University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	Algorithmics II (H)			
Coursework Number	1 of 1			
Deadline	Time:	4.30pm	Date:	15 November 2019
% Contribution to final course mark	20%		This should take this many hours:	20
Solo or Group ✓	Solo	✓	Group	
Submission Instructions	Via Moodle – see Sections 13 and 14			
Marking Criteria	See Section 15			
Please Note: This coursework cannot be redone				

Algorithmics II (H)

Assessed Exercise, 2019-20

Matching final-year students to projects

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 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/mod/resource/view.php?id=1353537`).

2. Deadline for submission

The hand-out date for the exercise is Friday 25 October 2019, by which time all the relevant material will have been covered in lectures. The deadline for submission is **4.30pm, Friday 15 November 2019**. 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 13. The aim will be to return marked exercises with individual feedback by Friday 6 December 2019. In addition, general feedback on the exercise will be provided via the course web page on Moodle.

3. Specification

In a Computing Science department, it is proposed to carry out the annual allocation of final-year students to projects by an automated matching scheme, taking into account the projects that are acceptable to each student, and the capacities of projects and lecturers. This matching problem can be solved using an application of network flow, which is the main focus of this exercise. The exercise has three parts: (a) complete an implementation of the Ford-Fulkerson algorithm for finding a maximum flow in a network, (b) design an appropriate network and demonstrate how it may be used in order to find the desired matching of students to projects, and (c) extend your code to handle lower quotas for the lecturers. Some skeleton code is provided and this should be extended in order to complete your solutions to the various parts of the exercise. The given code is written in Java and it is expected that this will be the language of implementation for your solutions. There is also a JUnit testing framework that is provided, which you will need to extend; to execute the tests it is recommended that you use the Eclipse IDE.

4. Getting started

On the course web page (`https://moodle.gla.ac.uk/course/view.php?id=971`), click on the link "Setup files" under the heading "Assessed Exercise". You will obtain a zip file entitled `AlgII_Setup_Files.zip`. After unzipping this file you will obtain a folder `AlgII_Setup_Files` which contains the following three subfolders: `Ass_Ex_Part_A`, `Ass_Ex_Part_B` and `Ass_Ex_Part_C`.

There are four tasks to complete as part of this exercise. Your code for Tasks 1 and 2 will be written within the subfolder `Ass_Ex_Part_A`, your code for Task 3 within the subfolder `Ass_Ex_Part_B` and your code for Task 4 within the subfolder `Ass_Ex_Part_C`.

5. More about the setup code

Folder `Ass_Ex_Part_A` should contain:

- a file `tasks_1-2_example.txt` containing a representation of a sample network (the format of this file is described in detail below);
- a file `pom.xml` (Project Object Model) for Maven;
- a folder `src` for the source code, with subfolders `main` and `test`;
    - subfolder `main/java/matching` contains:
        - package `matching.networkFlow` that contains several relevant classes, some of which are incomplete;
        - a class `FordFulk` that contains a place-holder for an implementation of the Ford-Fulkerson algorithm, some I/O code, and a method to run the algorithm;
        - a class `Main` that contains the `main` method, and is the entry point of the program;
    - subfolder `main/test` contains:
        - JUnit tests in subfolder `java/matching`;
        - `tasks_1-2_example.txt` in subfolder `resources`.

The given incomplete classes are to be developed into your solutions to Tasks 1 and 2 of this exercise.

Launch Eclipse and then select "File" → "Import" → "Existing Maven Project" and select folder `Ass_Ex_Part_A`. To run the code as it stands, right-click on the project from Package Explorer and choose "Run As" → "Run Configurations". Make sure you are editing a run configuration under "Java Application" and then enter the input filename `tasks_1-2_example.txt` in the arguments tab before pressing "Run". You should see some (incomplete) output.

To run the JUnit tests, right-click on the project from Package Explorer and choose "Run As" → "JUnit Test". The six given tests should execute, but you should find that only one passes, whilst two fail and three produce null pointer exceptions. Once you have developed your solution to Part A, you should find that all tests pass. You will also need to write more JUnit tests of your own.

N.B. If Eclipse is giving strange error messages, try the following: under "Project" → "Properties" → "Java Compiler", tick "Enable project specific settings". Ensure that "Use '`--release' option`' is unticked. Choose 11 for "Compiler compliance level".

Each class name in the package `networkFlow` is hopefully self-explanatory: the given class provides fields related to the entity (i.e., vertex / edge / directed graph, etc.) to which the class name refers and also provides methods that perform operations on the given entity. The class `Network` is a subclass of `DirectedGraph`, inheriting from the latter. Similarly the class `ResidualGraph` is a subclass of `Network`, again inheriting from the latter. Take some time to familiarise yourself with the given code. You will see that some methods have been left incomplete; as mentioned previously their completion forms the basis of the various tasks making up this exercise.

Folder `Ass_Ex_Part_B` should contain only an example input file `task_3_example.txt`. Upon completion of Part A of the exercise, you should copy the whole `src` folder and the file `pom.xml` from the `Ass_Ex_Part_A` folder into the `Ass_Ex_Part_B` folder, and these will form the starting point for your solution to Task 3.

Folder `Ass_Ex_Part_C` should contain only an example input file `task_4_example.txt`. Upon completion of Part B of the exercise, you should copy the whole `src` folder and the file `pom.xml` from the `Ass_Ex_Part_B` folder into the `Ass_Ex_Part_C` folder, and these will form the starting point for your solution to Task 4.

Note that Tasks 1 and 2 of this exercise should be completed without changing class declarations and completed methods. For these two tasks the only modifications should involve *completion* of the relevant methods and *addition* of JUnit tests. For Task 3, the class `FordFulk` will need to be modified, though none of the existing classes in the `networkFlow` package need be changed. (Additional classes may be added to this package if desired, but no additional packages are necessary.) A small penalty may be applied if any other modifications have been made. For Task 4, no additional classes are necessary, but existing classes will need to be modified. Some hints as to which modifications suffice are given in Section 11.

6. Part A, Task 1

This task involves the completion of the following methods in the `Network` class:

```
/**
 * Returns true if and only if the assignment of integers to the
 * flow fields of each edge in the network is a valid flow.
 * @return true, if the assignment is a valid flow
 */
public boolean isFlow() {
       // complete this method as part of Task 1
       return true;
}

/**
 * Gets the value of the flow.
 * @return the value of the flow
 */
public int getValue() {
       // complete this method as part of Task 1
       return 0;
}

/**
 * Prints the flow.
 * Display the flow through the network in the following format:
 * (u,v) c(u,v)/f(u,v)
 * where (u,v) is an edge, c(u,v) is the capacity of that edge and
 * f(u,v) is the flow through that edge - one line for each edge in
 * the network
 */
public void printFlow() {
       // complete this method as part of Task 1
}
```

7. Part A, Task 2

This task involves the implementation of the Ford-Fulkerson algorithm for finding a maximum flow in a given network. Specifically, the following methods are to be completed in the `ResidualGraph`, `Network` and `FordFulk` classes:

```
/**
 * Instantiates a new ResidualGraph object.
 * Builds the residual graph corresponding to the given network net.
 * Residual graph has the same number of vertices as net.
 * @param net the network
```

```
         */
        public ResidualGraph (Network net) {
                super(net.numVertices);
                // complete this constructor as part of Task 2
        }

        /**
         * Find an augmenting path if one exists.
         * Determines whether there is a directed path from the source to the sink
           in the residual graph -- if so, return a linked list containing the
           edges in the augmenting path in the form (s,v_1), (v_1,v_2), ...,
           (v_{k-1},v_k), (v_k,t); if not, return an empty linked list
         * @return the linked list
         */
        public LinkedList<Edge> findAugmentingPath () {
                // complete this method as part of Task 2
                return null;
        }

        /**
         * Calculates by how much the flow along the given path can be increased,
         * and then augments the network along this path by this amount.
         * @param path a list of edges along which the flow should be augmented
         */
        public void augmentPath(List<Edge> path) {
                // complete this method as part of Task 2
        }

        /**
         * Executes Ford-Fulkerson algorithm on the constructed network net.
         */
        public void fordFulkerson() {
                // complete this method as part of Task 2
        }
```

The input file for the method `readNetworkFromFile` in class `FordFulk` is assumed to be a text file with *n* lines, where *n* is the number of vertices in the network (including the source and the sink). The source is assumed to have label 0, whilst the sink is assumed to have label *n*-1. The first line gives the value of *n*. There then follow *n*-1 lines, where the (*j*+1)th line in this group ($0 \leq j \leq n-2$) has the following format:

$j\ i_1\ (c(j,i_1))\ i_2\ (c(j,i_2))\ \ldots\ i_r\ (c(j,i_r))$

The integers i_1, i_2, …, i_r refer to the vertices that belong to the adjacency list of vertex *j*. The integer $c(j, i_k)$ gives the capacity of the edge (j, i_k) ($1 \leq k \leq r$). All integers on the same line (whether enclosed in brackets or not) are assumed to be space-separated. Here is an example input file corresponding to the network introduced in lectures:

```
6
0 1 (2) 3 (4)
1 2 (1) 4 (3)
2 5 (2)
3 2 (3) 4 (1)
4 3 (1) 5 (4)
```

The above example input file is provided as `tasks_1-2_example.txt`. The output as given by the methods that you implemented as part of Tasks 1 and 2 should be the following:

```
The assignment is a valid flow
A maximum flow has value: 5
The flows along the edges are as follows:
(0,1) 2/2
(0,3) 4/3
(1,2) 1/0
```

```
(1,4) 3/2
(2,5) 2/2
(3,2) 3/2
(3,4) 1/1
(4,3) 1/0
(4,5) 4/3
```

Note that your solution need not be concerned with validation of the input file – this can be assumed to be in the specified format.

8. Part A, JUnit tests

To test out your solution to Tasks 1 and 2, add at least two non-trivial JUnit tests within the folder `src/test/java/matching` by modifying existing tests classes, and ensure that your new tests pass, along with the existing ones. Ensure that your new JUnit tests are contained in files named according to the convention "`StudentTest<ClassName>.java`" to avoid any name conflicts with the JUnit tests that we will add at the marking stage.

9. Part B, Task 3

As described above, copy your code from the `Ass_Ex_Part_A` folder to the `Ass_Ex_Part_B` folder. Some parts of the code will be modified further during the completion of this task.

This part of the exercise is concerned with the application of network flow to the problem of assigning students to projects. We assume that each student has a set of acceptable projects. For simplicity we assume that he/she does not explicitly rank this set in order of preference (the algorithms used here can be extended if we wish to take explicit preferences into account, but that is beyond the scope of this exercise). Additionally, a given project may be suitable for more than one student to work on simultaneously, though each project has a capacity indicating the maximum number of students that could be assigned to it. Similarly each lecturer has a capacity indicating the maximum number of students that he/she is willing to supervise (so for example, to give students a choice, lecturer 2 might offer projects 4, 7 and 10 with capacities 2, 1 and 3 respectively, though in practice he/she is only willing to supervise at most 5 students). We also take into account that Software Engineering (SE) students can only do projects that are marked as being suitable for this cohort.

Formally, let $X=\{x_1, x_2,\dots, x_n\}$ be a set of students, let $P=\{p_1, p_2,\dots, p_m\}$ be a set of projects and let $L=\{l_1, l_2,\dots, l_q\}$ be a set of lecturers. Each student $x_i \in X$ finds acceptable a non-empty set of projects $A_i \subseteq P$ and x_i is tagged according to whether he/she is an SE student. Each project $p_j \in P$ is offered by exactly one lecturer in L and is tagged according to whether p_j is suitable for SE students. Furthermore, each project $p_j \in P$ has a capacity denoted by c_j, whilst each lecturer $l_k \in L$ has a capacity denoted by d_k. The objective is to find an assignment M of students to projects such that:

(i) each student in X is assigned to at most one project in M ;
(ii) if student $x_i \in X$ is assigned to a project p_j in M, then $p_j \in A_i$ (i.e., x_i finds p_j acceptable) ;
(iii) if an SE student $x_i \in X$ is assigned to a project p_j in M, then p_j is suitable for SE students ;
(iv) each project $p_j \in P$ is assigned at most c_j students in M ;
(v) for each lecturer $l_k \in L$, at most d_k students are assigned in M to projects offered by l_k ;
(vi) subject to conditions (i)-(v), as many students are assigned to projects in M as possible.

The input file for this task is assumed to be a text file with $n+m+q+3$ lines, where n is the number of students, m is the number of projects and q is the number of lecturers. The first, second and third lines give the values of n, m and q respectively. The next n lines give information about the students. That is, for each i ($1 \le i \le n$), line $3+i$ has the following format:

```
i Si pr1 pr2 ... prt
```

where S_i is a single character that is 'Y' if *i* is an SE student, and 'N' otherwise. The integers p_{r_1}, p_{r_2},...., p_{r_t} correspond to the projects that *i* finds acceptable (assume that $t \geq 1$). (Note that it is possible that an SE student could inadvertently list projects that are not suitable for the SE cohort.) The next *m* lines give information about the projects. That is, for each *j* ($1 \leq j \leq m$) line 3+*n*+*j* has the following format:

```
j Tj lzj cj
```

where T_j is a single character that is 'Y' if *j* is a project suitable for an SE student, and 'N' otherwise. Integer l_{z_j} indicates the lecturer who offers project *j*, whilst c_j gives the capacity of *j* (assume that $c_j \geq 1$). The next *q* lines give information about the lecturers. That is, for each *k* ($1 \leq k \leq q$) line 3+*n*+*m*+*k* has the following format:

```
k dk
```

where d_k gives the capacity of lecturer *k* (assume that $d_k \geq 1$). All integers on the same line are assumed to be space-separated. Here is an example input file for this part of the exercise:

```
7
8
3
1 Y 1 7
2 N 1 2 3 4 5 6
3 N 2 1 4
4 N 2
5 N 1 2 3 4
6 N 2 3 4 5 6
7 N 5 3 8
1 Y 1 2
2 Y 1 1
3 Y 1 1
4 Y 2 1
5 Y 2 1
6 Y 2 1
7 N 3 1
8 Y 3 1
1 3
2 2
3 2
```

The above example input file is provided as `task_3_example.txt`. Extend your code from Tasks 1 and 2, in order to take as input a file in the above format and construct an assignment of students to projects satisfying conditions (i)-(vi) above. The output from your program should have the following format. The first *n* lines should state, for each student *i*,

```
Student i is assigned to project j, or
Student i is unassigned,
```

as appropriate. The next *m* lines should state, for each project *j*,

```
Project j with capacity cj is assigned aj student/s
```

The next *q* lines should state, for each lecturer *k*,

```
Lecturer k with capacity dk is assigned bk student/s
```

An example output corresponding to the above input is as follows (note that the following solution is not unique, and other solutions are possible):

```
Student 1 is assigned to project 1
Student 2 is assigned to project 1
Student 3 is assigned to project 2
Student 4 is unassigned
Student 5 is assigned to project 4
Student 6 is assigned to project 5
Student 7 is assigned to project 8

Project 1 with capacity 2 is assigned 2 students
Project 2 with capacity 1 is assigned 1 student
Project 3 with capacity 1 is assigned 0 students
Project 4 with capacity 1 is assigned 1 student
Project 5 with capacity 1 is assigned 1 student
Project 6 with capacity 1 is assigned 0 students
Project 7 with capacity 1 is assigned 0 students
Project 8 with capacity 1 is assigned 1 student

Lecturer 1 with capacity 3 is assigned 3 students
Lecturer 2 with capacity 2 is assigned 2 students
Lecturer 3 with capacity 2 is assigned 1 student
```

In extending your code from Tasks 1 and 2, you may find it helpful to create `Student`, `Project` and `Lecturer` subclasses of the `Vertex` class; no other modifications to the `networkFlow` package (and no additional packages) are necessary. The class `FordFulk` will, however, require modification to various parts, including the input/output mechanism.

Again your solution need not be concerned with validation of the input file – this can be assumed to be in the specified format.

10. Part B, JUnit tests

To test out your solution to Task 3, add at least two non-trivial JUnit tests within the folder `src/test/java/matching` by modifying existing tests classes, and ensure that your new tests pass, along with the existing ones. Again, ensure that your new JUnit tests are contained in files named according to the convention "`StudentTest<ClassName>.java`" to avoid any name conflicts with the JUnit tests that we will add at the marking stage.

11. Part C, Task 4

As described above, copy your code from the `Ass_Ex_Part_B` folder to the `Ass_Ex_Part_C` folder. Some parts of the code will be modified further during the completion of this task.

This part of the exercise is concerned with the modifying your code from Part B to take account of load balancing constraints on the part of the lecturers. Each lecturer l_k now has a *lower quota* q_k, a positive integer indicating the minimum number of students that must be assigned to l_k in any valid assignment. Lecturer l_k's capacity, denoted by d_k, is now also known as l_k's *upper quota*. The definition of a valid assignment of students to projects as given by criteria (i)-(vi) in Section 9 changes so that, for any lecturer l_k, the number of students assigned to l_k must be at least q_k and at most d_k. Specifically, criterion (v) changes as follows (all other criteria are unchanged):

(v) For each lecturer $l_k \in L$, at least q_k and at most d_k students are assigned in M to projects offered by l_k ;

An assignment of students to projects that satisfies the lecturers' lower quotas may not exist. The goal of this task is to extend your code from Part B in order to find an assignment or report that none exists.

For the purposes of this task, the input file format changes so that information about each lecturer's lower quota is provided. Specifically, the file format is exactly the same as for Task 3, except that the last q lines contain information about the lecturers' lower and upper quotas. That is, for each k ($1 \leq k \leq q$) line $3+n+m+k$ has the following format:

```
k qk dk
```

where q_k gives the lower quota of lecturer k (assume that $q_k \geq 0$) and d_k gives the upper quota of lecturer k (assume that $d_k \geq \max\{q_k, 1\}$). All integers on the same line are assumed to be space-separated. Here is an example input file for this part of the exercise:

```
7
8
3
1 N 5 3 8
2 N 2 3 4 5 6
3 N 1 2 3 4
4 N 2 7
5 N 2 1 4
6 N 1 2 3 4 5 6
7 Y 1 7
1 Y 1 2
2 Y 1 1
3 Y 1 1
4 Y 2 1
5 Y 2 1
6 Y 2 1
7 N 3 1
8 Y 3 1
1 1 3
2 3 3
3 2 2
```

The above example input file is provided as `task_4_example.txt`. Extend your code from Task 3, in order to take as input a file in the above format and find a valid assignment of students (satisfying conditions (i)-(iv) from Section 9, condition (v) from this section and condition (vi) from Section 9) or report that none exists. The output should be similar to that indicated in Section 9, except that the last q lines should state, for each lecturer k ($1 \leq k \leq q$):

```
Lecturer k with lower quota qk and upper quota dk is assigned bk student/s
```

An example output corresponding to the above input is as follows:

```
Student 1 is assigned to project 8
Student 2 is assigned to project 5
Student 3 is assigned to project 2
Student 4 is assigned to project 7
Student 5 is assigned to project 4
Student 6 is assigned to project 6
Student 7 is assigned to project 1

Project 1 with capacity 2 is assigned 1 student
Project 2 with capacity 1 is assigned 1 student
Project 3 with capacity 1 is assigned 0 students
Project 4 with capacity 1 is assigned 1 student
Project 5 with capacity 1 is assigned 1 student
Project 6 with capacity 1 is assigned 1 student
Project 7 with capacity 1 is assigned 1 student
Project 8 with capacity 1 is assigned 1 student
```

```
Lecturer 1 with lower quota 1 and upper quota 3 is assigned 2 students
Lecturer 2 with lower quota 3 and upper quota 3 is assigned 3 students
Lecturer 3 with lower quota 2 and upper quota 2 is assigned 2 students
```

If we were to raise the lower and upper quota of lecturer 3 to 3, the output would be as follows:

```
No assignment exists that meets all the lecturer lower quotas
```

Notes

1. This part of the exercise is more challenging, but the code required to extend your solution from Task 3 to a solution to Task 4 is relatively small. In particular, the following modifications suffice:

   - 7 lines added to / changed in class `Main`
   - 18 lines added to / changed in class `FordFulk`
   - 12 lines added to / changed in class `Lecturer`
   - 3 lines added to / changed in class `Network`

   The above list is merely intended to be a guide and of course there is a range of different ways of doing this. However this information is intended to indicate that lengthy alterations are definitely not necessary!

2. The number of marks available for Task 4 is deliberately relatively small. Therefore if you do not see how to do it after, say, an hour, it may be better not to spend any longer on this task and instead to move on to other assessments, project work or consolidating lecture material.

12. Part C, JUnit tests

To test out your solution to Task 4, add at least one non-trivial JUnit test within the folder `src/test/java/matching` by modifying existing tests classes, and ensure that your new tests pass, along with the existing ones. Again, ensure that your new JUnit tests are contained in files named according to the convention "`StudentTest<ClassName>.java`" to avoid any name conflicts with the JUnit tests that we will add at the marking stage.

13. What to submit

Your submission to this exercise should include your source code, code listing files (in pdf form) and an implementation report (written, e.g., in Word or LaTeX). 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*: include all of your .java files, .class files and any test files used for each of Parts A, B and C in three different sub-folders corresponding to each part. Be sure to submit *all* of the classes in each case, and not just the ones that you have altered. Also, ensure that you remove any debugging code that may generate large volumes of output on large input files.

- *Code listing files*: include three pdf files, containing formatted listings of the classes you have changed for each of Parts A, B and C. These files should include the following code listings:
  - For Part A, `FordFulk.java`, `Network.java` and `ResidualGraph.java`, together with the classes for any JUnit tests that you have added;
  - For Part B, `FordFulk.java` and any other classes added to the `networkFlow` package, together with the classes for any JUnit tests that you have added;
  - For Part C, any classes that have been modified, together with the classes for any JUnit tests that you have added.

In order to produce the pdf file for Part X, you should use the following Unix commands (e.g., on host sibu):

```
a2ps -A fill -Ma4 filename1.java filename2.java (etc) -o codeListingPartX.ps
ps2pdf -sPAPERSIZE=a4 codeListingPartX.ps
```

- *Implementation report*: this should include a status report for each task which, for any non-working implementation, should state clearly what happens on compilation (in the case of compile-time errors) or on execution (in the case of run-time errors). It should also explain the implementation of each of the methods that you completed in Tasks 1-4 above, making appropriate reference to the theory of network flow as covered in lectures. For Task 3, you should explain how the network was constructed to solve the project allocation problem, and for Task 4 you should describe the alterations that you made in order to cope with lecturer lower quotas.

  This document can be organised by giving headings for each of Tasks 1-4, and for each task the sub-headings "status report" and "justification of operation" can be included. The implementation explanations for each task need not (and should not) be lengthy: they will typically range from a few sentences for Task 1 up to at most half a page for each of Tasks 3 and 4. However you should avoid saying things like "I implemented the algorithm as per the lecture slides" – you need to demonstrate an understanding of the underlying concepts in all cases.

14. How to submit

In order to make your submission, follow the submission link in the section entitled "Assessed Exercise" within the Moodle page (`https://moodle.gla.ac.uk/course/view.php?id=971`) for the course. You will need to submit a zip file or tar.gz file containing your files listed in Section 13, 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 be organised as follows:

- The top-level folder should include your implementation report.
- There should be three sub-folders, named `Ass_Ex_Part_A`, `Ass_Ex_Part_B` and `Ass_Ex_Part_C`, each of which should include your .java, .class and test files and the code listing pdf file for the corresponding part of the exercise.

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. If you have used any external sources, be sure to acknowledge them in your implementation report.

After the deadline, the submissions will be run through the School's in-house plagiarism detection program. As part of the marking, your code for each of Parts A, B and C will subject to JUnit testing, which will include the provided tests, the tests you added and some additional tests.

15. Marking scheme

Submissions will be marked according to the following breakdown of numerical marks:

Completion of three methods in the `Network` class for Task 1 (*)	(4)
Completion of two methods in the `ResidualGraph` class and one method in the `FordFulk` class for Task 2 (*)	(9)
Modification of the `FordFulk` class and addition of classes to the `networkFlow` package (if necessary) for Task 3 (*)	(7)
Modifications to allow for lecturer lower quotas for Task 4 (*)	(3)
Implementation reports for Tasks 1, 2, 3 and 4	(10)
Addition of your own JUnit tests for Tasks 1, 2, 3 and 4	(5)
Quality of code (i.e. layout, comments etc.) and general presentation	(2)
Total	**(40)**

Numerical marks will then be converted to bands according to the School's standard percentage-band translation table.

(*) these marks will be awarded on the basis of correctness, efficiency and indication of understanding of the underlying theory of network flow.