Lowest Common Ancestor

(Binary Lifting)

Given a tree G and queries of the from (u_1, u_2), for each query, the task is to find the lowest common ancestor, $LCA(u_1, u_2)$ described in the last topic.

The algorithm discussed before used RMQ on *Euler Tour* of the tree. It found the vertex with the least depth on the path between u_1 and u_2. However, that algorithm can only do limited tasks.

This algorithm can be used to tell the weight of the heaviest edge or the lightest edge in the path between any two vertices. This application is interesting and has many uses in other problems. One such problem is finding the second best minimum spanning tree. For that purpose, we will learn this algorithm.

In Binary Lifting, we will maintain an array $ancestor[i][j]$ which stores the $2^{j\text{-th}}$ ancestor of the node i. This allows us to jump to any ancestor of any vertex in $O(\log V)$ time. This array can be filled with a simple DFS traversal. We will also maintain arrays $tin[]$ and $tout[]$ which record the in-time and out-time of each vertex during the DFS.

Consider a query $LCA(u, v)$. First, we check if u or v are ancestors of each other using the $tin[]$ and $tout[]$ arrays in $O(1)$ time. If yes, we return the result. Otherwise, we jump to an ancestor of u which is also the highest node that is not an ancestor of v. That is, we find a node q such that q is not an ancestor of v, but $ancestor[q][0]$ is. We can find q in $O(\log V)$ time, again using $ancestor[][]$.

Let $L = \lceil \log V \rceil$. Assign $i = L$. If $ancestor[u][i]$ is not an ancestor of v, then we assign $u = ancestor[u][i]$ and decrement i. If $ancestor[u][i]$ is an ancestor, then we just decrement i. Clearly, after doing this for all non-negative i the node u will be the desired node - i.e. u is still not an ancestor of v, but $ancestor[u][0]$ is.

Now, obviously, the answer to LCA will be $ancestor[u][0]$ - i.e., the node with the highest depth among the ancestors of the node u, which is also an ancestor of v.

Below is the implementation of the approach discussed above, in a `struct LCA`.

```cpp
// lcaBinaryLift.cpp

#include <vector>
#include <algorithm>

struct LCA {
    int timer, L, V;
    std::vector<int> tin, tout;
    std::vector<std::vector<int>> ancestor;

    LCA(std::vector<std::vector<int>> &adj, int _V, int root = 1) :
        timer(0),
        V(_V),
        tin(std::vector<int>(V + 1)),
        tout(std::vector<int>(V + 1)) {

        for (L = 1; L < V; L <<= 1); // L = ceil(log V)
```

```
        ancestor.assign(V + 1, std::vector<int>(L + 1));
        dfs(adj, root, root);
    }

    void dfs(std::vector<std::vector<int>> &adj, int u, int parent) {
        tin[u] = ++timer;
        ancestor[u][0] = parent;
        for (int i = 1; i <= L; ++i)
            ancestor[u][i] = ancestor[ancestor[u][i - 1]][i - 1];
        for (int v : adj[u]) {
            if (v != parent) {
                dfs(adj, v, u);
            }
        }
        tout[u] = ++timer;
    }

    bool isAncestor(int u, int v) {
        return (tin[u] <= tin[v] and tout[u] >= tout[v]);
    }

    int lca(int u, int v)
    {
        if (isAncestor(u, v))
            return u;
        if (isAncestor(v, u))
            return v;
        for (int i = L; i >= 0; --i) {
            if (not isAncestor(ancestor[u][i], v))
                u = ancestor[u][i];
        }
        return ancestor[u][0];
    }
};
```

So answering a LCA query will iterate i from L to 0 and checks in each iteration if one node is the ancestor of the other. So, each query can be answered in $O(\log V)$.

Therefore,

- Time complexity for preprocessing: $O(V \log V)$.
- Time complexity for LCA query: $O(\log V)$.
- Space complexity: $O(V \log V)$.