Introduction to Algorithms	Problem Set 1
CS 4820, Spring 2018	Due 11:59pm Thursday, February 1

Hand in your solutions electronically using CMS. Each solution should be submitted as a separate file. Collaboration is encouraged while solving the problems, but:

1. **list the names of those with whom you collaborated;**
2. **you must write up the solutions in your own words;**
3. **you must write your own code.**

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) *(5 points)*
For any positive integer n, let L_n denote an L-shaped region in the plane obtained by starting with a square of side length 2^n and deleting its upper right quadrant. For example, L_1 is the "L-shaped tromino tile" discussed in class on Wednesday. See Figure 1 for additional examples.

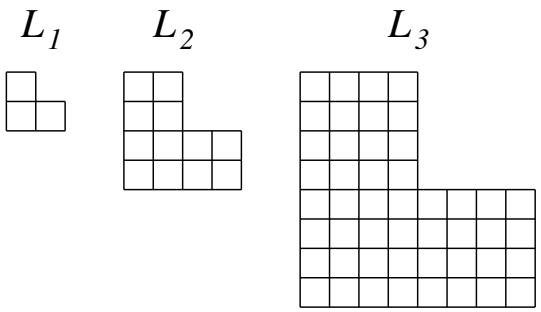

Figure 1: The regions L_1, L_2, L_3

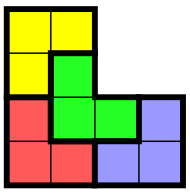

Figure 2: Tiling L_2 with copies of L_1

Prove, **using mathematical induction,** that for every positive integer n it is possible to tile L_n using copies of L_1. In other words, you should prove that L_n can be partitioned into regions, each congruent to L_1. See Figure 2 for an example when $n = 2$.

Try to make your proof as clear and precise as possible. You do not need to describe an algorithm to compute the tiling, nor analyze its running time. You only must prove that such a tiling exists.

(2) *(10 points)*
Consider the following scenario: n students get flown out to the Bay Area for a day of interviews at a large technology company. The interviews are organized as follows. There are m time slots during the day, and n interviewers, where $m \geq n$. Each student s has a fixed *schedule* which gives, for each of the n interviewers, the time slot in which s meets with that interviewer. This, in turn, defines a schedule for each interviewer i, giving the time slots in which i meets each student. The schedules have the property that

- each student sees each interviewer exactly once,
- no two students see the same interviewer in the same time slot, and
- no two interviewers see the same student in the same time slot.

Now, the interviewers decide that a full day of interviews like this seems pretty tedious, so they come up with the following scheme. Each interviewer i will pick a *distinct* student s. At the end of i's scheduled meeting with s, i will take s to one of the company's many in-house eateries, and they'll both blow off the entire rest of the day sipping espresso and rubbing elbows with celebrity chefs.

Specifically, the plan is for each interviewer i, and his or her chosen student s, to *truncate* their schedules at the time of their meeting; in other words, they will follow their original schedules up to the time slot of this meeting, and then they will cancel all their meetings for the entire rest of the day.

The crucial thing is, the interviewers want to plan this cooperatively so as to avoid the following *bad situation*: some student s whose schedule has not yet been truncated (and so is still following his/her original schedule) shows up for an interview with an interviewer who's already left for the day.

Give an efficient algorithm to arrange the coordinated departures of the interviewers and students so that this scheme works out and the *bad situation* described above does not happen.

Example:
Suppose $n = 2$ and $m = 4$; there are students s_1 and s_2, and interviewers i_1 and i_2. Suppose s_1 is scheduled to meet i_1 in slot 1 and meet i_2 in slot 3; s_2 is scheduled to meet i_1 in slot 2 and i_2 in slot 4. Then the only solution would be to have i_1 leave with s_2 and i_2 leave with s_1. If we scheduled i_1 to leave with s_1, then we'd have a bad situation in which i_1 has already left the building at the end of the first slot, but s_2 still shows up for a meeting with i_1 at the beginning of the second slot.

	Time slot			
	1	2	3	4
s_1	i_1		i_2	
s_2		i_1		i_2

	Time slot			
	1	2	3	4
s_1	i_1		(i_2)	
s_2		(i_1)		i_2

	Time slot			
	1	2	3	4
s_1	(i_1)		i_2	
s_2		i_1		i_2

(3) *(10 points)*
In this problem we consider the classical Gale-Shapley algorithm. Given n men and n women, and a preference list for each man and woman of all members of the opposite sex, give an implementation of the Gale-Shapley stable matching algorithm (with men proposing to women) that runs in $O(n^2)$ time, as explained on page 46 of the book.

Implement the algorithm in Java using the environment provided — use the framework code (Framework.java) we provide on CMS, 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 implement 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).

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, and set the name of the input file to be the name of one of the sample inputs (`Test`i`in.txt` for $i = 0, 1, \ldots, 4$). Each of the provided sample outputs (`Test`i`out.txt` for $i = 0, 1, \ldots, 4$) are the output of the Gale-Shapley algorithm when men propose to women 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 $n = 1000$ and $n = 2000$. The running time of your code should increase quadratically, not cubically, as the input gets bigger. We expect that even the $n = 2000$ 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. Both men and women are labeled with numbers $0, 1, \ldots, n-1$.
- In each of the next n lines, we are providing the preference list of a man. The ith line is the preference list of the ith man (the first woman in the list is the most preferred and the last woman is the least preferred).
- In each of the next n lines, we are providing the preference list of a woman. The ith line is the preference list of the ith woman (the first man in the list is the most preferred and the last man is the least preferred).

Each line of the output file corresponds to a man and a women that are matched by the algorithm.

The code reads in the input, and stores this in two $n \times n$ matrices: MenPrefs and WomenPrefs, where row i of the MenPrefs matrix lists the choices of man i in order (first woman is most preferred by man i), and similarly, row i of the WomenPrefs lists the choices of woman 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 use Java 8 for compiling and testing your program.