Graph

# 330. Implement BFS algorithm

Given a directed graph. The task is to do Breadth First Traversal of this graph starting from 0.  
**Note:**One can move from node u to node v only if there's an edge from u to v and find the BFS traversal of the graph starting from the 0th vertex, from left to right according to the graph. Also, you should only take nodes directly or indirectly connected from Node 0 in consideration.

**Example 1:**

**Input:**

**Output:** 0 1 2 3 4

**Explanation**:

0 is connected to 1 , 2 , 3.

2 is connected to 4.

so starting from 0, it will go to 1 then 2

then 3.After this 2 to 4, thus bfs will be

0 1 2 3 4.

**Example 2:**

**Input:**

**Output:** 0 1 2

**Explanation**:

0 is connected to 1 , 2.

so starting from 0, it will go to 1 then 2,

thus bfs will be 0 1 2 3 4.

**Your task:**  
You don’t need to read input or print anything. Your task is to complete the function **bfsOfGraph()** which takes the integer V denoting the number of vertices and adjacency list as input parameters and returns  a list containing the BFS traversal of the graph starting from the 0th vertex from left to right.

**Expected Time Complexity:**O(V + E)  
**Expected Auxiliary Space:**O(V)

**Constraints:**  
1 ≤ V, E ≤ 104

## Solution:

[Breadth-First Traversal (or Search)](http://en.wikipedia.org/wiki/Breadth-first_search) for a graph is similar to Breadth-First Traversal of a tree (See method 2 of [this post](https://www.geeksforgeeks.org/level-order-tree-traversal/)). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth-First Traversal of the following graph is 2, 0, 3, 1.



Following are the implementations of simple Breadth-First Traversal from a given source.   
The implementation uses an [adjacency list representation](http://en.wikipedia.org/wiki/Adjacency_list) of graphs. [STL](http://en.wikipedia.org/wiki/Standard_Template_Library)‘s [list container](http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#LIST) is used to store lists of adjacent nodes and the queue of nodes needed for BFS traversal.

// Program to print BFS traversal from a given

// source vertex. BFS(int s) traverses vertices

// reachable from s.

#include<iostream>

#include <list>

using namespace std;

// This class represents a directed graph using

// adjacency list representation

class Graph

{

int V; // No. of vertices

// Pointer to an array containing adjacency

// lists

list<int> \*adj;

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int v, int w);

// prints BFS traversal from a given source s

void BFS(int s);

};

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.

}

void Graph::BFS(int s)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

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

visited[i] = false;

// Create a queue for BFS

list<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push\_back(s);

// 'i' will be used to get all adjacent

// vertices of a vertex

list<int>::iterator i;

while(!queue.empty())

{

// Dequeue a vertex from queue and print it

s = queue.front();

cout << s << " ";

queue.pop\_front();

// Get all adjacent vertices of the dequeued

// vertex s. If a adjacent has not been visited,

// then mark it visited and enqueue it

for (i = adj[s].begin(); i != adj[s].end(); ++i)

{

if (!visited[\*i])

{

visited[\*i] = true;

queue.push\_back(\*i);

}

}

}

}

// Driver program to test methods of graph class

int main()

{

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Breadth First Traversal "

<< "(starting from vertex 2) \n";

g.BFS(2);

return 0;

}

**Output:**

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (for example Disconnected graph). To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the [DFS modified version](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)).

Time Complexity: O(V+E) where V is a number of vertices in the graph and E is a number of edges in the graph.

# 331. Implement DFS Algo

Given a connected undirected graph. Perform a Depth First Traversal of the graph.  
**Note:**Use recursive approach to find the DFS traversal of the graph starting from the 0th vertex from left to right according to the graph..

**Example 1:**

**Input:**

**Output:** 0 1 2 4 3

**Explanation**:

0 is connected to 1, 2, 4.

1 is connected to 0.

2 is connected to 0.

3 is connected to 4.

4 is connected to 0, 3.

so starting from 0, it will go to 1 then 2

then 4, and then from 4 to 3.

Thus dfs will be 0 1 2 4 3.

**Example 2:**

**Input:**

**Output:** 0 1 2 3

**Explanation**:

0 is connected to 1 , 3.

1 is connected to 2.

2 is connected to 1.

3 is connected to 0.

so starting from 0, it will go to 1 then 2

then back to 0 then 0 to 3

thus dfs will be 0 1 2 3.

**Your task:**  
You dont need to read input or print anything. Your task is to complete the function **dfsOfGraph()** which takes the integer V denoting the number of vertices and adjacency list as input parameters and returns  a list containing the DFS traversal of the graph starting from the 0th vertex from left to right according to the graph.

**Expected Time Complexity:**O(V + E)  
**Expected Auxiliary Space:**O(V)

**Constraints:**  
1 ≤ V, E ≤ 104

## Solution:

**Approach:**   
Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

**Algorithm:**   
Create a recursive function that takes the index of the node and a visited array.

1. Mark the current node as visited and print the node.
2. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

**Implementation:**  
Below are implementations of simple Depth First Traversal. The C++ implementation uses an adjacency list representation of graphs. STL’s list container is used to store lists of adjacent nodes.

// C++ program to print DFS traversal from

// a given vertex in a given graph

#include <bits/stdc++.h>

using namespace std;

// Graph class represents a directed graph

// using adjacency list representation

class Graph {

public:

map<int, bool> visited;

map<int, list<int> > adj;

// function to add an edge to graph

void addEdge(int v, int w);

// DFS traversal of the vertices

// reachable from v

void DFS(int v);

};

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::DFS(int v)

{

// Mark the current node as visited and

// print it

visited[v] = true;

cout << v << " ";

// 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])

DFS(\*i);

}

// Driver code

int main()

{

// Create a graph given in the above diagram

Graph g;

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Depth First Traversal"

" (starting from vertex 2) \n";

g.DFS(2);

return 0;

}

**Output:**

Following is Depth First Traversal (starting from vertex 2)

2 0 1 3

**Complexity Analysis:**

* **Time complexity:**O(V + E), where V is the number of vertices and E is the number of edges in the graph.
* **Space Complexity:** O(V), since an extra visited array of size V is required.

**Handling A Disconnected Graph:**

* **Solution:**   
  This will happen by handling a corner case.   
  The above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex, as in a Disconnected graph. To do a complete DFS traversal of such graphs, run DFS from all unvisited nodes after a DFS.   
  *The recursive function remains the same.*
* **Algorithm:**
  1. Create a recursive function that takes the index of the node and a visited array.
  2. Mark the current node as visited and print the node.
  3. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.
  4. Run a loop from 0 to the number of vertices and check if the node is unvisited in the previous DFS, call the recursive function with the current node.

**Implementation:**

// C++ program to print DFS

// traversal for a given given

// graph

#include <bits/stdc++.h>

using namespace std;

class Graph {

// A function used by DFS

void DFSUtil(int v);

public:

map<int, bool> visited;

map<int, list<int> > adj;

// function to add an edge to graph

void addEdge(int v, int w);

// prints DFS traversal of the complete graph

void DFS();

};

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::DFSUtil(int v)

{

// Mark the current node as visited and print it

visited[v] = true;

cout << v << " ";

// 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);

}

// The function to do DFS traversal. It uses recursive

// DFSUtil()

void Graph::DFS()

{

// Call the recursive helper function to print DFS

// traversal starting from all vertices one by one

for (auto i : adj)

if (visited[i.first] == false)

DFSUtil(i.first);

}

// Driver Code

int main()

{

// Create a graph given in the above diagram

Graph g;

g.addEdge(0, 1);

g.addEdge(0, 9);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(9, 3);

cout << "Following is Depth First Traversal \n";

g.DFS();

return 0;

}

**Output:**

Following is Depth First Traversal

0 1 2 3 9

**Complexity Analysis:**

* **Time complexity:**O(V + E), where V is the number of vertices and E is the number of edges in the graph.
* **Space Complexity:**O(V), since an extra visited array of size V is required.

# 332. [Detect Cycle in Directed Graph using BFS/DFS Algo](https://www.geeksforgeeks.org/detect-cycle-in-a-graph/)

Given a Directed Graph with **V** vertices (Numbered from **0** to **V-1**) and **E** edges, check whether it contains any cycle or not.

**Example 1:**

**Input:**

**Output:** 1

**Explanation**: 3 -> 3 is a cycle

**Example 2:**

**Input:**

**Output:** 0

**Explanation**: no cycle in the graph

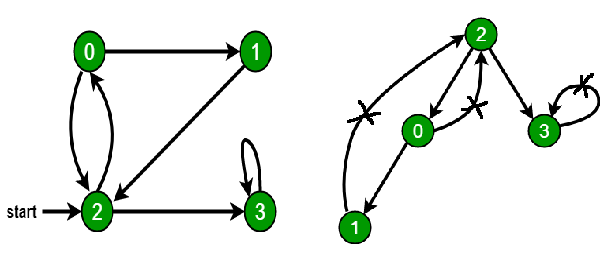
**Your task:**  
You don’t need to read input or print anything. Your task is to complete the function **isCyclic()** which takes the integer V denoting the number of vertices and adjacency list as input parameters and returns a boolean value denoting if the given directed graph contains a cycle or not.

**Expected Time Complexity:**O(V + E)  
**Expected Auxiliary Space:**O(V)

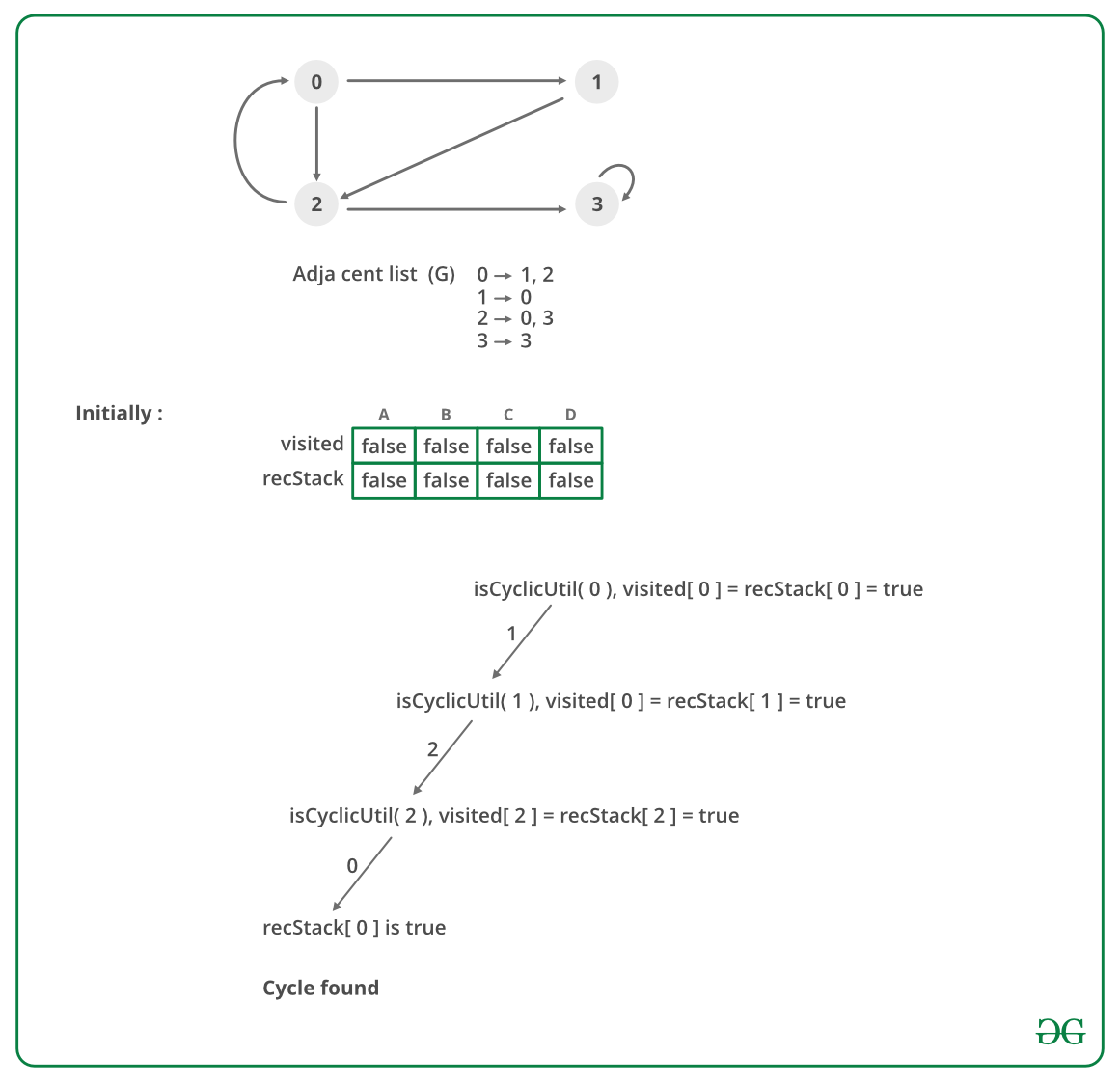
**Constraints:**  
1 ≤ V, E ≤ 105

## Solution:

* **Approach:** Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a [back edge](http://en.wikipedia.org/wiki/Depth-first_search#Output_of_a_depth-first_search) present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestors in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.



* For a disconnected graph, Get the DFS forest as output. To detect cycle, check for a cycle in individual trees by checking back edges.  
  To detect a back edge, keep track of vertices currently in the recursion stack of function for DFS traversal. If a vertex is reached that is already in the recursion stack, then there is a cycle in the tree. The edge that connects the current vertex to the vertex in the recursion stack is a back edge. Use**recStack[]** array to keep track of vertices in the recursion stack.  
  **Dry run of the above approach:**



In the above image there is a mistake node 1 is making a directed edge to 2 not with 0 please make a note.

* **Algorithm:**
  1. Create the graph using the given number of edges and vertices.
  2. Create a recursive function that initializes the current index or vertex, visited, and recursion stack.
  3. Mark the current node as visited and also mark the index in recursion stack.
  4. Find all the vertices which are not visited and are adjacent to the current node. Recursively call the function for those vertices, If the recursive function returns true, return true.
  5. If the adjacent vertices are already marked in the recursion stack then return true.
  6. Create a wrapper class, that calls the recursive function for all the vertices and if any function returns true return true. Else if for all vertices the function returns false return false.

**Implementation:**

// A C++ Program to detect cycle in a graph

#include<bits/stdc++.h>

using namespace std;

class Graph

{

int V; // No. of vertices

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

bool isCyclicUtil(int v, bool visited[], bool \*rs); // used by isCyclic()

public:

Graph(int V); // Constructor

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

bool isCyclic(); // returns true if there is a cycle in this graph

};

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.

}

// This function is a variation of DFSUtil() in https://www.geeksforgeeks.org/archives/18212

bool Graph::isCyclicUtil(int v, bool visited[], bool \*recStack)

{

if(visited[v] == false)

{

// Mark the current node as visited and part of recursion stack

visited[v] = true;

recStack[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] && isCyclicUtil(\*i, visited, recStack) )

return true;

else if (recStack[\*i])

return true;

}

}

recStack[v] = false; // remove the vertex from recursion stack

return false;

}

// Returns true if the graph contains a cycle, else false.

// This function is a variation of DFS() in https://www.geeksforgeeks.org/archives/18212

bool Graph::isCyclic()

{

// Mark all the vertices as not visited and not part of recursion

// stack

bool \*visited = new bool[V];

bool \*recStack = new bool[V];

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

{

visited[i] = false;

recStack[i] = false;

}

// Call the recursive helper function to detect cycle in different

// DFS trees

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

if (isCyclicUtil(i, visited, recStack))

return true;

return false;

}

int main()

{

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

if(g.isCyclic())

cout << "Graph contains cycle";

else

cout << "Graph doesn't contain cycle";

return 0;

}

**Output:**

Graph contains cycle

* **Complexity Analysis:**
  + **Time Complexity:**O(V+E).   
    Time Complexity of this method is same as time complexity of [DFS traversal](https://www.geeksforgeeks.org/archives/18212) which is O(V+E).
  + **Space Complexity:** O(V).   
    To store the visited and recursion stack O(V) space is needed.

Steps involved in detecting cycle in a directed graph using BFS.  
**Step-1:** Compute in-degree (number of incoming edges) for each of the vertex present in the graph and initialize the count of visited nodes as 0.  
**Step-2:**Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)  
**Step-3:** Remove a vertex from the queue (Dequeue operation) and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighboring nodes.
3. If in-degree of a neighboring nodes is reduced to zero, then add it to the queue.

**Step 4:** Repeat Step 3 until the queue is empty.  
**Step 5:**If count of visited nodes is **not** equal to the number of nodes in the graph has cycle, otherwise not.

**How to find in-degree of each node?**   
There are 2 ways to calculate in-degree of every vertex:   
Take an in-degree array which will keep track of   
**1)** Traverse the array of edges and simply increase the counter of the destination node by 1.

for each node in Nodes

indegree[node] = 0;

for each edge(src,dest) in Edges

indegree[dest]++

Time Complexity: O(V+E)

**2)** Traverse the list for every node and then increment the in-degree of all the nodes connected to it by 1.

for each node in Nodes

If (list[node].size()!=0) then

for each dest in list

indegree[dest]++;

Time Complexity: The outer for loop will be executed V number of times and the inner for loop will be executed E number of times, Thus overall time complexity is O(V+E).

The overall time complexity of the algorithm is O(V+E)

// A C++ program to check if there is a cycle in

// directed graph using BFS.

#include <bits/stdc++.h>

using namespace std;

// Class to represent a graph

class Graph {

int V; // No. of vertices'

// Pointer to an array containing adjacency list

list<int>\* adj;

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int u, int v);

// Returns true if there is a cycle in the graph

// else false.

bool isCycle();

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int u, int v)

{

adj[u].push\_back(v);

}

// This function returns true if there is a cycle

// in directed graph, else returns false.

bool Graph::isCycle()

{

// Create a vector to store indegrees of all

// vertices. Initialize all indegrees as 0.

vector<int> in\_degree(V, 0);

// Traverse adjacency lists to fill indegrees of

// vertices. This step takes O(V+E) time

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

for (auto v : adj[u])

in\_degree[v]++;

}

// Create an queue and enqueue all vertices with

// indegree 0

queue<int> q;

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

if (in\_degree[i] == 0)

q.push(i);

// Initialize count of visited vertices

int cnt = q.size();

// Create a vector to store result (A topological

// ordering of the vertices)

vector<int> top\_order;

// One by one dequeue vertices from queue and enqueue

// adjacents if indegree of adjacent becomes 0

while (!q.empty()) {

// Extract front of queue (or perform dequeue)

// and add it to topological order

int u = q.front();

q.pop();

top\_order.push\_back(u);

// Iterate through all its neighbouring nodes

// of dequeued node u and decrease their in-degree

// by 1

list<int>::iterator itr;

for (itr = adj[u].begin(); itr != adj[u].end(); itr++)

// If in-degree becomes zero, add it to queue

if (--in\_degree[\*itr] == 0)

{

q.push(\*itr);

//while we are pushing elements to the queue we will incrementing the cnt

cnt++;

}

}

// Check if there was a cycle

if (cnt != V)

return true;

else

return false;

}

// Driver program to test above functions

int main()

{

// Create a graph given in the above diagram

Graph g(6);

g.addEdge(0, 1);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(3, 4);

g.addEdge(4, 5);

if (g.isCycle())

cout << "Yes";

else

cout << "No";

return 0;

}

**Output:**

Yes

**Time Complexity:** O(V+E)

In **Topological Sort**, the idea is to visit the **parent node** followed by the **child node**. If the given graph contains a **cycle**, then there is at least **one node** which is a parent as well as a child so this will break **Topological Order**. Therefore, after the **topological sort**, check for every **directed edge** whether it follows the order or not.

Below is the implementation of the above approach:

// C++ Program to implement

// the above approach

#include <bits/stdc++.h>

using namespace std;

// n -> is number of edges in graph

// m -> is number of node in graph

int n, m ;

// Stack to store the

// visited vertices in

// the Topological Sort

stack<int> s;

// Store Topological Order

vector<int> tsort;

// Adjacency list to store edges

vector<int> adj[int(1e5) + 1];

// To ensure visited vertex

vector<int> visited(int(1e5) + 1);

// Function to perform DFS

void dfs(int u)

{

// Set the vertex as visited

visited[u] = 1;

for (auto it : adj[u]) {

// Visit connected vertices

if (visited[it] == 0)

dfs(it);

}

// Push into the stack on

// complete visit of vertex

s.push(u);

}

// Function to check and return

// if a cycle exists or not

bool check\_cycle()

{

// Stores the position of

// vertex in topological order

unordered\_map<int, int> pos;

int index = 0;

// Pop all elements from stack

while (!s.empty()) {

pos[s.top()] = index;

// Push element to get

// Topological Order

tsort.push\_back(s.top());

index += 1;

// Pop from the stack

s.pop();

}

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

for (auto it : adj[i]) {

// If parent vertex

// does not appear first

if (pos[i] > pos[it]) {

// Cycle exists

return true;

}

}

}

// Return false if cycle

// does not exist

return false;

}

// Function to add edges

// from u -> v

void addEdge(int u, int v)

{

adj[u].push\_back(v);

}

// Driver Code

int main()

{

n = 4, m = 5;

// Insert edges

addEdge(0, 1);

addEdge(0, 2);

addEdge(1, 2);

addEdge(2, 0);

addEdge(2, 3);

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

if (visited[i] == 0) {

dfs(i);

}

}

// If cycle exist

if (check\_cycle())

cout << "Yes";

else

cout << "No";

return 0;

}

**Output:**

Yes

***Time Complexity:****O(N + M)*  
***Auxiliary Space:****O(N)*

# 333. [Detect Cycle in UnDirected Graph using BFS/DFS Algo](https://practice.geeksforgeeks.org/problems/detect-cycle-in-an-undirected-graph/1)

Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.

**Example 1:**

**Input:**

**Output:** 1

**Explanation:** 1->2->3->4->1 is a cycle.

**Example 2:**

**Input:**

**Output:** 0

**Explanation:** No cycle in the graph.

**Your Task:**  
You don't need to read or print anything. Your task is to complete the function **isCycle()**which takes V denoting the number of vertices and adjacency list as input parameters and returns a boolean value denoting if the undirected graph contains any cycle or not, return 1 if a cycle is present else return 0.

**NOTE:**The adjacency list denotes the edges of the graph where edges[i][0] and edges[i][1] represent an edge.

**Expected Time Complexity:**O(V + E)  
**Expected Space Complexity:**O(V)

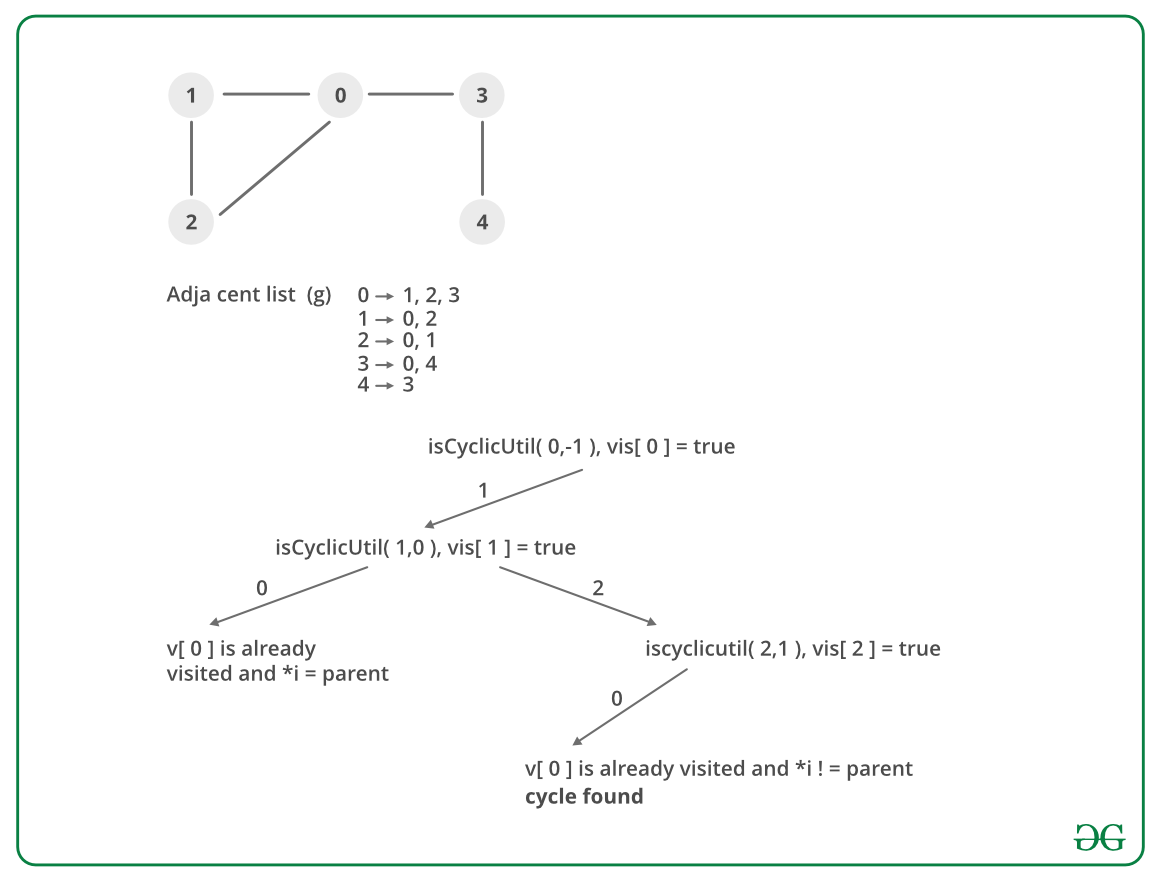
**Constraints:**  
1 ≤ V, E ≤ 105

## Solution:

**Approach:** Run a DFS from every unvisited node. [Depth First Traversal](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/) can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is joining a node to itself (self-loop) or one of its ancestor in the tree produced by DFS.   
To find the back edge to any of its ancestors keep a visited array and if there is a back edge to any visited node then there is a loop and return true.  
**Algorithm:**

1. Create the graph using the given number of edges and vertices.
2. Create a recursive function that have current index or vertex, visited array and parent node.
3. Mark the current node as visited .
4. Find all the vertices which are not visited and are adjacent to the current node. Recursively call the function for those vertices, If the recursive function returns true return true.
5. If the adjacent node is not parent and already visited then return true.
6. Create a wrapper class, that calls the recursive function for all the vertices and if any function returns true, return true.
7. Else if for all vertices the function returns false return false.

**Dry Run:**



**Implementation:**

// A C++ Program to detect

// cycle in an undirected graph

#include<iostream>

#include <list>

#include <limits.h>

using namespace std;

// Class for an undirected graph

class Graph

{

// No. of vertices

int V;

// Pointer to an array

// containing adjacency lists

list<int> \*adj;

bool isCyclicUtil(int v, bool visited[],

int parent);

public:

// Constructor

Graph(int V);

// To add an edge to graph

void addEdge(int v, int w);

// Returns true if there is a cycle

bool isCyclic();

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

// Add w to v’s list.

adj[v].push\_back(w);

// Add v to w’s list.

adj[w].push\_back(v);

}

// A recursive function that

// uses visited[] and parent to detect

// cycle in subgraph reachable

// from vertex v.

bool Graph::isCyclicUtil(int v,

bool visited[], int parent)

{

// 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 an adjacent vertex is not visited,

//then recur for that adjacent

if (!visited[\*i])

{

if (isCyclicUtil(\*i, visited, v))

return true;

}

// If an adjacent vertex is visited and

// is not parent of current vertex,

// then there exists a cycle in the graph.

else if (\*i != parent)

return true;

}

return false;

}

// Returns true if the graph contains

// a cycle, else false.

bool Graph::isCyclic()

{

// Mark all the vertices as not

// visited and not part of recursion

// stack

bool \*visited = new bool[V];

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

visited[i] = false;

// Call the recursive helper

// function to detect cycle in different

// DFS trees

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

{

// Don't recur for u if

// it is already visited

if (!visited[u])

if (isCyclicUtil(u, visited, -1))

return true;

}

return false;

}

// Driver program to test above functions

int main()

{

Graph g1(5);

g1.addEdge(1, 0);

g1.addEdge(0, 2);

g1.addEdge(2, 1);

g1.addEdge(0, 3);

g1.addEdge(3, 4);

g1.isCyclic()?

cout << "Graph contains cycle\n":

cout << "Graph doesn't contain cycle\n";

Graph g2(3);

g2.addEdge(0, 1);

g2.addEdge(1, 2);

g2.isCyclic()?

cout << "Graph contains cycle\n":

cout << "Graph doesn't contain cycle\n";

return 0;

}

**Output:**

Graph contains cycle

Graph doesn't contain cycle

**Complexity Analysis:**

* **Time Complexity:** O(V+E).   
  The program does a simple DFS Traversal of the graph which is represented using adjacency list. So the time complexity is O(V+E).
* **Space Complexity:** O(V).   
  To store the visited array O(V) space is required.

In this article, the [BFS](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/) based solution is discussed. We do a BFS traversal of the given graph. For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not a parent of v, then there is a cycle in the graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle.   
We use a parent array to keep track of the parent vertex for a vertex so that we do not consider the visited parent as a cycle.

// C++ program to detect cycle

// in an undirected graph

// using BFS.

#include <bits/stdc++.h>

using namespace std;

void addEdge(vector<int> adj[], int u, int v)

{

adj[u].push\_back(v);

adj[v].push\_back(u);

}

bool isCyclicConntected(vector<int> adj[], int s,

int V, vector<bool>& visited)

{

// Set parent vertex for every vertex as -1.

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

// Create a queue for BFS

queue<int> q;

// Mark the current node as

// visited and enqueue it

visited[s] = true;

q.push(s);

while (!q.empty()) {

// Dequeue a vertex from queue and print it

int u = q.front();

q.pop();

// Get all adjacent vertices of the dequeued

// vertex u. If a adjacent has not been visited,

// then mark it visited and enqueue it. We also

// mark parent so that parent is not considered

// for cycle.

for (auto v : adj[u]) {

if (!visited[v]) {

visited[v] = true;

q.push(v);

parent[v] = u;

}

else if (parent[u] != v)

return true;

}

}

return false;

}

bool isCyclicDisconntected(vector<int> adj[], int V)

{

// Mark all the vertices as not visited

vector<bool> visited(V, false);

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

if (!visited[i] && isCyclicConntected(adj, i,

V, visited))

return true;

return false;

}

// Driver program to test methods of graph class

int main()

{

int V = 4;

vector<int> adj[V];

addEdge(adj, 0, 1);

addEdge(adj, 1, 2);

addEdge(adj, 2, 0);

addEdge(adj, 2, 3);

if (isCyclicDisconntected(adj, V))

cout << "Yes";

else

cout << "No";

return 0;

}

**Output:**

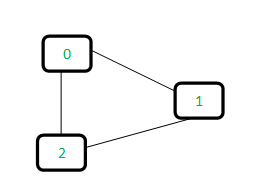
Yes

**Time Complexity:** The program does a simple BFS Traversal of graph and graph is represented using adjacency list. So the time complexity is O(V+E)

**Using Disjoint set data structure:**

*Union-Find Algorithm* can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an [algorithm to detect cycle](http://www.geeksforgeeks.org/archives/18516). This is another method based on *Union-Find*. This method assumes that the graph doesn’t contain any self-loops.

We can keep track of the subsets in a 1D array, let’s call it parent[].  
Let us consider the following graph:



For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.

Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

0 1 2

-1 -1 -1

Now process all edges one by one.  
*Edge 0-1:* Find the subsets in which vertices 0 and 1 are. Since they are in different subsets, we take the union of them. For taking the union, either make node 0 as parent of node 1 or vice-versa.

0 1 2 <----- 1 is made parent of 0 (1 is now representative of subset {0, 1})

1 -1 -1

*Edge 1-2:* 1 is in subset 1 and 2 is in subset 2. So, take union.

0 1 2 <----- 2 is made parent of 1 (2 is now representative of subset {0, 1, 2})

1 2 -1

*Edge 0-2:* 0 is in subset 2 and 2 is also in subset 2. Hence, including this edge forms a cycle.  
How subset of 0 is same as 2?   
0->1->2 // 1 is parent of 0 and 2 is parent of 1

Based on the above explanation, below are implementations:

// A union-find algorithm to detect cycle in a graph

#include <bits/stdc++.h>

using namespace std;

// a structure to represent an edge in graph

class Edge

{

public:

int src, dest;

};

// a structure to represent a graph

class Graph

{

public:

// 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

Graph\* createGraph(int V, int E)

{

Graph\* graph = new Graph();

graph->V = V;

graph->E = E;

graph->edge = new Edge[graph->E \* sizeof(Edge)];

return graph;

}

// A utility function to find the subset of an element i

int find(int parent[], int i)

{

if (parent[i] == -1)

return i;

return find(parent, parent[i]);

}

// A utility function to do union of two subsets

void Union(int parent[], int x, int y)

{

parent[x] = y;

}

// The main function to check whether a given graph contains

// cycle or not

int isCycle(Graph\* graph)

{

// Allocate memory for creating V subsets

int\* parent = new int[graph->V \* sizeof(int)];

// Initialize all subsets as single element sets

memset(parent, -1, sizeof(int) \* graph->V);

// Iterate through all edges of graph, find subset of

// both vertices of every edge, if both subsets are

// same, then there is cycle in graph.

for (int i = 0; i < graph->E; ++i) {

int x = find(parent, graph->edge[i].src);

int y = find(parent, graph->edge[i].dest);

if (x == y)

return 1;

Union(parent, x, y);

}

return 0;

}

// Driver code

int main()

{

/\* Let us create the following graph

0

| \

| \

1---2 \*/

int V = 3, E = 3;

Graph\* graph = createGraph(V, E);

// add edge 0-1

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

// add edge 1-2

graph->edge[1].src = 1;

graph->edge[1].dest = 2;

// add edge 0-2

graph->edge[2].src = 0;

graph->edge[2].dest = 2;

if (isCycle(graph))

cout << "graph contains cycle";

else

cout << "graph doesn't contain cycle";

return 0;

}

**Output:**

graph contains cycle

Note that the implementation of *union()* and *find()* is naive and takes O(n) time in the worst case. These methods can be improved to O(Logn) using *Union by Rank or Height*. We will soon be discussing *Union by Rank* in a separate post.

# 334. Search in a Maze

## Same as ques 254 of backtracking

# 335. Minimum Step by Knight

Given a square chessboard, the initial position of Knight and position of a target. Find out the minimum steps a Knight will take to reach the target position.

**Note:**  
The initial and the target position co-ordinates of Knight have been given accoring to 1-base indexing.

**Example 1:**

**Input:**

N=6

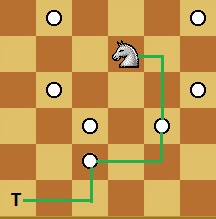
knightPos[ ] = {4, 5}

targetPos[ ] = {1, 1}

**Output:**

3

**Explanation:**



Knight takes 3 step to reach from

(4, 5) to (1, 1):

(4, 5) -> (5, 3) -> (3, 2) -> (1, 1).

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **minStepToReachTarget()** which takes the inital position of Knight (KnightPos), the target position of Knight (TargetPos) and the size of the chess board (N) as an input parameters and returns the minimum number of steps required by the knight to reach from its current position to the given target position.

**Expected Time Complexity:** O(N2).  
**Expected Auxiliary Space:** O(N2).

**Constraints:**  
1 <= N <= 1000  
1 <= Knight\_pos(X, Y), Targer\_pos(X, Y) <= N

## Solution:

**Approach:**   
This problem can be seen as shortest path in unweighted graph. Therefore we use [BFS](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/) to solve this problem. We try all 8 possible positions where a Knight can reach from its position. If reachable position is not already visited and is inside the board, we push this state into queue with distance 1 more than its parent state. Finally we return distance of target position, when it gets pop out from queue.  
Below code implements BFS for searching through cells, where each cell contains its coordinate and distance from starting node. In worst case, below code visits all cells of board, making worst-case time complexity as O(N^2)

// C++ program to find minimum steps to reach to

// specific cell in minimum moves by Knight

#include <bits/stdc++.h>

using namespace std;

// structure for storing a cell's data

struct cell {

int x, y;

int dis;

cell() {}

cell(int x, int y, int dis)

: x(x), y(y), dis(dis)

{

}

};

// Utility method returns true if (x, y) lies

// inside Board

bool isInside(int x, int y, int N)

{

if (x >= 1 && x <= N && y >= 1 && y <= N)

return true;

return false;

}

// Method returns minimum step

// to reach target position

int minStepToReachTarget(

int knightPos[], int targetPos[],

int N)

{

// x and y direction, where a knight can move

int dx[] = { -2, -1, 1, 2, -2, -1, 1, 2 };

int dy[] = { -1, -2, -2, -1, 1, 2, 2, 1 };

// queue for storing states of knight in board

queue<cell> q;

// push starting position of knight with 0 distance

q.push(cell(knightPos[0], knightPos[1], 0));

cell t;

int x, y;

bool visit[N + 1][N + 1];

// make all cell unvisited

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

for (int j = 1; j <= N; j++)

visit[i][j] = false;

// visit starting state

visit[knightPos[0]][knightPos[1]] = true;

// loop until we have one element in queue

while (!q.empty()) {

t = q.front();

q.pop();

// if current cell is equal to target cell,

// return its distance

if (t.x == targetPos[0] && t.y == targetPos[1])

return t.dis;

// loop for all reachable states

for (int i = 0; i < 8; i++) {

x = t.x + dx[i];

y = t.y + dy[i];

// If reachable state is not yet visited and

// inside board, push that state into queue

if (isInside(x, y, N) && !visit[x][y]) {

visit[x][y] = true;

q.push(cell(x, y, t.dis + 1));

}

}

}

}

// Driver code to test above methods

int main()

{

int N = 30;

int knightPos[] = { 1, 1 };

int targetPos[] = { 30, 30 };

cout << minStepToReachTarget(knightPos, targetPos, N);

return 0;

}

**Output:** 

20

**Complexity Analysis:** 

* **Time complexity:** O(N^2).   
  At worst case, all the cells will be visited so time complexity is O(N^2).
* **Auxiliary Space:** O(N^2).   
  The nodes are stored in queue. So the space Complexity is O(N^2).

# 336. Flood Fill

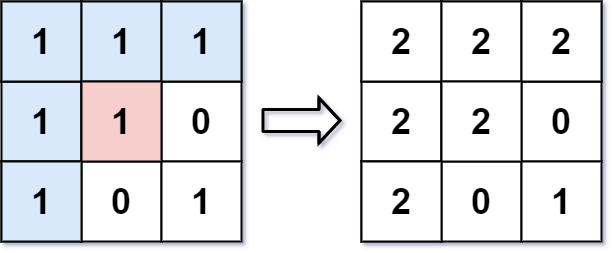
An image is represented by an m x n integer grid image where image[i][j] represents the pixel value of the image.

You are also given three integers sr, sc, and newColor. You should perform a **flood fill** on the image starting from the pixel image[sr][sc].

To perform a **flood fill**, consider the starting pixel, plus any pixels connected **4-directionally** to the starting pixel of the same color as the starting pixel, plus any pixels connected **4-directionally** to those pixels (also with the same color), and so on. Replace the color of all of the aforementioned pixels with newColor.

Return *the modified image after performing the flood fill*.

**Example 1:**



**Input:** image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, newColor = 2

**Output:** [[2,2,2],[2,2,0],[2,0,1]]

**Explanation:** From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.

Note the bottom corner is not colored 2, because it is not 4-directionally connected to the starting pixel.

**Example 2:**

**Input:** image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, newColor = 2

**Output:** [[2,2,2],[2,2,2]]

**Constraints:**

* m == image.length
* n == image[i].length
* 1 <= m, n <= 50
* 0 <= image[i][j], newColor < 216
* 0 <= sr < m
* 0 <= sc < n

## Solution:

**Intuition**

We perform the algorithm explained in the problem description: paint the starting pixels, plus adjacent pixels of the same color, and so on.

**Algorithm**

Say color is the color of the starting pixel. Let's floodfill the starting pixel: we change the color of that pixel to the new color, then check the 4 neighboring pixels to make sure they are valid pixels of the same color, and of the valid ones, we floodfill those, and so on.

We can use a function dfs to perform a floodfill on a target pixel.

class Solution {

public int[][] floodFill(int[][] image, int sr, int sc, int newColor) {

int color = image[sr][sc];

if (color != newColor) dfs(image, sr, sc, color, newColor);

return image;

}

public void dfs(int[][] image, int r, int c, int color, int newColor) {

if (image[r][c] == color) {

image[r][c] = newColor;

if (r >= 1) dfs(image, r-1, c, color, newColor);

if (c >= 1) dfs(image, r, c-1, color, newColor);

if (r+1 < image.length) dfs(image, r+1, c, color, newColor);

if (c+1 < image[0].length) dfs(image, r, c+1, color, newColor);

}

}

}

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of pixels in the image. We might process every pixel.
* Space Complexity: O(N)*O*(*N*), the size of the implicit call stack when calling dfs.

**My Implementation using BFS:**

class Solution {

public:

vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor) {

int color = image[sr][sc], m = image.size(), n = image[0].size();

if(color==newColor)

return image;

queue<pair<int, int>> q;

q.push({sr,sc});

image[sr][sc] = newColor;

while(!q.empty()){

int i = q.front().first, j = q.front().second;

q.pop();

if(i-1>=0 && image[i-1][j]==color){

image[i-1][j] = newColor;

q.push({i-1, j});

}

if(i+1<m && image[i+1][j]==color){

image[i+1][j] = newColor;

q.push({i+1, j});

}

if(j-1>=0 && image[i][j-1]==color){

image[i][j-1] = newColor;

q.push({i, j-1});

}

if(j+1<n && image[i][j+1]==color){

image[i][j+1] = newColor;

q.push({i, j+1});

}

}

return image;

}

};

# 337. [Clone a graph](https://leetcode.com/problems/clone-graph/)

Given a reference of a node in a [**connected**](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)#Connected_graph) undirected graph.

Return a [**deep copy**](https://en.wikipedia.org/wiki/Object_copying#Deep_copy) (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

class Node {

public int val;

public List<Node> neighbors;

}

**Test case format:**

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with val == 1, the second node with val == 2, and so on. The graph is represented in the test case using an adjacency list.

**An adjacency list** is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with val = 1. You must return the **copy of the given node** as a reference to the cloned graph.

**Example 1:**



**Input:** adjList = [[2,4],[1,3],[2,4],[1,3]]

**Output:** [[2,4],[1,3],[2,4],[1,3]]

**Explanation:** There are 4 nodes in the graph.

1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

**Example 2:**



**Input:** adjList = [[]]

**Output:** [[]]

**Explanation:** Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.

**Example 3:**

**Input:** adjList = []

**Output:** []

**Explanation:** This an empty graph, it does not have any nodes.

**Constraints:**

* The number of nodes in the graph is in the range [0, 100].
* 1 <= Node.val <= 100
* Node.val is unique for each node.
* There are no repeated edges and no self-loops in the graph.
* The Graph is connected and all nodes can be visited starting from the given node.

## Solution:

**My Implementation:**

Use DFS concept and hashtable. Keep making node and their adjacency list.

class Solution {

public:

Node\* fun(Node\* node, unordered\_map<int, Node\*> &mp){

Node\* nd = new Node(node->val);

mp[node->val] = nd;

for(Node\* nb: node->neighbors){

if(mp.find(nb->val)!=mp.end()){

nd->neighbors.push\_back(mp[nb->val]);

}

else{

nd->neighbors.push\_back(fun(nb, mp));

}

}

return nd;

}

Node\* cloneGraph(Node\* node) {

if(!node)

return node;

unordered\_map<int, Node\*> mp;

return fun(node, mp);

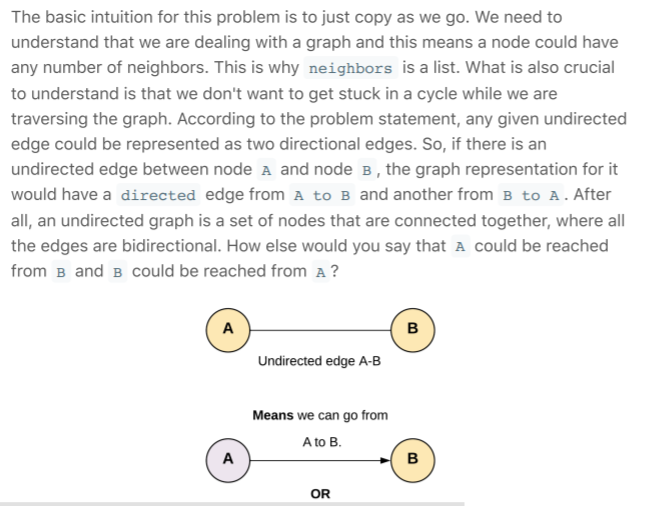
}

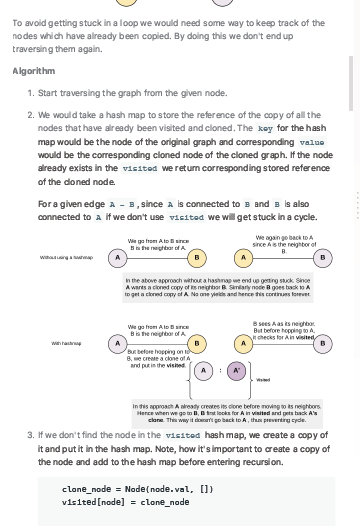
};

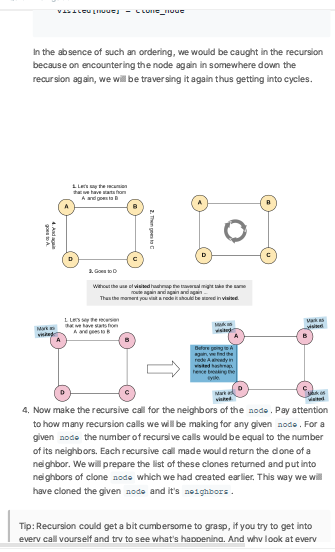
**Time Complexity:** O(V+E)

**Space Complexity:** O(V)









/\*

// Definition for a Node.

class Node {

public int val;

public List<Node> neighbors;

public Node() {}

public Node(int \_val,List<Node> \_neighbors) {

val = \_val;

neighbors = \_neighbors;

}

};

\*/

class Solution {

private HashMap <Node, Node> visited = new HashMap <> ();

public Node cloneGraph(Node node) {

if (node == null) {

return node;

}

// If the node was already visited before.

// Return the clone from the visited dictionary.

if (visited.containsKey(node)) {

return visited.get(node);

}

// Create a clone for the given node.

// Note that we don't have cloned neighbors as of now, hence [].

Node cloneNode = new Node(node.val, new ArrayList());

// The key is original node and value being the clone node.

visited.put(node, cloneNode);

// Iterate through the neighbors to generate their clones

// and prepare a list of cloned neighbors to be added to the cloned node.

for (Node neighbor: node.neighbors) {

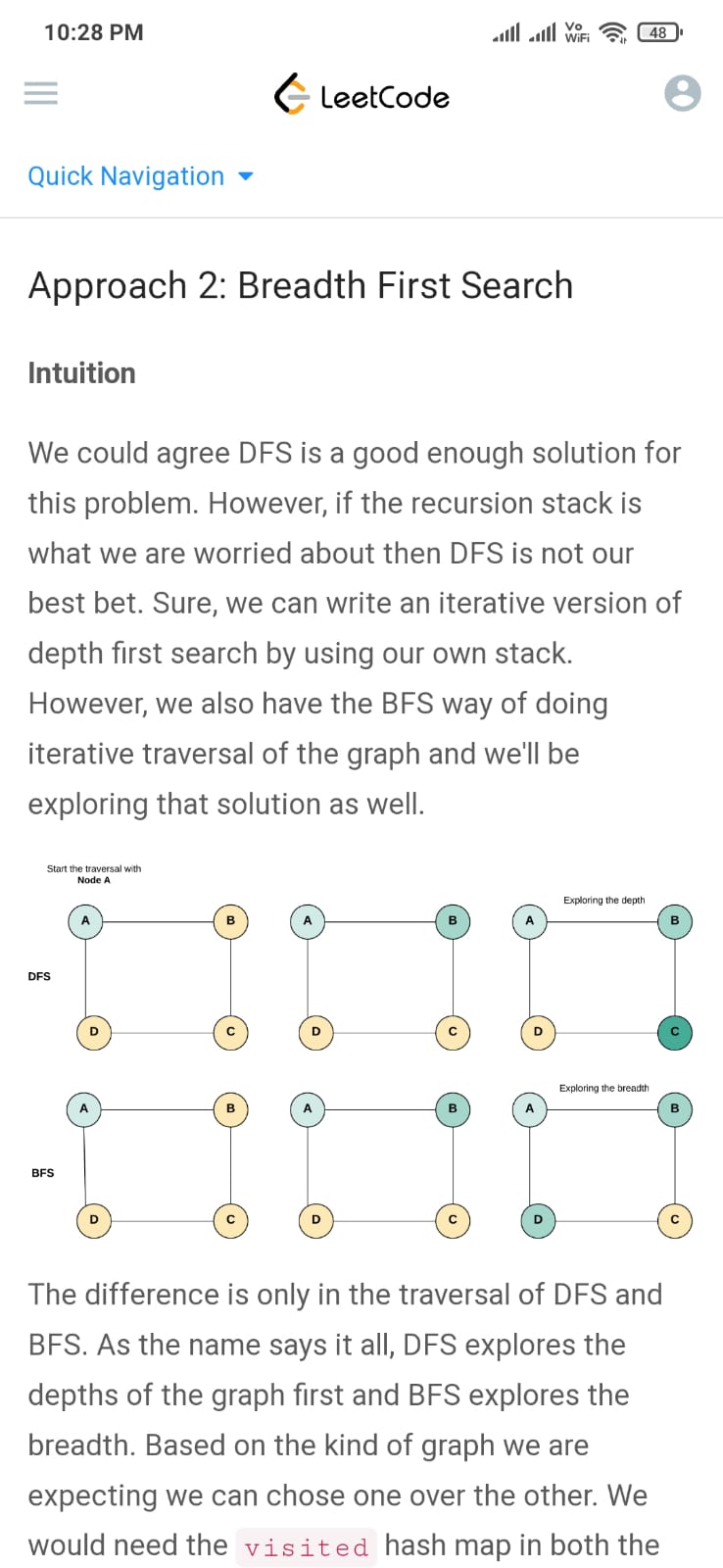
cloneNode.neighbors.add(cloneGraph(neighbor));

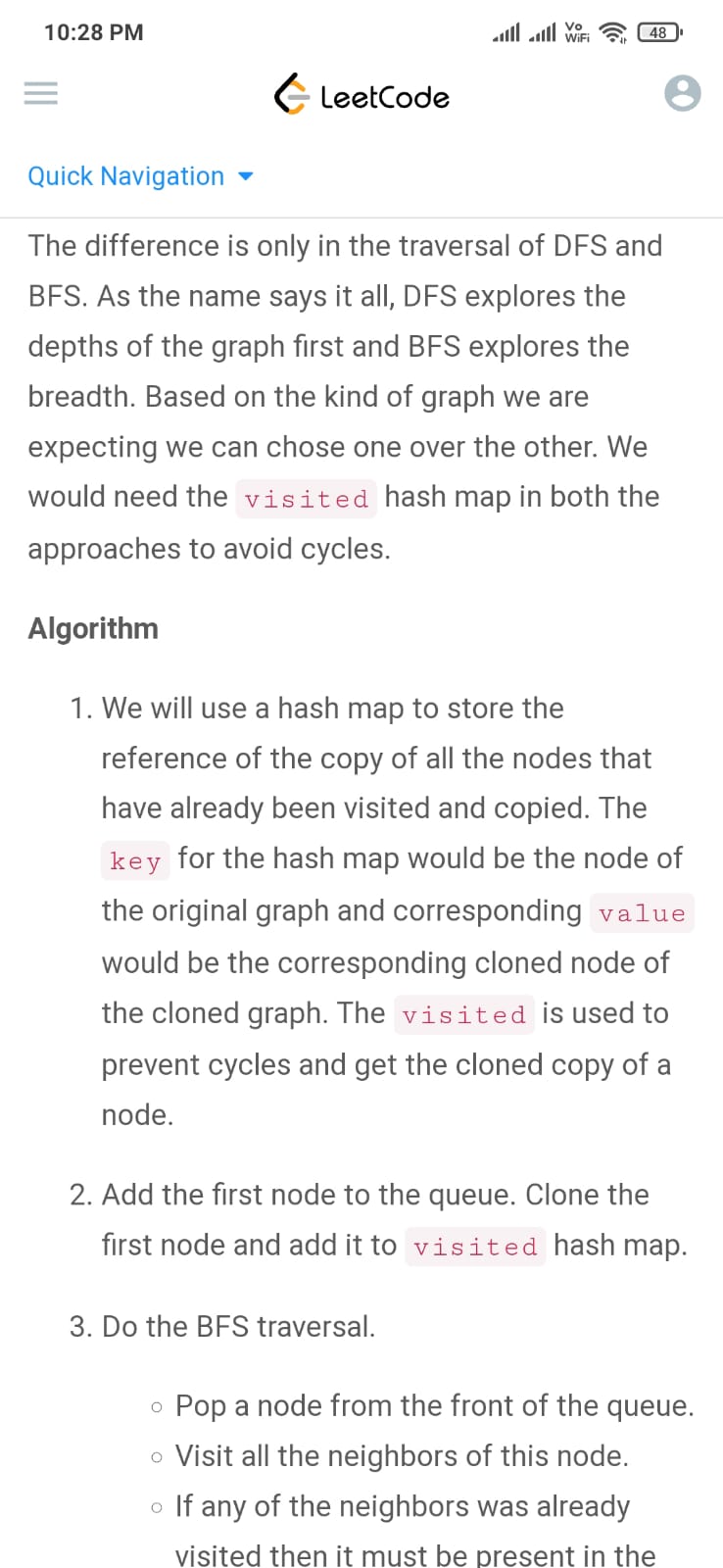
}

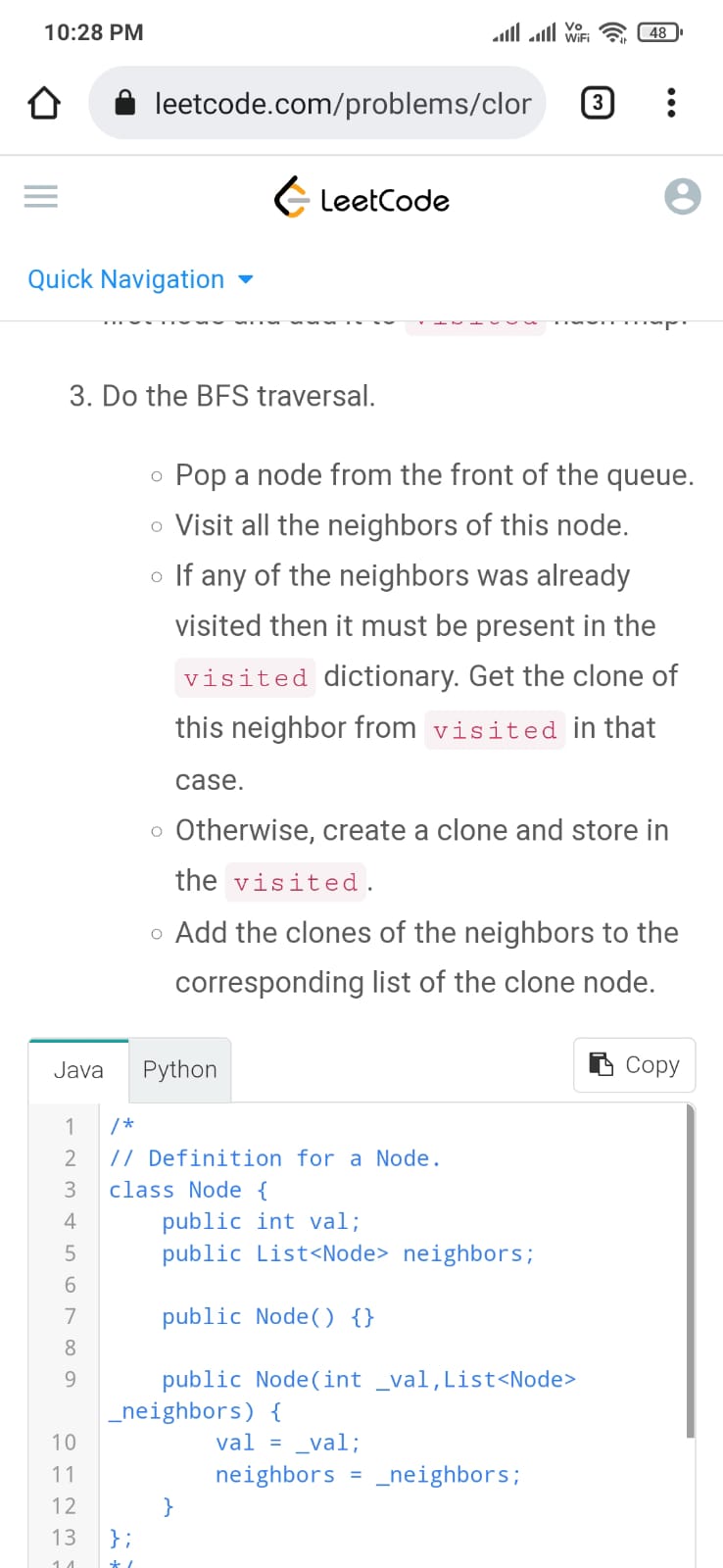
return cloneNode;

}

}







/\*

// Definition for a Node.

class Node {

public int val;

public List<Node> neighbors;

public Node() {}

public Node(int \_val,List<Node> \_neighbors) {

val = \_val;

neighbors = \_neighbors;

}

};

\*/

class Solution {

public Node cloneGraph(Node node) {

if (node == null) {

return node;

}

// Hash map to save the visited node and it's respective clone

// as key and value respectively. This helps to avoid cycles.

HashMap<Node, Node> visited = new HashMap();

// Put the first node in the queue

LinkedList<Node> queue = new LinkedList<Node> ();

queue.add(node);

// Clone the node and put it in the visited dictionary.

visited.put(node, new Node(node.val, new ArrayList()));

// Start BFS traversal

while (!queue.isEmpty()) {

// Pop a node say "n" from the from the front of the queue.

Node n = queue.remove();

// Iterate through all the neighbors of the node "n"

for (Node neighbor: n.neighbors) {

if (!visited.containsKey(neighbor)) {

// Clone the neighbor and put in the visited, if not present already

visited.put(neighbor, new Node(neighbor.val, new ArrayList()));

// Add the newly encountered node to the queue.

queue.add(neighbor);

}

// Add the clone of the neighbor to the neighbors of the clone node "n".

visited.get(n).neighbors.add(visited.get(neighbor));

}

}

// Return the clone of the node from visited.

return visited.get(node);

}

}

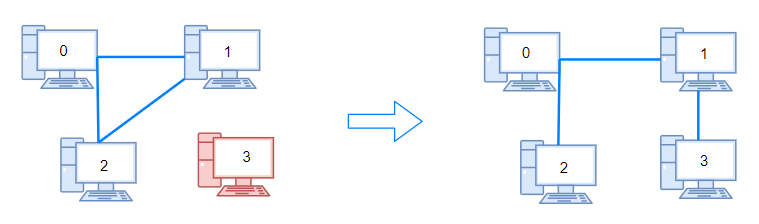
# 338. Making wired Connections

There are n computers numbered from 0 to n - 1 connected by ethernet cables connections forming a network where connections[i] = [ai, bi] represents a connection between computers ai and bi. Any computer can reach any other computer directly or indirectly through the network.

You are given an initial computer network connections. You can extract certain cables between two directly connected computers, and place them between any pair of disconnected computers to make them directly connected.

Return *the minimum number of times you need to do this in order to make all the computers connected*. If it is not possible, return -1.

**Example 1:**

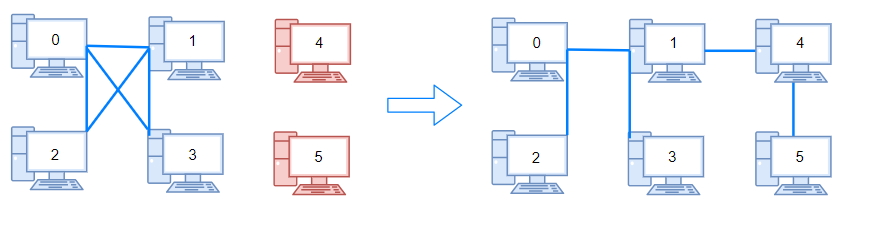


**Input:** n = 4, connections = [[0,1],[0,2],[1,2]]

**Output:** 1

**Explanation:** Remove cable between computer 1 and 2 and place between computers 1 and 3.

**Example 2:**



**Input:** n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]

**Output:** 2

**Example 3:**

**Input:** n = 6, connections = [[0,1],[0,2],[0,3],[1,2]]

**Output:** -1

**Explanation:** There are not enough cables.

**Constraints:**

* 1 <= n <= 105
* 1 <= connections.length <= min(n \* (n - 1) / 2, 105)
* connections[i].length == 2
* 0 <= ai, bi < n
* ai != bi
* There are no repeated connections.
* No two computers are connected by more than one cable.

## Solution:

**Using Disjoint set data structure**

class Solution {

public:

int find(int parent[], int x){

if(parent[x]==x)

return x;

return find(parent, parent[x]);

}

void \_union(int parent[], int a, int b){

int pa = find(parent, a), pb = find(parent, b);

parent[pb] = pa;

}

int makeConnected(int n, vector<vector<int>>& connections) {

if(connections.size()<n-1)

return -1;

int parent[n], i=0;

for(;i<n;i++)

parent[i] = i;

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

\_union(parent, connections[i][0], connections[i][1]);

}

unordered\_set<int> st;

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

st.insert(find(parent, i));

}

return st.size()-1;

}

};

**Using DFS:**

class Solution {

public:

void dfs(list<int>\* l, int x, bool\* visited){

visited[x] = true;

for(auto a : l[x]){

if(visited[a])continue;

dfs(l,a,visited);

}

}

int makeConnected(int n, vector<vector<int>>& connections) {

if(connections.size()<(n-1))return -1;

int edges = connections.size();

list<int>\* l = new list<int>[100000];

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

l[connections[i][0]].push\_back(connections[i][1]);

l[connections[i][1]].push\_back(connections[i][0]);

}

int components = 0;

bool\*visited = new bool[n];

for(int i = 0;i<n;i++)visited[i] = false;

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

if(!visited[i]){

components++;

dfs(l,i,visited);

}

}

return components-1;

}

};

# 339. word ladder

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

* Every adjacent pair of words differs by a single letter.
* Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
* sk == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return *the****number of words****in the****shortest transformation sequence****from* beginWord *to* endWord*, or*0*if no such sequence exists.*

**Example 1:**

**Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

**Output:** 5

**Explanation:** One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.

**Example 2:**

**Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

**Output:** 0

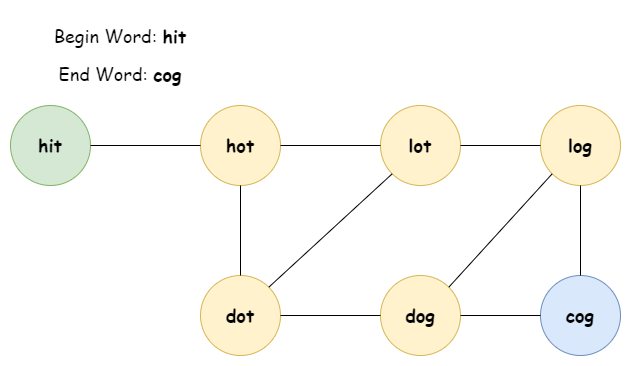
**Explanation:** The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

**Constraints:**

* 1 <= beginWord.length <= 10
* endWord.length == beginWord.length
* 1 <= wordList.length <= 5000
* wordList[i].length == beginWord.length
* beginWord, endWord, and wordList[i] consist of lowercase English letters.
* beginWord != endWord
* All the words in wordList are **unique**.

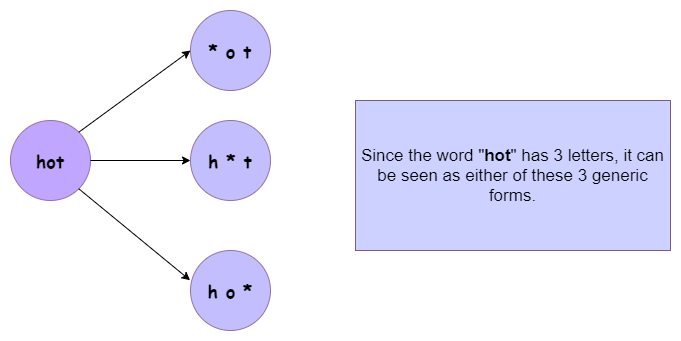
## Solution:

We are given a beginWord and an endWord. Let these two represent start node and end node of a graph. We have to reach from the start node to the end node using some intermediate nodes/words. The intermediate nodes are determined by the wordList given to us. The only condition for every step we take on this ladder of words is the current word should change by just one letter.



We will essentially be working with an undirected and unweighted graph with words as nodes and edges between words which differ by just one letter. The problem boils down to finding the shortest path from a start node to a destination node, if there exists one. Hence it can be solved using Breadth First Search approach.

One of the most important step here is to figure out how to find adjacent nodes i.e. words which differ by one letter. To efficiently find the neighboring nodes for any given word we do some pre-processing on the words of the given wordList. The pre-processing involves replacing the letter of a word by a non-alphabet say, \*.



This pre-processing helps to form generic states to represent a single letter change.

For e.g. Dog ----> D\*g <---- Dig

Both Dog and Dig map to the same intermediate or generic state D\*g.

The preprocessing step helps us find out the generic one letter away nodes for any word of the word list and hence making it easier and quicker to get the adjacent nodes. Otherwise, for every word we will have to iterate over the entire word list and find words that differ by one letter. That would take a lot of time. This preprocessing step essentially builds the adjacency list first before beginning the breadth first search algorithm.

For eg. While doing BFS if we have to find the adjacent nodes for Dug we can first find all the generic states for Dug.

1. Dug => \*ug
2. Dug => D\*g
3. Dug => Du\*

The second transformation D\*g could then be mapped to Dog or Dig, since all of them share the same generic state. Having a common generic transformation means two words are connected and differ by one letter.

#### Approach 1: Breadth First Search

**Intuition**

Start from beginWord and search the endWord using BFS.

**Algorithm**

1. Do the pre-processing on the given wordList and find all the possible generic/intermediate states. Save these intermediate states in a dictionary with key as the intermediate word and value as the list of words which have the same intermediate word.
2. Push a tuple containing the beginWord and 1 in a queue. The 1 represents the level number of a node. We have to return the level of the endNode as that would represent the shortest sequence/distance from the beginWord.
3. To prevent cycles, use a visited dictionary.
4. While the queue has elements, get the front element of the queue. Let's call this word as current\_word.
5. Find all the generic transformations of the current\_word and find out if any of these transformations is also a transformation of other words in the word list. This is achieved by checking the all\_combo\_dict.
6. The list of words we get from all\_combo\_dict are all the words which have a common intermediate state with the current\_word. These new set of words will be the adjacent nodes/words to current\_word and hence added to the queue.
7. Hence, for each word in this list of intermediate words, append (word, level + 1) into the queue where level is the level for the current\_word.
8. Eventually if you reach the desired word, its level would represent the shortest transformation sequence length.

Termination condition for standard BFS is finding the end word.

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

// Since all words are of same length.

int L = beginWord.length();

// Dictionary to hold combination of words that can be formed,

// from any given word. By changing one letter at a time.

Map<String, List<String>> allComboDict = new HashMap<>();

wordList.forEach(

word -> {

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

// Key is the generic word

// Value is a list of words which have the same intermediate generic word.

String newWord = word.substring(0, i) + '\*' + word.substring(i + 1, L);

List<String> transformations = allComboDict.getOrDefault(newWord, new ArrayList<>());

transformations.add(word);

allComboDict.put(newWord, transformations);

}

});

// Queue for BFS

Queue<Pair<String, Integer>> Q = new LinkedList<>();

Q.add(new Pair(beginWord, 1));

// Visited to make sure we don't repeat processing same word.

Map<String, Boolean> visited = new HashMap<>();

visited.put(beginWord, true);

while (!Q.isEmpty()) {

Pair<String, Integer> node = Q.remove();

String word = node.getKey();

int level = node.getValue();

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

// Intermediate words for current word

String newWord = word.substring(0, i) + '\*' + word.substring(i + 1, L);

// Next states are all the words which share the same intermediate state.

for (String adjacentWord : allComboDict.getOrDefault(newWord, new ArrayList<>())) {

// If at any point if we find what we are looking for

// i.e. the end word - we can return with the answer.

if (adjacentWord.equals(endWord)) {

return level + 1;

}

// Otherwise, add it to the BFS Queue. Also mark it visited

if (!visited.containsKey(adjacentWord)) {

visited.put(adjacentWord, true);

Q.add(new Pair(adjacentWord, level + 1));

}

}

}

}

return 0;

}

}

**Complexity Analysis**

* Time Complexity: O({M}^2 \times N)*O*(*M*2×*N*), where M*M* is the length of each word and N*N* is the total number of words in the input word list.
  + For each word in the word list, we iterate over its length to find all the intermediate words corresponding to it. Since the length of each word is M*M* and we have N*N* words, the total number of iterations the algorithm takes to create all\_combo\_dict is M \times N*M*×*N*. Additionally, forming each of the intermediate word takes O(M)*O*(*M*) time because of the substring operation used to create the new string. This adds up to a complexity of O({M}^2 \times N)*O*(*M*2×*N*).
  + Breadth first search in the worst case might go to each of the N*N* words. For each word, we need to examine M*M* possible intermediate words/combinations. Notice, we have used the substring operation to find each of the combination. Thus, M*M* combinations take O({M} ^ 2)*O*(*M*2) time. As a result, the time complexity of BFS traversal would also be O({M}^2 \times N)*O*(*M*2×*N*).

Combining the above steps, the overall time complexity of this approach is O({M}^2 \times N)*O*(*M*2×*N*).

* Space Complexity: O({M}^2 \times N)*O*(*M*2×*N*).
  + Each word in the word list would have M*M* intermediate combinations. To create the all\_combo\_dict dictionary we save an intermediate word as the key and its corresponding original words as the value. Note, for each of M*M* intermediate words we save the original word of length M*M*. This simply means, for every word we would need a space of {M}^2*M*2 to save all the transformations corresponding to it. Thus, all\_combo\_dict would need a total space of O({M}^2 \times N)*O*(*M*2×*N*).
  + Visited dictionary would need a space of O(M \times N)*O*(*M*×*N*) as each word is of length M*M*.
  + Queue for BFS in worst case would need a space for all O(N)*O*(*N*) words and this would also result in a space complexity of O(M \times N)*O*(*M*×*N*).

Combining the above steps, the overall space complexity is O({M}^2 \times N)*O*(*M*2×*N*) + O(M \* N)*O*(*M*∗*N*) + O(M \* N)*O*(*M*∗*N*) = O({M}^2 \times N)*O*(*M*2×*N*) space.

**Optimization:** We can definitely reduce the space complexity of this algorithm by storing the indices corresponding to each word instead of storing the word itself.

**My Implementation of above approach:**

class Solution {

public:

int ladderLength(string beginWord, string endWord, vector<string>& wordList) {

unordered\_map<string, vector<int>> mp;

wordList.push\_back(beginWord);

int n = wordList.size();

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

for(int j=0;j<wordList[i].size();j++){

string s = wordList[i];

s[j] = '\*';

mp[s].push\_back(i);

}

}

queue<pair<string, int>> q;

q.push({beginWord, 1});

vector<bool> visited(n, false);

visited[n-1] = true;

while(!q.empty()){

int level = q.front().second;

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

string t = q.front().first;

t[i] = '\*';

vector<int> v = mp[t];

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

if(!visited[v[j]]){

if(wordList[v[j]]==endWord)

return level+1;

visited[v[j]] = true;

q.push({wordList[v[j]], level+1});

}

}

}

q.pop();

}

return 0;

}

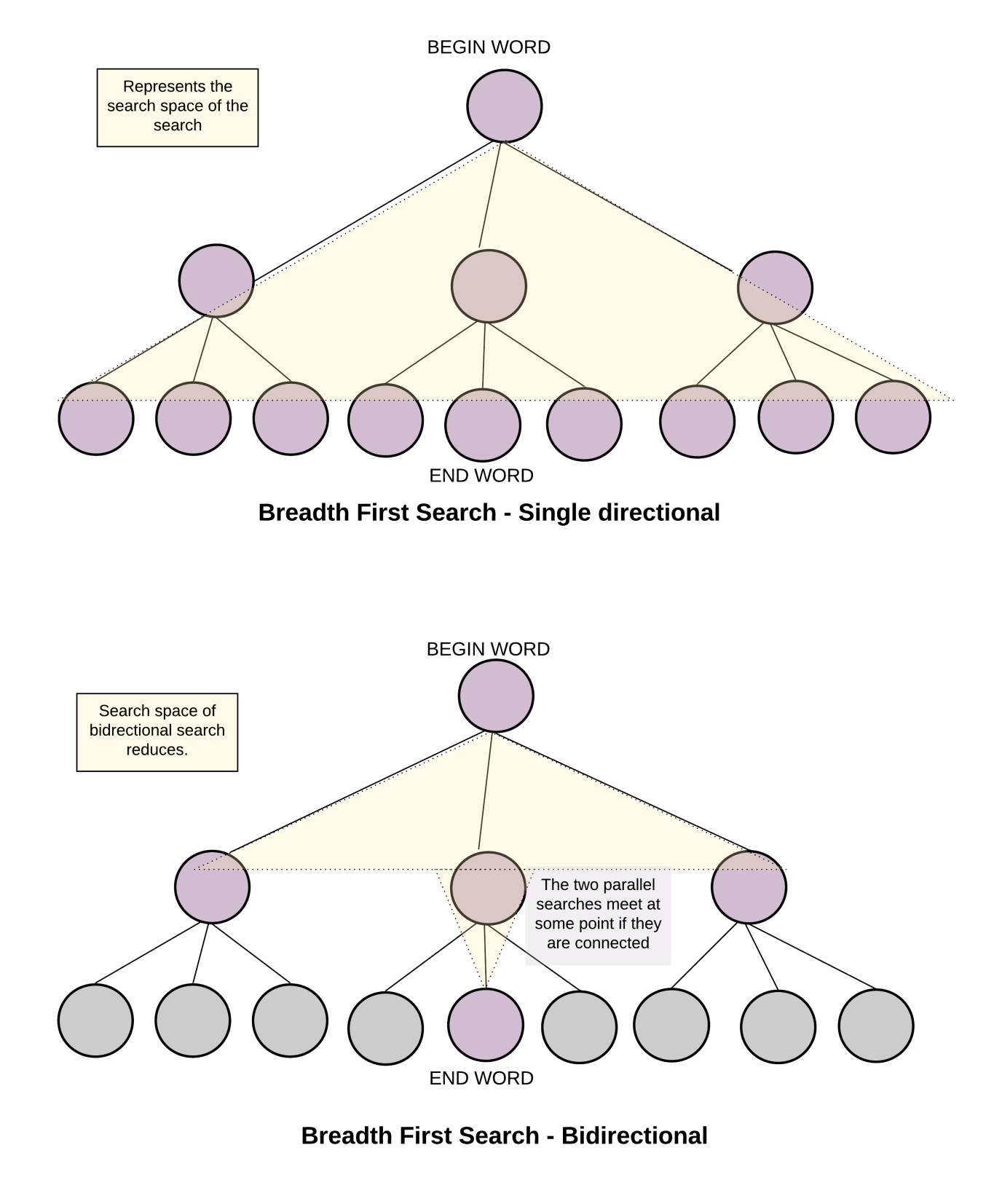
};

#### Approach 2: Bidirectional Breadth First Search

**Intuition**

The graph formed from the nodes in the dictionary might be too big. The search space considered by the breadth first search algorithm depends upon the branching factor of the nodes at each level. If the branching factor remains the same for all the nodes, the search space increases exponentially along with the number of levels. Consider a simple example of a binary tree. With each passing level in a complete binary tree, the number of nodes increase in powers of 2.

We can considerably cut down the search space of the standard breadth first search algorithm if we launch two simultaneous BFS. One from the beginWord and one from the endWord. We progress one node at a time from both sides and at any point in time if we find a common node in both the searches, we stop the search. This is known as bidirectional BFS and it considerably cuts down on the search space and hence reduces the time and space complexity.

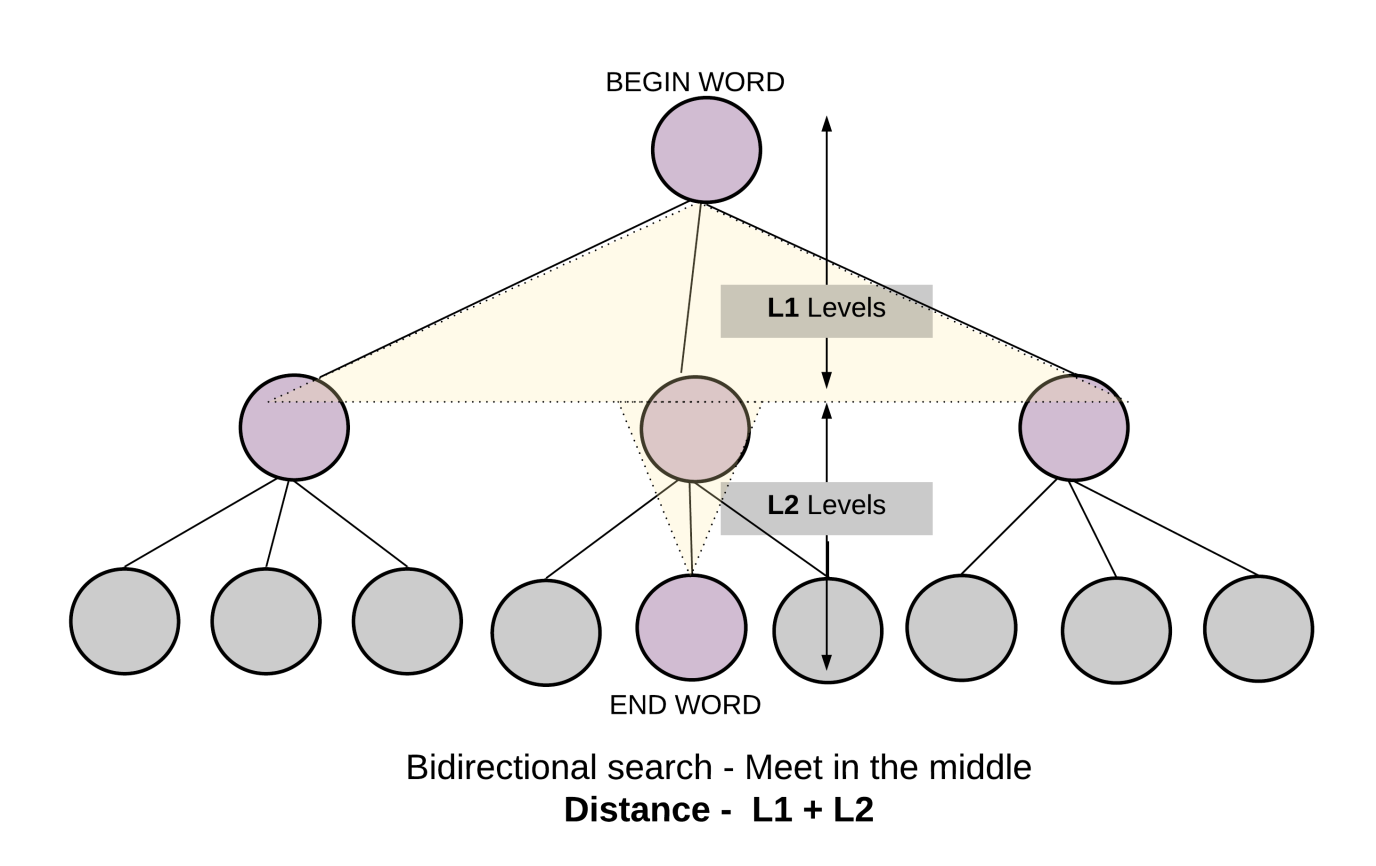


**Algorithm**

1. The algorithm is very similar to the standard BFS based approach we saw earlier.
2. The only difference is we now do BFS starting two nodes instead of one. This also changes the termination condition of our search.
3. We now have two visited dictionaries to keep track of nodes visited from the search starting at the respective ends.
4. If we ever find a node/word which is in the visited dictionary of the parallel search we terminate our search, since we have found the meet point of this bidirectional search. It's more like meeting in the middle instead of going all the way through.

Termination condition for bidirectional search is finding a word which is already been seen by the parallel search.

1. The shortest transformation sequence is the sum of levels of the meet point node from both the ends. Thus, for every visited node we save its level as value in the visited dictionary.



class Solution {

private int L;

private Map<String, List<String>> allComboDict;

Solution() {

this.L = 0;

// Dictionary to hold combination of words that can be formed,

// from any given word. By changing one letter at a time.

this.allComboDict = new HashMap<>();

}

private int visitWordNode(

Queue<Pair<String, Integer>> Q,

Map<String, Integer> visited,

Map<String, Integer> othersVisited) {

Pair<String, Integer> node = Q.remove();

String word = node.getKey();

int level = node.getValue();

for (int i = 0; i < this.L; i++) {

// Intermediate words for current word

String newWord = word.substring(0, i) + '\*' + word.substring(i + 1, L);

// Next states are all the words which share the same intermediate state.

for (String adjacentWord : this.allComboDict.getOrDefault(newWord, new ArrayList<>())) {

// If at any point if we find what we are looking for

// i.e. the end word - we can return with the answer.

if (othersVisited.containsKey(adjacentWord)) {

return level + othersVisited.get(adjacentWord);

}

if (!visited.containsKey(adjacentWord)) {

// Save the level as the value of the dictionary, to save number of hops.

visited.put(adjacentWord, level + 1);

Q.add(new Pair(adjacentWord, level + 1));

}

}

}

return -1;

}

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

if (!wordList.contains(endWord)) {

return 0;

}

// Since all words are of same length.

this.L = beginWord.length();

wordList.forEach(

word -> {

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

// Key is the generic word

// Value is a list of words which have the same intermediate generic word.

String newWord = word.substring(0, i) + '\*' + word.substring(i + 1, L);

List<String> transformations =

this.allComboDict.getOrDefault(newWord, new ArrayList<>());

transformations.add(word);

this.allComboDict.put(newWord, transformations);

}

});

// Queues for birdirectional BFS

// BFS starting from beginWord

Queue<Pair<String, Integer>> Q\_begin = new LinkedList<>();

// BFS starting from endWord

Queue<Pair<String, Integer>> Q\_end = new LinkedList<>();

Q\_begin.add(new Pair(beginWord, 1));

Q\_end.add(new Pair(endWord, 1));

// Visited to make sure we don't repeat processing same word.

Map<String, Integer> visitedBegin = new HashMap<>();

Map<String, Integer> visitedEnd = new HashMap<>();

visitedBegin.put(beginWord, 1);

visitedEnd.put(endWord, 1);

while (!Q\_begin.isEmpty() && !Q\_end.isEmpty()) {

// One hop from begin word

int ans = visitWordNode(Q\_begin, visitedBegin, visitedEnd);

if (ans > -1) {

return ans;

}

// One hop from end word

ans = visitWordNode(Q\_end, visitedEnd, visitedBegin);

if (ans > -1) {

return ans;

}

}

return 0;

}

}

**Complexity Analysis**

* Time Complexity: O({M}^2 \times N)*O*(*M*2×*N*), where M*M* is the length of words and N*N* is the total number of words in the input word list. Similar to one directional, bidirectional also takes O({M}^2 \times N)*O*(*M*2×*N*) time for finding out all the transformations. But the search time reduces to half, since the two parallel searches meet somewhere in the middle.
* Space Complexity: O({M}^2 \times N)*O*(*M*2×*N*), to store all M*M* transformations for each of the N*N* words in the all\_combo\_dict dictionary, same as one directional. But bidirectional reduces the search space. It narrows down because of meeting in the middle.

# 340. Dijkstra algo

Given a weighted, undirected and connected graph of V vertices and E edges, Find the shortest distance of all the vertex's from the source vertex S.  
**Note:**The Graph doesn't contain any negative weight cycle.

**Example 1:**

**Input:**

**S** = 0

**Output:**

0 9

**Explanation**:

The source vertex is 0. Hence, the shortest

distance of node 0 is 0 and the shortest

distance from node 9 is 9 - 0 = 9.

**Example 2:**

**Input:**

**S** = 2

**Output:**

4 3 0

**Explanation**:

For nodes 2 to 0, we can follow the path-

2-1-0. This has a distance of 1+3 = 4,

whereas the path 2-0 has a distance of 6. So,

the Shortest path from 2 to 0 is 4.

The other distances are pretty straight-forward.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **dijkstra()**  which takes number of vertices Vandan adjacency list adj as input parameters and returns a list of integers, where ith integer denotes the shortest distance of the ith node from Source node. Here adj[i] contains a list of lists containing two integers where the first integer j denotes that there is an edge between i and j and second integer w denotes that the weight between edge i and j is w.

**Expected Time Complexity:** O(V2).  
**Expected Auxiliary Space:** O(V2).

**Constraints:**  
1 ≤ V ≤ 1000  
0 ≤ adj[i][j] ≤ 1000

1 ≤ adj.size() ≤ [ (V\*(V - 1)) / 2 ]  
0 ≤ S < V

## Solution:

# For Adjacency matrix Representation

Dijkstra’s algorithm is very similar to [Prim’s algorithm for minimum spanning tree](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/). Like Prim’s MST, we generate a*SPT (shortest path tree)* with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.  
Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm   
**1)** Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.   
**2)** Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.   
**3)** While *sptSet* doesn’t include all vertices   
….**a)** Pick a vertex u which is not there in *sptSet* and has a minimum distance value.   
….**b)** Include u to *sptSet*.   
….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Let us understand with the following example: 



The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet*becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively). 



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet*includes all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).



***How to implement the above algorithm?***

We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store the shortest distance values of all vertices.

// A C++ program for Dijkstra's single source shortest path algorithm.

// The program is for adjacency matrix representation of the graph

#include <iostream>

using namespace std;

#include <limits.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[])

{

cout <<"Vertex \t Distance from Source" << endl;

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

cout << i << " \t\t"<<dist[i]<< endl;

}

// 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);

}

// 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;

}

**Output:**

Vertex Distance from Source

0 0

1 4

2 12

3 19

4 21

5 11

6 9

7 8

8 14

**Notes:**   
**1)** The code calculates the shortest distance but doesn’t calculate the path information. We can create a parent array, update the parent array when distance is updated (like [prim’s implementation](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/)) and use it to show the shortest path from source to different vertices.  
**2)** The code is for undirected graphs, the same Dijkstra function can be used for directed graphs also.  
**3)** The code finds the shortest distances from the source to all vertices. If we are interested only in the shortest distance from the source to a single target, we can break the for loop when the picked minimum distance vertex is equal to the target (Step 3.a of the algorithm).  
**4)** Time Complexity of the implementation is O(V^2). If the input [graph is represented using adjacency list](https://www.geeksforgeeks.org/graph-and-its-representations/), it can be reduced to O(E log V) with the help of a binary heap. Please see   
[Dijkstra’s Algorithm for Adjacency List Representation](https://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/) for more details.  
**5)** Dijkstra’s algorithm doesn’t work for graphs with negative weight cycles. It may give correct results for a graph with negative edges but you must allow a vertex can be visited multiple times and that version will lose its fast time complexity. For graphs with negative weight edges and cycles, [Bellman–Ford algorithm](http://en.wikipedia.org/wiki/Bellman-Ford_algorithm) can be used, we will soon be discussing it as a separate post.

# For Adjacency List Representation

**My Implementation:**

vector <int> dijkstra(int V, vector<vector<int>> adj[], int S)

{

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

dist[S] = 0;

set<pair<int, int>> st;

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

st.insert({dist[i], i});

}

while(!st.empty()){

auto beg = st.begin();

int min = (\*beg).first, ind = (\*beg).second;

st.erase(beg);

for(int i=0;i<adj[ind].size();i++){

int adjacent = adj[ind][i][0], weight = adj[ind][i][1];

if(dist[ind]+weight<dist[adjacent] && st.find({dist[adjacent], adjacent})!=st.end()){

st.erase(st.find({dist[adjacent] ,adjacent}));

dist[adjacent] = dist[ind]+weight;

st.insert({dist[adjacent] ,adjacent});

}

}

}

return dist;

}

**Time Complexity:** The time complexity of the above code/algorithm looks O(V^2) as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed O(V+E) times (similar to BFS). The inner loop has decreaseKey() operation which takes O(LogV) time. So overall time complexity is O(E+V)\*O(LogV) which is O((E+V)\*LogV) = O(ELogV)   
Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to O(E + VLogV) using Fibonacci Heap. The reason is, Fibonacci Heap takes O(1) time for decrease-key operation while Binary Heap takes O(Logn) time.

# 341. Implement Topological Sort

Given a Directed Acyclic Graph (DAG) with V vertices and E edges, Find any Topological Sorting of that Graph.

**Example 1:**

**Input:**

**Output:**

1

**Explanation**:

The output 1 denotes that the order is

valid. So, if you have, implemented

your function correctly, then output

would be 1 for all test cases.

One possible Topological order for the

graph is 3, 2, 1, 0.

**Example 2:**

**Input:**

**Output:**

1

**Explanation:**

The output 1 denotes that the order is

valid. So, if you have, implemented

your function correctly, then output

would be 1 for all test cases.

One possible Topological order for the

graph is 5, 4, 2, 1, 3, 0.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **topoSort()**  which takes the integer V denoting the number of vertices and adjacency list as input parameters and returns an array consisting of a the vertices in Topological order. As there are multiple Topological orders possible, you may return any of them. If your returned topo sort is correct then console output will be 1 else 0.

**Expected Time Complexity:** O(V + E).  
**Expected Auxiliary Space:** O(V).

**Constraints:**  
2 ≤ V ≤ 104  
1 ≤ E ≤ (N\*(N-1))/2

## Solution:

**My Implementation:**

vector<int> topoSort(int V, vector<int> adj[])

{

vector<int> indegree(V, 0);

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

for(int j=0;j<adj[i].size();j++){

indegree[adj[i][j]]++;

}

}

queue<int> q;

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

if(indegree[i]==0)

q.push(i);

}

vector<int> res;

while(!q.empty()){

int x = q.front(); q.pop();

res.push\_back(x);

for(int i=0;i<adj[x].size();i++){

indegree[adj[x][i]]--;

if(indegree[adj[x][i]]==0)

q.push(adj[x][i]);

}

}

return res;

}

**Time Complexity:** O(V+E)

**Auxiliary space:** O(V)

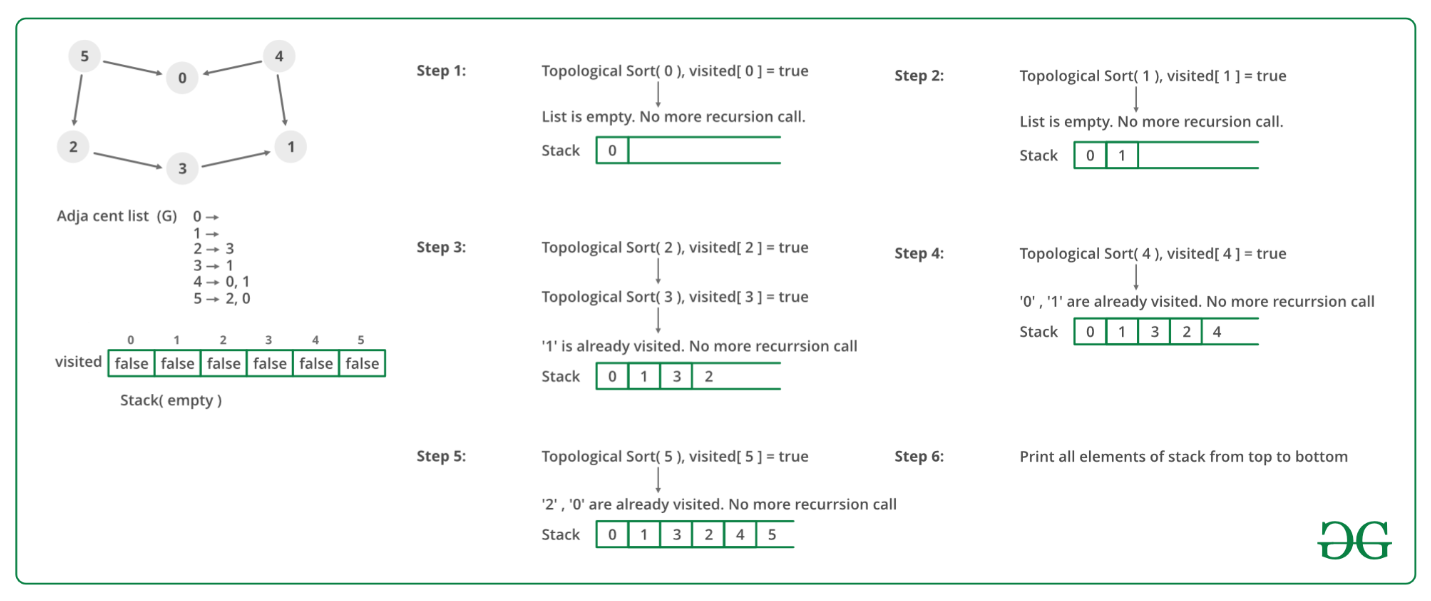
***Topological Sorting vs Depth First Traversal (DFS)***:

In [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting.

**Algorithm to find Topological Sorting:**

We recommend to first see the implementation of [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/). We can modify [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)to find Topological Sorting of a graph. In [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of the stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack.

Below image is an illustration of the above approach:



Following are the implementations of topological sorting. Please see the code for Depth [First Traversal for a disconnected Graph](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/) and note the differences between the second code given there and the below code.

// 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 {

// No. of vertices'

int V;

// 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:

// Constructor

Graph(int V);

// 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)

{

// Add w to v’s list.

adj[v].push\_back(w);

}

// 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 Code

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 \n";

// Function Call

g.topologicalSort();

return 0;

}

**Output**

Following is a Topological Sort of the given graph

5 4 2 3 1 0

**Complexity Analysis:**

* **Time Complexity:** O(V+E).   
  The above algorithm is simply DFS with an extra stack. So time complexity is the same as DFS which is.
* **Auxiliary space:** O(V).   
  The extra space is needed for the stack.

**Note:** Here, we can also use vector instead of the stack. If the vector is used then print the elements in reverse order to get the topological sorting.

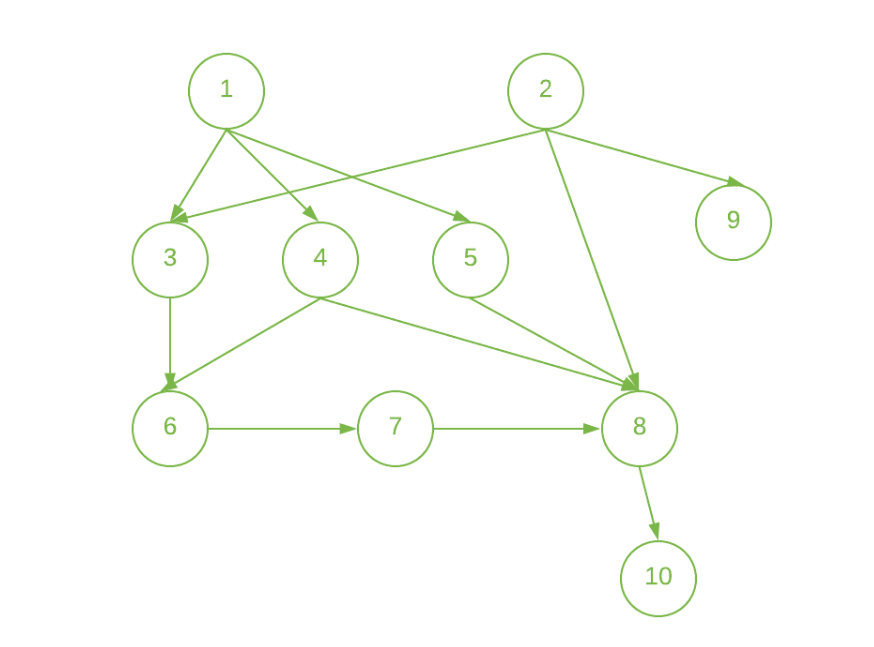
**Applications:**   
Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers.

# 342. Minimum time taken by each job to be completed given by a Directed Acyclic Graph

Given a **Directed Acyclic Graph** having **V**vertices and**E**edges, where each edge **{U, V}** represents the Jobs **U** and **V** such that Job **V** can only be started only after completion of Job **U**. The task is to determine the minimum time taken by each job to be completed where each Job takes unit time to get completed.

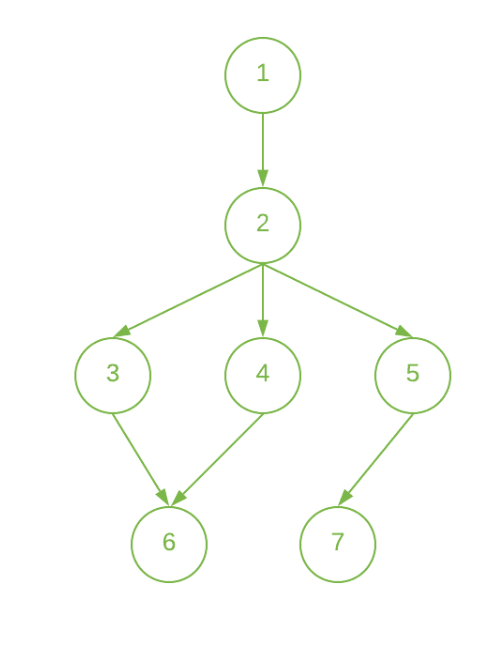
**Examples:**

***Input:****N****=****10, E = 13, Below is the given graph:*

*[](https://media.geeksforgeeks.org/wp-content/uploads/20200804212533/Semester1.png)*

***Output:****1 1 2 2 2 3 4 5 2 6****Explanation:*** *Start the jobs 1 and 2 at the beginning and complete them at 1 unit of time.   
Since, jobs 3, 4, 5, and 9 have the only dependency on one job (i.e 1st job for jobs 3, 4, and 5 and 2nd job for job 9). So, we can start these jobs at 1st unit of time and complete these at 2nd unit of time after the completion of the dependent Job.  
Similarly,   
Job 6 can only be done after 3rd and 4th jobs are done. So, start it at 2nd unit of time and complete it at 3rd unit of time.  
Job 7 can only be done after job 6 is done. So, you can start it at 3rd unit of time and complete it at 4th unit of time.  
Job 8 can only be done after 4th, 5th, and 7th jobs are done. So, start it at 4th unit of time and complete it at 5th unit of time.  
Job 10 can only be done after the 8th job is done. So, start it at 5th unit of time and complete it at 6th unit of time.*

***Input:****N = 7, E = 7, Below is the given graph:*

*[](https://media.geeksforgeeks.org/wp-content/uploads/20200808013603/Semester2.png)*

***Output:****1 2 3 3 3 4 4****Explanation:*** *Start the Job 1 at the beginning and complete it at 1st unit of time.  
The job 2 can only be done after 1st Job is done. So, start it at 1st unit of time and complete it at 2nd unit of time.  
Since, Job 3, 4, and 5 have the only dependency on 2nd Job. So, start these jobs at 2nd unit of time and complete these at 3rd unit of time.  
The Job 6 can only be done after the 3rd and 4th Job is done. So, start it at 3rd unit of time and complete it at 4th unit of time.  
The Job 7 can only be done after the 5th Job is done. So, start it at 3rd hour and complete it at 4th unit of time.*

## Solution:

**Approach:** The job can be started only if all the jobs that are prerequisites of the job that are done. Therefore, the idea is to use [Topological Sort](https://www.geeksforgeeks.org/topological-sorting/) for the given network. Below are the steps:

1. Finish the jobs that are not dependent on any other job.
2. Create an array **inDegree[]** to store the count of the dependent node for each node in the given network.
3. Initialize a [queue](https://www.geeksforgeeks.org/queue-data-structure/) and push all the vertex whose **inDegree[]** is 0.
4. Initialize the timer to 1 and store the current queue size(say **size**) and do the following:
   * Pop the node from the queue until the size is **0** and update the finishing time of this node to the **timer**.
   * While popping the node(say node **U**) from the queue decrement the **inDegree** of every node connected to it.
   * If **inDegree** of any node is **0** in the above step then insert that node in the queue.
   * Increment the timer after all the above steps.
5. Print the finishing time of all the nodes after we traverse every node in the above step.

Below is the implementation of the above approach:

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

#define maxN 100000

// Adjacency List to store the graph

vector<int> graph[maxN];

// Array to store the in-degree of node

int indegree[maxN];

// Array to store the time in which

// the job i can be done

int job[maxN];

// Function to add directed edge

// between two vertices

void addEdge(int u, int v)

{

// Insert edge from u to v

graph[u].push\_back(v);

// Increasing the indegree

// of vertex v

indegree[v]++;

}

// Function to find the minimum time

// needed by each node to get the task

void printOrder(int n, int m)

{

// Find the topo sort order

// using the indegree approach

// Queue to store the

// nodes while processing

queue<int> q;

// Pushing all the vertex in the

// queue whose in-degree is 0

// Update the time of the jobs

// who don't require any job to

// be completed before this job

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

if (indegree[i] == 0) {

q.push(i);

job[i] = 1;

}

}

// Iterate until queue is empty

while (!q.empty()) {

// Get front element of queue

int cur = q.front();

// Pop the front element

q.pop();

for (int adj : graph[cur]) {

// Decrease in-degree of

// the current node

indegree[adj]--;

// Push its adjacent elements

if (indegree[adj] == 0) {

job[adj] = job[cur] + 1;

q.push(adj);

}

}

}

// Print the time to complete

// the job

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

cout << job[i] << " ";

cout << "\n";

}

// Driver Code

int main()

{

// Given Nodes N and edges M

int n, m;

n = 10;

m = 13;

// Given Directed Edges of graph

addEdge(1, 3);

addEdge(1, 4);

addEdge(1, 5);

addEdge(2, 3);

addEdge(2, 8);

addEdge(2, 9);

addEdge(3, 6);

addEdge(4, 6);

addEdge(4, 8);

addEdge(5, 8);

addEdge(6, 7);

addEdge(7, 8);

addEdge(8, 10);

// Function Call

printOrder(n, m);

return 0;

}

**Output:**

1 1 2 2 2 3 4 5 2 6

***Time Complexity:****O(V+E), where V is the number of nodes and E is the number of edges.*  
***Auxiliary Space:****O(V)*

# 343. Find whether it is possible to finish all tasks or not from given dependencies

There are a total of N tasks, labeled from 0 to N-1. Some tasks may have prerequisites, for example to do task 0 you have to first complete task 1, which is expressed as a pair: [0, 1]  
Given the total number of **tasks N** and a list of **prerequisite pairs P**, find if it is possible to finish all tasks.

**Example 1:**

**Input:**

N = 4, P = 3

prerequisites = {{1,0},{2,1},{3,2}}

**Output:**

Yes

**Explanation**:

To do task 1 you should have completed

task 0, and to do task 2 you should

have finished task 1, and to do task 3 you

should have finished task 2. So it is possible.

**Example 2:**

**Input:**

N = 2, P = 2

prerequisites = {{1,0},{0,1}}

**Output:**

No

**Explanation**:

To do task 1 you should have completed

task 0, and to do task 0 you should

have finished task 1. So it is impossible.

**Your task:**  
You don’t need to read input or print anything. Your task is to complete the function **isPossible()** which takes the integer N denoting the number of tasks, P denoting the number of prerequisite pairs and prerequisite as input parameters and returns true if it is possible to finish all tasks otherwise returns false.

**Expected Time Complexity:**O(N + P)  
**Expected Auxiliary Space:**O(N + P)

**Constraints:**  
1 ≤ N ≤ 104  
1 ≤ P ≤ 105

## Solution:

**Solution:**We can consider this problem as a graph (related to [topological sorting](https://www.geeksforgeeks.org/topological-sorting/)) problem. All tasks are nodes of the graph and if task u is a prerequisite of task v, we will add a directed edge from node u to node v. Now, this problem is equivalent to detecting a cycle in the graph represented by prerequisites. If there is a cycle in the graph, then it is not possible to finish all tasks (because in that case there is no any topological order of tasks). Both BFS and DFS can be used to solve it.  
Since pair is inconvenient for the implementation of graph algorithms, we first transform it to a graph. If task u is a prerequisite of task v, we will add a directed edge from node u to node v.  
Prerequisite : [Detect Cycle in a Directed Graph](https://www.geeksforgeeks.org/detect-cycle-in-a-graph/)  
**Using**[**DFS**](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/) For DFS, it will first visit a node, then one neighbor of it, then one neighbor of this neighbor… and so on. If it meets a node which was visited in the current process of DFS visit, a cycle is detected and we will return false. Otherwise it will start from another unvisited node and repeat this process till all the nodes have been visited. Note that you should make two records: one is to record all the visited nodes and the other is to record the visited nodes in the current DFS visit.  
The code is as follows. We use a vector visited to record all the visited nodes and another vector onpath to record the visited nodes of the current DFS visit. Once the current visit is finished, we reset the onpath value of the starting node to false.

// CPP program to check whether we can finish all

// tasks or not from given dependencies.

#include <bits/stdc++.h>

using namespace std;

// Returns adjacency list representation from a list

// of pairs.

vector<unordered\_set<int> > make\_graph(int numTasks,

vector<pair<int, int> >& prerequisites)

{

vector<unordered\_set<int> > graph(numTasks);

for (auto pre : prerequisites)

graph[pre.second].insert(pre.first);

return graph;

}

// A DFS based function to check if there is a cycle

// in the directed graph.

bool dfs\_cycle(vector<unordered\_set<int> >& graph, int node,

vector<bool>& onpath, vector<bool>& visited)

{

if (visited[node])

return false;

onpath[node] = visited[node] = true;

for (int neigh : graph[node])

if (onpath[neigh] || dfs\_cycle(graph, neigh, onpath, visited))

return true;

return onpath[node] = false;

}

// Main function to check whether possible to finish all tasks or not

bool canFinish(int numTasks, vector<pair<int, int> >& prerequisites)

{

vector<unordered\_set<int> > graph = make\_graph(numTasks, prerequisites);

vector<bool> onpath(numTasks, false), visited(numTasks, false);

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

if (!visited[i] && dfs\_cycle(graph, i, onpath, visited))

return false;

return true;

}

int main()

{

int numTasks = 4;

vector<pair<int, int> > prerequisites;

// for prerequisites: [[1, 0], [2, 1], [3, 2]]

prerequisites.push\_back(make\_pair(1, 0));

prerequisites.push\_back(make\_pair(2, 1));

prerequisites.push\_back(make\_pair(3, 2));

if (canFinish(numTasks, prerequisites)) {

cout << "Possible to finish all tasks";

}

else {

cout << "Impossible to finish all tasks";

}

return 0;

}

**Output**

Possible to finish all tasks

**Using**[**BFS**](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/)   
BFS can be used to solve it using the idea of topological sort. If topological sorting is possible, it means there is no cycle and it is possible to finish all the tasks.  
BFS uses the indegrees of each node. We will first try to find a node with 0 indegree. If we fail to do so, there must be a cycle in the graph and we return false. Otherwise we have found one. We set its indegree to be -1 to prevent from visiting it again and reduce the indegrees of all its neighbors by 1. This process will be repeated for n (number of nodes) times. If we have not returned false, we will return true.

// A BFS based solution to check if we can finish

// all tasks or not. This solution is mainly based

// on Kahn's algorithm.

#include <bits/stdc++.h>

using namespace std;

// Returns adjacency list representation from a list

// of pairs.

vector<unordered\_set<int> > make\_graph(int numTasks,

vector<pair<int, int> >& prerequisites)

{

vector<unordered\_set<int> > graph(numTasks);

for (auto pre : prerequisites)

graph[pre.second].insert(pre.first);

return graph;

}

// Finds in-degree of every vertex

vector<int> compute\_indegree(vector<unordered\_set<int> >& graph)

{

vector<int> degrees(graph.size(), 0);

for (auto neighbors : graph)

for (int neigh : neighbors)

degrees[neigh]++;

return degrees;

}

// Main function to check whether possible to finish all tasks or not

bool canFinish(int numTasks, vector<pair<int, int> >& prerequisites)

{

vector<unordered\_set<int> > graph = make\_graph(numTasks, prerequisites);

vector<int> degrees = compute\_indegree(graph);

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

int j = 0;

for (; j < numTasks; j++)

if (!degrees[j])

break;

if (j == numTasks)

return false;

degrees[j] = -1;

for (int neigh : graph[j])

degrees[neigh]--;

}

return true;

}

int main()

{

int numTasks = 4;

vector<pair<int, int> > prerequisites;

prerequisites.push\_back(make\_pair(1, 0));

prerequisites.push\_back(make\_pair(2, 1));

prerequisites.push\_back(make\_pair(3, 2));

if (canFinish(numTasks, prerequisites)) {

cout << "Possible to finish all tasks";

}

else {

cout << "Impossible to finish all tasks";

}

return 0;

}

**Output**

Possible to finish all tasks

# 344. Find the no. of Isalnds

Given a grid of size n\*m (n is number of rows and m is number of columns grid has) consisting of '0's(Water) and '1's(Land). Find the number of islands.  
**Note:**An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically or diagonally i.e., in all 8 directions.

**Example 1:**

**Input:**

grid = {{0,1},{1,0},{1,1},{1,0}}

**Output:**

1

**Explanation:**

The grid is-

0 1

1 0

1 1

1 0

All lands are connected.

**Example 2:**

**Input:**

grid = {{0,1,1,1,0,0,0},{0,0,1,1,0,1,0}}

**Output:**

2

**Expanation:**

The grid is-

0 1 1 1 0 0 0

0 0 1 1 0 1 0

There are two islands one is colored in blue

and other in orange.

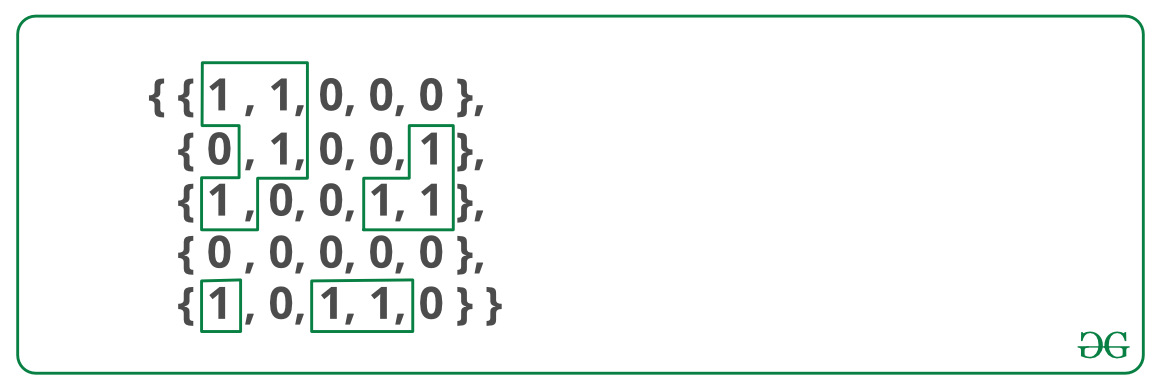
**Your Task:**  
You don't need to read or print anything. Your task is to complete the function **numIslands()**which takes grid as input parameter and returns the total number of islands.

**Expected Time Compelxity:**O(n\*m)  
**Expected Space Compelxity:**O(n\*m)

**Constraints:**  
1 ≤ n, m ≤ 500

## Solution:

A graph where all vertices are connected with each other has exactly one connected component, consisting of the whole graph. Such a graph with only one connected component is called a Strongly Connected Graph.  
The problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected components. BFS can also be used.  
***What is an island?***   
A group of connected 1s forms an island. For example, the below matrix contains 4 islands



A cell in 2D matrix can be connected to 8 neighbours. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursively call for 8 neighbours only.

// C++Program to count islands in boolean 2D matrix

#include <bits/stdc++.h>

using namespace std;

// A utility function to do DFS for a 2D

// boolean matrix. It only considers

// the 8 neighbours as adjacent vertices

void DFS(vector<vector<int>> &M, int i, int j, int ROW,

int COL)

{

//Base condition

//if i less than 0 or j less than 0 or i greater than ROW-1 or j greater than COL- or if M[i][j] != 1 then we will simply return

if (i < 0 || j < 0 || i > (ROW - 1) || j > (COL - 1) || M[i][j] != 1)

{

return;

}

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

{

M[i][j] = 0;

DFS(M, i + 1, j, ROW, COL); //right side traversal

DFS(M, i - 1, j, ROW, COL); //left side traversal

DFS(M, i, j + 1, ROW, COL); //upward side traversal

DFS(M, i, j - 1, ROW, COL); //downward side traversal

DFS(M, i + 1, j + 1, ROW, COL); //upward-right side traversal

DFS(M, i - 1, j - 1, ROW, COL); //downward-left side traversal

DFS(M, i + 1, j - 1, ROW, COL); //downward-right side traversal

DFS(M, i - 1, j + 1, ROW, COL); //upward-left side traversal

}

}

int countIslands(vector<vector<int>> &M)

{

int ROW = M.size();

int COL = M[0].size();

int count = 0;

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

{

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

{

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

{

M[i][j] = 0;

count++;

DFS(M, i + 1, j, ROW, COL); //right side traversal

DFS(M, i - 1, j, ROW, COL); //left side traversal

DFS(M, i, j + 1, ROW, COL); //upward side traversal

DFS(M, i, j - 1, ROW, COL); //downward side traversal

DFS(M, i + 1, j + 1, ROW, COL); //upward-right side traversal

DFS(M, i - 1, j - 1, ROW, COL); //downward-left side traversal

DFS(M, i + 1, j - 1, ROW, COL); //downward-right side traversal

DFS(M, i - 1, j + 1, ROW, COL); //upward-left side traversal

}

}

}

return count;

}

// Driver Code

int main()

{

vector<vector<int>> M = {{1, 1, 0, 0, 0},

{0, 1, 0, 0, 1},

{1, 0, 0, 1, 1},

{0, 0, 0, 0, 0},

{1, 0, 1, 0, 1}};

cout << "Number of islands is: " << countIslands(M);

return 0;

}

**Output**

Number of islands is: 5

**Time complexity:** O(ROW x COL)

The idea is to consider all 1 values as individual sets. Traverse the matrix and do a union of all adjacent 1 vertices. Below are detailed steps.  
**Approach:**   
1) Initialize result (count of islands) as 0   
2) Traverse each index of the 2D matrix.   
3) If the value at that index is 1, check all its 8 neighbours. If a neighbour is also equal to 1, take the union of the index and its neighbour.   
4) Now define an array of size row\*column to store frequencies of all sets.   
5) Now traverse the matrix again.   
6) If the value at index is 1, find its set.   
7) If the frequency of the set in the above array is 0, increment the result be 1.  
Following is implementation of the above steps.

// C++ program to find number of islands

// using Disjoint Set data structure.

#include <bits/stdc++.h>

using namespace std;

// Class to represent

// Disjoint Set Data structure

class DisjointUnionSets

{

vector<int> rank, parent;

int n;

public:

DisjointUnionSets(int n)

{

rank.resize(n);

parent.resize(n);

this->n = n;

makeSet();

}

void makeSet()

{

// Initially, all elements

// are in their own set.

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

parent[i] = i;

}

// Finds the representative of the set

// that x is an element of

int find(int x)

{

if (parent[x] != x)

{

// if x is not the parent of itself,

// then x is not the representative of

// its set.

// so we recursively call Find on its parent

// and move i's node directly under the

// representative of this set

parent[x]=find(parent[x]);

}

return parent[x];

}

// Unites the set that includes x and the set

// that includes y

void Union(int x, int y)

{

// Find the representatives(or the root nodes)

// for x an y

int xRoot = find(x);

int yRoot = find(y);

// Elements are in the same set,

// no need to unite anything.

if (xRoot == yRoot)

return;

// If x's rank is less than y's rank

// Then move x under y so that

// depth of tree remains less

if (rank[xRoot] < rank[yRoot])

parent[xRoot] = yRoot;

// Else if y's rank is less than x's rank

// Then move y under x so that depth of tree

// remains less

else if (rank[yRoot] < rank[xRoot])

parent[yRoot] = xRoot;

else // Else if their ranks are the same

{

// Then move y under x (doesn't matter

// which one goes where)

parent[yRoot] = xRoot;

// And increment the result tree's

// rank by 1

rank[xRoot] = rank[xRoot] + 1;

}

}

};

// Returns number of islands in a[][]

int countIslands(vector<vector<int>>a)

{

int n = a.size();

int m = a[0].size();

DisjointUnionSets \*dus = new DisjointUnionSets(n \* m);

/\* The following loop checks for its neighbours

and unites the indexes if both are 1. \*/

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

{

for (int k = 0; k < m; k++)

{

// If cell is 0, nothing to do

if (a[j][k] == 0)

continue;

// Check all 8 neighbours and do a Union

// with neighbour's set if neighbour is

// also 1

if (j + 1 < n && a[j + 1][k] == 1)

dus->Union(j \* (m) + k,

(j + 1) \* (m) + k);

if (j - 1 >= 0 && a[j - 1][k] == 1)

dus->Union(j \* (m) + k,

(j - 1) \* (m) + k);

if (k + 1 < m && a[j][k + 1] == 1)

dus->Union(j \* (m) + k,

(j) \* (m) + k + 1);

if (k - 1 >= 0 && a[j][k - 1] == 1)

dus->Union(j \* (m) + k,

(j) \* (m) + k - 1);

if (j + 1 < n && k + 1 < m &&

a[j + 1][k + 1] == 1)

dus->Union(j \* (m) + k,

(j + 1) \* (m) + k + 1);

if (j + 1 < n && k - 1 >= 0 &&

a[j + 1][k - 1] == 1)

dus->Union(j \* m + k,

(j + 1) \* (m) + k - 1);

if (j - 1 >= 0 && k + 1 < m &&

a[j - 1][k + 1] == 1)

dus->Union(j \* m + k,

(j - 1) \* m + k + 1);

if (j - 1 >= 0 && k - 1 >= 0 &&

a[j - 1][k - 1] == 1)

dus->Union(j \* m + k,

(j - 1) \* m + k - 1);

}

}

// Array to note down frequency of each set

int \*c = new int[n \* m];

int numberOfIslands = 0;

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

{

for (int k = 0; k < m; k++)

{

if (a[j][k] == 1)

{

int x = dus->find(j \* m + k);

// If frequency of set is 0,

// increment numberOfIslands

if (c[x] == 0)

{

numberOfIslands++;

c[x]++;

}

else

c[x]++;

}

}

}

return numberOfIslands;

}

// Driver Code

int main(void)

{

vector<vector<int>>a = {{1, 1, 0, 0, 0},

{0, 1, 0, 0, 1},

{1, 0, 0, 1, 1},

{0, 0, 0, 0, 0},

{1, 0, 1, 0, 1}};

cout << "Number of Islands is: "

<< countIslands(a) << endl;

}

**Output:** 

Number of Islands is: 5

This problem can also solved by applying BFS() on each component. In each BFS() call, a component or a sub-graph is visited. We will call BFS on the next un-visited component. The number of calls to BFS() gives the number of connected components. BFS can also be used.

A cell in 2D matrix can be connected to 8 neighbours. So, unlike standard BFS(), where we process all adjacent vertices, we process 8 neighbours only. We keep track of the visited 1s so that they are not visited again.

// A BFS based solution to count number of

// islands in a graph.

#include <bits/stdc++.h>

using namespace std;

// R x C matrix

#define R 5

#define C 5

// A function to check if a given cell

// (u, v) can be included in DFS

bool isSafe(int mat[R][C], int i, int j,

bool vis[R][C])

{

return (i >= 0) && (i < R) &&

(j >= 0) && (j < C) &&

(mat[i][j] && !vis[i][j]);

}

void BFS(int mat[R][C], bool vis[R][C],

int si, int sj)

{

// These arrays are used to get row and

// column numbers of 8 neighbours of

// a given cell

int row[] = { -1, -1, -1, 0, 0, 1, 1, 1 };

int col[] = { -1, 0, 1, -1, 1, -1, 0, 1 };

// Simple BFS first step, we enqueue

// source and mark it as visited

queue<pair<int, int> > q;

q.push(make\_pair(si, sj));

vis[si][sj] = true;

// Next step of BFS. We take out

// items one by one from queue and

// enqueue their univisited adjacent

while (!q.empty()) {

int i = q.front().first;

int j = q.front().second;

q.pop();

// Go through all 8 adjacent

for (int k = 0; k < 8; k++) {

if (isSafe(mat, i + row[k],

j + col[k], vis)) {

vis[i + row[k]][j + col[k]] = true;

q.push(make\_pair(i + row[k], j + col[k]));

}

}

}

}

// This function returns number islands (connected

// components) in a graph. It simply works as

// BFS for disconnected graph and returns count

// of BFS calls.

int countIslands(int mat[R][C])

{

// Mark all cells as not visited

bool vis[R][C];

memset(vis, 0, sizeof(vis));

// Call BFS for every unvisited vertex

// Whenever we see an univisted vertex,

// we increment res (number of islands)

// also.

int res = 0;

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

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

if (mat[i][j] && !vis[i][j]) {

BFS(mat, vis, i, j);

res++;

}

}

}

return res;

}

// main function

int main()

{

int mat[][C] = { { 1, 1, 0, 0, 0 },

{ 0, 1, 0, 0, 1 },

{ 1, 0, 0, 1, 1 },

{ 0, 0, 0, 0, 0 },

{ 1, 0, 1, 0, 1 } };

cout << countIslands(mat);

return 0;

}

**Output:**

5

Time Complexity : O(ROW \* COL) where ROW is number of ROWS and COL is number of COLUMNS in the matrix.

# 345. Given a sorted Dictionary of an Alien Language, find order of characters

Given a sorted dictionary of an alien language having N words and k starting alphabets of standard dictionary. Find the order of characters in the alien language.  
**Note:** Many orders may be possible for a particular test case, thus you may return any valid order and output will be 1 if the order of string returned by the function is correct else 0 denoting incorrect string returned.

**Example 1:**

**Input:**

N = 5, K = 4

dict = {"baa","abcd","abca","cab","cad"}

**Output:**

1

**Explanation:**

Here order of characters is

'b', 'd', 'a', 'c' Note that words are sorted

and in the given language "baa" comes before

"abcd", therefore 'b' is before 'a' in output.

Similarly we can find other orders.

**Example 2:**

**Input:**

N = 3, K = 3

dict = {"caa","aaa","aab"}

**Output:**

1

**Explanation:**

Here order of characters is

'c', 'a', 'b' Note that words are sorted

and in the given language "caa" comes before

"aaa", therefore 'c' is before 'a' in output.

Similarly we can find other orders.

**Your Task:**  
You don't need to read or print anything. Your task is to complete the function **findOrder()**which takes  the string array dict[], its size N and the integer K as input parameter and returns a string denoting the order of characters in the alien language.

**Expected Time Complexity:**O(N \* |S| + K) , where |S| denotes maximum length.  
**Expected Space Compelxity:**O(K)

**Constraints:**  
1 ≤ N, M ≤ 300  
1 ≤ K ≤ 26  
1 ≤ Length of words ≤ 50

## Solution:

**Approach 1:**

The idea is to create a graph of characters and then find [topological sorting](https://www.geeksforgeeks.org/topological-sorting/) of the created graph. Following are the detailed steps.  
1) Create a graph *g* with number of vertices equal to the size of alphabet in the given alien language. For example, if the alphabet size is 5, then there can be 5 characters in words. Initially there are no edges in graph.  
2) Do following for every pair of adjacent words in given sorted array.   
…..a) Let the current pair of words be *word1*and *word2*. One by one compare characters of both words and find the first mismatching characters.   
…..b) Create an edge in *g* from mismatching character of *word1*to that of *word2*.  
3) Print [topological sorting](https://www.geeksforgeeks.org/topological-sorting/) of the above created graph.

*The implementation of the above in C++.*

string findOrder(string dict[], int N, int K) {

vector<int> indegree(K, 0);

vector<int> adj[K];

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

int len = min(dict[i-1].size(), dict[i].size()), j=0;

while(j<len && dict[i-1][j]==dict[i][j]){

j++;

}

if(j!=len){

indegree[dict[i][j]-'a']++;

adj[dict[i-1][j]-'a'].push\_back(dict[i][j]-'a');

}

}

queue<int> q;

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

if(indegree[i]==0)

q.push(i);

string res = "";

while(!q.empty()){

int f = q.front(); q.pop();

res.push\_back('a'+f);

for(int i=0;i<adj[f].size();i++){

indegree[adj[f][i]]--;

if(indegree[adj[f][i]]==0)

q.push(adj[f][i]);

}

}

return res;

}

**Time Complexity:** The first step to create a graph takes O(n + alpha) time where n is number of given words and alpha is number of characters in given alphabet. The second step is also topological sorting. Note that there would be alpha vertices and at-most (n-1) edges in the graph. The time complexity of topological sorting is O(V+E) which is O(n + alpha) here. So overall time complexity is O(n + alpha) + O(n + alpha) which is O(n + alpha).

**Exercise:**The above code doesn’t work when the input is not valid. For example {“aba”, “bba”, “aaa”} is not valid, because from first two words, we can deduce ‘a’ should appear before ‘b’, but from last two words, we can deduce ‘b’ should appear before ‘a’ which is not possible. Extend the above program to handle invalid inputs and generate the output as “Not valid”.

**Approach 2: [Works for invalid input data]**

*We have implemented this approach in C#.*

Algorithm:

(1) Compare 2 adjacent words at a time (i.e, word1 with word2, word2 with word3, … , word(startIndex) and word(startIndex + 1)

(2) Then we compare one character at a time for the 2 words selected.

(2a) If both characters are different, we stop the comparison here and conclude that the character from word(startIndex) comes before the other.

(2b) If both characters are the same, we continue to compare until (2a) occurs or if either of the words has been exhausted.

(3) We continue to compare each word in this fashion until we have compared all words.

Once we find a character set in (2a) we pass them to class ‘AlienCharacters’ which takes care of the overall ordering of the characters. The idea is to maintain the ordering of the characters in a linked list (DNode). To optimize the insertion time into the linked list, a map (C# Dictionary) is used as an indexing entity, thus, bringing down the complexity to O(1). This is an improvement from the previous algorithm where topological sort was used for the purpose.

Boundary conditions:

1. The startIndex must be within range

2. When comparing 2 words, if we exhaust on one i.e, the length of both words is different. Compare only until either one exhausts.

Complexity Analysis:

The method-wise time complexities have been mentioned in the code below (C#) for better understanding.

If ‘N’ is the number of words in the input alien vocabulary/dictionary, ‘L’ length of the max length word, and ‘C’ is the final number of unique characters,

Time Complexity: O(N \* L)

Space Complexity: O(C)

using System;

using System.Collections.Generic;

using System.Linq;

namespace AlienDictionary

{

public class DNode

{

public string Char;

public DNode prev = null;

public DNode next = null;

public DNode(string character) => Char = character;

}

public class AlienCharacters

{

public AlienCharacters(int k) => MaxChars = k;

private int MaxChars;

private DNode head = null;

private Dictionary<string, DNode> index = new Dictionary<string, DNode>();

// As we use Dictionary for indexing, the time complexity for inserting

// characters in order will take O(1)

// Time: O(1)

// Space: O(c), where 'c' is the unique character count.

public bool UpdateCharacterOrdering(string predChar, string succChar)

{

DNode pNode = null, sNode = null;

bool isSNodeNew = false, isPNodeNew = false;

if(!index.TryGetValue(predChar, out pNode))

{

pNode = new DNode(predChar);

index[predChar] = pNode;

isPNodeNew = true;

}

if (!index.TryGetValue(succChar, out sNode))

{

sNode = new DNode(succChar);

index[succChar] = sNode;

isSNodeNew = true;

}

// before ordering is formed, validate if both the nodes are already present

if (!isSNodeNew && !isPNodeNew)

{

if (!Validate(predChar, succChar))

return false;

}

else if ((isPNodeNew && !isSNodeNew) || (isPNodeNew && isSNodeNew))

InsertNodeBefore(ref pNode, ref sNode);

else

InsertNodeAfter(ref pNode, ref sNode);

if (pNode.prev == null)

head = pNode;

return true;

}

// Time: O(1)

private void InsertNodeAfter(ref DNode pNode, ref DNode sNode)

{

sNode.next = pNode?.next;

if (pNode.next != null)

pNode.next.prev = sNode;

pNode.next = sNode;

sNode.prev = pNode;

}

// Time: O(1)

private void InsertNodeBefore(ref DNode pNode, ref DNode sNode)

{

// insert pnode before snode

pNode.prev = sNode?.prev;

if (sNode.prev != null)

sNode.prev.next = pNode;

sNode.prev = pNode;

pNode.next = sNode;

}

// Time: O(1)

private bool Validate(string predChar, string succChar)

{

// this is the first level of validation

// validate if predChar node actually occurs before succCharNode.

DNode sNode = index[succChar];

while(sNode != null)

{

if (sNode.Char != predChar)

sNode = sNode.prev;

else

return true; // validation successful

}

// if we have reached the end and not found the predChar before succChar

// something is not right!

return false;

}

// Time: O(c), where 'c' is the unique character count.

public override string ToString()

{

string res = "";

int count = 0;

DNode currNode = head;

while(currNode != null)

{

res += currNode.Char + " ";

count++;

currNode = currNode.next;

}

// second level of validation

if (count != MaxChars) // something went wrong!

res = "ERROR!!! Input words not enough to find all k unique characters.";

return res;

}

}

class Program

{

static int k = 4;

static AlienCharacters alienCharacters = new AlienCharacters(k);

static List<string> vocabulary = new List<string>();

static void Main(string[] args)

{

vocabulary.Add("baa");

vocabulary.Add("abcd");

vocabulary.Add("abca");

vocabulary.Add("cab");

vocabulary.Add("cad");

ProcessVocabulary(0);

Console.WriteLine(alienCharacters.ToString());

Console.ReadLine();

}

// Time: O(vocabulary.Count + max(word.Length))

static void ProcessVocabulary(int startIndex)

{

if (startIndex >= vocabulary.Count - 1)

return;

var res = GetPredSuccChar(vocabulary.ElementAt(startIndex), vocabulary.ElementAt(startIndex + 1));

if (res != null)

{

if (!alienCharacters.UpdateCharacterOrdering(res.Item1, res.Item2))

{

Console.WriteLine("ERROR!!! Invalid input data, the words maybe in wrong order");

return;

}

}

ProcessVocabulary(startIndex + 1);

}

//Time: O(max(str1.Length, str2.Length)

static Tuple<string, string> GetPredSuccChar(string str1, string str2)

{

Tuple<string, string> result = null;

if (str1.Length == 0 || str2.Length == 0)

return null; // invalid condition.

if(str1[0] != str2[0]) // found an ordering

{

result = new Tuple<string, string>(str1[0].ToString(), str2[0].ToString());

return result;

}

string s1 = str1.Substring(1, str1.Length - 1);

string s2 = str2.Substring(1, str2.Length - 1);

if (s1.Length == 0 || s2.Length == 0)

return null; // recursion can stop now.

return GetPredSuccChar(s1, s2);

}

}

}

**Output:**

c a b

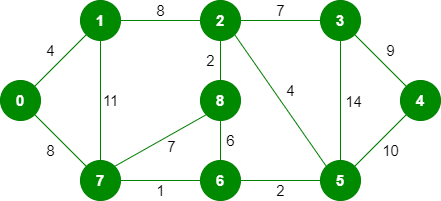
# 346. [Implement Kruksal’sAlgorithm](https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/)

***What is Minimum Spanning Tree?***   
Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.  
*How many edges does a minimum spanning tree has?*   
A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.   
*What are the applications of*the *Minimum Spanning Tree?*   
See [this](https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/)for applications of MST.

Below are the steps for finding MST using Kruskal’s algorithm

***1.****Sort all the edges in non-decreasing order of their weight.****2.****Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.****3.****Repeat step#2 until there are (V-1) edges in the spanning tree.*

Step #2 uses the [Union-Find algorithm](https://www.geeksforgeeks.org/union-find/) to detect cycles. So we recommend reading the following post as a prerequisite.   
[Union-Find Algorithm | Set 1 (Detect Cycle in a Graph)](https://www.geeksforgeeks.org/union-find/)   
[Union-Find Algorithm | Set 2 (Union By Rank and Path Compression)](https://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/)  
The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph. 



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

After sorting:

Weight Src Dest

1 7 6

2 8 2

2 6 5

4 0 1

4 2 5

6 8 6

7 2 3

7 7 8

8 0 7

8 1 2

9 3 4

10 5 4

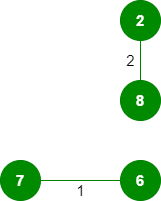
11 1 7

14 3 5

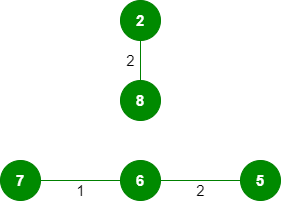
Now pick all edges one by one from the sorted list of edges   
**1.** *Pick edge 7-6:* No cycle is formed, include it. 

Kruskal’s Minimum Spanning Tree Algorithm

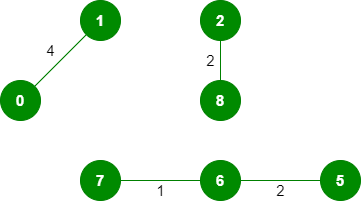
**2.** *Pick edge 8-2:* No cycle is formed, include it. 



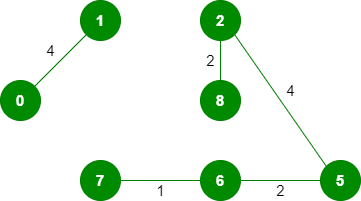
**3.** *Pick edge 6-5:* No cycle is formed, include it. 



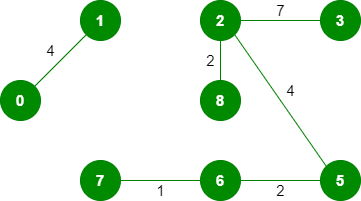
**4.** *Pick edge 0-1:* No cycle is formed, include it. 



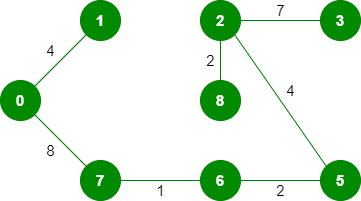
**5.** *Pick edge 2-5:* No cycle is formed, include it. 



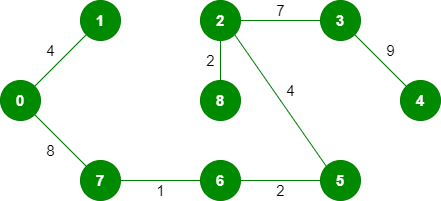
**6.***Pick edge 8-6:*Since including this edge results in the cycle, discard it.  
**7.** *Pick edge 2-3:* No cycle is formed, include it. 



**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.

Below is the implementation of the above idea:

#include <bits/stdc++.h>

using namespace std;

// DSU data structure

// path compression + rank by union

class DSU {

int\* parent;

int\* rank;

public:

DSU(int n)

{

parent = new int[n];

rank = new int[n];

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

parent[i] = -1;

rank[i] = 1;

}

}

// Find function

int find(int i)

{

if (parent[i] == -1)

return i;

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

}

// union function

void unite(int x, int y)

{

int s1 = find(x);

int s2 = find(y);

if (s1 != s2) {

if (rank[s1] < rank[s2]) {

parent[s1] = s2;

rank[s2] += rank[s1];

}

else {

parent[s2] = s1;

rank[s1] += rank[s2];

}

}

}

};

class Graph {

vector<vector<int> > edgelist;

int V;

public:

Graph(int V) { this->V = V; }

void addEdge(int x, int y, int w)

{

edgelist.push\_back({ w, x, y });

}

void kruskals\_mst()

{

// 1. Sort all edges

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

// Initialize the DSU

DSU s(V);

int ans = 0;

cout << "Following are the edges in the "

"constructed MST"

<< endl;

for (auto edge : edgelist) {

int w = edge[0];

int x = edge[1];

int y = edge[2];

// take that edge in MST if it does form a cycle

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

s.unite(x, y);

ans += w;

cout << x << " -- " << y << " == " << w

<< endl;

}

}

cout << "Minimum Cost Spanning Tree: " << ans;

}

};

int main()

{

/\* Let us create following weighted graph

10

0--------1

| \ |

6| 5\ |15

| \ |

2--------3

4 \*/

Graph g(4);

g.addEdge(0, 1, 10);

g.addEdge(1, 3, 15);

g.addEdge(2, 3, 4);

g.addEdge(2, 0, 6);

g.addEdge(0, 3, 5);

// int n, m;

// cin >> n >> m;

// Graph g(n);

// for (int i = 0; i < m; i++)

// {

// int x, y, w;

// cin >> x >> y >> w;

// g.addEdge(x, y, w);

// }

g.kruskals\_mst();

return 0;

}

**Output**

Following are the edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Cost Spanning Tree: 19

**Time Complexity:** O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be at most O(V2), so O(LogV) is O(LogE) the same. Therefore, the overall time complexity is O(ElogE) or O(ElogV)

# 347. Implement Prim’s Algorithm

Prim’s algorithm is also a [Greedy algorithm](https://www.geeksforgeeks.org/archives/18528). It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.   
A group of edges that connects two set of vertices in a graph is called [cut in graph theory](http://en.wikipedia.org/wiki/Cut_%28graph_theory%29). *So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).*

***How does Prim’s Algorithm Work?*** The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning*Tree. And they must be connected with the minimum weight edge to make it a *Minimum*Spanning Tree.

***Algorithm***   
**1)** Create a set *mstSet* that keeps track of vertices already included in MST.   
**2)** Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.   
**3)** While mstSet doesn’t include all vertices   
….**a)** Pick a vertex *u* which is not there in *mstSet*and has minimum key value.   
….**b)** Include *u*to mstSet.   
….**c)** Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*  
The idea of using key values is to pick the minimum weight edge from [cut](http://en.wikipedia.org/wiki/Cut_(graph_theory)). The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST. 

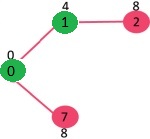
Let us understand with the following example: 

[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-11.jpg)

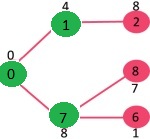
The set *mstSet*is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with the minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet*becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.

[](https://www.geeksforgeeks.org/wp-content/uploads/MST1.jpg)

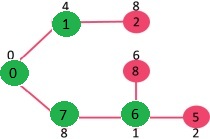
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.

[](https://www.geeksforgeeks.org/wp-content/uploads/MST2.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively). 

[](https://www.geeksforgeeks.org/wp-content/uploads/MST3.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.

[](https://www.geeksforgeeks.org/wp-content/uploads/MST4.jpg)

We repeat the above steps until *mstSet*includes all vertices of given graph. Finally, we get the following graph.

[](https://www.geeksforgeeks.org/wp-content/uploads/MST5.jpg)

***How to implement the above algorithm?***   
We use a boolean array mstSet[] to represent the set of vertices included in MST. If a value mstSet[v] is true, then vertex v is included in MST, otherwise not. Array key[] is used to store key values of all vertices. Another array parent[] to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

// 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;

// Number of vertices in the graph

#define V 5

// A utility function to find the vertex with

// minimum key value, from the set of vertices

// not yet included in MST

int minKey(int key[], bool mstSet[])

{

// Initialize min value

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; 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(int parent[], int graph[V][V])

{

cout<<"Edge \tWeight\n";

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

cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";

}

// Function to construct and print MST for

// a graph represented using adjacency

// matrix representation

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

{

// Array to store constructed MST

int parent[V];

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

int key[V];

// To represent set of vertices included in MST

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;

parent[0] = -1; // First node is always root of MST

// 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 code

int main()

{

/\* Let us create the following graph

2 3

(0)--(1)--(2)

| / \ |

6| 8/ \5 |7

| / \ |

(3)-------(4)

9 \*/

int graph[V][V] = { { 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

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

{ 0, 5, 7, 9, 0 } };

// Print the solution

primMST(graph);

return 0;

}

**Output:**

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

Time Complexity of the above program is O(V^2). If the input [graph is represented using adjacency list](https://www.geeksforgeeks.org/archives/27134), then the time complexity of Prim’s algorithm can be reduced to O(E log V) with the help of binary heap.

# Prim’s MST for Adjacency List Representation

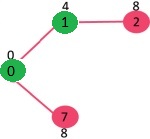
The time complexity for the matrix representation is O(V^2). In this post, O(ELogV) algorithm for adjacency list representation is discussed.   
As discussed in the previous post, in Prim’s algorithm, two sets are maintained, one set contains list of vertices already included in MST, other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in O(V+E) time using [BFS](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/). The idea is to traverse all vertices of graph using [BFS](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)and use a Min Heap to store the vertices not yet included in MST. Min Heap is used as a priority queue to get the minimum weight edge from the [cut](http://en.wikipedia.org/wiki/Cut_%28graph_theory%29). Min Heap is used as time complexity of operations like extracting minimum element and decreasing key value is O(LogV) in Min Heap.  
Following are the detailed steps.   
**1)**Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and key value of the vertex.   
**2)** Initialize Min Heap with first vertex as root (the key value assigned to first vertex is 0). The key value assigned to all other vertices is INF (infinite).   
**3)**While Min Heap is not empty, do following   
…..**a)** Extract the min value node from Min Heap. Let the extracted vertex be u.   
…..**b)** For every adjacent vertex v of u, check if v is in Min Heap (not yet included in MST). If v is in Min Heap and its key value is more than weight of u-v, then update the key value of v as weight of u-v.  
Let us understand the above algorithm with the following example: 



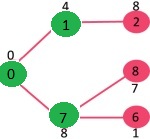
Initially, key value of first vertex is 0 and INF (infinite) for all other vertices. So vertex 0 is extracted from Min Heap and key values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.   
The vertices in green color are the vertices included in MST. 



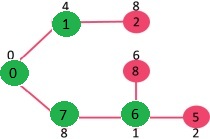
Since key value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 1 are updated (Key is updated if the a vertex is in Min Heap and previous key value is greater than the weight of edge from 1 to the adjacent). Min Heap contains all vertices except vertex 0 and 1. 



Since key value of vertex 7 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 7 are updated (Key is updated if the a vertex is in Min Heap and previous key value is greater than the weight of edge from 7 to the adjacent). Min Heap contains all vertices except vertex 0, 1 and 7. 



Since key value of vertex 6 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 6 are updated (Key is updated if the a vertex is in Min Heap and previous key value is greater than the weight of edge from 6 to the adjacent). Min Heap contains all vertices except vertex 0, 1, 7 and 6. 



The above steps are repeated for rest of the nodes in Min Heap till Min Heap becomes empty 



// C / C++ program for Prim's MST for adjacency list representation of graph

#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

// A structure to represent a node in adjacency list

struct AdjListNode {

int dest;

int weight;

struct AdjListNode\* next;

};

// A structure to represent an adjacency list

struct AdjList {

struct AdjListNode\* head; // pointer to head node of list

};

// A structure to represent a graph. A graph is an array of adjacency lists.

// Size of array will be V (number of vertices in graph)

struct Graph {

int V;

struct AdjList\* array;

};

// A utility function to create a new adjacency list node

struct AdjListNode\* newAdjListNode(int dest, int weight)

{

struct AdjListNode\* newNode = (struct AdjListNode\*)malloc(sizeof(struct AdjListNode));

newNode->dest = dest;

newNode->weight = weight;

newNode->next = NULL;

return newNode;

}

// A utility function that creates a graph of V vertices

struct Graph\* createGraph(int V)

{

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->V = V;

// Create an array of adjacency lists. Size of array will be V

graph->array = (struct AdjList\*)malloc(V \* sizeof(struct AdjList));

// Initialize each adjacency list as empty by making head as NULL

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

graph->array[i].head = NULL;

return graph;

}

// Adds an edge to an undirected graph

void addEdge(struct Graph\* graph, int src, int dest, int weight)

{

// Add an edge from src to dest. A new node is added to the adjacency

// list of src. The node is added at the beginning

struct AdjListNode\* newNode = newAdjListNode(dest, weight);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

// Since graph is undirected, add an edge from dest to src also

newNode = newAdjListNode(src, weight);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

// Structure to represent a min heap node

struct MinHeapNode {

int v;

int key;

};

// Structure to represent a min heap

struct MinHeap {

int size; // Number of heap nodes present currently

int capacity; // Capacity of min heap

int\* pos; // This is needed for decreaseKey()

struct MinHeapNode\*\* array;

};

// A utility function to create a new Min Heap Node

struct MinHeapNode\* newMinHeapNode(int v, int key)

{

struct MinHeapNode\* minHeapNode = (struct MinHeapNode\*)malloc(sizeof(struct MinHeapNode));

minHeapNode->v = v;

minHeapNode->key = key;

return minHeapNode;

}

// A utilit function to create a Min Heap

struct MinHeap\* createMinHeap(int capacity)

{

struct MinHeap\* minHeap = (struct MinHeap\*)malloc(sizeof(struct MinHeap));

minHeap->pos = (int\*)malloc(capacity \* sizeof(int));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct MinHeapNode\*\*)malloc(capacity \* sizeof(struct MinHeapNode\*));

return minHeap;

}

// A utility function to swap two nodes of min heap. Needed for min heapify

void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b)

{

struct MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

// A standard function to heapify at given idx

// This function also updates position of nodes when they are swapped.

// Position is needed for decreaseKey()

void minHeapify(struct MinHeap\* minHeap, int idx)

{

int smallest, left, right;

smallest = idx;

left = 2 \* idx + 1;

right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->key < minHeap->array[smallest]->key)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->key < minHeap->array[smallest]->key)

smallest = right;

if (smallest != idx) {

// The nodes to be swapped in min heap

MinHeapNode\* smallestNode = minHeap->array[smallest];

MinHeapNode\* idxNode = minHeap->array[idx];

// Swap positions

minHeap->pos[smallestNode->v] = idx;

minHeap->pos[idxNode->v] = smallest;

// Swap nodes

swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

// A utility function to check if the given minHeap is ampty or not

int isEmpty(struct MinHeap\* minHeap)

{

return minHeap->size == 0;

}

// Standard function to extract minimum node from heap

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap)

{

if (isEmpty(minHeap))

return NULL;

// Store the root node

struct MinHeapNode\* root = minHeap->array[0];

// Replace root node with last node

struct MinHeapNode\* lastNode = minHeap->array[minHeap->size - 1];

minHeap->array[0] = lastNode;

// Update position of last node

minHeap->pos[root->v] = minHeap->size - 1;

minHeap->pos[lastNode->v] = 0;

// Reduce heap size and heapify root

--minHeap->size;

minHeapify(minHeap, 0);

return root;

}

// Function to decrease key value of a given vertex v. This function

// uses pos[] of min heap to get the current index of node in min heap

void decreaseKey(struct MinHeap\* minHeap, int v, int key)

{

// Get the index of v in heap array

int i = minHeap->pos[v];

// Get the node and update its key value

minHeap->array[i]->key = key;

// Travel up while the complete tree is not hepified.

// This is a O(Logn) loop

while (i && minHeap->array[i]->key < minHeap->array[(i - 1) / 2]->key) {

// Swap this node with its parent

minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;

minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;

swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

// move to parent index

i = (i - 1) / 2;

}

}

// A utility function to check if a given vertex

// 'v' is in min heap or not

bool isInMinHeap(struct MinHeap\* minHeap, int v)

{

if (minHeap->pos[v] < minHeap->size)

return true;

return false;

}

// A utility function used to print the constructed MST

void printArr(int arr[], int n)

{

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

printf("%d - %d\n", arr[i], i);

}

// The main function that constructs Minimum Spanning Tree (MST)

// using Prim's algorithm

void PrimMST(struct Graph\* graph)

{

int V = graph->V; // Get the number of vertices in graph

int parent[V]; // Array to store constructed MST

int key[V]; // Key values used to pick minimum weight edge in cut

// minHeap represents set E

struct MinHeap\* minHeap = createMinHeap(V);

// Initialize min heap with all vertices. Key value of

// all vertices (except 0th vertex) is initially infinite

for (int v = 1; v < V; ++v) {

parent[v] = -1;

key[v] = INT\_MAX;

minHeap->array[v] = newMinHeapNode(v, key[v]);

minHeap->pos[v] = v;

}

// Make key value of 0th vertex as 0 so that it

// is extracted first

key[0] = 0;

minHeap->array[0] = newMinHeapNode(0, key[0]);

minHeap->pos[0] = 0;

// Initially size of min heap is equal to V

minHeap->size = V;

// In the following loop, min heap contains all nodes

// not yet added to MST.

while (!isEmpty(minHeap)) {

// Extract the vertex with minimum key value

struct MinHeapNode\* minHeapNode = extractMin(minHeap);

int u = minHeapNode->v; // Store the extracted vertex number

// Traverse through all adjacent vertices of u (the extracted

// vertex) and update their key values

struct AdjListNode\* pCrawl = graph->array[u].head;

while (pCrawl != NULL) {

int v = pCrawl->dest;

// If v is not yet included in MST and weight of u-v is

// less than key value of v, then update key value and

// parent of v

if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v]) {

key[v] = pCrawl->weight;

parent[v] = u;

decreaseKey(minHeap, v, key[v]);

}

pCrawl = pCrawl->next;

}

}

// print edges of MST

printArr(parent, V);

}

// Driver program to test above functions

int main()

{

// Let us create the graph given in above fugure

int V = 9;

struct Graph\* graph = createGraph(V);

addEdge(graph, 0, 1, 4);

addEdge(graph, 0, 7, 8);

addEdge(graph, 1, 2, 8);

addEdge(graph, 1, 7, 11);

addEdge(graph, 2, 3, 7);

addEdge(graph, 2, 8, 2);

addEdge(graph, 2, 5, 4);

addEdge(graph, 3, 4, 9);

addEdge(graph, 3, 5, 14);

addEdge(graph, 4, 5, 10);

addEdge(graph, 5, 6, 2);

addEdge(graph, 6, 7, 1);

addEdge(graph, 6, 8, 6);

addEdge(graph, 7, 8, 7);

PrimMST(graph);

return 0;

}

Output: 

0 - 1

5 - 2

2 - 3

3 - 4

6 - 5

7 - 6

0 - 7

2 - 8

**Time Complexity:** The time complexity of the above code/algorithm looks O(V^2) as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed O(V+E) times (similar to BFS). The inner loop has decreaseKey() operation which takes O(LogV) time. So overall time complexity is O(E+V)\*O(LogV) which is O((E+V)\*LogV) = O(ELogV) (For a connected graph, V = O(E))

**My Implementation:**

class Solution

{

public:

int minKey(vector<int> key, unordered\_set<int> st){

int min = INT\_MAX, minIndex = -1;

for(auto i = st.begin(); i!=st.end(); i++){

if(min>key[\*i]){

min = key[\*i];

minIndex = \*i;

}

}

return minIndex;

}

//Function to find sum of weights of edges of the Minimum Spanning Tree.

int spanningTree(int V, vector<vector<int>> adj[])

{

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

unordered\_set<int> st;

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

st.insert(i);

}

int ans = 0;

key[0] = 0;

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

int u = minKey(key, st);

ans += key[u];

st.erase(u);

for(int j=0;j<adj[u].size();j++){

if(st.find(adj[u][j][0])!=st.end() && key[adj[u][j][0]]>adj[u][j][1]){

key[adj[u][j][0]] = adj[u][j][1];

}

}

}

return ans;

}

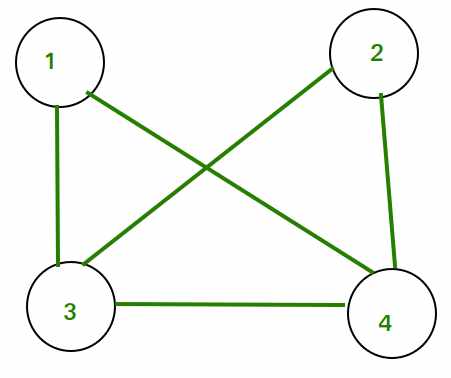
};

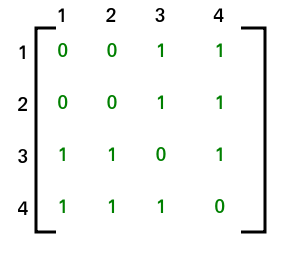
**Time Complexity:** O((V+E)\*V)

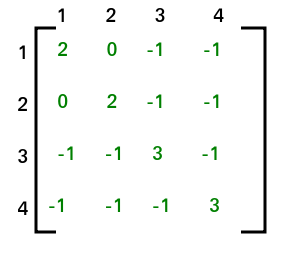
# 348. Total no. of Spanning tree in a graph

If a graph is a complete graph with n vertices, then total number of spanning trees is n(n-2) where n is the number of nodes in the graph. In complete graph, the task is equal to counting different labeled trees with n nodes for which have [Cayley’s formula](https://www.geeksforgeeks.org/g-fact-20-cayleys-formula-for-number-of-labelled-trees/).

**What if graph is not complete?**  
Follow the given procedure :-  
STEP 1: Create Adjacency Matrix for the given graph.  
STEP 2: Replace all the diagonal elements with the degree of nodes. For eg. element at (1,1) position of adjacency matrix will be replaced by the degree of node 1, element at (2,2) position of adjacency matrix will be replaced by the degree of node 2, and so on.  
STEP 3: Replace all non-diagonal 1’s with -1.  
STEP 4: Calculate co-factor for any element.  
STEP 5: The cofactor that you get is the total number of spanning tree for that graph.

Consider the following graph:  


Adjacency Matrix for the above graph will be as follows:  


After applying STEP 2 and STEP 3, adjacency matrix will look like  


The co-factor for (1, 1) is 8. Hence total no. of spanning tree that can be formed is 8.  
NOTE- Co-factor for all the elements will be same. Hence we can compute co-factor for any element of the matrix.

This method is also known as [Kirchhoff’s Theorem](https://en.wikipedia.org/wiki/Kirchhoff%27s_theorem). It can be applied to complete graphs also.

# 349. Implement Bellman Ford Algorithm

Given a graph and a source vertex *src*in graph, find shortest paths from *src*to all vertices in the given graph. The graph may contain negative weight edges.   
We have discussed [Dijkstra’s algorithm](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/) for this problem. Dijkstra’s algorithm is a Greedy algorithm and time complexity is O((V+E)LogV) (with the use of Fibonacci heap). *Dijkstra doesn’t work for Graphs with negative weights, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.*

**Algorithm**   
Following are the detailed steps.  
*Input:* Graph and a source vertex *src*   
*Output:* Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.  
**1)** This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.  
**2)** This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.   
…..**a)** Do following for each edge u-v   
………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]   
………………….dist[v] = dist[u] + weight of edge uv  
**3)** This step reports if there is a negative weight cycle in graph. Do following for each edge u-v   
……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”   
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn’t contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle  
***How does this work?*** Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges (Proof is simple, you can refer [this](http://courses.csail.mit.edu/6.006/spring11/lectures/lec15.pdf) or [MIT Video Lecture](http://www.youtube.com/watch?v=Ttezuzs39nk))  
**Example**   
Let us understand the algorithm with following example graph. The images are taken from [this](http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.prn.pdf)source.  
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*

Bellman–Ford Algorithm Example Graph 1

Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed. 

Bellman–Ford Algorithm Example Graph 2

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values). 

Bellman–Ford Algorithm Example Graph 3

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances.  
**Implementation:**

// A C++ program for Bellman-Ford's single source

// shortest path algorithm.

#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph

struct Edge {

int src, dest, weight;

};

// 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.

struct 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)

{

printf("Vertex Distance from Source\n");

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

printf("%d \t\t %d\n", i, dist[i]);

}

// 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 src)

{

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[src] = 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 u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INT\_MAX

&& dist[u] + weight < dist[v])

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

}

}

// 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 u = graph->edge[i].src;

int v = graph->edge[i].dest;

int weight = graph->edge[i].weight;

if (dist[u] != INT\_MAX

&& dist[u] + weight < dist[v]) {

printf("Graph contains negative weight cycle");

return; // If negative cycle is detected, simply

// return

}

}

printArr(dist, V);

return;

}

// Driver program to test above functions

int main()

{

/\* Let us create the graph given in above example \*/

int V = 5; // Number of vertices in graph

int E = 8; // Number of edges in graph

struct Graph\* graph = createGraph(V, E);

// add edge 0-1 (or A-B in above figure)

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = -1;

// add edge 0-2 (or A-C in above figure)

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 4;

// add edge 1-2 (or B-C in above figure)

graph->edge[2].src = 1;

graph->edge[2].dest = 2;

graph->edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 2;

// add edge 1-4 (or B-E in above figure)

graph->edge[4].src = 1;

graph->edge[4].dest = 4;

graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)

graph->edge[5].src = 3;

graph->edge[5].dest = 2;

graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)

graph->edge[6].src = 3;

graph->edge[6].dest = 1;

graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)

graph->edge[7].src = 4;

graph->edge[7].dest = 3;

graph->edge[7].weight = -3;

BellmanFord(graph, 0);

return 0;

}

**Output:**

Vertex Distance from Source

0 0

1 -1

2 2

3 -2

4 1

**Notes**   
**1)**Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.  
**2)** Bellman-Ford works better (better than Dijkstra’s) for distributed systems. Unlike Dijkstra’s where we need to find the minimum value of all vertices, in Bellman-Ford, edges are considered one by one.                                                                    
**3)**Bellman-Ford does not work with undirected graph with negative edges as it will declared as negative cycle.

# 350. Implement Floyd warshallAlgorithm

The [Floyd Warshall Algorithm](http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm) is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.   
**Example:**

**Input:**

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

{INF, 0, 3, INF},

{INF, INF, 0, 1},

{INF, INF, INF, 0} }

which represents the following graph

10

(0)------->(3)

| /|\

5 | |

| | 1

\|/ |

(1)------->(2)

3

Note that the value of graph[i][j] is 0 if i is equal to j

And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

**Output:**

Shortest distance matrix

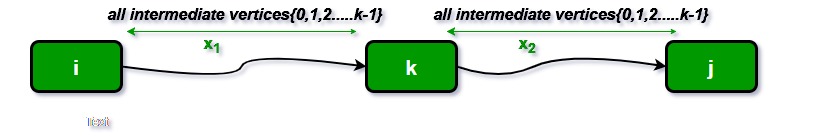
0 5 8 9

INF 0 3 4

INF INF 0 1

INF INF INF 0

**Floyd Warshall Algorithm**   
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. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.   
**1)** k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.   
**2)** k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]  
The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



Following is implementations of the Floyd Warshall algorithm.

// C++ Program for Floyd Warshall Algorithm

#include <bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 4

/\* Define Infinite as a large enough

value.This value will be used for

vertices not connected to each other \*/

#define INF 99999

// A function to print the solution matrix

void printSolution(int dist[][V]);

// Solves the all-pairs shortest path

// problem using Floyd Warshall algorithm

void floydWarshall(int graph[][V])

{

/\* dist[][] will be the output matrix

that will finally have the shortest

distances between every pair of vertices \*/

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

/\* Initialize the solution matrix same

as input graph matrix. Or we can say

the initial values of shortest distances

are based on shortest paths considering

no intermediate vertex. \*/

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

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

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

/\* Add all vertices one by one to

the set of intermediate vertices.

---> Before start of an iteration,

we have shortest distances between all

pairs of vertices such that the

shortest distances consider only the

vertices in set {0, 1, 2, .. k-1} as

intermediate vertices.

----> After the end of an iteration,

vertex no. k is added to the set of

intermediate vertices and the set becomes {0, 1, 2, ..

k} \*/

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

// Pick all vertices as source one by one

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

// Pick all vertices as destination for the

// above picked source

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

// If vertex k is on the shortest path from

// i to j, then update the value of

// dist[i][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];

}

}

}

// Print the shortest distance matrix

printSolution(dist);

}

/\* A utility function to print solution \*/

void printSolution(int dist[][V])

{

cout << "The following matrix shows the shortest "

"distances"

" between every pair of vertices \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;

}

}

// Driver code

int main()

{

/\* Let us create the following weighted graph

10

(0)------->(3)

| /|\

5 | |

| | 1

\|/ |

(1)------->(2)

3 \*/

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

{ INF, 0, 3, INF },

{ INF, INF, 0, 1 },

{ INF, INF, INF, 0 } };

// Print the solution

floydWarshall(graph);

return 0;

}

**Output:**

Following matrix shows the shortest distances between every pair of vertices

0 5 8 9

INF 0 3 4

INF INF 0 1

INF INF INF 0

**Time Complexity:**O(V^3)  
The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.   
Also, the value of INF can be taken as INT\_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT\_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

#include

#define INF INT\_MAX

..........................

if ( dist[i][k] != INF &&

dist[k][j] != INF &&

dist[i][k] + dist[k][j] < dist[i][j]

)

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

...........................

**Floyd Warshall**

The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph. The Graph is represented as adjancency matrix, and the matrix denotes the weight of the edegs (if it exists) else -1. **Do it in-place.**

**Example 1:**

**Input:** matrix = {{0,25},{-1,0}}

**Output:** {{0,25},{-1,0}}

**Explanation:** The shortest distance between

every pair is already given(if it exists).

**Example 2:**

**Input:** matrix = {{0,1,43},{1,0,6},{-1,-1,0}}

**Output:** {{0,1,7},{1,0,6},{-1,-1,0}}

**Explanation:** We can reach 3 from 1 as 1->2->3

and the cost will be 1+6=7 which is less than

43.

**Your Task:**  
You don't need to read, return or print anything. Your task is to complete the function **shortest\_distance()**which takes the matrix as input parameter and modify the distances for every pair in-place.

**Expected Time Complexity:**O(n3)  
**Expected Space Compelxity:**O(1)

**Constraints:**  
1 <= n <= 100

## Solution:

class Solution {

public:

void shortest\_distance(vector<vector<int>>&matrix){

int n = matrix.size();

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

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

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

if(i!=x && j!=x && matrix[i][x]!=-1 && matrix[x][j]!=-1){

if(matrix[i][x]+matrix[x][j] < matrix[i][j]){

matrix[i][j] = matrix[i][x]+matrix[x][j];

}

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

matrix[i][j] = matrix[i][x]+matrix[x][j];

}

}

}

}

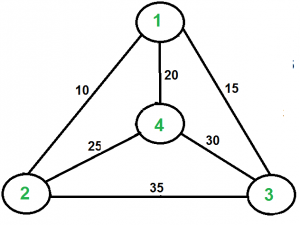
}

}

};

# 351. Travelling Salesman Problem

[Travelling Salesman Problem (TSP) :](https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/) Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.   
Note the difference between [Hamiltonian Cycle](https://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/) and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.   
For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.  
The problem is a famous NP-hard problem. There is no polynomial-time known solution for this problem. 



**Examples:**

Output of Given Graph:

minimum weight Hamiltonian Cycle :

10 + 25 + 30 + 15 := 80

## Solution:

**Naive Solution:**

In this post, the implementation of a simple solution is discussed.

1. Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
2. Generate all (n-1)! permutations of cities.
3. Calculate the cost of every permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.

Below is the implementation of the above idea

// CPP program to implement traveling salesman

// problem using naive approach.

#include <bits/stdc++.h>

using namespace std;

#define V 4

// implementation of traveling Salesman Problem

int travllingSalesmanProblem(int graph[][V], int s)

{

// store all vertex apart from source vertex

vector<int> vertex;

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

if (i != s)

vertex.push\_back(i);

// store minimum weight Hamiltonian Cycle.

int min\_path = INT\_MAX;

do {

// store current Path weight(cost)

int current\_pathweight = 0;

// compute current path weight

int k = s;

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

current\_pathweight += graph[k][vertex[i]];

k = vertex[i];

}

current\_pathweight += graph[k][s];

// update minimum

min\_path = min(min\_path, current\_pathweight);

} while (

next\_permutation(vertex.begin(), vertex.end()));

return min\_path;

}

// Driver Code

int main()

{

// matrix representation of graph

int graph[][V] = { { 0, 10, 15, 20 },

{ 10, 0, 35, 25 },

{ 15, 35, 0, 30 },

{ 20, 25, 30, 0 } };

int s = 0;

cout << travllingSalesmanProblem(graph, s) << endl;

return 0;

}

**Output**

80

**Dynamic Programming:**  
Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far. Now the question is how to get cost(i)?  
To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i*.  
We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

If size of S is 2, then S must be {1, i},

C(S, i) = dist(1, i)

Else if size of S is greater than 2.

C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don’t have nth in them.  
Using the above recurrence relation, we can write dynamic programming based solution. There are at most O(n\*2n) subproblems, and each one takes linear time to solve. The total running time is therefore O(n2\*2n). The time complexity is much less than O(n!), but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

We will soon be discussing approximate algorithms for travelling salesman problem.

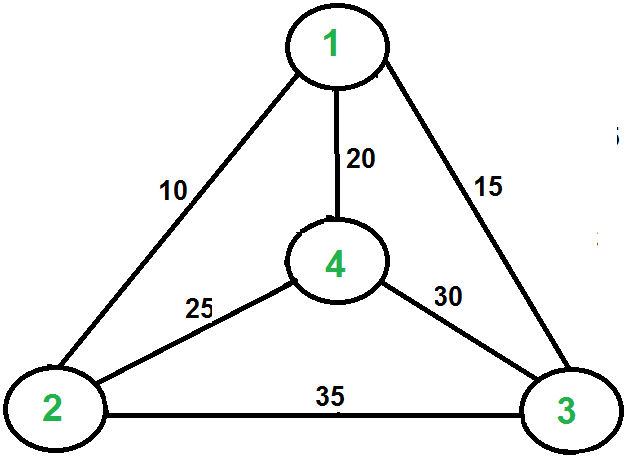
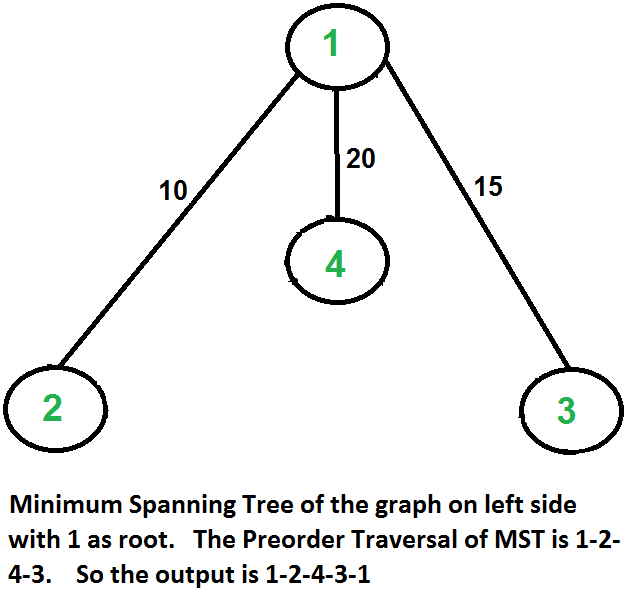
**Approximate using MST**

We introduced [Travelling Salesman Problem](https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/) and discussed Naive and Dynamic Programming Solutions for the problem in the [previous post](https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/),. Both of the solutions are infeasible. In fact, there is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There are approximate algorithms to solve the problem though. The approximate algorithms work only if the problem instance satisfies Triangle-Inequality.

**Triangle-Inequality:** The least distant path to reach a vertex j from i is always to reach j directly from i, rather than through some other vertex k (or vertices), i.e., dis(i, j) is always less than or equal to dis(i, k) + dist(k, j). The Triangle-Inequality holds in many practical situations.  
When the cost function satisfies the triangle inequality, we can design an approximate algorithm for TSP that returns a tour whose cost is never more than twice the cost of an optimal tour. The idea is to use **M**inimum **S**panning **T**ree (MST). Following is the MST based algorithm.

**Algorithm:**  
**1)** Let 1 be the starting and ending point for salesman.  
**2)** Construct MST from with 1 as root using [Prim’s Algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/).  
**3)** List vertices visited in preorder walk of the constructed MST and add 1 at the end.

Let us consider the following example. The first diagram is the given graph. The second diagram shows MST constructed with 1 as root. The preorder traversal of MST is 1-2-4-3. Adding 1 at the end gives 1-2-4-3-1 which is the output of this algorithm.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/Euler12.png)[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/MST_TSP.png)  
  
  
  
  
  
  
  
  
  
In this case, the approximate algorithm produces the optimal tour, but it may not produce optimal tour in all cases.

**How is this algorithm 2-approximate?** The cost of the output produced by the above algorithm is never more than twice the cost of best possible output. Let us see how is this guaranteed by the above algorithm.  
Let us define a term ***full walk*** to understand this. A full walk is lists all vertices when they are first visited in preorder, it also list vertices when they are returned after a subtree is visited in preorder. The full walk of above tree would be 1-2-1-4-1-3-1.  
Following are some important facts that prove the 2-approximateness.  
**1)**The cost of best possible Travelling Salesman tour is never less than the cost of MST. (The definition of [MST](http://en.wikipedia.org/wiki/Minimum_spanning_tree)says, it is a minimum cost tree that connects all vertices).  
**2)** The total cost of full walk is at most twice the cost of MST (Every edge of MST is visited at-most twice)  
**3)** The output of the above algorithm is less than the cost of full walk. In above algorithm, we print preorder walk as output. In preorder walk, two or more edges of full walk are replaced with a single edge. For example, 2-1 and 1-4 are replaced by 1 edge 2-4. So if the graph follows triangle inequality, then this is always true.

From the above three statements, we can conclude that the cost of output produced by the approximate algorithm is never more than twice the cost of best possible solution.

We have discussed a very simple 2-approximate algorithm for the travelling salesman problem. There are other better approximate algorithms for the problem. For example [Christofides algorithm](http://en.wikipedia.org/wiki/Christofides_algorithm) is 1.5 approximate algorithm. We will soon be discussing these algorithms as separate posts.

**BackTracking based solution:**

**Approach:** In this post, implementation of simple solution is discussed. 

* Consider city 1 (let say 0th node) as the starting and ending point. Since route is cyclic, we can consider any point as starting point.
* Start traversing from the source to its adjacent nodes in dfs manner.
* Calculate cost of every traversal and keep track of minimum cost and keep on updating the value of minimum cost stored value.
* Return the permutation with minimum cost.

Below is the implementation of the above approach:

// C++ implementation of the approach

#include <bits/stdc++.h>

using namespace std;

#define V 4

// Function to find the minimum weight Hamiltonian Cycle

void tsp(int graph[][V], vector<bool>& v, int currPos,

int n, int count, int cost, int& ans)

{

// If last node is reached and it has a link

// to the starting node i.e the source then

// keep the minimum value out of the total cost

// of traversal and "ans"

// Finally return to check for more possible values

if (count == n && graph[currPos][0]) {

ans = min(ans, cost + graph[currPos][0]);

return;

}

// BACKTRACKING STEP

// Loop to traverse the adjacency list

// of currPos node and increasing the count

// by 1 and cost by graph[currPos][i] value

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

if (!v[i] && graph[currPos][i]) {

// Mark as visited

v[i] = true;

tsp(graph, v, i, n, count + 1,

cost + graph[currPos][i], ans);

// Mark ith node as unvisited

v[i] = false;

}

}

};

// Driver code

int main()

{

// n is the number of nodes i.e. V

int n = 4;

int graph[][V] = {

{ 0, 10, 15, 20 },

{ 10, 0, 35, 25 },

{ 15, 35, 0, 30 },

{ 20, 25, 30, 0 }

};

// Boolean array to check if a node

// has been visited or not

vector<bool> v(n);

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

v[i] = false;

// Mark 0th node as visited

v[0] = true;

int ans = INT\_MAX;

// Find the minimum weight Hamiltonian Cycle

tsp(graph, v, 0, n, 1, 0, ans);

// ans is the minimum weight Hamiltonian Cycle

cout << ans;

return 0;

}

**Output:**

80

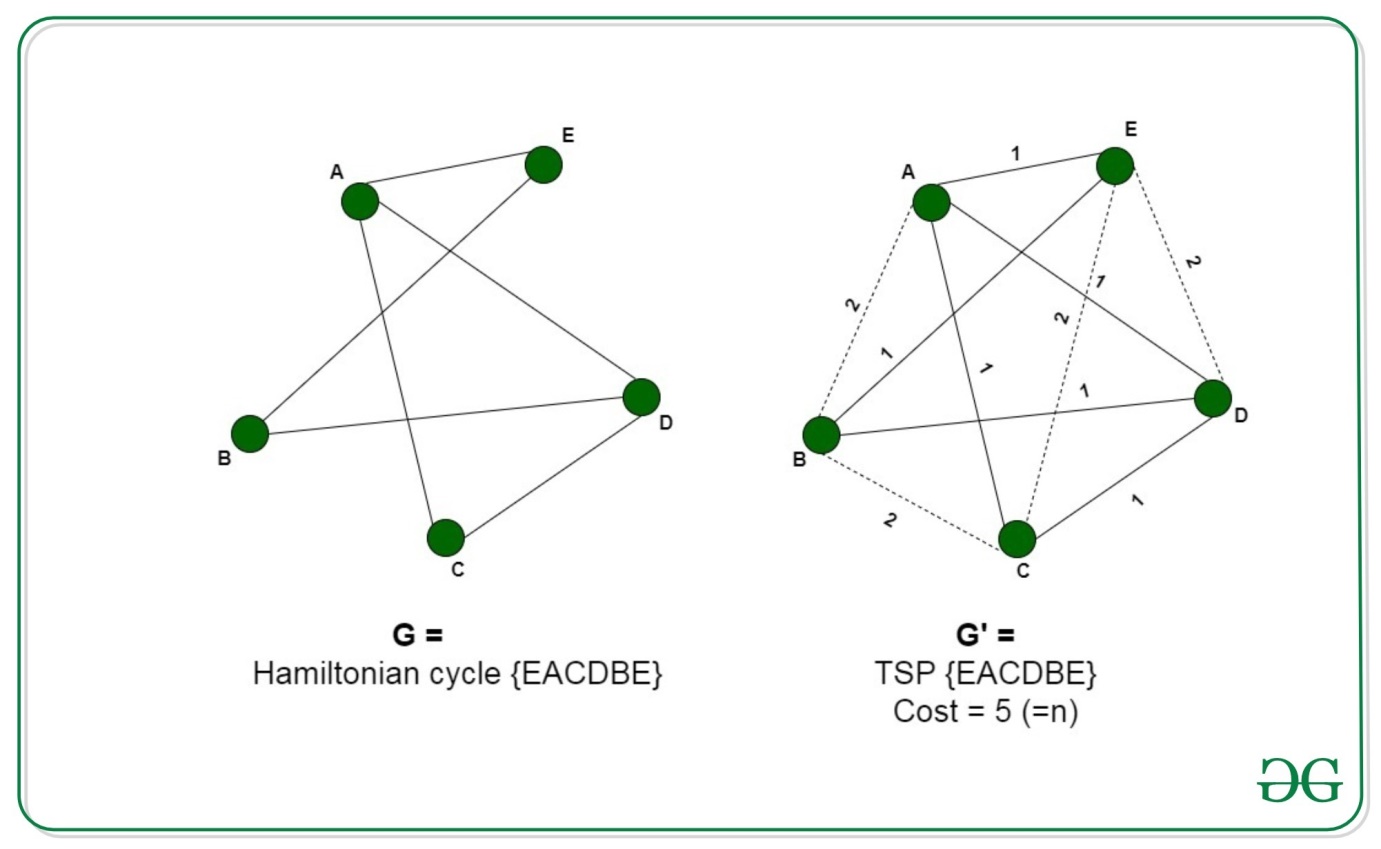
**Time Complexity:** O(N!), As for the first node there are N possibilities and for the second node there are n – 1 possibilities.  
For N nodes time complexity = N \* (N – 1) \* . . . 1 = O(N!)  
**Auxiliary Space:**O(N)

**Proof that traveling salesman problem is NP Hard**

Given a set of cities and the distance between each pair of cities, the [travelling salesman problem](https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/) finds the path between these cities such that it is the shortest path and traverses every city once, returning back to the starting point.

**Problem –** Given a graph **G(V, E)**, the problem is to determine if the graph has a TSP consisting of cost at most **K**.  
**Explanation –**  
In order to prove the Travelling Salesman Problem is NP-Hard, we will have to reduce a known NP-Hard problem to this problem. We will carry out a reduction from the Hamiltonian Cycle problem to the Travelling Salesman problem.  
Every instance of the Hamiltonian Cycle problem consists of a graph G =(V, E) as the input can be converted to a Travelling Salesman problem consisting of graph G’ = (V’, E’) and the maximum cost, K. We will construct the graph G’ in the following way:  
For all the edges e belonging to E, add the cost of edge c(e)=1. Connect the remaining edges, e’ belonging to E’, that are not present in the original graph **G**, each with a cost c(e’)= 2.  
And, set .  
The new graph G’ can be constructed in polynomial time by just converting G to a complete graph G’ and adding corresponding costs. This reduction can be proved by the following two claims:

* Let us assume that the graph **G** contains a Hamiltonian Cycle, traversing all the vertices V of the graph. Now, these vertices form a TSP with  Since it uses all the edges of the original graph having cost c(e)=1. And, since it is a cycle, therefore, it returns back to the original vertex.
* We assume that the graph G’ contains a TSP with cost, . The TSP traverses all the vertices of the graph returning to the original vertex. Now since none of the vertices are excluded from the graph and the cost sums to n, therefore, necessarily it uses all the edges of the graph present in **E**, with cost 1, hence forming a [hamiltonian cycle](https://www.geeksforgeeks.org/hamiltonian-cycle-backtracking-6/) with the graph **G**.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200601165759/tsp1.jpg)

Thus we can say that the graph **G’** contains a TSP if graph **G** contains Hamiltonian Cycle. Therefore, any instance of the Travelling salesman problem can be reduced to an instance of the hamiltonian cycle problem. Thus, the TSP is NP-Hard.

# 352. Graph Colouring Problem

[Graph coloring](http://en.wikipedia.org/wiki/Graph_coloring) problem is to assign colors to certain elements of a graph subject to certain constraints.

**Vertex coloring** is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. The other graph coloring problems like ***Edge Coloring*** (No vertex is incident to two edges of same color) and ***Face Coloring***(Geographical Map Coloring) can be transformed into vertex coloring.

**Chromatic Number:** The smallest number of colors needed to color a graph G is called its chromatic number. For example, the following can be colored minimum 2 colors. 

vertex_coloring

The problem to find chromatic number of a given graph is [NP Complete](https://www.geeksforgeeks.org/np-completeness-set-1/).

**Applications of Graph Coloring:**

The graph coloring problem has huge number of applications.

***1) Making Schedule or Time Table:***Suppose we want to make am exam schedule for a university. We have list different subjects and students enrolled in every subject. Many subjects would have common students (of same batch, some backlog students, etc). *How do we schedule the exam so that no two exams with a common student are scheduled at same time? How many minimum time slots are needed to schedule all exams?* This problem can be represented as a graph where every vertex is a subject and an edge between two vertices mean there is a common student. So this is a graph coloring problem where minimum number of time slots is equal to the chromatic number of the graph.

***2)***[***Mobile Radio Frequency Assignment***](http://www.zib.de/groetschel/teaching/SS2012/GraphCol%20and%20FrequAssignment.pdf)***:*** When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. How to assign frequencies with this constraint? What is the minimum number of frequencies needed? This problem is also an instance of graph coloring problem where every tower represents a vertex and an edge between two towers represents that they are in range of each other.

***3) Sudoku:***Sudoku is also a variation of Graph coloring problem where every cell represents a vertex. There is an edge between two vertices if they are in same row or same column or same block.

***4)***[***Register Allocation***](http://en.wikipedia.org/wiki/Register_allocation)***:***In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. This problem is also a graph coloring problem.

***5) Bipartite Graphs:***We can check if a graph is Bipartite or not by coloring the graph using two colors. If a given graph is 2-colorable, then it is Bipartite, otherwise not. See [this](https://www.geeksforgeeks.org/bipartite-graph/)for more details.

***6) Map Coloring:***Geographical maps of countries or states where no two adjacent cities cannot be assigned same color. Four colors are sufficient to color any map (See [Four Color Theorem](http://en.wikipedia.org/wiki/Four_color_theorem))

**There can be many more applications:** For example the below reference video lecture has a case study at 1:18.   
[Akamai](http://en.wikipedia.org/wiki/Akamai_Technologies)runs a network of thousands of servers and the servers are used to distribute content on Internet. They install a new software or update existing softwares pretty much every week. The update cannot be deployed on every server at the same time, because the server may have to be taken down for the install. Also, the update should not be done one at a time, because it will take a lot of time. There are sets of servers that cannot be taken down together, because they have certain critical functions. This is a typical scheduling application of graph coloring problem. It turned out that 8 colors were good enough to color the graph of 75000 nodes. So they could install updates in 8 passes.

We introduced [graph coloring and applications](http://www.geeksforgeeks.org/graph-coloring-applications/) in previous post. As discussed in the previous post, graph coloring is widely used. Unfortunately, there is no efficient algorithm available for coloring a graph with minimum number of colors as the problem is a known [NP Complete problem](http://www.geeksforgeeks.org/np-completeness-set-1/). There are approximate algorithms to solve the problem though. Following is the basic Greedy Algorithm to assign colors. It doesn’t guarantee to use minimum colors, but it guarantees an upper bound on the number of colors. The basic algorithm never uses more than d+1 colors where d is the maximum degree of a vertex in the given graph.

**Basic Greedy Coloring Algorithm:**

***1.****Color first vertex with first color.****2.****Do following for remaining V-1 vertices.   
…..****a)****Consider the currently picked vertex and color it with the   
lowest numbered color that has not been used on any previously   
colored vertices adjacent to it. If all previously used colors   
appear on vertices adjacent to v, assign a new color to it.*

Following is the implementation of the above Greedy Algorithm.

// A C++ program to implement greedy algorithm for graph coloring

#include <iostream>

#include <list>

using namespace std;

// A class that represents an undirected graph

class Graph

{

int V; // No. of vertices

list<int> \*adj; // A dynamic array of adjacency lists

public:

// Constructor and destructor

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

~Graph() { delete [] adj; }

// function to add an edge to graph

void addEdge(int v, int w);

// Prints greedy coloring of the vertices

void greedyColoring();

};

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w);

adj[w].push\_back(v); // Note: the graph is undirected

}

// Assigns colors (starting from 0) to all vertices and prints

// the assignment of colors

void Graph::greedyColoring()

{

int result[V];

// Assign the first color to first vertex

result[0] = 0;

// Initialize remaining V-1 vertices as unassigned

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

result[u] = -1; // no color is assigned to u

// A temporary array to store the available colors. True

// value of available[cr] would mean that the color cr is

// assigned to one of its adjacent vertices

bool available[V];

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

available[cr] = false;

// Assign colors to remaining V-1 vertices

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

{

// Process all adjacent vertices and flag their colors

// as unavailable

list<int>::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

if (result[\*i] != -1)

available[result[\*i]] = true;

// Find the first available color

int cr;

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

if (available[cr] == false)

break;

result[u] = cr; // Assign the found color

// Reset the values back to false for the next iteration

for (i = adj[u].begin(); i != adj[u].end(); ++i)

if (result[\*i] != -1)

available[result[\*i]] = false;

}

// print the result

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

cout << "Vertex " << u << " ---> Color "

<< result[u] << endl;

}

// Driver program to test above function

int main()

{

Graph g1(5);

g1.addEdge(0, 1);

g1.addEdge(0, 2);

g1.addEdge(1, 2);

g1.addEdge(1, 3);

g1.addEdge(2, 3);

g1.addEdge(3, 4);

cout << "Coloring of graph 1 \n";

g1.greedyColoring();

Graph g2(5);

g2.addEdge(0, 1);

g2.addEdge(0, 2);

g2.addEdge(1, 2);

g2.addEdge(1, 4);

g2.addEdge(2, 4);

g2.addEdge(4, 3);

cout << "\nColoring of graph 2 \n";

g2.greedyColoring();

return 0;

}

**Output:**

Coloring of graph 1

Vertex 0 ---> Color 0

Vertex 1 ---> Color 1

Vertex 2 ---> Color 2

Vertex 3 ---> Color 0

Vertex 4 ---> Color 1

Coloring of graph 2

Vertex 0 ---> Color 0

Vertex 1 ---> Color 1

Vertex 2 ---> Color 2

Vertex 3 ---> Color 0

Vertex 4 ---> Color 3

Time Complexity: O(V^2 + E) in worst case.

**Analysis of Basic Algorithm**   
The above algorithm doesn’t always use minimum number of colors. Also, the number of colors used sometime depend on the order in which vertices are processed. For example, consider the following two graphs. Note that in graph on right side, vertices 3 and 4 are swapped. If we consider the vertices 0, 1, 2, 3, 4 in left graph, we can color the graph using 3 colors. But if we consider the vertices 0, 1, 2, 3, 4 in right graph, we need 4 colors. 

graph_coloring2

So the order in which the vertices are picked is important. Many people have suggested different ways to find an ordering that work better than the basic algorithm on average. The most common is [Welsh–Powell Algorithm](http://mrsleblancsmath.pbworks.com/w/file/fetch/46119304/vertex%20coloring%20algorithm.pdf) which considers vertices in descending order of degrees.

**How does the basic algorithm guarantee an upper bound of d+1?**   
Here d is the maximum degree in the given graph. Since d is maximum degree, a vertex cannot be attached to more than d vertices. When we color a vertex, at most d colors could have already been used by its adjacent. To color this vertex, we need to pick the smallest numbered color that is not used by the adjacent vertices. If colors are numbered like 1, 2, …., then the value of such smallest number must be between 1 to d+1 (Note that d numbers are already picked by adjacent vertices).   
This can also be proved using induction. See [this](http://www.youtube.com/watch?v=h9wxtqoa1jY)video lecture for proof.   
We will soon be discussing some interesting facts about chromatic number and graph coloring.

# 353. Snake and Ladders Problem

You are given an n x n integer matrix board where the cells are labeled from 1 to n2 in a [**Boustrophedon style**](https://en.wikipedia.org/wiki/Boustrophedon) starting from the bottom left of the board (i.e. board[n - 1][0]) and alternating direction each row.

You start on square 1 of the board. In each move, starting from square curr, do the following:

* Choose a destination square next with a label in the range [curr + 1, min(curr + 6, n2)].
  + This choice simulates the result of a standard **6-sided die roll**: i.e., there are always at most 6 destinations, regardless of the size of the board.
* If next has a snake or ladder, you **must** move to the destination of that snake or ladder. Otherwise, you move to next.
* The game ends when you reach the square n2.

A board square on row r and column c has a snake or ladder if board[r][c] != -1. The destination of that snake or ladder is board[r][c]. Squares 1 and n2 do not have a snake or ladder.

Note that you only take a snake or ladder at most once per move. If the destination to a snake or ladder is the start of another snake or ladder, you do **not** follow the subsequent snake or ladder.

* For example, suppose the board is [[-1,4],[-1,3]], and on the first move, your destination square is 2. You follow the ladder to square 3, but do **not** follow the subsequent ladder to 4.

Return *the least number of moves required to reach the square*n2*. If it is not possible to reach the square, return*-1.

**Example 1:**



**Input:** board = [[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,35,-1,-1,13,-1],[-1,-1,-1,-1,-1,-1],[-1,15,-1,-1,-1,-1]]

**Output:** 4

**Explanation:**

In the beginning, you start at square 1 (at row 5, column 0).

You decide to move to square 2 and must take the ladder to square 15.

You then decide to move to square 17 and must take the snake to square 13.

You then decide to move to square 14 and must take the ladder to square 35.

You then decide to move to square 36, ending the game.

This is the lowest possible number of moves to reach the last square, so return 4.

**Example 2:**

**Input:** board = [[-1,-1],[-1,3]]

**Output:** 1

**Constraints:**

* n == board.length == board[i].length
* 2 <= n <= 20
* grid[i][j] is either -1 or in the range [1, n2].
* The squares labeled 1 and n2 do not have any ladders or snakes.

## Solution:

Use BFS:

class Solution {

public:

int snakesAndLadders(vector<vector<int>>& board) {

vector<int> p(1);

for (int i=board.size()-1, flip=0; i>=0; i--, flip=!flip) {

if (flip) {

p.insert(p.end(), board[i].rbegin(), board[i].rend());

} else {

p.insert(p.end(), board[i].begin(), board[i].end());

}

}

vector<bool> vis(p.size(), false);

queue<int> q;

q.push(1); vis[1] = true;

int level = 0;

while (!q.empty()) {

for (int sz=q.size(); sz>0; sz--) {

int front = q.front(); q.pop();

if (front == p.size() - 1) return level;

for (int i=1; i<=6; i++) {

int nc = i + front ;

if (nc < p.size() && !vis[nc]) {

int next = p[nc] == -1 ? nc: p[nc];

q.push(next); vis[nc] = true;

}

}

}

level++;

}

return -1;

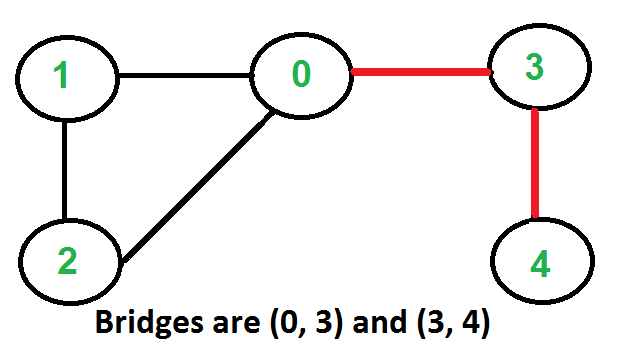
}

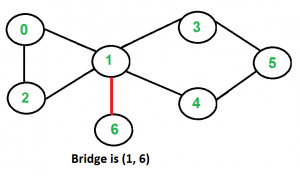
};

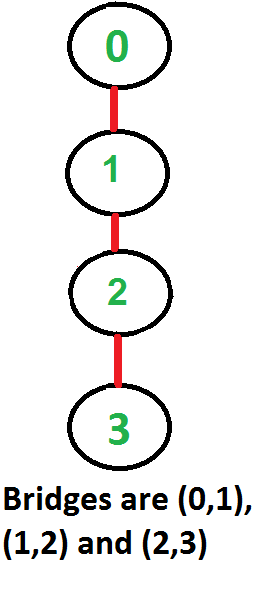
# 354. Find bridge in a graph

An edge in an undirected connected graph is a bridge if removing it disconnects the graph. For a disconnected undirected graph, definition is similar, a bridge is an edge removing which increases number of disconnected components.   
Like [Articulation Points](https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/), bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. For example, in a wired computer network, an articulation point indicates the critical computers and a bridge indicates the critical wires or connections.

Following are some example graphs with bridges highlighted with red color.







**How to find all bridges in a given graph?**   
A simple approach is to one by one remove all edges and see if removal of an edge causes disconnected graph. Following are steps of simple approach for connected graph.  
1) For every edge (u, v), do following   
…..a) Remove (u, v) from graph   
..…b) See if the graph remains connected (We can either use BFS or DFS)   
…..c) Add (u, v) back to the graph.  
Time complexity of above method is O(E\*(V+E)) for a graph represented using adjacency list. Can we do better?

**A O(V+E) algorithm to find all Bridges**   
The idea is similar to [O(V+E) algorithm for Articulation Points](https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/). We do DFS traversal of the given graph. In DFS tree an edge (u, v) (u is parent of v in DFS tree) is bridge if there does not exist any other alternative to reach u or an ancestor of u from subtree rooted with v. As discussed in the [previous post](https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/), the value low[v] indicates earliest visited vertex reachable from subtree rooted with v. *The condition for an edge (u, v) to be a bridge is, “low[v] > disc[u]”*.

Following are the implementations of above approach.

// A C++ program to find bridges in a given undirected graph

#include<iostream>

#include <list>

#define NIL -1

using namespace std;

// A class that represents an undirected graph

class Graph

{

int V; // No. of vertices

list<int> \*adj; // A dynamic array of adjacency lists

void bridgeUtil(int v, bool visited[], int disc[], int low[],

int parent[]);

public:

Graph(int V); // Constructor

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

void bridge(); // prints all bridges

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w);

adj[w].push\_back(v); // Note: the graph is undirected

}

// A recursive function that finds and prints bridges using

// DFS traversal

// u --> The vertex to be visited next

// visited[] --> keeps track of visited vertices

// disc[] --> Stores discovery times of visited vertices

// parent[] --> Stores parent vertices in DFS tree

void Graph::bridgeUtil(int u, bool visited[], int disc[],

int low[], int parent[])

{

// A static variable is used for simplicity, we can

// avoid use of static variable by passing a pointer.

static int time = 0;

// Mark the current node as visited

visited[u] = true;

// Initialize discovery time and low value

disc[u] = low[u] = ++time;

// Go through all vertices adjacent to this

list<int>::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

{

int v = \*i; // v is current adjacent of u

// If v is not visited yet, then recur for it

if (!visited[v])

{

parent[v] = u;

bridgeUtil(v, visited, disc, low, parent);

// Check if the subtree rooted with v has a

// connection to one of the ancestors of u

low[u] = min(low[u], low[v]);

// If the lowest vertex reachable from subtree

// under v is below u in DFS tree, then u-v

// is a bridge

if (low[v] > disc[u])

cout << u <<" " << v << endl;

}

// Update low value of u for parent function calls.

else if (v != parent[u])

low[u] = min(low[u], disc[v]);

}

}

// DFS based function to find all bridges. It uses recursive

// function bridgeUtil()

void Graph::bridge()

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

int \*disc = new int[V];

int \*low = new int[V];

int \*parent = new int[V];

// Initialize parent and visited arrays

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

{

parent[i] = NIL;

visited[i] = false;

}

// Call the recursive helper function to find Bridges

// in DFS tree rooted with vertex 'i'

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

if (visited[i] == false)

bridgeUtil(i, visited, disc, low, parent);

}

// Driver program to test above function

int main()

{

// Create graphs given in above diagrams

cout << "\nBridges in first graph \n";

Graph g1(5);

g1.addEdge(1, 0);

g1.addEdge(0, 2);

g1.addEdge(2, 1);

g1.addEdge(0, 3);

g1.addEdge(3, 4);

g1.bridge();

cout << "\nBridges in second graph \n";

Graph g2(4);

g2.addEdge(0, 1);

g2.addEdge(1, 2);

g2.addEdge(2, 3);

g2.bridge();

cout << "\nBridges in third graph \n";

Graph g3(7);

g3.addEdge(0, 1);

g3.addEdge(1, 2);

g3.addEdge(2, 0);

g3.addEdge(1, 3);

g3.addEdge(1, 4);

g3.addEdge(1, 6);

g3.addEdge(3, 5);

g3.addEdge(4, 5);

g3.bridge();

return 0;

}

Output:

Bridges in first graph

3 4

0 3

Bridges in second graph

2 3

1 2

0 1

Bridges in third graph

1 6

**Time Complexity:** The above function is simple DFS with additional arrays. So time complexity is same as DFS which is O(V+E) for adjacency list representation of graph.

# 355. Count Strongly connected Components (Kosaraju Algo)

Given a Directed Graph with**V**vertices **(**Numbered from**0 to V-1)** and **E** edges, Find the number of strongly connected components in the graph.

**Example 1:**

**Input:**

**Output:**

3

**Explanation**:

We can clearly see that there are 3 Strongly

Connected Components in the Graph

**Example 2:**

**Input:**

**Output:**

1

**Explanation**:

All of the nodes are connected to each other.

So, there's only one SCC.

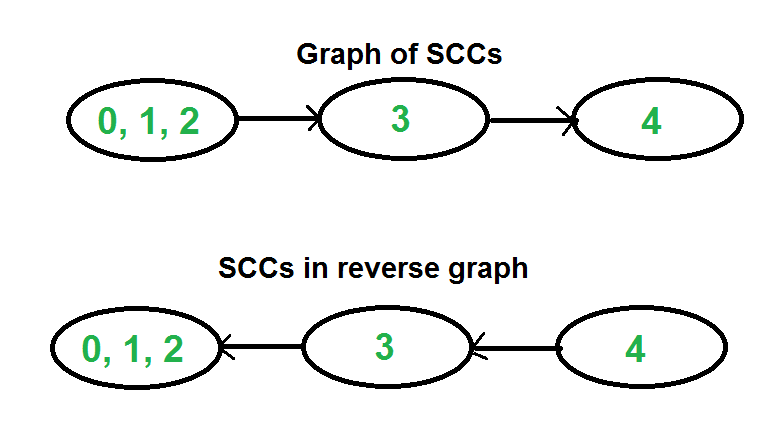
**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **kosaraju()** which takes the number of vertices V and adjacency list of the graph as inputs and returns an integer denoting the number of strongly connected components in the given graph.

**Expected Time Complexity:** O(V+E).  
**Expected Auxiliary Space:** O(V).

**Constraints:**  
1 ≤ V ≤ 5000  
0 ≤ E ≤ (V\*(V-1))  
0 ≤ u, v ≤ N-1  
Sum of E over all testcases will not exceed 25\*106

## Solution:

We can find all strongly connected components in O(V+E) time using [Kosaraju’s algorithm](http://en.wikipedia.org/wiki/Kosaraju%27s_algorithm). Following is detailed Kosaraju’s algorithm.  
**1)** Create an empty stack ‘S’ and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.  
**2)** Reverse directions of all arcs to obtain the transpose graph.  
**3)** One by one pop a vertex from S while S is not empty. Let the popped vertex be ‘v’. Take v as source and do DFS (call [DFSUtil(v)](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

**How does this work?**  
The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start from 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sink in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way for getting this sequence. However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other that its own SCC), will always be greater than finish time of vertices in the other SCC (See [this](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/strongComponent.htm)for proof). For example, in DFS of above example graph, finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And finish time of 3 is always greater than 4. DFS doesn’t guarantee about other vertices, for example finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appear after both 3 and 4.  
In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC {0, 1, 2} becomes sink and the SCC {4} becomes source. As discussed above, in stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in stack, we process vertices from sink to source (in reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.  
[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/SCCGraph2.png)

Following is C++ implementation of Kosaraju’s algorithm.

// C++ Implementation of Kosaraju's algorithm to print all SCCs

#include <iostream>

#include <list>

#include <stack>

using namespace std;

class Graph

{

int V; // No. of vertices

list<int> \*adj; // An array of adjacency lists

// Fills Stack with vertices (in increasing order of finishing

// times). The top element of stack has the maximum finishing

// time

void fillOrder(int v, bool visited[], stack<int> &Stack);

// A recursive function to print DFS starting from v

void DFSUtil(int v, bool visited[]);

public:

Graph(int V);

void addEdge(int v, int w);

// The main function that finds and prints strongly connected

// components

void printSCCs();

// Function that returns reverse (or transpose) of this graph

Graph getTranspose();

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

// A recursive function to print DFS starting from v

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

{

// Mark the current node as visited and print it

visited[v] = true;

cout << v << " ";

// 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::getTranspose()

{

Graph g(V);

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

{

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for(i = adj[v].begin(); i != adj[v].end(); ++i)

{

g.adj[\*i].push\_back(v);

}

}

return g;

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v’s list.

}

void Graph::fillOrder(int v, bool visited[], stack<int> &Stack)

{

// Mark the current node as visited and print it

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])

fillOrder(\*i, visited, Stack);

// All vertices reachable from v are processed by now, push v

Stack.push(v);

}

// The main function that finds and prints all strongly connected

// components

void Graph::printSCCs()

{

stack<int> Stack;

// Mark all the vertices as not visited (For first DFS)

bool \*visited = new bool[V];

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

visited[i] = false;

// Fill vertices in stack according to their finishing times

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

if(visited[i] == false)

fillOrder(i, visited, Stack);

// Create a reversed graph

Graph gr = getTranspose();

// Mark all the vertices as not visited (For second DFS)

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

visited[i] = false;

// Now process all vertices in order defined by Stack

while (Stack.empty() == false)

{

// Pop a vertex from stack

int v = Stack.top();

Stack.pop();

// Print Strongly connected component of the popped vertex

if (visited[v] == false)

{

gr.DFSUtil(v, visited);

cout << endl;

}

}

}

// Driver program to test above functions

int main()

{

// Create a graph given in the above diagram

Graph g(5);

g.addEdge(1, 0);

g.addEdge(0, 2);

g.addEdge(2, 1);

g.addEdge(0, 3);

g.addEdge(3, 4);

cout << "Following are strongly connected components in "

"given graph \n";

g.printSCCs();

return 0;

}

Output:

Following are strongly connected components in given graph

0 1 2

3

4

**Time Complexity:** The above algorithm calls DFS, finds reverse of the graph and again calls DFS. DFS takes O(V+E) for a graph represented using adjacency list. Reversing a graph also takes O(V+E) time. For reversing the graph, we simple traverse all adjacency lists.

The above algorithm is asymptotically best algorithm, but there are other algorithms like [Tarjan’s algorithm](http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm) and [path-based](http://en.wikipedia.org/wiki/Path-based_strong_component_algorithm) which have same time complexity but find SCCs using single DFS. The Tarjan’s algorithm is discussed in the following post.

[Tarjan’s Algorithm to find Strongly Connected Components](https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/)

**Applications:**  
SCC algorithms can be used as a first step in many graph algorithms that work only on strongly connected graph.  
In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages or play common games. The SCC algorithms can be used to find such groups and suggest the commonly liked pages or games to the people in the group who have not yet liked commonly liked a page or played a game.

# 356. Check whether a graph is Bipartite or Not

Given an adjacency list of a graph**adj**of V no. of vertices having 0 based index. Check whether the graph is bipartite or not.

**Example 1:**

**Input:**

**Output:** 1

**Explanation:** The given graph can be colored

in two colors so, it is a bipartite graph.

**Example 2:**

**Input:**

**Output:** 0

**Explanation:** The given graph cannot be colored

in two colors such that color of adjacent

vertices differs.

**Your Task:**  
You don't need to read or print anything. Your task is to complete the function **isBipartite()**which takes V denoting no. of vertices and adj denoting adjacency list of the graph and returns a boolean value true if the graph is bipartite otherwise returns false.

**Expected Time Complexity:**O(V + E)  
**Expected Space Complexity:**O(V)  
  
**Constraints:**  
1 ≤ V, E ≤ 105

## Solution:

**My Implementation using BFS:**

bool bfs(int V, vector<int> adj[], int node, vector<int> &color){

queue<int> q;

q.push(node);

int curr = 0;

while(!q.empty()){

int sz = q.size();

curr = !curr;

while(sz--){

int i = q.front(); q.pop();

for(int j=0;j<adj[i].size();j++){

if(color[adj[i][j]]==-1){

color[adj[i][j]] = curr;

q.push(adj[i][j]);

}

if(color[adj[i][j]]!=curr)

return false;

}

}

}

return true;

}

bool isBipartite(int V, vector<int>adj[]){

// Code here

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

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

if(color[i]==-1){

color[i] = 0;

if(bfs(V, adj, i, color)==false) return false;

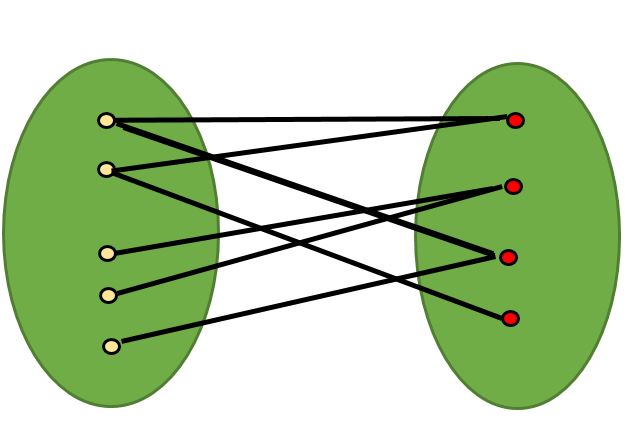
}

}

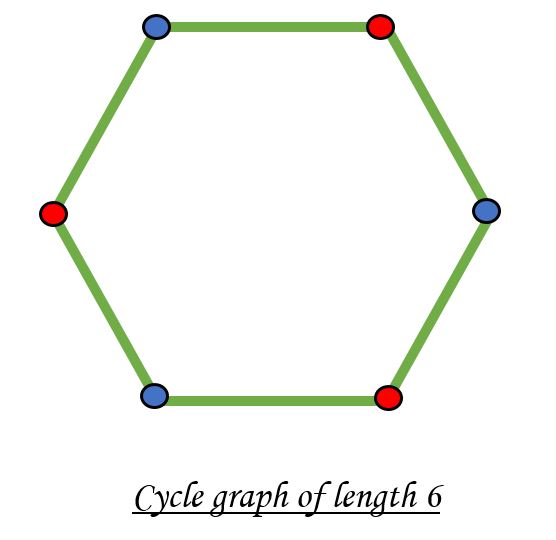
return true;

}

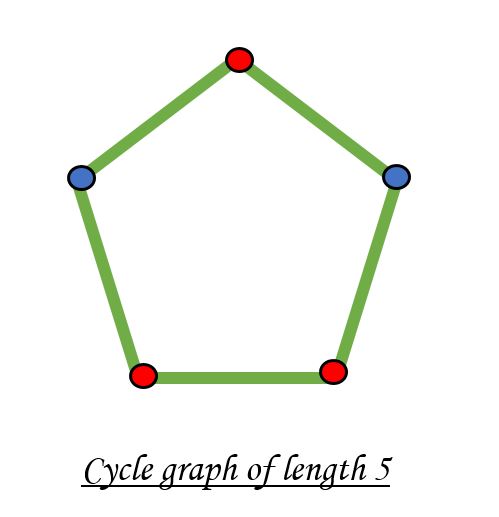
A [Bipartite Graph](http://en.wikipedia.org/wiki/Bipartite_graph) is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.



A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.



It is not possible to color a cycle graph with odd cycle using two colors.



*Algorithm to check if a graph is Bipartite:*   
One approach is to check whether the graph is 2-colorable or not using [backtracking algorithm m coloring problem](https://www.geeksforgeeks.org/backttracking-set-5-m-coloring-problem/). 

Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).   
1. Assign RED color to the source vertex (putting into set U).   
2. Color all the neighbors with BLUE color (putting into set V).   
3. Color all neighbor’s neighbor with RED color (putting into set U).   
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.   
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

// C++ program to find out whether a

// given graph is Bipartite or not

#include <iostream>

#include <queue>

#define V 4

using namespace std;

// This function returns true if graph

// G[V][V] is Bipartite, else false

bool isBipartite(int G[][V], int src)

{

// Create a color array to store colors

// assigned to all vertices. Vertex

// number is used as index in this array.

// The value '-1' of colorArr[i]

// is used to indicate that no color

// is assigned to vertex 'i'. The value 1

// is used to indicate first color

// is assigned and value 0 indicates

// second color is assigned.

int colorArr[V];

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

colorArr[i] = -1;

// Assign first color to source

colorArr[src] = 1;

// Create a queue (FIFO) of vertex

// numbers and enqueue source vertex

// for BFS traversal

queue <int> q;

q.push(src);

// Run while there are vertices

// in queue (Similar to BFS)

while (!q.empty())

{

// Dequeue a vertex from queue ( Refer http://goo.gl/35oz8 )

int u = q.front();

q.pop();

// Return false if there is a self-loop

if (G[u][u] == 1)

return false;

// Find all non-colored adjacent vertices

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

{

// An edge from u to v exists and

// destination v is not colored

if (G[u][v] && colorArr[v] == -1)

{

// Assign alternate color to this adjacent v of u

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

q.push(v);

}

// An edge from u to v exists and destination

// v is colored with same color as u

else if (G[u][v] && colorArr[v] == colorArr[u])

return false;

}

}

// If we reach here, then all adjacent

// vertices can be colored with alternate color

return true;

}

// Driver program to test above function

int main()

{

int G[][V] = {{0, 1, 0, 1},

{1, 0, 1, 0},

{0, 1, 0, 1},

{1, 0, 1, 0}

};

isBipartite(G, 0) ? cout << "Yes" : cout << "No";

return 0;

}

**Output:**

Yes

The above algorithm works only if the graph is connected. In above code, we always start with source 0 and assume that vertices are visited from it. One important observation is a graph with no edges is also Bipartite. Note that the Bipartite condition says all edges should be from one set to another.

We can extend the above code to handle cases when a graph is not connected. The idea is repeatedly called above method for all not yet visited vertices.

// C++ program to find out whether

// a given graph is Bipartite or not.

// It works for disconnected graph also.

#include <bits/stdc++.h>

using namespace std;

const int V = 4;

// This function returns true if

// graph G[V][V] is Bipartite, else false

bool isBipartiteUtil(int G[][V], int src, int colorArr[])

{

colorArr[src] = 1;

// Create a queue (FIFO) of vertex numbers a

// nd enqueue source vertex for BFS traversal

queue<int> q;

q.push(src);

// Run while there are vertices in queue (Similar to

// BFS)

while (!q.empty()) {

// Dequeue a vertex from queue ( Refer

// http://goo.gl/35oz8 )

int u = q.front();

q.pop();

// Return false if there is a self-loop

if (G[u][u] == 1)

return false;

// Find all non-colored adjacent vertices

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

// An edge from u to v exists and

// destination v is not colored

if (G[u][v] && colorArr[v] == -1) {

// Assign alternate color to this

// adjacent v of u

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

q.push(v);

}

// An edge from u to v exists and destination

// v is colored with same color as u

else if (G[u][v] && colorArr[v] == colorArr[u])

return false;

}

}

// If we reach here, then all adjacent vertices can

// be colored with alternate color

return true;

}

// Returns true if G[][] is Bipartite, else false

bool isBipartite(int G[][V])

{

// Create a color array to store colors assigned to all

// vertices. Vertex/ number is used as index in this

// array. The value '-1' of colorArr[i] is used to

// indicate that no color is assigned to vertex 'i'.

// The value 1 is used to indicate first color is

// assigned and value 0 indicates second color is

// assigned.

int colorArr[V];

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

colorArr[i] = -1;

// This code is to handle disconnected graph

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

if (colorArr[i] == -1)

if (isBipartiteUtil(G, i, colorArr) == false)

return false;

return true;

}

// Driver code

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;

}

**Output:**

Yes

Time Complexity of the above approach is same as that Breadth First Search. In above implementation is O(V^2) where V is number of vertices. If graph is represented using adjacency list, then the complexity becomes O(V+E).

**If Graph is represented using Adjacency List** .Time Complexity will be O(V+E).

Works for connected as well as disconnected graph.

#include <bits/stdc++.h>

using namespace std;

bool isBipartite(int V, vector<int> adj[])

{

// vector to store colour of vertex

// assigning all to -1 i.e. uncoloured

// colours are either 0 or 1

// for understanding take 0 as red and 1 as blue

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

// queue for BFS storing {vertex , colour}

queue<pair<int, int> > q;

//loop incase graph is not connected

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

//if not coloured

if (col[i] == -1) {

//colouring with 0 i.e. red

q.push({ i, 0 });

col[i] = 0;

while (!q.empty()) {

pair<int, int> p = q.front();

q.pop();

//current vertex

int v = p.first;

//colour of current vertex

int c = p.second;

//traversing vertexes connected to current vertex

for (int j : adj[v]) {

//if already coloured with parent vertex color

//then bipartite graph is not possible

if (col[j] == c)

return 0;

//if uncoloured

if (col[j] == -1) {

//colouring with opposite color to that of parent

col[j] = (c) ? 0 : 1;

q.push({ j, col[j] });

}

}

}

}

}

//if all vertexes are coloured such that

//no two connected vertex have same colours

return 1;

}

// { Driver Code Starts.

int main()

{

int V, E;

V = 4 , E = 8;

//adjacency list for storing graph

vector<int> adj[V];

adj[0] = {1,3};

adj[1] = {0,2};

adj[2] = {1,3};

adj[3] = {0,2};

bool ans = isBipartite(V, adj);

//returns 1 if bipartite graph is possible

if (ans)

cout << "Yes\n";

//returns 0 if bipartite graph is not possible

else

cout << "No\n";

return 0;

}

**Output**

Yes

**Exercise:**   
**1.** Can DFS algorithm be used to check the bipartite-ness of a graph? If yes, how?   
Solution :

// 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;

}

**Output**

Yes

# 357. [Detect Negative cycle in a graph](https://www.geeksforgeeks.org/detect-negative-cycle-graph-bellman-ford/)

Given a weighted directed graph with n nodes and m edges. Nodes are labeled from 0 to n-1, the task is to check if it contains a negative weight cycle or not.  
**Note:**edges[i] is defined as u, v and weight.

**Example 1:**

**Input:** n = 3, edges = {{0,1,-1},{1,2,-2},

{2,0,-3}}

**Output:** 1

**Explanation:** The graph contains negative weight

cycle as 0->1->2->0 with weight -1,-2,-3,-1.

**Example 2:**

**Input:** n = 3, edges = {{0,1,-1},{1,2,-2},

{2,0,3}}

**Output:** 0

**Explanation:** The graph does not contain any

negative weight cycle.

**Your Task:**  
You don't need to read or print anyhting. Your task is to complete the function **isNegativeWeightCycle()**which takes n and edges as input paramater and returns 1 if graph contains negative weight cycle otherwise returns 0.

**Expected Time Complexity:**O(n\*m)  
**Expected Space Compelxity:**O(n)

**Constraints:**  
1 <= n <= 100  
1 <= m <= n\*(n-1), where m is the total number of Edges in the directed graph.

## Solution:

**My Implementation:**

int isNegativeWeightCycle(int n, vector<vector<int>>edges){

// Code here

vector<int> key(n, INT\_MAX);

key[0] = 0;

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

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

if(key[edges[j][0]]!=INT\_MAX && key[edges[j][0]]+edges[j][2] < key[edges[j][1]]){

key[edges[j][1]] = key[edges[j][0]]+edges[j][2];

}

}

}

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

if(key[edges[j][0]]!=INT\_MAX && key[edges[j][0]]+edges[j][2] < key[edges[j][1]]){

return 1;

}

}

return 0;

}

The idea is to use [Bellman-Ford Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/).

Below is an algorithm to find if there is a negative weight cycle reachable from the given source.  
**1)** Initialize distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is the source vertex.  
**2)** This step calculates the shortest distances. Do the following |V|-1 times where |V| is the number of vertices in the given graph.   
**a)** Do the following for each edge u-v.  
     **b)**If dist[v] > dist[u] + weight of edge uv, then update dist[v].   
**c)**dist[v] = dist[u] + weight of edge uv.  
**3)** This step reports if there is a negative weight cycle in the graph. Do the following for each edge u-v   
     **a)**If dist[v] > dist[u] + weight of edge uv, then the “Graph has a negative weight cycle”

The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn’t contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.

// A C++ program to check if a graph contains negative

// weight cycle using Bellman-Ford algorithm. This program

// works only if all vertices are reachable from a source

// vertex 0.

#include <bits/stdc++.h>

using namespace std;

// a structure to represent a weighted edge in graph

struct Edge {

int src, dest, weight;

};

// 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.

struct 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[graph->E];

return graph;

}

// The main function that finds shortest distances

// from src to all other vertices using Bellman-

// Ford algorithm. The function also detects

// negative weight cycle

bool isNegCycleBellmanFord(struct Graph\* graph,

int src)

{

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[src] = 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 u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

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

}

}

// 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 u = graph->edge[i].src;

int v = graph->edge[i].dest;

int weight = graph->edge[i].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

return true;

}

return false;

}

// Driver program to test above functions

int main()

{

/\* Let us create the graph given in above example \*/

int V = 5; // Number of vertices in graph

int E = 8; // Number of edges in graph

struct Graph\* graph = createGraph(V, E);

// add edge 0-1 (or A-B in above figure)

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = -1;

// add edge 0-2 (or A-C in above figure)

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 4;

// add edge 1-2 (or B-C in above figure)

graph->edge[2].src = 1;

graph->edge[2].dest = 2;

graph->edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 2;

// add edge 1-4 (or A-E in above figure)

graph->edge[4].src = 1;

graph->edge[4].dest = 4;

graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)

graph->edge[5].src = 3;

graph->edge[5].dest = 2;

graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)

graph->edge[6].src = 3;

graph->edge[6].dest = 1;

graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)

graph->edge[7].src = 4;

graph->edge[7].dest = 3;

graph->edge[7].weight = -3;

if (isNegCycleBellmanFord(graph, 0))

cout << "Yes";

else

cout << "No";

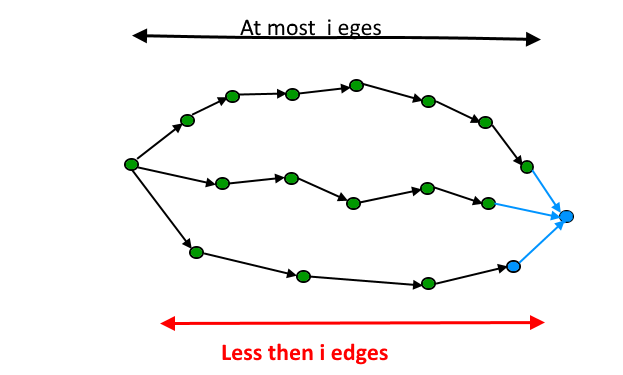
return 0;

}

**Output :**

No

**How does it work?**   
As discussed, the [Bellman-Ford algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/), for a given source, first calculates the shortest distances which have at most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be a maximum |V| – 1 edge on any simple path, that is why the outer loop runs |v| – 1 time. If there is a negative weight cycle, then one more iteration would give a short route.



**How to handle a disconnected graph (If the cycle is not reachable from the source)?**   
The above algorithm and program might not work if the given graph is disconnected. It works when all vertices are reachable from source vertex 0.  
To handle disconnected graphs, we can repeat the process for vertices for which distance is infinite.

// A C++ program for Bellman-Ford's single source

// shortest path algorithm.

#include <bits/stdc++.h>

using namespace std;

// a structure to represent a weighted edge in graph

struct Edge {

int src, dest, weight;

};

// 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.

struct 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[graph->E];

return graph;

}

// The main function that finds shortest distances

// from src to all other vertices using Bellman-

// Ford algorithm. The function also detects

// negative weight cycle

bool isNegCycleBellmanFord(struct Graph\* graph,

int src, int dist[])

{

int V = graph->V;

int E = graph->E;

// Step 1: Initialize distances from src

// to all other vertices as INFINITE

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

dist[i] = INT\_MAX;

dist[src] = 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 u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

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

}

}

// 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 u = graph->edge[i].src;

int v = graph->edge[i].dest;

int weight = graph->edge[i].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

return true;

}

return false;

}

// Returns true if given graph has negative weight

// cycle.

bool isNegCycleDisconnected(struct Graph\* graph)

{

int V = graph->V;

// To keep track of visited vertices to avoid

// recomputations.

bool visited[V];

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

// This array is filled by Bellman-Ford

int dist[V];

// Call Bellman-Ford for all those vertices

// that are not visited

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

if (visited[i] == false) {

// If cycle found

if (isNegCycleBellmanFord(graph, i, dist))

return true;

// Mark all vertices that are visited

// in above call.

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

if (dist[i] != INT\_MAX)

visited[i] = true;

}

}

return false;

}

// Driver program to test above functions

int main()

{

/\* Let us create the graph given in above example \*/

int V = 5; // Number of vertices in graph

int E = 8; // Number of edges in graph

struct Graph\* graph = createGraph(V, E);

// add edge 0-1 (or A-B in above figure)

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = -1;

// add edge 0-2 (or A-C in above figure)

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 4;

// add edge 1-2 (or B-C in above figure)

graph->edge[2].src = 1;

graph->edge[2].dest = 2;

graph->edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 2;

// add edge 1-4 (or A-E in above figure)

graph->edge[4].src = 1;

graph->edge[4].dest = 4;

graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)

graph->edge[5].src = 3;

graph->edge[5].dest = 2;

graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)

graph->edge[6].src = 3;

graph->edge[6].dest = 1;

graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)

graph->edge[7].src = 4;

graph->edge[7].dest = 3;

graph->edge[7].weight = -3;

if (isNegCycleDisconnected(graph))

cout << "Yes";

else

cout << "No";

return 0;

}

**Output :**

No

In this post, [Floyd Warshall Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/) based solution is discussed that works for both connected and disconnected graphs.  
Distance of any node from itself is always zero. But in some cases, as in this example, when we traverse further from 4 to 1, the distance comes out to be -2, i.e. distance of 1 from 1 will become -2. This is our catch, we just have to check the nodes distance from itself and if it comes out to be negative, we will detect the required negative cycle.

// C++ Program to check if there is a negative weight

// cycle using Floyd Warshall Algorithm

#include<bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 4

/\* Define Infinite as a large enough value. This

value will be used for vertices not connected

to each other \*/

#define INF 99999

// A function to print the solution matrix

void printSolution(int dist[][V]);

// Returns true if graph has negative weight cycle

// else false.

bool negCyclefloydWarshall(int graph[][V])

{

/\* dist[][] will be the output matrix that will

finally have the shortest

distances between every pair of vertices \*/

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

/\* Initialize the solution matrix same as input

graph matrix. Or we can say the initial values

of shortest distances are based on shortest

paths considering no intermediate vertex. \*/

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

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

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

/\* Add all vertices one by one to the set of

intermediate vertices.

---> Before start of a iteration, we have shortest

distances between all pairs of vertices such

that the shortest distances consider only the

vertices in set {0, 1, 2, .. k-1} as intermediate

vertices.

----> After the end of a iteration, vertex no. k is

added to the set of intermediate vertices and

the set becomes {0, 1, 2, .. k} \*/

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

{

// Pick all vertices as source one by one

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

{

// Pick all vertices as destination for the

// above picked source

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

{

// If vertex k is on the shortest path from

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

if (dist[i][k] + dist[k][j] < dist[i][j])

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

}

}

}

// If distance of any vertex from itself

// becomes negative, then there is a negative

// weight cycle.

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

if (dist[i][i] < 0)

return true;

return false;

}

// driver program

int main()

{

/\* Let us create the following weighted graph

1

(0)----------->(1)

/|\ |

| |

-1 | | -1

| \|/

(3)<-----------(2)

-1 \*/

int graph[V][V] = { {0 , 1 , INF , INF},

{INF , 0 , -1 , INF},

{INF , INF , 0 , -1},

{-1 , INF , INF , 0}};

if (negCyclefloydWarshall(graph))

cout << "Yes";

else

cout << "No";

return 0;

}

**Output:**

Yes

# 358. Longest path in a Directed Acyclic Graph

Given a Weighted **D**irected **A**cyclic **G**raph (DAG) and a source vertex s in it, find the longest distances from s to all other vertices in the given graph.  
The longest path problem for a general graph is not as easy as the shortest path problem because the longest path problem doesn’t have [optimal substructure property](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/). In fact, [the Longest Path problem is NP-Hard for a general graph](http://en.wikipedia.org/wiki/Longest_path_problem). However, the longest path problem has a linear time solution for directed acyclic graphs. The idea is similar to [linear time solution for shortest path in a directed acyclic graph.](https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/), we use [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/).   
We initialize distances to all vertices as minus infinite and distance to source as 0, then we find a [topological sorting](https://www.geeksforgeeks.org/topological-sorting/) of the graph. Topological Sorting of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a) ). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.  
Following figure shows step by step process of finding longest paths.

LongestPath

Following is complete algorithm for finding longest distances.   
**1)** Initialize dist[] = {NINF, NINF, ….} and dist[s] = 0 where s is the source vertex. Here NINF means negative infinite.   
**2)** Create a topological order of all vertices.   
**3)** Do following for every vertex u in topological order.   
………..Do following for every adjacent vertex v of u   
………………if (dist[v] < dist[u] + weight(u, v))   
………………………dist[v] = dist[u] + weight(u, v) 

Following is C++ implementation of the above algorithm.

// A C++ program to find single source longest distances

// in a DAG

#include <iostream>

#include <limits.h>

#include <list>

#include <stack>

#define NINF INT\_MIN

using namespace std;

// Graph is represented using adjacency list. Every

// node of adjacency list contains vertex number of

// the vertex to which edge connects. It also

// contains weight of the edge

class AdjListNode {

int v;

int weight;

public:

AdjListNode(int \_v, int \_w)

{

v = \_v;

weight = \_w;

}

int getV() { return v; }

int getWeight() { return weight; }

};

// Class to represent a graph using adjacency list

// representation

class Graph {

int V; // No. of vertices'

// Pointer to an array containing adjacency lists

list<AdjListNode>\* adj;

// A function used by longestPath

void topologicalSortUtil(int v, bool visited[],

stack<int>& Stack);

public:

Graph(int V); // Constructor

~Graph(); // Destructor

// function to add an edge to graph

void addEdge(int u, int v, int weight);

// Finds longest distances from given source vertex

void longestPath(int s);

};

Graph::Graph(int V) // Constructor

{

this->V = V;

adj = new list<AdjListNode>[V];

}

Graph::~Graph() // Destructor

{

delete [] adj;

}

void Graph::addEdge(int u, int v, int weight)

{

AdjListNode node(v, weight);

adj[u].push\_back(node); // Add v to u's list

}

// A recursive function used by longestPath. See below

// link for details

// https:// www.geeksforgeeks.org/topological-sorting/

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<AdjListNode>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i) {

AdjListNode node = \*i;

if (!visited[node.getV()])

topologicalSortUtil(node.getV(), visited, Stack);

}

// Push current vertex to stack which stores topological

// sort

Stack.push(v);

}

// The function to find longest distances from a given vertex.

// It uses recursive topologicalSortUtil() to get topological

// sorting.

void Graph::longestPath(int s)

{

stack<int> Stack;

int dist[V];

// 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);

// Initialize distances to all vertices as infinite and

// distance to source as 0

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

dist[i] = NINF;

dist[s] = 0;

// Process vertices in topological order

while (Stack.empty() == false) {

// Get the next vertex from topological order

int u = Stack.top();

Stack.pop();

// Update distances of all adjacent vertices

list<AdjListNode>::iterator i;

if (dist[u] != NINF) {

for (i = adj[u].begin(); i != adj[u].end(); ++i){

if (dist[i->getV()] < dist[u] + i->getWeight())

dist[i->getV()] = dist[u] + i->getWeight();

}

}

}

// Print the calculated longest distances

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

(dist[i] == NINF) ? cout << "INF " : cout << dist[i] << " ";

delete [] visited;

}

// Driver program to test above functions

int main()

{

// Create a graph given in the above diagram.

// Here vertex numbers are 0, 1, 2, 3, 4, 5 with

// following mappings:

// 0=r, 1=s, 2=t, 3=x, 4=y, 5=z

Graph g(6);

g.addEdge(0, 1, 5);

g.addEdge(0, 2, 3);

g.addEdge(1, 3, 6);

g.addEdge(1, 2, 2);

g.addEdge(2, 4, 4);

g.addEdge(2, 5, 2);

g.addEdge(2, 3, 7);

g.addEdge(3, 5, 1);

g.addEdge(3, 4, -1);

g.addEdge(4, 5, -2);

int s = 1;

cout << "Following are longest distances from "

"source vertex "

<< s << " \n";

g.longestPath(s);

return 0;

}

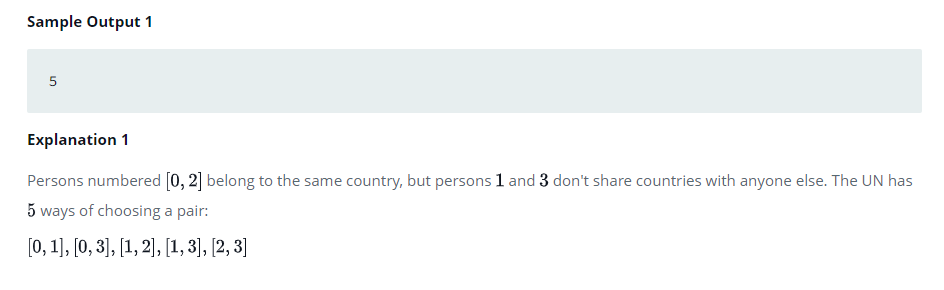
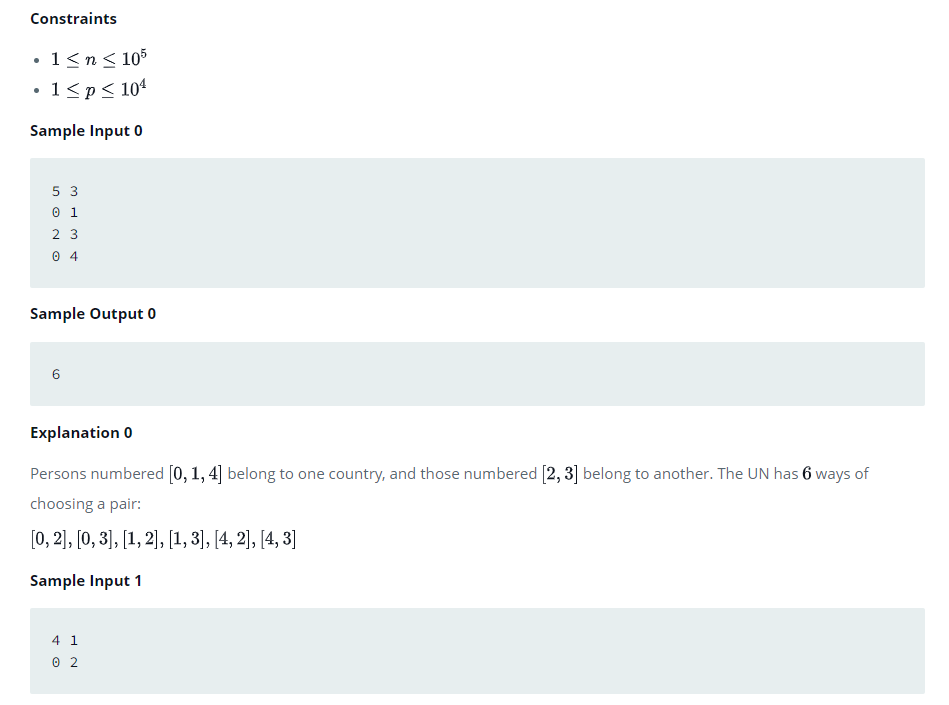
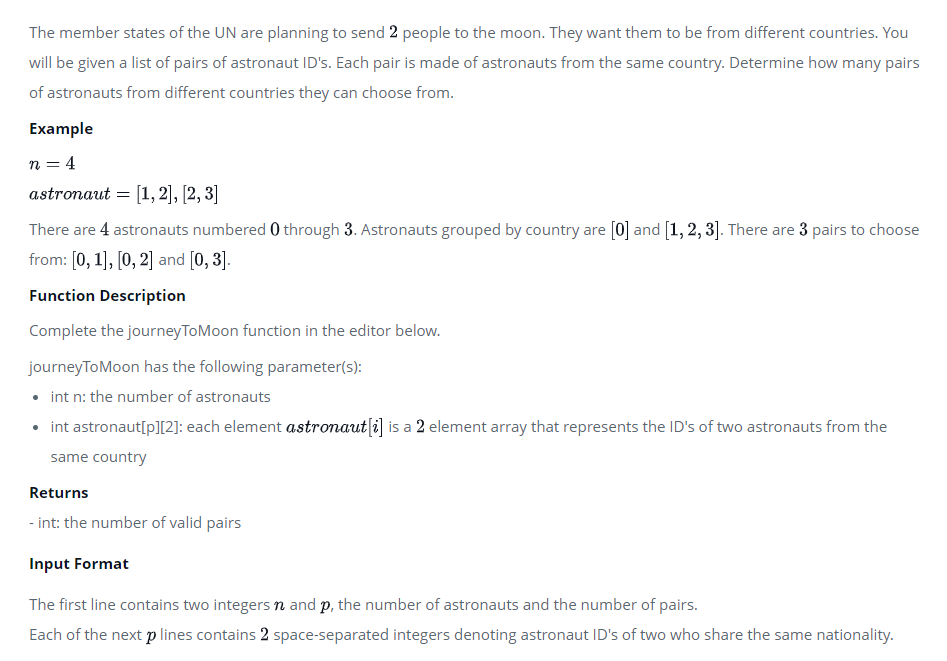
**Output:**

Following are longest distances from source vertex 1

INF 0 2 9 8 10

**Time Complexity:** Time complexity of topological sorting is O(V+E). After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is O(E). So the inner loop runs O(V+E) times. Therefore, overall time complexity of this algorithm is O(V+E).

# 359. [Journey to the Moon](https://www.hackerrank.com/challenges/journey-to-the-moon/problem)



## Solution:

**My Implementation:**

void dfs(vector<int> adj[], vector<int> &visited, int node, int &compo){

    for(int i=0;i<adj[node].size();i++){

        if(visited[adj[node][i]]==-1){

            visited[adj[node][i]] = 1;

            dfs(adj, visited, adj[node][i], compo);

        }

    }

    compo++;

}

long long int journeyToMoon(int n, vector<vector<int>> astronaut) {

    vector<int> lencompo, adj[n], visited(n, -1);

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

        adj[astronaut[i][0]].push\_back(astronaut[i][1]);

        adj[astronaut[i][1]].push\_back(astronaut[i][0]);

    }

    for(int i=0;i<n;i++){

        if(visited[i]==-1){

            visited[i] = 1;

            int compo = 0;

            dfs(adj, visited, i, compo);

            lencompo.push\_back(compo);

        }

    }

    long long int res=0;

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

        cout<<lencompo[i]<<endl;

        res += ((n-lencompo[i])\*lencompo[i]);

    }

    return res/2;

}

**Editorial Solution:**

This problem can be thought of as a graph problem. The first step is to determine the number of countries represente. This can be done using a depth first search. Everyone from a country is part of the same connected component. After creating the components, use combinatorics to determine the number of ways a pair can be formed.

Let us assume that component i contains Mi people. To get number of ways to select two persons from different components, we subtract the number of ways of selecting two persons from the same component from the total numbers of ways of selecting two persons, i.e.

Ways = N choose 2 - (∑(Mi Choose 2) for i = 1 to M)

Problem Setter's code:

//Animesh Sinha

#include <iostream>

#include <list>

#include <vector>

#include <stdio.h>

#include <iterator>

#include <cmath>

#define MAX 100000

using namespace std;

list<int> \*ad;

int \*visited;

int vertices;

void DFS(int u)

{

visited[u] = 1;

vertices++;

list<int>::iterator it;

for(it=ad[u].begin();it!=ad[u].end();it++)

{

if(visited[\*it] == 0)

{

visited[\*it] = 1;

DFS(\*it);

}

}

}

int main()

{

int i,m,u,v,numComponents=0,allv=0,temp=2,count=0;

long long int n;

int eachC[MAX];

cin >> n >> m;

if(n == 1)

{

cout <<"0\n";

return 0;

}

ad = new list<int>[n];

list<int>::iterator it;

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

{

cin >> u >> v;

ad[u].push\_back(v);

ad[v].push\_back(u);

}

visited = new int[n];

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

{

visited[i] = 0;

}

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

{

if(visited[i] == 0)

{

vertices = 0;

DFS(i);

eachC[numComponents] = vertices;

numComponents++;

}

}

long long int totalWays = n\*(n-1) / 2;

long long int sameWays = 0;

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

{

sameWays = sameWays + (eachC[i]\*(eachC[i]-1) / 2);

}

cout << (totalWays - sameWays) << endl;

return 0;

}

# 362. Oliver and the Game

Oliver and Bob are best friends. They have spent their entire childhood in the beautiful city of Byteland. The people of Byteland live happily along with the King.  
The city has a unique architecture with total **N** houses. The King's Mansion is a very big and beautiful bungalow having **address = 1**. Rest of the houses in Byteland have some unique address, (**say A**), are connected by roads and there is always a *unique path* between any two houses in the city. Note that the King's Mansion is also included in these houses.

Oliver and Bob have decided to play Hide and Seek taking the entire city as their arena. In the given scenario of the game, it's Oliver's turn to hide and Bob is supposed to find him.  
Oliver can hide in any of the houses in the city including the King's Mansion. As Bob is a very lazy person, for finding Oliver, he either goes *towards the King's Mansion* (he stops when he reaches there), or he moves *away from the Mansion* in any possible path till the last house on that path.

Oliver runs and hides in some house (**say X**) and Bob is starting the game from his house (**say Y**). If Bob reaches house X, then he surely finds Oliver.

Given Q queries, you need to tell Bob if it is possible for him to find Oliver or not.

The queries can be of the following two types:  
**0 X Y :** Bob moves towards the King's Mansion.  
**1 X Y :** Bob moves away from the King's Mansion

**INPUT :**  
The first line of the input contains a single integer **N**, total number of houses in the city. Next **N-1** lines contain two space separated integers **A** and **B** denoting a road between the houses at address **A** and **B**.  
Next line contains a single integer **Q** denoting the number of queries.  
Following Q lines contain three space separated integers representing each query as explained above.

**OUTPUT :**  
Print "**YES**" or "**NO**" for each query depending on the answer to that query.

**CONSTRAINTS :**  
1 ≤ N ≤ 10^5  
1 ≤ A,B ≤ N  
1 ≤ Q ≤ 5\*10^5  
1 ≤ X,Y ≤ N

**NOTE :**  
Large Input size. Use printf scanf or other fast I/O methods.

**Sample Input**

9

1 2

1 3

2 6

2 7

6 9

7 8

3 4

3 5

5

0 2 8

1 2 8

1 6 5

0 6 5

1 9 1

**Sample Output**

YES

NO

NO

NO

YES

Time Limit: 1

Memory Limit: 256

Source Limit:

**Explanation**

Query 1 Bob goes from 8 towards 1 meeting 2 in the path. Query 2 Bob goes from 8 away from 1 and never meets 2. Query 3 Bob goes from 5 away from 1 and never meets 6. Query 4 Bob goes from 5 towards 1 and never meets 6. Query 5 Bob goes from 1 away from 1 and meets finds Oliver at 9. he can take the following two paths 1 -> 2 -> 6 -> 9 OR 1 -> 2 -> 7 -> 8 9 appears in atleast one of them

## Solution:

This Problem can be solved by Depth First Search and Topological Sorting.

The idea is to maintain a global timer variable which stores an in time and an out time during DFS calls. See author's solution for clarity.

In any query we just need to check if one node is fully contained within another or not. In other words, if one node lies in the sub tree of the other node, then the answer might be YES depending on the Query type 0 or 1.

**Author's Solution**

#include<bits/stdc++.h>

using namespace std;

int vertex;

vector<vector<int> > tree; //used for representing the tree

vector<bool> visited;

vector<int> starttime; // starttime[i] notes the time at which DFS enters node i

vector<int> endtime; // endtime[i] notes the time at which DFS exits node i

int timer = 0; // a global variable that stores the timer at that instant

void makeTree() // takes the input and creates a directed graph representing the tree

{

scanf("%d",&vertex);

tree.resize(vertex+1);

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

{ int x,y;

scanf("%d%d",&x,&y);

tree[x].push\_back(y);

}

}

void measureTime(int v) // Performs Depth First Search

{

visited[v] = 1;

starttime[v] = timer++;

for(int i = 0 ; i < tree[v].size() ; i++) // calling measureTime() for adjacent nodes of node v and performing DFS

{

if( visited [ tree[ v ][ i ] ] == 0 )

measureTime(tree[v][i]);

}

endtime[v] = timer++;

}

int check(int x, int y)

{

if( starttime[x] > starttime[y] && endtime[x] < endtime[y] ) // checks weather node x lies in the subtree of node y or not

return 1;

return 0;

}

int main()

{

makeTree();

visited.resize(vertex+1,0);

starttime.resize(vertex+1,0);

endtime.resize(vertex+1,0);

measureTime(1);

int q;

scanf("%d",&q);

while(q--)

{

int type,x,y;

scanf("%d%d%d",&type,&x,&y);

if( !check(x,y) && !check(y,x) )

{

printf("NO\n");

continue;

}

if(type == 0)

{

if(check(y,x) == 1)

printf("YES\n");

else

printf("NO\n");

}

else

{

if(check(x,y) == 1)

printf("YES\n");

else

printf("NO\n");

}

}

return 0;

}

# 360. Cheapest Flights Within K Stops

There are n cities connected by some number of flights. You are given an array flights where flights[i] = [fromi, toi, pricei] indicates that there is a flight from city fromi to city toi with cost pricei.

You are also given three integers src, dst, and k, return ***the cheapest price****from*src*to*dst*with at most*k*stops.*If there is no such route, return-1.

**Example 1:**



**Input:** n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 1

**Output:** 200

**Explanation:** The graph is shown.

The cheapest price from city 0 to city 2 with at most 1 stop costs 200, as marked red in the picture.

**Example 2:**



**Input:** n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 0

**Output:** 500

**Explanation:** The graph is shown.

The cheapest price from city 0 to city 2 with at most 0 stop costs 500, as marked blue in the picture.

**Constraints:**

* 1 <= n <= 100
* 0 <= flights.length <= (n \* (n - 1) / 2)
* flights[i].length == 3
* 0 <= fromi, toi < n
* fromi != toi
* 1 <= pricei <= 104
* There will not be any multiple flights between two cities.
* 0 <= src, dst, k < n
* src != dst

## Solution:

#### Approach 1: Dijkstra's Algorithm

class Solution {

public int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {

// Build the adjacency matrix

int adjMatrix[][] = new int[n][n];

for (int[] flight: flights) {

adjMatrix[flight[0]][flight[1]] = flight[2];

}

// Shortest distances array

int[] distances = new int[n];

// Shortest steps array

int[] currentStops = new int[n];

Arrays.fill(distances, Integer.MAX\_VALUE);

Arrays.fill(currentStops, Integer.MAX\_VALUE);

distances[src] = 0;

currentStops[src] = 0;

// The priority queue would contain (node, cost, stops)

PriorityQueue<int[]> minHeap = new PriorityQueue<int[]>((a, b) -> a[1] - b[1]);

minHeap.offer(new int[]{src, 0, 0});

while (!minHeap.isEmpty()) {

int[] info = minHeap.poll();

int node = info[0], stops = info[2], cost = info[1];

// If destination is reached, return the cost to get here

if (node == dst) {

return cost;

}

// If there are no more steps left, continue

if (stops == K + 1) {

continue;

}

// Examine and relax all neighboring edges if possible

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

if (adjMatrix[node][nei] > 0) {

int dU = cost, dV = distances[nei], wUV = adjMatrix[node][nei];

// Better cost?

if (dU + wUV < dV) {

minHeap.offer(new int[]{nei, dU + wUV, stops + 1});

distances[nei] = dU + wUV;

}

else if (stops < currentStops[nei]) {

// Better steps?

minHeap.offer(new int[]{nei, dU + wUV, stops + 1});

}

currentStops[nei] = stops;

}

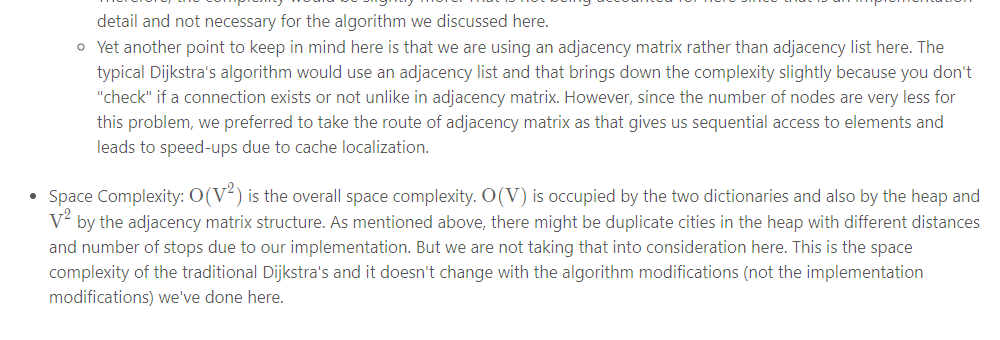
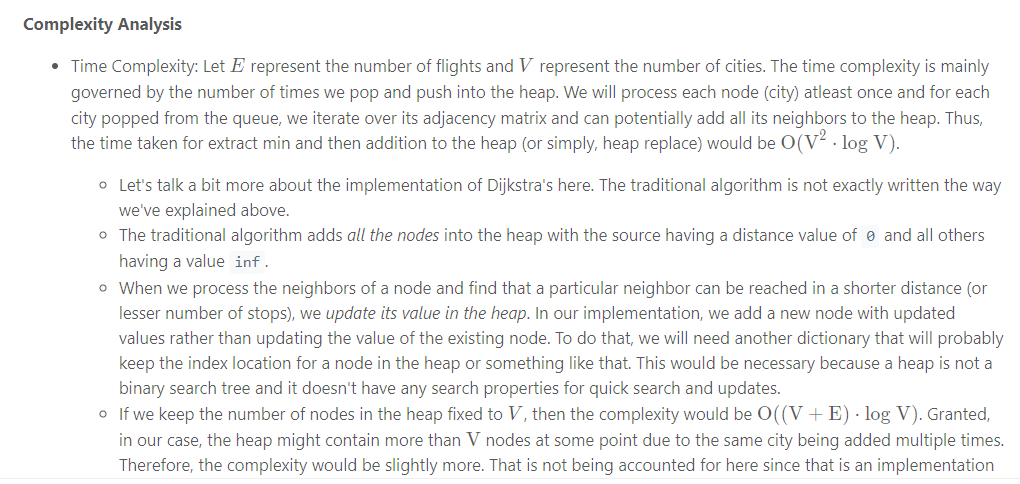
}

}

return distances[dst] == Integer.MAX\_VALUE? -1 : distances[dst];

}

}



#### Approach 2: Depth-First-Search with Memoization

class Solution {

private int[][] adjMatrix;

private HashMap<Pair<Integer, Integer>, Long> memo;

public int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {

this.adjMatrix = new int[n][n];

this.memo = new HashMap<Pair<Integer, Integer>, Long>();

for (int[] flight: flights) {

this.adjMatrix[flight[0]][flight[1]] = flight[2];

}

long ans = this.findShortest(src, K, dst, n);

return ans >= Integer.MAX\_VALUE ? -1 : (int)ans;

}

public long findShortest(int node, int stops, int dst, int n) {

// No need to go any further if the destination is reached

if (node == dst) {

return 0;

}

// Can't go any further if no stops left

if (stops < 0) {

return Integer.MAX\_VALUE;

}

Pair<Integer, Integer> key = new Pair<Integer, Integer>(node, stops);

// If the result of this state is already cached, return it

if (this.memo.containsKey(key)) {

return this.memo.get(key);

}

// Recursive calls over all the neighbors

long ans = Integer.MAX\_VALUE;

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

int weight = this.adjMatrix[node][neighbor];

// 0 value means no edge

if (weight > 0) {

ans = Math.min(ans, this.findShortest(neighbor, stops - 1, dst, n) + weight);

}

}

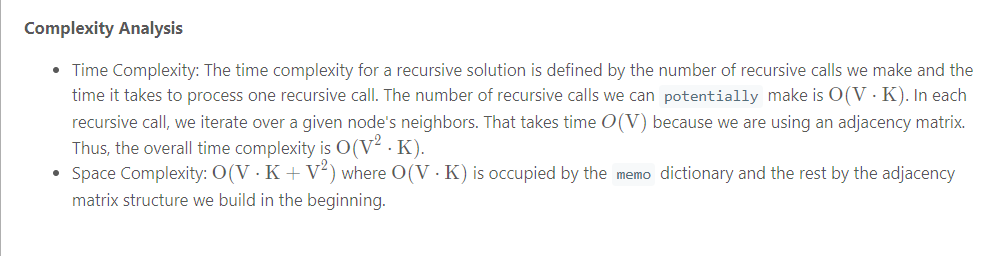
// Cache the result

this.memo.put(key, ans);

return ans;

}

}



#### Approach 3: Bellman-Ford

class Solution {

public int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {

// We use two arrays for storing distances and keep swapping

// between them to save on the memory

long[][] distances = new long[2][n];

Arrays.fill(distances[0], Integer.MAX\_VALUE);

Arrays.fill(distances[1], Integer.MAX\_VALUE);

distances[0][src] = distances[1][src] = 0;

// K + 1 iterations of Bellman Ford

for (int iterations = 0; iterations < K + 1; iterations++) {

// Iterate over all the edges

for (int[] edge : flights) {

int s = edge[0], d = edge[1], wUV = edge[2];

// Current distance of node "s" from src

long dU = distances[1 - iterations&1][s];

// Current distance of node "d" from src

// Note that this will port existing values as

// well from the "previous" array if they didn't already exist

long dV = distances[iterations&1][d];

// Relax the edge if possible

if (dU + wUV < dV) {

distances[iterations&1][d] = dU + wUV;

}

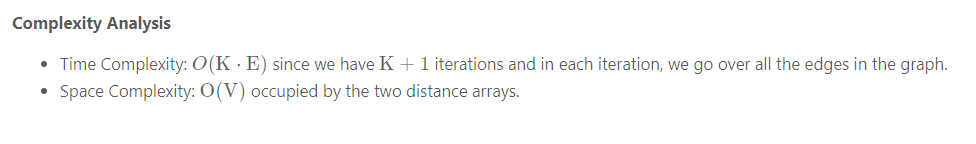
}

}

return distances[K&1][dst] < Integer.MAX\_VALUE ? (int)distances[K&1][dst] : -1;

}

}



#### Approach 4: Breadth First Search

class Solution {

public int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {

// Build the adjacency matrix

int adjMatrix[][] = new int[n][n];

for (int[] flight: flights) {

adjMatrix[flight[0]][flight[1]] = flight[2];

}

// Shortest distances dictionary

HashMap<Pair<Integer, Integer>, Long> distances = new HashMap<Pair<Integer, Integer>, Long>();

distances.put(new Pair<Integer, Integer>(src, 0), 0L);

// Number of stops done

int stops = 0;

// Final answer

long ans = Long.MAX\_VALUE;

// BFS Queue

LinkedList<Integer> bfsQueue = new LinkedList<Integer>();

bfsQueue.add(src);

// Iterate until we exhaust K+1 levels or the queue gets empty

while (!bfsQueue.isEmpty() && stops < K + 1) {

// Iterate on current level

int length = bfsQueue.size();

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

// Loop over neighbors of popped node

int node = bfsQueue.poll();

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

if (adjMatrix[node][nei] > 0) {

long dU = distances.getOrDefault(new Pair<Integer, Integer>(node, stops), Long.MAX\_VALUE);

long dV = distances.getOrDefault(new Pair<Integer, Integer>(nei, stops + 1), Long.MAX\_VALUE);

long wUV = adjMatrix[node][nei];

// No need to update the minimum cost if we have already exhausted our K stops.

if (stops == K && nei != dst) {

continue;

}

if (dU + wUV < dV) {

distances.put(new Pair<Integer, Integer>(nei, stops + 1), dU + wUV);

bfsQueue.add(nei);

// If the neighbor is infact the destination, update the answer accordingly

if (nei == dst) {

ans = Math.min(ans, dU + wUV);

}

}

}

}

}

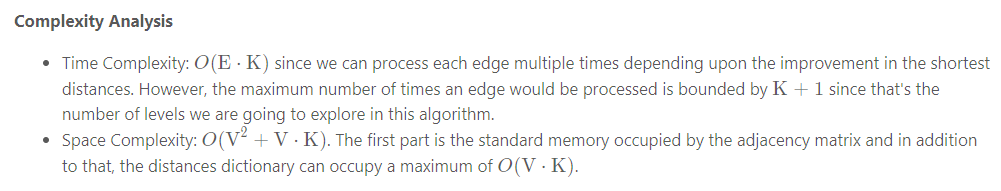
stops++;

}

return ans == Long.MAX\_VALUE ? -1 : (int) ans;

}

}



# 362. Water Jug problem using BFS

You are given an m liter jug and a n liter jug. Both the jugs are initially empty. The jugs don’t have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d is less than n.

(X, Y) corresponds to a state where X refers to the amount of water in Jug1 and Y refers to the amount of water in Jug2   
Determine the path from the initial state (xi, yi) to the final state (xf, yf), where (xi, yi) is (0, 0) which indicates both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0).

The operations you can perform are:

1. Empty a Jug, (X, Y)->(0, Y) Empty Jug 1
2. Fill a Jug, (0, 0)->(X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

**Examples:**

Input : 4 3 2

Output : {(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2)}

## Solution:

We have discussed the optimal solution in [The Two Water Jug Puzzle](https://www.geeksforgeeks.org/two-water-jug-puzzle/). In this post, a [BFS](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/) based solution is discussed.

Here, we keep exploring **all the different valid cases of the states of water in the jug simultaneously** until and unless we reach the required target water.

As provided in the problem statement, at any given state we can do either of the following operations:

1. Fill a jug

2. Empty a jug

3. Transfer water from one jug to another **until either of them gets completely filled or empty.**

**Running of the** **algorithm:**

We start at an initial state in the queue where both the jugs are empty. We then continue to explore all the possible intermediate states derived from the current jug state using the operations provided.

We also, maintain a visited matrix of states so that we avoid revisiting the same state of jugs again and again.

| Cases | Jug 1 | Jug 2 | Is Valid |
| --- | --- | --- | --- |
| Case 1 | Fill it | Empty it |  |
| Case 2 | Empty it | Fill it |  |
| Case 3 | Fill it | Fill it | Redundant case |
| Case 4 | Empty it | Empty it | Already visited (Initial State) |
| Case 5 | Unchanged | Fill it |  |
| Case 6 | Fill it | Unchanged |  |
| Case 7 | Unchanged | Empty |  |
| Case 8 | Empty | Unchanged |  |
| Case 9 | Transfer water from this | Transfer water into this |  |
| Case 10 | Transfer water into this | Transfer water from this |  |

From the table above, we can observe that the state **where both the jugs are filled is redundant** as we won’t be able to continue ahead / do anything with this state in any possible way.

So, we proceed, **keeping in mind all the valid state cases** (as shown in the table above) and we do a BFS on them.

In the BFS, we firstly skip the states which was already visited or if the amount of water in either of the jugs exceeded the jug quantity.

If we continue further, then we firstly mark the current state as visited and check if in this state, if we have obtained the target quantity of water in either of the jugs, we can empty the other jug and return the current state’s entire path.

But, if we have not yet found the target quantity, we then derive the intermediate states from the current state of jugs i.e. we derive the valid cases, mentioned in the table above (go through the code once if you have some confusion).

We keep repeating all the above steps until we have found our target or there are no more states left to proceed with.

#include <bits/stdc++.h>

using namespace std;

typedef pair<int,int> pii;

void printpath(map<pii,pii>mp ,pii u)

{

if(u.first==0 &&u.second==0)

{

cout<<0<<" "<<0<<endl;

return ;

}

printpath(mp,mp[u]);

cout<<u.first<<" "<<u.second<<endl;

}

void BFS(int a ,int b, int target)

{

map<pii, int>m;

bool isSolvable =false;

vector<tuple<int ,int ,int>>path;

map<pii, pii>mp;

queue<pii>q;

q.push(make\_pair(0,0));

while(!q.empty())

{

auto u =q.front();

// cout<<u.first<<" "<<u.second<<endl;

q.pop();

if(m[u]==1)

continue;

if ((u.first > a || u.second > b || u.first < 0 || u.second < 0))

continue;

// cout<<u.first<<" "<<u.second<<endl;

m[{u.first,u.second}]=1;

if(u.first == target || u.second==target)

{

isSolvable = true;

printpath(mp,u);

if (u.first == target) {

if (u.second != 0)

cout<<u.first<<" "<<0<<endl;

}

else {

if (u.first != 0)

cout<<0<<" "<<u.second<<endl;

}

return;

}

// completely fill the jug 2

if(m[{u.first,b}]!=1)

{q.push({u.first,b});

mp[{u.first,b}]=u;}

// completely fill the jug 1

if(m[{a,u.second}]!=1)

{ q.push({a,u.second});

mp[{a,u.second}]=u;}

//transfer jug 1 -> jug 2

int d = b - u.second;

if(u.first >= d)

{

int c = u.first - d;

if(m[{c,b}]!=1)

{q.push({c,b});

mp[{c,b}]=u;}

}

else

{

int c = u.first + u.second;

if(m[{0,c}]!=1)

{q.push({0,c});

mp[{0,c}]=u;}

}

//transfer jug 2 -> jug 1

d = a - u.first;

if(u.second >= d)

{

int c = u.second - d;

if(m[{a,c}]!=1)

{q.push({a,c});

mp[{a,c}]=u;}

}

else

{

int c = u.first + u.second;

if(m[{c,0}]!=1)

{q.push({c,0});

mp[{c,0}]=u;}

}

// empty the jug 2

if(m[{u.first,0}]!=1)

{ q.push({u.first,0});

mp[{u.first,0}]=u;}

// empty the jug 1

if(m[{0,u.second}]!=1)

{q.push({0,u.second});

mp[{0,u.second}]=u;}

}

if (!isSolvable)

cout << "No solution";

}

int main()

{

int Jug1 = 5, Jug2 = 7, target = 3;

cout << "Path from initial state "

"to solution state ::\n";

BFS(Jug1, Jug2, target);

return 0;

}

**Output**

Path of states of jugs followed is :

0 , 0

0 , 3

3 , 0

3 , 3

4 , 2

0 , 2

**Time Complexity: O(n\*m).**

**Space Complexity: O(n\*m)**. Where n and m are the quantity of jug1 and jug2 respectively.

# The Two Water Jug Puzzle

You are on the side of the river. You are given a **m** liter jug and a **n** liter jug where **0 < m < n**. Both the jugs are initially empty. The jugs don’t have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d < n. Determine the minimum no of operations to be performed to obtain d liters of water in one of jug.   
The operations you can perform are:

1. Empty a Jug
2. Fill a Jug
3. Pour water from one jug to the other until one of the jugs is either empty or full.

## Solution:

There are several ways of solving this problem including BFS and DP. In this article, an arithmetic approach to solving the problem is discussed. The problem can be modeled by means of the Diophantine equation of the form mx + ny = d which is solvable if and only if gcd(m, n) divides d. Also, the solution x,y for which equation is satisfied can be given using the [Extended Euclid algorithm for GCD](https://www.geeksforgeeks.org/basic-and-extended-euclidean-algorithms/).   
For example, if we have a jug J1 of 5 liters (n = 5) and another jug J2 of 3 liters (m = 3) and we have to measure 1 liter of water using them. The associated equation will be 5n + 3m = 1. First of all this problem can be solved since gcd(3,5) = 1 which divides 1 (See [this](https://www.geeksforgeeks.org/measure-1-litre-from-two-vessels-infinite-water-supply/) for detailed explanation). Using the Extended Euclid algorithm, we get values of n and m for which the equation is satisfied which are n = 2 and m = -3. These values of n, m also have some meaning like here n = 2 and m = -3 means that we have to fill J1 twice and empty J2 thrice.   
Now to find the minimum no of operations to be performed we have to decide which jug should be filled first. Depending upon which jug is chosen to be filled and which to be emptied we have two different solutions and the minimum among them would be our answer.

**Solution 1 (Always pour from m liter jug into n liter jug)**

1. Fill the m litre jug and empty it into n liter jug.
2. Whenever the m liter jug becomes empty fill it.
3. Whenever the n liter jug becomes full empty it.
4. Repeat steps 1,2,3 till either n liter jug or the m liter jug contains d litres of water.

Each of steps 1, 2 and 3 are counted as one operation that we perform. Let us say algorithm 1 achieves the task in C1 no of operations.

**Solution 2 (Always pour from n liter jug into m liter jug)**

1. Fill the n liter jug and empty it into m liter jug.
2. Whenever the n liter jug becomes empty fill it.
3. Whenever the m liter jug becomes full empty it.
4. Repeat steps 1, 2 and 3 till either n liter jug or the m liter jug contains d liters of water.

Let us say solution 2 achieves the task in C2 no of operations.  
Now our final solution will be a minimum of C1 and C2.  
Now we illustrate how both of the solutions work. Suppose there are a 3 liter jug and a 5 liter jug to measure 4 liters water so **m = 3,n = 5 and d = 4**. The associated Diophantine equation will be 3m + 5n = 4. We use pair (x, y) to represent amounts of water inside the 3-liter jug and 5-liter jug respectively in each pouring step.

**Using Solution 1, successive pouring steps are:**

(0,0)->(3,0)->(0,3)->(3,3)->(1,5)->(1,0)->(0,1)->(3,1)->(0,4)

Hence the no of operations you need to perform are **8.**

**Using Solution 2, successive pouring steps are:**

(0,0)->(0,5)->(3,2)->(0,2)->(2,0)->(2,5)->(3,4)

Hence the no of operations you need to perform are **6.**  
Therefore, we would use solution 2 to measure 4 liters of water in 6 operations or moves.

Based on the explanation here is the implementation.

// C++ program to count minimum number of steps

// required to measure d litres water using jugs

// of m liters and n liters capacity.

#include <bits/stdc++.h>

using namespace std;

// Utility function to return GCD of 'a'

// and 'b'.

int gcd(int a, int b)

{

if (b==0)

return a;

return gcd(b, a%b);

}

/\* fromCap -- Capacity of jug from which

water is poured

toCap -- Capacity of jug to which

water is poured

d -- Amount to be measured \*/

int pour(int fromCap, int toCap, int d)

{

// Initialize current amount of water

// in source and destination jugs

int from = fromCap;

int to = 0;

// Initialize count of steps required

int step = 1; // Needed to fill "from" Jug

// Break the loop when either of the two

// jugs has d litre water

while (from != d && to != d)

{

// Find the maximum amount that can be

// poured

int temp = min(from, toCap - to);

// Pour "temp" liters from "from" to "to"

to += temp;

from -= temp;

// Increment count of steps

step++;

if (from == d || to == d)

break;

// If first jug becomes empty, fill it

if (from == 0)

{

from = fromCap;

step++;

}

// If second jug becomes full, empty it

if (to == toCap)

{

to = 0;

step++;

}

}

return step;

}

// Returns count of minimum steps needed to

// measure d liter

int minSteps(int m, int n, int d)

{

// To make sure that m is smaller than n

if (m > n)

swap(m, n);

// For d > n we cant measure the water

// using the jugs

if (d > n)

return -1;

// If gcd of n and m does not divide d

// then solution is not possible

if ((d % gcd(n,m)) != 0)

return -1;

// Return minimum two cases:

// a) Water of n liter jug is poured into

// m liter jug

// b) Vice versa of "a"

return min(pour(n,m,d), // n to m

pour(m,n,d)); // m to n

}

// Driver code to test above

int main()

{

int n = 3, m = 5, d = 4;

printf("Minimum number of steps required is %d",

minSteps(m, n, d));

return 0;

}

**Output:**

Minimum number of steps required is 6

**Time Complexity:** O(N + M)  
**Auxiliary Space:**O(1) 

# 363. Find if there is a path of more thank length from a source

Given a graph, a source vertex in the graph and a number k, find if there is a simple path (without any cycle) starting from given source and ending at any other vertex such that the distance from source to that vertex is atleast ‘k’ length.

**Example:**

Input : Source s = 0, k = 58

Output : True

There exists a simple path 0 -> 7 -> 1

-> 2 -> 8 -> 6 -> 5 -> 3 -> 4

Which has a total distance of 60 km which

is more than 58.

Input : Source s = 0, k = 62

Output : False

In the above graph, the longest simple

path has distance 61 (0 -> 7 -> 1-> 2

-> 3 -> 4 -> 5-> 6 -> 8, so output

should be false for any input greater

than 61.

## Solution:

One important thing to note is, simply doing BFS or DFS and picking the longest edge at every step would not work. The reason is, a shorter edge can produce longer path due to higher weight edges connected through it.  
The idea is to use Backtracking. We start from given source, explore all paths from current vertex. We keep track of current distance from source. If distance becomes more than k, we return true. If a path doesn’t produces more than k distance, we backtrack.  
How do we make sure that the path is simple and we don’t loop in a cycle? The idea is to keep track of current path vertices in an array. Whenever we add a vertex to path, we check if it already exists or not in current path. If it exists, we ignore the edge.  
Below is implementation of above idea.

// Program to find if there is a simple path with

// weight more than k

#include<bits/stdc++.h>

using namespace std;

// iPair ==> Integer Pair

typedef pair<int, int> iPair;

// This class represents a dipathted graph using

// adjacency list representation

class Graph

{

int V; // No. of vertices

// In a weighted graph, we need to store vertex

// and weight pair for every edge

list< pair<int, int> > \*adj;

bool pathMoreThanKUtil(int src, int k, vector<bool> &path);

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int u, int v, int w);

bool pathMoreThanK(int src, int k);

};

// Returns true if graph has path more than k length

bool Graph::pathMoreThanK(int src, int k)

{

// Create a path array with nothing included

// in path

vector<bool> path(V, false);

// Add source vertex to path

path[src] = 1;

return pathMoreThanKUtil(src, k, path);

}

// Prints shortest paths from src to all other vertices

bool Graph::pathMoreThanKUtil(int src, int k, vector<bool> &path)

{

// If k is 0 or negative, return true;

if (k <= 0)

return true;

// Get all adjacent vertices of source vertex src and

// recursively explore all paths from src.

list<iPair>::iterator i;

for (i = adj[src].begin(); i != adj[src].end(); ++i)

{

// Get adjacent vertex and weight of edge

int v = (\*i).first;

int w = (\*i).second;

// If vertex v is already there in path, then

// there is a cycle (we ignore this edge)

if (path[v] == true)

continue;

// If weight of is more than k, return true

if (w >= k)

return true;

// Else add this vertex to path

path[v] = true;

// If this adjacent can provide a path longer

// than k, return true.

if (pathMoreThanKUtil(v, k-w, path))

return true;

// Backtrack

path[v] = false;

}

// If no adjacent could produce longer path, return

// false

return false;

}

// Allocates memory for adjacency list

Graph::Graph(int V)

{

this->V = V;

adj = new list<iPair> [V];

}

// Utility function to an edge (u, v) of weight w

void Graph::addEdge(int u, int v, int w)

{

adj[u].push\_back(make\_pair(v, w));

adj[v].push\_back(make\_pair(u, w));

}

// Driver program to test methods of graph class

int main()

{

// create the graph given in above fugure

int V = 9;

Graph g(V);

// making above shown graph

g.addEdge(0, 1, 4);

g.addEdge(0, 7, 8);

g.addEdge(1, 2, 8);

g.addEdge(1, 7, 11);

g.addEdge(2, 3, 7);

g.addEdge(2, 8, 2);

g.addEdge(2, 5, 4);

g.addEdge(3, 4, 9);

g.addEdge(3, 5, 14);

g.addEdge(4, 5, 10);

g.addEdge(5, 6, 2);

g.addEdge(6, 7, 1);

g.addEdge(6, 8, 6);

g.addEdge(7, 8, 7);

int src = 0;

int k = 62;

g.pathMoreThanK(src, k)? cout << "Yes\n" :

cout << "No\n";

k = 60;

g.pathMoreThanK(src, k)? cout << "Yes\n" :

cout << "No\n";

return 0;

}

**Output:** 

No

Yes

**Exercise:**  
Modify the above solution to find weight of longest path from a given source.  
**Time Complexity:** O(n!)   
**Explanation:**   
From the source node, we one-by-one visit all the paths and check if the total weight is greater than k for each path. So, the worst case will be when the number of possible paths is maximum. This is the case when every node is connected to every other node.   
Beginning from the source node we have n-1 adjacent nodes. The time needed for a path to connect any two nodes is 2. One for joining the source and the next adjacent vertex. One for breaking the connection between the source and the old adjacent vertex.   
After selecting a node out of n-1 adjacent nodes, we are left with n-2 adjacent nodes (as the source node is already included in the path) and so on at every step of selecting a node our problem reduces by 1 node.  
We can write this in the form of a recurrence relation as: F(n) = n\*(2+F(n-1))   
This expands to: 2n + 2n\*(n-1) + 2n\*(n-1)\*(n-2) + ……. + 2n(n-1)(n-2)(n-3)…..1   
As n times 2n(n-1)(n-2)(n-3)….1 is greater than the given expression so we can safely say time complexity is: n\*2\*n!   
Here in the question the first node is defined so time complexity becomes   
F(n-1) = 2(n-1)\*(n-1)! = 2\*n\*(n-1)! – 2\*1\*(n-1)! = 2\*n!-2\*(n-1)! = O(n!)

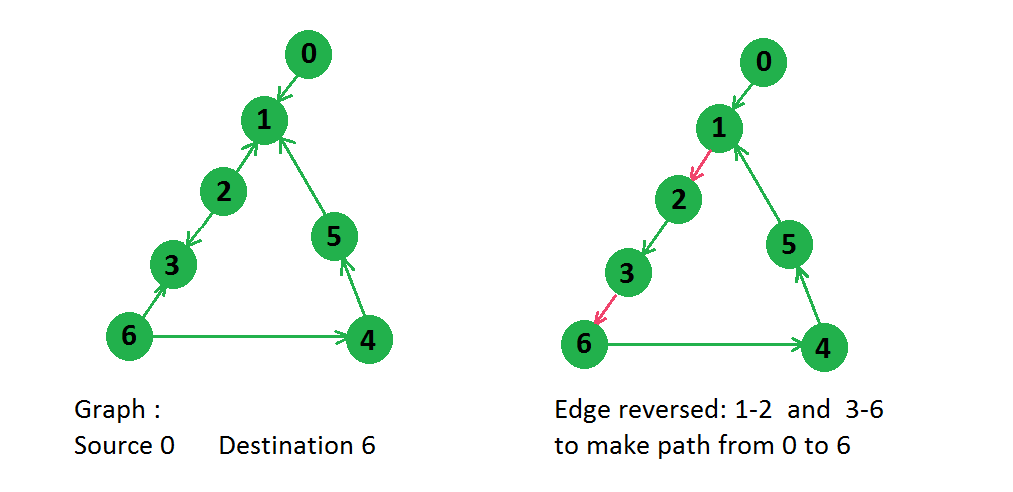
# 364. M-ColouringProblem

## Same as ques 259 of backtracking

# 365. Minimum edges to reverse to make a path from source to destination

Given a directed graph and a source node and destination node, we need to find how many edges we need to reverse in order to make at least 1 path from the source node to the destination node.

Examples:



In above graph there were two paths from node 0 to node 6,

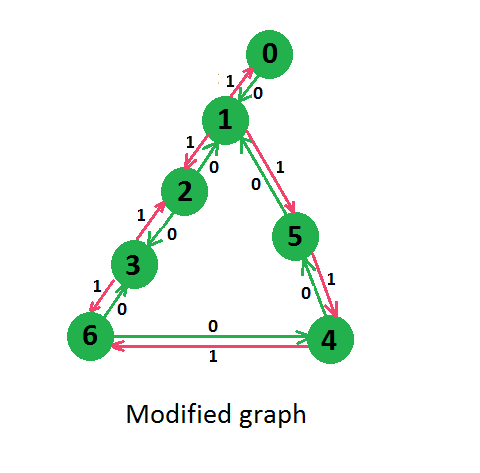
0 -> 1 -> 2 -> 3 -> 6

0 -> 1 -> 5 -> 4 -> 6

But for first path only two edges need to be reversed, so answer will be 2 only.

## Solution:

This problem can be solved assuming a different version of the given graph. In this version we make a reverse edge corresponding to every edge and we assign that a weight 1 and assign a weight 0 to original edge. After this modification above graph looks something like below, 



Now we can see that we have modified the graph in such a way that, if we move towards original edge, no cost is incurred, but if we move toward reverse edge 1 cost is added. So if we apply [Dijkstra’s shortest path](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/) on this modified graph from given source, then that will give us minimum cost to reach from source to destination i.e. minimum edge reversal from source to destination.

Below is the code based on above concept.

// C++ Program to find minimum edge reversal to get

// atleast one path from source to destination

#include <bits/stdc++.h>

using namespace std;

#define INF 0x3f3f3f3f

// This class represents a directed graph using

// adjacency list representation

class Graph

{

int V;

list<pair<int, int>> \*graph;

public:

// Constructor:

Graph(int V)

{

this->V = V;

graph = new list<pair<int, int>>[V];

}

// Adding edges into the graph:

void addEdge(int u, int v, int w)

{

graph[u].push\_back(make\_pair(v, w));

}

// Returns shortest path from source to all other vertices.

vector<int> shortestPath(int source)

{

// Create a set to store vertices that are being preprocessed

set<pair<int, int>> setds;

// Create a vector for distances and initialize all

// distances as infinite (INF)

vector<int> distance(V, INF);

// Insert source itself in Set and initialize its distance as 0.

setds.insert(make\_pair(0, source));

distance = 0;

/\* Looping till all shortest distance are finalized

then setds will become empty \*/

while (!setds.empty())

{

// The first vertex in Set is the minimum distance

// vertex, extract it from set.

pair<int, int> tmp = \*(setds.begin());

setds.erase(setds.begin());

// vertex label is stored in second of pair (it

// has to be done this way to keep the vertices

// sorted distance (distance must be first item

// in pair)

int u = tmp.second;

list<pair<int, int>>::iterator i;

for (i = graph[u].begin(); i != graph[u].end(); ++i)

{

// Get vertex label and weight of current adjacent

// of u.

int v = (\*i).first;

int weight = (\*i).second;

// If there is shorter path to v through u.

if (distance[v] > distance[u] + weight)

{

/\* If distance of v is not INF then it must be in

our set, so removing it and inserting again

with updated less distance.

Note : We extract only those vertices from Set

for which distance is finalized. So for them,

we would never reach here. \*/

if (distance[v] != INF)

setds.erase(setds.find(make\_pair(distance[v], v)));

// Updating distance of v

distance[v] = distance[u] + weight;

setds.insert(make\_pair(distance[v], v));

}

}

}

return distance;

}

Graph modelGraphWithEdgeWeight(int edge[][2], int E, int V)

{

Graph g(V);

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

{

// original edge : weight 0

g.addEdge(edge[i][0], edge[i][1], 0);

// reverse edge : weight 1

g.addEdge(edge[i][1], edge[i][0], 1);

}

return g;

}

int getMinEdgeReversal(int edge[][2], int E, int V, int source, int destination)

{

// get modified graph with edge weight.

Graph g = modelGraphWithEdgeWeight(edge, E, V);

// distance vector stores shortest path.

vector<int> dist = g.shortestPath(source);

// If distance of destination is still INF then we cannot reach destination. Hence, not possible.

if (dist[destination] == INF)

return -1;

else

return dist[destination];

}

};

int main()

{

int V = 7;

Graph g(V);

int edge[][2] = {{0, 1}, {2, 1}, {2, 3}, {5, 1}, {4, 5}, {6, 4}, {6, 3}};

int E = sizeof(edge) / sizeof(edge[0]);

int minEdgeToReverse = g.getMinEdgeReversal(edge, E, V, 0, 6);

if (minEdgeToReverse != -1)

cout << minEdgeToReverse << endl;

else

cout << "Not Possible." << endl;

return 0;

}

**Output:**

2

One more efficient approach to this problem would be by using [0-1 BFS concept](https://www.geeksforgeeks.org/0-1-bfs-shortest-path-binary-graph/).

Below is the implementation of that algorithm:

//Java code to find minimum edge reversal to get

//atleast one path from source to destination using 0-1 BFS

//Code By: Sparsh\_CBS

import java.util.\*;

class Node{

private int val;

private int weight;

private Integer parent;

Node(int val, int weight){

this.val = val;

this.weight = weight;

parent = null;

}

//We have used the concept of parent to avoid

//a child revisiting its parent and pushing it in

//the deque during the 0-1 BFS

Node(int val, int distance, Integer parent){

this.val = val;

this.weight = distance;

this.parent = parent;

}

public int getVal(){

return val;

}

public int getWeight(){

return weight;

}

public Integer getParent(){

return parent;

}

}

public class Gfg{

public static void main(String[] args) {

List<List<Integer>> adj = new ArrayList<>();

for(int i = 0; i < 7; i++)

adj.add(new ArrayList<>());

adj.get(0).add(1);

adj.get(2).add(1);

adj.get(5).add(1);

adj.get(2).add(3);

adj.get(6).add(3);

adj.get(6).add(4);

adj.get(4).add(5);

int ans = getMinRevEdges(adj, 0, 6);

if(ans == Integer.MAX\_VALUE)

System.out.println(-1);

else

System.out.println(ans);

}

private static int getMinRevEdges(List<List<Integer>> adj, int src, int dest){

int n = adj.size();

//Create the given graph into bidirectional graph

List<List<Node>> newAdj = getBiDirectionalGraph(adj);

//Now, Apply 0-1 BFS using Deque to get the shortest path

//In the implementation, we will only add the

//encountered node into the deque if and only if

//the distance at which it was earlier explored was

//strictly larger than the currently encountered distance

Deque<Node> dq = new LinkedList<>();

//Here Node is made up of : Node(int node\_val, int node\_distance, int node\_parent)

dq.offer(new Node(src,0,0));

int[] dist = new int[n];

//Set the distance of all nodes to infinity(Integer.MAX\_VALUE)

Arrays.fill(dist, Integer.MAX\_VALUE);

//set distance of source node as 0

dist[src] = 0;

while(!dq.isEmpty()){

Node curr = dq.pollFirst();

int currVal = curr.getVal();

int currWeight = curr.getWeight();

int currParent = curr.getParent();

//If we encounter the destination node, we return

if(currVal == dest)

return currWeight;

//Iterate over the neighbours of the current Node

for(Node neighbourNode: newAdj.get(currVal)){

int neighbour = neighbourNode.getVal();

if(neighbour == currParent)

continue;

int wt = neighbourNode.getWeight();

if(wt == 0 && dist[neighbour] > currWeight){

dist[neighbour] = currWeight;

dq.offerFirst(new Node(neighbour,currWeight, currVal));

}

else if(dist[neighbour] > currWeight+wt){

dist[neighbour] = currWeight+wt;

dq.offerLast(new Node(neighbour, currWeight+wt, currVal));

}

}

}

return Integer.MAX\_VALUE;

}

private static List<List<Node>> getBiDirectionalGraph(List<List<Integer>> adj){

int n = adj.size();

List<List<Node>> newAdj = new ArrayList<>();

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

newAdj.add(new ArrayList<>());

boolean[] visited = new boolean[n];

Queue<Integer> queue = new LinkedList<>();

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

if(!visited[i]){

visited[i] = true;

queue.offer(i);

while(!queue.isEmpty()){

int curr = queue.poll();

for(int neighbour: adj.get(curr)){

//original edges are to be assigned a weight of 0

newAdj.get(curr).add(new Node(neighbour, 0));

//make a fake edge and assign a weight of 1

newAdj.get(neighbour).add(new Node(curr, 1));

if(visited[neighbour]){

//if the neighbour was visited, then dont

// add it again in the queue

continue;

}

visited[neighbour] = true;

queue.offer(neighbour);

}

}

}

}

return newAdj;

}

}

***Output:***

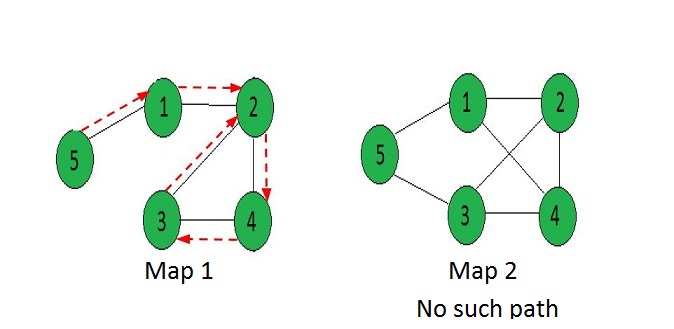
2

**Time Complexity:**O(V+E)

**Space Complexity:**O(V+2\*E)

# 366. Paths to travel each nodes using each edge(Seven Bridges of Königsberg)

There are n nodes and m bridges in between these nodes. Print the possible path through each node using each edges (if possible), traveling through each edges only once.



**Examples :**

Input : [[0, 1, 0, 0, 1],

[1, 0, 1, 1, 0],

[0, 1, 0, 1, 0],

[0, 1, 1, 0, 0],

[1, 0, 0, 0, 0]]

Output : 5 -> 1 -> 2 -> 4 -> 3 -> 2

Input : [[0, 1, 0, 1, 1],

[1, 0, 1, 0, 1],

[0, 1, 0, 1, 1],

[1, 1, 1, 0, 0],

[1, 0, 1, 0, 0]]

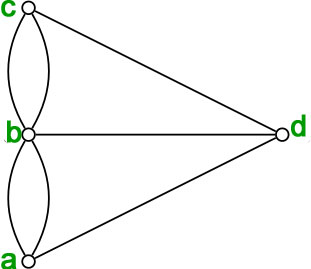
Output : "No Solution"

## Solution:

**Euler paths and circuits :**

* An Euler path is a path that uses every edge of a graph **exactly once**.
* An Euler circuit is a circuit that uses every edge of a graph exactly once.
* An Euler path starts and ends at different vertices.
* An Euler circuit starts and ends at the same vertex.

The **Konigsberg bridge**problem’s graphical representation : 



There are simple criteria for determining whether a multigraph has a Euler path or a Euler circuit. For any multigraph to have a Euler circuit, all the degrees of the vertices must be even.

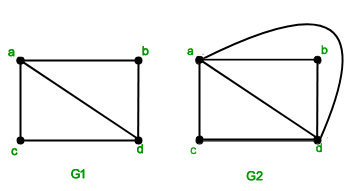
**Theorem –** “A connected multigraph (and simple graph) with at least two vertices has a Euler circuit if and only if **each** of its vertices has an **even degree**.”

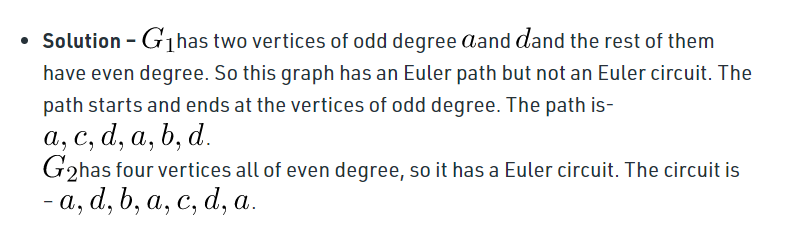
Proof of the above statement is that every time a circuit passes through a vertex, it adds twice to its degree. Since it is a circuit, it starts and ends at the same vertex, which makes it contribute one degree when the circuit starts and one when it ends. In this way, every vertex has an even degree.   
Since the konigsberg graph has vertices having odd degrees, a Euler circuit does not exist in the graph.

**Theorem –** “A connected multigraph (and simple graph) has an Euler path but not an Euler circuit if and only if it has **exactly two vertices** of odd degree.”

The proof is an extension of the proof given above. Since a path may start and end at different vertices, the vertices where the path starts and ends are allowed to have odd degrees.

* **Example –**Which graphs shown below have an Euler path or Euler circuit?





It is one of the famous problems in Graph Theory and known as problem of “Seven Bridges of Königsberg”. This problem was solved by famous mathematician Leonhard Euler in 1735. This problem is also considered as the beginning of Graph Theory.   
The problem back then was that: There was 7 bridges connecting 4 lands around the city of Königsberg in Prussia. Was there any way to start from any of the land and go through each of the bridges once and only once? Please see [these wikipedia images](https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg#/media/File:7_bridges.svg) for more clarity.

Euler first introduced graph theory to solve this problem. He considered each of the lands as a node of a graph and each bridge in between as an edge in between. Now he calculated if there is any [Eulerian Path](https://www.geeksforgeeks.org/mathematics-euler-hamiltonian-paths/) in that graph. If there is an Eulerian path then there is a solution otherwise not.   
Problem here, is a generalized version of the problem in 1735.

Below is the implementation :

// A C++ program print Eulerian Trail in a

// given Eulerian or Semi-Eulerian Graph

#include <iostream>

#include <string.h>

#include <algorithm>

#include <list>

using namespace std;

// A class that represents an undirected graph

class Graph

{

// No. of vertices

int V;

// A dynamic array of adjacency lists

list<int> \*adj;

public:

// Constructor and destructor

Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

~Graph()

{

delete [] adj;

}

// functions to add and remove edge

void addEdge(int u, int v)

{

adj[u].push\_back(v);

adj[v].push\_back(u);

}

void rmvEdge(int u, int v);

// Methods to print Eulerian tour

void printEulerTour();

void printEulerUtil(int s);

// This function returns count of vertices

// reachable from v. It does DFS

int DFSCount(int v, bool visited[]);

// Utility function to check if edge u-v

// is a valid next edge in Eulerian trail or circuit

bool isValidNextEdge(int u, int v);

};

/\* The main function that print Eulerian Trail.

It first finds an odd degree vertex (if there is any)

and then calls printEulerUtil() to print the path \*/

void Graph::printEulerTour()

{

// Find a vertex with odd degree

int u = 0;

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

if (adj[i].size() & 1)

{

u = i;

break;

}

// Print tour starting from oddv

printEulerUtil(u);

cout << endl;

}

// Print Euler tour starting from vertex u

void Graph::printEulerUtil(int u)

{

// Recur for all the vertices adjacent to

// this vertex

list<int>::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

{

int v = \*i;

// If edge u-v is not removed and it's a a

// valid next edge

if (v != -1 && isValidNextEdge(u, v))

{

cout << u << "-" << v << " ";

rmvEdge(u, v);

printEulerUtil(v);

}

}

}

// The function to check if edge u-v can be considered

// as next edge in Euler Tout

bool Graph::isValidNextEdge(int u, int v)

{

// The edge u-v is valid in one of the following

// two cases:

// 1) If v is the only adjacent vertex of u

int count = 0; // To store count of adjacent vertices

list<int>::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

if (\*i != -1)

count++;

if (count == 1)

return true;

// 2) If there are multiple adjacents, then u-v

// is not a bridge

// Do following steps to check if u-v is a bridge

// 2.a) count of vertices reachable from u

bool visited[V];

memset(visited, false, V);

int count1 = DFSCount(u, visited);

// 2.b) Remove edge (u, v) and after removing

// the edge, count vertices reachable from u

rmvEdge(u, v);

memset(visited, false, V);

int count2 = DFSCount(u, visited);

// 2.c) Add the edge back to the graph

addEdge(u, v);

// 2.d) If count1 is greater, then edge (u, v)

// is a bridge

return (count1 > count2)? false: true;

}

// This function removes edge u-v from graph.

// It removes the edge by replacing adjacent

// vertex value with -1.

void Graph::rmvEdge(int u, int v)

{

// Find v in adjacency list of u and replace

// it with -1

list<int>::iterator iv = find(adj[u].begin(),

adj[u].end(), v);

\*iv = -1;

// Find u in adjacency list of v and replace

// it with -1

list<int>::iterator iu = find(adj[v].begin(),

adj[v].end(), u);

\*iu = -1;

}

// A DFS based function to count reachable

// vertices from v

int Graph::DFSCount(int v, bool visited[])

{

// Mark the current node as visited

visited[v] = true;

int count = 1;

// Recur for all vertices adjacent to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (\*i != -1 && !visited[\*i])

count += DFSCount(\*i, visited);

return count;

}

// Driver program to test above function

int main()

{

// Let us first create and test

// graphs shown in above figure

Graph g1(4);

g1.addEdge(0, 1);

g1.addEdge(0, 2);

g1.addEdge(1, 2);

g1.addEdge(2, 3);

g1.printEulerTour();

Graph g3(4);

g3.addEdge(0, 1);

g3.addEdge(1, 0);

g3.addEdge(0, 2);

g3.addEdge(2, 0);

g3.addEdge(2, 3);

g3.addEdge(3, 1);

// comment out this line and you will see that

// it gives TLE because there is no possible

// output g3.addEdge(0, 3);

g3.printEulerTour();

return 0;

}

**Output:**

2-0 0-1 1-2 2-3

1-0 0-2 2-3 3-1 1-0 0-2

# 367. Vertex Cover Problem

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either ‘u’ or ‘v’ is in the vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. ***Given an undirected graph, the vertex cover problem is to find minimum size vertex cover***.   
The following are some examples. 

VertexCover

## Solution:

[Vertex Cover Problem](http://en.wikipedia.org/wiki/Vertex_cover) is a known [NP Complete problem](https://www.geeksforgeeks.org/np-completeness-set-1/), i.e., there is no polynomial-time solution for this unless P = NP. There are approximate polynomial-time algorithms to solve the problem though. Following is a simple approximate algorithm adapted from [CLRS book](http://www.flipkart.com/introduction-algorithms-english-3rd/p/itmdwxyrafdburzg?pid=9788120340077&affid=sandeepgfg).

**Naive Approach:**

Consider all the subset of vertices one by one and find out whether it covers all edges of the graph. For eg. in a graph consisting only 3 vertices the set consisting of the combination of vertices are:{0,1,2,{0,1},{0,2},{1,2},{0,1,2}} . Using each element of this set check whether these vertices cover all  all the edges of the graph. Hence update the optimal answer. And hence print the subset having minimum number of vertices which also covers all the edges of the graph.

**Approximate Algorithm for Vertex Cover:** 

1) Initialize the result as {}

2) Consider a set of all edges in given graph. Let the set be E.

3) Do following while E is not empty

...a) Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to result

...b) Remove all edges from E which are either incident on u or v.

4) Return result

Below diagram to show the execution of the above approximate algorithm: 

vertexCover

**How well the above algorithm perform?**   
It can be proved that the above approximate algorithm never finds a vertex cover whose size is more than twice the size of the minimum possible vertex cover (Refer [this](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/AproxAlgor/vertexCover.htm)for proof)  
**Implementation:**   
The following are C++ and Java implementations of the above approximate algorithm.

// Program to print Vertex Cover of a given undirected graph

#include<iostream>

#include <list>

using namespace std;

// This class represents a undirected graph using adjacency list

class Graph

{

int V; // No. of vertices

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

public:

Graph(int V); // Constructor

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

void printVertexCover(); // prints vertex cover

};

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.

adj[w].push\_back(v); // Since the graph is undirected

}

// The function to print vertex cover

void Graph::printVertexCover()

{

// Initialize all vertices as not visited.

bool visited[V];

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

visited[i] = false;

list<int>::iterator i;

// Consider all edges one by one

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

{

// An edge is only picked when both visited[u] and visited[v]

// are false

if (visited[u] == false)

{

// Go through all adjacents of u and pick the first not

// yet visited vertex (We are basically picking an edge

// (u, v) from remaining edges.

for (i= adj[u].begin(); i != adj[u].end(); ++i)

{

int v = \*i;

if (visited[v] == false)

{

// Add the vertices (u, v) to the result set.

// We make the vertex u and v visited so that

// all edges from/to them would be ignored

visited[v] = true;

visited[u] = true;

break;

}

}

}

}

// Print the vertex cover

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

if (visited[i])

cout << i << " ";

}

// Driver program to test methods of graph class

int main()

{

// Create a graph given in the above diagram

Graph g(7);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(3, 4);

g.addEdge(4, 5);

g.addEdge(5, 6);

g.printVertexCover();

return 0;

}

Output:

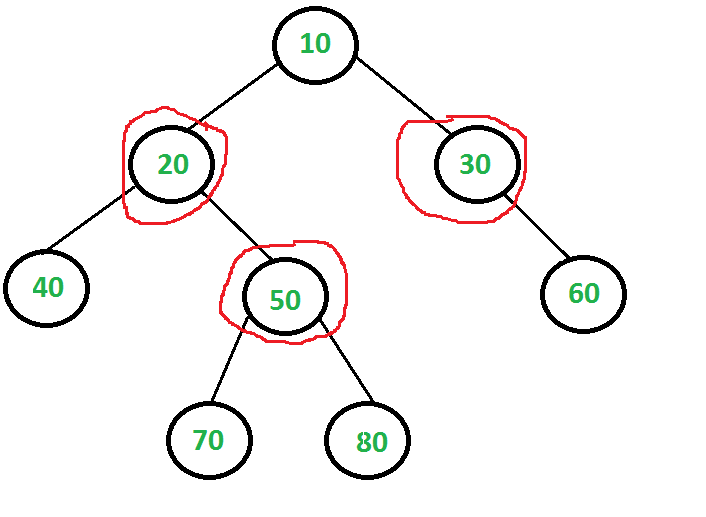
0 1 3 4 5 6

The Time Complexity of the above algorithm is O(V + E).

# Vertex Cover Problem for Tree

A [vertex cover of an undirected graph](https://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm-2/) is a subset of its vertices such that for every edge (u, v) of the graph, either ‘u’ or ‘v’ is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph.   
The problem to find minimum size vertex cover of a graph is [NP complete](https://www.geeksforgeeks.org/np-completeness-set-1/). But it can be solved in polynomial time for trees. In this post a solution for Binary Tree is discussed. The same solution can be extended for n-ary trees.

For example, consider the following binary tree. The smallest vertex cover is {20, 50, 30} and size of the vertex cover is 3.



## Solution:

The idea is to consider following two possibilities for root and recursively for all nodes down the root.   
***1) Root is part of vertex cover:***In this case root covers all children edges. We recursively calculate size of vertex covers for left and right subtrees and add 1 to the result (for root).

***2) Root is not part of vertex cover:*** In this case, both children of root must be included in vertex cover to cover all root to children edges. We recursively calculate size of vertex covers of all grandchildren and number of children to the result (for two children of root).

Below are implementation of above idea.

// A naive recursive C implementation for vertex cover problem for a tree

#include <stdio.h>

#include <stdlib.h>

// A utility function to find min of two integers

int min(int x, int y) { return (x < y)? x: y; }

/\* A binary tree node has data, pointer to left child and a pointer to

right child \*/

struct node

{

int data;

struct node \*left, \*right;

};

// The function returns size of the minimum vertex cover

int vCover(struct node \*root)

{

// The size of minimum vertex cover is zero if tree is empty or there

// is only one node

if (root == NULL)

return 0;

if (root->left == NULL && root->right == NULL)

return 0;

// Calculate size of vertex cover when root is part of it

int size\_incl = 1 + vCover(root->left) + vCover(root->right);

// Calculate size of vertex cover when root is not part of it

int size\_excl = 0;

if (root->left)

size\_excl += 1 + vCover(root->left->left) + vCover(root->left->right);

if (root->right)

size\_excl += 1 + vCover(root->right->left) + vCover(root->right->right);

// Return the minimum of two sizes

return min(size\_incl, size\_excl);

}

// A utility function to create a node

struct node\* newNode( int data )

{

struct node\* temp = (struct node \*) malloc( sizeof(struct node) );

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// Driver program to test above functions

int main()

{

// Let us construct the tree given in the above diagram

struct node \*root = newNode(20);

root->left = newNode(8);

root->left->left = newNode(4);

root->left->right = newNode(12);

root->left->right->left = newNode(10);

root->left->right->right = newNode(14);

root->right = newNode(22);

root->right->right = newNode(25);

printf ("Size of the smallest vertex cover is %d ", vCover(root));

return 0;

}

**Output:**

Size of the smallest vertex cover is 3

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. For example, vCover of node with value 50 is evaluated twice as 50 is grandchild of 10 and child of 20.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So Vertex Cover problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems,](https://www.geeksforgeeks.org/archives/tag/dynamic-programming) re-computations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Following is the implementation of Dynamic Programming based solution. In the following solution, an additional field ‘vc’ is added to tree nodes. The initial value of ‘vc’ is set as 0 for all nodes. The recursive function vCover() calculates ‘vc’ for a node only if it is not already set.

/\* Dynamic programming based program for Vertex Cover problem for

a Binary Tree \*/

#include <stdio.h>

#include <stdlib.h>

// A utility function to find min of two integers

int min(int x, int y) { return (x < y)? x: y; }

/\* A binary tree node has data, pointer to left child and a pointer to

right child \*/

struct node

{

int data;

int vc;

struct node \*left, \*right;

};

// A memoization based function that returns size of the minimum vertex cover.

int vCover(struct node \*root)

{

// The size of minimum vertex cover is zero if tree is empty or there

// is only one node

if (root == NULL)

return 0;

if (root->left == NULL && root->right == NULL)

return 0;

// If vertex cover for this node is already evaluated, then return it

// to save recomputation of same subproblem again.

if (root->vc != 0)

return root->vc;

// Calculate size of vertex cover when root is part of it

int size\_incl = 1 + vCover(root->left) + vCover(root->right);

// Calculate size of vertex cover when root is not part of it

int size\_excl = 0;

if (root->left)

size\_excl += 1 + vCover(root->left->left) + vCover(root->left->right);

if (root->right)

size\_excl += 1 + vCover(root->right->left) + vCover(root->right->right);

// Minimum of two values is vertex cover, store it before returning

root->vc = min(size\_incl, size\_excl);

return root->vc;

}

// A utility function to create a node

struct node\* newNode( int data )

{

struct node\* temp = (struct node \*) malloc( sizeof(struct node) );

temp->data = data;

temp->left = temp->right = NULL;

temp->vc = 0; // Set the vertex cover as 0

return temp;

}

// Driver program to test above functions

int main()

{

// Let us construct the tree given in the above diagram

struct node \*root = newNode(20);

root->left = newNode(8);

root->left->left = newNode(4);

root->left->right = newNode(12);

root->left->right->left = newNode(10);

root->left->right->right = newNode(14);

root->right = newNode(22);

root->right->right = newNode(25);

printf ("Size of the smallest vertex cover is %d ", vCover(root));

return 0;

}

**Output:**

Size of the smallest vertex cover is 3

**Approach for any general tree :**

1. Approach will be same dynamic programming approach as discussed.

2. For every node, if we exclude this node from vertex cover than we have to include its neighbouring nodes,

   and if we include this node in the vertex cover than we will take the minimum of the two possibilities of taking its neighbouring

   nodes in the vertex cover to get minimum vertex cover.

3. We will store the above information in the dp array.

// C++ implementation for the above approach

#include <bits/stdc++.h>

using namespace std;

// An utility function to add an edge in the tree

void addEdge(vector<int> adj[], int x, int y)

{

adj[x].push\_back(y);

adj[y].push\_back(x);

}

void dfs(vector<int> adj[], vector<int> dp[], int src,

int par)

{

for (auto child : adj[src]) {

if (child != par)

dfs(adj, dp, child, src);

}

for (auto child : adj[src]) {

if (child != par) {

// not including source in the vertex cover

dp[src][0] += dp[child][1];

// including source in the vertex cover

dp[src][1] += min(dp[child][1], dp[child][0]);

}

}

}

// function to find minimum size of vertex cover

void minSizeVertexCover(vector<int> adj[], int N)

{

vector<int> dp[N + 1];

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

// 0 denotes not included in vertex cover

dp[i].push\_back(0);

// 1 denotes included in vertex cover

dp[i].push\_back(1);

}

dfs(adj, dp, 1, -1);

// printing minimum size vertex cover

cout << min(dp[1][0], dp[1][1]) << endl;

}

// Driver Code

int main()

{

/\* 1

/ \

2 7

/ \

3 6

/ | \

4 8 5

\*/

// number of nodes in the tree

int N = 8;

// adjacency list representation of the tree

vector<int> adj[N + 1];

addEdge(adj, 1, 2);

addEdge(adj, 1, 7);

addEdge(adj, 2, 3);

addEdge(adj, 2, 6);

addEdge(adj, 3, 4);

addEdge(adj, 3, 8);

addEdge(adj, 3, 5);

minSizeVertexCover(adj, N);

return 0;

}

**Output**

3

**Time Complexity :** O(N)

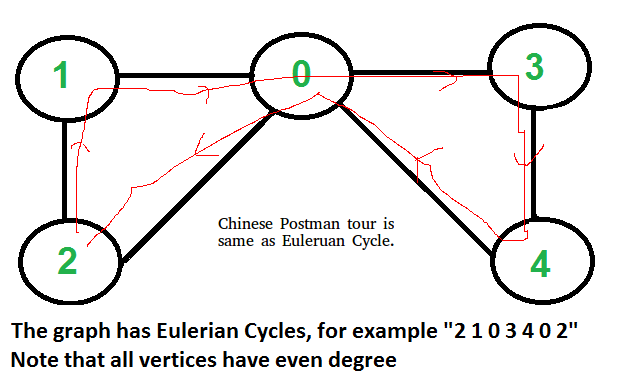
**Auxiliary space :** O(N)

# 368. Chinese Postman or Route Inspection

[Chinese Postman Problem](https://en.wikipedia.org/wiki/Route_inspection_problem) is a variation of [Eulerian circuit](https://www.geeksforgeeks.org/eulerian-path-and-circuit/) problem for undirected graphs. An Euler Circuit is a closed walk that covers every edge once starting and ending position is same. Chinese Postman problem is defined for connected and undirected graph. The problem is to find shortest path or circuity that visits every edge of the graph at least once.

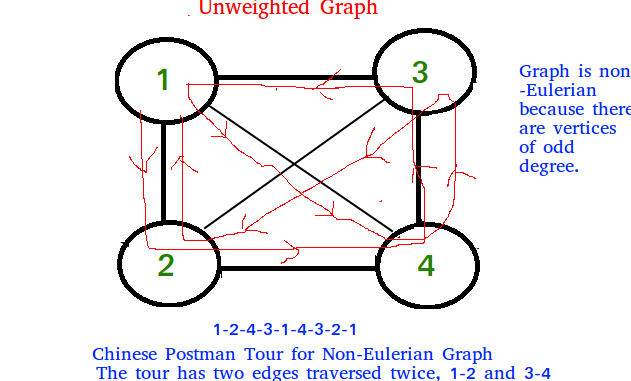
## Solution:

**If input graph contains Euler Circuit, then a solution of the problem is Euler Circuit**   
An undirected and connected graph has Eulerian cycle if “all vertices have even degree“.

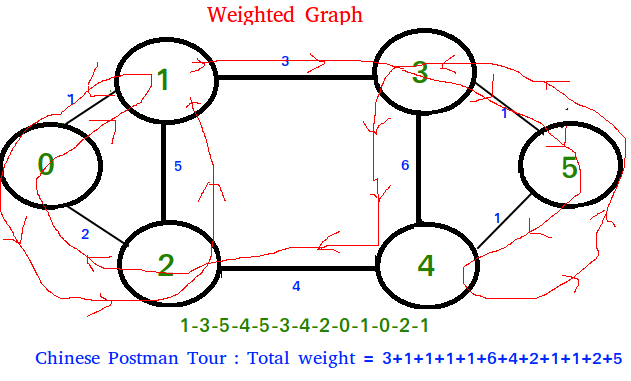


It doesn’t matter whether graph is weighted or unweighted, the Chinese Postman Route is always same as Eulerian Circuit if it exists. In weighted graph the minimum possible weight of Postman tour is sum of all edge weights which we get through Eulerian Circuit. We can’t get a shorter route as we must visit all edges at-least once.

**If input graph does NOT contain Euler Circuit**   
In this case, the task reduces to following.   
1) In unweighted graph, minimum number of edges to duplicate so that the given graph converts to a graph with Eulerian Cycle. 



2) In weighted graph, minimum total weight of edges to duplicate so that given graph converts to a graph with Eulerian Cycle.



Algorithm to find shortest closed path or optimal

Chinese postman route in a weighted graph that may

not be Eulerian.

step 1 : If graph is Eulerian, return sum of all

edge weights.Else do following steps.

step 2 : We find all the vertices with odd degree

step 3 : List all possible pairings of odd vertices

For n odd vertices total number of pairings

possible are, (n-1) \* (n-3) \* (n -5)... \* 1

step 4 : For each set of pairings, find the shortest

path connecting them.

step 5 : Find the pairing with minimum shortest path

connecting pairs.

step 6 : Modify the graph by adding all the edges that

have been found in step 5.

step 7 : Weight of Chinese Postman Tour is sum of all

edges in the modified graph.

step 8 : Print Euler Circuit of the modified graph.

This Euler Circuit is Chinese Postman Tour.

**Illustration :**

3

(a)-----------------(b)

1 / | | \1

/ | | \

(c) | 5 6| (d)

\ | | /

2 \ | 4 | /1

(e)------------------(f)

As we see above graph does not contain Eulerian circuit

because is has odd degree vertices [a, b, e, f]

they all are odd degree vertices .

First we make all possible pairs of odd degree vertices

[ae, bf], [ab, ef], [af, eb]

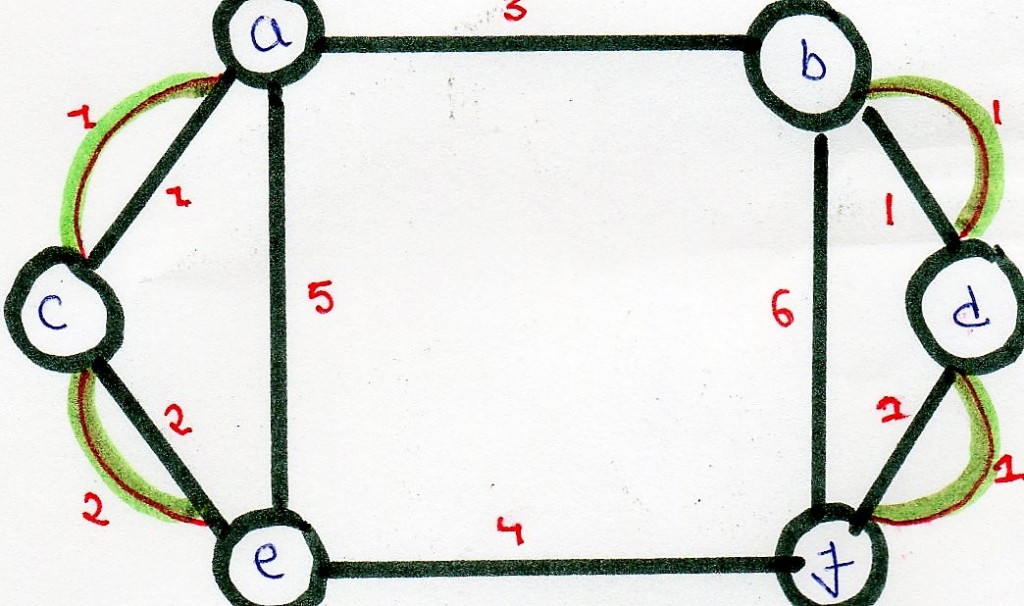
so pairs with min sum of weight are [ae, bf] :

ae = (ac + ce = 3 ), bf = ( bd + df = 2 )

Total : 5

We add edges ac, ce, bd and df to the original graph and

create a modified graph.



Optimal chinese postman route is of length : 5 + 23 =

28 [ 23 = sum of all edges of modified graph ]

Chinese Postman Route :

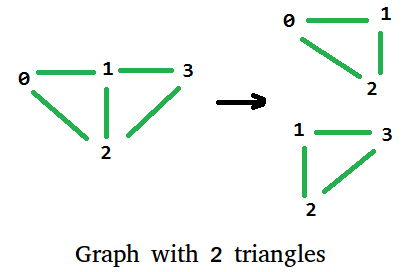
a - b - d - f - d - b - f - e - c - a - c - e - a

This route is Euler Circuit of the modified graph.

# 369. Number of Triangles in a Directed and Undirected Graph

# Number of Triangles in an Undirected Graph

Given an Undirected simple graph, We need to find how many triangles it can have. For example below graph have 2 triangles in it.



## Solution:

Let A[][] be the adjacency matrix representation of the graph. If we calculate A3, then the number of triangles in Undirected Graph is equal to trace(A3) / 6. Where trace(A) is the sum of the elements on the main diagonal of matrix A. 

Trace of a graph represented as adjacency matrix A[V][V] is,

trace(A[V][V]) = A[0][0] + A[1][1] + .... + A[V-1][V-1]

Count of triangles = trace(A3) / 6

Below is the implementation of the above formula.

// A C++ program for finding

// number of triangles in an

// Undirected Graph. The program

// is for adjacency matrix

// representation of the graph

#include <bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 4

// Utility function for matrix

// multiplication

void multiply(int A[][V], int B[][V], int C[][V])

{

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

{

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

{

C[i][j] = 0;

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

C[i][j] += A[i][k]\*B[k][j];

}

}

}

// Utility function to calculate

// trace of a matrix (sum of

// diagonal elements)

int getTrace(int graph[][V])

{

int trace = 0;

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

trace += graph[i][i];

return trace;

}

// Utility function for calculating

// number of triangles in graph

int triangleInGraph(int graph[][V])

{

// To Store graph^2

int aux2[V][V];

// To Store graph^3

int aux3[V][V];

// Initialising aux

// matrices with 0

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

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

aux2[i][j] = aux3[i][j] = 0;

// aux2 is graph^2 now printMatrix(aux2);

multiply(graph, graph, aux2);

// after this multiplication aux3 is

// graph^3 printMatrix(aux3);

multiply(graph, aux2, aux3);

int trace = getTrace(aux3);

return trace / 6;

}

// driver code

int main()

{

int graph[V][V] = {{0, 1, 1, 0},

{1, 0, 1, 1},

{1, 1, 0, 1},

{0, 1, 1, 0}

};

printf("Total number of Triangle in Graph : %d\n",

triangleInGraph(graph));

return 0;

}

**Output:**

Total number of Triangle in Graph : 2

**How does this work?**   
If we compute An for an adjacency matrix representation of the graph, then a value An[i][j] represents the number of distinct walks between vertex i to j in the graph. In A3, we get all distinct paths of length 3 between every pair of vertices.  
A triangle is a cyclic path of length three, i.e. begins and ends at the same vertex. So A3[i][i] represents a triangle beginning and ending with vertex i. Since a triangle has three vertices and it is counted for every vertex, we need to divide the result by 3. Furthermore, since the graph is undirected, every triangle twice as i-p-q-j and i-q-p-j, so we divide by 2 also. Therefore, the number of triangles is trace(A3) / 6.

**Time Complexity:**  
The time complexity of above algorithm is O(V3) where V is number of vertices in the graph, we can improve the performance to O(V2.8074) using [Strassen’s matrix multiplication](https://www.geeksforgeeks.org/strassens-matrix-multiplication/) algorithm.

**Another approach:**Using [Bitsets](https://www.geeksforgeeks.org/c-bitset-and-its-application/) as adjacency lists.

* For each node in the graph compute the corresponding adjacency list as a bitmask.
* If two nodes, i & j, are adjacent compute the number of nodes that are adjacent to i & j and add it to the answer.
* In the end, divide the answer by 6 to avoid duplicates.

In order to compute the number of nodes adjacent to two nodes, i & j, we use the bitwise operation **&** (and) on the adjacency list of i and j, then we count the number of ones.

Below is the implementation of the above approach:

#include<iostream>

#include<string>

#include<algorithm>

#include<cstring>

#include<vector>

#include<bitset>

using namespace std;

#define V 4

int main()

{

// Graph represented as adjacency matrix

int graph[][V] = {{0, 1, 1, 0},

{1, 0, 1, 1},

{1, 1, 0, 1},

{0, 1, 1, 0}};

// create the adjacency list of the graph (as bit masks)

// set the bits at positions [i][j] & [j][i] to 1, if there is an undirected edge between i and j

vector<bitset<V>> Bitset\_Adj\_List(V);

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

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

if(graph[i][j])

Bitset\_Adj\_List[i][j] = 1,

Bitset\_Adj\_List[j][i] = 1;

int ans = 0;

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

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

// if i & j are adjancent

// compute the number of nodes that are adjancent to i & j

if(Bitset\_Adj\_List[i][j] == 1 && i != j){

bitset<V> Mask = Bitset\_Adj\_List[i] & Bitset\_Adj\_List[j];

ans += Mask.count();

}

// divide the answer by 6 to avoid duplicates

ans /= 6;

cout << "The number of Triangles in the Graph is : " << ans;

// This code is contributed

// by Gatea David

}

**Output**

The number of Triangles in the Graph is : 2

**Time Complexity:**First we have the two for nested loops O(V2) flowed by Bitset operations & and count, both have a time complexity of O(V / Word RAM), where V = number of nodes in the graph and Word RAM is usually 32 or 64. So the final time complexity is O(V2\* V / 32) or O(V3).

# Number of Triangles in Directed and Undirected Graphs

Given a Graph, count number of triangles in it. The graph is can be directed or undirected.

**Example:**

Input: digraph[V][V] = { {0, 0, 1, 0},

{1, 0, 0, 1},

{0, 1, 0, 0},

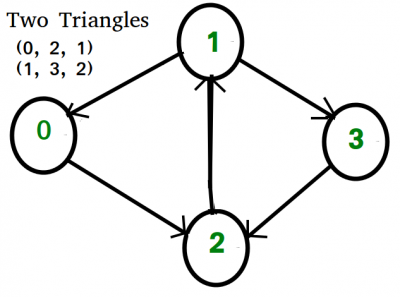
{0, 0, 1, 0}

};

Output: 2

Give adjacency matrix represents following

directed graph.



## Solution:

We have discussed a [method based on graph trace](https://www.geeksforgeeks.org/number-of-triangles-in-a-undirected-graph/) that works for undirected graphs. In this post a new method is discussed with that is simpler and works for both directed and undirected graphs.  
The idea is to use three nested loops to consider every triplet (i, j, k) and check for the above condition (there is an edge from i to j, j to k and k to i)   
However in an **undirected graph**, the triplet (i, j, k) can be permuted to give six combination (See [previous post](https://www.geeksforgeeks.org/number-of-triangles-in-a-undirected-graph/) for details). Hence we divide the total count by 6 to get the actual number of triangles.   
In case of **directed graph**, the number of permutation would be 3 (as order of nodes becomes relevant). Hence in this case the total number of triangles will be obtained by dividing total count by 3. For example consider the directed graph given below

Following is the implementation.

// C++ program to count triangles

// in a graph. The program is for

// adjacency matrix representation

// of the graph.

#include<bits/stdc++.h>

// Number of vertices in the graph

#define V 4

using namespace std;

// function to calculate the

// number of triangles in a

// simple directed/undirected

// graph. isDirected is true if

// the graph is directed, its

// false otherwise

int countTriangle(int graph[V][V],

bool isDirected)

{

// Initialize result

int count\_Triangle = 0;

// Consider every possible

// triplet of edges in graph

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

{

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

{

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

{

// Check the triplet if

// it satisfies the condition

if (graph[i][j] && graph[j][k]

&& graph[k][i])

count\_Triangle++;

}

}

}

// If graph is directed ,

// division is done by 3,

// else division by 6 is done

isDirected? count\_Triangle /= 3 :

count\_Triangle /= 6;

return count\_Triangle;

}

//driver function to check the program

int main()

{

// Create adjacency matrix

// of an undirected graph

int graph[][V] = { {0, 1, 1, 0},

{1, 0, 1, 1},

{1, 1, 0, 1},

{0, 1, 1, 0}

};

// Create adjacency matrix

// of a directed graph

int digraph[][V] = { {0, 0, 1, 0},

{1, 0, 0, 1},

{0, 1, 0, 0},

{0, 0, 1, 0}

};

cout << "The Number of triangles in undirected graph : "

<< countTriangle(graph, false);

cout << "\n\nThe Number of triangles in directed graph : "

<< countTriangle(digraph, true);

return 0;

}

**Output**

The Number of triangles in undirected graph : 2

The Number of triangles in directed graph : 2

**Comparison of this approach with**[**previous approach**](https://www.geeksforgeeks.org/number-of-triangles-in-a-undirected-graph/)**:**   
Advantages:

* No need to calculate Trace.
* Matrix- multiplication is not required.
* Auxiliary matrices are not required hence optimized in space.
* Works for directed graphs.

Disadvantages:

* The time complexity is O(n3) and can’t be reduced any further.

# 370. Minimise the cashflow among a given set of friends who have borrowed money from each other

## Same as ques 229 of Greedy.

# 371. Two Clique Problem

A Clique is a subgraph of graph such that all vertices in subgraph are completely connected with each other. Given a Graph, find if it can be divided into two Cliques.

**Examples:**

Input : G[][] = {{0, 1, 1, 0, 0},

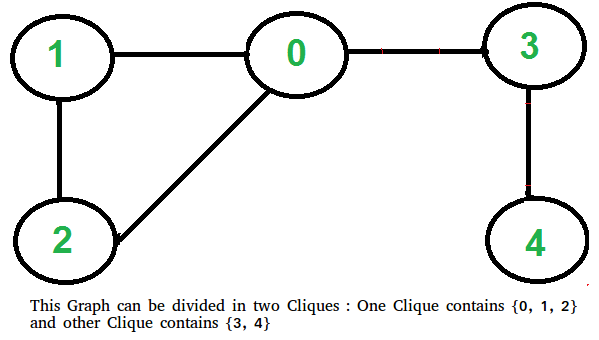
{1, 0, 1, 1, 0},

{1, 1, 0, 0, 0},

{0, 1, 0, 0, 1},

{0, 0, 0, 1, 0}};

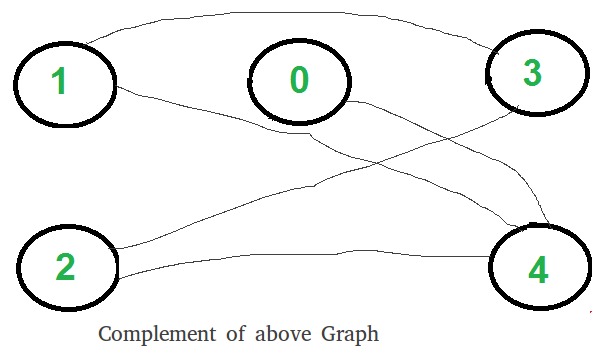
Output : Yes



## Solution:

This problem looks tricky at first, but has a simple and interesting solution. A graph can be divided in two cliques if its complement graph is [Bipartitie](https://www.geeksforgeeks.org/bipartite-graph/). So below are two steps to find if graph can be divided in two Cliques or not.

* Find the complement of Graph. Below is the complement graph is above shown graph. In complement, all original edges are removed. And the vertices which did not have an edge between them, now have an edge connecting them.



* Return true if complement is Bipartite, else false. The above shown graph is Bipartite. Checking whether a Graph is Biparitite or no is discussed [here](https://www.geeksforgeeks.org/bipartite-graph/).

**How does this work?**   
If complement is Bipartite, then graph can be divided into two sets U and V such that there is no edge connecting to vertices of same set. This means in original graph, these sets U and V are completely connected. Hence original graph could be divided in two Cliques.

**Implementation:**   
Below is the implementation of above steps.

// C++ program to find out whether a given graph can be

// converted to two Cliques or not.

#include <bits/stdc++.h>

using namespace std;

const int V = 5;

// This function returns true if subgraph reachable from

// src is Bipartite or not.

bool isBipartiteUtil(int G[][V], int src, int colorArr[])

{

colorArr[src] = 1;

// Create a queue (FIFO) of vertex numbers and enqueue

// source vertex for BFS traversal

queue <int> q;

q.push(src);

// Run while there are vertices in queue (Similar to BFS)

while (!q.empty())

{

// Dequeue a vertex from queue

int u = q.front();

q.pop();

// Find all non-colored adjacent vertices

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

{

// An edge from u to v exists and destination

// v is not colored

if (G[u][v] && colorArr[v] == -1)

{

// Assign alternate color to this adjacent

// v of u

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

q.push(v);

}

// An edge from u to v exists and destination

// v is colored with same color as u

else if (G[u][v] && colorArr[v] == colorArr[u])

return false;

}

}

// If we reach here, then all adjacent vertices can

// be colored with alternate color

return true;

}

// Returns true if a Graph G[][] is Bipartite or not. Note

// that G may not be connected.

bool isBipartite(int G[][V])

{

// Create a color array to store colors assigned

// to all vertices. Vertex number is used as index in

// this array. The value '-1' of colorArr[i]

// is used to indicate that no color is assigned to

// vertex 'i'. The value 1 is used to indicate first

// color is assigned and value 0 indicates

// second color is assigned.

int colorArr[V];

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

colorArr[i] = -1;

// One by one check all not yet colored vertices.

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

if (colorArr[i] == -1)

if (isBipartiteUtil(G, i, colorArr) == false)

return false;

return true;

}

// Returns true if G can be divided into

// two Cliques, else false.

bool canBeDividedinTwoCliques(int G[][V])

{

// Find complement of G[][]

// All values are complemented except

// diagonal ones

int GC[V][V];

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

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

GC[i][j] = (i != j)? !G[i][j] : 0;

// Return true if complement is Bipartite

// else false.

return isBipartite(GC);

}

// Driver program to test above function

int main()

{

int G[][V] = {{0, 1, 1, 1, 0},

{1, 0, 1, 0, 0},

{1, 1, 0, 0, 0},

{0, 1, 0, 0, 1},

{0, 0, 0, 1, 0}

};

canBeDividedinTwoCliques(G) ? cout << "Yes" :

cout << "No";

return 0;

}

**Output :**

Yes

Time complexity of above implementation is O(V2).