XenoHematology: "Efficient Blood Testing In An Alien Population"

Avraham Leff

October 27, 2022

1 Assignment-Specific Packaging

The general packaging is unchanged from the basic "Homework Requirements" (see slides from first lecture and "Homework Policies for COM 2545" on Piazza).

This assignment's "DIR" **must be named** *XenoHematology*, and your application's Java class **must be named** *XenoHematology.java*. Your write-up file **must be named** *XenoHematology.pdf*.

2 Motivation

This assignment is intended to give you experience applying (and extending) concepts learned in lecture, and more practice coding an algorithm and explaining its order of growth.

3 Background

You're working for the renowned *Inter-Planetary* Xenobiology department, and they need you to help solve a newly discovered xeno hematology problem.

Members of the xeno population need to be able donate blood to one another, but only some members of the population are *compatible* with one another. When a xeno donates blood to an *incompatible* xeno, the result is a very painful death. Thus, there is an urgent need for a inexpensive

way to determine whether two members of the xeno population are compatible or incompatible with one another. Unfortunately, the current test for "compatibility" is very expensive.

There is a glimmer of hope in this situation. Other xeno biologists have discovered some interesting facts about "compatibility".

- Compatibility is an equivalence relation.
- Incompatibility, in contrast, is not an equivalence relation. Although it's symmetric, it's neither transitive nor reflexive.

In addition to the above facts, the following facts may enable ways to greatly reduce the cost of testing for compatibility.

- If x is incompatible with y, and y is incompatible with z, then x is compatible with z. Because of the equivalence relation mentioned above, this fact has implications beyond a single x, y, z triple.
- If x is compatible with y and y is incompatible with z, then x is incompatible with z. Because of the equivalence relation mentioned above, this fact has implications beyond a single x, y, z triple.

Thus, by doing the expensive "compatibility" tests on some xeno members of the population x, y, z, we **may avoid the need to do a pair-wise "compatibility" test on all members of the population**! Your team will do random sampling of the population, administering the "compatibility" test on different xeno pairs. Depending on the test result on a pair (x, y), you either record `setCompatible(x, y)` or `setIncompatible(x, y)`.

Once a pair is determined to be "compatible" with one another, we know that they are not incompatible. Conversely, once a pair is determined to be "incompatible" with one another, we know that they are not compatible. Your program's API includes `areCompatible(x, y)` and `areIncompatible(x, y)` which returns the appropriate result based on the result of a previously administered test. In general, prior to administering the test, however, both `areCompatible` and `areIncompatible` **return** `false` **to reflect the fact we have no "compatibility" information about the specified pair**. However: if test results from x, y and y, x allow us to apply the facts discussed above, then there is **no need for pair-wise tests between** x, y, z in order to definitively answer whether they are "compatible" or "incompatible".

One more important, but conceptually dubious, assumption. Once `setCompatible(x, y)` or `setIncompatible(x, y)` has been invoked (i.e.,

you received a test result for this pair), **the results of subsequent tests are ignored** – even if they contradict that first result. The first result is always dispositive.

3.1 Observation

After thinking through this "background information", it may seem that we've discussed this problem in lecture. Before coding up your solution, be sure that you've fully thought this through ☺.

4 Requirements

> The performance requirements for this assignment are harder than usual. Say that we label the *size of the problem's xeno population* as n, and label the *number of the XenoHematology API invocations* as m.
>
> Then the implementation must exhibit at worst **linear** order of growth as a function of a problem size that's characterized by invoking m operations on a population of size n where we set $n = m$. In addition, it must scale nicely to (at least) $n = m = 2^{27}$.

> Aside from using the JDK, for this assignment, you may **only use code written by yourself or copied from the textbook**!

The assignment consists of both "programming" and "non-programming" components.

4.1 Non-Programming

This portion of the assignment is worth 20%.

Your writeup must contain, in this order:

1. In no more than three paragraphs, explain how your implementation meets the required order-of-growth.

Your explanation must sketch your algorithm's design in sufficient detail that your "Big-O" claims follows naturally. Don't give me the code

since I can see that for myself! Pseudo-code is the right level of detail. If appropriate, consider incorporating a diagram into your discussion.

This component will be graded on your *"communication skills"*: clarity, correctness, succinctness, and "convincingness".

You may "explain by reference" to material covered in lecture or textbook or to "well-known" cs-knowledge. If you choose to use a "reference to covered material", you **must offer a convincing explanation** that this material is relevant to solving our problem.

4.2 Programming

This portion of the assignment is worth 80%, and will be graded for "correctness" and ability to meet the "Big-O" requirements at scale.

> Please review the general requirements for a programming assignment! I've tried to reduce the chances of "mistakes" occurring through the use of a "skeleton interface", but ultimately, **you are responsible** for ensuring that I can compile and test your code without incident.

1. Begin by downloading a skeleton `XenoHematology.java` class from this git repository.
2. Then, implement the stubbed methods of *XenoHematology*.