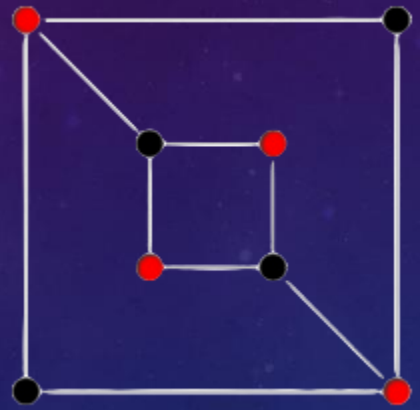
BIPARTITE GRAPH

```
bool dfs(int node, int c)
{
visited[node]=true,color[node]=c;
for(auto child: adj[node])
{
    if(!visited[child])
    {
        if(dfs(child,c^1)==false) return false;
    }
    else if(color[node]==color[child]) return false;
}
return true;
}
```

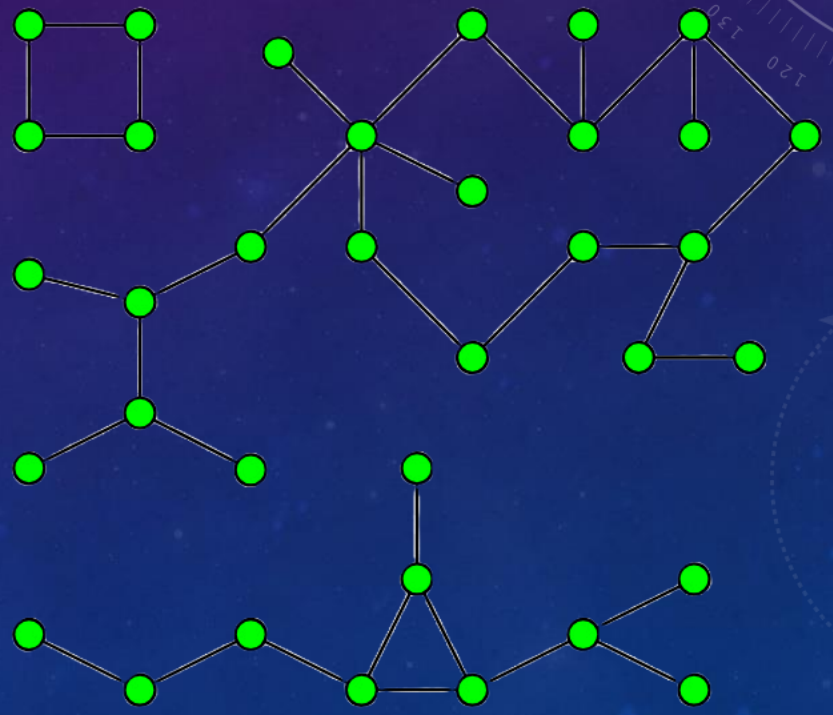


```
 bool flag=true;
 for(int i=1;i<=n;i++)
    {
        if(visited[i]==false){  if(dfs(i,0)==false) no;  }
    }
```

NO. OF CONNECTED COMPONENTS

You have to run dfs for all nodes and for each dfs mark all the child nodes as true. then dont run dfs if it is visited.
The total number of times you run dfs will be the number of connected components

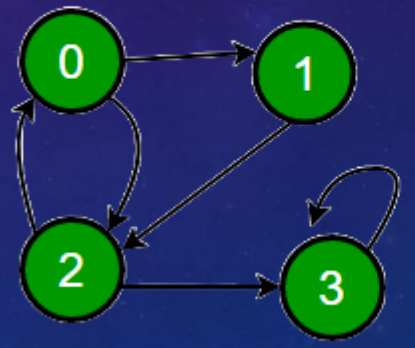
```
void dfs(int node)
{
visited[node]=true;
for(auto child: adj[node])
{
    if(!visited[child]) dfs(child);
}
}
```

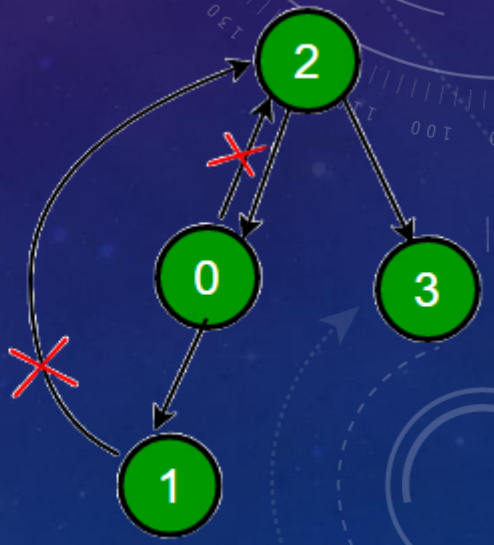


DETECT CYLES

YOU HAVE TO RUN DFS FOR ALL NODES AND IF YOU RUN DFS AND COME ACROSS A CASE WHERE THE CHILD != PARENT AND STILL IT IS VISITED YOU RETURN TRUE. FOR A PARTICULAR NODE YOU TAKE THE BITWISE OR OF ALL THE CHILDREN AND THEN RETURN THE RESULT TO THE PARENT CALL.

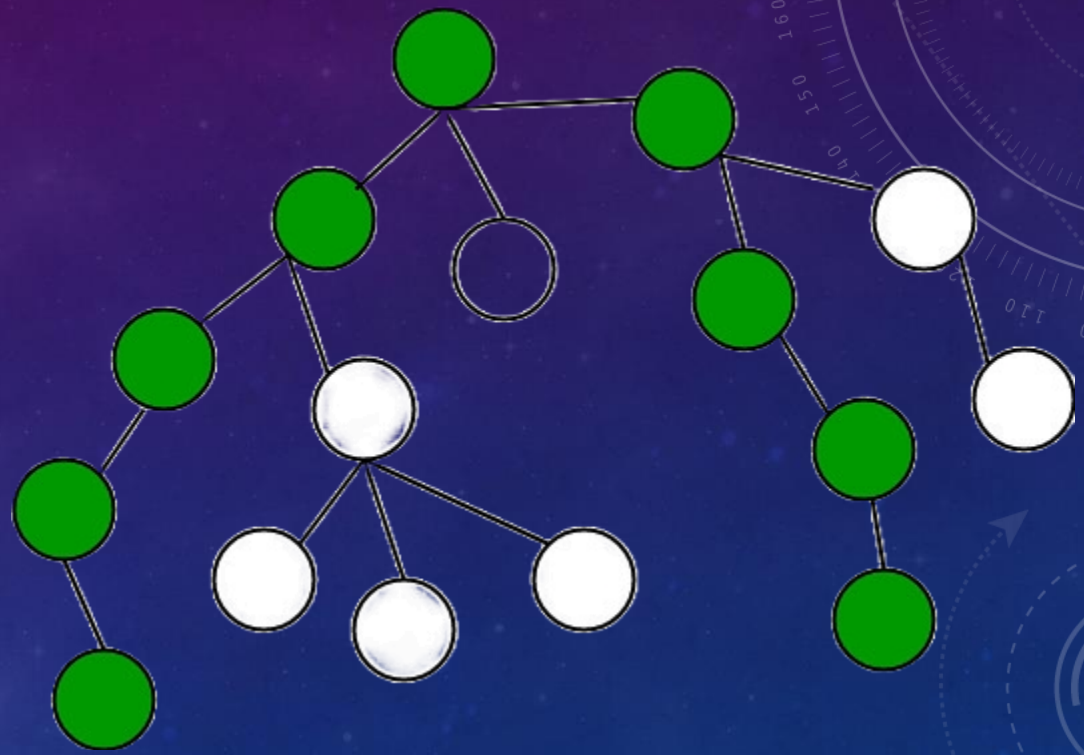
```
bool dfs(int node, int par=0)
{
visited[node] = true;
bool loop = false;
for(auto child: adj[node])
{
    if(visited[child] && child==par) continue;
    if(visited[child]) return true;
    loop|=dfs(child,node);
}
return loop;
}
```





DIAMETER OF TREE/GRAPH

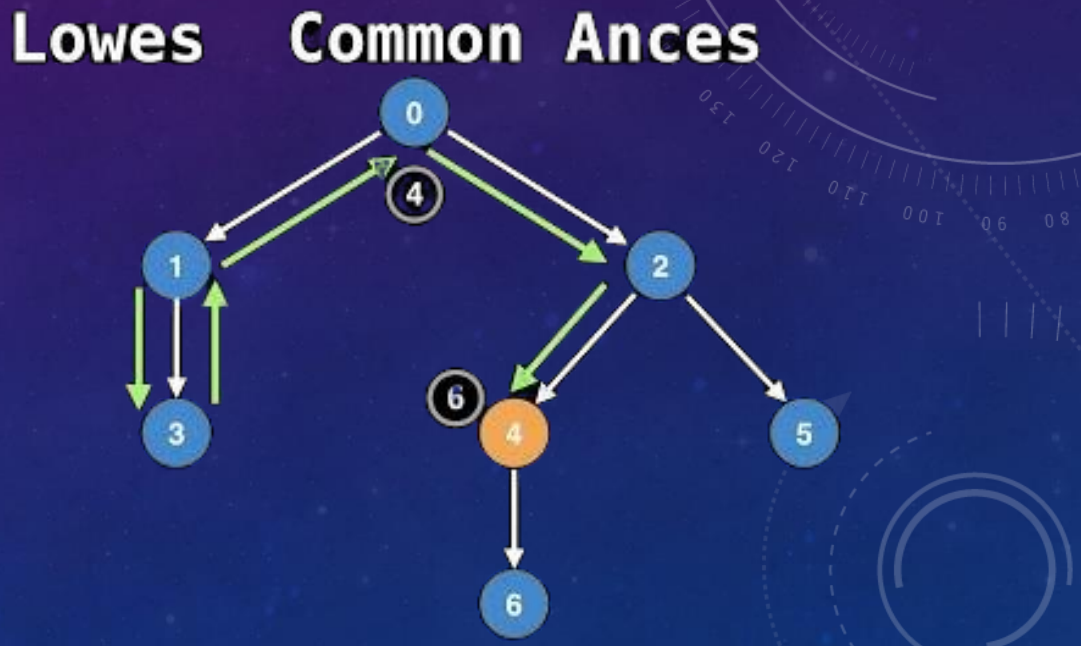
```
pair<int, int> dfs(int node)
{
    visited[node] = true;
    int len1 = 0, len2 = 0;
    for (auto child : adj[node])
    {
        if (!visited[child])
        {
            pair<int, int> childDepths = dfs(child);
            if (childDepths.first > len1)
            {
                len2 = len1;
                len1 = childDepths.first;
            }
            else if (childDepths.first > len2)
            {
                len2 = childDepths.first;
            }
        }
    }
    diameter = max(diameter, len1 + len2);
    return {1 + len1, node};
}
```

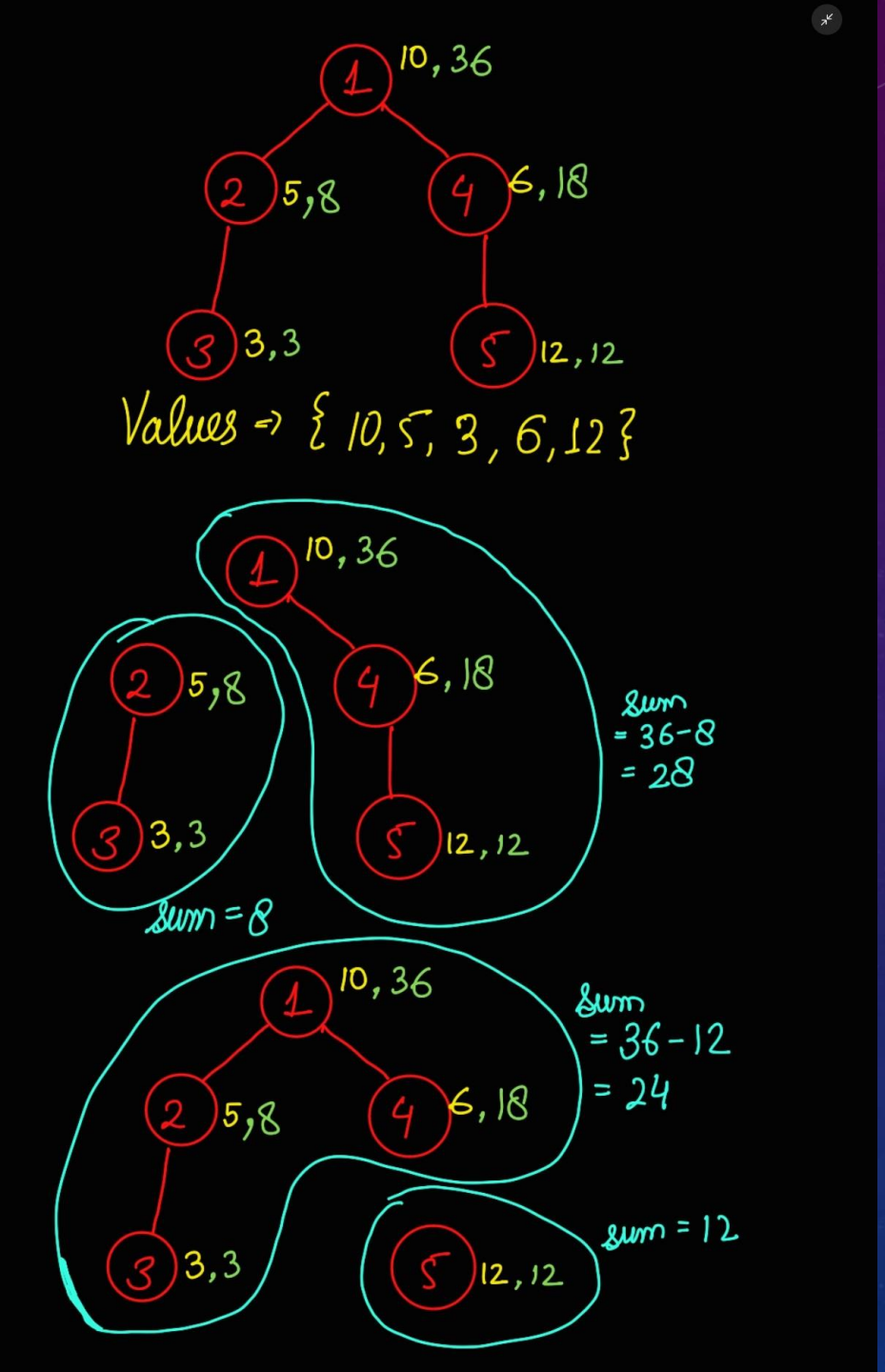


LCAO

WE RUN DFS AND STORE THE PARENT OF EVERY NODE. THEN WE GET THE PATHS TO DESIRED NODES FROM ROOT AND THEN WE RUN FOR LOOP FOR LCAO.

```
void dfs(int node, int par=0)
{
visited[node]=true;
parent[node]=par;
for(auto child: adj[node])
{
    if(!visited[child]) dfs(child,node);
}
}
vector<int> path(int node)
{
    vector<int> ans;
    while(node!=0) ans.pb(parent[node]),node=parent[node];
    reverse(all(ans));
    return ans;
}
```





EDGE DELETION

COMPUTE THE SUM OF SUBTREE AND GO TOWARDS ROOT NODE. THEN SUM OF THE TWO COMPONENTS AFTER THEIR EDGE IS DELETED WILL BE ***SUM[NODE]*** && ***SUM[ROOT]-SUM[NODE]***

```
int val[N];

void dfs(int vertex, int par){
    /** Take action on vertex after entering
     *  the vertex
     **/
    subtre_sum[vertex] += val[vertex];
    for(int child : g[vertex]){
        /** Take action on child before
         *  entering the child node
         **/
        if(child == par) continue;
        dfs(child, vertex);
        subtre_sum[vertex] += subtre_sum[child];
        /** Take action on child
         *  after exiting child node
         **/
    }
```

Topological Sorting

```
int n,m;
vector<int> g[100100];
int indegree[100100];
vector<int> topo;

void kahn(){
    queue<int> q;
    for(int i=1;i<=n;i++){
        if(indegree[i]==0){
            q.push(i);
        }
    }

    while(!q.empty()){
        int x = q.front();
        q.pop();

        topo.push_back(x);

        for(auto v:g[x]){
            indegree[v]--;
            if(indegree[v]==0){
                q.push(v);
            }
        }
    }
}
```

```
void solve(){

    cin>>n>>m;
    for(int i=0;i<m;i++){
        int a,b;
        cin>>a>>b;
        g[b].push_back(a);
        indegree[a]++;
    }
    kahn();
}
```

We can use Kahn's algorithm to get the topological ordering.
Topological ordering is for a DAG where 1 is pre-requisite for 2 and so on.

However if there is a cycle in the graph then using topological sorting the indegree can never become zero. So we can get the nodes which are involved in a cycle by simply checking for the nodes whose indegree is not zero.

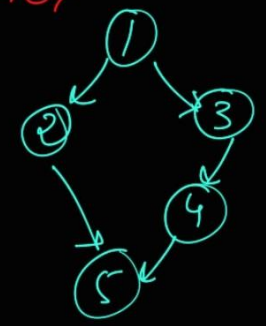
Topological Sorting

Finding the lexicographically smallest topological ordering

↳ Use minheap instead of queue

Finding the length of longest path in DAG

↳

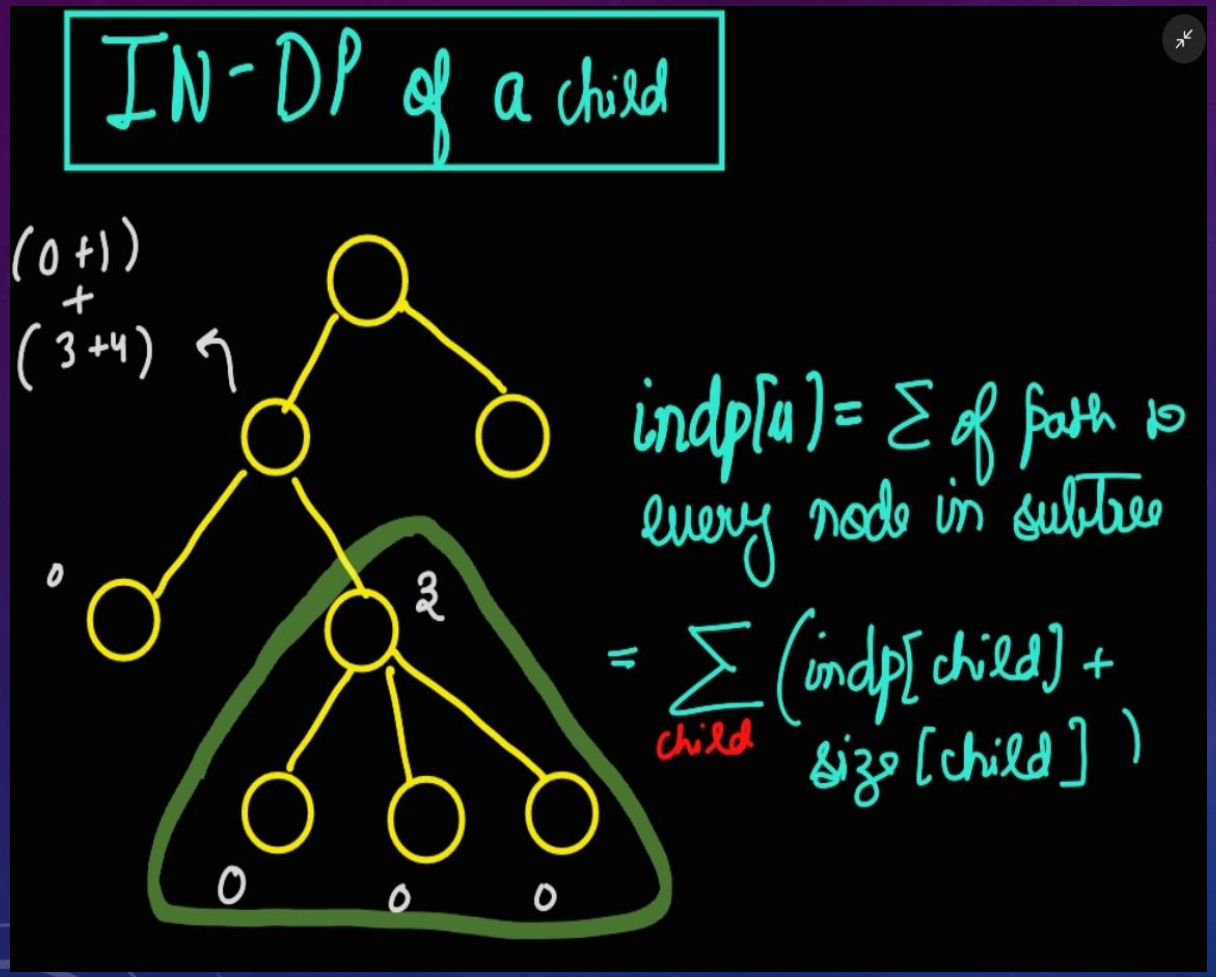


dp[i] = longest path starting from node i

dp[i] = max(1, dp[neigh] + 1)

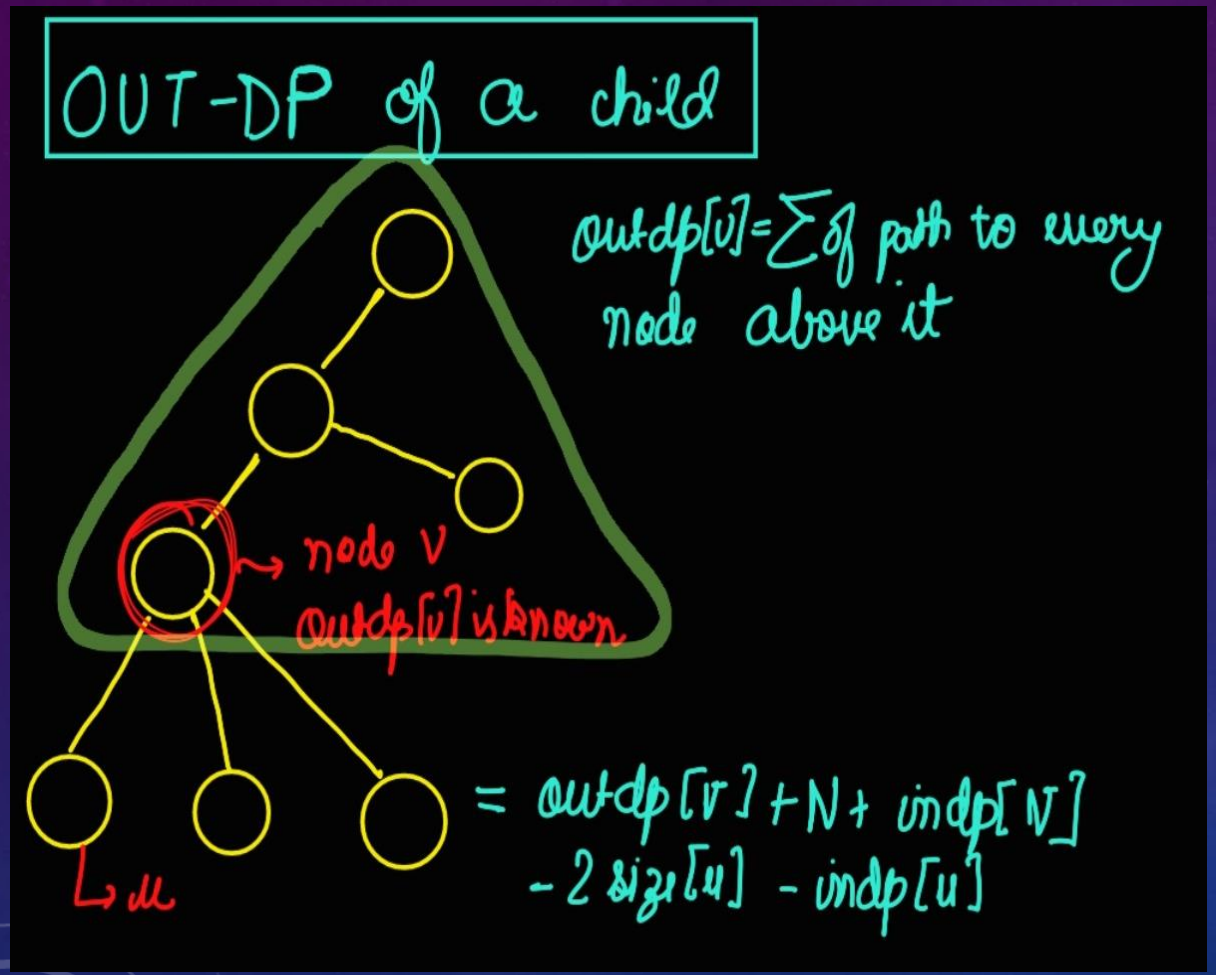
dp[5] = 1
dp[4] = 2
dp[3] = 3
dp[2] = 2
dp[1] = 4

IN DP OF A NODE




```
void dfs(int node, int par=0)
{
visited[node]=true;
size[node]=1;
indp[node]=0;
for(auto child: adj[node])
{
    if(!visited[child])
    {
        dfs(child,node);
        size[node]+=size[child];
       indp[node]+=indp[child]+size[child];
    }
}
}
```

OUT DP OF A NODE




```
void dfs(int node,int par=0)
{
visited[node]=true;
if(par==0) outdp[node]=0;
else outdp[node]=outdp[par]+indp[par]+N-2*size[node]-indp[node];
for(auto child: adj[node])
{
    if(!visited[child])
    dfs(child,node);
}
}
```

$$outdp[u] = outdp[v] + N + indp[v] - indp[u] - 2\,size[u]$$

$$\Rightarrow \underbrace{outdp[u] + indp[u]}_{\Sigma \text{ of all paths from one node to all other nodes in the tree}} = outdp[v] + indp[v] + N - 2\,size[u]$$