University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	Algorithmics II (H)			
Coursework Number	1 of 1			
Deadline	Time:	4.30pm	Date:	19 November 2021
% Contribution to final course mark	20%			
Solo or Group ✓	Solo	✓	Group	
Anticipated Hours	20			
Submission Instructions	See Section 5			
Please Note: This coursework cannot be re-assessed				

You must complete a Declaration of Originality when making your submission via Moodle.

Algorithmics II (H)

Assessed Exercise 2021-22
Assigning Junior Doctors to Hospitals

1 Introduction

1.1 General

This is the only assessed practical exercise for Algorithmics II (H), and accounts for 20% of the assessment for this course. As a rough guide, it is expected that it should be possible to obtain full marks by putting in no more than 20 hours of work, and you are advised not to spend significantly more time than this on the exercise. The language of implementation for this exercise is Java.

The exercise is to be done individually. Some discussion of the exercise among members of the class is to be expected, but close collaboration or copying of code, in any form, is strictly forbidden – see the School's plagiarism policy, contained in Appendix A of the Undergraduate Class Guide (available from `https://moodle.gla.ac.uk/course/view.php?id=21505`).

1.2 Deadline for submission

The hand-out date for the exercise is **Friday 29 October 2021**, by which time all the relevant material will have been covered in lectures. The deadline for submission is **4.30pm, Friday 19 November 2021**. The course web page on Moodle will be used for this exercise, providing setup files and an electronic submission mechanism. Guidance as to what should be submitted is given in Section 5.

1.3 Context

The National Health Service of Caledonia (NHSC) wishes to automate the process of allocating intending junior doctors to hospital posts using a matching algorithm. Doctors have preferences over the available hospitals, and likewise hospitals express preferences over the doctors who have applied to them. Each hospital also has a capacity, indicating the maximum number of doctors it can take on. Your task as an algorithm designer is to provide the NHSC with software for matching doctors to hospitals based on these preference lists and capacities. The algorithm should ensure that the computed matching is "stable", i.e., no doctor and hospital would prefer to be assigned to one another than to remain with their existing assignees. Algorithms are required for the cases that (i) preference lists are strictly ordered, and (ii) hospitals' preference lists may include ties (indicating the possibility that a given hospital might be indifferent between batches of doctors in its preference list).

1.4 Structure of the document

The remainder of this document is organised as follows. Section 2 defines the matching problems under consideration and describes the two matching algorithms that are to be implemented. Section 3 gives an overview of the skeleton code that is made available to you (this code includes a main class, an input file parser, relevant object classes and a dataset generator to help you test your code). Section 4 lists the implementation tasks that are required as part of this exercise. Section 5 explains how to submit your solution, and finally Section 6 summarises the marking scheme for the exercise.

2 Background

2.1 Problem definitions

An instance I of the *Hospitals / Residents problem with Ties* (*HRT*) comprises a set $D = \{d_1, d_2, \dots, d_{n_1}\}$ of doctors and a set $H = \{h_1, h_2, \dots, h_{n_2}\}$ of hospitals. Each doctor (respectively hospital) ranks a subset of the hospitals (respectively doctors) in order of preference, where hospitals' preference lists may include ties. Additionally, each hospital $h_j \in H$ has a *capacity* $c_j \in \mathbb{Z}^+$, meaning that h_j can be assigned at most c_j doctors, while each doctor can be assigned to at most one hospital. If all preference lists in I are strictly ordered, we say that I is an instance of the *Hospitals / Residents problem* (*HR*). A doctor $d_i \in D$ is said to find a hospital $h_j \in H$ *acceptable* if h_j belongs to d_i's preference list, and we define acceptability for a hospital in a similar way. We assume that preference lists are *consistent*, that is, given a doctor–hospital pair $(d_i, h_j) \in D \times H$, d_i finds h_j acceptable if and only if h_j finds d_i acceptable. If d_i does find h_j acceptable then we call (d_i, h_j) an *acceptable pair*.

Given that a hospital h_j's preference list may include ties, we say that h_j *prefers* doctor d_i to doctor d_k if d_i comes before d_k in h_j's list, and these doctors are not involved in the same tie in h_j's list. For example, in the following preference list, where ties are represented by round brackets, h_1 prefers d_1 to d_3 and prefers d_3 to d_6, but does not prefer d_3 to d_4 (h_1 ranks d_3 and d_4 equally and is said to be *indifferent* between them).

$$h_1 : (d_1\ d_2)\ (d_3\ d_4\ d_5)\ d_6$$

A *matching* M in I is a subset of acceptable pairs such that each doctor appears in at most one pair, and each hospital $h_j \in H$ appears in at most c_j pairs. Given a doctor $d_i \in D$, if $(d_i, h_j) \in M$ for some $h_j \in H$ then we say that d_i is *matched* and let $M(d_i)$ denote h_j, otherwise d_i is *unmatched*. Given a hospital $h_j \in H$, we let $M(h_j) = \{d_i \in D : (d_i, h_j) \in M\}$ denote the set of *assignees* of h_j in M. We say that h_j is *full* or *undersubscribed* in M if $|M(h_j)| = c_j$ or $|M(h_j)| < c_j$, respectively. We now define stability.

Definition 1. *Let I be an instance of HRT and let M be a matching in I. A doctor–hospital pair $(d_i, h_j) \in (D \times H) \setminus M$ is a* blocking pair *of M, or* blocks *M, if*

1. *(d_i, h_j) is an acceptable pair,*
2. *either d_i is unmatched in M or d_i prefers h_j to $M(d_i)$, and*
3. *either h_j is undersubscribed in M or h_j prefers d_i to some member of $M(h_j)$.*

M is said to be stable *if it admits no blocking pair.*

In an instance of HRT, stable matchings can have different sizes. To illustrate this, consider the HRT instance I_1 with two doctors and two hospitals shown in Figure 1. Here each hospital has capacity 1, d_1 prefers h_1 to h_2, d_2 finds only h_1 acceptable, and h_1 is indifferent between d_1 and d_2. Define the matchings $M_1 = \{(d_1, h_1)\}$ and $M_2 = \{(d_1, h_2), (d_2, h_1)\}$. Both matchings are stable: (d_2, h_1) does not block M_1 because h_1 does not prefer d_2 to its assignee d_1; likewise (d_1, h_1) does not block M_2 because h_1 does not prefer d_1 to its assignee d_2.

The fact that stable matchings can have different sizes motivates the problem of finding a maximum size stable matching in an instance of HRT. This problem is known to be NP-hard[1] [6, 7].

[1]This means that, assuming P$\neq$NP, there is no efficient (polynomial-time) algorithm to find a maximum size stable matching in an instance of HRT.

Doctors' preferences	Hospitals' preferences
d_1: h_1 h_2	h_1: $(d_1$ $d_2)$
d_2: h_1	h_2: d_1

Figure 1: An example instance I_1 of HRT.

2.2 RGS algorithm for HR

Given an instance I of HR, we now describe an algorithm, called the *Resident-oriented Gale / Shapley (RGS) algorithm for HR*, to find a stable matching in I [1]. In fact the RGS algorithm produces the unique stable matching M in I that is *doctor optimal*: that is, each assigned doctor has the best hospital that she can be assigned in any stable matching, whilst each unassigned doctor is unassigned in every stable matching [2].

The algorithm proceeds as follows. Initially M is empty. As long as there exists a doctor d_i who is unassigned and has not yet applied to every hospital on her list, d_i applies to her most-preferred hospital h_j that she has not yet applied to. If h_j is undersubscribed then d_i is provisionally assigned to h_j in M. Otherwise h_j is full. If h_j prefers d_i to its worst provisional assignee d_k then h_j rejects d_k and cancels its provisional assignment with d_k, and d_i becomes provisionally assigned to h_j instead. Otherwise h_j prefers d_k to d_i, so h_j rejects d_i. The algorithm terminates when every doctor is either assigned to a hospital or has applied to every hospital on her list.

A pseudocode description of the RGS algorithm for HR is given in Algorithm 1. The following theorem establishes the correctness of the algorithm.

Theorem 2 ([1, 2])**.** *Given an instance I of HR, all possible executions of the RGS algorithm for HR as applied to I produce, in linear time, the doctor-optimal stable matching in I.*

Doctors' preferences	Hospitals' preferences
d_1: h_2 h_1	h_1: d_1 d_3 d_2 d_5 d_6
d_2: h_1 h_2	h_2: d_2 d_6 d_1 d_4 d_5
d_3: h_1 h_3	h_3: d_4 d_3
d_4: h_2 h_3	
d_5: h_2 h_1	
d_6: h_1 h_2	

Figure 2: An example instance I_2 of HR.

We now illustrate an execution of the RGS algorithm as applied to the HR instance I_2 shown in Figure 2. Instance I_2 comprises six doctors and three hospitals, and each hospital has capacity 2. Initially M is empty and the algorithm proceeds as follows, if we assume that the doctors are considered in increasing indicial order.

1. d_1 applies to h_2 and becomes provisionally assigned to h_2 as h_2 is undersubscribed
2. d_2 applies to h_1 and becomes provisionally assigned to h_1 as h_1 is undersubscribed
3. d_3 applies to h_1 and becomes provisionally assigned to h_1 as h_1 is undersubscribed; h_1 becomes full
4. d_4 applies to h_2 and becomes provisionally assigned to h_2 as h_2 is undersubscribed; h_2 becomes full
5. d_5 applies to h_2; h_2 is full and rejects d_5 because h_2 prefers its worst assignee d_4 to d_5
6. d_5 applies to h_1; h_1 is full and rejects d_5 because h_1 prefers its worst assignee d_2 to d_5

Algorithm 1 RGS algorithm for HR

```
Require: HR instance $I$
Ensure: return the doctor-optimal stable matching $M$ in $I$
 1: $M := \emptyset$;
 2: while some doctor $d_i \in D$ is unmatched in $M$ and has not applied to all hospitals on her list do
 3:     $h_j$ := first hospital on $d_i$'s list to which $d_i$ has not yet applied;
 4:     $d_i$ applies to $h_j$;
 5:     if $h_j$ is undersubscribed then
 6:         $M := M \cup \{(d_i, h_j)\}$;
 7:     else
 8:         $d_k$ := worst doctor assigned to $h_j$;
 9:         if $h_j$ prefers $d_i$ to $d_k$ then
10:             $M := (M \cup \{(d_i, h_j)\}) \backslash \{(d_k, h_j)\}$;
11:             $h_j$ rejects $d_k$;
12:         else
13:             $h_j$ rejects $d_i$;
14:         end if
15:     end if
16: end while
17: return $M$;
```

7. d_6 applies to h_1; h_1 is full and rejects d_6 because h_1 prefers its worst assignee d_2 to d_6

8. d_6 applies to h_2; h_2 is full and rejects d_4 because h_2 prefers d_6 to its worst assignee d_4; d_6 becomes provisionally assigned to h_2

9. d_4 applies to h_3 and becomes provisionally assigned to h_3 as h_3 is undersubscribed

The algorithm now terminates with $M = \{(d_1, h_2), (d_2, h_1), (d_3, h_1), (d_4, h_3), (d_6, h_2)\}$ returned as the doctor-optimal stable matching.

2.3 Király's approximation algorithm for HRT

Király's approximation algorithm is an efficient algorithm for finding a stable matching, given an instance I of HRT, that is "close" to optimal in a precise sense. That is, it guarantees to find a stable matching M such that $|M| \geq \frac{2}{3}s^+(I)$, where $s^+(I)$ denotes the maximum size of a stable matching in I. Or, put another way, M always has size at least two-thirds of the size of a maximum size stable matching. For example, given an instance of HRT with 100 doctors where a maximum size stable matching has size 90, Király's algorithm guarantees to find a stable matching of size at least 60.

There are various versions of Király's algorithm depending on the nature of the given HRT instance [3, 4, 5], and in this assessed exercise we will be using the version for *one-sided ties* [3]. That is, we assume that ties belong to the hospitals' preference lists only, and doctors' preference lists are strictly ordered. This assumption is natural from the point of view of practical applications, as it is reasonable to expect doctors' lists to be short, and therefore doctors can choose between hospitals more easily, whereas typically hospitals' lists are very long, and hospitals may find it more difficult to distinguish strictly between their applicants.

Király's algorithm for HRT is similar to the RGS algorithm for HR, except that it allows a doctor d_i who has been rejected from every hospital on her preference list to have a "second chance" and apply to them all again in a second pass through her list. During this second pass, d_i is said to be *promoted* and has a higher priority at each hospital h_j in the sense that if d_i and some other doctor d_k are tied in h_j's list, and d_i is promoted and d_k is not, then the notion of *prefers* is extended so that h_j prefers d_i to d_k. A doctor can only be promoted once: after a second pass through her list, if she has again been rejected by every hospital on her list then

Algorithm 2 Király's approximation algorithm

```
Require: HRT instance $I$
Ensure: return a stable matching $M$ in $I$ such that $|M| \geq \frac{2}{3}s^+(I)$
 1: $M := \emptyset$;
 2: for each doctor $d_i \in D$ do
 3:    $promoted(d_i)$ := false;
 4:    $exhausted(d_i)$ := false;
 5: end for
 6: while some doctor $d_i \in D$ is unmatched and (!$promoted(d_i)$ or !$exhausted(d_i)$) do
 7:    if $exhausted(d_i)$ then
 8:       $promoted(d_i)$ = true;
 9:       $exhausted(d_i)$ = false;
10:       reactivate $d_i$;
11:    end if
12:    $h_j$ := first hospital on $d_i$'s list to which $d_i$ has not yet applied;
13:    $d_i$ applies to $h_j$;
14:    if $h_j$ is undersubscribed then
15:       $M := M \cup \{(d_i, h_j)\}$;
16:    else
17:       $d_k$ := worst doctor assigned to $h_j$;    {Any one, if there is more than one}
18:       if $h_j$ prefers $d_i$ to $d_k$ then
19:          $M := (M \cup \{(d_i, h_j)\}) \backslash \{(d_k, h_j)\}$;
20:          $h_j$ rejects $d_k$;
21:       else
22:          $h_j$ rejects $d_i$;
23:       end if
24:    end if
25:    if $d_i$ has been rejected from the last hospital on her list then
26:       $exhausted(d_i)$ = true;
27:    end if
28: end while
29: return $M$;
```

d_i will not be able to apply to any hospital again and will be unmatched in the final matching. We say that d_i is *exhausted* if d_i has been rejected from every hospital in her preference list (either during a first pass or a second pass through her list).

A pseudocode description of Király's algorithm is given in Algorithm 2. We now give an explanation of the algorithm. Initially the matching M is empty, and booleans for each doctor d_i are set to indicate that d_i has not been promoted yet and d_i is not exhausted yet. The main loop iterates as long as there is some doctor d_i who is unmatched and either d_i has not been promoted yet or d_i is not yet exhausted. If d_i is exhausted then, as we know d_i has not been promoted, we set d_i as promoted and set d_i to be not exhausted. We "reactivate" d_i, meaning that we now assume that the first hospital on d_i's list to which d_i has not yet applied is reset to the first hospital on d_i's list. Lines 12-24 of the algorithm are similar to the RGS algorithm for HR, except that two lines need to be interpreted carefully:

- Line 17: when selecting the "worst" doctor assigned to h_j, recall that promoted doctors are preferred to non-promoted doctors. For example, if h_j has capacity 2 and h_j's preference list comprises a single tie between three doctors d_1, d_2, d_3, the first two of which are assigned to h_j, where d_1 is promoted and d_2 is not, then the worst doctor assigned to h_j is d_2.

- Line 18: again recall that the notion of "prefers" has been extended to take into account promoted doctors. So if two doctors d_i and d_k are tied in h_j's preference list, and d_i has been promoted but d_k has not, then h_j prefers d_i to d_k, so d_k will be rejected. On the other hand if both doctors have been promoted, or neither doctor has been promoted, then h_j does not prefer d_i to d_k, so d_i will be rejected.

Finally, lines 25-27 set a doctor d_i to be exhausted if she has just been rejected by the last hospital on her preference list. The following theorem establishes the correctness of the algorithm.

Theorem 3 ([3]). *Given an instance I of HRT with strictly ordered doctor preference lists, all possible executions of Király's algorithm as applied to I produce, in linear time, a stable matching M in I such that $|M| \geq \frac{2}{3}s^+(I)$.*

We now illustrate an execution of Király's algorithm as applied to the HRT instance I_1 shown in Figure 1. Initially M is empty, and all doctors are set to be non-promoted and non-exhausted. The algorithm proceeds as follows, if we assume that the doctors are considered in increasing indicial order.

1. d_1 applies to h_1 and becomes provisionally assigned to h_1 as h_1 is undersubscribed
2. d_2 applies to h_1 and is rejected, as h_1 is full and does not prefer d_2 to d_1; d_2 is now exhausted
3. d_2 is promoted and reactivated, and applies again to h_1 in the second pass through her list. Hospital h_1 is full, but now prefers d_2 to d_1 as both are tied in h_1's list, but d_2 has been promoted and d_1 has not. Thus d_1 is rejected from h_1 and d_2 becomes provisionally assigned to h_1 instead
4. d_1 applies to h_2 and becomes provisionally assigned to h_2, as h_2 is undersubscribed.

The algorithm now terminates with $M = \{(d_1, h_2), (d_2, h_1)\}$, which in this case is in fact the unique maximum size stable matching.

3 Setup code

As mentioned in Section 1.4, skeleton code is provided, which includes a main class, an input file parser, relevant object classes and a dataset generator to help you test your code. Download the file `AlgII-AssEx-setup.zip` from the course web page on Moodle (see under "Assessed exercise" in the "Assessment" section. You should obtain the following Java source files:

- `Doctor.java` – a class to represent a single Doctor;
- `Hospital.java` – a class to represent a single Hospital;
- `Instance.java` – a class to represent an instance of HR or HRT;
- `Algorithm.java` – a class containing methods to execute the RGS algorithm for HR / Király's algorithm for HRT, to print out a matching and to check a matching for stability;
- `Parser.java` – a class containing methods to parse an instance and a matching from input files;
- `Main.java` – the main class containing the main method to run the program;
- `generator.jar` – a JAR file for generating an HR / HRT instance;
- `param.txt` – a parameter file for use with `generator.jar`;
- `ExampleInstance.txt` – a sample instance I_0 of HR;

- `MatchingStable.txt` – a matching that is stable in I_0;
- `MatchingUnstable.txt` – a matching that is unstable in I_0.

The program can be run in two different ways: (i) `java Main` ⟨`instance file`⟩ and (ii) `java Main` ⟨`instance file`⟩ ⟨`matching file`⟩. In case (i), given an HR or HRT instance I contained in ⟨`instance file`⟩, either the RGS algorithm for HR or Király's algorithm for HRT should be executed on I, depending on which task you are implementing. The program should check the computed matching for stability in I and print it. An example usage is `java Main ExampleInstance.txt`. In case (ii), given an HR / HRT instance I contained in ⟨`instance file`⟩ and a matching M contained in ⟨`matching file`⟩, the program should check the computed matching for stability in I and print it. An example usage is `java Main ExampleInstance.txt MatchingUnstable.txt`.

Take some time to familiarise yourself with the setup code in the `java` files. Pay specific attention to the instance variables, to see how an instance and a matching are represented internally. You will see that some methods are missing or incomplete – these are to be implemented as part of this exercise.

The file `generator.jar` is provided to enable you to generate instances of HR and HRT randomly when testing out your solutions. The generator should be executed using the following command-line syntax: `java -jar generator.jar` ⟨`parameter file`⟩ ⟨`instance file`⟩, where ⟨`parameter file`⟩ is the name of a text file containing information about the nature of the instance to be generated, and ⟨`instance file`⟩ is the name of the instance file that the generator will produce. Here is an example usage: `java -jar generator.jar param.txt instance.txt`.

The format of the parameter file is as follows:

- Line 1: number of doctors;
- Line 2: number of hospitals;
- Line 3: total capacity of the hospitals;
- Line 4: minimum length of each doctor's preference list;
- Line 5: maximum length of each doctor's preference list;
- Line 6: probability p of an entry in a hospital's list being tied with its successor.

An example parameter file is included as `param.txt`. It can be used to generate an HRT instance with 20 doctors, 5 hospitals, 20 posts in total (so each hospital has capacity 4), a preference list for each doctor that is between 3 and 5 in length, and a probability of 0.85 of each doctor being tied with its successor in a hospital's list. Note that the higher p is, the more "dense" the ties are. If $p = 0$ there are no ties, so we have an instance of HR, whereas if $p = 1$ then every hospital is indifferent between every doctor on its preference list. The generator program will produce an HR or HRT instance that conforms to the following format:

- Line 1: number of doctors n_1;
- Line 2: number of hospital n_2;
- Line $i + 2$ ($1 \leq i \leq n_1$): preference list of doctor d_i, starting with "`i: `";
- Lines $n_1 + j + 2$ ($1 \leq j \leq n_2$): preference list of hospital h_j, starting with "`j: cj: `", where c_j is the capacity of h_j.

Here is a small example HRT instance to illustrate the above format:

```
10
3
1: 3
2: 2 3 1
3: 2 3 1
4: 3 2 1
5: 3 1
6: 2
7: 2 3
8: 1 2 3
9: 1
10: 3 2 1
1: 4: (5 10) 9 (4 3 8 2)
2: 3: (3 7) (2 8 6 10 4)
3: 3: (7 4 10 5 2) (8 3 1)
```

The two matching files `MatchingStable.txt` and `MatchingUnstable.txt` are both given as a series of assigned doctor–hospital pairs (one per line).

4 Implementation tasks

There are four separate implementation tasks as part of this exercise. Each of these involves extending the skeleton code provided. In doing so, you can modify the structure of provided code as you wish (changing existing classes / instance variables / methods and adding new classes / instance variables / methods are permitted). However, you should not modify the parts of the `Main` class that deal with command-line arguments – this will ensure that the marker can execute your program as described under each of the following tasks. To complete this exercise it should not be necessary to add any new classes, but you will not be penalised if you choose to.

4.1 Task 1

Complete the method `printMatching` in `Algorithm.java` so that the matching that has been computed by the `run` method or read in from the matching file is printed to the console. The output should contain one line for each doctor d_i, showing which hospital d_i is assigned to, or else indicating that d_i is unmatched. Finally, the size of the matching should be displayed. Here are some of the lines of the required output from `printMatching` when executing `java Main ExampleInstance.txt MatchingUnstable.txt`:

```
Matching:
Doctor 1 is assigned to hospital 3
Doctor 2 is assigned to hospital 5
...
Doctor 9 is assigned to hospital 5
Doctor 10 is unmatched
...
Doctor 20 is assigned to hospital 3
Matching size: 18
```

4.2 Task 2

Complete the body of the `run` method in `Algorithm.java` in order to implement the RGS algorithm for HR. This will also involve adding and modifying code in other classes.

The worst-case complexity of the RGS algorithm for HR is $O(L)$, where L is the total length of the doctors' preference lists (or equivalently, the total length of the hospitals' preference lists). You should ensure that your implementation respects this theoretical worst-case complexity as far as possible. In order to achieve this, the following hints may be useful:

- To determine in $O(1)$ time whether a hospital h_j prefers one doctor to another, it is recommended to use h_j's ranking list, stored in `rankList`, as described in the lectures in connection with the Stable Marriage problem.
- To implement Line 8 of the RGS algorithm for HR efficiently, you may find the variable `rankOfWorstAssignee` in `Hospital.java` to be useful. For a given hospital h_j, this variable should initially point to the end of h_j's preference list. When h_j becomes full, it remains full and its worst assignee can never get worse. Hence this pointer can traverse from right to left during the algorithm's execution and never needs to move right. The pointer `rankOfWorstAssignee` can be updated when the identity of h_j's worst assigned doctor is required.

4.3 Task 3

Place your code from Tasks 1 and 2 in a folder called `HR`. Take a copy of your `HR` folder and rename the copied folder `HRT`. Your implementation of Task 3 should be carried out within the `HRT` folder to keep it separate from your solution for `HR`. Task 3 involves modifying the body of the `run` method in `Algorithm.java` in order to implement Király's approximation algorithm for HRT. This will also involve adding and modifying code in other classes, particularly `Doctor.java`, where you will need to add booleans that correspond to whether a doctor has been promoted / is exhausted.

Try to aim for an $O(L)$ implementation of Kiraly's approximation algorithm for HRT if you can. In order to achieve this, the following hints may be useful:

- Again, the concept of a ranking list should be helpful to enable a hospital to determine in $O(1)$ time whether it prefers one doctor to another. In the presence of ties, this requires a little explanation. Suppose a hospital h_j's preference list is d_1 $(d_2\ d_3)$ d_4. Then the ranks of d_1, d_2, d_3, d_4 h_j's list could be set to be either $1, 2, 2, 3$ or $1, 2, 2, 4$ respectively (it should not matter which). The parser code that you have been given assumes the former, so that the rank of a doctor d_i in a hospital h_j's list is the value of $k \geq 1$ such that d_i appears in the kth-most preferred tie in h_j's list, where a tie can be of length 1 for this purpose.
- Determining whether a hospital prefers one doctor to another will require the modified definition of *prefers*. Similarly determining the worst assignee of a hospital will require care. To do this, you may find it useful to maintain, for each hospital a list of doctors assigned at each rank position (if any), with promoted doctors coming before unpromoted doctors in this list. There should never need to be a linear search through such a list of assignees.

4.4 Task 4

Within your `HR` folder, complete the method `checkStability` in `Algorithm.java` so that, for a given instance of HR, the matching that has been computed by the `run` method or read in from the matching file is tested for stability. If there are any blocking pairs, these should be output one per line to the console. Finally, a single line should be output confirming whether the matching is stable. Here is the required output from `checkStability` when executing `java Main ExampleInstance.txt MatchingUnstable.txt`:

```
Blocking pair between doctor 11 and hospital 5
Blocking pair between doctor 11 and hospital 3
Blocking pair between doctor 11 and hospital 1
Blocking pair between doctor 15 and hospital 5
Matching is not stable
```

Your implementation of `checkStability` in the case of `HR` should have $O(L)$ worst-case complexity, where L is the total length of the doctors' preference lists.

Next, within your `HRT` folder, complete the method `checkStability` in `Algorithm.java` so that, for a given instance of HRT, the matching that has been computed by the `run` method or read in from the matching file is tested for stability. The output should be analogous to the example given above. Note that when you check for the stability of a given matching in a given instance of HRT, the notion of "prefers" is as defined in Section 2.1 (i.e., it has nothing to do with whether a doctor has been promoted or not). Again, try to aim for $O(L)$ worst-case complexity for your implementation of `checkStability` in the case of HRT.

5 How to submit

Your submission to this exercise should include your source code for Tasks 1-4 and two code listing files (in pdf form). All submissions are to be made electronically and no hard-copy submissions are required. More information about each of these components is given as follows:

- *Source code*: Ensure that each of the sub-folders `HR` and `HRT` contain your source files and compiled class files. Do not use packages or separate `bin` and `src` sub-folders. It should be possible for your code to be executed without the need to compile it.
- *Code listing files*: Produce two formatted code listing files in pdf form for the source files in each of your `HR` and `HRT` sub-folders. In order to create the code listing files, the preferred option is to use the following Unix commands within each of your `HR` and `HRT` sub-folders, but this is not obligatory:

  ```
  a2ps -A fill -Ma4 *.java -o codeListingX.ps
  ps2pdf -sPAPERSIZE=a4 codeListingX.ps
  ```

  where `X` is `HR` or `HRT`.
- *Status report*: this should contain the following information:
  - any external sources you have used (standard `Java` classes need not be mentioned);
  - any assumptions you have made during implementation, if applicable;
  - any known issues with the code, and (if possible) how they could be addressed (if you believe the code is working, all that is required is one sentence indicating this).

In order to make your submission, follow the submission link under "Assessed exercise" in the "Assessment" section of the course web page on Moodle . You will need to submit a zip file or tar.gz file which should be named `AlgII_<family name>_<given name>.zip` or `AlgII_<family name>_<given name>.tar.gz`, e.g., `AlgII_Manlove_David.zip`.

The zip or tar.gz file should contain your source code in the sub-folders `HR` and `HRT` as described above, and your two code listing files and status report in the top-level folder. Before you submit, ensure that your zip/tar.gz file contains the version of the files that you wish to have assessed. You can submit as many times as you wish; the last submission made before the exercise deadline will be the one that is used, and your code at the time of that submission should correspond to the code listing pdf files.

You will be required to complete a Declaration of Originality when submitting via Moodle. For the purposes of this exercise, the declarations that you make apply to all parts of your submission.

Task	Description	Marks
1	Print matching	2
2	RGS algorithm for HR – correctness	3
2	RGS algorithm for HR – efficiency	2
3	Király's approximation algorithm for HRT – correctness	4
3	Király's approximation algorithm for HRT – efficiency	3
4	Check stability (HR) – correctness and efficiency	3
4	Check stability (HRT) – correctness and efficiency	1
	Quality of code (commenting, modularity etc)	2
	Total	20

Table 1: Breakdown of marks for the exercise

6 Marking scheme

Upon submission, your solution will be executed against various acceptance tests. Each of these will be run using either `java Main ⟨instance file⟩` or `java Main ⟨instance file⟩ ⟨matching file⟩`. If there are any run-time errors you will lose marks.

Table 1 shows the breakdown of marks for the different parts of the exercise. This is intended to be a guide and the exact breakdown may be adjusted slightly when it comes to the marking. Your final mark will be converted to a percentage and then to an overall band (on the 23-point scale) for the exercise via the School's standard percentage-band translation table

Final remark: this specification document may appear lengthy, but on the other hand you will hopefully find that the actual volume of code that is required in order to gain full marks on each of the separate tasks is not especially large! The aim in this specification has been to try to explain the problems and algorithms that are involved, and what is expected of you in terms of your solutions, as clearly as possible.

References

[1] D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–15, 1962.

[2] D. Gusfield and R.W. Irving. *The Stable Marriage Problem: Structure and Algorithms.* MIT Press, 1989.

[3] Z. Király. Better and simpler approximation algorithms for the stable marriage problem. In *Proceedings of ESA '08: the 16th Annual European Symposium on Algorithms*, volume 5193 of *Lecture Notes in Computer Science*, pages 623–634. Springer, 2008.

[4] Z. Király. Better and simpler approximation algorithms for the stable marriage problem. *Algorithmica*, 60:3–20, 2011.

[5] Z. Király. Linear time local approximation algorithm for maximum stable marriage. *Algorithms*, 6(3):471–484, 2013.

[6] D.F. Manlove. *Algorithmics of Matching Under Preferences.* World Scientific, 2013.

[7] D.F. Manlove, R.W. Irving, K. Iwama, S. Miyazaki, and Y. Morita. Hard variants of stable marriage. *Theoretical Computer Science*, 276(1-2):261–279, 2002.