**Design and Analysis of Algorithm Lab Manual**

1) **Write a python program to implement Merge sort algorithm for sorting a list of integers in ascending order.**

def mergeSort(nlist):

print("Splitting ",nlist)

if len(nlist)>1:

mid = len(nlist)//2

lefthalf = nlist[:mid]

righthalf = nlist[mid:]

mergeSort(lefthalf)

mergeSort(righthalf)

i=j=k=0

while i < len(lefthalf) and j < len(righthalf):

if lefthalf[i] < righthalf[j]:

nlist[k]=lefthalf[i]

i=i+1

else:

nlist[k]=righthalf[j]

j=j+1

k=k+1

while i < len(lefthalf):

nlist[k]=lefthalf[i]

i=i+1

k=k+1

while j < len(righthalf):

nlist[k]=righthalf[j]

j=j+1

k=k+1

print("Merging ",nlist)

nlist = [14,46,43,27,57,41,45,21,70]

mergeSort(nlist)

print(nlist)

2) **Write a python programs to implement backtracking algorithm for the N-queens problem.**

def solve\_nqueens(n):

def is\_safe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, n), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve\_util(board, col):

if col == n:

return True

for i in range(n):

if is\_safe(board, i, col):

board[i][col] = 1

if solve\_util(board, col + 1):

return True

board[i][col] = 0

return False

board = [[0 for \_ in range(n)] for \_ in range(n)]

if solve\_util(board, 0):

for row in board:

print(" ".join("Q" if val == 1 else "." for val in row))

else:

print("No solution exists.")

if \_\_name\_\_ == "\_\_main\_\_":

n = 20

solve\_nqueens(n)

**3) Write a python program to implement the Job scheduling Algorithm**

def printjobschedule(array, t):

m = len(array)

for j in range(m):

for q in range(m - 1 - j):

if array[q][2] < array[q + 1][2]:

array[q], array[q + 1] = array[q + 1], array[q]

res = [False] \* t

# To store result

job = ['-1'] \* t

for q in range(len(array)):

# Find a free slot

for q in range(min(t - 1, array[q][1] - 1), -1, -1):

if res[q] is False:

res[q] = True

job[q] = array[q][0]

break

# print

print(job)

# Driver

array = [['a', 7, 202],

['b', 5, 29],

['c', 6, 84],

['d', 1, 75],

['e', 2, 43]]

print("Maximum profit sequence of jobs is- ")

printjobschedule(array, 3)

**4) Write a pyton program to implement Dijkstra’s algorithm for the Single source shortest path problem.**

class Graph():

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

self.V = vertices

self.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

def printSolution(self, dist):

print("Vertex \t Distance from Source")

for node in range(self.V):

print(node, "\t\t", dist[node])

# A utility function to find the vertex with

# minimum distance value, from the set of vertices

# not yet included in shortest path tree

def minDistance(self, dist, sptSet):

# Initialize minimum distance for next node

min = 1e7

# Search not nearest vertex not in the

# shortest path tree

for v in range(self.V):

if dist[v] < min and sptSet[v] == False:

min = dist[v]

min\_index = v

return min\_index

# Function that implements Dijkstra's single source

# shortest path algorithm for a graph represented

# using adjacency matrix representation

def dijkstra(self, src):

dist = [1e7] \* self.V

dist[src] = 0

sptSet = [False] \* self.V

for cout in range(self.V):

# Pick the minimum distance vertex from

# the set of vertices not yet processed.

# u is always equal to src in first iteration

u = self.minDistance(dist, sptSet)

# Put the minimum distance vertex in the

# shortest path tree

sptSet[u] = True

# Update dist value of the adjacent vertices

# of the picked vertex only if the current

# distance is greater than new distance and

# the vertex in not in the shortest path tree

for v in range(self.V):

if (self.graph[u][v] > 0 and

sptSet[v] == False and

dist[v] > dist[u] + self.graph[u][v]):

dist[v] = dist[u] + self.graph[u][v]

self.printSolution(dist)

# Driver program

g = Graph(9)

g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],

[4, 0, 8, 0, 0, 0, 0, 11, 0],

[0, 8, 0, 7, 0, 4, 0, 0, 2],

[0, 0, 7, 0, 9, 14, 0, 0, 0],

[0, 0, 0, 9, 0, 10, 0, 0, 0],

[0, 0, 4, 14, 10, 0, 2, 0, 0],

[0, 0, 0, 0, 0, 2, 0, 1, 6],

[8, 11, 0, 0, 0, 0, 1, 0, 7],

[0, 0, 2, 0, 0, 0, 6, 7, 0]

]

g.dijkstra(0)

**5) Write a python program that implements Prim’s algorithm to generate minimum cost spanning tree**

**# 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

**6) Write a python program that implements Kruskal’s algorithm to generate minimum cost spanning tree**

# 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()

**7) Write a python program to implement Dynamic Programming algorithm for the 0/1 Knapsack problem**

class KnapsackPackage(object):

""" Knapsack Package Data Class """

def \_\_init\_\_(self, weight, value):

self.weight = weight

self.value = value

self.cost = value / weight

def \_\_lt\_\_(self, other):

return self.cost < other.cost

class FractionalKnapsack(object):

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

pass

def knapsackGreProc(self, W, V, M, n):

packs = []

for i in range(n):

packs.append(KnapsackPackage(W[i], V[i]))

packs.sort(reverse=True)

result = 0

i = 0

while M > 0 and i < n:

if packs[i].weight <= M:

M -= packs[i].weight

result += packs[i].value

else:

fraction = M / packs[i].weight

result += fraction \* packs[i].value

M = 0

i += 1

print("Max Value:\t", result)

if \_\_name\_\_ == "\_\_main\_\_":

W = [15, 10, 2, 4]

V = [30, 25, 2, 6]

M = 37

n = 4

proc = FractionalKnapsack()

proc.knapsackGreProc(W, V, M, n)