**Task:-Floyd-Warshall Algorithm**

**Objective :-** *To find shortest distances between every pair of vertices in a given edge weighted directed Graph.*

**Introduction:-**

*Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).*

**Algorithm:-**

*Algorithm of Floyd washall –*

*for (int k = 0; k < v; k++)*

*{*

*for (int i = 0; i < v; i++)*

*{*

*for (int j = 0; j < v; j++)*

*{*

*a[i][j] = min(a[i][j], a[i][k] + a[k][j]);*

*}*

*}*

*}*

**Procedure:-**

* *We initialize the solution matrix same as the input graph matrix as a first step.*
* *Then we update the solution matrix by considering all vertices as an intermediate vertex.*
* *The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.*
* *When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.*

**Code:-**

#include <bits/stdc++.h>

using namespace std;

int main()

{

    int v, e;

    cin >> v >> e;

    int a[v][v];

    for (int i = 0; i < v; i++)

    {

        for (int j = 0; j < v; j++)

        {if (i==j)

        {a[i][j]=0;

            /\* code \*/

        }

            else{a[i][j] = INT16\_MAX;}

            /\* code \*/

        }

        /\* code \*/

    }

    for (int i = 0; i < e; i++)

    {

        int x, y, w;

        cin >> x >> y >> w;

        a[x][y] = w;

        /\* code \*/

    }

    for (int k = 0; k < v; k++)

    {

        for (int i = 0; i < v; i++)

        {

            for (int j = 0; j < v; j++)

            {

                a[i][j] = min(a[i][j], a[i][k] + a[k][j]);

            }

            /\* code \*/

        }

        /\* code \*/

    }

    for (int i = 0; i < v; i++)

    {

        for (int j = 0; j < v; j++)

        {

            if (a[i][j] == INT16\_MAX)

            {

                cout << "INF"

                     << " ";

                /\* code \*/

            }

            else

            {

                cout << a[i][j] << " ";

            }

            /\* code \*/

        }

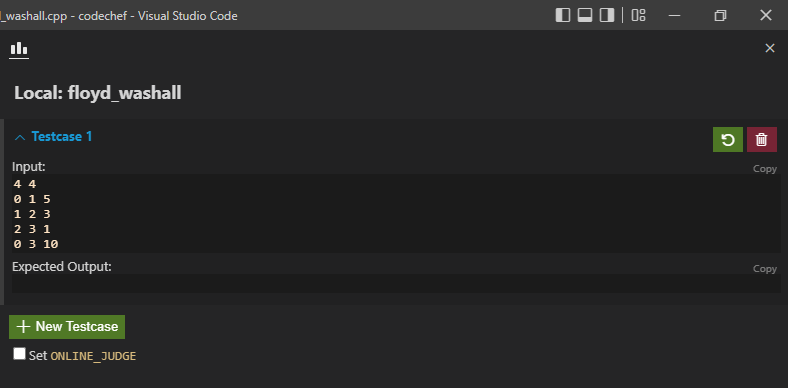
cout << endl;

        /\* code \*/

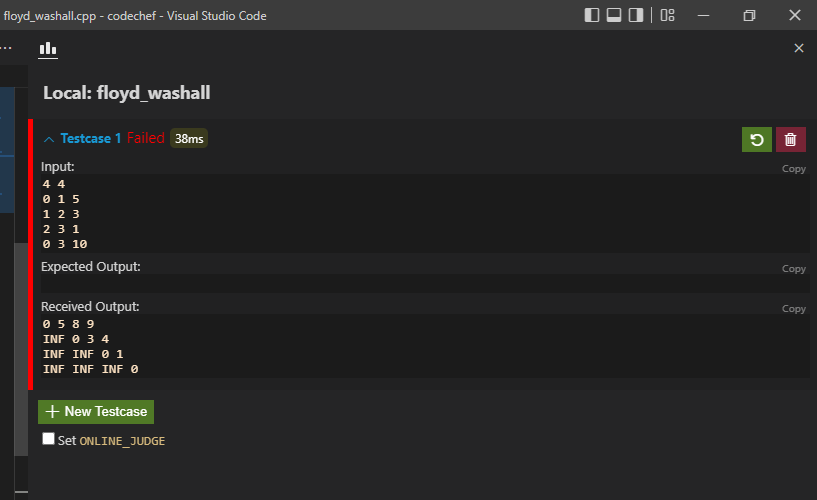
    }

}

**Input:-**



**Output:-**



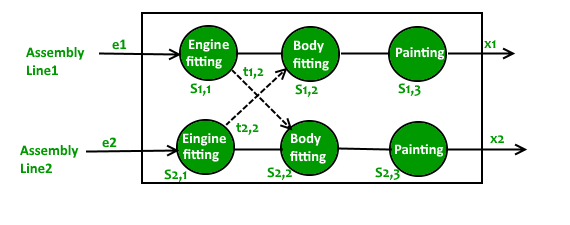
**Conclusion –**

*Time Complexity : O(n3)*

**Task:-Assembly line Algorithm**

**Objective :** *To find minimum time to pass through the two assembly lines.*

**Introduction :** *A car factory has two assembly lines, each with n stations. A station is denoted by Si,j where i is either 1 or 2 and indicates the assembly line the station is on, and j indicates the number of the station. The time taken per station is denoted by ai,j. Each station is dedicated to some sort of work like engine fitting, body fitting, painting, and so on. So, a car chassis must pass through each of the n stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station Si,j, it will continue to station Si,j+1unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line i at station j – 1 to station j on the other line takes time ti,j. Each assembly line takes an entry time ei and exit time xi which may be different for the two lines.*



**Algorithm:-**

*for (i = 1; i < NUM\_STATION; ++i)*

*{*

*T1[i] = min(T1[i - 1] + a[0][i],*

*T2[i - 1] + t[1][i] + a[0][i]);*

*T2[i] = min(T2[i - 1] + a[1][i],*

*T1[i - 1] + t[0][i] + a[1][i]);*

*}*

**Code :-**

#include <bits/stdc++.h>

using namespace std;

int main()

{

    int n;

    cin >> n;

    int a[2][n];

    int t[2][n];

    for (int i = 0; i < 2; i++)

    {

        for (int j = 0; j < n; j++)

        {

            cin >> a[i][j];

        }

    }

    for (int i = 0; i < 2; i++)

    {

        for (int j = 0; j < n; j++)

        {

            cin >> t[i][j];

        }

    }

 int e1,e2,x1,x2;

 cin>>e1>>e2>>x1>>x2;

 int result[2][n];

 int t1=e1+a[0][0],t2=e2+a[1][0];

 result[0][0]=t1,result[1][0]=t2;

 for (int i = 1; i < n; i++)

 {result[0][i]=min(t2+t[1][i]+a[0][i],t1+a[0][i]);

 result[1][i]=min(t1+t[0][i]+a[1][i],t2+a[1][i]);

 t1=result[0][i];

 t2=result[1][i];

 }

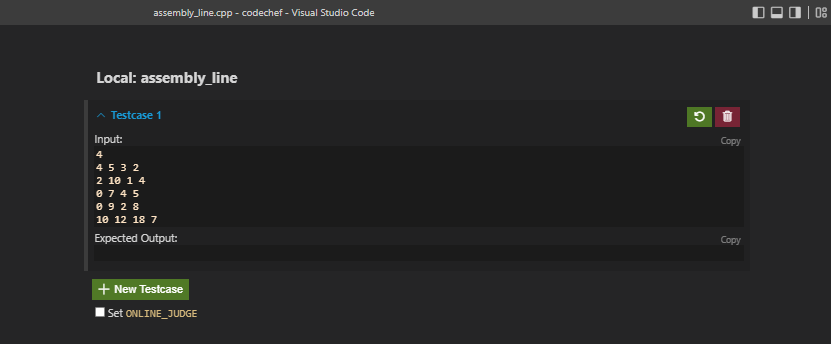
result[0][n-1]+=x1;

result[1][n-1]+=x2;

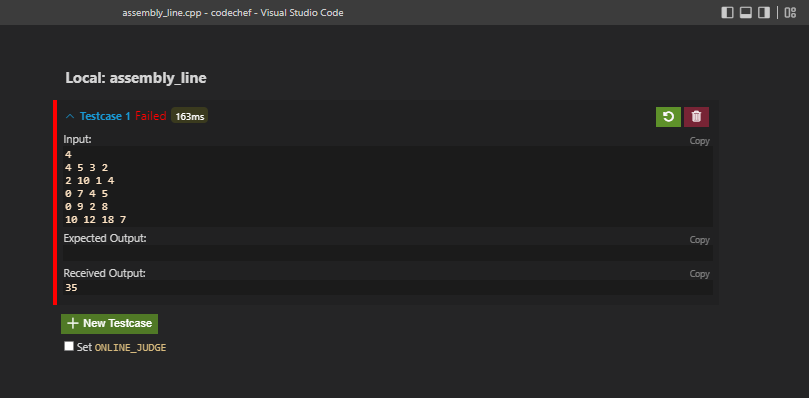
cout<<min(result[0][n-1],result[1][n-1])<<endl;

}

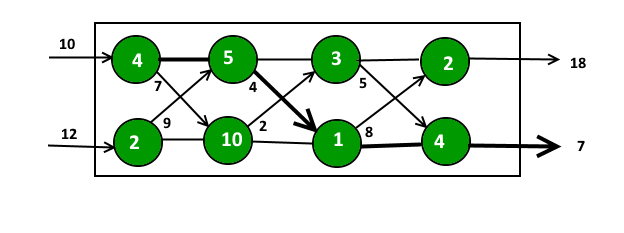
**Input :-**



**Output:-**



**Graph :**



**Conclusion:-**

*Time Complexity : O(n)*

**Task:-Multistage-graph Problem Algorithm**

**Objective :** *To find shortest path from source to destination in multistage graph.*

**Introduction :** *A****Multistage graph****is a directed graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only (In other words there is no edge between vertices of same stage and from a vertex of current stage to previous stage).  
We are given a multistage graph, a source and a destination, we need to find shortest path from source to destination. By convention, we consider source at stage 1 and destination as last stage.*

**Algorithm:-**

*for (int i = 0; i < e; i++)*

*{*

*int x, y, w;*

*cin >> x >> y >> w;*

*a[x][y] = w;*

*if (y==destination)*

*{distance[x]=w;*

*}*

*}*

*for (int i = 0; i < v-2; i++)*

*{for(int j=0;j<v;j++){*

*for(int k=0;k<v;k++){*

*if (a[j][k]!=INT16\_MAX&&distance[k]!=INT16\_MAX)*

*{*

*distance[j]=min(distance[j],distance[k]+a[j][k]);*

*}*

*}*

*} }*

**Code :-**

#include<bits/stdc++.h>

using namespace std;

int main(){

    int v,e;

    cin>>v>>e;

    int a[v][v];

    int distance[v];

    for (int i = 0; i < v; i++)

    {distance[i]=INT16\_MAX;

        /\* code \*/

    }

    int source,destination;

    cin>>source>>destination;

    distance[destination]=0;

    for (int i = 0; i < v; i++)

    {

        for (int j = 0; j < v; j++)

        {

            a[i][j] = INT16\_MAX;

            /\* code \*/

        }

        /\* code \*/

    }

    for (int i = 0; i < e; i++)

    {

        int x, y, w;

        cin >> x >> y >> w;

        a[x][y] = w;

    if (y==destination)

    {distance[x]=w;

    }

    }

    for (int i = 0; i < v-2; i++)

    {for(int j=0;j<v;j++){

        for(int k=0;k<v;k++){

            if (a[j][k]!=INT16\_MAX&&distance[k]!=INT16\_MAX)

            {

                distance[j]=min(distance[j],distance[k]+a[j][k]);

            }

        }

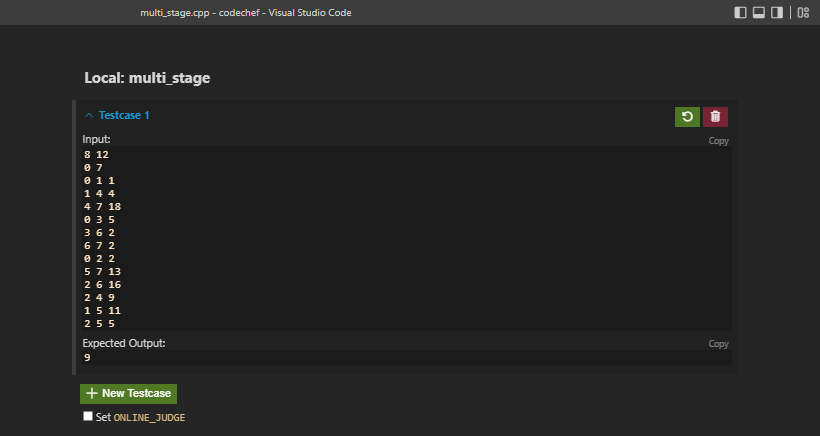
    }

    }

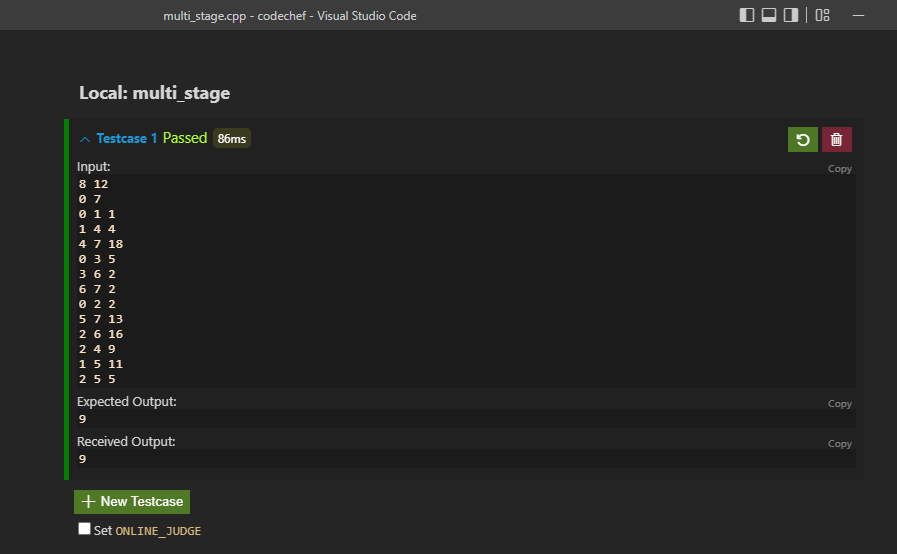
    cout<<distance[source]<<endl;

}

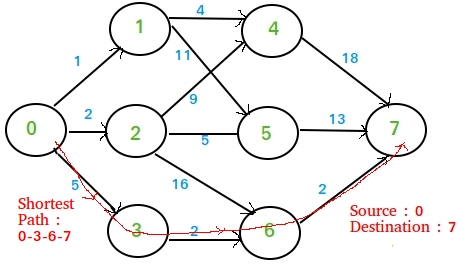
**INPUT :-**



**OUTPUT:-**



**Graph :**



**Time Complexity : O(n3)**

**Task:-Kruskal’s Algorithm**

**Introduction :**

*The Kruskal’s algorithm is a very famous greedy approach to obtain a minimum spanning tree (MST) out of a given connected undirected weighted graph. Given a connected undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A minimum spanning tree for a weighted, connected and undirected graph is a spanning tree with a weight less than or equal to ever other possible spanning tree. In Kruskal we start with an empty tree. Sort all the edges of the given graph in ascending order of weights. Then pick the edge with the minimum cost and check if on adding it to the current MST does it creates a cycle. If it does, ignore it, otherwise add it to the graph. Do this till all the vertices are covered. The graph obtained this way is the MST.*

**PROCEDURE :**

* *First, sort all the edges from low weight to high.*
* *Now, take the edge with the lowest weight and add it to the spanning tree.*
* *If the edge to be added creates a cycle, then reject the edge.*
* *Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.*

**The applications of Kruskal's algorithm are :**

*Kruskal's algorithm can be used to layout electrical wiring among cities. It can be used to lay down LAN connections.*

**Code :**

#include <bits/stdc++.h>

using namespace std;

int find\_set(int x, vector<set<int>> &s)

{

  for (int i = 0; i < s.size(); i++)

  {

    if (s[i].find(x) != s[i].end())

      return i;

  }

  return -1;

}

int main()

{

  int n, e;

  cin >> n >> e;

  vector<pair<int, pair<int, int>>> edges;

  edges.reserve(e);

  for (int i = 0; i < e; i++)

  {

    int x, y, c;

    cin >> x >> y >> c;

    edges.push\_back({c, {x, y}});

  }

  sort(edges.begin(), edges.end());

  vector<set<int>> s(n);

  for (int i = 0; i < n; i++)

    s[i].insert(i);

  int mincost = 0;

  vector<pair<int, int>> MST;

  for (int i = 0; i < n && MST.size() != n - 1; i++)

  {

    int a = find\_set(edges[i].second.first, s);

    int b = find\_set(edges[i].second.second, s);

    if (a == b)

      continue;

    set<int> temp;

    set\_union(s[a].begin(), s[a].end(), s[b].begin(), s[b].end(), inserter(temp, temp.begin()));

    s[a] = temp;

    s[b].clear();

    MST.push\_back({edges[i].second.first, edges[i].second.second});

    mincost += edges[i].first;

  }

  cout << "MST  :" << endl;

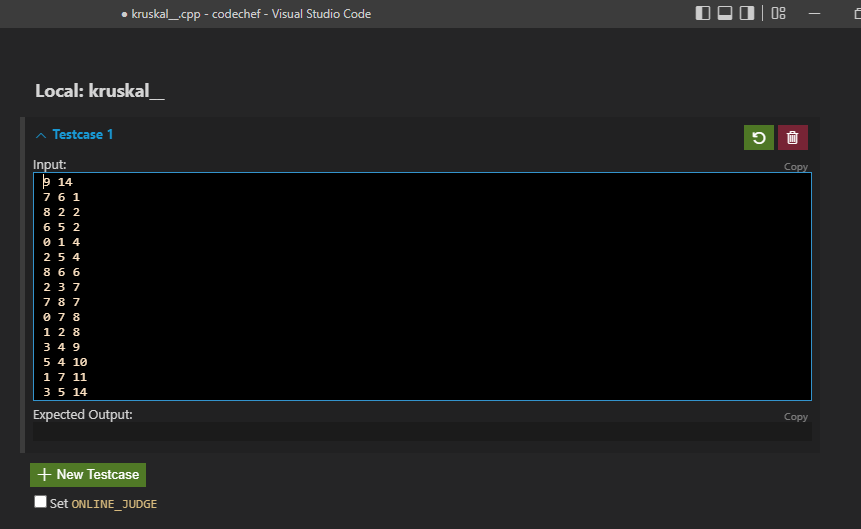
  for (int i = 0; i < MST.size(); i++)

    cout << MST[i].first << " " << MST[i].second << endl;

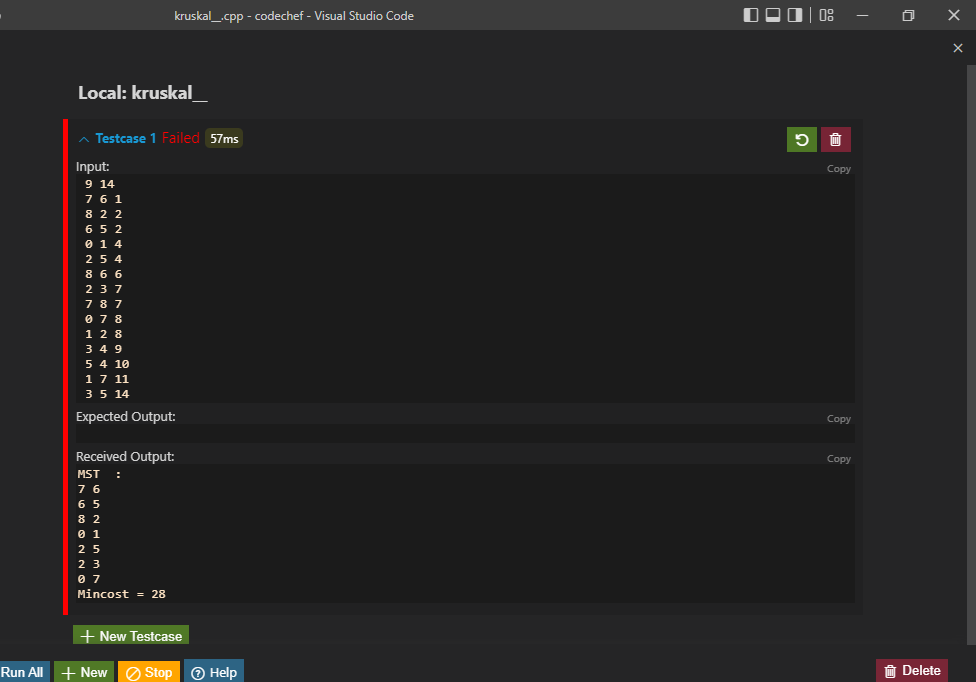
  cout << "Mincost = " << mincost << endl;

}

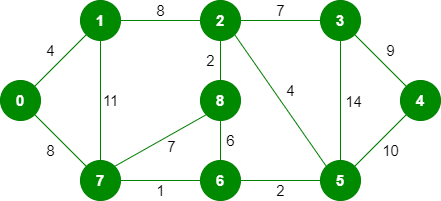
**INPUT:-**



**OUTPUT:-**



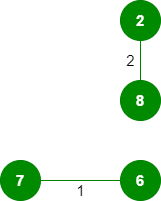
**Graph:-**



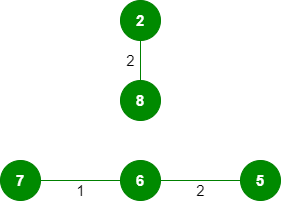
**1.** *Pick edge 7-6:* No cycle is formed, include it. 

Kruskal’s Minimum Spanning Tree Algorithm

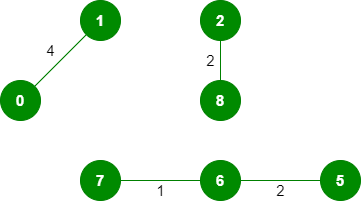
**2.** *Pick edge 8-2:* No cycle is formed, include it. 



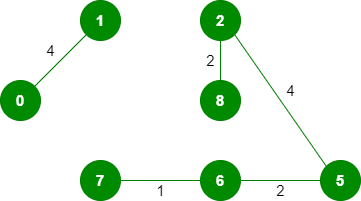
**3.** *Pick edge 6-5:* No cycle is formed, include it. 



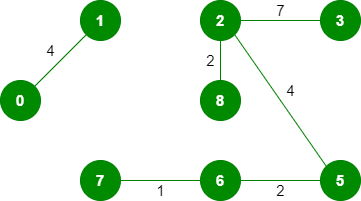
**4.** *Pick edge 0-1:* No cycle is formed, include it. 



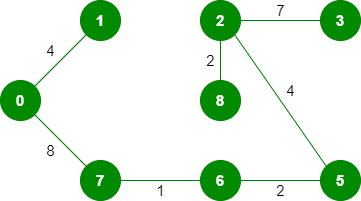
**5.** *Pick edge 2-5:* No cycle is formed, include it. 



**6.***Pick edge 8-6:*Since including this edge results in the cycle, discard it.  
**7.** *Pick edge 2-3:* No cycle is formed, include it. 

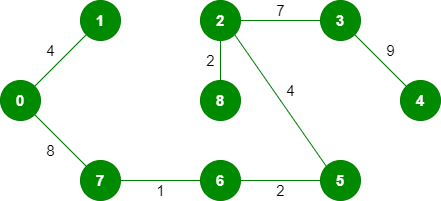


**8.** *Pick edge 7-8:* Since including this edge results in the cycle, discard it.  
**9.** *Pick edge 0-7:* No cycle is formed, include it. 



**10.** *Pick edge 1-2:*Since including this edge results in the cycle, discard it.  
**11.** *Pick edge 3-4:* No cycle is formed,

include it. 



Since the number of edges included equals (V – 1), the algorithm stops here.

**Conclusion:-**

Time Complexity : O(E logV)

E is the number of edges and V is the number of vertices.

We learnt the greedy technique of algorithm design. We learnt how to implement the Kruskal’s algorithm and how to find the MST for a given connected undirected weighted graph.

**Task:-0/1 Knapsack Problem**

**Objective :** *To get the maximum profit in the knapsack of capacity W by filling given weights and profits of n items.*

**Introduction :** *Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don’t pick it (0-1 property).*

**Algorithm:-**

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

*{*

*int i, w;*

*vector<vector<int>> K(n + 1, vector<int>(W + 1));*

*for (i = 0; i <= n; i++)*

*{*

*for (w = 0; w <= W; w++)*

*{*

*if (i == 0 || w == 0)*

*K[i][w] = 0;*

*else if (wt[i - 1] <= w)*

*K[i][w] = max(val[i - 1] +*

*K[i - 1][w - wt[i - 1]],*

*K[i - 1][w]);*

*else*

*K[i][w] = K[i - 1][w];*

*}*

*}*

*return K[n][W];*

*}*

**Code :**

#include <bits/stdc++.h>

using namespace std;

int max(int a, int b)

{

  return (a > b) ? a : b;

}

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

{

  int i, w;

  vector<vector<int>> K(n + 1, vector<int>(W + 1));

  for (i = 0; i <= n; i++)

  {

    for (w = 0; w <= W; w++)

    {

      if (i == 0 || w == 0)

        K[i][w] = 0;

      else if (wt[i - 1] <= w)

        K[i][w] = max(val[i - 1] +

                          K[i - 1][w - wt[i - 1]],

                      K[i - 1][w]);

      else

        K[i][w] = K[i - 1][w];

    }

  }

  return K[n][W];

}

int main()

{int p;

cin>>p;

  int val[p] ;

  int wt[p];

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

{cin>>val[i];

    /\* code \*/

}

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

{cin>>wt[i];

    /\* code \*/

}

  int W ;

  cin>>W;

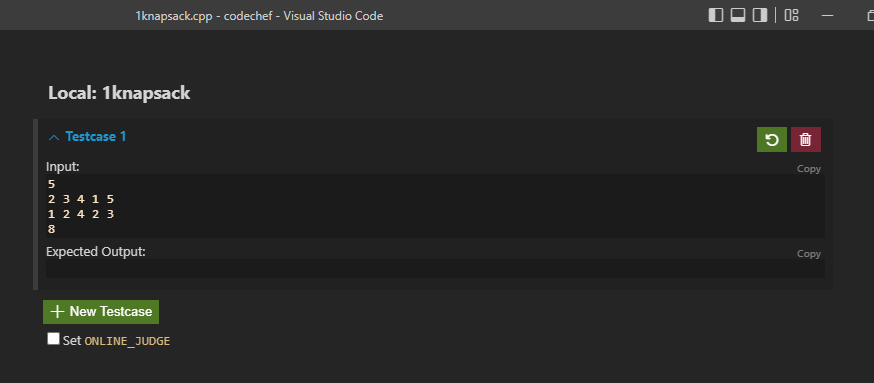
  int n = sizeof(val) / sizeof(val[0]);

  cout << "Max Profit is: " << knapSack(W, wt, val, n);

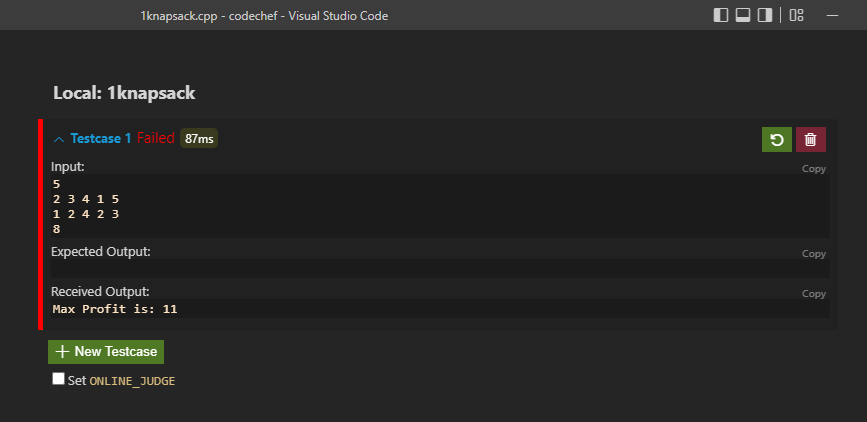
  return 0;

}

**INPUT:-**



**OUTPUT:-**



**Conclusion:-**

*Time Complexity : O(N\*W)*