Lowest Common Ancestor

(Using RMQ)

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, i.e. a vertex u that lies on the path from root to u_1 and the path from root to u_2, and the the vertex u should be the lowest (lowest depth possible).

The idea is to preprocess the tree using DFS at root. We do an *Euler tour* of the tree and store the order of visiting vertices in a container $euler$ and store the depth of all nodes in $depth$ as we perform the DFS. We also maintain an array $firstFoundAt[]$ which stores the position of the vertex i in the container $euler$ when it was first encountered during the Euler tour.

Using this information, we can find the LCA of any two vertices in the tree. It is obvious to see that from $euler[firstFoundAt[u_1]]$ to $euler[firstFoundAt[u_2]]$, this is a path between u_1 and u_2. The $LCA(u_1, u_2)$ lies on this path. It is the vertex with lowest depth in all the vertices on the path said above.

We can maintain a segment tree to return the vertex in the Euler tour with minimum depth. This is how the problem of lowest common ancestor is associated with Range Minimum Query (RMQ). The segment tree performs update and minimum extraction operations in $(O \log V)$ time with $O(V)$ preprocessing time needed for building the tree.

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

```cpp
// lcaSegmentTree.cpp

#include <vector>
#include <algorithm>

struct LCA {
    const int INF = 0x3f3f3f3f;
    std::vector<int> depth, euler, firstFoundAt, segTree;
    std::vector<bool> visited;
    int V, segTreeSize;

    LCA(std::vector<std::vector<int>> &adj, int _V, int root = 1) :
        V(_V),
        depth(std::vector<int>(V + 1)),
        firstFoundAt(std::vector<int>(V + 1)),
        visited(std::vector<bool>(V + 1)) {

        euler.reserve(2 * V);
        depth[root] = -1;
        eulerTourDFS(adj, root, root);
        depth[root] = INF;
        int m = euler.size(), sz;
        for (sz = 1; sz < m; sz <<= 1);
        segTree.resize(sz << 1);
        for (int i = 0; i < m; ++i)
            segTree[sz + i] = euler[i];

        for (int i = sz - 1; i >= 1; --i) {
            int l = segTree[i << 1], r = segTree[i << 1 | 1];
            segTree[i] = (depth[l] > depth[r] ? r : l);
        }
        segTreeSize = sz << 1;
    }

    void eulerTourDFS(std::vector<std::vector<int>> &adj, int u, int parent) {
        visited[u] = true;
        depth[u] = depth[parent] + 1;
        firstFoundAt[u] = int(euler.size());
        euler.push_back(u);
        for (int v : adj[u]) {
            if (not visited[v]) {
                eulerTourDFS(adj, v, u);
                euler.push_back(u);
            }
        }
    }

    int query(int L, int R, int l = 0, int r = - 1, int v = 1)
    {
        if (r == -1)
            r += segTreeSize;
        if (R < l or L > r)
            return 0;
        if (L <= l and R >= r)
            return segTree[v];
        int mid = l + ((r - l) >> 1);
        int left = query(L, R, l, mid, v << 1);
        int right = query(L, R, mid + 1, r, v << 1 | 1);
        return (depth[left] > depth[right] ? right : left);
    }

    int lca(int u, int v) {
        int l = firstFoundAt[u];
        int r = firstFoundAt[v];
        if (l > r)
            std::swap(l, r);
        return query(l, r);
    }
};

int main() {}
```

The segment query requires $O(\log V)$ time. The preprocessing time for the segment tree is $O(V)$ and for the Euler Tour DFS is $O(V + E)$.

Therefore,

- Time complexity for preprocessing: $O(V + E)$.

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

The main idea was to understand how the *Euler Tour* of the tree with RMQ can be used to find the LCA of any two vertices in the tree. Instead of a segment tree, a sparse table can also be used.

The RMQ operation in a sparse table requires $O(1)$ time but $O(V \log V)$ preprocessing time to build the table.

Similarly, a square-root decomposition tree can be used whose query requires $O(\sqrt{V})$ time and $O(V)$ preprocessing time.