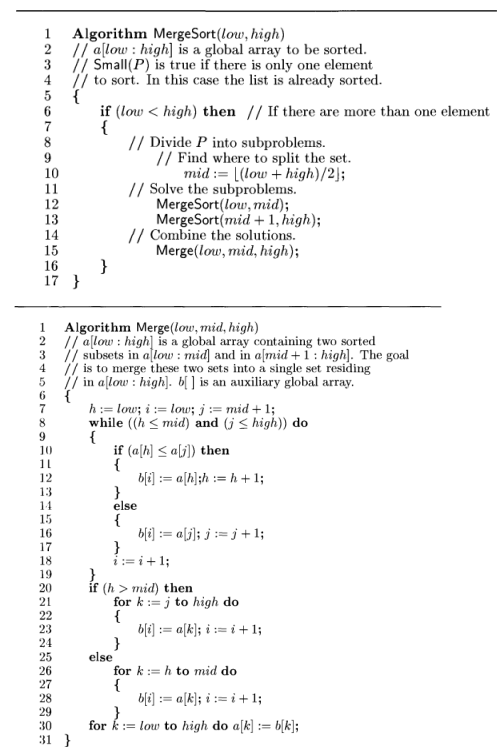
**Merge Sort:**

****

import time

import matplotlib.pyplot as plt

def merge\_sort(arr, low, high):

    if low < high:

        mid = (low + high) // 2

        merge\_sort(arr, low, mid)

        merge\_sort(arr, mid + 1, high)

        merge(arr, low, mid, high)

def merge(arr, low, mid, high):

    b = [0] \* (high + 1)

    h = low

    j = mid + 1

    i = low

    while h <= mid and j <= high:

        if arr[h] <= arr[j]:

            b[i] = arr[h]

            h += 1

        else:

            b[i] = arr[j]

            j += 1

        i += 1

    if h > mid:

        for k in range(j, high + 1):

            b[i] = arr[k]

            i += 1

    else:

        for k in range(h, mid + 1):

            b[i] = arr[k]

            i += 1

    for k in range(low, high + 1):

        arr[k] = b[k]

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

    sizes = []

    times = []

    # Taking multiple input sizes from the user

    num\_sizes = int(input("Enter the number of different input sizes you want to test: "))

    for \_ in range(num\_sizes):

        size = int(input("Enter input size: "))

        sizes.append(size)

    for n in sizes:

        print(f"\nEnter {n} elements for input size {n}:")

        arr = [int(input()) for \_ in range(n)]

        print("Original array:", arr)

        start\_time = time.time()

        merge\_sort(arr, 0, n - 1)

        end\_time = time.time()

        times.append(end\_time - start\_time)

        print("Sorted array:", arr)

        print(f"Time taken for size {n}: {end\_time - start\_time} seconds")

    # Plotting the results

    plt.figure(figsize=(10, 6))

    plt.plot(sizes, times, marker='o', color='b', label="Merge Sort Time Complexity")

    plt.xlabel("Input Size (n)")

    plt.ylabel("Time (seconds)")

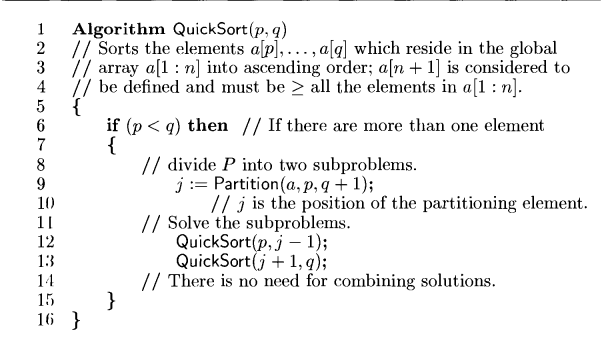
    plt.title("Time Complexity of Merge Sort")

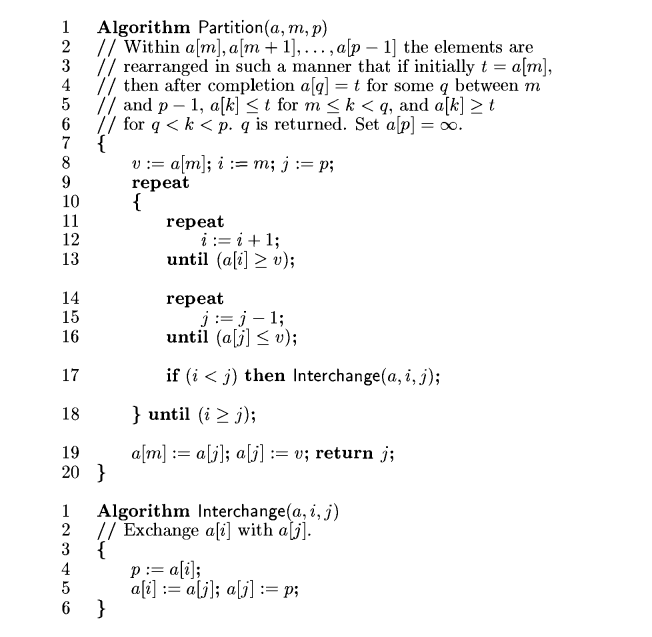
    plt.legend()

    plt.grid(True)

    plt.show()

**Quick Sort:**

****

****

import matplotlib.pyplot as plt

import random

import time

import sys

sys.setrecursionlimit(3000)

def partition(arr, low, high):

pivot = arr[high]

i = low - 1

for j in range(low, high):

if arr[j] < pivot:

i += 1

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

arr[i + 1], arr[high] = arr[high], arr[i + 1]

return i + 1

def quick\_sort(arr, low, high):

if low < high:

pi = partition(arr, low, high)

quick\_sort(arr, low, pi - 1)

quick\_sort(arr, pi + 1, high)

def sort\_and\_print\_example():

example\_arr = list(map(int, input("Enter the array elements separated by spaces: ").split()))

print("Original array:", example\_arr)

quick\_sort(example\_arr, 0, len(example\_arr) - 1)

print("Sorted array:", example\_arr)

def plot\_time\_complexity():

sizes = input("Enter array sizes separated by commas (e.g., 100,200,500): ")

sizes = list(map(int, sizes.split(',')))

best\_times = []

average\_times = []

worst\_times = []

for size in sizes:

arr\_best = random.sample(range(size), size)

start\_time = time.time()

quick\_sort(arr\_best, 0, size - 1)

best\_times.append(time.time() - start\_time)

arr\_avg = random.sample(range(size), size)

start\_time = time.time()

quick\_sort(arr\_avg, 0, size - 1)

average\_times.append(time.time() - start\_time)

arr\_worst = list(range(size))

start\_time = time.time()

quick\_sort(arr\_worst, 0, size - 1)

worst\_times.append(time.time() - start\_time)

plt.figure(figsize=(10, 6))

plt.plot(sizes, best\_times, label="Best Case (O(n log n))", marker="o")

plt.plot(sizes, average\_times, label="Average Case (O(n log n))", marker="o")

plt.plot(sizes, worst\_times, label="Worst Case (O(n^2))", marker="o")

plt.xlabel("Array Size (n)")

plt.ylabel("Execution Time (seconds)")

plt.title("Time Complexity of QuickSort")

plt.legend()

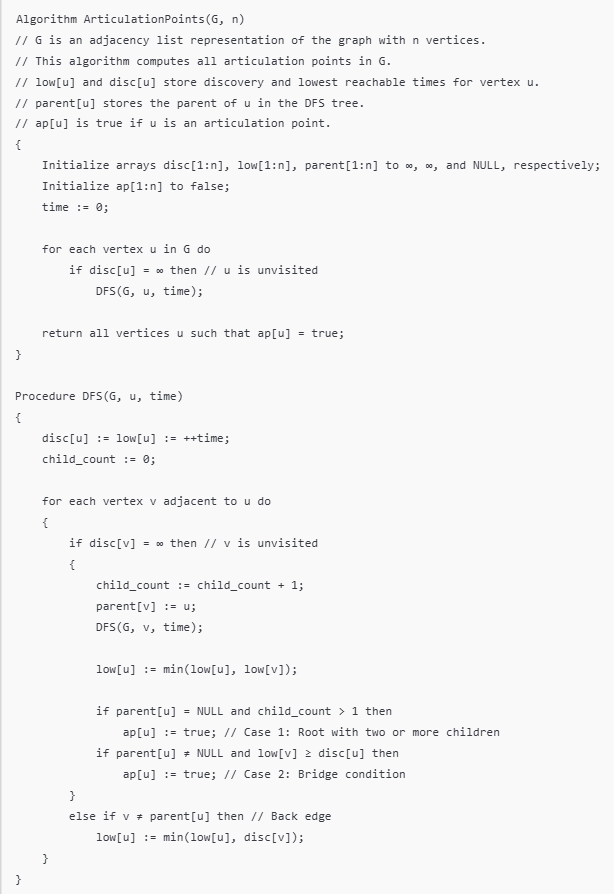
plt.grid(True)

plt.show()

sort\_and\_print\_example()

plot\_time\_complexity()

**2) Articulation points**:



**CODE:**

from collections import defaultdict

class Graph:

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

self.V = vertices

self.graph = defaultdict(list)

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

self.graph[u].append(v)

self.graph[v].append(u)

def \_articulation\_point\_util(self, u, visited, parent, low, disc, ap):

children = 0

visited[u] = True

disc[u] = self.time

low[u] = self.time

self.time += 1

for v in self.graph[u]:

if not visited[v]:

parent[v] = u

children += 1

self.\_articulation\_point\_util(v, visited, parent, low, disc, ap)

low[u] = min(low[u], low[v])

if parent[u] is None and children > 1:

ap[u] = True

if parent[u] is not None and low[v] >= disc[u]:

ap[u] = True

elif v != parent[u]:

low[u] = min(low[u], disc[v])

def find\_articulation\_points(self):

visited = [False] \* self.V

disc = [float("Inf")] \* self.V

low = [float("Inf")] \* self.V

parent = [None] \* self.V

ap = [False] \* self.V

self.time = 0

for i in range(self.V):

if not visited[i]:

self.\_articulation\_point\_util(i, visited, parent, low, disc, ap)

articulation\_points = []

for index, value in enumerate(ap):

if value:

articulation\_points.append(index)

return articulation\_points

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

print("Articulation Points in Graphs")

V = int(input("Enter the number of vertices: "))

E = int(input("Enter the number of edges: "))

g = Graph(V)

print("Enter the edges (u v):")

for \_ in range(E):

u, v = map(int, input().split())

g.add\_edge(u, v)

articulation\_points = g.find\_articulation\_points()

print("Articulation Points in the graph are:")

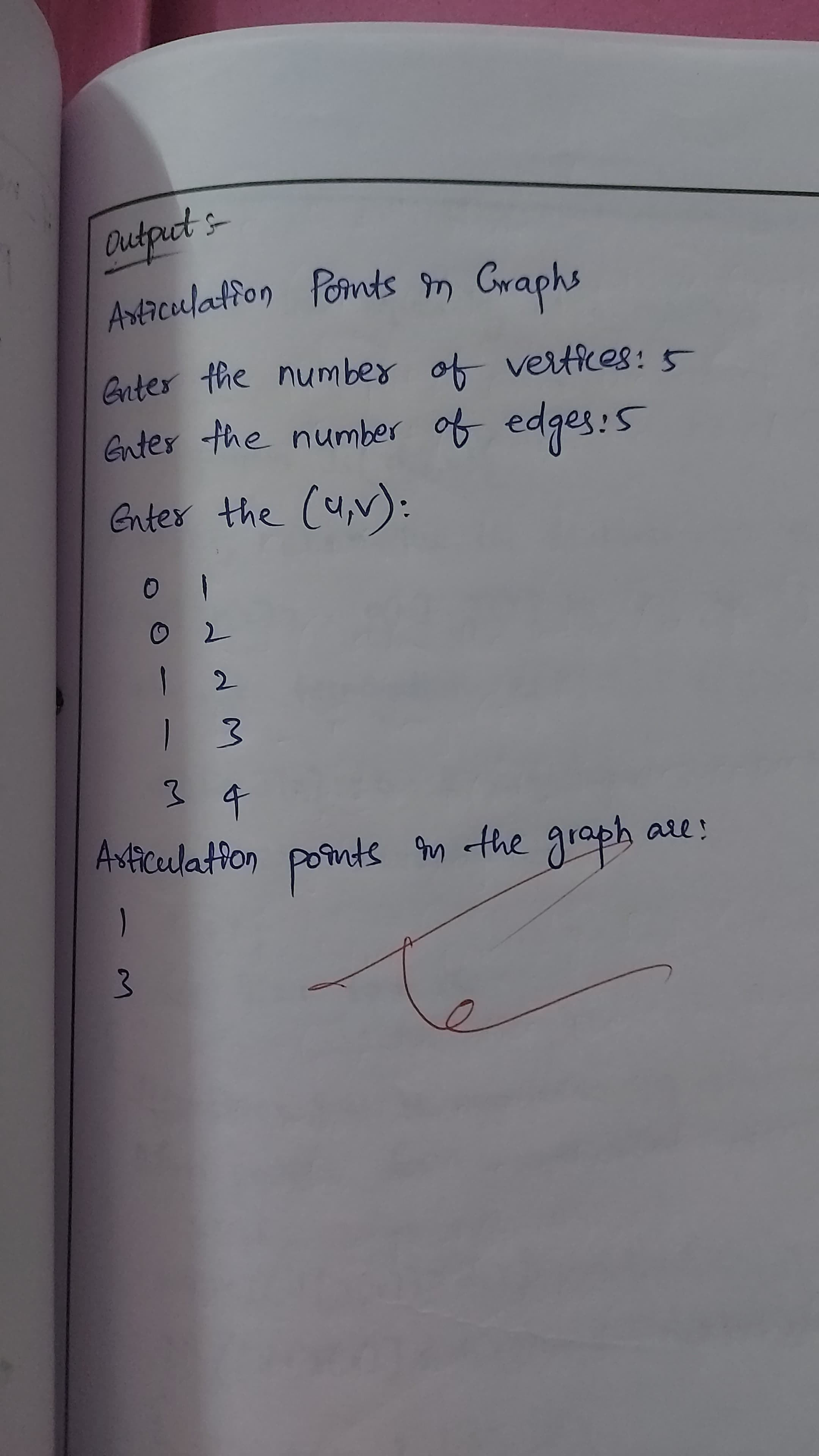
if articulation\_points:

for point in articulation\_points:

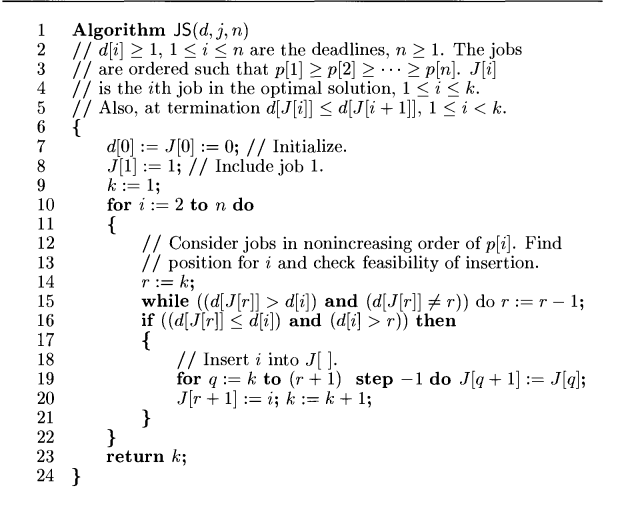
print(point)

else:

print("No articulation points found.")



**3) job sequencing with deadlines:**



**Code:**

def AlgorithmJS(jobs, n):

jobs.sort(key=lambda x: x[1], reverse=True)

J = [0] \* (n + 1)

J[0] = 0

k = 0

total\_profit = 0

deadlines = [False] \* (n + 1)

for i in range(n):

job\_num = jobs[i][0]

job\_profit = jobs[i][1]

job\_deadline = jobs[i][2]

for j in range(min(job\_deadline,n), 0, -1):

if not deadlines[j]:

J[j] = job\_num

deadlines[j] = True

k += 1

total\_profit += job\_profit

break

return k, total\_profit, J[1:k + 1]

def main():

n = int(input("Enter the number of jobs: "))

jobs = []

print("Enter the jobs")

for i in range(n):

job\_info = list(map(int, input(f"Job {i + 1}: ").split()))

jobs.append(job\_info)

num\_jobs, total\_profit, selected\_jobs = AlgorithmJS(jobs, n)

print("Number of jobs", num\_jobs)

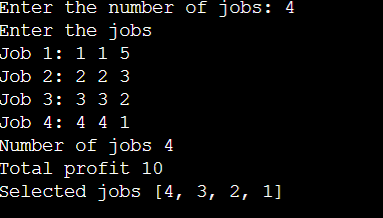
print("Total profit", total\_profit)

print("Selected jobs", selected\_jobs)

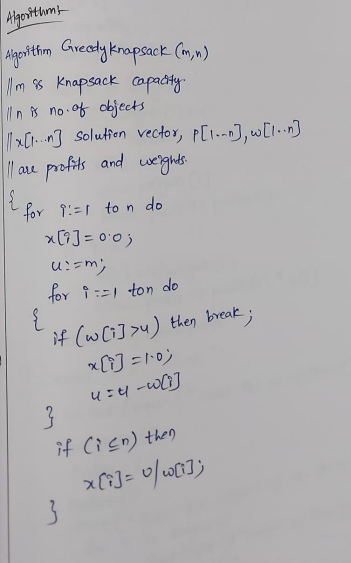
if \_\_name\_\_ == "\_\_main\_\_":

main()

output:



**4) fractional knapsack :**

****

**CODE:**

def greedy\_knapsack(weights, profits, capacity, n):

items = [(profits[i] / weights[i], weights[i], profits[i]) for i in range(n)]

items.sort(key=lambda x: x[0], reverse=True)

x = [0.0] \* n

u = capacity

total\_profit = 0.0

for i in range(n):

if items[i][1] > u:

break

x[i] = 1.0

u -= items[i][1]

total\_profit += items[i][2]

if i < n:

x[i] = u / items[i][1]

total\_profit += items[i][2] \* x[i]

return x, total\_profit

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

n = int(input("Enter the number of items: "))

weights = []

profits = []

print("Enter the weights of the items:")

for i in range(n):

w = float(input(f"Weight of item {i + 1}: "))

weights.append(w)

print("Enter the profits of the items:")

for i in range(n):

p = float(input(f"Profit of item {i + 1}: "))

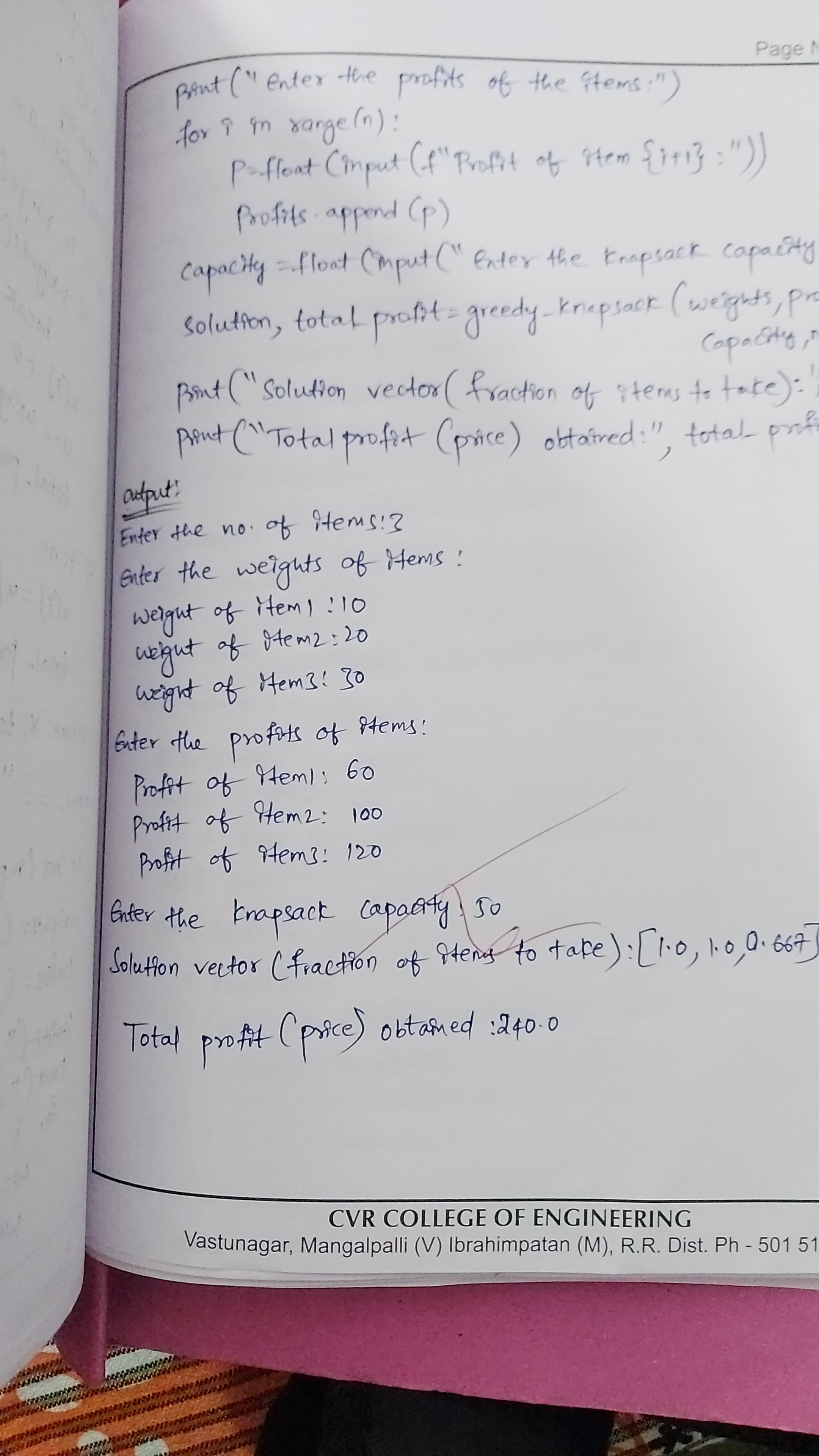
profits.append(p)

capacity = float(input("Enter the knapsack capacity: "))

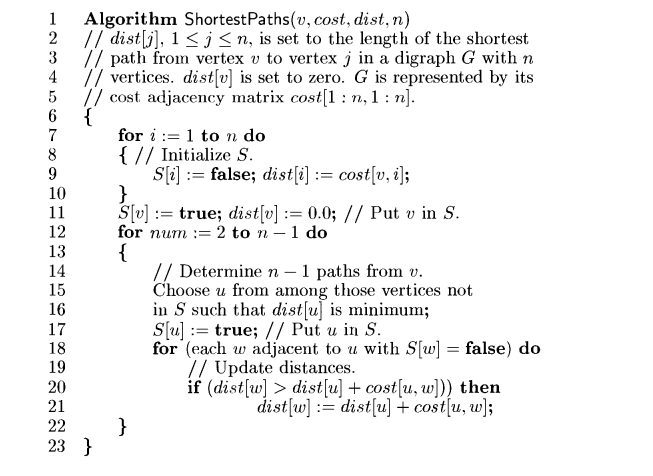
solution, total\_profit = greedy\_knapsack(weights, profits, capacity, n)

print("Solution vector (fraction of items to take):", solution)

print("Total profit obtained:", total\_profit)



**5) djasktras single source shortest path:**



import numpy as np

def shortest\_paths(u, cost, dist, n):

S = [False] \* n

for i in range(n):

S[i] = False

dist[i] = cost[u][i]

S[u] = True

dist[u] = 0.0

for \_ in range(1, n):

min\_dist = float('inf')

for v in range(n):

if not S[v] and dist[v] < min\_dist:

min\_dist = dist[v]

u = v

S[u] = True

for w in range(n):

if cost[u][w] != float('inf') and not S[w]:

if dist[w] > dist[u] + cost[u][w]:

dist[w] = dist[u] + cost[u][w]

def main():

n = int(input("Enter the number of vertices: "))

print("Enter the adjacency matrix (use 'inf' for no edge):")

cost = []

for i in range(n):

row = input(f"Row {i + 1}: ").split()

cost.append([float('inf') if x == 'inf' else float(x) for x in row])

cost = np.array(cost)

start\_vertex = int(input("Enter the starting vertex (1 to {}): ".format(n))) - 1

if start\_vertex < 0 or start\_vertex >= n:

print("Invalid starting vertex.")

return

dist = [float('inf')] \* n

shortest\_paths(start\_vertex, cost, dist, n)

print("Shortest distances from vertex", start\_vertex + 1, ":")

for i in range(n):

if dist[i] == float('inf'):

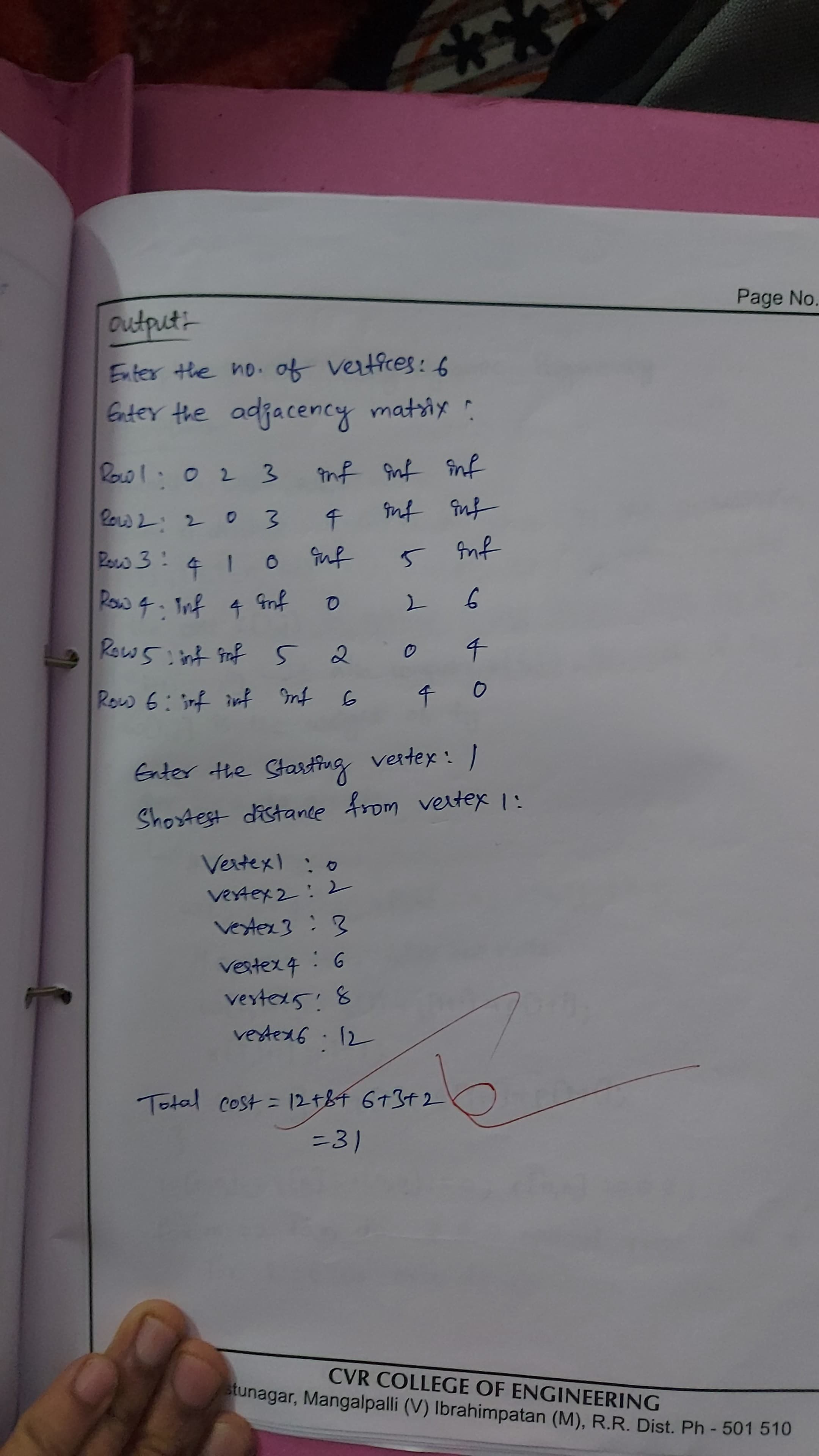
print(f"Vertex {i + 1}: inf")

else:

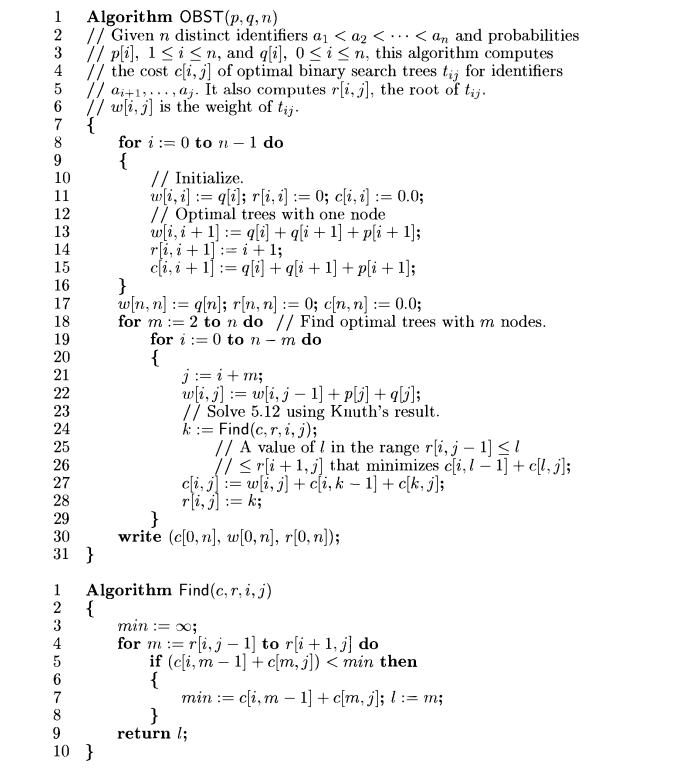
print(f"Vertex {i + 1}: {dist[i]}")

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

main()



**OBST:**



import sys

def find(c, r, i, j):

min\_cost = sys.maxsize

min\_k = -1

for m in range(r[i][j-1], r[i+1][j] + 1):

if c[i][m-1] + c[m][j] < min\_cost:

min\_cost = c[i][m-1] + c[m][j]

min\_k = m

return min\_k

def OBST(p, q, n):

w = [[0 for \_ in range(n+1)] for \_ in range(n+1)]

c = [[0 for \_ in range(n+1)] for \_ in range(n+1)]

r = [[0 for \_ in range(n+1)] for \_ in range(n+1)]

for i in range(n+1):

w[i][i] = q[i]

c[i][i] = 0

r[i][i] = 0

if i < n:

w[i][i+1] = q[i] + q[i+1] + p[i+1]

c[i][i+1] = w[i][i+1]

r[i][i+1] = i + 1

for m in range(2, n+1):

for i in range(n - m + 1):

j = i + m

w[i][j] = w[i][j-1] + p[j] + q[j]

k = find(c, r, i, j)

c[i][j] = w[i][j] + c[i][k-1] + c[k][j]

r[i][j] = k

return c, w, r

def display\_table(table, n, title):

print(f"\n{title}:")

for i in range(n+1):

for j in range(n+1):

print(f"{table[i][j]:>5}", end=" ")

print()

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

n = int(input("Enter the number of keys (n): "))

p = [0] \* (n+1)

q = [0] \* (n+1)

print(f"Enter {n} success probabilities (p[i] for key a[i]):")

for i in range(1, n+1):

p[i] = float(input(f"p[{i}] = "))

print(f"Enter {n+1} failure probabilities (q[i] for gaps between keys):")

for i in range(n+1):

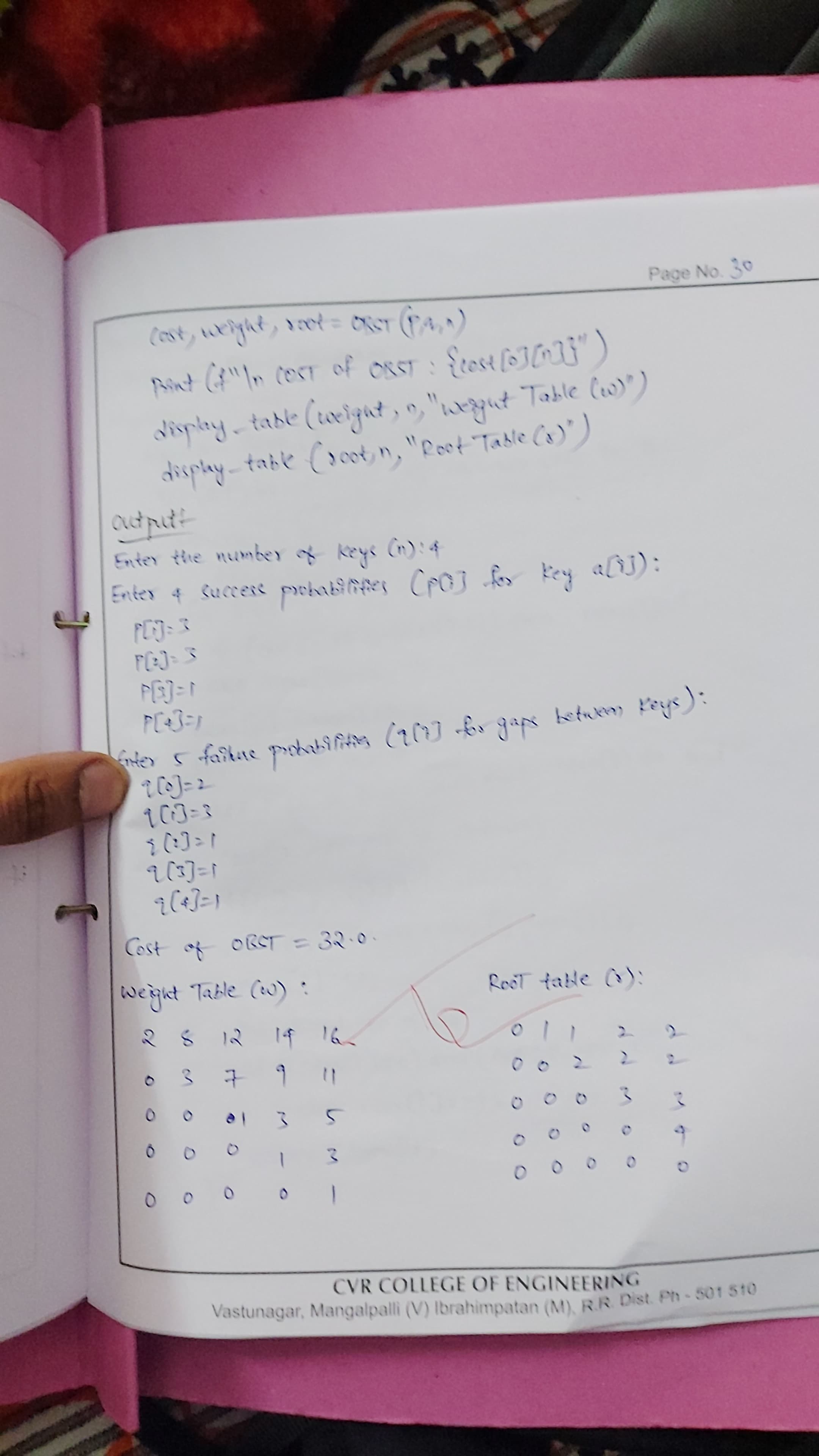
q[i] = float(input(f"q[{i}] = "))

cost, weight, root = OBST(p, q, n)

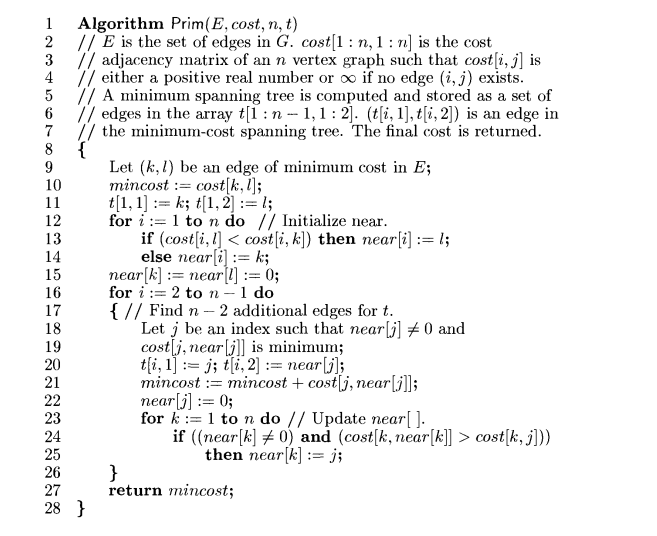
print(f"\nCost of OBST: {cost[0][n]}")

display\_table(weight, n, "Weight Table (w)")

display\_table(root, n, "Root Table (r)")



**PRIMS ALGORITHM:**



**CODE:**

INF = float('inf')

def prims(t, cost, n):

near = [-1] \* n

tcost = 0

minCost = INF

k = l = -1

# Find the initial minimum edge

for i in range(n):

for j in range(n):

if cost[i][j] < minCost and i != j and cost[i][j] != 0:

minCost = cost[i][j]

k = i

l = j

t[0][0] = k

t[0][1] = l

tcost += minCost

# Initialize the near array

for i in range(n):

if cost[i][k] < cost[i][l]:

near[i] = k

else:

near[i] = l

near[k] = near[l] = -1

# Find the remaining edges of the Minimum Cost Spanning Tree (MCST)

for i in range(1, n - 1):

mCost = INF

min = -1

# Find the closest vertex to the MST

for j in range(n):

if near[j] != -1 and cost[j][near[j]] < mCost:

mCost = cost[j][near[j]]

min = j

t[i][0] = min

t[i][1] = near[min]

tcost += mCost

near[min] = -1

for j in range(n):

if near[j] != -1 and cost[j][min] < cost[j][near[j]]:

near[j] = min

return tcost

def main():

n = int(input("Enter the number of vertices: "))

t = [[0, 0] for \_ in range(n - 1)] # Store edges in MCST

cost = []

print("Enter the cost matrix:")

for i in range(n):

row = list(map(int, input().split()))

for j in range(n):

if row[j] == 0 and i != j:

row[j] = INF

cost.append(row)

minCost = prims(t, cost, n)

print("Total cost of MCSPT:", minCost)

print("Edges in the MCSPT:")

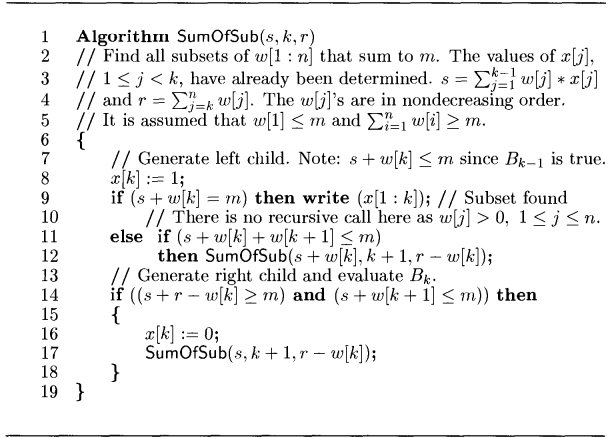
for i in range(n - 1):

print(f"{t[i][0] + 1}-{t[i][1] + 1}")

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

main()

**Sum of Sub sets:**



**CODE:**

def sum\_of\_sub(s, k, r, w, x, m, n):

x[k] = 1

if s + w[k] == m:

print([w[i] for i in range(k + 1) if x[i] == 1])

elif s + w[k] + w[k + 1] <= m:

sum\_of\_sub(s + w[k], k + 1, r - w[k], w, x, m, n)

if s + r - w[k] >= m and s + w[k + 1] <= m:

x[k] = 0

sum\_of\_sub(s, k + 1, r - w[k], w, x, m, n)

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

n = int(input("Enter the number of elements in the set: "))

w = [0] + list(map(int, input("Enter the elements in non-decreasing order: ").split()))

m = int(input("Enter the target sum (m): "))

if sorted(w[1:]) != w[1:]:

print("The input weights must be in non-decreasing order.")

else:

x = [0] \* (n + 1)

r = sum(w[1:])

print("Subsets that sum to", m, "are:")

sum\_of\_sub(0, 1, r, w, x, m, n)