(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.

Solution We proved in class that the Gale-Shapley algorithm with n proposers and n responders runs through its while loop of a free proposers making an offer $O(n^2)$ times. To implement this in $O(n^2)$ time, we can spend $O(n^2)$ time pre-processing, setting up useful data structures, and then be able to implement each iteration through the while loop in $O(1)$ time. Recall the steps of the while loop

1. select a free proposer m
2. identify the highest ranked respondent w on m's list that m has not yet made an offer to
3. if w is currently matched to a proposer m', check if w prefers m' to m. If w prefers m' no need to update the matching, else add the pair (m, w) to the matching and possibly delete (m', w) if w was matched.

To do these, its clearly useful to maintain a list of free proposers, `FreeProposers`, starting with all the proposers, and the while loop ends when this list is empty. This takes care of the first item.

To think of the second item, we need an array that tells us for each Proposer, how far down their offer list they went so far. Say an array `NumOffers` where item `NumOffers`$[m]$ is the number of offers proposer m made so far, starting with all 0. This way we can look up their next favored respondent in $O(1)$ time.

It is also useful to keep an array where we can look up in $O(1)$ time for any respondent who their current partner is, say an array `Match` where `Match`$[w]$ is the proposer respondent w is matched to.

The hardest part is to be able to decide if w prefers m or m' in $O(1)$ time. To do this, we need to build for each respondent a list of how they rank proposers, an array `Rank` where `Rank`$[w][m]$ will return the rank of proposer m on w's list of preferences. This is an n^2 size array, and takes $O(n^2)$ to build, which is part of pre-processing. After this the decision is simply checking if `Rank`$[w][m] >$ `Rank`$[w][m']$.

Common mistakes The most common mistake was to not build the `Rank` array, but rather walk through w's list every time they need to choose between two proposers. This takes $O(n)$ per iteration, and there are $O(n^2)$ iterations. Students who did this ended up with 7-8/10 points, as their code would fail on some of the 3 large examples we were testing on. Many students also had a proposer attempt to propose to the same woman multiple times, or (equally as bad) iterated through a proposer's preference list at each round to determine the first respondent to whom they had not yet proposed.

(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.

Solution (a) We can run the same algorithm as the classical stable matching. Say residents propose to hospitals. Each hospital tentatively accepts the best resident that proposed so far, and each resident, when not tentatively matched, makes a proposal to the next hospital on his/her list. When the list runs out, the resident remains unmatched. To make sure the implementation is efficient, we start by setting up the matrix `Ranking` of hospitals' ranking of residents as was done for problem 1. This can be done in $O(nm)$ time, and allows us to answer queries if a hospital h prefers resident r or r' in $O(1)$ time.

We claim that this algorithm solves the problem. First, the running time:

- Setting up the `Ranking` matrix takes $O(nm)$ time.
- There are n residents, and each can make at most k proposals, so there will be at most nk proposals made. Clearly $k \leq n$.
- Given a new proposal, the hospital needs to check which resident they prefer, which we can do in $O(1)$ time using the `Ranking` matrix. This will be a total of $O(kn)$ time if one considers every possible proposal.
- Finally, we need a few simple pointers, and a list. Each resident needs a pointer to its next possible proposal, and we need to keep a list of all unmatched residents, and make one of them make its next proposal.

For correctness, we need to prove that the resulting matching is stable. Consider a resident hospital pair (r, h) with h on r's list. We need to argue that (r, h) doesn't form an instability.

- Notice that if a hospital ever gets a proposal, it will remain matched throughout, and that its matched resident is improving over time. We have seen this property of the classical Gale-Shapley algorithm. Here it is true for the same reason, the algorithm makes the hospitals select the better of the proposals each time.
- The residents make offers in decreasing order.
- Now if (r, h) are not part of the matching, there can be two reasons for this:
  - either r never made a proposal to hospital h, and if this is the case, r must be matched to a hospital h' that r prefers to h, so (r, h) is not an instability
  - or r made a proposal to h, but was rejected. Using the top observation above, then h must remain matched throughout the process, and ends up being matched to a resident r' they prefer to r, so again (r, h) is not an instability.

Alternate solution 1 Offers can be made also by hospitals. So hospitals sort their applicants by preferences, and we get hospitals to make offers to residents, i.e., hospitals playing the role of men in the Gale-Shapley algorithm. This also works just as well, and the proof of correctness is analogous. For the running time, we now need to preprocess the residents' preferences so we can decide in $O(1)$ time if a resident r prefers hospitals h or h'.

Alternate solution 2 They can also reduce to the original stable matching problem: extend the preference list to a full list by adding the unlisted options on the bottom in any order. Now use Gale-Shapley to get a stable matching. Finally if any pair is matched even though the resident didn't even apply to the hospital, just delete this match, and keep both parties unmatched. We claim that the resulting matching satisfies the new definition of being stable.

Solution (b) True; it is possible that a resident could get a better match by applying to hospitals other than their top k. The simplest example is with $k = 1$ and 2 hospitals and 2 residents. If both residents have the same top choice, the resident not selected by the hospital would have been better off applying to the other hospital.

Common mistakes One common mistake was to not mention preprocessing the preferences of the proposed-to group, which is essential in keeping each iteration on $O(1)$ time. Also, many of the correctness proofs did not cover cases where r and h could be unmatched, when (r, h) is a pair not in the matching.

(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.

Solution The simple solution is to **reduce** to the stable matching problem, then solve it in $O(n^2)$ time with the Gale-Shapley algorithm. This is an application of the stable matching problem, where we want to match each ship to a port where it is going to undergo maintenance. The n ships and the n ports would be the analogues of the men and the women. What we need to identify is the preference rules for the ships and the ports given the full schedules of the ships for the current month.

The correct preference rules are that ships prefer earlier ports along their schedule and the ports prefer later ships. This rule uniquely determines preference lists for ships and ports, because we are given that according to the original (not truncated) schedules, no two ships visit the same port on the same day and each ship eventually visits each port.

Given these preference rules, we run the stable matching algorithm and claim that the resulting stable match would produce no conflicts.

We prove this latter claim by contradiction. Assume that ships s_1 and s_2 were at the port p_1 simultaneously, where p_1 is the destination port for s_1 (notice that this is the only possible way of achieving a conflict, since if neither s_1 nor s_2 had reached their destination ports, then they were following their original schedule, which we know is collision-free). Since s_1 was in p_1 before s_2, then p_1 prefers s_2 to s_1 (ports prefer late arriving ships). And since s_2 visited p_1 before its matched destination p_2, then s_2 prefers p_1 to p_2 (ships prefer earlier ports). Therefore the pair (s_2, p_1) is an instability - a contradiction.

Alternate algorithms. It is also possible to reinvent the algorithm: ships propose to port in the order of their schedule, and ports prefer later ships.

- ships stop at their first port (a proposal
- if a later ship comes, the ship docking at the port moves on to its next destination. imagine for stating the algorithm that it arrives to the next port at the originally scheduled time (despite leaving later).
- terminate when each ship is docked.

This is the same as Gale-Shapley in the reduction abobe with the ships as proposers. However, as this solution reinvents the algorithm (rather than recognizing that its the same algorithm), we have to prove its properties again. Properties that need proving are

- algorithm terminates + running time, this includes also that ports once they tentatively have a ship, will always have one, and these are later and later ships (assuming ships propose).
- "no bad situation" can arise. Prove this by contradiction. Assume there is a bad situation: ship s is showing up and port p is occupied. If s is coming to port p than its matched port p' must be later in her schedule, i.e., she must have moved on from p. But when p asked s to move on, it already has a latter arriving ship dock there, and ports assigned ships only get later in the schedule. A contradiction.

Common mistakes. The most common mistake was missing some part of the proof, or possibly getting some of the preferences backwards. If both ports and ships have the same order of preferences, the resulting algorithm similar to just using greedy, and no longer works.

A few people used greedy directly. No greedy algorithm works (as best as we know. There are two natural versions

Version 1. Take an arbitrary port, and assign last ship/port pair (then solve the remaining problem recursively). To see why this doesn't work consider the example where the docking times are in this order (with boldface marking the algorithm's choices for docking:

$$(p_1, s_2), (p_2, s_2), (p_2, s_1), (p_1, s_1)$$

So ship s_1 and port p_1 are the first pair assigned (that is last arrival time), and then we match (p_2, s_2). The bad situation arises when ship s_1 shows up in port p_2.

Version 2. Consider port who finishes first, and make it take the last ship it meets. This seems a bit harder to brake. (It works if there are only 2 pairs!)
In the example below, s_1 and p_1 is the first pair assigned (that is p_1 finished first and s_1 was his last ship), now next s_2 and p_2 are assigned, and then s_3 and p_3. Suppose the times were in this order (with boldface marking the algorithm's choices for docking):

$$(p_1, s_3), (p_1, s_2), (p_2, s_3), (p_3, s_3), (p_3, s_1), (p_1, s_1), (p_2, s_2), (p_2, s_1), (p_3, s_2)$$

The first assigned is (p_1, s_1), then (p_2, s_2), and finally (p_3, s_3). Now the bad situation arises when s_1 shows up at p_3 is taken before s shows up!

Version 2' This is the analogous version with ships/port interchanged: so we consider the ship that starts last, and assign it to doc at its first port. Bad example for this is the above with ships/port's roles reversed

$$(p_2, s_3), (p_1, s_2), (p_2, s_2), (p_1, s_1), (p_1, s_3), (p_3, s_3), (p_2, s_2), (p_2, s_1), (p_3, s_1)$$

Note that there is no need to rewrite out the algorithm if doing a reduction. You only need to write out how to transform the inputs of your problem into the inputs of the algorithm. For example, in

Gale-Shapley, you only need to explain how to transform the ship's schedule into preference lists and then prove properties about the output.

Furthermore, by rewriting out a proven algorithm,there is always the possibility of making a mistake. For example, a common mistake for students that reimplemented Gale-Shapley is that they often forget that they can't just choose a random proposer at each iteration. The proposer must be chosem from the set of unmatched proposers. Otherwise, the algorithm will just loop forever.