1) **COMBINATION SUM**

def combination\_sum(candidates, target):

def backtrack(remaining, combo, start):

if remaining == 0:

result.append(list(combo))

return

elif remaining < 0:

return

for i in range(start, len(candidates)):

combo.append(candidates[i])

backtrack(remaining - candidates[i], combo, i) # Not i + 1 because we can reuse the same elements

combo.pop()

result = []

backtrack(target, [], 0)

return result

print(combination\_sum([2, 3, 6, 7], 7))

print(combination\_sum([2, 3, 5], 8))

print(combination\_sum([2], 1))

Input: candidates = [2,3,6,7], target = 7

Output: [[2,2,3],[7]]

Input: candidates = [2,3,5], target = 8

Output: [[2,2,2,2],[2,3,3],[3,5]]

Input: candidates = [2], target = 1

Output: []

2) **PERMUTATION**

def permute(nums):

def backtrack(start=0):

if start == len(nums):

result.append(nums[:])

for i in range(start, len(nums)):

nums[start], nums[i] = nums[i], nums[start]

backtrack(start + 1)

nums[start], nums[i] = nums[i], nums[start]

result = []

backtrack()

return result

print(permute([1, 2, 3]))

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

Input: nums = [0,1]

Output: [[0,1],[1,0]]

Input: nums = [1]

Output: [[1]]

3) **Prim’s Algorithm**

import sys

def prims\_algorithm(graph):

num\_vertices = len(graph)

selected\_nodes = [False] \* num\_vertices

selected\_nodes[0] = True

edges = []

for \_ in range(num\_vertices - 1):

minimum = sys.maxsize

x = 0

y = 0

for i in range(num\_vertices):

if selected\_nodes[i]:

for j in range(num\_vertices):

if not selected\_nodes[j] and graph[i][j]:

if minimum > graph[i][j]:

minimum = graph[i][j]

x = i

y = j

edges.append((x, y, minimum))

selected\_nodes[y] = True

return edges

graph = [

[0, 2, 0, 6, 0],

[2, 0, 3, 8, 5],

[0, 3, 0, 0, 7],

[6, 8, 0, 0, 9],

[0, 5, 7, 9, 0]

]

mst\_edges = prims\_algorithm(graph)

print("Edges in the Minimum Spanning Tree:")

for edge in mst\_edges:

print(f"Edge: {edge[0]} - {edge[1]} with weight {edge[2]}")

4) **Kruskal’s Algorithm**

class DisjointSet:

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

self.parent = list(range(n))

self.rank = [0] \* n

def find(self, u):

if self.parent[u] != u:

self.parent[u] = self.find(self.parent[u])

return self.parent[u]

def union(self, u, v):

root\_u = self.find(u)

root\_v = self.find(v)

if root\_u != root\_v:

if self.rank[root\_u] > self.rank[root\_v]:

self.parent[root\_v] = root\_u

elif self.rank[root\_u] < self.rank[root\_v]:

self.parent[root\_u] = root\_v

else:

self.parent[root\_v] = root\_u

self.rank[root\_u] += 1

def kruskal(n, edges):

edges.sort(key=lambda x: x[2]) # Sort edges by weight

ds = DisjointSet(n)

mst = []

total\_cost = 0

for u, v, weight in edges:

if ds.find(u) != ds.find(v):

ds.union(u, v)

mst.append((u, v, weight))

total\_cost += weight

return mst, total\_cost

edges = [(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)]

n = 4

mst, cost = kruskal(n, edges)

print("Minimum Spanning Tree:", mst)

print("Total Cost:", cost)

5) **Boruvkas algorithm**

class Graph:

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

self.V = vertices

self.graph = []

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

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

def find(self, parent, i):

if parent[i] == i:

return i

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

def 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

def boruvka(self):

parent = []

rank = []

cheapest = []

num\_trees = self.V

mst\_weight = 0

for node in range(self.V):

parent.append(node)

rank.append(0)

cheapest = [-1] \* len(self.graph)

while num\_trees > 1:

for i in range(len(self.graph)):

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

set\_u = self.find(parent, u)

set\_v = self.find(parent, v)

if set\_u != set\_v:

if cheapest[set\_u] == -1 or cheapest[set\_u][2] > w:

cheapest[set\_u] = [u, v, w]

if cheapest[set\_v] == -1 or cheapest[set\_v][2] > w:

cheapest[set\_v] = [u, v, w]

for node in range(self.V):

if cheapest[node] != -1:

u, v, w = cheapest[node]

set\_u = self.find(parent, u)

set\_v = self.find(parent, v)

if set\_u != set\_v:

mst\_weight += w

self.union(parent, rank, set\_u, set\_v)

print(f"Edge {u} - {v} with weight {w} is included in the MST")

cheapest = [-1] \* len(self.graph)

num\_trees -= 1

print(f"Total weight of MST is {mst\_weight}")

g = Graph(4)

g.add\_edge(0, 1, 10)

g.add\_edge(0, 2, 6)

g.add\_edge(0, 3, 5)

g.add\_edge(1, 3, 15)

g.add\_edge(2, 3, 4)

g.boruvka()