Solution — Evolution

1 Modeling

We are given a list of n species $s_0, \dots, s_{n-1}$ together with their ages $a_0, \dots, a_{n-1}$. 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 $a_i \leq a_j$ holds. Moreover, m queries $q_0, \dots, q_{m-1}$ are given and each query q is a pair (s_q, b_q) of a species and an age.

For each species s let $P(s)$ be the unique s-to-root path in the tree.

The problem asks to compute for each query q the oldest species that is both on $P(s_q)$ and at most b_q years old. Note that since it is assured that b_q is at least the age of s_q, such a species can always be found.

2 Algorithm Design

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

$O(m \cdot n)$-**solution (30 points)** - A solution which processes each query in $O(n)$ time is easy to derive: for each query q start at the species s_q and walk $P(s_q)$ towards the root, while the species you visit is at most b_q years old. The last species satisfying this condition is the answer to the query. Note that the length of $P(s_q)$ 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.

$O(m \cdot \log n)$-**solution (100 points)** - Walking along $P(s_q)$ in the previous solution probably felt like a waste of time, because the ages along the paths are sorted and thus searching for the answer could be done using binary search in $O(\log n)$ time (see slides from week 2). However, in order to do binary search, we actually need to have the path explicitly stored, for example in an array.

Unfortunately, explicitly storing all paths $P(s)$ for all species s requires $\Theta(n^2)$ time, which is too slow.

However, we can perform a depth first search (DFS) starting from the root that keeps the invariant of storing $P(s)$ in an array where s is the species the DFS is currently visiting. During such a modified DFS we have $P(s)$ stored for all s in an array at some point in time (because the DFS visits all vertices).

The last observation we have to make is that we do not have to process the queries in the order in which they were given. We can rather store for each species s a list of queries in which the species is contained in, this is all queries (s_q, b_q) with $s = s_q$. Given this information, we can process the queries in the order in which we visit species during the DFS.

In summary we get the following algorithm: start a DFS at the root of the tree. Keep the following invariant: if the DFS 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 occurs in by doing binary search on $P(s)$.

The DFS in the tree takes $O(n)$ time and each of the m queries takes $O(\log n)$ time. Hence, the total running time is $O(n + m \cdot \log n) = O(m \cdot \log n)$, since n and m are of the same order of magnitude.

3 Implementation

Before providing a complete implementation of an $O(m \cdot \log n)$ 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 `std::cin` to read a string (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 `std::map<std::string,int>` or `std::unordered_map<std::string,int>`. The latter is the data structure of choice here because it is faster (`std::map` is implemented with a tree, while `std::unordered_map` is a hash-map). In order to map the indices back to the species for output, we use `std::vector<std::string>`. Hence, the code to read in the species and their ages looks as follows:

```cpp
std::unordered_map<std::string,int> species_to_index;
std::vector<std::string> species(n);
std::vector<int> age(n);

for(int i = 0; i < n; ++i) {
  std::string name;
  std::cin >> name;
  species_to_index[name] = i;
  species[i] = name;

  std::cin >> age[i];
}
```

Global variables or passing local variables by reference. In the $O(m \cdot \log n)$ solution we 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 for even matrices).

Data structure for the tree. In the $O(m \cdot n)$ solution we 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 `std::vector<int> parent(n)` where `parent[i]` stores the parent (its index) of the i-th species.

In the $O(m \cdot \log n)$ solution we do a DFS starting from the root. Hence, for each species we need to know all its children. Thus, an adjacency list `std::vector<std::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.

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

  std::vector<std::vector<int> > tree(n);
  for(int i = 0; i < n-1; ++i){
    std::string child;
    std::cin >> child;

    std::string parent;
    std::cin >> parent;

    tree[species_to_index[parent]].push_back(species_to_index[name]);
  }
```

4 Appendix

The following code is an implementation of the $O(m \cdot \log n)$ solution.

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


// binary search:
// find largest index i s.t. p(i) := age[path[i]] <= b is true
int binary(int b, const std::vector<int>& path, const std::vector<int>& age){

  // establish invariant
  int l = -1; // l is largest index where we know p(l) is false
  int r = path.size() -1; // r is smallest index where we know p(r) is true

  while(l+1 < r){ // always at least one element strictly inbetween
    int m = (l+r)/2; // m!=l and m!=r so interval shrinks

    // propagate invariant
    if(age[path[m]] <= b) r = m;
    else l = m;
  }
  return path[r];
}

// dfs
void dfs(int u, const std::vector<std::vector<int> >& tree,
  std::vector<int>& path,
  const std::vector<std::vector<std::pair<int,int> > >& query,
  std::vector<int>& result,
  const std::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 dfs
  for(int i = 0; i < tree[u].size(); ++i){
    int v = tree[u][i];
    path.push_back(v); // maintain path invariant
    dfs(v,tree,path,query,result,age);
  }

  path.pop_back(); // maintain path invariant
}

int main(){
  std::ios_base::sync_with_stdio(false);
  int t; std::cin >> t;
  while(t--){
    int n, q; std::cin >> n >> q;

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

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

    // read tree
    std::vector<std::vector<int> > tree(n);
    for(int i = 0; i < n-1; ++i){
      std::string child; std::cin >> child;
      std::string parent; std::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
    std::vector<std::vector<std::pair<int,int> > > query(n);
    for(int i = 0; i < q; ++i){
      std::string name; std::cin >> name;
      int b; std::cin >> b;
      query[species_to_index[name]].push_back(std::make_pair(b,i));
    }

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

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