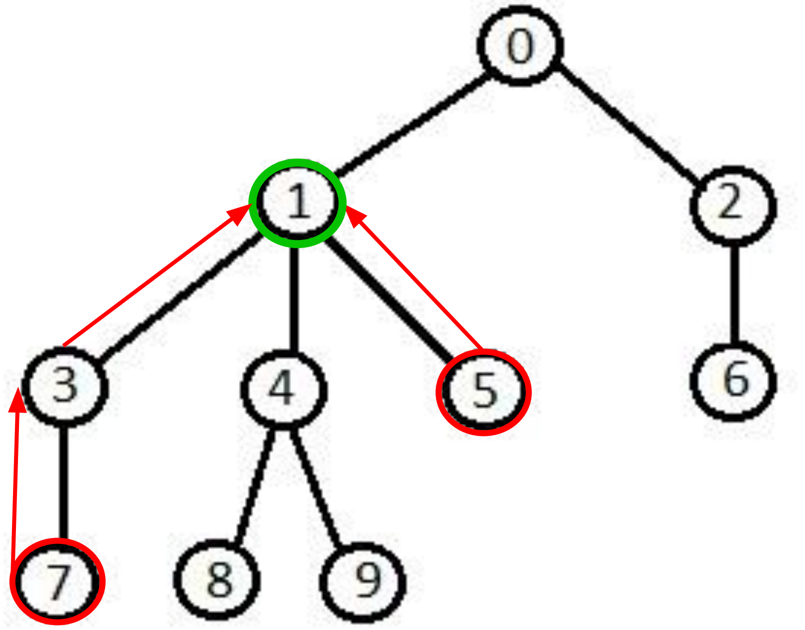
LCA in O(log n) time

By Andi Qu

What is LCA?

LCA is the lowest common ancestor (common ancestor with maximal depth) of 2 nodes.



Why do we care?

- It’s quite common in hard problems (e.g. USACO Platinum)
    - USACO Platinum December 2015 “Max Flow”
    - USACO Platinum December 2018 “Gathering”
    - USACO Platinum January 2019 “Exercise”
- It’s useful in the real world
    - Merging algorithms in VCS (3-way merge)
    - Finding social media influencers
    - Literally finding lowest common ancestors in genetics

Other common names for LCA

- Binary Lifting
  - Actually more common than LCA sometimes
- DP on trees
  - Slightly less common
- “That stupid Fenwick tree technique”
  - Slightly less common

Common LCA strategies

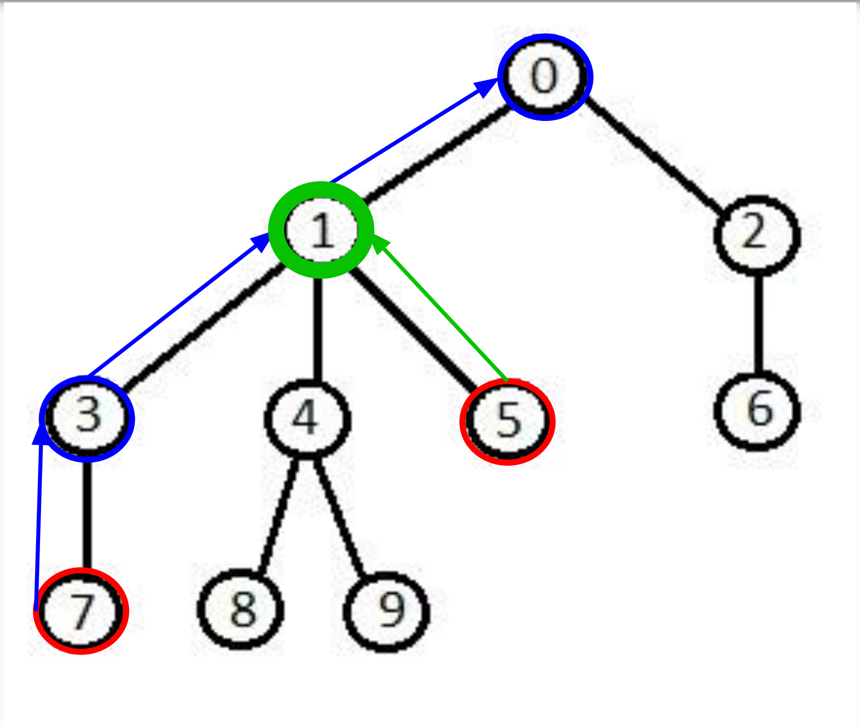
- **⟨O(n), O(n)⟩** (Naive brute force method)
- **⟨O(n), O(√n)⟩** (Square root decomposition)
- **⟨O(n log n), O(log n)⟩** (Sparse table + binary lifting + DP)
- **⟨O(n), O(log n)⟩** (Everyone's favourite data structure)
- **⟨O(n log n), O(1)⟩** (Sparse table again but somehow better)

Where n is the number of nodes in the tree.

An ⟨O(n), O(n)⟩ solution

1. Traverse up the tree from one of the nodes
2. Mark all visited nodes as visited
3. Traverse up the tree from the other node
4. Return the first visited node that is marked as visited

An ⟨O(n), O(n)⟩ solution



O(n) Solution - Code

```cpp
vector<int> tree[MAXN];
int parent[MAXN];
bool visited[MAXN];

void dfs(int current = 1, int p = -1) { // Get parents
   parent[current] = p;
   visited[current] = true;
   for (auto& i : tree[current]) {
       if (!visited[i]) {
           dfs(i, current);
       }
   }
}

int lca(int a, int b) { // Answer queries
   fill(visited, visited + MAXN, false);
   while (a != -1) {
       visited[a] = true;
       a = parent[a];
   }
   while (!visited[b]) {
       b = parent[b];
   }
   return b;
}
```

Can we do better?

Our O(n) solution is far too slow, and will usually get a TLE in a contest, since we usually need to answer multiple LCA queries in a single test case.

We want to get a O(√n) or O(log n) solution with some preprocessing.

(But O(log n) is better than O(√n) so we're just going to find a O(log n) solution.)

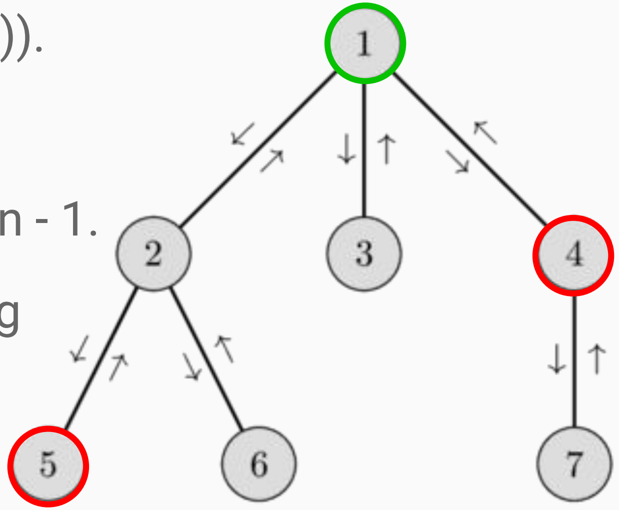
Consider the Euler tour of the tree

Notice that LCA(a, b) = RMQ(find(euler, a), find(euler, b)).

We know that we can answer RMQ in O(log n) time.

The length of the Euler tour for a tree with n nodes is 2n - 1.

This mean we can answer LCA with O(n) preprocessing and O(log n) queries!



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

How do we answer RMQ in O(log n) time?

Preprocessing - Euler tour

```
vector<int> tree[MAXN];
int euler[MAXN * 2 - 1];
int first_occurrence[MAXN];
int euler_size = 0;
bool visited[MAXN];

void euler_tour(int current = 1) { // Basically a dfs
   visited[current] = true;
   first_occurrence[current] = euler_size;
   euler[euler_size++] = current;
   for (auto& i : tree[current]) {
       if (!visited[i]) {
           euler_tour(i);
           euler[euler_size++] = current;
       }
   }
}
```

Preprocessing - Segment tree

```
int segtree[4 * euler_size];

void build(int v = 1, int tl = 0, int tr = euler_size - 1) { // Standard segtree
    if (tl == tr) {
        segtree[v] = euler[tl];
    } else {
        int tm = (tl + tr) / 2;
        build(v * 2, tl, tm);
        build(v * 2 + 1, tm + 1, tr);
        segtree[v] = min(segtree[v * 2], segtree[v * 2 + 1]);
    }
}
```

Query

```
int query(int v, int tl, int tr, int l, int r) { // Standard RMQ
    if (l > r) return INT_MAX;
    if (l == tl && r == tr) return segtree[v];
    int tm = (tl + tr) / 2;
    return min(query(v * 2, tl, tm, l, min(r, tm)),
               query(v * 2 + 1, tm + 1, tr, max(l, tm + 1), r));
}

int lca(int l, int r) {
    int left = first_occurrence[l], right = first_occurrence[r];
    if (left > right) {
        return query(1, 0, euler_size - 1, right, left);
    } else {
        return query(1, 0, euler_size - 1, left, right);
    }
}
```

LCA using a Cartesian tree (Restricted RMQ)

Since we are able to reduce finding LCA to finding RMQ, technically we are able to find LCA with ⟨**O(n), O(1)**⟩ using the Fischer-Heun structure.

But this is a bad idea:

- Cartesian trees are really complicated (both to understand and to code).
- In practice, the ⟨**O(n), O(log n)**⟩ segment tree approach is a bit faster.

See https://web.stanford.edu/class/cs166/lectures/01/Slides01.pdf pg 11 and onwards for details

(And https://www.topcoder.com/community/competitive-programming/tutorials/range-minimum-query-and-lowest-common-ancestor/ for more LCA stuff)

More RMQ

- **⟨$O(n^2)$, $O(1)$⟩** (full preprocessing)
- **⟨$O(n \log \log n)$, $O(1)$⟩** (hybrid approach)
- **⟨$O(n)$, $O(\log n)$⟩** (hybrid approach)
- **⟨$O(n)$, $O(\log \log n)$⟩** (hybrid approach)

But don’t do these - they’re complicated and unnecessary

Example problem - Usaco 2018-2019 December Platinum Q3 “Gathering”

Cows have assembled from around the world for a massive gathering. There are N cows, and $N-1$ pairs of cows who are friends with each other. Every cow knows every other cow through some chain of friendships.

They had great fun, but the time has come for them to leave, one by one. They want to leave in some order such that as long as there are still at least two cows left, every remaining cow has a remaining friend. Furthermore, due to issues with luggage storage, there are M pairs of cows (a_i, b_i) such that cow a_i must leave before cow b_i. Note that the cows a_i and b_i may or may not be friends.

Help the cows figure out, for each cow, whether she could be the last cow to leave. It may be that there is no way for the cows to leave satisfying the above constraints.

Solution to the USACO problem

The problem can be rephrased as follows: removing leaves from a tree one by one while respecting order constraints, determine the possible final nodes.

The official solution requires the use of a greedy algorithm and a DFS...

But everyone knows that DFS actually stands for Dull, Flat, and Soulless, and that Greed is one of the seven deadly sins...

So we use LCA instead (aka Love, Care, and Affection)!

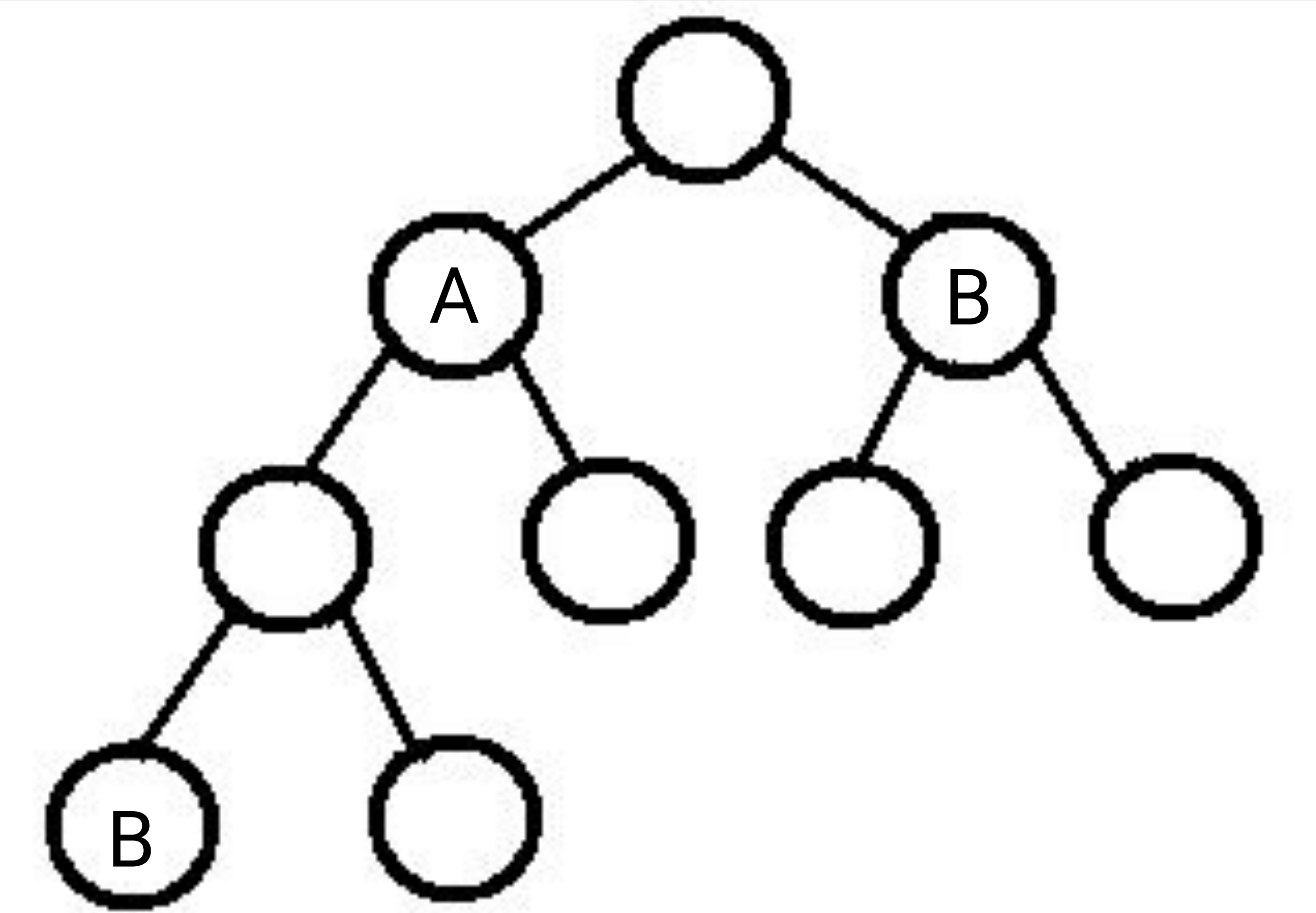
Solution insight

Taking 1 as the root, if we have cow *a* must leave before cow *b*, then we have the following:

- If LCA(*a*, *b*) = a, then every node except for the subtree of *a* containing *b* is invalidated.
- Otherwise, every node in the subtree with root *a* is invalidated.

This is because we must have *a* removed before *b*, meaning that you must remove those nodes to remove *a* and therefore remove *b*.

Illustration



Solution sketch

- Make the LCA segment tree.
- Do LCA on each query (a, b).
- Invalidate nodes as per the condition mentioned earlier.
  - Use a tree prefix sum or a **segment tree** avoid having to do a DFS every time.
- Run a DFS and see which nodes are valid and which are not.

The runtime of this algorithm is ⟨**O(N), O(N log N + M log N)**⟩.

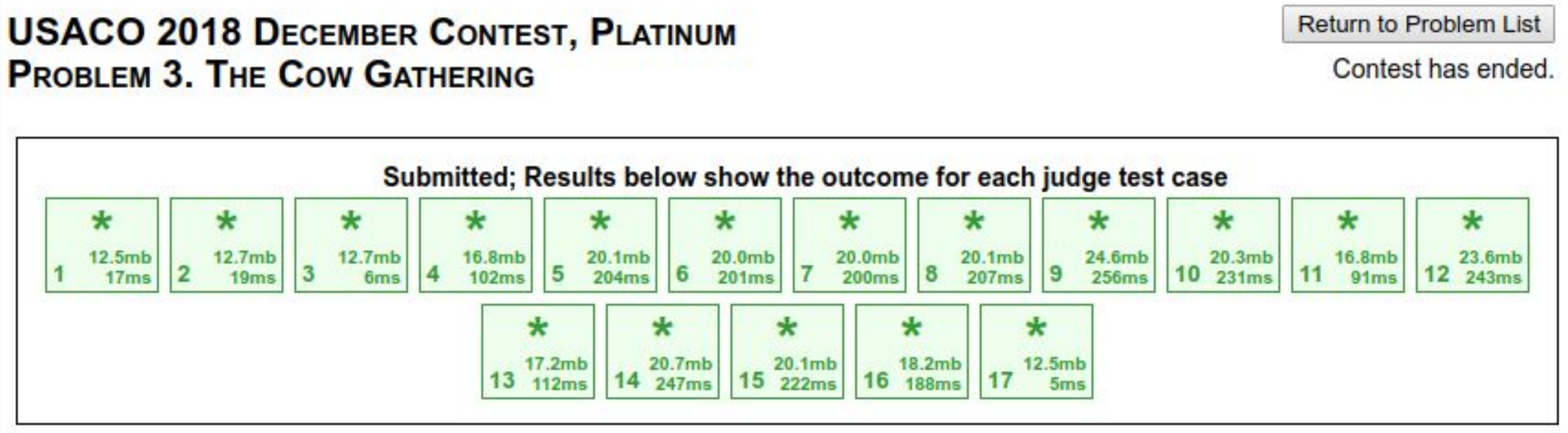
(⟨**O(N), O(NM log N)**⟩ if you don't use the tree prefix sum or **segment tree**).

Code (In C++)   (Thanks, Benq)

```cpp
#pragma GCC optimize ("O3")
#pragma GCC target ("sse4")

#include <bits/stdc++.h>
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/rope>

using namespace std;
using namespace __gnu_pbds;
using namespace __gnu_cxx;

typedef long long ll;
typedef long double ld;
typedef complex<ld> cd;

typedef pair<int, int> pi;
typedef pair<ll,ll> pl;
typedef pair<ld,ld> pd;

typedef vector<int> vi;
typedef vector<ld> vd;
typedef vector<ll> vl;
typedef vector<pi> vpi;
typedef vector<pl> vpl;
typedef vector<cd> vcd;

template <class T> using Tree = tree<T, null_type, less<T>, rb_tree_tag,tree_order_statistics_node_update>;

#define FOR(i, a, b) for (int i = (a); i < (b); i++)
#define F0R(i, a) for (int i = 0; i < (a); i++)
#define FORd(i,a,b) for (int i = (b)-1; i >= (a); i--)
#define F0Rd(i,a) for (int i = (a)-1; i >= 0; i--)
#define trav(a, x) for (auto& a : x)

#define mp make_pair
#define pb push_back
#define f first
#define s second
#define lb lower_bound
#define ub upper_bound

#define sz(x) (int)x.size()
#define beg(x) x.begin()
#define en(x) x.end()
#define all(x) beg(x), en(x)
#define resz resize

const int MOD = 1000000007;
const ll INF = 1e18;
const int MX = 100001;
const ld PI = 4*atan((ld)1);

template<class T> void ckmin(T &a, T b) { a = min(a, b); }
template<class T> void ckmax(T &a, T b) { a = max(a, b); }

namespace io {
    // TYPE ID (StackOverflow)

    template<class T> struct like_array : is_array<T>{};
    template<class T, size_t N> struct like_array<array<T,N>> : true_type{};
    template<class T> struct like_array<vector<T>> : true_type{};
    template<class T> bool is_like_array(const T& a) { return like_array<T>::value; }

    // I/O

    void setIn(string s) { freopen(s.c_str(),"r",stdin); }
    void setOut(string s) { freopen(s.c_str(),"w",stdout); }
    void setIO(string s = "") {
        ios_base::sync_with_stdio(0); cin.tie(0);
        if (sz(s)) { setIn(s+".in"), setOut(s+".out"); }
    }

    // INPUT

    template<class T> void re(T& x) { cin >> x; }
    template<class Arg, class... Args> void re(Arg& first, Args&... rest);
    void re(double& x) { string t; re(t); x = stod(t); }
    void re(ld& x) { string t; re(t); x = stold(t); }

    template<class T> void re(complex<T>& x);
    template<class T1, class T2> void re(pair<T1,T2>& p);
    template<class T> void re(vector<T>& a);
    template<class T, size_t SZ> void re(array<T,SZ>& a);

    template<class Arg, class... Args> void re(Arg& first, Args&... rest) { re(first); re(rest...); }
    template<class T> void re(complex<T>& x) { T a,b; re(a,b); x = cd(a,b); }
    template<class T1, class T2> void re(pair<T1,T2>& p) { re(p.f,p.s); }
    template<class T> void re(vector<T>& a) { F0R(i,sz(a)) re(a[i]); }
    template<class T, size_t SZ> void re(array<T,SZ>& a) { F0R(i,SZ) re(a[i]); }

    // OUTPUT

    template<class T1, class T2> ostream& operator<<(ostream& os, const pair<T1,T2>& a) {
        os << '{' << a.f << ", " << a.s << '}'; return os;
    }
    template<class T> ostream& printArray(ostream& os, const T& a, int SZ) {
        os << '{';
        F0R(i,SZ) {
            if (i) {
                os << ", ";
                if (is_like_array(a[i])) cout << "\n";
            }
            os << a[i];
        }
        os << '}';
        return os;
    }
    template<class T> ostream& operator<<(ostream& os, const vector<T>& a) {
        return printArray(os,a,sz(a));
    }
    template<class T, size_t SZ> ostream& operator<<(ostream& os, const array<T,SZ>& a) {
        return printArray(os,a,SZ);
    }

    template<class T> void pr(const T& x) { cout << x << '\n'; }
    template<class Arg, class... Args> void pr(const Arg& first, const Args&... rest) {
        cout << first << ' '; pr(rest...);
    }
}

using namespace io;

int N,M, dir[MX];

void finish() {
    FOR(i,1,N+1) pr(0);
    exit(0);
}

template<int SZ> struct Topo {
    int N, in[SZ], ok[SZ];
    vi res, adj[SZ];

    void addEdge(int x, int y) {
        adj[x].pb(y), in[y] ++;
    }

    void sort() {
        queue<int> todo;
        FOR(i,1,N+1) if (in[i] == 0) {
            ok[i] = 1;
            todo.push(i);
        }
        while (sz(todo)) {
            int x = todo.front(); todo.pop();
            res.pb(x);
            for (int i: adj[x]) {
                in[i] --;
                if (!in[i]) todo.push(i);
            }
        }
        if (sz(res) == N) {
            FOR(i,1,N+1) pr(ok[i]);
        } else {
            finish();
        }
    }
};

template<int SZ> struct LCA {
    const int MAXK = 32-__builtin_clz(SZ);

    int N, R = 1; // vertices from 1 to N, R = root
    vi adj[SZ];
    int par[32-__builtin_clz(SZ)][SZ], depth[SZ];

    void addEdge(int u, int v) {
        adj[u].pb(v), adj[v].pb(u);
    }

  void dfs(int u, int prev){
        par[0][u] = prev;
        depth[u] = depth[prev]+1;
        for (int v: adj[u]) if (v != prev) dfs(v, u);
    }

    void init(int _N) {
        N = _N;
        dfs(R, 0);
        FOR(k,1,MAXK) FOR(i,1,N+1)
            par[k][i] = par[k-1][par[k-1][i]];
    }

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

        F0Rd(k,MAXK) if (depth[u] >= depth[v]+(1<<k))  u = par[k][u];
        F0Rd(k,MAXK) if (par[k][u] != par[k][v]) u = par[k][u], v = par[k][v];

        if(u != v) u = par[0][u], v = par[0][v];
        return u;
    }

    int dist(int u, int v) {
        return depth[u]+depth[v]-2*depth[lca(u,v)];
    }

    bool isAnc(int a, int b) {
        F0Rd(i,MAXK) if (depth[b]-depth[a] >= (1<<i)) b = par[i][b];
        return a == b;
    }

    int getAnc(int a, int b) {
        F0Rd(i,MAXK) if (depth[b]-depth[a]-1 >= (1<<i)) b = par[i][b];
        return b;
    }
};

LCA<MX> L;
Topo<MX> T;

void setDir(int x, int y) {
    if (dir[x] && dir[x] != y) finish();
    dir[x] = y;
}

void dfs0(int x) {
    for (int y: L.adj[x]) if (y != L.par[0][x]) {
        dfs0(y);
        if (x != 1 && dir[y] == -1) setDir(x,-1);
    }
}

void dfs1(int x) {
    int co = 0;
    if (dir[x] == 1) co ++;
    for (int y: L.adj[x]) if (y != L.par[0][x]) {
        if (dir[y] == -1) co ++;
    }
    for (int y: L.adj[x]) if (y != L.par[0][x]) {
        if (dir[y] == -1) co --;
        if (co) setDir(y,1);
        if (dir[y] == -1) co ++;
    }
    for (int y: L.adj[x]) if (y != L.par[0][x]) dfs1(y);
}

void genEdge() {
    dfs0(1);
    dfs1(1);
    FOR(i,2,N+1) {
        if (dir[i] == -1) {
            T.addEdge(i,L.par[0][i]);
        } else if (dir[i] == 1) {
            T.addEdge(L.par[0][i],i);
        }
    }
}

int main() {
    // you should actually read the stuff at the bottom
    setIO("gathering");
    re(N,M);
    F0R(i,N-1) {
        int a,b; re(a,b);
        L.addEdge(a,b);
    }
    L.init(N);
    F0R(i,M) {
        int a,b; re(a,b); // if you root the tree at b, then every node in the subtree corresponding to a is bad
        if (L.isAnc(a,b)) { // a is ancestor of b
            int B = L.getAnc(a,b);
            setDir(B,-1);
        } else {
            setDir(a,1);
        }
        T.addEdge(b,a);
    }
    genEdge(); T.N = N; T.sort();

    // you should actually read the stuff at the bottom
}
```

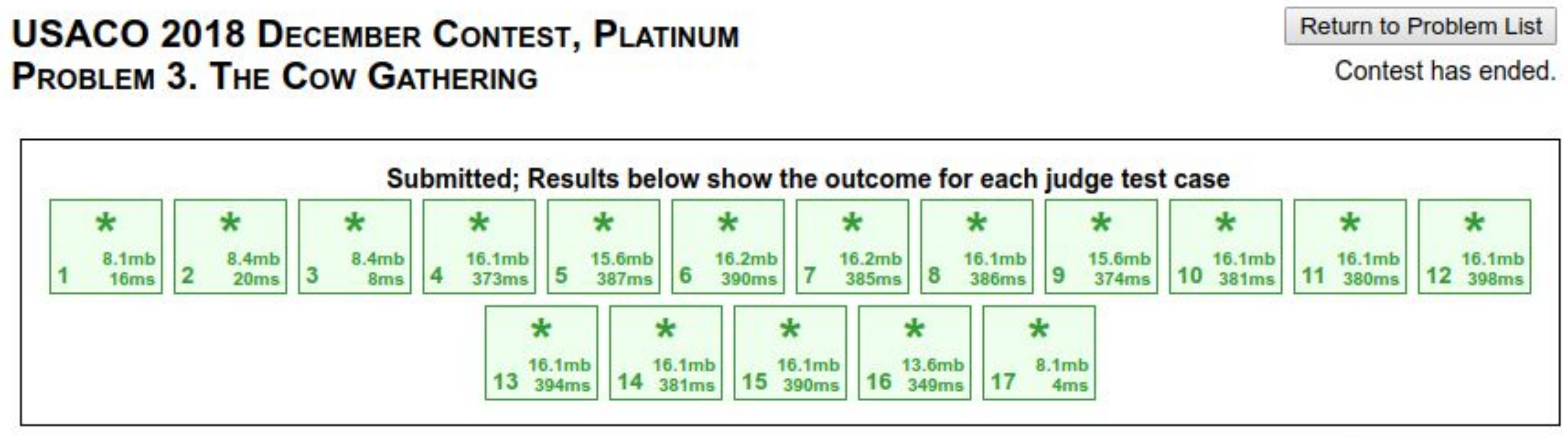
Proof that it works

Behold:



Proof that it’s better

Behold:



Questions?