**Selection Sort**

def selectionSort(arr):

for i in range(len(arr)):

min = float('-inf')

for j in range(i + 1, len(arr)):

if arr[i] > arr[j]:

arr[i],arr[j] = arr[j], arr[i]

return arr

print(selectionSort([89,56,45,34,65,76]))

**Prism Algorithm**

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

**Kruskal Algorithm**

class Graph:

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

self.V = vertices

self.graph = []

# Function to add an edge to graph

def addEdge(self, u, v, w):

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

# A utility function to find set of an element i

# (truly uses path compression technique)

def find(self, parent, i):

if parent[i] != i:

# Reassignment of node's parent

# to root node as

# path compression requires

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

return parent[i]

# A function that does union of two sets of x and y

# (uses union by rank)

def union(self, parent, rank, x, y):

# Attach smaller rank tree under root of

# high rank tree (Union by Rank)

if rank[x] < rank[y]:

parent[x] = y

elif rank[x] > rank[y]:

parent[y] = x

# If ranks are same, then make one as root

# and increment its rank by one

else:

parent[y] = x

rank[x] += 1

# The main function to construct MST

# using Kruskal's algorithm

def KruskalMST(self):

# This will store the resultant MST

result = []

# An index variable, used for sorted edges

i = 0

# An index variable, used for result[]

e = 0

# Sort all the edges in

# non-decreasing order of their

# weight

self.graph = sorted(self.graph,

key=lambda item: item[2])

parent = []

rank = []

# Create V subsets with single elements

for node in range(self.V):

parent.append(node)

rank.append(0)

# Number of edges to be taken is less than to V-1

while e < self.V - 1:

# Pick the smallest edge and increment

# the index for next iteration

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

i = i + 1

x = self.find(parent, u)

y = self.find(parent, v)

# If including this edge doesn't

# cause cycle, then include it in result

# and increment the index of result

# for next edge

if x != y:

e = e + 1

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

self.union(parent, rank, x, y)

# Else discard the edge

minimumCost = 0

print("Edges in the constructed MST")

for u, v, weight in result:

minimumCost += weight

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

print("Minimum Spanning Tree", minimumCost)

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

g = Graph(4)

g.addEdge(0, 1, 10)

g.addEdge(0, 2, 6)

g.addEdge(0, 3, 5)

g.addEdge(1, 3, 15)

g.addEdge(2, 3, 4)

# Function call

g.KruskalMST()