| **Introduction to Algorithms** | **Problem Set 1** |
|---|---|
| **CS 4820 Spring 2016** | **Due 11:59pm Thursday, February 4** |

Your homework submissions need to be typeset (hand-drawn figures are OK). See the course web page for suggestions on typing formulas. Solution to each question needs to be uploaded to CMS as a separate pdf file. 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)** 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, gine 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 your algorithm in Java using the environment provided.

For the code, you must use the framework code (*Framework.java*) we provide on CMS. We will put the code you have submitted in the part of the framework that we have specified by the commented line
*//YOUR CODE GOES HERE*
and will run the code. So, you should only submit that part of your code in *Fragment.txt* file. Please read the provided framework carefully before starting to write your code. You can test your code with the test cases provided on CMS. *Framwork.java* will take 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 (*SampleTest_i.txt* in which $0 \leq i \leq 5$). Each of the provided sample outputs (*SampleOutput_i.txt* in which $0 \leq i \leq 5$) are the output of the Gale Shapley algorithm when men propose to women for the corresponding sample test case.

The format of the input file is the following:

- First line has one number, $n$. Men are labeled with numbers $1, 2, \ldots, n$, and women are also labeled with numbers $1, 2, \ldots, n$.
- In each of the next $n$ lines, we are providing the preference list of a man. The $i$th line is the preference list of the $i$th 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 $i$th line is the preference list of the $i$th woman (the first man in the list is the most preferred and the last man is the least preferred).

In each line of the output file, there should be 2 numbers $x$ and $y$ which mean that in the stable matching, man $x$ will get matched to woman $y$. Our framework handles reading the input from the input file and writing the output in the output file, so your job is to only implement the algorithm.

Our code sets up the preferences of the $n$ men in an $n$ by $n$ matrix *MenPrefs*, where row $i$ of the matrix lists the choices of man $i$ in order (first woman is best). We use an $n$ by $n$ matrix *WomenPrefs* where $j$th row of this matrix lists men in the order of preferences of woman $j$ (first man is best). Your code needs to output a stable matching by putting each matched pair in the *MatchedPairsList* and needs to run in at most $O(n^2)$ time.

**(2)** 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, from 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 less than $k$ or they can also 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:
  - 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 $h$.

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

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 $O(n^2)$ time.

**(3)** Consider the following scenario. $n$ computer science students get flown out to the Pacific Northwest for a day of interviews at a large software company. The interviews are organized as follows. There are $m$ time slots during the day, and $n$ interviewers, where $m > n$. 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 $S$ out for coffee at one of the numerous local cafes, and they'll both blow off the entire rest of the day drinking espresso and watching it rain.

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.[1]

**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.

[1]Note that there might be multiple such solutions, and your algorithm can return any one of them.