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
- Compare the run time of an efficient algorithm against an inefficient algorithm
- Show an application of the stable matching problem

Problem Description

In this project, you will implement a variation of the stable marriage problem 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.

Gale and Shapley published their paper on the Stable Matching Problem in 1962. However, we now have a new problem with a different twist.

The situation is the following: There are n UT ECE students who all have a big software lab programming assignment due the next morning! The lab requires them to run certain jobs on m university's servers. One server can run more than one user's jobs. However, each server has a maximum number of jobs it can run. Since users are scattered around campus and are completing the assignment at different times, the requests for the jobs are sent at different times and at varying distances to each of the servers. Users simply want the server closest to them (ex. If Alice is 3 miles from Server A and 1 mile from Server B, she will prefer Server B to Server A). The servers will create a preference list for the users using the following scheme. Each server computes the following value for each user:

$Pref$(Server A, User 1) = (30(# of User 1 jobs) + 70(Distance between User 1 and Server A))

If two users have the same $Pref$ value, then the tie will be broken by the server choosing whichever user sent their job-completion request first. (ex. If Alice and Bob both have a $Pref$ value of 5 with respect to Server A but Alice sent her request first, then Server A will prefer Alice to Bob.) We also have that no two users send a request at the exact same time.

We will assume that the total number of jobs that can be run on the m servers is less than or equal to the number of users. The interest is in finding a way of assigning each user to at most one server, in such a way that all available slots for jobs on the server are filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital).
We say that an assignment of users to servers is *stable* if neither of the following situations arises:

- First type of instability: There are users u and u', and a server s, such that
  - u is assigned to s, and
  - u' is assigned to no server, and
  - s prefers u' to u
- Second type of instability: There are users u and u', and servers s and s', such that
  - u is assigned to s, and
  - u' is assigned to s', and
  - s prefers u' to u, and
  - u' prefers s to s'.

So we basically have the Stable Matching Problem as presented in class, except that (i) servers generally want more than one user, and (ii) there is a surplus of ECE users. There are several parts to the problem.

Part 1: Write a report [30 points]
Write a short report that includes the following information:

(a) Show that there is always a stable assignment of users to servers.

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

(c) Give the runtime complexity of your algorithm in Big O notation and explain why.

(d) Give a proof of your algorithm's correctness. Remember that you must prove both that your algorithm terminates and gives a correct result.

(e) Consider a Brute Force Implementation of the algorithm where you find all combinations of possible matchings and verify if they are a stable marriage one by one. Give the runtime complexity of this brute force algorithm in Big O notation and explain why.

(f) In the following two sections you will implement code for a brute force solution and an efficient solution. In your report, use the provided data files to plot the number of servers (x-axis) against the time in ms it takes for your code to run (y-axis). There are four small data files and four large data files included in the input provided. The large data files may be too large for the brute force algorithm to finish running on your machine. If that is the case, do not worry about plotting the brute force results for the large data files. Your plot should therefore contain 8 points from your efficient algorithm and 4-8 points from the brute force algorithm.

Please make sure the points from different algorithms are distinct so that you can easily compare the runtimes from the brute force algorithm and your efficient algorithm. Scale the plot so that the comparisons are easy to make (we recommend a logarithmic scaling). Also take note of the trend in run time as the number of servers increases.

Part 2: Implement a Brute Force Solution [30 points]
A brute force solution to this problem involves generating all possible permutations of users and servers, and checking whether each one is a stable matching, until a stable matching is found. For this part of the assignment, you are to implement a function that verifies whether or not a given matching is stable. We have provided most of the brute force solution already, including input, function skeletons, abstract classes, and a data structure. Your code will go inside a skeleton file called `Program1.java`. A file named `Matching.java` contains the data structure for a matching. See the instructions section for more information.
Inside `Program1.java` is a placeholder for a verify function called `isStableMatching()`. Implement this function to complete the Brute Force algorithm (the rest of the code for the brute force algorithm is already provided).
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.

Part 3: Implement an Efficient Algorithm [40 points]
We can do better than the above using an algorithm similar to the Gale-Shapley algorithm. Implement the efficient algorithm you devised in your report. Again, you are provided several files to work with. Implement the function `stableMarriageGaleShapley()` 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

- If you do not know how to download Java or are having trouble choosing and running an IDE, there is a post on Piazza with applicable information.
- 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 servers
  - **n**: Number of users

  - **server_preference**: An ArrayList of ArrayLists containing each of the server's preferences of users, in order from most preferred to least preferred. The servers are in order from 0 to $m-1$. Each server has an ArrayList that ranks its preferences of users who are identified by numbers 0 through $n-1$.
  - **user_preference**: An ArrayList of ArrayLists containing each of the user's preferences for servers, in order from most preferred to least preferred. The users are in order from 0 to $n-1$. Each user has an ArrayList that ranks its preferences of servers who are identified by numbers 0 to $m-1$.
  - **servers_slots**: An ArrayList that specifies how many slots each server has. The index of the value corresponds to which server it represents.
  - **user_matching**: An ArrayList to hold the final matching. This ArrayList (should) hold the number of the server each user 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 `setUserMatching(<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 user. The value at that index indicates to which server he/she is matched. A value of -1 at that index indicates that the user is not matched up. For example, if user 0 is matched to server 55, user 1 is unmatched, and user 2 is matched to server 3, the ArrayList should contain {55, -1, 3}.

- You must implement the methods `isStableMaching()` and `stableMarriageGaleShapley()` 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.

- `Driver.java` is the main driver program. Use command line arguments to choose between brute force and your efficient algorithm and to specify an input file. Use -b for brute force, -g for the efficient algorithm, and input file name for specified input (i.e. `java -classpath . Driver [-g] [-b] <filename>` on a linux machine). As a test, the 4-8-5.in input file should output:
  - User 0 Server -1
  - User 1 Server 3
  - User 2 Server 0
  - User 3 Server 1
  - User 4 Server 2
  - User 5 Server -1
  - User 6 Server -1
  - User 7 Server 0

- 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 `verify()` 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).
- If you are unsure how to do the plot in the report, we recommend the following resources. If you are on linux: `http://www.gnuplot.info/`. If you are using Windows: `http://www.online-tech-tips.com/ms-office-tips/excel-tutorial-how-to-make-a-simple-graph-or-chart-in-excel/`.
- We suggest using System.nanoTime() to calculate your runtime.
- 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 BEFORE 12:59 pm on February 19, 2019.