Solution — Evolution

1 Modeling

We are given a list of n species $s_0, \ldots, s_{n-1}$ together with their ages $a_0, \ldots, a_{n-1}$. The species are organized in a rooted tree, which is given as a list of $n-1$ directed edges (s_i, s_j) indicating that s_i is a child of s_j. This tree satisfies the min-heap property and thus if s_i is a child of s_j, then $a_i \geq a_j$ holds. Moreover, m queries $q_0, \ldots, q_{m-1}$ are given and each query q_i is a pair (s_{q_i}, b_{q_i}) of a species and an age.

The problem asks to compute for each query q_i the oldest species which is an ancestor of s_{q_i} (i.e., it appears on the unique s_{q_i}-to-root path, denoted $P(s_{q_i})$) which is at most b_{q_i} years old. Since it is assured that $b_{q_i} \geq a_{q_i}$, such a species can always be found.

Constraints. The number of species n is at most 50 000, and the number of queries q is also at most 50 000. This suggests that in order to get 100 points, an $O(q \cdot n)$ time solution will certainly be too slow. In the first test set, we are guaranteed that n and q are at most 1 000. This means that here we can probably get away with a solution that runs in time $O(q \cdot n)$.

2 Algorithm design

$O(q \cdot n)$-solution (30 points). A solution which processes each query in $O(n)$ time is easy: for each query q_i start at the species s_{q_i} and walk in the tree towards the root (i.e., walk along the unique s_{q_i}-to-root path $P(s_{q_i})$) while the species s_j you visit satisfy the age condition $a_j \leq b_{q_i}$. The last species satisfying the condition is the answer to the query. Note that the length of $P_{s_{q_i}}$ can be $\Theta(n)$ for each query if for example the whole tree is just a path and species close to the leaf are queried whereas the answer is the root.

$O(q \cdot \log n)$-solution (100 points). Walking along $P(s_{q_i})$ in the previous solution probably felt like a waste of time, because the path is sorted and thus searching for the answer could be done using binary search in $O(\log n)$ time (see lecture 2 slides). However, in order to do binary search, we actually need to have the path explicitly stored for example in an array or a set. We observe that we can construct the path $P(s)$ from top down (i.e., starting from the root) incrementally from $P(s')$ where s' is the parent of s. Thus, while we construct $P(s)$ incrementally, we can answer all queries to ancestors of s on the fly using binary search. This observation leads to the conclusion that we have to process queries while constructing paths incrementally top down. The last thing is to find a good order for construction. However, here a depth first search (DFS) starting from the root does the trick.

In summary we get the following algorithm: start a DFS at the root of the tree. Keep the following invariant: if the search is at species s, then $P(s)$ is stored in a data structure where we

can do binary search on. If the search visits a species s which occurs in some query, compute answers to all the queries it is occurs in by doing binary search on $P(s)$.

The DFS in the tree takes $O(n)$ time and each of the q queries takes $O(\log n)$ time. Hence, the total running time is $O(n + q \cdot \log n) = O(q \cdot \log n)$.

3 Implementation

Before providing a complete implementation of an $O(q \cdot \log n)$ solution we want to discuss some implementation details.

Reading and processing string input. The species are given as strings in the input. You can use `cin` to read strings (don't forget the `#include <string>` and note that the default delimiter is a white space). In order to build the tree later, we need to map the species to indices. This can be done using a map `map<string,int>` (or `unordered_map<string,int>`, which is faster and the data structure of choice in this case, if you are interested read about it in the c++ reference). In order to map the indices back to the species for output, we use `vector<string>`. Hence, the code to read in the species and their ages looks as follows:

```
1  unordered_map<string,int> species_to_index;
2  vector<string> species(n);
3  vector<int> age(n);
4  for(int i = 0; i < n; ++i) {
5    string name; cin >> name;
6    species_to_index[name] = i;
7    species[i] = name;
8    int a; cin >> a;
9    age[i] = a;
10 }
```

Global variables or passing local variables by reference. In the $O(q \cdot \log n)$ solution we will implement a DFS and a binary search method, which for example need the tree or the ages of the species as arguments. Now you can either store these as global variables or pass them to the methods by reference. This is a design choice you have to make. Global variables usually require writing less code but are error prone if you forget to re-initialize them. We provide a solution with local variables (for an example of global variables see for example the solution of even matrices).

Data structure for the tree. In the $O(q \cdot n)$ solution we can compute the paths $P(s)$ bottom-up starting from the species s and walking towards the root. Hence, for each species we only need to know the parent and we can simply use a `vector<int> parent(n)` where `parent[i]` stores the parent (its index) of the i-th species.

In the $O(q \cdot \log n)$ solution we do a DFS starting from the root. Hence, for each species we need to know all its children. Thus, an adjacency list `vector<vector<int> > tree(n)` where `tree[i]` is a vector containing (the indices of) all the children of the i-th species does the trick. We also need to find the root, which is easy because it is the oldest species. Finding the root and constructing the tree is done in the following code snippet.

```
  int root = max_element(age.begin(), age.end()) - age.begin();

  vector<vector<int> > tree(n);
  for(int i = 0; i < n-1; ++i){
    string child; cin >> child;
    string parent; cin >> parent;
    tree[species_to_index[parent]].push_back(species_to_index[name]);
  }
```

4 Appendix

The following code is an implementation of the $O(q \cdot \log n)$ solution explained in the previous section.

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <string>
#include <utility>
#include <algorithm>

using namespace std;

// binary search
int binary(int b, vector<int>& path, vector<int>& age){
  int l = 0; int r = path.size()-1;
  while(l != r){
    int m = (l+r)/2;
    if(age[path[m]] > b) l = m+1; else r = m;
  }
  return path[l];
}

// dfs
void dfs(int u, vector<vector<int> >& tree, vector<int>& path,
  vector<vector<pair<int,int> > >& query, vector<int>& result, vector<int>& age){

  // process queries
  for(int i = 0; i < query[u].size(); ++i){
    result[query[u][i].second] = binary(query[u][i].first,path,age);
  }

  // continue search
  for(int i = 0; i < tree[u].size(); ++i){
    int v = tree[u][i];
    path.push_back(v);
    dfs(v,tree,path,query,result,age);
  }
  path.pop_back();
}

int main(){
  ios_base::sync_with_stdio(false);
  int t; cin >> t;

  // do test cases
  while(t--){
```

```
    int n, q; cin >> n >> q;

    // read names and ages
    unordered_map<string,int> species_to_index;
    vector<string> species(n);
    vector<int> age(n);
    for(int i = 0; i < n; ++i) {
      string name; cin >> name;
      species_to_index[name] = i;
      species[i] = name;
      int a; cin >> a;
    }

    // find root
    int root = max_element(age.begin(), age.end()) - age.begin();

    // read tree
    vector<vector<int> > tree(n);
    for(int i = 0; i < n-1; ++i){
      string child; cin >> child;
      string parent; cin >> parent;
          tree[species_to_index[parent]].push_back(species_to_index[child]);
    }

    // read queries; for each species store a vector of queries
    // consisting of the age b and the index of the query i
    vector<vector<pair<int,int> > > query(n);
    for(int i = 0; i < q; ++i){
      string name; cin >> name;
      int b; cin >> b;
      query[species_to_index[name]].push_back(make_pair(b,i));
    }

    // process queries in one tree traversal
    vector<int> path; path.push_back(root);
    vector<int> result(q);
    dfs(root,tree,path,query,result,age);

    // output result
    for(int i = 0; i < q; ++i){
      cout << species[result[i]];
          if(i < q-1) cout << "␣";
    }
    cout << endl;
  }
  return 0;
}
```