//1 . Write a program to implement Fractional knapsack using Greedy algorithm and also find the maximum profit

#include <bits/stdc++.h>

using *namespace* std;

// Structure for an item which stores weight and corresponding value of Item

*struct* Item {

*int* value, weight;

};

// Comparison function to sort Item according to val/weight ratio

*bool* cmp(*struct* Item *a*, *struct* Item *b*)

{

*double* r1 = (*double*)*a*.value / (*double*)*a*.weight;

*double* r2 = (*double*)*b*.value / (*double*)*b*.weight;

    return r1 > r2;

}

// Main greedy function to solve problem

*double* fractionalKnapsack(*int* *W*, *struct* Item *arr*[], *int* *N*)

{

    // Sorting Item on basis of ratio

    sort(*arr*, *arr* + *N*, cmp);

*double* finalvalue = 0.0;

    // Looping through all items

    for (*int* i = 0; i < *N*; i++) {

        // If adding Item won't overflow,

        // add it completely

        if (*arr*[i].weight <= *W*) {

*W* -= *arr*[i].weight;

            finalvalue += *arr*[i].value;

        }

        // If we can't add current Item,

        // add fractional part of it

        else {

            finalvalue += *arr*[i].value \* ((*double*)*W* / (*double*)*arr*[i].weight);

            break;

        }

    }

    // Returning final value

    return finalvalue;

}

// Driver code

*int* main()

{

*int* W ;

    cout<<"Enter weight of knapsack: "<<endl;

    cin>>W;

*int* N ;

    cout<<"Enter number of items: "<<endl;

    cin>>N;

    Item arr[N];

    for(*int* i=0;i<N;i++)

    {

        cout<<"Enter value and weight of item "<<i+1<<endl;

        cin>>arr[i].value>>arr[i].weight;

    }

    cout << "Maximum value we can obtain = "<< fractionalKnapsack(W, arr, N);

    return 0;

}

// // 2.Write a program to implement 0/1 knapsack using dynamic programming and also find the maximum profit

// /\* A Naive recursive implementation of

// 0-1 Knapsack problem \*/

#include <bits/stdc++.h>

using *namespace* std;

// A utility function that returns

// maximum of two integers

*int* max(*int* *a*, *int* *b*)

{

    return (*a* > *b*) ? *a* : *b*;

}

// Returns the maximum value that

// can be put in a knapsack of capacity W

*int* knapSack(*int* *W*, *int* *wt*[], *int* *val*[], *int* *n*)

{

    // Base Case

    if (*n* == 0 || *W* == 0)

        return 0;

    // If weight of the nth item is more

    // than Knapsack capacity W, then

    // this item cannot be included

    // in the optimal solution

    if (*wt*[*n* - 1] > *W*)

        return knapSack(*W*, *wt*, *val*, *n* - 1);

    // Return the maximum of two cases:

    // (1) nth item included

    // (2) not included

    else

        return max(*val*[*n* - 1] + knapSack(*W* - *wt*[*n* - 1], *wt*, *val*, *n* - 1),knapSack(*W*, *wt*, *val*, *n* - 1));

}

// Driver code

*int* main()

{

*int* W ;

    cout<<"enter weight of knapsak: "<<endl;

    cin>>W;

*int* n;

    cout<<"enter number of items: "<<endl;

    cin>>n;

*int* val[n];

*int* wt[n];

    for(*int* i=0;i<n;i++)

    {

        cout<<"enter value and weight of item "<<i+1<<endl;

        cin>>val[i]>>wt[i];

    }

    cout << knapSack(W, wt, val, n);

    return 0;

}

// 3. Write a program to implement Bellman-Ford Algorithm using Dynamic Programming and verify the time complexity

//  A C++ program for Bellman-Ford's single source

//  shortest path algorithm.

#include <bits/stdc++.h>

using *namespace* std;

// a structure to represent a weighted edge in graph

*struct* Edge

{

*int* src, dest, weight;

};

// a structure to represent a connected, directed and

// weighted graph

*struct* Graph

{

    // V-> Number of vertices, E-> Number of edges

*int* V, E;

    // graph is represented as an array of edges.

*struct* Edge \*edge;

};

// Creates a graph with V vertices and E edges

*struct* Graph \*createGraph(*int* *V*, *int* *E*)

{

*struct* Graph \*graph = new Graph;

    graph->V = V;

    graph->E = E;

    graph->edge = new Edge[E];

    return graph;

}

// A utility function used to print the solution

*void* printArr(*int* *dist*[], *int* *n*)

{

    printf("Vertex Distance from Source\n");

    for (*int* i = 0; i < n; ++i)

        printf("%d \t\t %d\n", i, dist[i]);

}

// The main function that finds shortest distances from src

// to all other vertices using Bellman-Ford algorithm. The

// function also detects negative weight cycle

*void* BellmanFord(*struct* Graph \**graph*, *int* *src*)

{

*int* V = graph->V;

*int* E = graph->E;

*int* dist[V];

    // Step 1: Initialize distances from src to all other

    // vertices as INFINITE

    for (*int* i = 0; i < V; i++)

        dist[i] = INT\_MAX;

    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple

    // shortest path from src to any other vertex can have

    // at-most |V| - 1 edges

    for (*int* i = 1; i <= V - 1; i++)

    {

        for (*int* j = 0; j < E; j++)

        {

*int* u = graph->edge[j].src;

*int* v = graph->edge[j].dest;

*int* weight = graph->edge[j].weight;

            if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

                dist[v] = dist[u] + weight;

        }

    }

    // Step 3: check for negative-weight cycles. The above

    // step guarantees shortest distances if graph doesn't

    // contain negative weight cycle. If we get a shorter

    // path, then there is a cycle.

    for (*int* i = 0; i < E; i++)

    {

*int* u = graph->edge[i].src;

*int* v = graph->edge[i].dest;

*int* weight = graph->edge[i].weight;

        if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

        {

            printf("Graph contains negative weight cycle");

            return; // If negative cycle is detected, simply

                    // return

        }

    }

    printArr(dist, V);

    return;

}

// Driver's code

*int* main()

{

    /\* Let us create the graph given in above example \*/

*int* V;

*int* E;

    cout << "Enter the number of vertices: ";

    cin >> V;

    cout << "Enter the number of edges: ";

    cin >> E;

*struct* Graph \*graph = createGraph(V, E);

    cout << "Enter the edges in the format: source destination weight" << endl;

    for (*int* i = 0; i < E; i++)

    {

        cout << "enter source , destination and weight of edge " << i + 1 << endl;

        cin >> graph->edge[i].src >> graph->edge[i].dest >> graph->edge[i].weight;

    }

    BellmanFord(graph, 0);

    return 0;

}

//4.Write a recursive program to find the solution of placing n queens on the chessboard so that no two queens attack each other using Backtracking

/\* C++ program to solve N Queen Problem using backtracking \*/

#include <bits/stdc++.h>

#define N 4

using *namespace* std;

/\* A utility function to print solution \*/

*void* printSolution(*int* *board*[N][N])

{

    for (*int* i = 0; i < N; i++) {

        for (*int* j = 0; j < N; j++)

            cout << " " << board[i][j] << " "<<endl;

    }

}

/\* A utility function to check if a queen can be placed on board[row][col]. Note that this

function is called when "col" queens are already placed in columns from 0 to col -1.

So we need to check only left side for attacking queens \*/

*bool* isSafe(*int* *board*[N][N], *int* *row*, *int* *col*)

{

*int* i, j;

    /\* Check this row on left side \*/

    for (i = 0; i < col; i++)

        if (board[row][i])

            return false;

    /\* Check upper diagonal on left side \*/

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

        if (board[i][j])

            return false;

    /\* Check lower diagonal on left side \*/

    for (i = row, j = col; j >= 0 && i < N; i++, j--)

        if (board[i][j])

            return false;

    return true;

}

/\* A recursive utility function to solve N Queen problem \*/

*bool* solveNQUtil(*int* *board*[N][N], *int* *col*)

{

    /\* base case: If all queens are placed then return true \*/

    if (col >= N)

        return true;

    /\* Consider this column and try placing this queen in all rows one by one \*/

    for (*int* i = 0; i < N; i++) {

        /\* Check if the queen can be placed on board[i][col] \*/

        if (isSafe(board, i, col)) {

            /\* Place this queen in board[i][col] \*/

            board[i][col] = 1;

            /\* recur to place rest of the queens \*/

            if (solveNQUtil(board, col + 1))

                return true;

            /\* If placing queen in board[i][col] doesn't lead to a solution, then remove queen from board[i][col] \*/

            board[i][col] = 0; // BACKTRACK

        }

    }

    /\* If the queen cannot be placed in any row in this column col then return false \*/

    return false;

}

/\* This function solves the N Queen problem using Backtracking. It mainly uses solveNQUtil() to

solve the problem. It returns false if queens cannot be placed, otherwise, return true and

prints placement of queens in the form of 1s. Please note that there may be more than one

solutions, this function prints one of the feasible solutions.\*/

*bool* solveNQ()

{

*int* board[N][N] = { { 0, 0, 0, 0 },

                        { 0, 0, 0, 0 },

                        { 0, 0, 0, 0 },

                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {

        cout << "Solution does not exist";

        return false;

    }

    printSolution(board);

    return true;

}

// driver program to test above function

*int* main()

{

    solveNQ();

    return 0;

}

// CPP program to implement traveling salesman

// problem using naive approach.

#include <bits/stdc++.h>

using *namespace* std;

#define V 4

// implementation of traveling Salesman Problem

*int* travllingSalesmanProblem(*int* *graph*[][V], *int* *s*)

{

    // store all vertex apart from source vertex

    vector<*int*> vertex;

    for (*int* i = 0; i < V; i++)

        if (i != *s*)

            vertex.push\_back(i);

    // store minimum weight Hamiltonian Cycle.

*int* min\_path = INT\_MAX;

    do {

        // store current Path weight(cost)

*int* current\_pathweight = 0;

        // compute current path weight

*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*];

        // update minimum

        min\_path = min(min\_path, current\_pathweight);

    } while (

        next\_permutation(vertex.begin(), vertex.end()));

    return min\_path;

}

// Driver Code

*int* main()

{

    // int n;

    // cout<<"No of nodes:";

    // cin<<n;

    // int graph[n][n];

    // for (int i = 1; i <= n; i++)

    // {

    // for (int j; j < n; j++)

    //     {

    //  cin >> graph[i][j];

    //     }

    // }

    // matrix representation of graph

*int* graph[][V] = { { 0, 10, 15, 20 },

                    { 10, 0, 35, 25 },

                    { 15, 35, 0, 30 },

                    { 20, 25, 30, 0 } };

*int* s = 0;

    cout << travllingSalesmanProblem(graph, s) << endl;

    return 0;

}