PART - A

1. Write a program to sort a list of N elements using Selection Sort Technique.

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_index = i

        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j

        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr


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

print("Enter the elements:")
elements = [int(input()) for _ in range(n)]

print("Sorted list:", selection_sort(elements))
```

Output:

```
Enter the number of elements in the list: 5
Enter the elements:
42
17
23
31
10
Before Sorting: [42, 17, 23, 31, 10]
After Sorting: [10, 17, 23, 31, 42]
```

2. Write a program to read 'n' numbers, find minimum and maximum value in an array using divide and conquer.

```
def find_min_max(arr, start, end):
    if start == end:
        return arr[start], arr[start]

    if end - start == 1:
        return (arr[start], arr[end]) if arr[start] < arr[end] else (arr[end], arr[start])

    mid = (start + end) // 2

    min1, max1 = find_min_max(arr, start, mid)
    min2, max2 = find_min_max(arr, mid + 1, end)

    return min1 if min1 < min2 else min2, max1 if max1 > max2 else max2

n = int(input("Enter the number of elements: "))
print("Enter the elements: ")
arr = [int(input()) for _ in range(n)]

min_val, max_val = find_min_max(arr, 0, n - 1)

print("min1mum value:", min_val)
print("max1mum value:", max_val)
```

Output:

```
Enter the number of elements: 8
Enter the elements:
45
23
67
12
89
34
56
78
Minimum value: 12
Maximum value: 89
```

3. Sort a given set of n integer elements using the Merge Sort method and compute its time complexity. Run the program for varied values of n > 5000, and record the time taken to sort.

```
import random
import time

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result += left[i:]
    result += right[j:]
    return result

def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

n = int(input('Enter the number of elements (more than 5000): '))
arr = [random.randint(1, 1000000) for _ in range(n)]

print(f'Before Sorting: {arr}')
start_time = time.time()
sorted_array = merge_sort(arr)
end_time = time.time()
print(f'After Sorting: {sorted_array}')

execution_time = end_time - start_time
print(f"Sorting time for n={n}: {execution_time:.6f} seconds")
```

Output:
Enter the number of elements (more than 5000): 10000
Before Sorting: [287455, 95047, 802097, 639128, 124292, 41960, 974115, 655593, 619842, 674871, ...]
After Sorting: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...]
Sorting time for n=10000: 0.028579 seconds

4. Sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort.

```
import random
import time

def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = random.choice(arr)

    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)

n = int(input('Enter the number of elements (more than 5000): '))
arr = [random.randint(1, 1000000) for _ in range(n)]

print(f'Before Sorting: {arr}')
start_time = time.time()
sorted_arr = quick_sort(arr)
end_time = time.time()
print(f'After Sorting: {sorted_arr}')

execution_time = end_time - start_time
print(f"Sorting time for n={n}: {execution_time:.6f} seconds")
```

Output:

Enter the number of elements (more than 5000): 10000
Before Sorting: [789303, 743102, 563925, 738179, 864153, 506862, 539726, 723575, 134097, 811143, ...]
After Sorting: [25, 69, 80, 97, 109, 122, 129, 132, 144, 152, ...]
Sorting time for n=10000: 0.038093 seconds

5. Write a program to sort a list of N elements using Insertion Sort Technique.

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key

n = int(input("Enter the number of elements: "))
print("Enter the elements: ")
arr = [int(input()) for _ in range(n)]

insertion_sort(arr)

print("Sorted array is:", arr)
```

Output:

```
Enter the number of elements: 8
Enter the elements:
34
12
5
78
23
45
67
89
Sorted array is: [5, 12, 23, 34, 45, 67, 78, 89]
```

6. Write a program to implement the BFS algorithm for a graph.

```
def bfs(visited, graph, start_node):
    visited.append(start_node)
    queue.append(start_node)

    while queue:
        current_node = queue.pop(0)
        print(current_node, end=" ")

        for neighbor in graph[current_node]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)

visited = []
graph = {}
queue = []

while True:
    node = input("Enter a node for the graph or q to quit: ")
    if node.lower() == 'q':
      break
    neighbors = input(f"Enter neighbors for node {node} separated by spaces or press Enter for
none: ")
    graph[node] = neighbors.split()

start_node = input("Enter the start node: ")

print("BFS traversal is:")
bfs(visited, graph, start_node)
```

Output:

Enter a node for the graph or q to quit: A
Enter neighbors for node A separated by spaces or press Enter for none: B C
Enter a node for the graph or q to quit: B
Enter neighbors for node B separated by spaces or press Enter for none: D E
Enter a node for the graph or q to quit: C
Enter neighbors for node C separated by spaces or press Enter for none: F
Enter a node for the graph or q to quit: D
Enter neighbors for node D separated by spaces or press Enter for none:
Enter a node for the graph or q to quit: E
Enter neighbors for node E separated by spaces or press Enter for none: F
Enter a node for the graph or q to quit: F
Enter neighbors for node F separated by spaces or press Enter for none:
Enter a node for the graph or q to quit: G
Enter neighbors for node G separated by spaces or press Enter for none:
Enter a node for the graph or q to quit: q
Enter the start node: A
BFS traversal is:
A B C D E F

7. Write a program to implement the DFS algorithm for a graph.

```
def dfs(visited, graph, node):
    print(node, end=" ")
    visited.append(node)

    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(visited, graph, neighbor)

visited = []
graph = {}

while True:
    node = input("Enter a node (type 'q' to quit): ")
    if node == 'q':
        break

    neighbors = input(f"Enter neighbors for node {node} separated by spaces or press Enter for
none: ")

    graph[node] = neighbors.split()

start_node = input("Enter the start node: ")

print("DFS traversal is:")
dfs(visited, graph, start_node)
```

Output:
Enter a node (type 'q' to quit): A
Enter neighbors for node A separated by spaces or press Enter for none: B C
Enter a node (type 'q' to quit): B
Enter neighbors for node B separated by spaces or press Enter for none: D E
Enter a node (type 'q' to quit): C
Enter neighbors for node C separated by spaces or press Enter for none: F
Enter a node (type 'q' to quit): D
Enter neighbors for node D separated by spaces or press Enter for none:
Enter a node (type 'q' to quit): E
Enter neighbors for node E separated by spaces or press Enter for none: F
Enter a node (type 'q' to quit): F
Enter neighbors for node F separated by spaces or press Enter for none:
Enter a node (type 'q' to quit): G
Enter neighbors for node G separated by spaces or press Enter for none:
Enter a node (type 'q' to quit): q
Enter the start node: A
DFS traversal is:
A B D E F C

8. Write a program to implement Strassen's Matrix Multiplication of 2*2 Matrixes.

```
def strassen_matrix_multiply(A, B):
    a11, a12, a21, a22 = A[0][0], A[0][1], A[1][0], A[1][1]
    b11, b12, b21, b22 = B[0][0], B[0][1], B[1][0], B[1][1]

    M1 = (a11 + a22) * (b11 + b22)
    M2 = b11 * (a21 + a22)
    M3 = a11 * (b12 - b22)
    M4 = a22 * (b21 - b11)
    M5 = b22 * (a11 + a12)
    M6 = (a21 - a11) * (b11 + b12)
    M7 = (a12 - a22) * (b21 + b22)

    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    return [[C11, C12], [C21, C22]]

def input_matrix(prompt):
    return [
        [int(input(f"{prompt}[{i}][{j}]: ")) for j in range(2)]
        for i in range(2)
    ]

print("Matrix A:")
A = input_matrix("A")

print("\nMatrix B:")
B = input_matrix("B")

result = strassen_matrix_multiply(A, B)

print("\nResult of Strassen's Matrix Multiplication:")
for row in result:
    print(row)
```

Output:
Matrix A:
A[0][0]: 11
A[0][1]: 12
A[1][0]: 13
A[1][1]: 14

Matrix B:
B[0][0]: 21
B[0][1]: 22
B[1][0]: 23
B[1][1]: 24

Result of Strassen's Matrix Multiplication:
[475, 498]
[553, 580]

PART - B

1. Write a program to implement a backtracking algorithm for solving problems like N queens.

```
def is_safe(board, row, col, N):
    for i in range(row):
        if (
            board[i][col] == 1
            or (col - row + i >= 0 and board[i][col - row + i] == 1)
            or (col + row - i < N and board[i][col + row - i] == 1)
        ):
            return False
    return True

def solve_n_queens_util(board, row, N):
    if row == N:
        return True

    for col in range(N):
        if is_safe(board, row, col, N):
            board[row][col] = 1
            if solve_n_queens_util(board, row + 1, N):
                return True
            board[row][col] = 0

    return False

def solve_n_queens(N):
    board = [[0] * N for _ in range(N)]
    if not solve_n_queens_util(board, 0, N):
        print("No solutions exist.")
        return

    for row in board:
        print(' '.join(['Q' if col == 1 else '-' for col in row]))

N = int(input("Enter the number of queens (N): "))
solve_n_queens(N)
```

Output:
Enter the number of queens (N): 8
Q - - - - - - -
- - - - Q - - -
- - - - - - - Q
- - - - - Q - -
- - Q - - - - -
- - - - - - Q -
- Q - - - - - -
- - - Q - - - -

Enter the number of queens (N): 3
No solutions exist.

2. Design and implement it to find a subset of a given set S = {Sl, S2,.....,Sn} of n positive integers whose SUM is equal to a given positive integer d. For example, if S={1, 2, 5, 6, 8} and d= 9, there are two solutions {1,2,6}and {1,8}. Display a suitable message, if the given problem instance doesn't have a solution.

```
def find_subsets_with_sum(nums, target_sum):
    def recursive(index, current_sum, current_subset):
        if current_sum == target_sum:
            subsets.append(list(current_subset))
            return
        if current_sum > target_sum or index == len(nums):
            return
        current_subset.add(nums[index])
        recursive(index + 1, current_sum + nums[index], current_subset)
        current_subset.remove(nums[index])
        recursive(index + 1, current_sum, current_subset)

    subsets = []
    recursive(0, 0, set())
    return subsets

elements = set(map(int, input("Enter the elements (separated by spaces):").split()))
target_sum = int(input("Enter the target sum:"))
result = find_subsets_with_sum(list(elements), target_sum)

if result:
    print("Subsets with sum", target_sum, "found:")
    for subset in result:
        print(set(subset))
else:
    print("No subset found with sum", target_sum)
```

Output:

```
Enter the elements (separated by spaces):1 2 3 4 5
Enter the target sum:8
Subsets with sum 8 found:
{1, 2, 5}
{1, 3, 4}
{3, 5}

Enter the elements (separated by spaces):2 1 5 7
Enter the target sum:4
No subset found with sum 4
```

3. Write a program to find shortest paths to other vertices using Dijkstra's algorithm.

```
def dijkstra(graph, start):
    num_vertices = len(graph)
    distances = [float('inf')] * num_vertices
    penultimate_vertices = [None] * num_vertices
    distances[start] = 0
    visited = [False] * num_vertices

    for _ in range(num_vertices):
        min_distance = float('inf')
        min_vertex = -1

        for v in range(num_vertices):
            if not visited[v] and distances[v] < min_distance:
                min_distance = distances[v]
                min_vertex = v

        if min_vertex == -1:
            break

        visited[min_vertex] = True

        for v in range(num_vertices):
            if not visited[v] and graph[min_vertex][v] > 0:
                new_distance = distances[min_vertex] + graph[min_vertex][v]
                if new_distance < distances[v]:
                    distances[v] = new_distance
                    penultimate_vertices[v] = min_vertex

    return distances, penultimate_vertices

def get_path(penultimate_vertices, destination):
    path = [destination]
    while penultimate_vertices[destination] is not None:
        destination = penultimate_vertices[destination]
        path.insert(0, destination)
    return path

no_vertices = int(input("Enter the number of vertices: "))
graph = []

print("Enter the weight adjacency matrix:")
for i in range(no_vertices):
```

```
    row = list(map(int, input(f"Enter the weights of the vertices for vertex {i}: ").split()))
    graph.append(row)

start_vertex = int(input(f"Enter the start vertex (0 to {no_vertices - 1}): "))
distances, penultimate_vertices = dijkstra(graph, start_vertex)

for vertex, distance in enumerate(distances):
    if vertex != start_vertex:
        path_to_vertex = get_path(penultimate_vertices, vertex)
        print(f"Shortest distance from {start_vertex} to {vertex} is {distance}, path:
{path_to_vertex}")
```

Output:

```
Enter the number of vertices: 3
Enter the weight adjacency matrix:
Enter the weights of the vertices for vertex 0: 0 2 4
Enter the weights of the vertices for vertex 1: 2 0 1
Enter the weights of the vertices for vertex 2: 4 1 0
Enter the start vertex (0 to 2): 0
Shortest distance from 0 to 1 is 2, path: [0, 1]
Shortest distance from 0 to 2 is 3, path: [0, 2]
```

4. Write a program to perform Knapsack Problem using Greedy Solution.

```
def knapsack(weights, values, W):
    n = len(weights)
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, W + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][W], W

n = int(input("Enter the number of items: "))
weights = []
values = []

for i in range(n):
    weight, value = map(int, input(f"Enter weights and values for item {i + 1} (separated by
space): ").split())
    weights.append(weight)
    values.append(value)

W = int(input("Enter the maximum weight capacity of the knapsack: "))
max_value, w = knapsack(weights, values, W)

print(f"The maximum value and total weight that can be obtained are: {max_value}, {w}")
```

Output:

```
Enter the number of items: 3
Enter weights and values for item 1 (separated by space): 2 5
Enter weights and values for item 2 (separated by space): 3 8
Enter weights and values for item 3 (separated by space): 4 9
Enter the maximum weight capacity of the knapsack: 10

The maximum value and total weight that can be obtained are: 22, 10
```

5. Write a program to implement a greedy algorithm for job sequencing with deadlines.

```
def job_sequencing_with_deadline(jobs):
    jobs.sort(key=lambda x: x[2], reverse=True)
    max_deadline = max(jobs, key=lambda x: x[1])[1]
    slots = [-1] * (max_deadline + 1)
    total_profit = 0

    for job_id, deadline, profit in jobs:
        while deadline > 0 and slots[deadline] != -1:
            deadline -= 1
        if deadline > 0:
            slots[deadline] = job_id
            total_profit += profit

    return total_profit, [job for job in slots if job != -1]

num_jobs = int(input("Enter the number of jobs: "))
jobs = []
for i in range(num_jobs):
    job_id = i+1
    deadline = int(input(f"Enter deadline for jobs{i+1}: "))
    profit = int(input(f"Enter profit for jobs{i+1}: "))
    jobs.append((job_id, deadline, profit))

profit, sequence = job_sequencing_with_deadline(jobs)

print("\nResult")
print("Sequence of jobs: ", sequence)
print("Total profit: ", profit)
```

Output:

```
Enter the number of jobs: 2
Enter deadline for jobs1: 1
Enter profit for jobs1: 50
Enter deadline for jobs2: 2
Enter profit for jobs2: 30

Result
Sequence of jobs:  [1, 2]
Total profit:  80
```

6. Write a program to perform Travelling Salesman Problem.

```
import sys

def nearest_neighbor(graph):
    num_vertices = len(graph)
    visited = [False] * num_vertices
    current_vertex = 0
    visited[current_vertex] = True
    tour = [current_vertex + 1]
    total_distance = 0

    for _ in range(num_vertices - 1):
        nearest_vertex = None
        min_distance = sys.maxsize

        for vertex in range(num_vertices):
            if not visited[vertex] and graph[current_vertex][vertex] < min_distance:
                nearest_vertex = vertex
                min_distance = graph[current_vertex][vertex]

        tour.append(nearest_vertex + 1)
        total_distance += min_distance
        visited[nearest_vertex] = True
        current_vertex = nearest_vertex

    tour.append(tour[0])
    total_distance += graph[current_vertex][tour[0] - 1]
    return tour, total_distance

n = int(input("Enter the number of nodes: "))
graph = [[0] * n for _ in range(n)]

for i in range(n):
    for j in range(n):
        graph[i][j] = int(input(f"Enter the distance from node {i + 1} to node {j + 1}: "))

tour, total_distance = nearest_neighbor(graph)
print(f"Optimal Tour: {tour}")
print(f"Total Distance: {total_distance}")
```

Output:
Enter the number of nodes: 3
Enter the distance from node 1 to node 1: 3
Enter the distance from node 1 to node 2: 4
Enter the distance from node 1 to node 3: 1
Enter the distance from node 2 to node 1: 5
Enter the distance from node 2 to node 2: 2
Enter the distance from node 2 to node 3: 7
Enter the distance from node 3 to node 1: 4
Enter the distance from node 3 to node 2: 7
Enter the distance from node 3 to node 3: 8
Optimal Tour: [1, 3, 2, 1]
Total Distance: 13

7. Write a program that implements Prim’s algorithm to generate minimum cost spanning Tree.

```
def find_min_edge(matrix, Vt):
    min_edge = None
    V = len(matrix)

    for u in Vt:
        for v in range(V):
            if v not in Vt and matrix[u][v] > 0:
                min_edge = (u, v, matrix[u][v])

    return min_edge

def prim(matrix):
    V = len(matrix)
    start_vertex = 0
    Vt = {start_vertex}
    Et = []
    total_weight = 0

    while len(Vt) < V:
        min_edge = find_min_edge(matrix, Vt)
        if min_edge is None:
            break
        u, v, weight = min_edge
        Vt.add(v)
        Et.append((u, v, weight))
        total_weight += weight

    return Et, total_weight

def print_minimum_spanning_tree(minimum_spanning_tree, total_weight):
    print("\nMinimum Spanning Tree:")
    for u, v, weight in minimum_spanning_tree:
        print(f"Edge: {u}-{v}, Weight: {weight}")

    print(f"Total Weight of MST: {total_weight}")


num_vertices = int(input("Enter the number of vertices: "))
graph = []
print("Enter the weight adjacency Matrix:")
for _ in range(num_vertices):
```

```
    row = list(map(int, input(f"Enter weight of the vertex {_ + 1}: ").split()))
    graph.append(row)
minimum_spanning_tree, total_weight = prim(graph)
print_minimum_spanning_tree(minimum_spanning_tree, total_weight)
```

Output:

Enter the number of vertices: 5
Enter the weight adjacency Matrix:
Enter weight of the vertex 1: 0 2 0 4 0
Enter weight of the vertex 2: 2 0 3 3 5
Enter weight of the vertex 3: 0 3 0 0 7
Enter weight of the vertex 4: 6 0 0 0 9
Enter weight of the vertex 5: 0 5 7 9 0

Minimum Spanning Tree:
Edge: 0-1, Weight: 2
Edge: 1-2, Weight: 3
Edge: 1-3, Weight: 3
Edge: 1-4, Weight: 5
Total Weight of MST: 13

8. Write a program that implements Kruskal's algorithm to generate minimum cost spanning tree.

```
def find_parent(parent, i):
    return i if parent[i] == i else find_parent(parent, parent[i])

def kruskal(graph):
    num_vertices = len(graph)
    parent = list(range(num_vertices))
    edges = []

    for u in range(num_vertices):
        for v, w in enumerate(graph[u]):
            if w > 0:
                edges.append((u, v, w))

    edges.sort(key=lambda x: x[2])
    mst = []

    for u, v, w in edges:
        parent_u = find_parent(parent, u)
        parent_v = find_parent(parent, v)

        if parent_u != parent_v:
            mst.append((u, v, w))
            parent[parent_u] = parent_v

    return mst

def print_mst(mst):
    total_weight = sum(w for _, _, w in mst)
    print("\nMinimum spanning tree using Kruskal's algorithm:")
    print("Edges Weight")

    for u, v, w in mst:
        print(f"{u}-{v} {w}")
    print(f"Total weight: {total_weight}")

V = int(input("Enter the number of vertices: "))
print ("enter the weighted adjancy matrix (enter 0 for no edges): ")
graph = [list(map(int, input().split())) for _ in range(V)]
mst = kruskal(graph)
print_mst(mst)
```

Output:
Enter the number of vertices: 4
Enter the weighted adjacency matrix (enter 0 for no edges):
0 2 0 4
2 0 3 3
0 3 0 0
6 0 0 0

Minimum spanning tree using Kruskal's algorithm:
Edges Weight
0-1 2
1-2 3
1-3 3
Total weight: 8