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:

- Get familiar 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 adapted from the textbook Chapter 1, Exercise 5, 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.

In this problem we will consider the following situation: There are n apartments being rented by their landlords to n prospective tenants. A landlord can own more than one apartment. Each landlord ranks tenants according to their own criteria of credit rating, education, rental history, etc. Similarly each tenant ranks apartments according to their respective criteria of location, price, year of construction, etc. Landlords can be indifferent among certain tenants and similarly tenants can be indifferent among apartments. For example (with $n = 4$), a tenant T could say that apartment $A1$ is ranked in first place; second place is a tie between apartment $A2$ and apartment $A3$ (no preference between them) and apartment $A4$ is in last place. We will say that T prefers apartment A to apartment A' if A is ranked higher than A' on his/her preference list i.e. they are not tied. We say that an assignment of tenants to apartments (a matching), S, is weakly stable unless there exists a tenant T and an apartment A such that each of T and (the landlord owning) A strictly prefers the other to their partner in S. In other words, **a match is weakly stable if there are entities who are indifferent to whether they are paired with their current partners, or with each other, but not weakly stable if they prefer each other.** For example:
Let's say we have 2 landlords *L1, L2* each owning one apartment *A1* and *A2* respectively and 2 tenants *T1,T2*
L1 has no preference; *L2* has no preference; *T1* prefers A*2* over A*1*; *T2* has no preference;
If A*1* is matched to *T1*; A*2* is matched to *T2*; then this is weakly stable.
However, if *L2* prefers *T2* over *T1*, then the matching isn't weakly stable anymore since *T1* and *A2* strictly prefer each other than their current pairing (so it is unstable now)

So we basically have the Stable Matching Problem as presented in class, except that preference lists may rank multiple people with the same priority, and with a slightly different definition of stability.

Part 1: Write a report [30 points]

Write a short report that includes the following information:
(a) Prove that there always exists a perfect matching that is weakly stable.
(b) Give an algorithm in pseudocode (either an outline or paragraph works) to find a stable assignment. Hint: it should be very similar to the Gale-Shapley Algorithm

(c) Give a proof of your algorithm's correctness. Remember that you must prove both that your algorithm terminates and gives a correct result.
(d) Give the runtime complexity of your algorithm in Big-O notation and explain why your proposal accurately captures the runtime of your algorithm.
(e) Consider a Brute Force Implementation of the algorithm where you find all combinations of possible matchings and verify whether they form a weakly stable match 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 couples (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 apartments increases.

Part 2: Implement a Brute Force Solution [30 points]

A brute force solution to this problem involves generating all possible permutations of tenants and apartments, 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 `stableMatchingGaleShapley()` inside of `Program1.java`. Of the files we have provided, please only modify **`Program1.java`**, so that your solution remains compatible with the grading program. 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 1.7. **It is YOUR responsibility to ensure that your solution compiles with the standard Java 1.7 configuration (JDK 7)**. For most (if not all) students this should not be a problem. The LRC machines have Java 1.7 installed. You can check the version of Java on any UNIX machine by running the command "`java –version`". If you have doubts, email a TA or post your question on Piazza.

- If you do not know how to install Java or are having trouble choosing and running an IDE, post on Piazza for additional assistance or come speak to a TA

- There are several `.java` files, but you only need to make modifications to `Program1.java`. You can modify `Driver.java` if it helps you to test or debug your program, however we will not grade the contents of `Driver.java`. Do not modify the other files we have given you. However, you may add additional source files to your solution if you so desire (just be sure to submit them all). 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. If not, there is probably a problem with your Java configuration.

- The main data structure for a matching is defined and documented in `Matching.java`. A Matching object includes:
  – **l**: The number of landlords.
  – **n**: The number of tenants (= the number of apartments)
  – landlord_own: An ArrayList of ArrayLists containing each of the *l* landlords' ownership of apartments. The apartments are listed in order from 0 to *n-1* in these lists, and each landlord can have from 1 to *n* apartments. For example, landlord l_0 can own 3 apartments a_2, a_3 and a_4, landlord l_1 can own 2 apartments a_0 and a_1 while landlord l_2 can own a single apartment a_5. The landlords are in order from 0 to *l-1* in **landlord_own**
  – landlord_pref: An ArrayList of ArrayLists containing each of the landlord's preferences for tenants. The tenants are listed in order from *0* to $n-1$ in these lists. The format of the list is the same as above. The landlords are in order from *0* to *l-1* in **landlord_pref**
  – **tenant_pref**: An ArrayList of ArrayLists containing each of the tenant's preferences for apartments. The apartments are listed in order from *0* to *n-1* in these lists, and the numbers stored in the list represent the priority of how the tenant would prefer the apartments. For example, if two apartments a_1 and a_2 are tied for first priority, and a_0 has second priority, then the preference list will read: {2,1,1}. The tenants are in order from *0* to $n-1$ in **tenant_pref**.
  – tenant_matching: An ArrayList to hold the final matching. This ArrayList will hold the index of the apartment each tenant is assigned to (in order of the tenants). 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 `setTenantMatching(<your solution>)` or constructing a new `Matching(match, <your solution>)`, where `match` is the Matching we pass into the function. The index of this ArrayList corresponds to each tenant. The value at that index indicates to which apartment he/she is matched.

- You must implement the methods `isStableMatching()` and `stableMatchingGaleShapley()` in the file `Program1.java`. You may add methods to this file if you feel it is necessary or useful. You may add additional source files if you so desire.

- To compile the given code, from the command line go to the directory which contains your code, and compile it with
  `javac *.java`

- `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] <path to input file>`).

- 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 driver to verify the correctness of your solutions.

- The format of the input file, e.g. `4.in` is as follows

```
3 4
1
0
2 3
1 1 2 2
2 1 1 1
1 1 1 1
2 3 1 1
1 2 2 2
1 1 1 1
1 2 1 1
```

  - First line gives the number of landlords and apartments, so in this case 3 = number of landlords and 4 = number of tenants/apartments
  - The next three lines indicate what landlords own what apartments, here L0 owns A1, L1 owns A0 and L2 owns A2, A3
  - This is followed by landlord preference lists, each landlord's list is on a separate line, e.g. the line 1 1 2 2, gives the preference list for L0 in the order of T0, T1, T2, & T3. It appears that L0 likes T0 and T1 equally (indifferent) in first preference and then likes T2 and T3 equally (indifferent) in second preference
  - The last four lines are the tenant preference lists for each of the four tenants

One possible weakly stable matching:
Tenant 0 Apartment 2
Tenant 1 Apartment 0
Tenant 2 Apartment 1
Tenant 3 Apartment 3
Another possible weakly stable matching:
Tenant 0 Apartment 3
Tenant 1 Apartment 2
Tenant 2 Apartment 1
Tenant 3 Apartment 0
Incorrect matching:
Tenant 0 Apartment 0
Tenant 1 Apartment 2
Tenant 2 Apartment 3
Tenant 3 Apartment 1
(consider Tenant3 and Apartment0; Owner of Apartment0 would prefer Tenant3 over Tenant0; Tenant3 would prefer Apartment0 over Apartment1; Hence, the matching is not weakly stable)

- 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 (i.e. all java files except `Driver.java, Permutation.java, Matching.java and AbstractProgram1.java`) not your .class files. So please take a moment to separate out the required .java files before zipping them.
- your 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 11:59pm on the day it is due. No late submissions will be accepted.

Example for testing your code:

```
3 3
0
1
2
2 1 3
3 2 1
1 3 2
2 1 3
3 2 1
1 3 2
```

Three possible weakly stable matchings:

Tenant 0 Apartment 1	Tenant 0 Apartment 0	Tenant 0 Apartment 2
Tenant 1 Apartment 2	Tenant 1 Apartment 1	Tenant 1 Apartment 0
Tenant 2 Apartment 0	Tenant 2 Apartment 2	Tenant 2 Apartment 1