Introduction to Algorithms	**Problem Set 1 (3 problems)**
CS 4820 Fall 2018	**Due 11:59pm Thursday, August 30**

Your homework submissions need to be typeset (hand-drawn figures are OK). See the course web page for suggestions on typing formulas.

The solution to each question needs to be uploaded to CMS as a separate pdf file. To help provide anonymity in your grading, do not write your name on the homework (CMS will know it's your submission). Remember that when a problem asks you to design an algorithm, you must also prove the algorithm's correctness and analyze its running time. The running time must be bounded by a polynomial function of the input size.

(1) Implementing Gale-Shapley. In this problem we consider the classical Gale-Shapley algorithm. Given n proposers and n respondents, and a preference list for each proposer and respondent of all members of the opposite sex, give an implementation of the Gale-Shapley stable matching algorithm that runs in $O(n^2)$ time, as explained on page 46 of the book.

We have provided a starting environment for your solution in the file `Framework.java` (available on CMS). Use the framework code to read the input and write the output in a specific form (this makes it easy for us to test your algorithm). The only thing you need to implement is the algorithm, and you are restricted to implementing this between the lines `//YOUR CODE STARTS HERE` and `//YOUR CODE ENDS HERE`. This is to make sure you can only use classes from `java.util` (imported at the start of the file). Submit your modified `Framework.java` file with the the algorithm filled in.

Warning: Be aware that the running time of calling a method of a built-in Java class is usually not constant time, and take this into account when you think about the overall running time of your code. For instance, if you use a LinkedList, and use the `indexOf` method, this will take time linear in the number of elements in the list.

You can test your code with the test cases provided on CMS. `Framework.java` takes two command line arguments, the first is the name of the input file, and the second is the name of the output file. The input file should be in the same folder in which your compiled Java code is. After you compile and run your code, the output file will also be in the same folder. In order to test your code with the provided test cases, copy the test cases in the folder in which you have compiled your code. Then, when executing your code, set the name of the input file to be the name of one of the sample inputs (`Test{i}in.txt` for i in {0, 1, ..., 5}). For the smaller test instances numbered 0-3, (`Test{i}out.txt` for i in {0, 1, ..., 3}) are the output of the Gale-Shapley algorithm for the corresponding sample test case. The first four test examples are small; you can use these to help test the correctness of your code by running the algorithm, and checking if your code generates the same output as the one we gave you.

(Note that the resulting matching does not depend on the order of proposals made.) The larger instances are useful to help test the running time. The last two have $n = 1\ 000$ and $n = 2\ 000$. The running time of your code should increase quadratically, not cubically, as the input gets bigger. So doubling n should roughly increase the running time by a factor of 4, and **not by a factor of 8**. We expect that even the $n = 2\ 000$ instance should take less than 1-2 seconds to run if your code is $O(n^2)$.

The format of the input file is the following:

- First line has one number, n. Proposers and respondents are labeled with numbers $0, 1, \ldots, n-1$.
- In each of the next n lines, we are providing the preference list of a proposer. The ith line is the preference list of the ith proposer (the first respondent in the list is the most preferred and the last respondent is the least preferred).
- In each of the next n lines, we are providing the preference list of a respondent. The ith line is the preference list of the ith respondent (the first proposer in the list is the most preferred and the last proposer is the least preferred).

Each line of the output file corresponds to a proposer and a respondent that are matched by the algorithm. The code reads in the input, and stores this in two $n \times n$ matrices: ProposerPrefs and RespondentPrefs, where row i of the ProposerPrefs matrix lists the choices of proposer i in order (first respondent is most preferred by proposer i), and similarly, row i of the RespondentPrefs lists the choices of respondent i in order. Your code needs to output a stable matching by putting each matched pair in the MatchedPairsList and needs to run in $O(n^2)$ time.

We will use Java 8 to compile and test your program.

(2) Resident matching. We studied the stable matching in class assuming that each side has a full preference list, and that the number of boys and girls is the same. Stable matching is used in many applications, including assigning residents to hospitals or children to schools in many cities (e.g., in Boston). In the case of residents applying to hospitals, residents are limited to apply only to a small set of hospitals. To make this concrete, assume that we have the following (somewhat simplified) arrangement.

- There are n applicants and a set of m hospitals. We will assume that each applicant is asked to list k hospitals that (s)he is interested in, and lists them in the order of his/her preference. (In reality, they can list fewer than k, or they can pay extra for some extra preferences, so the lists may not be all of the same length, but let's ignore this complication for the exercise.)
- Now each hospital gets the full list of applicants that listed the particular residency option, and they need to rank all applicants. (Another simplification we will assume is that the options are all different, while in reality hospitals typically need multiple residents of the same specialty).
- In this situation, we cannot expect to match all residents to hospitals. We will extend the definition of a stable matching to be a matching that satisfied the following conditions which we'll call weak stability:
  - For all matched pairs (r, h) the hospital h was listed on r's preference list,
  - for any pair (r, h) of resident r, and hospital slot h on r's preference list, if the pair (r, h) is not part of the matching, then one of the following must hold:
    - h is assigned to a different resident r' and h prefers r' to r
    - r is assigned to a different hospital h' and r prefers h' to h.

This is very similar to the traditional stable matching problem, but some hospitals as well as some prospective residents may remain unmatched.

(a) Show that there is a stable matching for any set of prospective residents and hospitals and any preference lists, and give an algorithm to find one in time at most $O(nm)$ time.

(b) Prospective residents have an awkward decision to make of what k programs should they apply for. In the definition above we assumed that residents will be applying to their top k choices of hospitals. Can it happen that a resident would get a better match if applying to some other hospitals, not his or her top k? Either prove that this is not possible, or give an example when this happens.

(3) Maintenance Month. Peripatetic Shipping Lines, Inc., is a shipping company that owns n ships, and provides service to n ports. Each of its ships has a schedule which says, for each day of the month, which of the ports it's currently visiting, or whether it's out at sea. (You can assume the "month" here has m days, for some $m > n$.) Each ship visits each port for exactly one day during the month. For safety reasons, PSL Inc. has the following strict requirement:

(*) *No two ships can be in the same port on the same day.*

The company wants to perform maintenance on all the ships this month, via the following scheme. They start with the regular monthly schedule, that does satisfy the requirement (*). They want to truncate each ship's schedule: for each ship S_i, there will be some day when it arrives in its scheduled port and simply remains there for rest of the month (for maintenance). This means that S_i will not visit the remaining ports on its schedule (if any) that month, but this is okay. So the truncation of S_i's schedule will simply consist of its original schedule up to a certain specified day on which it is in a port P; the remainder of the truncated schedule simply has it remain in port P. Now the company's question to you is the following: Given the schedule for each ship, find a truncation of each so that condition (*) continues to hold: no two ships are ever in the same port on the same day.

Show that such a set of truncations can always be found, and give an efficient algorithm to find one.

Example: Suppose we have two ships, Ships A and B, and two ports, P_1 and P_2, and that a "month" has four days. Suppose the first ship's schedule is

port P_1; at sea; port P_2; at sea

and the second ship's schedule is

at sea; port P_1; at sea; port P_2

Ship	Day 1	Day 2	Day 3	Day 4
Ship A	Port P_1	at sea	Port P_2	at sea
Ship B	at sea	Port P_1	at sea	Port P_2

For this example, the only valid way to choose truncations would be to have the first ship remain in port P_2 starting on day 3, and have the second ship remain in port P_1 starting on day 2.