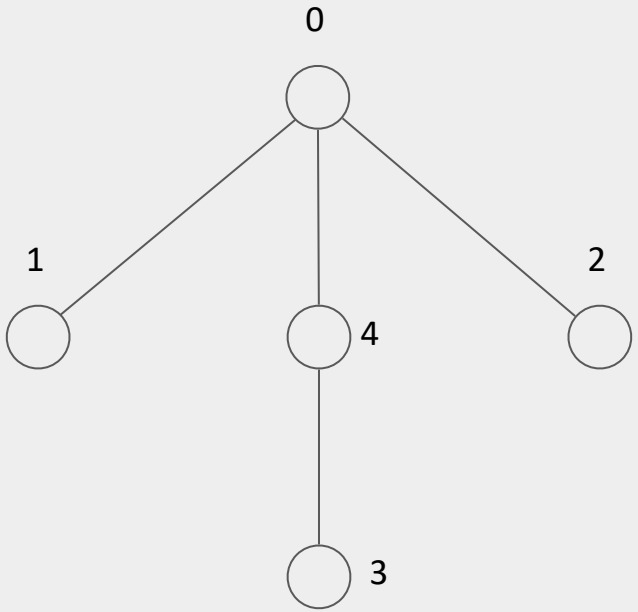
LCA

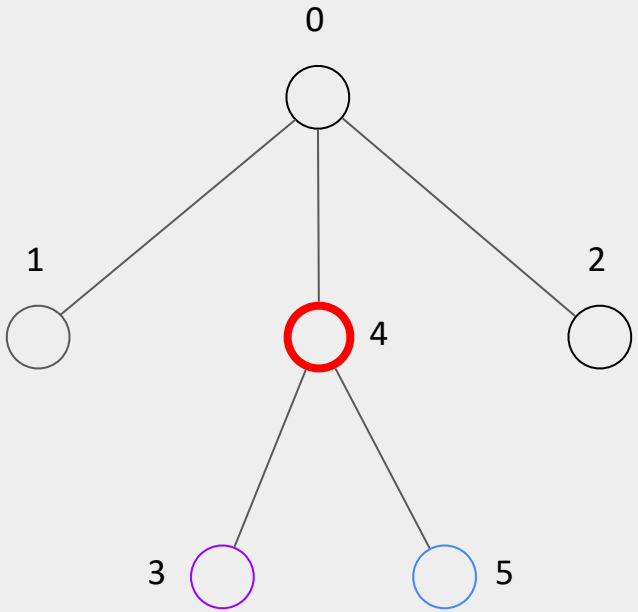
0 is ancestor for (1, 3, 4, 2), 4 is ancestor for 3.

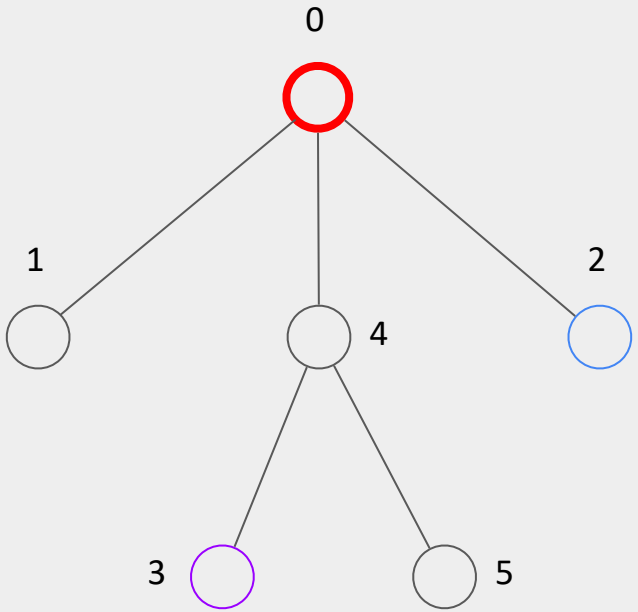


LCA

Let's call the Least Common Ancestor (LCA) of (a, b) the node c, such that c is a common ancestor of both a and b and, among all such nodes, c has the greatest depth.

Furthermore, for theoretical simplicity, we assume that a is not an ancestor of b, and b is not an ancestor of a.


0
1
2
4
3
5


0
1
2
4
3
5

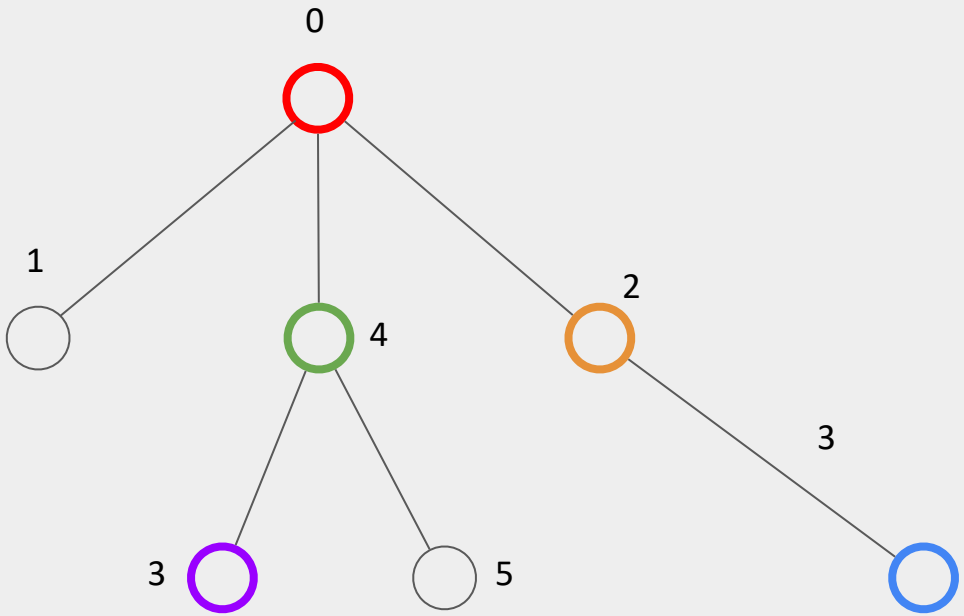
How to solve this?

We are already able to check whether one vertex is an ancestor of another or not. To do this, we simply need to use 'tin' and 'tout'.

How to solve this?

Let's traverse all vertices and find the LCA(a, b) according to the definition, which is a vertex x that is an ancestor of both a and b and has the maximum depth.

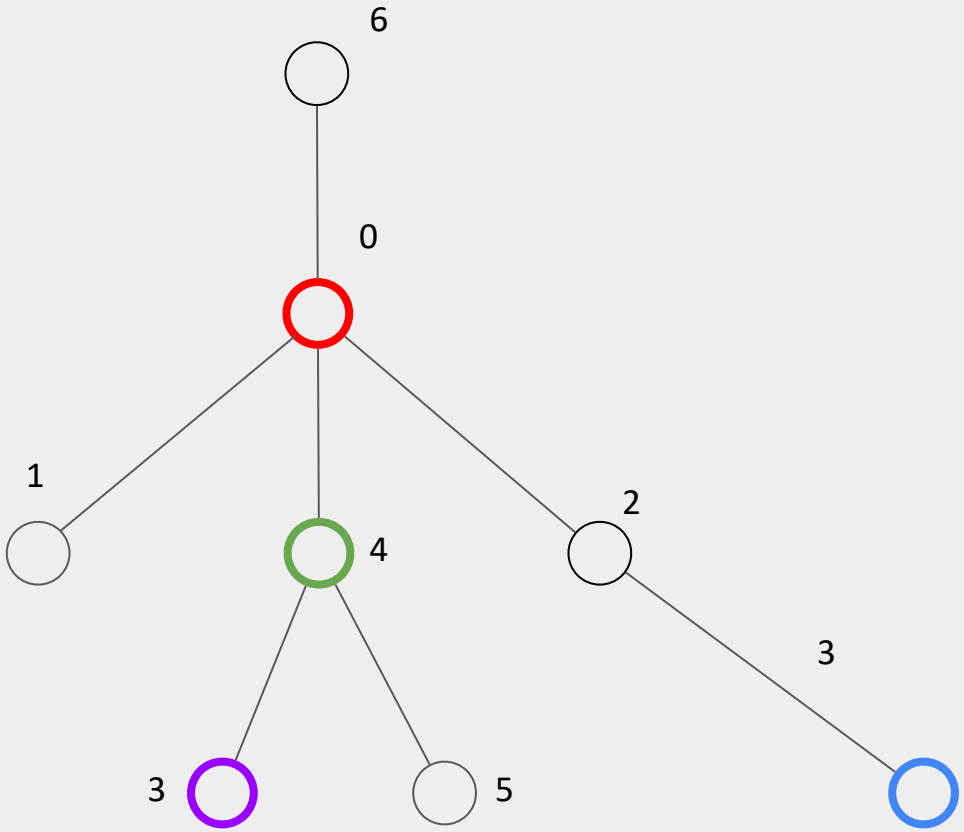
blue - b, violet - a.
red - parent of a an b, green - parent of only a, yellow - only of b



Optimization

Actually, you don't even need to traverse all the vertices; you can just go upwards from vertex a and find the first vertex c that is an ancestor of b.

blue - b, violet - a.
red - parent of a an b, green - parent of only a



Faster

Let’s calculate dp.

dp[i][dist] - ancestor of i with distance dist.

dp[i][0] = i, dp[i][1] = parent[i]

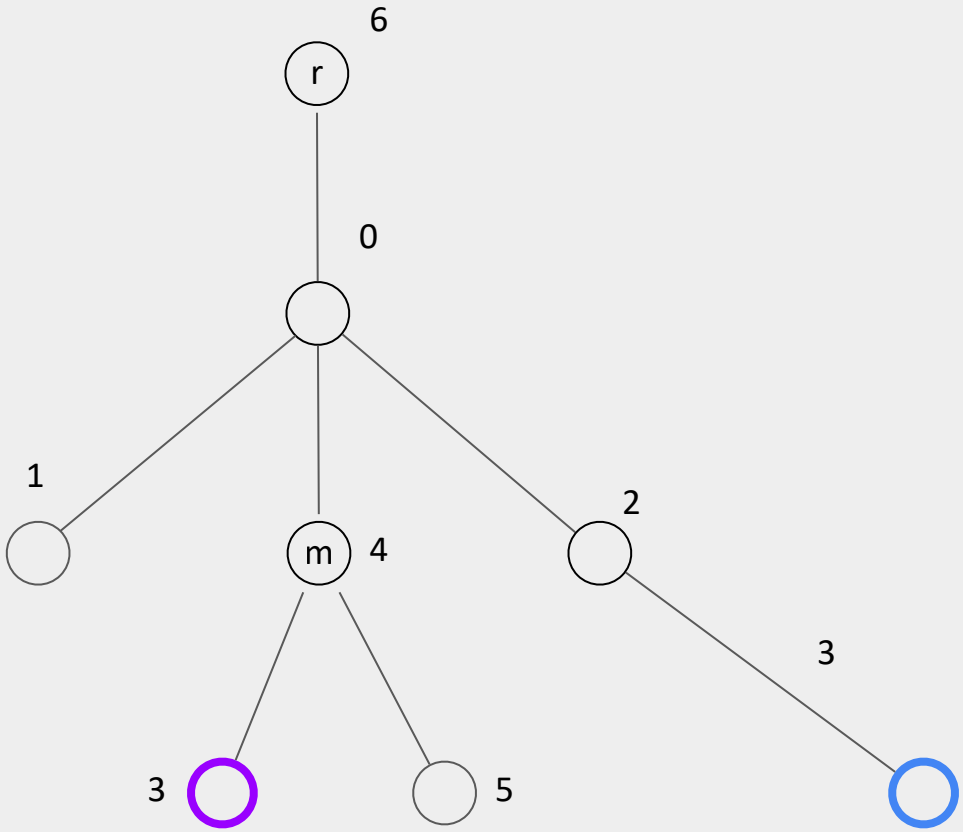
dp[i][dist] = dp[dp[i][dist - 1]][1]

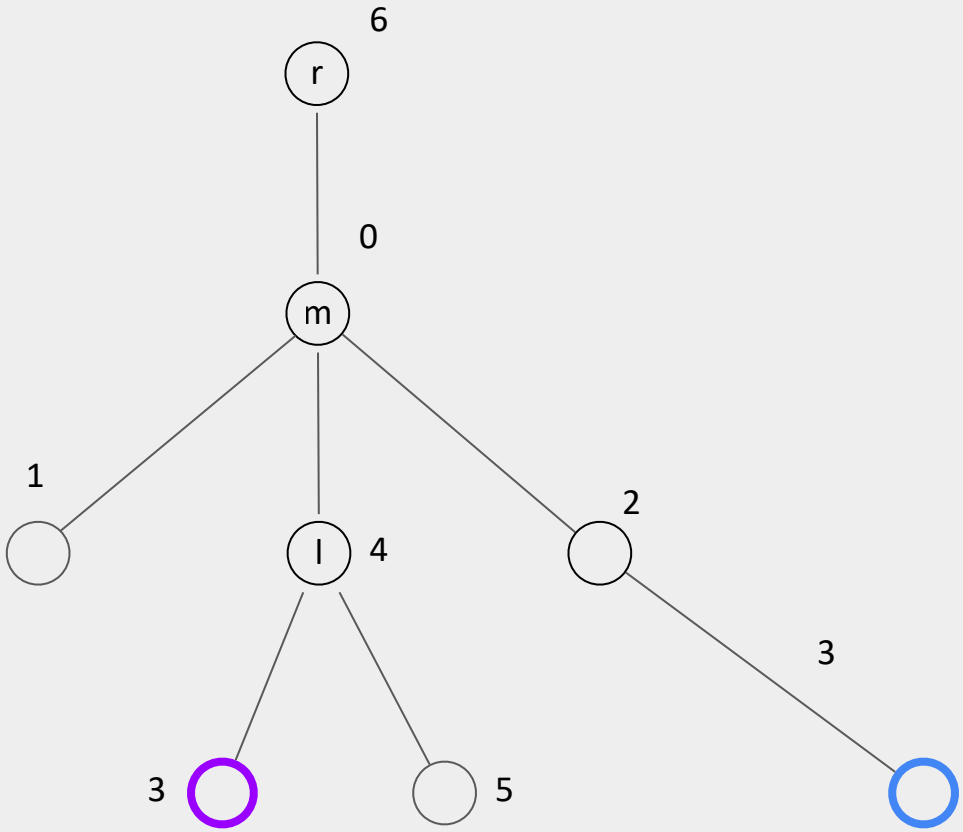
Faster

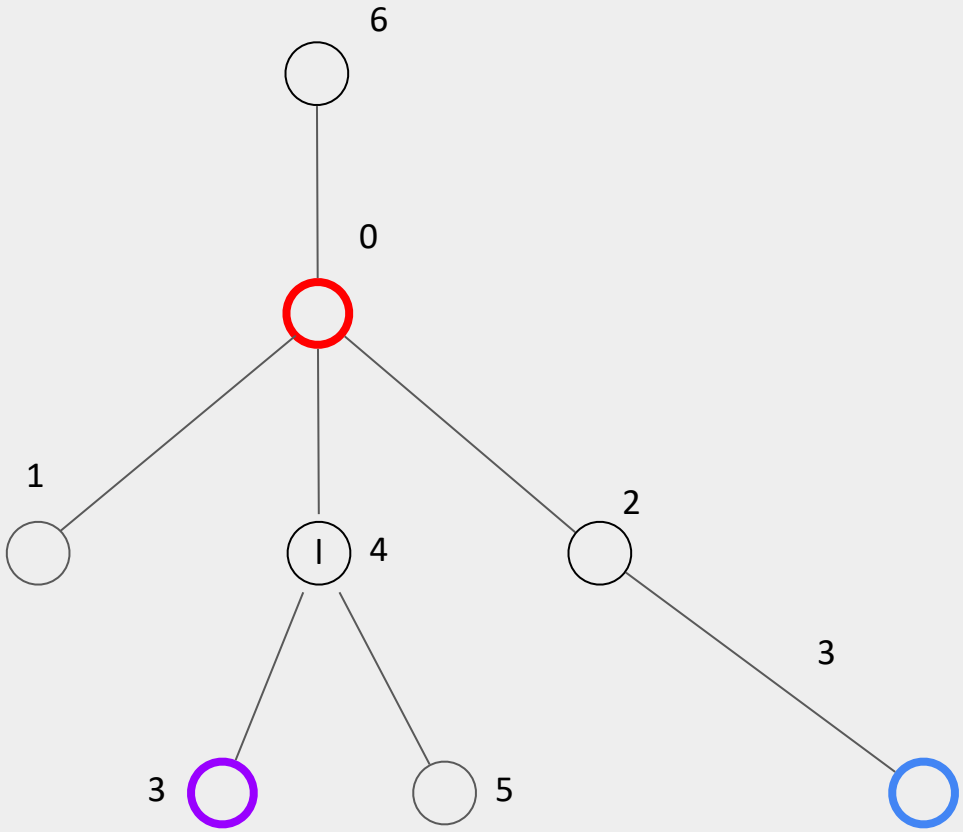
Now we can perform a binary search using this approach. The function 'is ancestor' will be 1 if going from the root, then 0; therefore, it suits binary search.

The left boundary (definitely not an ancestor) will be a vertex itself.

The right boundary (definitely an ancestor) will be the root.


6
r
0
1
m
4
2
3
3
5


6
r
0
m
1
l
4
2
3
3
5


6
0
1
2
4
3
3
5

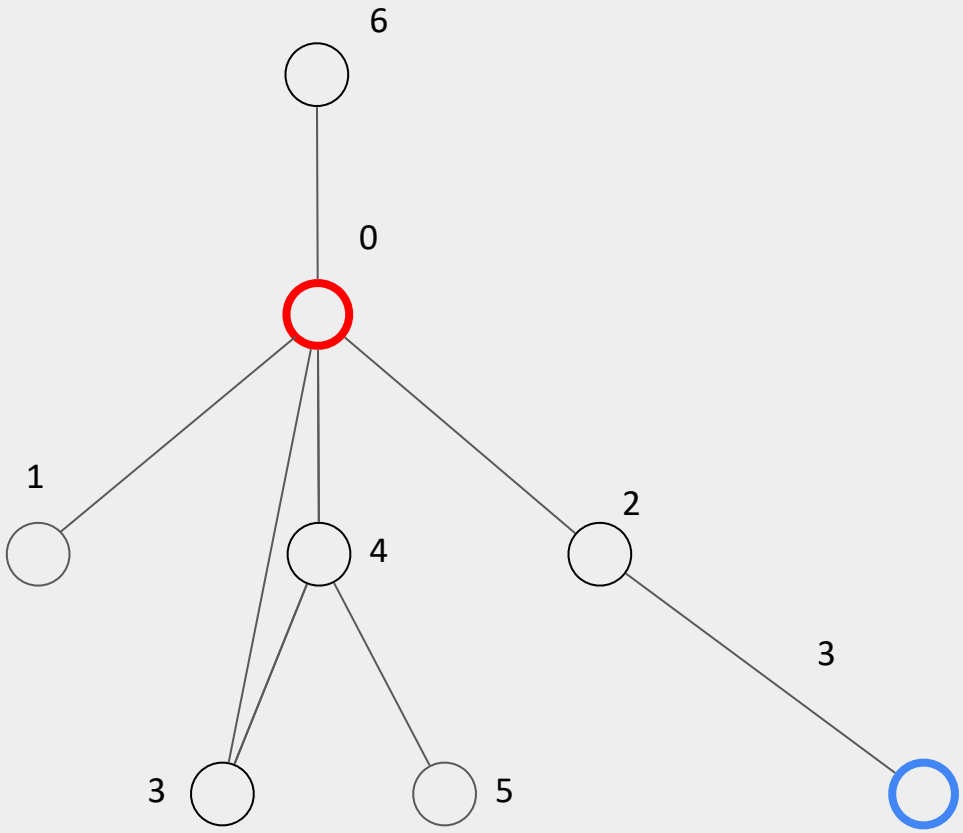
I’m speed

dp[i][j] - parent of i with distance (2^j).

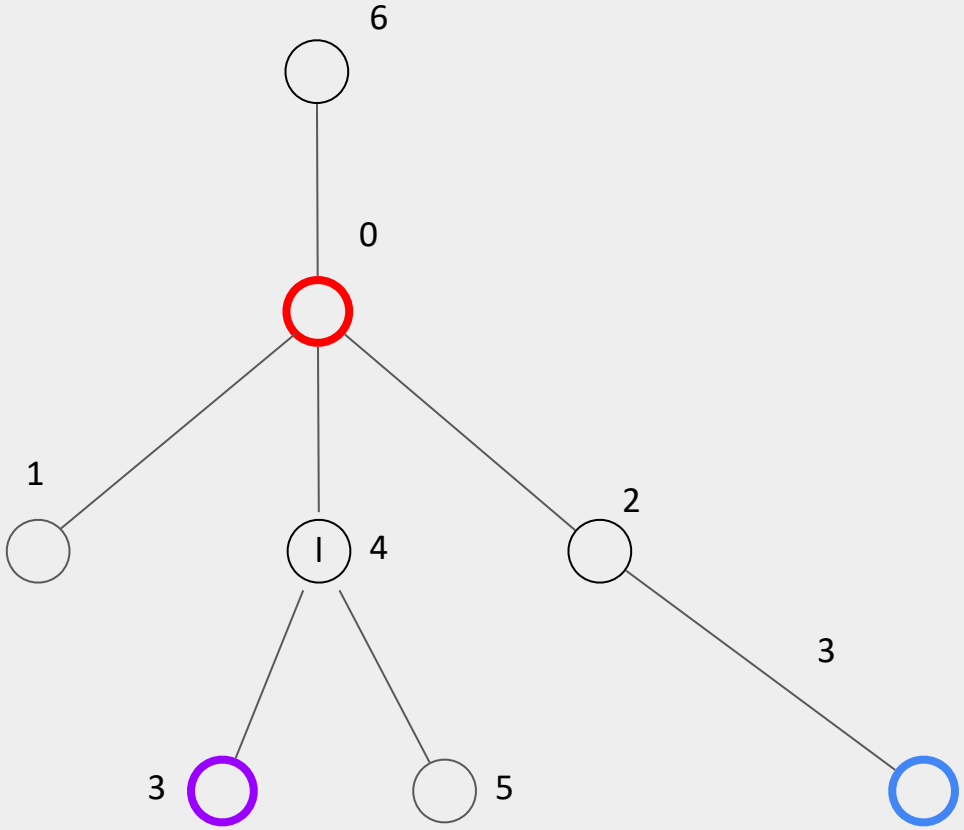
dp[i][0] = parent[i], dp[root][j] = root

dp[i][j + 1] = dp[dp[i][j]][j], because we found vertex a in distance 2^j from i, and then for this vertex a we also found an ancestor with distance 2^j. 2^(j + 1) = 2^j + 2^j

This dp name is binary up(binup)

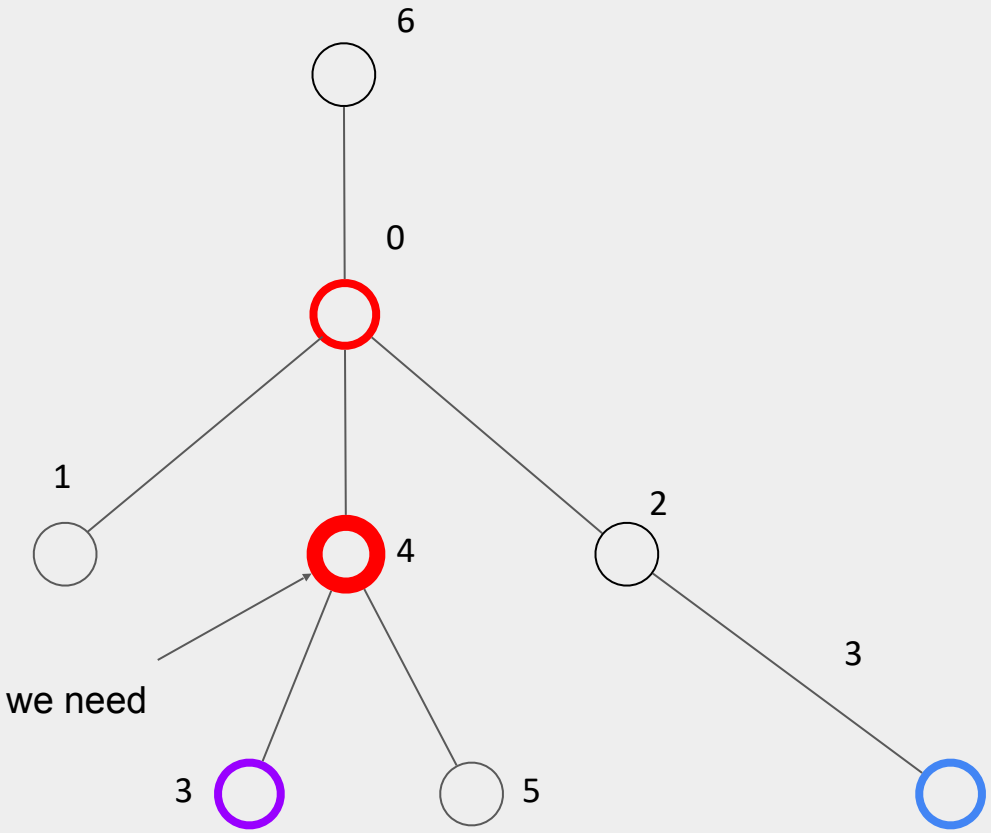

6
0
1
4
2
3
3
5

dp[3][0] = 4, dp[3][1] = 0, dp[3][2] = 6
dp[4][0] = 0, dp[4][1] = 6, dp[4][2] = 6



I’m speed

Let's now search for not the LCA but the child of LCA and the ancestor of a.


6
0
1
4
2
3
we need
3
5

I’m speed

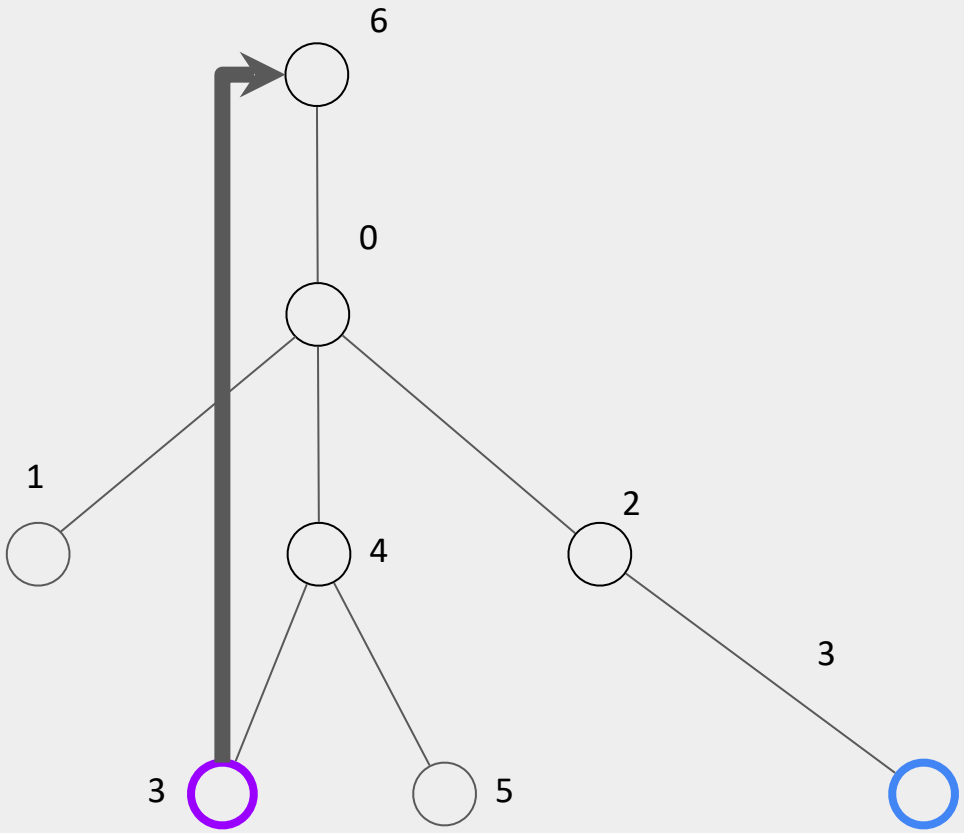
Let's go through the powers of two from i = log(n) to 1.

Suppose we are currently at the power of i, then let's check the ancestor of a at a distance of 2^i.

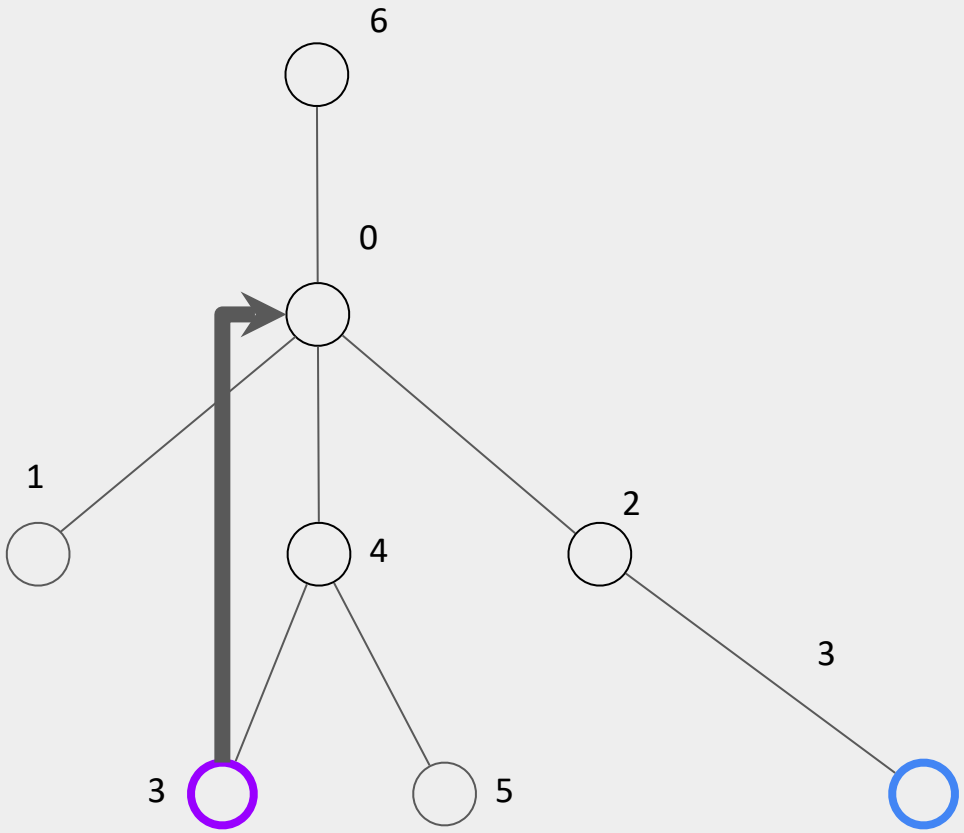
If it is not an ancestor of b, then we move to that ancestor.

If it is an ancestor of b, then we do nothing

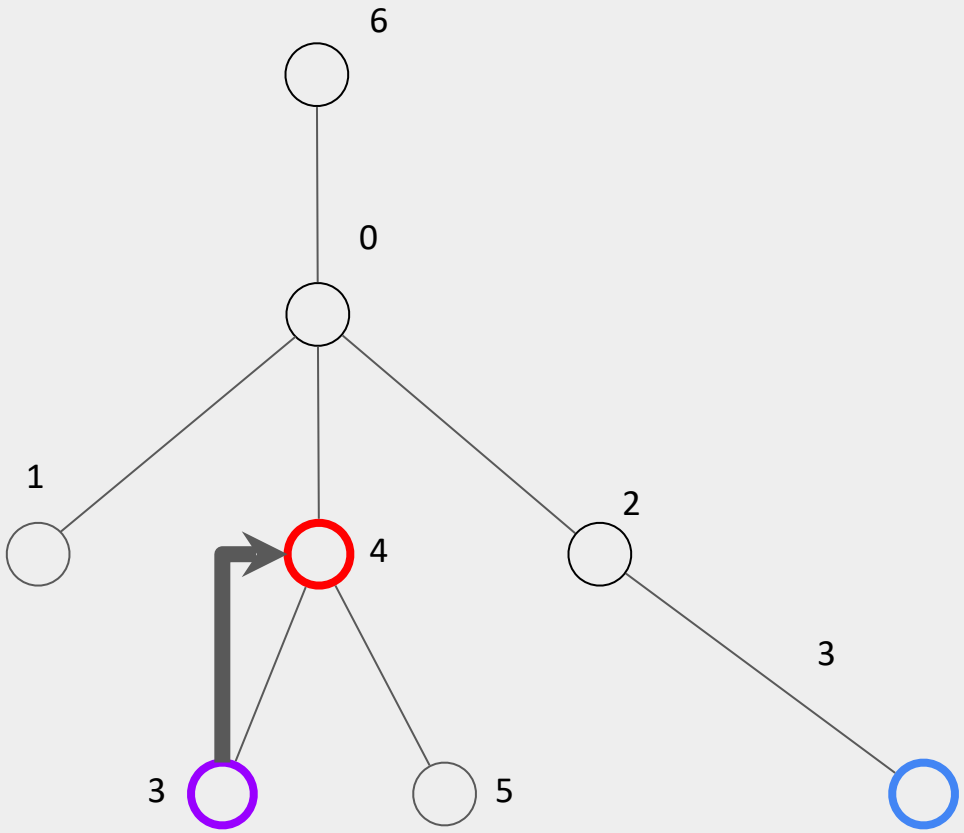
power = 2, vertex = 3, dist = 2^2 = 4, vertex_check = root



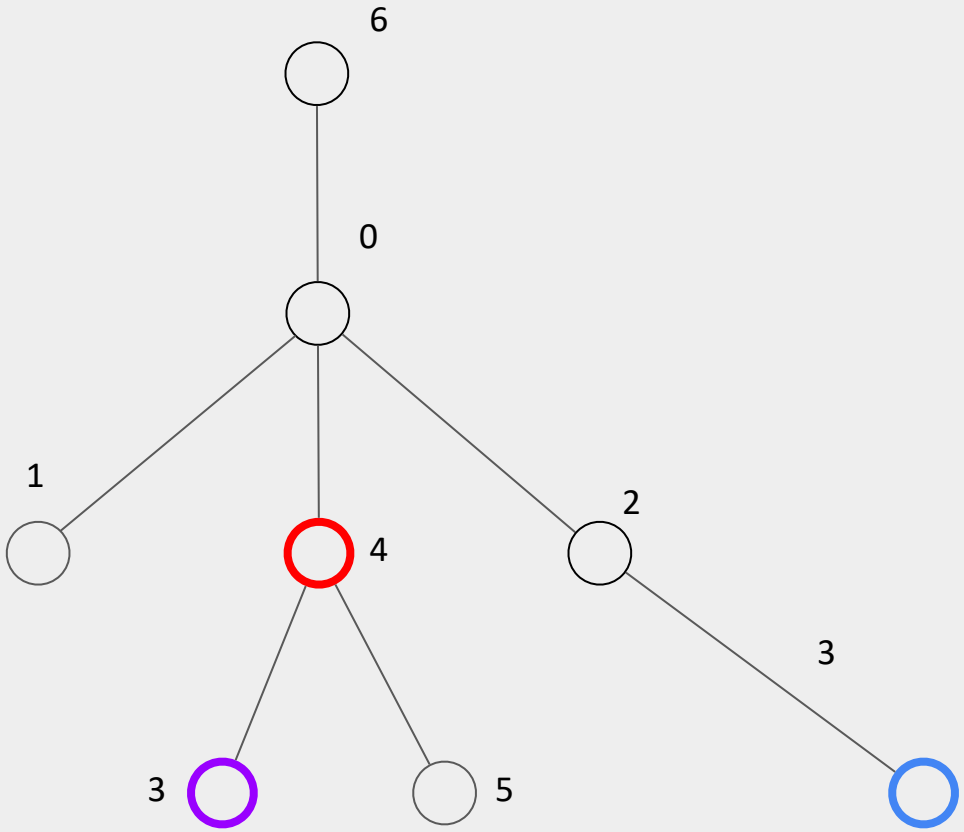
power = 1, vertex = 3, dist = 2^1 = 2, vertex_check = 0



power = 0, vertex = 3, dist = 2^0 = 1, vertex_check = 4



power = .., vertex = 4, the end of algo



I’m speed

Thus, we have found a vertex, which ancestor is the answer.

We achieved this in O(log(n)) per query, as for each query, we iterate through powers of two and simply check the element in the 'dp' array. The precomputation takes O(n log n) time, as for each vertex, we computed the logarithm of 'n' ancestors.

```cpp
const int MAXN = 100005;
const int MAXLOG = 20;

vector<int> tin(MAXN), tout(MAXN), parent(MAXN);
vector<vector<int>> up(MAXN, vector<int>(MAXLOG));

int timer = 0;

void dfs(int v, int p, vector<vector<int>>& adj) {
    tin[v] = ++timer;
    up[v][0] = p;

    for (int i = 1; i < MAXLOG; ++i) {
        up[v][i] = up[up[v][i - 1]][i - 1];
    }

    for (int u : adj[v]) {
        if (u != p) {
            dfs(u, v, adj);
        }
    }

    tout[v] = ++timer;
}
```

```
31.  bool isAncestor(int u, int v) {
32.      return tin[u] <= tin[v] && tout[u] >= tout[v];
33.  }
34.
35.  int lca(int u, int v) {
36.      if (isAncestor(u, v)) {
37.          return u;
38.      }
39.      if (isAncestor(v, u)) {
40.          return v;
41.      }
42.      for (int i = MAXLOG - 1; i >= 0; --i) {
43.          if (!isAncestor(up[u][i], v)) {
44.              u = up[u][i];
45.          }
46.      }
47.      return up[u][0];
48.  }
```

```
    vector<vector<int>> adj(n);

    for (int i = 0; i < n - 1; ++i) {
        int u, v;
        cin >> u >> v;
        u--; v--;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    int root = 0;

    dfs(root, root, adj);

    int u, v;
    cin >> u >> v;
    u--; v--;
    cout << "LCA of " << u << " and " << v << " is " << lca(u, v) << endl;

    return 0;
}
```

Success #stdin #stdout 0.01s 15956KB

comments (0)

stdin

copy

```
7
1 2
1 3
2 4
2 5
3 6
3 7
4 6
```

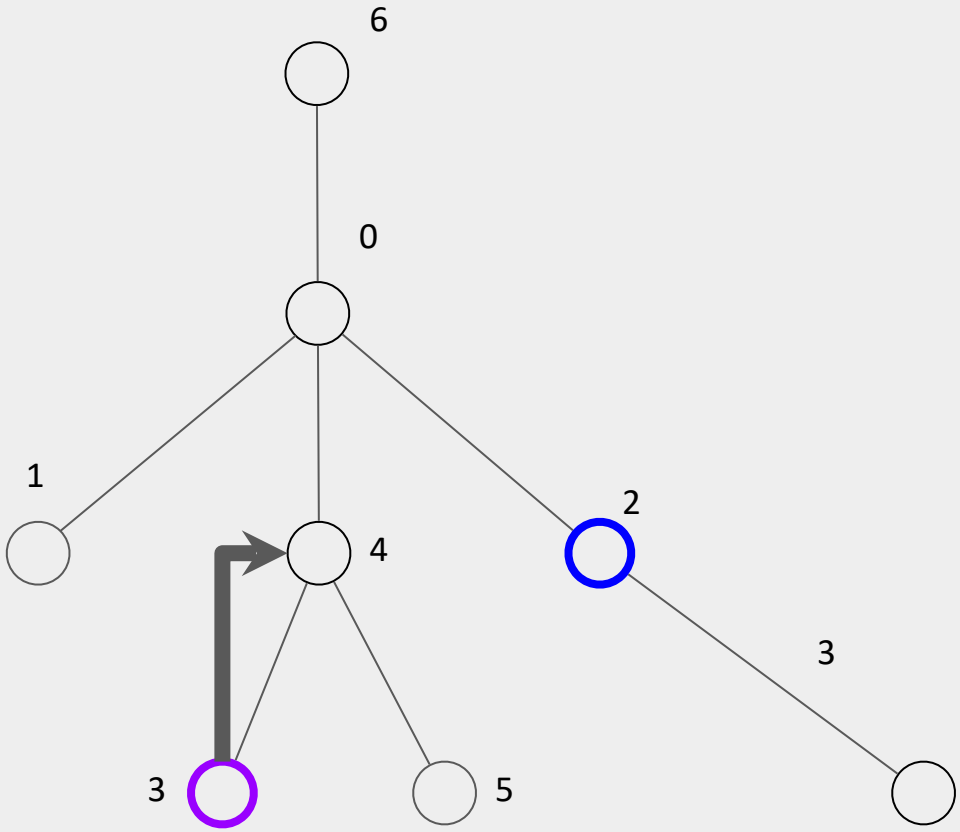
stdout

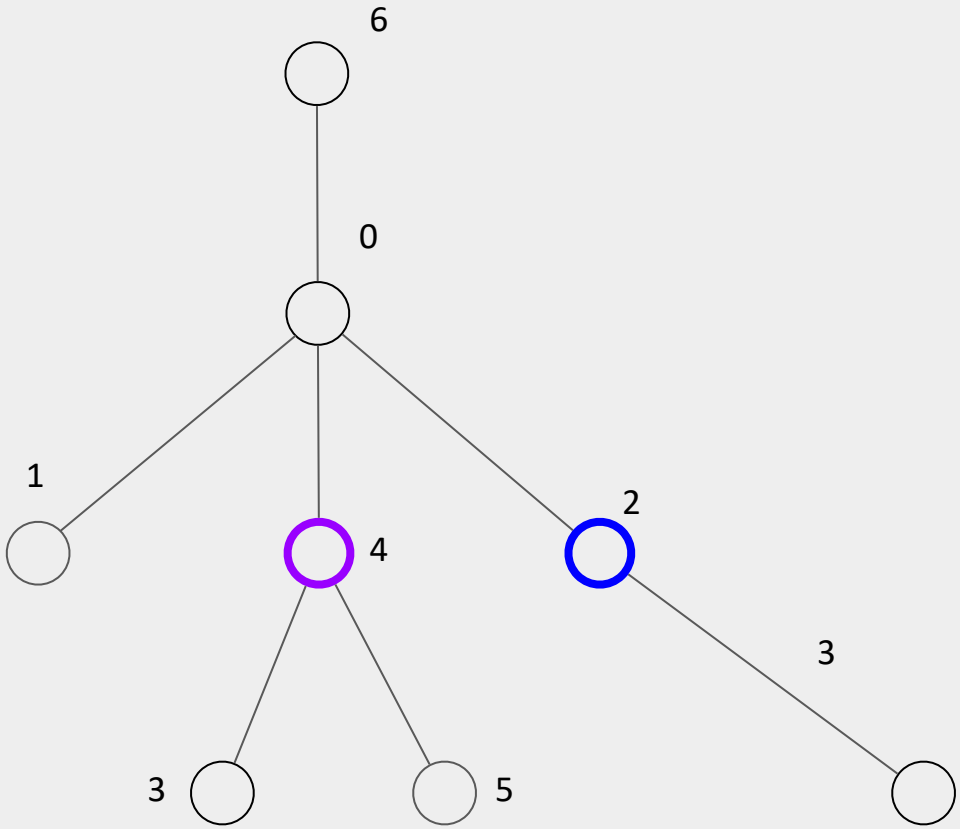
copy

```
LCA of 3 and 5 is 0
```

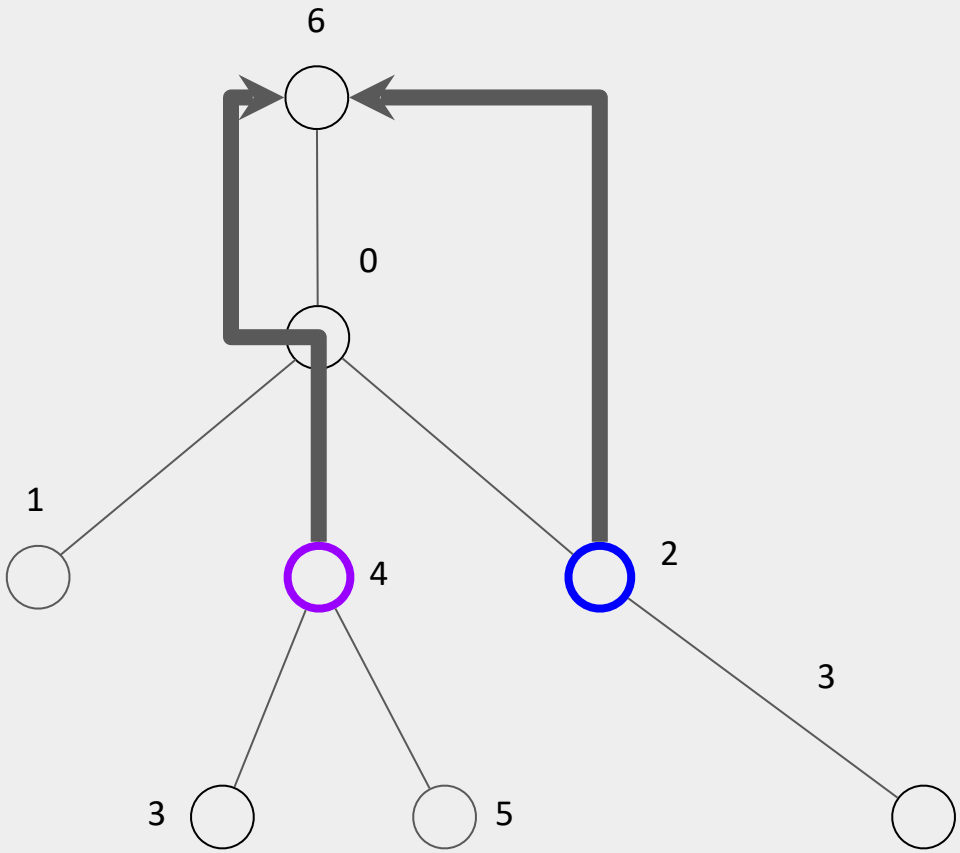
Other usage of binup

Also, there is a beautiful alternative method. Instead of checking for an ancestor directly, we can first reach the same depth for the vertices and then check for ancestor equality.

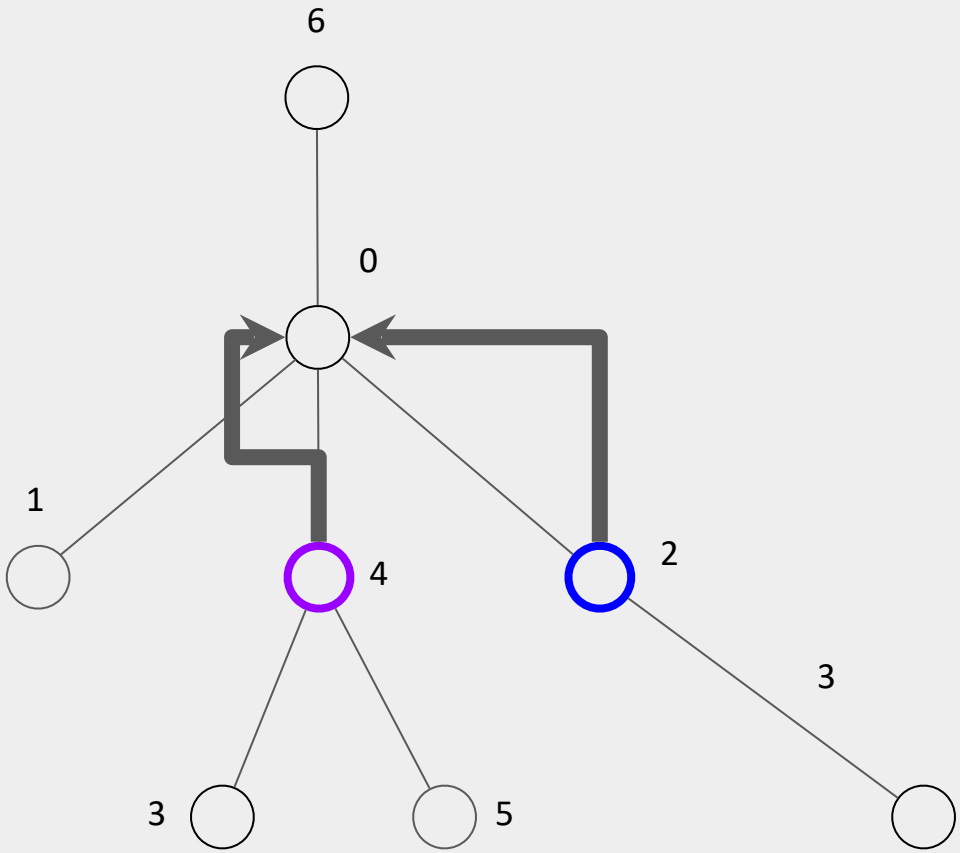

6
0
1
4
2
3
3
5


6
0
1
4
2
3
3
5

power = 1, vertex = (4, 2), dist = 2^1 = 2, vertex_check = (root, root)



power = 0, vertex = (4, 2), dist = 2^0 = 1, vertex_check = (0, 0)



```
const int MAXN = 100005;
const int MAXLOG = 20;

int parent[MAXN];
int depth[MAXN];
int table[MAXN][MAXLOG];

void preprocess(int n) {
    for (int i = 0; i < n; i++) {
        table[i][0] = parent[i];
    }

    for (int j = 1; (1 << j) < n; j++) {
        for (int i = 0; i < n; i++) {
            table[i][j] = table[table[i][j - 1]][j - 1];
        }
    }
}
```

```
int lca(int u, int v) {
    if (depth[v] < depth[u]) {
        swap(u, v);
    }

    for (int i = MAXLOG - 1; i >= 0; i--) {
        if (depth[v] - (1 << i) >= depth[u]) {
            v = table[v][i];
        }
    }

    if (u == v) {
        return u;
    }

    for (int i = MAXLOG - 1; i >= 0; i--) {
        if (table[u][i] != -1 && table[u][i] != table[v][i]) {
            u = table[u][i];
            v = table[v][i];
        }
    }

    return parent[u];
}
```

```
int main() {
    parent[0] = 0;
    parent[1] = 0;
    parent[2] = 0;
    parent[3] = 1;
    parent[4] = 1;
    parent[5] = 2;
    parent[6] = 2;

    depth[0] = 0;
    for (int i = 1; i < MAXN; i++) {
        depth[i] = depth[parent[i]] + 1;
    }
    preprocess(MAXN);

    int u = 4, v = 6;
    cout << "LCA of " << u << " and " << v << " is " << lca(u, v) << endl;

    return 0;
}

```

Success #stdin #stdout 0.01s 11928KB

comments (0)

stdin

copy

```
Standard input is empty
```

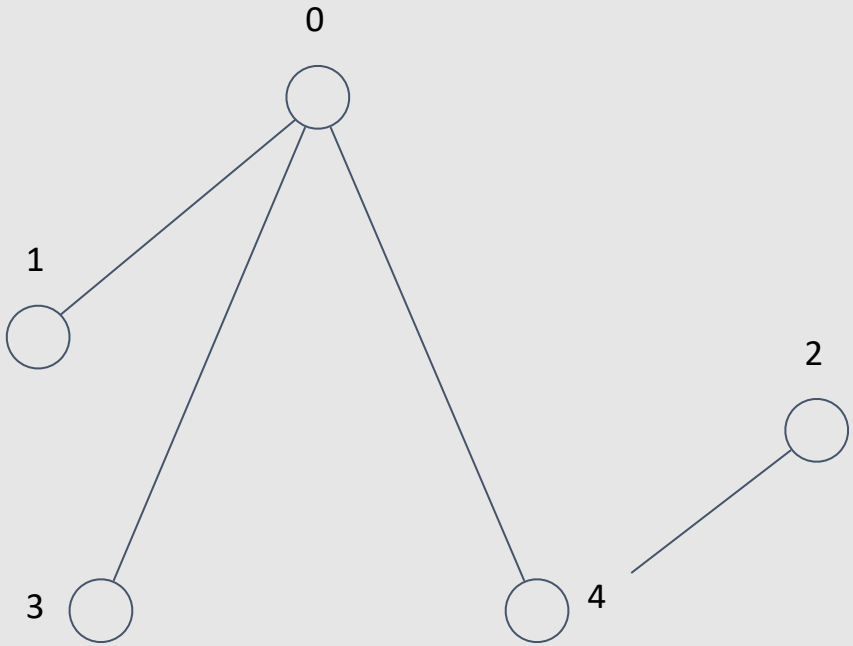
stdout

copy

```
LCA of 4 and 6 is 0
```

The second euler tour.

Let's recall what the second Euler tour is. We add a vertex to the Euler tour if we have just entered it or just returned from it.


0
1
2
3
4

Euler tour

- Example - [0, 1, 0, 3, 0, 4, 2, 4, 0]

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

void dfs(int u, const vector<vector<int> > &g, vector<bool> &visited, int &timer, vector<int> &eul
er) {
    if (visited[u]) {
        return;
    }
    euler.push_back(u);
    visited[u] = true;
    for (auto v: g[u]) {
        dfs(v, g, visited, timer, euler);
        euler.push_back(u);
    }
}

int main() {
    int n, m, timer = 0;
    cin >> n >> m;
    vector<vector<int> > graph(n);
    vector<bool> visited(n);
    vector<int> euler;

    for (int i = 0; i < m; i++) {
        int from, to;
        cin >> from >> to;
        from--;
        to--;
        graph[from].push_back(to);
    }
```

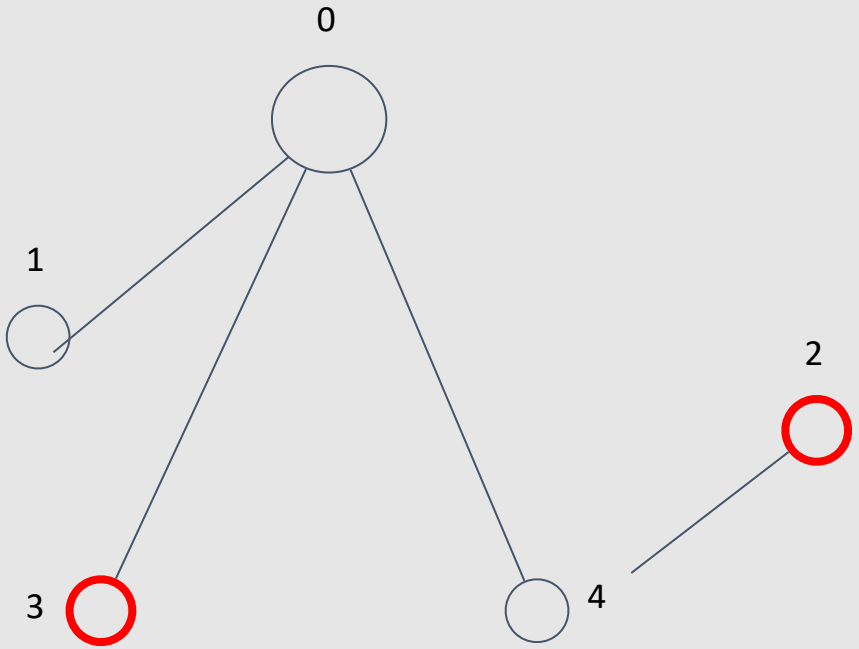
Euler tour

Let's take a closer look at this traversal. Suppose we want to find, for example, the LCA of vertices 3 and 2. Notice that if we take any segment between their encounters in the Euler tour, we will have only three types of vertices:

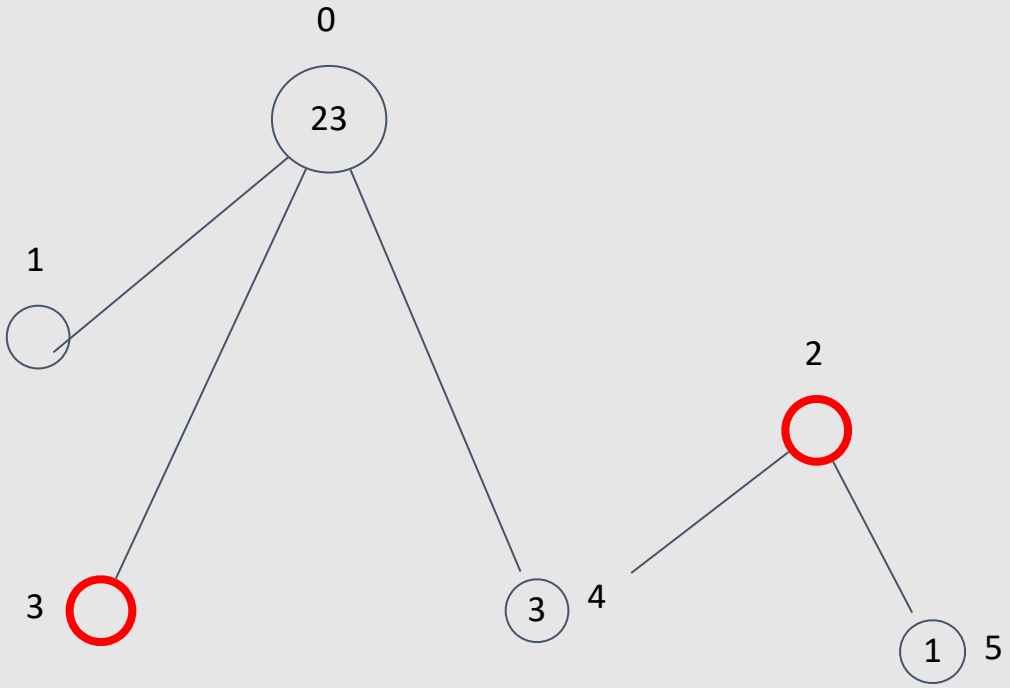
Euler tour

1) Irrelevant nodes - vertices that are not ancestors of either a or b.
2) Nodes that are ancestors of a.
3) Nodes that are ancestors of b.

[0, 1, 0, 3, 0, 4, 2, 4, 0]



[0, 1, 0, 3, 0, 4, 2, 5, 2, 4, 0]



Euler tour

Notice that in this segment, there will always be LCA(a, b) because we need to pass through it to reach from a to b. But how do we find it?

Idea

In fact, it's quite simple: the LCA has the shortest distance from the root in the segment, and consequently, the smallest depth. Therefore, we need to find the index of the minimum element in the Euler tour. That is, we encounter a problem we're already familiar with - RMQ (Range Minimum Query).

```cpp
const int MAXN = 100005;
const int MAXLOG = 20;

vector<int> euler, tin(MAXN), depth(MAXN);
vector<vector<int>> adj(MAXN);

void dfs(int v, int p, int d) {
    tin[v] = euler.size();
    euler.push_back(v);
    depth[v] = d;

    for (int u : adj[v]) {
        if (u != p) {
            dfs(u, v, d + 1);
            euler.push_back(v);
        }
    }
}

void buildTree(int v, int tl, int tr, vector<int>& tree) {
    if (tl == tr) {
        tree[v] = euler[tl];
    } else {
        int tm = (tl + tr) / 2;
        buildTree(v * 2, tl, tm, tree);
        buildTree(v * 2 + 1, tm + 1, tr, tree);
        int l = tree[v * 2];
        int r = tree[v * 2 + 1];
        tree[v] = (depth[l] < depth[r]) ? l : r;
    }
}
```

```
39. int queryTree(int v, int tl, int tr, int l, int r, vector<int>& tree) {
40.     if (l > r) return -1;
41.     if (l == tl && r == tr) return tree[v];
42. 
43.     int tm = (tl + tr) / 2;
44.     int left = queryTree(v * 2, tl, tm, l, min(r, tm), tree);
45.     int right = queryTree(v * 2 + 1, tm + 1, tr, max(l, tm + 1), r, tree);
46. 
47.     if (left == -1) return right;
48.     if (right == -1) return left;
49. 
50.     return (depth[left] < depth[right]) ? left : right;
51. }
52. 
53. int lca(int u, int v, int n, vector<int>& tree) {
54.     int left = tin[u];
55.     int right = tin[v];
56.     if (left > right) swap(left, right);
57. 
58.     return queryTree(1, 0, n - 1, left, right, tree);
59. }
```

```cpp
int main() {
    int n;
    cin >> n;

    for (int i = 0; i < n - 1; ++i) {
        int u, v;
        cin >> u >> v;
        u--;
        v--;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    int root = 0;

    dfs(root, -1, 0);

    vector<int> tree(4 * euler.size());
    buildTree(1, 0, euler.size() - 1, tree);

    int u, v;
    cin >> u >> v;
    u--;
    v--;
    cout << "LCA of " << u << " and " << v << " is " << lca(u, v, euler.size(), tree) << endl;

    return 0;
}
```

Success #stdin #stdout 0.01s 6272KB

stdin

```
7
1 2
1 3
2 4
2 5
3 6
3 7
4 6
```

stdout

```
LCA of 3 and 5 is 0
```

Asymptotic

The asymptotic complexity of this solution is O(n log n) for precomputation and O(log(n)) for each query.

Sparse table

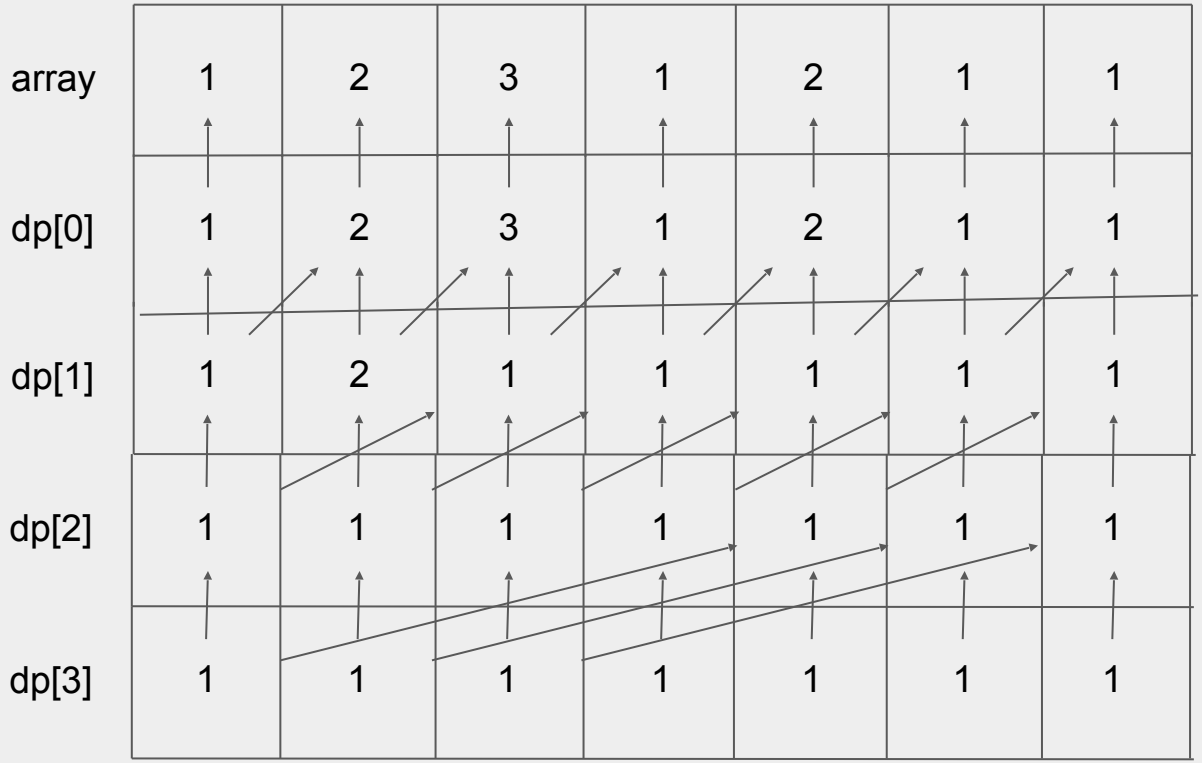
Furthermore, the problem doesn't involve any queries to change elements, so there might be a faster solution. Let's come up with a new structure based on 'binup'.

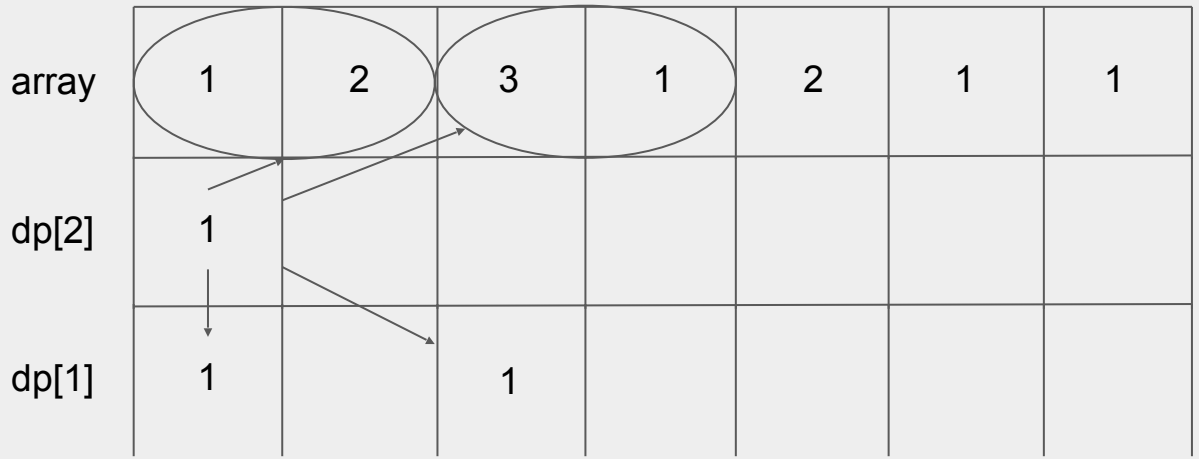
Sparse table

min[i][j] = min(a[i]..a[i + (2^j) - 1])

min[i][0] = a[i]

min[i][j + 1] = min(min[i][j], min[i + (2^j)][j])


array
1 2 3 1 2 1 1
dp[0]
1 2 3 1 2 1 1
dp[1]
1 2 1 1 1 1 1
dp[2]
1 1 1 1 1 1 1
dp[3]
1 1 1 1 1 1 1


array
1
2
3
1
2
1
1
dp[2]
1
dp[1]
1
1

RMQ

Suppose we have a segment [l, r], and we want to find the minimum within it.

How can we do that? We could do it similarly to 'binup', but that would take O(log(n)) time. But let's try to get O(1) approach.

RMQ

The first important observation is that the minimum operation is quite useful because min(a, a) = a.

Let's then simply find two (maybe overlapping) segments of the maximum power of two that is less, than r - l + 1.

min[1, 5]? length = 5, max_power = 2

RMQ

$c = \lfloor \log(r - l + 1) \rfloor$

$min[l, r] = min(dp[l][c], dp[r - (2^c)][c])$

```cpp
const int MAXN = 100005;
const int MAXLOG = 20;

vector<int> a(MAXN);
vector<vector<int>> table(MAXN, vector<int>(MAXLOG));

void buildSparseTable(int n) {
    for (int i = 0; i < n; ++i) {
        table[i][0] = a[i];
    }

    for (int j = 1; (1 << j) <= n; ++j) {
        for (int i = 0; i + (1 << j) <= n; ++i) {
            table[i][j] = min(table[i][j - 1], table[i + (1 << (j - 1))][j - 1]);
        }
    }
}
```

```
23.  int query(int l, int r) {
24.      int len = r - l + 1;
25.      int k = log2(len);
26.
27.      return min(table[l][k], table[r - (1 << k) + 1][k]);
28.  }
```

```
    buildSparseTable(n);

    int q;
    cin >> q;

    while (q--) {
        int l, r;
        cin >> l >> r;
        l--;
        r--;
        cout << "Minimum element in range [" << l << ", " << r << "]: " << query(l, r) << endl;
    }

    return 0;
}
```

Success #stdin #stdout 0.02s 15164KB

comments (0)

stdin

copy

```
7
1 2 3 2 1 3 2
3
1 4
2 4
2 5
```

stdout

copy

```
Minimum element in range [0, 3]: 1
Minimum element in range [1, 3]: 2
Minimum element in range [1, 4]: 1
```

Improvement

What's wrong with this code?

Actually, computing logarithms like this is not efficient because it takes quite a long time.

Let's precompute logarithms instead.

```
int query(int l, int r) {
    int len = r - l + 1;
    int k = log_2[len];

    return min(table[l][k], table[r - (1 << k) + 1][k]);
}

int main() {
    int n;
    cin >> n;
    log_2[1] = 0;
    log_2[0] = 0;

    for (int i = 0; i < n; ++i) {
        cin >> a[i];
    }

    for (int i = 2; i < MAXN; ++i) {
        log_2[i] = log_2[i / 2] + 1;
    }
```

Where can we use it?

We need an idempotent operation.

The definition from wikipedia says “It is the property of certain operations whereby they can be applied multiple times without changing the result beyond the initial application.”

Examples

min/max

or/and

gcd/lcm

abs

Tarjans algorithm

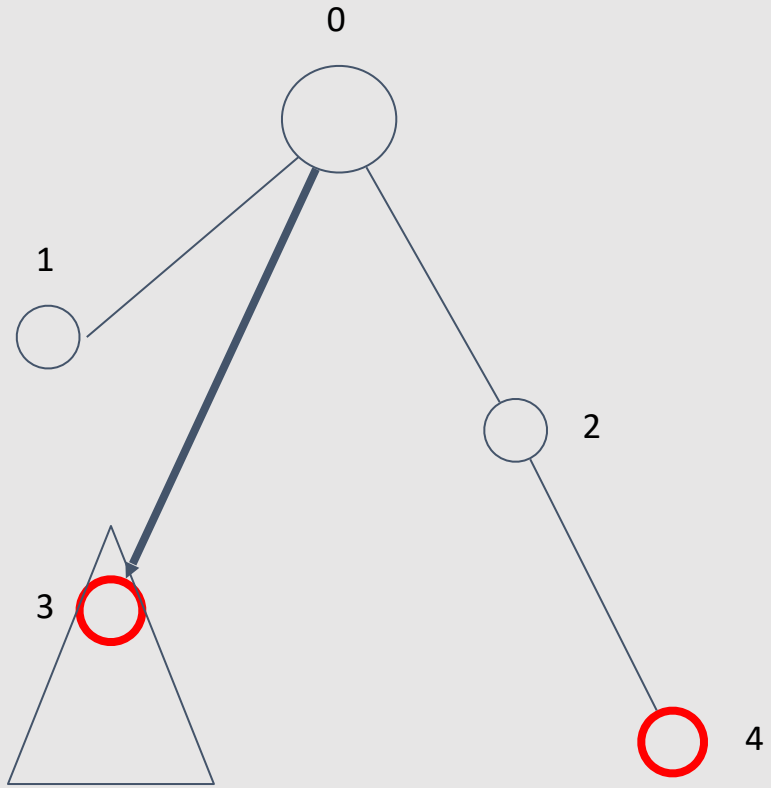
Suppose now we have all the queries given in advance(offline task). Is it possible to come up with something else?

Tarjans algorithm

It turns out that it's possible to find the LCA using DSU (Disjoint Set Union).

Let's start a DFS recursively from the root. Suppose we have just exited vertex u, reached from vertex v via edge (u, v). This means that we have completely traversed the subtree of vertex u. Therefore, if for any vertex in the subtree of u, the LCA has not yet been found, then it will be at least v.

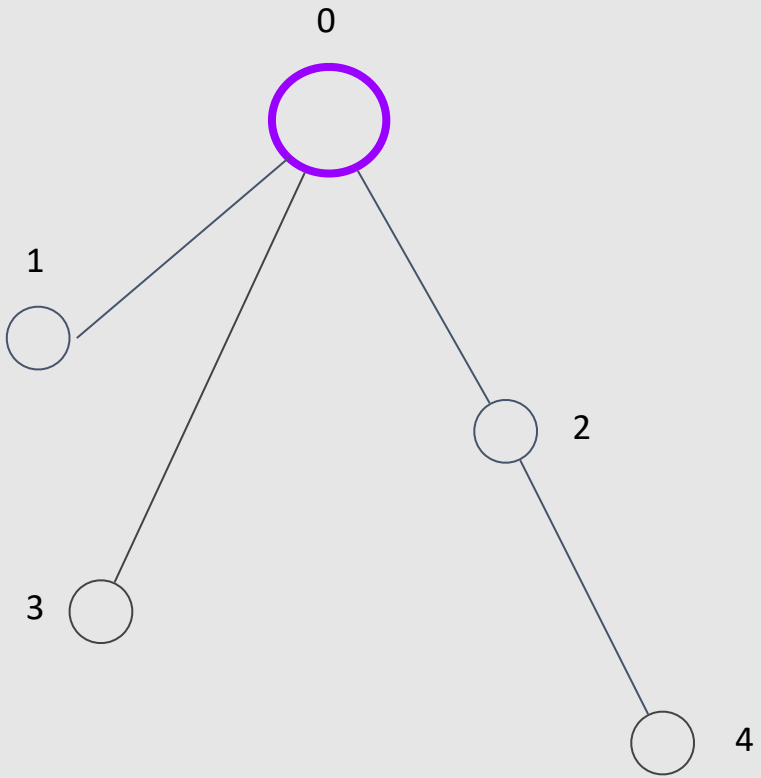
now in dsu all subtree of 3 in 0-th set.



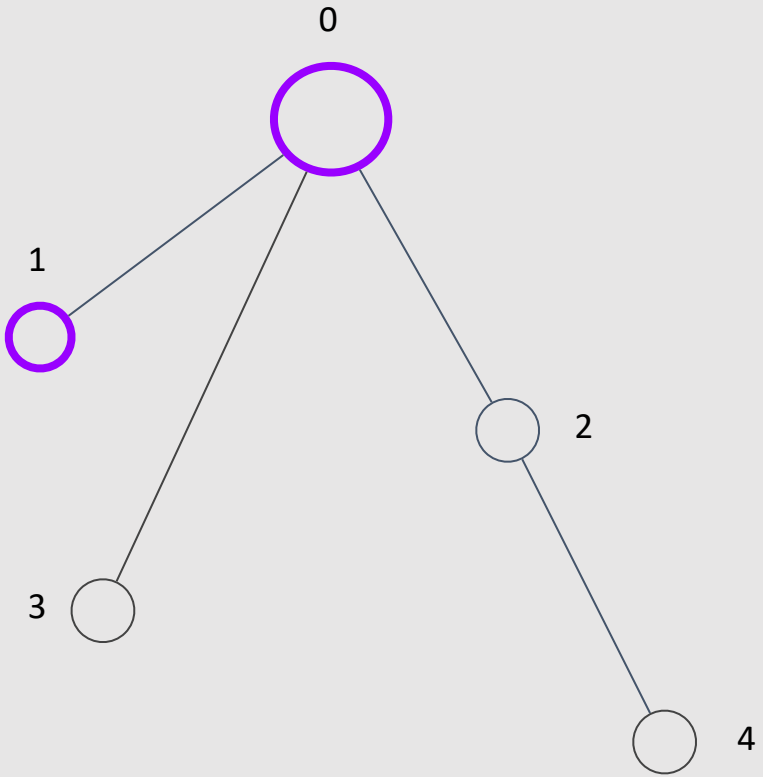
Tarjans algorithm

Suppose we have reached a certain vertex a for which we need to find LCA(a, b) and b is already visited. In this case, the answer will be dsu[b]. This is because it is the minimum vertex that is still open, and both a and b are its descendants.

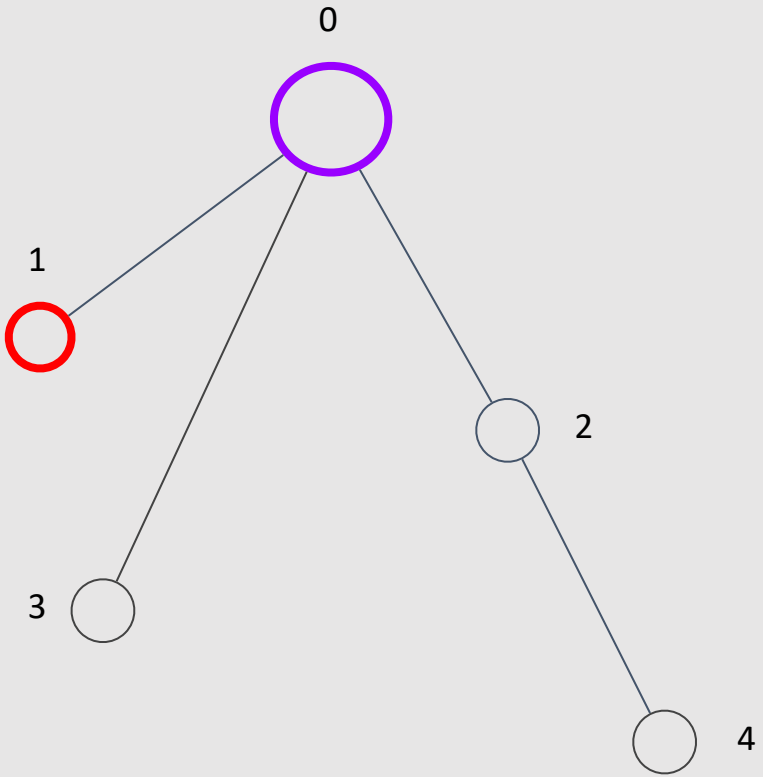
dsu = [0, 1, 2, 3, 4], query = {{3, 4}, {2, 4}}



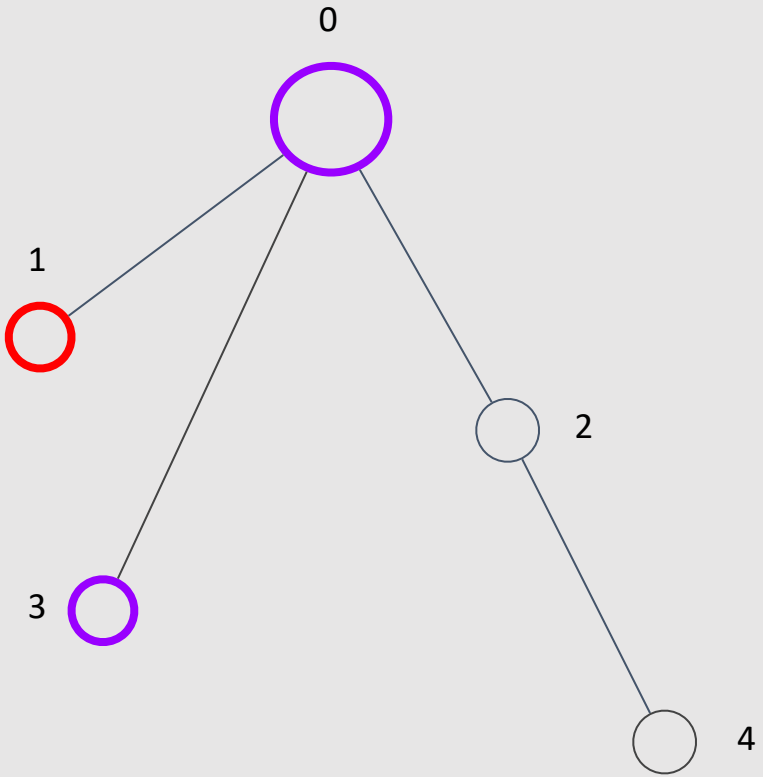
dsu = [0, 1, 2, 3, 4], query = {{3, 4}, {2, 4}}



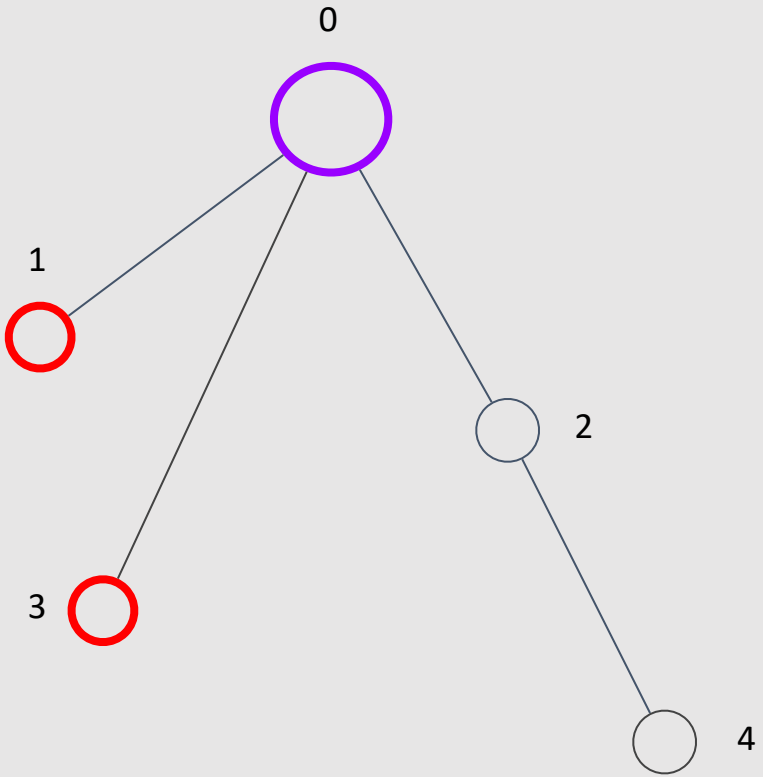
dsu = [0, 0, 2, 3, 4], query = {{3, 4}, {2, 4}}



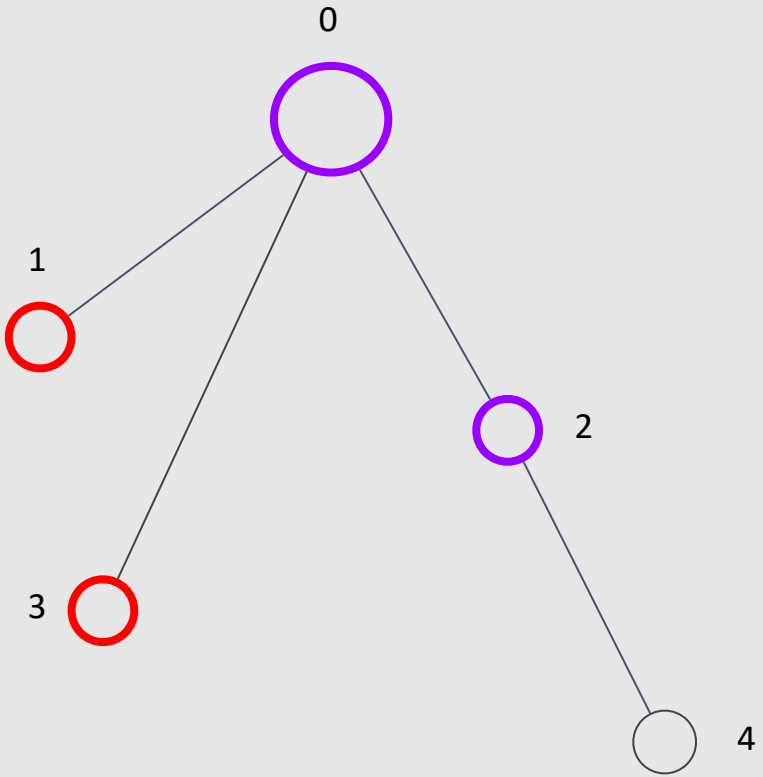
dsu = [0, 0, 2, 3, 4], query = {{3, 4}, {2, 4}}



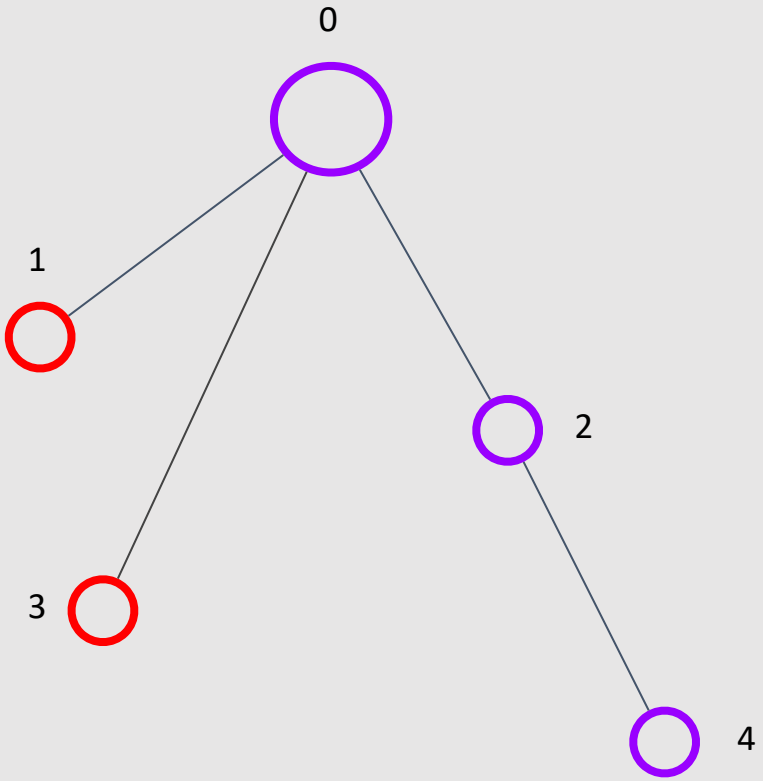
dsu = [0, 0, 2, 0, 4], query = {{3, 4}, {2, 4}}



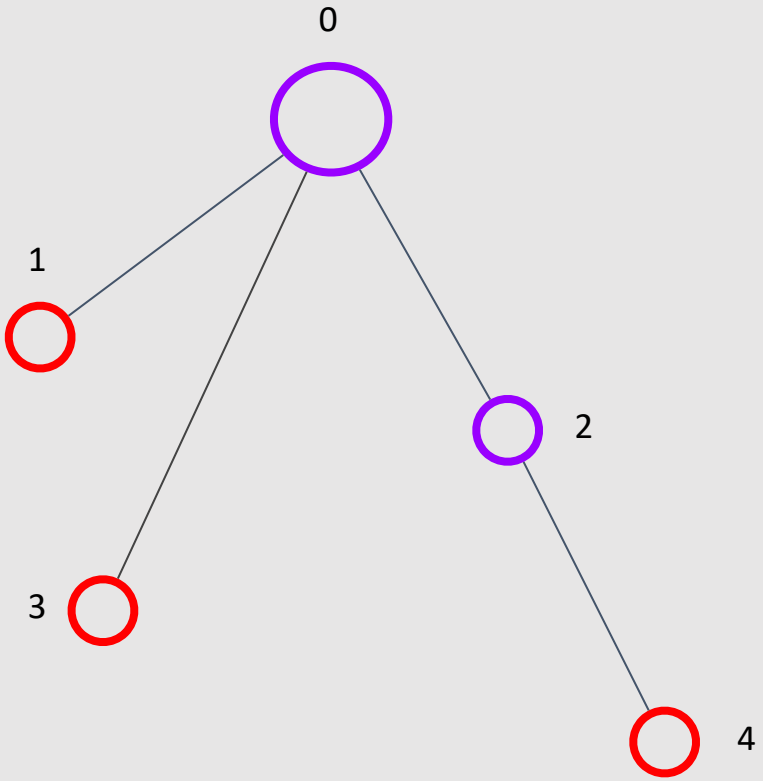
dsu = [0, 0, 2, 0, 4], query = {{3, 4}, {2, 4}}



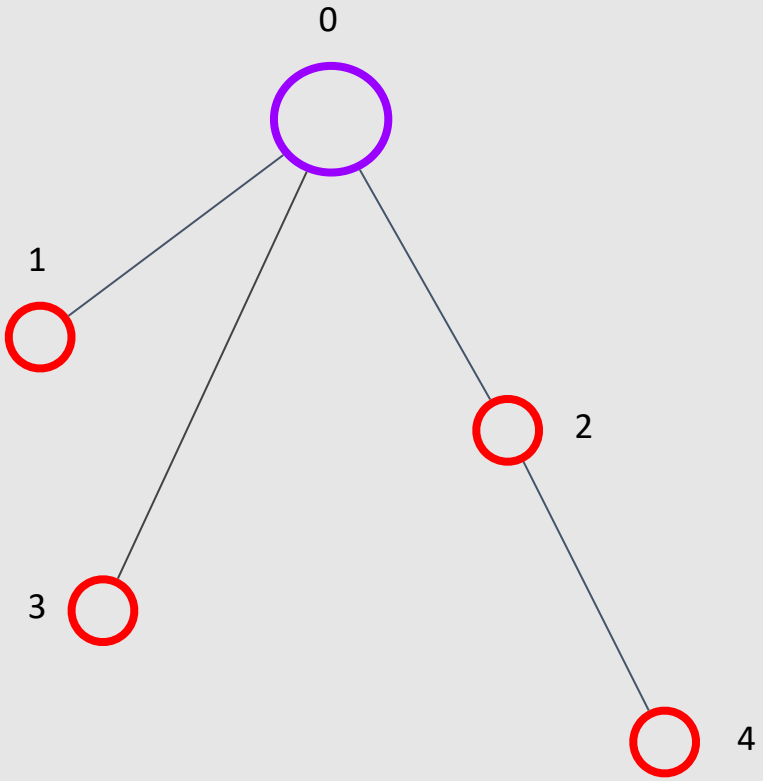
dsu = [0, 0, 2, 0, 4], query = {{3, 4}, {2, 4}}, answers = {0, …}



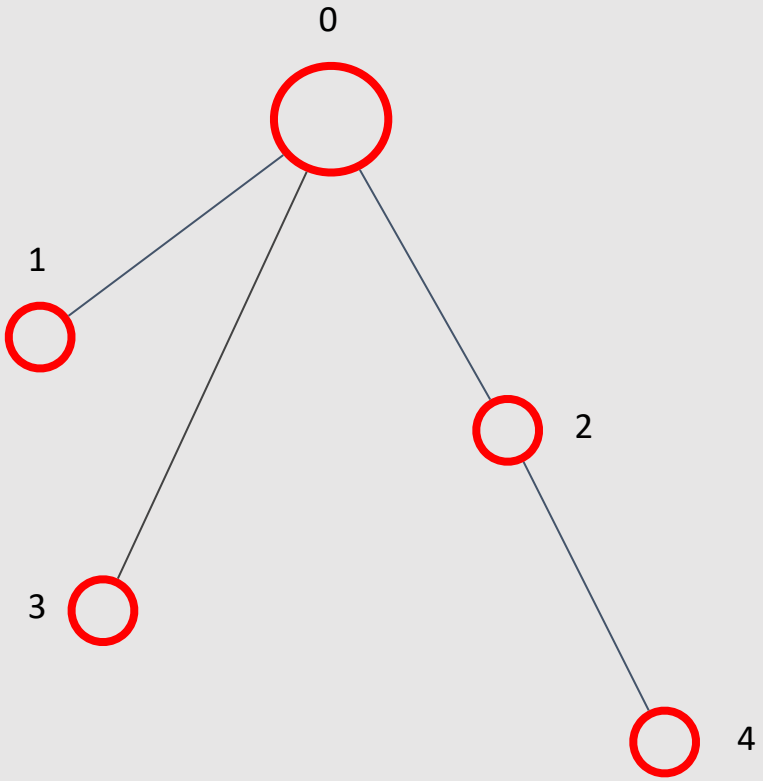
dsu = [0, 0, 2, 0, 2], query = {{3, 4}, {2, 4}}, answers = {0, 2}



dsu = [0, 0, 0, 0, 0], query = {{3, 4}, {2, 4}}, answers = {0, 2}



dsu = [0, 0, 0, 0, 0], query = {{3, 4}, {2, 4}}, answers = {0, 2}



```cpp
const int MAXN = 100005;

vector<bool> visited(MAXN);
vector<vector<int>> queries(MAXN);
vector<int> ancestor(MAXN);

int find(int v) {
    if (v == ancestor[v]) {
        return v;
    }
    return ancestor[v] = find(ancestor[v]);
}

void tarjanLCA(int v, vector<vector<int>>& adj) {
    ancestor[v] = v;

    visited[v] = true;

    for (int u : adj[v]) {
        if (!visited[u]) {
            tarjanLCA(u, adj);
            ancestor[find(u)] = v;
        }
    }

    for (int q : queries[v]) {
        if (visited[q]) {
            int lca = find(q);
            cout << "LCA of " << v << " and " << q << " is " << lca << endl;
        }
    }
}
```

```cpp
int main() {
    int n;
    cin >> n;

    vector<vector<int>> adj(n);

    for (int i = 0; i < n - 1; ++i) {
        int u, v;
        cin >> u >> v;
        u--;
        v--;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    int q;
    cin >> q;

    for (int i = 0; i < q; ++i) {
        int u, v;
        cin >> u >> v;
        u--;
        v--;
        queries[u].push_back(v);
        queries[v].push_back(u);
        cout << u << " " << v << "\n";
    }

    tarjanLCA(0, adj);

    return 0;
}
```

stdin

```
7
1 2
1 3
2 4
2 5
3 6
3 7
1
4 6
```

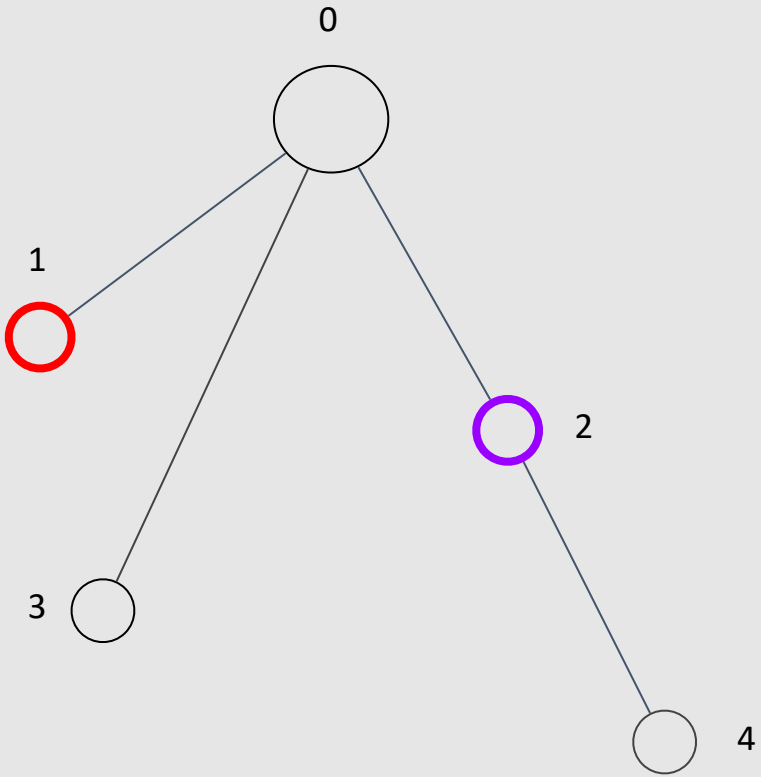
stdout

```
3 5
LCA of 5 and 3 is 0
```

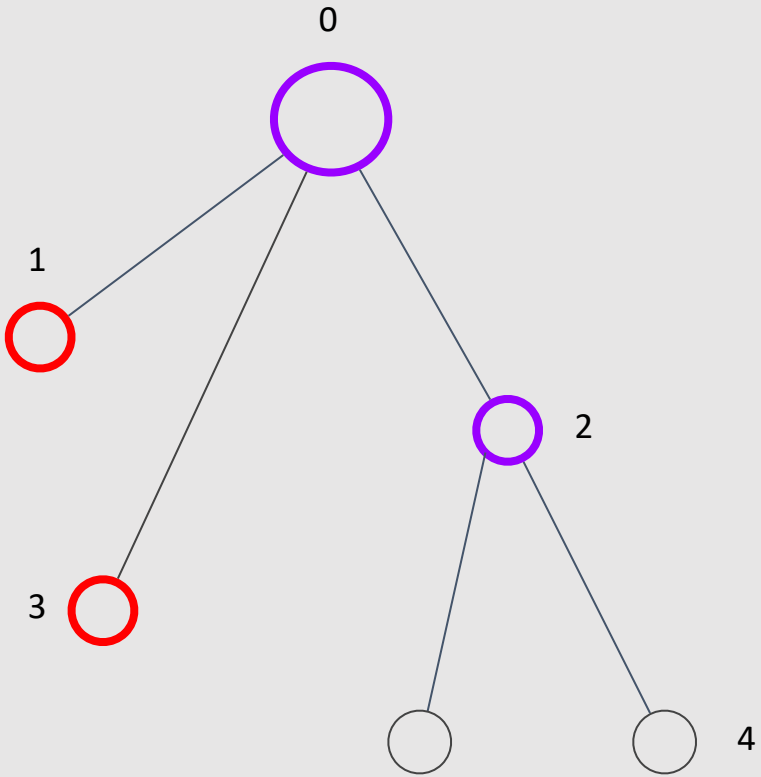
Task 1

lca with adding vertex?

lca(1, 2) = ?



add(2)



Task 1

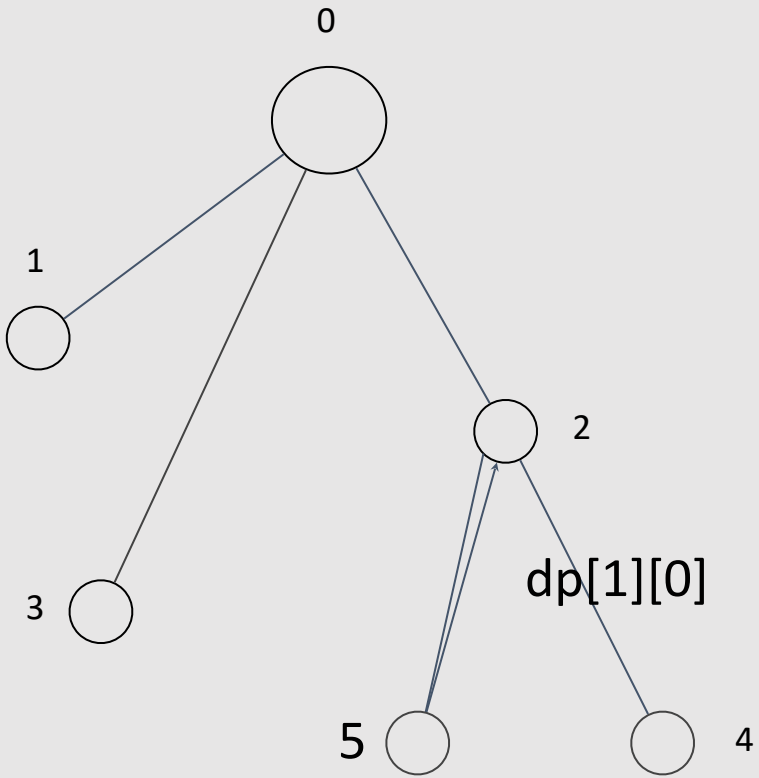
dp[i][j] - ancestor of i in distance (2^j).

vector<array> dp

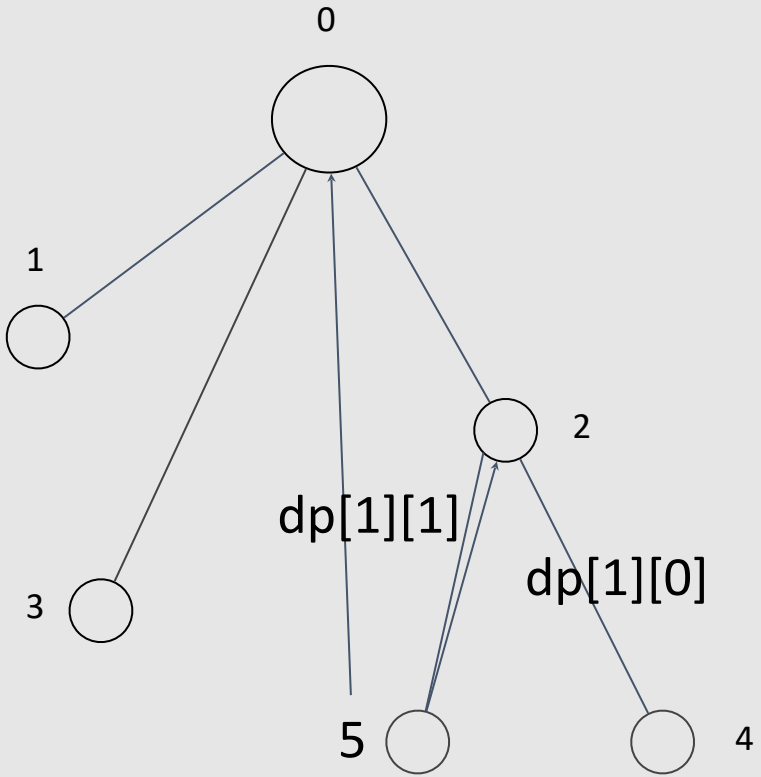
dp.push_back/append array

dp[new_vertex][0] = parent

add(2)



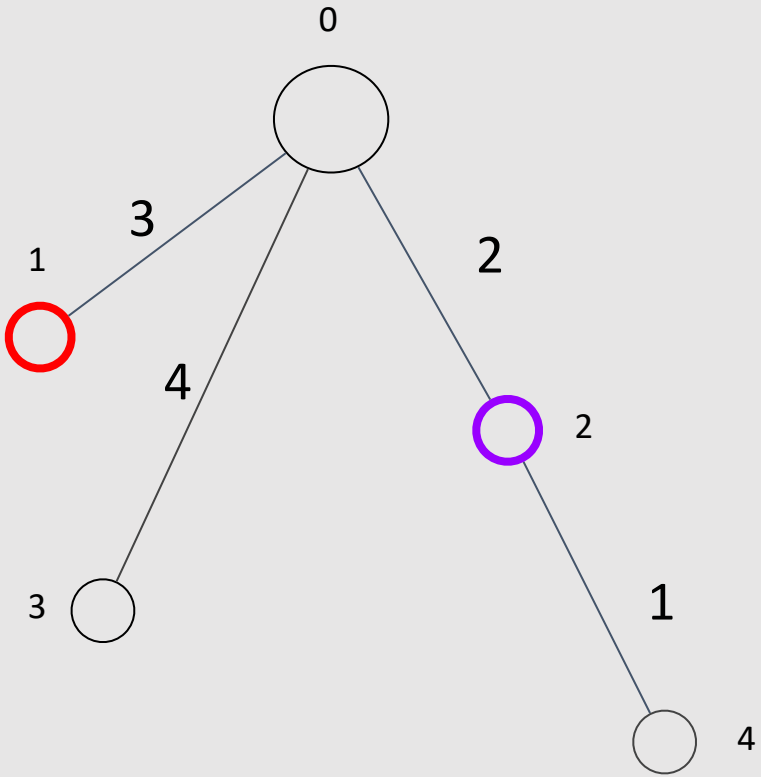
add(2)



Task 2

sum on path

sum from vertex 1 to 2 = 5

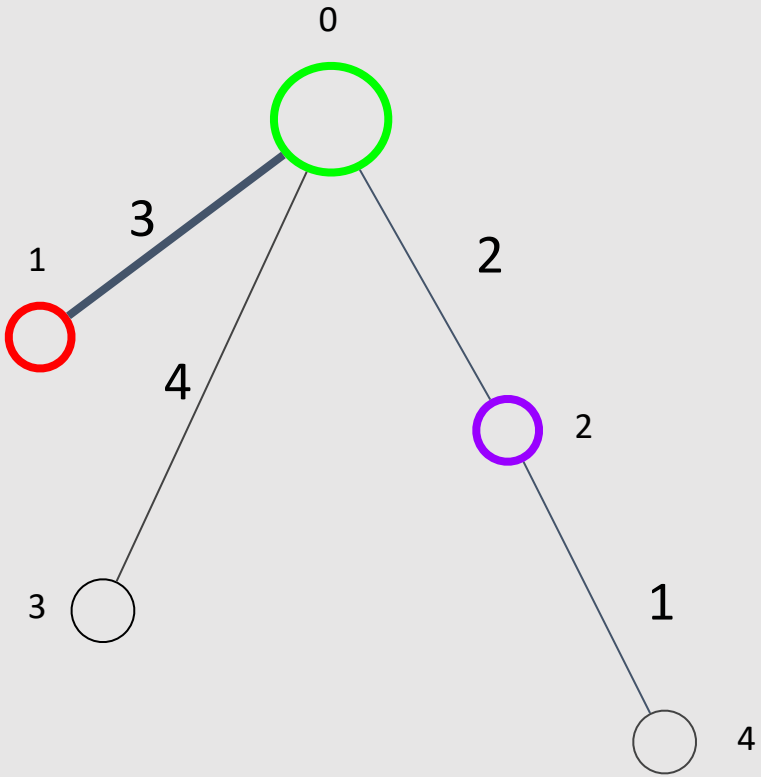


Let’s change task

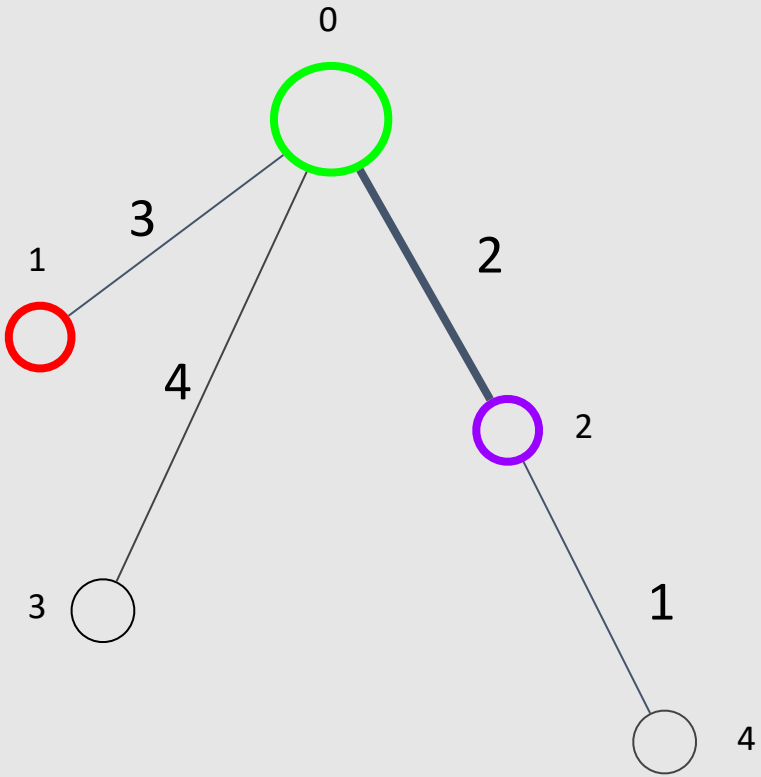
Now we dont want to calculate sum on path.

But we will calculate sum on the branch.

sum from vertex 1 to 2 = 5



sum from vertex 1 to 2 = 5



Let’s change task

sum[i][dist] - sum on the way from this vertex to the parent of this vertex by dist = 2^dist.

sum[i][dist + 1] = sum[i][dist] + sum[dp[i][dist]][dist];

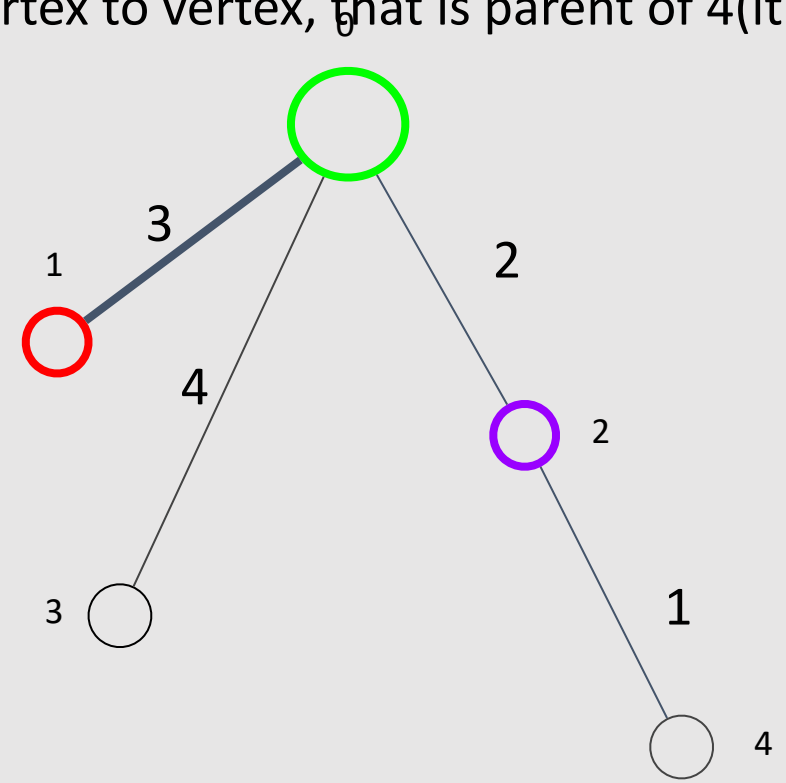
sum[4][0] = 1, sum[2][0] = 2

sum[4][1] = sum[2][0] + sum[4][0] = 3

sum[4][0] = sum from 4 vertex to vertex, that is parent of 4(it's two)

sum from 2 to 4 = 1

sum[3][0] = 4



What we can keep

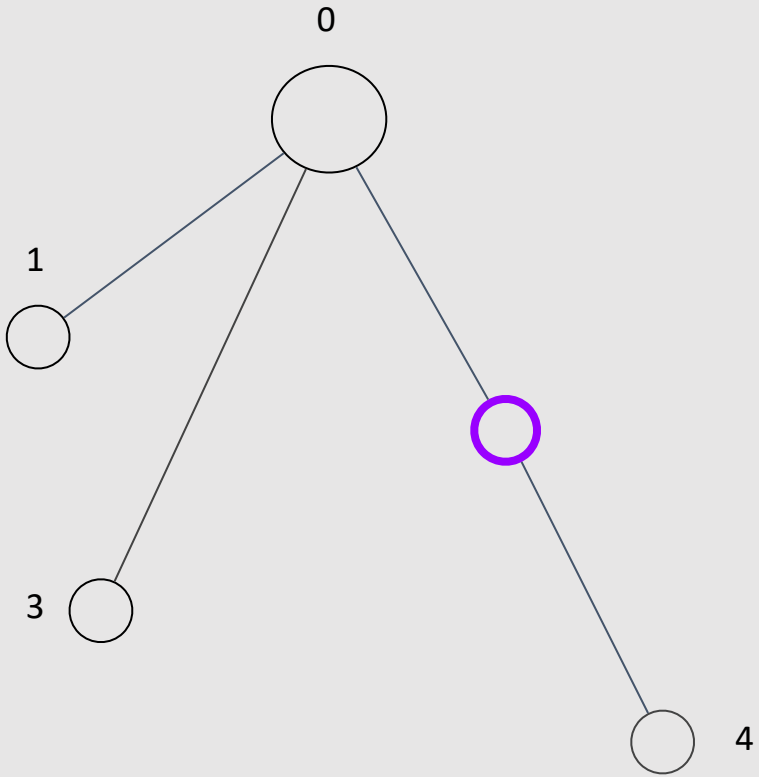
We can keep all such functions, that f(a,b) = f(a, mid) o f(mid, b)

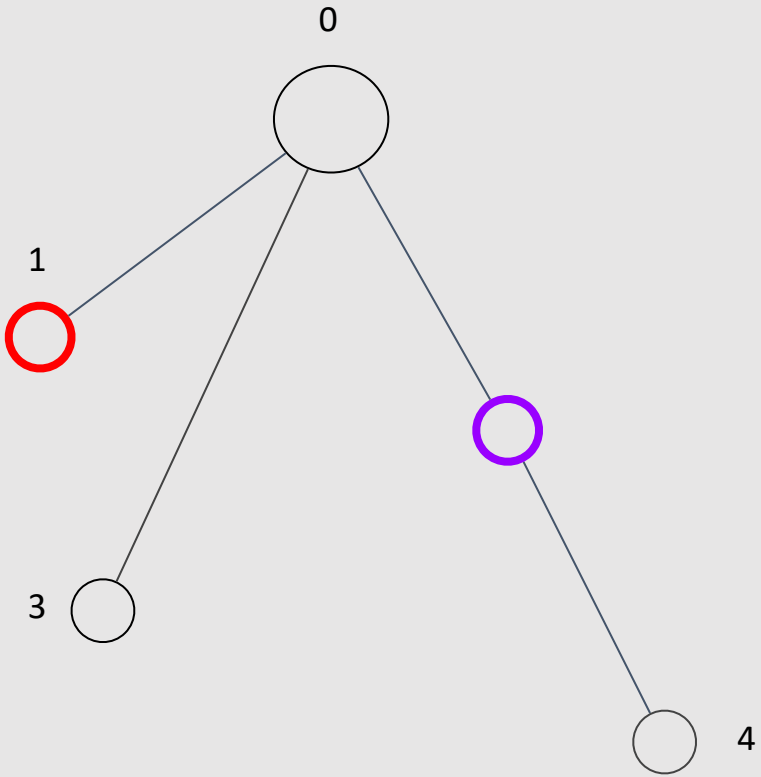
Task 3

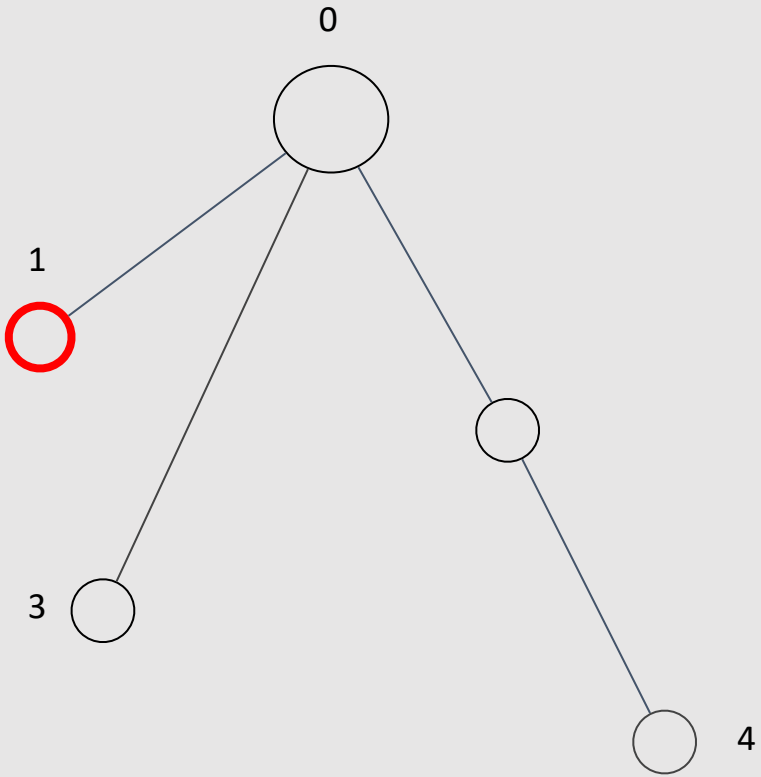
You need to find longest path(diameter) with adding vertices.

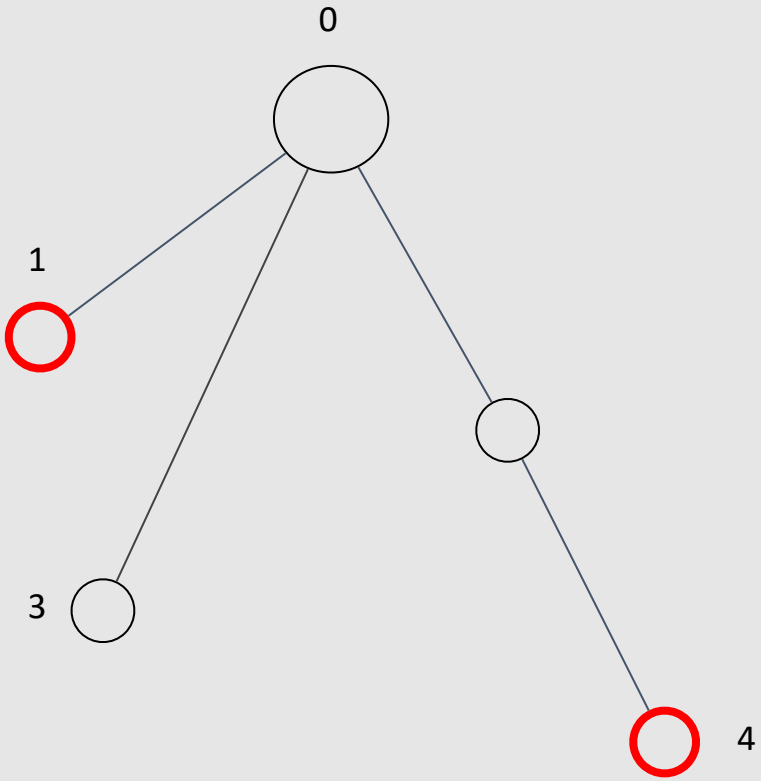
? find longest path(diameter) in tree

add as son of a.


0
1
3
4


0
1
3
4

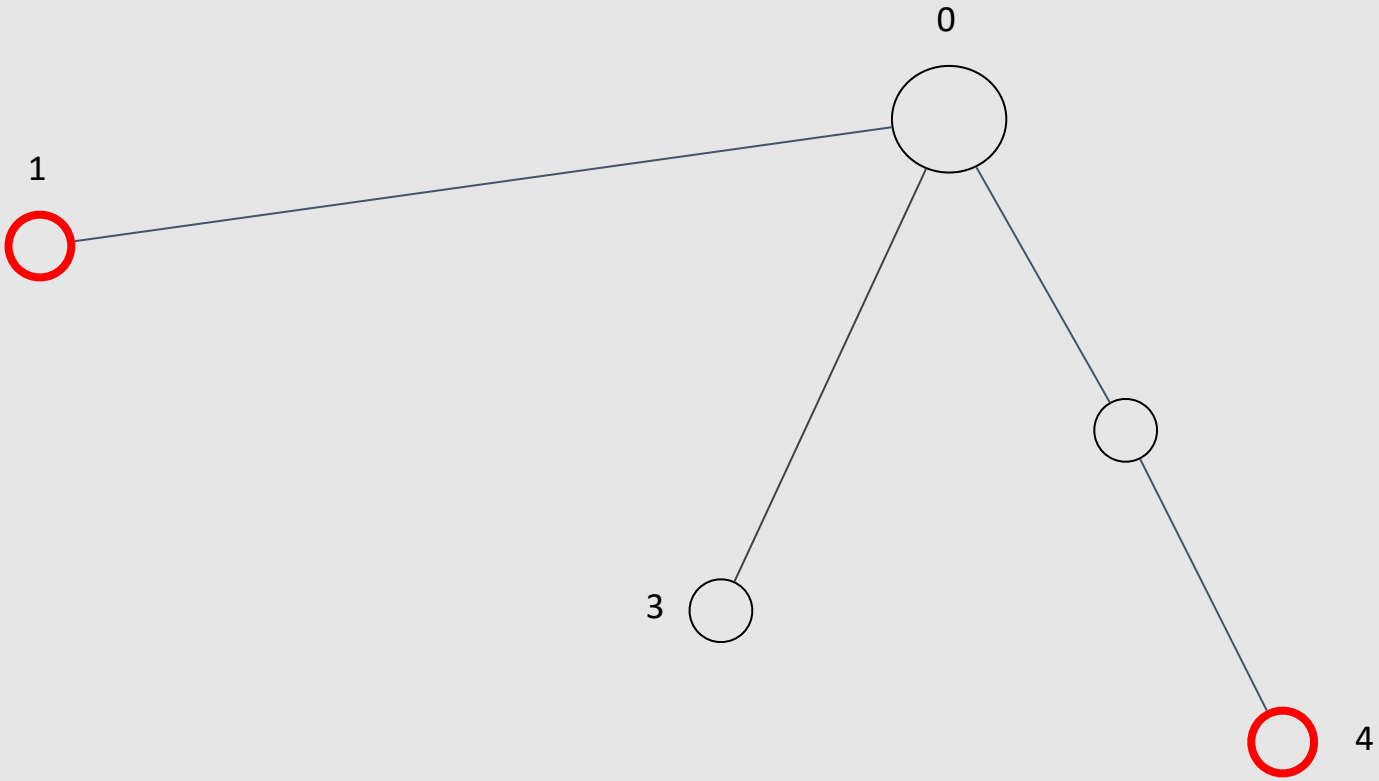

0
1
3
4


0
1
3
4

Task 3

2 diff situation

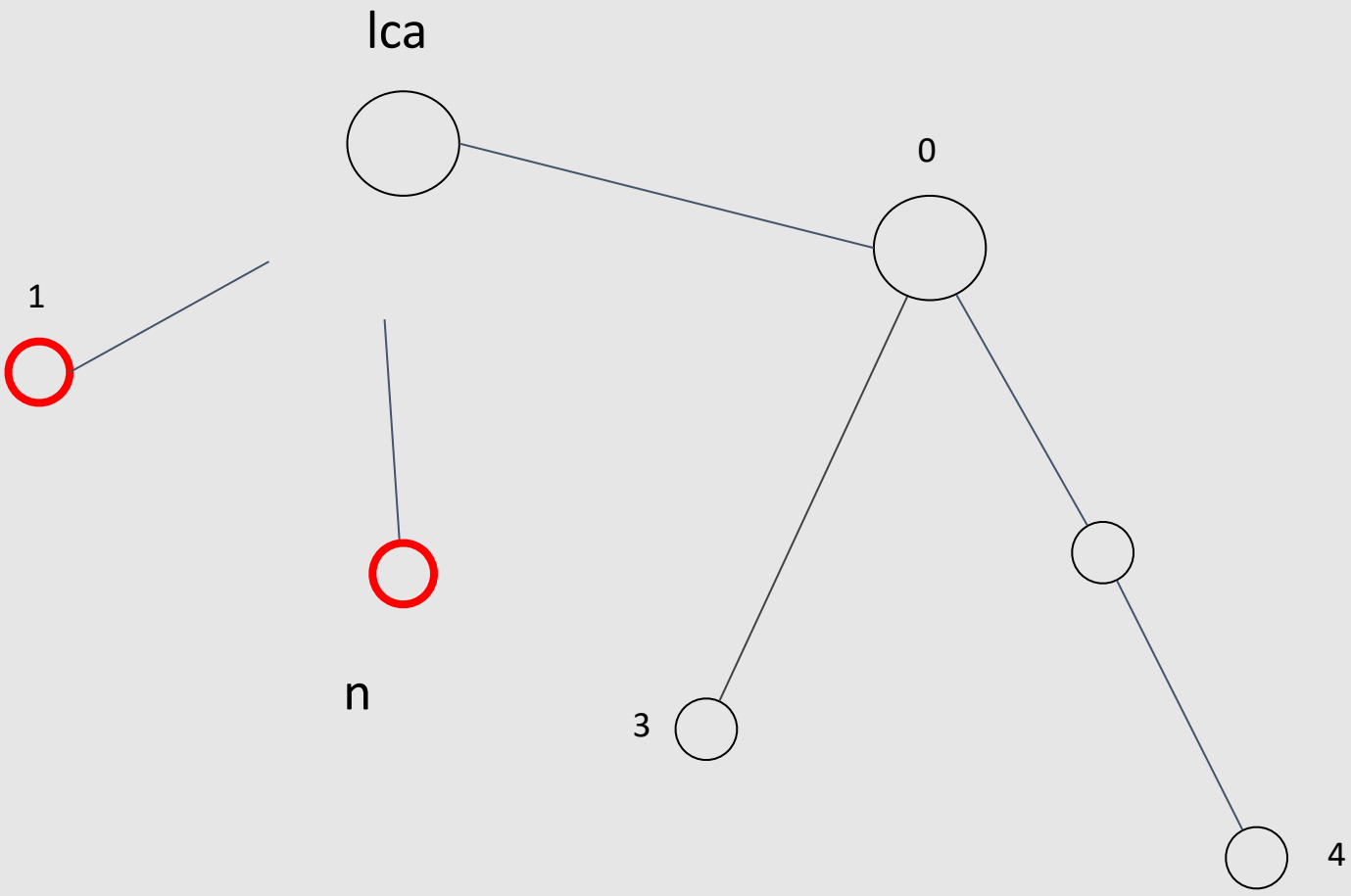
one - diameter cross the root


0
1
3
4

Task 3

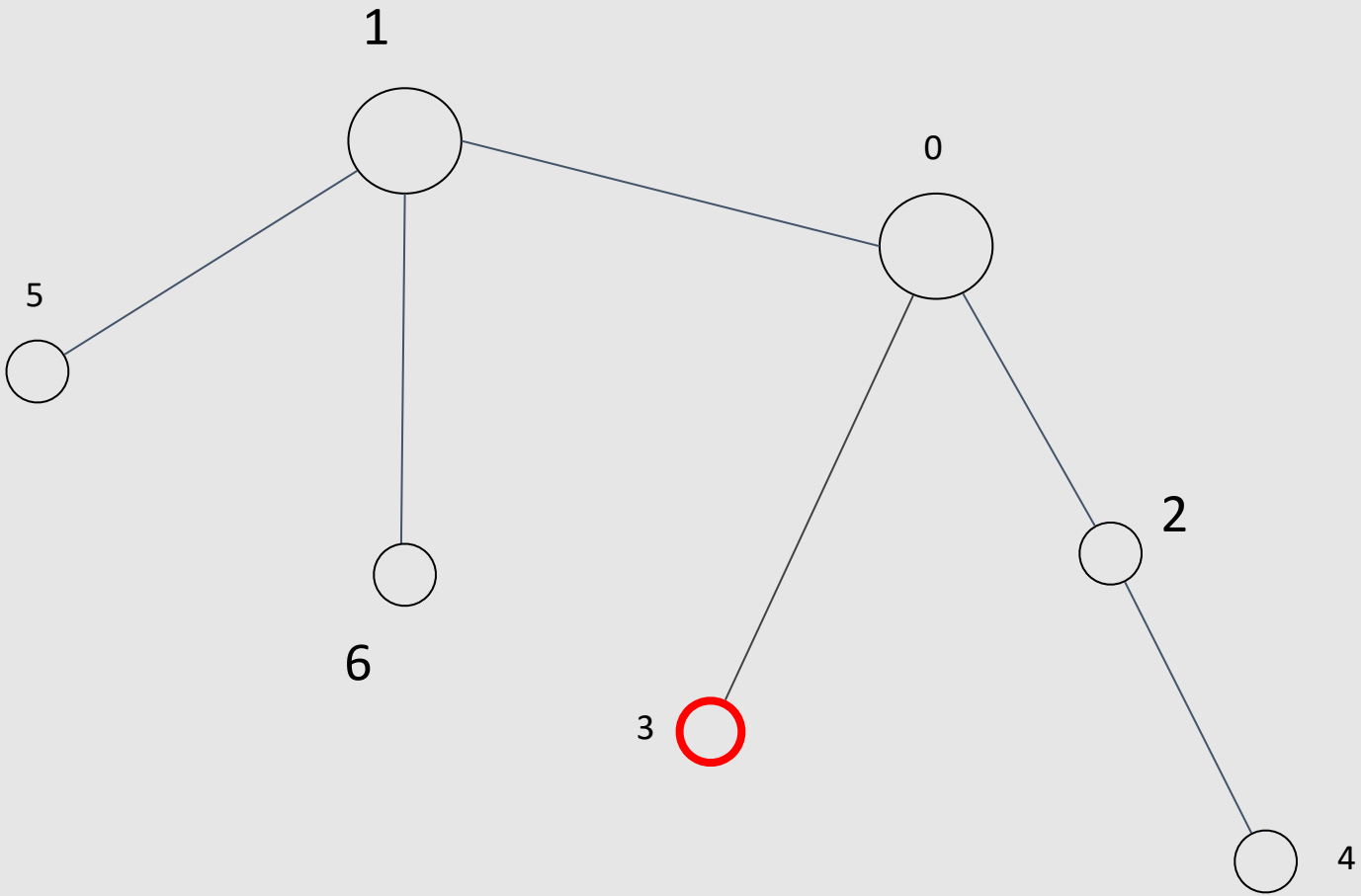
2 diff situation

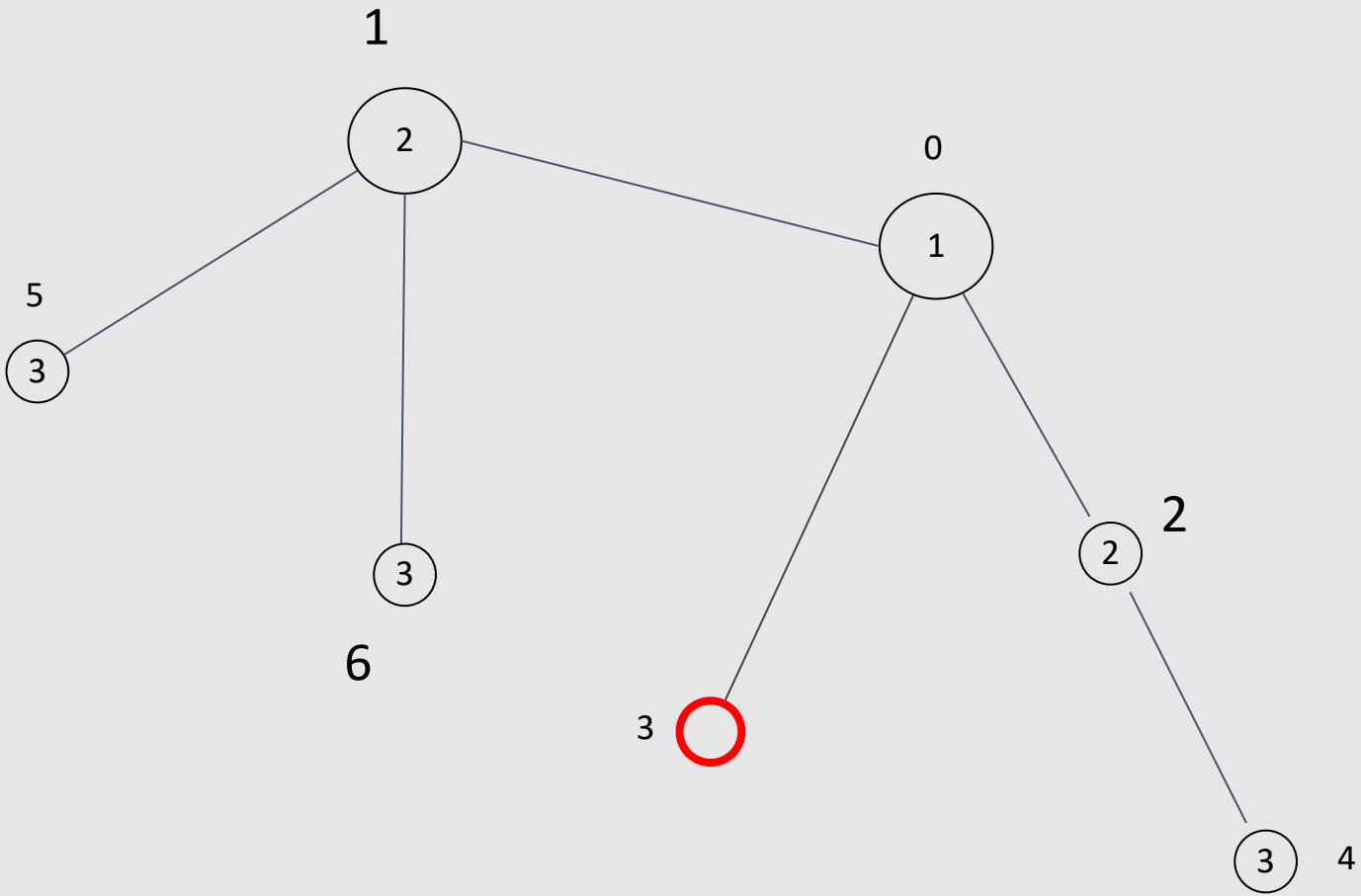
second - diameter doesn’t cross the root

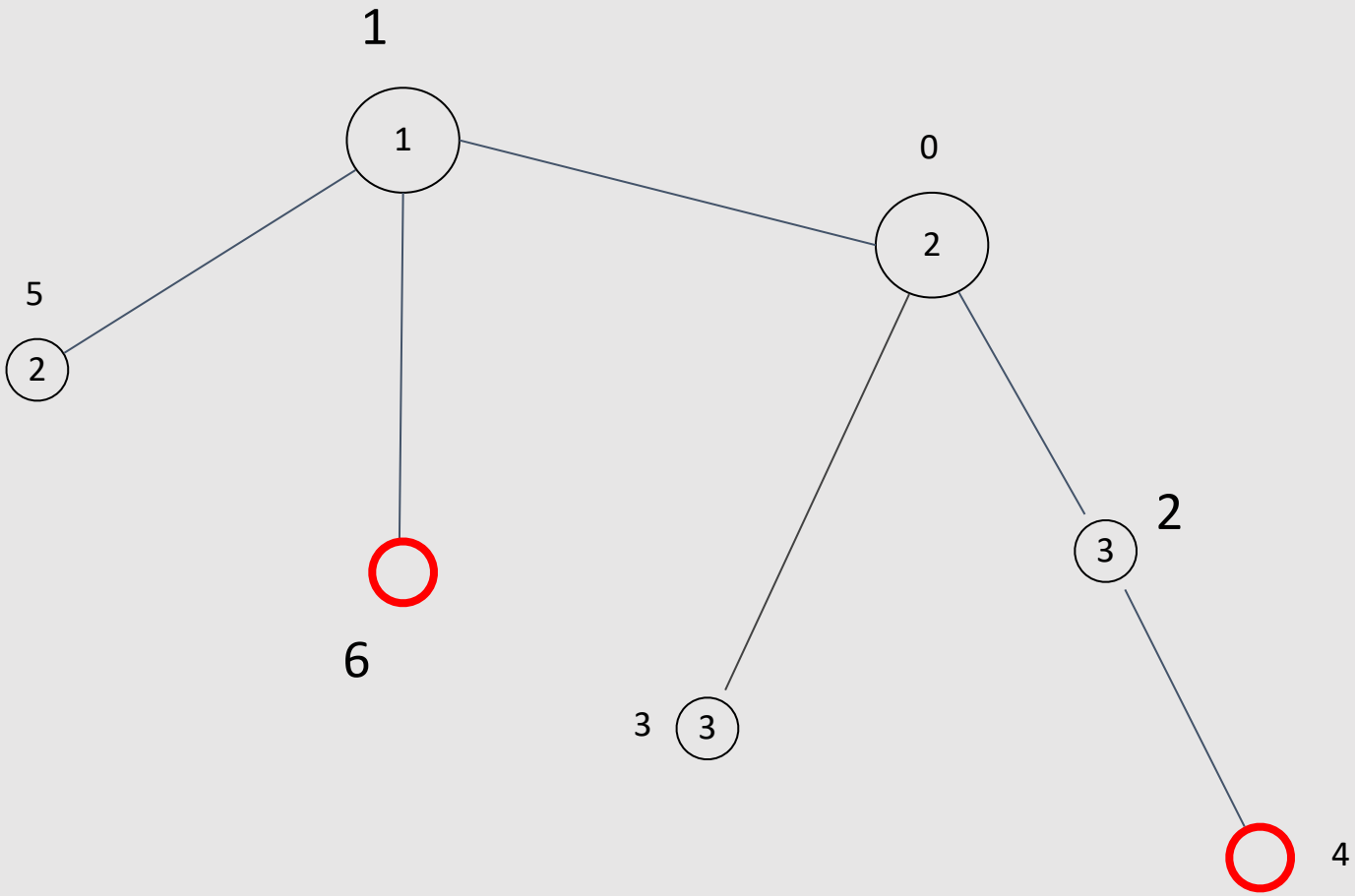

lca
0
1
n
3
4

Task 3

Example


1
0
5
2
6
3
4


1
2
0
1
5
3
3
6
2
2
3
3
4


1
1
0
2
5
2
2
3
6
3
3
4

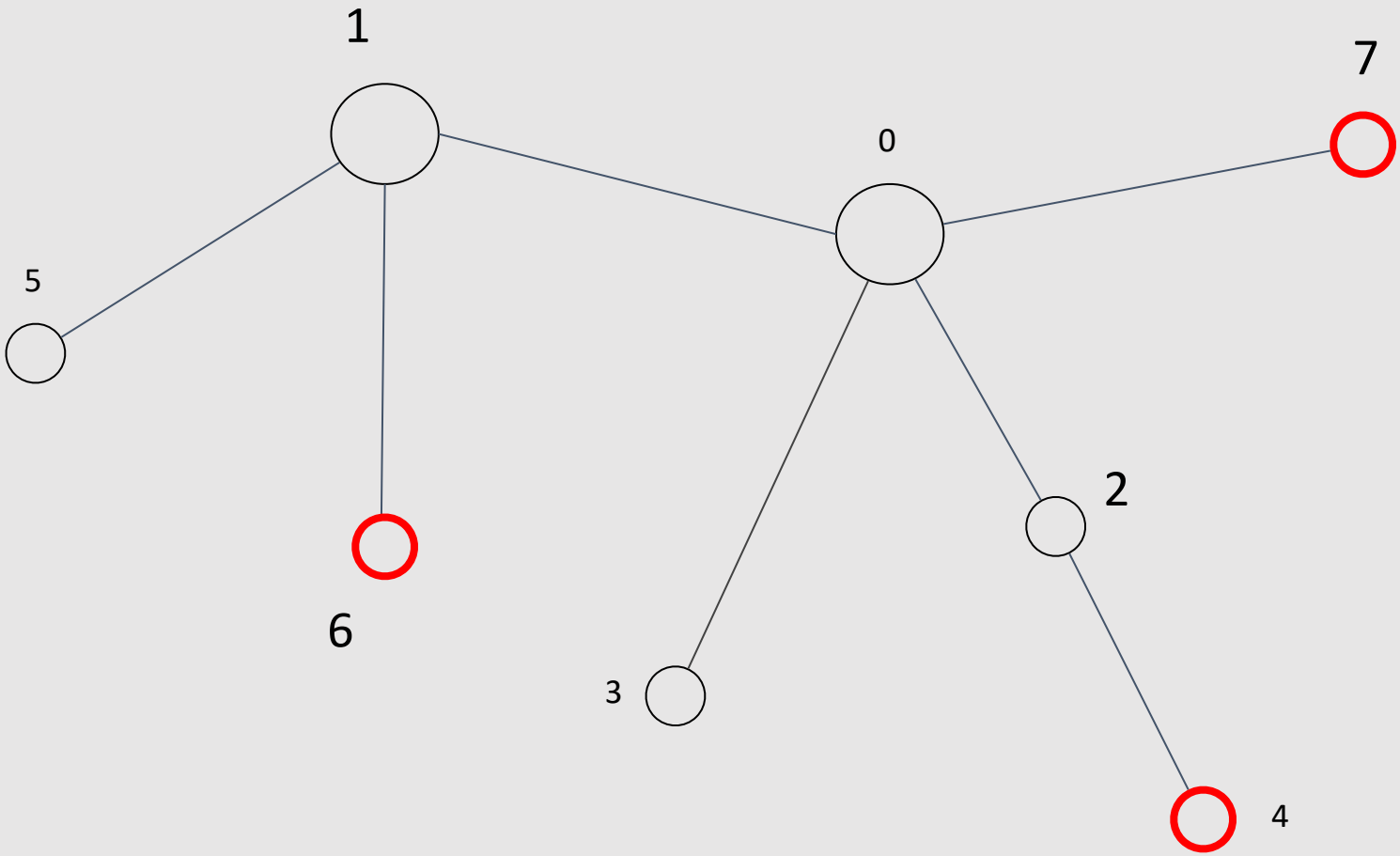
Task 3

change task

Every time we keep current diameter - (a, b)

add as son of a - add and recalculate binary up(task1)

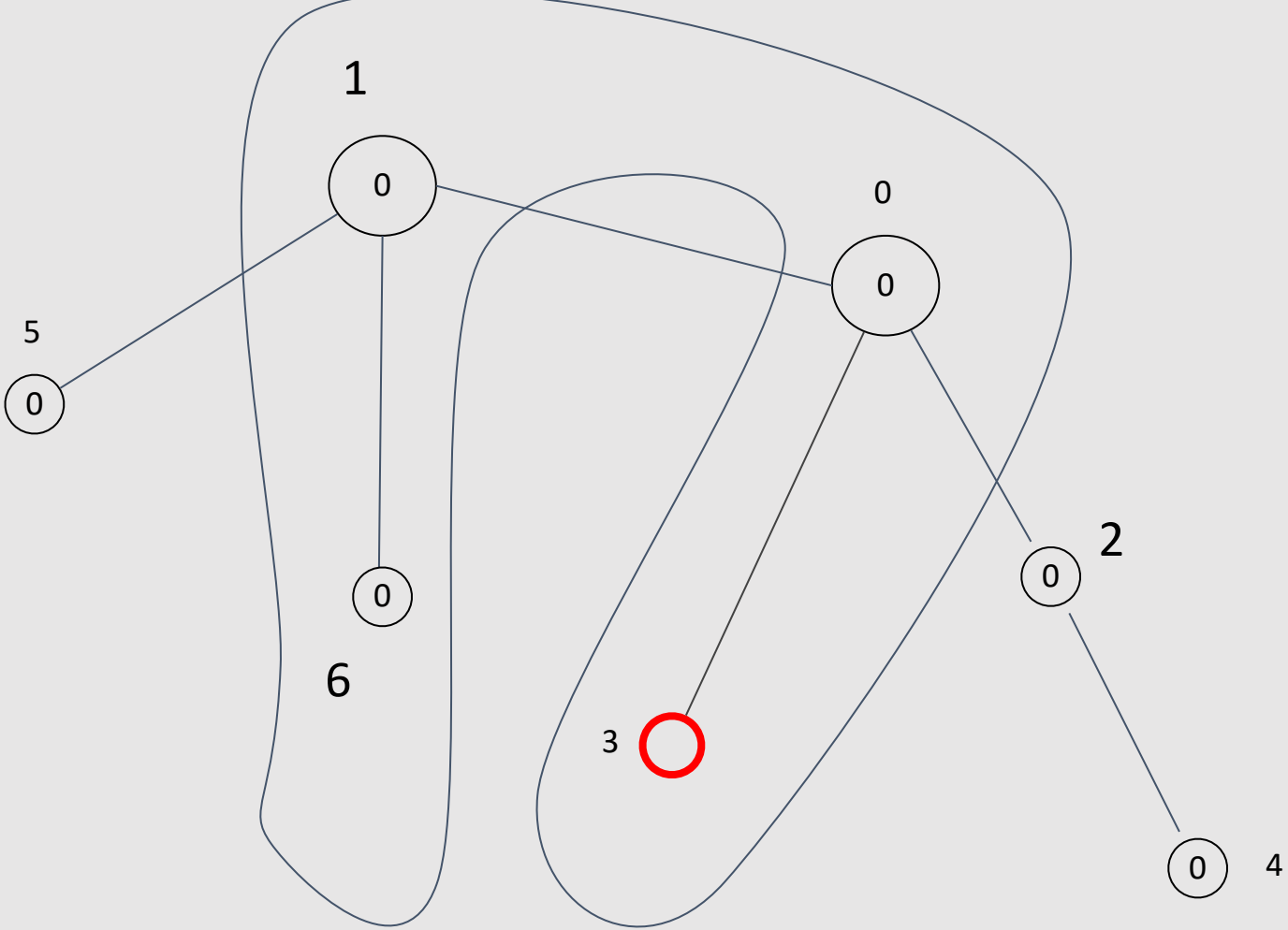
when we want to find longest path - we have three options (a, b), (a, new_vertex), (new_vertex, b)

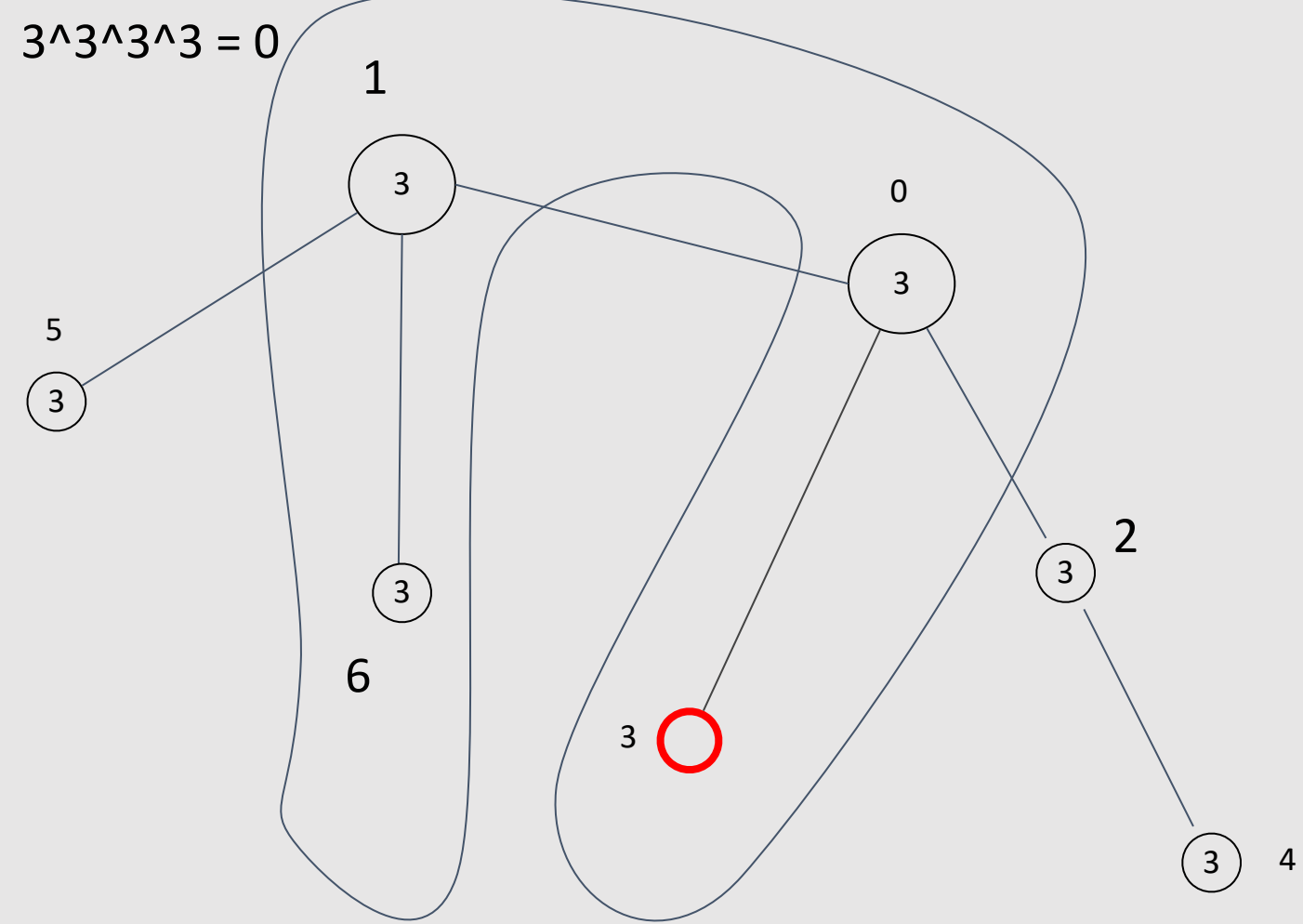

1
7
0
5
2
6
3
4

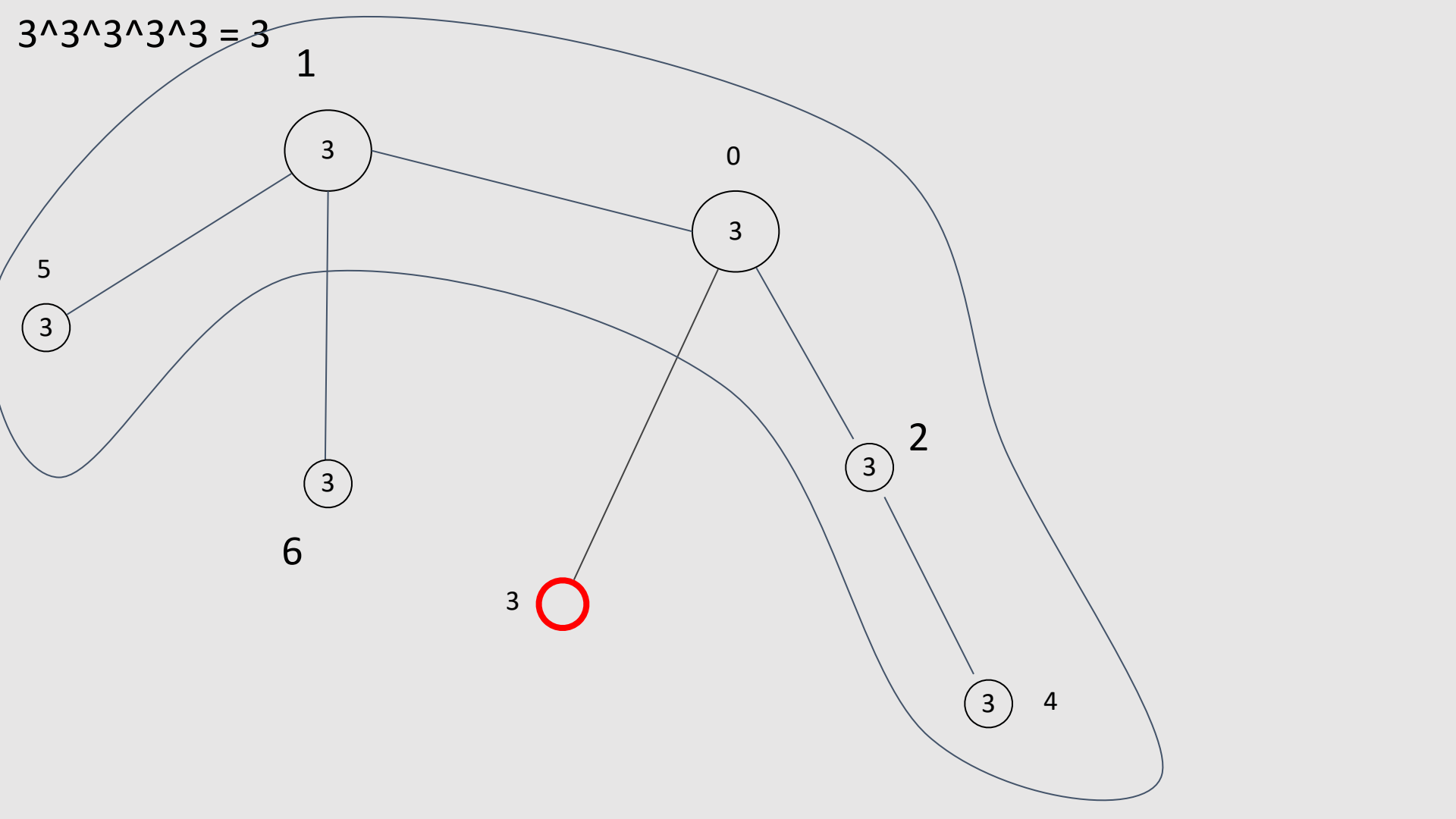
Task 4

xor on path ?

xor = in tree


1
0
0
0
5
0
0
2
0
6
3
0
4


3^3^3^3 = 0
1
3
0
3
5
3
3
2
3
6
3
3
4


3^3^3^3^3 = 3
1
3
0
3
5
3
3
2
3
6
3
3
4

Task 4

current xor is the result of all (xor=)

the answer can be either current_xor or 0.

if number of vertices in path is even, the answer is 0

either way it’s current_xor

Task 4

number of vertices between a and b= number of vertices between a and lca and between b and lca

number of vertices between a and lca = depth[lca] - depth[a]

Task 4

(depth[lca] - depth[a] + depth[lca] - depth[b]) % 2 = (2

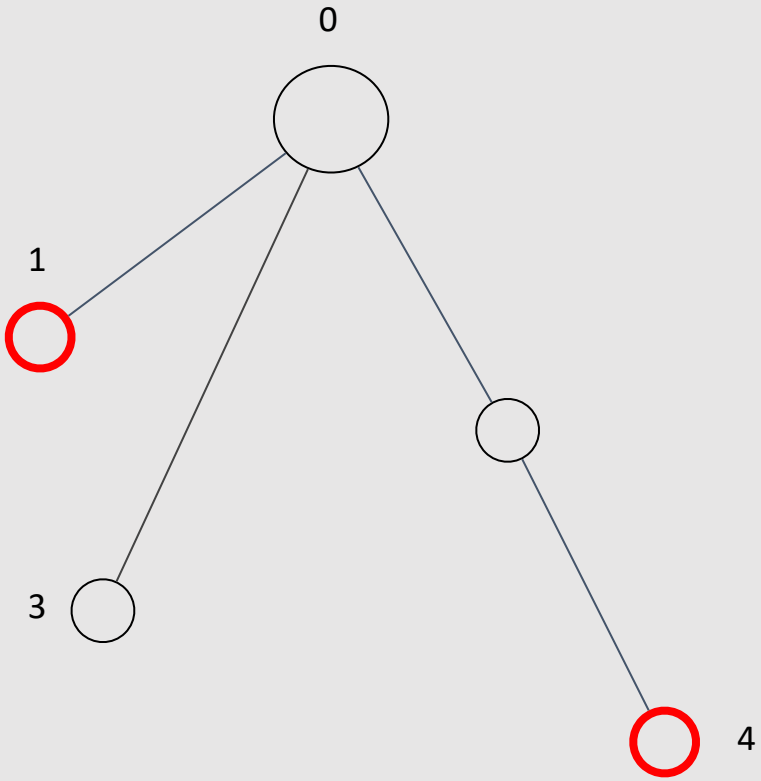
depth[lca] - depth[a] - depth[b]) % 2 = (depth[a] - depth[b]) % 2

Task 5

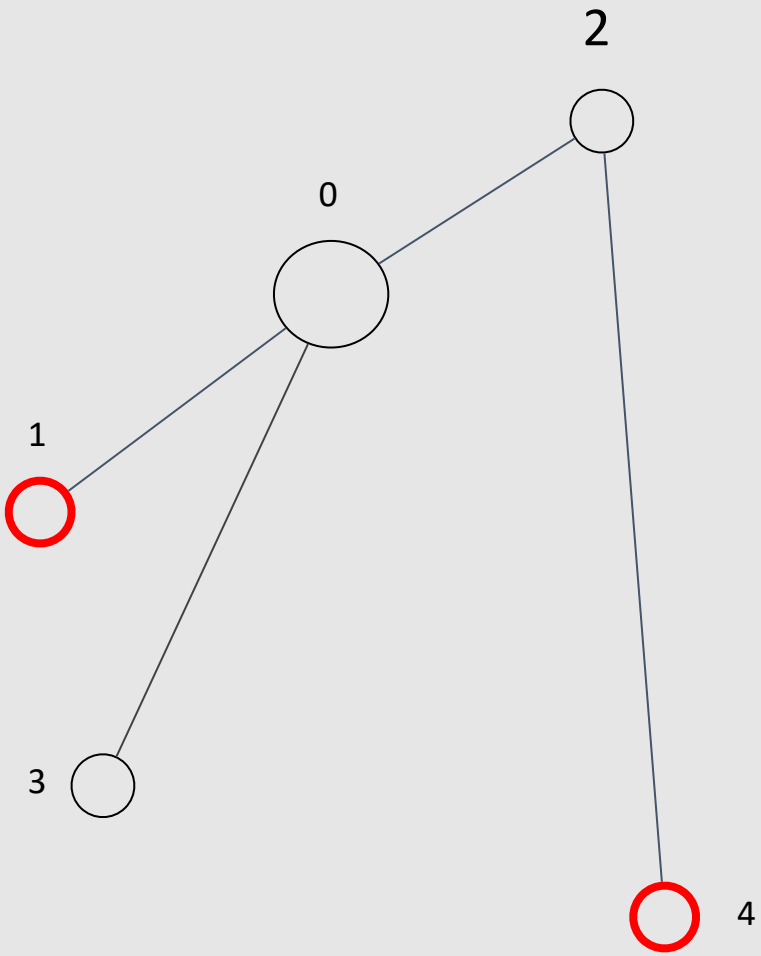
Lca(a, b)

root = x

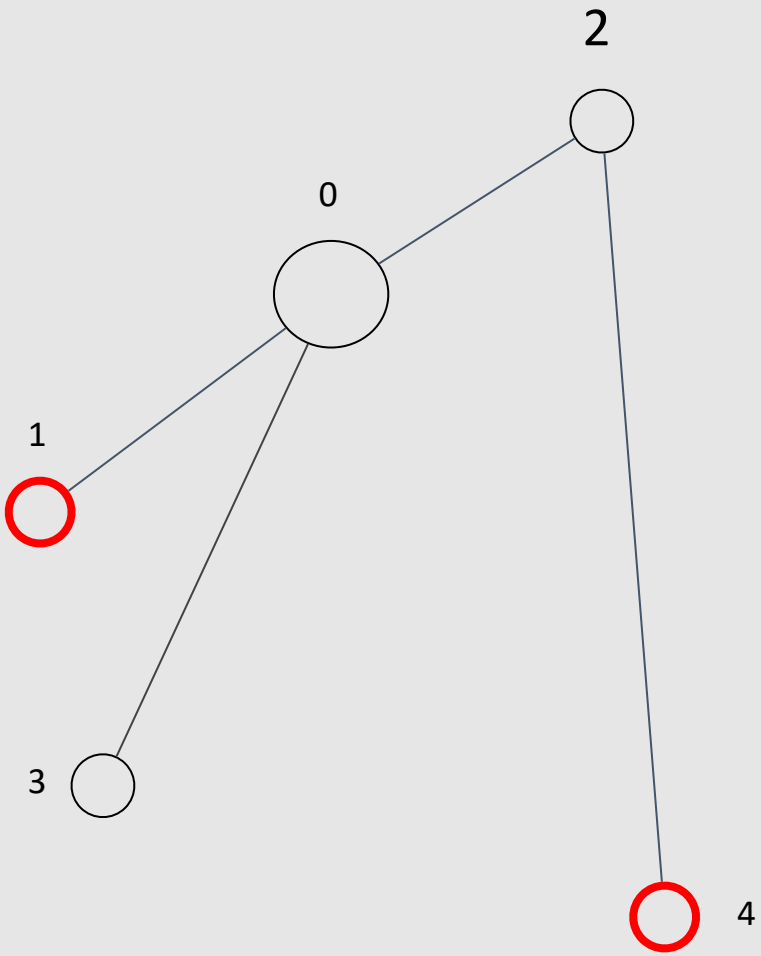
LCA(1, 4) = 0



root = 2



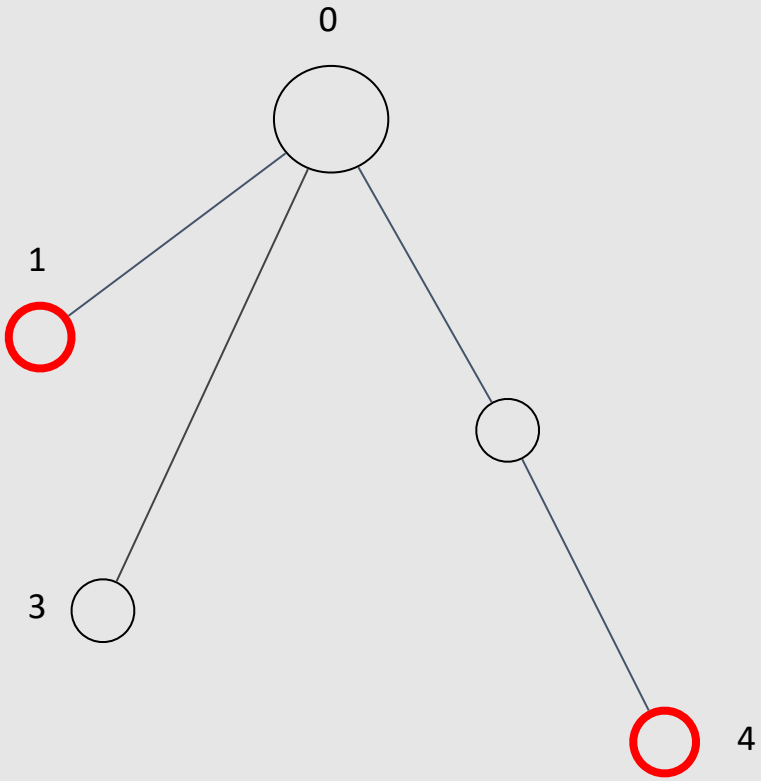
LCA(1, 4) = 2

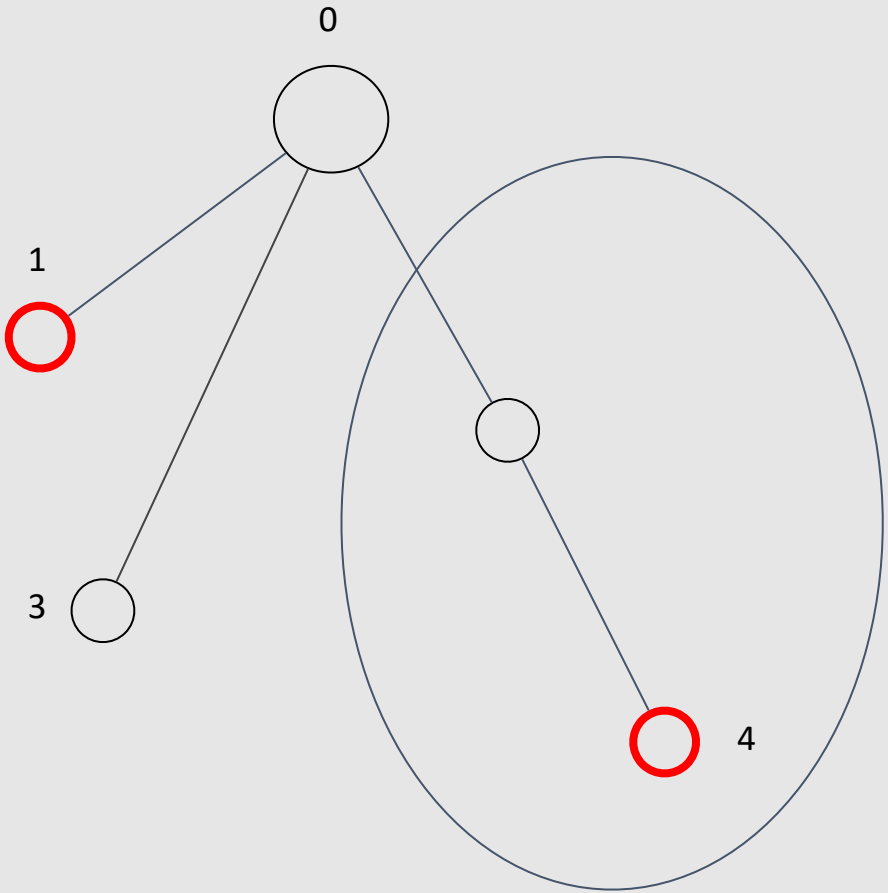


Task 6

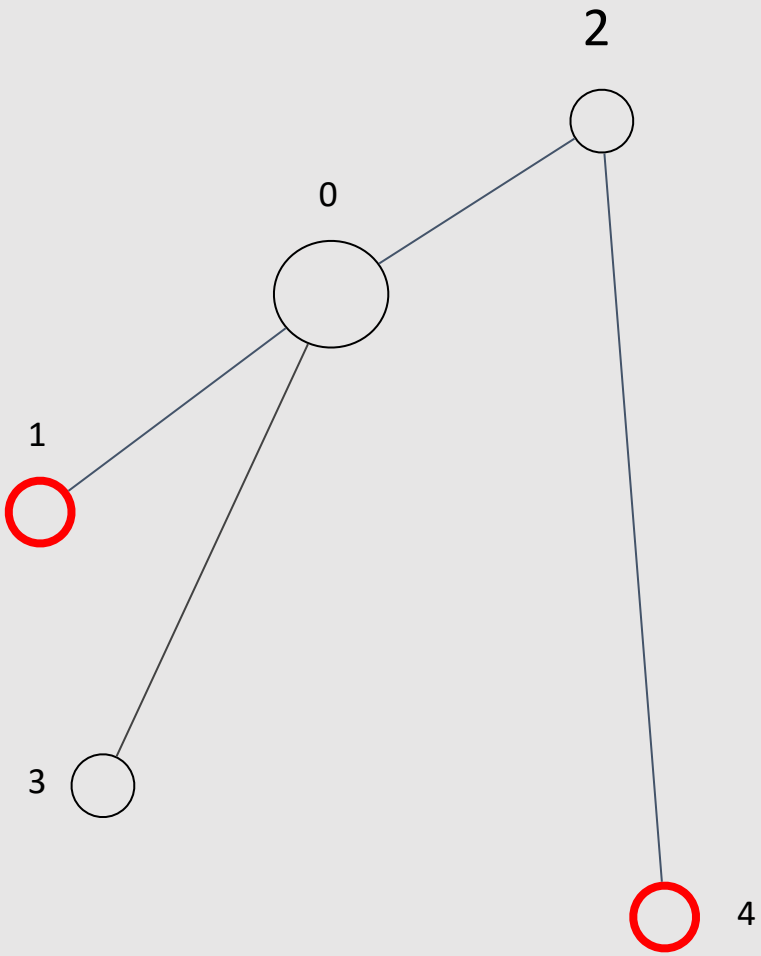
Disjoint sparse table

LCA(1, 4) = 0




0
1
3
4

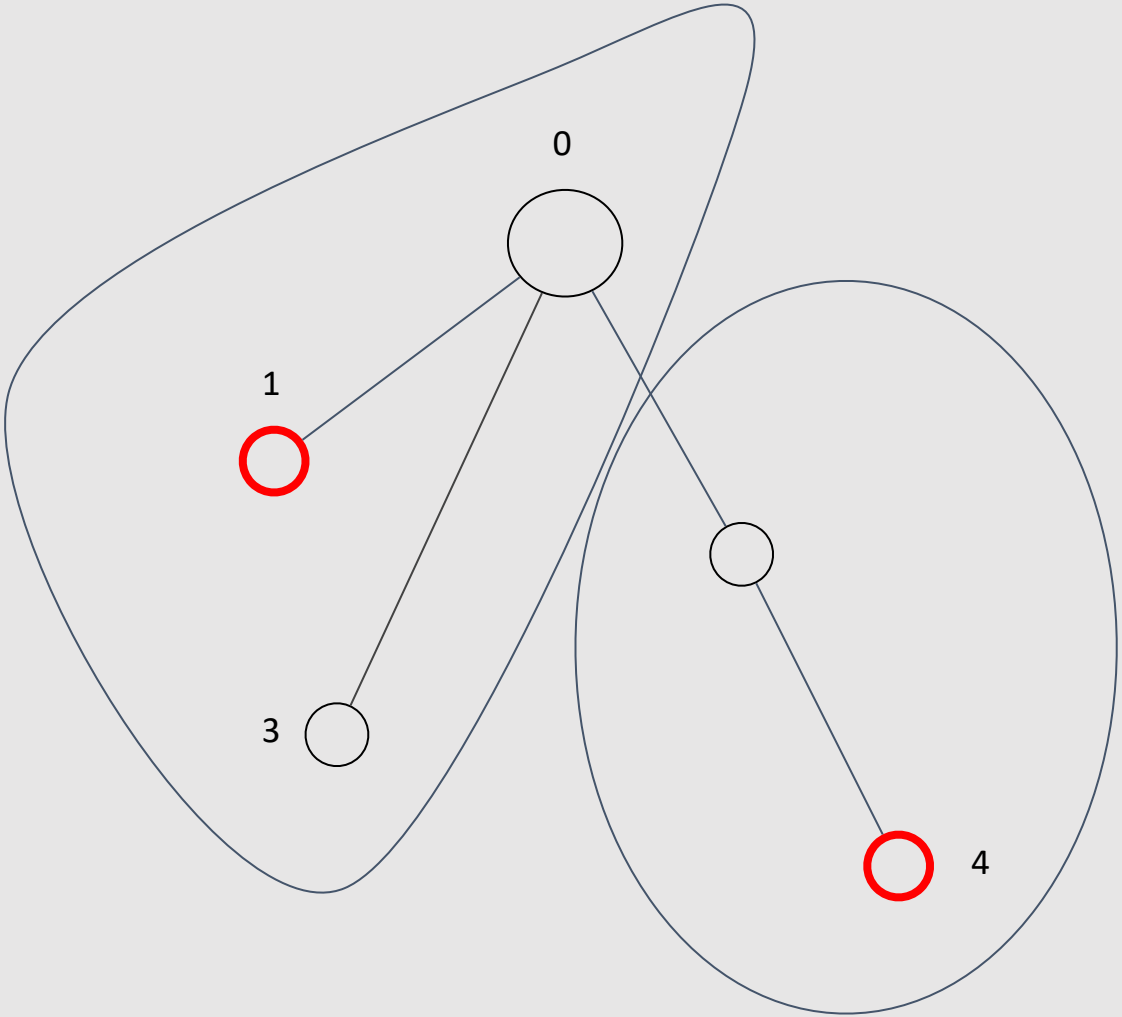
LCA(1, 4) = 2

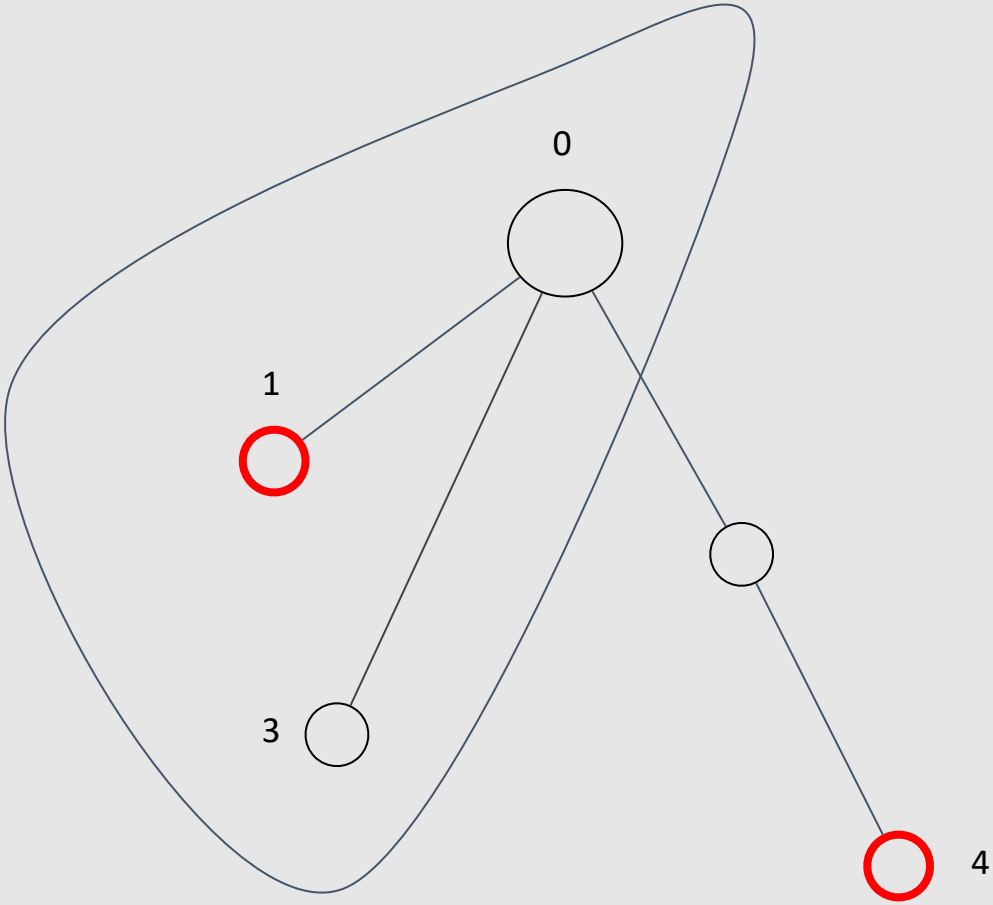


Task 6

Let’s check two cases

first one: the vertex in the subtree of root, that doesn’t contain new_root


0
1
3
4


0
1
3
4

Task 6

Let’s check two cases

second one: the vertex in the subtree of root, that contain new_root

lca(a, b) ^ lca(a, root) ^ lca(b, root)

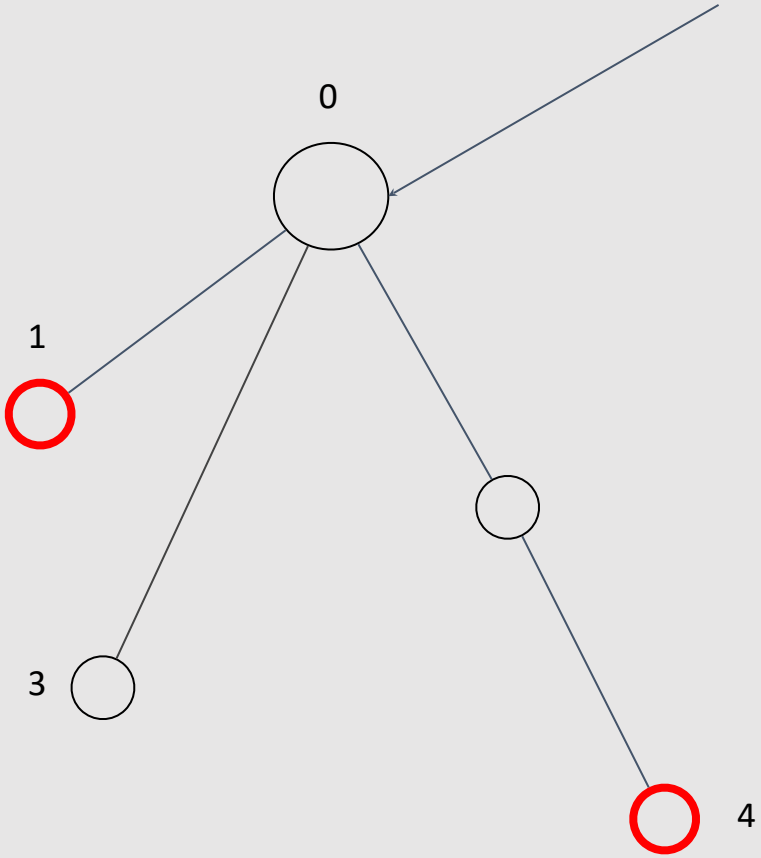
a ^ a = 0

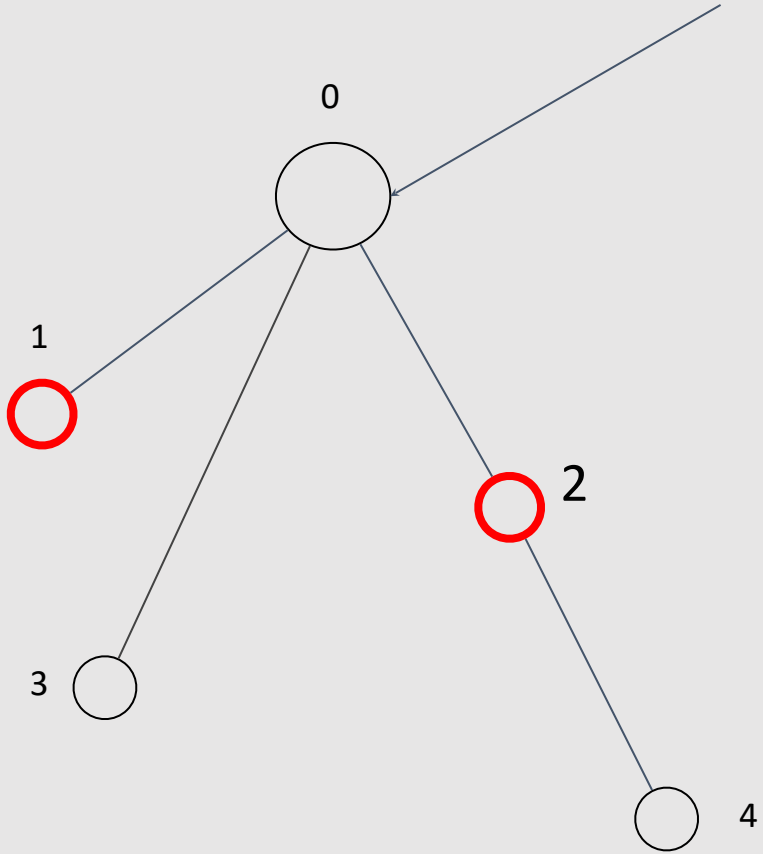
Task 6

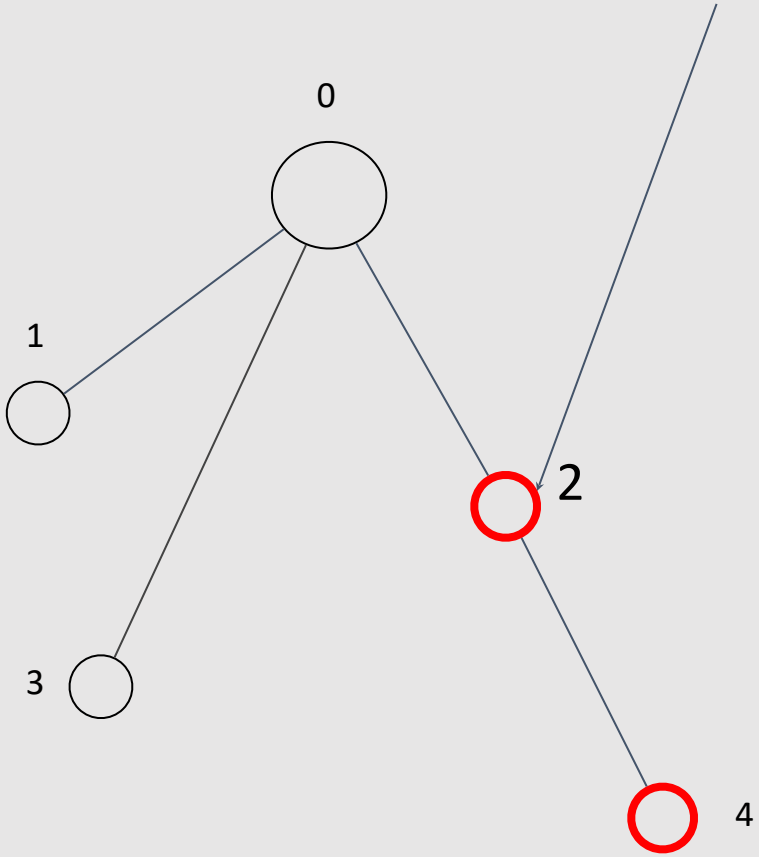
if we have the query change root,

root = root

when we need to find an answer = lca(a, b) ^ lca(a, root) ^ lca(root, b)


0
1
3
4

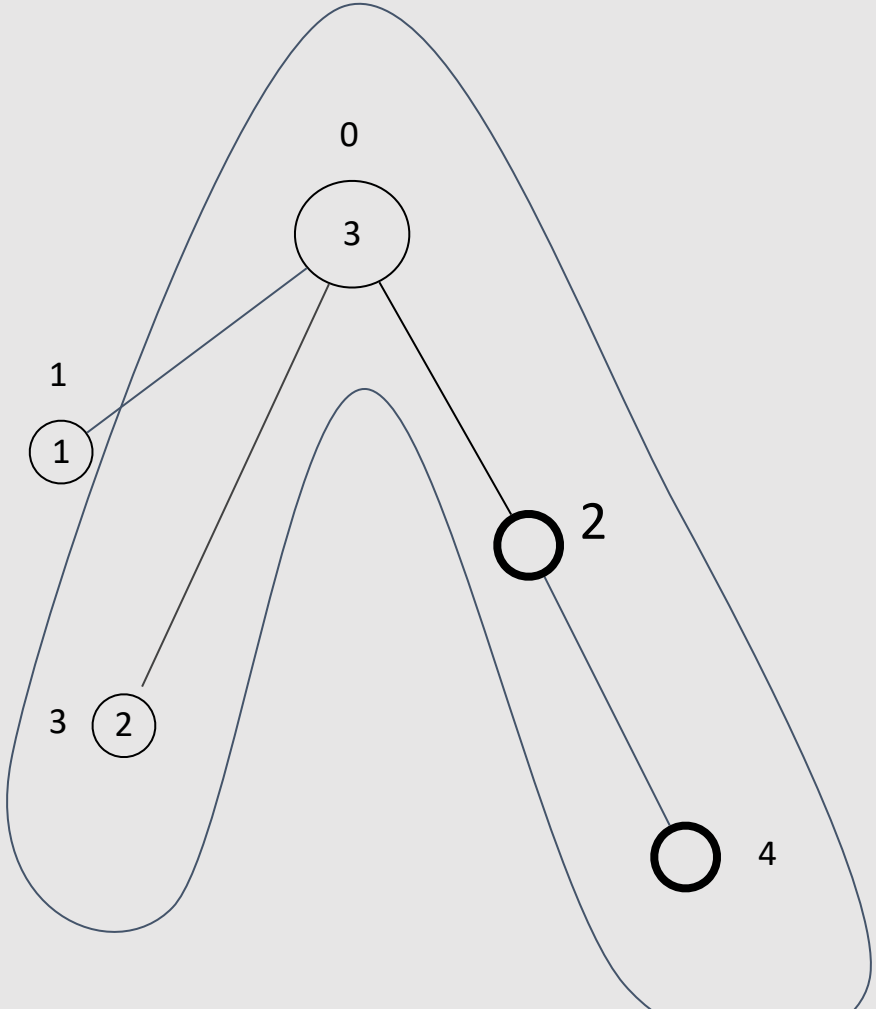

0
1
2
3
4

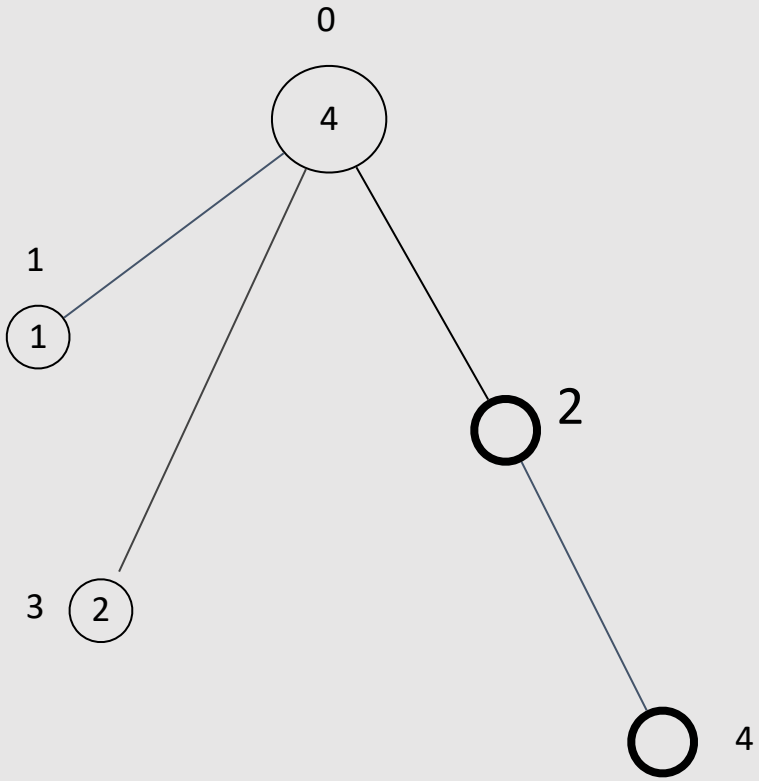

0
1
2
3
4

Task 7

sum(a, b)

ar[a] = c


0
3
1
1
2
3
2
4


0
4
1
1
2
3
2
4

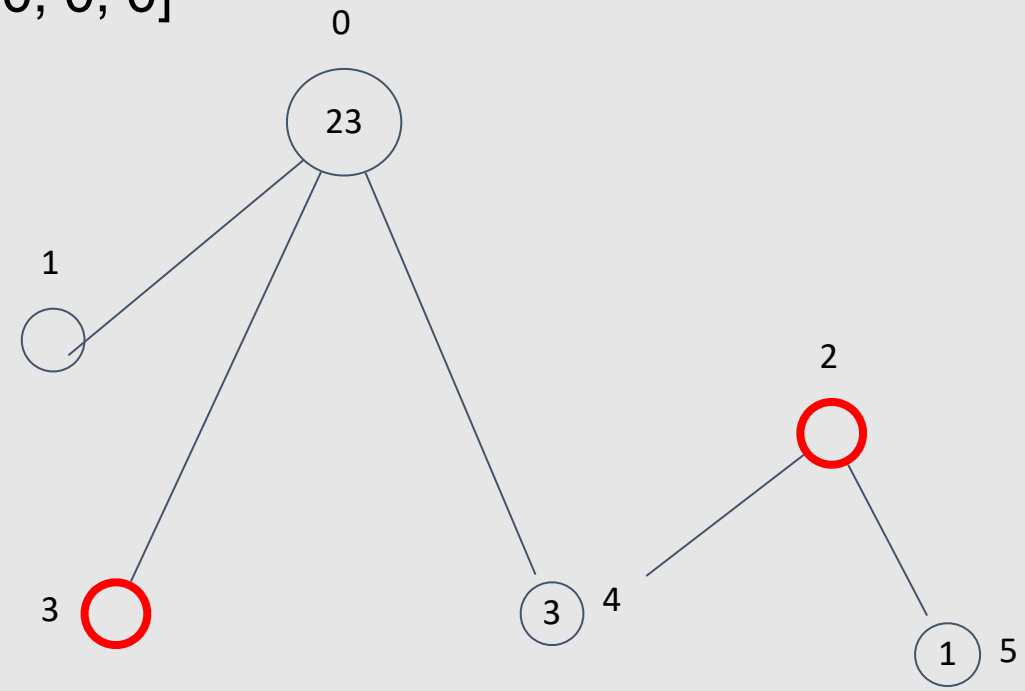
Task 7

two parts of task:

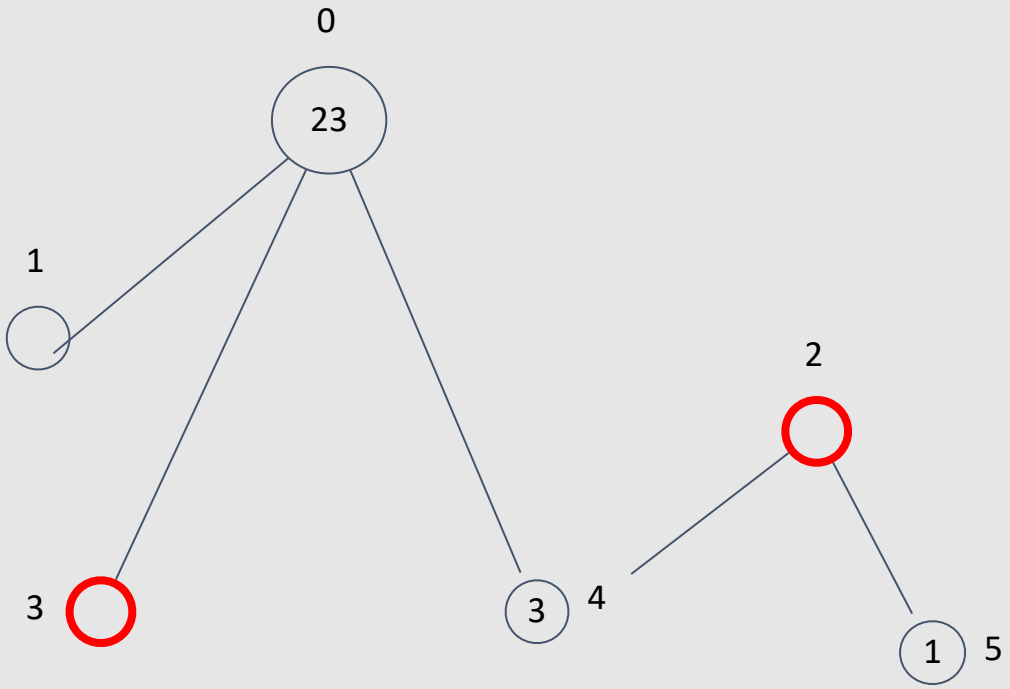
1) we answer to sum on path without addings with binary up
2) We answer to all addings at one once using segment tree.

euler = [0, 1, 1, 3, 3, 4, 2, 5, 5, 2, 5, 4, 0]

a = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

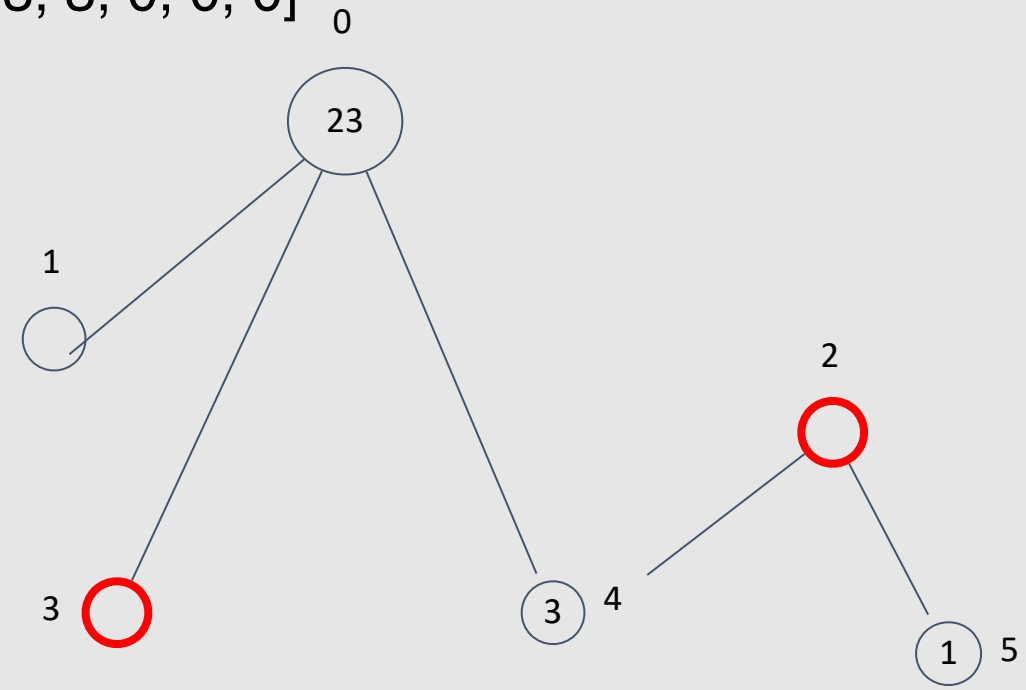


a[2] += 3

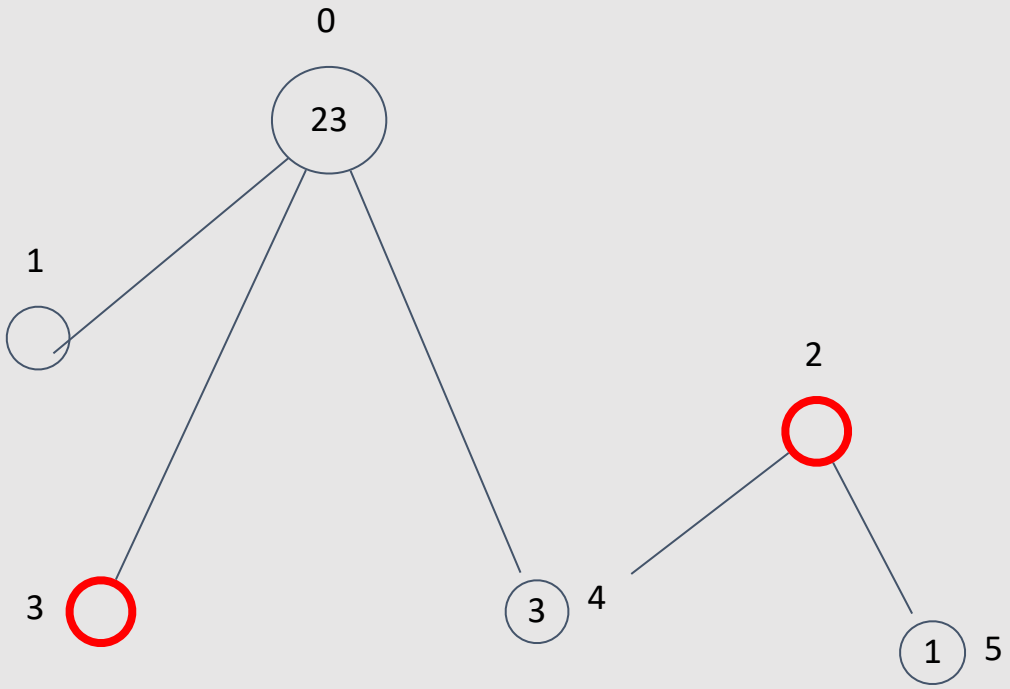


euler = [0, 1, 1, 3, 3, 4, 2, 5, 5, 2, 5, 4, 0]

a = [0, 0, 0, 0, 0, 0, 3, 3, 3, 3, 0, 0, 0]

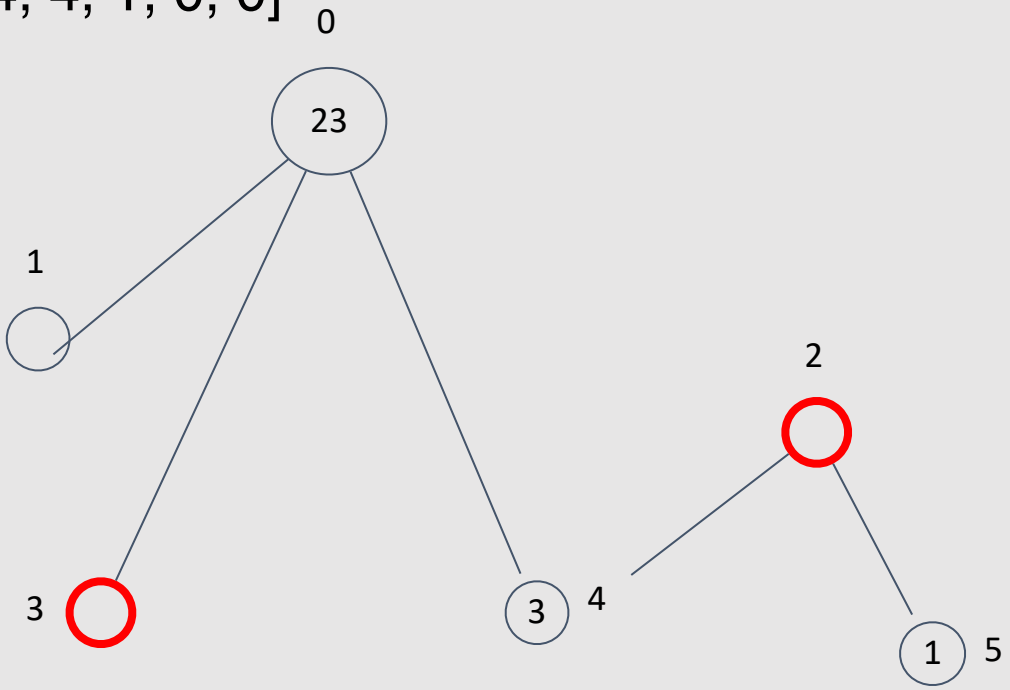


a[4] += 1

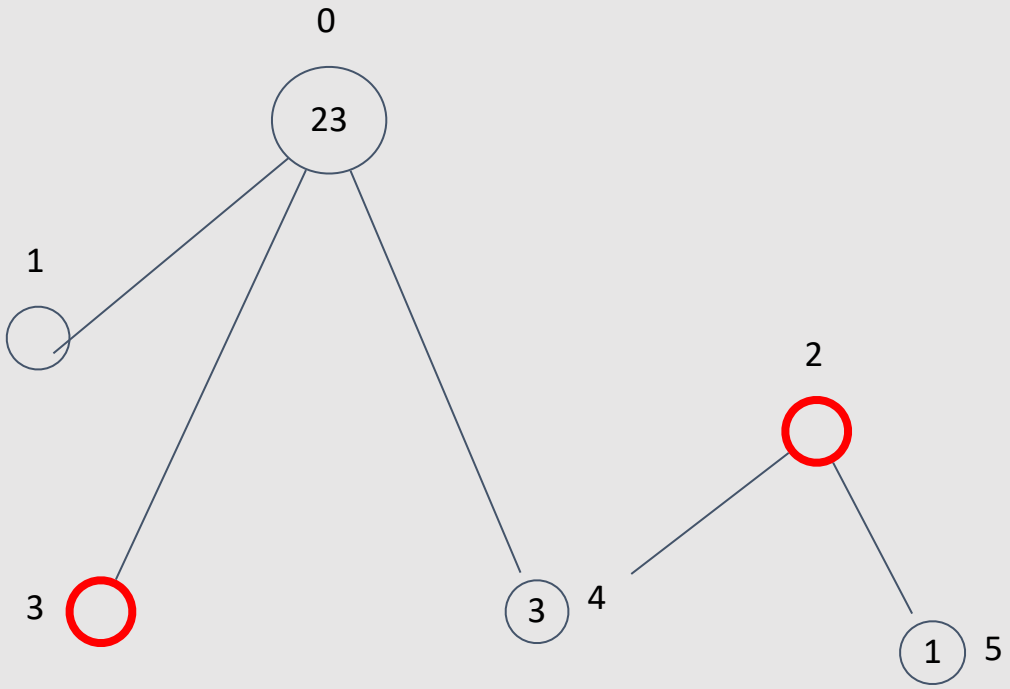


euler = [0, 1, 1, 3, 3, 4, 2, 5, 5, 2, 4, 0]

a = [0, 0, 0, 0, 0, 1, 4, 4, 4, 4, 1, 0, 0]



? (3, 5) = (2 + 23 + 3 + 1 + 1) + 0 + 5



All codes

binup(alt) - https://ideone.com/lNrwm9

binup(reg) - https://ideone.com/cetrCM

sparse_table(c++) - https://ideone.com/0uHSfH

sparse with log(c++) - https://ideone.com/Gz2CVK

euler tour(c++) - https://ideone.com/83hAyo

Lca offline - https://ideone.com/S4N8pZ