**Week 6.**

**/\*Ques1. Given a (directed/undirected) graph, design an algorithm**

**and implement it using a program to find if a path exists between**

**two given vertices or not. (Hint: use DFS)\*/**

#include <iostream>

#include <vector>

#include <stdbool.h>

using namespace std;

void DFS(int source, vector<vector<int>> matrix, vector<bool> &visited);

int main()

{

int ch;

cout << "1. For Adjacency Matrix as a Input." << endl;

cout << "2. For Edges and Vertex as a Input." << endl;

cin >> ch;

if (ch == 1)

{

int source, destination, size, t;

cout << "Input size :";

cin >> size;

vector<bool> visited(size, false);

vector<vector<int>> matrix;

vector<int> temp;

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

{

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

{

cin >> t;

temp.push\_back(t);

}

matrix.push\_back(temp);

temp.clear();

}

cout << "Input Source and Destination :";

cin >> source >> destination;

DFS(source - 1, matrix, visited);

if (visited[destination - 1] == true)

cout << "\nYes Path Exists.";

else

cout << "\nNo Such Path Exists.";

}

else

{

int source, destination;

int edges, vertex;

cout << "Input no of Vertex and Edges :";

cin >> vertex >> edges;

vector<bool> visited(vertex);

vector<vector<int>> matrix;

vector<int> temp;

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

{

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

{

temp.push\_back(0);

}

matrix.push\_back(temp);

temp.clear();

}

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

{

int u, v;

cin >> u >> v;

matrix[u - 1][v - 1] = 1;

}

cout << "Input Source and Destination :";

cin >> source >> destination;

DFS(source - 1, matrix, visited);

if (visited[destination - 1] == true)

cout << "\nYes Path Exists.";

else

cout << "\nNo Such Path Exists.";

}

return 0;

}

void DFS(int source, vector<vector<int>> matrix, vector<bool> &visited)

{

printf("%d ", source);

visited[source] = true;

for (int neigh = 0; neigh < matrix.size(); neigh++)

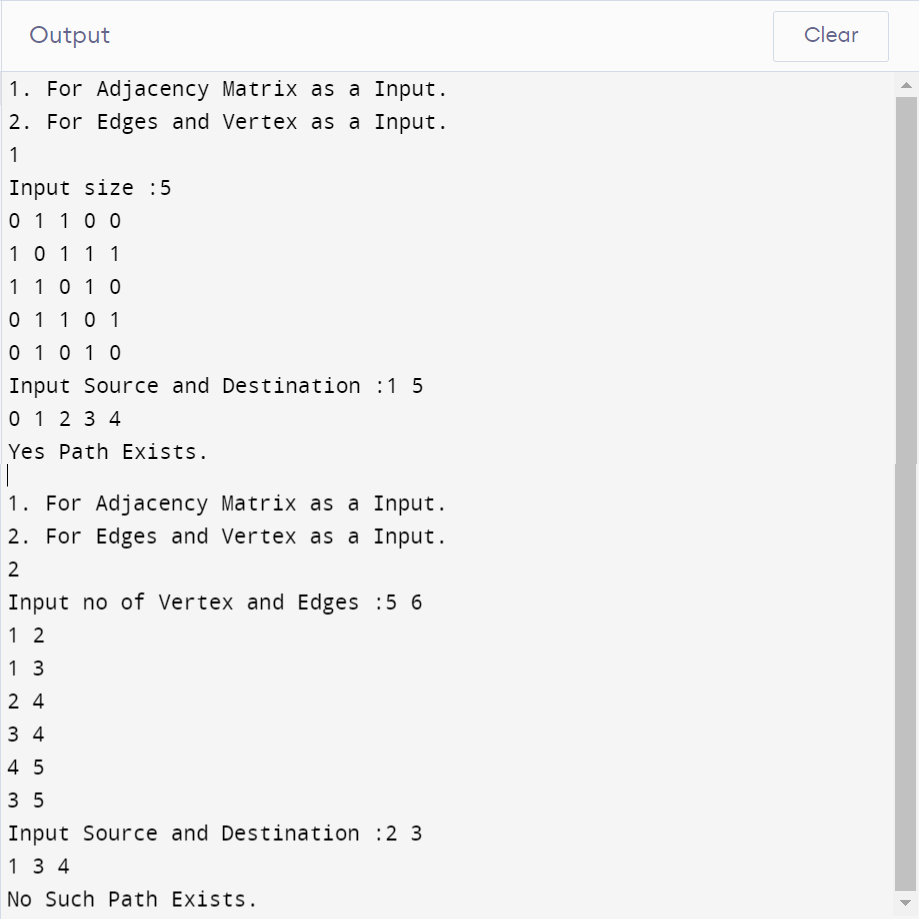
{

if (matrix[source][neigh] == 1 && visited[neigh] == false)

DFS(neigh, matrix, visited);

}

}



**/\*Ques2.Given a graph, design an algorithm and implement it using**

**a program to find if a graph is bipartite or not. (Hint: use BFS)\*/**

#include <iostream>

#include <queue>

using namespace std;

bool bipar(int \*\*AdjM, int V)

{

queue<int> Q;

int \*color = new int[V]();

int curr = 1;

color[0] = curr;

Q.push(0);

while (!Q.empty())

{

int u = Q.front();

Q.pop();

if (AdjM[u][u])

return false;

curr \*= -1;

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

{

if (AdjM[u][v] && color[v] == 0)

{

color[v] = curr;

Q.push(v);

}

else if (AdjM[u][v] && color[u] == color[v])

return false;

}

}

return true;

}

int main()

{

int v, s, d;

cin >> v; // Size of adjacency matrix

int \*\*AdjM; // Adjacency Matrix

AdjM = (int \*\*)malloc(v \* sizeof(int \*));

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

AdjM[i] = (int \*)malloc(v \* sizeof(int));

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

{

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

{

cin >> AdjM[i][j];

}

}

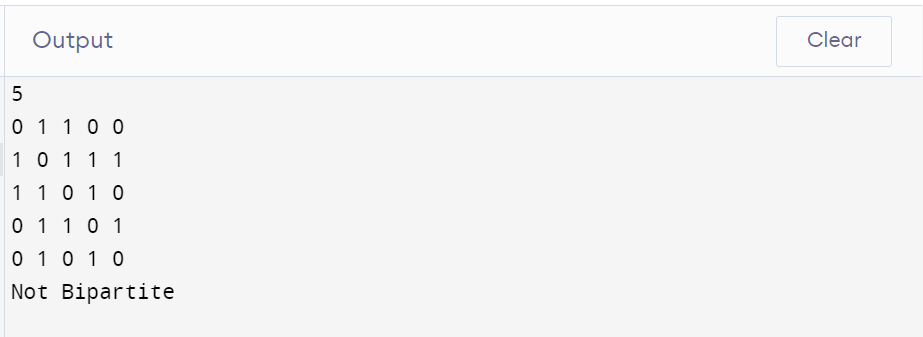
if (bipar(AdjM, v) == true)

cout << "Yes Bipartite";

else

cout << "Not Bipartite";

}



**Week 8.**

**/\*Ques3.Given a directed graph, design an algorithm and implement**

**it using a program to find whether cycle exists in the graph or not \*/**

#include <iostream>

#include <vector>

#include <stdbool.h>

using namespace std;

bool IsCycle(int src, vector<vector<int>> &adj, vector<bool> &stack, vector<bool> &visited)

{

stack[src] = true;

if (visited[src] == false)

{

for (auto j : adj[src])

{

if (visited[j] == false && IsCycle(j, adj, stack, visited))

return true;

if (stack[j] == true)

return true;

}

}

stack[src] = false;

return false;

}

int main()

{

int vertex = 0, edges;

cin >> edges;

vector<vector<int>> matrix;

vector<int> temp;

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

{

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

{

int t;

cin >> t;

temp.push\_back(t);

if (t == 1)

vertex++;

}

matrix.push\_back(temp);

temp.clear();

}

vector<vector<int>> adj(vertex);

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

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

if (matrix[i][j] == 1)

adj[i].push\_back(j);

bool cycle = false;

vector<bool> stack(edges, false);

vector<bool> visited(edges, false);

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

{

if (visited[i] == false && IsCycle(i, adj, visited, stack))

{

cycle = true;

break;

}

}

if (cycle)

cout << "Yes Cycle Exixts.";

else

cout << "No Cycle Exits";

return 0;

}



**Week7.**

**/\*Ques1. After end term examination, Akshay wants to party with his friends. All his friends are living as paying guest and it has been decided to first gather at Akshay’s house and then move towards party location. The problem is that no one knows the exact address of his house in the city. Akshay as a computer science wizard knows how to apply his theory subjects in his real life and came up with an amazing idea to help his friends. He draws a graph by looking in to location of his house and his friends’ location (as a node in the graph) on a map. He wishes to find out shortest distance and path covering that distance from each of his friend’s location to his house and then whatsapp them this path so that they can reach his house in minimum time. Akshay has developed the program that implements Dijkstra’s algorithm but not sure about correctness of results. Can you also implement the same algorithm and verify the correctness of Akshay’s results? (Hint: Print shortest path and distance from friends’ location to Akshay’s house)\*/**

#include <iostream>

#include <bits/stdc++.h>

using namespace std;

int minDisIndex(int \*dis, bool \*vis, int v)

{

int i;

int minDis = INT\_MAX;

int minIndex = -1;

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

{

if (vis[i] == false && dis[i] <= minDis)

{

minDis = dis[i];

minIndex = i;

}

}

return minIndex;

}

void dijkstra(vector<vector<int>> mat, int v, int s)

{

int dis[v];

bool vis[v];

int parent[v];

int i, j;

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

{

dis[i] = INT\_MAX;

vis[i] = false;

parent[i] = -1;

}

dis[s] = 0;

parent[s] = s;

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

{

int m = minDisIndex(dis, vis, v);

vis[m] = true;

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

{

if (dis[m] != INT\_MAX && !vis[j] && mat[m][j])

{

if (dis[j] > dis[m] + mat[m][j])

{

dis[j] = dis[m] + mat[m][j];

parent[j] = m;

}

}

}

}

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

{

if (i == s)

{

cout << i + 1 << " : " << dis[i] << endl;

continue;

}

cout << i + 1;

j = i;

while (parent[j] != s)

{

cout << " " << parent[j] + 1;

j = parent[j];

}

cout << " " << s + 1 << " : " << dis[i] << endl;

}

}

int main()

{

int i, j;

int v;

cin >> v;

vector<vector<int>> mat(v, vector<int>(v));

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

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

cin >> mat[i][j];

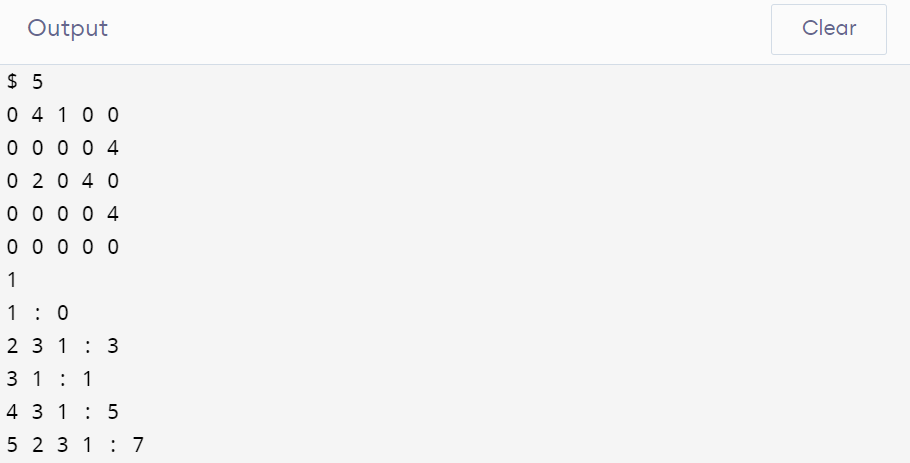
int s;

cin >> s;

dijkstra(mat, v, s - 1);

return 0;

}

****

**/\*Ques2.Design an algorithm and implement it using a program to solve previous question's problem using Bellman-Ford's shortest path algorithm.\*/**

#include <iostream>

#include <bits/stdc++.h>

using namespace std;

void bellmanFord(vector<vector<int>> mat, int v, int s)

{

int dis[v];

int parent[v];

int i, j;

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

{

dis[i] = INT\_MAX;

}

dis[s] = 0;

parent[s] = s;

for (int k = 0; k < v - 1; k++)

for (i = 0; i < v - 1; i++)

{

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

{

if (dis[i] != INT\_MAX && mat[i][j])

{

if (dis[j] > dis[i] + mat[i][j])

{

dis[j] = dis[i] + mat[i][j];

parent[j] = i;

}

}

}

}

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

{

if (i == s)

{

cout << i + 1 << " : " << dis[i] << endl;

continue;

}

cout << i + 1;

j = i;

while (parent[j] != s)

{

cout << " " << parent[j] + 1;

j = parent[j];

}

cout << " " << s + 1 << " : " << dis[i] << endl;

}

}

int main()

{

int i, j;

int v;

cin >> v;

vector<vector<int>> mat(v, vector<int>(v));

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

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

cin >> mat[i][j];

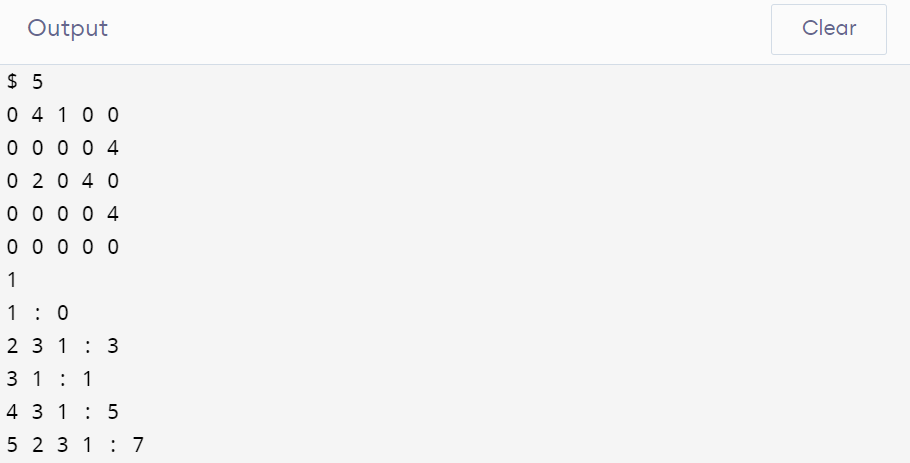
int s;

cin >> s;

bellmanFord(mat, v, s - 1);

return 0;

}



**/\*Ques3. Given a directed graph with two vertices ( source and destination). Design an algorithm and implement it using a program to find the weight of the shortest path from source to destination with exactly k edges on the path.\*/**

#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

int min(int a, int b)

{

return a < b ? a : b;

}

int shortestPath(int V, int graph[][V], int u, int v, int k)

{

if (k == 0 && u == v)

return 0;

if (k == 1 && graph[u][v] != INT\_MAX)

return graph[u][v];

if (k <= 0)

return INT\_MAX;

int res = INT\_MAX;

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

{

if (graph[u][i] != INT\_MAX && u != i && v != i)

{

int rec\_res = shortestPath(V, graph, i, v, k - 1);

if (rec\_res != INT\_MAX)

res = min(res, graph[u][i] + rec\_res);

}

}

return res;

}

int main()

{

int V, src, des, k;

scanf("%d", &V);

int graph[V][V];

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

{

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

{

scanf("%d", &graph[i][j]);

}

}

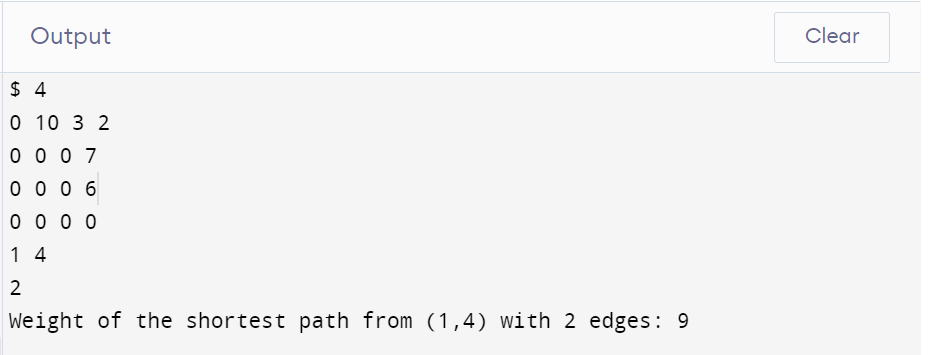
scanf("%d%d", &src, &des);

scanf("%d", &k);

printf("Weight of the shortest path from (%d,%d) with %d edges: %d", src, des, k, shortestPath(V, graph, src - 1, des - 1, k));

return 0;

}

****

**Week 8.**

**/\*Ques1. Assume that a project of road construction to connect some cities is given to your friend. Map of these cities and roads which will connect them (after construction) is provided to him in the form of a graph. Certain amount of rupees is associated with construction of each road. Your friend has to calculate the minimum budget required for this project. The budget should be designed in such a way that the cost of connecting the cities should be minimum and number of roads required to connect all the cities should be minimum (if there are N cities then only N-1 roads need to be constructed). He asks you for help. Now, you have to help your friend by designing an algorithm which will find minimum cost required to connect these cities. (use Prim's algorithm)\*/**

#include <bits/stdc++.h>

using namespace std;

#define V 7

int minimumnodevertex(vector<int> weight, vector<bool> process)

{

int vertex;

int minimum = INT\_MAX;

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

{

if (minimum > weight[i] && process[i] == false)

{

minimum = weight[i];

vertex = i;

}

}

return vertex;

}

void findMST(int graph[V][V])

{

vector<int> parent(V, -1);

vector<int> weight(V, INT\_MAX);

vector<bool> process(V, false);

int MinimumWight = 0;

weight[0] = 0;

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

{

int minvertex = minimumnodevertex(weight, process);

process[minvertex] = true;

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

{

if (graph[minvertex][j] != 0 && process[j] != true && graph[minvertex][j] < weight[j])

{

weight[j] = graph[minvertex][j];

parent[j] = minvertex;

}

}

}

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

MinimumWight += graph[parent[i]][i];

cout << "Minimum Spanning Weight :" << MinimumWight;

}

int main()

{

int graph[V][V];

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

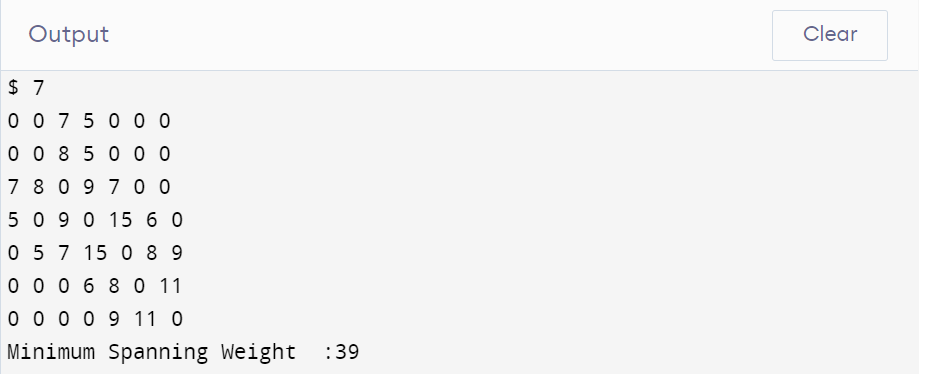
for (int j = 0; j < V; j++)

cin >> graph[i][j];

findMST(graph);

return 0;

}



**/\*Ques2.Implement the previous problem using Kruskal's algorithm.\*/**

#include <iostream>

#include <algorithm>

#include <vector>

#include <limits.h>

using namespace std;

vector<pair<int, pair<int, int>>> graph;

vector<pair<int, pair<int, int>>> result;

int parent[10000];

using namespace std;

void make(int i)

{

parent[i] = i;

}

int find(int V)

{

if (V == parent[V])

return V;

return find(parent[V]);

}

void Union(int a, int b, int i)

{

a = find(a);

b = find(b);

if (a != b)

{

parent[b] = a;

result.push\_back(graph[i]);

}

}

void Kruskal(int V)

{

sort(graph.begin(), graph.end());

int E = graph.size();

int s, d, w;

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

{

s = graph[i].second.first;

d = graph[i].second.second;

Union(s, d, i);

}

}

int main()

{

int V, E = 0;

cin >> V;

vector<vector<int>> matrix;

vector<int> temp;

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

{

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

{

int t;

cin >> t;

temp.push\_back(t);

if (t != 0)

E++;

}

matrix.push\_back(temp);

temp.clear();

}

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

make(i);

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

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

if (matrix[i][j] != 0)

graph.push\_back(make\_pair(matrix[i][j], make\_pair(i, j)));

int sum = 0;

Kruskal(V);

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

{

int w = result[i].first;

sum = sum + w;

}

cout << "Minimum Spanning weight :" << sum;

return 0;

}



**/\*Ques3.Assume that same road construction project is given to another person. The amount he will earn from this project is directly proportional to the budget of the project. This person is greedy, so he decided to maximize the budget by constructing those roads who have highest construction cost. Design an algorithm and implement it using a program to find the maximum budget required for the project.\*/**

#include <iostream>

#include <algorithm>

#include <vector>

#include <limits.h>

using namespace std;

vector<pair<int, pair<int, int>>> graph;

vector<pair<int, pair<int, int>>> result;

int parent[10000];

using namespace std;

void make(int i)

{

parent[i] = i;

}

int find(int V)

{

if (V == parent[V])

return V;

return find(parent[V]);

}

void Union(int a, int b, int i)

{

a = find(a);

b = find(b);

if (a != b)

{

parent[b] = a;

result.push\_back(graph[i]);

}

}

void Kruskal(int V)

{

sort(graph.rbegin(), graph.rend());

int E = graph.size();

int s, d, w;

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

{

s = graph[i].second.first;

d = graph[i].second.second;

Union(s, d, i);

}

}

int main()

{

int V, E = 0;

cin >> V;

vector<vector<int>> matrix;

vector<int> temp;

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

{

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

{

int t;

cin >> t;

temp.push\_back(t);

if (t != 0)

E++;

}

matrix.push\_back(temp);

temp.clear();

}

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

make(i);

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

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

if (matrix[i][j] != 0)

graph.push\_back(make\_pair(matrix[i][j], make\_pair(i, j)));

int sum = 0;

Kruskal(V);

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

{

int w = result[i].first;

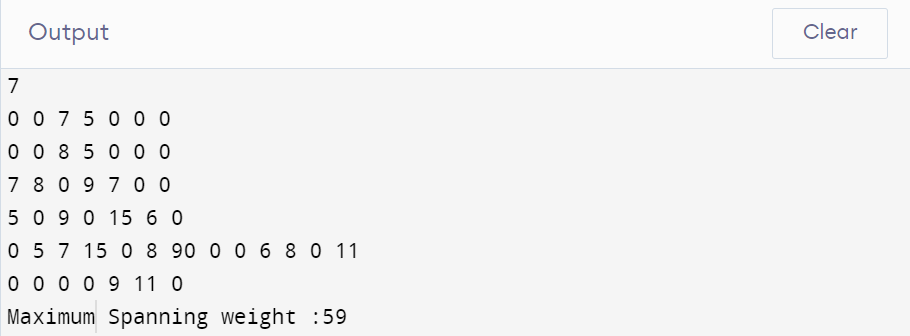
sum = sum + w;

}

cout << "Maximum Spanning weight :" << sum;

return 0;

}



**Week 9.**

**/\*Ques1. Given a graph, Design an algorithm and implement it using a program to implement Floyed-Warshall all pair shortest algorithm.\*/**

#include <bits/stdc++.h>

using namespace std;

#define V 5

#define INF 99999

void printSolution(int dist[][V])

{

cout << "Shortest Distance Matrix: \n";

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

{

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

{

if (dist[i][j] == INF)

cout << "INF"

<< " ";

else

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

}

cout << endl;

}

}

void floydWarshall(int graph[][V])

{

int dist[V][V], i, j, k;

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

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

dist[i][j] = graph[i][j];

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

{

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

{

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

{

if (dist[i][j] > (dist[i][k] + dist[k][j]) && (dist[k][j] != INF && dist[i][k] != INF))

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

}

}

}

printSolution(dist);

}

int main()

{

int graph[V][V] = {{0, 10, 5, 5, INF},

{INF, 0, 5, 5, 5},

{INF, INF, 0, INF, 10},

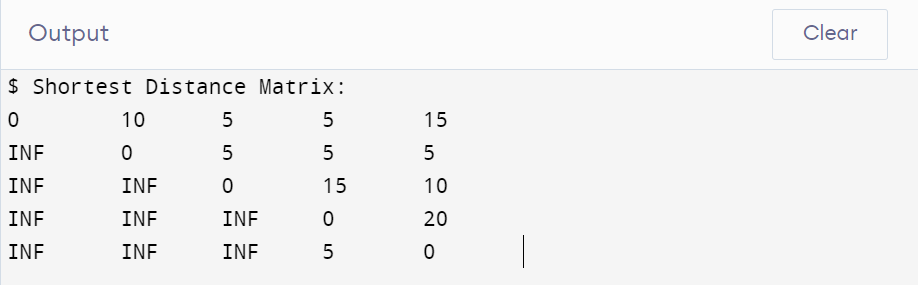
{INF, INF, INF, 0, 20},

{INF, INF, INF, 5, 0}};

floydWarshall(graph);

return 0;

}



**/\*Ques2.Given a knapsack of maximum capacity w . N items are provided , each having its own value and weight . You have to Design an algorithm and implement it using a program to find the list of the selected items such that the final selected content has weight w and has maximum value . You can take fractions of items , i.e . the items can be broken into smaller pieces so that you have to carry only a fraction x of item i , where 0<=x<=1\*/**

#include <iostream>

#include <vector>

#include <map>

using namespace std;

int main()

{

float weight, volume, capacity, PerUnit;

map<float, pair<float, int>, greater<float>> mymap;

int n;

cin >> n;

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

{

cin >> weight >> volume;

PerUnit = volume / weight;

mymap.insert({PerUnit, make\_pair(weight, i + 1)});

}

cin >> capacity;

float MaxProfit = 0, we = 0;

vector<pair<int, float>> item\_weight;

for (auto it = mymap.begin(); it != mymap.end(); it++)

{

MaxProfit += it->first \* it->second.first;

we += it->second.first;

if (we < capacity)

item\_weight.push\_back(make\_pair(it->second.second, it->second.first));

if (we > capacity)

{

float temp = 0;

MaxProfit -= it->first \* it->second.first;

we -= it->second.first;

temp = capacity - we;

MaxProfit += it->first \* temp;

we += temp;

item\_weight.push\_back(make\_pair(it->second.second, temp));

}

if (we == capacity)

break;

}

cout << "Maximum Value:" << MaxProfit << endl;

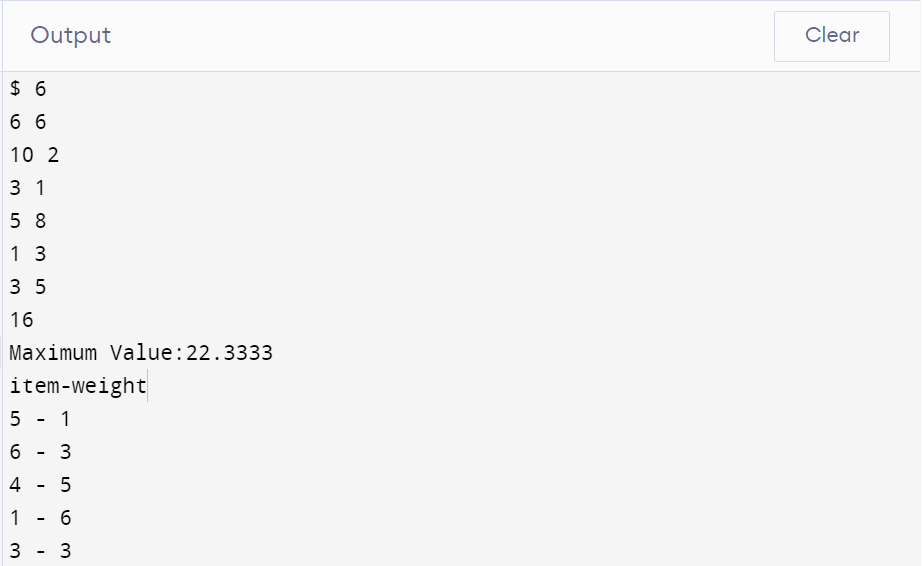
cout << "item-weight" << endl;

for (int i = 0; i < item\_weight.size(); i++)

cout << item\_weight[i].first << " - " << item\_weight[i].second << endl;

return 0;

}

****

**/\*Ques3. Given an array of elements . Assume arr [ i ] represents the size of file i . Write an algorithm and a program to merge all these files into single file with minimum computation . For given two files A and B with sizes m and n , computation cost of merging them is O ( m + n ) . ( Hint : use greedy approach )\*/**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{

int n, sum = 0, MinimumCost = 0;

cin >> n;

vector<int> vrr(n);

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

cin >> vrr[i];

sort(vrr.begin(), vrr.end());

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

{

vrr[i] = vrr[i - 1] + vrr[i];

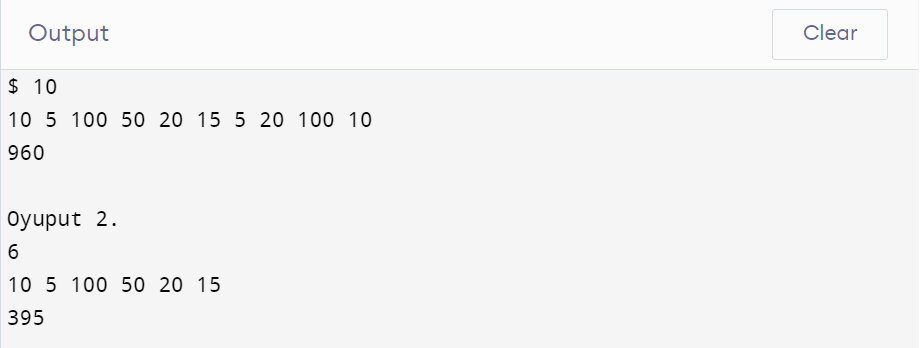
MinimumCost += vrr[i];

}

cout << MinimumCost;

return 0;

}



**Week 10.**

**/\*Ques1. Given a list of activities with their starting time and finishing time. Your goal is to select maximum number of activities that can be performed by a single person such that selected activities must be non-conflicting. Any activity is said to be non-conflicting if starting time of an activity is greater than or equal to the finishing time of the other activity. Assume that a person can only work on a single activity at a time.\*/**

#include <iostream>

#include <vector>

#include <map>

#include <algorithm>

using namespace std;

int main()

{

int size;

cin >> size;

vector<int> result;

multimap<int, pair<int, int>> activity;

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

{

int s, e;

cin >> s >> e;

activity.insert({e, make\_pair(s, i + 1)});

}

int count = 0, next = -1;

for (auto it = activity.begin(); it != activity.end(); it++)

{

if (it->second.first > next)

{

count++;

next = it->first;

result.push\_back(it->second.second);

}

}

cout << "No. of non-conflicting activities: " << count << endl;

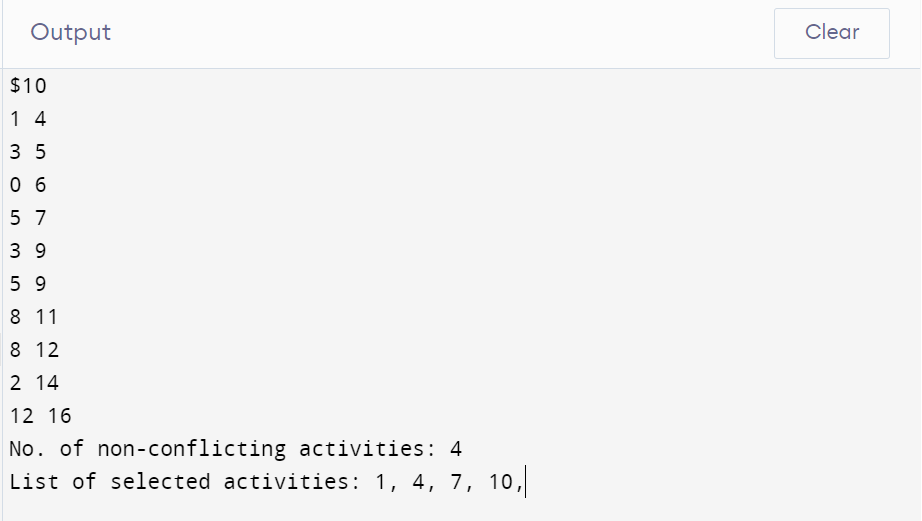
cout << "List of selected activities: ";

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

cout << result[i] << ", ";

return 0;

}



**/\*Ques3. Given an unsorted array of elements, design an algorithm and implement it using a program to find whether majority element exists or not. Also find median of the array. A majority element is an element that appears more than n/2 times, where n is the size of array.\*/**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{

int n;

cin >> n;

vector<int> vrr(n);

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

cin >> vrr[i];

int median = vrr[n / 2];

sort(vrr.begin(), vrr.end());

int count = 1, temp = 0;

for (int i = 0; i < n - 1; i++)

{

if (vrr[i] == vrr[i + 1])

count++;

else

{

if (count >= (n / 2))

{

cout << "Yes";

temp = 1;

}

count = 1;

}

}

if (!temp)

cout << "NO";

cout << "\n"

<< median;

return 0;

}

