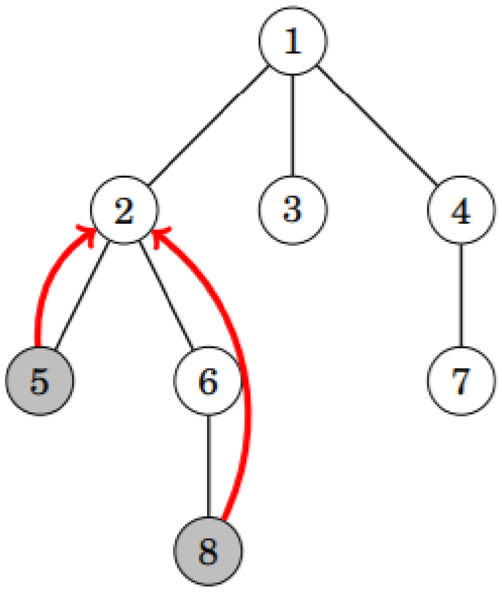
Dynamic Programming - 3

Lowest Common Ancestor

Problem Statement

The lowest common ancestor (LCA) is a concept in graph theory and computer science. Let ‘T’ be a rooted tree with ‘N’ nodes. The lowest common ancestor is defined between two nodes, ‘u’ and ‘v’, as the lowest node in ‘T’ that has both ‘u’ and ‘v’ as descendants (where we allow a node to be a descendant of itself).



For the given tree, The LCA of nodes 5 and 8 will be node 2, as node 2 is the first node that lies in the path from node 5 to root node 1 and from node 8 to root node 1.

The path from node 5 to root node looks like 5 → 2 → 1.

The path from node 8 to root node looks like 8 → 6 → 2 → 1.

Since 2 is the first node that lies in both paths. Hence LCA will be 2.

Given any two nodes ‘u’ and ‘v’, find the LCA for the two nodes in the given Tree ‘T’.

Explanation:

The above problem can be solved using the binary lifting technique. Here, we will be dividing our solution into various parts and we will do pre-processing to reduce the time complexity.

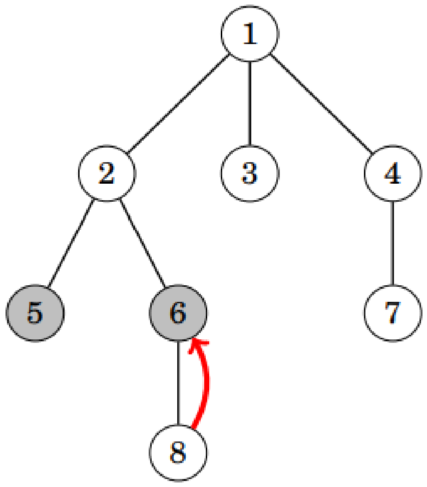
Step 1:

For each node, we will precompute its ancestor above him, its ancestor two nodes above, its ancestor four above, etc. Let's store them in the array **dp**, i.e. **dp[i][u]** is the **2^i**-th ancestor above the node **u** with **u=1...N**, **i=0...ceil(log(N))**. This information allows us to jump from any node to any ancestor above it in O(logN) time. We can compute this array using a DFS traversal of the tree. So total complexity for this entire pre-processing comes out to be O(NlogN).

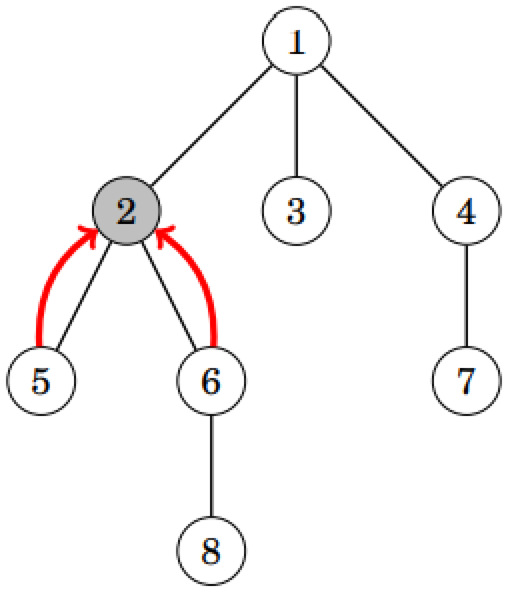
Step 2:

Now, we use two pointers that initially point to the two nodes whose LCA we have to find.

Now, one of the pointers moves upwards so that both pointers point to nodes at the same level. i.e. we will move our pointer upward, which has a higher depth until both of the pointers reach the same depth. This can be done in O(log(N)) by using a sparse table.



After this, we determine the minimum number of steps needed to move both pointers upwards to point to the same node. The node to which the pointers point after this is the lowest common ancestor. The example suffices to move both pointers one step upwards to node 1, which is the lowest common ancestor.



Algorithm:

Initialize a variable ‘level’ to log(N) + 3;

Initialize a 2d array ‘dp’ of dimension ‘(N + 1) * level’ to find the ‘kth’ ancestor of any tree node using the sparse table technique.

Initialize an array ‘depth’ of length ‘N + 1’ to store the depth of all the nodes from root node 1.

- Func dfs(curr, par)
    - Update dp[cur][0] to par.
    - Update depth[cur] to depth[par] + 1.
    - Run a for loop from i = ‘1’ to ‘level’. → standard filing of a sparse table
        - If dp[cur][i - 1] is not ‘0’.
            - Update dp[cur][i] to dp[dp[cur][i - 1]][i -1].
    - Iterate in all the children of node ‘cur’ and call dfs as dfs(child, cur)

- Func get_lca(u, v):
    - Swap ‘u’ and ‘v’ if the depth of node ‘u’ is greater than node ‘v.’,I.e. if depth[u] > depth[v] swap(u, v)
    - Initialize a variable dis equals to depth[v] - depth[u].
    - Run a for loop from i = ‘0’ to ‘level’. → making both the nodes at the same depth.
        - If ‘ith’ bit in ‘dis’ is set i.e. d & (1 << i) == 1.
            - Update ‘v’ to dp[v][i]
    - If node ‘u’ and ‘v’ are the same, return ‘u’ as we have found the LCA.

    - Run a for loop from i = ‘level - 1’ to ‘0’.
        - If (2 ^ i)th parent of ‘u’ and ‘v’ is same, i.e. dp[u][i] == dp[v][i],

            ■ We do nothing
        ■ Else
            ■ We update ‘u’ to dp[u][i]
            ■ We update ‘v’ to dp[v][i]
    ○ We return the parent of ‘u’, i.e. dp[u][0].

Func Main():

- dfs(1, 0)
- For each query
    - Print get_lca(u, v)

Code :

```
const int MAX_N = 1e5 + 10;
const int lvl = 20;
vector<int> graph[MAX_N];
int dp[MAX_N][lvl];
int depth[MAX_N];
void dfs(int cur, int par)
{
    dp[cur][0] = par;
    depth[cur] = depth[par] + 1;
    for (int i = 1; i < lvl; ++i)
    {
        if(dp[cur][i - 1])
        {
          dp[cur][i] = dp[dp[cur][i - 1]][i - 1];
        }
    }
    for (auto& i : graph[cur])
    {
        if(i != par)
        {
            dfs(i, cur);
        }
    }
```

```
}

int get_lca(int u, int v)
{
    if (depth[u] > depth[v])
    {
        swap(u, v);
    }

    int d = depth[v] - depth[u];
    for (int i = 0; i < lvl; ++i)
    {
        if(d & (1 << i))
        {
            v = dp[v][i];
        }
    }

    if (u == v)
    {
        return u;
    }
    for (int i = lvl - 1; i >= 0; --i)
    {
        if(dp[u][i] != dp[v][i])
        {
            u = dp[u][i];
            v = dp[v][i];
        }
    }

    return dp[u][0];
}

vector<int> lca(int n, vector<vector<int>> edge, vector<vector<int>> query)
{
    for (int i = 0; i < n + 1; ++i)
    {
        graph[i].clear();
        depth[i] = 0;
        for (int j = 0; j < lvl; ++j)
        {
            dp[i][j] = 0;
        }
    }
    for (auto& i : edge)
    {
        graph[i[0]].push_back(i[1]);
        graph[i[1]].push_back(i[0]);
    }

    dfs(1, 0);
    vector<int> ans;
    for (auto& i : query)
    {
        ans.push_back(get_lca(i[0], i[1]));
    }
    return ans;
}
```

Time Complexity

O(Q * log(N) + N * log(N)), where N is the number of nodes of the tree and Q is the number of queries. **N * log(N)** is for pre-processing that we are doing to create the sparse table. For each query, we are taking **log(N)** time to answer it, so a total of **Q * log(N).** Hence overall complexity is **O(Q * log(N) + N * log(N)).**

Space Complexity

O(N * log(N)), where N is the number of nodes of the tree.

We are creating a 2-D array of size **N * log(N)**. Hence overall complexity is **O(N * log(N)).**

Maximize Profit

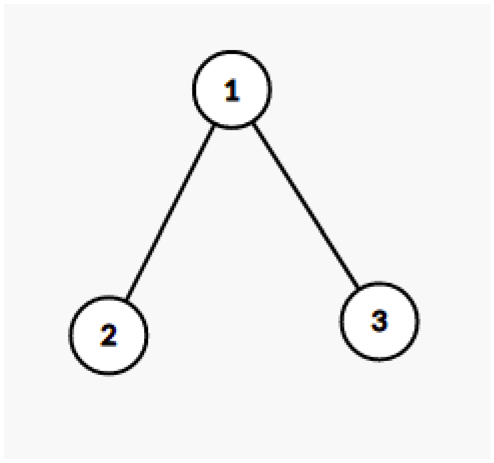
Problem Statement

You are given a tree having 'N' *(1 <= N <= 10^5)* nodes, where each node has some coins 'costs[i]', (*1 <= costs[i] <= 10^9*) attached to it.

You are allowed to collect the coins from any node. Your task is to maximize the sum of chosen coins given that you cannot choose coins from adjacent nodes(i.e., nodes which are directly connected by an edge).

Explanation:

Let's take a simple tree.



Now if we take coins at node 1 then for sure we can not take coins present at nodes 2 and 3, but if we skip node 1 then, we have two options for each of the other two nodes .i.e we can either take or skip them.

To generalize this, we can say that if we take the parent node, we need to skip all its direct children nodes, and if we skip the parent node, we can either take its children node or skip it.

For easy implementation, we can define two 2-dimensional dp arrays namely dp1[N] and dp2[N], where,

dp1[v]= Maximum sum of coins possible by choosing coins from subtree of node v including coin from node v also.

dp2[v]= Maximum sum of coins possible by choosing coins from subtree of node v such that we don't include coins at node v in our answer.

So, Our final answer is a maximum of two cases i.e. max(dp1[1],dp2[1]).

And defining recursion is even easier, in this case,

dp1[v]=costs[v]+$\sum_{i=1}^{n} dp2[vi]$**.** (since we cannot include any of the children) and

dp2[v]=$\sum_{i=1}^{n} max(dp1[vi], dp2[vi])$(since we can include children now, but we can also choose not to include them in the subset, hence the max of both cases).

Algorithm:

```
Func dfs(s, p)
        ○ Update dp1[s]=costs[s]. // costs[s] is the value of a coin stored at node s.
        ○ Update dp2[s]=0.
        ○ Traverse all the nodes connected to s.
              ■ If the current node(i.e., curr)  is not the parent of s.
              ■ call dfs as dfs(curr, s)
              ■ dp1[s]+=dp2[curr]
              ■ dp2[s]+=max(dp1[curr],dp2[curr]).
Func Main():
    ● dfs(1, 0)
    ● coût<<max(dp1[1],dp2[1]);
```

Code:

```
long long int solve(int node, int parentNode, bool b, vector<int> v[],
vector<int> &c, vector<vector<long long int>> &dp)
{

    if(dp[node][b] != -1)
    {
        return dp[node][b];
    }

    long long int sum = 0;
    for (int i = 0; i < (int)v[node].size(); i++)
    {
        if(v[node][i] == parentNode)
        {
            continue;
        }
        if (b == 1)
        {
            sum += solve(v[node][i], node, 0, v, c, dp);
        }
        else
        {
            sum += max(solve(v[node][i], node, 0, v, c, dp),
solve(v[node][i], node, 1, v, c, dp));
        }
    }
    if(b == 1)
    {
        sum += c[node];
    }


    dp[node][b] = sum;

    return sum;
}

long long int maximizeProfit(int n, vector<int> c, vector<vector<int>>
edges)
{
    vector<int> v[n];
    vector<vector<long long int>> dp(n, {-1, -1});

    for(int i = 0; i + 1 < n; i++)
    {
        int x = edges[i][0];
        int y = edges[i][1];

        x--;
        y--;
        v[x].push_back(y);
        v[y].push_back(x);
    }

    long long int ans = max(solve(0, 0, 1, v, c, dp), solve(0, 0, 0, v, c,
dp));

    return ans;

}
```

Time Complexity

O(N), where ‘N’ is the number of nodes in the tree. Since we are going to visit each node at most two times, the final time complexity becomes O(N).

Space Complexity

O(N), where ‘N’ is the number of nodes in the tree. We will be using two 2d arrays of size ‘N’ to memorize the states. So space complexity becomes O(N).