Lab Assignment

Name – Jignesh Ameta
Roll No. – 14
Class – MCA 2nd Sem
Subject – Design and Analysis of Algorithms (DAA)Lab

Date of Submission -

Index

Q1 – WAP in Python for implementing Binary search & linear search (menu driven) using iterative and recursive approach ?

Ans :-

Algorithm :-

Step 1: Start
Step 2: Input the number of elements n
Step 3: Input n elements into array arr[]
Step 4: Repeat the following until user exits:

```
   Display menu:
      Linear Search
      Binary Search
      Exit
   Read user choice choice
```

If choice == 1 (Linear Search):

Step 5:

```
   Display method menu:
      Iterative
      Recursive
   Read user method method
   Input the element key to search
```

Step 6:

```
   If method == 1:
      Call linear_search_iterative(arr[], key)
      Function:
         FUNCTION linear_search_iterative(arr[], key):
            FOR i = 0 TO n - 1 DO:
               IF arr[i] == key:
                  RETURN i
            RETURN -1
   Else if method == 2:
      Call linear_search_recursive(arr[], key, index = 0)
      Function:
         FUNCTION linear_search_recursive(arr[], key, index):
            IF index >= n:
               RETURN -1
            IF arr[index] == key:
               RETURN index
            RETURN linear_search_recursive(arr, key, index + 1)
```

Step 7:

```
   If result ≠ -1 → print index where found
   Else → print “Element not found”
```

If choice == 2 (Binary Search):

Step 8:

Sort the array arr[]

Display method menu:

Iterative

Recursive

Read user method method

Input the element key to search

Step 9:

```
If method == 1:
    Call binary_search_iterative(arr[], key)
    Function :
        FUNCTION binary_search_iterative(arr[], key):
            low = 0
            high = n - 1
            WHILE low ≤ high:
                mid = (low + high) // 2
                IF arr[mid] == key:
                    RETURN mid
                ELSE IF key < arr[mid]:
                    high = mid - 1
                ELSE:
                    low = mid + 1
            RETURN -1
Else if method == 2:
    Call binary_search_recursive(arr[], key, low=0, high=n-1)
    Function:
        FUNCTION binary_search_recursive(arr[], key, low, high):
            IF low > high:
                RETURN -1
            mid = (low + high) // 2
            IF arr[mid] == key:
                RETURN mid
            ELSE IF key < arr[mid]:
                RETURN binary_search_recursive(arr, key, low, mid - 1)
            ELSE:
                RETURN binary_search_recursive(arr, key, mid + 1, high)
```

Step 10:

If result ≠ -1 → print index where found

Else → print “Element not found”

If choice == 3:

Step 11:

Exit the program

Step 12: End

Implementation :-

```
# Iterative Linear Search
def linear_search_iterative(arr, key):
    for i in range(len(arr)):
        if arr[i] == key:
            return i
    return -1

# Recursive Linear Search
def linear_search_recursive(arr, key, index=0):
    if index >= len(arr):
        return -1
    if arr[index] == key:
        return index
    return linear_search_recursive(arr, key, index + 1)

# Iterative Binary Search
def binary_search_iterative(arr, key):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1

    return -1

# Recursive Binary Search
def binary_search_recursive(arr, key, low, high):
    if low > high:
        return -1

    mid = (low + high) // 2

    if arr[mid] == key:
        return mid
    elif arr[mid] < key:
        return binary_search_recursive(arr, key, mid + 1, high)
    else:
        return binary_search_recursive(arr, key, low, mid - 1)

# Main Menu-Driven Program
def main():
    arr = []
    print("Enter the number of elements:")
    n = int(input())
```

```
    print(f"Enter {n} elements:")
    for _ in range(n):
        arr.append(int(input()))

    while True:
        print("\nMENU:")
        print("1. Linear Search")
        print("2. Binary Search")
        print("3. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1 or choice == 2:
            print("Choose Method:")
            print("1. Iterative")
            print("2. Recursive")
            method = int(input("Enter method: "))

            key = int(input("Enter the element to search: "))

            if choice == 1:
                if method == 1:
                    index = linear_search_iterative(arr, key)
                else:
                    index = linear_search_recursive(arr, key)

            elif choice == 2:
                arr.sort()
                print(f"Sorted Array: {arr}")
                if method == 1:
                    index = binary_search_iterative(arr, key)
                else:
                    index = binary_search_recursive(arr, key, 0, len(arr) - 1)

            if index != -1:
                print(f"Element found at index {index}")
            else:
                print("Element not found.")

        elif choice == 3:
            print("Exiting program.")
            break
        else:
            print("Invalid choice. Try again.")

# Run the program
main()
```

Output :-

```
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python "/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab/01_BinaryAndLinearSearchRec.py"
Enter the number of elements:
5
Enter 5 elements:
23
4
3
2
5

MENU:
1. Linear Search
2. Binary Search
3. Exit
Enter your choice: 1
Choose Method:
1. Iterative
2. Recursive
Enter method: 1
Enter the element to search: 3
Element found at index 2

MENU:
1. Linear Search
2. Binary Search
3. Exit
Enter your choice: 2
Choose Method:
1. Iterative
2. Recursive
Enter method: 2
Enter the element to search: 3
Sorted Array: [2, 3, 4, 5, 23]
Element found at index 1

MENU:
1. Linear Search
2. Binary Search
3. Exit
Enter your choice: 
```

Q2 – WAP in Python for implementing Quick Sort with Iterative and Recursive Approach ?

Ans :-

Algorithm :-

Step 1: Start

Step 2: Input number of elements n

Step 3: Input n elements into array arr[]

Step 4: Choose Sorting Method
If the user selects Recursive Quick Sort, go to Step 5
If the user selects Iterative Quick Sort, go to Step 6

Step 5: Recursive Quick Sort Algorithm
Call function: quick_sort_recursive(arr[], low = 0, high = n - 1)
Recursive Quick Sort Logic:
If low < high, continue
Partition the array:
Choose a pivot (typically the last element).
Rearrange the array:
Elements ≤ pivot → left
Elements > pivot → right
Return pivot index p
Recurse on subarrays:
Call quick_sort_recursive(arr[], low, p - 1)
Call quick_sort_recursive(arr[], p + 1, high)

Step 6: Iterative Quick Sort Algorithm
Call function: quick_sort_iterative(arr[], low = 0, high = n - 1)
Iterative Quick Sort Logic:
Initialize a stack to simulate recursion.
Push initial low and high indices to stack.
Repeat while stack is not empty:
Pop high and low values
Partition the subarray using same pivot logic.
If left subarray exists (p - 1 > low):
Push low and p - 1 to stack
If right subarray exists (p + 1 < high):
Push p + 1 and high to stack

Step 7: Print the sorted array

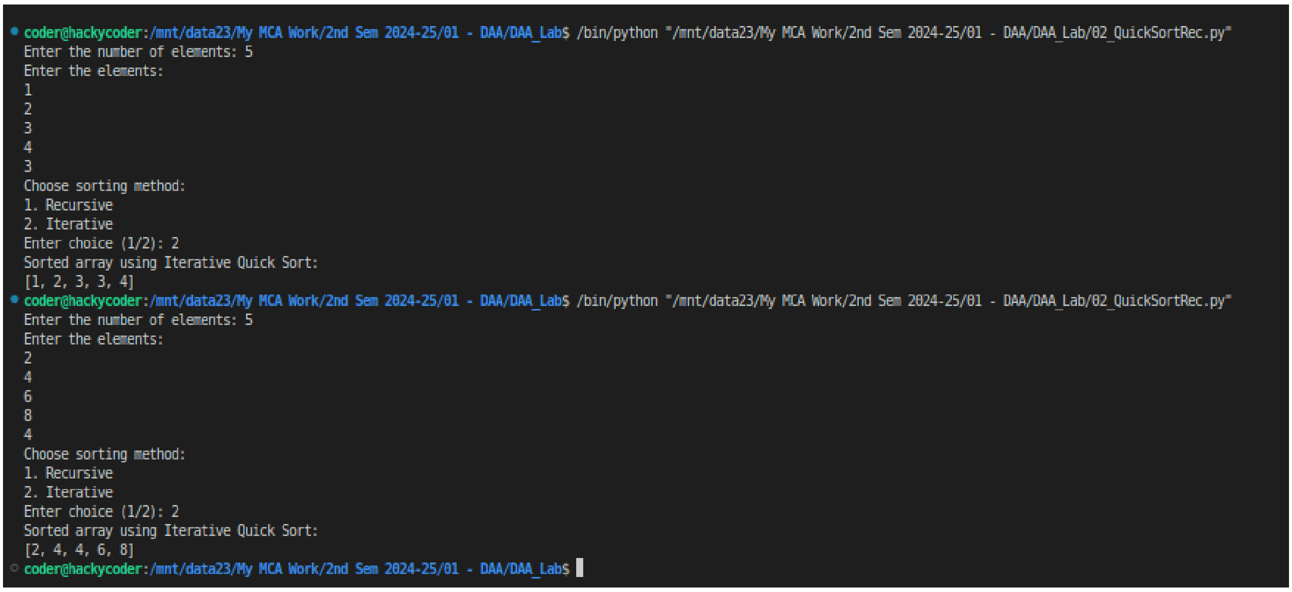
Step 8: End

Implementation :-

```
def quick_sort_recursive(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort_recursive(arr, low, pi - 1)
        quick_sort_recursive(arr, pi + 1, high)

def quick_sort_iterative(arr):
    size = len(arr)
    stack = [(0, size - 1)]

    while stack:
        low, high = stack.pop()
        if low < high:
            pi = partition(arr, low, high)
            stack.append((low, pi - 1))
            stack.append((pi + 1, high))

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

# ------------------- Input & Execution -------------------

n = int(input("Enter the number of elements: "))
arr = []

print("Enter the elements:")
for _ in range(n):
    arr.append(int(input()))

method = int(input("Choose sorting method:\n1. Recursive\n2. Iterative\nEnter choice (1/2): "))

if method == 1:
    quick_sort_recursive(arr, 0, n - 1)
    print("Sorted array using Recursive Quick Sort:")
    print(arr)

elif method == 2:
    quick_sort_iterative(arr)
    print("Sorted array using Iterative Quick Sort:")
    print(arr)

else:
```

print("Invalid choice.")

Output :-



Q3 – WAP in Python for implementing Merge Sort with Iterative and Recursive Approach ?

Ans :-

Algorithm :-

Merge Sort (Recursive Approach) — Algorithm Steps:

Algorithm: merge_sort(arr)

```
Start
    If the length of arr is less than or equal to 1:
        Return arr (it is already sorted)
    Calculate the middle index of the array:
        mid = len(arr) // 2
    Recursively divide and sort the left half:
        left = merge_sort(arr[:mid])
    Recursively divide and sort the right half:
        right = merge_sort(arr[mid:])
    Merge the two sorted halves into a single sorted array:
        result = merge(left, right)
    Return result
End
```

Algorithm: merge(left, right)

```
Start
    Initialize an empty list result
    Initialize i = 0, j = 0 (pointers for left and right)
    While i < length of left and j < length of right:
        If left[i] <= right[j]:
            Append left[i] to result
            Increment i
        Else:
            Append right[j] to result
            Increment j
    Append the remaining elements from left[i:] to result
    Append the remaining elements from right[j:] to result
    Return result
End
```

Merge Sort (Iterative Approach) — Algorithm Steps:

Algorithm: merge_sort_iterative(arr)

```
Start
    Let width = 1 (initial size of subarrays to merge)
    Let n = length of arr
    While width < n:
        Set i = 0
        While i < n:
            Set left = i
            Set mid = min(i + width, n)
            Set right = min(i + 2*width, n)
            Merge the subarrays arr[left:mid] and arr[mid:right]
            Replace arr[left:right] with the merged result
```

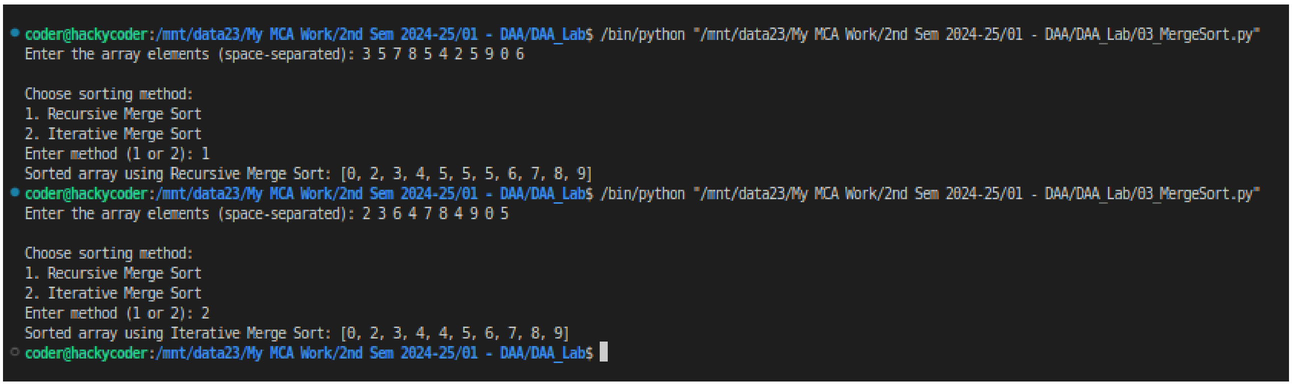
```
        Increment i by 2 * width
      Double the width: width = 2 * width
    Return arr
  End
```

Implementation :-

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

  # Compare elements and merge
  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

  # Append remaining elements
  result.extend(left[i:])
  result.extend(right[j:])
  return result

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

  mid = len(arr) // 2
  left = merge_sort_recursive(arr[:mid])
  right = merge_sort_recursive(arr[mid:])
  return merge(left, right)

def merge_sort_iterative(arr):
  width = 1
  n = len(arr)

  while width < n:
    for i in range(0, n, 2 * width):
      left = i
      mid = min(i + width, n)
      right = min(i + 2 * width, n)

      # Merge arr[left:mid] and arr[mid:right]
      left_part = arr[left:mid]
      right_part = arr[mid:right]
      merged = merge(left_part, right_part)

      # Replace in original array
      arr[left:right] = merged

    width *= 2
```

```
    return arr

# ----------- Main Program ------------
arr = list(map(int, input("Enter the array elements (space-separated): ").split()))

print("\nChoose sorting method:")
print("1. Recursive Merge Sort")
print("2. Iterative Merge Sort")
method = int(input("Enter method (1 or 2): "))

if method == 1:
    result = merge_sort_recursive(arr)
    print("Sorted array using Recursive Merge Sort:", result)
elif method == 2:
    result = merge_sort_iterative(arr)
    print("Sorted array using Iterative Merge Sort:", result)
else:
    print("Invalid method selected.")
```

Output :-



```
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python "/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab/03_MergeSort.py"
Enter the array elements (space-separated): 3 5 7 8 5 4 2 5 9 0 6

Choose sorting method:
1. Recursive Merge Sort
2. Iterative Merge Sort
Enter method (1 or 2): 1
Sorted array using Recursive Merge Sort: [0, 2, 3, 4, 5, 5, 5, 6, 7, 8, 9]
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python "/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab/03_MergeSort.py"
Enter the array elements (space-separated): 2 3 6 4 7 8 4 9 0 5

Choose sorting method:
1. Recursive Merge Sort
2. Iterative Merge Sort
Enter method (1 or 2): 2
Sorted array using Iterative Merge Sort: [0, 2, 3, 4, 4, 5, 6, 7, 8, 9]
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$
```

Q4 – WAP in Python for implementing Strassen Matrix Multiplication with Divide and Conqur Approach ?
Ans :-
Algorithm :-

Step 1: Input and Validation

• Begin with two input matrices **A** and **B,** both of size **n×n**.

• Check to ensure both matrices are square and of the same size.

• If **n** is not a power of two (e.g., 2, 4, 8, 16, ...), increase the matrix size by adding rows and columns filled with zeros until it becomes a power of two.

Step 2: Base Case

If the matrix size is 1 (i.e., each matrix has only one element), simply multiply these two elements and return the result.

Step 3: Divide the Matrices

Split each **n×n** matrix into four smaller submatrices, each of size **2n×2n**

A11,A12,A21,A22 from matrix **A**

B11,B12,B21,B22 from matrix **B**

Step 4: Recursively Compute Seven Products

Instead of the usual eight multiplications, compute only seven using the following formulas:

1. **M1=(A11+A22)×(B11+B22)**
2. **M2=(A21+A22)×B11**
3. **M3=A11×(B12−B22)**
4. **M4=A22×(B21−B11)**
5. **M5=(A11+A12)×B22**
6. **M6=(A21−A11)×(B11+B12)**
7. **M7=(A12−A22)×(B21+B22)**

Each of these multiplications is performed by recursively applying the same Strassen multiplication algorithm.

Step 5: Combine Products to Form Result Submatrices

Using the seven products, compute the four submatrices of the resulting matrix **C**:

1. **C11=M1+M4−M5+M7**
2. **C12=M3+M5**
3. **C21=M2+M4**
4. **C22=M1−M2+M3+M6**

Step 6: Reconstruct the Final Matrix

Join these four submatrices **C11,C12,C21,C22** to form the complete product matrix **C.**

Step 7: Final Steps

If the matrix was padded, remove the extra rows and columns added in Step 1 to restore the result matrix to the original size **n×n**.

Output the final product matrix **C**.

Implementation :-

```
# Matrix addition
def add_matrix(A, B):
    n = len(A)
    return [[A[i][j] + B[i][j] for j in range(n)] for i in range(n)]

# Matrix subtraction
def sub_matrix(A, B):
    n = len(A)
    return [[A[i][j] - B[i][j] for j in range(n)] for i in range(n)]

# Split a matrix into 4 equal submatrices
def split_matrix(M):
    n = len(M)
    mid = n // 2
    A11 = [row[:mid] for row in M[:mid]]
    A12 = [row[mid:] for row in M[:mid]]
    A21 = [row[:mid] for row in M[mid:]]
    A22 = [row[mid:] for row in M[mid:]]
    return A11, A12, A21, A22

# Combine 4 submatrices into one
def join_quadrants(C11, C12, C21, C22):
    top = [c11 + c12 for c11, c12 in zip(C11, C12)]
    bottom = [c21 + c22 for c21, c22 in zip(C21, C22)]
    return top + bottom

# Strassen's recursive multiplication
def strassen(A, B):
    n = len(A)

    if n == 1:
        return [[A[0][0] * B[0][0]]]

    A11, A12, A21, A22 = split_matrix(A)
    B11, B12, B21, B22 = split_matrix(B)

    M1 = strassen(add_matrix(A11, A22), add_matrix(B11, B22))
    M2 = strassen(add_matrix(A21, A22), B11)
    M3 = strassen(A11, sub_matrix(B12, B22))
    M4 = strassen(A22, sub_matrix(B21, B11))
```

```
    M5 = strassen(add_matrix(A11, A12), B22)
    M6 = strassen(sub_matrix(A21, A11), add_matrix(B11, B12))
    M7 = strassen(sub_matrix(A12, A22), add_matrix(B21, B22))

    C11 = add_matrix(sub_matrix(add_matrix(M1, M4), M5), M7)
    C12 = add_matrix(M3, M5)
    C21 = add_matrix(M2, M4)
    C22 = add_matrix(sub_matrix(add_matrix(M1, M3), M2), M6)

    return join_quadrants(C11, C12, C21, C22)

# Pad matrix to next power of 2
def pad_matrix(M):
    n = len(M)
    m = 1
    while m < n:
        m *= 2
    padded = [[0]*m for _ in range(m)]
    for i in range(n):
        for j in range(n):
            padded[i][j] = M[i][j]
    return padded, n

# Crop matrix back to original size
def crop_matrix(M, size):
    return [row[:size] for row in M[:size]]

# Read matrix from user with input validation
def read_matrix(name):
    try:
        n = int(input(f"\nEnter the size of square matrix {name} (n x n): "))
        if n <= 0:
            raise ValueError("Matrix size must be a positive integer.")
        matrix = []
        print(f"Enter the elements of matrix {name} row by row (space-separated):")
        for i in range(n):
            row = input(f"Row {i+1}: ").strip().split()
            if len(row) != n:
                raise ValueError(f"Each row must have exactly {n} elements.")
            matrix.append([int(x) for x in row])
        return matrix
    except ValueError as e:
        print("Input Error:", e)
        return None

# Main program with full exception handling
def main():
    print(" Strassen's Matrix Multiplication ")

    A = read_matrix("A")
    if A is None:
        return
```

```
    B = read_matrix("B")
    if B is None:
        return

    # Check if both matrices are square and same size
    try:
        if len(A) != len(B):
            raise ValueError("Matrices A and B must be of the same size.")

        if any(len(row) != len(A) for row in A + B):
            raise ValueError("All rows must have the same number of elements (square matrices only).")

        # Pad matrices to next power of 2
        A_pad, orig_size = pad_matrix(A)
        B_pad, _ = pad_matrix(B)

        # Perform Strassen multiplication
        C_pad = strassen(A_pad, B_pad)

        # Crop to original size
        C = crop_matrix(C_pad, orig_size)

        # Output result
        print("\n Resultant Matrix C = A x B:")
        for row in C:
            print(" ".join(map(str, row)))

    except Exception as e:
        print("Error:", e)

# Run the program
if __name__ == "__main__":
    main()
```

Output :-

```
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python "/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab/04_StrassenMatrixMultiplication.py"
🔢 Strassen's Matrix Multiplication 🔢

Enter the size of square matrix A (n x n): 3
Enter the elements of matrix A row by row (space-separated):
Row 1: 2 4 5
Row 2: 6 5 7
Row 3: 7 5 3

Enter the size of square matrix B (n x n): 3
Enter the elements of matrix B row by row (space-separated):
Row 1: 4 5 7
Row 2: 8 9 -7
Row 3: -4 -5 6

 Resultant Matrix C = A x B:
20 21 16
36 40 49
56 65 32
```

Q5 – WAP in Python for implementing Prim's Algorithm Using Greedy Approach ?

Ans :-

Algorithm :-

Step 1: Initialization

Set a constant INF = float('inf') to represent an infinite weight.

Initialize:

selected[]: to track vertices included in MST (False for all initially).

key[]: to store the minimum weight edge that connects the vertex to the MST (INF for all initially).

parent[]: to store the parent of each vertex in MST (-1 for all initially).

Set key[0] = 0 → start from the first vertex (index 0).

Step 2: Main Loop to Build MST

Repeat n times (where n is the number of vertices):

Find the vertex u with the smallest key value not yet selected:

Set min_key = INF and u = -1.

For each vertex v:

If not selected[v] and key[v] < min_key:

Update min_key = key[v]

Set u = v

Include vertex u into MST:

Set selected[u] = True

Update key[] and parent[] for adjacent vertices of u:

For every vertex v:

If there is an edge (graph[u][v] != 0)

And v is not yet selected

And graph[u][v] < key[v]:

Update key[v] = graph[u][v]

Set parent[v] = u

Step 3: Output the MST

Initialize total_cost = 0

For all vertices from 1 to n-1:

Print parent[i] - i as the MST edge with its weight.

Add the weight to total_cost.

Step 4: Input Function

take_input() reads:

Number of vertices n

An n x n adjacency matrix where 0 means no connection

Calls prims_algorithm(graph, n) to compute MST

Implementation :-

```
INF = float('inf')

def prims_algorithm(graph, n):
    selected = [False] * n
    key = [INF] * n
    parent = [-1] * n

    key[0] = 0  # Start from the first vertex

    for _ in range(n):
        min_key = INF
        u = -1
        for v in range(n):
            if not selected[v] and key[v] < min_key:
                min_key = key[v]
                u = v

        selected[u] = True

        for v in range(n):
            if graph[u][v] != 0 and not selected[v] and graph[u][v] < key[v]:
                key[v] = graph[u][v]
                parent[v] = u

    print("\nMinimum Spanning Tree (MST) Edges and Weights:")
    total_cost = 0
    for i in range(1, n):
        print(f"{parent[i]} - {i} \tWeight: {graph[i][parent[i]]}")
        total_cost += graph[i][parent[i]]

    print(f"Total cost of MST: {total_cost}")

def take_input():
    n = int(input("Enter the number of vertices: "))
    print("Enter the adjacency matrix (use 0 for no connection):")
    graph = []
    for i in range(n):
        row = list(map(int, input(f"Row {i + 1}: ").split()))
        if len(row) != n:
            print("Error: Row length must match number of vertices.")
            return
        graph.append(row)

    prims_algorithm(graph, n)

# Run
take_input()
```

Output :-

```
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python "/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab/05_Prim'sAlgorithm.py"
Enter the number of vertices: 3
Enter the adjacency matrix (use 0 for no connection):
Row 1: 0 2 0
Row 2: 0 0 4
Row 3: 0 2 4

Minimum Spanning Tree (MST) Edges and Weights:
0 - 1   Weight: 0
1 - 2   Weight: 2
Total cost of MST: 2
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python "/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab/05_Prim'sAlgorithm.py"
Enter the number of vertices: 5
Enter the adjacency matrix (use 0 for no connection):
Row 1: 0 2 4 3 0
Row 2: 0 4 3 2 2
Row 3: 2 4 0 0 4
Row 4: 2 3 4 0 5
Row 5: 3 4 5 5 0

Minimum Spanning Tree (MST) Edges and Weights:
0 - 1   Weight: 0
1 - 2   Weight: 4
1 - 3   Weight: 3
1 - 4   Weight: 4
Total cost of MST: 11
```

Q6 – WAP in Python for implementing Krauskal Algorithm Using Greedy Approach ?
Ans :-
Algorithm :-

Step 1: Disjoint Set Data Structure Initialization

Create a DisjointSet class:

parent: each vertex is initially its own parent.

rank: used for union by rank optimization.

DisjointSet Methods:

find(item):

Finds the representative (root) of the set for a given item.

Uses path compression to flatten the structure.

union(u, v):

Merges the sets containing u and v.

Uses union by rank to attach the smaller tree to the root of the larger tree.

Returns True if merged (i.e., edge can be added to MST), else False.

Step 2: Kruskal's Algorithm Function

Function: kruskal_with_union_find(vertices, edges)

Input:

vertices: list of vertex names/labels.

edges: list of tuples (u, v, weight) representing graph edges.

Steps:

Initialize Disjoint Set with vertices.

Sort the edges by increasing weight.

Initialize an empty list mst[] to store MST edges.

Iterate over sorted edges:

For each edge (u, v, weight):

If u and v are in different sets (ds.union(u, v) returns True):

Add the edge to mst[].

Return the constructed MST list.

Step 3: Output the MST

Print each edge of the MST with its weight.

Greedy Nature of Kruskal's Algorithm

Always picks the next lightest edge that does not form a cycle, which is ensured using Union-Find.

Greedy approach helps minimize total weight at every step.

Implementation :-

```
class DisjointSet:
    def __init__(self, vertices):
        self.parent = {v: v for v in vertices}
        self.rank = {v: 0 for v in vertices}

    def find(self, item):
        if self.parent[item] != item:
            self.parent[item] = self.find(self.parent[item])  # Path compression
        return self.parent[item]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)

        if root_u == root_v:
            return False  # They are already in the same set

        # Union by rank
        if self.rank[root_u] < self.rank[root_v]:
            self.parent[root_u] = root_v
        elif self.rank[root_u] > self.rank[root_v]:
            self.parent[root_v] = root_u
        else:
            self.parent[root_v] = root_u
            self.rank[root_u] += 1
        return True


def kruskal_with_union_find(vertices, edges):
    ds = DisjointSet(vertices)
    mst = []

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

    for u, v, weight in edges:
        if ds.union(u, v):
            mst.append((u, v, weight))

    return mst


# Example usage
if __name__ == "__main__":
    vertices = ['A', 'B', 'C', 'D', 'E']
    edges = [
        ('A', 'B', 1),
        ('A', 'C', 3),
        ('B', 'C', 1),
        ('B', 'D', 6),
        ('C', 'D', 4),
```

```
    ('C', 'E', 2),
    ('D', 'E', 5)
]

mst = kruskal_with_union_find(vertices, edges)
print("Minimum Spanning Tree using Union-Find (Disjoint Set):")
for u, v, weight in mst:
    print(f"{u} - {v}: {weight}")
```

Output :-

```
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python "/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab/06_KrauskalAlgorithm.py"
Enter the number of vertices: 3
Enter the adjacency matrix (use 0 for no connection):
Row 1: 0 2 1
Row 2: 2 3 1
Row 3: 0 2 0

Minimum Spanning Tree (MST) Edges and Weights:
0 - 2   Weight: 1
1 - 2   Weight: 1
Total cost of MST: 2
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python "/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab/06_KrauskalAlgorithm.py"
Enter the number of vertices: 5
Enter the adjacency matrix (use 0 for no connection):
Row 1: 0 2 0 3 2
Row 2: 0 4 3 2 0
Row 3: 4 5 3 2 3
Row 4: 5 6 4 0 3
Row 5: 0 0 3 2 4

Minimum Spanning Tree (MST) Edges and Weights:
0 - 1   Weight: 2
0 - 4   Weight: 2
1 - 3   Weight: 2
2 - 3   Weight: 2
Total cost of MST: 8
```

Q7 - WAP in Python for implementing Fractional Knapsack Algorithm Using Greedy Method ?

Ans :-

Algorithm :-

Step 1: Start
Step 2: Input number of items n
Step 3: For each item i = 1 to n
    Input weight[i] and value[i]
Step 4: Input knapsack capacity C
Step 5: For each item i = 1 to n
    Compute value-to-weight ratio ratio[i] = value[i] / weight[i]
    Store as tuple (ratio[i], weight[i], value[i])
Step 6: Sort all items in descending order of ratio[i]
Step 7: Initialize total_value = 0
Step 8: For each item in sorted order:
      If C >= weight[i]:
          Take the full item
          total_value = total_value + value[i]
          C = C – weight[i]
      Else:
          Take fraction of item
          total_value = total_value + ratio[i] * C
          Stop (knapsack full)
Step 9: Output total_value as the maximum profit
Step 10: End

Implementation :-

```
# ---------------------- Fractional Knapsack using Greedy ----------------------
def fractional_knapsack(weights, values, capacity):
    n = len(weights)

    # Compute value-to-weight ratio for each item
    # Store tuple (ratio, weight, value)
    value_per_weight = [
        (values[i] / weights[i], weights[i], values[i]) for i in range(n)]

    # Greedy Step: Sort items by decreasing ratio (value/weight)
    value_per_weight.sort(reverse=True)

    total_value = 0.0
    for ratio, weight, value in value_per_weight:

        # If item can fully fit, take it all
        if capacity >= weight:
            total_value += value
            capacity -= weight
        else:
            # If only part can fit, take fraction of it
            total_value += ratio * capacity
            break   # capacity full
```

```
    return total_value

# ---------------------- Input Section ----------------------
print("Enter number of items:")
n = int(input())

weights = []
values = []

print("Enter weight and value for each item:")
for i in range(n):
    w = int(input(f"Weight of item {i+1}: "))
    v = int(input(f"Value of item {i+1}: "))
    weights.append(w)
    values.append(v)

capacity = int(input("Enter knapsack capacity: "))

# ---------------------- Output Section ----------------------
print("\n=== Results ===")

# Greedy based Fractional Knapsack
print("\n2. Fractional Knapsack (Greedy Approach):")
max_value_frac = fractional_knapsack(weights, values, capacity)
print("Maximum value (Fractional Knapsack):", round(max_value_frac, 2))
```

Output :-

```
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python "/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab/07_KnapsackAlgorithm.py"
Enter number of items:
5
Enter weight and value for each item:
Weight of item 1: 5
Value of item 1: 15
Weight of item 2: 10
Value of item 2: 25
Weight of item 3: 15
Value of item 3: 35
Weight of item 4: 20
Value of item 4: 45
Weight of item 5: 25
Value of item 5: 65
Enter knapsack capacity: 18

=== Results ===

1. 0/1 Knapsack (DP Approach):
Maximum value (0/1 Knapsack): 40

2. Fractional Knapsack (Greedy Approach):
Maximum value (Fractional Knapsack): 48.8
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$
```

Q8 – WAP in Python for implementing DFS & BFS (Graph Traversal Algorithm) using Recursive & Iterative method ?
Ans :-
Algorithm :-

BFS Algorithm

Step 1: Start
Step 2: Input graph, start
Step 3: Initialize visited = ∅, queue = [start], bfs_order = []
Step 4: While queue not empty:
  - Remove front node
  - If node not in visited:
    • Mark visited
    • Append to bfs_order
    • Add all neighbors to queue
Step 5: Print bfs_order
Step 6: End

DFS (Recursive) Algorithm

Step 1: Start
Step 2: Input graph, start
Step 3: Initialize visited = ∅, dfs_order = []
Step 4: Procedure dfs(node)
  - Mark node visited, append to dfs_order
  - For each neighbor not visited → call dfs(neighbor)
Step 5: Call dfs(start)
Step 6: Print dfs_order
Step 7: End

DFS (Iterative using Stack) Algorithm

Step 1: Start
Step 2: Input graph, start
Step 3: Initialize visited = ∅, stack = [start], dfs_order = []
Step 4: While stack not empty:
  - Pop node
  - If node not in visited:
    • Mark visited
    • Append to dfs_order
    • Push neighbors (in reverse order) into stack
Step 5: Print dfs_order
Step 6: End

Implementation :-

```
from collections import deque
# BFS Implementation
def bfs(graph, start):
    visited = set()
```

```python
    queue = deque([start])
    bfs_order = []
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            bfs_order.append(node)
            queue.extend(graph[node])  # Add all neighbors
    return bfs_order

# DFS Implementation (Recursive)
def dfs_recursive(graph, node, visited=None, dfs_order=None):
    if visited is None:
        visited = set()
    if dfs_order is None:
        dfs_order = []
    visited.add(node)
    dfs_order.append(node)
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited, dfs_order)
    return dfs_order

# DFS Implementation (Iterative using stack)
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]
    dfs_order = []
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            dfs_order.append(node)
            stack.extend(reversed(graph[node]))  # reverse for proper order
    return dfs_order

# ---------------------------
# User Input Section
# ---------------------------
if __name__ == "__main__":
    n = int(input("Enter number of vertices: "))
    graph = {}
    for i in range(n):
        vertex = input(f"Enter vertex {i+1} name: ")
        neighbors = input(f"Enter neighbors of {vertex} (space-separated): ").split()
        graph[vertex] = neighbors
    start = input("Enter start vertex: ")
    print("\nGraph:", graph)
    print("BFS Traversal:", bfs(graph, start))
    print("DFS Traversal (Recursive):", dfs_recursive(graph, start))
    print("DFS Traversal (Iterative):", dfs_iterative(graph, start))
```

Output :-

```
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python3 "/home
Enter number of vertices: 4
Enter vertex 1 name: 0
Enter neighbors of 0 (space-separated): 1 2 3
Enter vertex 2 name: 1
Enter neighbors of 1 (space-separated): 0 2 3
Enter vertex 3 name: 2
Enter neighbors of 2 (space-separated): 0 1 3
Enter vertex 4 name: 3
Enter neighbors of 3 (space-separated): 0 1 2
Enter start vertex: 2

Graph: {'0': ['1', '2', '3'], '1': ['0', '2', '3'], '2': ['0', '1', '3'], '3': ['0', '1', '2']}
BFS Traversal: ['2', '0', '1', '3']
DFS Traversal (Recursive): ['2', '0', '1', '3']
DFS Traversal (Iterative): ['2', '0', '1', '3']
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$
```

Q9 - WAP in Python for implementing Integer(0/1) Knapsack Algorithm Using DP Method ?

Ans :-

Algorithm :-

Step 1: Start
Step 2: Input number of items n
Step 3: For each item i = 1 to n
    Input weight[i] and value[i]
Step 4: Input knapsack capacity C
Step 5: Create a DP table K of size (n+1) × (C+1)
    Initialize all entries K[i][w] = 0
Step 6: For each item i = 1 to n:
    For each capacity w = 1 to C:
Case 1: If weight[i-1] ≤ w (item can fit):
    - Compute include = value[i-1] + K[i-1][w - weight[i-1]]
    - Compute exclude = K[i-1][w]
    - Set K[i][w] = max(include, exclude)
Case 2: Else (item cannot fit):
    - Set K[i][w] = K[i-1][w]
Step 7: Final result = K[n][C] (bottom-right cell of DP table)
Step 8: Output maximum value that can be achieved
Step 9: End

Implementation :-

```
# ---------------------- 0/1 Knapsack using Dynamic Programming ----------------------
def knapsack_01(weights, values, capacity):
    n = len(weights)

    # DP Table: K[i][w] will store the maximum value
    # that can be attained with items 0..i and capacity w
    K = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    # Build DP table in bottom-up manner
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):

            # If current item can fit in knapsack
            if weights[i - 1] <= w:
                # Option 1: Include this item -> values[i-1] + value of remaining capacity
                include = values[i - 1] + K[i - 1][w - weights[i - 1]]

                # Option 2: Exclude this item -> value remains same as previous row
                exclude = K[i - 1][w]

                # Take max of including or excluding item
                K[i][w] = max(include, exclude)
            else:
                # Item cannot fit, so take previous row's value
                K[i][w] = K[i - 1][w]
```

```
    # Final answer is stored at bottom-right corner of DP table
    return K[n][capacity]

# ---------------------- Input Section ----------------------
print("Enter number of items:")
n = int(input())
weights = []
values = []
print("Enter weight and value for each item:")
for i in range(n):
    w = int(input(f"Weight of item {i+1}: "))
    v = int(input(f"Value of item {i+1}: "))
    weights.append(w)
    values.append(v)

capacity = int(input("Enter knapsack capacity: "))

# ---------------------- Output Section ----------------------
print("\n=== Results ===")

# Dynamic Programming based 0/1 Knapsack
print("\n1. 0/1 Knapsack (DP Approach):")
max_value_01 = knapsack_01(weights, values, capacity)
print("Maximum value (0/1 Knapsack):", max_value_01)
```

Output :-

```
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bi
Enter number of items:
4
Enter weight and value for each item:
Weight of item 1: 2
Value of item 1: 12
Weight of item 2: 1
Value of item 2: 10
Weight of item 3: 3
Value of item 3: 20
Weight of item 4: 2
Value of item 4: 15
Enter knapsack capacity: 5

=== Results ===

1. 0/1 Knapsack (DP Approach):
Maximum value (0/1 Knapsack): 37
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$
```

Q10 – WAP in Python for implementing TSP(Traveling Sales Problem) Algorithm Using Greedy Method?
Ans :-
Algorithm :-

Step 1: Start
Step 2: Input number of cities n and cost adjacency matrix graph[n][n]
Step 3: Initialize visited[] = False, mark start city 0 as visited
    Set min_cost = ∞
Step 4: Define recursive tsp(current, count, cost)
    If all cities visited and edge to start exists →
        Update min_cost = min(min_cost, cost + return_edge)
    Else for each unvisited city i:
        Mark visited → call tsp(i, count+1, cost + graph[current][i]) → backtrack
Step 5: Call tsp(0, 1, 0)
Step 6: Output min_cost
Step 7: End

Implementation :-

```
import sys

def tsp(graph, visited, current_pos, n, count, cost, start_pos, min_cost):
    if count == n and graph[current_pos][start_pos] > 0:
        min_cost[0] = min(min_cost[0], cost + graph[current_pos][start_pos])
        return
    for i in range(n):
        if not visited[i] and graph[current_pos][i] > 0:
            visited[i] = True
            tsp(graph, visited, i, n, count + 1, cost + graph[current_pos][i], start_pos, min_cost)
            visited[i] = False  # backtrack
def take_input_and_run_tsp():
    n = int(input("Enter the number of cities (vertices): "))
    print("Enter the cost adjacency matrix (use 0 if no direct path):")
    graph = []
    for i in range(n):
        row = list(map(int, input(f"Row {i + 1}: ").split()))
        if len(row) != n:
            print("Error: Row length must be equal to number of cities.")
            return
        graph.append(row)
    visited = [False] * n
    visited[0] = True
    min_cost = [sys.maxsize]
    tsp(graph, visited, 0, n, 1, 0, 0, min_cost)
    print(f"\nMinimum cost to complete TSP tour: {min_cost[0]}")

# Run
take_input_and_run_tsp()
```

Output :-

```
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python /mnt/data23/My\ MCA\ Work/2nd\ Sem\ 2024-25/01\ -\ DAA/DAA_Lab/08_TSPAlgorithm.py
Enter the number of cities (vertices): 5
Enter the cost adjacency matrix (use 0 if no direct path):
Row 1: 0 10 3 7 9
Row 2: 2 5 7 10 14
Row 3: 4 6 9 20 15
Row 4: 16 18 20 10 5
Row 5: 25 30 14 16 19

Minimum cost to complete TSP tour: 34
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python /mnt/data23/My\ MCA\ Work/2nd\ Sem\ 2024-25/01\ -\ DAA/DAA_Lab/08_TSPAlgorithm.py
Enter the number of cities (vertices): 4
Enter the cost adjacency matrix (use 0 if no direct path):
Row 1: 0 10 15 20
Row 2: 10 0 35 25
Row 3: 15 35 0 30
Row 4: 20 25 30 0

Minimum cost to complete TSP tour: 80
coder@hackycoder:/mnt/data23/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$
```

Q11 – WAP in Python for implementing TSP(Traveling Sales Problem) Algorithm Using Dynamic Programming?
Ans :-
Algorithm :-

Step 1: Start
Step 2: Input number of cities n and cost adjacency matrix graph[n][n]
    (Use graph[u][v] > 0 to mean an edge exists; 0 = no direct path.)
Step 3: Let ALL = (1 << n) - 1
    Create dp[0..ALL][0..n-1] and set every entry = ∞
Step 4: Base case: dp[1 << 0][0] = 0 // visited only city 0, ended at 0
Step 5: For mask from 1 to ALL:
  For each u in 0..n-1 such that (mask >> u) & 1 == 1 and dp[mask][u] ≠ ∞:
    For each v in 0..n-1 such that (mask >> v) & 1 == 0 and graph[u][v] > 0:
      new_mask = mask | (1 << v)
      dp[new_mask][v] = min(dp[new_mask][v], dp[mask][u] + graph[u][v])
Step 6: final_mask = ALL
    min_tour_cost = ∞
    For every u = 0..n-1 with graph[u][0] > 0:
        min_tour_cost = min(min_tour_cost, dp[final_mask][u] + graph[u][0])
Step 7: Output min_tour_cost (if ∞, no Hamiltonian cycle exists)
Step 8: End

Implementation :-

```
import sys
def tsp_dp(graph):
    n = len(graph)
    # dp[mask][i] will be the minimum cost to visit all cities in 'mask'
    # ending at city 'i'.
    # Initialize with infinity
    dp = [[sys.maxsize for _ in range(n)] for _ in range(1 << n)]
    # Base case: starting at city 0, cost to visit only city 0 is 0
    dp[1 << 0][0] = 0
    # Iterate over all possible masks (subsets of cities)
    for mask in range(1, 1 << n):
        for u in range(n):
            # If city u is in the current mask
            if (mask >> u) & 1:
                # If dp[mask][u] is still infinity, it means we haven't found a path to u yet
                if dp[mask][u] == sys.maxsize:
                    continue
                # Try to extend the path to an unvisited city v
                for v in range(n):
                    # If v is not in the current mask and there's a path from u to v
                    if not ((mask >> v) & 1) and graph[u][v] > 0:
                        new_mask = mask | (1 << v)
                        dp[new_mask][v] = min(dp[new_mask][v], dp[mask][u] + graph[u][v])
    # After filling the DP table, find the minimum cost to return to the starting city (city 0)
    # from any city, having visited all cities.
    min_tour_cost = sys.maxsize
    final_mask = (1 << n) - 1 # Mask where all cities are visited
```

```
    for u in range(n):
        if graph[u][0] > 0: # Check if there's a path from u back to city 0
            min_tour_cost = min(min_tour_cost, dp[final_mask][u] + graph[u][0])
    return min_tour_cost
def take_input_and_run_tsp_dp():
    n = int(input("Enter the number of cities (vertices): "))
    print("Enter the cost adjacency matrix (use 0 if no direct path):")
    graph = []
    for i in range(n):
        row = list(map(int, input(f"Row {i + 1}: ").split()))
        if len(row) != n:
            print("Error: Row length must be equal to number of cities.")
            return
        graph.append(row)
    min_cost = tsp_dp(graph)
    print(f"\nMinimum cost to complete TSP tour (DP): {min_cost}")
# Run
take_input_and_run_tsp_dp()
```

Output :-

```
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /
my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab/10_TSPAlgorithmDP.py"
Enter the number of cities (vertices): 4
Enter the cost adjacency matrix (use 0 if no direct path):
Row 1: 0 10 15 20
Row 2: 10 0 35 25
Row 3: 15 45 0 30
Row 4: 20 25 30 0

Minimum cost to complete TSP tour (DP): 80
```

Q12 – WAP in Python for implementing Sum of Subset Algorithm Using Backtracking Method?

Ans :-

Algorithm :-

Step 1: Start
Step 2: Input number set weights[] and target sum T
Step 3: Display all possible subsets of weights[]
Step 4: Initialize a queue with state (index = 0, current_sum = 0, subset = ∅)
  Set found = False
Step 5: While queue not empty:
  Remove state (i, sum, subset)
  - If sum == T → print subset, set found = True
  - Else if i ≥ n or sum > T → skip
  - Else:
    Add state (i+1, sum + weights[i], subset ∪ {weights[i]})
    Add state (i+1, sum, subset)
Step 6: If found == False → print “No subset found”
Step 7: End

Implementation :-

```
from collections import deque
from itertools import chain, combinations
def sum_of_subsets_iterative(weights, target_sum):
    n = len(weights)
    weights.sort()
    queue = deque()
    queue.append((0, 0, []))  # (index, current_sum, current_subset)
    found = False
    while queue:
        i, current_sum, subset = queue.popleft()
        if current_sum == target_sum:
            print("Subset found:", subset)
            found = True
            continue
        if i >= n or current_sum > target_sum:
            continue
        # Include weights[i]
        queue.append((i + 1, current_sum + weights[i], subset + [weights[i]]))
        # Exclude weights[i]
        queue.append((i + 1, current_sum, subset))
    if not found:
        print("No subset found.")
# === Main Program ===
weights = list(map(int, input("Enter the set of weights (space separated): ").split()))
target_sum = int(input("Enter the target sum: "))
def all_subsets(weights):
    subsets = list(chain.from_iterable(combinations(weights, r) for r in range(len(weights)+1)))
    print("All subsets:")
    for subset in subsets:
        print(list(subset))
```

```
all_subsets(weights)
print("\nSubsets whose sum is", target_sum, ":")
sum_of_subsets_iterative(weights, target_sum)
```

Output :-

```
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python3 "/hom
Enter the set of weights (space separated): 30 24 12 35
Enter the target sum: 47
All subsets:
[]
[30]
[24]
[12]
[35]
[30, 24]
[30, 12]
[30, 35]
[24, 12]
[24, 35]
[12, 35]
[30, 24, 12]
[30, 24, 35]
[30, 12, 35]
[24, 12, 35]
[30, 24, 12, 35]

Subsets whose sum is 47 :
Subset found: [12, 35]
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$
```

Q13 – WAP in Python for implementing N Queen Problem Algorithm Using Backtracking Method?
Ans :-
Algorithm :-

Step 1: Start
Step 2: Input board size n (number of queens)
Step 3: Initialize board[0..n-1] = -1 // stores column positions
Step 4: Define recursive procedure queens(row)
- If row == n → all queens placed → return True
- For each column col = 0..n-1:
    Place queen at (row, col)
    If no conflict with previous rows → call queens(row+1)
    If recursive call succeeds → return True
- If no column works → return False

Step 5: Define noConflicts(row)
- For each earlier row i:
    If same column or diagonal → return False
- Else return True

Step 6: Call queens(0)
- If True → print board with queens placed
- Else → print “No solution exists”

Step 7: End

Implementation :-

```
def calcQueens(size):
	board = [-1] * size
	if queens(board, 0, size):
		return board
	else:
		return None
def queens(board, current, size):
	if current == size:
		return True
	else:
		for i in range(size):
			board[current] = i
			if noConflicts(board, current):
				if queens(board, current + 1, size):
					return True
def noConflicts(board, current):
	for i in range(current):
		if board[i] == board[current]:
			return False
		if abs(board[current] - board[i]) == current - i:
			return False
	return True
if __name__ == "__main__":
	try:
		n = int(input("Enter the number of queens: "))
```

```
        if n <= 0:
            print("Number of queens must be positive.")
        else:
            solution = calcQueens(n)
            if solution:
                print(f"A solution exists for {n} queens:")
                for row in range(n):
                    line = ['Q' if solution[row] == col else '_' for col in range(n)]
                    print(' '.join(line))
            else:
                print(f"No solution exists for {n} queens.")
    except ValueError:
        print(f"No solution exists for {n} queens.")
    except ValueError:
        print("Invalid input. Please enter an integer.")
```

Output :-

```
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bir
Enter the number of queens: 5
A solution exists for 5 queens:
Q _ _ _ _
_ _ Q _ _
_ _ _ _ Q
_ Q _ _ _
_ _ _ Q _
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bir
Enter the number of queens: 10
A solution exists for 10 queens:
Q _ _ _ _ _ _ _ _ _
_ _ Q _ _ _ _ _ _ _
_ _ _ _ _ Q _ _ _ _
_ _ _ _ _ _ _ Q _ _
_ _ _ _ _ _ _ _ _ Q
_ _ _ _ Q _ _ _ _ _
_ _ _ _ _ _ _ _ Q _
_ Q _ _ _ _ _ _ _ _
_ _ _ Q _ _ _ _ _ _
_ _ _ _ _ _ Q _ _ _
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$
```

Q14 – WAP in Python for implementing Integer(0/1) Knapsack Algorithm Using Backtracking Method?

Ans :-

Algorithm :-

Step 1: Start
Step 2: Input n, weights[0..n-1], values[0..n-1], capacity
Step 3: Initialize max_value = 0, best_items = []
Step 4: Define knapsack(index, cur_w, cur_v, items)
    - If index == n → if cur_v > max_value then max_value = cur_v, best_items = copy(items); return
    - If cur_w + weights[index] <= capacity → items.append(index); knapsack(index+1, cur_w+weights[index], cur_v+values[index], items); items.pop()
    - knapsack(index+1, cur_w, cur_v, items)
Step 5: Call knapsack(0, 0, 0, [])
Step 6: Print max_value and best_items
Step 7: End

Implementation :-

```
# 0/1 Knapsack using Backtracking
def knapsack_backtracking(weights, values, capacity):
    n = len(weights)
    max_value = 0
    best_items = []

    def backtrack(index, current_weight, current_value, current_items):
        nonlocal max_value, best_items

        # Base case: if all items are considered
        if index == n:
            if current_value > max_value:
                max_value = current_value
                best_items = list(current_items)
            return

        # Case 1: Include the current item if it fits
        if current_weight + weights[index] <= capacity:
            current_items.append(index)
            backtrack(index + 1,
                      current_weight + weights[index],
                      current_value + values[index],
                      current_items)
            current_items.pop()  # Backtrack

        # Case 2: Exclude the current item
        backtrack(index + 1, current_weight, current_value, current_items)

    backtrack(0, 0, 0, [])
    return max_value, best_items
```

```
# ---------------------- Input Section ----------------------
print("=== Knapsack using Backtracking (0/1) ===")
n = int(input("Enter number of items: "))

weights = []
values = []

print("Enter weight and value for each item:")
for i in range(n):
    w = int(input(f"Weight of item {i+1}: "))
    v = int(input(f"Value of item {i+1}: "))
    weights.append(w)
    values.append(v)

capacity = int(input("Enter knapsack capacity: "))

# ---------------------- Output Section ----------------------
# 0/1 Knapsack Backtracking
max_value_bt, best_items_bt = knapsack_backtracking(weights, values, capacity)
print("\n0/1 Knapsack (Backtracking):")
print("Maximum value:", max_value_bt)
print("Items taken (indices):", best_items_bt)
```

Output :-

```
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bi
=== Knapsack using Backtracking (0/1) ===
Enter number of items: 4
Enter weight and value for each item:
Weight of item 1: 0
Value of item 1: 5
Weight of item 2: 1
Value of item 2: 15
Weight of item 3: 2
Value of item 3: 20
Weight of item 4: 3
Value of item 4: 10
Enter knapsack capacity: 35

0/1 Knapsack (Backtracking):
Maximum value: 50
Items taken (indices): [0, 1, 2, 3]
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$
```

Q15 – WAP in Python for implementing Knapsack Algorithm Using Branch And Bounding Method?

Ans :-

Algorithm :-

Step 1: Start
Step 2: Input n, items[0..n-1] (weight,value), capacity
Step 3: Compute ratio = value/weight for each item and sort items by ratio desc
Step 4: Define bound(cur_w, cur_v, idx) — greedy fractional fill from idx
Step 5: max_profit = 0; PQ = [(-bound(0,0,0), 0, 0, 0)]
Step 6: While PQ not empty:
    - Pop (-b, w, v, i) → b = -(-b)
    - If b <= max_profit → continue
    - If i == n → max_profit = max(max_profit, v); continue
    - If w + items[i].weight <= capacity → push (-bound(w+wi, v+vi, i+1), w+wi, v+vi, i+1); max_profit = max(max_profit, v+vi)
    - Push (-bound(w, v, i+1), w, v, i+1) if its bound > max_profit
Step 7: Print max_profit
Step 8: End

Implementation :-

```
import heapq

class Item:
    def __init__(self, weight, value):
        self.weight = weight
        self.value = value
        self.ratio = value / weight

def knapsack_branch_and_bound(items, capacity):
    n = len(items)
    items.sort(key=lambda x: x.ratio, reverse=True)

    max_profit = 0

    # Priority queue stores tuples: (-upper_bound, current_weight, current_value, item_index)
    # We use negative upper_bound to make it a min-heap for max-profit
    pq = [(-calculate_bound(0, 0, 0, items, capacity), 0, 0, 0)] # (bound, current_weight,
current_value, item_index)

    while pq:
        bound, current_weight, current_value, item_index = heapq.heappop(pq)
        bound = -bound # Convert back to positive bound

        if bound < max_profit:
            continue

        if item_index == n:
            max_profit = max(max_profit, current_value)
            continue
```

```
        # Include the current item
        if current_weight + items[item_index].weight <= capacity:
            new_weight = current_weight + items[item_index].weight
            new_value = current_value + items[item_index].value
            new_bound = calculate_bound(new_weight, new_value, item_index + 1, items, capacity)
            if new_bound > max_profit:
                heapq.heappush(pq, (-new_bound, new_weight, new_value, item_index + 1))
            max_profit = max(max_profit, new_value) # Update max_profit if this path is better

        # Exclude the current item
        new_bound = calculate_bound(current_weight, current_value, item_index + 1, items, capacity)
        if new_bound > max_profit:
            heapq.heappush(pq, (-new_bound, current_weight, current_value, item_index + 1))

    return max_profit

def calculate_bound(current_weight, current_value, item_index, items, capacity):
    n = len(items)
    if current_weight >= capacity:
        return current_value
    bound = current_value
    total_weight = current_weight
    for i in range(item_index, n):
        if total_weight + items[i].weight <= capacity:
            total_weight += items[i].weight
            bound += items[i].value
        else:
            remain = capacity - total_weight
            bound += items[i].ratio * remain
            break
    return bound

# ---------------------- Input Section ----------------------
print("=== 0/1 Knapsack using Branch and Bound ===")
n = int(input("Enter number of items: "))

weights = []
values = []
items = []

print("Enter weight and value for each item:")
for i in range(n):
    w = int(input(f"Weight of item {i+1}: "))
    v = int(input(f"Value of item {i+1}: "))
    weights.append(w)
    values.append(v)
    items.append(Item(w, v))

capacity = int(input("Enter knapsack capacity: "))
```

```
# ---------------------- Output Section ----------------------
# 0/1 Knapsack Branch and Bound
max_value_bnb = knapsack_branch_and_bound(items, capacity)
print("\n0/1 Knapsack (Branch and Bound):")
print("Maximum value:", max_value_bnb)
```

Output :-

```
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bi
=== 0/1 Knapsack using Branch and Bound ===
Enter number of items: 5
Enter weight and value for each item:
Weight of item 1: 1
Value of item 1: 12
Weight of item 2: 2
Value of item 2: 24
Weight of item 3: 3
Value of item 3: 36
Weight of item 4: 4
Value of item 4: 48
Weight of item 5: 5
Value of item 5: 60
Enter knapsack capacity: 75

0/1 Knapsack (Branch and Bound):
Maximum value: 180
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$
```

Q16 – WAP in Python for implementing TSP(Traveling Sales Problem) Algorithm Using Branch And Bounding Method?

Ans :-

Algorithm :-

Step 1: Start
Step 2: Input n and adjacency matrix graph[n][n]
Step 3: Compute initial_lower_bound = sum(min outgoing edge from each city)
Step 4: min_cost = ∞, best_path = []
Step 5: PQ = [(initial_lower_bound, 0, [0], 1<<0)] # (lower_bound, current_city, path, visited_mask)
Step 6: While PQ not empty:
- Pop (lb, cur, path, mask)
- If len(path) == n:
    - cost = sum(graph[path[i]][path[i+1]] for i in 0..n-2) + graph[path[-1]][path[0]]
    - If cost < min_cost: min_cost = cost, best_path = path + [path[0]]
    - continue
- If lb >= min_cost: continue # prune
- For each next in 0..n-1 not visited in mask:
    - new_mask = mask | (1<<next)
    - new_path = path + [next]
    - path_cost = sum(graph[new_path[i]][new_path[i+1]] for i in 0..len(new_path)-2)
    - remaining_lb = sum(min outgoing edge from each unvisited city excluding visited ones)
    - new_lb = path_cost + remaining_lb
    - Push (new_lb, next, new_path, new_mask) into PQ

Step 7: Print min_cost and best_path
Step 8: End

Implementation :-

```
import heapq

def tsp_branch_and_bound(graph):
    num_cities = len(graph)

    # Priority queue to store (cost, path, visited_mask)
    # Cost is the lower bound of the path
    pq = []

    # Initial state: start from city 0
    # (cost, current_path, visited_mask)
    # current_path is a list of cities visited so far
    # visited_mask is a bitmask to keep track of visited cities

    # Calculate initial lower bound (sum of minimum outgoing edges from each city)
    initial_lower_bound = 0
    for i in range(num_cities):
        min_edge = float('inf')
        for j in range(num_cities):
            if i != j:
                min_edge = min(min_edge, graph[i][j])
```

```
        initial_lower_bound += min_edge

    # Add the starting node to the priority queue
    # (lower_bound, current_city, path_list, visited_mask)
    heapq.heappush(pq, (initial_lower_bound, 0, [0], 1 << 0))

    min_cost = float('inf')
    best_path = []

    while pq:
        lower_bound, current_city, path, visited_mask = heapq.heappop(pq)

        # If all cities are visited
        if len(path) == num_cities:
            # Add the cost to return to the starting city
            current_total_cost = sum(graph[path[i]][path[i+1]] for i in range(num_cities - 1)) +
graph[path[-1]][path[0]]

            if current_total_cost < min_cost:
                min_cost = current_total_cost
                best_path = path + [path[0]] # Add starting city to complete the cycle
            continue

        # If current lower bound is already greater than or equal to the best found cost, prune
        if lower_bound >= min_cost:
            continue

        # Explore neighbors
        for next_city in range(num_cities):
            if not (visited_mask &(1 << next_city)): # If next_city has not been visited
                new_visited_mask = visited_mask | (1 << next_city)
                new_path = path + [next_city]

                # Calculate new lower bound
                # This is a simplified lower bound calculation.
                # A more sophisticated one would involve minimum spanning trees or assignment
problems.
                # For simplicity, we'll just add the cost of the new edge to the current lower bound.
                # This is not strictly correct for a tight lower bound but serves as an example.

                # A better lower bound for the remaining path could be calculated here.
                # For this example, we'll use a simple heuristic:
                # current path cost + minimum outgoing edge from next_city to an unvisited city

                current_path_cost = sum(graph[new_path[i]][new_path[i+1]] for i in range(len(new_path)
- 1))

                remaining_lower_bound = 0
                unvisited_cities = []
                for i in range(num_cities):
                    if not (new_visited_mask & (1 << i)):
                        unvisited_cities.append(i)
```

```
                if unvisited_cities:
                    # Find minimum outgoing edge from each unvisited city
                    for city_u in unvisited_cities:
                        min_out_edge = float('inf')
                        for city_v in range(num_cities):
                            if city_u != city_v and not (new_visited_mask & (1 << city_v)):
                                min_out_edge = min(min_out_edge, graph[city_u][city_v])
                        if min_out_edge != float('inf'):
                            remaining_lower_bound += min_out_edge

                # The lower bound should also include the cost to return to the start from the last
unvisited city
                # This is a very simplified lower bound. For a true B&B, you'd need a more robust
calculation.
                # For now, let's just use the current path cost as a base for the lower bound.
                new_lower_bound = current_path_cost + remaining_lower_bound

                heapq.heappush(pq, (new_lower_bound, next_city, new_path, new_visited_mask))
    return min_cost, best_path

def take_input_and_run_tsp_bnb():
    n = int(input("Enter the number of cities (vertices): "))
    print("Enter the cost adjacency matrix (use 0 for no direct path, or a very large number like 999
for no direct path):")
    graph = []
    for i in range(n):
        row = list(map(int, input(f"Row {i + 1}: ").split()))
        if len(row) != n:
            print("Error: Row length must be equal to number of cities.")
            return
        # Replace 0 with float('inf') for non-existent paths if needed,
        # but for TSP, 0 usually means no direct path and should be handled.
        # Here, we assume 0 means no direct path and will be skipped in calculations.
        # If the user enters a large number for no path, it will be used.
        graph.append(row)

    min_cost, best_path = tsp_branch_and_bound(graph)

    if min_cost == float('inf'):
        print("\nNo TSP tour found.")
    else:
        print(f"\nMinimum cost to complete TSP tour (Branch and Bound): {min_cost}")
        # Adjust path to show city numbers starting from 1 if desired
        print("Best path:", [city + 1 for city in best_path])

# Run
if __name__ == "__main__":
    take_input_and_run_tsp_bnb()
```

Output :-

```
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$ /bin/python3 "/home/hackycoder/my_Da
Enter the number of cities (vertices): 4
Enter the cost adjacency matrix (use 0 for no direct path, or a very large number like 999 for no direct path):
Row 1: 1 3 4 5
Row 2: 3 2 1 5
Row 3: 5 3 2 4
Row 4: 2 4 3 1

Minimum cost to complete TSP tour (Branch and Bound): 10
Best path: [1, 2, 3, 4, 1]
hackycoder@hackycoder:~/my_Data/My MCA Work/2nd Sem 2024-25/01 - DAA/DAA_Lab$
```