Programming Hand in 2

Brooks & Odderskov

For this and the remaining two handins you must work in your semester project group.

For each of the below exercises there is an associated template. You can find all templates in the zip file 'dmaProg2.zip'. Create a new project in IntelliJ for all your DMA programming handins. It is up to you how you want to structure your hand ins but it may be a good idea to have a module for each of the four hand ins. Unpack the zip file to the newly created project folder.

Do not change any of the existing code, e.g. do not rename methods, etc., although you may need to change the reference to the package (first line in templates). You will only need to add the body of your methods in the relevant template. For each of the programs, replace the comment with your code. Note, you do not need to create methods for user inputs. All you really need to do is to write the logic of the methods, i.e. the algorithms.

Each exercise is accompanied by a test class. You can use this to test whether your method works correctly. Note, as opposed to the first programming exercises, there are no tests for complexity. You must analyze your code your self and comment on the complexity. Comments are made in the code as per the example shown in class and which is similar to what you must do in your SEP1 project.

For the three final programming hand ins, it will be useful to get acquainted with data structures in Java's standard library. You've already been introduced to ArrayLists. Aarhus University has made a nice guide to which libraries you need. You can find the guide here.

To find fast solutions to some of the problems of this hand in, you will need to use a Balanced binary search tree (BBST), so you can use a `TreeSet` in Java. In class, we will go through /have gone through AVL trees as an example of a BBST. Note, the link mentions a red/black search tree which is simply another kind of BBST but which has "the same" implementation in Java (i.e. `TreeSet`).

The scoring is stated in each exercises but note that algorithm analysis gives you extra points. Naturally, you only need to analyse the code you have written so you do not need to comment on existing code in the templates.

You upload your hand in as a zip-file where you simply zip the handin2 module with all your code/files.

If you get stuck doing any of the exercises, it is recommended seeking out the student instructor at the study café (on Discord).

Program 1: `gcd`

In this problem, the input to your program are two positive integers, and your program must output the greatest common divisor of the two integers. For instance, if the input is (98, 84) your program should output 14.

Concretely, you must implement three public methods named `findGCD1`, `findGCD2`, and `findGCD3` that each take two integers as argument and return an `int`:

- `findGCD1` must use a for loop and no while loops and no recursive calls
- `findGCD2` must use a while loop and no for loops and no recursive calls
- `findGCD3` must use recursion and no loops of any kind

Use the template **GCD.java**.

Scoring:

- 1/3 point for correct `findGCD1` and 1/3 point for correct algorithm analysis
- 1/3 point for correct `findGCD2` and 1/3 point for correct algorithm analysis
- 1/3 point for correct `findGCD3` and 1/3 point for correct algorithm analysis

Program 2: `extendedEA`

In this problem, you must implement the Extended Euclidean Algorithm (EAA). You can read about both the Euclidean Algorithm and EAA here, where you also will find implementations as inspiration to your own implementation. As is mentioned in the link, EAA finds integer coefficients x and y such that:

$$ax + by = \gcd(a, b)$$

EAA was also mentioned in the lesson about multiplicative inverses in the first part of the course so you may want to look up the theory in the first course book also. Feel free to research the topic further online.

The input to your program are two positive integers, and your program must output an array containing two values, x and y from the above stated linear combination. For instance, if the input is (34, 13) your program should output [5, -13].

Concretely, you must implement a public method named `extendedEA` that take two positive integers as argument and returns an `int[]`. Use the template **extendedEA.java**.

Note, we have created a main method that includes a scanner for user inputs. This main method must have access to a class named `GCD` that contains a method named `findGCD`, i.e. must be able to call `GCD.findGCD(a, b)` that returns the greatest common divisor of a and b. You can simply use one of the methods from Program 1 above.

If everything is coded and set up correctly, then inputting 34 and 13 will result in the following output:

```
Extended Euclidian Algorithm
Expression calculator for gcd(a, b)
Give value for a: 34
Give value for b: 13

[5, -13]
1 = 34(5) + 13(-13)
The inverse of 34 mod 13 is 5

Process finished with exit code 0
```

And inputting 34 and 14 will result in the following output:

```
Extended Euclidian Algorithm
Expression calculator for gcd(a, b)
Give value for a: 34
Give value for b: 14

[-2, 5]
2 = 34(-2) + 14(5)
No inverse exists

Process finished with exit code 0
```

Scoring:

- 2 points for correct algorithm and an additional point for correct algorithm analysis

Program 3: `squareMultiply`

In this problem, the input to your program are three positive integers, and your program should output the modular exponentiation. For instance, if the inputs are 3, 5, and 8, respectively, then your program should output 3.

Concretely, you must implement a public method named `modExpoCalculator` that takes three `int` as argument and returns an `int`. Use the template **squareMultiply.java**.

Scoring:

- 1 point for correct algorithm and an additional point for correct algorithm analysis

Program 4: `dodgeBall`

Crazy Dodgeball is a game in which two teams play against each other using a ball. At any point in time, the players on your team stand on a line, and the following two things can happen:

- A new player comes alive and is added to the line at position x
- A ball is thrown at position x
  - If a ball is thrown at position x and there is a player at position x, that player is out and is removed from the line.
  - Otherwise, the player that is closest to position x moves to position x and throws the ball back at the other team.

Here is an example, starting with four players at positions 1, 7, 10 and 20:

- Ball thrown at 10: 1, 7, 20 (10 dead; distance 0)
- Ball thrown at 18: 1, 7, 18 (moved from 20 to 18; distance 2)
- Ball thrown at 5: 1, 5, 18 (moved from 7 to 5; distance 2)
- New player at 11: 1, 5, 11, 18
- Ball thrown at 8: 1, 8, 11, 18 (moved from 5 to 8; distance 3)

This example corresponds to `test1` in the `testDodge` class. Looking at how the test case is constructed may sometimes be useful in figuring out how to construct the code (this is similar to "test-driven" development).

In this exercise you must implement a class named `dodgeBall` with the following methods:

- `void addPlayer(int x)` $\longrightarrow$ adds a new player at position x in the game
- `int throwBall(int x)` $\longrightarrow$ updates the set of players when a ball is thrown at position x and returns the distance from the closest player to the ball

At no point in time are there two players that stand at the same position.
If a ball is thrown at position x and there are two players that are both closest to the ball, that is, they stand at positions $x - d$ and $x + d$ for some integer d, then the player with the numerically smallest position moves to the ball.
For example, if there are two players that stand at positions 1 and 7, and a ball is thrown at position 4, then the player at position 1 moves to position 4.
Concretely, you should use the template **dodgeBall.java** and implement the methods `addPlayer` and throwBall in `dodgeBall`, and you are allowed to add your own private fields to the class.
Input constraints:

- When `throwBall(x)` is called, there is at least one player on the line
- Number of operations ≤ 500000
- $1 \leq x \leq 5000000$ (whenever `addPlayer(x)` or `throwBall(x)` is invoked)

Scoring:

- 1 point for correct algorithm
- 1 extra point for correct and fast algorithm
- 1 additional point for correct algorithm analysis

An algorithm that spends $O(\log N)$ time on `addPlayer()` and `throwBall()` is fast enough for the extra point.

Hint for the fast solution: Use a balanced binary search tree.