Elad Ohayon

The proposed time complexity of the algorithm to solve the XenoHematology problem is estimated to be O(n) with an interesting observation. This algorithm consists of a set containing the custom class, IncompatibleTree (not a tree :)), that holds 2 sets one to track compatibles and one to track incompatibles along with Sedgewick's Union Find. However, the set of IncompatibleTree isn't filled until we have data on members of the population, meaning the runtime is **dependent** on the amount of data we have, as we do more tests the run time converges to O(n). Upon initialization, the array used for union-find must be filled with numbers 0…populationSize-1, making the constructor O(n). Since Union find is implemented through a tree, all operations are O(log n) except the initialization. Going through the program method by method, the first method, setIncompatible has a runtime of O(n). This method first preforms checks which are O(1) for valid input, and O(logn) to determine whether xeno1 and xeno2 were previously set compatible through union find. Next, this method loops through the set of IncompatibleTree to determine whether we already have data on xeno1 and xeno2 which behaves at some fraction *O(n) where with more data, the ratio converges to 1/2 as we must look at more IncompatibleTree's to locate xeno1 and xeno2, but 0.5N pairs at most. If we have previous data on xeno1 and xeno2 the time complexity will be O(n) with respect to the number of elements in each set since we merge IncompatibleTrees, else the complexity is O(1) since simple set.add() is done. This program behaves at starting O(1) (only if we have no data), jumps to some fraction * O(n) and converges to O(n), but since we don't care about constants the runtime is simply O(n).

AreIncompatible() has the exact same behvaior as setInompatible. The program checks the input for O(1), does a O(logn) check if the two numbers are compatible, and lastly loops through every instance of IncompatibleTree in the set to determine whether the 2 numbers are in the same IncompatibleTree meaning they are incompatible. Moving onto setCompatible, this function also has a runtime complexity of O(1) (no data) jumps to fraction *O(n) and converges to O(n) as it must check if a pair was previously declared incompatible to prevent setting an incompatible pair compatible which is O(n) at most, union the two values together which is O(log n). Lastly, areCompatible modifies the proper instance of IncompatibleTree to set xeno1's incompatibles equal to xeno2's Incompatibles, merging if necessary. This has the exact same logic as setIncompatile just that we change the set for compatibles instead making the runtime vary from fraction * O(n) to O(n) where we drop the constant.

Lastly, the method areCompatible, is the only one that runs in O(log n) time as it merely runs O(1) checks to determine if the inputs are valid and then connected xeno1 and xeno2 to determine compatibility which is a O(log n) operation since it is implemented through a tree. Overall, the time complexity of this algorithm is O(n) since all its methods converge to O(n) areCompatible. However, **the cost to setup the algorithm is O(n)** for the union find and constants are dropped.