Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS212 - Data structures and algorithms

Assignment 1 Specifications: Recursion

Release date: 17-02-2025 at 06:00

Due date: 14-03-2025 at 23:59

Total marks: 170

Contents

1 General Instructions

- *Read the entire assignment thoroughly before you start coding.*
- This assignment should be completed individually; no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of the built-in Java data structures. Doing so will result in a zero mark. You may only use native 1-dimensional arrays where applicable.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experiences a runtime error, you will be awarded a zero mark. Runtime errors are considered unsafe programming.
- Read the entire specification before you start coding.
- **Ensure your code compiles with Java 8**
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Please note that there is a late deadline which is 1 hour after the initial deadline, but you will lose 20% of your mark.

2 Warning

For this assignment, you are **only allowed to use recursion**. The following rules must be adhered to for the entirety of this assignment. Failure to comply with any of these rules **will result in a mark of 0**. The words "`for`","`do`" and "`while`" may not appear anywhere in any of the files you upload. Make sure you do not use any variable or function names that contain these words, and also **do not use these words in your comments**. For safety reasons, before uploading make sure to search (Ctrl F) through your files to make sure you don't find these keywords.

3 Overview

Recursion is an important tool for programming, as some problems have solutions that can easily be solved using recursion. Recursion occurs when a function calls itself directly or indirectly through other functions. Make sure you do practical 1 before attempting this assignment as practical 1 will give you some tips for how to implement recursive functions.

The N-Queens problem is a well known recursive backtracking example where you are tasked with placing N number of queens on a NxN chessboard such that none of the queens threaten each other according to the usual rules of chess. In this assignment, you will be implementing a version of this but we will be using queens, bishops and rooks. You will also be tasked with trying to fit the largest number of pieces on the board, since it is possible to fit more than N pieces in some scenarios.

4 Tasks

You are tasked with creating a program that will solve a variation of the N-Queens problem by implementing the classes shown below. Please note that these are the bare minimum and you will need to add helper functions to implement this recursively. Also note that using any iterative loops will be seen as a fail and will result in a mark of 0, as was explained in Section 2.

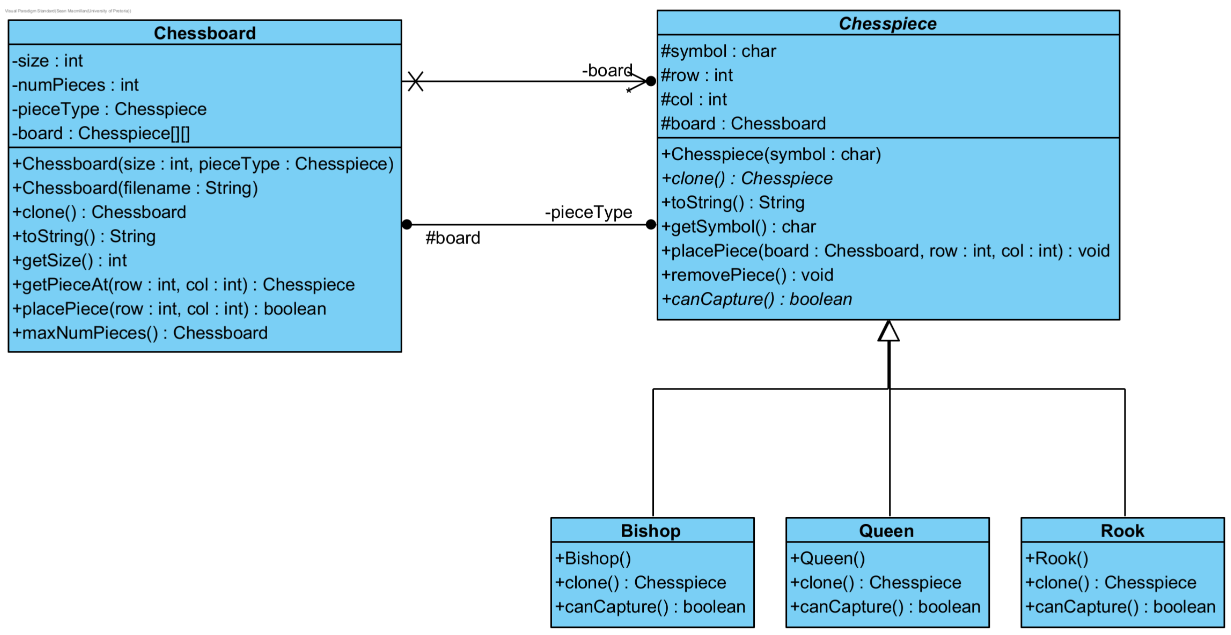

Figure 1: UML

4.1 Chesspiece

This is an abstract class used to represent the chess pieces that can be placed on the chessboard.

4.1.1 Members

- symbol
  - This is the symbol that represents the chess piece which will be used for printing.
- row
  - This is the row in which the piece is placed.
- col
  - This is the column in which the piece is placed.
- board
  - This is the Chessboard that this piece is placed on.

4.1.2 Functions

- Chesspiece
  - Initialise the symbol to the passed-in parameter.
  - Initialise the coordinates to `[-1,-1]`.
  - Initialise the board to `null`.
- clone
  - This is an abstract function that can be used to make a copy of the current object.
- toString
  - This returns a string representation of the current `Chesspiece`.
  - Do not change this, since FitchFork will use this to mark your assignment.
- getSymbol
  - Return the symbol of this `Chesspiece`.
- placePiece
  - Set the corresponding member variables to the passed-in parameters.
- removePiece
  - Set the coordinates to `[-1,-1]` and the board to `null`.
- canCapture
  - This will return `true` if this piece can capture any piece on the board, and return `false` if it cannot capture any piece on the board.

4.2 Bishop, Queen and Rook

These classes all inherit from Chesspiece and are the chess pieces that can be placed on boards. All of these classes act similarly and as such, to save space, will be covered in one section.

4.2.1 Functions

- Constructor
  - The constructors should call the parent constructor using the appropriate symbol:
    - Bishop: 'B'
    - Queen: 'Q'
    - Rook: 'R'
- clone
  - This should create and return a copy of the current `Chesspiece`.
  - The row and column variables should be copied over, while the new piece should have its board variable set to `null`.
- canCapture
  - This should return `true` if this piece can capture any piece on the board, and `false` if it cannot capture any pieces.
  - The rules for captures follow normal chess rules, and if you are unsure what these are you can read about it here.

4.3 Chessboard

This class represents a NxN chessboard and will be used to calculate the most optimal placing of a specific chess piece.

4.3.1 Members

- size
  - This is the number of cells in the rows and columns of the chessboard.
  - Only sizes in the range size $\in [0, 5)$ are allowed.
  - *Note: The reason why sizes 5 and up are not allowed is to prevent overloading our servers. Once you are satisfied with your mark on FitchFork, feel free to remove the restriction and play around with it on your computer to see the cool patterns that emerge, but be warned this might take some time since this is not an optimized algorithm.*
- pieceType
  - This is the type of `Chesspiece` that can be placed on the `Chessboard`. Note that this means only one type of Chesspiece can be placed on a board at a time.
  - This piece will only be used to create clones, thus it will have coordinates of `[-1,-1]` and have a board of `null`.
- board
  - This is a 2D array that will store the chessboard cells.
  - Empty spots will be filled with `null`.

- numPieces
  - This integer shows how many pieces are currently placed on the board.

4.3.2 Functions

- Chessboard
  - This constructor will be used to create an empty `Chessboard`.
  - If the passed-in size is not in the allowed range, then set the size variable to 0, otherwise set it to the passed-in parameter.
  - Set the `pieceType` member variable to the passed-in parameter.
  - Initialise the board and `numPieces` members to reflect the current state of the board.
- Chessboard
  - This constructor will be used to create a `Chessboard` with initial positions stored in a textfile.
  - If the textfile does not exist, then create a `Chessboard` with size 0 and `pieceType` of type Queen.
  - The first line of the textfile will be a number, read this in as the size of the board. If the size of the board is not in the allowed range, then set the size variable to 0.
  - On the second line of the textfile will be a String. Read this in. If it is equal to "Queen","Rook" or "Bishop" then initialise the `pieceType` to the corresponding `Chesspiece`. If it is not equal to any of those strings, then initialise the `pieceType` to a Queen.
  - After that is an unknown number of lines. Each line represents a coordinate on which the appropriate piece is placed. Each line will contain two numbers with a space between them. The first number is the row, and the second number is the column. Call the `Chessboard::placePiece` function with these coordinates.
- clone
  - This will create a copy of the current `Chessboard`.
  - Copy over all of the member variables. When copying the Chesspieces, make sure that you use the clone function to make copies of the pieces.
- toString
  - This returns a string representation of the `Chessboard`.
  - Do not change this, since Fitchfork will use this to mark your assignment.
  - Use this function as an example for how to convert nested loops to recursive functions.
- getSize
  - Returns the size member.
- getPieceAt
  - Returns the `Chesspiece` at the passed-in coordinate.
  - If the passed-in coordinate is invalid, then return `null`.

- placePiece
  - This function will attempt to place a copy of the `pieceType` at the passed-in coordinate. If a copy was placed then `true` is returned, otherwise `false` is returned.
  - If the passed-in coordinate is invalid or there is already a piece there, then return `false`.
  - If the clone of the `pieceType` can be placed at the passed-in coordinate without being able to capture any pieces, then update the member variables to show that the piece has been placed at the coordinate and return `true`. If the piece can capture a piece already on the board, then do not add this piece and do not make any changes to the member variables (or revert the changes, if you already changed it) and return `false`.
  - *Hint: Since only one type of piece is allowed to be on the board, you only have to check whether this piece can capture any other piece. This is because if any other piece already on the board can capture this piece, this piece can also capture that piece.*
- maxNumPieces
  - This should return the `Chessboard` that can fit the largest number of chess pieces of the same type as `pieceType` on the board.
  - You should implement a recursive backtracking strategy to calculate this. The pseudocode is given below:

```
1. Starting at the top left, loop through all of the open squares on the board. Start by looping
   through all the columns in a row, before moving on to the next row. For each empty square do
   the following:
   (a) Create a clone of the current board.
   (b) Call the placePiece function on the clone of the board at the coordinates of the empty
       square.
   (c) If the chess piece could not be placed then continue to the next empty square.
   (d) If the chess piece was placed; call the maxNumPieces function on the clone of the board to
       get a new solution, then see if this solution is better than the current most optimal solution
       found. Hint: Your recursive helper function should have a variable where you store the
       current best solution. Initialise this to the current board at the first function call.
   (e) If the new solution has more pieces than the old best solution then, return the new solution.
       If the new solution has the same or less pieces than the old solution then return the old
       solution.
```

5 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, approximately 10% of the assignment marks will be assigned to your testing skills. To do this, you will need to submit a testing main (inside the Main.java file) that will be used to test an Instructor-provided solution. You may add any helper functions to the Main.java file to aid your testing. In order to determine the coverage of your testing the jacoco tool. The following set of commands will be used to run jacoco:

```
javac *.java
rm -Rf cov
mkdir ./cov
java -javaagent:jacocoagent.jar=excludes=org.jacoco.*,destfile=./cov/output.exec -cp ./ Main
mv *.class ./cov
java -jar ./jacococli.jar report ./cov/output.exec --classfiles ./cov --html ./cov/report
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

and we will scale this ratio according to the size of the class.
The mark you will receive for the testing coverage task is determined using table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the function stipulated in this specification will be considered to determine your mark. Remember that your main will be testing the instructor-provided code and as such it can only be assumed that the functions outlined in this specification are defined and implemented.

6 Upload checklist

The following files should be in the root of your archive

- Chessboard.java
- Chesspiece.java
- Bishop.java
- Queen.java
- Rook.java
- Main.java
- Any textfiles needed by your Main

7 Allowed libraries

- java.io.File
- java.io.FileNotFoundException
- java.util.Scanner

8 Submission

You need to submit your source files on the FitchFork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place the above-mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. Your code must be able to be compiled with the Java 8 standard.

For this practical, you will have 3 upload opportunities and your best mark will be your final mark. Upload your archive to the appropriate slot on the FitchFork website well before the deadline. **No late submissions will be accepted!**