UNIT-III

What is Greedy Algorithm?

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment.

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.

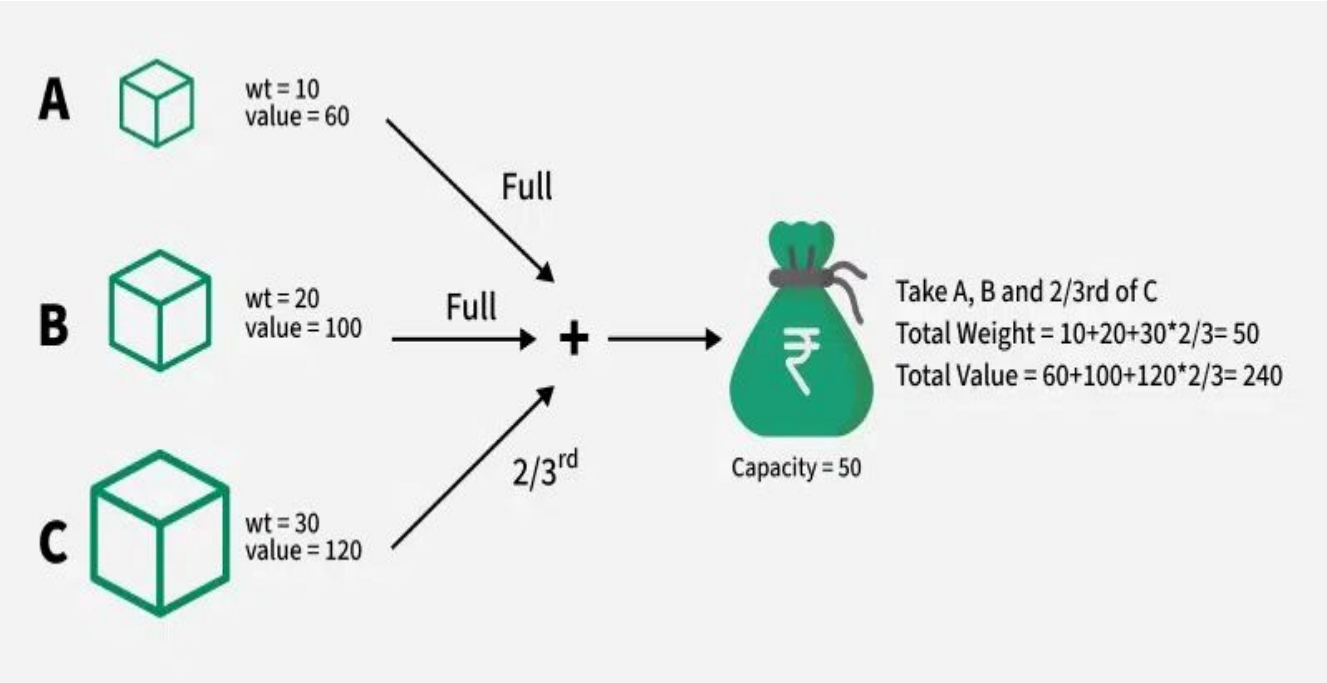
This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset.

Characteristics of a greedy method:

- To construct the solution in an optimal way, this algorithm creates two sets where one set contains all the chosen items, and another set contains the rejected items.
- A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

Greedy Algorithms General Structure

A **greedy algorithm** solves problems by making the best choice at each step. Instead of looking at all possible solutions, it focuses on the option that seems best right now.



Problem structure:

Most of the problems where greedy algorithms work follow these two properties:

1). **Greedy Choice Property:-** This property states that choosing the best possible option at each step will lead to the best overall solution. If this is not true, a greedy approach may not work.

2). **Optimal Substructure:-** This means that you can break the problem down into smaller parts, and solving these smaller parts by making greedy choices helps solve the overall problem

Different Types of Greedy Algorithm

- Selection Sort
- Knapsack Problem
- Minimum Spanning Tree
- Single-Source Shortest Path Problem
- Job Scheduling Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Huffman Coding
- Ford-Fulkerson Algorithm

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 a job sequencing with a deadline.
- This algorithm is also used to solve the fractional knapsack problem.

Greedy method is one of the strategies used for solving the optimization problems.

The **Knapsack problem** is an example of the combinational optimization problem. This problem is also commonly known as the "**Rucksack Problem**". The name of the problem is defined from the maximization problem as mentioned below:

Given a bag with maximum weight capacity of W and a set of items, each having a weight and a value associated with it. Decide the number of each item to take in a collection such that the total weight is less than the capacity and the total value is maximized.

How does the Greedy Algorithm works?

Greedy Algorithm solve optimization problems by making the **best local choice** at each step in the hope of finding the global optimum. It's like taking the best option available at each moment, hoping it will lead to the best overall outcome.

Here's how it works:

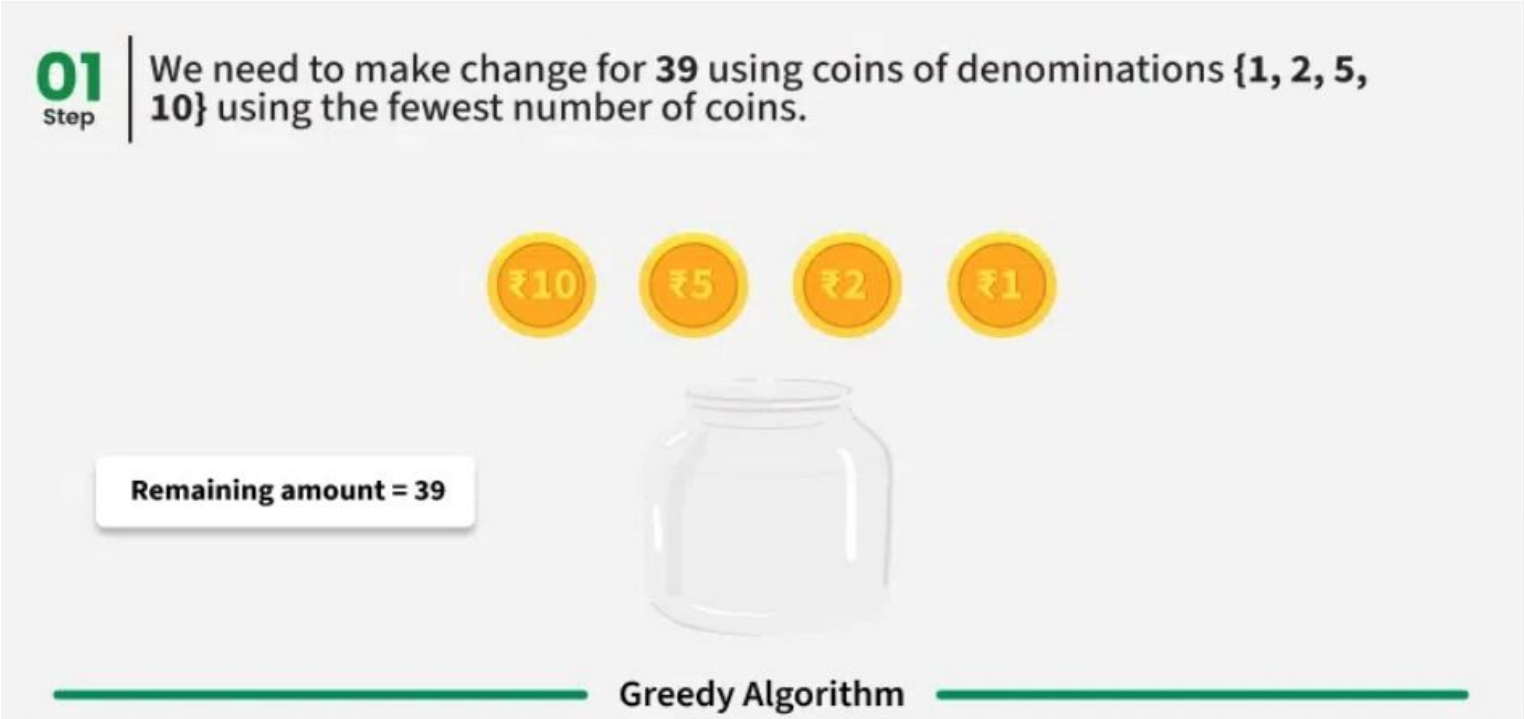
- Start with the **initial state** of the problem. This is the **starting point** from where you begin making choices.
- Evaluate **all possible choices** you can make from the current state. Consider all the options available at that specific moment.
- Choose the option that seems best at that moment, regardless of future consequences. This is the **"greedy"** part – **you take the best option available now, even if it might not be the best in the long run.**
- Move to the new state based on your chosen option. This becomes your **new starting point** for the next iteration.
- Repeat **steps 2-4** until you reach the **goal state** or no further progress is possible. Keep making the best local choices until you reach the end of the problem or get stuck.

Example:

Let's say you have a **set of coins** with values **[1, 2, 5, 10]** and you need to give **minimum number of coin** to someone change for **39**.

*The **greedy algorithm** for making change would work as follows:*

- ***Step-1**: Start with the **largest coin** value that is **less than or equal** to the amount to be changed. In this case, the largest coin less than or equal to **39** is 10.*
- ***Step- 2**: Subtract the largest coin value from the amount to be changed, and **add** the coin to the solution. In this case, **subtracting 10 from 39 gives 29**, and we add **one 10-coin** to the solution.*
- *Repeat **steps 1 and 2** until the amount to be changed becomes 0.*



02 Step

Start with the highest denomination (10). Since greedy algorithm works by taking the largest coin possible first, we start with the coin of denomination 10.

Remaining amount = 9

We pick 10 as many times as we can without exceeding the target (39). So, 39 ÷ 10 = 3 coins of 10.

03 Step | Now take the next highest denomination (5).

Remaining amount = 4

We pick as many 5's as we can for the remaining amount (9), so 9 ÷ 5 = 1 coin of 5.

04 Step | Now take the next highest denomination (2).

Remaining amount = 0

Now, we pick the coin of denomination 2 for the remaining amount (4). 4 ÷ 2 = 2 coins of 2.

05 Step | We have successfully made the target sum (39) using a total of **6 coins**: 3*10 + 1*5 + 2*2 = 39.

Remaining amount = 0

```
// C++ Program to find the minimum number of coins to construct a given amount using greedy approach
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int minCoins(vector<int> &coins, int amount) {
    int n = coins.size();
    sort(coins.begin(), coins.end());
    int res = 0;
      // Start from the coin with highest denomination
    for(int i = n - 1; i >= 0; i--) {
        if(amount >= coins[i]) {
    // Find the maximum number of ith coin we can use
int cnt = (amount / coins[i]);
                // Add the count to result
            res += cnt;
```

```
// Subtract the corresponding amount from the total amount
amount -= (cnt * coins[i]);
        }
// Break if there is no amount left
        if(amount == 0)
            break;
    }
    return res;
}
int main() {
    vector<int> coins = {5, 2, 10, 1};
    int amount = 39;
    cout << minCoins(coins, amount);
    return 0;
}
```

Types of Knapsack Problem:

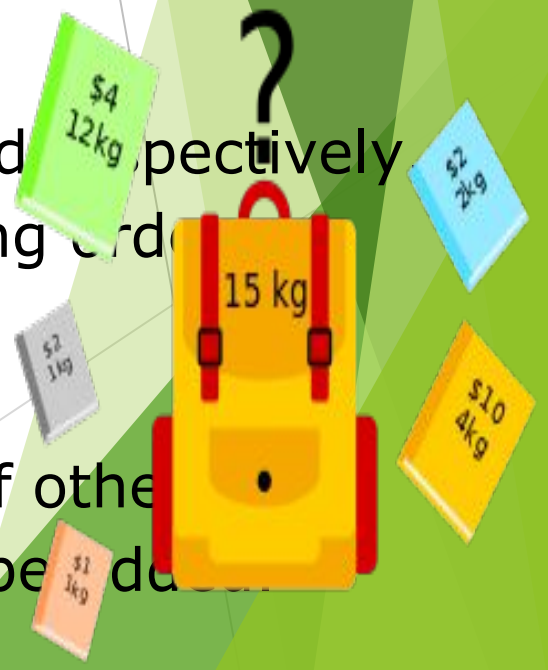
- The knapsack problem can be classified into the following types:
- Fractional Knapsack Problem
- 0/1 Knapsack Problem

Fractional Knapsack Problem :

The weights (Wi) and profit values (Pi) of the items to be added in the knapsack are taken as an input for the fractional knapsack algorithm and the subset of the items added in the knapsack without exceeding the limit and with maximum profit is achieved as the output.

Algorithm

- Consider all the items with their weights and profits mentioned respectively
- Calculate P_i/W_i of all the items and sort the items in descending order based on their P_i/W_i values.
- Without exceeding the limit, add the items into the knapsack.
- If the knapsack can still store some weight, but the weights of other items exceed the limit, the fractional part of the next time can be added.
- Hence, giving it the name fractional knapsack problem.



Examples :

- For the given set of items and the knapsack capacity of 10 kg, find the subset of the items to be added in the knapsack such that the profit is maximum.

Items	1	2	3	4	5
Weights (in kg)	3	3	2	5	1
Profits	10	15	10	12	8

Solution :

Step 1 :

Given, n = 5

$Wi = \{3, 3, 2, 5, 1\}$

$Pi = \{10, 15, 10, 12, 8\}$

Calculate P_i/W_i for all the items

Items	1	2	3	4	5
Weights (in kg)	3	3	2	5	1
Profits	10	15	10	20	8
P_i/W_i	3.3	5	5	4	8

- **Step 2 :**Arrange all the items in descending order based on P_i/W_i.

Items	5	2	3	4	1
Weights (in kg)	1	3	2	5	3
Profits	8	15	10	20	10
P_i/W_i	8	5	5	4	3.3

- **Step 3 :**

Without exceeding the knapsack capacity, insert the items in the knapsack with maximum profit.

Knapsack = {5, 2, 3}

However, the knapsack can still hold 4 kg weight, but the next item having 5 kg weight will exceed the capacity.

Therefore, only 4 kg weight of the 5 kg will be added in the knapsack.

Items	5	2	3	4	1
Weights (in kg)	1	3	2	5	3
Profits	8	15	10	20	10
Knapsack	1	1	1	4/5	0

Hence, the knapsack holds the weights = [(1 * 1) + (1 * 3) + (1 * 2) + (4/5 * 5)] = 10, with maximum profit of

[(1 * 8) + (1 * 15) + (1 * 10) + (4/5 * 20)] = 49.

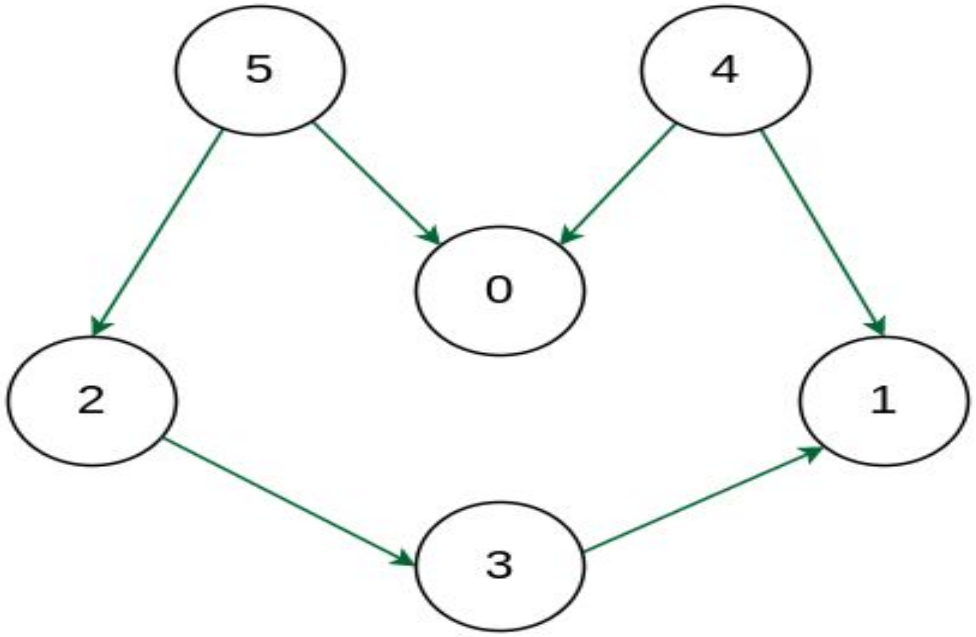
Applications :

Few of the many real-world applications of the knapsack problem are −

- Cutting raw materials without losing too much material
- Picking through the investments and portfolios
- Selecting assets of asset-backed securitization
- Generating keys for the Merkle-Hellman algorithm
- Cognitive Radio Networks
- Power Allocation
- Network selection for mobile nodes

Topological sort :

- Topological sorting for **Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge u-v, vertex **u** comes before **v** in the ordering.

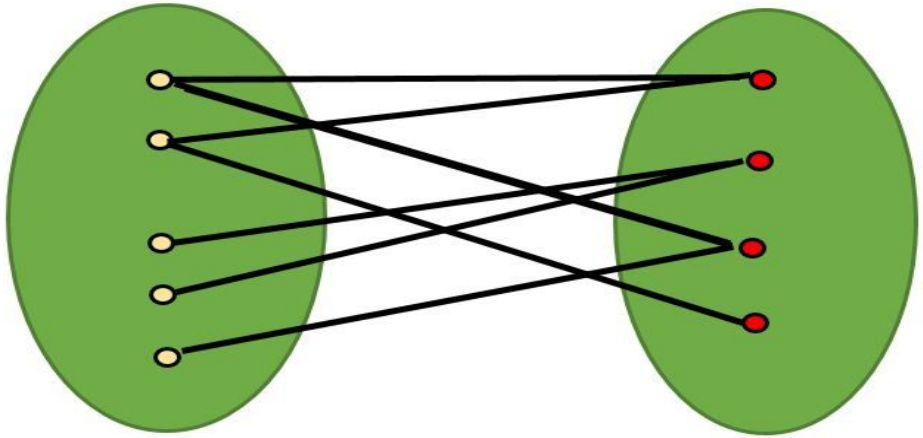


Output: *5 4 2 3 1 0*

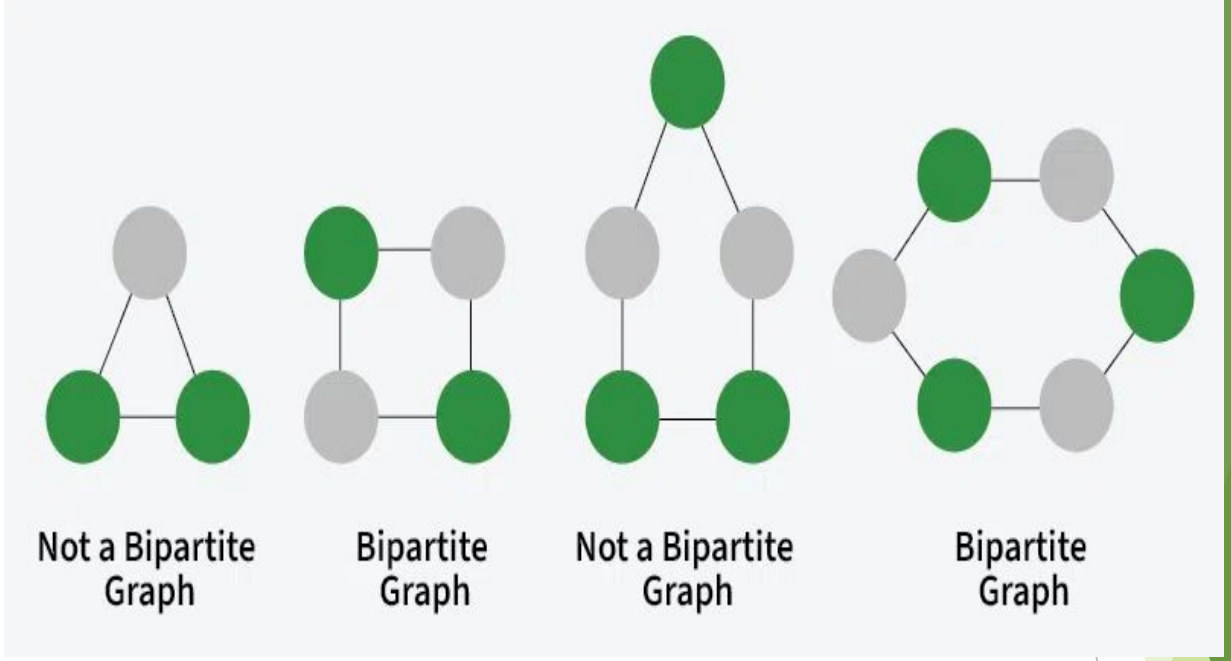
Explanation: *The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges). A topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. Another topological sorting of the following graph is "4 5 2 3 1 0".*

What is Bipartite Graph?

- A **bipartite graph** can be colored with two colors such that no two adjacent vertices share the same color. This means we can divide the graph's vertices into two distinct sets where:
- All edges connect vertices from one set to vertices in the other set.
- No edges exist between vertices within the same set.
- **An alternate definition:** Formally, a graph G = (V, E) is bipartite if and only if its vertex set V can be partitioned into two non-empty subsets X and Y, such that every edge in E has one endpoint in X and the other endpoint in Y. This partition of vertices is also known as bi-partition.



Example of Bipartite Graph



Application of Bipartite Graph

Bipartite graphs help solve matching problems. For example, they can assign tasks to employees or courses to students.

They can also create recommendation systems. One group represents users and the other represents items. If a user rates an item, a connection is made between them. This helps suggest items to users based on what they like.

Additionally, bipartite graphs can show relationships in social networks. One group represents people and the other represents groups. If someone belongs to a group, there's a connection between them.

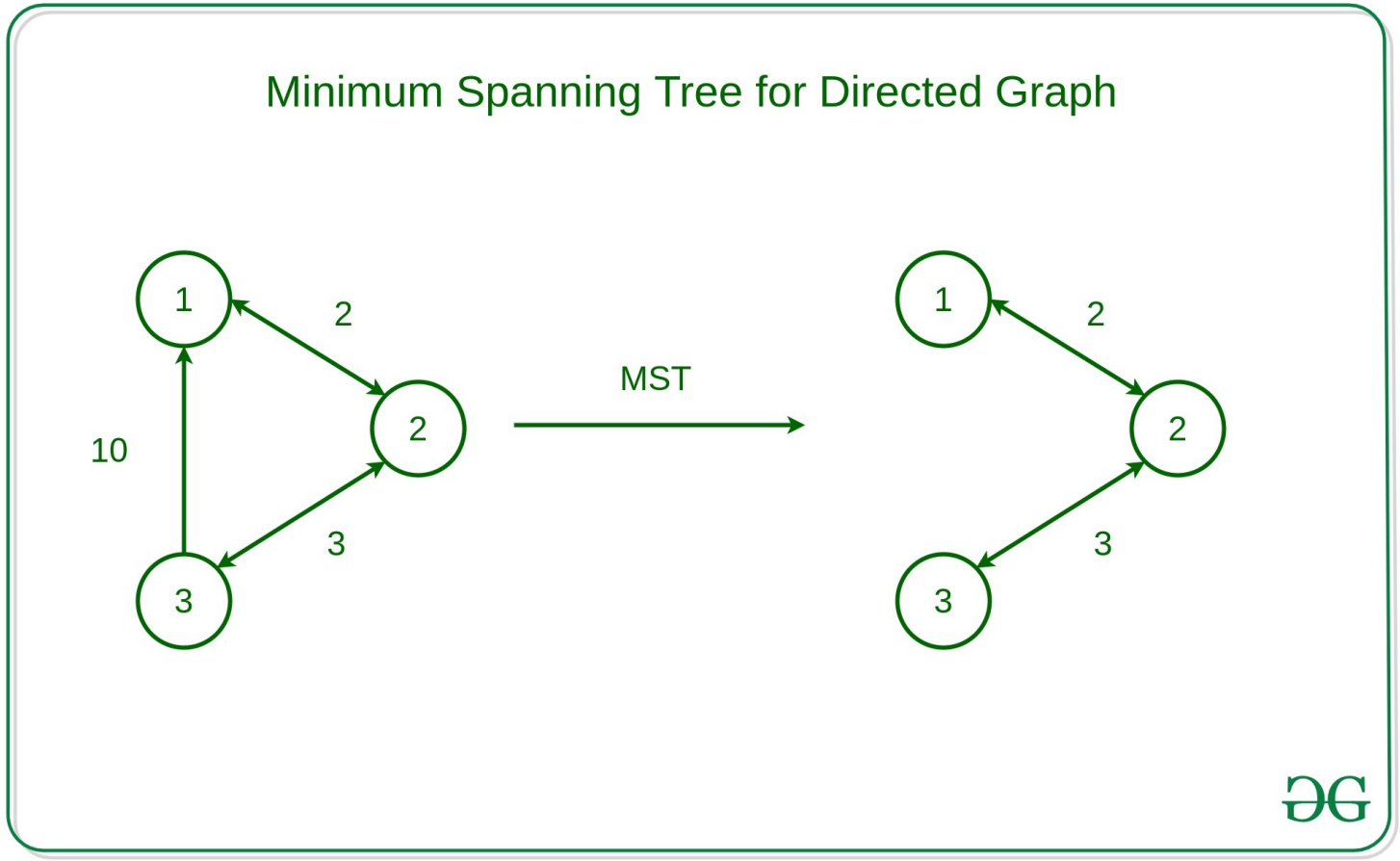
Bipartite graphs are instrumental in solving stable marriage problems and other matching scenarios.

- **What is a 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 $\mathbf{n^{n-2}}$ 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 $\mathbf{5^{5-2} = 125.}$
- The weight of a spanning tree is determined by the sum of weight of all the edge involved in it.

- **Applications of the spanning tree :**
- 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

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

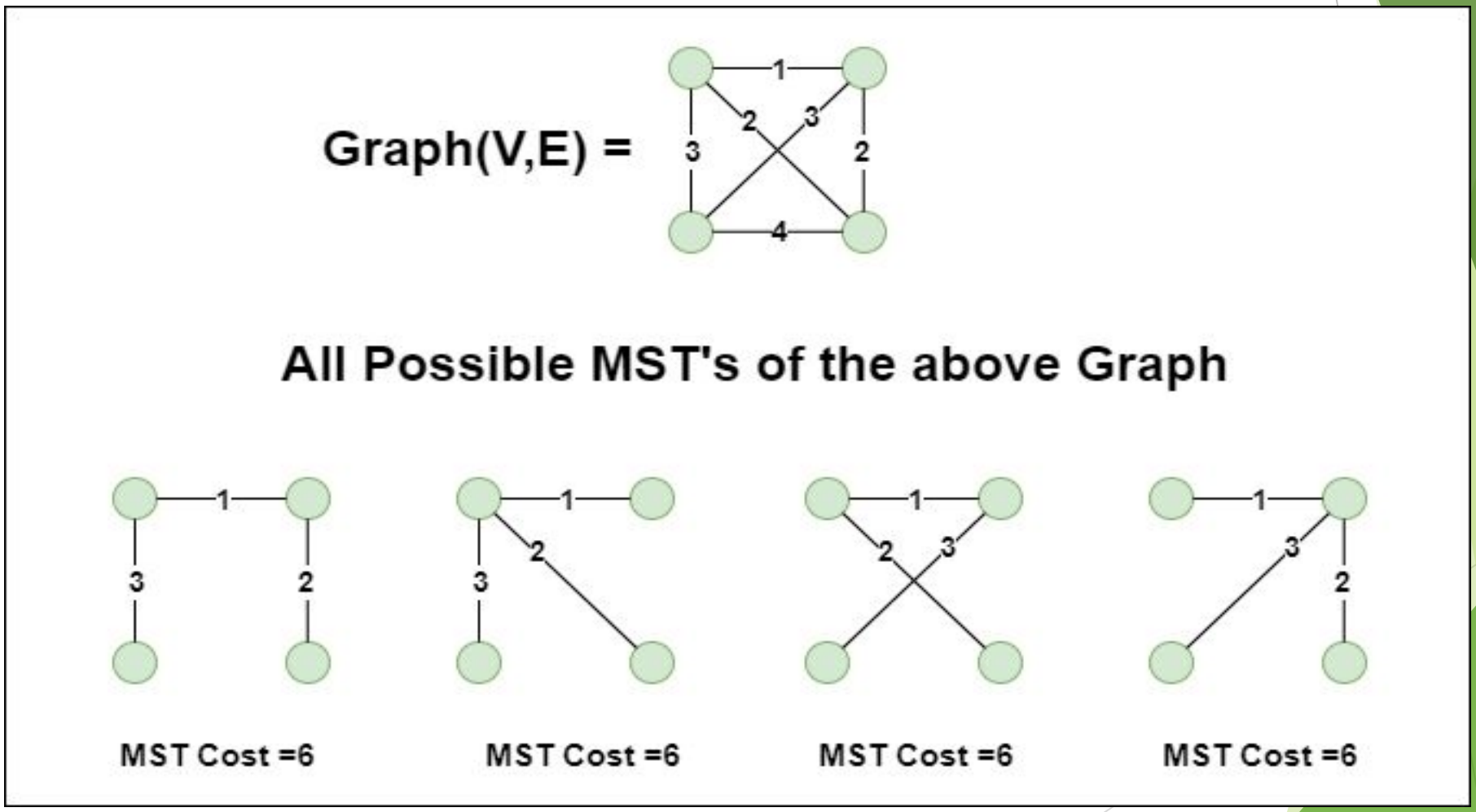


Properties of Minimum Spanning Tree:

- A minimum spanning tree connects all the vertices in the graph, ensuring that there is a path between any pair of nodes.
- An **MST** is **acyclic**, meaning it contains no cycles. This property ensures that it remains a tree and not a graph with loops.
- An **MST** with **V** vertices (where **V** is the number of vertices in the original graph) will have exactly **V – 1** edges, where **V** is the number of vertices.
- An **MST** is optimal for minimizing the total edge weight, but it may not necessarily be unique.
- The cut property states that if you take any cut (a partition of the vertices into two sets) in the original graph and consider the minimum-weight edge that crosses the cut, that edge is part of the **MST**.

Minimum Spanning Tree of a Graph may not be Unique:

Like a spanning tree, there can also be many possible MSTs for a graph as shown in the below image:



Algorithms to find Minimum Spanning Tree:

There are several algorithms to find the minimum spanning tree from a given graph, some of them are listed below:

i) **Kruskal's Minimum Spanning Tree Algorithm:**

ii) **Prim's Minimum Spanning Tree Algorithm:**

Kruskal's Minimum Spanning Tree Algorithm:

This is one of the popular algorithms for finding the minimum spanning tree from a connected, undirected graph. This is a greedy algorithm. The algorithm workflow is as below:

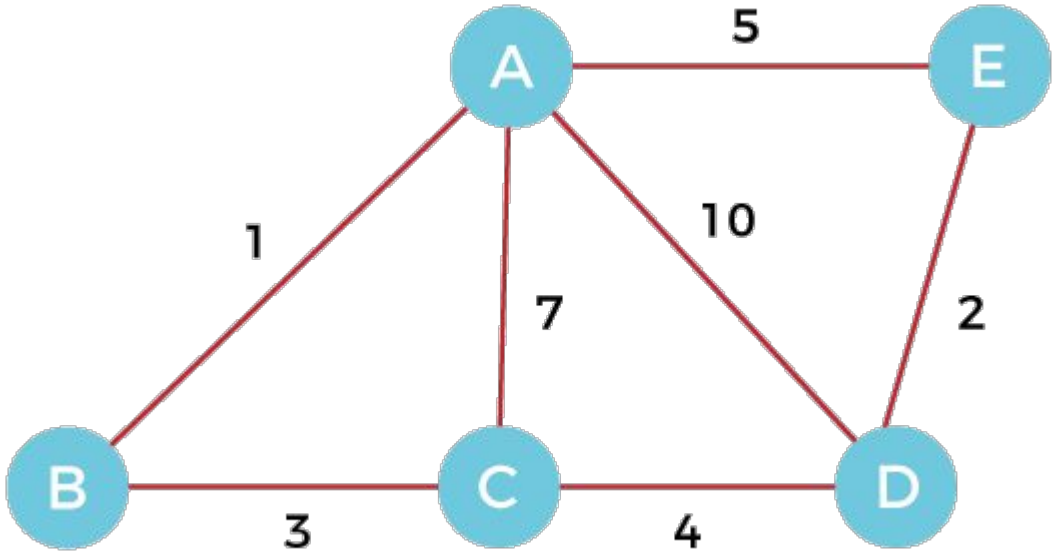
- First, it sorts all the edges of the graph by their weights,
- Then starts the iterations of finding the spanning tree.
- At each iteration, the algorithm adds the next lowest-weight edge one by one, such that the edges picked until now does not form a cycle.

This algorithm can be implemented efficiently using a DSU (Disjoint-Set) data structure to keep track of the connected components of the graph. This is used in a variety of practical applications such as network design, clustering, and data analysis.

Example of Kruskal's algorithm :

Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.

Suppose a weighted graph is -



The weight of the edges of the above graph is given in the below table -

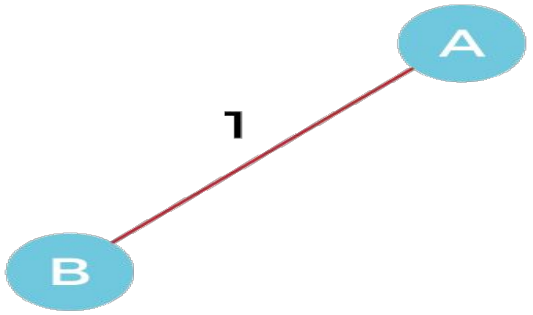
Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

Now, sort the edges given above in the ascending order of their weights.

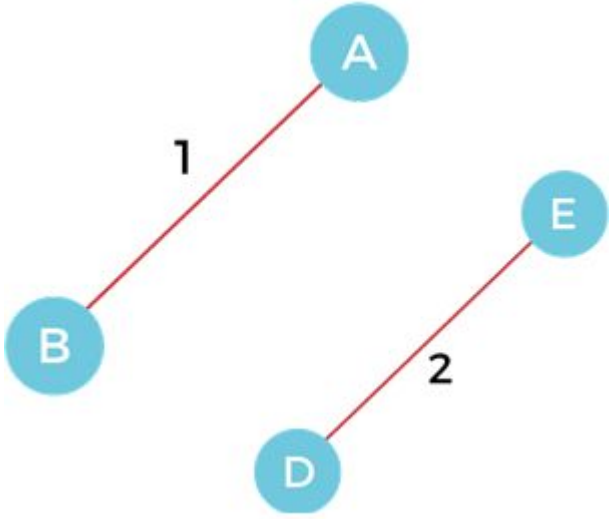
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Now, let's start constructing the minimum spanning tree.

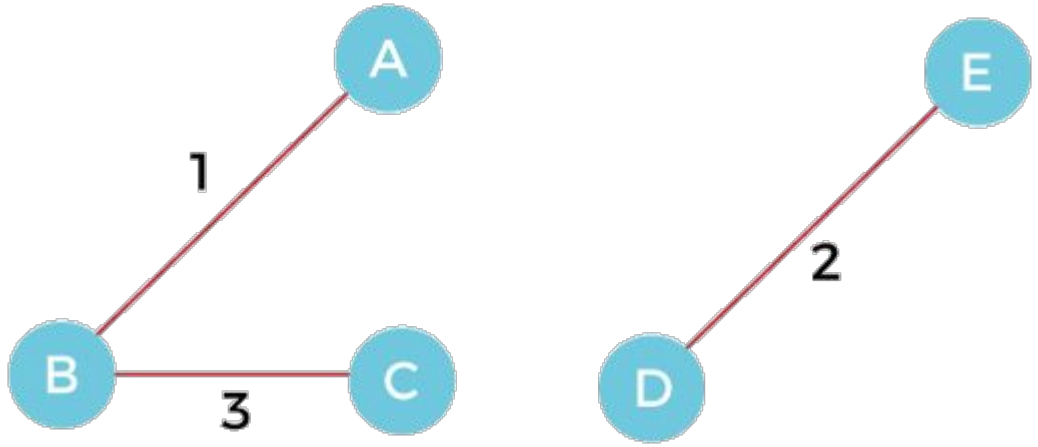
Step 1 - First, add the edge **AB** with weight **1** to the MST.



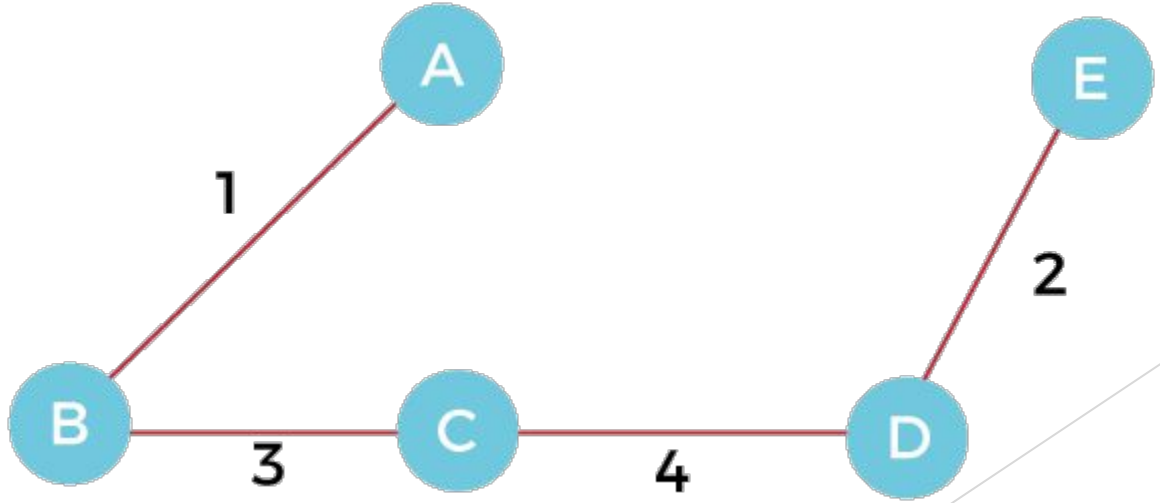
Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



Step 3 - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.



Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.

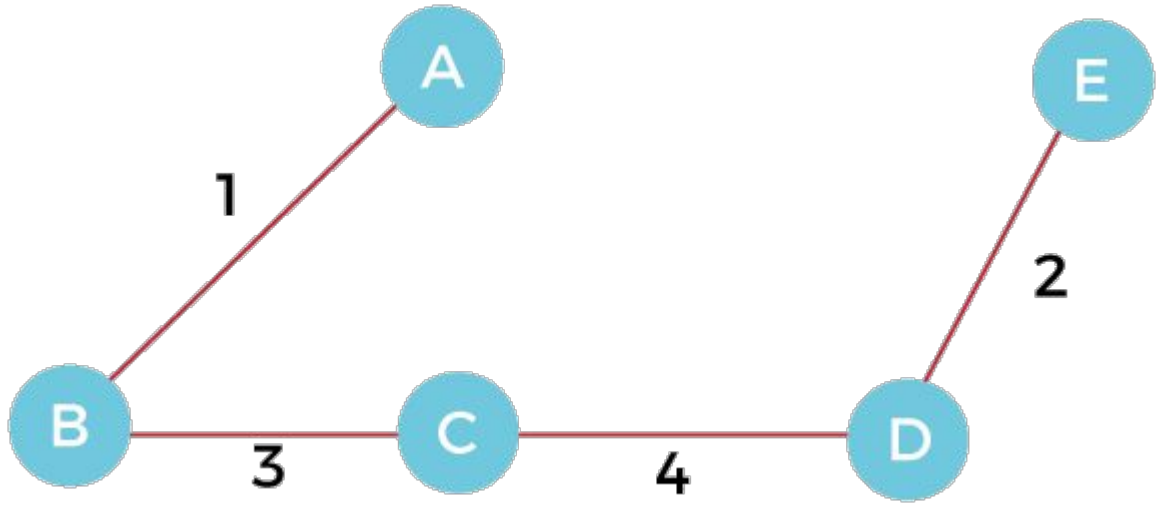


Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

Step 6 - Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

Step 7 - Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10

Complexity of Kruskal's algorithm

Now, let's see the time complexity of Kruskal's algorithm.

- **Time Complexity**

  The time complexity of Kruskal's algorithm is O(E logE) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

Prim's Minimum Spanning Tree Algorithm:

This is also a greedy algorithm. This algorithm has the following workflow:

- It starts by selecting an arbitrary vertex and then adding it to the MST.
- Then, it repeatedly checks for the minimum edge weight that connects one vertex of MST to another vertex that is not yet in the MST.
- This process is continued until all the vertices are included in the MST.

To efficiently select the minimum weight edge for each iteration, this algorithm uses priority_queue to store the vertices sorted by their minimum edge weight currently. It also simultaneously keeps track of the MST using an array or other data structure suitable considering the data type it is storing.

This algorithm can be used in various scenarios such as image segmentation based on color, texture, or other features. For Routing, as in finding the shortest path between two points for a delivery

How does Prim's Algorithm Work?

- ***Step 1:*** *Determine an arbitrary vertex as the starting vertex of the MST.*

  Step 2: *Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).*

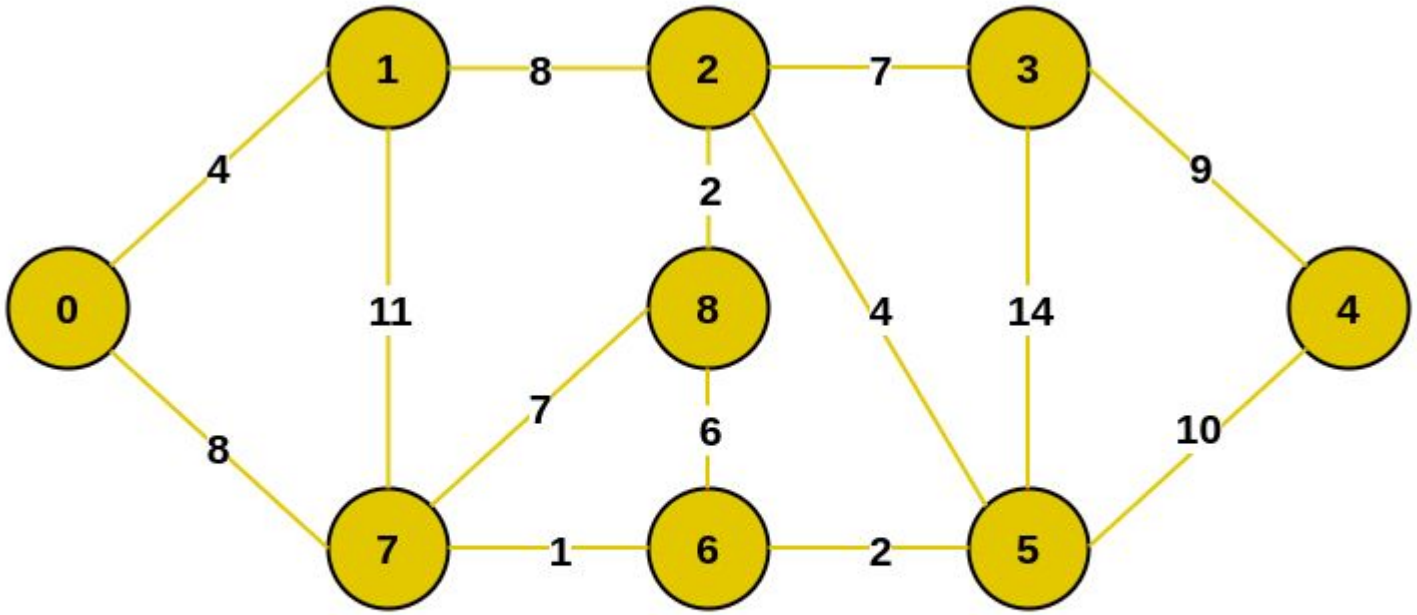
  Step 3: *Find edges connecting any tree vertex with the fringe vertices.*

  Step 4: *Find the minimum among these edges.*

  Step 5: *Add the chosen edge to the MST if it does not form any cycle.*

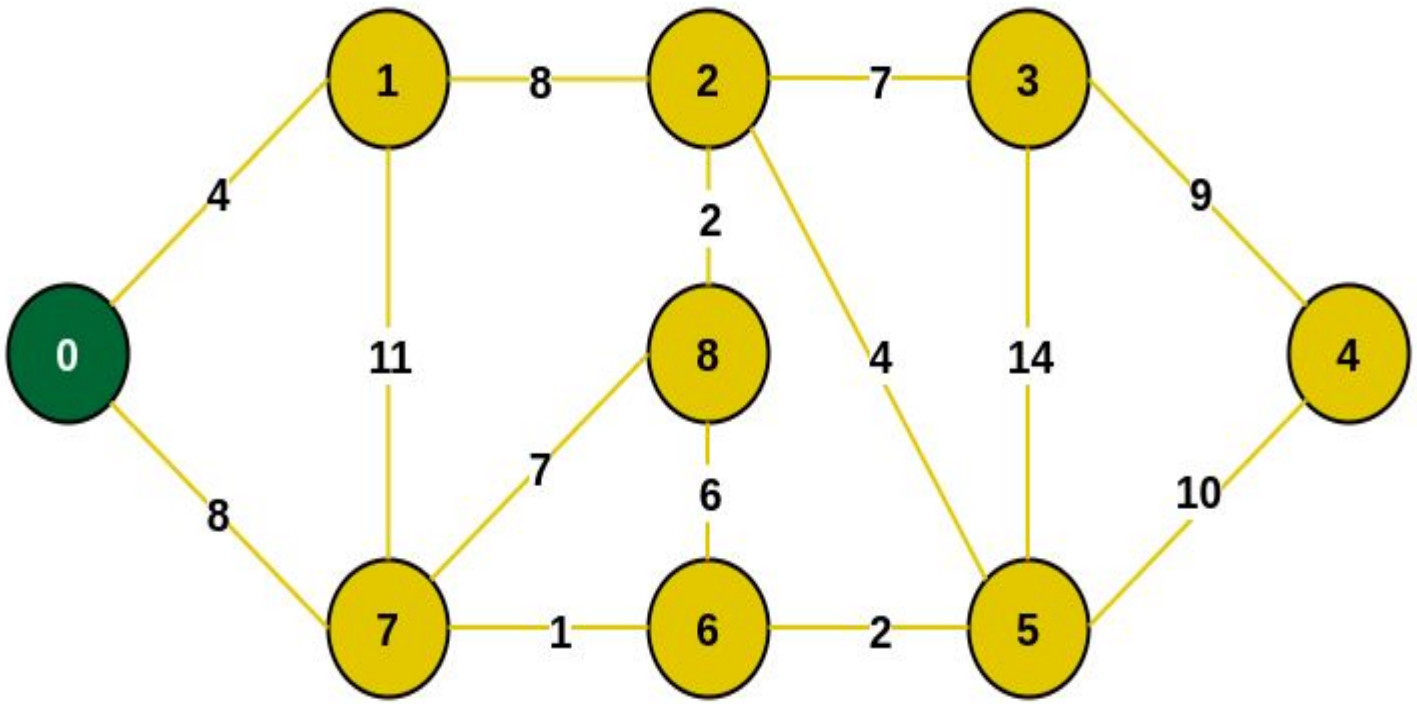
  Step 6: *Return the MST and exit*

Consider the following graph as an example for which we need to find the Minimum Spanning Tree (MST).



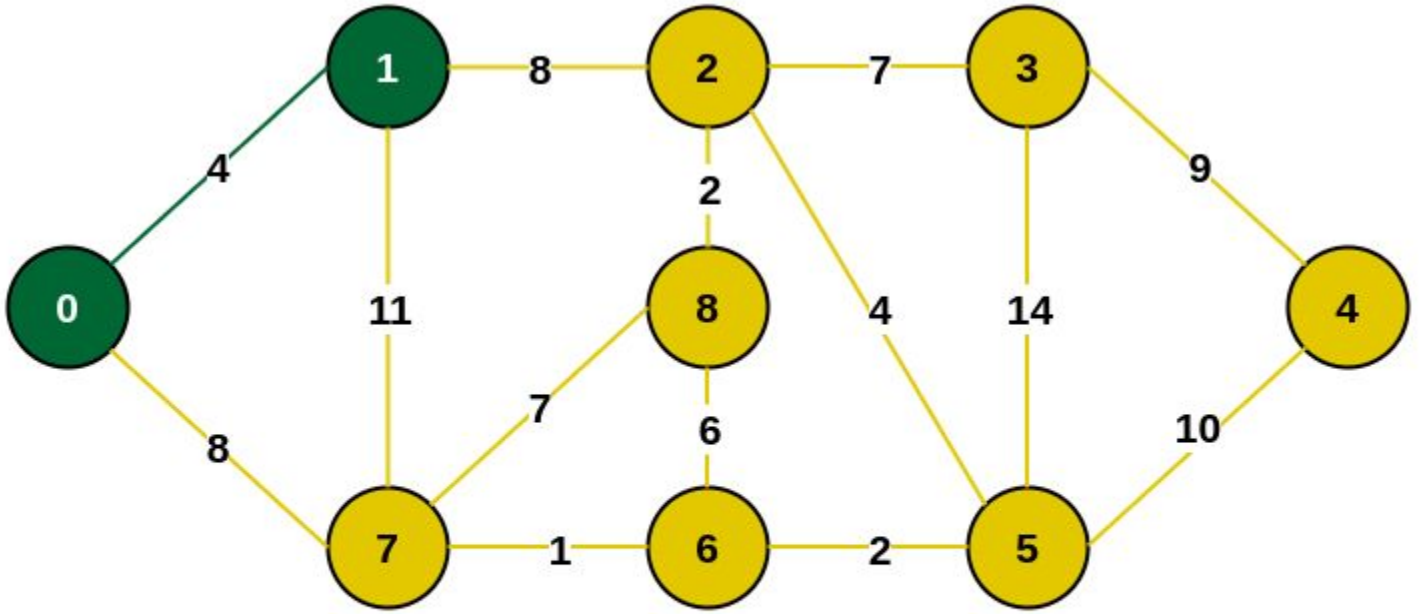
Example of a Graph

Step 1: *Firstly, we select an arbitrary vertex that acts as the starting vertex of the Minimum Spanning Tree. Here we have selected vertex* ***0*** *as the starting vertex.*



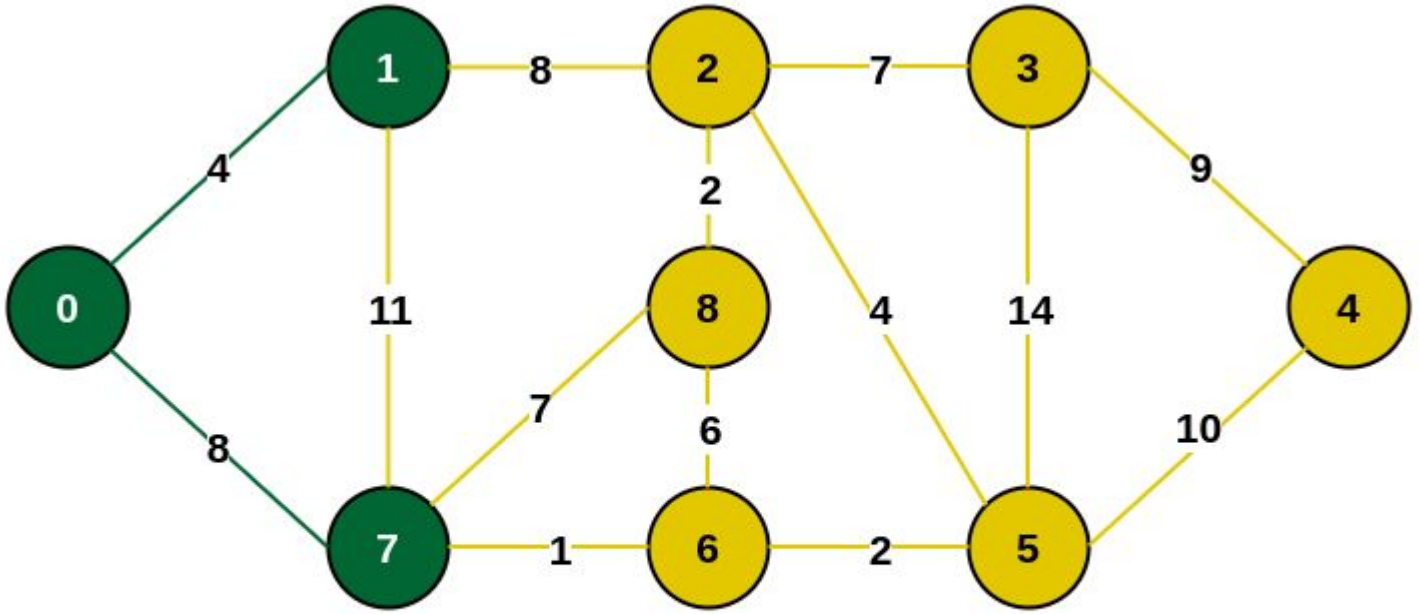
Select an arbitrary starting vertex. Here we have selected 0

Step 2: *All the edges connecting the incomplete MST and other vertices are the edges {0, 1} and {0, 7}. Between these two the edge with minimum weight is {0, 1}. So include the edge and vertex 1 in the MST.*



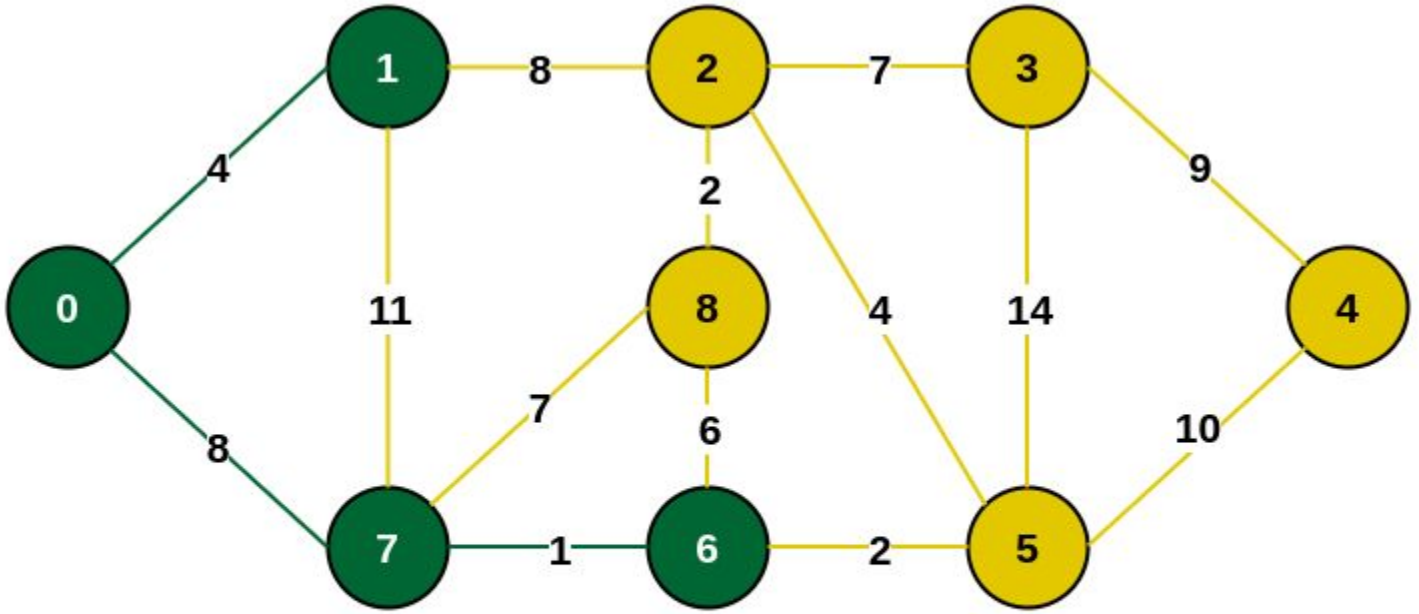
Minimum weighted edge from MST to other vertices is 0-1 with weight 4

Step 3: *The edges connecting the incomplete MST to other vertices are {0, 7}, {1, 7} and {1, 2}. Among these edges the minimum weight is 8 which is of the edges {0, 7} and {1, 2}. Let us here include the edge {0, 7} and the vertex 7 in the MST. [We could have also included edge {1, 2} and vertex 2 in the MST].*



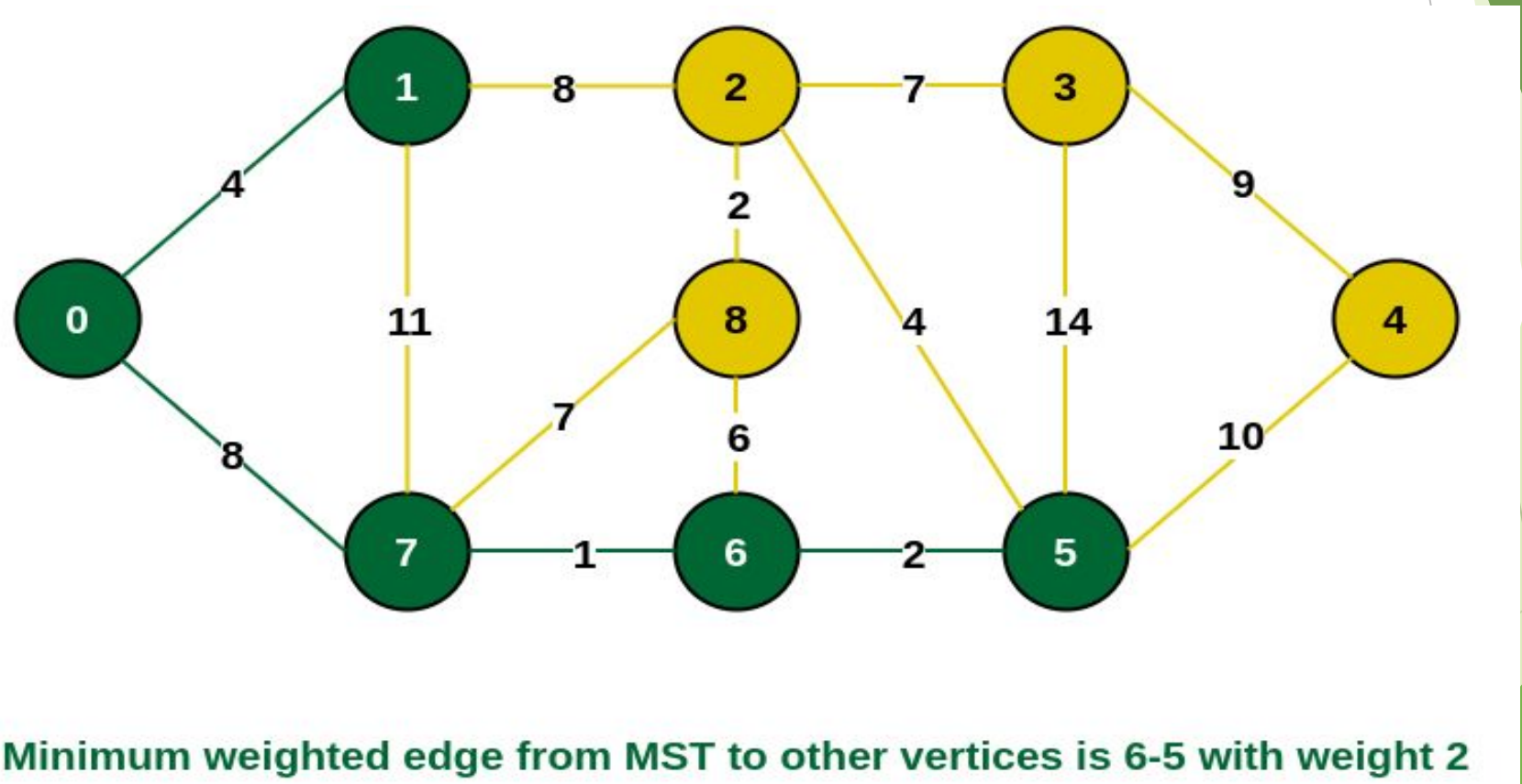
Minimum weighted edge from MST to other vertices is 0-7 with weight 8

Step 4: *The edges that connect the incomplete MST with the fringe vertices are {1, 2}, {7, 6} and {7, 8}. Add the edge {7, 6} and the vertex 6 in the MST as it has the least weight (i.e., 1).*

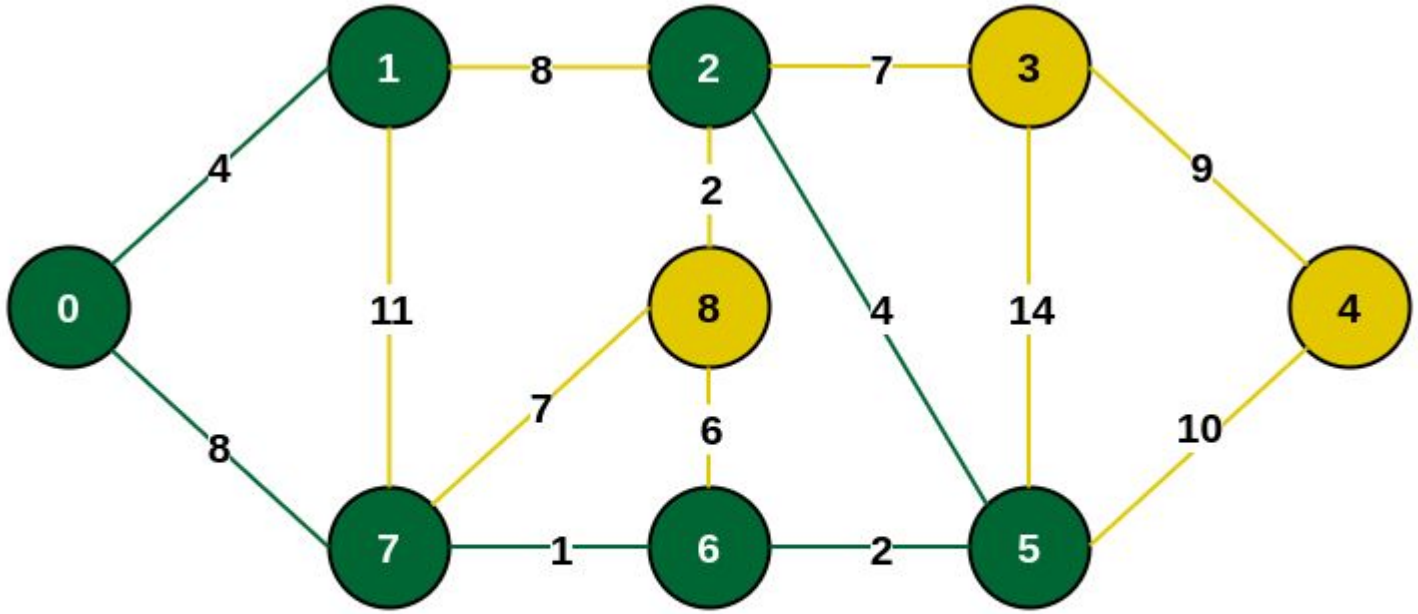


Minimum weighted edge from MST to other vertices is 7-6 with weight 1

Step 5: *The connecting edges now are {7, 8}, {1, 2}, {6, 8} and {6, 5}. Include edge {6, 5} and vertex 5 in the MST as the edge has the minimum weight (i.e., 2) among them.*

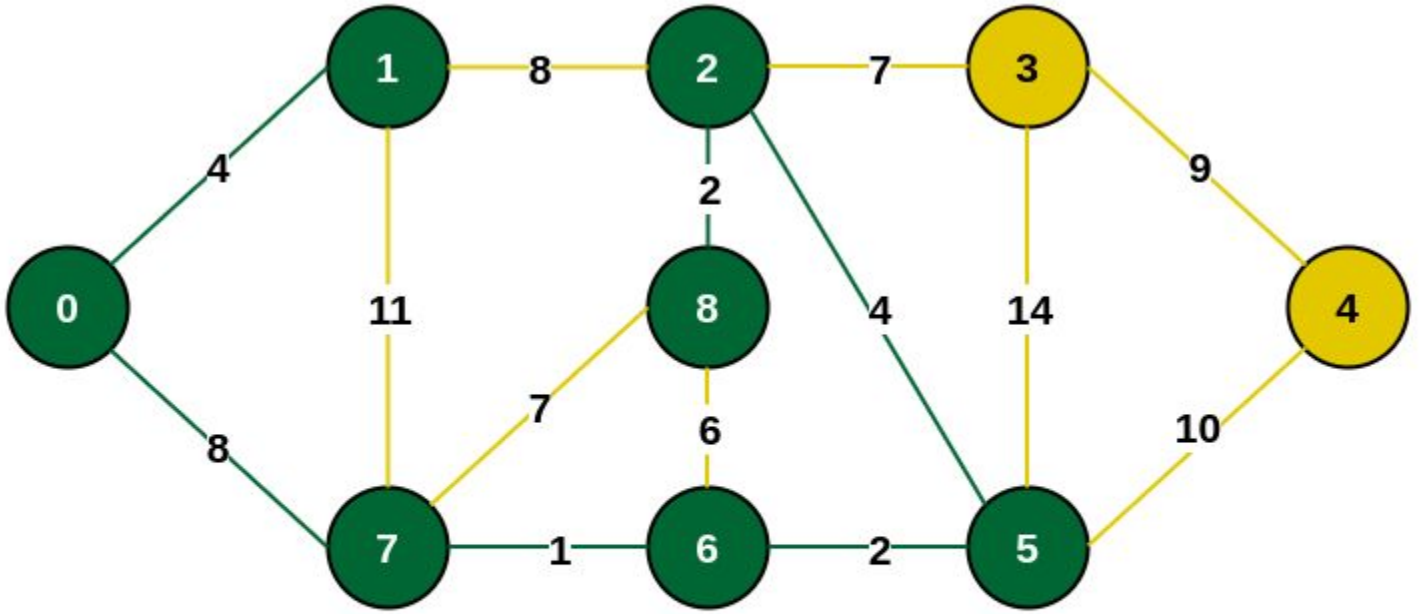


Step 6: *Among the current connecting edges, the edge {5, 2} has the minimum weight. So include that edge and the vertex 2 in the MST.*



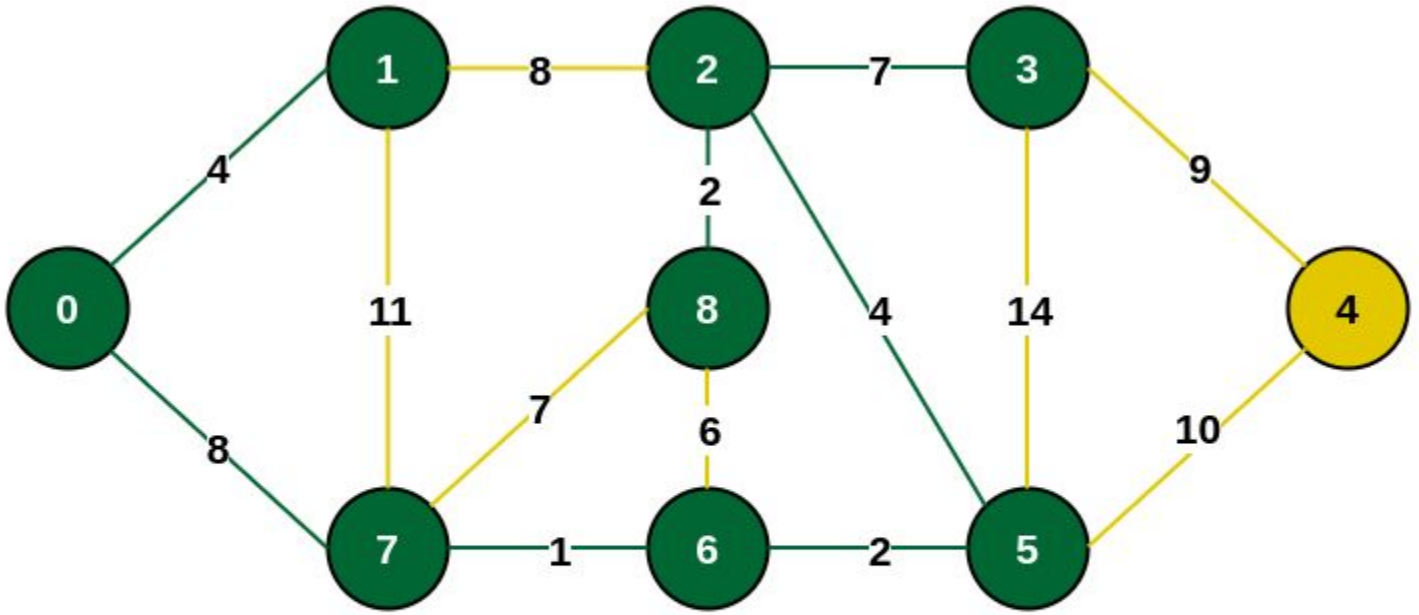
Minimum weighted edge from MST to other vertices is 5-2 with weight 4

Step 7: *The connecting edges between the incomplete MST and the other edges are {2, 8}, {2, 3}, {5, 3} and {5, 4}. The edge with minimum weight is edge {2, 8} which has weight 2. So include this edge and the vertex 8 in the MST.*



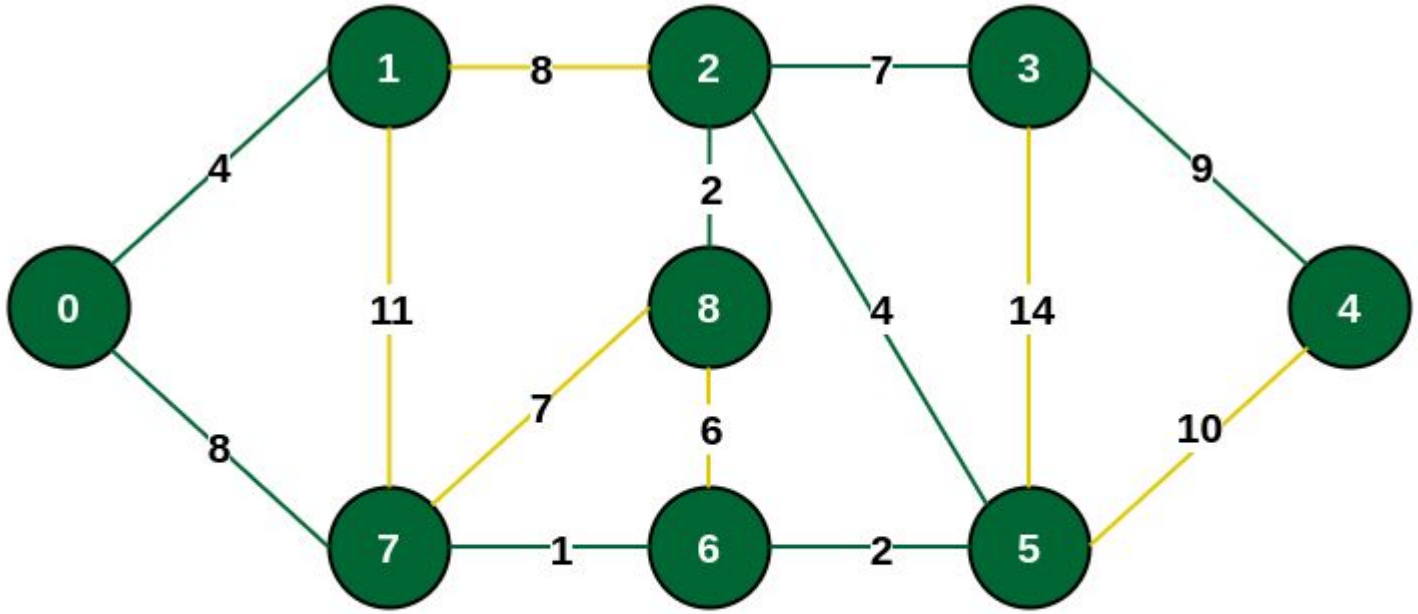
Minimum weighted edge from MST to other vertices is 2-8 with weight 2

Step 8: *See here that the edges {7, 8} and {2, 3} both have same weight which are minimum. But 7 is already part of MST. So we will consider the edge {2, 3} and include that edge and vertex 3 in the MST.*



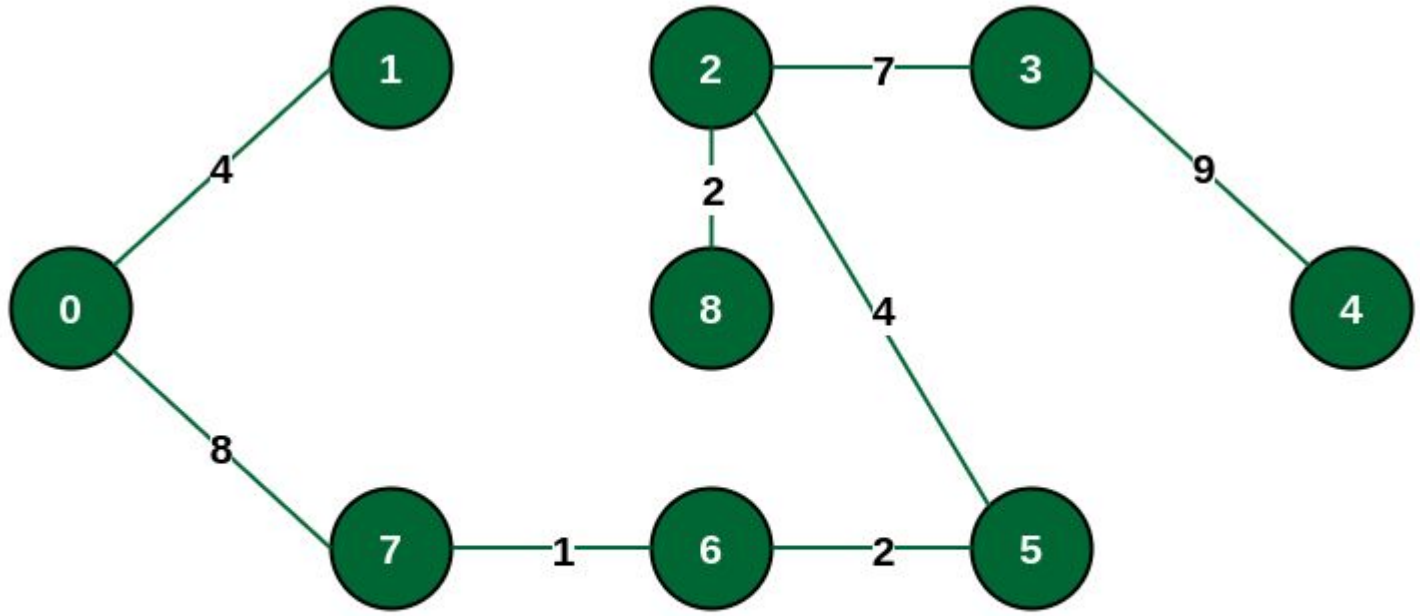
Minimum weighted edge from MST to other vertices is 2-3 with weight 7

Step 9: *Only the vertex 4 remains to be included. The minimum weighted edge from the incomplete MST to 4 is {3, 4}.*



Minimum weighted edge from MST to other vertices is 3-4 with weight 9

*The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = **37***



The final structure of MST

Applications of Minimum Spanning Trees:

- **Network design**: Spanning trees can be used in network design to find the minimum number of connections required to connect all nodes. Minimum spanning trees, in particular, can help minimize the cost of the connections by selecting the cheapest edges.
- **Image processing**: Spanning trees can be used in image processing to identify regions of similar intensity or color, which can be useful for segmentation and classification tasks.
- **Biology**: Spanning trees and minimum spanning trees can be used in biology to construct phylogenetic trees to represent evolutionary relationships among species or genes.
- **Social network analysis**: Spanning trees and minimum spanning trees can be used in social network analysis to identify important connections and relationships among individuals or groups.

Single-Source Shortest Path :

Dijkstra's algorithm is a popular algorithms for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist ***Edsger W. Dijkstra*** *in 1956.*

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

- Dijkstra's algorithm can work on both directed graphs and undirected graphs as this algorithm is designed to work on any type of graph as long as it meets the requirements of having non-negative edge weights and being connected.
- **In a directed graph,** each edge has a direction, indicating the direction of travel between the vertices connected by the edge. In this case, the algorithm follows the direction of the edges when searching for the shortest path.
- **In an undirected graph,** the edges have no direction, and the algorithm can traverse both forward and backward along the edges when searching for the shortest path.

Need for Dijkstra's Algorithm (Purpose and Use-Cases)

The need for Dijkstra's algorithm arises in many applications where finding the shortest path between two points is crucial.

For example, It can be used in the routing protocols for computer networks and also used by map systems to find the shortest path between starting point and the Destination (as explained in How does Google Maps work?)

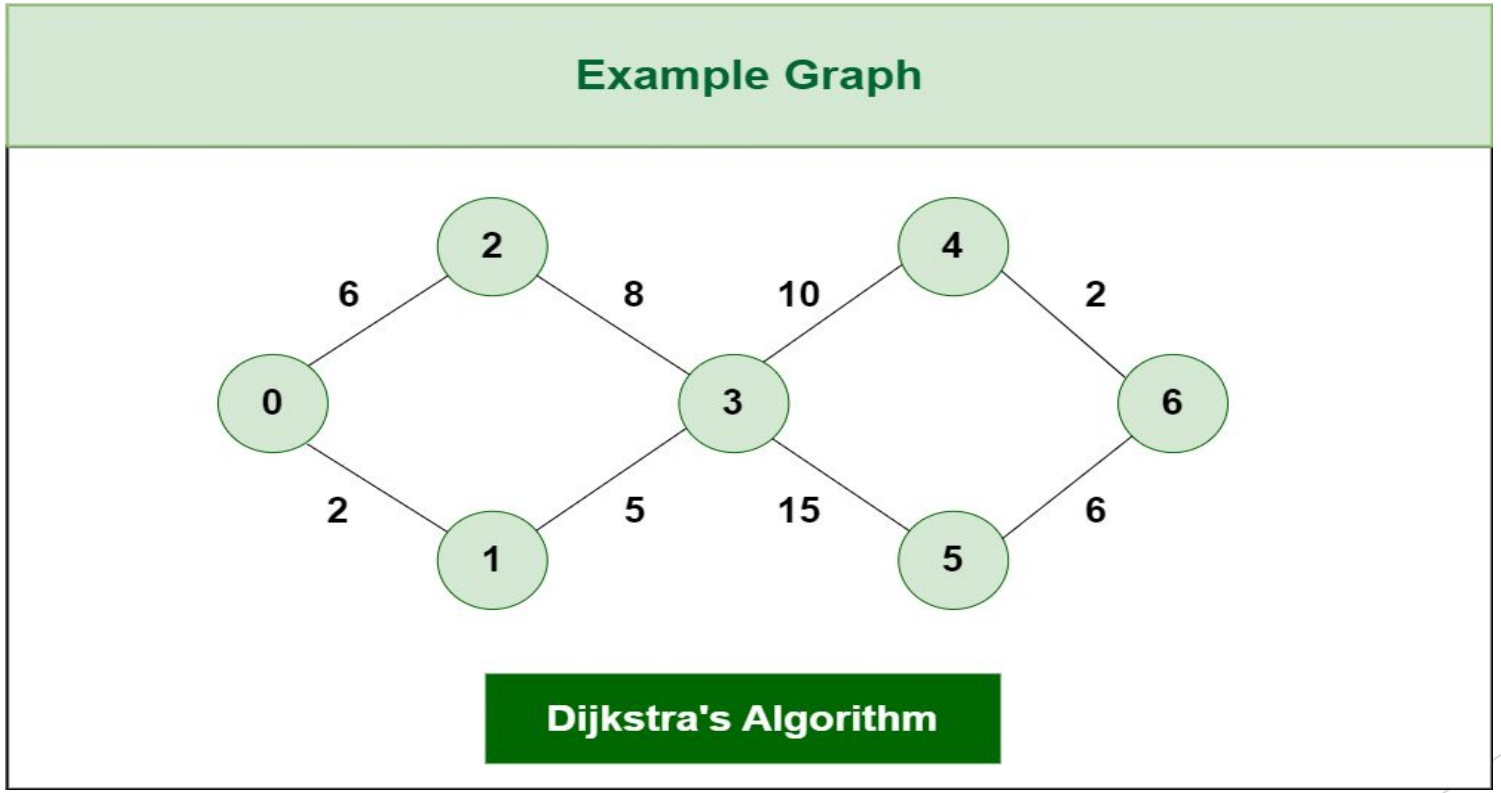
Algorithm for Dijkstra's Algorithm:

1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.
3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes are unvisited.

How does Dijkstra's Algorithm works?

Let's see how Dijkstra's Algorithm works with an example given below: Dijkstra's Algorithm will generate the shortest path from Node 0 to all other Nodes in the graph.

Consider the below graph:



The algorithm will generate the shortest path from node 0 to all the other nodes in the graph.

For this graph, we will assume that the weight of the edges represents the distance between two nodes.

As, we can see we have the shortest path from,

Node 0 to Node 1, from

Node 0 to Node 2, from

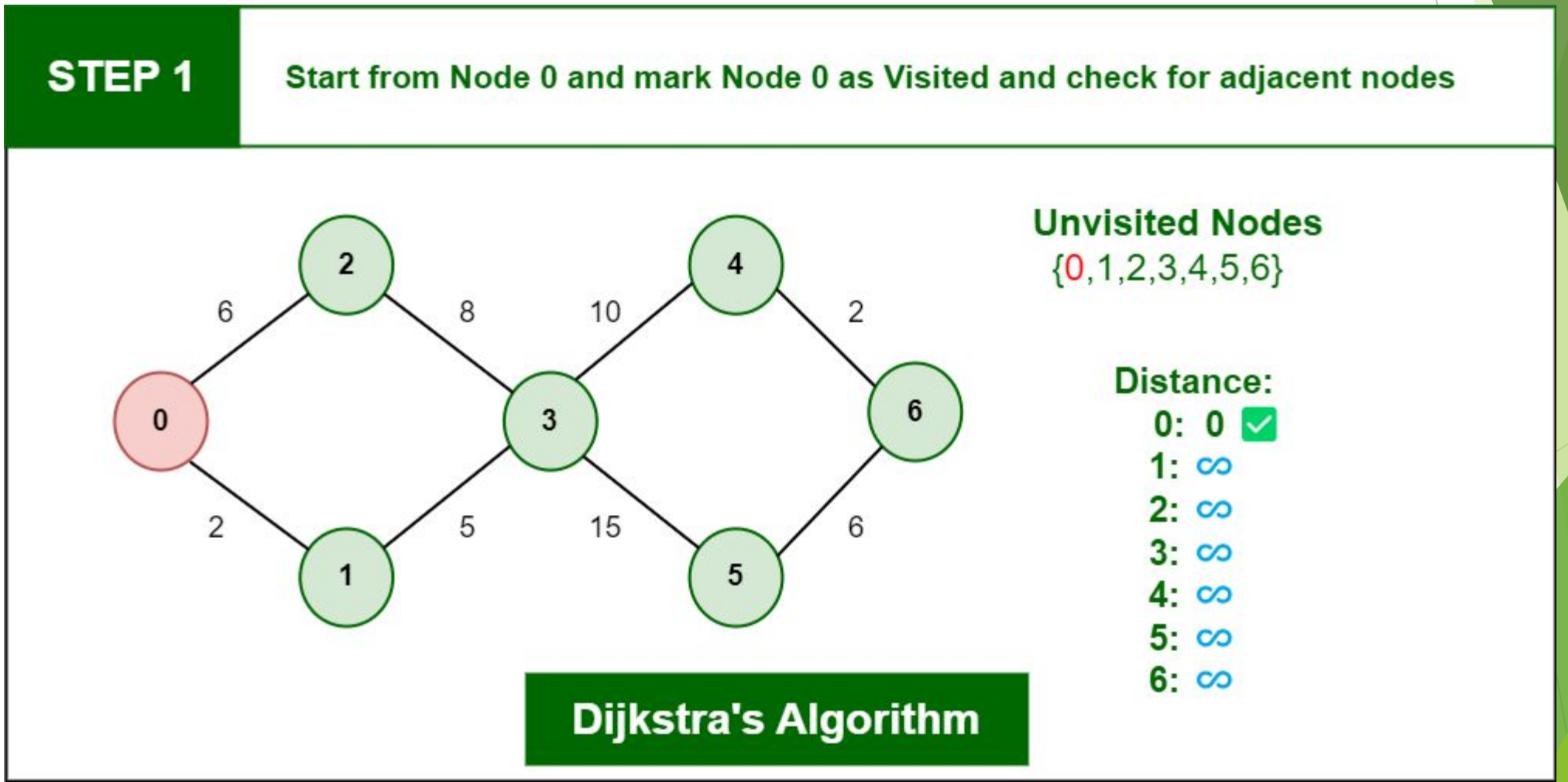
Node 0 to Node 3, from

Node 0 to Node 4, from

Node 0 to Node 6.

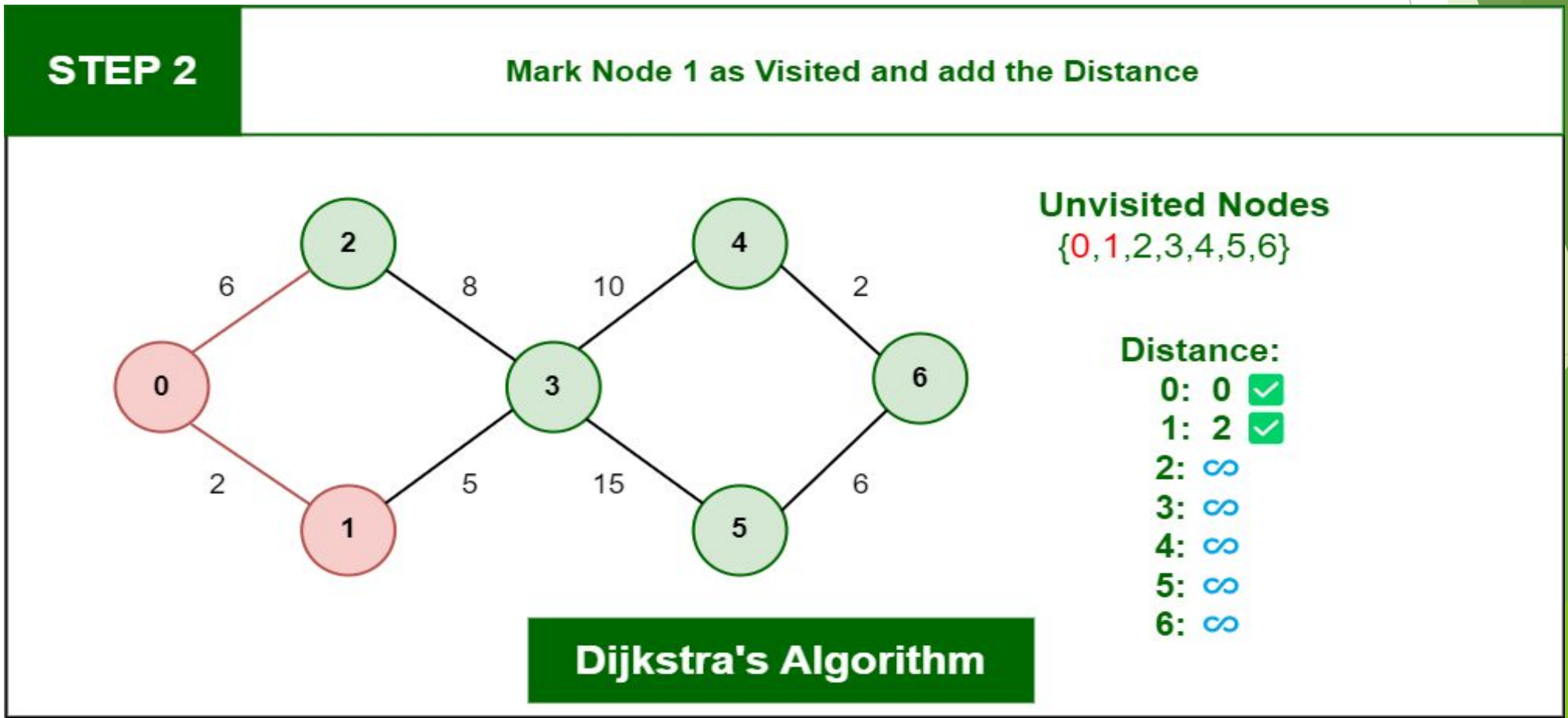
Initially we have a set of resources given below :
The Distance from the source node to itself is 0. In this example the source node is 0.
The distance from the source node to all other node is unknown so we mark all of them as infinity.
Example: 0 -> 0, 1-> ∞,2-> ∞,3-> ∞,4-> ∞,5-> ∞,6-> ∞.
we'll also have an array of unvisited elements that will keep track of unvisited or unmarked Nodes.
Algorithm will complete when all the nodes marked as visited and the distance between them added to the path. ***Unvisited Nodes:- 0 1 2 3 4 5 6.***

Step 1: *Start from Node 0 and mark Node as visited as you can check in below image visited Node is marked red.*



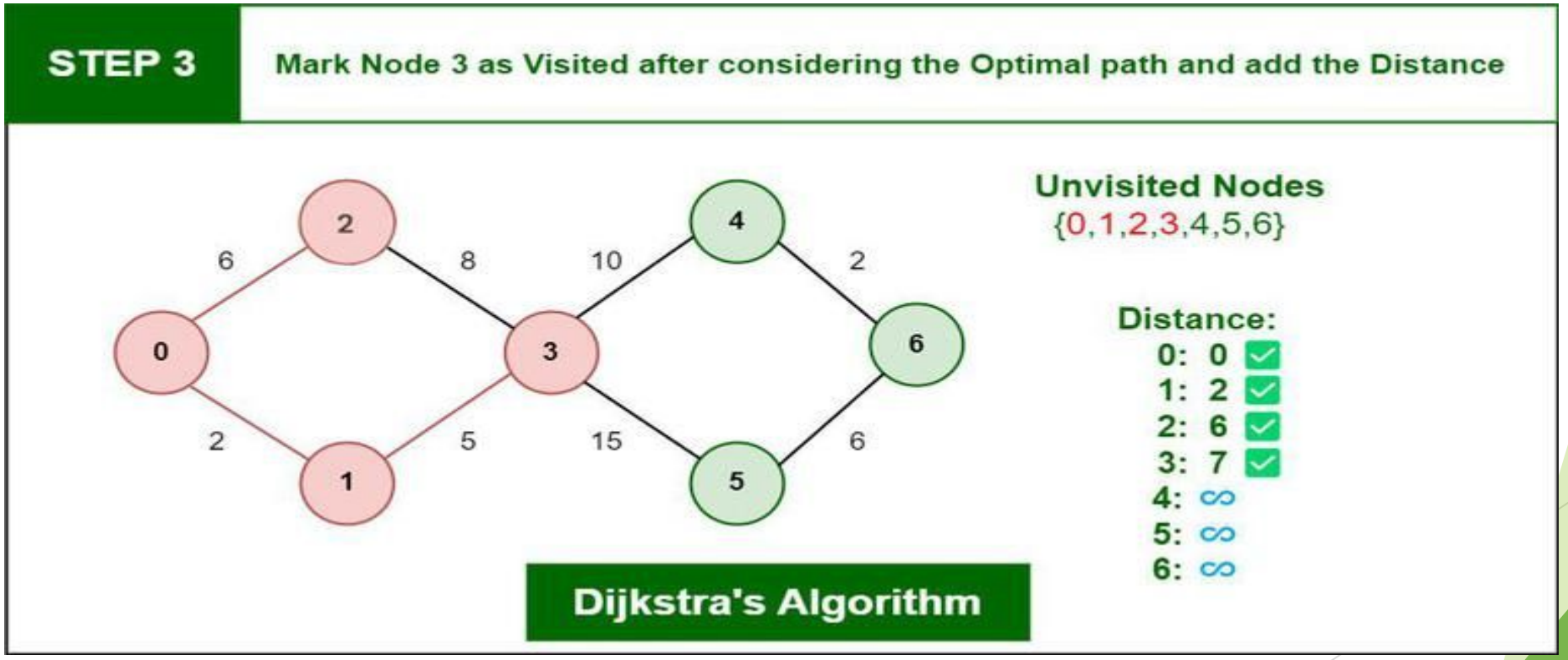
Step 2: *Check for adjacent Nodes, Now we have to choices (Either choose Node1 with distance 2 or either choose Node 2 with distance 6) and choose Node with minimum distance. In this step* ***Node 1*** *is Minimum distance adjacent Node, so marked it as visited and add up the distance.*

Distance: Node 0 -> Node 1 = 2



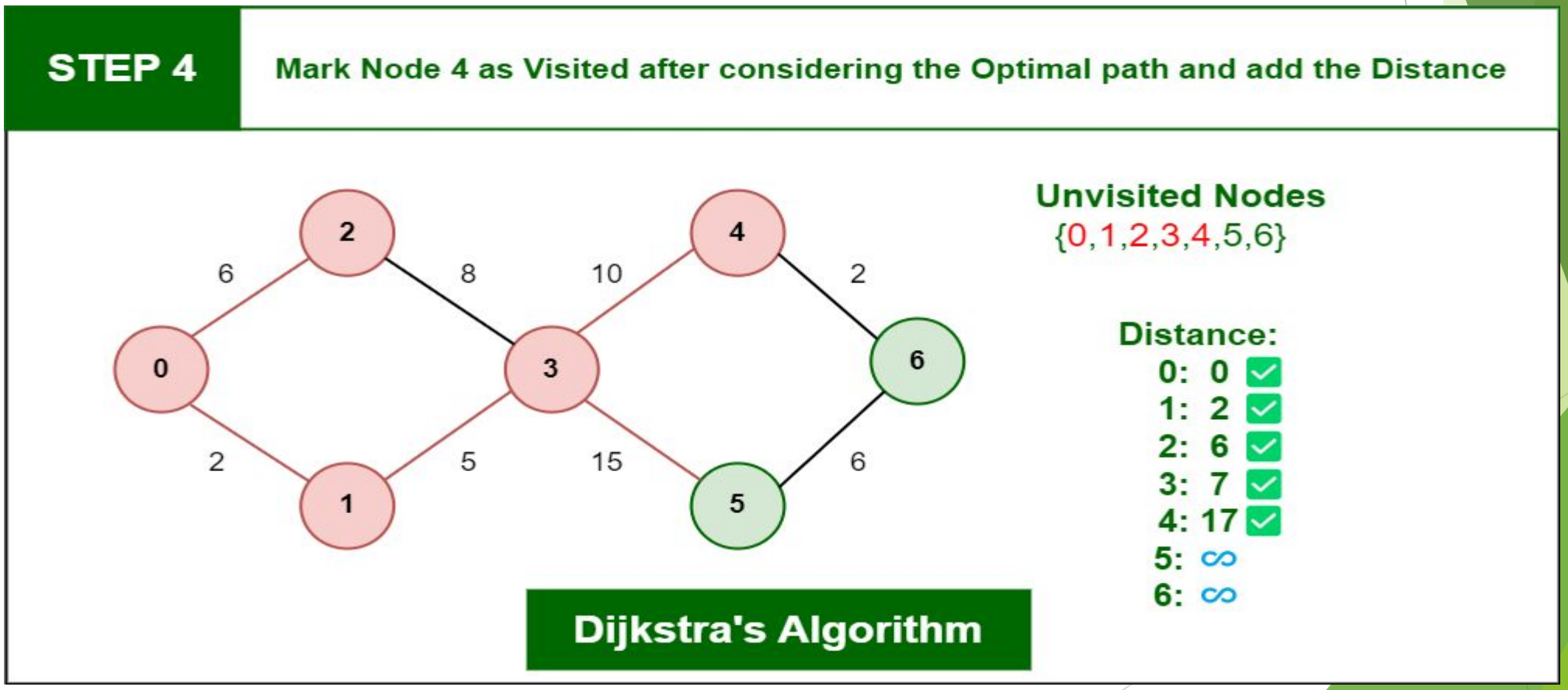
Step 3: *Then Move Forward and check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:*

Distance: Node 0 -> Node 1 -> Node 3 = 2 + 5 = 7



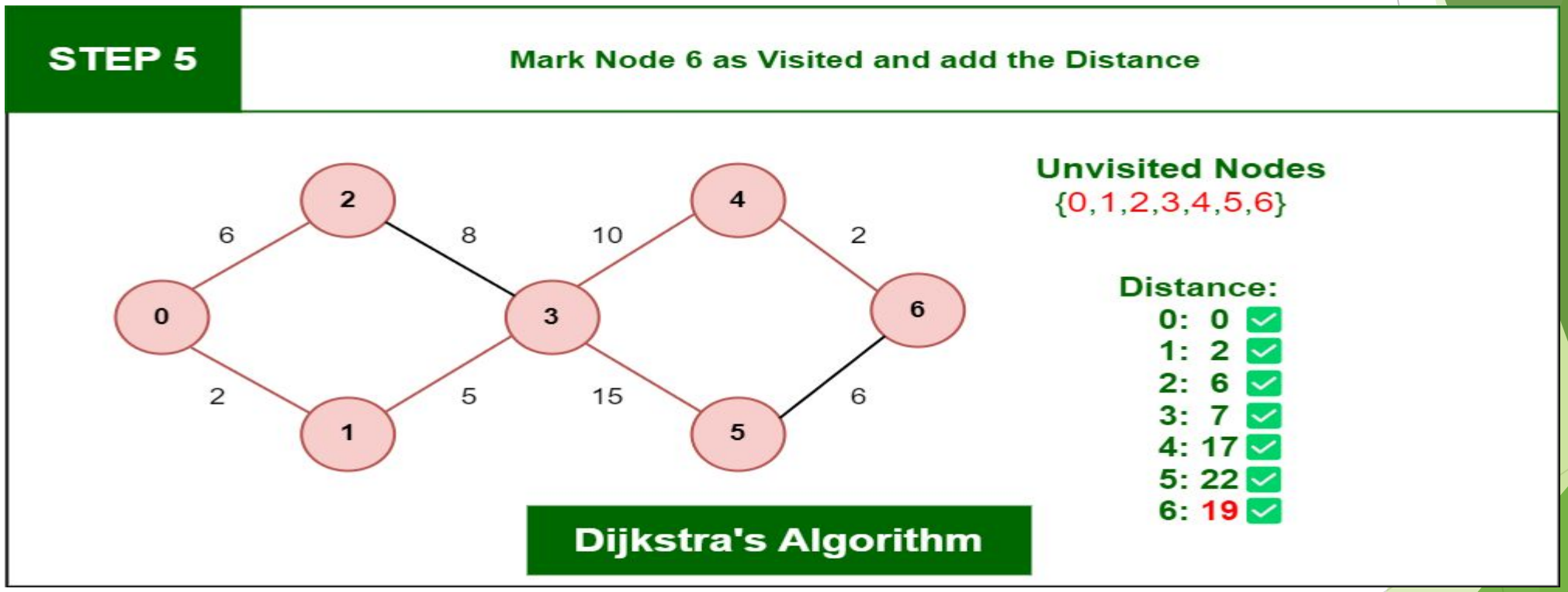
Step 4: *Again we have two choices for adjacent Nodes (Either we can choose Node 4 with distance 10 or either we can choose Node 5 with distance 15) so choose Node with minimum distance. In this step* ***Node 4*** *is Minimum distance adjacent Node, so marked it as visited and add up the distance.*

Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 = 2 + 5 + 10 = 17



Step 5: *Again, Move Forward and check for adjacent Node which is* ***Node 6****, so marked it as visited and add up the distance, Now the distance will be:*

Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 -> Node 6 = 2 + 5 + 10 + 2 = 19



So, the Shortest Distance from the Source Vertex is 19 which is optimal one

Dijkstra's Algorithm has several real-world use cases, some of which are as follows:

1. **Digital Mapping Services in Google Maps:** Many times we have tried to find the distance in G-Maps, from one city to another, or from your location to the nearest desired location. There encounters the Shortest Path Algorithm, as there are various routes/paths connecting them but it has to show the minimum distance, so Dijkstra's Algorithm is used to find the minimum distance between two locations along the path. Consider India as a graph and represent a city/place with a vertex and the route between two cities/places as an edge, then by using this algorithm, the shortest routes between any two cities/places or from one city/place to another city/place can be calculated.
2. **Social Networking Applications:** In many applications you might have seen the app suggests the list of friends that a particular user may know. How do you think many social media companies implement this feature efficiently, especially when the system has over a billion users. The standard Dijkstra algorithm can be applied using the shortest path between users measured through handshakes or connections among them. When the social networking graph is very small, it uses standard Dijkstra's algorithm along with some other features to find the shortest paths, and however, when the graph is becoming bigger and bigger, the standard algorithm takes a few several seconds to count and alternate advanced algorithms are used.

3.Telephone Network: As we know, in a telephone network, each line has a bandwidth, ‘b’. The bandwidth of the transmission line is the highest frequency that line can support. Generally, if the frequency of the signal is higher in a certain line, the signal is reduced by that line. Bandwidth represents the amount of information that can be transmitted by the line. If we imagine a city to be a graph, the vertices represent the switching stations, and the edges represent the transmission lines and the weight of the edges represents ‘b’. So as you can see it can fall into the category of shortest distance problem, for which the Dijkstra is can be used.

4.IP routing to find Open shortest Path First: Open Shortest Path First (OSPF) is a link-state routing protocol that is used to find the best path between the source and the destination router using its own Shortest Path First. Dijkstra’s algorithm is widely used in the routing protocols required by the routers to update their forwarding table. The algorithm provides the shortest cost path from the source router to other routers in the network.

5.Flighting Agenda: For example, If a person needs software for making an agenda of flights for customers. The agent has access to a database with all airports and flights. Besides the flight number, origin airport, and destination, the flights have departure and arrival time. Specifically, the agent wants to determine the earliest arrival time for the destination given an origin airport and start time. There this algorithm comes into use.

6.Designate file server: To designate a file server in a LAN(local area network), Dijkstra's algorithm can be used. Consider that an infinite amount of time is required for transmitting files from one computer to another computer. Therefore to minimize the number of "hops" from the file server to every other computer on the network the idea is to use Dijkstra's algorithm to minimize the shortest path between the networks resulting in the minimum number of hops.

7.Robotic Path: Nowadays, drones and robots have come into existence, some of which are manual, some automated. The drones/robots which are automated and are used to deliver the packages to a specific location or used for a task are loaded with this algorithm module so that when the source and destination is known, the robot/drone moves in the ordered direction by following the shortest path to keep delivering the package in a minimum amount of time.

Huffman Coding Algorithm :

Data may be compressed using the Huffman Coding technique to become smaller without losing any of its information. After David Huffman, who created it in the beginning? **Data that contains frequently repeated characters is typically compressed using Huffman coding.**

A well-known Greedy algorithm is Huffman Coding. The size of code allocated to a character relies on the frequency of the character, which is why it is referred to be a greedy algorithm. **The short-length variable code is assigned to the character with the highest frequency, and vice versa for characters with lower frequencies. It employs a variable-length** encoding, which means that it gives each character in the provided data stream a different variable-length code.

Prefix Rule :

Essentially, this rule states that the code that is allocated to a character shall not be another code's prefix. If this rule is broken, various ambiguities may appear when decoding the Huffman tree that has been created.

Let's look at an illustration of this rule to better comprehend it: For each character, a code is provided, such as:

1. a - 0
2. b - 1
3. c - 01

Assuming that the produced bit stream is 001, the code may be expressed as follows when decoded:

1. 0 0 1 = aab
2. 0 01 = ac

- The Huffman Code is obtained for each distinct character in primarily two steps:

- Create a Huffman Tree first using only the unique characters in the data stream provided.
- Second, we must proceed through the constructed Huffman Tree, assign codes to the characters, and then use those codes to decode the provided text.

Steps to Take in Huffman Coding

The steps used to construct the Huffman tree using the characters provided

1. Input:
2. string str = "abbcdbccdaabbeeebeab"

If Huffman Coding is employed in this case for data compression, the following information must be determined for decoding:

- For each character, the Huffman Code
- Huffman-encoded message length (in bits), average code length
- Utilizing the formulas covered below, the final two of them are discovered.

How Can a Huffman Tree Be Constructed from Input Characters?

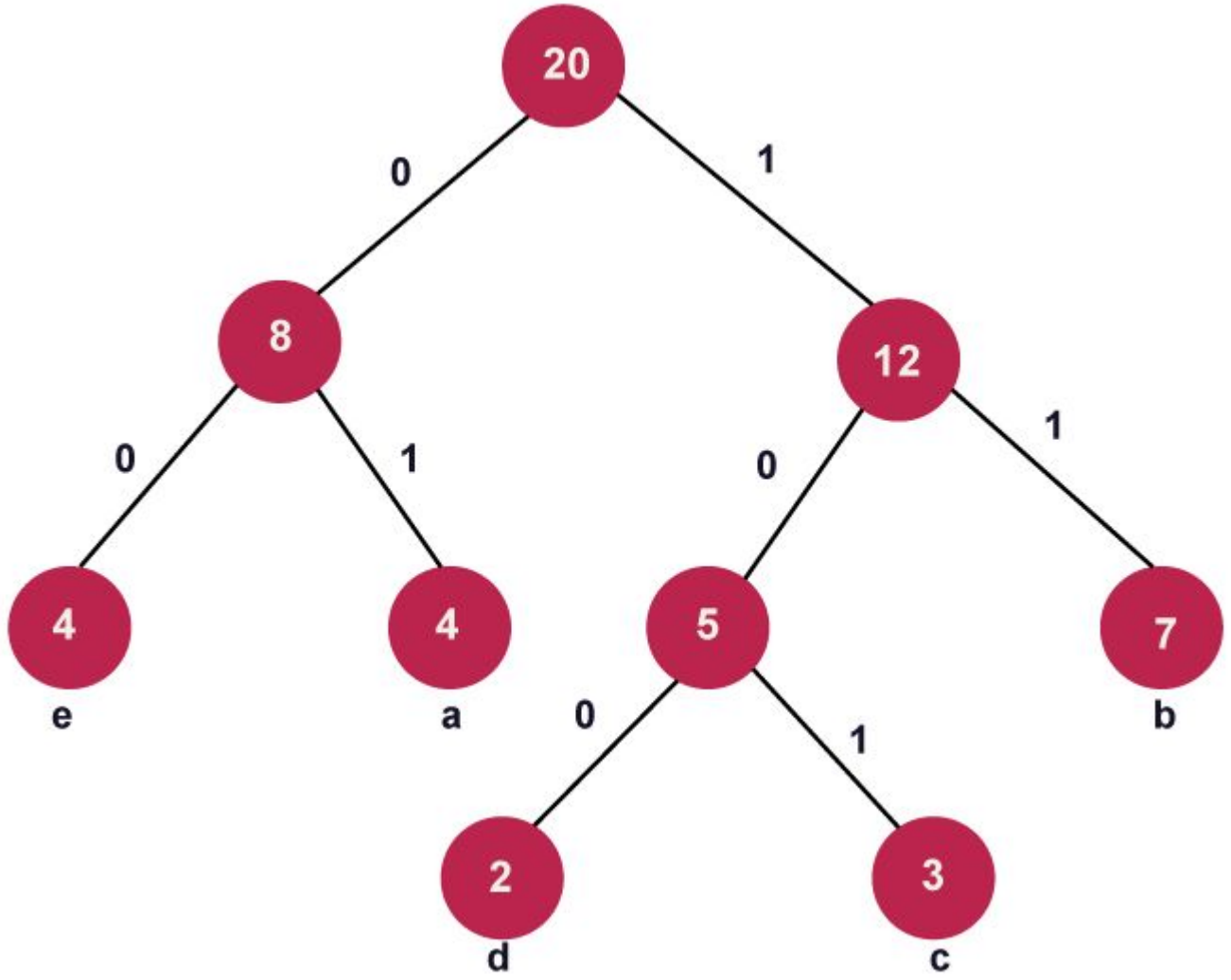
The frequency of each character in the provided string must first be determined

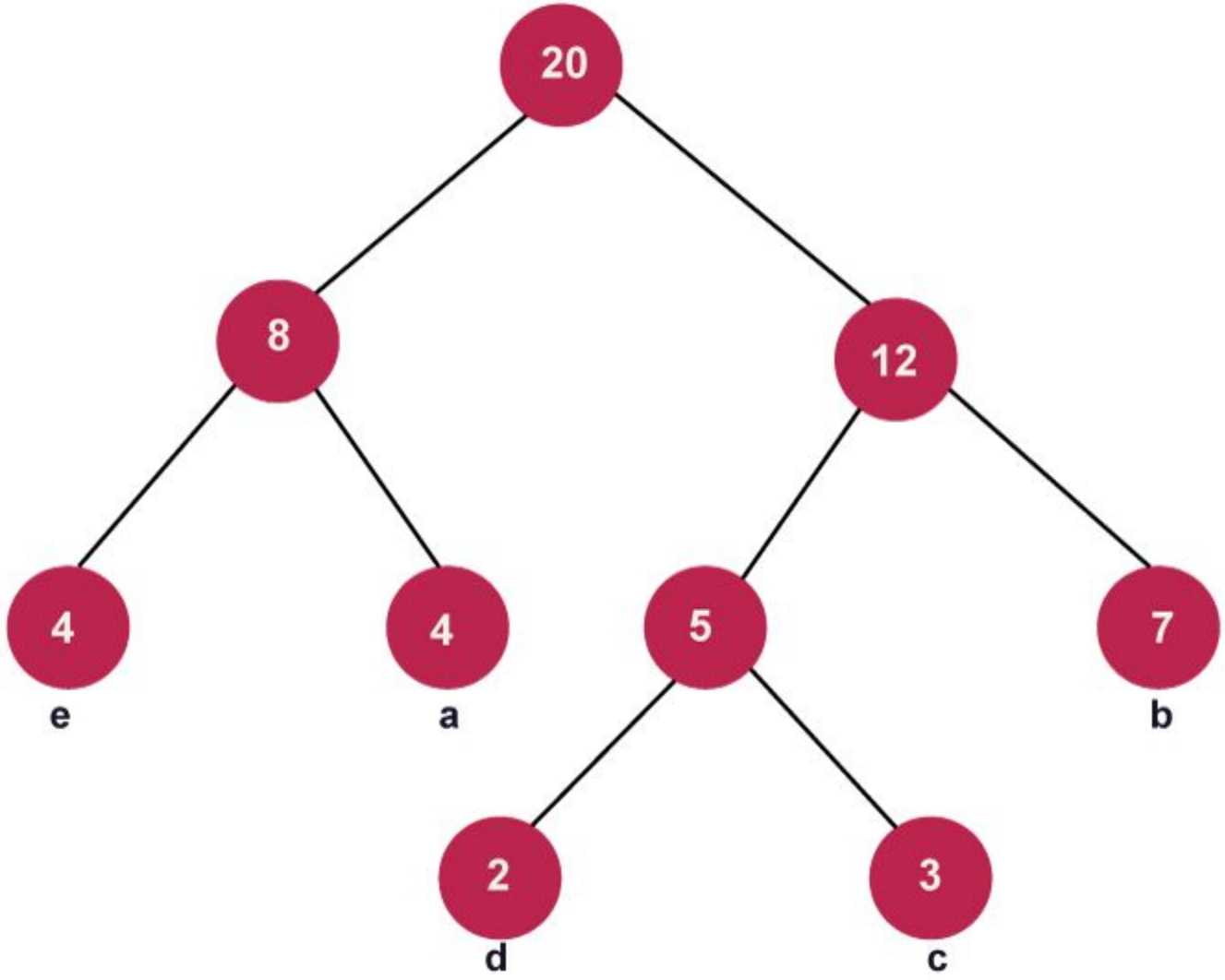
Character	Frequency
a	4
b	7
c	3
d	2
e	4

1. Sort the characters by frequency, ascending. These are kept in a Q/min-heap priority queue.
2. For each distinct character and its frequency in the data stream, create a leaf node.
3. Remove the two nodes with the two lowest frequencies from the nodes, and the new root of the tree is created using the sum of these frequencies.
    1. Make the first extracted node its left child and the second extracted node its right child while extracting the nodes with the lowest frequency from the min-heap.
    2. To the min-heap, add this node.
    3. Since the left side of the root should always contain the minimum frequency.
4. Repeat steps 3 and 4 until there is only one node left on the heap, or all characters are represented by nodes in the tree. The tree is finished when just the root node remains.

Examples of Huffman Coding

Let's use an illustration to explain the algorithm:




20
8
12
4
4
5
7
e
a
b
2
3
d
c

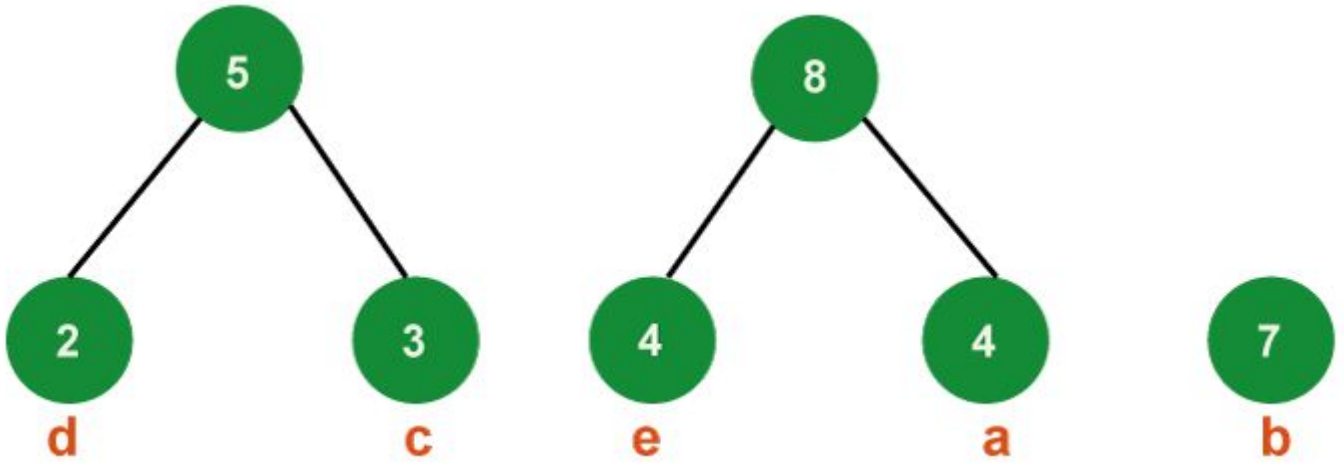
Algorithm for Huffman Coding

Step 1: Build a min-heap in which each node represents the root of a tree with a single node and holds 5 (the number of unique characters from the provided stream of data).

Step 2: Obtain two minimum frequency nodes from the min heap in step two. Add a third internal node, frequency 2 + 3 = 5, which is created by joining the two extracted nodes.

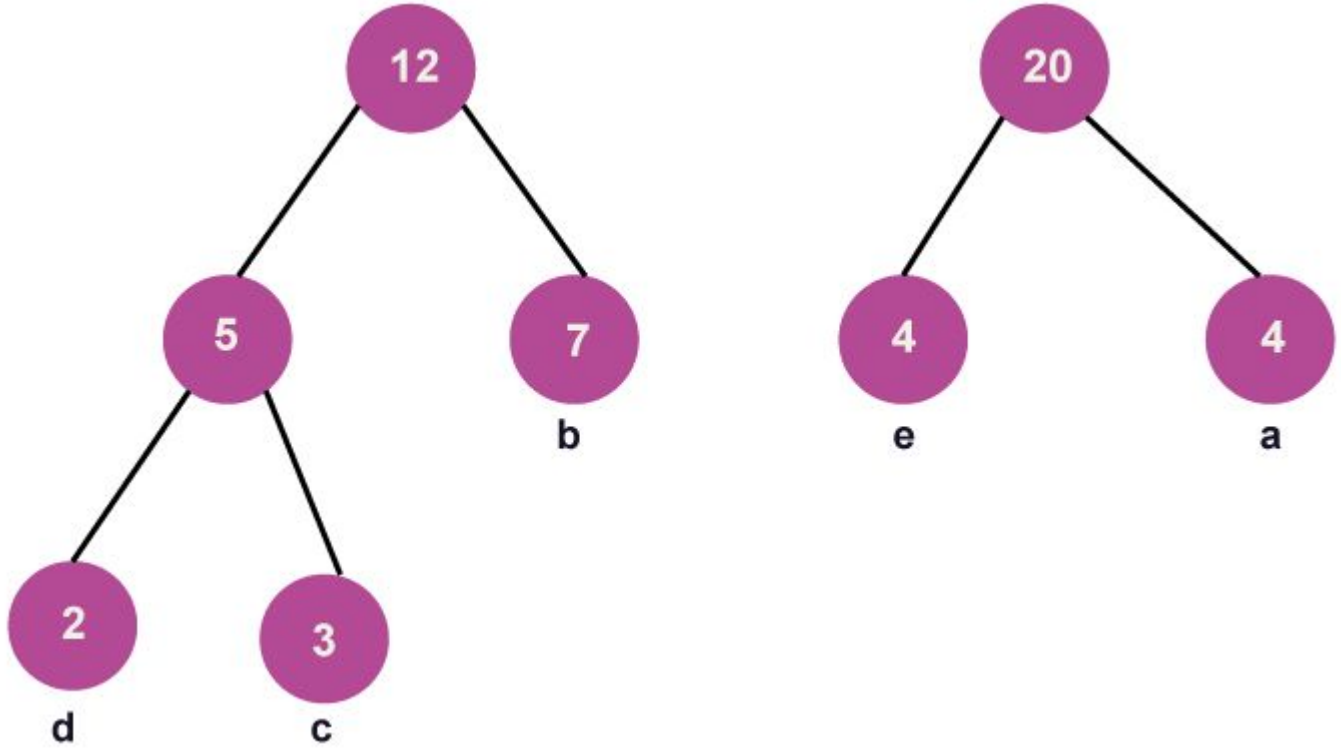
•Now, there are 4 nodes in the min-heap, 3 of which are the roots of trees with a single element each, and 1 of which is the root of a tree with two elements.

Step 3: Get the two minimum frequency nodes from the heap in a similar manner in step three. Additionally, add a new internal node formed by joining the two extracted nodes; its frequency in the tree should be 4 + 4 = 8.



Step 4: Get the two minimum frequency nodes in step four. Additionally, add a new internal node formed by joining the two extracted nodes; its frequency in the tree should be 5 + 7 = 12.

- When creating a Huffman tree, we must ensure that the minimum value is always on the left side and that the second value is always on the right side. Currently, the image below shows the tree that has formed:



Step 5: Get the following two minimum frequency nodes in step 5. Additionally, add a new internal node formed by joining the two extracted nodes; its frequency in the tree should be 12 + 8 = 20.

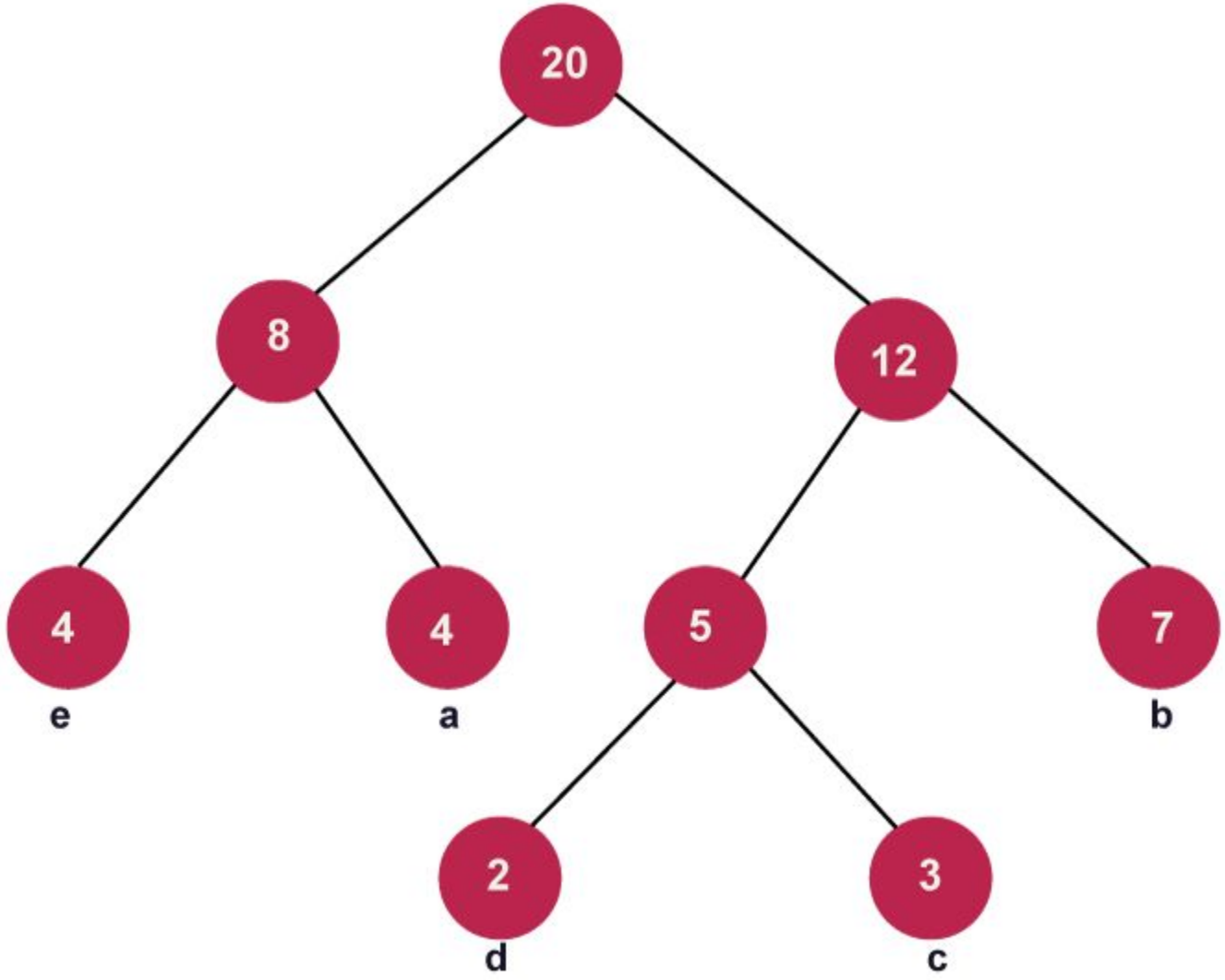
Continue until all of the distinct characters have been added to the tree. The Huffman tree created for the specified cast of characters is shown in the above image.

Now, for each non-leaf node, assign 0 to the left edge and 1 to the right edge to create the code for each letter.

Rules to follow for determining edge weights:

- We should give the right edges weight 1 if you give the left edges weight 0.
- If the left edges are given weight 1, the right edges must be given weight 0.
- Any of the two aforementioned conventions may be used.
- However, follow the same protocol when decoding the tree as well.

Following the weighting, the modified tree is displayed as follows:


20
8
12
4
4
5
7
e
a
b
2
3
d
c

END