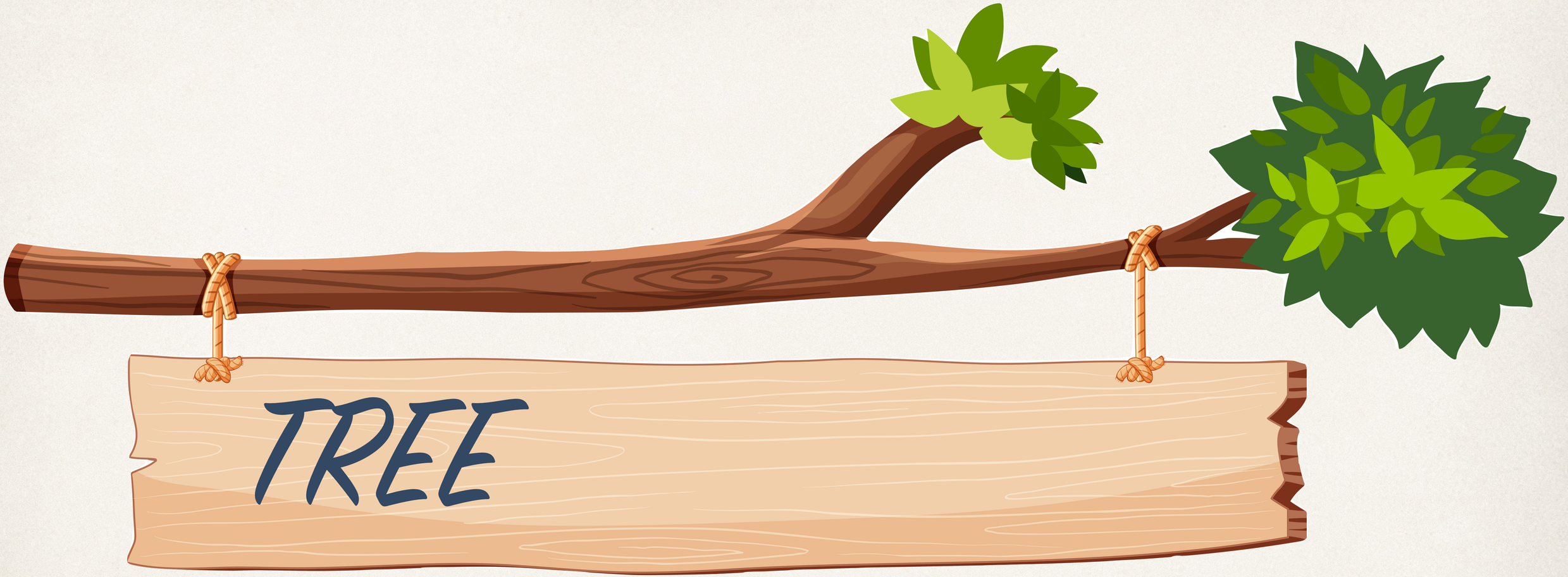

TREE

Tree

- Tree Traversal
- Depth of the node
- Height of the node
- Diameter of the tree
- Binary Lifting

Some properties and definition of the tree

- A graph is a **tree** iff it is connected and contains N nodes and N-1 edges.
- A graph is a **tree** iff every pair of nodes has exactly one simple path between them
- A graph is a **tree** iff it is connected and does not contain any cycles

General Tree terminology

A leaf of the tree is nay node in the tree with degree 1

- if the tree is rooted , the root with a single child is not typically considered a leaf but depending on the problem ,this is not always the case

A star graph has two common definitions. Try to understand what they mean-they typically appear in subtasks

- Def1: Only one node has degree greater than 1
- Only one node has degree greater than 2

A forest is a graph such that each connected component is tree.

Rooted tree terminology:

A **root** of a tree is any node of the tree that is considered to be at the ‘top’

A **parent** of node n is the first node along the path from n to the root

- A root does not have a parent , This is typically done in code by setting the parent of the root to be -1

The **ancestors** of a node are its parent and parent’s ancestors

- Typically a node is considered its own ancestors as well(such as in the subtree definiton)

The **subtree** of a node are the set of nodes have n as an ancestors

- A node is typically considered to be in its own subtree
- Note: This is easily confused with subgraphs

The depth of level of a node is its distance from the root

Tree Traversal

- We will apply the dfs here but we can modify the dfs code, the way we apply in the graph.
- Basic difference between graph and tree is that graph may contain the loops whereas the tree don’t.
- So We don’t need to maintain the visited array which we do in the case of the graph. By the way we need suitable measure that we don’t go back in the nodes
- So we have to check that the parent is not visited if its child is visited.Or in other way we can say the we spot the backtracking when the parent node is visited again
- So for initial node is 1 which is actually
- The next codes are of the dfs to calculate the height and depth of the nodes of the trees.

- We can give the default parent of first node as 0.

```
const int N=1024;
int g[N][N],depth[N],height[N];

void dfs(int vertex, int par=0){
    // Take action on vertex after entering the vertex
    for(int child: g[vertex]){
        // Take action on child before entering the child node

        depth[child]=depth[vertex]+1;

        if(child==par) continue;
        dfs(child,vertex);

        height[vertex]=max(height[vertex],height[child]+1);
        /* Take action on child after exiting
        child node*/
    }
    /**Take action on the vertex before exiting on vertex
     * **/
}
```

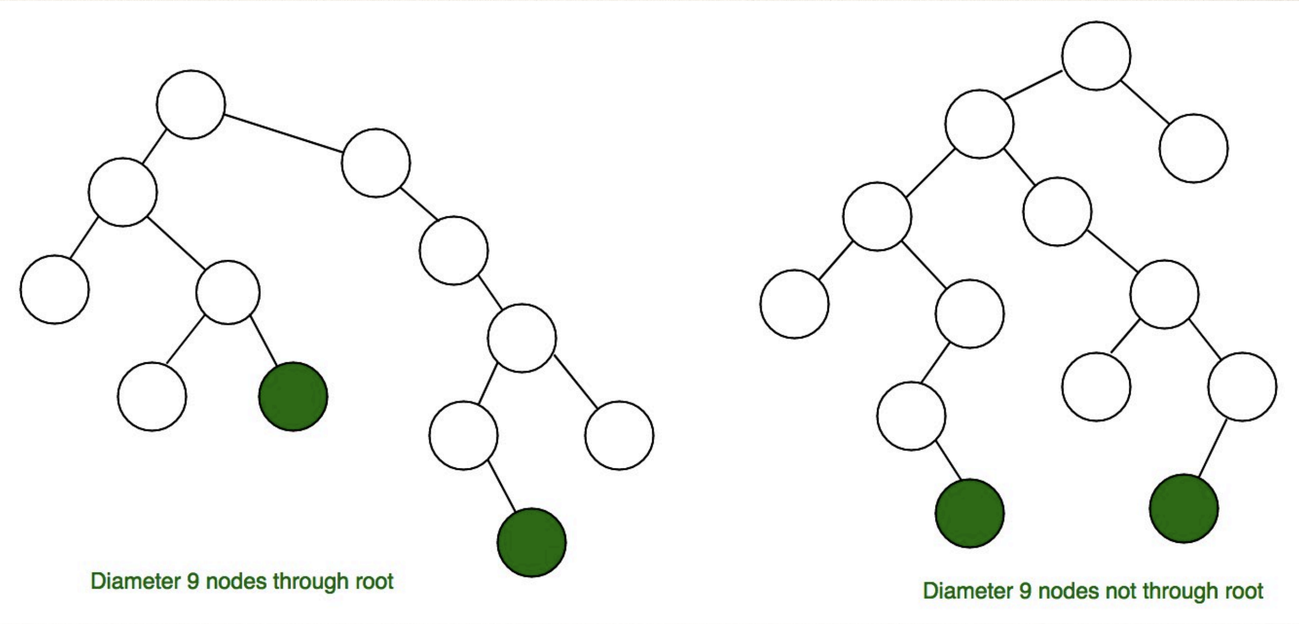
Subtree sum of the node

- For the given tree find the subtree sum for the given node
- Instead of solving subtree sum for each node we can precompute the subtree sum for the given node using the dfs
- So basically using the dfs for the given q queries the time complexity is O(n)
- Instead of without precomputing the time complexity is O(q*n);

```cpp
const int N=1e5;
int subtree_sum[N];
void dfs(int node,vector<int>adj[],int par=0){
    int vertex=node;
    subtree_sum[vertex]+=vertex;
    /*when we are entering into the child*/

    for(auto child: adj[vertex]){
        /*when we are entering into the child node*/

        if(child==par)continue;
        dfs(child,adj,vertex);
        /*when we are returning from the child node*/

        subtree_sum[vertex]+=subtree_sum[child];
    }

    /*Take action on vertex before exiting from the vertex*/
}

// same question you can practice
```

Diameter of a tree

- The diameter/width of a tree is defined as the number of nodes on the longest path between two end nodes.
- Next we will discuss two O(n) time algorithms for calculating the diameter of a tree.
- The first algorithm is based on dynamic programming, and the second algorithm uses two depth-first searches.

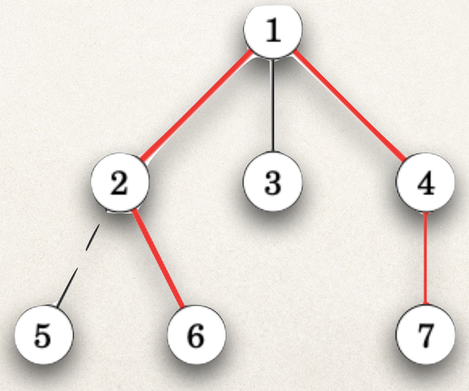


NOTE:

1. A general way to approach many tree problems is to first root the tree arbitrarily.
2. After this, we can try to solve the problem separately for each subtree.

Our first algorithm for calculating the diameter is based on this idea.

- An important observation is that every path in a rooted tree has a highest point: the highest node that belongs to the path.
- Thus, we can calculate for each node the length of the longest path whose highest point is the node.
- One of those paths corresponds to the diameter of the tree.
- For example, in the following tree, node 1 is the highest point on the path that corresponds to the diameter:

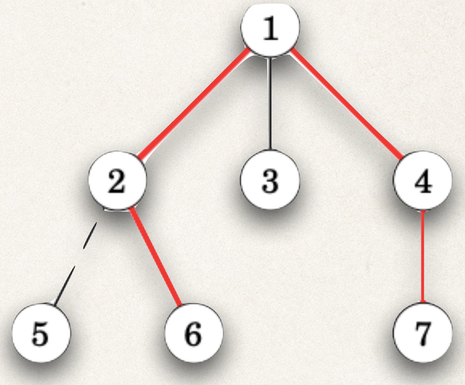


We calculate for each node x two values: ·

- **toLeaf(x)**: the maximum length of a path from x to any leaf
- **maxLength(x):** the maximum length of a path whose highest point is x

For example, in the above tree,

- toLeaf(1) = 2, because there is a path 1 → 2 → 6,
- and maxLength(1) = 4, because there is a path 6 → 2 → 1 → 4 → 7.
- In this case, maxLength(1) equals the diameter.



Dynamic programming can be used to calculate the above values for all nodes in O(n) time.
First, to calculate toLeaf(x),

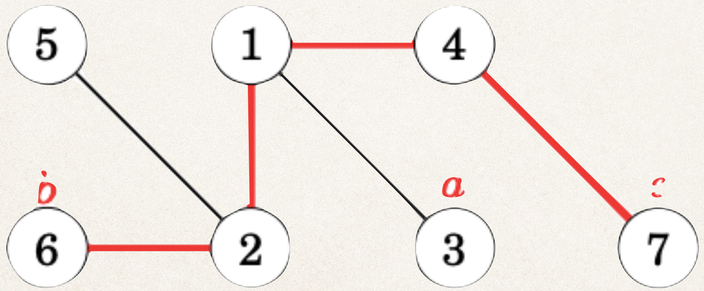
- we go through the children of x, choose a child c with maximum toLeaf(c) and add one to this value.
- Then, to calculate maxLength(x), we choose two distinct children a and b such that the sum toLeaf(a)+toLeaf(b) is maximum and add two to this sum.

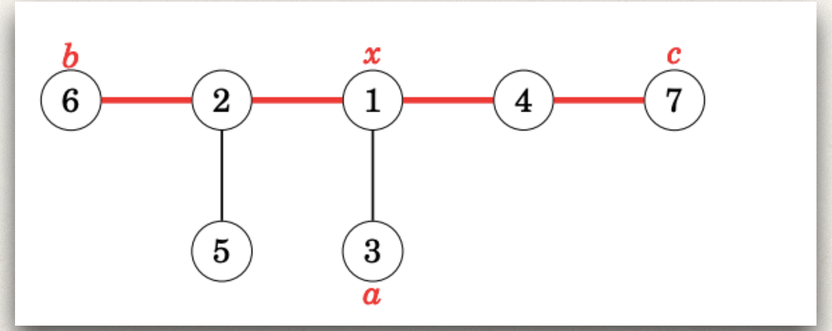
Another efficient way to calculate the diameter of a tree is based on two depth first searches.

- First, we choose an arbitrary node a in the tree and find the farthest node b from a.
- Then, we find the farthest node c from b.
- The diameter of the tree is the distance between b and c. In the following graph, a, b and c could be:

This is an **elegant** method, but why does it work?

- It helps to draw the tree differently so that the path that corresponds to the diameter is horizontal, and all other nodes hang from it:
- Node x indicates the place where the path from node a joins the path that corresponds to the diameter.
- The farthest node from a is node b, node c or some other node that is at least as far from node x. Thus, this node is always a valid choice for an endpoint of a path that corresponds to the diameter





```cpp
#include <bits/stdc++.h>
using namespace std;
const int N=1e5+10;
vector<int>g[N];
int depth[N];
void dfs(int v,int par=-1){
    for(int child: g[v]){
        if(child==par)continue;
        depth[child]=depth[v]+1;
        dfs(child,v);
    }
}
Void input(){
  int n;
    cin>>n;
    for(int i=0;i<n-1;i++){
        int x,y;
        cin>>x>>y;
        g[x].push_back(y);
        g[y].push_back(x);
    }
}
```

```cpp
int32_t main(){
    input();
    dfs(1);
    int mx_depth= -1;
    int mx_d_node;
    for(int i=1;i⩽n;i++){
        if(mx_depth<depth[i]){
            mx_depth= depth[i];
            mx_d_node=i;
        }
        depth[i]=0;
    }
    dfs(mx_d_node);
    mx_depth= -1;
    for(int i=1;i⩽n;i++){
        if(mx_depth<depth[i]){
            mx_depth= depth[i];
        }
    }
    cout<<mx_depth<<endl;
    return 0;
}
```

Lowest common Ancestor

Given a tree G. Given queries of the form (v_1, v_2), for each query you need to find the lowest common ancestor (or least common ancestor), i.e. a vertex v that lies on the path from the root to v_1 and the path from the root to v_2, and the vertex should be the lowest.

In other words, the desired vertex v is the most bottom ancestor of v_1 and v_2.

- It is obvious that their lowest common ancestor lies on a shortest path from v_1 and v_2.
- Also, if v_1 is the ancestor of v_2, v_1 is their lowest common ancestor.

Before answering the queries, we need to **preprocess** the tree.

Euler Tower

We make a DFS traversal starting at the root and we build a list euler which stores the order of the vertices that we visit (a vertex is added to the list when we first visit it, and after the return of the DFS traversals to its children).
This is also called an Euler tour of the tree.

It is clear that the size of this list will be O(N). We also need to build an array first $[0..,N-1]$. which stores for each vertex i its first occurrence in euler.
That is, the first position in euler such that euler$[first[i]] = i$.
Also by using the DFS we can find the height of each node (distance from root to it) and store it in the array height$[0..,N-1]$.

So how can we answer queries using the Euler tour and the additional two arrays?

Suppose the query is a pair of v_1 and v_2.

Consider the vertices that we visit in the Euler tour between the first visit of v_1 and the first visit of v_2.

It is easy to see, that the LCA(v_1, v_2) is the vertex with the lowest height on this path.

We already noticed, that the LCA has to be part of the shortest path between v_1 and v_2.

Clearly it also has to be the vertex with the smallest height.

And in the Euler tour we essentially use the shortest path, except that we additionally visit all subtrees that we find on the path.

But all vertices in these subtrees are lower in the tree than the LCA and therefore have a larger height.

So the LCA(v_1 , v_2) can be uniquely determined by finding the vertex with the smallest height in the Euler tour between first(v_1) and first(v_2).

The tour starting at vertex 6 and ending at 4 we visit the vertices [6, 2, 1, 3, 1, 4].
Among those vertices the vertex 1 has the lowest height, therefore LCA(6, 4) = 1.

To recap:

- To answer a query we just need to find the vertex with smallest height in the array euler in the range from first[v_1] to first[v_2].
- Thus, the LCA problem is reduced to the RMQ problem (finding the minimum in an range problem).
- Using Sqrt-Decomposition, it is possible to obtain a solution answering each query in $O(\sqrt{N})$ with preprocessing time of O(N)
- Using a Segment Tree you can answer each query in O(logN) with preprocessing in O(N) time.
  Since there will almost never be any update to the stored values, a Sparse Table might be a better choice, allowing O(1) query answering with O(N log N) build time.

```
struct LCA
{
    vector<int> height, euler, first, segtree;
    vector<bool> visited;
    int n;
    LCA(vector<vector<int>> &adj, int root = 0)
    {
        n = adj.size();
        height.resize(n);
        first.resize(n);
        euler.reserve(n * 2);
        visited.assign(n, false);
        dfs(adj, root);
        int m = euler.size();
        segtree.resize(m * 4);
        build(1, 0, m - 1);
    }
    dfs();
    build();
    lca();
}
```

```cpp
int lca(int u, int v)
{
    int left = first[u], right = first[v];
    if (left > right)
        swap(left, right);
    return query(1, 0, euler.size() - 1, left, right);
}
```

```cpp
void dfs(vector<vector<int>> &adj, int
node, int h = 0)
{
    visited[node] = true;
    height[node] = h;
    first[node] = euler.size();
    euler.push_back(node);
    for (auto to : adj[node])
    {
        if (!visited[to])
        {
            dfs(adj, to, h + 1);
            euler.push_back(node);
        }
    }
}
```

```
void build(int node, int b, int e)
{
    if (b == e)
    {
        segtree[node] = euler[b];
    }
    else
    {
        int mid = (b + e) / 2;
        build(node << 1, b, mid);
        build(node << 1 | 1, mid + 1, e);
        int l = segtree[node << 1]
        int r = segtree[node << 1 | 1];
        segtree[node] = (height[l] < height[r]) ? l : r;
    }
}
```

```
int query(int node, int b, int e, int L, int R)
{
    if (b > R || e < L)
        return -1;

    if (b ⩾ L && e ⩽ R)
        return segtree[node];

    int mid = (b + e)/2;

    int left = query(node << 1, b, mid, L, R);

    int right = query(node << 1 | 1, mid + 1, e, L, R);
    if (left == -1)
        return right;
    if (right == -1)
        return left;
    return height[left] < height[right] ? left : right;
}
```

Important Links

- A beautiful technique to solve the xor related Problem codeforces