added: 10 Jun 2008 22:48
Edited: 14 Mar 2012 2:09

Lowest common ancestor. Being in O (log N) (binary lifting method)

Contents [hide]

Suppose we are given a tree G.
The input receives requests form (V1, V2), for each request is required to find their least common ancestor, ie a vertex V, which lies in the path from the root to V1, the path from the root to V2, and all of these peaks should be chosen lowermost. In other words, the required vertex V - ancestor and V1, and V2, and among all such common ancestors selected lower. It is obvious that the lowest common ancestor of vertices V1 and V2 - it is their common ancestor, which lies on the shortest path from V1 to V2. In particular, for example, if V1 is the ancestor of V2, then V1 is their least common ancestor.

In English, this problem is called the problem LCA - Least Common Ancestor.

Here will be considered an algorithm which is written much faster than the one described here .

Asymptotic behavior of the resulting algorithm will be: preprocessing of the **O (N log N)** and the response to each request for **O (Log N)** .

Algorithm

Predposchitaem for each vertex of its ancestor 1st, 2nd ancestor, 4th, etc. We denote this array by P, that is P [i] [j] - a 2^j -th ancestor of vertex i, i = 1..N, j = 0..⌈logN⌉. Also, for each vertex we find the time to call her and output depth-first search (see. "Depth-first search") - it is, we need to determine in O (1) whether one vertex is an ancestor of the other (not necessarily direct). This preprocessing can be done in **O (N Log N)** .

Suppose now entered another request - a pair of vertices (A, B). Immediately check whether the ancestor of one vertex other - in this case, it is the result. If A is not an ancestor of B, and B is not an ancestor of A, then we will climb the ancestors of A, until we find the highest (ie, closest to the root) vertex, which is not an ancestor (not necessarily direct) B (t. e. a vertex X, such that X is not an ancestor of B, and P [X] [0] - the ancestor of B). In this case, find the vertex X is in O (log N), using an array of P.

We describe this process in more detail. Let L = ⌈logN⌉. Assume first that I = L. If P [A] [I] is not an ancestor of B, then assign A = P [A] [I], and reduce I. If P [A] [I] is an ancestor of B, then simply reduce I. Obviously, when I would be less than zero, the vertex A just and will be required vertex - ie such that A is not an ancestor of B, but P [A] [0] - the ancestor of B.

Now, obviously, the answer to the LCA will be P [A] [0] - ie smallest vertex

among the ancestors of the original vertices A, which is also the ancestor of B.

Asymptotics. The whole algorithm is responding to a request from the change l of L = ⌈logN⌉ to 0 and check each step in O (1) whether one vertex is an ancestor of the other. Consequently, for each query answer is found in O (log N).

Implementation

```
int n, l;
vector <vector <int>> g;
vector <int> tin, tout;
int timer;
vector <vector <int>> up;

void dfs (int v, int p = 0) {
        tin [v] = ++ timer;
        up [v] [0] = p;
        for (int i = 1; i <= l; ++ i)
                up [v] [i] = up [up [v] [i-1]] [i-1];
        for (size_t i = 0; i <g [v] .size (); ++ i) {
                int to = g [v] [i];
                if (to! = p)
                        dfs (to, v);
        }
        tout [v] = ++ timer;
}

bool upper (int a, int b) {
        return tin [a] <= tin [b] && tout [a]> = tout [b];
}

int lca (int a, int b) {
        if (upper (a, b)) return a;
        if (upper (b, a)) return b;
        for (int i = l; i> = 0; --i)
                if (! upper (up [a] [i], b))
                        a = up [a] [i];
        return up [a] [0];
}

int main () {

        Reading ... n and g ...

        tin.resize (n), tout.resize (n), up.resize (n);
        l = 1;
        while ((1 << l) <= n) ++ l;
        for (int i = 0; i <n; ++ i) up [i] .resize (l + 1);
        dfs (0);
```

```
        for (;;) {
                int a, b; // Current request
                int res = lca (a, b); // Response to a request
        }

}
```