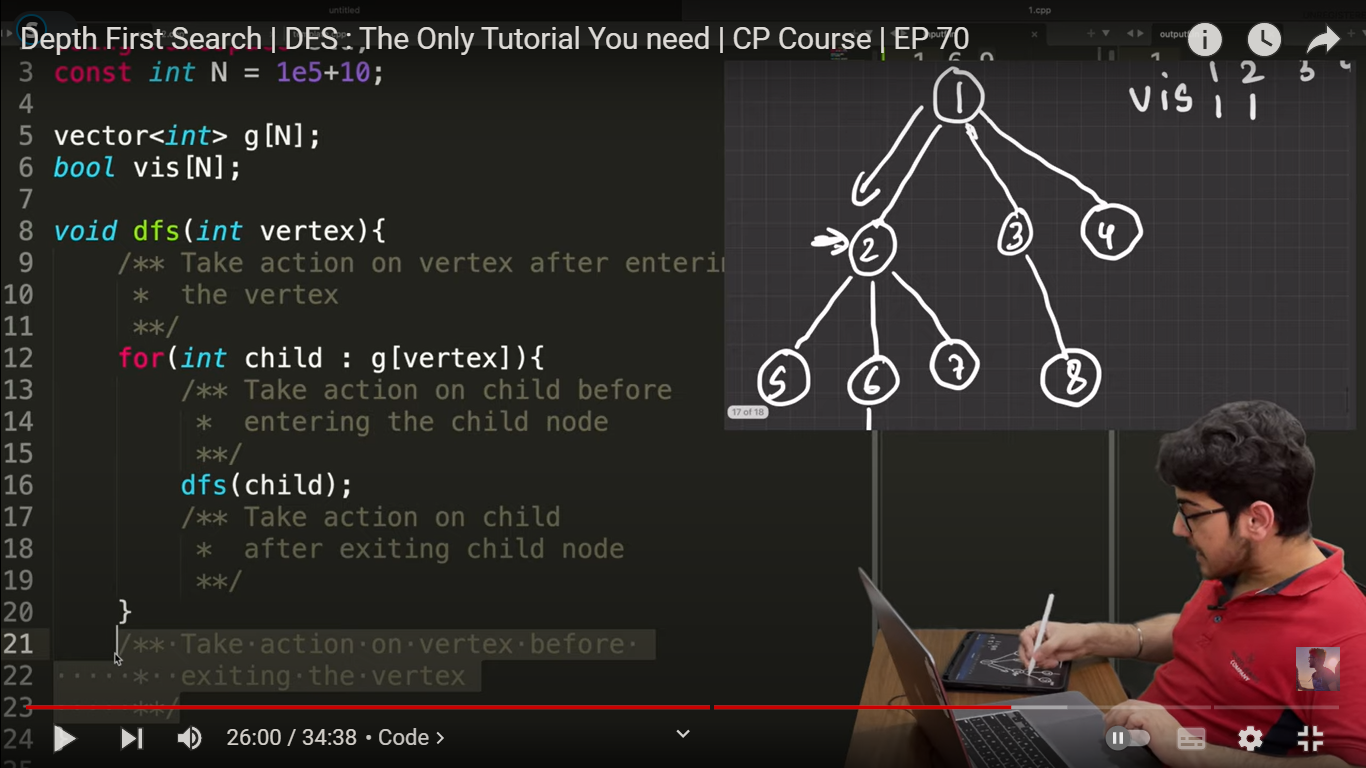
1> DFS: Main Code Sections Of DFS



//Take Action After Exiting from //Every child Node for given vertex.

2> Topological Sort Using DFS:

This algo used in OS for ordering jobs which are dependent on other process to complete.

In DFS toposort we make User of Stack to find out process for which no other process dependent on it. And push inside stack. So highly dependent process goes to lowest bottom of stack. And process with zero dependency comes on top. This works only on (DAG Directed Acyclic Graph). Cause if graph is not directed then there is no dependency.

Time = O (V+E) ; Space = O(V)

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

    {

        // code here

        stack<long> st;

        vector<bool> vis(V,false);

        for(long i=0; i<V; i++){

            if(!vis[i]){

                dfs(i, adj, st, vis);

            }

        }

        vector<int> ans(V);

        for(auto &i:ans){

            i = st.top();

            st.pop();

        }

        return ans;

    }

    void dfs(int v, vector<int> adj[], stack<long> &st, vector<bool> &vis){

        vis[v] = true;

        for(int &node:adj[v]){

            if(!vis[node])

            dfs(node, adj, st, vis);

        }

        st.push(v);

    }

3> Topological Sort Using BFS (Kahn’s Algo):

In BFS toposort we find out dependency for each and every process (i.e., indegree of every node and then process with zero indegree is pushed into Q. And process marked as completed. Now as this process is completed we other process which were dependent on first process we will reduce their dependency by 1. And if it becomes 0 we’ll push inside Q.

Time = O (V+E) ; Space = O(V)

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

    {

        vector<int> inDeg(V);

        for(int i=0; i<V; i++){

            for(int &ele:adj[i])

                inDeg[ele]++;

        }

        vector<int> ans;

        bfs(adj, V, ans, inDeg);

        return ans;

    }

    void bfs(vector<int> adj[], int N, vector<int>&ans, vector<int>&inDeg)

    {

            queue<int> q;

            for(int i=0; i<N; i++){

                if(inDeg[i]==0)

                    q.push(i);

            }

            while(!q.empty()){

                int ele = q.front();

                ans.push\_back(ele);

                q.pop();

                for(int &i:adj[ele]){

                    inDeg[i]--;

                    if(inDeg[i]==0)

                        q.push(i);

                }

            }

    }

4> Dijkstra’s Algorithm: Algorithm to find out shortest path for all nodes from a given single source. This does not work for negative weighted graphs. This makes use of priority queue (Min Heap). Or Set can also be used. Here “dist” is output vector.

The Idea behind this is to keep pushing inside Q until we find out smaller and smaller distances. If we find out smaller we’ll update all node belonging to that vertex.

[**Network Delay Problem**](https://leetcode.com/problems/network-delay-time/)

*set*<*pair*<*int*, *int*>> st;

*vector*<*int*> dist(*n*, iMax);

        st.insert({0, *k*});

        dist[*k*] = 0;

        while(!st.empty())

        {

*auto* vertexPair = \*st.begin();

*int* vertexDist = vertexPair.first;

*int* vertex = vertexPair.second;

            st.erase(st.begin());

            for(*auto* &nodePair:gList[vertex])

            {

*int* node = nodePair.first;

*int* nodeDist = nodePair.second;

                if((dist[vertex]+nodeDist) < dist[node]){

                    dist[node] = dist[vertex]+nodeDist;

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

                }

/\*The Idea behind this is to keep pushing inside Q until we find out smaller and smaller distances. If we find out smaller we’ll update all node belonging to that vertex \*/

            }

        }

Using Priority Queue:

*int* networkDelayTime(*vector*<*vector*<*int*>>& *times*, *int* *n*, *int* *k*)

    {

*vector*<*vector*<*pair*<*int*, *int*>>> gList(*n*+1);

        for(*auto* v:*times*){

            gList[v[0]].push\_back({v[1], v[2]});

        }

        //Greater<> always creates a min Heap & less max-heap

*priority\_queue*<*pair*<*int*, *int*>, *vector*<*pair*<*int*, *int*>> , *greater*<*pair*<*int*, *int*>>> q;

*vector*<*int*> dist(*n*+1, iMax);

        q.push({0, *k*});

        dist[*k*] = 0;

        while(!q.empty())

        {

*auto* vertexPair = q.top();

*int* vertexDist = vertexPair.first;

*int* vertex = vertexPair.second;

            q.pop();

            for(*auto* &nodePair:gList[vertex])

            {

*int* node = nodePair.first;

*int* nodeDist = nodePair.second;

                if((dist[vertex] + nodeDist) < dist[node]){

                    dist[node] = dist[vertex] + nodeDist;

                    q.push({dist[node], node});

                }

/\*The Idea behind this is to keep pushing inside Q until we find out smaller and smaller distances. If we find out smaller, we’ll update all node belonging to that vertex \*/

            }

        }

*int* ans = 0;

        for(*int* i=1; i<=*n*; i++){

            if(dist[i] == iMax)

                return -1;

            ans = max(ans, dist[i]);

        }

        return ans;

    }

5> Cycle Detection Directed Graph (Using DFS):

In Directed graph to detect a cycle we consider visited[] and recursion Stack[] to check cycle.

Corner case:

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

    {

*vector*<*bool*> vis(*V*,false), recStack(*V*,false);

        for(*int* i=0; i<*V*; i++)

        {

            if(!vis[i]){

                if(DFS(*adj*, i, vis, recStack)) return true;

            }

        }

        return false;

    }

*bool* DFS(*vector*<*int*> *adj*[], *int* &*vertex*, *vector*<*bool*>&*vis*, *vector*<*bool*>&*recStack*)

    {

*vis*[*vertex*] = 1;

*recStack*[*vertex*] = 1;

*bool* ans = false;

        for(*auto* node:*adj*[*vertex*]){

            if(!*vis*[node]){

                ans |= DFS(*adj*, node, *vis*, *recStack*);

            }

            else if(*recStack*[node]) return true;

        }

*recStack*[*vertex*] = 0;

        return ans;

    }

6> Cycle Detection Undirected Graph (Using DFS):

Here, We make use of visited[] and parent to identify from where node is traversed.

Corner case:

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

    {

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

        for(*int* i=0; i<*V*; i++){

            if(!vis[i]) {

                if(DFS(i, *adj*, vis, -1))

                    return true;

            }

        }

        return false;

    }

*bool* DFS(*int* *vertex*, *vector*<*int*> *adj*[], *vector*<*bool*> &*vis*, *int* *parent*)

    {

*vis*[*vertex*] = 1;

        for(*auto* node:*adj*[*vertex*])

        {

            if(!*vis*[node]){

                if(DFS(node, *adj*, *vis*, *vertex*))

                    return true;

            }

            else if(*parent* != node) return true;

        }

        return false;

    }

7> Bipartite Graph (Using DFS):

The Graph that can be colored using exactly 2 colors provided that no 2 adjacent nodes have same color then it is called as bipartite graph.

*bool* isBipartite(*int* *V*, *vector*<*int*>*adj*[]){

*vector*<*int*> vis(*V*), col(*V*);

         for(*int* i=0; i<*V*; i++){

             if(!vis[i]){

                 if(DFS(i, *adj*, vis, col, -1)) return false;

             }

         }

         return true;

    }

*bool* DFS(*int* &*vertex*, *vector*<*int*> *adj*[], *vector*<*int*>&*vis*, *vector*<*int*>&*col*, *int* *vcol*)

    {

*vis*[*vertex*] = 1;

*col*[*vertex*] = *vcol*\*-1;

*bool* ans = false;

        for(*auto* node:*adj*[*vertex*]){

            if(!*vis*[node]){

                ans |= DFS(node, *adj*, *vis*, *col*, *col*[*vertex*]);

            }

            else if(*col*[node] == *col*[*vertex*]) return true;

        }

        return ans;

    }