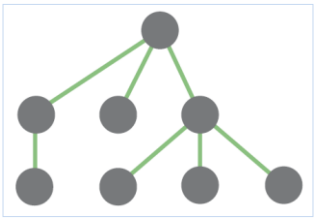
10 TREE ALGORITHMS

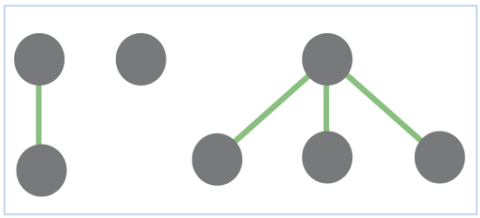
Introduction

A **tree** is an undirected graph in which any two vertices are connected by exactly one path, or equivalently a **connected acyclic undirected graph** (a connected graph with no cycles).

A disjoint collection of trees is called a **forest**.



Tree

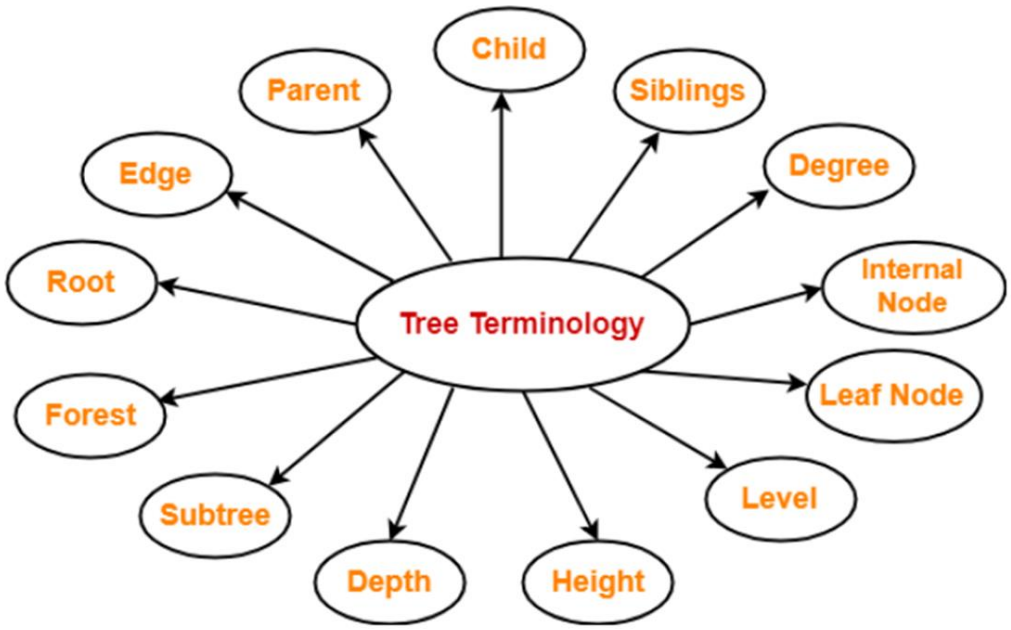


Forest

Tree Properties

- Tree is a **non-linear** data structure which organizes data in a **hierarchical** structure.
- There is **one and only one path** between every pair of vertices in a tree.
- A tree with n vertices has exactly **n-1 edges**. Any connected graph with n vertices and (n-1) edges is a tree.
- A graph is a tree if and only if it is **minimally connected** (if removal of any one edge from it disconnects the graph).

Tree Terminology



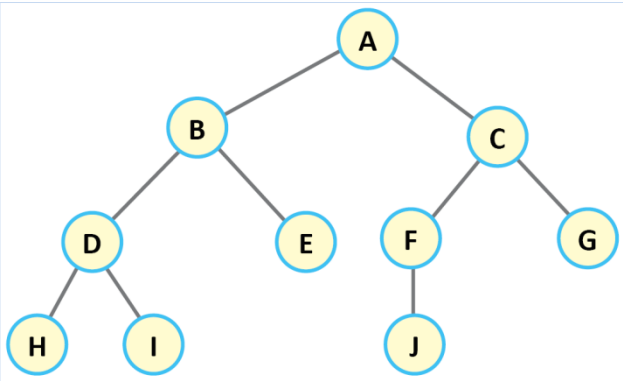
Tree Traversals

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges we always start from the **root** node.

DFS Traversals:

- Preorder (Root, Left, Right)
- Inorder (Left, Root, Right)
- Postorder (Left, Right, Root)

BFS or Level Order Traversal



- Preorder: A B D H I E C F J G
- Inorder: H D I B E A J F C G
- Postorder: H I D E B J F G C A
- Levelorder: A B C D E F G H I J

Example 1: Tree Traversals

Print the pre-order, in-order, post-order and level-order traversals of a tree with N nodes. Nodes are conveniently numbered 1 … N and node 1 is the root. The first line of input contains N. Each of following lines represents an edge with two integers. During the traversal, visit the next child with the smallest ID before the other children.

Sample Input

```
1 1
1 2
3 1
6 2
3 8
3 7
7 11
5 2
2 4
10 5
5 9
```

Sample Output

```
Preorder: 1 2 4 5 9 10 6 3 7 11 8
Inorder: 4 2 9 5 10 6 1 11 7 3 8
Postorder: 4 9 10 5 6 2 11 7 8 3 1
Levelorder: 1 2 3 4 5 6 7 8 9 10 11
```

```
// Binary Tree Traversals using an adjacency set.
#include <iostream>
#include <vector>
#include <set>
#include <queue>

using namespace std;

int N;
vector<set<int>> tree; //adjacency set
//-----------------------------
void readInput()
{
	cin >> N;
	tree.resize(N + 1);
	for (int i = 0; i < N - 1; i++)
	{
		int u, v;
		cin >> u >> v;
		tree[u].insert(v);
		tree[v].insert(u);
	}
}
//-----------------------------
void preOrder(int u, int p)
{
	cout << u << " ";
	for (int v : tree[u])
	{
		if (v == p)
			continue;
		preOrder(v, u);
	}
}
//-----------------------------
void inOrder(int u, int p)
{
	auto it = tree[u].begin();
	while (it != tree[u].end())
	{
		if (*it == p)
		{
			it++;
			continue;
		}
		inOrder(*it, u);
		it++;
		break;
	}
	cout << u << " ";
	if (it != tree[u].end() && *it != p)
		inOrder(*it, u);
}

//-----------------------------
void postOrder(int u, int p)
{
	auto it = tree[u].begin();
	while (it != tree[u].end())
	{
		if (*it == p)
```

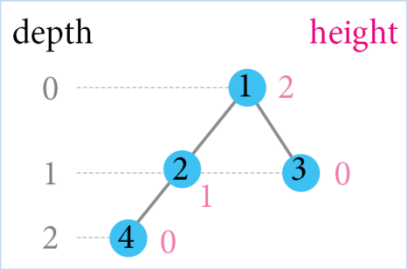
```
                {
                        it++;  continue;
                }
                postOrder(*it, u);    it++;
        }
        cout << u << " ";
}
//-----------------------------
void levelOrder(int root)
{
        vector<bool> visited(N + 1, false);
        queue<int> q;
        q.push(root);
        visited[root] = true;

        while (!q.empty())
        {
                int u = q.front();
                q.pop();
                cout << u << " ";
                for (int v : tree[u])
                {
                        if (visited[v])
                                continue;
                        visited[v] = true;
                        q.push(v);
                }
        }
}
//-----------------------------
int main()
{
        readInput();
        preOrder(1, 1);
        cout << endl;
        inOrder(1, 1);
        cout << endl;
        postOrder(1, 1);
        cout << endl;
        levelOrder(1);
}
```

Height and Depth of Nodes

In a tree structure the **height** of a node is the number of edges on the longest path from the node to a leaf.

The **depth** of a node is the number of edges from the node to the tree's root node.

The height of leaves is 0, and depth of the root is 0.



Example 2: Heights and Depths

Find the height and depth of each node of a tree with N nodes. Nodes are conveniently numbered 1 … N and node 1 is the root.

Sample Input

```
4
1 2
3 1
2 4
```

Sample Output

```
2 1 0 0
0 1 1 2
```

```
// Height and Depth Tree.cpp : DFS solution. df1 and df2 can be combined.
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int N;
vector<vector<int>> adjList;
vector<int> resVec;
//---------------------------------
void readInput()
{
	cin >> N;
	adjList.resize(N + 1);
	for (int i = 0; i < N - 1; i++)
	{
		int u, v;
		cin >> u >> v;
		adjList[u].push_back(v);
		adjList[v].push_back(u);
	}
}
//---------------------------------
//Find heights. If u is a leaf, height is 0.
//Otherwise maximum height of children + 1
void dfs1(int u, int p)
```

```
{
	int temp = 0;
	for (int v : adjList[u])
	{
		if (v == p)
			continue;
		dfs1(v, u);
		temp = max(temp, resVec[v] + 1);
	}

	resVec[u] = temp;
}
//---------------------------------
void dfs2(int u, int p)
{
	for (int v : adjList[u])
	{
		if (v == p)
			continue;
		resVec[v] = resVec[u] + 1;
		dfs2(v, u);
	}
}
//---------------------------------
//Print and reset resVec
void printResult()
{
	for (int i = 1; i <= N; i++)
	{
		cout << resVec[i] << " ";
		resVec[i] = 0;
	}
	cout << endl;
}
//---------------------------------
int main()
{
	readInput();

	resVec.resize(N + 1, 0);
	dfs1(1, 1);
	printResult(); //Print heights of nodes

	dfs2(1, 1);
	printResult(); //Print depths of nodes
}
```

Exercise 1: Heights and Depths

1. Calculate the depths with a BFS function.
2. Calculate Heights and Depths in one DFS function.

Subtrees

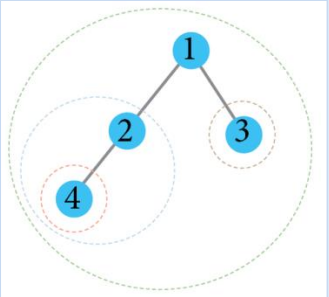
A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

Example 3: Sizes of subtrees

Find size of each subtree (number of nodes) of a tree with N nodes. Nodes are conveniently numbered 1 … N and node 1 is the root.

Sample Input
4
1 2
3 1
2 4

Sample Output
4 2 1 1



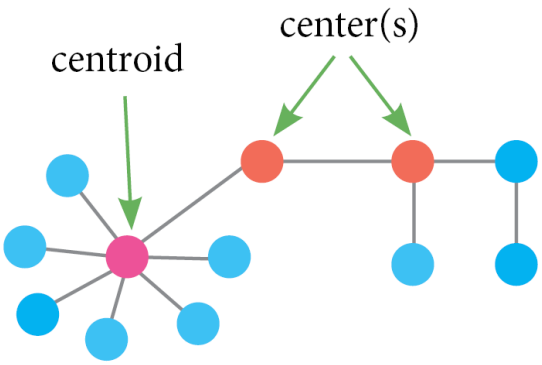

```
// SubTrees.cpp : DFS Solution.
#include <iostream>
#include <vector>

using namespace std;

int N;
vector<vector<int>> adjList;
vector<int> subTrees;
//---------------------------------

void readInput()
{
	cin >> N;
	adjList.resize(N + 1);
	for (int i = 0; i < N - 1; i++)
	{
		int u, v;
		cin >> u >> v;
		adjList[u].push_back(v);
		adjList[v].push_back(u);
	}
}
//---------------------------------
//If u is a leaf, size of subtree is 1.
//Otherwise 1 + sum of its subtrees (children)
void dfs(int u, int p)
{
	int temp = 1;
	for (int v : adjList[u])
	{
		if (v == p)
			continue;
		dfs(v, u);
		temp += subTrees[v];
	}

	subTrees[u] = temp;
}
//---------------------------------
void printResult()
{
	for (int i = 1; i <= N; i++)
		cout << subTrees[i] << " ";
```

```
	cout << endl;
}
//---------------------------------
int main()
{
	readInput();
	subTrees.resize(N + 1, 0);
	dfs(1, 1);
	printResult(); //Print heights of nodes
}
```

Center and Centroid

The **center** of a tree is a vertex with minimal eccentricity. The *eccentricity* of a vertex *X* in a tree is the maximum distance between the vertex *X* and any other vertex of the graph.

A vertex *u* is a **centroid** if after removing *u* from the tree, the size of any of the resulting connected components is at most n/2.



Example 4: Center(s) of a Tree

Find the center(s) of a tree with N nodes. Nodes are conveniently numbered 1 … N and node 1 is the root.

Sample Input

```
11
1 7
7 3
8 7
1 4
10 4
6 4
4 5
4 2
11 4
9 4
```

Sample Output

```
1 7
```

Hint: There can be one or two centers (two adjacent nodes). Keep removing leaf nodes from your tree until there are no more leaves.

```
// Center(s) of a Tree: Queue solution.
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

int N;
vector<vector<int>> adjList;
vector<int> degrees; //To determine leaves.
vector<int> distances;
//---------------------------------
void readInput()
{
	cin >> N;
	adjList.resize(N + 1);
	degrees.resize(N + 1, 0);
	for (int i = 0; i < N - 1; i++)
	{
		int u, v;
		cin >> u >> v;
		adjList[u].push_back(v);
		adjList[v].push_back(u);
		degrees[u]++, degrees[v]++;
	}
}
//---------------------------------
//Keep removing leaves with a queue.
void go()
{
	distances.resize(N + 1, 0);
	queue<int> q;
	for (int i = 1; i <= N; i++)
		if (degrees[i] == 1)
			q.push(i);

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

		for (int v : adjList[u])
		{
			degrees[v]--;
			if (degrees[v] == 1)
			{
				distances[v] = distances[u] + 1;
				q.push(v);
			}
		}
	}
}
//---------------------------------
//nodes with max distance are the centers.
void printResult()
{
	int maxDist = *max_element(distances.begin(), distances.end());
	for (int i = 1; i <= N; i++)
		if (distances[i] == maxDist)
			cout << i << " ";
}
//---------------------------------
int main()
{
	readInput();
	go();
	printResult();
}
```

Example 5: Centroid(s)

Find the centroid(s) of a tree with N nodes. Nodes are conveniently numbered 1 … N and node 1 is the root.

Sample Input

```
11
1 7
7 3
8 7
1 4
10 4
6 4
4 5
4 2
11 4
9 4
```

Sample Output

```
4
```

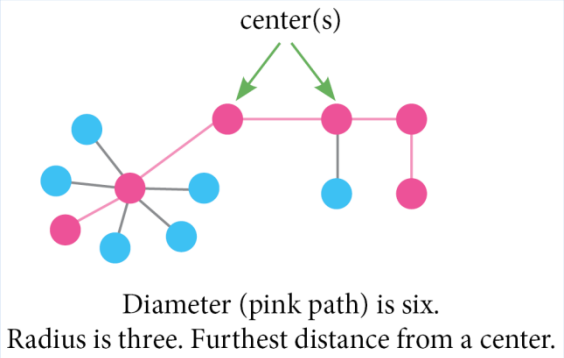
Hint: There can be one or two centroids (two adjacent nodes). Pick an arbitrary root, then run a DFS computing the size of each subtree, and then move starting from root to the largest subtree until we reach a vertex where no subtree has size greater than N/2. This vertex would be the centroid of the tree.

```
// Centroid of a Tree: DFS solution.
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

int N;
vector<vector<int>> adjList;
vector<int> resVec, subTrees;
//---------------------------------
void readInput()
{
	cin >> N;
	adjList.resize(N + 1);
	for (int i = 0; i < N - 1; i++)
	{
		int u, v;
		cin >> u >> v;
		adjList[u].push_back(v);
		adjList[v].push_back(u);
	}
}
//---------------------------------
//Calculate sub-tree sizes
void dfs1(int u, int p)
{
	subTrees[u] = 1;
	for (int v : adjList[u])
	{
		if (v == p)
			continue;
		dfs1(v, u);
		subTrees[u] += subTrees[v];
	}
}
//---------------------------------
//Find centroids
void dfs2(int u, int p)
{
```

```
        bool isCentroid = (N - subTrees[u] <= N / 2); //test the parent side

        for (int v : adjList[u])
        {
                if (v == p)
                        continue;

                if (subTrees[v] > N / 2)
                        isCentroid = false;

                if (subTrees[v] >= N / 2)
                        dfs2(v, u);
        }

        if (isCentroid)
                resVec.push_back(u);
}
//---------------------------------
//Unvisited nodes (degree > 0) are centers.
void printResult()
{
        sort(resVec.begin(), resVec.end());
        for (int x : resVec)
                cout << x << " ";
}
//---------------------------------
int main()
{
        readInput();
        subTrees.resize(N + 1, 0);
        dfs1(1, 1); //get subtree sizes
        dfs2(1, 1);
        printResult();
}
```

Radius and Diameter

- The **eccentricity** of the vertex v is the maximum distance from v to any vertex.
- The **radius** of a tree is the **minimum eccentricity** among the vertices of the tree.
- The diameter of a tree is the number of nodes on the longest path between two end nodes (**maximum eccentricity**).



Diameter (pink path) is six.
Radius is three. Furthest distance from a center.

Example 6: Radius and Diameter

Find the radius and diameter of a tree with *N* nodes. Nodes are conveniently numbered 1 … *N* and node 1 is the root.

Sample Input

```
11
1 7
7 3
8 7
1 4
10 4
6 4
4 5
4 2
11 4
9 4
```

Sample Output

```
3 6
```

Hint to find the diameter: Find the furthest node B from an arbitrary node A. Then find the furthest node C from node B. The diameter is the distance between node B and node C.

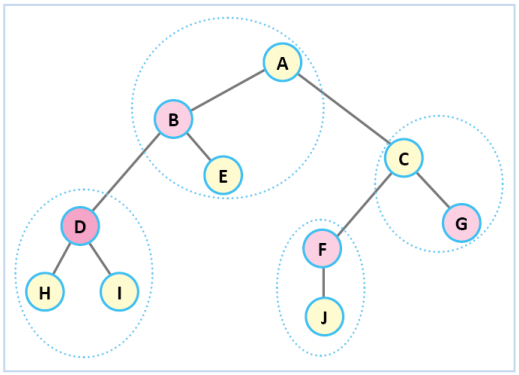
Hint to find the radius: Radius is half of the diameter. Consider the number of centers.

```
// Tree Diameter and Radius: BFS solution.
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

int N;
vector<vector<int>> adjList;
vector<int> distances;
//---------------------------------
void readInput()
{
	cin >> N;
	adjList.resize(N + 1);
	for (int i = 0; i < N - 1; i++)
	{
		int u, v;
		cin >> u >> v;
		adjList[u].push_back(v);
		adjList[v].push_back(u);
	}
}

//---------------------------------
void BFS(int start)
{
	queue<int> q;
	distances.clear();
	distances.resize(N + 1, N + 1);
	q.push(start);
	distances[start] = 1;
	while (!q.empty())
	{
		int u = q.front();
		q.pop();

		for (int v : adjList[u])
		{
			if (distances[v] < N + 1)
				continue;
			distances[v] = distances[u] + 1;
```

```
                    q.push(v);
                }
        }
}
//---------------------------------
int main()
{
        readInput();
        BFS(1);
        auto it = max_element(distances.begin() + 1, distances.end());
        BFS(it - distances.begin());
        int diameter = *max_element(distances.begin() + 1, distances.end());
        int radius = diameter / 2.0 + 0.5;
        cout << diameter << " " << radius << endl;
}
```

Minimum Vertex Cover

Given an undirected graph the objective is to determine a minimum subset of the vertices which covers all edges.



The problem to find minimum size vertex cover of a graph is NP complete. But it can be solved in polynomial time for trees.

Example 7: Minimum Vertex Cover

Find the size of the minimum vertex cover and in how many different ways you can obtain minimum vertex cower of a tree with *N* nodes. Nodes are conveniently numbered 1 … *N* and node 1 is the root. *N* is 6 in the sample input.

Sample Input

```
6
1 2
1 3
4 5
4 1
6 2
```

Sample Output

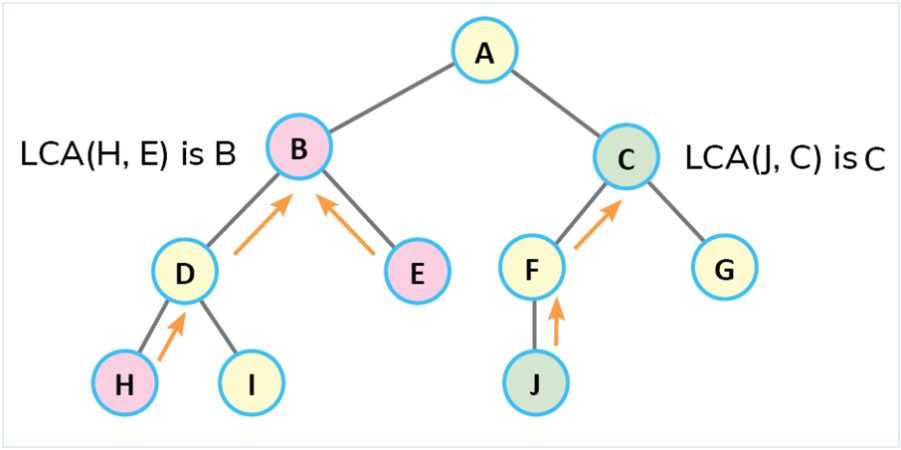
```
3 8
```

Hint: Greedy solution. Select parent of each leaf if the leaf was not already selected. Keep removing leaves.

```
// Vertex Cower Tree: Greedy Solution with a queue. O(N).
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <queue>

using namespace std;

int N;
vector<vector<int>> adjList;
vector<int> nodes;
//--------------------------------------
void readInput()
{
	cin >> N;
	adjList.resize(N);
	for (int i = 0; i < N - 1; i++)
	{
		int u, v;
		cin >> u >> v;
		u--; v--;
		adjList[u].push_back(v);
		adjList[v].push_back(u);
	}
}
//--------------------------------------
//nodes[i] -> -1: internal node, 0: leaf, 1:selected
int go()
{
	queue<int> q;

	for (int i = 0; i < N; i++)
		if (adjList[i].size() == 1)
		{
			nodes[i] = 0;
			q.push(i);
		}

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

		for (int v : adjList[u])
		{
			if (nodes[v] >= 0) //not -1
				continue;

			nodes[v] = 0;
			if (nodes[u] == 0)
				nodes[v] = 1;

			q.push(v);
		}
	}
	return  count(nodes.begin(), nodes.end(), 1);
}
//--------------------------------------
int main()
{
	readInput();
	if (N <= 2)
		cout << 1 << endl;
	else
	{
		nodes.resize(N, -1);
		cout << go() << endl;
	}
}
```

Lowest Common Ancestor (LCA)

The lowest common ancestor (LCA) of the nodes *v* and *w* of a tree *T* is the shared ancestor of *v* and *w* that is located farthest from the root. The lowest common ancestor is also known as the **least common ancestor**.



Example 8: LCA Queries

Answer Q Lowest Common Ancestor (LCA) queries on a tree with N nodes. Nodes are conveniently numbered 1 … N and node 1 is the root. For each query (u, v) print the node id of the LCA of u and v. N=4 and Q=4 in the sample input.

Sample Input

```
5 4
1 2
1 3
5 3
4 3
3 4
2 4
5 1
4 5
```

Sample Output

```
3
1
1
5
```

Solution 1: Traversing Up to the LCA. O(N + QH).

Pre-calculation: Depth and parent information of each node. DFS in O(N) time.

Answering a query: Bring both the nodes to the same height by making the node with greater depth jump one parent up the tree till both the nodes are at the same height. Once, both the nodes are at the same height then start jumping one parent up for both the nodes simultaneously till both the nodes become equal and that node will be the LCA of the two originally given nodes. This process takes O(H) time where H is the height of the tree.

```
// LCA : O(QH) solution. H is the tree height
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int N, Q; //Number of nodes and queries
vector<vector<int>> adjList;
vector<int> parent, depth;
//---------------------------------
void readInput()
{
	cin >> N >> Q;
	adjList.resize(N + 1);
	for (int i = 0; i < N - 1; i++)
	{
		int u, v;
		cin >> u >> v;
		adjList[u].push_back(v);
		adjList[v].push_back(u);
	}
}
//---------------------------------
//Parents and depths
void DFS(int u, int p)
{
	int maxH = 0; //max height of children
	for (int v : adjList[u])
	{
		if (v == p)
			continue;
		depth[v] = depth[u] + 1;
		parent[v] = u;
		DFS(v, u);
	}
}

//---------------------------------
int lca(int u, int v)
{
	while (depth[u] > depth[v])
		u = parent[u];

	while (depth[v] > depth[u])
		v = parent[v];

	while (u != v)
	{
		u = parent[u];
		v = parent[v];
	}

	return u; //or return v
}
//---------------------------------
void queries()
{
```

```
	for (int i = 0; i < Q; i++)
	{
		int u, v;
		cin >> u >> v;
		cout << lca(u, v) << endl;
	}
}
//---------------------------------
int main()
{
	readInput();
	parent.resize(N + 1, 0);
	depth.resize(N + 1, 0);

	DFS(1, 1);
	queries();
}
```

Solution 2: Binary Lifting Up. O(NlgN + QlgN).

Pre-calculation: Binary Lifting is a dynamic programming approach where we pre-compute an array memo[1, n][1, log(n)] where memo[i][j] contains 2^j-th ancestor of node i. For computing the values of memo[][], the following recursion may be used

```
memo[i][j] = parent[i] if j = 0 and

memo[i][j] = memo[memo[i][j – 1]][j – 1] if j > 0.
```

Answering a query: We first check whether a node is an ancestor of other or not and if one node is ancestor of other then it is the LCA of these two nodes otherwise we find a node which is not the common ancestor of both u and v and is highest (i.e. a node x such that x is not the common ancestor of u and v but memo[x][0] is) in the tree. After finding such a node (let it be x), we print the first ancestor of x i.e. memo[x][0] which will be the required LCA.

```
// LCA queries with binary lifting up
#include <fstream>
#include <vector>

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");
//-----------------------------------
int N, M, Q; //number of nodes, edges and queries.
int LOG;
vector<vector<int>> adjList;
vector<pair<int,int>> queries;
vector<int> depth;
vector <vector<int>> P; //P[i][j] stores the 2^j th ancestor of node i
//-----------------------------------
void readInput()
{
	cin >> N >> Q;
	adjList.resize(N + 1);
	for (int i = 0; i < N - 1; i++)
	{
		int u, v;
		cin >> u >> v;
		adjList[u].push_back(v);
		adjList[v].push_back(u);
	}
```

```
	queries.resize(Q);
	for (int i = 0; i < Q; i++)
		cin >> queries[i].first >> queries[i].second;
}
//-----------------------------------
void dfs(int u, int p) //current and parent nodes
{
	depth[u] = depth[p] + 1;

	for (int i = 0; i <= LOG; i++)
		if (i == 0)
			P[u][i] = p;
		else
			P[u][i] = P[P[u][i - 1]][i-1];

	for (int v : adjList[u])
		if (v != p)
			dfs(v, u);
}
//-----------------------------------
//Construct P matrix and find depth of each node
void init()
{
	LOG = log2(N);
	if (N%LOG > 0)
		LOG++;

	P.resize(N, vector<int>(LOG + 1, -1));
	depth.resize(N + 1, -1);

	dfs(1, 1);
}
//-----------------------------------
//return lca of u and v
int go(int u, int v)
{
	//linf up the lower level node (u) to the same level with v.
	if (depth[u] < depth[v])
		swap(u, v);

	int dist = depth[u] - depth[v];
	while (dist > 0)
	{
		u = P[u][log2(dist)];
		dist -= pow(2, log2(dist));
	}

	/* An alternative method to lift up u
	for (int i = LOG; i >= 0; i--)
		if ((depth[u] - pow(2, i)) >= depth[v])
			u = memo[u][i];
	*/

	if (u == v)
		return u;

	// Keep raising the two nodes by equal amount
	// untill each node has been raised uptill its (k-1)th ancestor
	// such that the (k)th ancestor is the LCA.
	//This for loop works like a binary search.
	for (int j = log2(depth[u]); j >= 0; --j)
		if (P[u][j] != P[v][j]) //different ancestors in this depth
		{
			u = P[u][j];
			v = P[v][j];
		}

	return P[u][0]; //parent of u and v
}
//-----------------------------------
int main()
```

```
{
	readInput();
	init();
	for (auto q : queries)
		cout << go(q.first, q.second) << endl;
	return 0;
}
```

Alternative Solutions for LCA Queries

- Squareroot Decomposition of heights. O(N + sqrt(H)Q)
- Euler tour + range minimum queries with sparse table. O(NlgN +Q). This method will be discussed in the "Queries on Trees" chapter.
- Tarjan's off-line LCQ Algorithm. O(N + Q).

Exercise 2: Ancestor-Descendant Relationship Queries

Modify the LCA binary lifting up program to answer ancestor-descendant relationship queries. *query(u, v)* function should return *true* if u is an ancestor of *v*.

Sample Input

```
5 4
1 2
1 3
5 3
4 3
3 4
2 4
5 1
1 5
```

Sample Output

```
Yes
No
No
Yes
```

Useful Links and References

https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm
https://www.tutorialspoint.com/centers-of-a-tree
https://tanujkhattar.wordpress.com/2016/01/10/centroid-decomposition-of-a-tree/
https://www.techiedelight.com/find-lowest-common-ancestor-lca-two-nodes-binary-tree/
https://www.geeksforgeeks.org/lca-in-a-tree-using-binary-lifting-technique/?ref=rp