Programming Assignment #1

Some Preliminaries

Programming assignments are to be done individually. Do not make your code publicly available (such as a Github repo) as this enables others to cheat and you will be held responsible. You may discuss the problem and general concepts with other students, but there should be no sharing of code. You may not submit code other than that which you write yourself or is provided with the assignment. This restriction specifically prohibits downloading code from the Internet. If any code you submit is in violation of this policy, you will receive no credit for the entire assignment.

This programming assignment is due **Monday, February 6th at 11:59 PM.** If you are unable to complete the assignment by this time, you may submit the assignment late until Wednesday, February 8th at 11:59 PM for a 20-point penalty.

The Goals

The goals of this lab are:

- Familiarize yourself with programming in Java
- Place the stable matching algorithm in the context of engineering solutions with high social impact
- Show an application of the stable matching problem
- Understand the difference between the two optimal stable matchings

The Context: Engineering Solutions with High Social Impact

Before we get started, please watch the following video on Engineering Solutions with Social Impact:
`https://www.youtube.com/watch?v=UCDwNWSXFHk`
You might want to jot down some notes to use in the next section.

The Problem: Stable Matching in Organ Transplant

The United Network for Organ Sharing (UNOS) works to help provide organs to needy patients. Let's start by reading a little bit about this problem space, in general, and its relationship to stable matching, specifically:
`https://mashable.com/archive/big-data-organ-transplants`

** Part 1: Report: Background ** [20 points]

The first section of your report will develop your background knowledge and thinking related to this problem. You should construct this portion of your report in three pieces. Assume the reader of your report is an educated outsider – for instance a roommate or other student at UT who may be an engineering or science major but not an expert in algorithms or computing. Each "piece" below can be brief (just a sentence or two or a short paragraph).

(a) For the first piece, describe the organ donation problem *in your own words*. Write this in a way a well informed outsider can comprehend (hint: run it by such a well-informed outsider and ask them for a critique).

(b) For the second piece, explain how algorithms can be used to achieve social impact in the context of the organ donation problem. Here, it might make sense to explicitly relate the problem to the stable matching problem (and potential algorithms).

(c) For the third and final piece, explain how issues of fairness and / or equity might arise in this solution and your thoughts about how stable matching algorithms might be used to address these issues.

In this programming assignment, we will work on a simplified version of this problem. In particular, you are working on an algorithm for UNOS to match organ donors who have volunteered to hospitals that have patients who need transplants. The hospitals rank the donors by the probability that the transplant works with their patients. However, the donors rank the hospitals based on how far they are willing to travel to donate to the hospital. Your job is to devise and implement an algorithm to automate this process based on the Gale Shapley algorithm presented in class.

There are n organ donors, each interested in donating to one of m hospitals. Each hospital has a set number of transplants they need to help their patients, which can vary between hospitals.

Every organ donor submits a preference list that ranks all m hospitals, and every hospital's clinical board creates a preference list of all n organ donors based on the organ donor's potential to match with the hospital's patients. We will assume that there are at least as many organ donors as the total openings available across all m hospitals.

This means that all needs for kidneys will be filled, but some organ donors may be left unmatched to a hospital. The interest lies in finding a way of assigning each organ donor to at most one hospital in such a way that all available openings are filled.

An assignment of donors to hospitals is *stable* if neither of the following situations arises:

- First type of instability: There are organ donors i and i', and a hospital c, such that
  - i is assigned to c, and
  - i' is assigned to no hospital, and
  - c prefers i' to i
- Second type of instability: There are organ donors i and i', and hospital c and c', so that
  - i is assigned to c, and
  - i' is assigned to c', and
  - c prefers i' to i, and
  - i' prefers c to c'.

As a result, we basically have the Stable Matching Problem as presented in class, except that (i) a hospital may want one or more organ donors, and (ii) there is potentially a surplus of organ donors. There are several parts to this problem.

**** Part 2: Report: Algorithm Design **** [20 points]
Continuing the report you started above, add a section on algorithm design that addresses each of the following:

(a) Give an algorithm in pseudocode (either an outline or paragraph works) to find a stable assignment that is **hospital** optimal.

(b) Give the runtime complexity of your algorithm in (a) in Big O notation and explain why. **Note: Full credit will be given to solutions that have a complexity of $O(mn)$.**

(c) Give an algorithm in pseudocode (either an outline or paragraph works) to find a stable assignment that is **donor** optimal.

(d) Give the runtime complexity of your algorithm in (c) in Big O notation and explain why. **Note: Try to make your algorithm as efficient as you can, but you will get full credit even if it does not match the runtime in (b) as long as you clearly explain your runtime and the difficulty of optimizing it further.**

For the programming assignment, you do *not* need to submit a proof that your algorithm returns a stable matching, or of donor/hospital optimality.

**** Part 3: Implementation: Check Stability of a Given Matching **** [15 points]
Given a Matching object `problem`, you should implement a boolean function to determine if the pairing of organ donor to hospitals (stored in the variable returned by `problem.getdonorMatching()`) is stable or not. Your code will go inside a function called `isStableMatching(Matching problem)` inside `Program1.java`. A file named `Matching.java` contains the data structure for a matching. Note that you do not need to optimize the runtime of this function, a brute force approach is sufficient. See the instructions section for more information on how to test this method.

**** Part 4: Implementation: Gale Shapley Algorithm **** [45 points]
Implement both algorithms from parts (a) (hospital optimal) and (c) (donor optimal) of your report. Again, you are provided several files to work with. Implement the function that yields a donor optimal solution `stableMatchingGaleShapley_donoroptimal()` and hospital optimal solution `stableMatchingGaleShapley_hospitaloptimal()` inside of `Program1.java`.

Of the files we have provided, please only modify `Problem1.java`, so that your solution remains compatible with ours. However, feel free to add any additional Java files (of your own authorship) as you see fit.

Instructions

- Download and import the code into your favorite development environment. We will be grading in Java 1.8. Therefore, we recommend you use Java 1.8 and NOT other versions of

Java, as we can not guarantee that other versions of Java will be compatible with our grading scripts. **It is YOUR responsibility to ensure that your solution compiles with Java 1.8.** If you have doubts, email a TA or post your question on Piazza.

- If you do not know how to download Java or are having trouble choosing and running an IDE, email a TA, post your question on Piazza, or visit the TAs during Office Hours.
- **Do not add any package statements to your code.** Some IDEs will make a new package for you automatically. If your IDE does this, make sure that you remove the package statements from your source files before turning in the assignment.
- There are several `.java` files, but you only need to make modifications to `Program1.java`. **Do not modify the other files.** However, you may add additional source files in your solution if you so desire. **Do not add extra imports to** `Program1.java`; the included imports should be all you need for your solution. There is a lot of starter code; carefully study the code provided for you, and ensure that you understand it before starting to code your solution. The set of provided files should compile and run successfully before you modify them.
- The main data structure for a matching is defined and documented in `Matching.java`. A Matching object includes:
  - **m**: Number of hospitals
  - **n**: Number of organ donors
  - **hospital_preference**: An ArrayList of ArrayLists containing each of the hospital's preferences of organ donors, in order from most preferred to least preferred. The hospitals are in order from 0 to $m-1$. Each hospital has an ArrayList that ranks its preferences of organ donors who are identified by numbers 0 through $n-1$.
  - **donor_preference**: An ArrayList of ArrayLists containing each of the donor's preferences for hospitals, in order from most preferred to least preferred. The organ donors are in order from 0 to $n-1$. Each organ donor has an ArrayList that ranks its preferences of hospitals that are identified by numbers 0 to $m-1$.
  - **hospital_openings**: An ArrayList that specifies how many openings each hospital has. The index of the value corresponds to which hospital it represents.
  - **donor_matching**: An ArrayList to hold the final matching. This ArrayList (should) hold the number of the hospital each donor is assigned to. This field will be empty in the `Matching` which is passed to your functions. The results of your algorithm should be stored in this field either by calling `setdonorMatching(<your_solution>)` or constructing a `new Matching(data, <your_solution>)`, where `data` is the Matching we pass into the function. The index of this ArrayList corresponds to each donor. The value at that index indicates to which hospital they are matched. A value of -1 at that index indicates that the donor is not matched up. For example, if donor 0 is matched to hospital 55, donor 1 is unmatched, and donor 2 is matched to hospital 3, the ArrayList should contain {55, -1, 3}. If using the flag [-s], an input with an existing matching can be given to check the correctness of the `isStableMatching()` function.

- You must implement the methods
  - `isStableMatching()`
  - `stableMatchingGaleShapley_donoroptimal()`
  - `stableMatchingGaleShapley_hospitaloptimal()`

  in the file `Program1.java`. You may add methods to this file if you feel it necessary or useful. You may add additional source files if you so desire.

- Test cases take the format of text files, which either have the file extension of `.in` or `.extended.in`. Here's how to interpret each test case, line by line:
  - Line 1: `m n`
  - Line 2: `m` space separated integers, denoting the number of openings available in each hospital. The first integer represents the number of open openings in hospital 0, the next integer represents the number for hospital 1, and so on.
  - The next `m` lines are the preference lists of the hospitals, where each space-separated integer represents an organ donor. The list goes from left to right, from most to least desirable. The first of these `m` lines is the preference list for hospital 0, the next line is for hospital 1, and so on.
  - The next `n` lines are the preference lists of the organ donors, where each space-separated integer represents a hospital. The list goes from left to right, from most to least desirable. The first of these `n` lines is the preference list for donor 0, the next line is for donor 1, and so on.
  - Last line (optional): `n` space separated integers representing a donor-hospital matching. If the first integer is x, then the first donor is assigned to hospital x, the second donor to the second integer, and so on. This is a way of hard coding in a matching to test your implementation of `isStableMatching()` in Part 2 before you complete Part 3. To see examples, see the last lines of the test cases with the file extension `.extended.in`.

- `Driver.java` is the main driver program. Use command line arguments to choose between your checker and your hospital optimal or donor optimal algorithms and to specify an input file. Use -h for hospital optimal, -d for donor optimal, and -s for importing an existing matching (to check correctness of `isStableMatching()`). (i.e. `java -classpath . Driver [-h] [-d] [-s] <filename>` on a linux machine). As a test, the 3-10-3.in input file should output the following for both a donor and hospital optimal solution:
  - donor 0 hospital -1
  - donor 1 hospital 1
  - donor 2 hospital -1
  - donor 3 hospital -1
  - donor 4 hospital -1
  - donor 5 hospital -1
  - donor 6 hospital -1

  - donor 7 hospital 2
  - donor 8 hospital 0
  - donor 9 hospital -1

- When you run `Driver.java`, it will tell you if the results of your algorithm(s) pass the `isStableMatching()` function that *you coded* for this particular set of data. When we grade your program, however, we will use *our* implementation of `isStableMatching()` to verify the correctness of your solutions.
- Make sure your program compiles on the LRC machines before you submit it.
- We will be checking programming style. A penalty of up to 10 points will be given for poor programming practices (e.g. do not name your variables foo1, foo2, int1, and int2).

Getting Started:

(a) Download the starter material from canvas.

(b) Do an initial compile of the starter code in your favorite Java IDE or on the ECE LRC Linux machines(reccomended), see note below on how to compile.

(c) Test your code using the inputs given and the Driver class.

(d) Submit your code to Gradescope to validate your code. You have unlimited submissions before the due date. Your latest submission will be the only submission we look at.

NOTE: To avoid receiving a 0 for the coded portion of this assignment, you MUST ensure that your code correctly compiles **with the original, unmodified starter files** on Java 1.8. Do not modify the signatures of or remove existing methods of `Program1.java`. Do not add package statements. Do not add extra imports. We recommend testing compilation of your code using the ECE LRC Linux machines (using “javac *.java” and “java Driver inputfile.in”) after redownloading the starter files from Canvas.

What To Submit

You should submit to Gradescope `Program1.java` (and, any extra `.java` files you added or modified). Please do not submit `Driver.java`. Failure to follow these instructions will result in a penalty of up to 10 points.

Your PDF report should be legibly scanned and submitted to Gradescope. Both your code and PDF report must be submitted by 11:59 PM on Monday, February 6, 2023. If you are unable to complete the assignment by this time, you may submit the assignment late until Wednesday, February 8, 2023 at 11:59 PM for a 20 point penalty.