The University of Melbourne
School of Computing and Information Systems
COMP90041 Programming and Software Development
Lecturer: Prof. Rui Zhang
Semester 1, 2019

Project C
Due: 3pm, Friday 31 May, 2019

1 Introduction

The aim of this project is to add some more advanced features to the system developed in Project B. The features to be added are:

- Sort the players with more specific rules
- Handling of invalid input via Exceptions
- Write (read) the game statistics into (from) a file, i.e., one which is stored on the hard disk between executions of Nimsys
- A new type of player - an AI (Artificial Intelligence) player, whose moves are automatically determined by the computer rather than a game user
- A victory guaranteed strategy for the AI player
- An advanced Nim game and a victory guaranteed strategy for the AI player in this new game (**for bonus marks**)

The system should still operate as specified in Project B, but with additional functionality, due to the addition of the aforementioned features. Thus, it is advised that you use your Project B solution as a starting point for implementing Project C.

Knowledge Coverage

- Exceptions, covered in Week 9 lecture
- File I/O operations, covered in Week 10 lecture
- Polymorphism, you may use either inheritance or interface to design the AI player in addition to the human player; the corresponding concepts are covered in Week 8 lecture

2 Requirements

In following description, all command line displays are put in a box. This is only for easier understanding the format. **The box should NOT be printed out by your program, only the contents in the box should be printed.** The command prompt is illustrated below:

2.1 Sort the players with more specific rules

In Project B, when resolving the ties of winning ratio, you are required to sort the player by the username in either ascending alphabetical order or descending alphabetical order.

In Project C, **ranking should still consider winning ratio first**. When resolving the ties of winning ratio, you are required to sort the player by the **username** in **ONLY** ascending alphabetical order, for all the commands.

Example Execution:
Suppose we have three players in (username, First Name, Last Name) format as follow: (LS, Luke, Skywalker), (HS, Han, Solo), (DV, Darth, Vader). They all have the same wining ratio.

1. rank all users (default, i.e., descending order)

```
$rankings
0%  | 00 games | Darth Vader
0%  | 00 games | Han Solo
0%  | 00 games | Luke Skywalker

$
```

2. rank all users in descending order

```
$rankings desc
0%  | 00 games | Darth Vader
0%  | 00 games | Han Solo
0%  | 00 games | Luke Skywalker

$
```

3. rank all users in ascending order

```
$rankings asc
0%  | 00 games | Darth Vader
0%  | 00 games | Han Solo
0%  | 00 games | Luke Skywalker

$
```

2.2 Invalid input handling via Exceptions

The system should check inputs for validity. For this task, you will not be required to implement exception handling for all possible invalid inputs - just a subset of them. The range of potential invalid inputs you are **required** to address by Exceptions (**not** via if-then statements) are listed below, along with the required behaviour of your program. Note that some of this input checking was also a requirement in Project B. Where this is the case, you need to modify your code so that the invalid input is handled via Exceptions. The rest of the invalid input handling cases described in Project B do **not** need modification.

- Invalid command - The user enters a command which is not a valid Nimsys command. Here, invalid command suggests the input command is not among the specified commands, i.e., `addplayer`, `editplayer`, `removeplayer`, `displayplayer` `resetstats`, `rankings`, `startgame`, and `exit`.
  Example:

```
$createplayer lskywalker,Skywalker,Luke
'createplayer' is not a valid command.

$
```

- Invalid number of arguments - The user enters a valid Nimsys command, but does not provide the correct number of arguments. **Note:** You only need to check for insufficient number of arguments, and simply ignore any extra arguments, i.e., an insufficient argument count will generate an Exception while an excessive count will not. Different commands may have different number of arguments; your program should be able to check invalid number of arguments for **all** commands.
  Example:

```
$addplayer lskywalker
Incorrect number of arguments supplied to command.

$
```

- Invalid move (during a game) - The player tries to remove an invalid number of stones from the game. For the move to be valid, it must be an integer between 1 and N inclusive, where N is the minimum of the upper bound and the number of stones remaining. Any other inputs (e.g. fractions, decimals, non-numeric entries) should be detected as invalid.
  Example (Upper bound is 3 stones here):

```
7 stones left: * * * * * * *
Han's turn - remove how many?
4

Invalid move. You must remove between 1 and 3 stones.

7 stones left: * * * * * * *
Han's turn - remove how many?
```

After implementing the invalid input checking, the scenarios detailed above should **not** cause your program to crash - rather, your program should display the appropriate error message, and continue execution, as illustrated in the examples. You may assume that, aside from the cases explicitly mentioned above, the input to your program will be valid.

2.3 The player statistics file

In Project B, no program data are stored to disk, so all player data are lost when exiting the program. Here, the task is to store these data upon exiting the program, and to restore them on subsequent executions. Thus, if one was to exit your program (using the 'exit' command), and then start it again (by running 'java Nimsys' at the shell prompt), your program should be restored to the state it was in immediately before exiting. That is, it should be as if the program never exited at all.

This can be achieved by storing your player data in a file. At the beginning of the execution of your program, if the file exists, it is opened and its contents loaded into the system. When your program exits, this file will be updated with new/modified players, and then closed. If the file does not already

exist, it will need to be created. It is up to you to decide the most appropriate format, e.g., text or binary, of this file. The name of the file should be **players.dat**, and it should be stored in the same directory as your program.

All player information should be stored, i.e., usernames, given / family names, and number of games played / won. Note that you do not need to store information about games in progress, since a game should never be in progress when the program exits properly, i.e., via the 'exit' command.

2.4 The AI (Artificial Intelligence) player

Here, a new type of player is to be added - an AI player. This player type should be controlled by the program, not by a human player. Aside from this, an AI player should be the same as a human player. That is, they should have all the same information associated with them (i.e., username, given/family names, and number of games played/won), and they should be stored in the system and appear in player lists/rankings, just as human players are. They should also be manipulated via all the same commands (with the exception of 'addplayer', since we now need to indicate whether we are adding a human player or an AI player to the system - see below).

The only difference between a human player and an AI player is in the way that they make a move. Instead of prompting for a move to be entered via standard input, the AI player should choose their own move, based on the state of the game. Thus, the only difference between a human player and an AI player should be in the method used to make a move. This suggests that the object-oriented principle of polymorphism should be applied here. Java offers polymorphism via two main avenues - inheritance, and interfaces. In this case, inheritance is the more appropriate choice. Conceptually speaking, we can think of human players and AI players as specialized players, i.e., a human player *is a* player, and an AI player *is a* player. They are identical in almost every way, and so most of their attributes and methods can be inherited - the only exception to this is the method used to make a move, which will need to be rewritten to act autonomously. You can add an abstract `NimPlayer` class, which will be used to represent the behaviour/attributes common to both Human and AI players. You can modify your original `NimPlayer` class used in Project B to be the new abstract `NimPlayer` class, and the human player class (`NimHumanPlayer`) and AI player class (`NimAIPlayer`) can extend the abstract player class.

Part of your mark for this project will be based on how well you apply polymorphism in your implementation of the human and the AI player, so it is important that you do use the principle of polymorphism in your design.

To allow for AI players to be added to the system, you should create a new command - 'addaiplayer'. This command should operate in exactly the same way as 'addplayer' (refer to Project B for details). The only difference is that the resulting player is an AI player. Note that all other commands, e.g., 'removeplayer' and 'editplayer' should work for both human players and AI players. Provided below is an example of the use of the 'addaiplayer' command:

```
$addaiplayer artoo,D2,R2

$
```

In this task you need to modify the provided **NimAIPlayer.java** to implement the AI player functionality. Note that this file and **Testable.java** are provided to you for auto-testing the task of Section 2.6. You do NOT need to modify the advancedMove() method until you work on the task of Section 2.6.

After implementing the AI player, the 'startgame' command should allow games to be started with one or both players being AI players. The game should proceed exactly as per the Project B spec, except that when it comes to an AI player's turn, there should be no reading of input from standard input - instead, the move should be immediately made by the AI. Provided below is an example execution. Here, Luke is a human player, and R2 D2 is an AI player:

```
$startgame 10,3,lskywalker,artoo

Initial stone count: 10
Maximum stone removal: 3
Player 1: Luke Skywalker
Player 2: R2 D2

10 stones left: * * * * * * * * * *
Luke's turn - remove how many?
3

7 stones left: * * * * * * *
R2's turn - remove how many?

5 stones left: * * * * *
Luke's turn - remove how many?
3

2 stones left: * *
R2's turn - remove how many?

1 stones left: *
Luke's turn - remove how many?
1

Game Over
R2 D2 wins!

$
```

The move an AI player makes given a specific situation shall follow certain strategy such that the victory is guaranteed for the AI player if it holds the ability to win when the game commences. For details, please see the following section.

2.5 The victory guaranteed strategy for AI players

When the number of remaining stones satisfy certain conditions, the player about to make a move may have a strategy to guarantee the victory. In this section, your task is to implement this strategy for the AI player as its `removeStone()` method. Please note that the `removeStone()` method will determine and return the number of stones to be removed. We describe such strategy in the following paragraph.

Simply, for a player to guarantee the victory no matter how the rival player moves in the future, it needs to ensure that the rival player is always left with $k(M+1)+1$ stones, where $k \in \{0, 1, 2, ...\}$ and M is the maximum number of stones can be removed at a time. Additonally, the commencing condition should satisfy that the rival player first move and there are $k(M+1)+1$ stones, or alternatively, in every winning player's turn there are some number of stones which cannot be expressed as $k(M+1)+1$.

The task is to implement the AI player's behavior such that it always wins if **either one of the following holds**:

- initial number of stones is $k(M+1)+1$, where $k \in \{0, 1, 2, ...\}$, and the rival player moves first,
- initial number of stones is **not** $k(M+1)+1$, where $k \in \{0, 1, 2, ...\}$, and the rival player moves second

For example, suppose Winfred and Louise are playing a game where a maximum of M stones can be removed each turn. Here, the notation [a, b] will be used to indicate all integers from a to b inclusive. So the number of stones a player must remove is in the range [1, M]. For Winfred to ensure his victory, the possible game states are ([a, b] here means there are at least a and up to b stones left to be removed by the players):

- Louise's turn, 1 stone left
- Winfred's turn, [2, M+1] stones left
- Louise's turn, (M+1)+1 stones left
- Winfred's turn, [(M+1)+2, 2(M+1)] stones left
- Louise's turn, 2(M+1)+1 stones left
- Winfred's turn, [2(M+1)+2, 3(M+1)] stones left
- Louise's turn, 3(M+1)+1 stones left
- and so on...

Notice that Winfred always has a way of moving to the next state, while Louise is always forced to move to the next state, i.e., the numbers of stones left for Louise are fixed in each of her turns.

If the winning condition does not hold, the AI player can remove any number of stones upto M, and hope the other player makes a mistake.

2.6 An advanced Nim game and its victory guaranteed strategy (Bonus)

(**This is a challenging task and it takes time to complete. You might find it worth more spending the time on assignments of other subjects. If you are successful in this task, you can earn back 1.5 marks that you may have lost in this project or previous two projects. The total mark for Project C is up to 11.5. However, the total mark for the three projects cannot exceed 40.**)

In this task, you are required to implement an advanced Nim game which has different rules from the original Nim game. Similar to the implementation of AI players, the polymorphism (inheritence and interface) provided by Java is expected to be demonstrated in your program. That is, both the original Nim game and this new advanced Nim game are specialized *game* with different rules. Either the inheritance or the interface can be chosen to reflect this relationship; the decision is up to you to suit your own design. Since we are going to have two types of games, the `removeStone()` method in the human player and AI player should also have two implementations, i.e., in game instances of different types of game, different `removeStone()` implementations are used, particularly for the AI player. For this advanced game setting, you can use the name `advancedMove()` for the method.

The rules of the advanced Nim game are as follows. The major rule in this advanced Nim game is that a player is **only allowed to remove 1 stone or 2 adjacent stones in each move**. Here, adjacent stones are stones who are neighbors to each other. Hence, the upper bound of stones to be removed is always 2. However, the requirement on the **adjacent stones** makes the position of the stones matters, i.e., removing the same number of stones at different positions may produce different game states, while in the original Nim game stones are conceptually the same, i.e., removing the same number of stones always produces the same game states. For example, given 5 stones represented as $< * * * * * >$,

- in the original Nim game removing 2 stones will always produce the state $< * * * >$
- in the advanced Nim game, depending on which stones are removed, removing 2 stones produces multiple states. In the following example, note that state 2 differs from state 3 because in state 2 the first two remained stones cannot be removed in a single move since they are not adjacent while in state 3 the first two remained stones can be removed in a single move.

1. removing the first and the second, results to $< \times \times * * * >$
2. removing the second the third, results to $< * \times \times * * >$
3. removing the third and the fourth, results to $< * * \times \times * >$
4. removing the fourth and the fifth, results to $< * * * \times \times >$

Similar to the original Nim game, each player takes turns to remove either one stone or two adjacent stones. **A player wins if he / she removed the last stone**.

There should be a new command defined in the `Nimsys`, named `startadvancedgame`, which will commence a new advanced game following the above rules. The syntax of this new command is `startadvancedgame initialstones,username1,username2`.

During the game, each player enters the move in the form of two integers `position number`, indicating the position and the number of stones to be removed. The stones left are displayed on one line, each stone is printed in the form of `<position, *>` if the stone is presented or `<position, x>` if the stone has already been removed.

Following is an example of the expected display of this command:

```
$startadvancedgame 10,lskywalker,artoo

Initial stone count: 10
Stones display: <1,*> <2,*> <3,*> <4,*> <5,*> <6,*> <7,*> <8,*> <9,*> <10,*>
Player 1: Luke Skywalker
Player 2: R2 D2

10 stones left: <1,*> <2,*> <3,*> <4,*> <5,*> <6,*> <7,*> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
3 2

8 stones left: <1,*> <2,*> <3,x> <4,x> <5,*> <6,*> <7,*> <8,*> <9,*> <10,*>
R2's turn - which to remove?

6 stones left: <1,*> <2,*> <3,x> <4,x> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
1 2

4 stones left: <1,x> <2,x> <3,x> <4,x> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
R2's turn - which to remove?

2 stones left: <1,x> <2,x> <3,x> <4,x> <5,*> <6,x> <7,x> <8,x> <9,x> <10,*>
Luke's turn - which to remove?
5 1

1 stones left: <1,x> <2,x> <3,x> <4,x> <5,x> <6,x> <7,x> <8,x> <9,x> <10,*>
R2's turn - which to remove?

Game Over
R2 D2 wins!

$
```

Any invalid input listed below shall be caught by **try / catch syntax**, and a line stating `Invalid move.` should be printed out:

- invalid position; given N stones when the game commences, the position should be $[1, N]$, any other values are invalid;
- invalid number of stones; **the number should be either 1 or 2**, any other values are invalid;
- the stones at the specified positions have already been removed

Here is an example output:

```
6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
12 1

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
1 3

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
2 2

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
1 2

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
```

After implementing the game rules, you are also required to design the strategy for the AI player. Because the game rules change in the advanced Nim game, the victory guaranteed strategy designed for the original Nim game does not work in this new game. Your task here is to implement the AI in this new game. The hints are: if the AI player **is the first one who moves** when the game commences, there is **always a victory guaranteed strategy** for the AI player; if the AI player **is not the first one who moves** when the game commences, it is still **possible** for the AI player to win as long as the rival player does not play exactly following the winning strategy. Hence, the implementation of the strategy for the AI player in this game should enable the AI player to:

- win if it moves first when game commences,
- win if it moves second, given that the rival player does not follow exactly the wining strategy.

The details of the strategy are left for you to design. To simplify the code design, you may assume **a maximum of 11 stones** in each game played. **In order to get bonus marks, you need to implement the strategy to meet all of the below requirements**:

- Your AI player class who can play the advanced game should be named as `NimAIPlayer`, please note this name is case sensitive and mandatory;
- Your AI player class should have a constructor with **no parameters**;
- Your AI player class should implement the interface `Testable` (provided), which is listed as follows:

```
public interface Testable {
        public String advancedMove(boolean[] available, String lastMove);
}
```

Here, the `boolean[] available` represents the stones remained to be removed, `true` as remained and `false` as removed, e.g., $< * \times \times \times * >$ can be represented as [true false false false true]. The `lastMove` represents the last move made by the rival player, e.g., if the rival player removed the second stone in the last turn, then `lastMove` is of value "2 1", if the second and the third stones are removed in the last turn, then `lastMove` is of value "2 2". The `advancedMove()` method returns the move chosen by your AI player this turn, in the form of `position number`. For example, if your AI player chooses to remove the first and the second stones, the returned string should be "1 2".

The definition of this interface should be put into your source directory as a file named `Testable.java` and submitted together with your solution. **The victory guaranteed strategy designed by you should be implemented in this `advancedMove()` method**. This method will be invoked when we test the correctness of your implementation of the victory guaranteed strategy.

Overall, your AI player class will look like (provided):

```
public class NimAIPlayer implements Testable ... {
        // you may further extend a class or implement an interface
        // to accomplish the task in Section 2.6
        public NimAIPlayer() {}
        ...
        public String advancedMove(boolean[] available, String lastMove) {
                // the implementation of the victory
                // guaranteed strategy designed by you
                ...
        }
        ...
}
```

Your solution will be evaluated using test cases in three cases:

1. Your AI player moves first to play against a dummy player, who moves randomly;
2. Your AI player moves first to play against an oracle AI player, who enumerates all the possibilities and try best to win;
3. Your AI player moves second to play against a dummy player, who moves randomly;

The solution will be assessed based on the wining ratio of your solution in the three cases. Figure 1 shows the testing procedure of a submitted solution. Specifically, for three cases, case 1, case 2 and case 3, 0.5 marks will be granted ONLY if your AI winning ratio is 100% in each case, suggesting your AI player passes all the test cases. Otherwise, a non-100% ratio for any case will not get any marks for that particular case.

Checklist For Solution

- **Blank line and whitespace related issues**
  - ☐ Make sure in terms of format, your output matches with the expected output on the submission system.

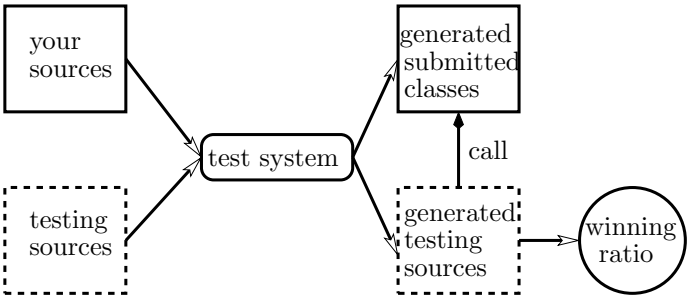

Figure 1: How testing on the advanced Nim game solution works; the testing procedure of the task in Section 2.6, the submitted codes will be complied and run together with the testing sources, while the generated testing classes will generate the winning ratio of the submitted solution.

- **File I/O mechanism related issues**
  - ☐ Make sure your program runs fine no matter whether the `players.dat` file exists.
  - ☐ Make sure every `exit` command triggers data to be written to the `players.dat` file.
- **Polymorphism related issues**
  - ☐ Make sure the AI player is implemented using inheritance mechanism.
  - ☐ Make sure you are leveraging the polymorphism to invoke the methods, i.e., using object declared in parent class to invoke methods overridden in the child class.
  - ☐ If attempting for bonus marks, make sure the advanced game is implemented using either inheritance or interface mechanism
- **Victory guaranteed strategy related issues**
  - ☐ Make sure your AI player can play without errors whether it plays as the first one or second one to move.
  - ☐ Try your best to win as many test cases as possible on the submission system with your AI player.
- **Submission issues**
  - ☐ Make sure there is only **one Scanner** object throughout your program.
- **Other issues**
  - ☐ The wining ratio of players with 0 games is 0.
  - ☐ The maximum number of players can be set as 100.
  - ☐ The maximum number of stones in each game is assumed to be 11 for the task of Section 2.6.
  - ☐ `Collections` such as `ArrayList` are acceptable.
  - ☐ Players are by default sorted according to the lexicographical order of the usernames.
  - ☐ Usernames can be assumed to be all lower-cased.
  - ☐ "*What it means when lastMove is a blank String/null in the advanced Nim game?*"
    It means that there is no last move and your `NimAIPlayer` object is to make the first move.
  - ☐ "*My code works perfectly on my machine but it does not get any winning ratio in the last test.*"
    First, check the FAQ items above and make sure you have handled each of them. Second, make sure that you do not change the Boolean array input parameter in `advancedMove()`. We just need your move returned as a String, and we will update the Boolean array.

Important Notes About Submission

Immediately after you make a submission using the "submit" command, computer automatic test will be conducted on your program by automatically compiling, running, and comparing your outputs for several test cases with generated expected outputs. The automatic test will deem your output wrong if your output does not match the expected output, even if the difference is just having an extra space or missing a comma. Therefore it is crucial that **your output follows exactly the same format shown in the examples above.**

The keyword `import` is available for you to use standard java packages. However, please **DO NOT** use the `package` keyword to organise your source files into packages. The automatic test system cannot deal with packages. If you are using Netbeans as the IDE, please be aware that the project name may automatically be used as the package name. Please remove the line like

```
package ProjC;
```

at the beginning of the source files before you submit them to the system.

Please use **ONLY ONE** Scanner object throughout your program. Otherwise the automatic test will cause your program to generate exceptions and terminate. The reason is that in the automatic test, multiple lines of test inputs are sent all together to the program. As the program receives the inputs, it will pass them all to the currently active Scanner object, leaving the rest Scanner objects nothing to read and hence cause run-time exception. Therefore it is crucial that **your program has only one Scanner object.** Arguments such as "It runs correctly when I do manual test, but fails under automatic test" will not be accepted.

3 Your Task

Implement the new version of Nimsys in Java according to the above specification. Specifically, it includes following subtasks:

1. try / catch syntax to detect the listed invalid inputs in Section 2.2;
2. file based mechanism for player data as described in Section 2.3;
3. the AI player implemented using polymorphism mechanism in Section 2.4;
4. the victory guaranteed strategy in the original Nim game for the AI player in Section 2.5;
5. (**bonus**) the advanced Nim game and the corresponding victory guaranteed strategy for AI players in Section 2.6. This is a challenging task. If you cannot work out the strategy, do not get stuck on it. Please complete all other tasks first and leave it as the last one.
6. See LMS for a detailed marking scheme.

4 Assessment

This project is worth 10% of the total marks for the subject. Remember that there is a 50% hurdle requirement (i.e. 20/40) for the three projects.

Please note that if you attempt the bonus task, you can get 1.5 bonus marks. The total mark for Project C is up to 11.5. However, the total mark for the three projects cannot exceed 40. **This is a challenging task and it takes time to complete, please allocate your time wisely.**

The deadline for the project is **3pm, Friday 31 May, 2019**. **There is a 20% penalty per day for late submissions. Suppose your project gets a mark of 10 but is submitted within 1 day**

after the deadline, then you get 20% penalty and the mark will be 8 after penalty. There will be no mark for submissions after 3pm 4 June, 2019 .

Your Java program will be assessed based on correctness of the output as well as quality of code implementation. See LMS for a detailed marking scheme.

5 Submission

The entry point of your program should be in a class called Nimsys (in a file called Nimsys.java). Thus, your program will be invoked via:

```
java Nimsys
```

Your Java classes should be stored together in their own directory under your home directory on the student server. Then, you can submit your work using the following command:

```
submit COMP90041 projC *.java

For late submissions, use the following command:
submit COMP90041 projC.late *.java
```

Note that *.java includes all Java files in current directory, so you must make sure that you don't include irrelevant Java files in the current directory. **Note that you must submit all Java files you have used for your project, not just** `Nimsys.java`. If you submit you code multiple times, the later submission will overwrite the previous one. If you submit all your java source codes and then modify one source code, you need to submit all of your source codes again, not just the modified one.

You should then verify your submission using the following command. This will store the verification information in the file '`feedback.txt`', which you can then view:

```
verify COMP90041 projC > feedback.txt

For late submissions, use the following command:
verify COMP90041 projC.late > feedback.txt
```

You should issue the above commands from within the same directory as where your project files are stored (to get there you may need to use the `cd` 'Change Directory' command). Note that you can submit as many times as you like before the deadline.

How you edit, compile and run your Java program is up to you. You are free to use any editor or development environment. However, **you need to ensure that your program compiles and runs correctly on the student servers**.

The test cases used to mark your submissions will be **different** from the sample tests given. You should test your program extensively to ensure it is correct for other input values with the same format as the sample tests. The tests for section 2.5 are **hidden**.

Submit your program to the student servers **a couple of days before the deadline** to ensure that they work (you can still improve your program). **"I can't get my code to work on the student server but it worked on my Windows machine" is not an acceptable excuse for late submissions.**

6 Individual Work

Note well that this project is part of your final assessment, so cheating is not acceptable. Any form of material exchange, whether written, electronic or any other medium is considered cheating, and so is

the soliciting of help from electronic newsgroups. Providing undue assistance is considered as serious as receiving it, and in the case of similarities that indicate exchange of more than basic ideas, formal disciplinary action will be taken for all involved parties. A sophisticated program that undertakes deep structural analysis of Java code identifying regions of similarity will be run over all submissions in "compare every pair" mode.