][1] = 6 - 3 = 3$$

When  $i = 3, j = 2$

$$M[3][2] = 18 - 3 = 15$$

When  $i = 3, j = 3$

$$M[3][3] = \infty - 3 = \infty$$

When  $i = 3, j = 4$

$$M[3][4] = 3 - 3 = 0$$

The matrix is shown below after the evaluation of the fourth row:

	0	1	2	3	4
0	$\infty$	10	20	0	1
1	13	$\infty$	14	2	0
2	1	3	$\infty$	0	2
3	16	3	15	$\infty$	0
4	16	4	7	16	$\infty$

Consider the fifth row:

When  $i = 4, j = 0$

$$M[4][0] = 16 - 4 = 12$$

When  $i = 4, j = 1$

$$M[4][1] = 4 - 4 = 0$$

When  $i = 4, j = 2$

$$M[4][2] = 7 - 4 = 3$$

When  $i = 4, j = 3$

$$M[4][3] = 16 - 4 = 12$$

When  $i = 4, j = 4$

$$M[4][4] = \infty - 4 = \infty$$

The matrix is shown below after the evaluation of the fifth row:

	0	1	2	3	4
0	$\infty$	10	20	0	1
1	13	$\infty$	14	2	0
2	1	3	$\infty$	0	2
3	16	3	15	$\infty$	0
4	12	0	3	12	$\infty$

The above matrix is the reduced matrix with respect to the rows.

Now we reduce the matrix with respect to the columns. Before reducing the matrix, we first find the minimum value of all the columns. The minimum value of first column is 1, the minimum value of the second column is 0, the minimum value of the third column is 3, the minimum value of the fourth column is 0, and the minimum value of the fifth column is 0, as shown in the below matrix:

**Now we reduce the matrix.**

Consider the first column.

When  $i = 0, j = 0$

$$M[0][0] = \infty - 1 = \infty$$

When  $i = 1, j = 0$

$$M[1][0] = 13 - 1 = 12$$

When  $i = 2, j = 0$

$$M[2][0] = 1 - 1 = 0$$

When  $i = 3, j = 0$

$$M[3][0] = 16 - 1 = 15$$

When  $i = 4, j = 0$

$$M[4][0] = 12 - 1 = 11$$

The matrix is shown below after the evaluation of the first column:

	0	1	2	3	4
0	$\infty$	10	20	0	1
1	12	$\infty$	14	2	0
2	0	3	$\infty$	0	2
3	15	3	15	$\infty$	0
4	11	0	3	12	$\infty$

Since the minimum value of the first and the third columns is non-zero, we will evaluate only first and third columns. We have evaluated the first column. Now we will evaluate the third column.

Consider the third column.

When  $i = 0, j = 2$

$$M[0][2] = 20 - 3 = 17$$

When  $i = 1, j = 2$

$$M[1][2] = 13 - 1 = 12$$

When  $i = 2, j = 2$

$$M[2][2] = 1 - 1 = 0$$

When  $i = 3, j = 2$

$$M[3][2] = 16 - 1 = 15$$



When  $i = 4, j = 2$

$$M[4][2] = 12 - 1 = 11$$

The matrix is shown below after the evaluation of the third column:

	0	1	2	3	4
0	$\infty$	10	17	0	1
1	12	$\infty$	12	2	0
2	0	3	0	0	2
3	15	3	15	$\infty$	0
4	11	0	0	12	$\infty$

The above is the reduced matrix. The minimum value of rows is 21, and the columns is 4. Therefore, the total minimum value is  $(21 + 4)$  equals to 25.

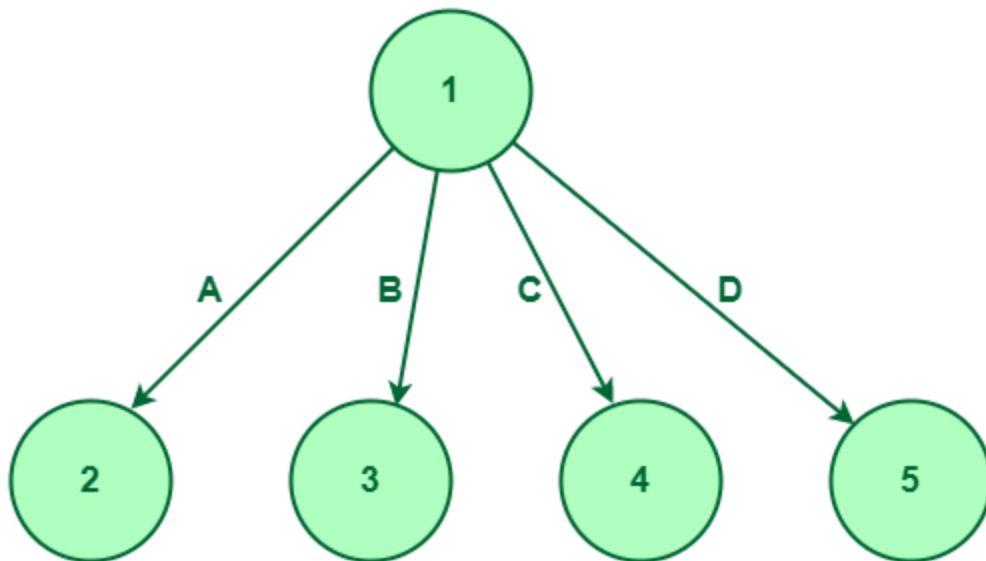
-----

### **FIFO Branch and Bound:**

#### FIFO Branch and Bound

First-In-First-Out is an approach to the branch and bound problem that uses the queue approach to create a state-space tree. In this case, the breadth-first search is performed, that is, the elements at a certain level are all searched, and then the elements at the next level are searched, starting with the first child of the first node at the previous level.

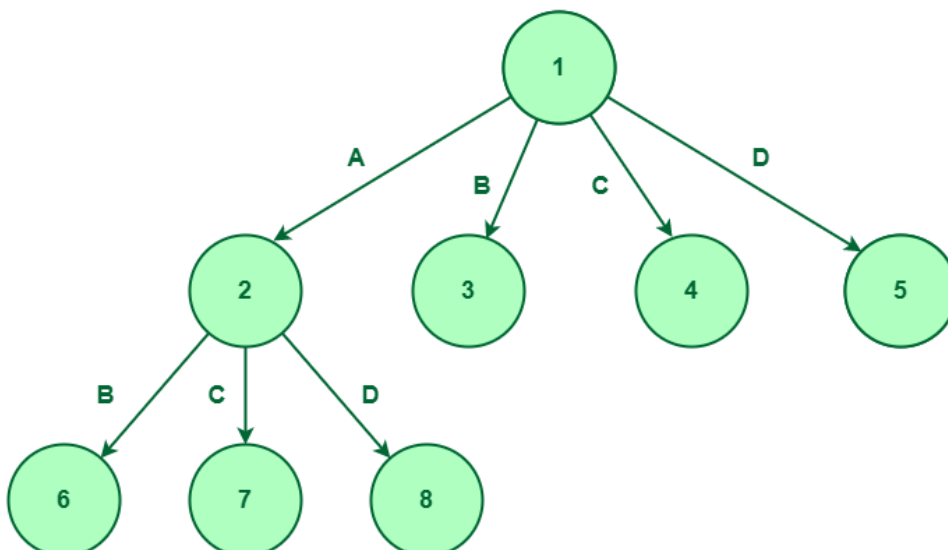
For a given set  $\{A, B, C, D\}$ , the state space tree will be constructed as follows :



State Space tree for set {A, B, C, D}

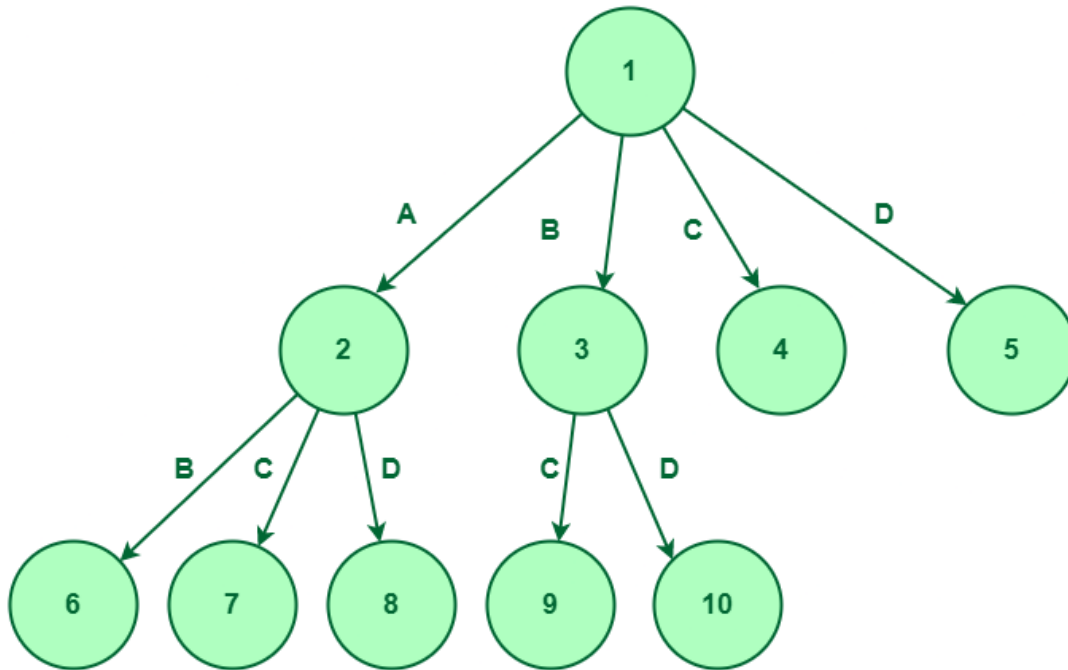
The above diagram shows that we first consider element A, then element B, then element C and finally we'll consider the last element which is D. We are performing BFS while exploring the nodes.

So, once the first level is completed. We'll consider the first element, then we can consider either B, C, or D. If we follow the route then it says that we are doing elements A and D so we will not consider elements B and C. If we select the elements A and D only, then it says that we are selecting elements A and D and we are not considering elements B and C.



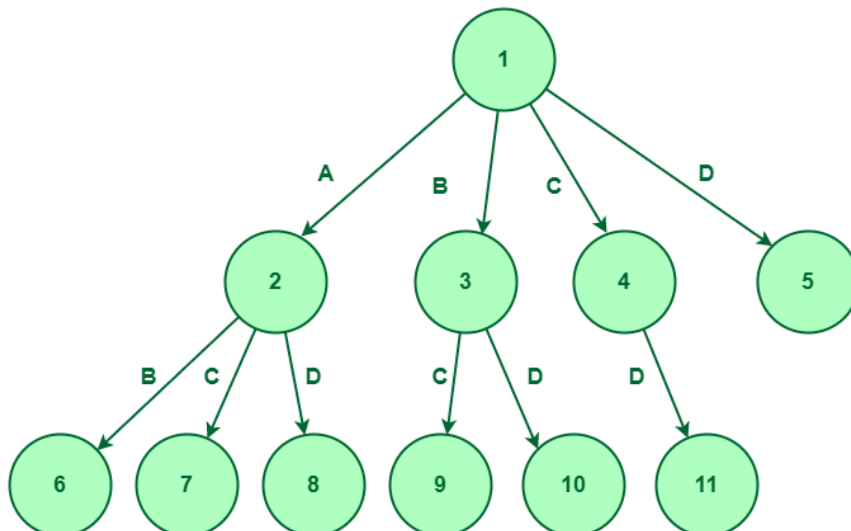
Selecting element A

Now, we will expand node 3, as we have considered element B and not considered element A, so, we have two options to explore that is elements C and D. Let's create nodes 9 and 10 for elements C and D respectively.



Considered element B and not considered element A

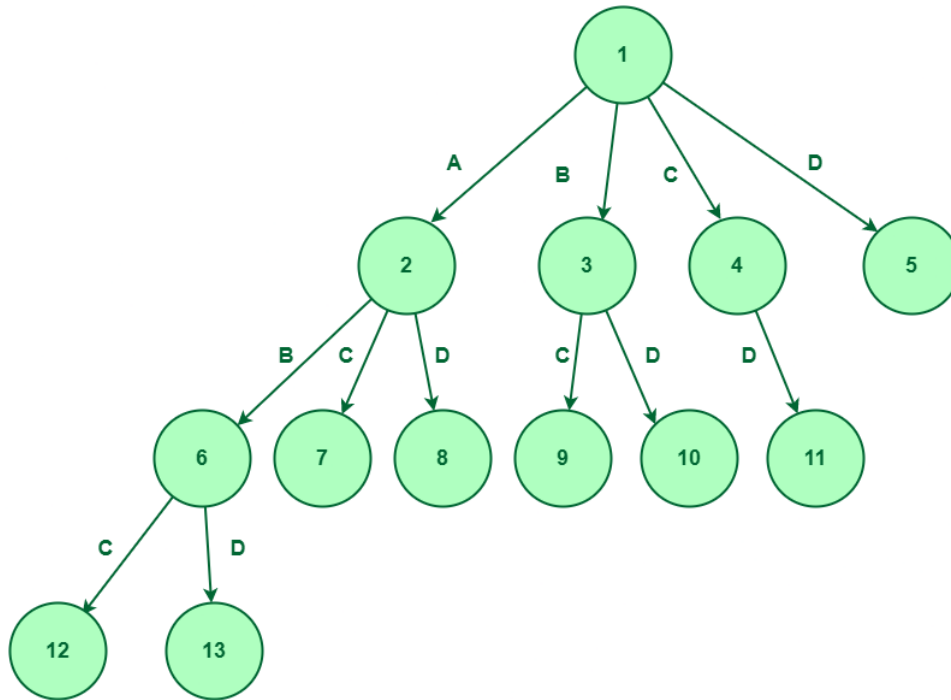
Now, we will expand node 4 as we have only considered elements C and not considered elements A and B, so, we have only one option to explore which is element D. Let's create node 11 for D.



Considered elements C and not considered elements A and B

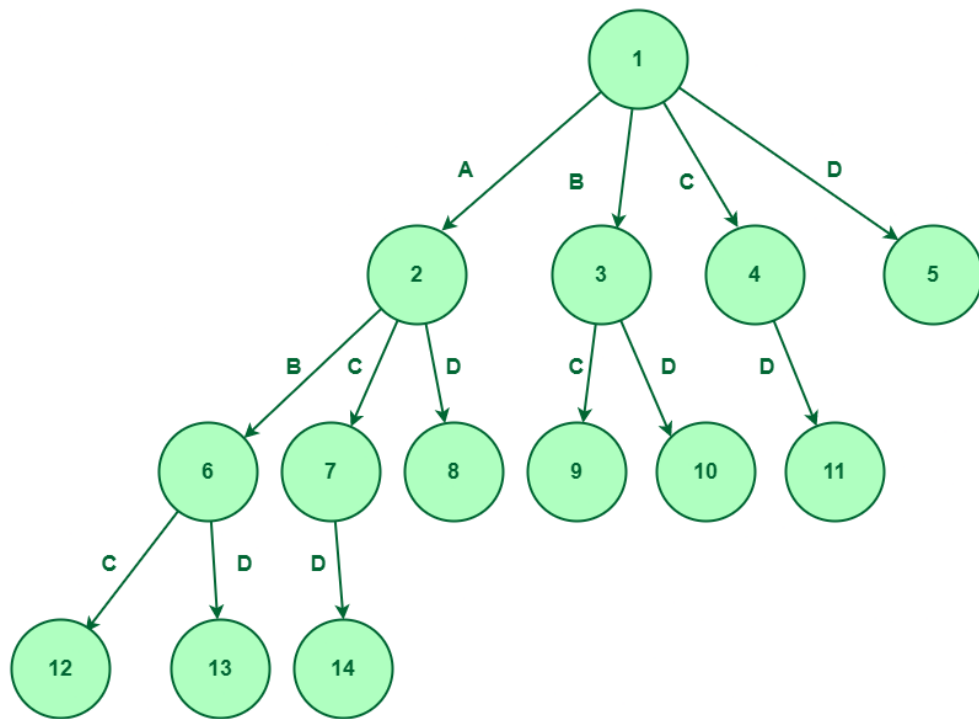
Till node 5, we have only considered elements D, and not selected elements A, B, and C. So, We have no more elements to explore, Therefore on node 5, there won't be any expansion.

Now, we will expand node 6 as we have considered elements A and B, so, we have only two option to explore that is element C and D. Let's create node 12 and 13 for C and D respectively.



Expand node 6

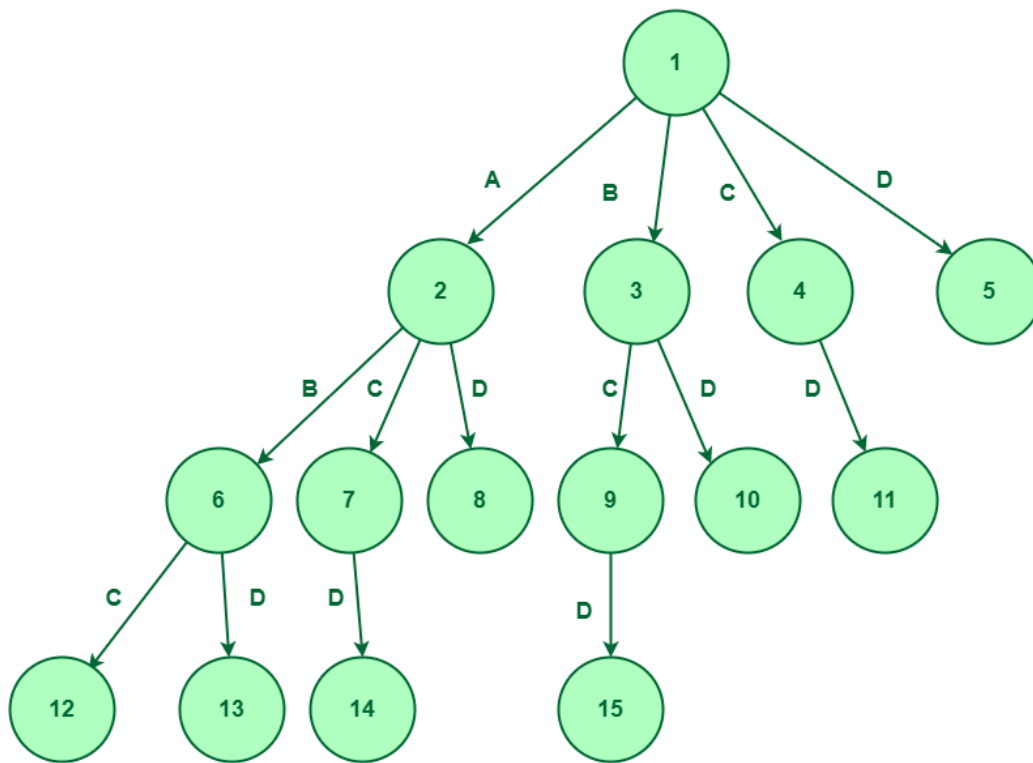
Now, we will expand node 7 as we have considered elements A and C and not consider element B, so, we have only one option to explore which is element D. Let's create node 14 for D.



Expand node 7

Till node 8, we have considered elements A and D, and not selected elements B and C, So, We have no more elements to explore, Therefore on node 8, there won't be any expansion.

Now, we will expand node 9 as we have considered elements B and C and not considered element A, so, we have only one option to explore which is element D. Let's create node 15 for D.



Expand node 9

---

## **NP-Hard and NP-Complete Problems:**

### **Basic concepts:**

A problem is called NP (nondeterministic polynomial) if its solution can be guessed and verified in polynomial time; nondeterministic means that no particular rule is followed to make the guess. If a problem is NP and all other NP problems are polynomial-time reducible to it, the problem is NP-complete.

**NP-complete** problems are a subset of the larger class of **NP (nondeterministic polynomial time) problems**. **NP** problems are a class of computational problems that can be solved in polynomial time by a non-deterministic machine and can be verified in polynomial time by a deterministic Machine. A problem **L** in **NP** is **NP-complete** if all other problems in **NP** can be reduced to **L** in polynomial time. If any **NP-complete** problem can be solved in polynomial time, then every problem in **NP** can be solved in polynomial time. **NP-complete** problems are the hardest problems in the **NP** set.

A decision problem **L** is **NP-complete** if it follow the below two properties:

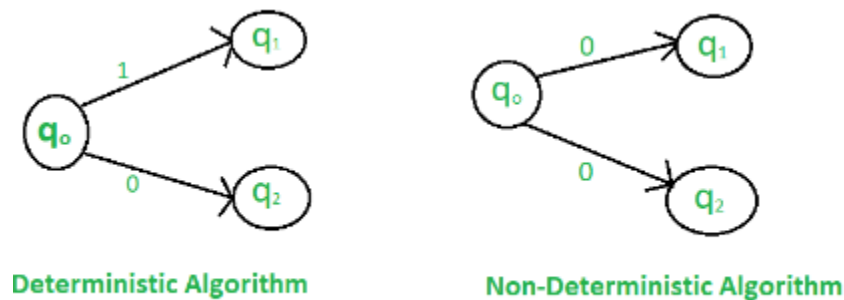
1. **L** is in **NP** (Any solution to NP-complete problems can be checked quickly, but no efficient solution is known).
2. Every problem in **NP** is reducible to **L** in polynomial time (Reduction is defined below).

A problem is **NP-Hard** if it obeys Property 2 above and need not obey Property 1. Therefore, a problem is **NP-complete** if it is both **NP** and **NP-hard**.

-----

### Deterministic and Non-deterministic Algorithms:

In a **deterministic algorithm**, for a given particular input, the computer will always produce the same output going through the same states but in the case of the **non-deterministic algorithm**, for the same input, the compiler may produce different output in different runs. In fact, non-deterministic algorithms can't solve the problem in polynomial time and can't determine what is the next step. The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it.



To implement a non-deterministic algorithm, we have a couple of languages like Prolog but these don't have standard programming language operators and these operators are not a part of any standard programming languages.

**Some of the terms related to the non-deterministic algorithm are defined below:**

- **choice(X):** chooses any value randomly from the set X.
- **failure():** denotes the unsuccessful solution.
- **success():** The solution is successful and the current thread terminates.

**Ex:**

**Problem Statement :** Search an element  $x$  on  $A[1:n]$  where  $n \geq 1$ , on successful search return  $j$  if  $a[j]$  is equals to  $x$  otherwise return 0.

**Non-deterministic Algorithm for this problem :**

```
j = choice(a, n)
if (A[j] == x) then
{
    write(j);
    success();
}
write(0); failure();
```

-----

### **NP-Hard and NP-Complete Classes:**

A problem is considered to be in NPC, that is NP-complete class, if it is a part of NP, and is as tough as any other problem in NP. And, a problem is considered to be NP-hard if all the problems in NP can be reduced in polynomial time. If a problem can be solved in polynomial time and is in NP, it means they have a polynomial time complexity.

These types of problems are referred to as NP-complete problems.

The NP-complete phenomenon is important for theoretical and practical purposes.

### **How to define NP-completeness?**

To be NP-complete it must fulfill the following criteria:

1. The language must be in NP.
2. It must be polynomial-time reducible.

If the language fails to satisfy the first criteria but fulfills the second one it is considered to be NP-hard. The NP-hard problem cannot be solved in polynomial time. But, if a problem is proved to be NPC, we do not focus on finding the most efficient algorithm for it, instead, we focus on designing an approximation algorithm.

Some examples of NP-complete problems are as follows:

1. To determine if a graph has a Hamiltonian cycle.
2. To determine if a boolean formula is satisfactory.



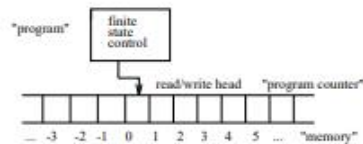
Some examples of NP-hard problems are as follows:

1. The traveling salesperson problem or TSP.
  2. Set cover problem
  3. The circuit satisfiability problem.
  4. Vertex cover problem.
- 

### **Cook's Theorem:**

Cook's Theorem proves that satisfiability is *NP*-complete by reducing all non-deterministic Turing machines to *SAT*.

Each Turing machine has access to a two-way infinite tape (read/write) and a finite state control, which serves as the program.



A program for a non-deterministic TM is:

1. Space on the tape for guessing a solution and certificate to permit verification.

2. A finite set of tape symbols
3. A finite set of states  $\Theta$  for the machine, including the start state  $q_0$  and final states  $Z_{yes}, Z_{no}$
4. A transition function, which takes the current machine state, and current tape symbol and returns the new state, symbol, and head position.

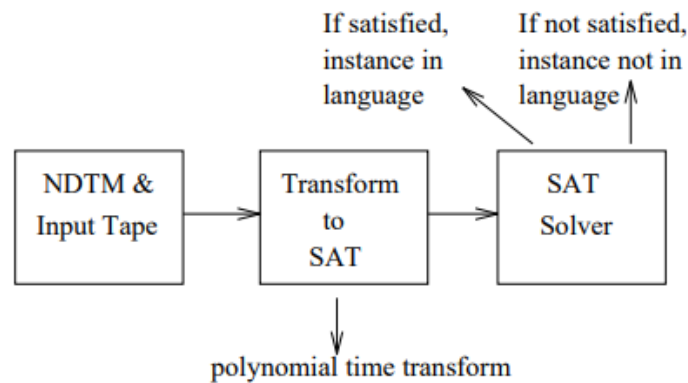
We know a problem is in  $NP$  if we have a NDTM program to solve it in worst-case time  $p[n]$ , where  $p$  is a polynomial and  $n$  is the size of the input.

## Cook's Theorem - Satisfiability is NP-complete!

---

**Proof:** We must show that any problem in  $NP$  is at least as hard as SAT. Any problem in  $NP$  has a non-deterministic TM program which solves it in polynomial time, specifically  $P(n)$ .

We will take this program and create from it an instance of satisfiability such that it is satisfiable if and only if the input string was in the language.



If a polynomial time transform exists, then SAT must be *NP*-complete, since a polynomial solution to SAT gives a polynomial time algorithm to anything in *NP*.

Our transformation will use boolean variables to maintain the state of the TM:

Variable	Range	Intended meaning
$Q[i, j]$	$0 \leq i \leq p(n)$ $0 \leq k \leq r$	At time $i$ , $M$ is in state $q_k$
$H[i, j]$	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n) + 1$	At time $i$ , the read-write head is scanning tape square $j$
$S[i, j, k]$	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n) + 1$ $0 < k < v$	At time $i$ , the contents of tape square $j$ is symbol $S_k$

Note that there are  $rp(n) + 2p^2(n) + 2p^2(n)v$  literals, a polynomial number if  $p(n)$  is polynomial.

We will now have to add clauses to ensure that these variables takes on the values as in the TM computation.

The group 6 clauses enforce the transition function of the machine. If the read-write head is not on tape square  $j$  at time  $i$ , it doesn't change ....

There are  $O(p^2(n))$  literals and  $O(p^2(n))$  clauses in all, so the transformation is done in polynomial time!

-----