ADA LAB 4

0/1 Knapsack Problem:

The Knapsack problem is an example of the combinational optimization problem. This problem is also commonly known as the “Rucksack Problem“. The name of the problem is defined from the maximization problem as mentioned below:

Given a bag with maximum weight capacity of W and a set of items, each having a weight and a value associated with it. Decide the number of each item to take in a collection such that the total weight is less than the capacity and the total value is maximized.

Using Dynamic Programming:

```
#include <bits/stdc++.h>
using namespace std;

// Returns the value of maximum profit
int knapSackRec(int W, int wt[], int val[], int index, int** dp)
{
    // base condition
    if (index < 0)
        return 0;
    if (dp[index][W] != -1)
        return dp[index][W];

    if (wt[index] > W) {

        // Store the value of function call
        // stack in table before return
        dp[index][W] = knapSackRec(W, wt, val, index - 1, dp);
        return dp[index][W];
    }
    else {
        // Store value in a table before return
        dp[index][W] = max(val[index]
                           + knapSackRec(W - wt[index], wt, val,
                                         index - 1, dp),
                           knapSackRec(W, wt, val, index - 1, dp));

        // Return value of table after storing
        return dp[index][W];
    }
```

```
}

int knapSack(int W, int wt[], int val[], int n)
{
    // double pointer to declare the
    // table dynamically
    int** dp;
    dp = new int*[n];

    // loop to create the table dynamically
    for (int i = 0; i < n; i++)
        dp[i] = new int[W + 1];

    // loop to initially filled the
    // table with -1
    for (int i = 0; i < n; i++)
        for (int j = 0; j < W + 1; j++)
            dp[i][j] = -1;
    return knapSackRec(W, wt, val, n - 1, dp);
}

// Driver Code
int main()
{
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);
    cout << knapSack(W, weight, profit, n);
    return 0;
}
```

Single Source Shortest Path Problem:

It is a shortest path problem where the shortest path from a given source vertex to all other remaining vertices is computed.
Dijkstra's Algorithm and Bellman Ford Algorithm are the famous algorithms used for solving single-source shortest path problem.

Using Dynamic Programming:

```
#include <iostream>

#define MAX_NODES 100
#define INF 99999 // A large value to represent infinity

using namespace std;

int graph[MAX_NODES][MAX_NODES];
int dist[MAX_NODES];
bool visited[MAX_NODES];
int n, m;

int findMinDistance() {
    int min = INF, min_index;

    for (int v = 0; v < n; v++)
        if (!visited[v] && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

void dijkstra(int src) {
    for (int i = 0; i < n; i++)
        dist[i] = INF, visited[i] = false;

    dist[src] = 0;

    for (int count = 0; count < n - 1; count++) {
        int u = findMinDistance();
        visited[u] = true;

        for (int v = 0; v < n; v++)
            if (!visited[v] && graph[u][v] && dist[u] != INF
                && dist[u] + graph[u][v] < dist[v])
```

```
                dist[v] = dist[u] + graph[u][v];
    }

    cout << "Vertex\tDistance from Source" << endl;
    for (int i = 0; i < n; i++)
        cout << i << "\t" << dist[i] << endl;
}

int main() {
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;

    // Initialize the graph matrix with 0s
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            graph[i][j] = 0;

    cout << "Enter edges (u v w) for each edge:" << endl;
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        graph[u][v] = w;
        graph[v][u] = w; // For undirected graph, add this line
    }

    int source;
    cout << "Enter source vertex: ";
    cin >> source;

    dijkstra(source);

    return 0;
}
```