Lab 2: Recursion

F28SG – Introduction to Data Structures and Algorithms (6 marks)

Plagiarism policy

This lab is not group work, and you are assessed individually.

Therefore, the work you submit for this lab **must be entirely your own**. You must not share your code solution with other students, and you must not copy code solutions from others. The university plagiarism policy is clear:

> *"Plagiarism involves the act of taking the ideas, writings or inventions of another person and using these as if they were one's own, whether intentionally or not."*
> ***Definition 2.1.***

The disciplinary action for plagiarism is an award of an F grade (fail) for the course. Serious instances of plagiarism will result in Compulsory Withdrawal from the university.

Q1) Recursive methods (3 points)

In this part you will implement some recursive methods. Your starting point is the file **Recursion.java**. Your task is to complete the methods as described below.

Unit tests in RecursionTest.java will check your implementations.

Q1A) Recursive sum of numbers

Write a recursive method `sum` that, given a number n, returns the sum of all positive numbers up to n - that is, it computes:

```
sum(n) = 1 + 2 + ... + n
```

See **Hint 1** if you are stuck.

Q1B) Recursive multiplication using only addition

Write a recursive method `multiply` which given two integers `m` and `n` as arguments, computes `m*n` using only Java's addition operator `+` and recursion. Your implementation cannot use Java's multiplication operator `*`. See **Hint 2** if you are stuck.

Q1C) Compute Fibonacci number

The Fibonacci sequence is 0,1,1,2,3,5,8,13,21,34,.... You should write a method `Fibonacci` which given n as an argument, returns the nth Fibonacci number using recursion, that is

```
Fibonacci(0) = 0

Fibonacci(1) = 1

Fibonacci(2) = 1

Fibonacci(3) = 2
```

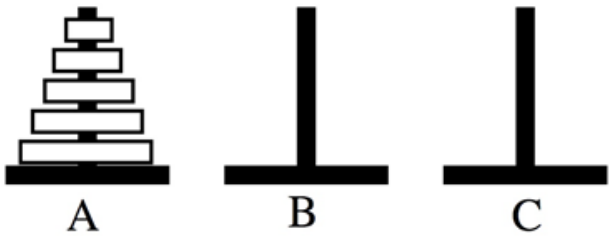
```
Fibonacci(4) = 3

Fibonacci(5) = 5

Fibonacci(6) = 8

Fibonacci(7) = 13

Fibonacci(8) = 21
```

and so on.

See **Hint 3** if you are stuck.

Q2) The Tower of Hanoi (2 points)

The Tower of Hanoi is a game that has a very elegant recursive solution. The goal is to move all rings of tower A to tower B:



However, you must adhere to the following rules

- you can only move one ring at a time
- you can only move the top ring
- you cannot put a larger ring on top of a smaller ring

To better understand the game, there are several places online where you can play it. For example, here:

http://www.mathsisfun.com/games/towerofhanoi.html

Your job is to implement a recursive method that solves the game for *n* number of rings.

You will see how the game is played in the **test/HanoiTest.java** file, i.e.

```
Hanoi game = new Hanoi(rings);

game.playHanoi();
```

Where `rings` is an integer value which is the number of rings starting on tower A.

A template for your implementation is in the file **Hanoi.java**.

Q2A) Implement moving one ring

You should implement moving one ring by implementing the method:

```
public static void moveOneRing(int from, int to){
    /* TODO */
}
```

Where integers `from` and `to` represent indexes: 0 for tower A, 1 for tower B and 2 for tower C.

To simulate the move of a ring from one tower to another, update the stacks stored in the `towers` array variable in **Hanoi.java**, which stores one stack for each of the 3 towers. Hanoi stacks A, B and C are implemented with this array in **Hanoi.java** as follows:

```
private Stack[] towers = new Stack[TOWERS];
```

Where `towers[0]` is a Stack data structure that represents Hanoi tower A. In other words, there is a Stack data structure at position 0 in the array `towers`. This means that `towers[1]` is a Stack for Hanoi tower B, `towers[2]`is a Stack for Hanoi tower C.

If you want to place a ring on top of a Hanoi tower, you'll need to *push* a ring onto the Stack data structure that represents that Hanoi tower.

For example, to place a ring called `ring` onto Hanoi tower B:

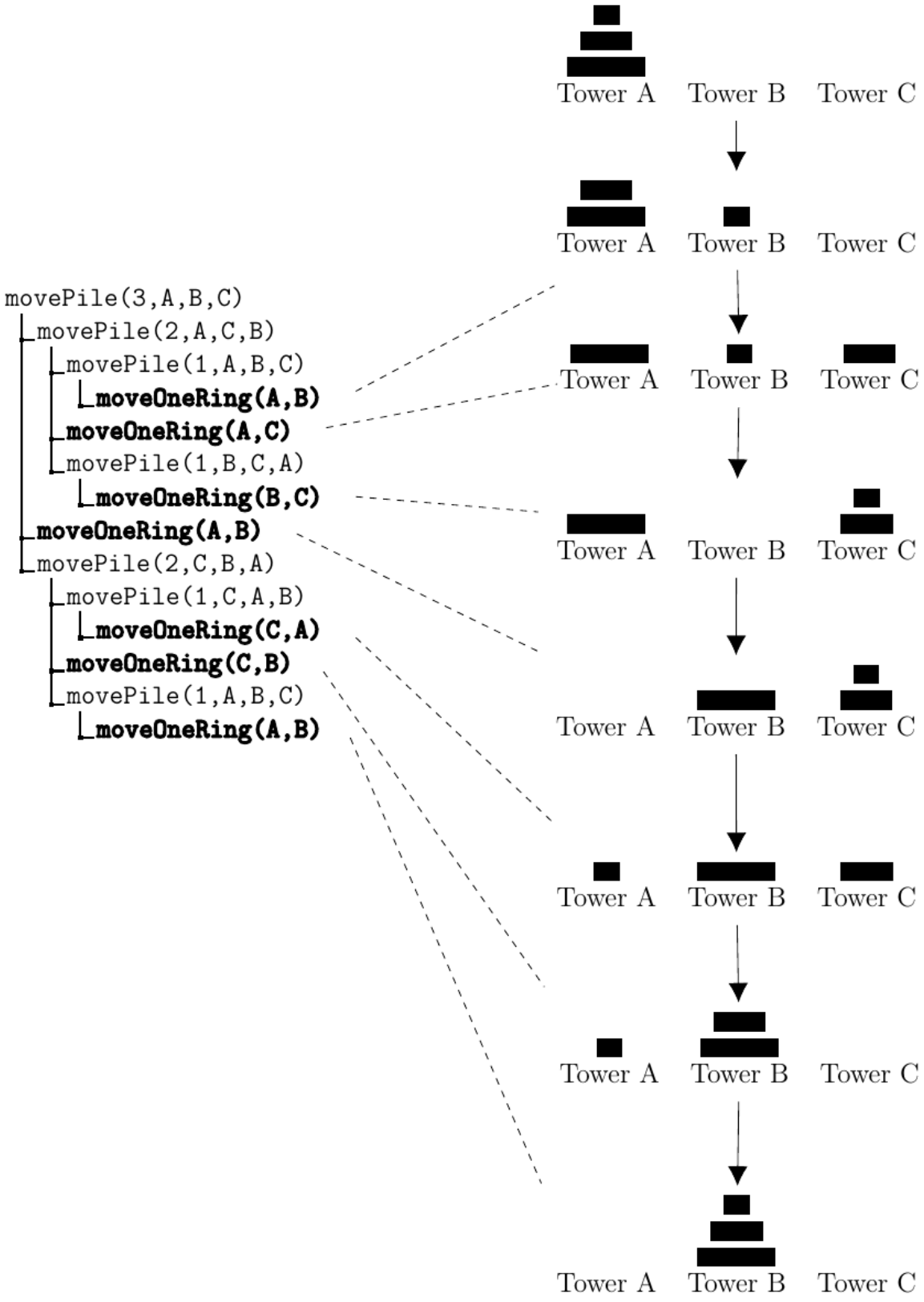
```
towers[1].push( ring );
```

The idea is to take a ring from the top of the `from` tower (where `from` is an array index position) and put it on the top of the `to` tower.

The tests are not expected to pass at this point.

Q2B) Implementing moving a pile

You are to implement the Hanoi game in a recursive programming style. The `movePile` method should coordinate the order of calls to `moveOneRing` that you implemented for the previous question. The sequencing of `moveOneRing` calls should stick to the rules of the Hanoi game and should successfully move all rings from the `from` tower to the `to` tower. The next page shows an example of moving three rings from tower A to tower B.


movePile(3,A,B,C)
movePile(2,A,C,B)
movePile(1,A,B,C)
moveOneRing(A,B)
moveOneRing(A,C)
movePile(1,B,C,A)
moveOneRing(B,C)
moveOneRing(A,B)
movePile(2,C,B,A)
movePile(1,C,A,B)
moveOneRing(C,A)
moveOneRing(C,B)
movePile(1,A,B,C)
moveOneRing(A,B)
Tower A
Tower B
Tower C

The 2nd part requires `movePile` to be implemented:

```
public void movePile(int n, int from, int to, int via){
    /* TODO */
}
```

In the `playHanoi()` method it is used like this:

```
movePile(numRings,0,1,2);
```

Where `numRings` are the number of rings initially on tower A.

The `movePile` method should be a recursive implementation.

- **Base case** : when `n` == 1, move the one ring on tower with index `from`, to the tower with index `to`.
- **Step case** : use recursive calls to move `n` rings from tower `from` to tower `to`. Coordinate these `moveOneRing` calls (passing the correct `from` and `to` arguments), using recursive calls inside `movePile`.

See the Javdoc documentation for the `movePile` method in `Hanoi.java` for an execution example.

Hint 4 which explains the algorithm in more detail.

The tests in HanoiTests.java runs the game using 1, 2.. 10 and 20 rings starting at tower A. Each time the end of game should result in all rings on tower B, and no rings on towers A and C. During execution, at no time can a larger ring be placed on top of a smaller ring. The `checkTower` method ensures this invariant. If it is ever violated, a `HanoiException` is thrown and the JUnit test will fail.

If you have correctly implemented the two Java methods, all JUnit tests should pass now.

Run the file with JUnit to test your implementation.

Q3) Code Quality (1 point)

Code quality is vitally important for so many reasons. Not least, for readability and maintainability, not just for yourself but for others too since in industry, software engineering is almost always a group exercise. Real world software engineering is mostly about refactoring and testing code, rather than writing new code (more code means higher maintenance costs!).

An additional mark is awarded if your code is deemed to be of high quality:

- **Code simplicity**
  - Is the code as short as it can be?
  - Is the Big-Oh complexity as small as it can possibly be?
  - Is the control flow (loops, while statements, if statements, etc.) simple to follow?
- **Documentation**
  - Are you using comments ***inside your implementation methods and test methods*** to provide an algorithmic commentary about what the code is doing.

    Here is a **good** comment:

    ```
    // creates unidirectional connection from the predecessor node to the new node
    prevNode.nextNode = newNode;
    ```

    Here is a **less useful** comment:

    ```
    // sets preNode.nextNode to newNode
    prevNode.nextNode = newNode;
    ```

  - Is Javadoc syntax used for documenting the code? ***Hint!*** In Eclipse move your cursor to the text definition of a field, method or a class, then use the following keyboard shortcut: ***Alt + Shift + j*** , and then fill in the generated Javadoc template. If you are using MacOS, the shortcut is: ***⌘ + Alt + j***
- **Conventions**
  - Are sensible Java code conventions used, e.g. for code indentation, declarations, statements, etc? See Sections 4, 5, 6, 7 and 9 of *Java Code Conventions*. ***Hint!*** *Use the keyboard shortcut in Eclipse:* ***Control + Shift + f*** , or on MacOS: ***⌘ + Shift + f***

Additional challenge: tail recursion

Many problems have very elegant recursive solutions. However, when using recursion extra memory is used since we have to push each method call to the program stack (see lecture 3). For example, in the factorial example

```
public static int factorial(int n){
        if (n == 0) return 1;
        else   return n*factorial(n-1);
  }
```

each call is pushed to the program stack, and when we reach the base case each step is popped and the result is multiplied with n. A more efficient type of recursion is called **tail recursion** and it happens when the recursive call is the very last operation of a method. In such cases the compiler can optimize the code by turning the recursion into normal iteration. For example, we can turn the factorial method into a tail recursive version by carry with us the result so far:

```
public static int tailFactorial(int n,int res){
        if (n == 0) return res;
        else return tailFactorial(n-1,n*res);
  }
```

On termination we do not need to step back through each call made, but we can return the value directly.

The task here is to implement the `sum` and `multiplication` methods in **Tail.java** using tail recursive variants `tailSum` and `tailMultiply` that you are to implement:

```
public static int tailSum(int n,int sum){
        // your code
  }


  public static int tailMultiply(int m,int n,int sum){
     // your code
  }
```

To pass all tests in **TailTest.java**, use the `tailFactorial` method in the factorial method in **Tail.java** .

Hints

Hint 1! *The method is very similar to the factorial method from Lecture 4.*

Hint 2! *m * n can be seen as m added together n times. This can be implemented recursively where each step adds m to the result of m * (n-1), until n reaches 0.*

Hint 3! *A Fibonacci number is the sum of the previous two Fibonacci numbers. You therefore need two base cases for the first two numbers in the sequence.*

Hint 4! *If you want to move N rings from tower A to tower B using tower C, then you can reduce the problem to* ***first*** *move the top (N-1) rings from tower A to tower C (using B),* ***then*** *move ring N from A to B, and* ***then*** *move the top (N-1) from C to B using A. If you continue the recursion this way, then you will end up with a base case when N=1 where you only move one ring.*