Question 1:

Implement a program that solves the Traveling Salesman Problem (TSP) using dynamic programming.The program should take input regarding a graph representing cities and their distances.The output should display the minimum cost of the Traveling Salesman Problem for the given graph.

Input Format :

- The first line contains an integer, n, representing the number of cities.
- The next n lines contain the adjacency matrix for the graph, where each line contains n space-separated integers representing the distances between cities.

Output Format:

- An integer representing the minimum cost of the Traveling Salesman Problem for the given graph.

Title for Question 1: Dynamic Programming for TSP

Solution:

```
#include<iostream>
#include<vector>
#include<climits>
#include<algorithm>

using namespace std;

int main()
{
 int n,s=0;
 cin>>n;
 int graph[n][n];
 vector<int> vertex;
 int min_path = INT_MAX;

 for(int i=0;i<n;i++)
     for(int j=0;j<n;j++)
         cin>>graph[i][j];
 for (int i = 0; i < n; i++)
  if (i != s)
   vertex.push_back(i);

 do {
  int current_pathweight = 0;

  int k = s;
  for (int i = 0; i < vertex.size(); i++) {
   current_pathweight += graph[k][vertex[i]];
   k = vertex[i];
  }
```

```
  current_pathweight += graph[k][s];
  min_path = min(min_path, current_pathweight);

 } while (next_permutation(vertex.begin(), vertex.end()));
 cout <<"Minimum Cost of TSP: "<< min_path << endl;

}
```

TestCases:

S.No	Inputs	Outputs
1	4 0 10 15 20 10 0 35 25 15 35 0 30 20 25 30 0	Minimum Cost of TSP: 80
2	5 0 29 20 21 16 29 0 15 26 25 20 15 0 29 30 21 26 29 0 22 16 25 30 22 0	Minimum Cost of TSP: 99
3	3 0 10 15 10 0 35 15 35 0	Minimum Cost of TSP: 60
4	6 0 2 9 10 7 3 2 0 10 5 8 7 9 10 0 3 1 10 10 5 3 0 6 2 7 8 1 6 0 4 3 7 10 2 4 0	Minimum Cost of TSP: 18
5	4 0 1 2 3 1 0 4 5 2 4 0 6 3 5 6 0	Minimum Cost of TSP: 14
6	2 0 1 1 0	Minimum Cost of TSP: 2

White List:

Black List:

Question 2:

Design a algorithm based on a greedy approach to approximate the solution for the Traveling Salesman Problem. Evaluate the effectiveness of your algorithm by comparing the results with optimal solutions for different instances of the problem.

Input Format:

- The first line contains an integer, n, representing the number of cities.
- The next n lines contain the adjacency matrix for the graph, where each line contains n space-separated integers representing the distances between cities.

Output Format:

- An integer representing the approximate distance using the Greedy TSP Approximation algorithm.

Title for Question 2: Greedy TSP Approximation

Solution:

```cpp
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

const int INF = INT_MAX;

int greedyTSPApproximation(int n, vector<vector<int>>& graph) {
    // Initialize variables
    int totalDistance = 0;
    vector<bool> visited(n, false);

    // Start from the first city (City 0)
    int currentCity = 0;
    visited[currentCity] = true;

    // Greedy nearest neighbor approach
    for (int i = 1; i < n; ++i) {
        int nearestCity = -1;
        int minDistance = INF;

        // Find the nearest unvisited city
        for (int j = 0; j < n; ++j) {
            if (!visited[j] && graph[currentCity][j] < minDistance) {
                nearestCity = j;
                minDistance = graph[currentCity][j];
            }
        }

        // Update total distance and mark the city as visited
        totalDistance += minDistance;
        visited[nearestCity] = true;

        // Move to the nearest city
        currentCity = nearestCity;
    }

    // Add the distance from the last visited city to the initial city
    totalDistance += graph[currentCity][0];

    return totalDistance;
}

int main() {
    // Input: Get the number of cities from the user
    int n;
   // cout << "Enter the number of cities: ";
    cin >> n;

    // Input: Get the adjacency matrix from the user
    vector<vector<int>> graph(n, vector<int>(n));
  //  cout << "Enter the adjacency matrix for the graph:" << endl;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> graph[i][j];
        }
    }

    // Greedy TSP Approximation
```

```
    int approximateDistance = greedyTSPApproximation(n, graph);

    // Output the approximate solution
    cout << "Approximate Distance using Greedy TSP: " << approximateDista

    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	5 0 10 20 30 40 10 0 25 35 45 20 25 0 15 50 30 35 15 0 55 40 45 50 55 0	Approximate Distance using Greedy TSP: 145
2	4 0 1 2 3 1 0 4 5 2 4 0 6 3 5 6 0	Approximate Distance using Greedy TSP: 14
3	6 0 10 15 20 25 30 10 0 35 40 45 50 15 35 0 55 60 65 20 40 55 0 75 80 25 45 60 75 0 90 30 50 65 80 90 0	Approximate Distance using Greedy TSP: 295
4	3 0 5 10 5 0 15 10 15 0	Approximate Distance using Greedy TSP: 30
5	7 0 12 23 34 45 56 67 12 0 11 22 33 44 55 23 11 0 67 78 89 90 34 22 67 0 55 66 77 45 33 78 55 0 21 32 56 44 89 66 21 0 43 67 55 90 77 32 43 0	Approximate Distance using Greedy TSP: 276
6	4 0 1 2 3 1 0 2 3 2 2 0 1 3 3 1 0	Approximate Distance using Greedy TSP: 7

White List:

Black List:

Question 3:

Design a function implementing the Nearest Neighbor heuristic for the Traveling Salesman Problem. Apply the heuristic to a given graph and analyze the quality of the solution obtained compared to an optimal solution.

Input Format:

- The first line contains an integer n representing the number of cities (2 <= n <= 10).
- The next n lines contain the distance matrix for the graph. Each line contains n integers, where the j-th integer represents the distance from the i-th city to the j-th city. All distances are non-negative integers.

Output Format:

- The output should display the Nearest Neighbor TSP solution and the total distance of the route.

Title for Question 3: TSP with Nearest Neighbor Heuristic

Solution:

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

const int INF = INT_MAX;

// Function to find the nearest neighbor for a given city
int findNearestNeighbor(int currentCity, const vector<vector<int>>& graph
    int nearestCity = -1;
    int minDistance = INF;
    int n = graph.size();

    for (int i = 0; i < n; ++i) {
        if (!visited[i] && graph[currentCity][i] < minDistance) {
            nearestCity = i;
            minDistance = graph[currentCity][i];
        }
    }

    return nearestCity;
}

// Function implementing the Nearest Neighbor heuristic for TSP
vector<int> nearestNeighborTSP(const vector<vector<int>>& graph) {
    int n = graph.size();

    // Initialize visited array
    vector<bool> visited(n, false);

    // Start from the first city (City 0)
    int currentCity = 0;
    visited[currentCity] = true;

    // Initialize the solution path with the starting city
    vector<int> path;
    path.push_back(currentCity);

    // Find the nearest neighbor for each city
    for (int i = 1; i < n; ++i) {
        int nearestCity = findNearestNeighbor(currentCity, graph, visited

        // Mark the nearest city as visited
        visited[nearestCity] = true;

        // Move to the nearest city
        currentCity = nearestCity;

        // Add the nearest city to the solution path
        path.push_back(currentCity);
    }

    return path;
}
```

```
// Function to calculate the total distance of a route in the TSP
int calculateTotalDistance(const vector<int>& path, const vector<vector<i
    int totalDistance = 0;
    int n = path.size();

    for (int i = 0; i < n - 1; ++i) {
        totalDistance += distances[path[i]][path[i + 1]];
    }

    // Return to the starting city
    totalDistance += distances[path[n - 1]][path[0]];

    return totalDistance;
}

int main() {
    // Input: Get the number of cities from the user
    int n;
  //  cout << "Enter the number of cities: ";
    cin >> n;

    // Input: Get the distance matrix from the user
    vector<vector<int>> distances(n, vector<int>(n));
// cout << "Enter the distance matrix for the graph:" << endl;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> distances[i][j];
        }
    }

    // Apply Nearest Neighbor heuristic
    vector<int> solution = nearestNeighborTSP(distances);

    // Output the solution
    cout << "Nearest Neighbor TSP Solution: ";
    for (int city : solution) {
        cout << city << " ";
    }
    cout << "\nTotal Distance: " << calculateTotalDistance(solution, dist

    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	4 0 10 15 30 10 0 65 25 15 65 0 30 30 25 30 0	Nearest Neighbor TSP Solution: 0 1 3 2 Total Distance: 80
2	3 0 10 20 10 0 15 20 15 0	Nearest Neighbor TSP Solution: 0 1 2 Total Distance: 45
3	4 0 2 9 10 1 0 6 4 15 7 0 8 6 3 12 0	Nearest Neighbor TSP Solution: 0 1 3 2 Total Distance: 33
4	5 0 3 8 9 7 2 0 6 7 8 15 10 0 6 3 9 6 8 0 4 12 5 7 4 0	Nearest Neighbor TSP Solution: 0 1 2 4 3 Total Distance: 25

S.No	Inputs	Outputs
5	6 0 8 5 14 7 10 9 0 9 12 4 8 13 10 0 3 6 2 4 5 15 0 11 6 7 6 2 10 0 9 10 12 8 6 14 0	Nearest Neighbor TSP Solution: 0 2 5 3 1 4 Total Distance: 29
6	6 0 5 8 12 15 7 6 0 6 10 8 13 4 15 0 9 6 11 14 7 9 0 12 4 9 11 10 13 0 6 3 14 12 5 4 0	Nearest Neighbor TSP Solution: 0 1 2 4 5 3 Total Distance: 42

White List:

Black List:

Question 4:

Implement the knapsack function to find the maximum possible value that can be obtained by filling a knapsack of capacity W. However, there is a twist in this dilemma. For each item, you can either choose to include it in the knapsack or exclude it. However, once an item is included, you can't exclude it later. Once included, it stays in the knapsack. Your task is to modify the knapsack function to handle this dilemma and return the maximum value possible considering the given constraints.

Input Format:

- The first line contains two integers n and W, where n is the number of items and W is the knapsack capacity.
- The next two lines contain n space-separated integers each. The first line represents the weights of the items, and the second line represents their corresponding values.

Output Format:

- Output a single integer, the maximum value that can be obtained considering the given constraints.

Title for Question 4: Basic 0/1 Knapsack

Solution:

```
#include <iostream>
#include <vector>

using namespace std;

int knapsack(int W, const vector<int>& weights, const vector<int>& values
    int n = weights.size();
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

    for (int i = 1; i <= n; ++i) {
        for (int w = 1; w <= W; ++w) {
            if (weights[i - 1] <= w) {
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w
            } else {
```

```
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][W];
}

int main() {
    int n, W;
    cin >> n; // Number of items
    cin >> W; // Knapsack capacity

    vector<int> weights(n);
    vector<int> values(n);

    // Input weights and values
    for (int i = 0; i < n; ++i) {
        cin >> weights[i];
    }

    for (int i = 0; i < n; ++i) {
        cin >> values[i];
    }

    cout << "Maximum value in the knapsack: " << knapsack(W, weights, val
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	3 5 2 1 3 3 2 1	Maximum value in the knapsack: 5
2	4 10 5 4 6 3 10 40 30 50	Maximum value in the knapsack: 90
3	2 7 3 4 2 1	Maximum value in the knapsack: 3
4	5 12 2 5 8 3 7 3 8 4 1 6	Maximum value in the knapsack: 14
5	3 6 2 4 3 1 3 5	Maximum value in the knapsack: 6
6	1 5 2 7	Maximum value in the knapsack: 7

White List:

Black List:

Question 5:

You are a treasure hunter who has just discovered a hidden cave filled with ancient artifacts and treasures. Each item has a weight and a value associated with it. However, your backpack can only carry a limited amount of weight. Given that you can take fractions of the treasures to maximize the total value you extract, your task is to determine the maximum value you can carry out of the cave.

Input Format:

- The first line contains an integer n (1 ? n ? 1000), representing the number of items.
- The second line contains an integer W (1 ? W ? 10000), representing the knapsack capacity.
- The next n lines contain two integers each: weights[i] (1 ? weights[i] ? 100) and values[i] (1 ? values[i] ? 100), representing the weight and value of the i-th item.

Output Format:

- Print a single line with the message "Maximum value in the knapsack (fractional): ", followed by the maximum total value you can carry out, rounded to a precision of up to two decimal places.

Title for Question 5: Fractional Knapsack

Solution:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct Item {
    int weight;
    int value;
    double valuePerWeight;  // Value per unit weight
};

bool compareItems(const Item& a, const Item& b) {
    return a.valuePerWeight > b.valuePerWeight;
}

double fractionalKnapsack(int W, vector<int>& weights, vector<int>& value
    int n = weights.size();
    vector<Item> items(n);

    for (int i = 0; i < n; ++i) {
        items[i].weight = weights[i];
        items[i].value = values[i];
        items[i].valuePerWeight = static_cast<double>(values[i]) / weight
    }

    sort(items.begin(), items.end(), compareItems);

    double maxTotalValue = 0.0;
    int currentWeight = 0;

    for (const Item& item : items) {
        if (currentWeight + item.weight <= W) {
            // Take the whole item
            maxTotalValue += item.value;
            currentWeight += item.weight;
        } else {
            // Take a fraction of the item
            double remainingWeight = W - currentWeight;
            maxTotalValue += item.valuePerWeight * remainingWeight;
```

```
            break;
        }
    }

    return maxTotalValue;
}

int main() {
    int n, W;
    cin >> n; // Number of items
    cin >> W; // Knapsack capacity

    vector<int> weights(n);
    vector<int> values(n);

    // Input weights and values
    for (int i = 0; i < n; ++i) {
        cin >> weights[i];
    }

    for (int i = 0; i < n; ++i) {
        cin >> values[i];
    }

    cout << "Maximum value in the knapsack (fractional): " << fractionalK
    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	3 10 5 30 2 20 3 25	Maximum value in the knapsack (fractional): 45.3
2	4 15 7 40 5 30 4 25 3 20	Maximum value in the knapsack (fractional): 10
3	2 8 4 25 6 35	Maximum value in the knapsack (fractional): 11.6
4	5 20 3 50 5 60 6 70 8 30 2 40	Maximum value in the knapsack (fractional): 140.96
5	1 5 3 15	Maximum value in the knapsack (fractional): 15
6	6 12 4 8 2 6 6 10 5 9 3 7 1 5	Maximum value in the knapsack (fractional): 15

White List:

Black List:

Question 6:

Assume you are developing a software tool for a digital library to find similarities in text content. The tool will be used to compare two strings (which could be sentences, paragraphs, etc.) and determine the length of the longest common subsequence. This can be used for applications like document comparison, plagiarism detection, or content analysis.

Input Format:

- The first line of input contains the first string.
- The second line of input contains the second string.
- Each string can consist of characters (including spaces) and is terminated by a newline character.

Output Format:

- The output is a single line that starts with "Length of Longest Common Subsequence: " followed by the length of the longest common subsequence.

Title for Question 6: Longest Common Subsequence

Solution:

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int longestCommonSubsequence(const string &str1, const string &str2) {
    int m = str1.size();
    int n = str2.size();

    vector<vector<int> > dp(m + 1, vector<int>(n + 1));

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 0;
            } else if (str1[i - 1] == str2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    return dp[m][n];
}

int main() {
    string str1, str2;

 //   cout << "Enter first sequence: ";
    cin >> str1;

  //  cout << "Enter second sequence: ";
    cin >> str2;

    cout << "Length of Longest Common Subsequence: " ;
      cout << longestCommonSubsequence(str1, str2);
      cout<< endl;

    return 0;
}
```

TestCases:

S.No	Inputs	Outputs
1	ABCBDAB BDCABC	Length of Longest Common Subsequence: 4
2	HELLO WORLD	Length of Longest Common Subsequence: 1
3	ALGORITHM LOGARITHM	Length of Longest Common Subsequence: 7
4	12345 54321	Length of Longest Common Subsequence: 1
5	AAABBB BBBAAA	Length of Longest Common Subsequence: 3
6	EMPTY STRING	Length of Longest Common Subsequence: 1

White List:

Black List:
