1. Illustrate recurrence equation.

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.

Substitution Method:

The Substitution Method Consists of two main steps:

Guess the Solution.

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

## Iteration Methods

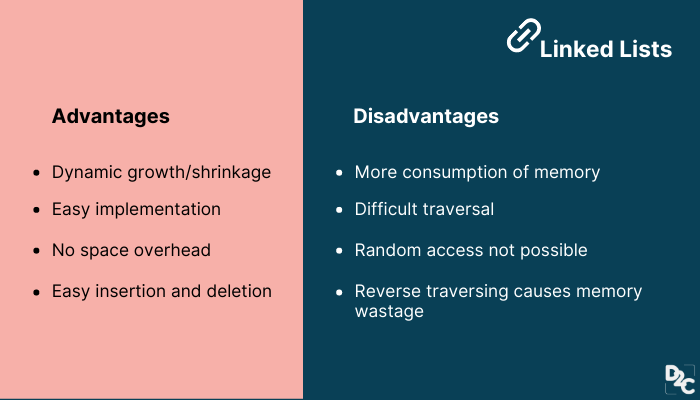
It means to expand the recurrence and express it as a summation of terms of n and initial condition.

Recursion Tree Method

Master Method

2. Write limitations of array and linked list.

* The size of the array should be known in advance.
* The array is a static data structure with a fixed size so, the size of the array cannot be modified further and hence no modification can be done during runtime.
* Insertion and deletion operations are costly in arrays as elements are stored in contiguous memory.
* If the size of the declared array is more than the required size then, it can lead to memory wastage.



3. Write algorithm for GCD of 2 numbers.

Step 1: Start

Step 2: Declare variable n1, n2, gcd=1, i=1

Step 3: Input n1 and n2

Step 4: Repeat until i<=n1 and i<=n2

Step 4.1: If n1%i==0 && n2%i==0:

Step 4.2: gcd = i

Step 5: Print gcd

Step 6: Stop

**Basic Euclidean Algorithm for GCD:**

The algorithm is based on the below facts.

* If we subtract a smaller number from a larger one (we reduce a larger number), GCD doesn’t change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
* Now instead of subtraction, if we divide the smaller number, the algorithm stops when we find the remainder 0.

4. Explain single source shortest path.

In a **shortest- paths problem**, we are given a weighted, directed graphs G = (V, E), with weight function **w: E → R** mapping edges to real-valued weights. The weight of path p = (v0,v1,..... vk) is the total of the weights of its constituent edges:

Single Source Shortest Paths

We define the shortest - path weight from u to v by δ(u,v) = min (w (p): u→v), if there is a path from u to v, and δ(u,v)= ∞, otherwise.

The **shortest path** from vertex s to vertex t is then defined as any path p with weight w (p) = δ(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 ∈ V to every vertex v ∈ 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.

5. What is spanning tree.

A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

A complete undirected graph can have **nn-2** number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be **55-2 = 125.**

* There can be more than one spanning tree of a connected graph G.
* A spanning tree does not have any cycles or loop.
* A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
* A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
* There can be a maximum **nn-2** number of spanning trees that can be created from a complete graph.
* A spanning tree has **n-1** edges, where 'n' is the number of nodes.
* If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

* Cluster Analysis
* Civil network planning
* Computer network routing protocol

6. Illustrate working of Doubly linked list with suitable example.

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

* **Link** − Each link of a linked list can store a data called an element.
* **Next** − Each link of a linked list contains a link to the next link called Next.
* **Prev** − Each link of a linked list contains a link to the previous link called Prev.
* **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

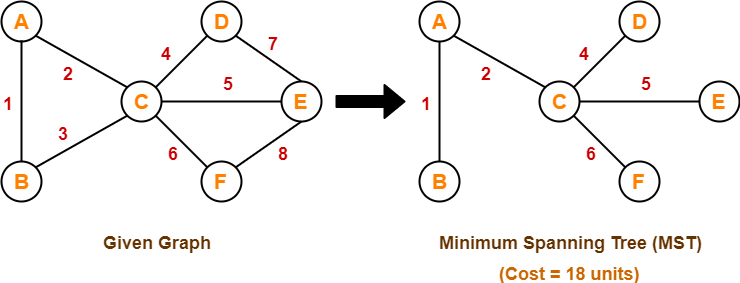
* Doubly Linked List contains a link element called first and last.
* Each link carries a data field(s) and two link fields called next and prev.
* Each link is linked with its next link using its next link.
* Each link is linked with its previous link using its previous link.
* The last link carries a link as null to mark the end of the list.
* **Insertion** − Adds an element at the beginning of the list.
* **Deletion** − Deletes an element at the beginning of the list.
* **Insert Last** − Adds an element at the end of the list.
* **Delete Last** − Deletes an element from the end of the list.
* **Insert After** − Adds an element after an item of the list.
* **Delete** − Deletes an element from the list using the key.
* **Display forward** − Displays the complete list in a forward manner.
* **Display backward** − Displays the complete list in a backward manner.

7. Find minimum spanning tree using prim and kruskal’s algorithm.



https://www.gatevidyalay.com/prims-and-kruskal-algorithm-difference/

Consider the following example-



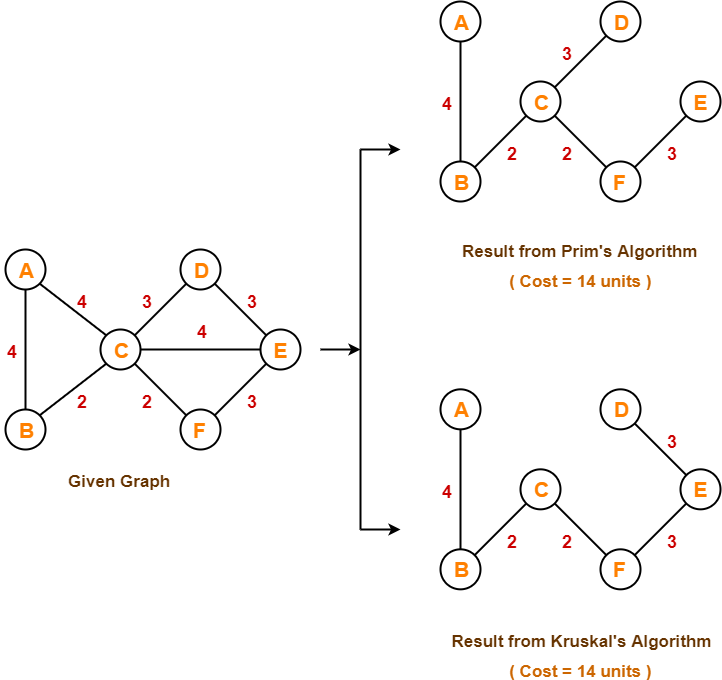
Here, both the algorithms on the above given graph produces the same MST as shown.

## ****Concept-02:****

* If all the edge weights are not distinct, then both the algorithms may not always produce the same MST.
* However, cost of both the MSTs would always be same in both the cases.

### **Example-**

Consider the following example-



Here, both the algorithms on the above given graph produces different MSTs as shown but the cost is same in both the cases.

## ****Concept-03:****

Kruskal’s Algorithm is preferred when-

* The graph is sparse.
* There are less number of edges in the graph like E = O(V)
* The edges are already sorted or can be sorted in linear time.

Prim’s Algorithm is preferred when-

* The graph is dense.
* There are large number of edges in the graph like E = O(V2).

## ****Concept-04:****

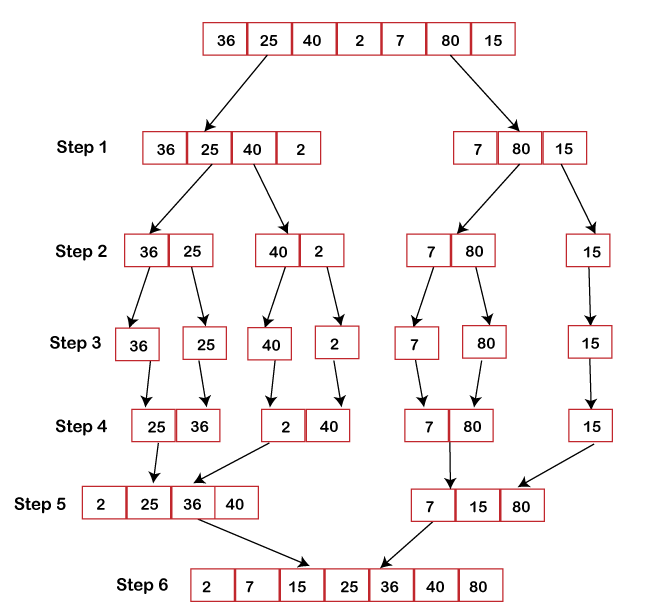
Difference between Prim’s Algorithm and Kruskal’s Algorithm-

|  |  |
| --- | --- |
| **Prim’s Algorithm** | **Kruskal’s Algorithm** |
| The tree that we are making or growing always remains connected. | The tree that we are making or growing usually remains disconnected. |
| Prim’s Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree. | Kruskal’s Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest. |
| Prim’s Algorithm is faster for dense graphs. | Kruskal’s Algorithm is faster for sparse graphs. |

8. Write any algorithm and analyze for its best, worst and average case time complexity

Merge sort is yet another sorting algorithm that falls under the category of Divide and Conquer technique. It is one of the best sorting techniques that successfully build a recursive algorithm.

In this technique, we segment a problem into two halves and solve them individually. After finding the solution of each half, we merge them back to represent the solution of the main problem.



**Best Case Complexity:** The merge sort algorithm has a best-case time complexity of **O(n\*log n)** for the already sorted array.

**Average Case Complexity:** The average-case time complexity for the merge sort algorithm is **O(n\*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

**Worst Case Complexity:** The worst-case time complexity is also **O(n\*log n)**, which occurs when we sort the descending order of an array into the ascending order.

**Space Complexity:** The space complexity of merge sort is **O(n)**.

The concept of merge sort is applicable in the following areas:

* Inversion count problem
* External sorting
* E-commerce applications

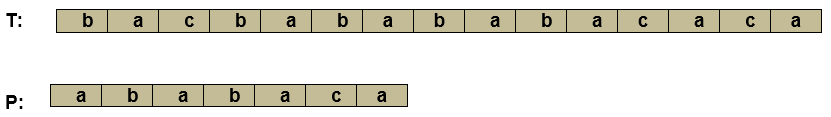
9. Explain working of KMP algorithm with suitable example.

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs.

**1. The Prefix Function (Π):** The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

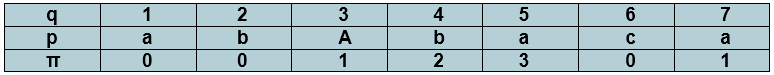
**2. The KMP Matcher:** With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

**Example:** Given a string 'T' and pattern 'P' as follows:



Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

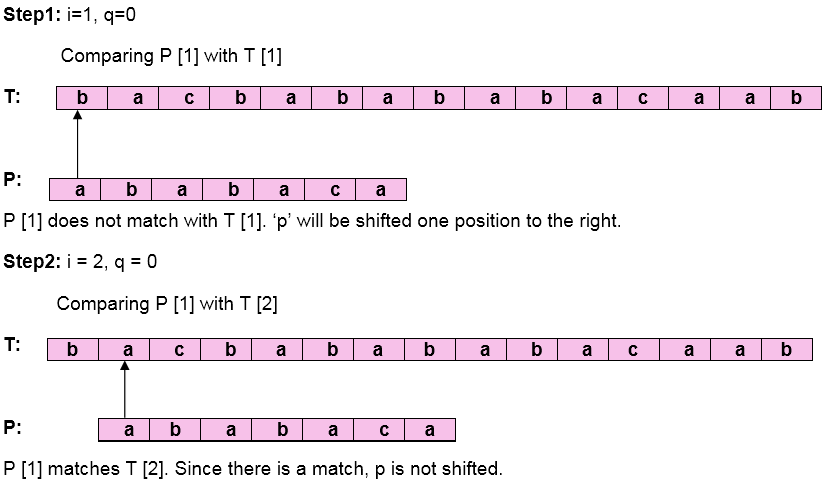
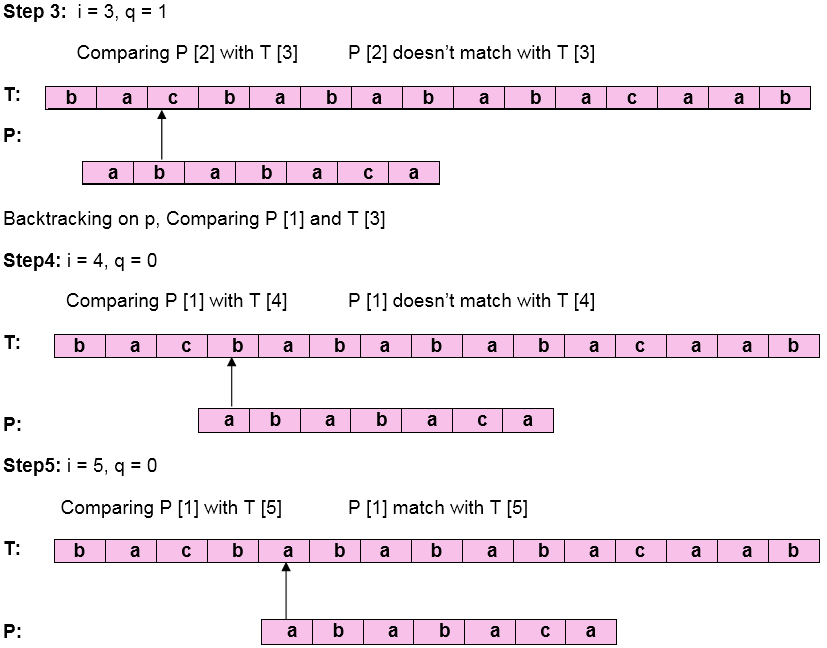
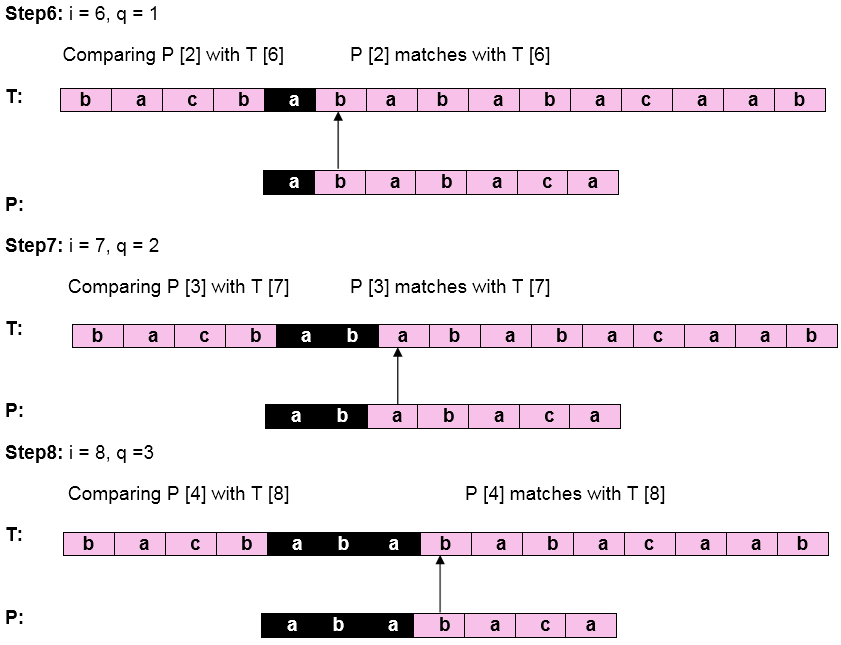
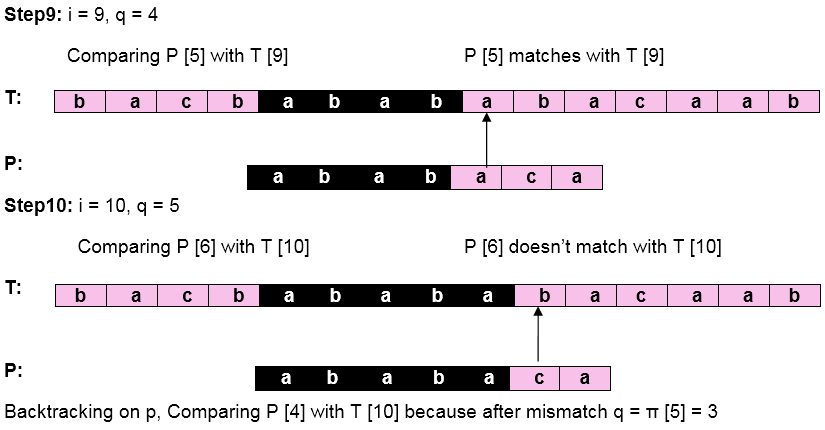
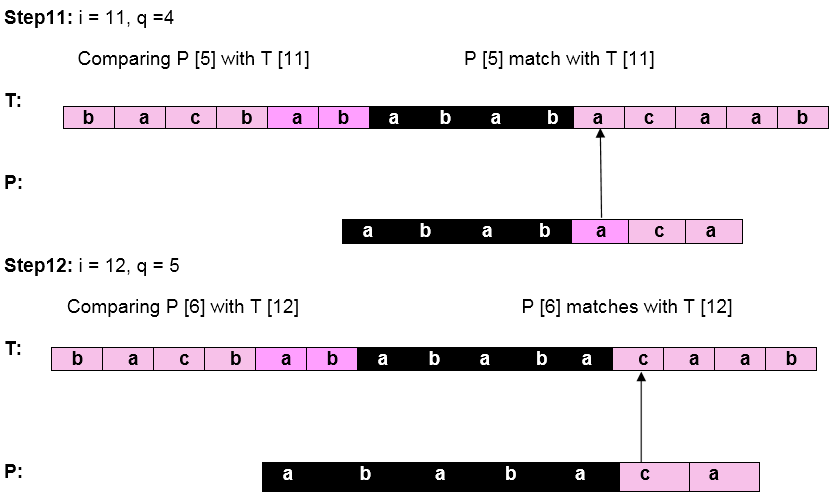
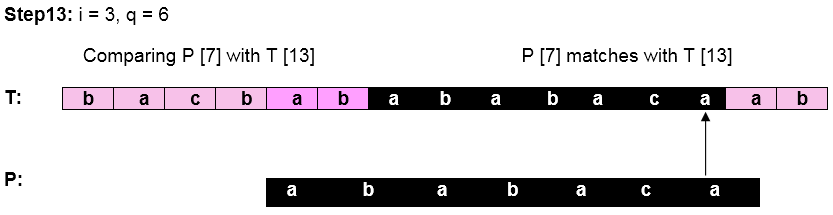
For 'p' the prefix function, ? was computed previously and is as follows:



**Solution:**

Initially: n = size of T = 15

m = size of P = 7

Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

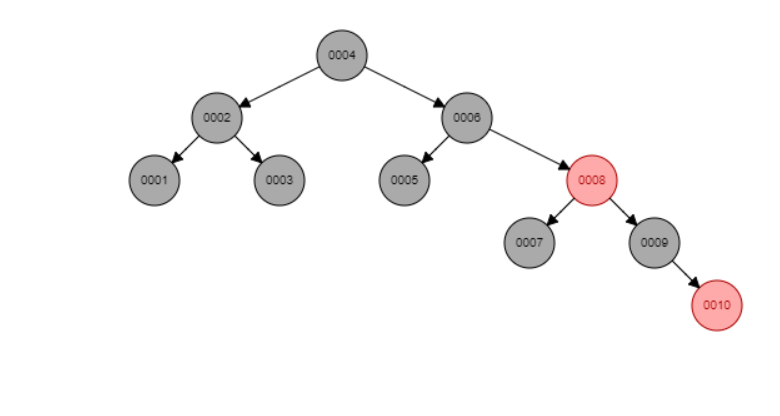
10.

Construct Red black tree:

1 2 3 4 5 6 7 8 9 10

**The red-Black tree** is a binary search tree. The prerequisite of the red-black tree is that we should know about the binary search tree. In a binary search tree, the values of the nodes in the left subtree should be less than the value of the root node, and the values of the nodes in the right subtree should be greater than the value of the root node.

Each node in the Red-black tree contains an extra bit that represents a color to ensure that the tree is balanced during any operations performed on the tree like insertion, deletion, etc. In a binary search tree, the searching, insertion and deletion take **O(log2n)** time in the average case, **O(1)** in the best case and **O(n)** in the worst case.



11.

What is the relation between P and NP class problems? Is P=NP? If No, then what will happen if P will become equal to NP?

1. **P Class**
2. **NP Class**
3. **CoNP Class**
4. **NP hard**
5. **NP complete**

**P Class**

The P in the P class stands for **Polynomial Time.** It is the collection of decision problems(problems with a “yes” or “no” answer) that can be solved by a deterministic machine in polynomial time.

**Features:**

1. The solution to P problems is easy to find.
2. P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

This class contains many natural problems like:

1. **Calculating the greatest common divisor.**
2. **Finding a maximum matching.**
3. **Decision versions of linear programming.**

**NP Class**

The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

**Features:**

1. The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
2. Problems of NP can be verified by a Turing machine in polynomial time.

This class contains many problems that one would like to be able to solve effectively: **Boolean Satisfiability Problem (SAT).**

**Hamiltonian Path Problem.**

**Graph coloring.**

**Co-NP Class**

Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is No, then there is proof that can be checked in polynomial time.

**Features:**

1. If a problem X is in NP, then its complement X’ is also is in CoNP.
2. For an NP and CoNP problem, there is no need to verify all the answers at once in polynomial time, there is a need to verify only one particular answer “yes” or “no” in polynomial time for a problem to be in NP or CoNP.

Some example problems for C0-NP are:

1. **To check prime number.**
2. **Integer Factorization.**

**NP-hard class**

An NP-hard problem is at least as hard as the hardest problem in NP and it is the class of the problems such that every problem in NP reduces to NP-hard.

**Features:**

1. All NP-hard problems are not in NP.
2. It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.
3. A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

Some of the examples of problems in Np-hard are:

1. **Halting problem.**
2. **Qualified Boolean formulas.**
3. **No Hamiltonian cycle.**

**NP-complete class**

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

**Features:**

1. NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
2. If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

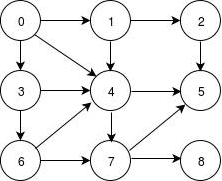
Some example problems include:

1. **Decision version of 0/1 Knapsack.**
2. **Hamiltonian Cycle.**
3. **Satisfiability.**
4. **Vertex cover.**

|  |  |
| --- | --- |
| **Complexity Class** | **Characteristic feature** |
| **P** | Easily solvable in polynomial time. |
| **NP** | Yes, answers can be checked in polynomial time. |
| **Co-NP** | No, answers can be checked in polynomial time. |
| **NP-hard** | All NP-hard problems are not in NP and it takes a long time to check them. |
| **NP-complete** | A problem that is NP and NP-hard is NP-complete. |

12.

Traverse the following graph using Depth First Traversal 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.

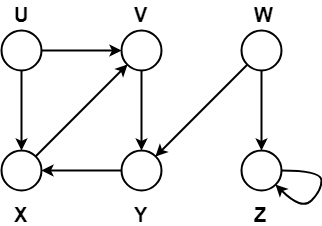
### **Applications of DFS algorithm**

The applications of using the DFS algorithm are given as follows -

* DFS algorithm can be used to implement the topological sorting.
* It can be used to find the paths between two vertices.
* It can also be used to detect cycles in the graph.
* DFS algorithm is also used for one solution puzzles.
* DFS is used to determine if a graph is bipartite or not.

**Similar Problem with Steps**

Compute the DFS tree for the graph given below-



Also, show the discovery and finishing time for each vertex and classify the edges.

## ****Solution-****

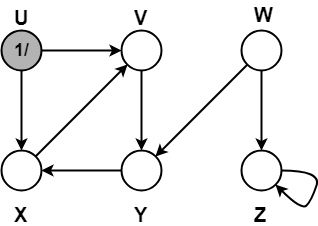
Initially for all the vertices of the graph, we set the variables as-

* color[v] = WHITE
* π[v] = NIL
* time = 0 (Global)

Let us start processing the graph from vertex U.

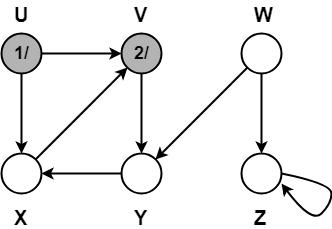
### **Step-01:**

* color[U] = GREY
* time = 0 + 1 = 1
* d[U] = 1



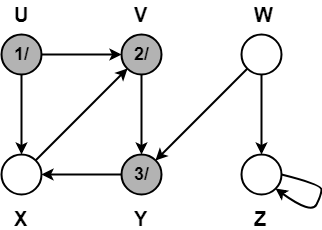
### **Step-02:**

* π[V] = U
* color[V] = GREY
* time = 1 + 1 = 2
* d[V] = 2



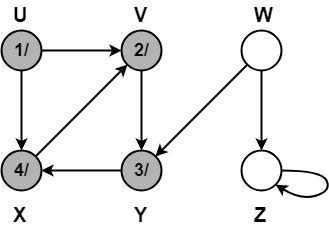
### **Step-03:**

* π[Y] = V
* color[Y] = GREY
* time = 2 + 1 = 3
* d[Y] = 3



### **Step-04:**

* π[X] = Y
* color[X] = GREY
* time = 3 + 1 = 4
* d[X] = 4

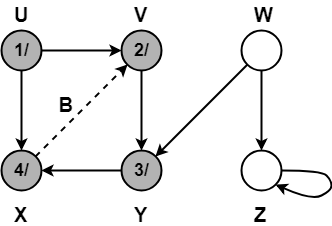


### **Step-05:**

When DFS tries to extend the visit from vertex X to vertex V, it finds-

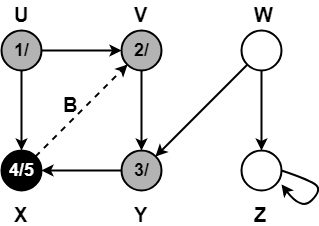
* Vertex V is an ancestor of vertex X since it has already been discovered
* Vertex V is GREY in color.

Thus, edge XV is a back edge.



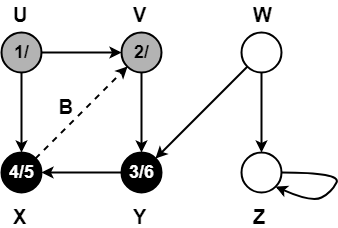
### **Step-06:**

* color[X] = BLACK
* time = 4 + 1 = 5
* f[X] = 5



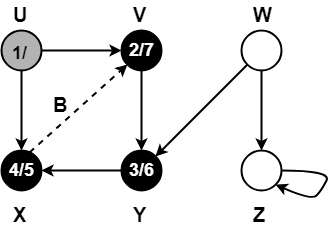
### **Step-07:**

* color[Y] = BLACK
* time = 5 + 1 = 6
* f[Y] = 6



### **Step-08:**

* color[V] = BLACK
* time = 6 + 1 = 7
* f[V] = 7



### **Step-09:**

When DFS tries to extend the visit from vertex U to vertex X, it finds-

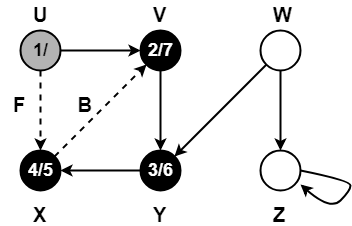
* Vertex X has already been completely processed i.e. vertex X has finished and is black.
* But vertex U has still not finished.

Alternatively,

When DFS tries to extend the visit from vertex U to vertex X, it finds-

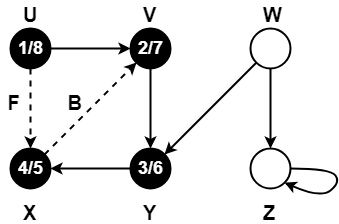
* Color(X) = BLACK
* d(X) > d(U)

Thus, edge UX is a forward edge.



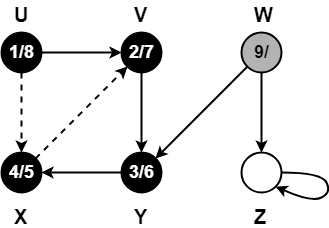
### **Step-10:**

* color[U] = BLACK
* time = 7 + 1 = 8
* f[U] = 8



### **Step-11:**

* color[W] = GREY
* time = 8 + 1 = 9
* d[W] = 9



### **Step-12:**

When DFS tries to extend the visit from vertex W to vertex Y, it finds-

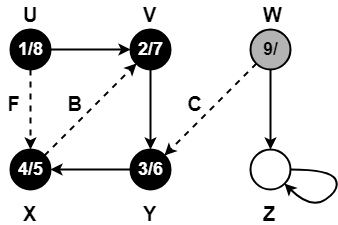
* Vertex Y has already been completely processed i.e. vertex Y has finished.
* Vertex Y is neither a descendant nor an ancestor of vertex W.

Alternatively,

When DFS tries to extend the visit from vertex W to vertex Y, it finds-

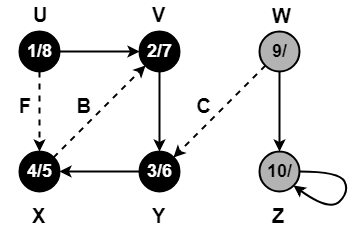
* Color(Y) = BLACK
* d(Y) < d(W)

Thus, edge WY is a cross edge.



### **Step-13:**

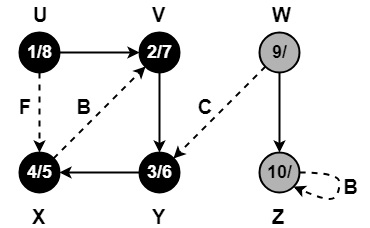
* π[Z] = W
* color[W] = GREY
* time = 9 + 1 = 10
* d[W] = 10



### **Step-14:**

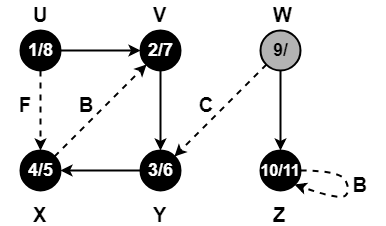
Since, self-loops are considered as back edges.

Therefore, self-loop present on vertex Z is considered as a back edge.



### **Step-15:**

* color[Z] = BLACK
* time = 10 + 1 = 11
* f[Z] = 11



### **Step-16:**

* color[W] = BLACK
* time = 11 + 1 = 12
* f[W] = 12

