Tree :

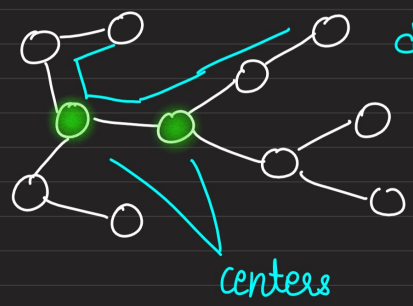
↳ undirected connected acyclic graph

$|V| = n$ then $|E| = n-1$

- Only 1 simple path between 2 nodes.

Trees
- * DFS/BFS on Tree
  - Diameter
  - Centre
  - leaf → no child
- * DFS on subtree
  - size
  - depth
  - centroid
- DP on Trees → part of Advanced DP
- Query on Tree. *

Diameter = length of longest path in a tree.
(d)
↳ not unique.



Center(s) : Point(s) through which all the diameter passes.

Diameter :-

Algorithm

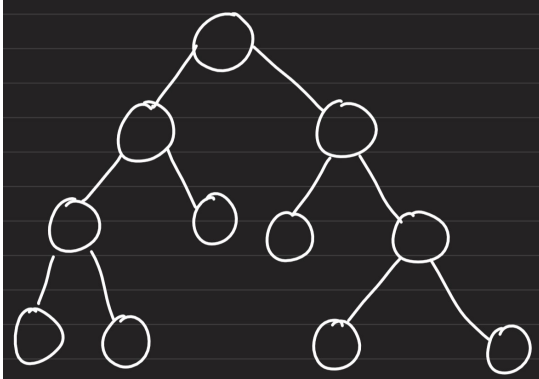
- Choose a random point X
- Find the farthest point from X, say Y
- Find the farthest point from Y, say Z

Y - Z is a diameter.

↳ 2 times DFS & we are done

↳ To print path, one more DFS.

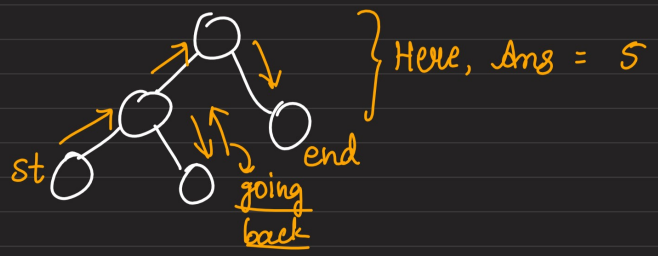
PYQs



Find the minimum time to traverse all the nodes of the tree starting from anywhere and ending at anywhere.

(No need to return to the starting point.)

eg:



Suppose, we need to return to starting point as well.
then Ans = 2 (N-1) ∵optimal.
↳ edge will be travelled twice.

But we don't need to return back

Ans = 2E - path path ≡ save cost

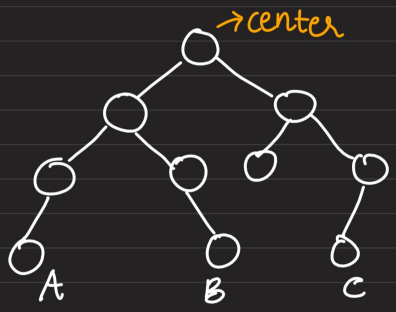
minimize ans
maximize path

∴ path = diameter

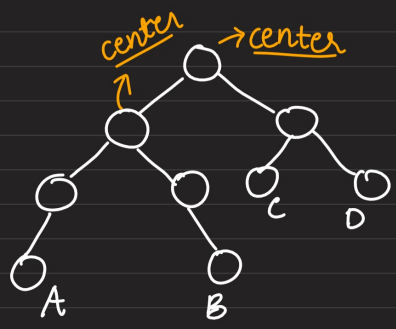
Ans = 2E - diameter.

Center of a tree (All diameters passes through center(s))

Node at the middle of the diameter.



#diameters = 2
(A-C, B-C)



#diameters = 4
(A-C, A-D
B-C, B-D)

A tree can have multiple centers. (either 1 or 2 centers)

Center is property of tree and every diameter will pass through every centre of the tree.

P1. Find center(s).

sol^n: Find diameter, track back to $\frac{|\text{diameter}|}{2}$ nodes to get the center. We will use parent links to get the center.

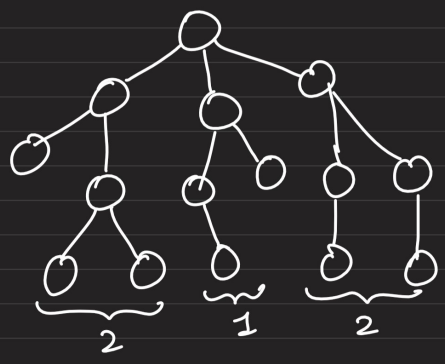
$D = 5$ } → 2 centers, go $\frac{D}{2}$ & $(\frac{D}{2}+1)$ steps up to get centers.

$D = 4$ } → 1 center, go $\frac{D}{2}$ steps up the parent link to get centers.

P2 Find no. of diameters.

sol^n: Case I: 1 center (|diameter| = even)

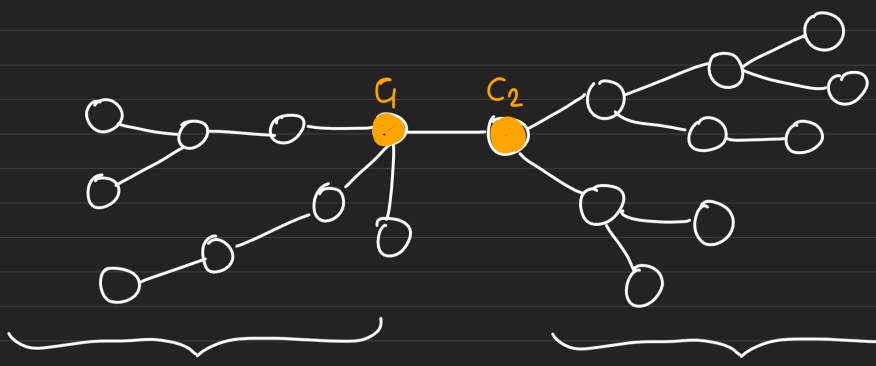
Root tree at this center.



diameters
$= 2 \times 1 + 1 \times 2 + 2 \times 2$
$= 8$ Ans

nodes of depth = $\frac{|\text{diameter}|}{2}$ in each neighbouring subtree, then take $\sum$ all pair product.

Case II : 2 centers ⇒ |diameter| = odd



count(# nodes at $\frac{d-1}{2}$) — c_1

count(# nodes at $\frac{d-1}{2}$) — c_2

Ans = $c_1 * c_2$ (here $3 \times 3 = 9$)

We can call, dfs $(C_1, C_2, 0)$ → dfs on C_1 with $p = C_2$

dfs $(C_2, C_1, 0)$ → dfs on C_2 with $p = C_1$

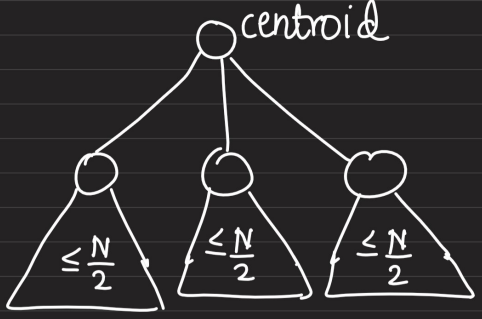
Implementation

↳

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

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

int cnt = 0; //counts nodes is a sutree of node nn
             //ans parent pp with depth = d
void cnt_nodes(int nn, int pp, int &d)
{
    if(depth[nn]==d) cnt++;
    for(auto v: g[nn])
    {
        if(v!=pp)
        {
            cnt_nodes(v,nn,d);
        }
    }
}
```

```cpp
void solve()
{
    int u,v;
    cin>>n;
    g.resize(n+1); par.resize(n+1); depth.assign(n+1,0);
    for(int i=1;i<=n-1;i++)
    {
        cin>>u>>v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    if(n==1)
    {
        cout<<"1\n";
        return;
    }
    dfs(1,0,0);
    int a = 1;
    for(int i=1;i<=n;i++)
    {
        if(depth[i]>depth[a])
        {
            a = i;
        }
    }
    dfs(a,0,0);
    int b = a;
    for(int i=1;i<=n;i++)
    {
        if(depth[i]>depth[b])
        {
            b=i;
        }
    }
    int len_d = depth[b];
    //Tracking back len_d/2 times to get the center
    int cen1 = b;
    int ops = 0;
    while(ops!=(len_d)/2)
    {
        cen1 = par[cen1];
        ops++;
    }
    int cen2 = -1;
    if(len_d%2==1)
    {
        //2 centers
        cen2 = par[cen1];
    }
    // cout<<cen1<<' '<<cen2<<'\n';

    //Finding nodes of a perticular depth in a subtree
    if(len_d%2==0) //One Center
    {
        vector<lli> vals;
        int d = len_d/2;
        dfs(cen1,0,0);
        for(auto v: g[cen1])
        {
            cnt_nodes(v,cen1,d);
            vals.push_back(cnt);
            cnt = 0;
        }

        lli ans = 0,sum = vals[0];
        for(int i=1;i<vals.size();i++)
        {
            ans += 1ll*sum*vals[i];
            sum += 1ll*vals[i];
        }
        cout<<ans<<'\n';
        return;
    }
    else //2 centers
    {
        int d = len_d/2;
        cnt = 0;
        dfs(cen1,0,0);
        cnt_nodes(cen1,cen2,d);
        int x = cnt;

        cnt = 0;
        dfs(cen2,0,0);
        cnt_nodes(cen2,cen1,d);
        int y = cnt;

        lli ans = 1ll*x*y;
        cout<<ans<<'\n';
        return;
    }
}

int main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);cout.tie(0);

    solve();
}
```

Centroid:



size of every subtree originating from centroid is $\leq \frac{N}{2}$

N = total no. of nodes

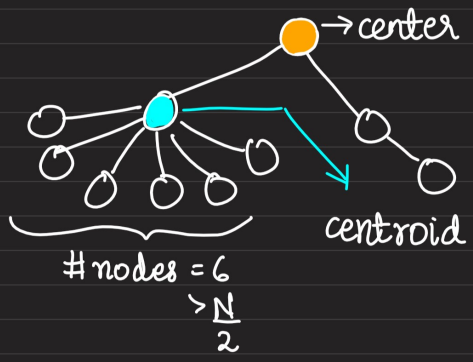
There can be multiple centroids.

Q1. Is there a limit on the #centroids?

Ans → Tree can have at max 2 centroids.

Q2. Is center = centroid ?

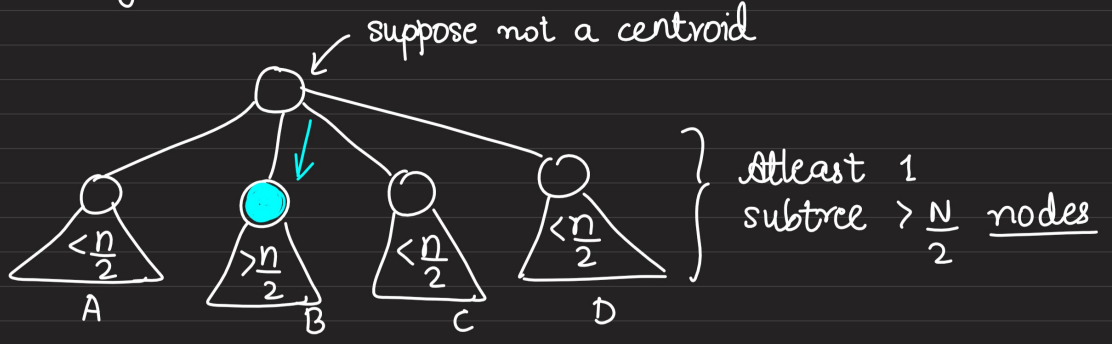
Ans → No



N = 10

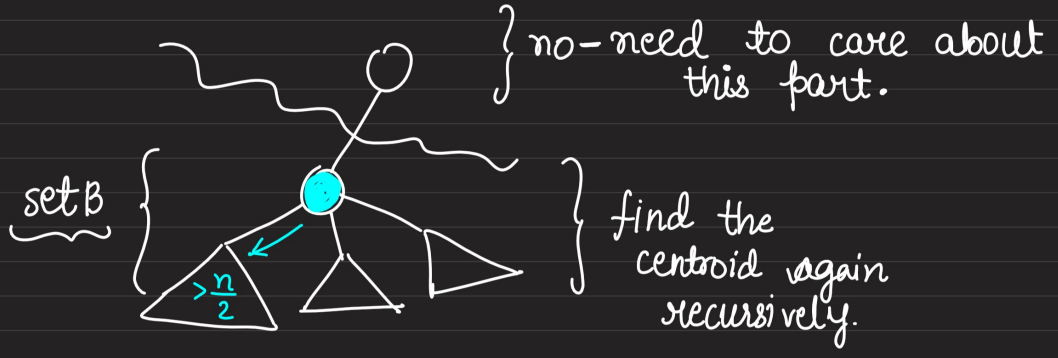
counter example.

Finding Centroid :

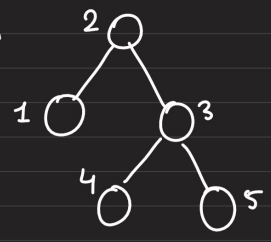


Actually, 2 subtree cannot have $> \frac{N}{2}$ size.

Centroid is in → setB *

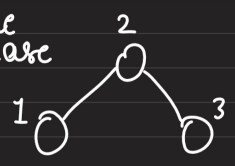


We can calculate size of all subtrees by only one dfs.

Q. 

Find $\sum_{1 \le i < j \le n} \sum dist(i,j)$

$N \le 2 \times 10^5$

Soln: Sample test case 

$= dist(1,2) + dis(1,3) + dist(2,3)$

$= 1 + 2 + 1 = 4$

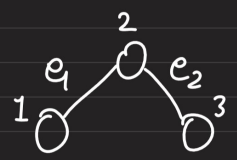
Edge Contribution Technique :

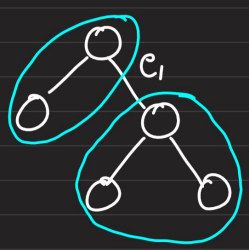
for all the edges, we want to count how many times a edge is there in the paths.

$D(1,2) + D(2,3) + D(1,3)$

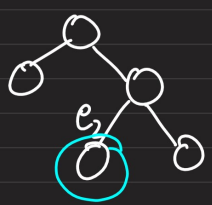
$= e_1 + e_2 + (e_1 + e_2)$

$= 2e_1 + 2e_2$

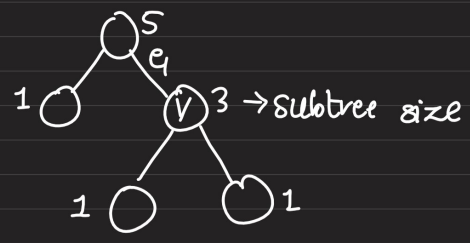




contribution of e_1 $= 2 \times 3 = 6$



contribution of e_2 $= 1 \times 4 = 4$



$e_i = S[v] * (n - S[v])$

simply do summation for all the edges.

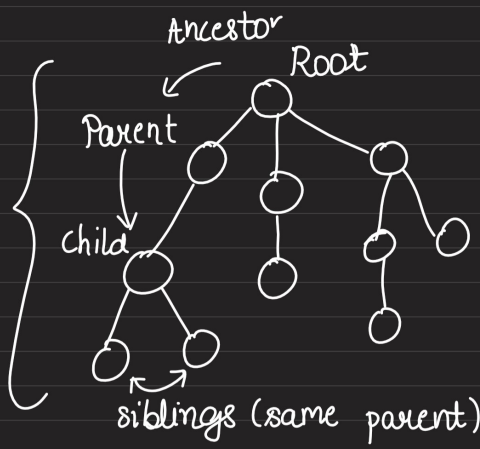
This is now O(N) solution.

```cpp
vector<vector<lli>> g;
vector<lli> par,subtreeSZ;

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

```cpp
void solve()
{
    lli n;cin>>n;
    g.resize(n+1); par.resize(n+1); subtreeSZ.resize(n+1);
    vector<ii> edgeList;
    lli u,v;
    for(int i=0;i+1<n;i++)
    {
        cin>>u>>v;
        edgeList.push_back({u,v});
        g[u].push_back(v);
        g[v].push_back(u);
    }
    dfs(1,0);
    lli ans = 0;
    for(auto edge : edgeList)
    {
        lli u = edge.F;
        lli v = edge.S;
        lli sz = min(subtreeSZ[u],subtreeSZ[v]);
        ans += sz * (n-sz);
    }
    cout<<ans<<'\n';
}
```

DFS on SubTree

- Either root is given or we choose it randomly. Rooted tree
- Root gives us parent child relation.



Terms :

Parent (N)

Child (N)

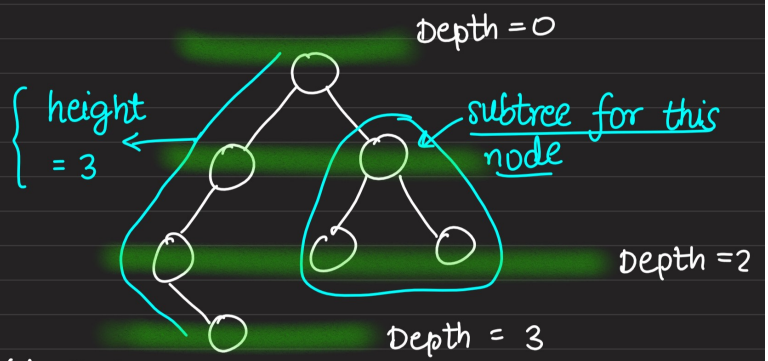
Ancestor (N)

Siblings (N)

Subtree (N)

height (T) = max (depths)

Depth (N)

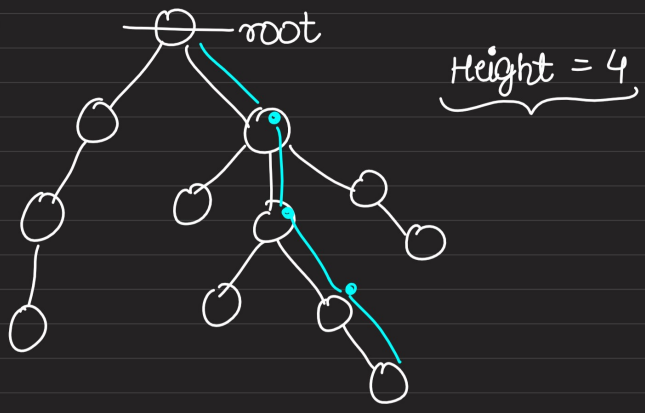


Child (N) = neighbours except parent.

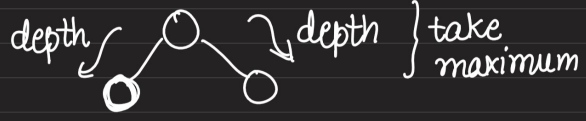
#Neighbours − 1

Q1. Given Tree
↳ find height

soln:



- dfs



depth [node] = depth [parent] + 1.

- bfs

do bfs from root. Take maximum dist among all nodes.

Q2. Sizes of Subtree :

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

int count_size(int nn, int pp) //nn-> node, dd-> depth of node, pp-> parent
{
    par[nn] = pp;
    //Base case
    if(g[nn].size()==1 && g[nn][0]==pp)
    {
        return subtrr_size[nn]=1;;
    }
    //Cache
    if(subtrr_size[nn]!=-1)
    {
        return subtrr_size[nn];
    }
    //Transitions
    int ans = 1;
    for(auto v : g[nn])
    {
        //here we dont need to check visited as there are no cycles,
        //only parent is visited, so, we already know that it is not visited
        if(v!=pp)
        {
            ans += count_size(v,nn); //this will find depth of all the nodes
        }
    }
    //save and return
    return subtrr_size[nn] = ans;
}
```

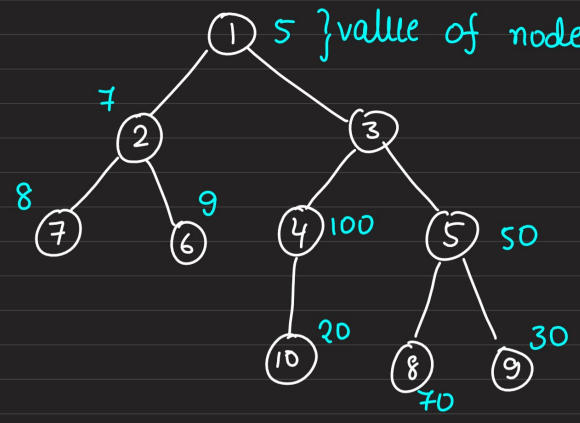
} → child node

DP approach

```cpp
vector<vector<int>> child;
vector<int> depth; //depth[child] = depth[parent] + 1
vector<int> par;
vector<int> isLeaf;
vector<int> subtreeSz;
vector<int> numChild;

void dfs(int nn, int pp, int dd) //nn-> node, dd-> depth of node, pp-> parent
{
    par[nn] = pp;
    depth[nn] = dd;
    subtreeSz[nn] =1;
    numChild[nn] = 0;
    for(auto v : g[nn])
    {
        //here we dont need to check visited as there are no cycles,
        //only parent is visited, so, we already know that it is not visited
        if(v!=pp)
        {
            numChild[nn]++;
            dfs(v,nn,dd+1); //this will find depth of all the nodes
            child[nn].push_back(v);
            subtreeSz[nn] += subtreeSz[v]; //do it during backtracking
        }
    }
    if(numChild[nn]==0)
    {
        isLeaf[nn]=1;
    }
}
```

Recursion approach

Q3.



Find. i) $\sum$ values in its subtree.

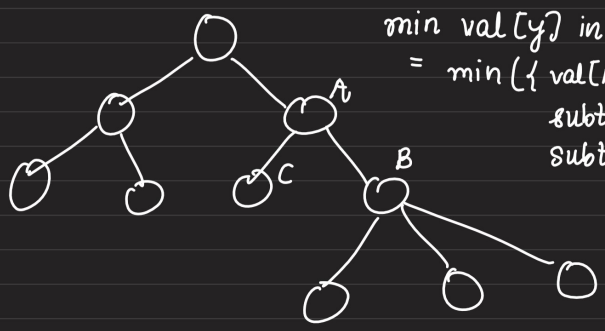
ii) $\max [val[x] - val[Y])$ $y \in$ subtree (x)

iii) $\sum_{x=1}^{N} \sum_{y \in subtree(x)} [val[x] == val[y])$

iv) $\sum_{x=1}^{N} \min \left(|val[x] - val[y]| \right)$ $y \in$ subtree (x) , $x \neq y$

soln: i) $\sum$ val in subtree = val [node] + $\sum$ val [childs]

ii) find minimum value of each subtree



min val[y] in subtree A
= min ({ val[A],
subtree [B]
subtree [C])

iii) $\sum_{x=1}^{N} \sum_{y \in subtree(x)} [val[x] == val[y]]$ → Reversal Approach

equivalent to

$\sum_{x=1}^{N} \sum_{y \in ancestor} [val[x] == val[y]]$

Here we maintain map of values of all the ancestors.

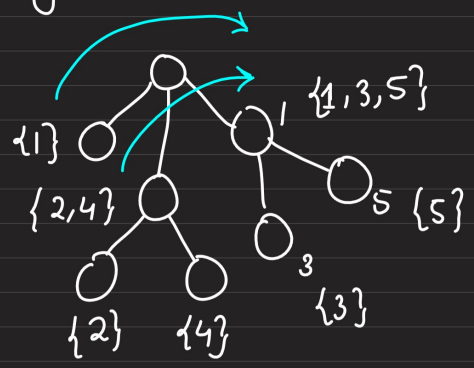
```
lli count_similars = 0;
void dfs(int nn,map<int,int> &explorings,int pp) //exploring is the map of ancestors values
{
    par[nn] = pp;
    count_similars += explorings[val[nn]];
    explorings[val[nn]]++;
    for(auto v: g[nn])
    {
        if(v!=pp)
        {
            dfs(v,explorings,nn);
        }
    }
    explorings[nn]--;
}
```

iv) $\sum_{x=1}^{N} \min(|val[x] - val[y]|), \; y \in subtree(x), \; x \neq y$

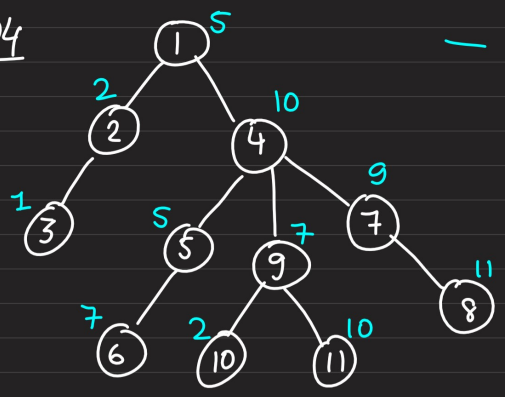
Here, we cannot use ancestor logic. We will use DSU.

- We create set of subtree & find the minimum among them.
  - ↳ lower_bound (x) (it) } O(log n)
  - ↳ it--

Create sets of the subtree, merge them using small to big concept to get set for their parent.



Q4



— → value of node

} root, tree is given

for → all nodes x, find max (| val[x] - val[y] |)
where y is ancestor of x.

Soln: for, x = 11 ans = val[1] - val[11]
= 5

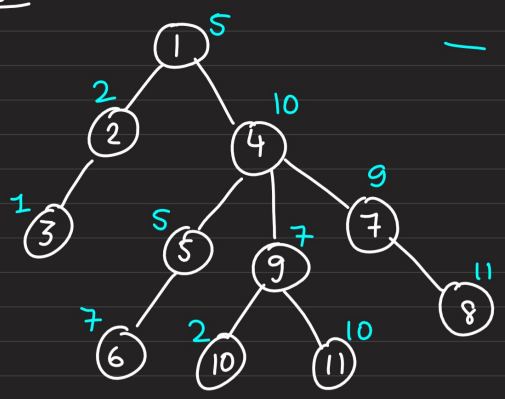
x = 9 ans = | val[9] - val[4] |
= | 7 - 10 | = 3

Keep track of minimum & maximum valued ancestor for every node.

ans = max(val[x] - val[minp],
val[x] - val[maxp])

```
vector<vector<lli>> g;
vector<lli> val;
vector<lli> ans;

void dfs(lli nn, lli pp, lli maxVal, lli minVal)
{
    ans[nn] = max(abs(maxVal - val[nn]),abs(minVal - val[nn]));
    for(auto v: g[nn])
    {
        if(v!=pp)
        {
            dfs(v,nn,max(maxVal,val[nn]),min(minVal,val[nn]));
        }
    }
}
```

Qs

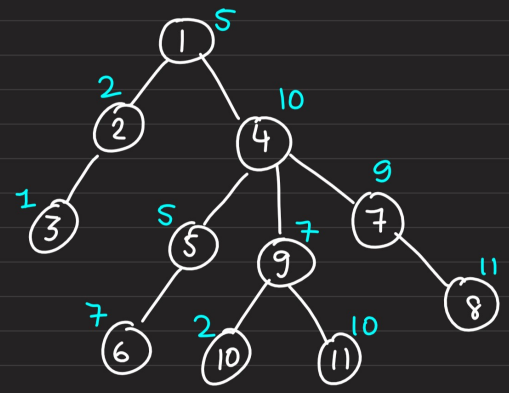


— → value of node

} root, tree is given

for → all nodes x, find min (|val[x] - val[y]|)
where y is ancestor of x.

Soln: We want to create a data-structure that will return closest value to val[x]



for x = 10

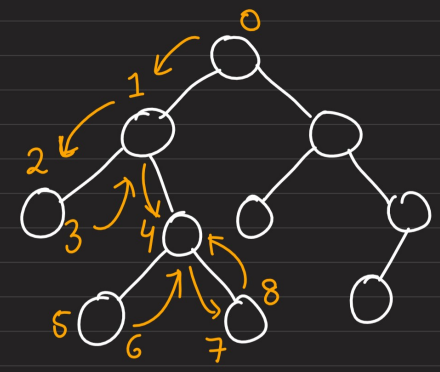
dns = |2-5| = 3

→ Maintain multiset.

Euler Tour on Tree (Query on Tree)

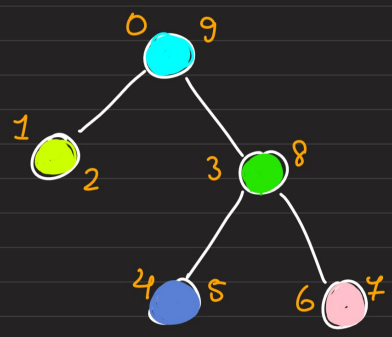
↳ All edges covered exactly twice.

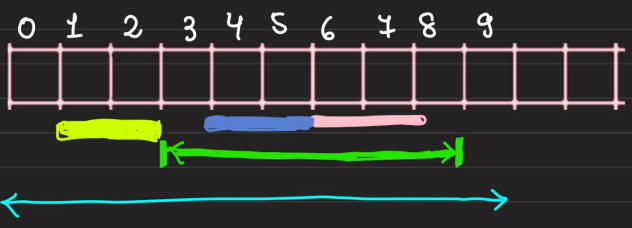
nothing just DFS + backtracking ← every edge covered twice

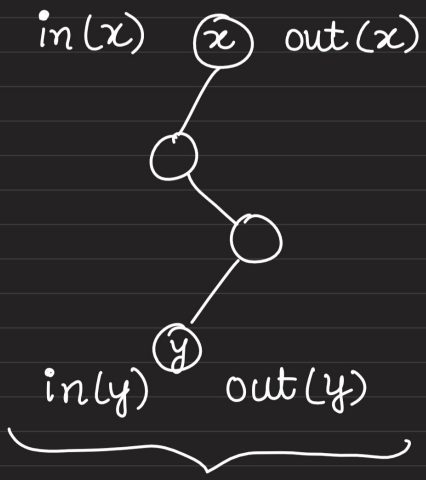


in time — out time — time(s) at which nodes where traversed.

Property







→ in(x) < in(y)

→ out(x) > out(y)

Relation: in(x) < in(y) < out(y) < out(x)

y ∈ subtree(x)

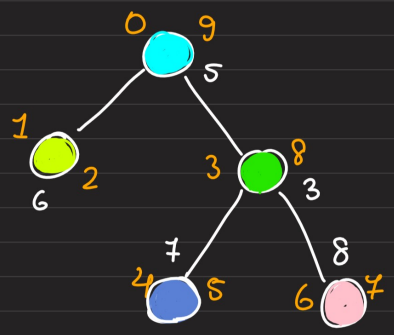
for, $y \in$ subtree(x)

$In(x) < In(y) < Out(y) < Out(x)$ } O(1) check

↳ We can check this inequality if we want to check if $y \in$ subtree(x) or not.

In one dfs, we can find the in/out values. Then, we can process the queries.

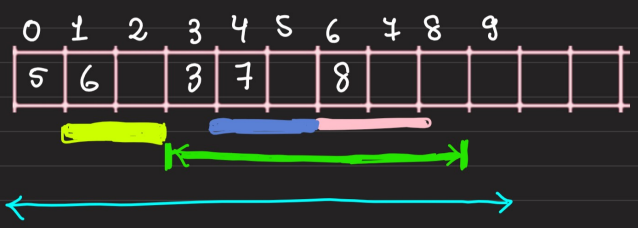
Building In/Out values :-



– } node value

– } In/Out

Flatening the tree



} We keep value of node at its invalue.

Standard Problems :

1) Find min^{n} no. of colours to colour all the nodes of the tree so that no 2 neighbours have the same colour.

Ans → 2 (Colour Alternatively)

All tree are Bi-partite.

2) Find min^{n} no. of colours to colour all the nodes of the tree so that no 2 neighbours and no 2 siblings have the same colour.

Ans → max (degree of node(s)) + 1