XenoHematology Writeup

When making my algorithm, I saw that I needed to find elements in an equivalence relation, which suggested using a Union Find. But there were additional requirements: if 2 elements were compatible, they were incompatible with the same elements, and 2 elements that were incompatible with the same element had to be compatible with each other. This meant that there were really only 2 possible blood types, it just wasn't clear which xeno had which one. To simulate this, I wrote a modified version of the Union Find, which I named Rival Union Find. In my structure, in addition to storing each node's parent, if the node is a root node, my structure also stores its "rival", which is my term for a union incompatible with this union – that is, xenos with the opposite blood type. My structure uses an array to store the index of the rival of each root node that has a rival (it is `-1` otherwise).

Each of the API methods in my implementation cleans the given inputs and calls the appropriate method(s) in `RivalUnionFind`, and so shares those methods' efficiencies: `setIncompatible()` calls `setRival()`, `setCompatible()` calls `union()`, `areCompatible()` calls `findUnion()` twice (to check if the xenos are in the same union), and `areIncompatible()` calls `findUnion()` and `findRival()` (to check if the union of one xeno is the rival of the other). The `findUnion(el1)` method is largely unchanged from Sedgewick's `find()`. I used path compression to give the method O(1) order of growth, as we discussed in lecture. The `findRival(el)` method finds a given node's union's rival, its root's rival, by finding the root and checking the rival in the array (`rival[findUnion(el)]`), giving it the same order of growth as `findUnion()`: O(1). My `union(el1, el2)` method is a modified version of Sedgewick's, finding the root of each element and merging the unions by setting the parent of one root as the parent of the other. I also might merge the rivals of each root, if necessary. Since my method only changes array values and calls `findUnion()` and `findRival(el)`, it is O(1), like those methods. My `setRival(el1, el2)` method either makes `el2`'s union into `el1`'s rival if `el1` doesn't yet have an rival by editing the array, an O(1) operation, or calls `union()` to join the union of `el2` and `el1`'s preexisting rival, an O(1) operation, making it also O(1).

Since each individual operation is O(1) irrespective of the size of the input population or number of arguments, invoking *n* operations leads to O(*n*) order of growth overall.