Solve rod cutting problem using recursion and bottom up approach.

// C++ program to find maximum

// profit from rod of size n

#include <bits/stdc++.h>

using namespace std;

int cutRod(vector<int> &price) {

int n = price.size();

vector<int> dp(price.size()+1, 0);

// Find maximum value for all

// rod of length i.

for (int i=1; i<=n; i++) {

for (int j=1; j<=i; j++) {

dp[i] = max(dp[i], price[j-1]+dp[i-j]);

}

}

return dp[n];

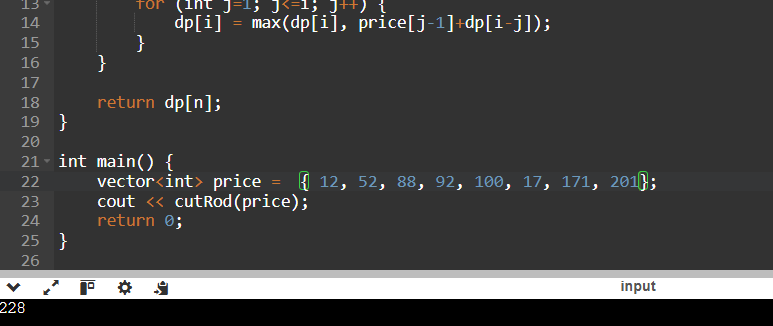
}

int main() {

vector<int> price = { 12, 52, 88, 92, 100, 17, 171, 201};

cout << cutRod(price);

return 0;

}

Implement Bellman-Ford algorithm problem using Dynamic Programming.

// C++ program to find single source shortest path Using Bellman

// -Ford algorithm

#include <iostream>

#include <vector>

using namespace std;

vector<int> bellmanFord(int V, vector<vector<int>>& edges, int src) {

// Initially distance from source to all

// other vertices is not known(Infinite).

vector<int> dist(V, 1e8);

dist[src] = 0;

// Relaxation of all the edges V times, not (V - 1) as we

// need one additional relaxation to detect negative cycle

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

for (vector<int> edge : edges) {

int u = edge[0];

int v = edge[1];

int wt = edge[2];

if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {

// If this is the Vth relaxation, then there is

// a negative cycle

if(i == V - 1)

return {-1};

// Update shortest distance to node v

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

}

}

}

return dist;

}

int main() {

int V = 5;

vector<vector<int>> edges = {{1, 3, 20}, {4, 3, -12}, {2, 4, 41},

{1, 2, 81}, {0, 1, 45}};

int src = 0;

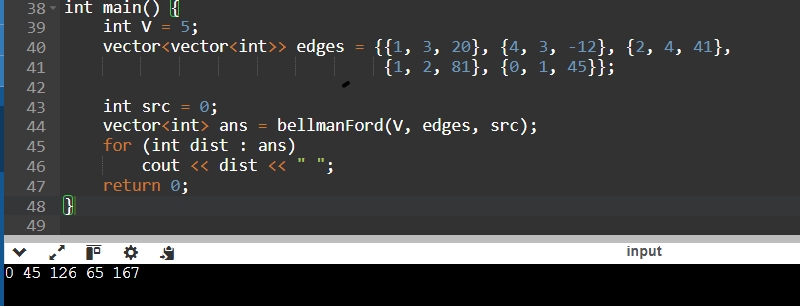
vector<int> ans = bellmanFord(V, edges, src);

for (int dist : ans)

cout << dist << " ";

return 0;

}



Implement all pairs shortest path for a graph using Floyd-Warshall algorithm

// C++ Program for Floyd Warshall Algorithm

#include <bits/stdc++.h>

using namespace std;

// Solves the all-pairs shortest path

// problem using Floyd Warshall algorithm

void floydWarshall(vector<vector<int>> &graph) {

int V = graph.size();

// Add all vertices one by one to

// the set of intermediate vertices.

for (int k = 0; k < V; k++) {

// Pick all vertices as source one by one

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

// Pick all vertices as destination

// for the above picked source

for (int j = 0; j < V; j++) {

// If vertex k is on the shortest path from

// i to j, then update the value of graph[i][j]

if ((graph[i][j] == -1 ||

graph[i][j] > (graph[i][k] + graph[k][j]))

&& (graph[k][j] != -1 && graph[i][k] != -1))

graph[i][j] = graph[i][k] + graph[k][j];

}

}

}

}

int main() {

vector<vector<int>> graph = {

{0, 4, -1, 5, -1},

{-1, 0, 1, -1, 6},

{2, -1, 0, 3, -1},

{-1, -1, 1, 0, 2},

{1, -1, -1, 4, 0}

};

floydWarshall(graph);

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

for(int j = 0; j<graph.size(); j++) {

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

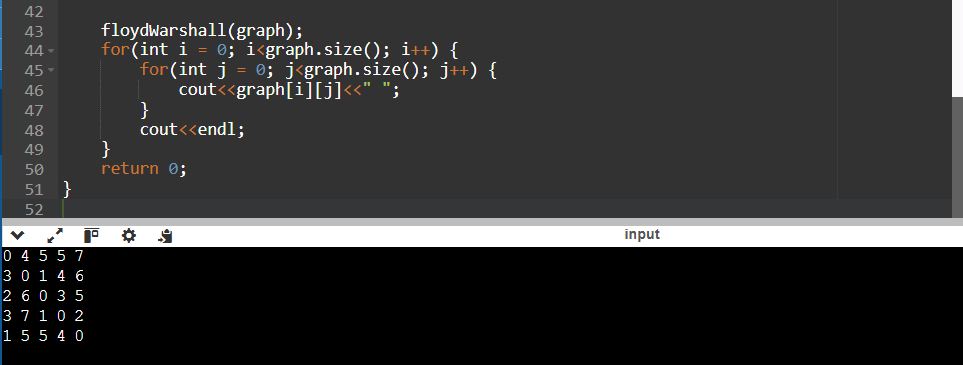
}

cout<<endl;

}

return 0;

}



Implement Prim’s and Kruskal’s algorithm to generate minimum cost 8. spanning tree using Greedy approach.

// A C++ program for Prim's Minimum

// Spanning Tree (MST) algorithm. The program is

// for adjacency matrix representation of the graph

#include <bits/stdc++.h>

using namespace std;

// A utility function to find the vertex with

// minimum key value, from the set of vertices

// not yet included in MST

int minKey(vector<int> &key, vector<bool> &mstSet) {

// Initialize min value

int min = INT\_MAX, min\_index;

for (int v = 0; v < mstSet.size(); v++)

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

// A utility function to print the

// constructed MST stored in parent[]

void printMST(vector<int> &parent, vector<vector<int>> &graph) {

cout << "Edge \tWeight\n";

for (int i = 1; i < graph.size(); i++)

cout << parent[i] << " - " << i << " \t"

<< graph[parent[i]][i] << " \n";

}

// Function to construct and print MST for

// a graph represented using adjacency

// matrix representation

void primMST(vector<vector<int>> &graph) {

int V = graph.size();

// Array to store constructed MST

vector<int> parent(V);

// Key values used to pick minimum weight edge in cut

vector<int> key(V);

// To represent set of vertices included in MST

vector<bool> mstSet(V);

// Initialize all keys as INFINITE

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

key[i] = INT\_MAX, mstSet[i] = false;

// Always include first 1st vertex in MST.

// Make key 0 so that this vertex is picked as first

// vertex.

key[0] = 0;

// First node is always root of MST

parent[0] = -1;

// The MST will have V vertices

for (int count = 0; count < V - 1; count++) {

// Pick the minimum key vertex from the

// set of vertices not yet included in MST

int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set

mstSet[u] = true;

// Update key value and parent index of

// the adjacent vertices of the picked vertex.

// Consider only those vertices which are not

// yet included in MST

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

// graph[u][v] is non zero only for adjacent

// vertices of m mstSet[v] is false for vertices

// not yet included in MST Update the key only

// if graph[u][v] is smaller than key[v]

if (graph[u][v] && mstSet[v] == false

&& graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

// Print the constructed MST

printMST(parent, graph);

}

// Driver's code

int main() {

vector<vector<int>> graph = { { 0, 2, 0, 10, 0 },

{ 2, 0, 3, 12, 5 },

{ 0, 13, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

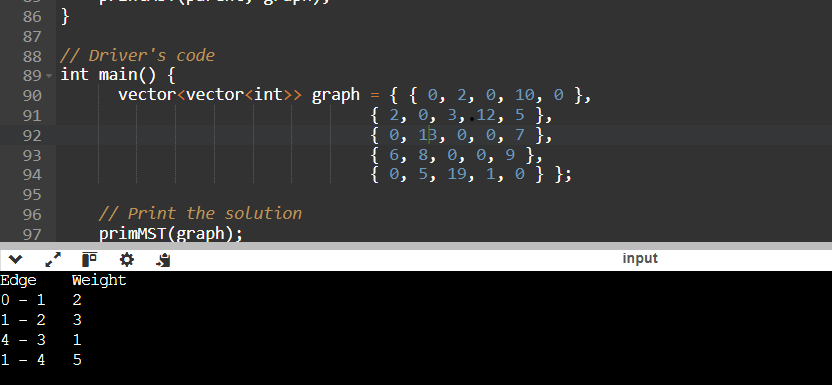
{ 0, 5, 19, 1, 0 } };

// Print the solution

primMST(graph);

return 0;

}



#include <bits/stdc++.h>

using namespace std;

// Disjoint set data struture

class DSU {

vector<int> parent, rank;

public:

DSU(int n) {

parent.resize(n);

rank.resize(n);

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

parent[i] = i;

rank[i] = 1;

}

}

int find(int i) {

return (parent[i] == i) ? i : (parent[i] = find(parent[i]));

}

void unite(int x, int y) {

int s1 = find(x), s2 = find(y);

if (s1 != s2) {

if (rank[s1] < rank[s2]) parent[s1] = s2;

else if (rank[s1] > rank[s2]) parent[s2] = s1;

else parent[s2] = s1, rank[s1]++;

}

}

};

bool comparator(vector<int> &a,vector<int> &b){

if(a[2]<=b[2])return true;

return false;

}

int kruskalsMST(int V, vector<vector<int>> &edges) {

// Sort all edhes

sort(edges.begin(), edges.end(),comparator);

// Traverse edges in sorted order

DSU dsu(V);

int cost = 0, count = 0;

for (auto &e : edges) {

int x = e[0], y = e[1], w = e[2];

// Make sure that there is no cycle

if (dsu.find(x) != dsu.find(y)) {

dsu.unite(x, y);

cost += w;

if (++count == V - 1) break;

}

}

return cost;

}

int main() {

// An edge contains, weight, source and destination

vector<vector<int>> edges = {

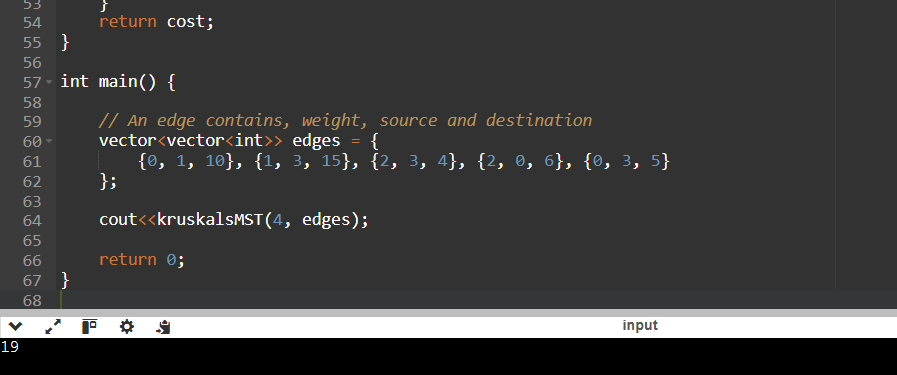
{0, 1, 10}, {1, 3, 15}, {2, 3, 4}, {2, 0, 6}, {0, 3, 5}

};

cout<<kruskalsMST(4, edges);

return 0;

}



Make use of Ford-Fulkerson algorithm to compute maximum flow.

// C++ program for implementation of Ford Fulkerson

// algorithm

#include <iostream>

#include <limits.h>

#include <queue>

#include <string.h>

using namespace std;

// Number of vertices in given graph

#define V 6

/\* Returns true if there is a path from source 's' to sink

't' in residual graph. Also fills parent[] to store the

path \*/

bool bfs(int rGraph[V][V], int s, int t, int parent[])

{

// Create a visited array and mark all vertices as not

// visited

bool visited[V];

memset(visited, 0, sizeof(visited));

// Create a queue, enqueue source vertex and mark source

// vertex as visited

queue<int> q;

q.push(s);

visited[s] = true;

parent[s] = -1;

// Standard BFS Loop

while (!q.empty()) {

int u = q.front();

q.pop();

for (int v = 0; v < V; v++) {

if (visited[v] == false && rGraph[u][v] > 0) {

// If we find a connection to the sink node,

// then there is no point in BFS anymore We

// just have to set its parent and can return

// true

if (v == t) {

parent[v] = u;

return true;

}

q.push(v);

parent[v] = u;

visited[v] = true;

}

}

}

// We didn't reach sink in BFS starting from source, so

// return false

return false;

}

// Returns the maximum flow from s to t in the given graph

int fordFulkerson(int graph[V][V], int s, int t)

{

int u, v;

// Create a residual graph and fill the residual graph

// with given capacities in the original graph as

// residual capacities in residual graph

int rGraph[V]

[V]; // Residual graph where rGraph[i][j]

// indicates residual capacity of edge

// from i to j (if there is an edge. If

// rGraph[i][j] is 0, then there is not)

for (u = 0; u < V; u++)

for (v = 0; v < V; v++)

rGraph[u][v] = graph[u][v];

int parent[V]; // This array is filled by BFS and to

// store path

int max\_flow = 0; // There is no flow initially

// Augment the flow while there is path from source to

// sink

while (bfs(rGraph, s, t, parent)) {

// Find minimum residual capacity of the edges along

// the path filled by BFS. Or we can say find the

// maximum flow through the path found.

int path\_flow = INT\_MAX;

for (v = t; v != s; v = parent[v]) {

u = parent[v];

path\_flow = min(path\_flow, rGraph[u][v]);

}

// update residual capacities of the edges and

// reverse edges along the path

for (v = t; v != s; v = parent[v]) {

u = parent[v];

rGraph[u][v] -= path\_flow;

rGraph[v][u] += path\_flow;

}

// Add path flow to overall flow

max\_flow += path\_flow;

}

// Return the overall flow

return max\_flow;

}

// Driver program to test above functions

int main()

{

// Let us create a graph shown in the above example

int graph[V][V]

= { { 0, 16, 13, 0, 0, 0 }, { 0, 0, 10, 12, 0, 0 },

{ 0, 4, 0, 0, 14, 0 }, { 0, 0, 9, 0, 0, 20 },

{ 0, 0, 0, 7, 0, 4 }, { 0, 0, 0, 0, 0, 0 } };

cout << "The maximum possible flow is "

<< fordFulkerson(graph, 0, 5);

return 0;

}

