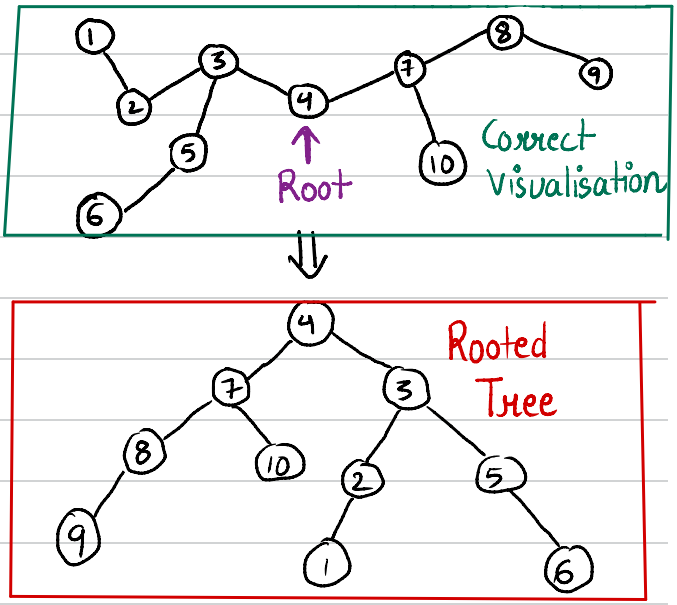
September 27, 2025

TREE FOUNDATIONS AND FRAMEWORK

* Trees → a general modification of graph which has no cycle and it is connected.

How tree looks like :



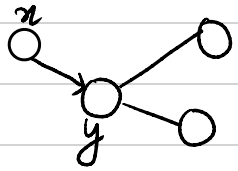
All concepts like BFS/DFS is applicable to Trees.

We will mostly use DFS instead of BFS because here in Trees, there is no concept of shortest path (BFS is used to find it) but there is a concept of unique path (there is no cycle). Any and every path is unique which we can easily find DFS.

Reachability = Shortest Path.

* Modification in DFS here

Give nodes, x & y, exploring y's neighbours :




```
visited[y] = 1
for(auto v: g[y])
    if(visited[v] != 1)
        DFS(v)
```

Now, the concept is there is no need to check for visited[v]. Because there is always a unique path and hence the node will be unvisited. Updated Code :

```
visit[y] = 1
for(auto u: g[y])
    if(u != parent[y])
        dfs(v)
```

Just track the last parent.

* How to structure the code :

Maintain the parent as the parameter then we won't even have to save it in the array like :

```
DFS(x, parent)
visit[y] = 1
for(auto u: g[y])
    if(v != parent)
        DFS(v)
```

* Why don't we need 'no. of edges' as input here?

⇒ Because there is no cycle. Hence, to connect n nodes, (n-1) edges are needed.

* DFS in Trees code :

```cpp
#include<bits/stdc++.h>
using namespace std;

int n;
vector<vector<int>> g;
void dfs(int nn,int pp){
    cout<<nn<<endl;
    for(auto v:g[nn]){
        if(v!=pp){
            dfs(v,nn);
        }
    }
}

void solve(){
    cin>>n;
    g.resize(n+1);
    for(int i=0;i<n-1;i++){
        int a,b;
        cin>>a>>b;
        g[a].push_back(b);
        g[b].push_back(a);
    }

    dfs(1,0);
}
```

node, parent

Input

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

Output

```
1
2
3
4
```

```cpp
int n;
vector<vector<int>> g;
vector<int> depth;

void dfs(int nn,int pp,int dd){
    depth[nn] = dd;
    for(auto v:g[nn]){
        if(v!=pp){
            dfs(v,nn,dd+1);
        }
    }
}

void solve(){
    cin>>n;
    g.resize(n+1);
    depth.resize(n+1);
    for(int i=0;i<n-1;i++){
        int a,b;
        cin>>a>>b;
        g[a].push_back(b);
        g[b].push_back(a);
    }

    dfs(1,0,0);
```

depth

Similarly, can maintain parent as well.

```cpp
int n;
vector<vector<int>> g;
vector<int> depth,par;

void dfs(int nn,int pp,int dd){
    depth[nn] = dd;
    par[nn] = pp;
    for(auto v:g[nn]){
        if(v!=pp){
            dfs(v,nn,dd+1);
        }
    }
}

void solve(){
    cin>>n;
    g.resize(n+1);
    depth.resize(n+1);
    par.resize(n+1);
    for(int i=0;i<n-1;i++){
        int a,b;
        cin>>a>>b;
        g[a].push_back(b);
        g[b].push_back(a);
    }
dfs(1,0,0);

for(int i=1;i<=n;i++){
    cout<<i<<" "<<par[i]<<" "<<depth[i]<<endl;
}
```

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

Output

```
1 0 0
2 1 1
3 2 2
4 2 2
```

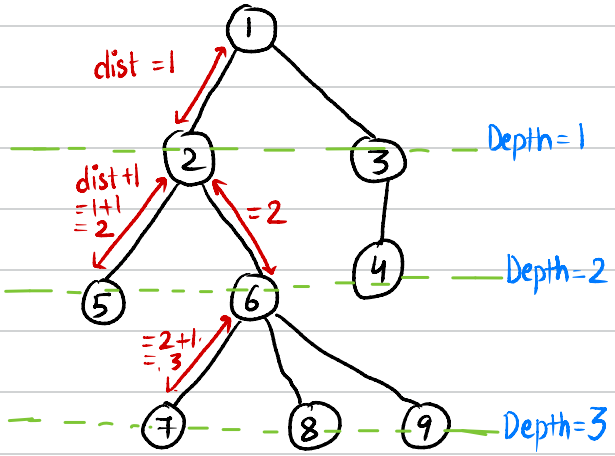
```
4
1 2
2 3
1 4
```

Output

```
1 0 0
2 1 1
3 2 2
4 1 1
```

* Find distance to all node from 1.

depth

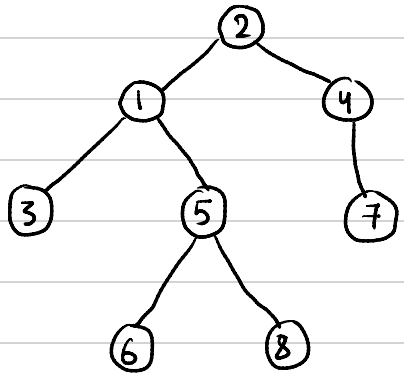


Code to do the same :

AlgoZenith

* Subtree

For any particular node, anything inside it (child + subchild)

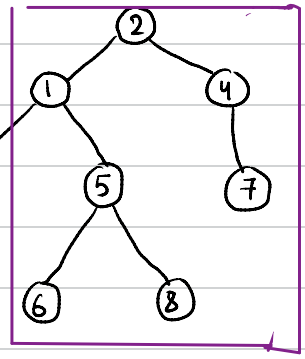


Which nodes are in the subtree of 1?

Since this is not a rooted tree, we cannot comment on which nodes will come under 1.

Suppose the tree is rooted at 3

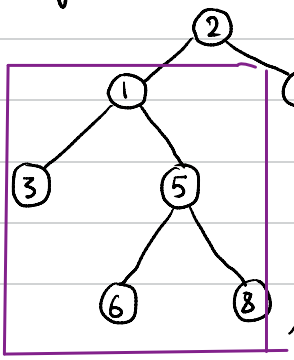
Then, the subtree will look like:



Subtree of 1 when 3 is rooted

1 → {1, 5, 6, 8, 2, 4, 7}

Let's say 2 is rooted, then:



1 → {1, 3, 5, 6, 8} Subtree

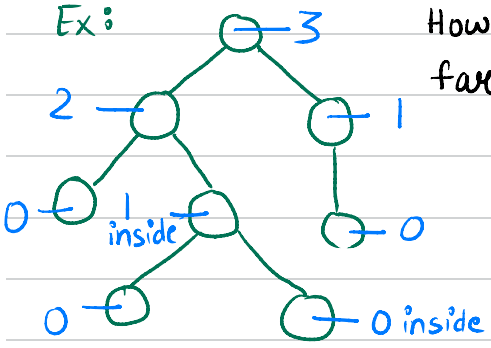
* Given a tree, for every node: Calculate number of nodes in its subtree.

```
dfs(x, pp)
    sub[x] = 1
    for(v : g[x])
        if(v != pp)
            dfs(v, nn)
            sub[x] += sub[v]
```

```
#include<bits/stdc++.h>
using namespace std;

int n;
vector<vector<int>> g;
// ancestor
vector<int> depth,par;
// subtree
vector<int> subsz;

void dfs(int nn,int pp,int dd){
    depth[nn] = dd;
    par[nn] = pp;

    subsz[nn] = 1;

    for(auto v:g[nn]){
        if(v!=pp){
            dfs(v,nn,dd+1);
            subsz[nn] += subsz[v];
        }
    }
}
```

* For every node, calculate how far, deep inside there is a node:

Ex:

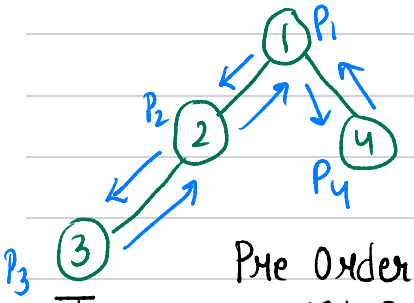


How far is the farthest node?

AlgoZenith

```cpp
#include<bits/stdc++.h>
using namespace std;

int n;
vector<vector<int>> g;
// ancestor
vector<int> depth,par;
// subtree
vector<int> subsz,subfar;

void dfs(int nn,int pp,int dd){
    depth[nn] = dd;
    par[nn] = pp;

    subsz[nn] = 1;
    subfar[nn] = 0;

    for(auto v:g[nn]){
        if(v!=pp){
            dfs(v,nn,dd+1);
            subsz[nn] += subsz[v];
            subfar[nn] = max(subfar[nn],1+subfar[v]);
        }
    }
}
```

* Pre order traversal Vs Post



P_1 P_2 P_3 P_4

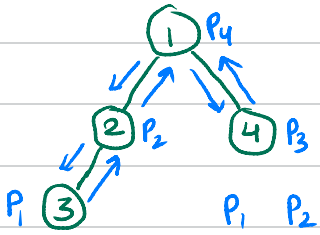
Pre Order: 1 2 3 4

Is this a valid Pre-Order traversal : {1, 4, 2, 3} ?

YES. Pre-Order traversal of a tree is not unique.

Post-Order Traversal:

For any node, its child will come before. Ex:



P_1 P_2 P_3 P_4

Post-Order ⇒ 3 2 4 1

// Pre-Order

```cpp
int n;
vector<vector<int>> g;

void dfs(int nn,int pp){
    cout<<nn<<endl;
    for(auto v:g[nn]){
        if(v!=pp){
            dfs(v,nn);
        }
    }
}

void solve(){
    cin>>n;
    g.resize(n+1);
    for(int i=0;i<n-1;i++){
        int a,b;
        cin>>a>>b;
        g[a].push_back(b);
        g[b].push_back(a);
    }
    dfs(1,0);
}
```

```
Input
4
1 4
1 2
2 3

Output
1
4
2
3
```

// Post-Order

```cpp
int n;
vector<vector<int>> g;

void dfs(int nn,int pp){
    for(auto v:g[nn]){
        if(v!=pp){
            dfs(v,nn);
        }
    }
    cout<<nn<<endl;
}

void solve(){
    cin>>n;
    g.resize(n+1);
    for(int i=0;i<n-1;i++){
        int a,b;
        cin>>a>>b;
        g[a].push_back(b);
        g[b].push_back(a);
    }
    dfs(1,0);
}
```

```
4
1 2
2 3
1 4

Output
3
2
4
1
```

Ques. Given a tree, find out the number of different pre-order traversals possible.

For 1 node, how many ways are there to get its pre-order ?

pre[x] = 1

Now, Suppose it has 2 childs then how many ways?

pre[x] = 2

What if there are 3?

pre[x] = 3 X 2 X 1 = 6

What if there are n childs for a node then how many ways for pre-order?

⇒ (# of childs)!

= n!

Coming back to our question now:

When we are deciding for a node

a → look at levels

b → in which order you'll visit them

c → order inside subtree

For c,

# of ways(child1) = 1

# of ways(child2) = 2

# of ways(child3) = 3

& so on

Depending on b, it could be (#child)! X (1 * 2 * 3 ... * Kth child ways)

```
int fact[100100]; ||Pre-calc

int n;
vector<vector<int>> g;
vector<int> ways;
void dfs(int nn,int pp){
    ways[nn] = 1;
    int child = 0;
    for(auto v:g[nn]){
        if(v!=pp){
            dfs(v,nn);
            ways[nn] *= ways[v];
            child++;
        }
    }
    ways[nn] *= fact[child];
}
```

* Frameworks

- Foundation || Basic Properties
  - DFS
  - Subtree
  - Ancestors
- Diameter
  Nodes with maximum distance in a tree determines its diameter.
- Center
  Mid point node of diameter's path. There can be more than one center.
- Centroid
  A node whose all subtrees sizes is less than half of the whole tree's size.
- Contribution Technique
  Edge

* How to find diameter?

From one node, find the farthest node corresponding to it

From one node, just do DFS and you will get the farthest node (A)

From that, again do DFS, and find another farthest node (B).

The A & B are diameter's end point.

```cpp
#include<bits/stdc++.h>
using namespace std;

int n;
vector<vector<int>> g;
// ancestor
vector<int> depth;

void dfs(int nn,int pp,int dd){
    depth[nn] = dd;
    for(auto v:g[nn]){
        if(v!=pp){
            dfs(v,nn,dd+1);
        }
    }
}

void solve(){
    cin>>n;
    g.resize(n+1);
    depth.resize(n+1);
    for(int i=0;i<n-1;i++){
        int a,b;
        cin>>a>>b;
        g[a].push_back(b);
        g[b].push_back(a);
    }

    dfs(1,0,0);

    int x = 1;
    for(int i=1;i<=n;i++){
        if(depth[i]>depth[x])x=i;
    }

    dfs(x,0,0);

    int y = 1;
    for(int i=1;i<=n;i++){
        if(depth[i]>depth[y])y=i;
    }

    cout<<x<<" "<<y<<endl;
```

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

Output

```
3 4
```

From 1, DFS gave 3.

From 3, DFS gave 4.

Hence 3 & 4 became endpoints of diameter

* More on Frameworks (No Updates)

Fuse data structures

1. Ancestoral Maintanence
   Data Structures on Ancestors.

2. Small to Large Merging / BSU on sack
   Data Structures on Subtrees

3. DS on Paths
   Binary Lifting, LCA finding