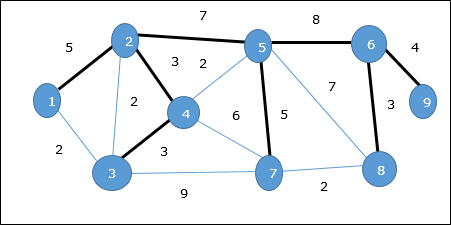
**Assignment No.3 (b)**

**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.

**Minimum Spanning tree Applications**

* To find paths in the map
* To design networks like telecommunication networks, water supply networks, and electrical grids.



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.

There are two methods to find Minimum Spanning Tree

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

# **Kruskal's Algorithm :**

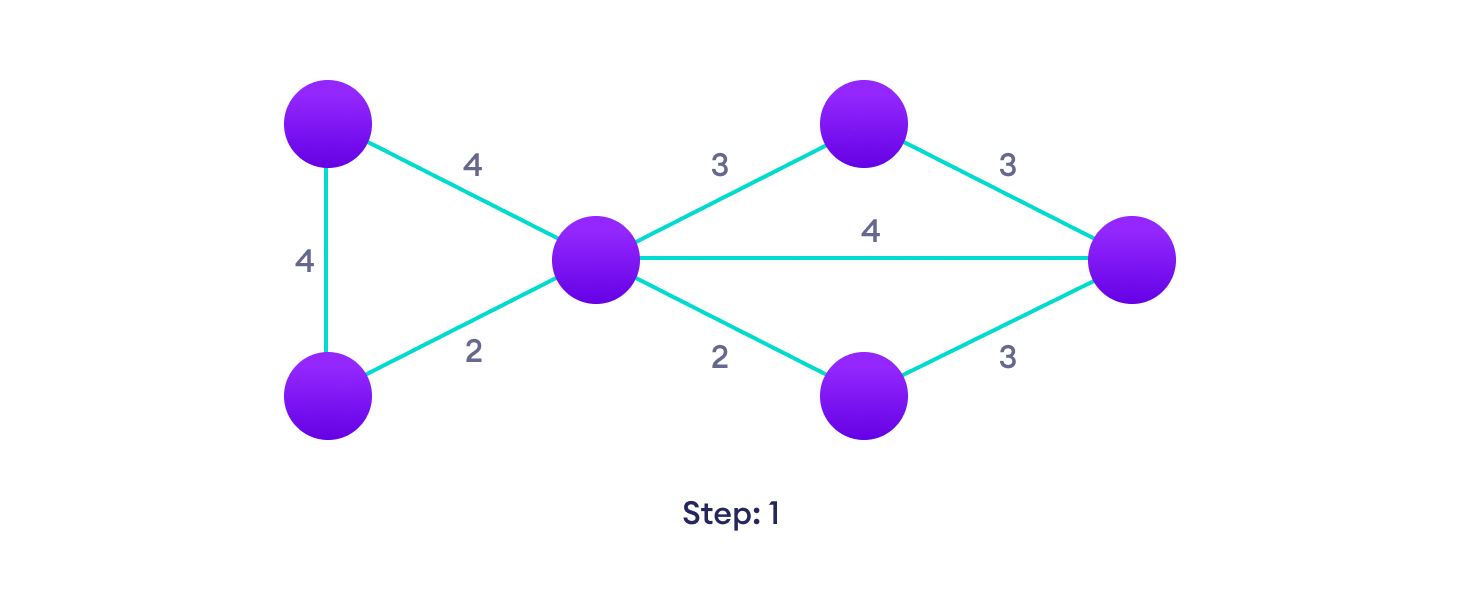
Kruskal's algorithm is a [minimum spanning tree](https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree#minimum-spanning) algorithm that takes a graph as input and finds the subset of the edges of that graph which

* form a tree that includes every vertex
* has the minimum sum of weights among all the trees that can be formed from the graph

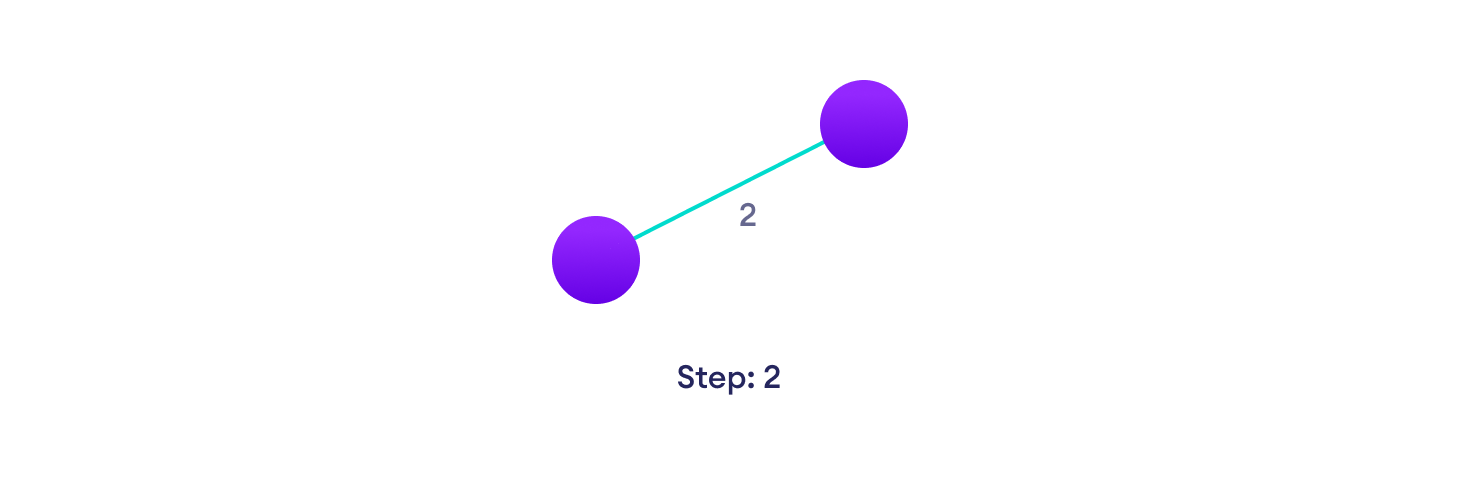
**Algorithm :**

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

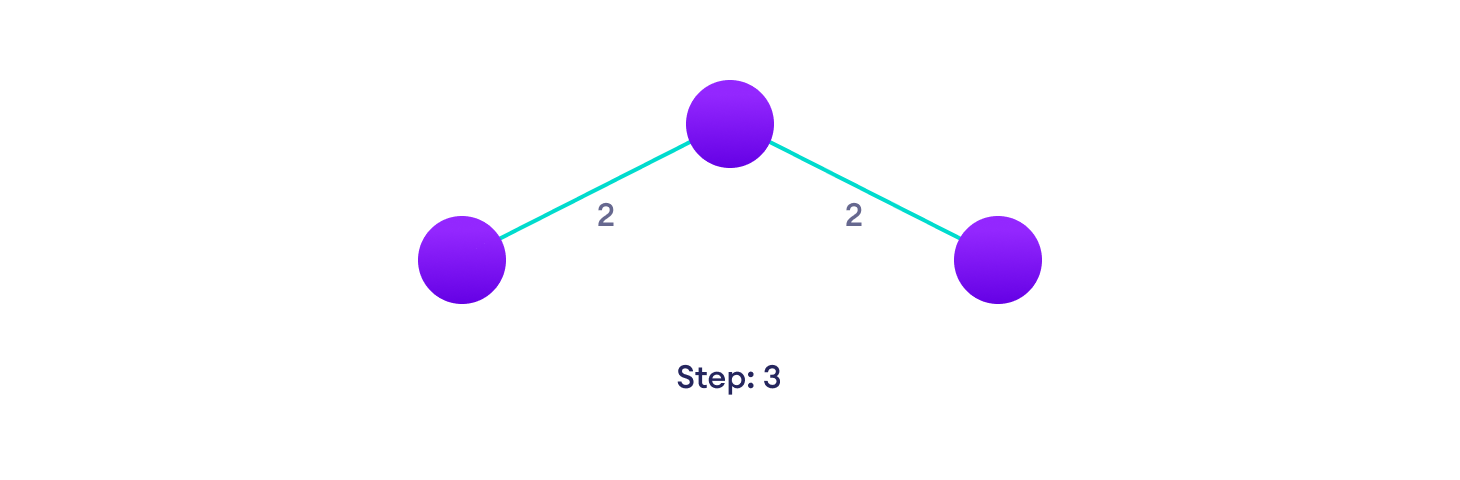
## Example of Kruskal's algorithm :



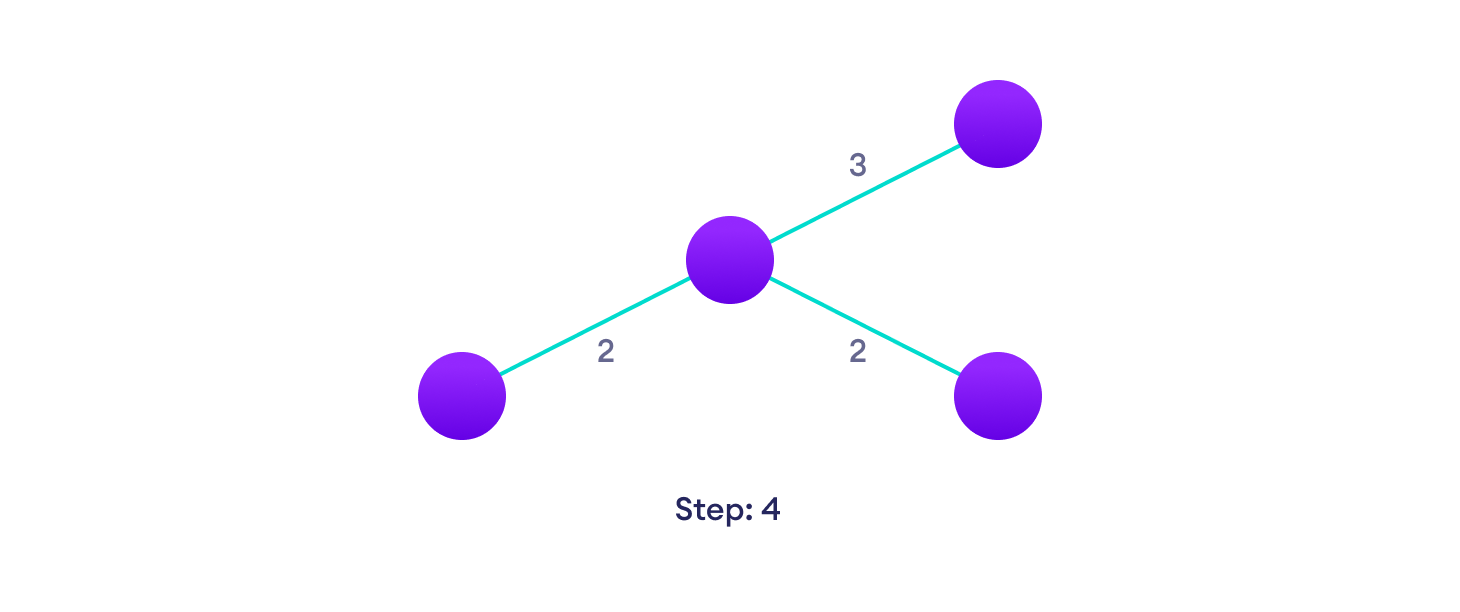
Start with a weighted graph



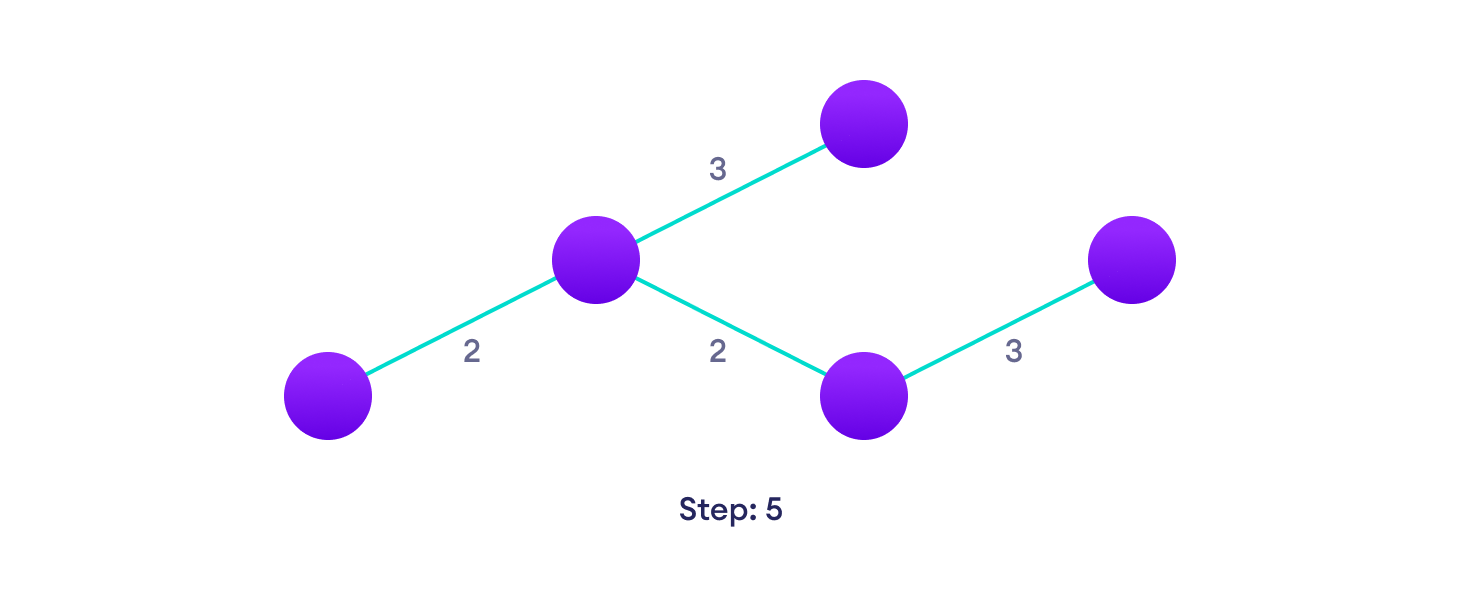
Choose the edge with the least weight, if there are more than 1, choose anyone



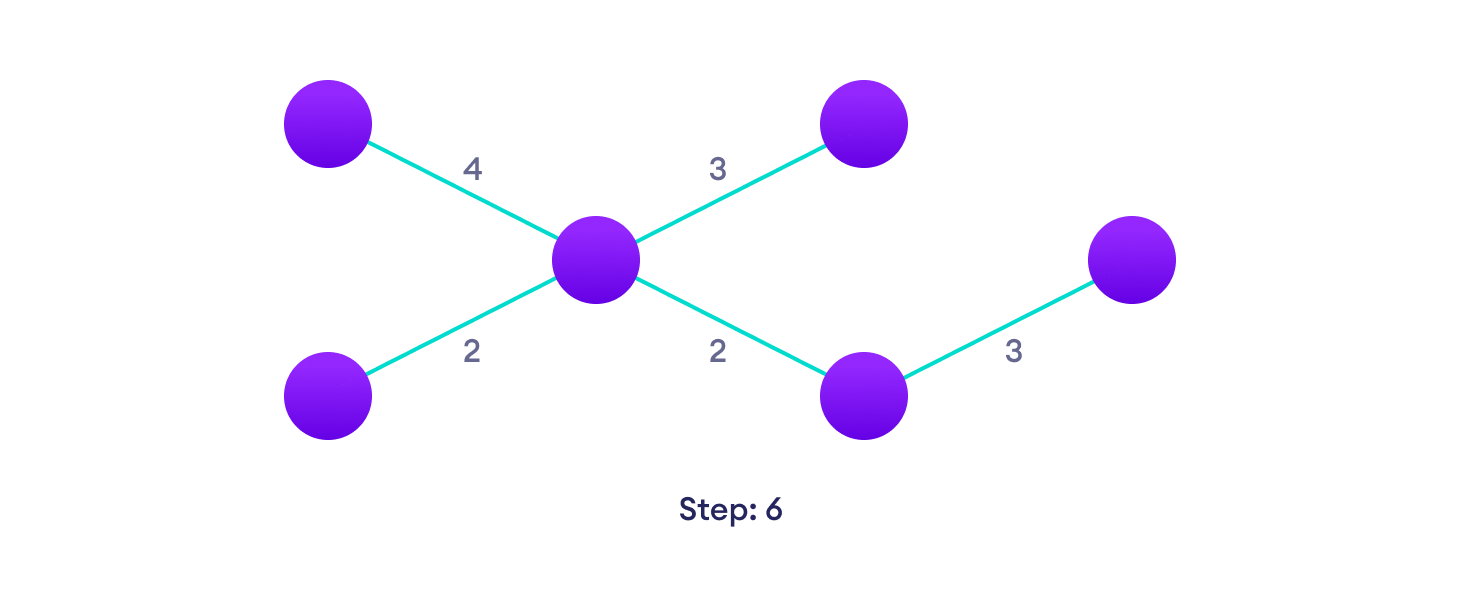
Choose the next shortest edge and add it



Choose the next shortest edge that doesn't create a cycle and add it



Choose the next shortest edge that doesn't create a cycle and add it



Repeat until you have a spanning tree

## Kruskal Algorithm Pseudocode :

KRUSKAL(G):

A = ∅

For each vertex v ∈ G.V:

MAKE-SET(v)

For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):

if FIND-SET(u) ≠ FIND-SET(v):

A = A ∪ {(u, v)}

UNION(u, v)

return A

## Kruskal's Algorithm Complexity :

The time complexity Of Kruskal's Algorithm is: O(E log E).

**Kruskal's Algorithm Applications**

* In order to layout electrical wiring
* In computer network (LAN connection)

Conclusion :

**Implementation :**

# Kruskal's algorithm in Python

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = []

def add\_edge(self, u, v, w):

self.graph.append([u, v, w])

# Search function

def find(self, parent, i):

if parent[i] == i:

return i

return self.find(parent, parent[i])

def apply\_union(self, parent, rank, x, y):

xroot = self.find(parent, x)

yroot = self.find(parent, y)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

else:

parent[yroot] = xroot

rank[xroot] += 1

# Applying Kruskal algorithm

def kruskal\_algo(self):

result = []

i, e = 0, 0

self.graph = sorted(self.graph, key=lambda item: item[2])

parent = []

rank = []

for node in range(self.V):

parent.append(node)

rank.append(0)

while e < self.V - 1:

u, v, w = self.graph[i]

i = i + 1

x = self.find(parent, u)

y = self.find(parent, v)

if x != y:

e = e + 1

result.append([u, v, w])

self.apply\_union(parent, rank, x, y)

for u, v, weight in result:

print("%d - %d: %d" % (u, v, weight))

g = Graph(6)

g.add\_edge(0, 1, 4)

g.add\_edge(0, 2, 4)

g.add\_edge(1, 2, 2)

g.add\_edge(1, 0, 4)

g.add\_edge(2, 0, 4)

g.add\_edge(2, 1, 2)

g.add\_edge(2, 3, 3)

g.add\_edge(2, 5, 2)

g.add\_edge(2, 4, 4)

g.add\_edge(3, 2, 3)

g.add\_edge(3, 4, 3)

g.add\_edge(4, 2, 4)

g.add\_edge(4, 3, 3)

g.add\_edge(5, 2, 2)

g.add\_edge(5, 4, 3)

g.kruskal\_algo()

**Output :**

1 - 2: 2

2 - 5: 2

2 - 3: 3

3 - 4: 3

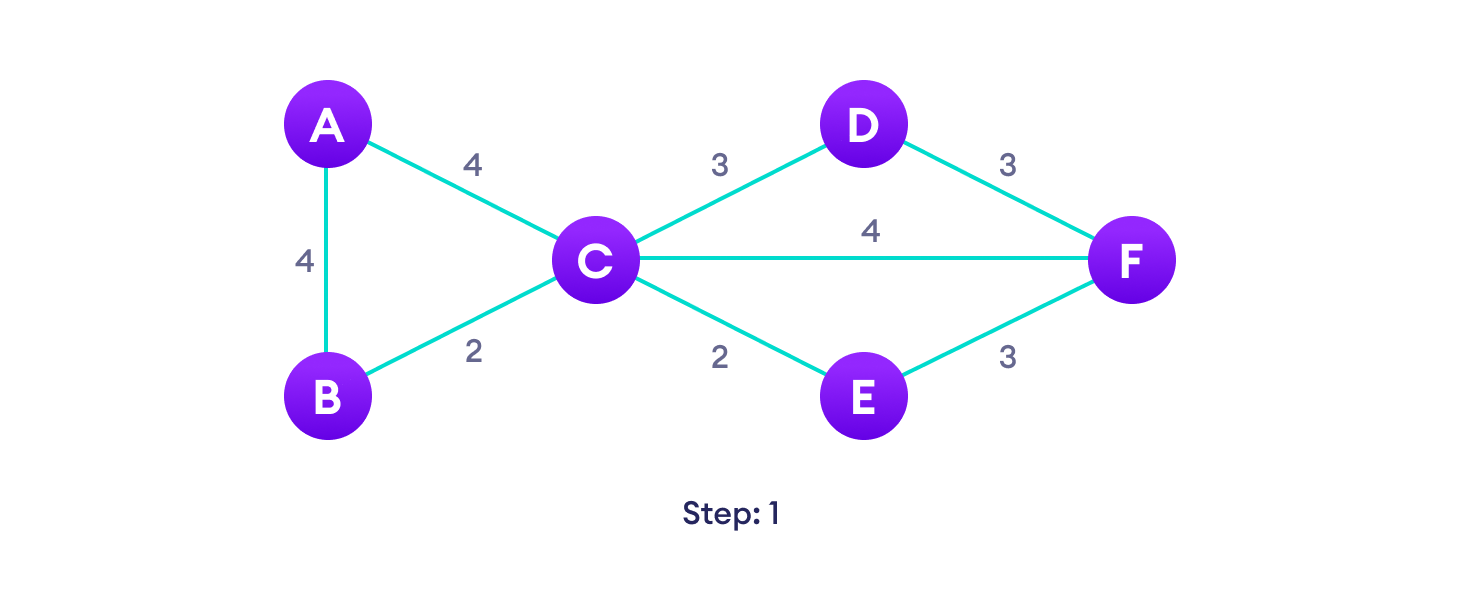
0 - 1: 4

**Prim's Algorithm**

Prim's algorithm is a [minimum spanning tree](https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree#minimum-spanning) algorithm that takes a graph as input and finds the subset of the edges of that graph which

* form a tree that includes every vertex
* has the minimum sum of weights among all the trees that can be formed from the graph

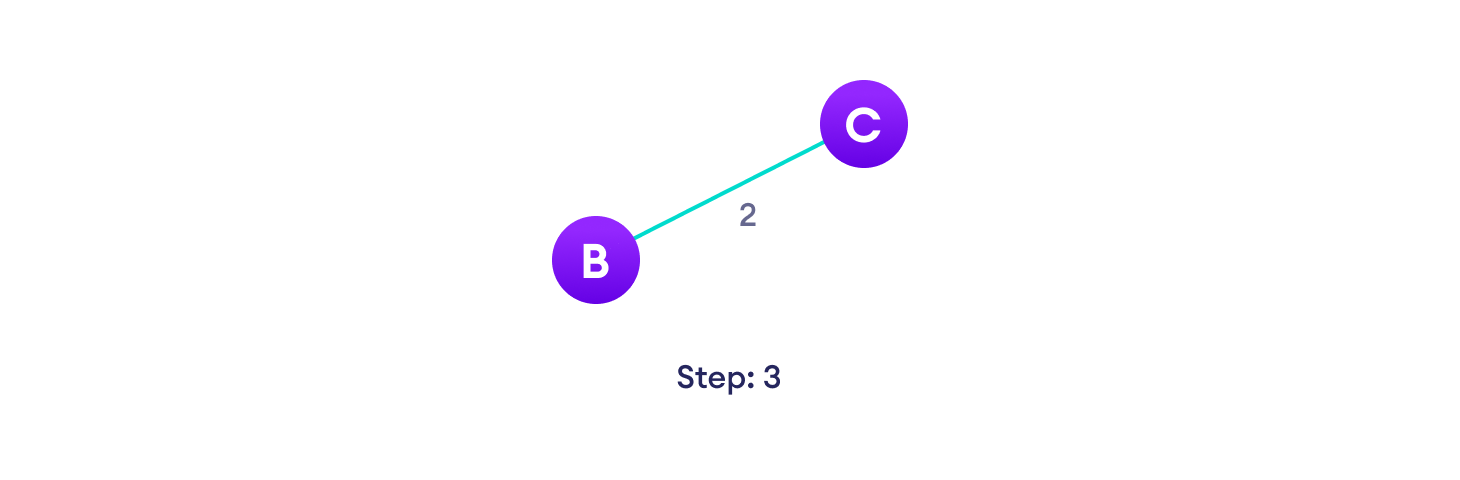
## Example of Prim's algorithm



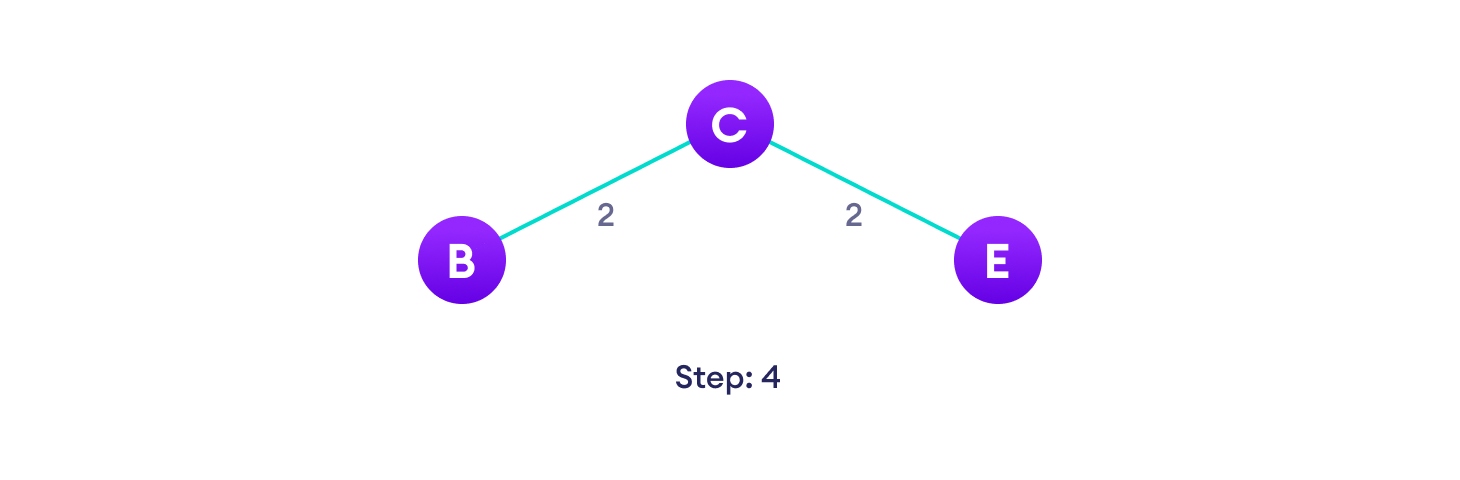
Start with a weighted graph



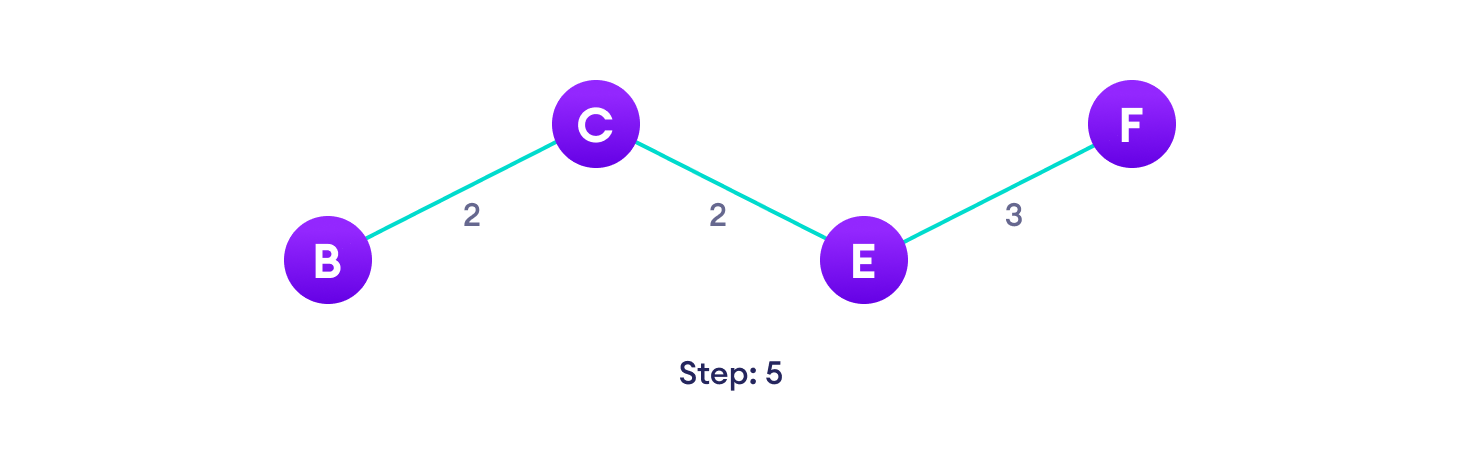
Choose a vertex



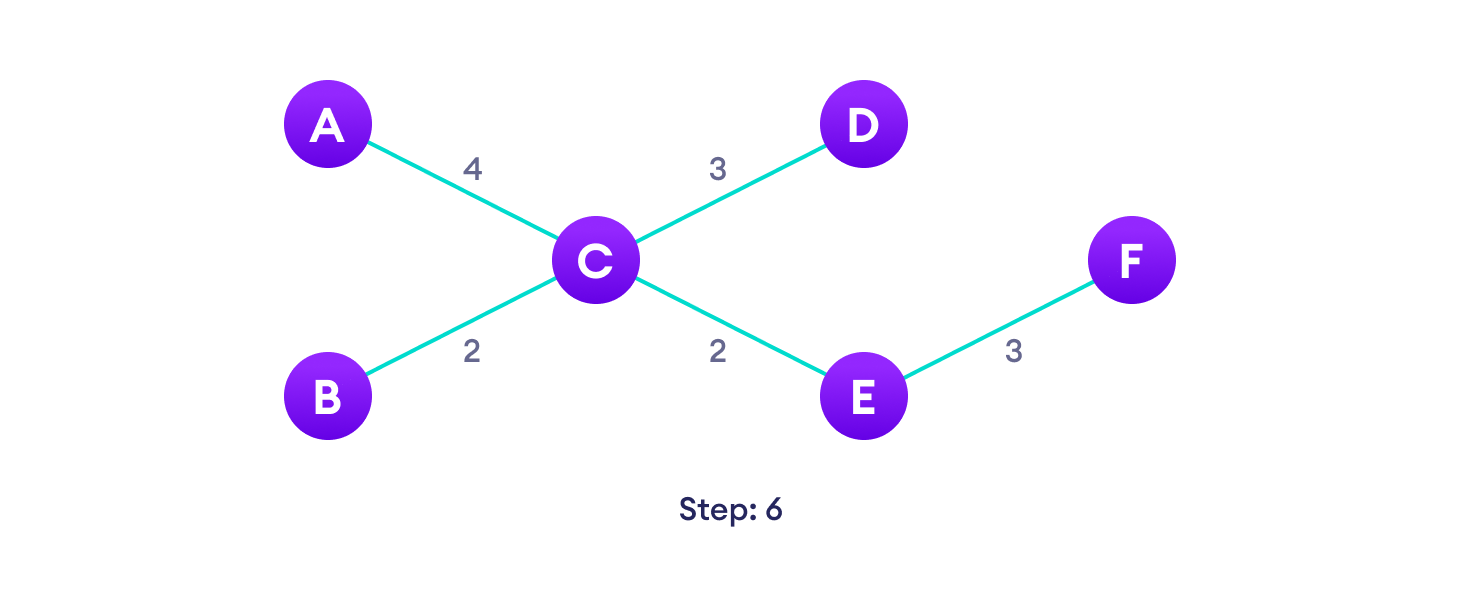
Choose the shortest edge from this vertex and add it



Choose the nearest vertex not yet in the solution



Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Repeat until you have a spanning tree

## Prim's Algorithm pseudocode

T = ∅;

U = { 1 };

while (U ≠ V)

let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;

T = T ∪ {(u, v)}

U = U ∪ {v}

## Prim's Algorithm Complexity

The time complexity of Prim's algorithm is O(E log V).

**Prim's Algorithm Application**

* Laying cables of electrical wiring
* In network designed
* To make protocols in network cycles

Conclusion :

**Implementation**

# Prim's Algorithm in Python

INF = 9999999

# number of vertices in graph

V = 5

# create a 2d array of size 5x5

# for adjacency matrix to represent graph

G = [[0, 9, 75, 0, 0],

[9, 0, 95, 19, 42],

[75, 95, 0, 51, 66],

[0, 19, 51, 0, 31],

[0, 42, 66, 31, 0]]

# create a array to track selected vertex

# selected will become true otherwise false

selected = [0, 0, 0, 0, 0]

# set number of edge to 0

no\_edge = 0

# the number of egde in minimum spanning tree will be

# always less than(V - 1), where V is number of vertices in

# graph

# choose 0th vertex and make it true

selected[0] = True

# print for edge and weight

print("Edge : Weight\n")

while (no\_edge < V - 1):

# For every vertex in the set S, find the all adjacent vertices

#, calculate the distance from the vertex selected at step 1.

# if the vertex is already in the set S, discard it otherwise

# choose another vertex nearest to selected vertex at step 1.

minimum = INF

x = 0

y = 0

for i in range(V):

if selected[i]:

for j in range(V):

if ((not selected[j]) and G[i][j]):

# not in selected and there is an edge

if minimum > G[i][j]:

minimum = G[i][j]

x = i

y = j

print(str(x) + "-" + str(y) + ":" + str(G[x][y]))

selected[y] = True

no\_edge += 1

**Output :**

Edge : Weight

0-1:9

1-3:19

3-4:31

3-2:51