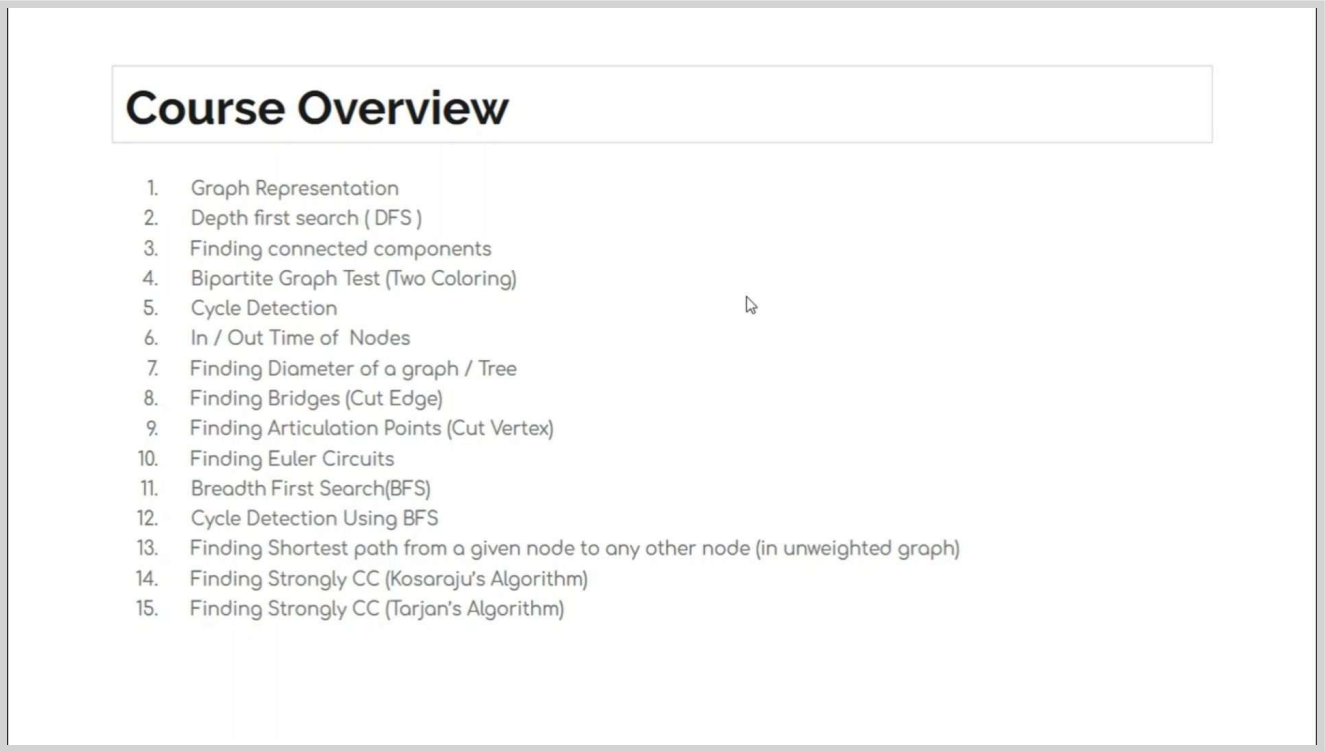
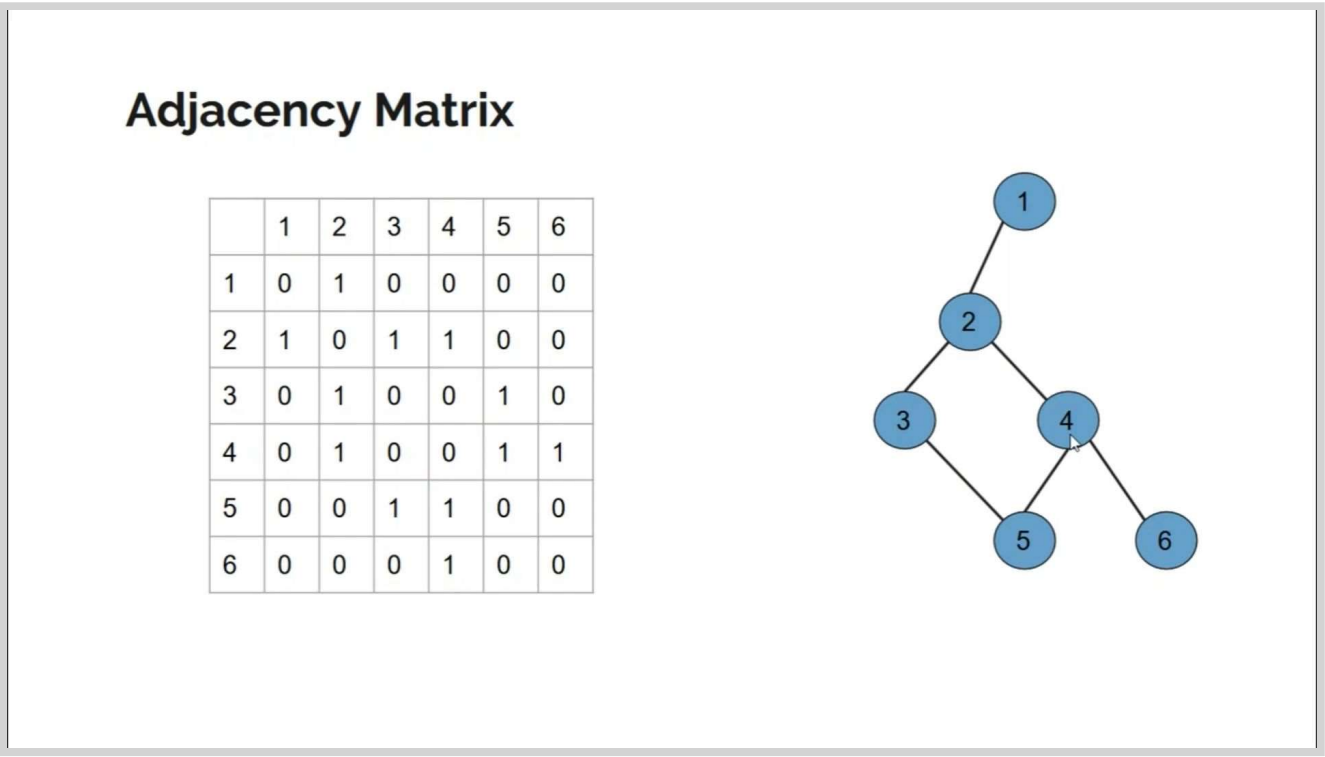
Graph Theory - 1

08:04



1. Graph Representation

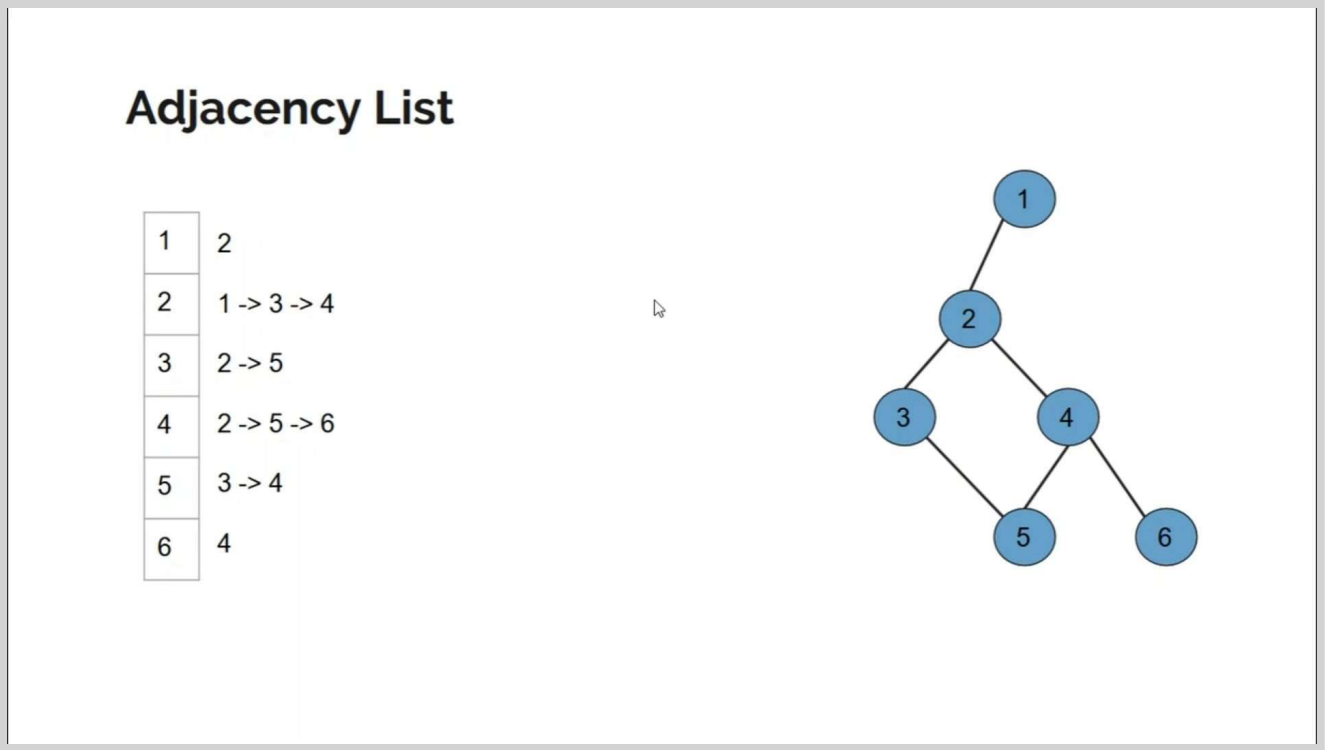
02:08



	1	2	3	4	5	6
1	0	1	0	0	0	0
2	1	0	1	1	0	0
3	0	1	0	0	1	0
4	0	1	0	0	1	1
5	0	0	1	1	0	0
6	0	0	0	1	0	0

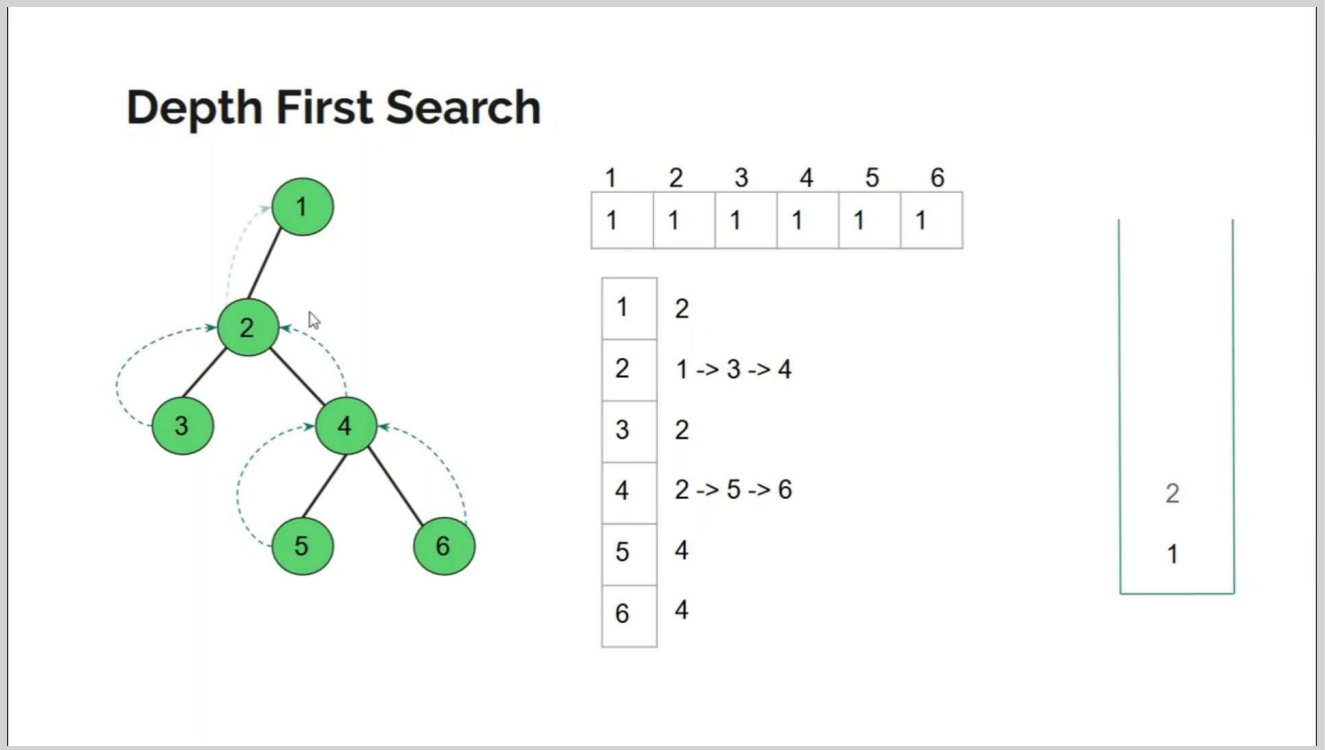
O(n*2) -> for bfs and dfs and also memory wise slow.

05:23



2. Depth First Search

12:28



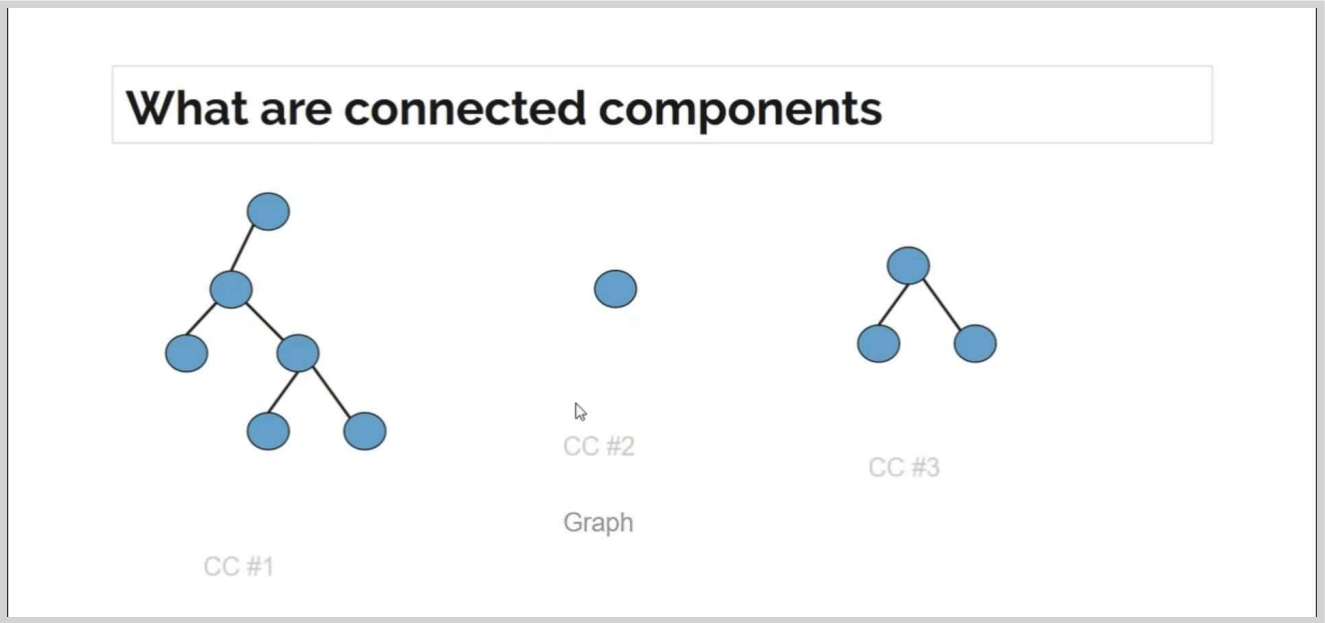
Code:

```cpp
void dfs(vector<vector<int>> &adj,vector<bool> &visited,int root){
    cout<<root<<" ";
    // tasks while first visit.
    visited[root]=1;
    for(auto child: adj[root]){
        if(!visited[child])
            dfs(adj,visited,child);
    }
    // tasks while returning.
}
```

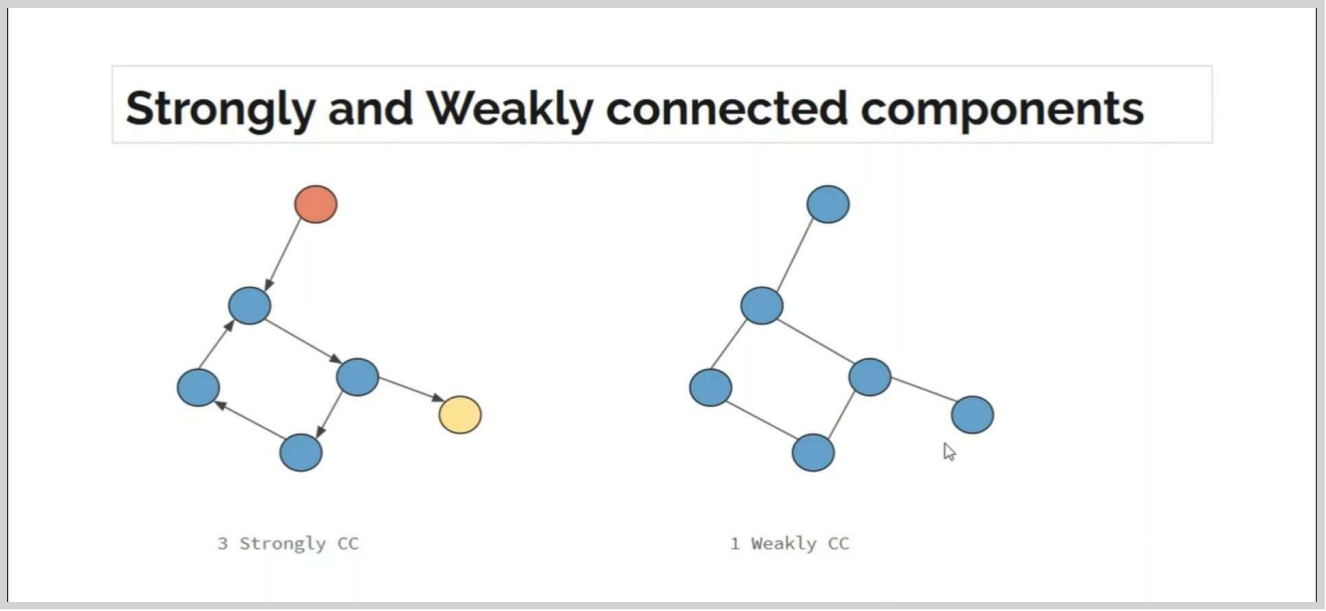

3. Count Connected Components

Number of times we need to make DFS calls to the graph.

01:16



04:13

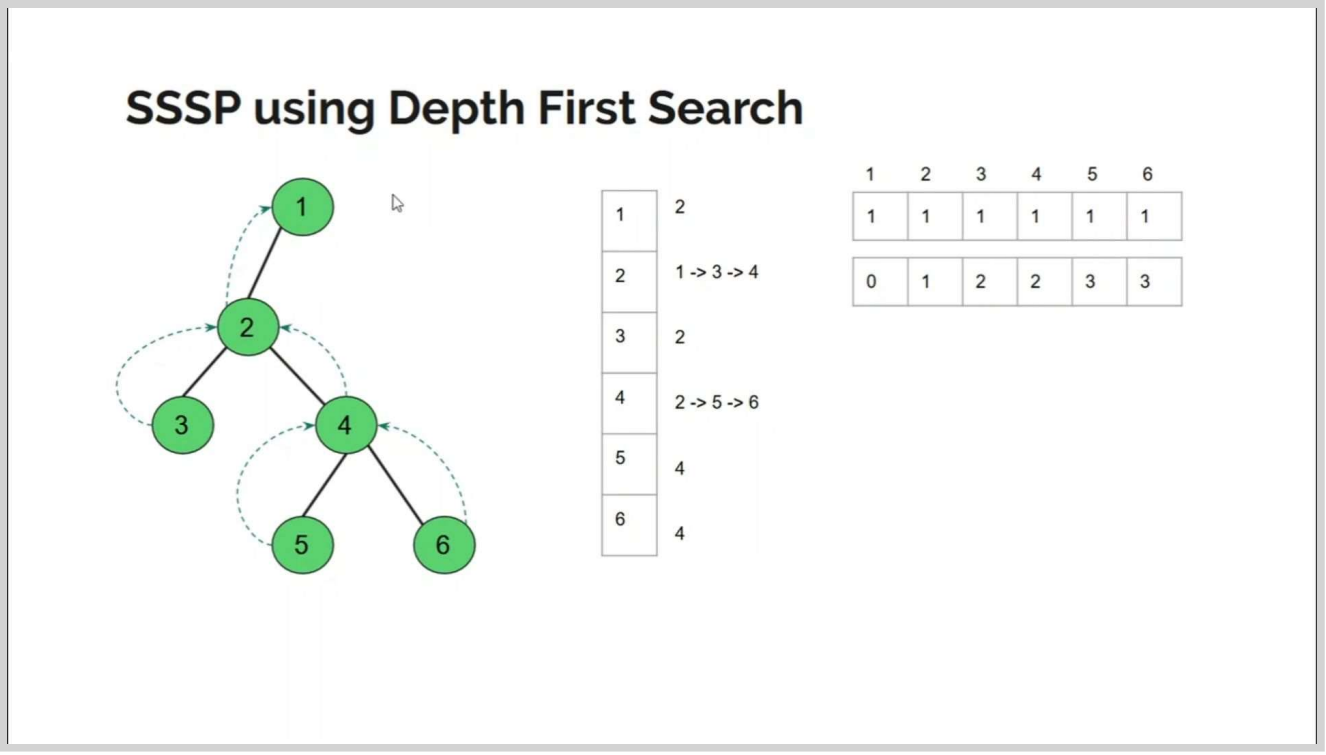


Code: Problem link

```
// Connected Components in Undirected Graph

int CountConnectedComponents(vector<vector<int>> &adj, vector<bool> &visited, int root)
{
    int connected_components = 0;
    for(int root=0;root<n;root++){
        if(!visited[root]){
            dfs(adj,visited,root); // simple dfs on graph
            connected_components++;
        }
    }
    return connected_components;
}
```


4. Single source shortest path (on Tree) using DFS.

08:13



The shortest path to every other node from the given node, this DFS will work only on Trees.

Code: Problem Link

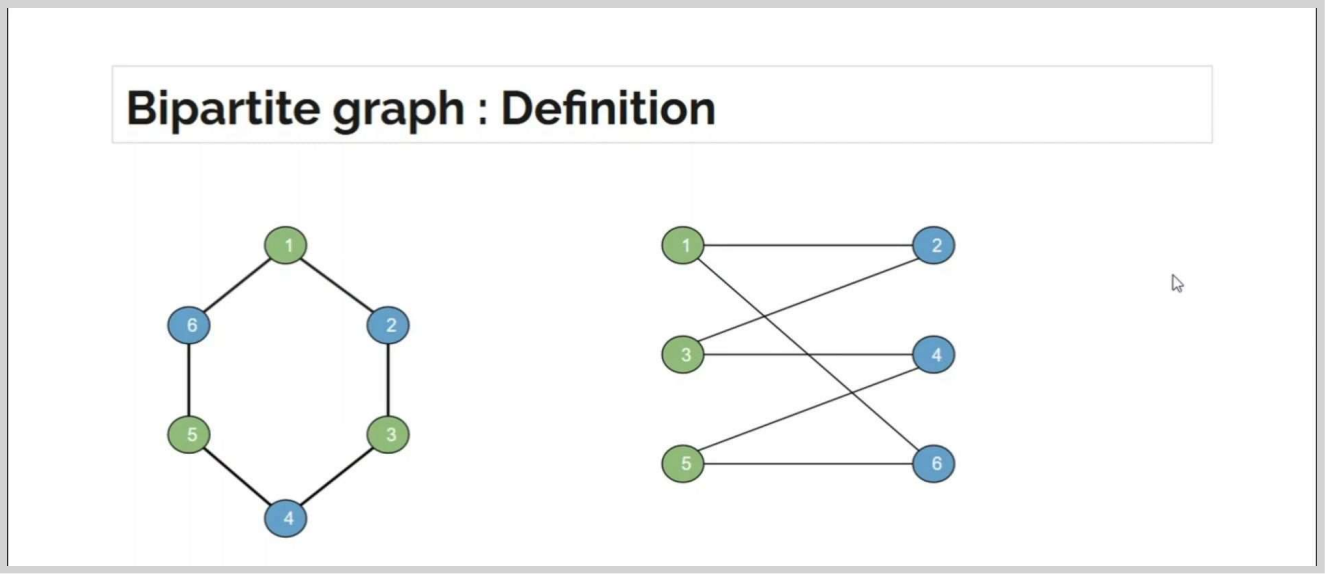
```
// Single Source Shortest Path on Trees using DFS

void SingleSourceShortestPath(vector<vector<int>> &adj, vector<bool> &visited, int source,int dist)
{
    distance[source] = dist;
    visited[source] = 1;
    for (auto child : adj[source])
    {
        if (!visited[child])
            SingleSourceShortestPath(adj, visited, child, dist + 1);
    }
}
SingleSourceShortestPath(adj,visited,source,0)
```


Tree: Undirected Graph (no cycle + n-1 edges + 1 CC)

5. Bipartite Graph Test

2 Colouring Problem

04:37



Code:

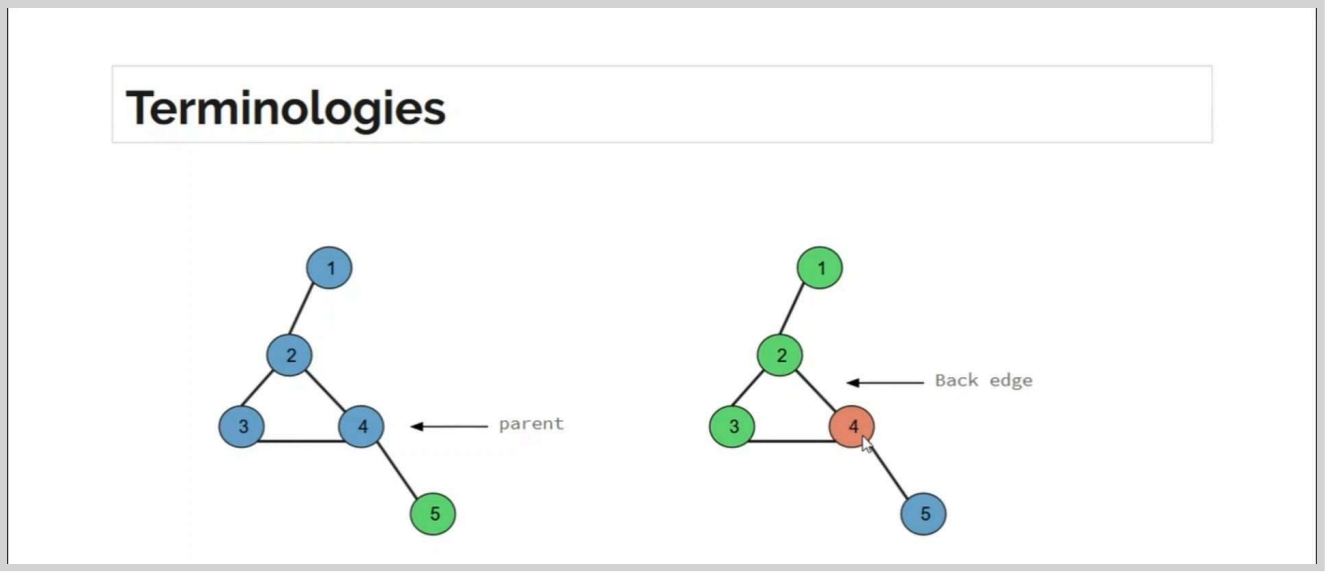
```cpp
// Check whether Graph is bipartite or not

bool checkBipartite(vector<vector<int>> &adj, vector<bool> &visited, int root,bool color)
{
    colors[root] = color;
    visited[root] = 1;
    for (auto child : adj[root])
    {
        if (!visited[child]){
            if(checkBipartite(adj, visited, child,(color^1))==false)
                return false;
        }
        else if(colors[root]==colors[child]){
            return false;
        }
    }
    return true;
}
```


Problem Link

6. Cycle Detection (using DFS)

whether a undirected graph contains cycle or not.

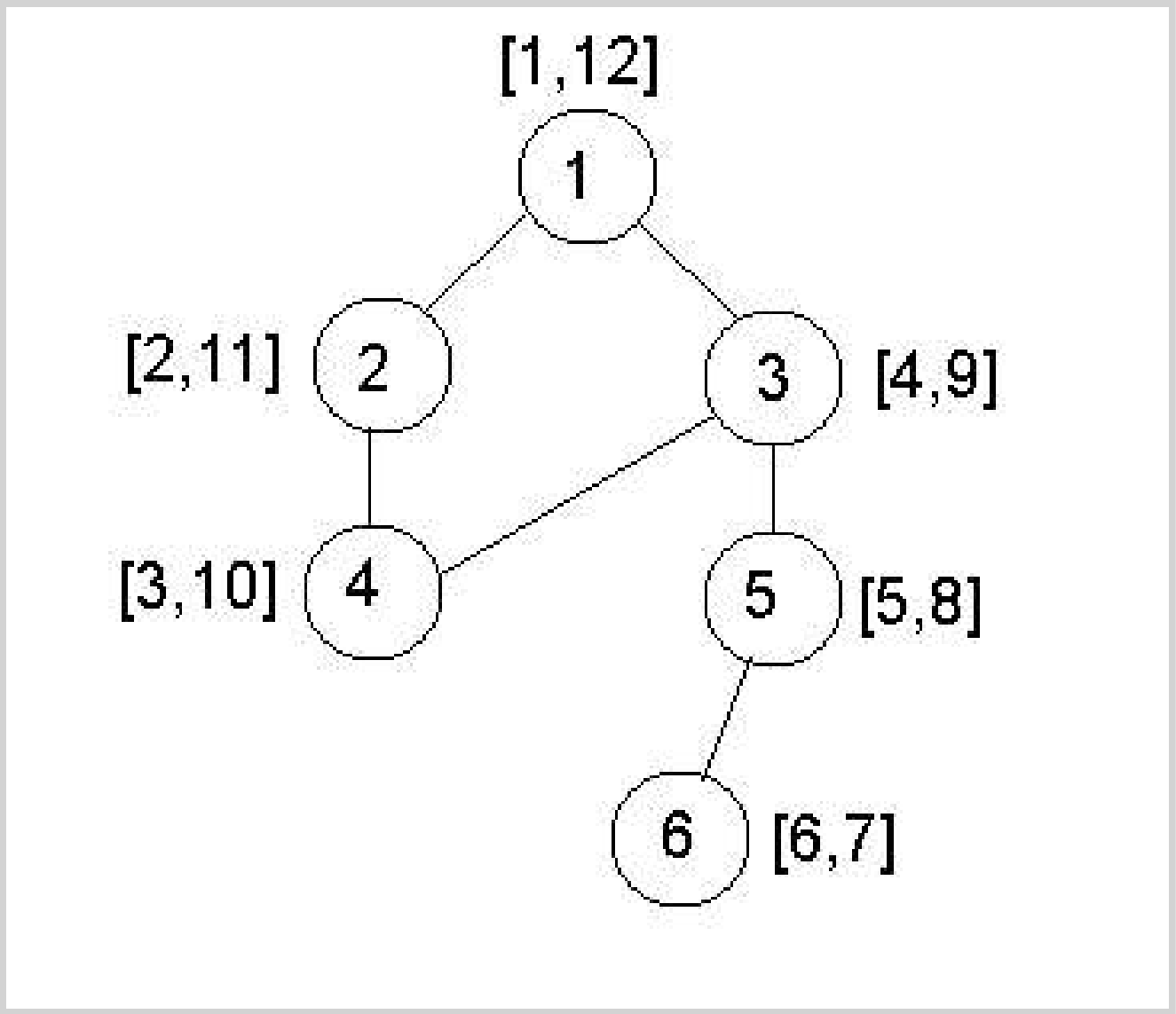
06:02



Code:

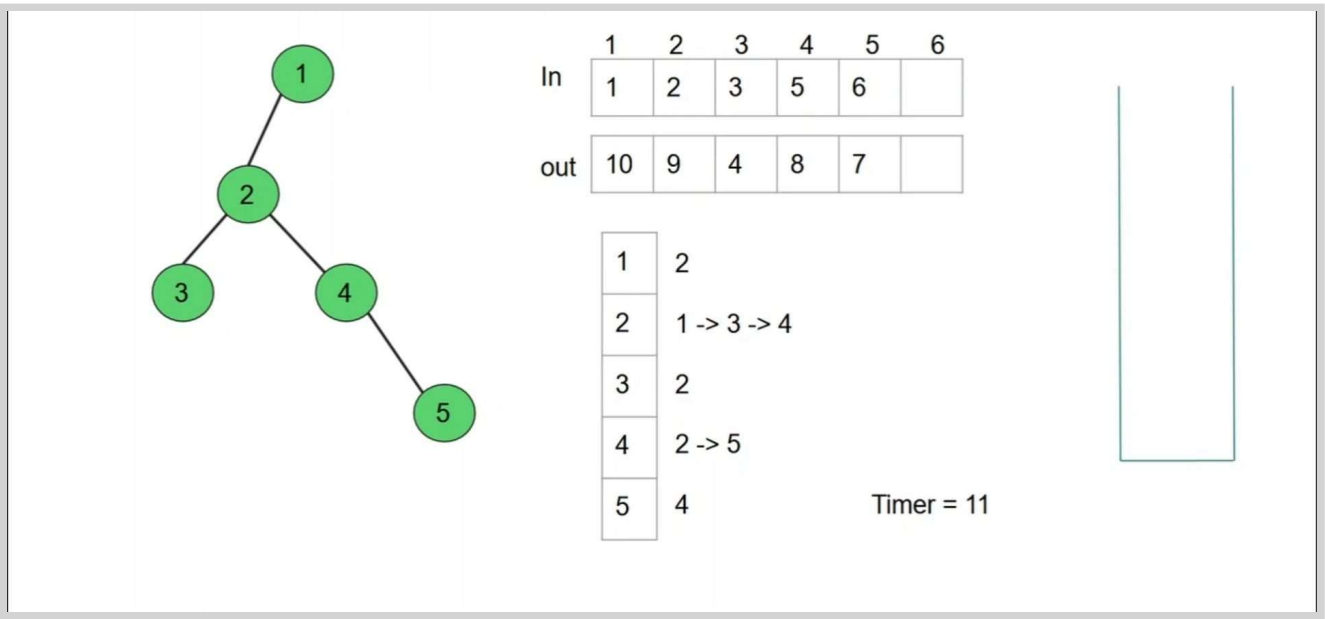
```cpp
// Check whether Undirected Graph has cycle or not using DFS

bool containsCycle(vector<vector<int>> &adj, vector<bool> &visited, int root,int parent){
    visited[root] = 1;
    for (auto child : adj[root])
    {
        if (!visited[child]){
            if (checkCycle(adj, visited, child, root) == true)
                return true;
        }
        else if(parent != child){
            return true;
        }
    }
    return false;
}
// calling
containsCycle(adj,visited,0,-1);
```


7. In/Out Time of Nodes



To find whether **u** lies in the sub-tree of **v** or not we just compare the in and out time of **u** and **v**. If **in[u] > in[v]** and **out[u] < out[v]** then **u** lies in the sub-tree of **v** otherwise not.

09:57

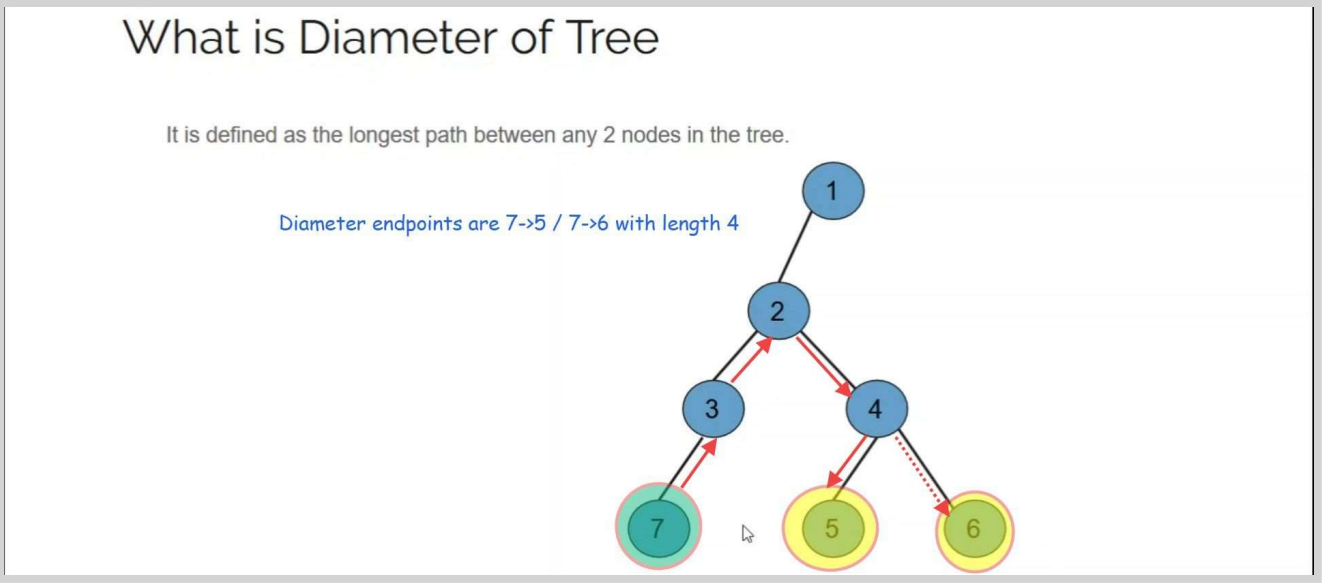


Code:

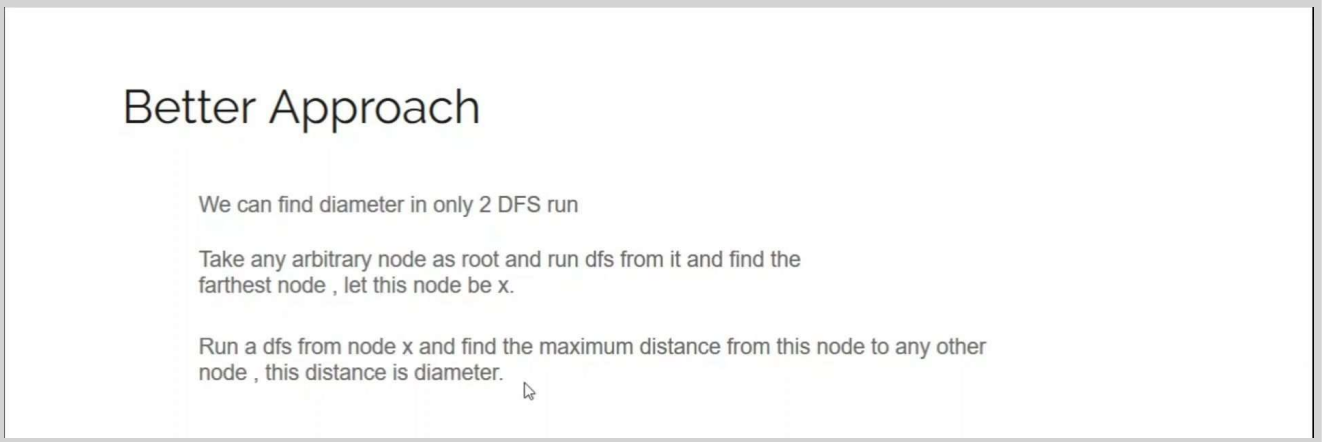
```cpp
// In Out Time for each node in Graph using DFS

int timer=0;
bool dfs(vector<vector<int>> &adj, vector<bool> &visited, int root, vector<int> &in, vector<int> &out)
{
    in[root] = timer;
    timer++;
    visited[root] = 1;
    for (auto child : adj[root])
    {
        if (!visited[child])
        {
            dfs(adj, visited, child,in,out);
        }
    }
    out[root] = timer;
    timer++;
}
```


8. Diameter of the Tree

01:05

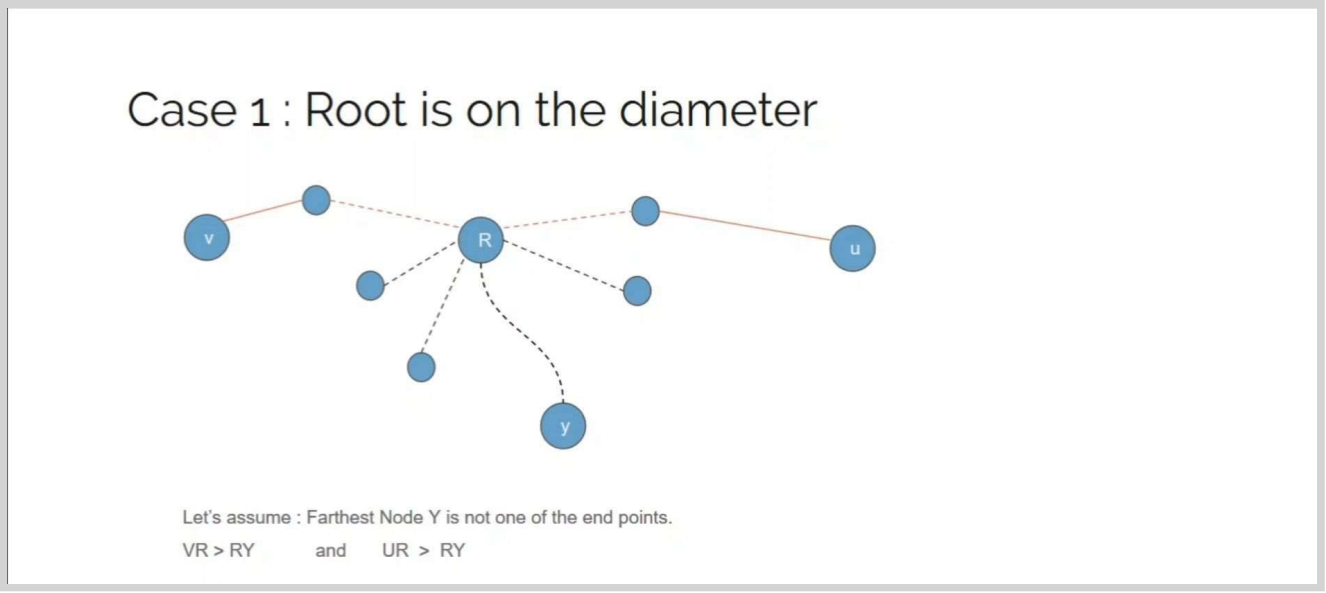


04:39



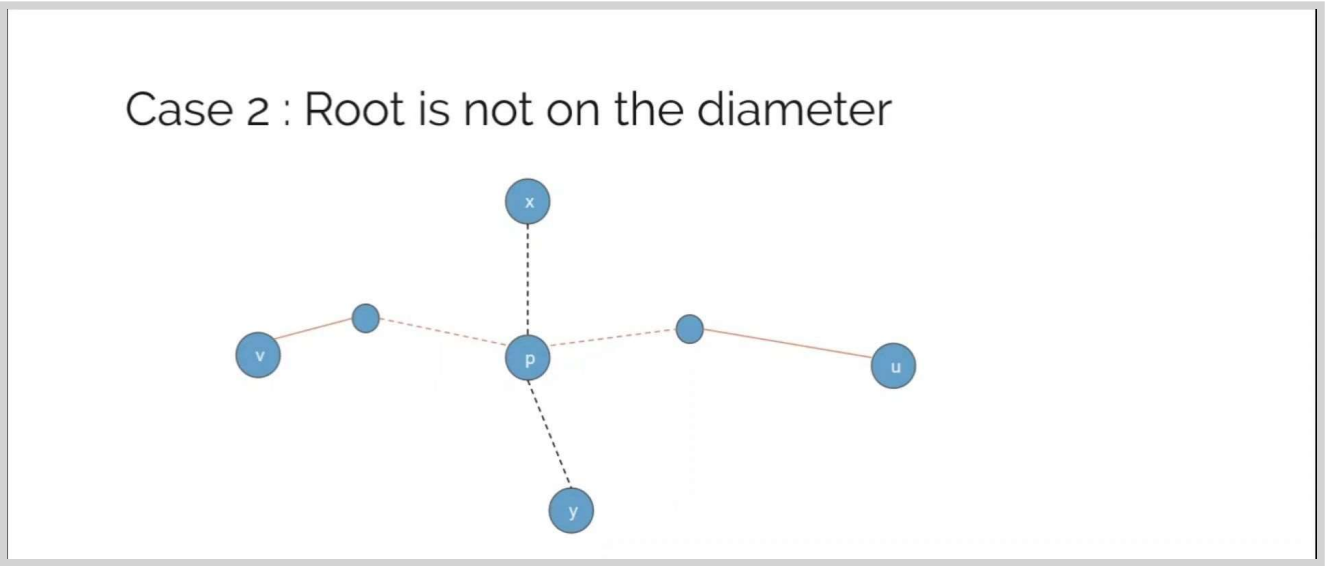
Proof of this approach:

11:15



we have **R->y** (farthest node) but u,v are the end of diameter so, by contradiction **y** is the end point of the diameter.

14:41



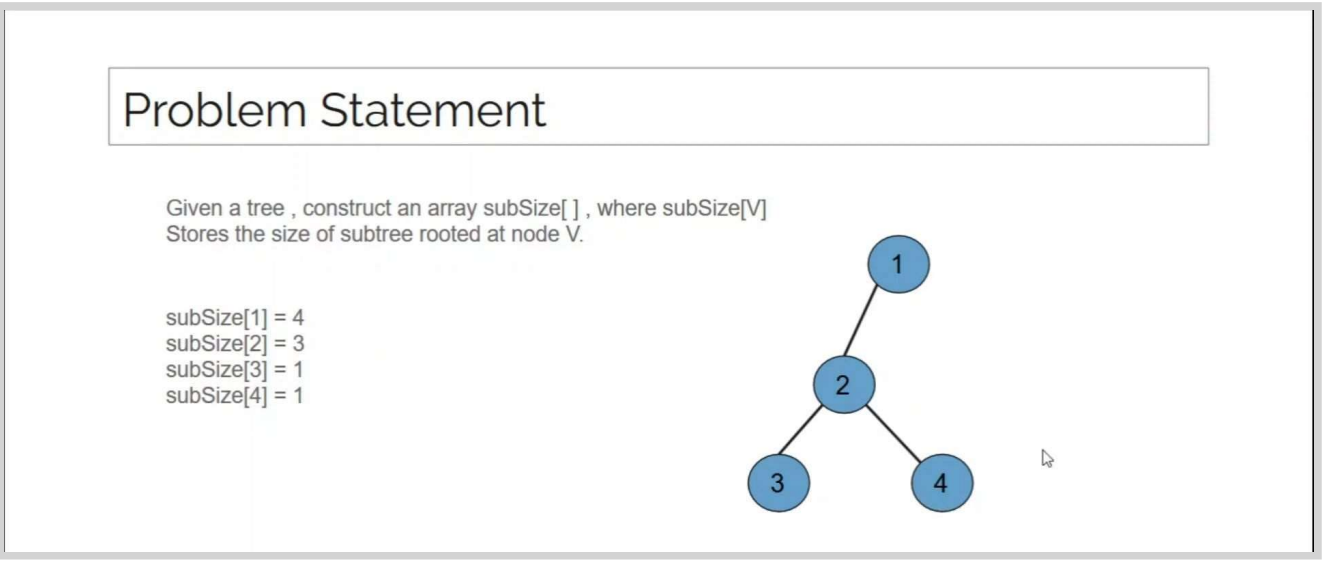
Code:

```cpp
// Diameter of a tree in Detail

void endPoint(vector<vector<int>> &adj, vector<bool> &visited, vector<int> &endNodes, int root, int distance)
{
    visited[root] = 1;
    if (dis == maxDis){
        endNodes[root] = 1;
    }
    for (auto child : adj[root]){
        if (!visited[child]){
            endPoint(adj, visited, child, distance + 1);
        }
    }
}

void Diameter(vector<vector<int>> &adj, vector<bool> &visited, vector<int> &maxDisNode,int &node,int root, int distance)
{
    visited[root] = 1;
    if (distance > maxDis)
    {
        maxDis = distance;
        node = root;
        maxDisNode[root] = 1;
    }
    for (auto child : adj[root])
    {
        if (!visited[child]){
            Diameter(child, distance + 1);
        }
    }
}
// function calls
maxDis = -1;
Diameter(adj,visited,maxDisNode,node,1,0);
// never forgot to clear visited arr before another dfs
maxDis = -1;
for (int i = 0; i < n + 1; i++)
    visited[i] = 0;
Diameter(node, 0);
for (int i = 1; i <= n; i++)
{
    for (int i = 0; i < n + 1; i++)
        visited[i] = 0;
    if (maxDisNode[i])
    {
        endPoint(adj,visited,endNodes,i, 0);
    }
}
```


9. Subtree size in O(N) using DFS

Given Tree, return array, i.e. subTreeSize[node] size of subtree rooted at node.

01:26



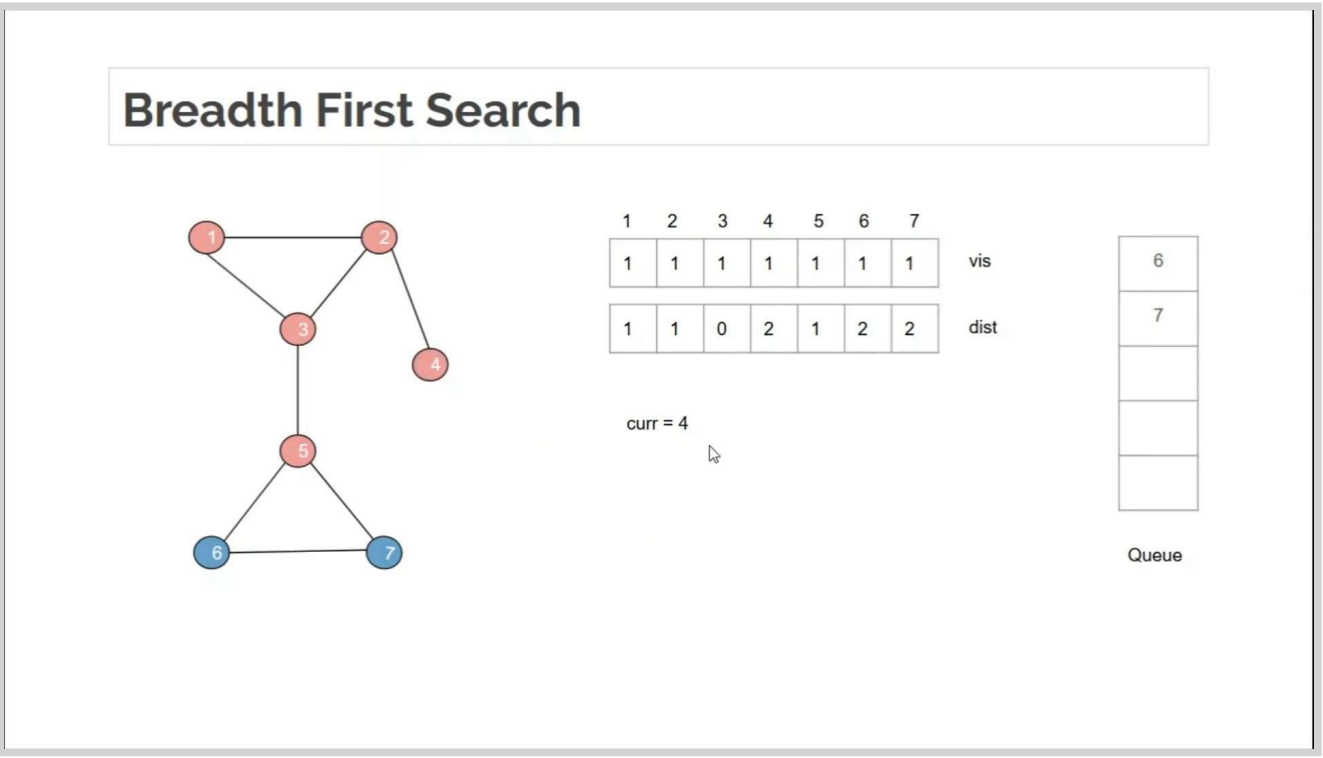
Code

```cpp
// Sub Tree Size on Trees using DFS

int subTreeSize(vector<vector<int>> &adj, vector<bool> &visited, vector<int> &subtreeSize, int root)
{
    visited[root] = 1;
    subtreeSize[root]=1;
    for (auto child : adj[root])
    {
        if (!visited[child])
            subtreeSize[root] += subTreeSize(adj, visited, subtreeSize, child);
    }
    return subtreeSize[root];
}

vector<int> subtreeSize(n,0);
subTreeSize(adj, visited, subtreeSize, 0);
```


10. BFS (Breadth First Search)

SSSP in unweighted graph

06:16



If we use stack instead of Queue then it will be DFS.

```cpp
// Level wise BFS

void bfs(vector<vector<int>> &adj, int root, int n){
    vector<bool> visited(n, 0);
    queue<int> q;
    q.push(root);
    visited[root] = 1;
    while (!q.empty())
    {
        int sz = q.size();
        while (sz--)
        {
            int curr = q.front();
            q.pop();
            for (auto nb : adj[curr])
            {
                if (!visited[nb])
                {
                    // Do some work
                    q.push(nb);
                    visited[nb] = 1;
                }
            }
        }
    }
}
```


Applications:

1) Given 4-digit prime no. min steps to change it to another 4-digit prime. operation : change only one digit.

Soln. generate all 4-digit prime the connect them if and only if they diff by one digit only. then find the min path.

2) Given relations between numbers

2 = 3, 3!=4, 1=2.....we need to find whether the equations holds true or not.

Soln : Make graph between = edge then for != case ....after making the complete graph check whether != relation lies in diff component or not.

11. Articulation Point And Bridges in Graph

Prerequisites:

DFS Graph:

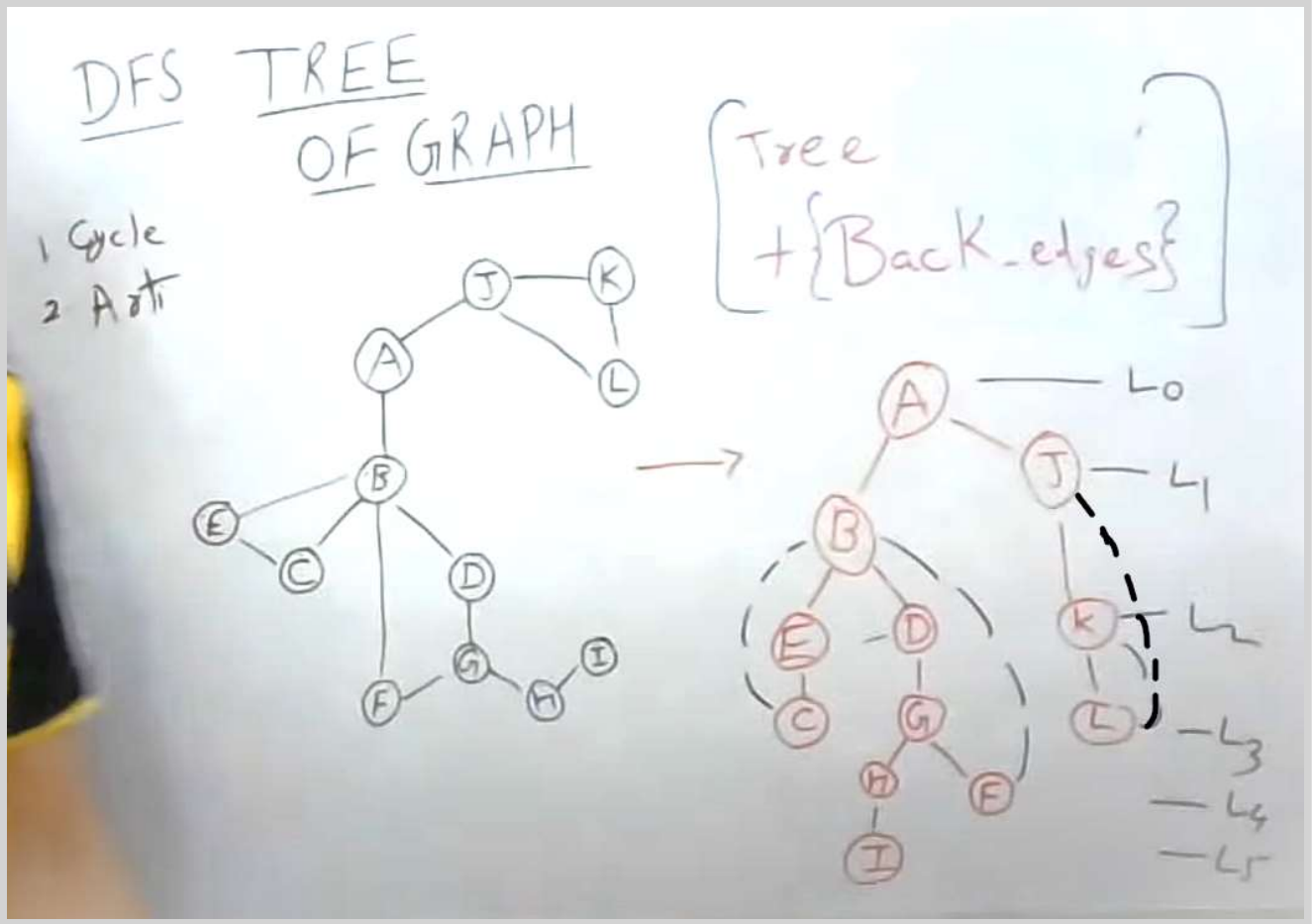
Article Link: https://codeforces.com/blog/entry/68138

DFS Tree + Back Edges

Implementation

Normal DFS but if the neighbor is visited and it is not the parent then it's a back edge

calling `visit(1)` looks like.



Uses:

1) **Cycle Detection:**

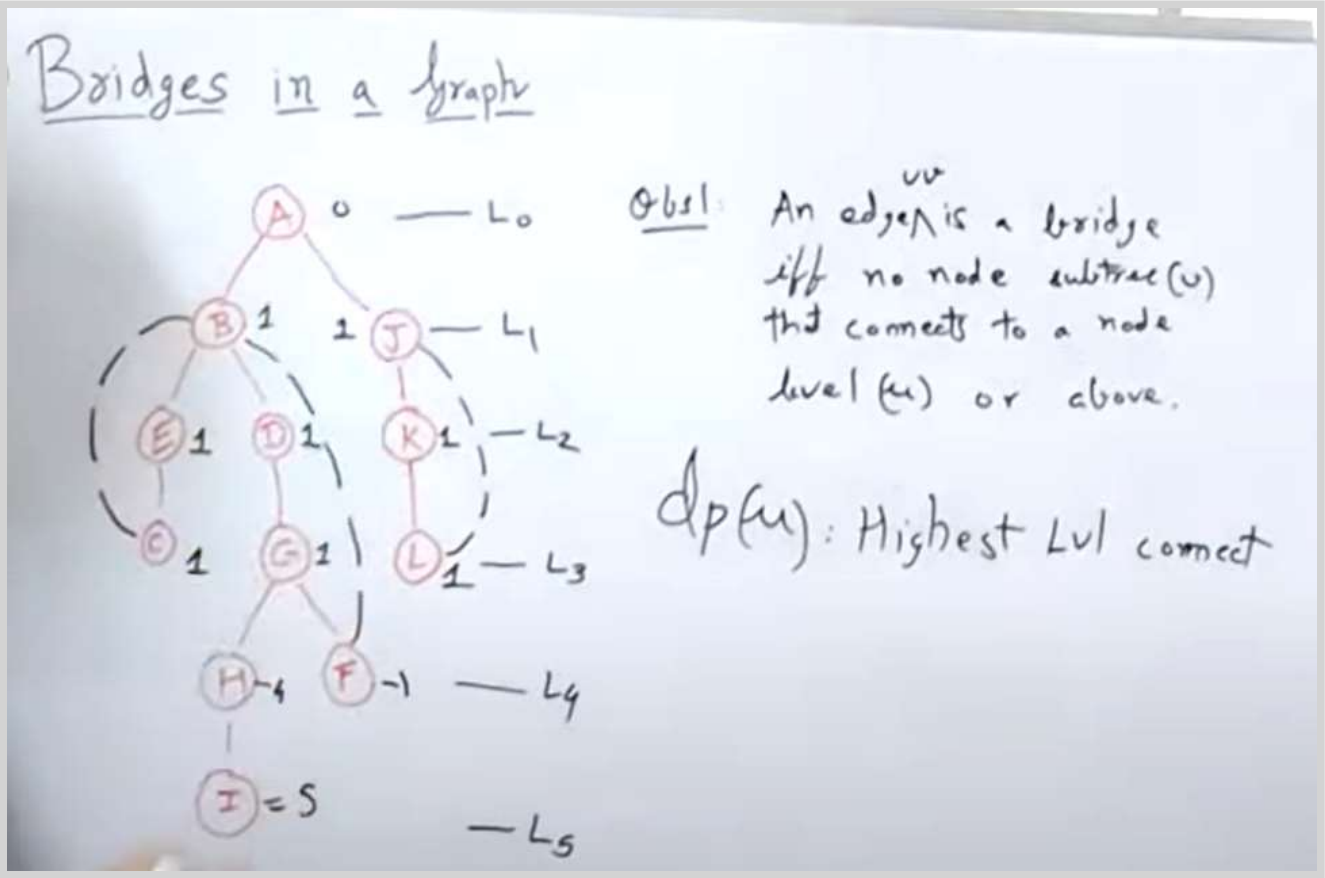
If more than one back edges then Graph has a cycle.

2) **Articulation Point:**

Mark Levels in Representation then for any node, if there is a back edge from subtree to above the tree then this is not an articulation point.

3) **Bridges:**

1. Back edges Can never be a bridge.
2. Mark Levels in Representation then for any node the edge from this node to just the next node is a bridge, if there is a back edge from subtree to above the / or to itself then this is not a Bridge.



DP[u]: The highest level that is connected by back edges.

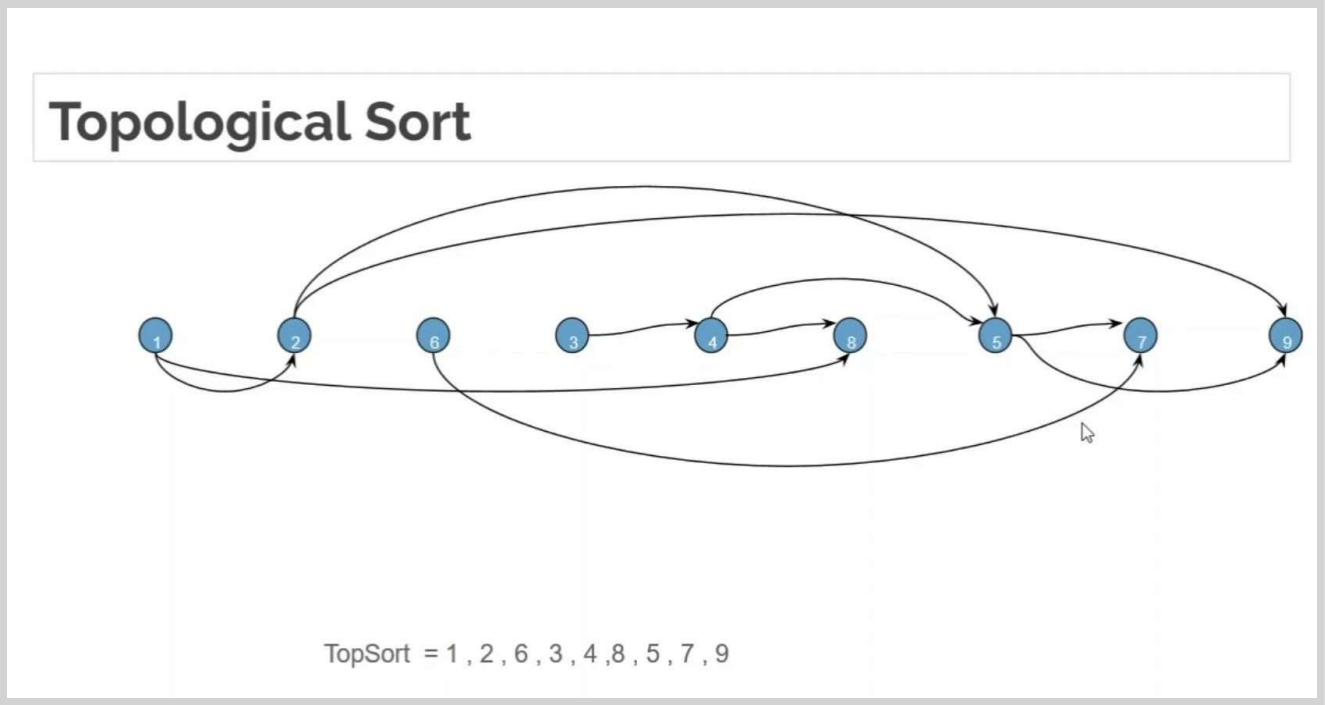
Links: https://codeforces.com/blog/entry/71146

Code: **Tarjan's Algorithm**

12. Topological Sorting

1. Used in Dependency Resolution.
2. If a valid topo sort is possible then there is no cycle in the directed graph. **DAG**

06:20

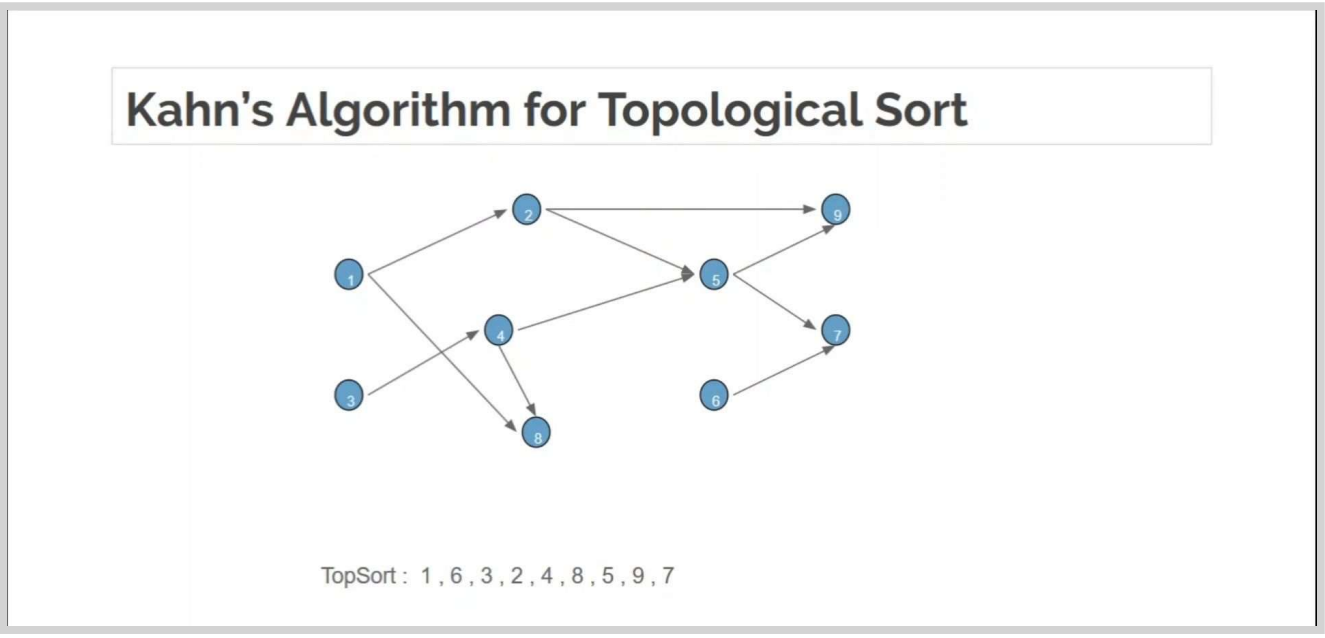


Kahn's Algorithm

1. Make the in-degree array.
2. Start with the zero indegree node and remove all the outgoing edges and add to the topological sort array.
3. Continue step 1.

Valid Toposort: result.size()==N (DAG)

04:30



Code:

```cpp
// Topological Sorting of Directed graph

/* valid iff no cycle is present → DAG */

vector<int> topoSort(vector<vector<int>> &adj, vector<bool> &visited, vector<int> &inDegree, int N)
{
    vector<int> result;
    // min heap
    priority_queue<int, vector<int>, greater<int>> min_heap;

    for (int i = 0; i < N; i++){
        if (inDegree[i] == 0)
            min_heap.push(i);
    }

    while (!min_heap.empty())
    {
        int node = min_heap.top();
        min_heap.pop();
        // cout<<node<<" ";
        result.pb(node);
        for (auto child : adj[node]){
            inDegree[child]--;
            if (inDegree[child] == 0)
                min_heap.push(child);
        }
    }
    //if(result.size()==N) // valid topo sort
    return result;
}
```

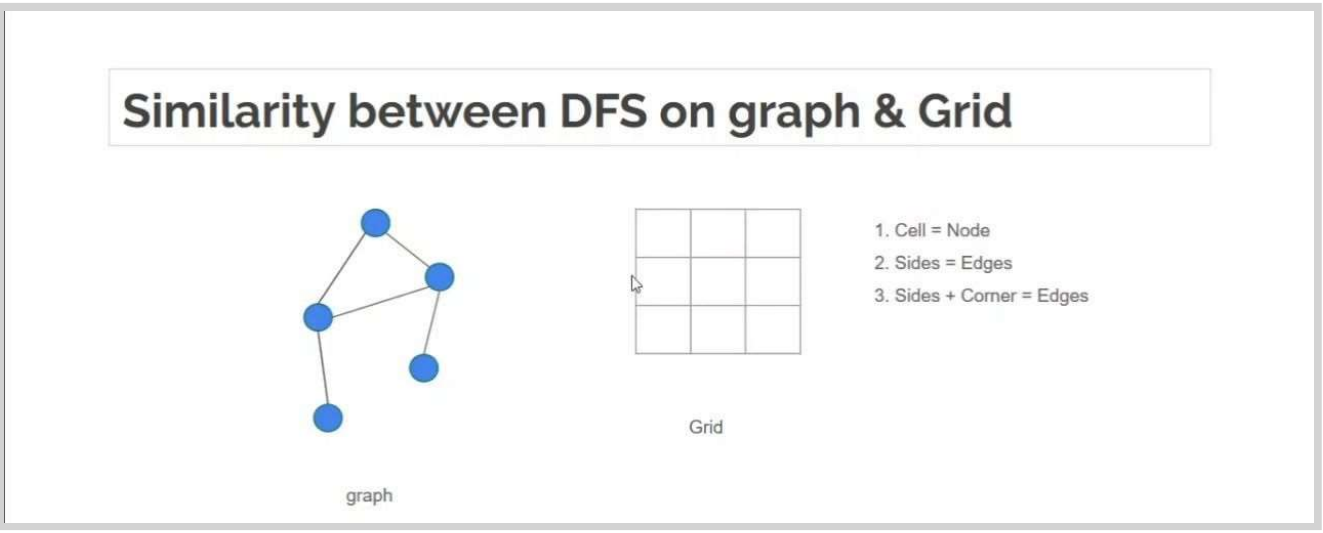
snappify.io

13. Graph Algorithms on 2D Grid

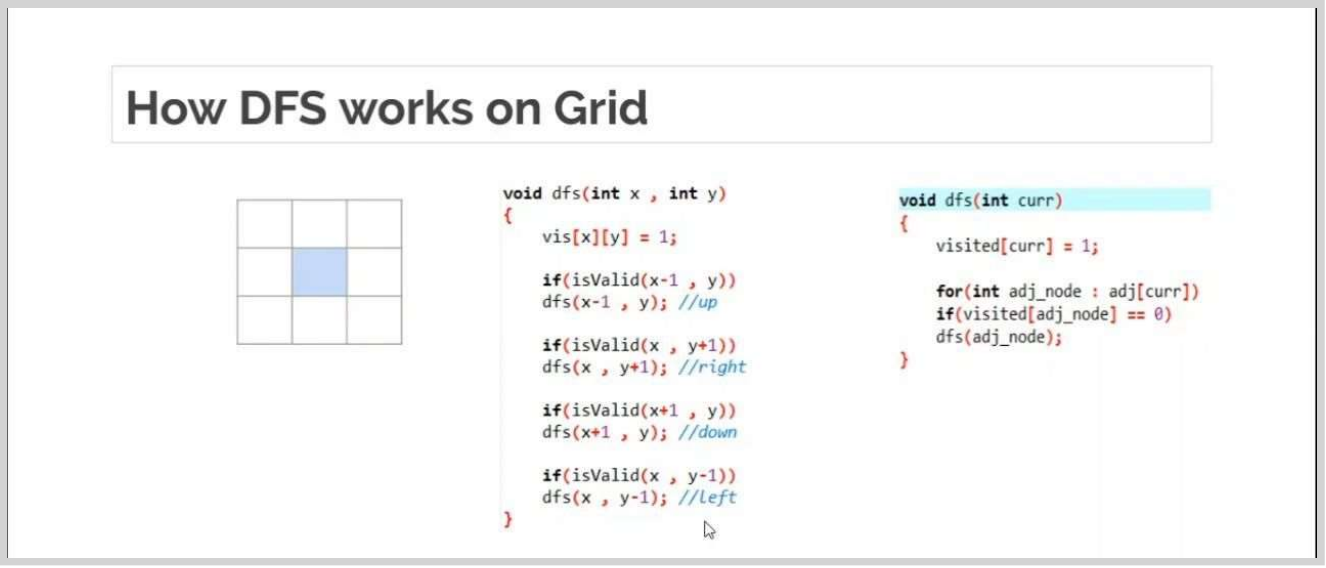
BFS, DFS, Dijkstra's on Grid

1. DFS on Grid

02:17

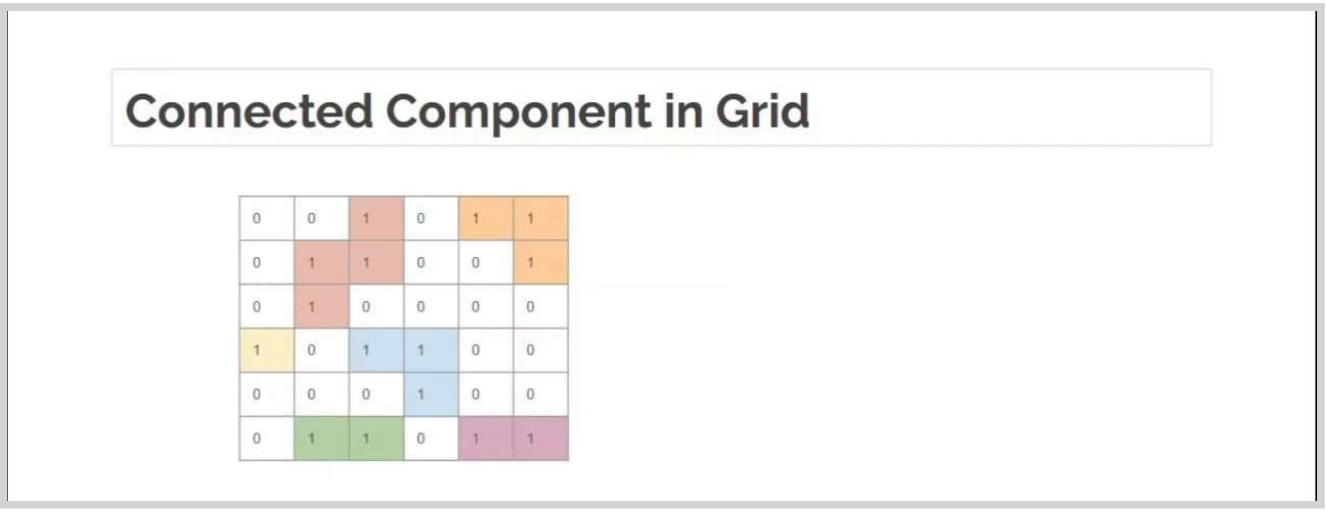


07:38



Connected Component in 2D Grid

[03:03]



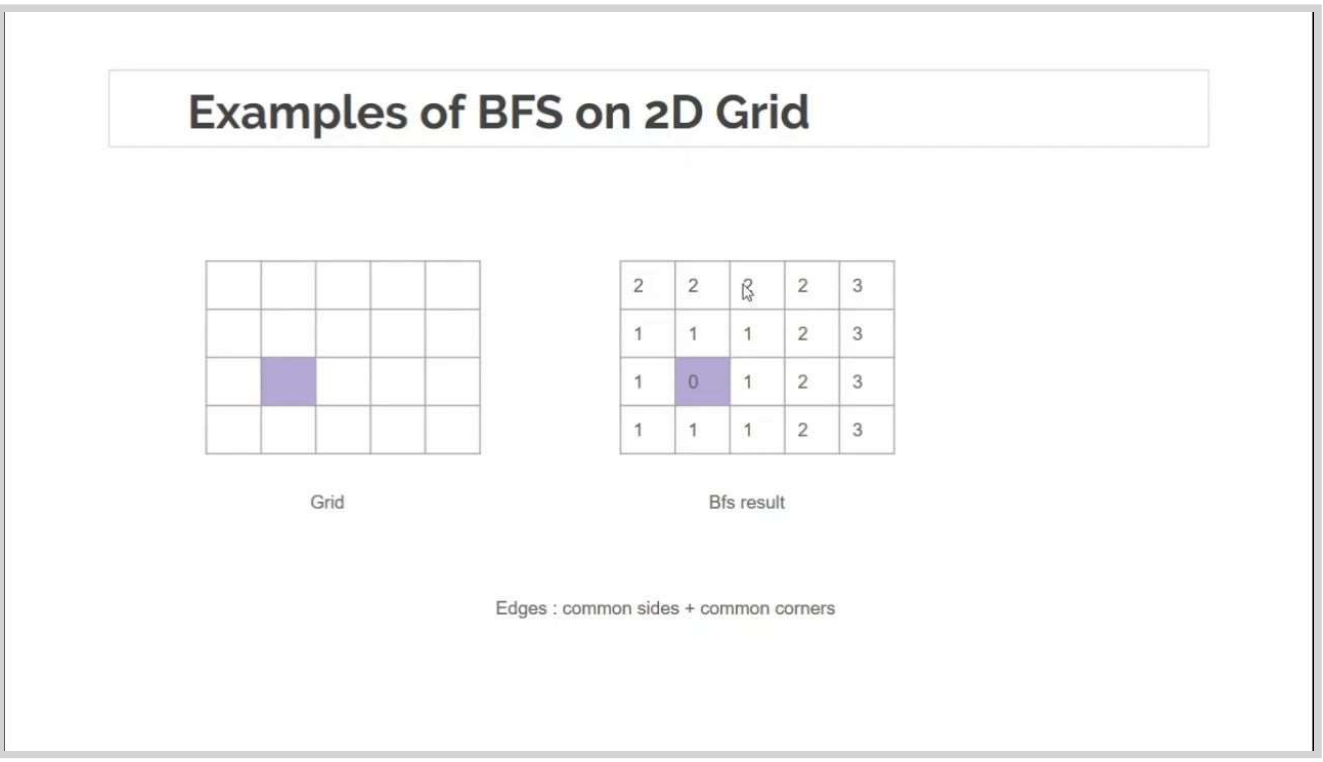
Code:

```cpp
int dx[] = {-1, 0, 1, 0};
int dy[] = {0, 1, 0, -1};

bool isValid(vector<vector<int>> &grid,vector<vector<bool>> &visited,int x,int y){
    // add extra check according to coditions in problem
    if (x>=0 and x<grid.size() and y>=0 and y<grid[0].size() and !visited[x][y] and grid[x][y])
        return 1;
    return 0;
}

// DFS on 2D Grid
void dfsOnGrid(vector<vector<int>> &grid,vector<vector<bool>> &visited,int x,int y){
    visited[x][y] = 1;
    cout<<x<<" "<<y<<endl;
    for(int i=0;i<4;i++){
        if(isValid(grid,visited,x+dx[i], y+dy[i]))
            dfs(grid,visited,x+dx[i], y+dy[i]);
    }
}

// Function calling
int connected_component=0;
for(int i=0;i<N;i++){
    for(int j=0;j<M;j++){
        if(!visited[i][j] and grid[i][j]){
            connected_component++;
            dfsOnGrid(grid,visited,i,j);
        }
    }
}
```

snappify.io

2. BFS on Grid

04:23



Minimum distance from the source.

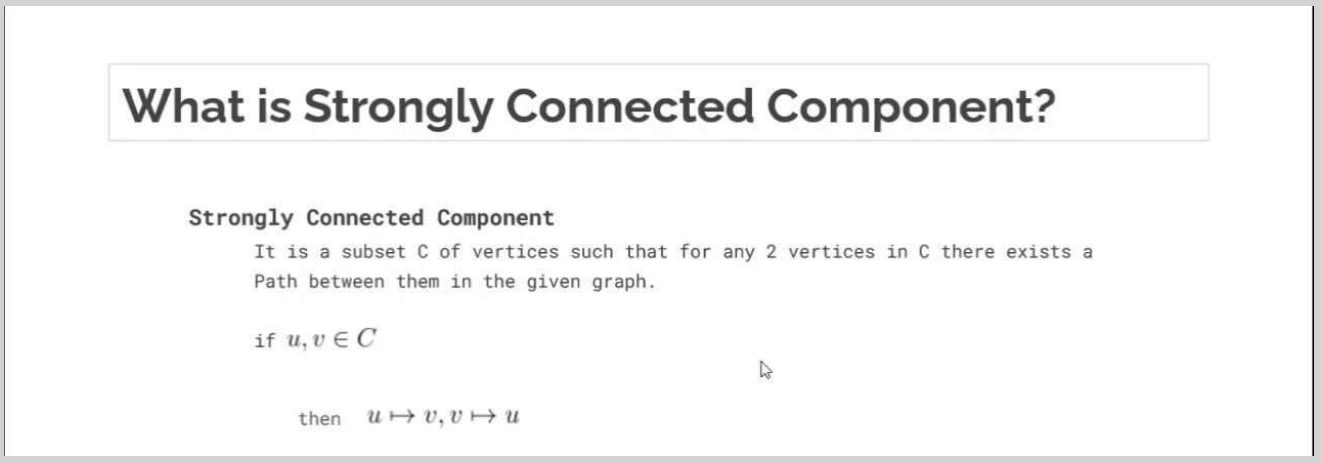
Code:

```cpp
// BFS on 2D Grid

void bfsOnGrid(vector<vector<int>> &grid,vector<vector<bool>> &visited,vector<vector<int>> &distance,int srcX, int srcY){

    pair<int,int> queue;
    q.push({srcX, srcY});
    distance[srcX][srcY] = 0;
    visited[srcX][srcY] = 1;

    while (!q.empty()){
        int x = q.front().first, y = q.front().second;
        q.pop();
        for (int i = 0; i < 4; i++){
            if (isValid(grid, visited, x+dx[i], y+dy[i])){
                dist[x+dx[i]][y+dy[i]] = dist[x][y]+1;
                vis[x+dx[i]][y+dy[i]] = 1;
                q.push({x+dx[i], y+dy[i]});
            }
        }
    }
}
```

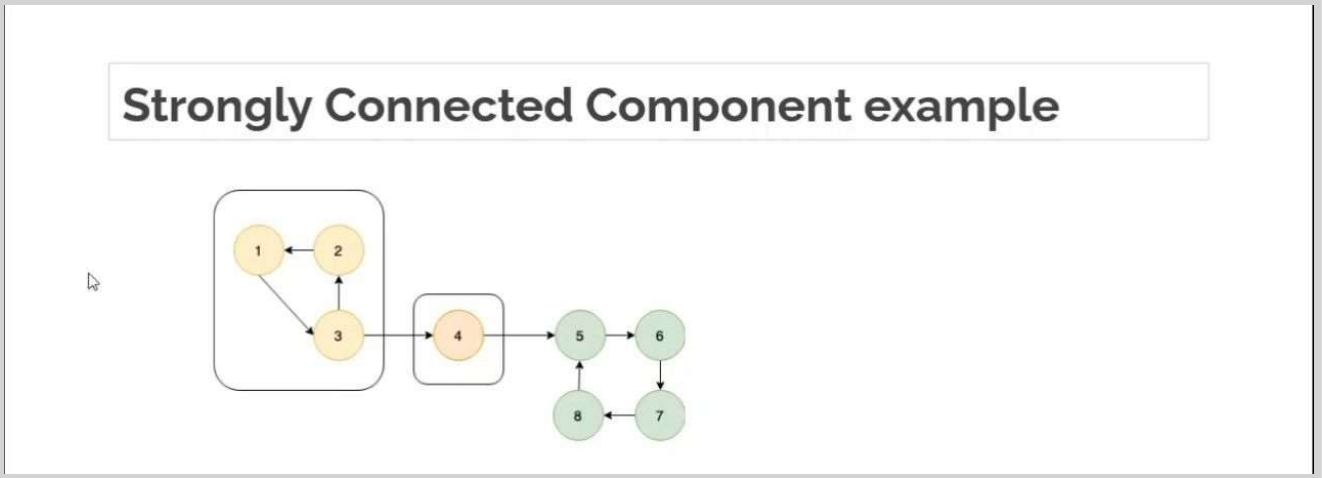
snappify.io

14. Kosaraju's Algorithm for Strongly Connected Component.

00:58

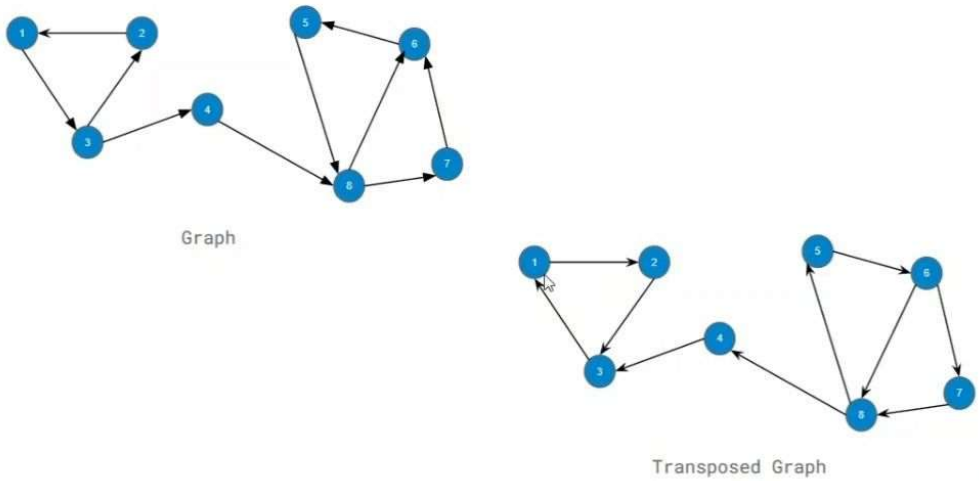


01:17



03:01

SCC & Transposed Graph



Graph

Transposed Graph

SCC not changed in Transposed graph(edges reversed)

05:01

Condensation Graph

Condensation Graph

A graph made with SCC of the original graph.
Each SCC of original graph acts as a vertex in Condensation Graph & there is an edge from SCC C_i to C_j iff there exist a node v in C_i and u in C_j and there is an edge from v to u.

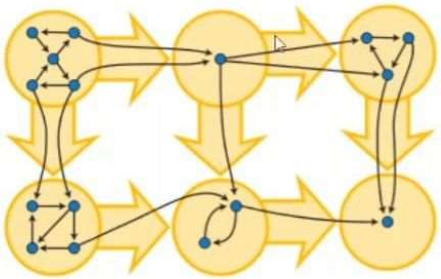


Image Source : wikipedia

07:49

Acyclic Property of Condensation Graph

Acyclic Nature

Condensation Graph does not contain any cycle in it.

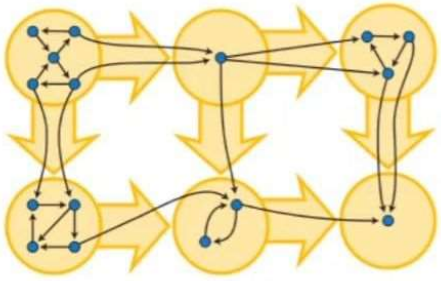
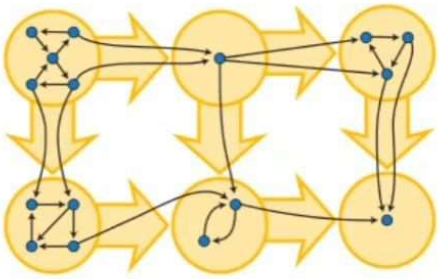


Image Source : wikipedia

12:51

Out time of each SCC & their relation

Lemma : if C_i & C_j are SCC and there exists edge from C_i to C_j then out[C_i] > out[C_j].



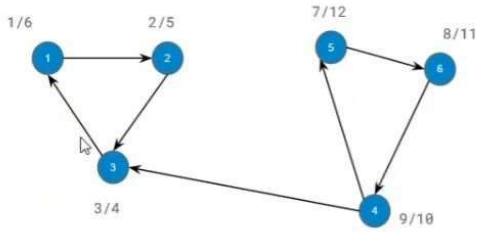
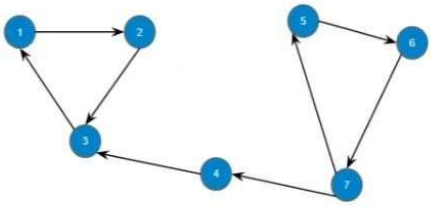


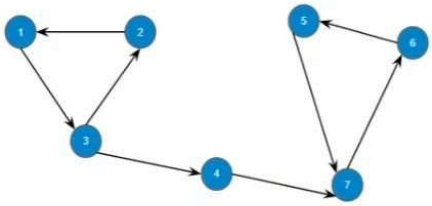
Image Source : wikipedia

19:31

Directed Acyclic Graph & Indegree 0 node

Claim
A DAG has at least 1 node with in-degree 0.



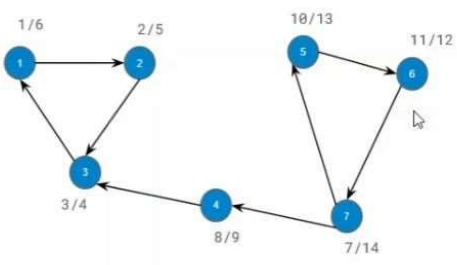


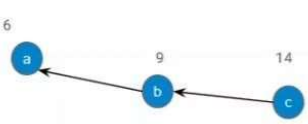
Transposed Graph

01:47

Kosaraju's algorithm

Run DFS on the graph and assign out time of each node , then sort the list by Out time of nodes.





Condensation Graph

Start DFS based on the out time because we can visit all the nodes.
in DAG the SCC with in-degree 0 will be visited at last in DFS for Sure.

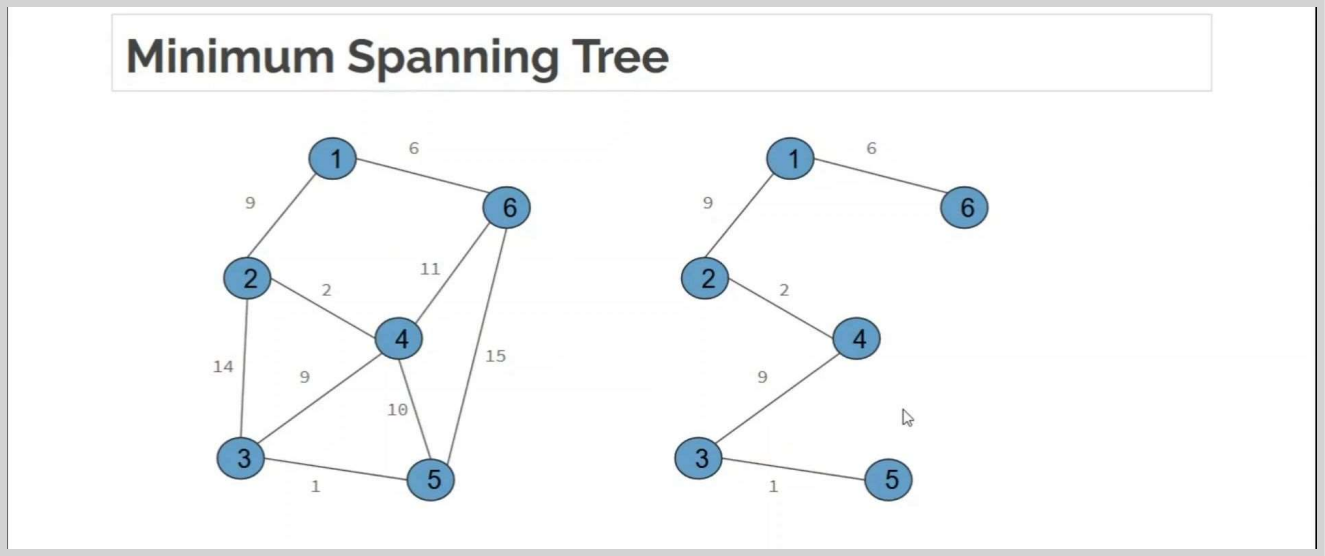
```cpp
// Kosaraju's Algorithm for Strongly Connected Component

void dfs1(vector<vector<int>> &adj, vector<bool> &visited, vector<int> &order, int node)
{
    visited[node] = true;
    for (auto child : adj[node])
        if (!visited[child])
            dfs1(child);
    order.push_back(v);
}

void dfs2(vector<vector<int>> &adj_rev, vector<bool> &visited, vector<int> &order, int node)
{
    visited[node] = true;
    component.push_back(node);
    for (auto child : adj_rev[node])
        if (!visited[child])
            dfs2(u);
}

int main()
{
    int n;
    // ... read n ...
    vector<vector<int>> adj(n), adj_rev(n);
    vector<bool> visited;
    vector<int> order, component;
    for (;;){
        int a, b;
        // ... read next directed edge (a,b) ...
        adj[a].push_back(b);
        adj_rev[b].push_back(a);
    }
    visited.assign(n, false);
    for (int i = 0; i < n; i++)
        if (!visited[i])
            dfs1(i);
    visited.assign(n, false);
    reverse(order.begin(), order.end());
    for (auto v : order)
        if (!visited[v]){
            dfs2(v);
            // ... processing component ...
            component.clear();
        }
}
```


GRAPH THEORY - 2

08:33

1. Minimum Spanning Tree

01:48

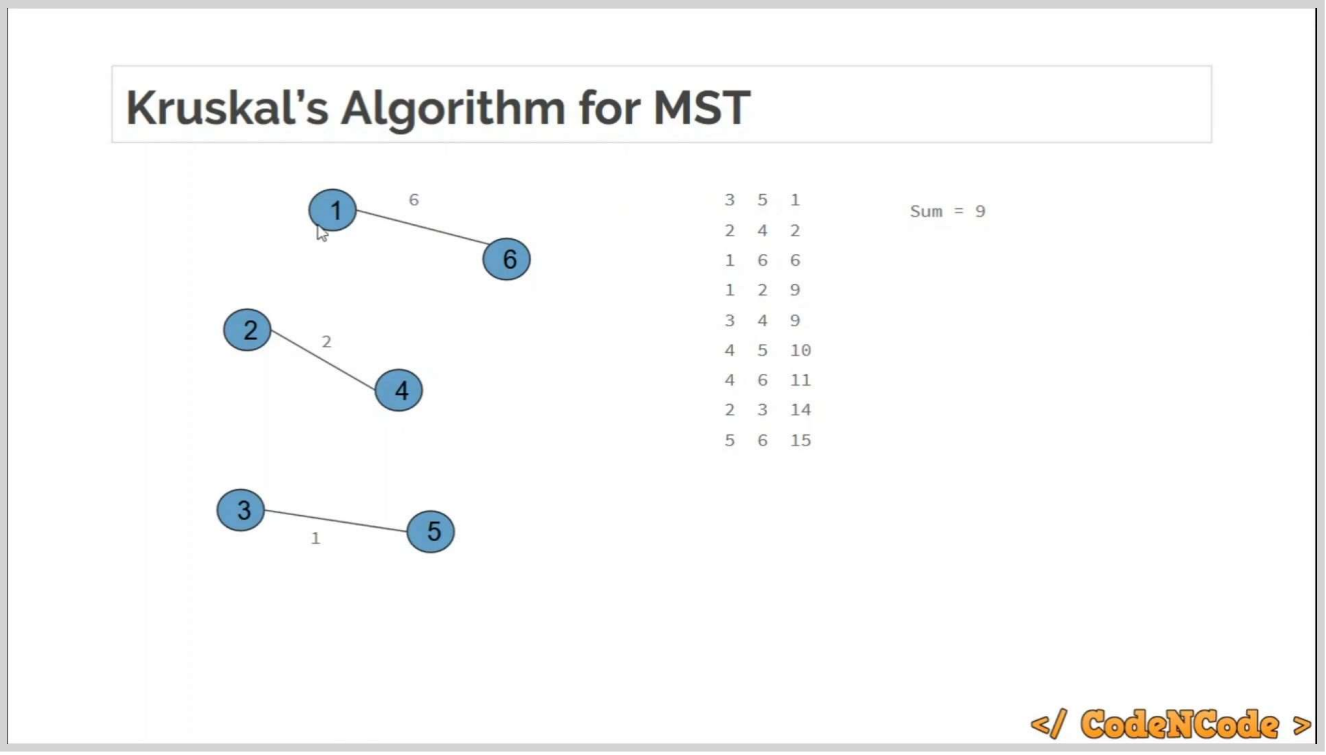


Kruskal's Algorithm for MST

Based on **DSU**

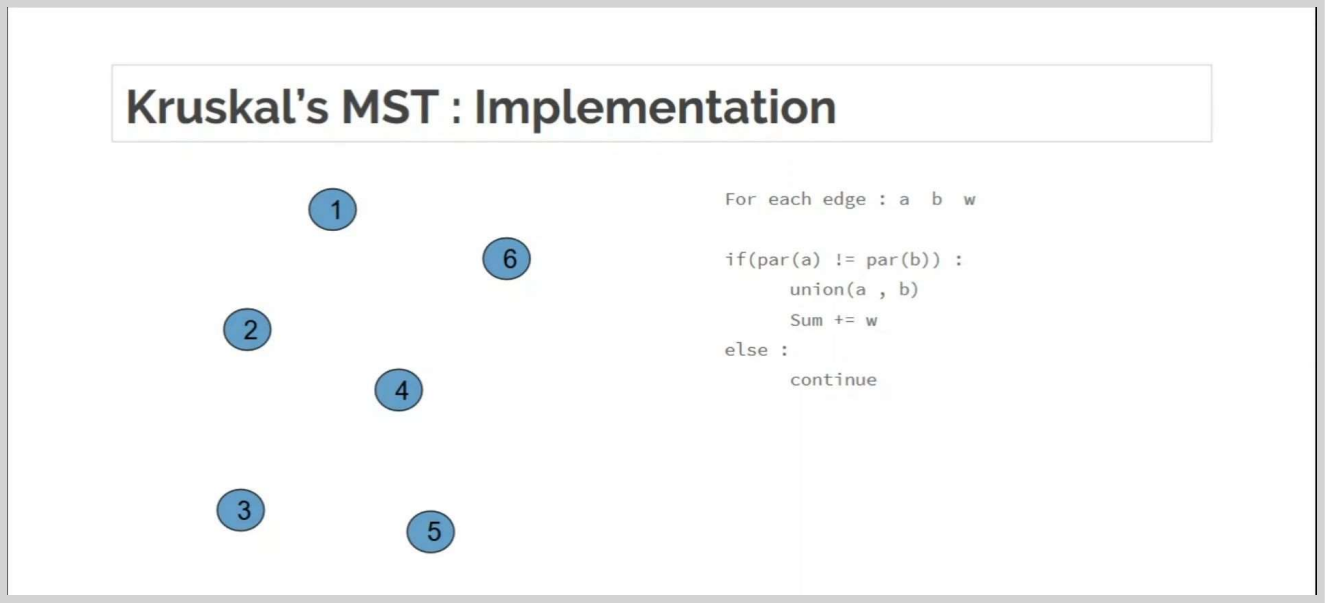
1. Sort the edges w.r.t weight
2. Start with disconnected comp and Sum=0 now start adding edges to the graph in ascending order.
3. If we got an edge that creates a cycle in most then remove max wt edge but all the nodes added were already in so add and remove this edge/skip this edge.

04:08



Implementation: using DSU

[02:15]



O(MLogN)

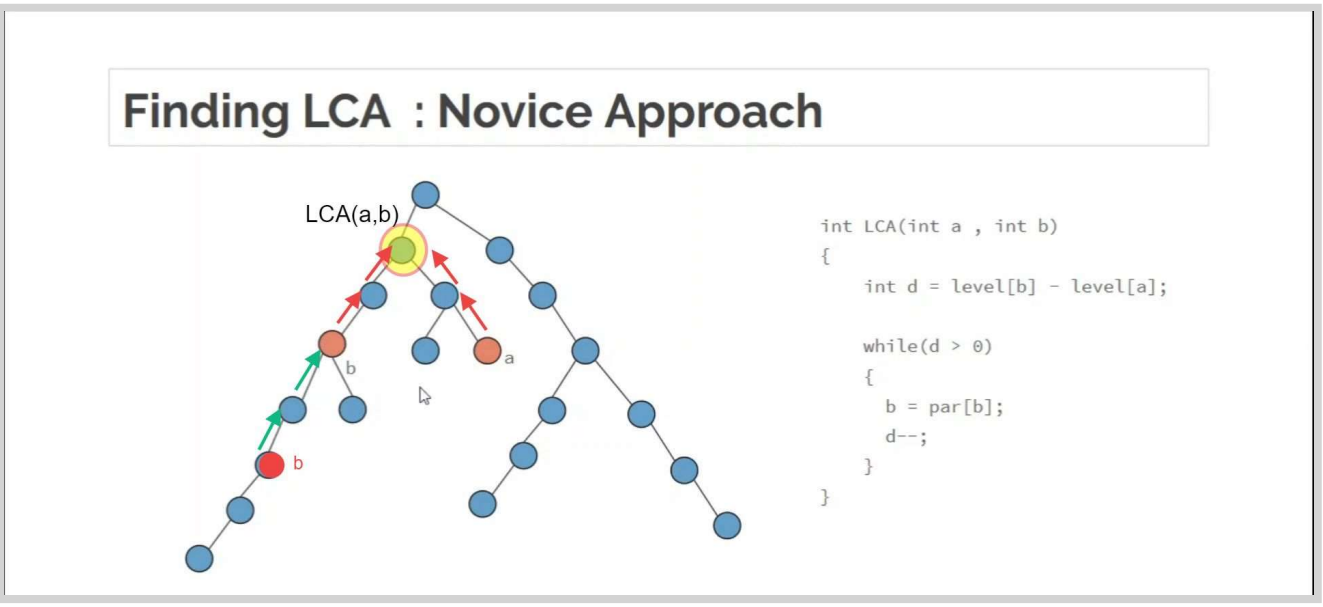
Just as in the simple version of the Kruskal algorithm, we sort all the edges of the graph in non-decreasing order of weights. Then put each vertex in its own tree (i.e. its set) via calls to the `make_set` function - it will take a total of $O(N)$. We iterate through all the edges (in sorted order) and for each edge determine whether the ends belong to different trees (with two `find_set` calls in $O(1)$ each). Finally, we need to perform the union of the two trees (sets), for which the DSU `union_sets` function will be called - also in $O(1)$. So we get the total time complexity of $O(M \log N + N + M) = O(M \log N)$.

Code:

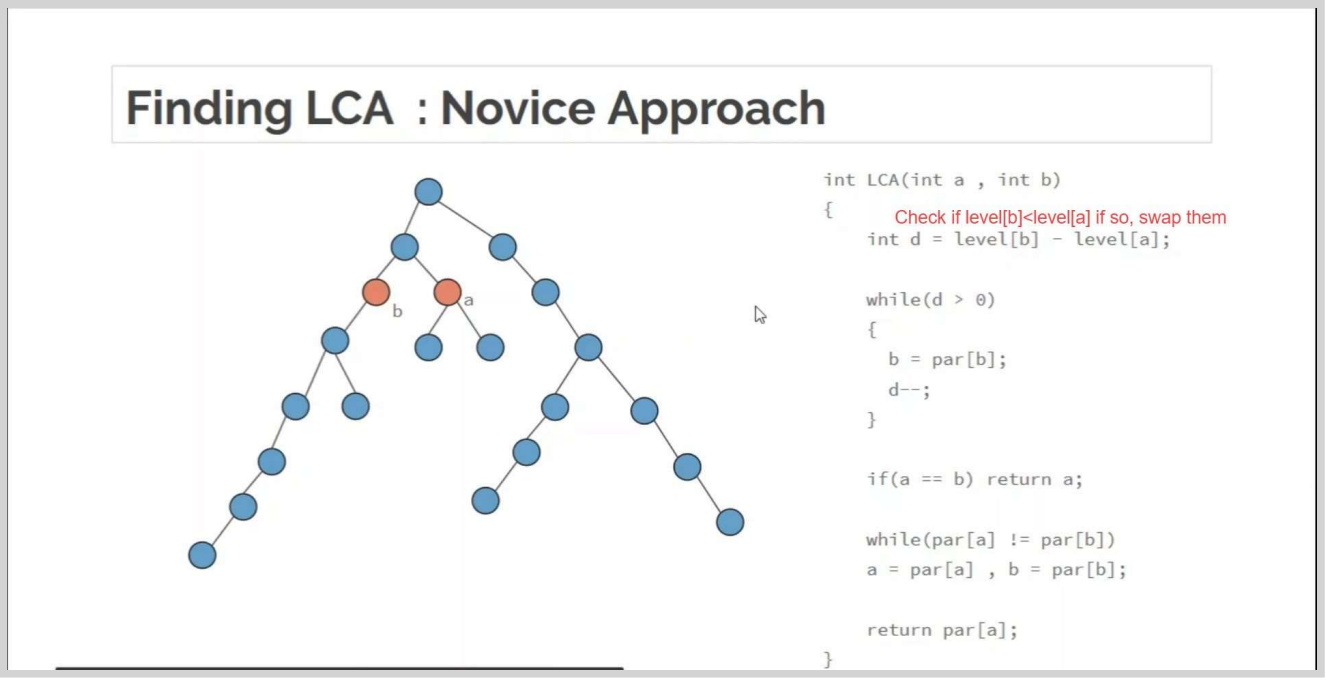
```cpp
struct Edge
{
    int u, v, weight;
    bool operator<(Edge const &other)
    {
        return weight < other.weight;
    }
};

vector<Edge> MST(vector<vector<Edge>> &adj,vector<Edge> &edges,int n){
        int n;
        int cost = 0;
        dsu d(n);
        vector<Edge> result;
        sort(edges.begin(), edges.end());
        for (Edge e : edges){
            if (d.find_set(e.u) != d.find_set(e.v)){
                cost += e.weight;
                result.push_back(e);
                d.union_sets(e.u, e.v);
            }
        }
}
```


2. Lowest Common Ancestor

05:03



09:05

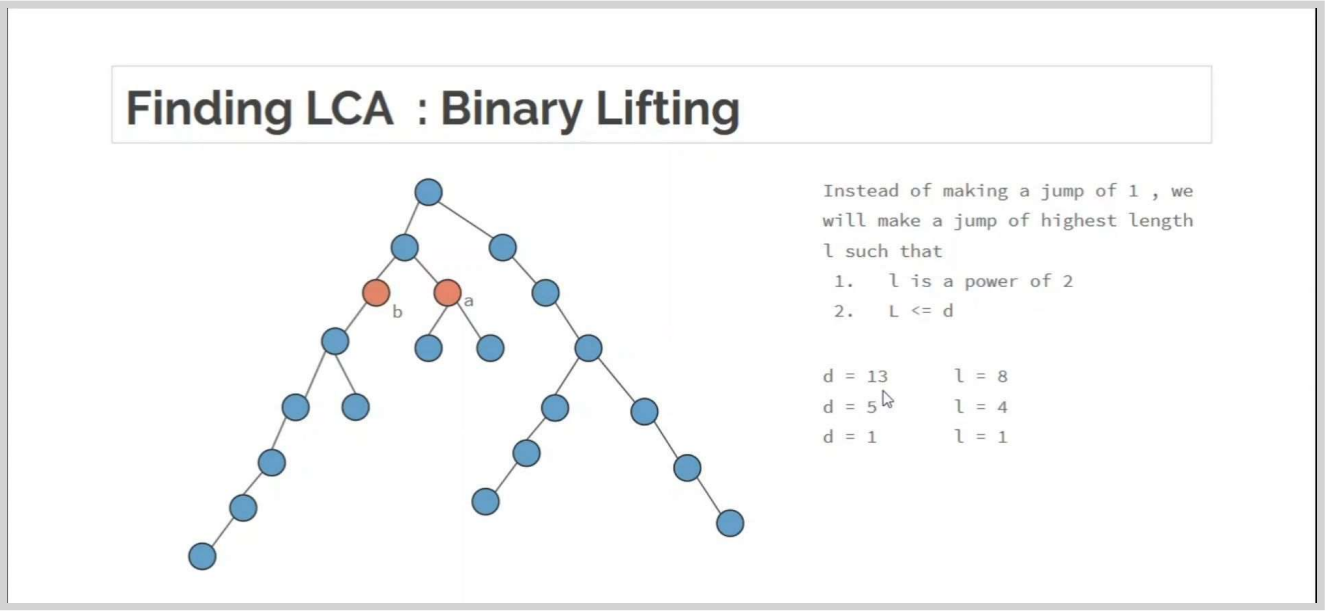


Time and Space: O(N)

Optimization: Jumps matter

Max LogN jumps

06:06



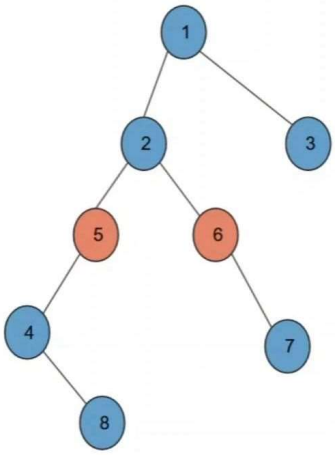
binary rep of d (13)=> 8 + 4 + 1 (1101)

We need **Jumps destination** to jump in the power of 2 similar to the parent array in the Novice approach.

Building sparse table:

10:49

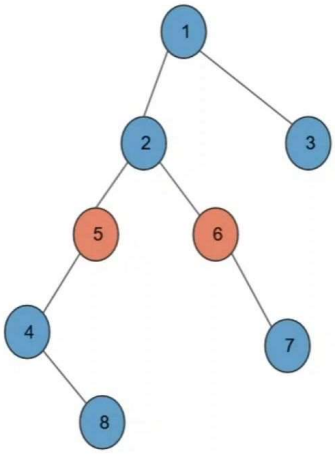
Structure of sparse table




```
N = # of nodes
maxN = log2(N)

int LCA[N+1][maxN];
Where LCA[i][j] = (2^j)th parent of i
```

11:52

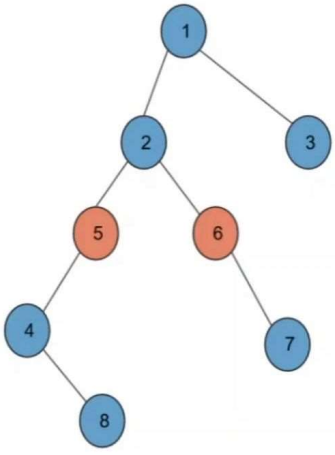
Structure of sparse table



1	-1	-1	-1	-1
2	-1	-1	-1	-1
3	-1	-1	-1	-1
4	-1	-1	-1	-1
5	-1	-1	-1	-1
6	-1	-1	-1	-1
7	-1	-1	-1	-1
8	-1	-1	-1	-1

12:54

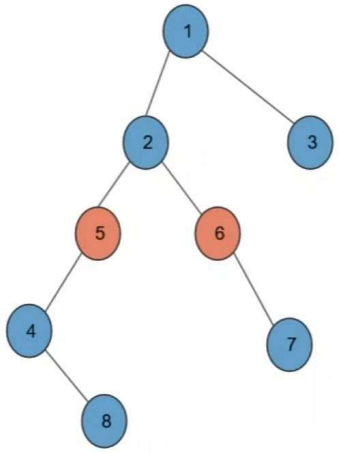
Structure of sparse table




```
void dfs(int node ,int par)
{
   LCA[node][0] = par;

   for(int child : adj[node])
   if(child != par)
   {
      dfs(child , node);
   }

}
```

15:44

Structure of sparse table



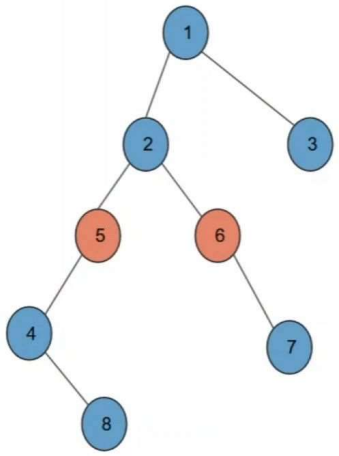
1	-1	-1	-1	-1
2	1	-1	-1	-1
3	1	-1	-1	-1
4	5	-1	-1	-1
5	2	-1	-1	-1
6	2	-1	-1	-1
7	6	-1	-1	-1
8	4	-1	-1	-1

Now using DP to fill the rest of the values

using 2^i => 2^(i-1) (parent) + 2^(i-1) (parent-parent)

22:38

Structure of sparse table

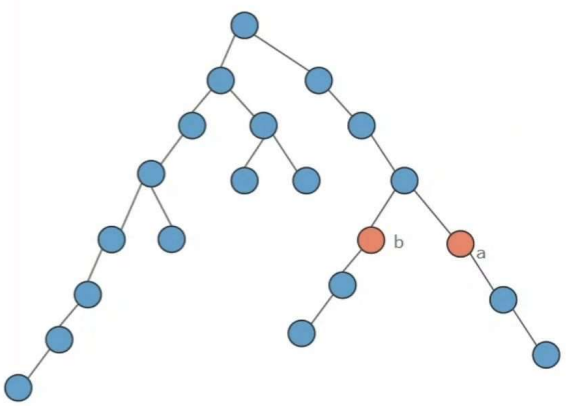



```
void init()
{
    dfs(1 , -1);
    for(int j=1; j<=maxN; j++)
    {
          for(int i=1; i<=N; i++)
          {
            if(LCA[i][j-1] != -1)
            {
                int par = LCA[i][j-1];
                LCA[i][j] = LCA[par][j-1];
            }
          }
    }
}
```

Implementation:

33:33

Finding LCA : Binary Lifting



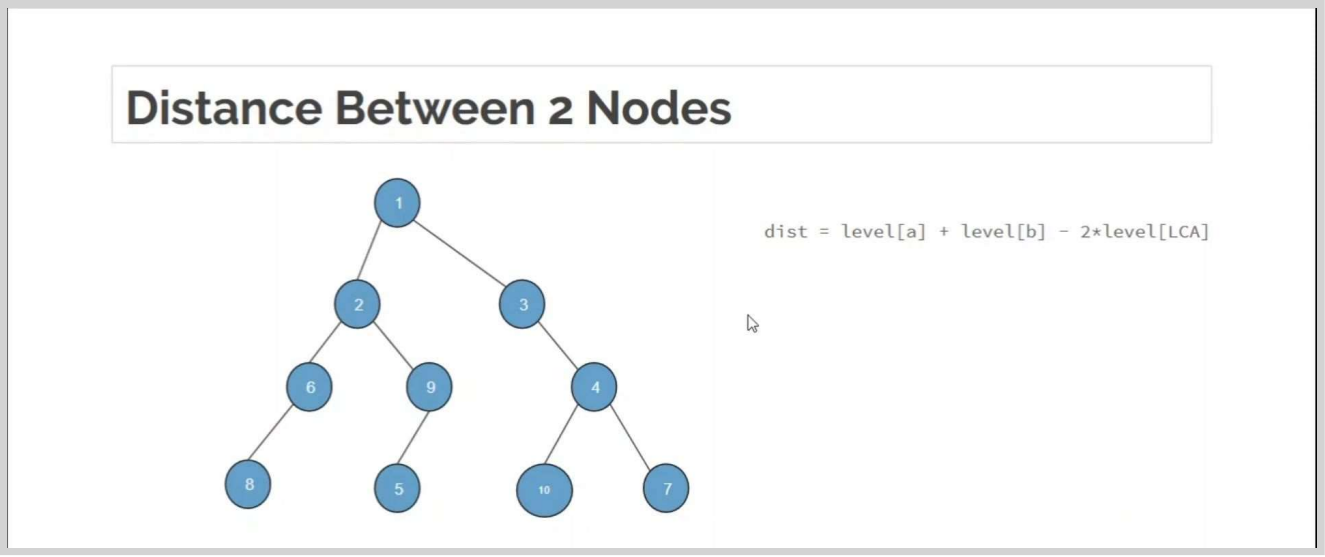

```
int LCA(int a , int b)
{
    if(level[a] > level[b]) swap(a , b);
    int d = level[b] - level[a];
    while(d > 0)
    {
        int i = log2(d);
        b = LCA[b][i];
        d -= (1 << i);
    }

    if(a == b) return a;

    for(int i=maxN; i>=0; i--)
    if(LCA[a][i] !=-1 && (LCA[a][i] != LCA[b][i]) )
    a = LCA[a][i] , b = LCA[b][i];

    return par[a];
}
```

when height becomes the same then we start making jumps of max 2-powers and don't move until they exist and their target is not equal.
Code:

```cpp
// Lowest Common Ancestor using Binary Lifting

void dfs(vector<vector<int>> &adj,vector<int> &level,vector<int> &parent,int root,int par)
{
    dp[root][0] = parent;
    // cout<<root<<" ";
    level[root] = 1 + ((par==-1)?0:level[par]);
    parent[root] = par;
    for (auto child : adj[root])
    {
        if (child != par)
            dfs(adj,child,root);
    }
}

void initialize(vector<vector<int>> &adj, int n, int maxN)
{
    dfs(adj,level,parent,0,-1);
    for (int j = 1;j<maxN; j++)
    {
        for (int i = 0;i<n; i++)
        {
            // if prev 2^(j-1) parent exist
            if (dp[i][j - 1] != -1){
                int par = dp[i][j - 1];
                dp[i][j] = dp[par][j - 1];
            }
        }
    }
}

int LCA(int u, int v, vector<int> adj[], int n, int maxN)
{
    if (level[u] < level[v])
        swap(u, v);

    int diff_level = (level[u] - level[v]);

    while (diff_level)
    {
        int i = log2(diff_level);
        u = dp[u][i];
        diff_level -= (1 << i);
    }
    if (u == v)
        return u;
    // to make it search in logN time
    for (int i = maxN; i >= 0; i--)
    {
        if ((dp[u][i] != -1) and (dp[u][i] != dp[v][i])) // must exist and have diff par
        {
            u = dp[u][i];
            v = dp[v][i];
        }
    }
    return parent[u];
}

// function calling
int maxN = log2(n);
initialize();
LCA();
```


Min Distance Between two Nodes (Tree only).

03:15



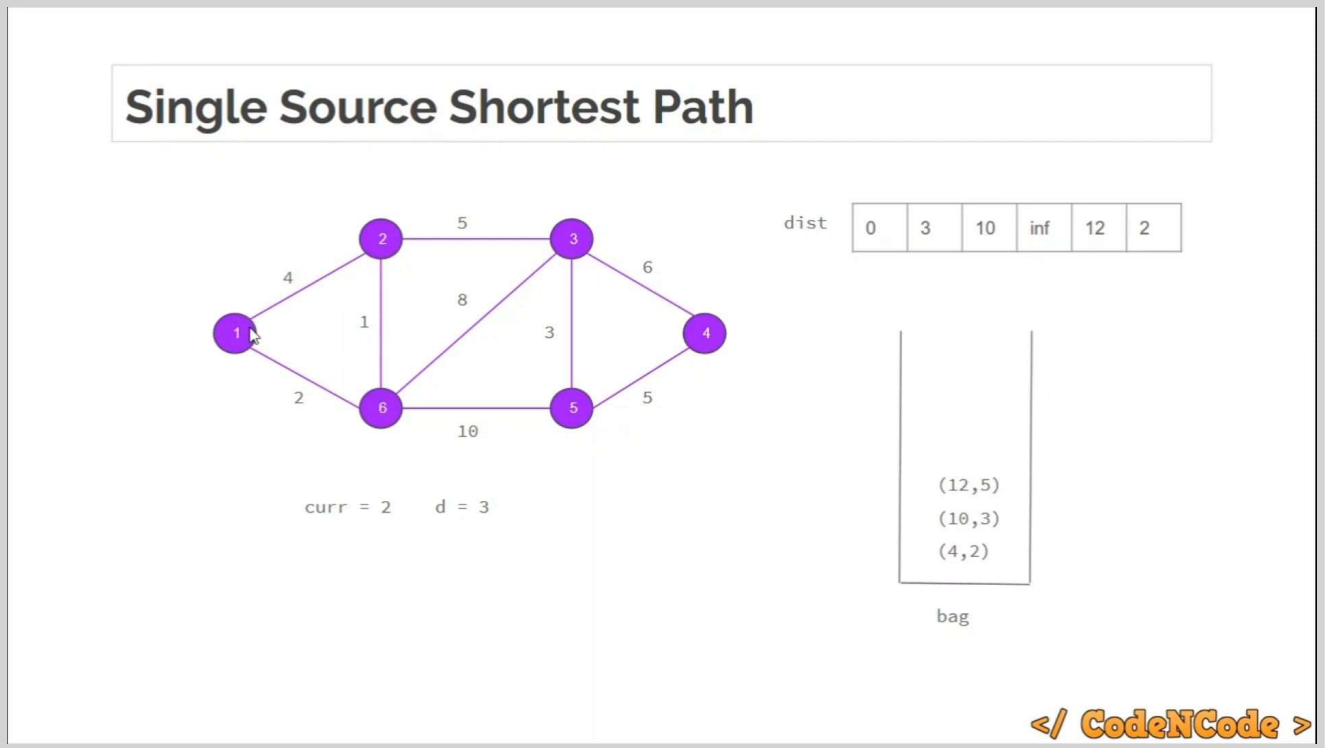
LCA => O(NLogN) using binary lifting
then Each query in O(1) time.

Single Source Shortest Path

3. Dijkstra's Algorithm

Given a graph and a source vertex in the graph, find the **shortest paths** from the source to all vertices in the given graph.
Its a Greedy algorithm,

10:13



start from the source vertex then try to minimize the dist array of all adjacent nodes from this curr node if the array is updated then push it into priority_queue.
In the end, we will get a min dist array.

** If the operation on the node is already completed don't do it again because it will not update the dist array for any of the adj nodes.

Code:

```cpp
// Dijkastra SSSP for weighted Graph

void Dijkastra(vector<vector<pii>> &adj,vector<int> &distance,vector<bool> &flag,int source,int v,int e){

    distance[source] = 0;
    // min_heap = {dis,node}
    priority_queue<pii, vector<pii>, greater<pii>> min_heap;

    min_heap.push({0, source});

    while (!min_heap.empty())
    {
        int currNode = min_heap.top().second;
        int currDis_from_source = min_heap.top().first;

        min_heap.pop();

        flag[currNode] = 1; // work of this node will be completed

        if (currDis_from_source > distance[currNode])
            continue;
        for (auto edge : adj[currNode])
        {
            if (!flag[edge.first] && (currDis_from_source + edge.second < distance[edge.first]))
            {
                distance[edge.first] = currDis_from_source + edge.second;
                min_heap.push({distance[edge.first], edge.first});
                /*
                    // store path in dijkarstra.
                        path[edge.first] = currNode;
                */
            }
        }
    }

    for (int i = 1; i <= v; i++)
    {
        if (distance[i] == 0)
            continue;
        if (distance[i] == inf)
            cout << -1 << " ";
        else
            cout << distance[i] << " ";
    }
    cout << endl;
}

// function calling
vector<int> distance(v,inf);
vector<bool> flag(v,0);
Dijkastra(adj,distance,flag,source,v,e);
//edge:u→{v,wt}

//Restore Path
vector<int> restore_path(int s, int t, vector<int> const &path)
{
    vector<int> p;

    for (int v = t; v != s; v = path[v])
        p.push_back(v);
    p.push_back(s);

    reverse(p.begin(), p.end());
    return p;
}
```

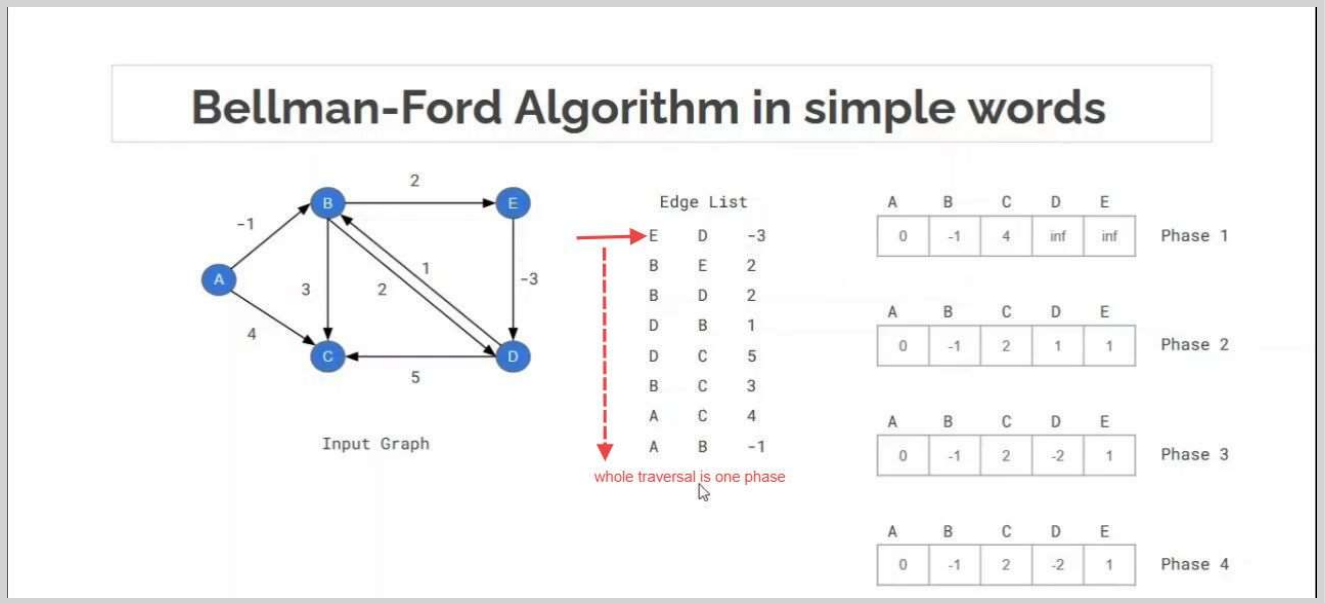
time : O(ElogV)

Drawback: Neg weight cycle

4. Bellman-Ford Algorithm

05:29

16:12



Relax weights (min dist) of the node in the dist array of v's from u's.