// A dynamic programming based

// solution for 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)
{
	int i, w;
	vector<vector<int> > K(n + 1, vector<int>(W + 1));

	// Build table K[][] in bottom up manner
	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];
}


// Driver Code
int main()
{
	int profit[] = { 60, 100, 120 };
	int weight[] = { 10, 20, 30 };
	int W = 50;
	int n = sizeof(profit) / sizeof(profit[0]);

	cout << knapSack(W, weight, profit, n);
	return 0;
}
//************************************************************************************//
```

Coin Change

```
// K is the maximum number of coin types
// N is the maximum amount of money to handle
int coins[K], dp[N];
int main() {
    int n, k; // target amount of money and number of coin types
```

```
    cin >> n >> k;
    for (int i = 0; i < k; ++i) {
        cin >> coins[i];
    }
    dp[0] = 1;
    for (int i = 0; i < k; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (j - coins[i] >= 0)
                dp[j] += dp[j - coins[i]];
        }
    }
    cout << dp[n] << "\n";
}

//************************************************************************************************//
// Dynamic Programming C++ implementation
// of LIS problem

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

// lis() returns the length of the longest
// increasing subsequence in arr[] of size n
int lis(int arr[], int n)
{
	int lis[n];

	lis[0] = 1;

	// Compute optimized LIS values in
	// bottom up manner
	for (int i = 1; i < n; i++) {
		lis[i] = 1;
		for (int j = 0; j < i; j++)
			if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
				lis[i] = lis[j] + 1;
	}

	// Return maximum value in lis[]
	return *max_element(lis, lis + n);
}

// Driver program to test above function
int main()
{
	int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
	int n = sizeof(arr) / sizeof(arr[0]);

	// Function call
	printf("Length of lis is %d\n", lis(arr, n));
	return 0;
}

//**********************************************************************************************//
// Dynamic Programming C++ implementation
```

// of LCS problem

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

int longestCommonSubsequence(string& text1, string& text2)
{
	int n = text1.size();
	int m = text2.size();

	// initializing 2 vectors of size m
	vector<int> prev(m + 1, 0), cur(m + 1, 0);

	for (int idx2 = 0; idx2 < m + 1; idx2++)
		cur[idx2] = 0;

	for (int idx1 = 1; idx1 < n + 1; idx1++) {
		for (int idx2 = 1; idx2 < m + 1; idx2++) {
			// if matching
			if (text1[idx1 - 1] == text2[idx2 - 1])
				cur[idx2] = 1 + prev[idx2 - 1];

			// not matching
			else
				cur[idx2]
					= 0 + max(cur[idx2 - 1], prev[idx2]);
		}
		prev = cur;
	}

	return cur[m];
}

int main()
{
	string S1 = "AGGTAB";
	string S2 = "GXTXAYB";

	// Function call
	cout << "Length of LCS is "
		<< longestCommonSubsequence(S1, S2);

	return 0;
}
```

//**//

isItTree

```
const int N = 1e6 + 5;
vector<int> g[N];
bool vis[N];
vector<int> path;
int n,m;
int par[N];
void dfs(int node,int parent)
```

```
{
   vis[node] = true;
   par[node] = parent;
   for (auto x : g[node])
   {
      if (!vis[x])
      {
         dfs(x, node);
      }
   }
}
void DO_Your_Best()
{
   cin >> n >> m;
   if (m != n - 1)
   {
      cout << "NO\n";
      return;
   }
   while (m--)
   {
      int a, b;
      cin >> a >> b;
      g[a].push_back(b);
      g[b].push_back(a);
   }
   dfs(1 - 1);
   bool comp = true;
   for (int i = 1; i <= n; ++i)
   {
      if (!vis[i])
         comp = false;
   }
   if (comp)
      cout << "YES\n";
   else
      cout << "NO\n";
}
//*********************************************************************************************//
```

Counting Rooms

```
#pragma GCC optimize("O3")
#include <bits/stdc++.h>
using namespace std;
#define int long long int
#define ll long long
#define all(v) ((v).begin()),((v).end())
#define ordered_set tree<int, null_type,less<int>, rb_tree_tag,tree_order_statistics_node_update>
int mul(const int &a, const int &b,int MOD) {
   return (a % MOD + MOD) * (b % MOD + MOD) % MOD;
}
// in case negative (2*mod)
int add(const int &a, const int &b,int MOD){
   return (a + b + 2 * MOD) % MOD;
}
```

```
const int MN = 1000;
char grid[MN][MN];
int row_num,col_num;
bool visited[MN][MN];
int dx4[] = {0,0,-1,1};
int dy4[] = {-1,1,0,0};
void dfss(int r,int c){
   if(r<0||r>=row_num||c<0||c>=col_num||grid[r][c]=='#'||visited[r][c])return;
   visited[r][c]= true;
//    for (int i = 0; i < 4; ++i) {
//        dfss(r + dx4[i] , c + dy4[i]);
//    }
   dfss(r,c-1);
   dfss(r , c+1);
   dfss(r+1 , c);
   dfss(r -1 , c);
}
void DO_Your_Best() {
   cin>> row_num >> col_num;
   for (int i = 0; i < row_num; ++i) {
      for (int j = 0; j < col_num; ++j) {
         cin >> grid[i][j];
      }
   }
   int cnt = 0 ;
   for (int i = 0; i < row_num; ++i) {
      for (int j = 0; j < col_num; ++j) {
         if(grid[i][j] == '.' && !visited[i][j]){
            dfss(i,j);
            cnt++;
         }
      }
   }
   cout << cnt << '\n';
}

signed main(){
#ifdef Clion
   freopen("input.txt", "r", stdin);
   freopen("output.txt", "w", stdout);
#endif
   ios_base ::sync_with_stdio(0);cin.tie(0);cout.tie(0);
   int t = 1;
   //  cin >> t;
   while(t--){
      DO_Your_Best();
   }
//***********************************************************************************************//
```
dfs and cycle solution
```
#pragma GCC optimize("O3")
#include <bits/stdc++.h>
using namespace std;
#define int long long int
#define ll long long
#define all(v) ((v).begin()),((v).end())
```

```
#define ordered_set tree<int, null_type,less<int>, rb_tree_tag,tree_order_statistics_node_update>
int mul(const int &a, const int &b,int MOD) {
    return (a % MOD + MOD) * (b % MOD + MOD) % MOD;
}
// in case negative (2*mod)
int add(const int &a, const int &b,int MOD){
    return (a + b + 2 * MOD) % MOD;
}
const int N=1e6+5;
vector<int>g[N];
vector<int>top;
bool vis[N],path[N],cycle;
int n,m;
void dfs(int node){
    vis[node]= true;
    path[node]= true;
    for(auto x:g[node]){
        if(path[x])
            cycle = true;
        if(!vis[x])dfs(x);
    }
    top.push_back(node);
    path[node]= false;
}
void DO_Your_Best() {
    cin >> n >> m;
    while (m--){
        int a , b;
        cin >> a >> b;
        g[a].push_back(b);
    }
    cycle = false;
    for (int i = 1; i <= n ; ++i) {
        if(!vis[i])
            dfs(i);
    }
    reverse(all(top));
    vector<int>ind(n+1);
    for (int i = 0; i < n; ++i) {
        ind[top[i]] = i;
    }
    if(cycle){
        cout << "IMPOSSIBLE\n";
        return;
    }
//    for (int i = 1; i <=n  ; ++i) {
//        for(auto it : g[i]){
//            if(ind[it] < ind[i]){
//                cout << "IMPOSSIBLE\n";
//                return;
//            }
//        }
//    }
    for(auto it : top)
        cout << it << ' ';
```

```
}

signed main(){
   ios_base ::sync_with_stdio(0);cin.tie(0);cout.tie(0);
   int t = 1;
   //  cin >> t;
   while(t--){
      DO_Your_Best();
   }
}
//*********************************************************************************************//
```

Program to count Number of connected components in an undirected graph

```
// C++ program for above approach
#include <bits/stdc++.h>
using namespace std;

// Graph class represents a undirected graph
// using adjacency list representation
class Graph {
	// No. of vertices
	int V;

	// Pointer to an array containing adjacency lists
	list<int>* adj;

	// A function used by DFS
	void DFSUtil(int v, bool visited[]);

public:
	// Constructor
	Graph(int V);

	void addEdge(int v, int w);
	int NumberOfconnectedComponents();
};

// Function to return the number of
// connected components in an undirected graph
int Graph::NumberOfconnectedComponents()
{

	// Mark all the vertices as not visited
	bool* visited = new bool[V];

	// To store the number of connected components
	int count = 0;
	for (int v = 0; v < V; v++)
		visited[v] = false;

	for (int v = 0; v < V; v++) {
		if (visited[v] == false) {
			DFSUtil(v, visited);
			count += 1;
```

```
            }
    }

    return count;
}

void Graph::DFSUtil(int v, bool visited[])
{

    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices
    // adjacent to this vertex
    list<int>::iterator i;

    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

// Add an undirected edge
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);
}

// Driver code
int main()
{
    Graph g(5);
    g.addEdge(1, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 4);

    cout << g.NumberOfconnectedComponents();

    return 0;
}
//*******************************************************************************************************//
```

Program for Dijkstra's Single Source Shortest Path Algorithm

```
// A C++ program for Dijkstra's single source shortest path
// algorithm. The program is for adjacency matrix
// representation of the graph
```

```
#include <limits.h>
#include <stdio.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum
// distance value, from the set of vertices not yet included
// in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
	// Initialize min value
	int min = INT_MAX, min_index;

	for (int v = 0; v < V; v++)
		if (sptSet[v] == false && dist[v] <= min)
			min = dist[v], min_index = v;

	return min_index;
}

// A utility function to print the constructed distance
// array
void printSolution(int dist[], int n)
{
	printf("Vertex Distance from Source\n");
	for (int i = 0; i < V; i++)
		printf("\t%d \t\t\t\t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
	int dist[V]; // The output array. dist[i] will hold the
				// shortest
	// distance from src to i

	bool sptSet[V]; // sptSet[i] will be true if vertex i is
					// included in shortest
	// path tree or shortest distance from src to i is
	// finalized

	// Initialize all distances as INFINITE and stpSet[] as
	// false
	for (int i = 0; i < V; i++)
		dist[i] = INT_MAX, sptSet[i] = false;

	// Distance of source vertex from itself is always 0
	dist[src] = 0;

	// Find shortest path for all vertices
	for (int count = 0; count < V - 1; count++) {
		// Pick the minimum distance vertex from the set of
```

```
		// vertices not yet processed. u is always equal to
		// src in the first iteration.
		int u = minDistance(dist, sptSet);

		// Mark the picked vertex as processed
		sptSet[u] = true;

		// Update dist value of the adjacent vertices of the
		// picked vertex.
		for (int v = 0; v < V; v++)

			// Update dist[v] only if is not in sptSet,
			// there is an edge from u to v, and total
			// weight of path from src to v through u is
			// smaller than current value of dist[v]
			if (!sptSet[v] && graph[u][v]
				&& dist[u] != INT_MAX
				&& dist[u] + graph[u][v] < dist[v])
				dist[v] = dist[u] + graph[u][v];
	}

	// print the constructed distance array
	printSolution(dist, V);
}

// driver program to test above function
int main()
{
	/* Let us create the example graph discussed above */
	int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
						{ 4, 0, 8, 0, 0, 0, 0, 11, 0 },
						{ 0, 8, 0, 7, 0, 4, 0, 0, 2 },
						{ 0, 0, 7, 0, 9, 14, 0, 0, 0 },
						{ 0, 0, 0, 9, 0, 10, 0, 0, 0 },
						{ 0, 0, 4, 14, 10, 0, 2, 0, 0 },
						{ 0, 0, 0, 0, 0, 2, 0, 1, 6 },
						{ 8, 11, 0, 0, 0, 0, 1, 0, 7 },
						{ 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

	dijkstra(graph, 0);

	return 0;
}

//*********************************************************************************************//
```

Topological Sorting

```
// A C++ program to print topological sorting of a DAG
#include <iostream>
#include <list>
#include <stack>
using namespace std;

// Class to represent a graph
class Graph {
```

```
	int V; // No. of vertices'

	// Pointer to an array containing adjacency listsList
	list<int>* adj;

	// A function used by topologicalSort
	void topologicalSortUtil(int v, bool visited[], stack<int>& Stack);

public:
	Graph(int V); // Constructor

	// function to add an edge to graph
	void addEdge(int v, int w);

	// prints a Topological Sort of the complete graph
	void topologicalSort();
};

Graph::Graph(int V)
{
	this->V = V;
	adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
	adj[v].push_back(w); // Add w to v’s list.
}

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[],
								stack<int>& Stack)
{
	// Mark the current node as visited.
	visited[v] = true;

	// Recur for all the vertices adjacent to this vertex
	list<int>::iterator i;
	for (i = adj[v].begin(); i != adj[v].end(); ++i)
		if (!visited[*i])
			topologicalSortUtil(*i, visited, Stack);

	// Push current vertex to stack which stores result
	Stack.push(v);
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
void Graph::topologicalSort()
{
	stack<int> Stack;

	// Mark all the vertices as not visited
	bool* visited = new bool[V];
	for (int i = 0; i < V; i++)
```

```
		visited[i] = false;

	// Call the recursive helper function to store Topological
	// Sort starting from all vertices one by one
	for (int i = 0; i < V; i++)
		if (visited[i] == false)
			topologicalSortUtil(i, visited, Stack);

	// Print contents of stack
	while (Stack.empty() == false) {
		cout << Stack.top() << " ";
		Stack.pop();
	}
}

// Driver program to test above functions
int main()
{
	// Create a graph given in the above diagram
	Graph g(6);
	g.addEdge(5, 2);
	g.addEdge(5, 0);
	g.addEdge(4, 0);
	g.addEdge(4, 1);
	g.addEdge(2, 3);
	g.addEdge(3, 1);

	cout << "Following is a Topological Sort of the given graph: ";
	g.topologicalSort();

	return 0;
}
//*********************************************************************************************//
```

is it bipartite graph

```
// C++ program to find out whether a given graph is Bipartite or not.
// Using recursion.
#include <iostream>

using namespace std;
#define V 4


bool colorGraph(int G[][V],int color[],int pos, int c){

	if(color[pos] != -1 && color[pos] !=c)
		return false;

	// color this pos as c and all its neighbours and 1-c
	color[pos] = c;
	bool ans = true;
	for(int i=0;i<V;i++){
		if(G[pos][i]){
			if(color[i] == -1)
				ans &= colorGraph(G,color,i,1-c);
```

```
				if(color[i] !=-1 && color[i] != 1-c)
					return false;
			}
			if (!ans)
				return false;
		}

		return true;
}

bool isBipartite(int G[][V]){
		int color[V];
		for(int i=0;i<V;i++)
			color[i] = -1;

		//start is vertex 0;
		int pos = 0;
		// two colors 1 and 0
		return colorGraph(G,color,pos,1);

}

int main()
{
		int G[][V] = {{0, 1, 0, 1},
			{1, 0, 1, 0},
			{0, 1, 0, 1},
			{1, 0, 1, 0}
		};

		isBipartite(G) ? cout<< "Yes" : cout << "No";
		return 0;
}
// This code is contributed By Mudit Verma
//**********************************************************************************************//
```