EE W360C: Algorithms
University of Texas at Austin — Programming Assignment #1
Dr. Parikh — Due: Monday, June 24 (Report) / July 1 (Code) 2019, 11:59pm

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.

For this problem we would like to match PhD advisors with college graduates. Assume each advisor chooses students based on a mix of locality to their school and academic merit. Each advisor will take on only one new student. The college graduates rank their preferred advisors according to how much they like their research work, with no ties. The advisors create rankings based on the graduates GPAs. **If there's a tie between two student's GPAs the advisor will prefer the student closest to themselves.** Assume there will be no ties after location is taken into account. Also, assume the number of advisors and students is the same.

With this information, assume advisors came up with an algorithm of selecting students. Advisors are given the lists of applicants and could decide (based on the above strategy) which student to offer admission to. All advisors made decisions independently. Once all first round decisions were made, they were all sent to the graduates, and each graduate could opt to take any offered admission (assume this is done based solely on their original preference list). After offers had been accepted, advisors were allowed to make a second round of offers and then a third. At the end of three rounds, any unassigned students were not admitted to any PhD program. Assume that for each round of offers, a student's preference list does not change, and that after accepting an offer, they can't reject it and accept another.

Part 1: Write a report [30 points]

Write a short report that includes the following information:

(a) This algorithm is incredibly inefficient. Assume there are n advisors and n students. In the worst case, what would be the number of pairs, x, made? What would the preference lists of the advisors and students look like to end up with only x pairs?

(b) Provide a better algorithm in pseudocode that finds 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 (i.e. no-one is unmatched, matching is stable) result.

(d) Give the runtime complexity of your algorithm in Big-O notation and explain how you derived it from the pseudocode.

(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 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 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 advisors increases.

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

A brute force solution to this problem involves generating all possible permutations of students and advisers, 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 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 1.8. **It is YOUR responsibility to ensure that your solution compiles with the standard Java 1.8 configuration (JDK 8).** For most (if not all) students this should not be a problem. If you have doubts, email a TA or post your question on Piazza.
- 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, as 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:
  - **m**: The number of advisors
  - **n**: Number of graduate students (should be the same as **m**)
  - **student_GPAs**: An ArrayList containing each student's GPAs. The students are in order from 0 to $n-1$.
  - **student_preferences**: An ArrayList of ArrayLists containing each of the student's preferences for advisors, 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 advisors who are identified by numbers 0 to $m-1$.
  - **advisor_locations**: An ArrayList that specifies x and y coordinates for the advisors location. The index of the value corresponds to which advisor it represents.
  - **student_locations**: An ArrayList that specifies x and y coordinates for the graduate student's location. The index of the value corresponds to which student it represents.
  - **student_matching**: An ArrayList to hold the final matching. This ArrayList (should) hold the number of the advisor 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 advisor he/she is matched.
- 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.in input file should output:
  - Student 0 Adviser 0
  - Student 1 Adviser 3
  - Student 2 Adviser 2
  - Student 3 Adviser 1
- 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.
- Do not add a package to your code, use the default package. If you have a line of code at the top of your Java file that says `package <some package name>;` that is wrong.
- 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`.
- In this lab you will be using `double` values to represent a student's GPA. Normally this would make it difficult to compare the values directly (due to floating point precision errors). We have designed the test cases such that this should not be a problem. It is fine to use `if (double_val1 == double_val2) {}` in this lab, but in future courses this is dangerous and prone to failure.
- Assume `ArrayList.indexOf` has a time complexity of $O(1)$.

What To Submit - June 24

You should submit a pdf file titled `eid_lastname_firstname.pdf` that contains Part 1(a) through 1(e). The grade you will receive on (a) and (e) will be final. The grade on (b)-(d) is not (you may

re-submit with changes later), but it is useful to put effort in it as this will allow you to receive timely feedback on possible weaknesses of your implementation.

What To Submit - July 1

You should submit a single zip file titled `eid_lastname_firstname.zip` that contains:

- all of your java files
- a pdf called `eid_lastname_firstname.pdf`, containing Part 1(f), and, if you made any changes, 1(b) through (d).

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 11:59pm on the day it is due.

Some Tips

- Part 1(b) - carefully planning out the pseudocode on which your code will be based - is very important in order to make your life easier (and runtime complexity lower) when writing your code later. Ideally, you want to have steps that are clearly defined and can easily be converted into code. It is time to think about how to efficiently implement certain steps that we immediately consider O(1) when we see them in class... For example, step 2 of Gale-Shapley (according to lecture 4 slides), calls for fetching a man m who is free and hasn't proposed to every $w \in W$. This of course isn't something that directly corresponds to a java instruction. So you need to figure out: where will you store the necessary information (e.g. data structure, object)? How will you make sure it takes O(1) time to find such a man?
- Don't worry if you can't figure out how to make a step take the minimum asymptotic time possible. As long as your (non-brute-force) code doesn't take forever to run, you will not be penalized.
- You won't be needing any data structure more complicated than an array or an ArrayList for the purposes of this assignment.
- Add comments in your code! Make it easy to tell what each chunk of code does.
- Java has a function that sorts a list in time O(nlogn), where n is the number of elements to be sorted.
- You may find it helpful to use Comparator(s).
- For part 1(b), there's no need to go into extreme detail. Many of the steps will follow naturally once you have appropriately created the data structures and objects you will need. In the example of "fetching a man m who is free and hasn't proposed to every $w \in W$", it would suffice to state in your pseudocode that you create a data structure that contains all free men, and then replace the abstract step by "remove a man from the list of free men", which is clearly O(1).