Recursion

Table of Contents

1. Session 2 (lab): Recursion

    1.1. Iteration vs. Recursion
    1.2. Get the maximum item of an array
    1.3. From integer to binary and from binary to integer

2. Homework

    2.1. Palidromes

1. Session 2 (lab): Recursion

1.1. Iteration vs. Recursion

In this exercise, you will solve a series of simple problems using both iterative and recursive strategies. This way you could appreciate the difference between these strategies, and learn when it is worth using one strategy or another.

Section 1. How many birthday candles have you blown out throughout your life?

Each year you blow out as many birthday candles as your new age. Therefore, if you were 6, the total number of birthday candles that you had blown out throughout your life would be:

```
count(6) = 1 + 2 + 3 + 4 + 5 + 6 = 21
```

Write a class, called `CandleCounterIterative`. This class will implement the following interface:

```
public interface CandleCounter {
    String getImplementationDescription();
    long count(long years);
}
```

The `getImplementationDescription` method must return the string "Iterative". The `count` method must return the number of candles a person has blown out throughout her life (the current age of the person will be the parameter of the method). This method must be implemented using an iterative strategy.

Try your implementation using the CandleCounterTest.java class. Comment the two last lines of the main method (you will use them later on).

You can see an example of the execution of the `CandleCounterTest` class below:

```
; java CandleCounterTest
Testing Iterative:
    count(0): 0, OK
    count(1): 1, OK
    count(2): 3, OK
    count(6): 21, OK
    count(10): 55, OK
    count(1000): 500500, OK
```

Section 2. Counting candles recursively

Write a class, called `CandleCounterRecursive`, which implements the previous interface (`CandleCounter`).

For this implementation of the interface, the `getImplementationDescription` method must return the "Recursive" string. The `count` method must be implemented using an recursive strategy this time.

Notice that the total number of candles blown out so far is the number of candles blown out this year (the current age) plus the total number of candles blown out up until last year.

You can see an example of execution of the `CandleCounterTest` class below. This time, uncomment the previous to the last line of the main in order to test also the recursive strategy.

```
; java CandleCounterTest
Testing Iterative:
    count(0): 0, OK
    count(1): 1, OK
    count(2): 3, OK
    count(6): 21, OK
    count(10): 55, OK
    count(1000): 500500, OK
Testing Recursive:
    count(0): 0, OK
    count(1): 1, OK
    count(2): 3, OK
    count(6): 21, OK
    count(10): 55, OK
    count(1000): 500500, OK
```

Section 3.

In some cases, neither the iterative solution (with loops) nor the recursive one are the most efficient alternatives. For instance, in the previous case, the total number of birthday candles blown out throughout your life is a well known integer series, that can be expressed as follows:

count(n) = 1 + 2 + ... + n = $\Sigma^{n}_{i=0}$ i

Use your calculus knowledge in order to solve the previous expression. Write a new class, called `CandleCounterEfficient`, which will implement also the `CandleCounter` interface, and which will implement the solution to this calculus problem.

This time, the `getImplementationDescription` method must return the "Efficient" string. The `count` method must be implemented using neither recursion nor iteration, i.e., using directly the solution you obtained to the previous calculus expression.

An example of execution of the `CandleCounterTest` class is shown below. Uncomment the last line of the main in order to test the efficient strategy.

```
; java CandleCounterTest
Testing Iterative:
    count(0): 0, OK
    count(1): 1, OK
    count(2): 3, OK
    count(6): 21, OK
    count(10): 55, OK
    count(1000): 500500, OK
Testing Recursive:
    count(0): 0, OK
    count(1): 1, OK
    count(2): 3, OK
    count(6): 21, OK
    count(10): 55, OK
    count(1000): 500500, OK
Testing Efficient:
    count(0): 0, OK
    count(1): 1, OK
    count(2): 3, OK
    count(6): 21, OK
    count(10): 55, OK
    count(1000): 500500, OK
```

Section 4. Counting bacteria

Some problems are more easily described using recursion than iteration.

For instance, most bacteria reproduce by means of the binary fission. This means that if a bacterias takes a day to reproduce, each day we will have double the number of bacteria than the previous day. Let us suppose that the first day there is an only bacterias (`count(1) = 1`). Then, the number of bacteria 6 days after will be:

```
count(6) = 2 * count(5)
         = 2 * 2 * count(4)
         = 2 * 2 * 2 * count(3)
         = 2 * 2 * 2 * 2 * count(2)
         = 2 * 2 * 2 * 2 * 2 * count(1)
         = 2 * 2 * 2 * 2 * 2 * 1
         = 2 * 2 * 2 * 2 * 2
         = 2 * 2 * 2 * 4
         = 2 * 2 * 8
         = 2 * 16
         = 32
```

Following the three previous sections, implement the following interface:

```
public interface BacteriaCounter {
    String getImplementationDescription();
    long count(int days);
}
```

Provide three implementations for this interface: `BacteriaCounterIterative`, `BacteriaCounterRecursive` and `BacteriaCounterEfficient`. Follow the same guidelines than in the previous sections.

This time, write first the recursive solution, then the iterative one and finally, the efficient one.

You can test your implementation using the BacteriaCounterTest.java class.

You can see an example of the `BacteriaCounterTest` class execution below:

```
; java BacteriaCounterTest
Testing Iterative:
    count(1): 1, OK
    count(2): 2, OK
    count(3): 4, OK
    count(6): 32, OK
    count(10): 512, OK
    count(20): 524288, OK
Testing Recursive:
    count(1): 1, OK
    count(2): 2, OK
    count(3): 4, OK
    count(6): 32, OK
    count(10): 512, OK
    count(20): 524288, OK
Testing Efficient:
    count(1): 1, OK
    count(2): 2, OK
    count(3): 4, OK
    count(6): 32, OK
    count(10): 512, OK
    count(20): 524288, OK
```

Section 5. When to use recursion or iteration?

Among the scientific community, it is a generally accepeted fact that any problem that can be solved by recursion, can be also solved by iteration, and viceversa (Church-Turing's thesis).

It is also true that recursive solutions are less efficient than iterative ones (if we neglect compiler optimizations).

The combination of these two facts may make you think that it is never worth to use recursive solutions, since the iterative equivalents are much more interesting. However, this is not always true.

Discuss with your partners when it is worth to use recursive solutions, and write down your conclusions.

Hint: remember the problem approach when we wanted to count candles:

```
count(6) = 1 + 2 + 3 + 4 + 5 + 6 = 21
```

and compare it to the approach when we wanted to count bacteria:

```
count(6) = 2 * count(5)
         = 2 * 2 * count(4)
         = 2 * 2 * 2 * count(3)
         = 2 * 2 * 2 * 2 * count(2)
         = 2 * 2 * 2 * 2 * 2 * count(1)
         = 2 * 2 * 2 * 2 * 2 * 1
         = 2 * 2 * 2 * 2 * 2
         = 2 * 2 * 2 * 4
         = 2 * 2 * 8
         = 2 * 16
         = 32
```

1.2. Get the maximum item of an array

Write a class, called `ArrayUtils`, which must contain a static method, `public static int getMax(int numbers[])`. This method must return the highest integer of the array that the method receives as argument. Implement this method using a recursive strategy.

Hint: the standard recursive way to solve these problems is to follow a "divide and conquer" strategy, where with each call to the recursive method, smaller array chunks are processed.

Hint: avoid inefficiencies in your code by not creating new arrays for each "chunk" of the original array. Instead, you can create an auxiliar method that receives the original array and a position, which delimit the relevant "chunk" at each recursive step.

You can check your implementation with the ArrayUtilsTest.java class.

You can see below an example of the execution of such class:

```
; java ArrayUtils
Test 1: OK
Test 2: OK
Test 3: OK
Test 4: OK
Test 5: OK
```

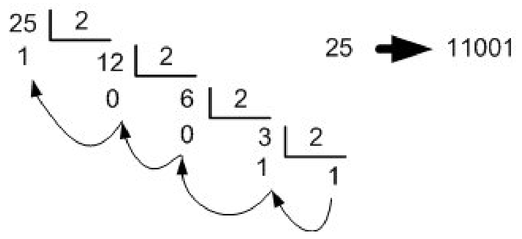
1.3. From integer to binary and from binary to integer

This exercise is intended to improve the one implemented in class, about translation from int to binary. Thus, we are going to implement a class, called `BinOperations`, which implements the interface:

```
public interface BinConverter {
    public String int2bin(int n);
    public int bin2int(String s);
}
```

Implement, using recursion, the following methods:

- `public String int2bin(int n)`: Accepts an integer as parameter and returns a String with its binary representation. If the received number is negative, the method throws a `IllegalArgumentException` exception.



- `public int bin2int(String s)`: Accepts a String object wich contains the binary representation of a positive integer, and returns this integer value. If the received parameter does not correpond with the expected format, the method throws an `IllegalArgumentException` exception.

Please remember that you must implememt both methods using recursion, not iteration.

The following methods will help you while programming the `bit2int` method. Go to the Java API to find how to use them:

- `public static int parseInt(String s)`, from `Integer` class.
- `public String substring(int beginIndex, int endIndex)`, from `String` class.

Hint: the standard recursive way to solve `bin2int` is to follow a "divide and conquer" strategy, where with each call to the recursive method, smaller strings chunks are processed. This time, it is worth to create substrings of the original string with each recursive call, due to the complexity of preserving the original string unmodified in every recursive call.

You can test your implementation with the class BinOperationsTest.java.

You can see an execution example below:

```
; java BinOperationsTest
Testing int2bin(-100): throws an IllegalArgumentException, OK
Testing int2bin(-1): throws an IllegalArgumentException, OK
Testing int2bin(0): "0", OK
Testing int2bin(1): "1", OK
Testing int2bin(2): "10", OK
Testing int2bin(3): "11", OK
Testing int2bin(4): "100", OK
Testing int2bin(5): "101", OK
Testing int2bin(6): "110", OK
Testing int2bin(7): "111", OK
Testing int2bin(8): "1000", OK
Testing int2bin(9): "1001", OK
Testing int2bin(10): "1010", OK
Testing int2bin(25): "11001", OK
Testing int2bin(256): "100000000", OK
Testing int2bin(2344): "100100101000", OK

Testing bin2int("00121011"): throws an IllegalArgumentException, OK
Testing bin2int("00110alcortes"): throws an IllegalArgumentException, OK
Testing bin2int("hello"): throws an IllegalArgumentException, OK
Testing bin2int("0"): 0, OK
Testing bin2int("1"): 1, OK
Testing bin2int("10"): 2, OK
Testing bin2int("11"): 3, OK
Testing bin2int("100"): 4, OK
Testing bin2int("101"): 5, OK
Testing bin2int("110"): 6, OK
Testing bin2int("111"): 7, OK
Testing bin2int("1000"): 8, OK
Testing bin2int("1001"): 9, OK
Testing bin2int("1010"): 10, OK
Testing bin2int("11001"): 25, OK
Testing bin2int("100000000"): 256, OK
Testing bin2int("100100101000"): 2344, OK
```

2. Homework

2.1. Palidromes

A palindrome is a word, phrase, number or other sequence of units that can be read the same way in either direction (the adjustment of punctuation and spaces between words is generally permitted). A well known example could be "Was it a rat I saw?". As you can check the sequence of characters is the same in both reading directions.

Write a class, called `Palindrome`, which must have a method, `public static boolean check(String s)`. This method must return `true` if the string in the argument is a palindrome, or `false` otherwise.

In order to simplify the problem, assume that palindromes do not ignore accented letters nor punctuation, but they do ignore capitalization.

Implement the recursive strategy efficiently, avoiding to copy the original string or substring in each recursive step.

Test your implementation using the PalindromeTest.java class.

An example of execution would be as follows:

```
; java PalindromeTest
Testing palindrome "": true, OK
Testing palindrome "abccba": true, OK
Testing palindrome "Ana": true, OK
Testing palindrome "Level": true, OK
Testing palindrome "Dabale arroz a la zorra el abad": true, OK
Testing palindrome "Satan oscillate my metallic sonatas": true, OK
Testing non-palindrome "Hola mundo": false, OK
Testing non-palindrome "Si a la ingenieria telematica": false, OK
```