**GRAPH THORY**

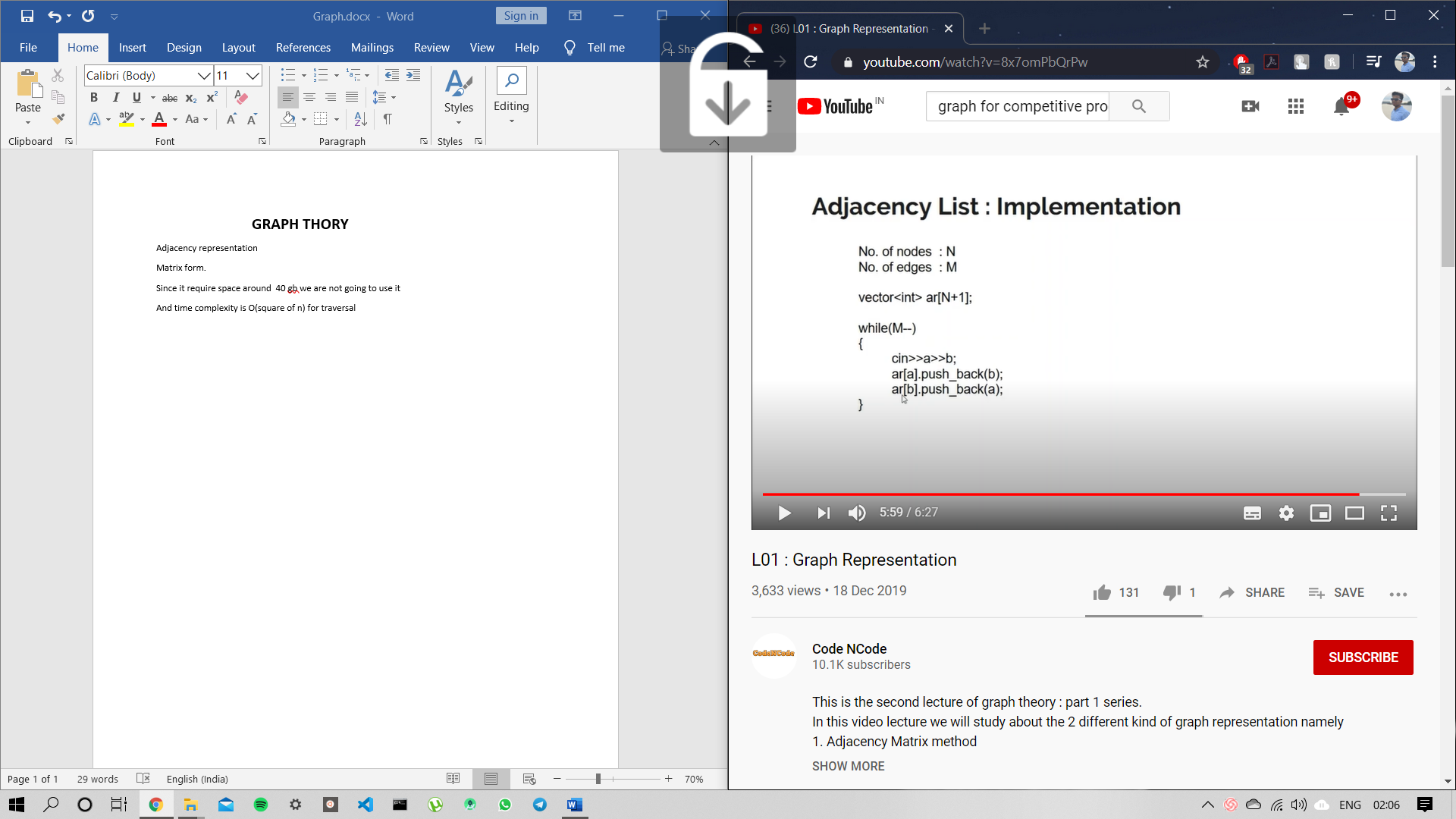
Adjacency representation

Matrix form.

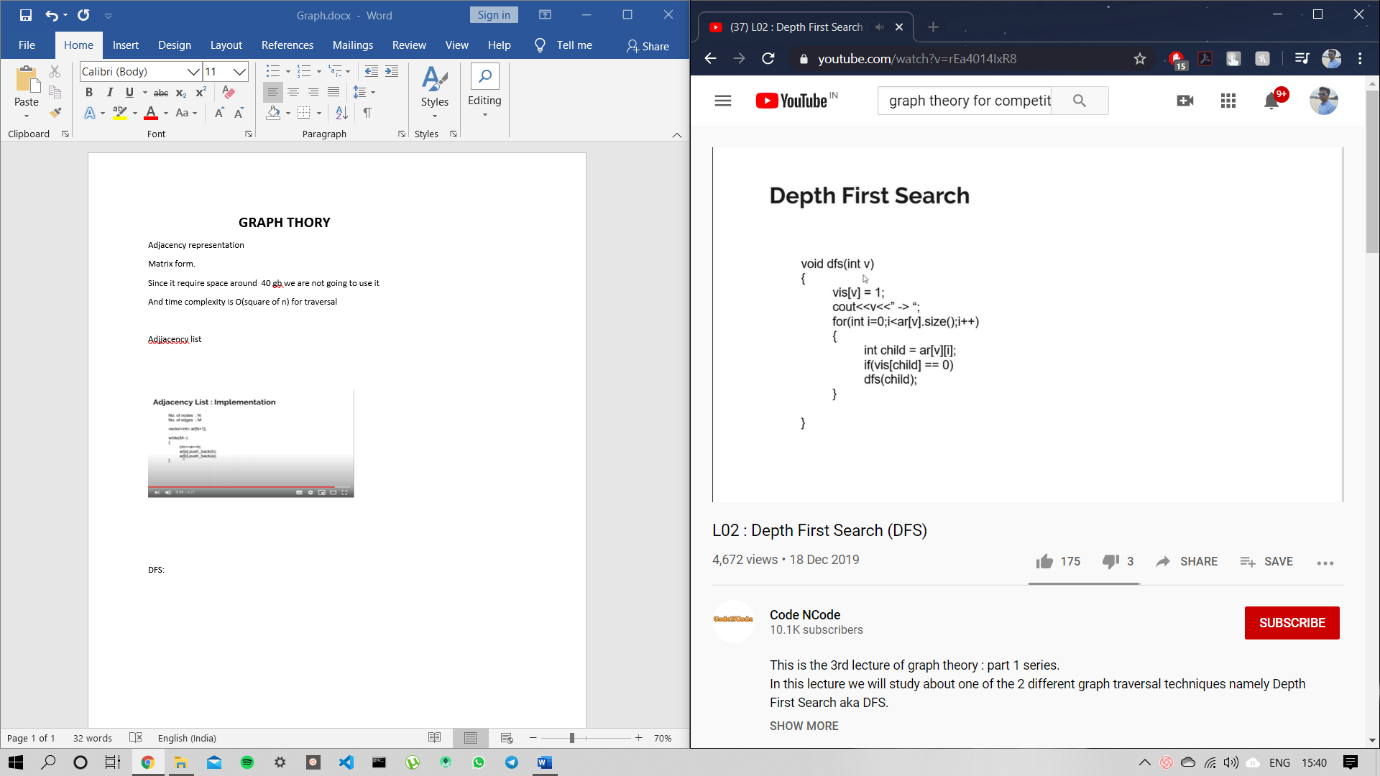
Since it require space around 40 gb we are not going to use it

And time complexity is O(square of n) for traversal

Adjacency list



DFS:



Note : tree does not contain cycle , difference between graph and cycle is there is no cycle in tree.

Single Source shortest path: gives shortest distance between source and any other node in graph

Code:

vector<int> v[10001];

int vis[10001] , distance[10001];

void dfs(int node ,int dis)

{

vis[node]=1;

distance[node]=dis;

for(int child :v[node])

{

if (vis[child] == 0)

{

dfs(child, dis[node]+1);

}

}

}

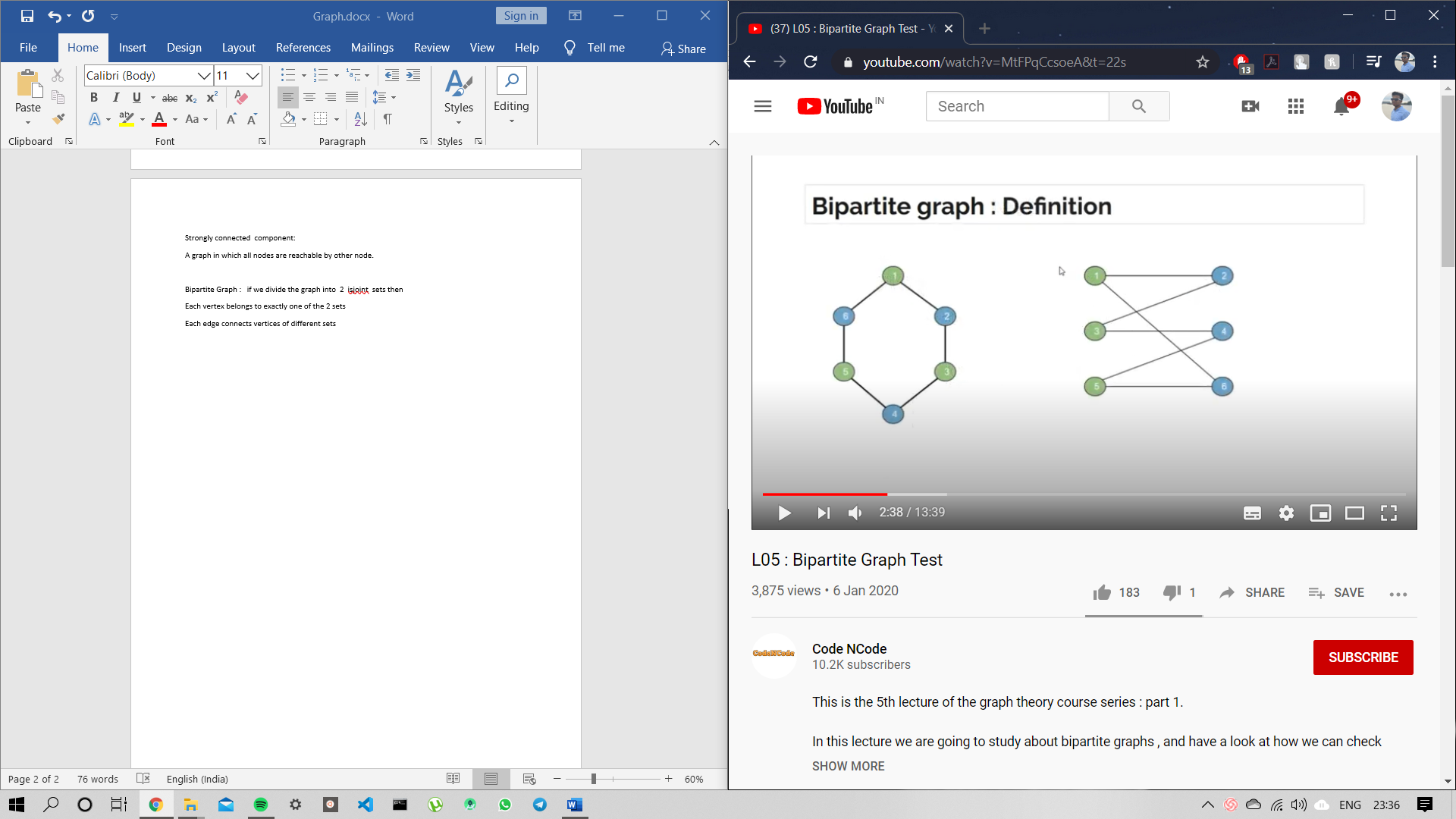
Strongly connected component:

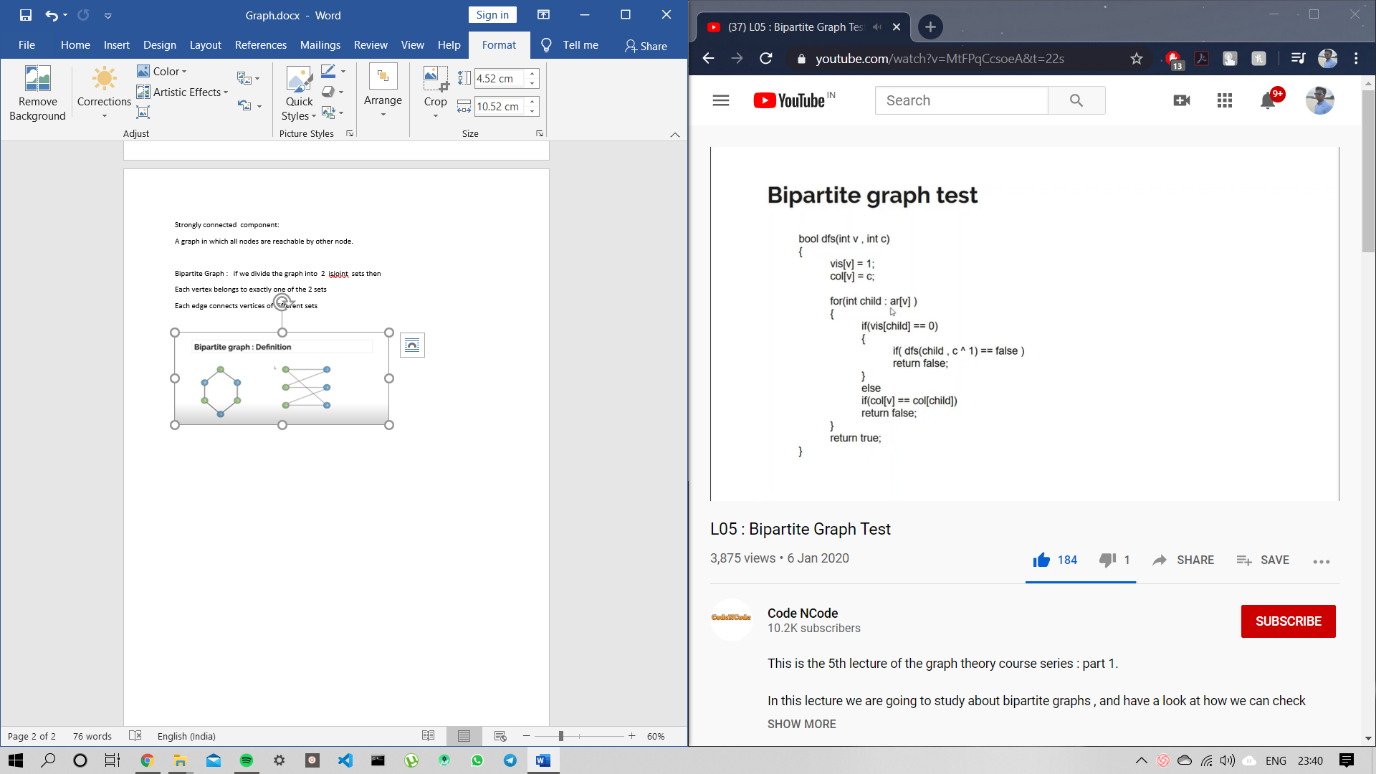
A graph in which all nodes are reachable by other node.

Bipartite Graph : if we divide the graph into 2 is joint sets then

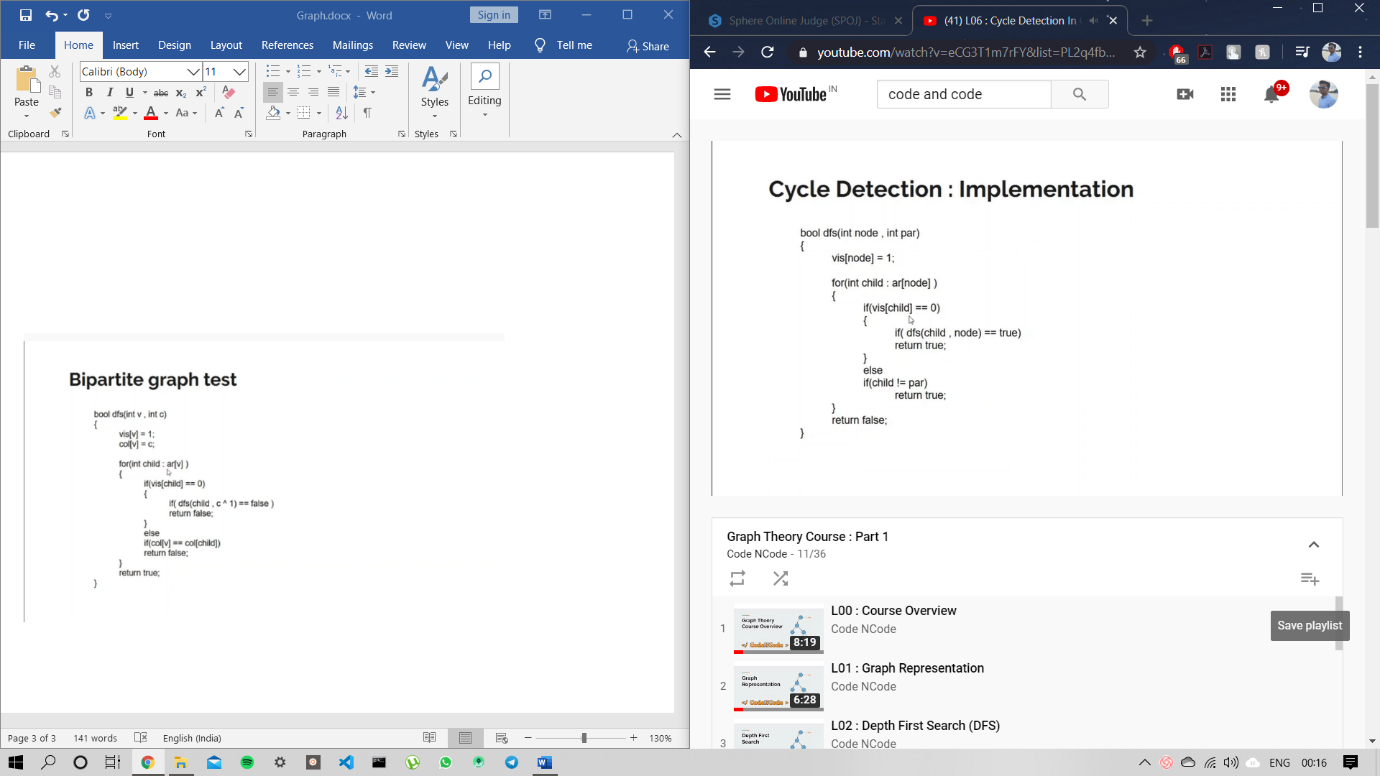
Each vertex belongs to exactly one of the 2 sets

Each edge connects vertices of different sets

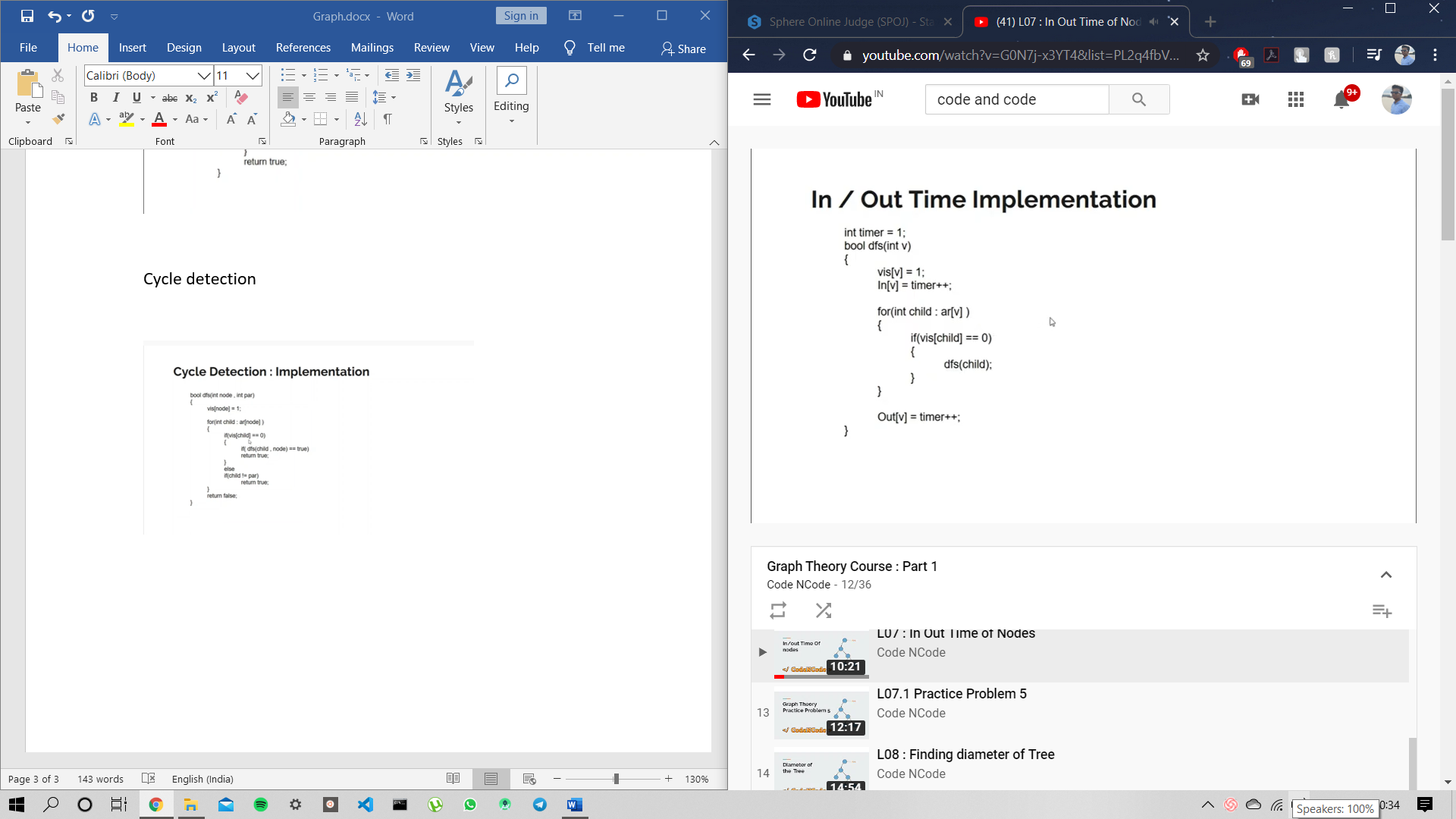




Cycle detection



In /out time in graph



Diameter of a tree:

Longest path between any 2 node in the

Novice Approach

Run DFS N time

Time complexity N^2

Haha……….

Better approach

Run DFS 2 times

Take any arbitrary node run dfs then find the farthest node ,let the node be x.

Run a dfs from node x and find farthest node . update the farthest node acc to condition and find the maximum distance.

Nodes in subtree of given node.

Pass the node through dfs function and hence we get the ans

Int dfs(int node)

{  
 vis[node]=1;

Int cur\_size=1;

For(int child : v[node])

{

If(vis[child]==0)

{

Cur\_size+=dfs(child);

}

}

subSize[node]=cur\_size;

return cur\_size;

}

BFS Breadth first Search:

Queue<int> q;

Int bfs(int src)

{

Queue.insert(node);

Vid[node]=1;

Dis[node]=0;

While(!q.empty)

{

Int cur= q.front();

q.pop();

for(int child: ar[child])

{

If(vis[child]==0)

{

q.push(child);

dis[chils]+=1;

vis[child]=1;

}

}

}

Return 0;

}

Bridge:- An edge when removed makes the graph disconnected.

Finding Bridges

Declare array of in [] to store in time

Declare array of low [] to store lowest ancestor that can be reached .

Timer =0 , in[1]=0 and low[1] = 0;

Vi::ar[100];

Int in[101],low[101],vis[101],timer=0;

Void dfs(int node ,int par)

Vis[node]=1;

In[node]=low[node]=timer;

Timer++;

For(int child: a[node])

If(child==parent continue;

If(vis[child]==1)

//backedge

Low[node] = min(lo[node] , in[child])

Else

Dfs(child,node)

Int n , m ,x, y;

Fo()

Cin>>x>>Y;

Ar[x].pb(y),ar[y].pb(x);

Dfs(1, -1) //since it is parent so we can pass anything.

Dfs and Bfs on 2d grid

Void dfs(int x, int y)

{  
 visited[i][j] = 1;

If(isvalid(x-1,j))

{

Dfs(x-1,j);

}

// similar for right , down , left .

}

KOSARAJU ALGORITHM :

Strongly connected components (directed graph)

Points to remember

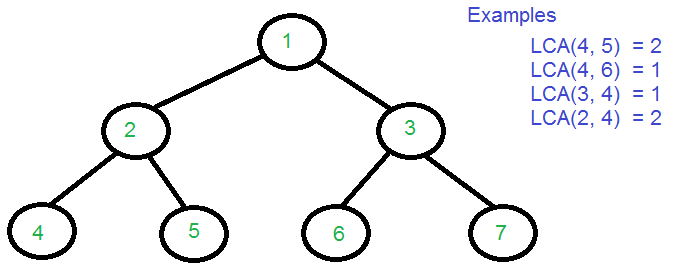
* Graph and transposed graph (indegree changes to outdegree and vice versa)
* Condensation graph
* Condensation graph must be acyclic
* Outdegree of one of scc is greater then other scc in condensed graph.
* There is atleast one scc having indegree 0 .

Now the question is which scc should be called first.

Run dfs on the graph assign out time of each node , then sort the list by out time of node.

**Least common ancestor**

**LCA is the node from where both the node can be reached with minimum distance .**



Naïve approach

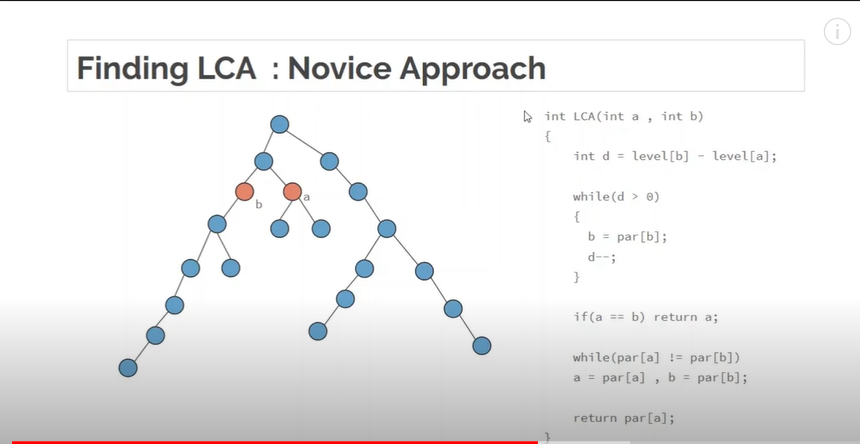
Calculate the level of both node // level can be calculated by dfs.

Make both node to the same height

Check whether they lie on same path if so then the LCA is either one of them

Else

Reduce the level of each node one by one and check



Space complexity of each case is O(N)

Time complexity of each case is O(N)

Which is quite high .

Part 2 for lowest common ancestor

Since in part one we were calculating linearly so the time complexity war order (n).

Instead we will jump length of l

Where l is power of 2 and l <=d

Explanation :

D=13 , l=8

D=5, l=4

D=1, l=1

For this we need to have a parent array of every node where it stores the parent at ith index where it would be parent of 2^i.

For each node we have to keep a list of at max log(n).

2^i <= n

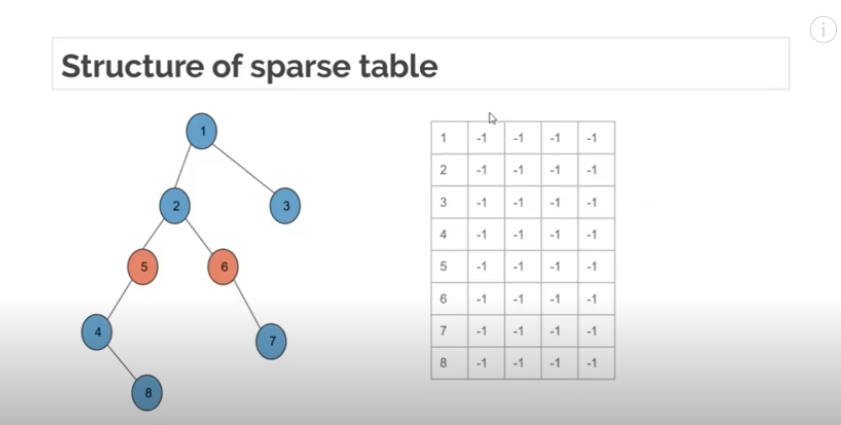
I <= log2(n)

Maxn = log[n]

Well create lca[n+1][maxn] = {-1} //non of the parents exist

Where lca[i][j] = 2^j parent of node i.

Eg:



Find 2^0 parent by running dfs by passing node and its parent

But for finding every 2^ith parent by dfs it would be very time costly.

So we can do is use dynamic programming

2^I = 2\* (2^i-1)

Suppose for node 5 2 power 1 can be written as 2 times 2power0.

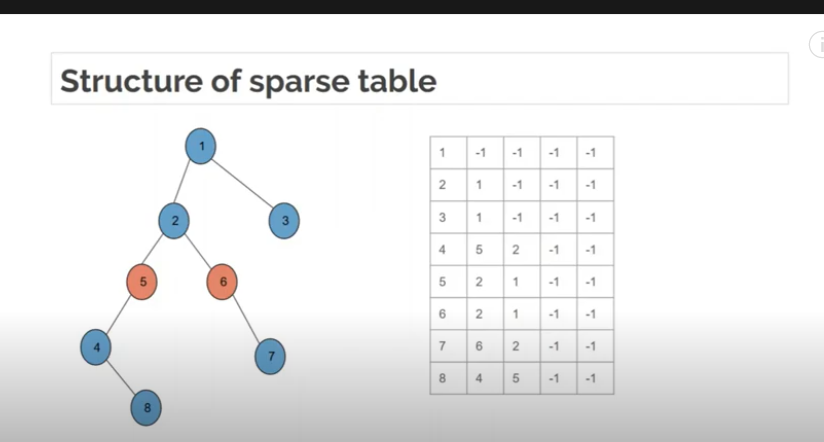
Another example

2^2th parent of 8

2^2= 2\* 2^1

2^1 of 8 is 5 and w can ask 5 for its 2^1th parent which is 1 so 2^2th parent of 8 is 1.

After filling 2^1th parent list



**Distance between 2 nodes**

Time complexity

Simple calculate LCA using binary lifting and the ans is level of node 1 + levelof node2 – 2\*times level of LCA

**Dijkstra’s Algorithm**