**Longest Path in a Directed Acyclic Graph:**

Given a Weighted Directed Acyclic Graph (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.** In fact, the **Longest Path problem is NP-Hard for a general graph. However, the longest path problem has a linear time solution for directed acyclic graphs.**

We initialize distances to all vertices as minus infinite and distance to source as 0, then we find a topological sorting of the graph. Topological Sorting of a graph represents a linear ordering of the graph. 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.

**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);**

**}**

Now, see how it works. We mark the current edge as true.

Then we will check for all nodes which are adjacent to the it **(obviously, it has to be a direct acyclic graph).** Now, we call topologicalSortUtil for those vertices in a recursive manner and finally when a node does not have nay child left, it will be added in the stack.

Now, let’ s know about topological sorting.

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. **Topological Sorting for a graph is not possible if the graph is not a DAG.**

The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).

  
  
  
Now, we already know that the initial vertex or source vertex must be a vertex with in-degree as 0.

So, it would be 4 or 5.

A graph can have more than one topological sorting.

For instance, the following graph have:

5 4 2 3 1 0 as a topological sorting

4 5 2 3 1 0 as a topological sorting.

How to topologically sort a graph:

**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();**

**}**

**}**

**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**

**//Why we cannot just start from one index and do the topological sort**

**//because, graph could be diconnected..I.e. graph is not strongly connected and contains several connected component of the graph**

**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] << " ";**

**}**

Now, this is the portion where we update the longest distance.

**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();**

**}**

**}**

What do we do?

We have the topological sorting of the graph in the stack. Now, we get the stack.top(). after that, we visit all the adjacent nodes of it. And, check if dist[v]<weight(uv)+dist[u]. **if it is, we update dist[v] as we are looking for longest path for all vertices from a given vertex.**

**Find A Mother Vertex In The Graph:**

A mother vertex in a graph G = (V,E) is a vertex v such that all other vertices in G can be reached by a path from v.

We strongly recommend you to minimize your browser and try this yourself first.

How to find mother vertex?

**Case 1:- Undirected Connected Graph :** In this case, all the vertices are mother vertices as we can reach to all the other nodes in the graph.

**Case 2:- Undirected/Directed Disconnected Graph :** In this case, there is no mother vertices as we cannot reach to all the other nodes in the graph.

**Case 3:- Directed Connected Graph :** In this case, we have to find a vertex -v in the graph such that we can reach to all the other nodes in the graph through a directed path.

We can find a mother vertex in O(V+E) time. The idea is based on Kosaraju’s Strongly Connected Component Algorithm.

In a graph of strongly connected components, mother vertices are always vertices of source component in component graph. The idea is based on below fact.

If there exist mother vertex (or vertices), then one of the mother vertices is the last finished vertex in DFS. (Or a mother vertex has the maximum finish time in DFS traversal).

A vertex is said to be finished in DFS if a recursive call for its DFS is over, i.e., all descendants of the vertex have been visited.

**How does the above idea work?**

Let the last finished vertex be v. Basically, we need to prove that there cannot be an edge from another vertex u to v if u is not another mother vertex (Or there cannot exist a non-mother vertex u such that u-→v is an edge). There can be two possibilities.

Recursive DFS call is made for u before v. If an edge u-→v exists, then v must have finished before u because v is reachable through u and a vertex finishes after all its descendants.

Recursive DFS call is made for v before u. In this case also, if an edge u-→v exists, then either v must finish before u (which contradicts our assumption that v is finished at the end) OR u should be reachable from v (which means u is another mother vertex).

**Transitive Closure of a Graph using DFS:**

Given a directed graph, find out if a vertex v is reachable from another vertex u for all vertex pairs (u, v) in the given graph. Here reachable mean that there is a path from vertex u to v. The reach-ability matrix is called transitive closure of a graph.

Now, we can always solve this problem using Floyd Warshall. But, that will take O(V3).

However, this could be improved upto O(v2). The idea is given below:

**Create a matrix tc[V][V] that would finally have transitive closure of given graph.** Initialize all entries of tc[][] as 0.

Call DFS for every node of graph to mark reachable vertices in tc[][]. In recursive calls to DFS, we don’t call DFS for an adjacent vertex if it is already marked as reachable in tc[][].

Below is implementation of the above idea. The code uses adjacency list representation of input graph and builds a matrix tc[V][V] such that tc[u][v] would be true if v is reachable from u.

**// C++ program to print transitive closure of a graph**

**#include<bits/stdc++.h>**

**using namespace std;**

**class Graph**

**{**

**int V; // No. of vertices**

**bool \*\*tc; // To store transitive closure**

**list<int> \*adj; // array of adjacency lists**

**void DFSUtil(int u, int v);**

**public:**

**Graph(int V); // Constructor**

**// function to add an edge to graph**

**void addEdge(int v, int w) { adj[v].push\_back(w); }**

**// prints transitive closure matrix**

**void transitiveClosure();**

**};**

**Graph::Graph(int V)**

**{**

**this->V = V;**

**adj = new list<int>[V];**

**tc = new bool\* [V];**

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

**{**

**tc[i] = new bool[V];**

**memset(tc[i], false, V\*sizeof(bool));**

**}**

**}**

**// A recursive DFS traversal function that finds**

**// all reachable vertices for s.**

**void Graph::DFSUtil(int s, int v)**

**{**

**// Mark reachability from s to t as true.**

**tc[s][v] = true;**

**// Find all the vertices reachable through v**

**list<int>::iterator i;**

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

**if (tc[s][\*i] == false)**

**DFSUtil(s, \*i);**

**}**

**// The function to find transitive closure. It uses**

**// recursive DFSUtil()**

**void Graph::transitiveClosure()**

**{**

**// Call the recursive helper function to print DFS**

**// traversal starting from all vertices one by one**

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

**DFSUtil(i, i); // Every vertex is reachable from self.**

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

**{**

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

**cout << tc[i][j] << " ";**

**cout << endl;**

**}**

**}**

**// Driver code**

**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 << "Transitive closure matrix is \n";**

**g.transitiveClosure();**

**return 0;**

**}**

**Find k-cores of an undirected graph:**Given a graph G and an integer K, K-cores of the graph are connected components that are left after all vertices of degree less than k have been removed

The standard algorithm to find a k-core graph is to remove all the vertices that have degree less than- ‘K’ from the input graph. We must be careful that removing a vertex reduces the degree of all the vertices adjacent to it, hence the degree of adjacent vertices can also drop below-‘K’. And thus, we may have to remove those vertices also. This process may/may not go until there are no vertices left in the graph.

To implement above algorithm, we do a modified DFS on the input graph and delete all the vertices having degree less than ‘K’, then update degrees of all the adjacent vertices, and if their degree falls below ‘K’ we will delete them too.

**Time complexity of the above solution is O(V + E) where V is number of vertices and E is number of edges.**

**// C++ program to find K-Cores of a graph**

**#include<bits/stdc++.h>**

**using namespace std;**

**// This class represents a undirected 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 u, int v);**

**// A recursive function to print DFS starting from v**

**bool DFSUtil(int, vector<bool> &, vector<int> &, int k);**

**// prints k-Cores of given graph**

**void printKCores(int k);**

**};**

**// A recursive function to print DFS starting from v.**

**// It returns true of degree of v after processing is less**

**// than k else false**

**// It also updates degree of adjacent if degree of v**

**// is less than k. And if degree of a processed adjacent**

**// becomes less than k, then it reduces of degree of v also,**

**bool Graph::DFSUtil(int v, vector<bool> &visited,**

**vector<int> &vDegree, int k)**

**{**

**// 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)**

**{**

**// degree of v is less than k, then degree of adjacent**

**// must be reduced**

**if (vDegree[v] < k)**

**vDegree[\*i]--;**

**// If adjacent is not processed, process it**

**if (!visited[\*i])**

**{**

**// If degree of adjacent after processing becomes**

**// less than k, then reduce degree of v also.**

**if (DFSUtil(\*i, visited, vDegree, k))**

**vDegree[v]--;**

**}**

**//**This is done for a undirected graph.

**}**

**// Return true if degree of v is less than k**

**return (vDegree[v] < k);**

**}**

**Graph::Graph(int V)**

**{**

**this->V = V;**

**adj = new list<int>[V];**

**}**

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

**{**

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

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

**}**

**// Prints k cores of an undirected graph**

**void Graph::printKCores(int k)**

**{**

**// INITIALIZATION**

**// Mark all the vertices as not visited and not**

**// processed.**

**vector<bool> visited(V, false);**

**vector<bool> processed(V, false);**

**int mindeg = INT\_MAX;**

**int startvertex;**

**// Store degrees of all vertices**

**vector<int> vDegree(V);**

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

**{**

**vDegree[i] = adj[i].size();**

**if (vDegree[i] < mindeg)**

**{**

**mindeg = vDegree[i];**

**startvertex=i;**

**}**

**}**

Find the degree with minimum degree. That will be our source vertex

**DFSUtil(startvertex, visited, vDegree, k);**

**// DFS traversal to update degrees of all**

**// vertices.**

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

**if (visited[i] == false)**

**DFSUtil(i, visited, vDegree, k);**

**// PRINTING K CORES**

**cout << "K-Cores : \n";**

K is our user input

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

**{**

**// Only considering those vertices which have degree**

**// >= K after BFS**

**if (vDegree[v] >= k)**

**{**

**cout << "\n[" << v << "]";**

**// Traverse adjacency list of v and print only**

**// those adjacent which have vDegree >= k after**

**// BFS.**

**list<int>::iterator itr;**

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

**if (vDegree[\*itr] >= k)**

**cout << " -> " << \*itr;**

**}**

**}**

**}**

**// Driver program to test methods of graph class**

**int main()**

**{**

**// Create a graph given in the above diagram**

**int k = 3;**

**Graph g1(9);**

**g1.addEdge(0, 1);**

**g1.addEdge(0, 2);**

**g1.addEdge(1, 2);**

**g1.addEdge(1, 5);**

**g1.addEdge(2, 3);**

**g1.addEdge(2, 4);**

**g1.addEdge(2, 5);**

**g1.addEdge(2, 6);**

**g1.addEdge(3, 4);**

**g1.addEdge(3, 6);**

**g1.addEdge(3, 7);**

**g1.addEdge(4, 6);**

**g1.addEdge(4, 7);**

**g1.addEdge(5, 6);**

**g1.addEdge(5, 8);**

**g1.addEdge(6, 7);**

**g1.addEdge(6, 8);**

**g1.printKCores(k);**

**cout << endl << endl;**

**Graph g2(13);**

**g2.addEdge(0, 1);**

**g2.addEdge(0, 2);**

**g2.addEdge(0, 3);**

**g2.addEdge(1, 4);**

**g2.addEdge(1, 5);**

**g2.addEdge(1, 6);**

**g2.addEdge(2, 7);**

**g2.addEdge(2, 8);**

**g2.addEdge(2, 9);**

**g2.addEdge(3, 10);**

**g2.addEdge(3, 11);**

**g2.addEdge(3, 12);**

**g2.printKCores(k);**

**return 0;**

**}**

Notice, how K core is printed:

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

**{**

**// Only considering those vertices which have degree**

**// >= K after BFS**

**if (vDegree[v] >= k)**

**{**

**cout << "\n[" << v << "]";**

**// Traverse adjacency list of v and print only**

**// those adjacent which have vDegree >= k after**

**// BFS.**

**list<int>::iterator itr;**

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

**if (vDegree[\*itr] >= k)**

**cout << " -> " << \*itr;**

**}**

**}**

**Degeneracy :** Degeneracy of a graph is the largest value k such that the graph has a k-core

**Count the number of nodes at given level in a tree using BFS.**

I can surely do that.

(use queue in case of BFS. Use function stack (by recursive calling) or stack in case of DFS)

**Count all possible paths between two vertices**

Count the total number of ways or paths that exist between two vertices in a directed graph.

These paths doesn’t contain a cycle, the simple enough reason is that a cycle contain infinite number of paths and hence they create problem. **Hence, for a directed graph, even if the graph contains a cycle, we have to handle it with caution. So that, it cannot create a problem.**

**Solution: Backtracking:**

**// C++ program to count all**

**// paths from a source**

**// to a destination.**

**#include<bits/stdc++.h>**

**using namespace std;**

**// A directed graph using**

**// adjacency list**

**// representation**

**class Graph**

**{**

**// No. of vertices**

**// in graph**

**int V;**

**list<int> \*adj;**

**// A recursive function**

**// used by countPaths()**

**void countPathsUtil(int, int,**

**bool [],**

**int &);**

**public:**

**// Constructor**

**Graph(int V);**

**void addEdge(int u, int v);**

**int countPaths(int s, int d);**

**};**

**Graph::Graph(int V)**

**{**

**this->V = V;**

**adj = new list<int>[V];**

**}**

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

**{**

**// Add v to u’s list.**

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

**}**

**// Returns count of**

**// paths from 's' to 'd'**

**int Graph::countPaths(int s, int d)**

**{**

**// Mark all the vertices**

**// as not visited**

**bool \*visited = new bool[V];**

**memset(visited, false, sizeof(visited));**

**// Call the recursive helper**

**// function to print**

**// all paths**

**int pathCount = 0;**

**countPathsUtil(s, d, visited, pathCount);**

**return pathCount;**

**}**

**// A recursive function to**

**// print all paths from**

**// 'u' to 'd'. visited[]**

**// keeps track of vertices**

**// in current path. path[]**

**// stores actual vertices**

**// and path\_index is current**

**// index in path[]**

**void Graph::countPathsUtil(int u, int d,**

**bool visited[],**

**int &pathCount)**

**{**

**visited[u] = true;**

**// If current vertex is**

**// same as destination,**

**// then increment count**

**if (u == d)**

**pathCount++;**

**// If current vertex**

**// is not destination**

**else**

**{**

**// Recur for all the**

**// vertices adjacent to**

**// current vertex**

**list<int>::iterator i;**

**for (i = adj[u].begin(); i !=**

**adj[u].end(); ++i)**

**if (!visited[\*i])**

**countPathsUtil(\*i, d, visited,**

**pathCount);**

**}**

**visited[u] = false;**

**}**

**// Driver Code**

**int main()**

**{**

**// Create a graph given**

**// in the above diagram**

**Graph g(4);**

**g.addEdge(0, 1);**

**g.addEdge(0, 2);**

**g.addEdge(0, 3);**

**g.addEdge(2, 0);**

**g.addEdge(2, 1);**

**g.addEdge(1, 3);**

**int s = 2, d = 3;**

**cout << g.countPaths(s, d);**

**return 0;**

**}**

Now, this recursive function is simple:

**visited[u] = true;**

**// If current vertex is**

**// same as destination,**

**// then increment count**

**if (u == d)**

**pathCount++;**

**// If current vertex**

**// is not destination**

**else**

**{**

**// Recur for all the**

**// vertices adjacent to**

**// current vertex**

**list<int>::iterator i;**

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

**{**

**if (!visited[\*i])**

**{**

**countPathsUtil(\*i, d, visited,pathCount);**

**}**

**}**

**}**

**visited[u] = false;**

//Backtracking step

Now, even here, we prevent cycle visiting by:

**if (!visited[\*i])**

**{**

**countPathsUtil(\*i, d, visited,pathCount);**

**}**

This part increments count:

**if (u == d)**

**{**

**pathCount++;**

**}**

Now, note that, this counts all possible paths between two given nodes.

We find that using backtracking.

**Minimum initial vertices to traverse whole matrix with given conditions:**

We are given a matrix that contains different values in its each cell. Our aim is to find the minimal set of positions in the matrix such that entire matrix can be traversed starting from the positions in the set.

We can move only to those neighbors that contain value less than or to equal to the current cell’s value. A neighbor of cell is defined as the cell that shares a side with the given cell.

From the above examples, we can easily identify that in order to use minimum number of positions we have to start from the positions having highest cell value. Therefore we pick the positions that contain the highest value in the matrix. We take the vertices having highest value in separate array. We perform DFS on every vertex starting from the highest value. If we encounter any vertex which is not yet visited during dfs then we have to include this vertex in our set. When all the cells have been processed then the set contains the required vertices.

**How does this work?**

We need to visit all vertices and to reach largest values we must start with them. If two largest values are not adjacent, then both of them must be picked. If two largest values are adjacent, then any of them can be picked as moving to equal value neighbors is allowed.

// CPP program to find minimum initial

// vertices to reach whole matrix.

**#include <bits/stdc++.h>**

**using namespace std;**

**const int MAX = 100;**

// (n, m) is current source cell from which

// we need to do DFS. N and M are total no.

// of rows and columns.

**void dfs(int n, int m, bool visit[][MAX],**

**int adj[][MAX], int N, int M)**

**{**

**// Marking the vertex as visited**

**visit[n][m] = 1;**

**// If below neighbor is valid and has**

**// value less than or equal to current**

**// cell's value**

**if (n + 1 < N &&**

**adj[n][m] >= adj[n + 1][m] &&**

**!visit[n + 1][m])**

**dfs(n + 1, m, visit, adj, N, M);**

**// If right neighbor is valid and has**

**// value less than or equal to current**

**// cell's value**

**if (m + 1 < M &&**

**adj[n][m] >= adj[n][m + 1] &&**

**!visit[n][m + 1])**

**dfs(n, m + 1, visit, adj, N, M);**

**// If above neighbor is valid and has**

**// value less than or equal to current**

**// cell's value**

**if (n - 1 >= 0 &&**

**adj[n][m] >= adj[n - 1][m] &&**

**!visit[n - 1][m])**

**dfs(n - 1, m, visit, adj, N, M);**

**// If left neighbor is valid and has**

**// value less than or equal to current**

**// cell's value**

**if (m - 1 >= 0 &&**

**adj[n][m] >= adj[n][m - 1] &&**

**!visit[n][m - 1])**

**dfs(n, m - 1, visit, adj, N, M);**

**}**

**void printMinSources(int adj[][MAX], int N, int M)**

**{**

**// Storing the cell value and cell indices**

**// in a vector.**

**vector<pair<long int, pair<int, int> > > x;**

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

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

**x.push\_back(make\_pair(adj[i][j],**

**make\_pair(i, j)));**

**// Sorting the newly created array according**

**// to cell values**

**sort(x.begin(), x.end());**

**// Create a visited array for DFS and**

**// initialize it as false.**

**bool visit[N][MAX];**

**memset(visit, false, sizeof(visit));**

**// Applying dfs for each vertex with**

**// highest value**

**for (int i = x.size()-1; i >=0 ; i--)**

**{**

**// If the given vertex is not visited**

**// then include it in the set**

**if (!visit[x[i].second.first][x[i].second.second])**

**{**

**cout << x[i].second.first << " "**

**<< x[i].second.second << endl;**

**dfs(x[i].second.first, x[i].second.second,**

**visit, adj, N, M);**

**}**

**}**

**}**

**// Driver code**

**int main()**

**{**

**int N = 2, M = 2;**

**int adj[N][MAX] = {{3, 3},**

**{1, 1}};**

**printMinSources(adj, N, M);**

**return 0;**

**}**

Now, how the code is written? The code is written in such a manner, that it will first maintain a pair which contains adj[I][j]’s value as first variable , and pair(I,j) as second variable. After that, we sorted the list. **What’s the significance of the sorted list?**Now, we get the vertices in the decreasing order of adj[I][j] with proper i and j.

So, the cell with the highest cell value will be traversed first.

Now, from that list, we will pick up elements one by one and do the following:

We will first check if the node is visited or not. If it is visited, we will not proceed.

We will otherwise, include this as the final set of answer. And, traverse it’s adjacents using dfs.

**Note that: we will only add a vertex when we are traversing the pair whose first part is adj[i][j] and second part is (i,j).**

**Note: also we could sort the pair based container whose first part is adj[i][j] and second part is (i,j) normally. As, traverse it in reverse order.**

Now, consider the matrix:{{3, 3},

                       {1, 1}};

**Obviously, the resultant vertex will be** (0,1)

**Shortest path to reach one prime to other by changing single digit at a time:**

Given two four digit prime numbers, suppose 1033 and 8179, we need to find the shortest path from 1033 to 8179 by altering only single digit at a time such that every number that we get after changing a digit is prime. For example a solution is 1033, 1733, 3733, 3739, 3779, 8779, 8179

The question can be solved by BFS and it is a pretty interesting to solve as a starting problem for beginners. We first find out all 4 digit prime numbers till 9999 using technique of Sieve of Eratosthenes. And then using those numbers formed the graph using adjacency list. After forming the adjacency list, we used simple BFS to solve the problem.

**void SieveOfEratosthenes(vector<int>& v)**

**{**

**// Create a boolean array "prime[0..n]" and initialize**

**// all entries it as true. A value in prime[i] will**

**// finally be false if i is Not a prime, else true.**

**int n = 9999;**

**bool prime[n + 1];**

**memset(prime, true, sizeof(prime));**

**for (int p = 2; p \* p <= n; p++) {**

**// If prime[p] is not changed, then it is a prime**

**if (prime[p] == true) {**

**// Update all multiples of p**

**for (int i = p \* p; i <= n; i += p)**

**prime[i] = false;**

**}**

**}**

**// Forming a vector of prime numbers**

**for (int p = 1000; p <= n; p++)**

**if (prime[p])**

**v.push\_back(p);**

**// cout<<v.size();**

**}**

Now, this is sieve of Eratosthenes.

**class graph {**

**int V;**

**list<int>\* l;**

**public:**

**graph(int V)**

**{**

**this->V = V;**

**l = new list<int>[V];**

**}**

**void addedge(int V1, int V2)**

**{**

**l[V1].push\_back(V2);**

**l[V2].push\_back(V1);**

**}**

**int bfs(int in1, int in2);**

**};**

**// in1 and in2 are two vertices of graph which are**

**// actually indexes in pset[]**

**int graph::bfs(int in1, int in2)**

**{**

**int visited[V];**

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

**queue<int> que;**

**visited[in1] = 1;**

**que.push(in1);**

**list<int>::iterator i;**

**int f = 0;**

**while (!que.empty()) {**

**int p = que.front();**

**que.pop();**

**for (i = l[p].begin(); i != l[p].end(); i++) {**

**if (!visited[\*i]) {**

**visited[\*i] = visited[p] + 1;**

**que.push(\*i);**

**}**

**if (\*i == in2) {**

**return visited[\*i] - 1;**

**}**

**}**

**}**

**}**

**// Returns true if num1 and num2 differ by single**

**// digit.**

**bool compare(int num1, int num2)**

**{**

**// To compare the digits**

**string s1 = to\_string(num1);**

**string s2 = to\_string(num2);**

**int c = 0;**

**if (s1[0] != s2[0])**

**c++;**

**if (s1[1] != s2[1])**

**c++;**

**if (s1[2] != s2[2])**

**c++;**

**if (s1[3] != s2[3])**

**c++;**

**// If the numbers differ only by a single**

**// digit return true else false**

**return (c == 1);**

**}**

**int shortestPath(int num1, int num2)**

**{**

**// Generate all 4 digit**

**vector<int> pset;**

**SieveOfEratosthenes(pset);**

**// Create a graph where node numbers are indexes**

**// in pset[] and there is an edge between two**

**// nodes only if they differ by single digit.**

**graph g(pset.size());**

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

**for (int j = i + 1; j < pset.size(); j++)**

**if (compare(pset[i], pset[j]))**

**g.addedge(i, j);**

**// Since graph nodes represent indexes of numbers**

**// in pset[], we find indexes of num1 and num2.**

**int in1, in2;**

**for (int j = 0; j < pset.size(); j++)**

**if (pset[j] == num1)**

**in1 = j;**

**for (int j = 0; j < pset.size(); j++)**

**if (pset[j] == num2)**

**in2 = j;**

**return g.bfs(in1, in2);**

**}**

**// Driver code**

**int main()**

**{**

**int num1 = 1033, num2 = 8179;**

**cout << shortestPath(num1, num2);**

**return 0;**

**}**

Now, the code is pretty simple. We first generate all prime numbers upto the upper limit. Then, we check if a prime number can be achieved by changing one digit of another prime, If it is, we add an edge between the two in the graph.

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

**for (int j = i + 1; j < pset.size(); j++)**

**if (compare(pset[i], pset[j]))**

**g.addedge(i, j);**

Now, we will do BFS between the first and last prime.

**Now,** note that point, the shortest path between two nodes might not be found using DFS, So, we should always do BFS in that case. **(you can surely thought of at least some directed graph example)**

**Water Jug problem using BFS:**

You are given a m litre jug and a n litre 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 litres of water where d is less than n.

(X, Y) corresponds to a state where X refers to amount of water in Jug1 and Y refers to amount of water in Jug2.

Determine the path from initial state (xi, yi) to 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).

**So, the operations which are allowed pragmatically:**

Empty a Jug, (X, Y)->(0, Y) Empty Jug 1

Fill a Jug, (0, 0)->(X, 0) Fill Jug 1

Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

(d can be any valid amount)

**#include <bits/stdc++.h>**

**#define pii pair<int, int>**

**#define mp make\_pair**

**using namespace std;**

**void BFS(int a, int b, int target)**

**{**

**// Map is used to store the states, every**

**// state is hashed to binary value to**

**// indicate either that state is visited**

**// before or not**

**map<pii, int> m;**

**bool isSolvable = false;**

**vector<pii> path;**

**queue<pii> q; // queue to maintain states**

**q.push({ 0, 0 }); // Initialing with initial state**

**while (!q.empty()) {**

**pii u = q.front(); // current state**

**q.pop(); // pop off used state**

**// if this state is already visited**

**if (m[{ u.first, u.second }] == 1)**

**continue;**

**// doesn't met jug constraints**

**if ((u.first > a || u.second > b ||**

**u.first < 0 || u.second < 0))**

**continue;**

**// filling the vector for constructing**

**// the solution path**

**path.push\_back({ u.first, u.second });**

**// marking current state as visited**

**m[{ u.first, u.second }] = 1;**

**// if we reach solution state, put ans=1**

**if (u.first == target || u.second == target) {**

**isSolvable = true;**

**if (u.first == target) {**

**if (u.second != 0)**

**// fill final state**

**path.push\_back({ u.first, 0 });**

**}**

**else {**

**if (u.first != 0)**

**// fill final state**

**path.push\_back({ 0, u.second });**

**}**

**// print the solution path**

**int sz = path.size();**

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

**cout << "(" << path[i].first**

**<< ", " << path[i].second << ")\n";**

**break;**

**}**

**// if we have not reached final state**

**// then, start developing intermediate**

**// states to reach solution state**

**q.push({ u.first, b }); // fill Jug2**

**q.push({ a, u.second }); // fill Jug1**

**for (int ap = 0; ap <= max(a, b); ap++) {**

**// pour amount ap from Jug2 to Jug1**

**int c = u.first + ap;**

**int d = u.second - ap;**

**// check if this state is possible or not**

**if (c == a || (d == 0 && d >= 0))**

**q.push({ c, d });**

**// Pour amount ap from Jug 1 to Jug2**

**c = u.first - ap;**

**d = u.second + ap;**

**// check if this state is possible or not**

**if ((c == 0 && c >= 0) || d == b)**

**q.push({ c, d });**

**}**

**q.push({ a, 0 }); // Empty Jug2**

**q.push({ 0, b }); // Empty Jug1**

**}**

**// No, solution exists if ans=0**

**if (!isSolvable)**

**cout << "No solution";**

**}**

**// Driver code**

**int main()**

**{**

**int Jug1 = 4, Jug2 = 3, target = 2;**

**cout << "Path from initial state "**

**"to solution state :\n";**

**BFS(Jug1, Jug2, target);**

**return 0;**

**}**

Now, check the BFS function:

**void BFS(int a, int b, int target)**

**{**

Map is used to store states, every state is hashed to a binary value (1 for visited, 0 for not visited)

**map<pii, int> m;**

Initially isSolvable is set as false

**bool isSolvable = false;**

This path will contain the path (traversed by BFS)

**vector<pii> path;**

Since, BFS, a queue is needed to be maintained

This queue will contain the current state as a pair of (first jug value, second jug value)

**queue<pii> q; // queue to maintain states**

Initially (0,0) is being pushed to the queue. Indicating that, both jugs are empty at thee initial state

**q.push({ 0, 0 }); // Initialing with initial state**

**while (!q.empty()) {**

**pii u = q.front(); // current state**

**q.pop(); // pop off used state**

if this state is already visited, we will skip. Otherwise, it will cause an infinite loop

**if (m[{ u.first, u.second }] == 1)**

**continue;**

**// doesn't met jug constraints**

If jug constraints are not met, then we will skip.

**if ((u.first > a || u.second > b ||**

**u.first < 0 || u.second < 0))**

**continue;**

**// filling the vector for constructing**

**// the solution path**

Now, notice that, if it is not already visited, or it follows the jug constraints, it is being pushed to the solution path . So, the **front element of the queue** is pushed into the solution path

**path.push\_back({ u.first, u.second });**

**// marking current state as visited**

After it is pushed in the solution path (vector), we will visit adjacent nodes. From here, what could be the adjacent nodes?

What could be the adjacent nodes?

Suppose, the current state is x,y

I.e. the fist jug is carrying x litres of water and second jug is carrying y litres of water.

Now, Suppose, the limit of first jug is m litres and second jug is n litres

So, one step is (x,n) and another state is (m,y)  
 Another state is we are pouring water from one jug to another jug.

And, empty jugs are are also states (0,n) and (m,0)

**m[{ u.first, u.second }] = 1;**

**// if we reach solution state, put ans=1**

**if (u.first == target || u.second == target) {**

**isSolvable = true;**

**if (u.first == target) {**

**if (u.second != 0)**

**//Now, whether the second container is fully empty or not, will not be our concern**

**// fill final state**

**path.push\_back({ u.first, 0 });**

**}**

**else {**

**//that means u.second reaches the target**

**//Now, whether the first container is fully empty or not, will not be our concern**

**if (u.first != 0)**

**// fill final state**

**path.push\_back({ 0, u.second });**

**}**

**//since, we already reach the solution. We would like to print it.**

**// print the solution path**

**int sz = path.size();**

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

**cout << "(" << path[i].first**

**<< ", " << path[i].second << ")\n";**

**break;**

**}**

**// if we have not reached final state**

**// then, start developing intermediate**

**// states to reach solution state**

**q.push({ u.first, b }); // fill Jug2**

**q.push({ a, u.second }); // fill Jug1**

**for (int ap = 0; ap <= max(a, b); ap++) {**

**// pour amount ap from Jug2 to Jug1**

**int c = u.first + ap;**

**int d = u.second - ap;**

**// check if this state is possible or not**

**if (c == a || (d == 0 && d >= 0))**

**q.push({ c, d });**

**// Pour amount ap from Jug 1 to Jug2**

**c = u.first - ap;**

**d = u.second + ap;**

**// check if this state is possible or not**

**if ((c == 0 && c >= 0) || d == b)**

**q.push({ c, d });**

**}**

**q.push({ a, 0 }); // Empty Jug2**

**q.push({ 0, b }); // Empty Jug1**

**}**

**// No, solution exists if ans=0**

**if (!isSolvable)**

**cout << "No solution";**

**}**

Now, we already mentioned the state:

Suppose, the current state is x,y

I.e. the fist jug is carrying x litres of water and second jug is carrying y litres of water.

Now, Suppose, the limit of first jug is m litres and second jug is n litres

So, one step is (x,n) and another state is (m,y)  
Another state is we are pouring water from one jug to another jug.

And, empty jugs are are also states (0,n) and (m,0)

First cases, filling either of the jugs:  
  
**q.push({ u.first, b }); // fill Jug2**

**q.push({ a, u.second }); // fill Jug1**

Second case, we are pouring water from one jug to another jug.

**for (int ap = 0; ap <= max(a, b); ap++)**

**{**

**// pour amount ap from Jug2 to Jug1**

**int c = u.first + ap;**

**int d = u.second - ap;**

**// check if this state is possible or not**

**if (c == a || (d == 0 && d >= 0))**

**q.push({ c, d });**

**// Pour amount ap from Jug 1 to Jug2**

**c = u.first - ap;**

**d = u.second + ap;**

**// check if this state is possible or not**

**if ((c == 0 && c >= 0) || d == b)**

**q.push({ c, d });**

**}**

Third cases, empty jugs are are also states (0,n) and (m,0)

**q.push({ a, 0 }); // Empty Jug2**

**q.push({ 0, b }); // Empty Jug1**

**Count number of trees in a forest:**

1. Apply DFS on every node.

2. Increment count by one if every connected node is visited from one source.

3. Again perform DFS traversal if some nodes yet not visited.

4. Count will give the number of trees in forest.

**// CPP program to count number of trees in**

**// a forest.**

**#include<bits/stdc++.h>**

**using namespace std;**

**// A utility function to add an edge in an**

**// undirected graph.**

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

**{**

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

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

**}**

**// A utility function to do DFS of graph**

**// recursively from a given vertex u.**

**void DFSUtil(int u, vector<int> adj[],**

**vector<bool> &visited)**

**{**

**visited[u] = true;**

**for (int i=0; i<adj[u].size(); i++)**

**if (visited[adj[u][i]] == false)**

**DFSUtil(adj[u][i], adj, visited);**

**}**

**// Returns count of tree is the forest**

**// given as adjacency list.**

**int countTrees(vector<int> adj[], int V)**

**{**

**vector<bool> visited(V, false);**

**int res = 0;**

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

**{**

**if (visited[u] == false)**

**{**

**DFSUtil(u, adj, visited);**

**res++;**

**//here it is incremented**

**//that is how we got count of trees in a forest**

**}**

**}**

**return res;**

**}**

**// Driver code**

**int main()**

**{**

**int V = 5;**

**vector<int> adj[V];**

**addEdge(adj, 0, 1);**

**addEdge(adj, 0, 2);**

**addEdge(adj, 3, 4);**

**cout << countTrees(adj, V);**

**return 0;**

**}**

**Boggle (Find all possible words in a board of characters)**It can be solved using plain DFS.  
  
It can be solved using DFS combined with trie.

**Minimum Time To Rot All Oranges:**

Given a matrix of dimension m\*n where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:

**0: Empty cell**

**1: Cells have fresh oranges**

**2: Cells have rotten oranges**

So we have to determine what is the minimum time required so that all the oranges become rotten. A rotten orange at index [i,j] can rot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (up, down, left and right). If it is impossible to rot every orange then simply return -1.

1) Create an empty Q.

2) Find all rotten oranges and enqueue them to Q. Also enqueue

a delimiter to indicate beginning of next time frame.

3) While Q is not empty do following

3.a) While delimiter in Q is not reached

(i) Dequeue an orange from queue, rot all adjacent oranges.

While rotting the adjacents, make sure that time frame

is incremented only once. And time frame is not incremented

if there are no adjacent oranges.

3.b) Dequeue the old delimiter and enqueue a new delimiter. The

oranges rotten in previous time frame lie between the two

delimiters.

// C++ program to find minimum time required to make all

// oranges rotten

**#include<bits/stdc++.h>**

**#define R 3**

**#define C 5**

**using namespace std;**

function to check whether a cell is valid / invalid

**bool isvalid(int i, int j)**

**{**

**return (i >= 0 && j >= 0 && i < R && j < C);**

**}**

structure for storing coordinates of the cell

**struct ele**

**{**

**int x, y;**

**};**

Function to check whether the cell is delimiter which is (-1, -1). Defined by us

**bool isdelim(ele temp)**

**{**

**return (temp.x == -1 && temp.y == -1);**

**}**

Function to check whether there is still a fresh orange remaining

bool checkall(int arr[][C])

{

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

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

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

return true;

return false;

}

**// This function finds if it is possible to rot all oranges or not.**

**// If possible, then it returns minimum time required to rot all,**

**// otherwise returns -1**

**int rotOranges(int arr[][C])**

**{**

**// Create a queue of cells**

We are solving this problem with BFS. Hence, queue is important. This queue will contain the rotten oranges position

**queue<ele> Q;**

**ele temp;**

**int ans = 0;**

**// Store all the cells having rotten orange in first time frame**

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

**{**

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

**{**

**if (arr[i][j] == 2)**

**{**

**temp.x = i;**

**temp.y = j;**

**Q.push(temp);**

**}**

**}**

**}**

Separate these rotten oranges from the oranges which will rotten

due the oranges in first time frame using delimiter which is (-1, -1)

**temp.x = -1;**

**temp.y = -1;**

**Q.push(temp);**

Process the grid while there are rotten oranges in the Queue

**while (!Q.empty())**

**{**

This flag is used to determine whether even a single fresh orange gets rotten due to rotten oranges in current time frame so we can increase the count of the required time.

Because, if we did not find a single orange rotten in the current time frame, we will neither increment the time frame count or nor inserting a new delimiter to the queue.

**bool flag = false;**

**// Process all the rotten oranges in current time frame.**

**while (!isdelim(Q.front()))**

**{**

**temp = Q.front();**

**// Check right adjacent cell that if it can be rotten**

**if (isvalid(temp.x+1, temp.y) && arr[temp.x+1][temp.y] == 1)**

**{**

**// if this is the first orange to get rotten, increase**

**// count and set the flag.**

**if (!flag) ans++, flag = true;**

**// Make the orange rotten**

**arr[temp.x+1][temp.y] = 2;**

**// push the adjacent orange to Queue**

**temp.x++;**

**Q.push(temp);**

**temp.x--; // Move back to current cell**

**}**

**// Check left adjacent cell that if it can be rotten**

**if (isvalid(temp.x-1, temp.y) && arr[temp.x-1][temp.y] == 1) {**

**if (!flag) ans++, flag = true;**

**arr[temp.x-1][temp.y] = 2;**

**temp.x--;**

**Q.push(temp); // push this cell to Queue**

**temp.x++;**

**}**

**// Check top adjacent cell that if it can be rotten**

**if (isvalid(temp.x, temp.y+1) && arr[temp.x][temp.y+1] == 1) {**

**if (!flag) ans++, flag = true;**

**arr[temp.x][temp.y+1] = 2;**

**temp.y++;**

**Q.push(temp); // Push this cell to Queue**

**temp.y--;**

**}**

**// Check bottom adjacent cell if it can be rotten**

**if (isvalid(temp.x, temp.y-1) && arr[temp.x][temp.y-1] == 1) {**

**if (!flag) ans++, flag = true;**

**arr[temp.x][temp.y-1] = 2;**

**temp.y--;**

**Q.push(temp); // push this cell to Queue**

**}**

**Q.pop();**

**}**

**// Pop the delimiter**

**Q.pop();**

**// If oranges were rotten in current frame than separate the**

**// rotten oranges using delimiter for the next frame for processing.**

**if (!Q.empty()) {**

**temp.x = -1;**

**temp.y = -1;**

**Q.push(temp);**

**}**

**// If Queue was empty than no rotten oranges left to process so exit**

**}**

**// Return -1 if all arranges could not rot, otherwise -1.**

**return (checkall(arr))? -1: ans;**

**}**

**// Drive program**

**int main()**

**{**

**int arr[][C] = { {2, 1, 0, 2, 1},**

**{1, 0, 1, 2, 1},**

**{1, 0, 0, 2, 1}};**

**int ans = rotOranges(arr);**

**if (ans == -1)**

**cout << "All oranges cannot rot\n";**

**else**

**cout << "Time required for all oranges to rot => " << ans << endl;**

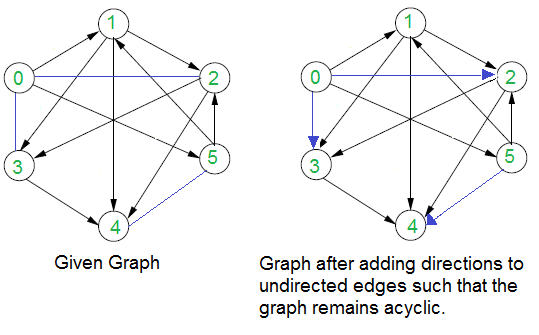
**return 0;**

**}**

**Assign directions to edges so that the directed graph remains acyclic:**

Given a graph with both directed and undirected edges. It is given that the directed edges don’t form cycle. How to assign directions to undirected edges so that the graph (with all directed edges) remains acyclic even after the assignment?

For example, in the below graph, blue edges don’t have directions.



The idea is to use Topological Sorting. Following are two steps used in the algorithm.

1) Consider the subgraph with directed edges only and find topological sorting of the subgraph. In the above example, topological sorting is {0, 5, 1, 2, 3, 4}. Below diagram shows topological sorting for the above example graph.

1. Use above topological sorting to assign directions to undirected edges. For every undirected edge (u, v), assign it direction from u to v if u comes before v in topological sorting, else assign it direction from v to u. Below diagram shows assigned directions in the example graph.

**Topological Sorting**

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).

We recommend to first see implementation of DFS here. We can modify DFS to find Topological Sorting of a graph. In DFS, 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 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 stack.

**// A C++ program to print topological sorting of a DAG**

**#include<iostream>**

**#include <list>**

**#include <stack>**

**using namespace std;**

**// Class to represent a graph**

**class Graph**

**{**

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

**// Pointer to an array containing adjacency listsList**

**list<int> \*adj;**

**// A function used by topologicalSort**

**void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);**

**public:**

**Graph(int V); // Constructor**

**// function to add an edge to graph**

**void addEdge(int v, int w);**

**// prints a Topological Sort of the complete graph**

**void topologicalSort();**

**};**

**Graph::Graph(int V)**

**{**

**this->V = V;**

**adj = new list<int>[V];**

**}**

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

**{**

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

**}**

**// A recursive function used by topologicalSort**

**void Graph::topologicalSortUtil(int v, bool visited[],**

**stack<int> &Stack)**

**{**

**// Mark the current node as visited.**

**visited[v] = true;**

**// Recur for all the vertices adjacent to this vertex**

**list<int>::iterator i;**

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

**if (!visited[\*i])**

**topologicalSortUtil(\*i, visited, Stack);**

**// Push current vertex to stack which stores result**

**Stack.push(v);**

**}**

**// The function to do Topological Sort. It uses recursive**

**// topologicalSortUtil()**

**void Graph::topologicalSort()**

**{**

**stack<int> Stack;**

**// Mark all the vertices as not visited**

**bool \*visited = new bool[V];**

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

**visited[i] = false;**

**// Call the recursive helper function to store Topological**

**// Sort starting from all vertices one by one**

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

**if (visited[i] == false)**

**topologicalSortUtil(i, visited, Stack);**

**// Print contents of stack**

**while (Stack.empty() == false)**

**{**

**cout << Stack.top() << " ";**

**Stack.pop();**

**}**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**// Create a graph given in the above diagram**

**Graph g(6);**

**g.addEdge(5, 2);**

**g.addEdge(5, 0);**

**g.addEdge(4, 0);**

**g.addEdge(4, 1);**

**g.addEdge(2, 3);**

**g.addEdge(3, 1);**

**cout << "Following is a Topological Sort of the given graph n";**

**g.topologicalSort();**

**return 0;**

**}**

**All Topological Sorts of a Directed Acyclic Graph:**

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

Given a DAG, print all topological sorts of the graph.

In a Directed acyclic graph many a times we can have vertices which are unrelated to each other because of which we can order them in many ways. These various topological sorting is important in many cases, for example if some relative weight is also available between the vertices, which is to minimize then we need to take care of relative ordering as well as their relative weight, which creates the need of checking through all possible topological ordering.

We can go through all possible ordering via backtracking , the algorithm step are as follows :

* Initialize all vertices as unvisited.
* Now choose vertex which is unvisited and has zero indegree and decrease indegree of all those vertices by 1 (corresponding to removing edges) which are adjacent now add this vertex to result and call the recursive function again and backtrack.
* After returning from function reset values of visited, result and indegree for enumeration of other possibilities.

**// C++ program to print all topological sorts of a graph**

**#include <bits/stdc++.h>**

**using namespace std;**

**class Graph**

**{**

**int V; // No. of vertices**

**// Pointer to an array containing adjacency list**

**list<int> \*adj;**

**// Vector to store indegree of vertices**

**vector<int> indegree;**

**// A function used by alltopologicalSort**

**void alltopologicalSortUtil(vector<int>& res,**

**bool visited[]);**

**public:**

**Graph(int V); // Constructor**

**// function to add an edge to graph**

**void addEdge(int v, int w);**

**// Prints all Topological Sorts**

**void alltopologicalSort();**

**};**

**// Constructor of graph**

**Graph::Graph(int V)**

**{**

**this->V = V;**

**adj = new list<int>[V];**

**// Initialising all indegree with 0**

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

**indegree.push\_back(0);**

**}**

**// Utility function to add edge**

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

**{**

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

**// increasing inner degree of w by 1**

**indegree[w]++;**

**}**

**// Main recursive function to print all possible**

**// topological sorts**

**void Graph::alltopologicalSortUtil(vector<int>& res,**

**bool visited[])**

**{**

**// To indicate whether all topological are found**

**// or not**

**bool flag = false;**

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

**{**

**// If indegree is 0 and not yet visited then**

**// only choose that vertex**

**if (indegree[i] == 0 && !visited[i])**

**{**

**// reducing indegree of adjacent vertices**

**list<int>:: iterator j;**

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

**indegree[\*j]--;**

**// including in result**

**res.push\_back(i);**

**visited[i] = true;**

Make the current node as visited

**alltopologicalSortUtil(res, visited);**

**// resetting visited, res and indegree for**

**// backtracking**

**visited[i] = false;**

**res.erase(res.end() - 1);**

Delete the current node form the result

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

**indegree[\*j]++;**

Again, for all the adjacent vertices of current vertex, indegree should be incremented by 1 as part of the backtrack

**flag = true;**

**}**

**}**

**// We reach here if all vertices are visited.**

**// So we print the solution here**

**if (!flag)**

**{**

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

**cout << res[i] << " ";**

**cout << endl;**

**}**

**}**

**// The function does all Topological Sort.**

**// It uses recursive alltopologicalSortUtil()**

**void Graph::alltopologicalSort()**

**{**

**// Mark all the vertices as not visited**

**bool \*visited = new bool[V];**

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

**visited[i] = false;**

**vector<int> res;**

**alltopologicalSortUtil(res, visited);**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**// Create a graph given in the above diagram**

**Graph g(6);**

**g.addEdge(5, 2);**

**g.addEdge(5, 0);**

**g.addEdge(4, 0);**

**g.addEdge(4, 1);**

**g.addEdge(2, 3);**

**g.addEdge(3, 1);**

**cout << "All Topological sorts\n";**

**g.alltopologicalSort();**

**return 0;**

**}**

**Kahn’s algorithm for Topological Sorting**

A DAG G has at least one vertex with in-degree 0 and one vertex with out-degree 0.

Proof: There’s a simple proof to the above fact is that a DAG does not contain a cycle which means that all paths will be of finite length. Now let S be the longest path from u(source) to v(destination). Since S is the longest path there can be no incoming edge to u and no outgoing edge from v, if this situation had occurred then S would not have been the longest path

=> indegree(u) = 0 and outdegree(v) = 0

**Algorithm:**

Steps involved in finding the topological ordering of a DAG:

**Step-1:** Compute in-degree (number of incoming edges) for each of the vertex present in the DAG 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.

**How to calculate the indegree of a 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).

**How to Prove A graph is A Tree:**

A graph is a tree under the following condition:

* The graph will not contain any cycle
* Every node is connected with other node ( There is difference between connected graph and complete graph. Here, we are looking for connected graph.

**How To Check For Cycle:**

We can either use BFS or DFS. For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle (See Detect cycle in an undirected graph for more details).

**How to check for connectivity?**

Since the graph is undirected, we can start BFS or DFS from any vertex and check if all vertices are reachable or not. If all vertices are reachable, then graph is connected, otherwise not.

(must be undirected. Directed graph has a different way of dtecting connectivvity)

(from any vertex, not from every vertex)