```
// Fractional Knapsack DAA1
// Write a program to implement Fractional knapsack using Greedy algorithm and also find
// the maximum profit

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

struct item
{
   int value, weight;
};

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;
}

int main()
{

   int n, c;
   double fvalue = 0;
   cout << "Enter the capacity of knapsack - " << endl;
   cin >> c;

   cout << "How many objects do you have? " << endl;
   cin >> n;

   item arr[n];
```

```
    for (int i = 0; i < n; i++)
    {
        cout << "Enter the value and weight of item " << i + 1 << ":" << endl;
        cin >> arr[i].value >> arr[i].weight;
    }

    // item arr[]={{40,80},{10,10},{50,40},{30,20},{60,90}};
    sort(arr, arr + n, cmp);

    for (int i = 0; i < n; i++)
    {
        if (arr[i].weight <= c)
        {
            c -= arr[i].weight;
            fvalue += arr[i].value;
        }
        else
        {
            fvalue += arr[i].value * ((double)c / (double)arr[i].weight);
            break;
        }
    }
    cout << "The maximum value is : " << fvalue << endl;

    return 0;
}

/*
Output-

Enter the value of C
```

```
50
how many objects do u have?
3
Enter the value and weight of item 1:
60
10
Enter the value and weight of item 2:
100
20
Enter the value and weight of item 3:
120
30
The maximum value is : 240
*/
```

```
// C++ program to solve 0/1 knapsack program using DP Iterative 0/1 KnapSack

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

int main()
{
    int n, w;
    cin >> n >> w;

    vector<int> profit(n + 1), weight(n + 1);

    for (int i = 1; i <= n; i++)
    {
        cin >> profit[i];
    }
    for (int i = 1; i <= n; i++)
    {
        cin >> weight[i];
    }

    int dp[n + 1][w + 1];
    for (int i = 0; i <= n; i++)
    {
        for (int j = 0; j <= w; j++)
        {
            dp[i][j] = 0;
        }
    }

    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= w; j++)
        {
            if (j - weight[i] >= 0)
            {
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + profit[i]);
            }
            else
            {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }

    cout << "Ans : " << dp[n][w] << "\n";
    for (int i = 0; i <= n; i++)
    {
        for (int j = 0; j <= w; j++)
        {
            cout << dp[i][j] << " ";
        }
        cout << '\n';
    }

    return 0;
}

/*

*Output -
4 8
```

```
1 2 5 6
2 3 4 5
Ans : 8
0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1 1
0 0 1 2 2 3 3 3 3
0 0 1 2 5 5 6 7 7
0 0 1 2 5 6 6 7 8
*/
```

```
// Bellman_Ford

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

// a structure to represent a weighted edge in graph
struct Edge
{
   int u; // source
   int v; // destination
   int wt;
};

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

// 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 u)
{
    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[u] = 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 s = graph->edge[j].u;
            int d = graph->edge[j].v;
            int w = graph->edge[j].wt;
            if ((dist[s] != INT_MAX) && ((dist[s] + w) < dist[d]))
            {
                dist[d] = dist[s] + w;
            }
        }
    }

    // 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 s = graph->edge[i].u;
        int d = graph->edge[i].v;
        int w = graph->edge[i].wt;
        if ((dist[s] != INT_MAX) && (dist[s] + w < dist[d]))
        {
            cout << "Graph contains negative weight cycle" << endl;
            return; // If negative cycle is detected, simply
        }
    }
```

```
    printarr(dist, V);

    return;

}


int main()

{

    int vertices, edges, source;

    cout << "Enter the number of Vertices: " << endl;

    cin >> vertices;

    cout << "Enter the number of Edges: " << endl;

    cin >> edges;

    struct Graph *graph = createGraph(vertices, edges);

    for (int i = 0; i < edges; i++)

    {

        cout << "Enter Edge"

             << " " << i + 1 << " "

             << "properties Source, destination, weight respectively" << endl;

        cin >> graph->edge[i].u;

        cin >> graph->edge[i].v;

        cin >> graph->edge[i].wt;

    }

    BellmanFord(graph, 0);

    return 0;

}


// Test case 1

/*

Enter the number of Vertices:

5

Enter the number of Edges:

8
```

```
Enter Edge 1 properties Source, destination, weight respectively
 0 1 -1
Enter Edge 2 properties Source, destination, weight respectively
 0 2 4
Enter Edge 3 properties Source, destination, weight respectively
 1 2 3
Enter Edge 4 properties Source, destination, weight respectively
 1 3 2
Enter Edge 5 properties Source, destination, weight respectively
 1 4 2
Enter Edge 6 properties Source, destination, weight respectively
 3 2 5
Enter Edge 7 properties Source, destination, weight respectively
 3 1 1
Enter Edge 8 properties Source, destination, weight respectively
 4 3 -3
Vertex   Distance from Source
0          0
1          -1
2          2
3          -2
4          1
*/
```

```
// N Queens



#include <bits/stdc++.h>
#define N 4
using namespace std;

int board[N][N];
int tb = 1; // total_board = tb

void displayboard()
{
   cout << "Board: " << tb
        << endl;
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
      {
         cout << board[i][j] << " ";
      }
      cout << endl;
   }
   cout << endl;
   tb++;
}

bool isSafe(int row, int col)
{
   for (int i = 0; i < row; i++) // for columns
   {
      if (board[i][col] == 1)
```

```
        {
            return false;
        }
    }
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) // for upper diagonal
    {
        if (board[i][j] == 1)
        {
            return false;
        }
    }

    for (int i = row - 1, j = col + 1; i >= 0 && j < N; i--, j++) // lower diagonal
    {
        if (board[i][j] == 1)
        {
            return false;
        }
    }

    return true;
}

void nQueens(int row) // backtrack
{
    if (row == N)
    {
        displayboard();
        return;
    }
    for (int i = 0; i < N; i++)
    {
        board[row][i] = 1;
        if (isSafe(row, i))
        {
            nQueens(row + 1);
        }
        board[row][i] = 0;
    }
}

int main()
{
    nQueens(0);
    return 0;
}

/*
Board: 1
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

Board: 2
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

*/
```

```
//  TSP

#include <bits/stdc++.h>
using namespace std;
const int N = 100;


int final_path[N+1];
bool visited[N];
int final_res = INT_MAX;

void copyToFinal(int curr_path[],int n)
{
    for (int i=0; i<n; i++)
        final_path[i] = curr_path[i];
    final_path[n] = curr_path[0];
}

int firstMin(int adj[N][N], int i,int n)
{
    int min = INT_MAX;
    for (int k=0; k<n; k++)
        if (adj[i][k]<min && i != k)
            min = adj[i][k];
    return min;
}


int secondMin(int adj[N][N], int i,int n)
{
    int first = INT_MAX, second = INT_MAX;
    for (int j=0; j<n; j++)
    {
        if (i == j)
            continue;

        if (adj[i][j] <= first)
        {
            second = first;
            first = adj[i][j];
        }
        else if (adj[i][j] <= second &&
                adj[i][j] != first)
            second = adj[i][j];
    }
    return second;
}



void TSPRec(int adj[N][N], int curr_bound, int curr_weight,
            int level, int curr_path[],int n)
{

    if (level==n)
    {

        if (adj[curr_path[level-1]][curr_path[0]] != 0)
        {

            int curr_res = curr_weight +
                    adj[curr_path[level-1]][curr_path[0]];
    if (curr_res < final_res)
    {
      copyToFinal(curr_path,n);
      final_res = curr_res;
    }
  }
  return;
}


for (int i=0; i<n; i++)
{

  if (adj[curr_path[level-1]][i] != 0 &&
    visited[i] == false)
  {
    int temp = curr_bound;
    curr_weight += adj[curr_path[level-1]][i];


    if (level==1)
     curr_bound -= ((firstMin(adj, curr_path[level-1],n) +
            firstMin(adj, i, n))/2);
    else
     curr_bound -= ((secondMin(adj, curr_path[level-1],n) +
            firstMin(adj, i, n))/2);


    if (curr_bound + curr_weight < final_res)
    {
            curr_path[level] = i;
            visited[i] = true;


            TSPRec(adj, curr_bound, curr_weight, level+1,
                curr_path,n);
          }


          curr_weight -= adj[curr_path[level-1]][i];
          curr_bound = temp;


          memset(visited, false, sizeof(visited));
          for (int j=0; j<=level-1; j++)
            visited[curr_path[j]] = true;
        }
      }
    }


    void TSP(int adj[N][N],int n)
    {
      int curr_path[N+1];


      int curr_bound = 0;
      memset(curr_path, -1, sizeof(curr_path));
      memset(visited, 0, sizeof(curr_path));
    for (int i=0; i<n; i++)
        curr_bound += (firstMin(adj, i,n) +
                       secondMin(adj, i,n));



    curr_bound = (curr_bound&1)? curr_bound/2 + 1 :
                                 curr_bound/2;



    visited[0] = true;
    curr_path[0] = 0;



    TSPRec(adj, curr_bound, 0, 1, curr_path,n);
}

// Driver code
int main()
{
    //Adjacency matrix for the given graph
    int n;
    int adj[N][N];

    cout<<"Enter the number of vetices"<<endl;
    cin>>n;

    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            cout<<"Enter the from "<<i+1<<"to "<<j+1<<endl;
            cin>>adj[i][j];
        }
    }

    TSP(adj,n);

    printf("Minimum cost : %d\n", final_res);
    printf("Path Taken : ");
    for (int i=0; i<=n; i++)
        printf("%d ", final_path[i]);

    cout<<endl;
    return 0;
}

/*
Enter the number of vetices
4
Enter the from 1to 1
0
Enter the from 1to 2
10
Enter the from 1to 3
15
Enter the from 1to 4
20
Enter the from 2to 1
10
Enter the from 2to 2
0
Enter the from 2to 3
35
Enter the from 2to 4
```

```
25
Enter the from 3to 1
15
Enter the from 3to 2
35
Enter the from 3to 3
0
Enter the from 3to 4
30
Enter the from 4to 1
20
Enter the from 4to 2
25
Enter the from 4to 3
30
Enter the from 4to 4
0
Minimum cost : 80
Path Taken : 0 1 3 2 0
*/
```