Programming Assignment #1

Programming assignments are to be done individually. 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.

The goals of this lab are:

- Familiarize you with programming in Java
- Show an application of the stable matching problem
- Understand the difference between the two optimal stable matchings.

Problem Description

In this project, you will implement a variation of the stable marriage problem adapted from the textbook Chapter 1, Exercise 4, and write a small report. We have provided Java code skeletons that you will fill in with your own solution. Please read through this document and the documentation in the starter code thoroughly before beginning.

The situation is the following: There are n students aiming to secure one of m Summer 2020 internship opportunities. Each student has a ranking of the internships in order of preference. Each company prefers students based on their GPA, number of projects, and months of experience. However, each company will weigh these factors differently. There may be more students seeking internships than there are internship positions. We aim to assign each student to at most one internship, in such a way that all available internships are filled. (Since there may be a surplus of students, there would be some students who do not get an internship.)

We say that an assignment of students to internships is *stable* if neither of the following situations arises:

- First type of instability: There are students s and s', and an internship I, such that
  - s is assigned to I, and
  - s' is assigned to no internship, and
  - I prefers s' to s
- Second type of instability: There are students s and s', and internships I and I', so that
  - s is assigned to I, and
  - s' is assigned to I', and
  - I prefers s' to s, and

– s' prefers I to I'.

So we have a modified version of the Stable Matching Problem as presented in class, where (i) companies generally want more than one student, and (ii) there is a shortage of internships. There are several parts to the problem.

Part 1: Write a report [20 points]
Write a short report that includes the following information:

(a) Give an algorithm in pseudocode (either an outline or paragraph works) to find a stable assignment that is **student** optimal. *Hint: it should be very similar to the Gale-Shapley algorithm, with students taking the role of the men, and internships of the women.*

(b) Give the runtime complexity of your algorithm in (a), in terms of m and n, in Big O notation and explain why. **Note : Try to make your algorithm as efficient as you can, but you may get full credit even if it is not O(mn) as long as you clearly explain your running time and the difficulty of optimizing it further.**

Note : For the programming assignment, you don't need to submit a proof that your algorithm returns a stable matching, or of internship or student optimality.

In the following two sections you will complete code for brute force and optimized methods to find a stable solution. Each internship prefers students based on a weighted sum of a student's GPA, months of experience, and number of projects differently. The sum of each of these weights is equal to 100. Each student has these attributes. For any internship (i) deciding which student (s) they prefer, they calculate the value:

- Obj(i, s) = $s_{GPA} * i_{weightGPA} + s_{monthsExp} * i_{weightExp} + s_{numProjects} * i_{weightProjects}$

You may choose to implement the computeInternshipPreferences() function if it is useful for your design.

Part 2: Finish the Brute Force Solution [20 points]
A brute force solution to this problem involves generating all possible permutations of students and internships, and checking whether each one is a stable matching. **Your job is to complete the function to check whether a matching is stable.** To find the stable matching that is student optimal, you need to find the matching where every student gets their best valid partner. For this part of the assignment, you are to implement a function that verifies whether or not a given matching is stable and find the student optimal one. We have provided most of the brute force solution already, including input, function skeletons, abstract classes, and a data structure. Inside `Program1.java` is a placeholder for a verify function called `isStableMatching()`. Implement this function to complete the Brute Force student Optimal algorithm (Note: a brute force algorithm is already provided, but the provided function returns the first possible matching. Once you complete the isStableMatching function, it will then return the first stable matching). A file named `Matching.java` contains the data structure for a matching. See the instructions section for more information.

Of the files we have provided, please only modify `Program1.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.

Part 3: Implement an Efficient Algorithm [60 points]
We can do much better than the brute force solution by using a modified version of the Gale-Shapley algorithm. Implement the efficient algorithm you devised in your report for student optimal solution. Again, you are provided several files to work with. Implement the function that yields a student optimal solution, `stableMarriageGaleShapley_studentoptimal()`, inside of `Program1.java`.
Of the files we have provided, please only modify `Program1.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 preferred 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.
- 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. 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**: The number of internships
  - **n**: Number of students
  - **internship_preference**: An ArrayList of ArrayLists to hold each of the internship's preferences of students, in order from most preferred to least preferred. The internships are in order from 0 to $m-1$. Each internship has an ArrayList that ranks its preferences of students who are identified by numbers 0 through $n-1$. *Using this field is optional.*
  - **student_preference**: An ArrayList of ArrayLists containing each of the student's preferences for internships, in order from most preferred to least preferred. The students are in order from 0 to $n-1$. Each student has an ArrayList that ranks its preferences of internships who are identified by numbers 0 to $m-1$.
  - **internship_slots**: An ArrayList that specifies how many slots each internship has. The index of the value corresponds to which internship it represents.

  - **student_matching**: An ArrayList to hold the final matching. This ArrayList (should) hold the number of the internship each student 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 `setstudentMatching(<your_solution>)` or constructing a `new Matching(marriage, <your_solution>)`, where `marriage` is the Matching we pass into the function. The index of this ArrayList corresponds to each student. The value at that index indicates to which internship he/she is matched. A value of -1 at that index indicates that the student is not matched up. For example, if student 0 is matched to internship 55, student 1 is unmatched, and student 2 is matched to internship 3, the ArrayList should contain {55, -1, 3}.

- You must implement the methods `isStableMatching()` and `stableMarriageGaleShapley_studentoptimal()` 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.

- The file `Permutation.java` is provided to help you generate matchings for the brute force solution. You should only have to use `getNextMatching(Matching data)` in your solution. See the function `stableMarriageBruteForce(Matching marriage)` in `AbstractProgram1.java`.

- `Driver.java` is the main driver program. Use command line arguments to choose between brute force and your efficient algorithms and to specify an input file. Use -b for brute force and -g for the efficient student optimal algorithm, and input file name for specified input (i.e. `java -classpath . Driver [-g] [-b] <filename>` on a linux machine).

- When you run the driver, it will tell you if the results of your efficient algorithm 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).

- Before you submit, be sure to turn your report into a PDF and name your PDF file `eid_lastname_firstname.pdf`.

What To Submit

You should submit a single zip file titled `eid_lastname_firstname.zip` that contains all of your java files and pdf report `eid_lastname_firstname.pdf`. Do not put these files in a folder before you zip them (i.e. the files should be in the root of the ZIP archive). Your solution must be submitted via Canvas BY 11:59 pm on September 24, 2019. If you are unable to complete the lab by this time you may submit the lab late until September 27th at 11:59PM for a 20 point penalty.