CS-107 : Self Assessment
Cryptography & Steganography

Hamza REMMAL, Rafael PIRES & Jamila SAM

-

version 2.0.2

Contents

1 Introduction **3**

2 Structure and supplied code **3**

2.1 Structure . . . 3
2.2 Supplied code . . . 3
2.3 Advanced code . . . 4
2.4 Tests . . . 5

3 Task 1: Implementation of utility methods **7**

3.1 Bit manipulation: `Bit.java` file . . . 7
3.1.1 Method `getXthBit` . . . 7
3.1.2 Method `getLSB` . . . 8
3.1.3 Methods `embedInXthBit` and `embedInLSB` . . . 8
3.1.4 Byte decomposition . . . 9
3.2 Handling `String`: the `Text.java` file . . . 9
3.2.1 Methods implemented for you . . . 9
3.2.2 It is your turn to implement . . . 10
3.3 Image manipulation: `Image.java` file . . . 10
3.3.1 Discovering the ARGB format . . . 10
3.3.2 A second image format: Grayscale . . . 12
3.3.3 Black pixel or white pixel? The binary format . . . 12
3.3.4 From pixel manipulation to image manipulation . . . 12
3.4 Tests . . . 14

4 Task 2: Cryptographic algorithms **15**

4.1 Your First Algorithm: Caesar's Algorithm . . . 15
4.2 Caesar revisited: Vigenère's algorithm . . . 16
4.3 Encryption with XOR . . . 17
4.4 Vernam encryption: One-Time-Pad . . . 17
4.5 An advanced algorithm: Cipher Block Chaining (CBC) . . . 18
4.6 Tests . . . 19

5 Task 3: Steganography **20**
5.1 Direct image encoding . . . 20
5.1.1 Embedding images . . . 20
5.1.2 Unveiling . . . 21
5.1.3 Tests . . . 21
5.2 Text encoding . . . 22
5.2.1 Inlay and unveiling . . . 22
5.3 Tests . . . 23
5.4 Combining cryptography and steganography . . . 23

6 Capture the Flag Challenge: Find the Flag **24**
6.1 Introduction . . . 24
6.2 Instructions . . . 24
6.2.1 Play Fair! . . . 24

7 Extensions **25**
7.1 Elements of cryptanalysis . . . 25
7.2 CBC extension . . . 25
7.3 Command interpreter . . . 25

8 Theoretical supplement **27**
8.1 ARGB representation . . . 27
8.2 Signed and unsigned integers . . . 28
8.3 Random number generators . . . 29

This document uses color and contains clickable links. It is best viewed in digital format.

1 Introduction

Protecting digital data and systems is a crucial issue of our time. The field of cryptography studies techniques for securing our data, provides so-called "crypto-systems" for encoding and decoding data.

On the other hand, Steganography is the art of concealment: this involves hiding information, such as an image or text, in a medium such as another image or text. The aim is to make it impossible to see that the medium is hiding the message. Unlike cryptography, which makes information incomprehensible, but still apparent[1], steganography aims to ensure that the information itself is not even detected. It is used, for example, in situations where the messages we transmit are under surveillance, such as under a dictatorship, or by terrorist groups[2]. More legally, it is used for digital watermarking which enables a file to be invisibly (or visibly) marked, so that its origin can be traced; for example, in the event of a file leak.

The aim of this self-assessment is to give you a basic introduction to some classic crypto-systems as well as introducing you to the field of steganography. In fact, you will be combining the two by hiding encrypted messages in images!

2 Structure and supplied code

2.1 Structure

The project is divided into three main tasks :

- Implementing utilities: in this first task, you will implement a number of methods whose role will be to help you implement the rest of the project more easily ;
- Cryptographic algorithms: in a second step, you will implement a number of cryptographic algorithms such as Caesar's algorithm or CBC encoding ;
- Steganographic algorithms: To complete your project, you will need to set up two algorithms, one easy and another a little bit more complicated, which will enable you to hide data in images.

Steps 2 et 3 can be implemented independently.

2.2 Supplied code

To access the provided material, carefully follow the instructions given in the project's description.

We also provide the javadoc documentation of the project under the following link. Make yourself familiar with the material through this documentation (the code itself may not necessarily be within your grasp, and we do not expect you to understand the details).

[1] We know that information exists and that they want to prevent us from reading it.
[2] This is also of interest to forensic scientists (see references at the end of the document).

- All mandatory code must be produced in :
  - `Bit.java`,
  - `Text.java`,
  - `Image.java`,
  - `Encrypt.java`,
  - `Decrypt.java`,
  - `ImageSteganography.java`,
  - `TextSteganography.java`
- The headers of the methods to be implemented are supplied and **must not be modified**.
- The supplied `SignatureChecks.java` file lists all the signatures that **must not be changed**. In particular, it is used as a control tool during submissions. It will allow you to call all the required methods **without testing their operation**. Check that this program **compiles** correctly, before submitting.
- The notation `aClass::aMethod` is read: "the `aMethod` method of the `aClass` class".

2.3 Advanced code

Some of the necessary functions required for the project, such as file and image manipulation, are too advanced for this course. Therefore, a utility file, `Helper.java`, is provided with the implementation of these methods. It contains :

- **Image manipulation** methods
  - The `public static int[][] readImage(String path)` method, which takes as parameter the access path to the image to be read. This method returns a two-dimensional integer array (`int[][]`) representing an image in the *ARGB* format.
  - The `public static void writeImage(String path, int[][] image)` method, which takes as parameters: the path where to store the image and a two-dimensional array representing the image in *ARGB* format
- Methods for **handling files**
  - The `public static String read(String path)` method, which reads the contents of a file whose path is given as an argument
  - The `public static void write(String path, byte[] content)` method, which writes the content of the array *content* to a file following the access path given by *path*
- **Display** methods
  - The `public static void show(int[][] image, String title)` method, which creates a new window and displays the image in parameter. The window title corresponds to the value of the second parameter.
  - The `public static void dialog(String title, String message)` method, with similar behavior to `Helper::show`, displays text in a new graphical window.
- The `public static byte[] generateRandomBytes(int length)` method will randomly generate a sequence of bytes whose length depends on the *length* parameter
- Method:1 for **error handling**:

– The `public static <T> T fail(String fmt, Object ... args)` method terminates program execution. It takes as parameters the format of the `fmt` message, as well as the *objects* needed to format the final message. (Formatting works in a very similar way to `String::format`)

The use of these methods in the appropriate contexts will be described in the statement or given as an example in the main program. For the moment, it is just a matter of taking note of their existence.

2.4 Tests

Important :

You are responsible for checking that your program behaves correctly. The supplied `Main.java` file, partially written, is a simple way of testing your implementations. **It is up to you to complete `Main.java` by invoking your methods in the right way to check that your program is error-free.**

To give you feedback on your project, we will use automated tests, which will pass randomly generated input data to the various functions of your program. **So there will also be tests to check how *special cases* are handled**. Therefore, it is important that your program correctly handles any and all *valid* input data.

When it comes to error handling, it is customary to test method input parameters; for example, to check that an array is not null, and/or is the right size, etc. These tests generally make debugging easier, and help you to reason about a function's behavior.

We will assume that function arguments are valid (unless explicitly stated otherwise). To guarantee this assumption, we will need to use Java assertions[3]. An assertion is written in the form: `assert expr;` where `expr` is a Boolean expression. If the expression is evaluated to `false`, then the program throws an error and stops, otherwise it continues normally. For example, to check that a method parameter named `key` is not `null`, you can write `assert key != null;` at the beginning of the method body (assertions don't necessarily have to appear at the beginning of the body).

Assertions must be activated to work. This is done by running the program with the `-ea` option:

- for IntelliJ
- for Eclipse.

Finally, we strongly recommend that you familiarize yourself with the use of the debugger right from the start. As soon as you test your first function, try setting breakpoints to examine the values of the variables involved.

[3] We will come back to them in detail later, but their use is intuitive enough for us to use them already

Here is a summary of the main instructions/guidelines for coding the project :

- Method parameters will be considered error-free unless explicitly stated otherwise.
- Method headers must remain unchanged: the `SignatureCheck.java` file must therefore contain no **compilation** errors, to guarantee compatibility with the checker.
- In addition to the imposed methods, you are free to define any additional methods you feel are relevant. **Modularize and try to produce clean code!**
- Checking that your program behaves correctly is your responsibility. Nevertheless, we provide the `Main.java` file, illustrating how to invoke your methods to test them. The test examples provided are **not exhaustive,** and you are authorized/encouraged to modify `Main.java` for further verification. Remember to run the tests provided one by one to "check" the behavior of your code.
- Your code must respect standard naming conventions.
- The instruction `System.exit()` must not be used.
- The project will be coded without the use of external libraries. If you have any doubts about the use of a particular library, ask us and, above all, pay attention to the alternatives that IntelliJ (or Eclipse) offers to import on your machine.
- Your project **must not be stored in a public repository** (such as a public GitHub). You can use the on-premise GitLab instance provided by EPFL, as shown in the course, and it is highly recommended that you learn to collaborate using `git`.
- Use the Ed forum. Before asking your question, check that there are no similar questions. This will make it easier to use the forum and avoid getting lost in the questions. Please also respect the forum structure: **Use category *Mini-projet 1***.

3 Task 1: Implementation of utility methods

To make it easier for you to implement cryptographic and steganographic algorithms, we will dedicate this first task of the project to the implementation of three types (classes) of utilities. The first methods, implemented in the `Bit.java` file, will relate to the manipulation of *bits*. The methods of the second class, implemented in the `Text.java` file, will help you manipulate strings. You will implement the methods in `Image.java`, which will help you to manipulate images easily. The role of these various utilities will become clear as you make progress.

> It Is a good idea to test what you develop as you progress. Start by reading the advice in the `Tests` section (see 3.4).

3.1 Bit manipulation: `Bit.java` file

Either for *cryptography* or *steganography*, bit manipulation will be a fundamental part of your implementation.

> Here you will use binary operators and code in the `Bit.java` file.

Note: to check your results easily, you can use 2's complement decimal/binary converters, such as the one available here.

3.1.1 Method `getXthBit`

To begin with, you will be asked to implement the `Bit::getXthBit` method:

```
public static boolean getXthBit(int value, int pos)
```

This method must extract the bit at position `pos` in the sequence of bits forming an integer `value`. The positions are numbered from right to left (the least significant bit is at position zero). If the value of the bit is `1`, this method should return the boolean value `true`, otherwise it will return the value `false`.

Here's an example where we extract each of the 32 bits from the `-12` literal:

```
int value = -12;
// Integer.SIZE gives the size of integers (in terms of number of bits)
boolean[] computed = new boolean[Integer.SIZE];
for (int i = 0; i < Integer.SIZE; ++i) {
    computed[i] = Bit.getXthBit(value, i);
    }
```

The content of the `computed` array, after execution of this code, will be as follows:

```
{ false, false, true, false, true, true, true, true, true, true,
  true, true, true, true, true, true, true, true, true, true, true,
  true, true, true, true, true, true, true, true, true, true, true
}
```

Remember that for each method you implement, you will be asked to use assertions to check that the input data is valid. So you will typically need to think about *edge cases* when formulating these assertions. For example, to formulate assertions relating to `Bit::getXthBit`, you need to ask yourself questions such as "What should happen if `Bit::getXthBit` is invoked with parameters `(value = 0, bit = 100)`?

3.1.2 Method `getLSB`

Now that you can extract all the bits from an integer, we focus on one particular bit, the *LSB (the least significant bit)*. Access to the least significant bit is often useful. You are therefore asked to implement the `Bit::getLSB` method with the signature :

```
public static boolean getLSB(int value)
```

This will tell you whether the *LSB* of an integer is *1* or *0*. Using the same convention as above, we will represent the integer value `1` by the boolean `true`, and the integer value `0` by the boolean `false`.

For example, calls to `Bit::getLSB` on the values `-12`, `12`, `0` and `255` should return `false`, `false`, `false` and `true` respectively.

3.1.3 Methods `embedInXthBit` and `embedInLSB`

Extracting bits from an integer is not the only operation required to complete the project. The reciprocal operation of "modifying" given bits will prove just as useful. For this step, you are asked to implement the `Bit::embedInXthBit` and `Bit::embedInLSB` methods, reciprocals of the `Bit::getXthBit` and `Bit::getLSB` methods respectively.

The `Bit::embedInXthBit` method will take as input an integer, as well as the position of the bit to be modified and the new value to be given to the latter. As a return value, the method must return the integer, but with the bit in the given position set to the new value. This can be summarized in the following signature:

```
public static int embedInXthBit(int value, boolean m, int pos)
```

As explained above, the parameters of this method are :

- the integer to be modified (`value`);
- the new value of the selected bit (`m`) in the returned integer; the value `true` to set the bit to `1` and `false` to set it to `0`
- the position of the bit to be modified (`pos`).

For example:

```
embedInXthBit(-12, false, 2); // returns the integer -16
```

The second method, the reciprocal of `Bit::getLSB`, will similarly return an integer equal to the parameter where the least significant bit has been changed:

```
public static int embedInLSB(int value, boolean m)
```

For example:

```
embedInLSB(-12, true); // returns the integer -11
```

3.1.4 Byte decomposition

To complete this first stage of your project, you will be asked to implement a first method (`Bit::toBitArray`) that will allow you to decompose an integer into an array of booleans representing its corresponding bits. You will also implement the inverse method of `Bit::toBitArray`, which lets you reshape an integer of type `byte` from a boolean array (`boolean[]`).

The method signatures are as follows:

```
public static boolean[] toBitArray(byte value)
```

```
public static byte toByte(boolean[] bitArray)
```

For example:

```
Bit.toBitArray(-4); // return [true, true, true, true, true, true, false,
    false]
```

> Remember to add the logic to validate your method parameters (assertions). The boolean array is assumed to exist (different from `null`).

3.2 Handling `String`: the `Text.java` file

Throughout the project, you will be working with strings. We will represent them in **three** different formats/types.

- The `String` type: natural representation of strings in Java;
- The `byte[]` type: memory representation of character strings;
- The `boolean[]` type represents a character string in *bit map* format ("black & white", details of the format to follow).

A character string can be represented using the predefined type `String`. Working with this type alone, however, can cause problems. For example, some characters cannot be displayed, making it difficult to check the results of the algorithms. This is the main reason why we are going to use the other two alternative formats.

3.2.1 Methods implemented for you

The memory representation of a `String` depends by default on the encoding of the machine used.. For the sake of simplicity, we will adopt a specific encoding, *UTF-8*, and provide you with the following two methods:

```
public static byte[] toBytes(String str)
```

```
public static String toString(byte[] bytes)
```

The first takes a character string as input and returns its representation as an array of bytes in UTF-8 encoding. The second performs the inverse operation, interpreting a sequence of bytes in UTF-8 encoding. This allows you to obtain machine-independent results, and the statement will suggest when these methods should be used.

3.2.2 It is your turn to implement

For this step, you will need to implement two methods. The first will allow you to switch from the `String` format representation to the `boolean[]` format representation. The second is the inverse of the first.

To begin with, we ask you to implement the `Text::toBitArray` method, whose signature is :

```
public static boolean[] toBitArray(String str)
```

The role of this method is to convert a string into an array of booleans by interpreting its UTF-8 encoding. It will convert the string into an array of `bytes` (according to UTF-8), then translate each of the bits in these `bytes` into a Boolean in sequence. Remember to use what you have coded in the `Bit.java` file.

For example, calling the `Text::toBitArray` method with the character string *`"ô$"`* as parameter ("o circumflex" and "dollar sign") will return the following array, representing a character string in *bit map* format:

```
{ true, true, false, false, false, false, true, true,
true, false, true, true, false, true, false, false, false, false,
true, false, false, true, false, false }
```

The second method, the inverse of the one you have just implemented, will take as parameter a string in *bit map* format, and return the representation in `String` format.

```
public static String toString(boolean[] bitArray)
```

This method is an *overload* of the `toString` method already provided. The boolean array is assumed to exist (different from `null`, to be verified using an assertion). Note that the size of a byte can be obtained using the formula `Byte.SIZE`. If the length of `bitarray` is not a multiple of this size, the extra data at the end of this array can be ignored.

3.3 Image manipulation: `Image.java` file

In this project, you will have to manipulate images modeled as arrays of *pixels*. You will now be asked to code various utilities needed for the required processes.

3.3.1 Discovering the ARGB format

As explained in the theoretical complement (see section 8), a pixel is nothing more than a color in ARGB representation.

- For this part of the project, **the use of binary operators is mandatory**.
- The theoretical complements show you that the primary components of an ARGB color are between between 0 (inclusive) and 255 (inclusive). This should help you understand why the `int` type is sufficient to represent a pixel without loss of information. There is, however, an important question to consider: does the fact that Java doesn't allow you to use unsigned numbers change anything? This is a point to which we will have to pay close attention in the following.

In order to familiarize yourself with color modeling in ARGB format, you are asked to implement:

- four methods of the `Image` class for extracting the primary components of a pixel;
- as well as a reciprocal method for forming a pixel from its primary components.

Here are the signatures of the methods to be implemented, along with a description of their expected behavior:

- The `public static byte alpha(int pixel)` method will extract the *alpha* (transparency) component of a given pixel.
- The `public static byte red(int pixel)` method will extract the pixel's red component.
- The `public static byte green(int pixel)` method will extract the green component of the pixel.
- Finally, you should be able to extract the blue component from a pixel using the `public static byte blue(int pixel)` method.

As an example, a code snippet is provided below to help you with your work.

```
int pixel = 0b11111111_11110000_00001111_01010101 ;// -1044651 (0xFFF00F55)
// We extract the different values:
byte alpha = Image.alpha(pixel); // -1 ( which is 255, 0xFF, unsigned)
byte red   = Image.red(pixel) ;  // -16 (which is 240, 0xF0, unsigned))
byte green = Image.green(pixel); // 15 (0x0F)
byte blue  = Image.blue(pixel);  // 85 (0x55)
```

Now that you are able to extract the primary components of a given pixel, it is time to try your hand at forming a pixel using its primary components. To do this, implement the method `Image::argb` with signature :

```
public static int argb(byte alpha, byte red, byte green, byte blue)
```

The following example should help you understand what is expected:

```
int pixel = argb (-1, -16, 15, 85);// -1044651 (0xFF_F0_0F_55)
```

- When implementing this method, please note that a color must be unsigned (value between 0 and 255). Consequently, the bytes supplied as parameters must be converted to unsigned values before the integer is reconstituted.
- Remember to test your implementation using the tests provided in the `Main.java` file, which you are encouraged to extend as you wish.

3.3.2 A second image format: Grayscale

Some of the processing required will involve working with simplified image formats. Specifically, these will be grayscale or black-and-white images.

To begin with, you will be asked to implement the `Image::gray` method, which will calculate the gray level of a given pixel. This can easily be calculated as the average of the primary components (*transparency* excluded). The method to be implemented will have the following signature:

```
public static int gray(int pixel)
```

> By construction, the integer returned will always have a value between 0 and 255: in fact, the maximum value corresponds to the situation where each byte of the constructed integer is 255 (in which case the average is also 255) and the minimum value is zero (each byte is zero, and therefore their average is also zero).

Here's an illustrative example of how this method should work :

```
gray(2146438997) // 2146438997 = 0xFFF00F55 = (240 + 15 + 85) / 3 = 113
```

additional tests are provided in the `Main.java` file.

3.3.3 Black pixel or white pixel? The binary format

The last image format you will have to implement is not a commonly used format. It Is a "home-made" format that will make certain tasks. We will call this format the *binary format* and define it from the *greyscale format*.

To convert a pixel to binary format, you will need to implement the `Image::binary` method, which will take the *gray scale* (`gray`, which must be a value between 0 and 255) as parameter and a threshold (`threshold`). This method should return the value `true`, representing a white pixel, if the pixel's gray level is greater or equal than the threshold. Alternatively, it should return the value `false`, which will represent a black pixel, if the gray level is strictly below the given threshold.

The signature of the `Image::binary` method is as follows:

```
public static boolean binary(int gray, int threshold)
```

Due to the simplicity of `Image::binary`, we do not provide an example of how to execute the method. However, a number of tests are available in `Main.java`.

3.3.4 From pixel manipulation to image manipulation

We now have tools for working with individual pixels in various formats. Now it is time to use them to work at full image level.

The first method you will need to implement is the `Image::toGray` method. It will convert an image from *ARGB* format to *grayscale* format. The method signature will look like this:

```
public static int[][] toGray(int[][] image)
```

The `image` parameter will be considered as correct if it is non-null.

At the same time, implement the `Image::toBinary` method, which allows you to convert an image from *grayscale* format to *binary* format. The method signature is provided below:

```
public static boolean[][] toBinary(int[][] image, int threshold)
```

> For the moment, the implemented methods allow conversion from *ARGB* to *grayscale* and *binary* formats. In your opinion, is it possible to implement methods reciprocal to the ones mentioned above? If not, why not?

Let's now look at the reciprocal functions of `Image::toGray` and `Image::toBinary`.

We start by looking at `Image::fromGray`, which converts an image from grayscale to ARGB format.

> The questions that arise are: "Is it possible to reshape a pixel from its grayscale? If not, what would be the best approximation of a pixel from its gray level?"

As you may have guessed, we need to use an approximation because of the loss of information (retrieving a full color from its corresponding gray scale is not possible).

Implement the `Image::fromGray` method, whose signature is:

```
public static int[][] fromGray(int[][] image)
```

and whose execution obeys the following specification:

> Let a pixel in the grayscale image be `gray`. The corresponding pixel in the ARGB image to be returned is such that :
>
> - it is completely opaque (`0xFF` for the `alpha` component);
> - and each of its red, green and blue components has the value `gray`.

Let's now look at moving from binary format to grayscale format. Remember that the *binary* format only allows us to represent two colors (black and white). The *grayscale* format, on the other hand, allows us to represent a total of 256 colors.

Clearly, converting from grayscale to binary results again in a loss of information. It is hence impossible to recalculate the grayscale from the *binary* representation. However, it is still possible to approximate the calculations.

With this in mind, you are asked to implement the `Image::fromBinary` method, which will convert an image from *binary* format to *grayscale* format. The method signature will look like this:

```
public static int[][] fromBinary(boolean[][] image)
```

This method should convert, according to the following rule:

> If a pixel in `image` is white (`true`), its corresponding grayscale value will be 255 (let's call this value it `gray`) On the other hand, if the pixel is black, `gray` will take the value 0.

The grayscale pixel to be calculated in this case is an integer whose most significant byte is `0xFF` and whose other bytes have the value `gray`.

> Code duplication must be avoided.

Remember that tests are provided in the `Main.java` file.

3.4 Tests

To test the behavior of your utilities, use the `Main.java` program. Here you can invoke all the methods you need for testing. Some test examples are provided in the form of methods you can use and draw inspiration from. These examples are not exhaustive. Add to them as you need to test all the methods implemented.

> A useful debugging trick is to use IntelliJ's ability to display values in hexadecimal or binary format. Values in hexadecimal or binary format: in the debugger, right-click on an integer value and then *View as > Hex or View as > binary.*

4 Task 2: Cryptographic algorithms

In this second part of the project, you will discover the world of cryptography. This is a basic introduction to the field, presenting some of the fundamental algorithms that paved the way for much more complex techniques.

You will have to implement the methods of the files `Encrypt.java` and `Decrypt.java` under the package `ch.epfl.cs107.crypto`.

The first file contains *encryption* methods. We are going to look at different methods for encrypting a text using a key. The aim is to achieve results whose meaning is no longer recoverable without the key once encrypted. It Is important to note that many cryptographic algorithms can be broken despite this.

The second file covers decryption methods. We Are only interested in simple (or simplified) techniques for recovering the original text.

> The encryption/decryption methods will all work with messages expressed as sequences of *bytes* (representing sequences of characters in a more universally "readable" format than `String`).

4.1 Your First Algorithm: Caesar's Algorithm

The first technique to be discussed is *shifting*. Shifting is one of the oldest forms of cryptography. To make a text unreadable, all you have to do is shift all the letters by a certain number of positions. To recover the original text, all that is needed is to make the reverse shift. The number of shifts required is encryption key. To put the idea into concrete terms, here's a simple example: Let's imagine an alphabet with four letters A, B, C and D. We can assign numbers to each of the symbols, say 0, 1, 2 and 3 respectively. This gives us the following encoding:

A	B	C	D
0	1	2	3

This encoding can be used to encrypt a message expressed in this alphabet. Let's take the message **BCDACAD** and B as the encryption key.

Caesar's algorithm consists in adding the value of the key code to each letter of the message. This addition is done modulo the size of the alphabet (in this case, 4), to always remain within the alphabet. As a reminder, x modulo y gives the remainder of the integer division of x by y (for example, 3 mod 4 = 3, 7 mod 4 = 3).

The following table shows the result of the encryption performed on the previous example:

Text	B (1)	C (2)	D (3)	A (0)	C (2)	A (0)	D (3)
Key	B (1)	B (1)	B (1)	B (1)	B (1)	B (1)	B (1)
Results (mod 4)	C (2)	D (3)	A (0)	B (1)	D (3)	B (1)	A (0)

The message **BCDACAD** encrypted with key B is therefore **CDABDBA**. To decrypt, shift one backward (or 3 = 4–1). This is the same as using the reverse key, in this case D.

Please note that in our case, the characters are not only positive and have values between -128 and 127 inclusive. The result must remain within this range, operating on a principle analogous to modulo calculation, i.e., 127+1 must give -128. Remember to convert the result into bytes, as operations on bytes return an int (integer promotion). A simple "cast" is all it takes.
Be sure to read the supplements on bytes (section 8.2) to solve this part easily.

As part of your project, you will have to implement the following two encoding and decoding functions:

```
public static byte[] caesar(byte[] plainText, byte key)
```

```
public static byte[] caesar(byte[] cipher, byte key)
```

Although the function signature is similar, the implementation will be different. The first, defined in the `Encrypt` class, will encode the `plainText` parameter using Caesar's algorithm with the `key` key. The second function will perform the opposite operation, decrypting the `cipher` using the `key` key.

Before you start coding these methods, ask yourself what is the alphabet's size is if you work with bytes.

4.2 Caesar revisited: Vigenère's algorithm

The Vigenère cipher is a more advanced version of the Caesar. It is very similar, but instead of using a single character key, it uses a word (or phrase) that is repeated throughout the message to encrypt it. We apply the same method as Caesar's cipher to shift the text. Let's use the same example as in Section 3.2, i.e. **BCDACAD** with this time the key **ABC**, keeping the encoding strictly positive (from 0 to 3):

Text	B (1)	C (2)	D (3)	A (0)	C (2)	A (0)	D (3)
Key	A (0)	B (1)	C (2)	A (0)	B (1)	C (2)	A (0)
Results (mod 4)	B (1)	D (3)	B (1)	A (0)	D (3)	C (2)	D (3)

This time, each character can be shifted differently (the first B is shifted by 0, the first C by 1, the first D by 2, etc.).
Similar to the implementation of Caesar's algorithm, you will need to implement the function

```
public static byte[] vigenere(byte[] plainText, byte[] keyword)
```

of the `Encrypt` class, in addition to the function

```
public static byte[] vigenere(byte[] cipher, byte[] keyword)
```

of the `Decrypt` class.
The name and order of the parameters is similar to that of the methods implemented for the Caesar algorithm, so you will be able to infer the role of each of the parameters functions for the Vigenère algorithm.

4.3 Encryption with XOR

The XOR ("exclusive OR") is a well-known logic function in computer science when working with binary representation. It behaves as follows:

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

It can be used to encrypt text. As we are working with a byte coding of the symbols, the implementation of this encryption algorithm will be greatly simplified. In fact, a byte is nothing more than 8 bits in binary, and the "exclusive or" between two bytes can be calculated by simply calling the binary operator (https://www.baeldung.com/java-xor-operator). In the method

```
xor(byte[] plainText, byte key)
```

each symbol of the coded message is obtained by calculating the "exclusive OR" between the character to be coded and the key.

Here's an example of how to execute XOR, again with the text "I want":

```
byte[] plainText = {105, 32, 119, 97, 110, 116};
byte key = 50;
byte[] cipherText = xor(plainText, key);
//Expected value for cipherText: {91, 18, 69, 83, 92, 70}
```

Note: to decrypt, all you need to do is encrypt the encrypted message with the same key, because applying XOR to a value twice will retrieve that same value.

4.4 Vernam encryption: One-Time-Pad

The "One-Time Pad" is a simple and highly effective technique. It consists of using a disposable mask made up of a sequence of bytes of the same size as the text to be encrypted. The i-th symbol of the coded message is simply obtained by an "exclusive OR" between the i-th character of the message to be coded and the i-th character of the mask. To decrypt, the same algorithm is applied to the coded message, using the same mask. The mask is then not reused, to prevent anyone guessing it. The major drawback of this technique is that it is very resource-intensive. For each exchange, we have to generate a new random mask, and one the size of the text, making it impractical.

Complete the method :

```
oneTimePad(byte[] plainText, byte[] pad)
```

Consider that the `pad` is given to you directly by the second argument. Use assertions to ensure that the `pad` is at least as long as the message.

Also code the variant :

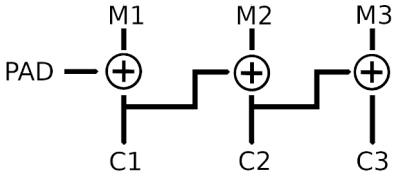

Figure 1: *CBC encryption process, from left to right: the encrypted block C1 of size T is obtained by XOR between the PAD key of size T and the block M1 of size T of the original message. Block C1 is then used as the encryption key for the next block M2, again of size T, of the original message. The encrypted block Ci is obtained by the same principle, using Ci-1 as the encryption key, and is XOR with block Mi..*

```
oneTimePad(byte[] plainText, byte[] pad, byte[] result)
```

that encrypts the `plainText` message using Vernam encryption and a randomly generated `pad`. When the method has finished executing, the two parameters `pad` and `result` will respectively contain the random `pad` used and the result of the encoding. The `pad` must in fact be returned to us for decoding.

> Why can't the function return the result of the encoding? What criticism can be made of this function signature?

4.5 An advanced algorithm: Cipher Block Chaining (CBC)

Cipher Block Chaining is a variant of *One-Time Pad* that is generally used to encrypt a continuous data stream, such as a TV channel or stream. It is usually combined with another encryption method, of your choice, for added security. In our case, we are going to look at the simplified case without combining it with another encryption method.

The concept is simple. You encrypt the text block by block. You start by using the given pad as the first key. The size (let's call it the T) of the pad will be the size of a block. You then consider the first T letters of your text, and XOR the pad, as in "One-Time Pad". This gives you the first part of the result. You will then use the T letters of the resulting ciphertext as a key to encrypt the next T letters of the message, and so on. You will find in Figure 1, the diagram illustrating this algorithm.

Complete the method

```
cbc(byte[] plaintext, byte[] iv)
```

so that it encrypts the text as shown. Bear in mind that the last part of the text won't necessarily have the size of the block, so you will stop XORing where the text ends.

> What criticism could you make of the simplified algorithm proposed for coding CBC?

For more information on the complete CBC, please refer to the "Extensions" part of the project (section 7.2).

4.6 Tests

To test the behavior of your new methods, use the `Main.java` program again. Once again, these examples are not exhaustive. You are encouraged to complete them to test all the methods implemented in this part of the project.

5 Task 3: Steganography

This final part of the project focuses on hiding an image or a text within another image. You will be using one of the simplest methods: Least Significant Bit Encoding, or LSB, which involves separating the information you want to hide into individual bits, and storing them into the least significant (and therefore with the least effect) bit of each pixel in the medium.

For this project, you will be asked to implement two types of encoding:

- Direct image encoding: hide (encode) and reveal a black & white image. It involves:
  - converting a supplied image into a black & white image;
  - encoding the supplied black & white image, the message or the *load*, into another image, the *cover*;
  - extracting the least significant bits from the cover to reveal the hidden message.
- Text encoding: encode and reveal text. It involves :
  - preparing a text message (i.e., converting a `String` to an array of bits representing the UTF-8 encoding of each character in the `String`) ;
  - encoding the bit array in a *support* image;
  - extracting the least significant bits and convert them to a `String`.

Warning None of the methods you will be implementing should modify the original support (parameters). It must return a modified version of the latter in which the message has been hidden.

5.1 Direct image encoding

In this first part, you will hide a binary image, the *load* composed solely of black and white pixels, in another image, the cover (which may be in color). To do that, you will have to change the last bit of each pixel in the cover to the value of the corresponding pixel in the load.

To do this, you will first convert a color image into a binary image (load), passing through a grayscale version; then you will iterate over each pixel of another image (support) to modify the last bit, so that the value of the load pixel is stored in that bit.

You will also implement the unveiling of a hidden image in a similar way: you will iterate over each pixel of the cover and read the value of the last bit to reformat the load as a binary image. These steps are described in details below.

To implement this part, remember to use the appropriate utility methods in the `Bit.java` file. Images provided as parameters will be assumed to be correctly sized (not `null`, not containing rows equal to `null`, not empty and not containing empty rows).

5.1.1 Embedding images

In the `ImageSteganography.java` file, implement the method

```
embedBW(int[][] cover, boolean[][] load)
```

which hides a binary `load` image in an ARGB `cover` image by changing the last bit of each `cover` pixel to the value of the corresponding `load` pixel.

So you iterate over each `load` pixel, starting at the top left [4], and replace the last bit of the `cover` pixel at the same position with the value of the `load` pixel.

Important: the embedding must not be applied to the **original support, which must remain unchanged**. .

You should also bear in mind that the load must be smaller than or equal to the support.

Then implement

```
embedGray(int[][] cover, int[][] grayImage, int threshold)
```

which hides a grayscale image `grayImage` in a `cover` image. The latter will first be converted into a binary image according to the `threshold` threshold.

In a similar fashion, implement the

```
embedARGB(int[][] cover, int[][] argbImage, int threshold)
```

which hides a color image in a `cover` image. The latter is first converted into a grayscale image, then into a binary image according to the `threshold`.

5.1.2 Unveiling

To be able to decode the hidden image, you will have to implement the following method,

```
revealBW(int[][] image)
```

which forms a binary image by reading the values of the least significant bit of each pixel of the `cover` ARGB image.

Example of use

```
int[][] cover = Helper.readImage("cover.jpg");
int[][] messageImage = Helper.readImage("message.jpg");
int[][] gray = Image.toGray(messageImage);
boolean[][] bw = Image.toBinary(gray, 240);
Helper.show(bw, "Black and white image to hide");
int[][] hidden = ImageSteganography.embedBW(cover, bw);
Helper.show(hidden, "Cover with hidden image");
//returns the medium containing the hidden message
boolean[][] decoded = ImageSteganography.revealBW(hidden);
Helper.show(decoded, "Black and white revealed image");
```

5.1.3 Tests

Refer to the `Main.java` program to start testing your implementation. Feel free to complete the tests provided.

[4] the point (0,0) of the image coordinate system is at the top left

5.2 Text encoding

We now need to deal with the case where the load is no longer a binary image, but text in the form of a `String`. This text will first have to be converted into an array of Booleans, via an integer array representation, so that it can be hidden in the medium (which is still an image). With the least significant bit method used in this project, it is only possible to embed a sequence of bits. To hide text (a `String`), we need to start by converting it into an array of bits.

To do this, we convert each character to its numerical value (`byte`), then each integer to its binary representation (`boolean[]`). The result is an array of Booleans corresponding to the binary representation of the `String` you wish to encode.

Furthermore, when we reveal the encoded message, we obtain an array of Booleans. We need to convert this array into the `String` to which it corresponds. To do this, we will apply the inverse operation to that used to transform the `String` into a binary array: we will convert the array of Booleans into an array of integers, then convert this array of integers into characters, and thus into a `String`.

Remember to use all the appropriate utilities in the `Bit.java` file.

5.2.1 Inlay and unveiling

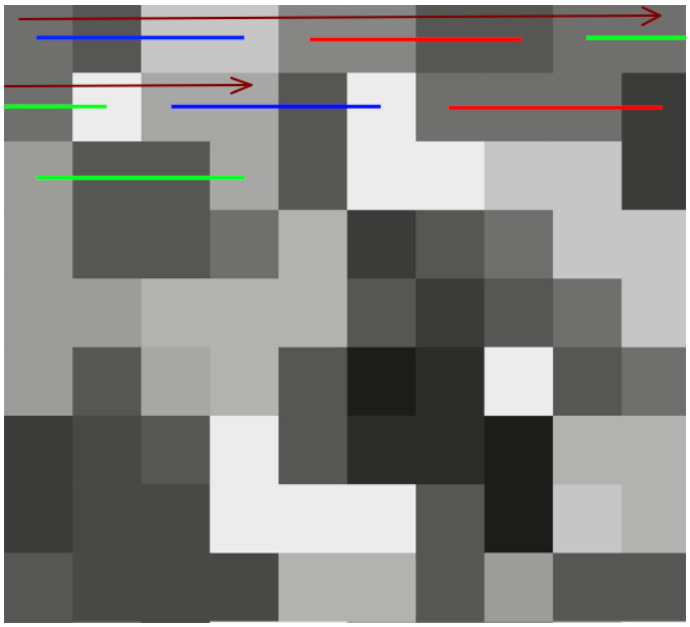

Figure 2: The 4 bits of the first character are embedded in the least significant bit of the first 4 pixels, those of the second character are embedded in the least significant bit of the next 4 pixels, and so on (in this example, we want to hide 6 characters encoded on 4 bits each instead of 16). The arrows indicate the direction of traversal.

Figure 2 illustrates how to embed the String: each new color corresponds to a new character. This figure uses 4 bits for a character, but you will have 16.

Implement

```
embedBitArray(int[][] cover, boolean[] message)
```

which inserts the `message` array into the least significant bits of the first X [5] pixels of the `cover` image, where X is the length of the `message` array. The image `cover` must remain unchanged. If the message is too long to be fully hidden in the image, it will simply be truncated (the rest of the message that cannot be hidden is ignored).

Then implement

```
revealBitArray(int[][] cover)
```

which returns the one-dimensional Boolean array corresponding to the least significant bits of `cover`.

Now implement

```
embedText(int[][] cover, byte[] message)
```

and

```
revealText(int[][] cover)
```

using the methods you have coded so far, hide or reveal a `byte[]` in/from a `cover` image.

5.3 Tests

To test this part of the project, use the `Main.java` program again.

5.4 Combining cryptography and steganography

At this stage, you have coded all the tools needed to enable the combined use of cryptography and steganography. If you wish, you can code methods that explicitly allow such combinations.

For example:

```
int[][] embedText(int[][] cover, byte[] message, String
    encryptionMethodName, byte[] key)
```

```
byte[] revealText(int[][] image, String encryptionMethodName, byte[] key)
```

[5] Starting at pixel 0,0, and traversing from left to right and then up and down

6 Capture the Flag Challenge: Find the Flag

6.1 Introduction

Welcome to this capture the flag challenge! You will need to solve a series of puzzles to uncover a hidden message in images. You will use the provided clues to find the final flag.

6.2 Instructions

1. **The following files are provided:**

   - `hint1.txt`
   - `hint2.txt`
   - `hint3.txt`
   - `hint4.txt`
   - `image1.jpg`
   - `image2.jpg`

2. **Use the clues provided in the `hint*.txt` files in order to find the hidden messages in the `image1.jpg` and `image2.jpg`.**

3. **Solve the clues to discover the hidden messages in the images. The images should be processed in the following order:**

   - `image1.jpg`
   - `image2.jpg`

4. **Implement your solution in the `challenge()` function of the `Challenge` class**

5. **Return the final flag in the format `FLAG{*********}`.**

6. **Take a look at the `Helper.java` class. It might prove very useful ;-).**

6.2.1 Play Fair!

Please do not share the flag with others. This helps keep the challenge fun for everyone :-)
Good luck!

7 Extensions

If, on a personal level, you'd like to take the project further, this section offers suggestions for extensions.

If you wish to render code with extensions, take care to document them in a README file and to preserve the quality of the code (it would be a pity if the addition of extensions lowered the quality of your code and therefore had a negative impact on the style score).

Implementing extensions means you have to be more independent, and assistants are less likely to help you.

7.1 Elements of cryptanalysis

This project will give you a basic introduction to some of the classic crypto-systems. Cryptanalysis, on the other hand, is the discipline that focuses on ways of breaking these crypto-systems, particularly with a view to verifying their robustness.

You can therefore extend the project by adding methods that attempt to decrypt encrypted messages without knowing the key. Naive "brute force" methods, which try all possible keys, can easily be applied to Caesar or XOR encoding for example.

Note that breaking an encoded message using a CBC is possible using "Brute Force", but extremely costly. We would have to test all possible block sizes with 256 possibilities per character of the original "pad." Imagine a block size of 5. If we test the possibilities from size 1 to 5, that would be $256 + 256^2 + 256^3 + 256^4 + 256^5$; that's over a billion possibilities.

It is possible to do better than "brute force" approaches by using more elaborate techniques such as character frequency analysis. Further information on this subject can be supplied on request.

7.2 CBC extension

You can take the CBC concept a step further by adding an encryption method of your choice after the XOR to solidify the implementation, and then decrypt the whole thing. For more information on CBC, take a look at the figures on wikipedia.

7.3 Command interpreter

Another bonus idea is to implement a shell to encrypt and decrypt messages. A shell is a basic command-line program, which you program to include commands entered the console. Here's an example of a basic shell:

```
...
boolean isFinished = false;
Scanner scan = new Scanner(System.in);

while(!isFinished){
    System.out.print("Voulez vous terminer votre programme? [Oui/Non]");
    String s = scan.nextLine();

    if(s.equals("Oui")){
        isFinished = true;
    }else{
        System.out.println("Le programme ne se termine pas.");
    }
```

```
}
...
```

This code corresponds to a program that runs in a loop until you say yes in response to the question. So you can make a shell that has functions and commands, so that you can give it text to encrypt and have it decode it. Of course, you need to make it as "user-friendly" as possible, i.e., easy to access for any user. A "help" command listing all your program's commands and functions is required.

8 Theoretical supplement

8.1 ARGB representation

The representation of a ARGB pixel using an integer is as follows: The color is broken down into 4 components with a value between 0 and 255:

- **Alpha** is the pixel's opacity. If it is 0, then the pixel is "invisible", regardless of its other components. If it is between 1 and 254, then it is transparent and lets the colours of the image pass behind it. If its value is 255, then the pixel is perfectly opaque.
- **Red** is the red light intensity value of the pixel.
- **Green** is the intensity value of the green light in the pixel.
- **Blue** is the blue light intensity value of the pixel.

Assuming that the least significant bit of an integer has the index 0 and the most significant bit has the index 31, then these 4 components are placed on the same integer in such a way: bits 31 to 24 define the alpha value, bits 23 to 16 the red value, bits 15 to 8 the green value and bits 7 to 0 the blue value. For example, if we want to represent a red pixel in RGBA we need the alpha and red component to be at most 255. This gives us:

Alpha	Red	Green	Blue
255	255	0	0
0xFF	0xFF	0x00	0x0
11111111	11111111	00000000	00000000
31 24	23 16	15 8	7 0

The most practical way to represent a color in Java is to use the more concise hexadecimal notation. For example, to define the red pixel, we can write:

```
int red = 0xFF_FF_00_00;
```

The "bitwise" operators make it easy to extract channel values from the integer representation :

```
// Our color in binary
int x = 0b00000000_00100000_11000000_11111111; // -> 2146559 (0x20c0ff)
// Shifts 8 bits to the right
int y = x >> 8; // -> 0b00000000_00100000_11000000
// binary AND, which has the effect of keeping only the 8
// first bits
int z = y & 0b11111111; // -> 0b11000000
// We have recovered our green component at 192
// we could also have written: int z = y & 0xff;
```

8.2 Signed and unsigned integers

In memory, an integer is simply a sequence of bits. Its actual value depends entirely on how the bits that make it up are interpreted. There are several ways of interpreting a sequence of bits, each with its own benefits (2's complement (Le complément à deux), 1's complement, "sign and magnitude" and "offset binary").

In Java, integers are all encoded as *2's complement*. This means that integers in Java can be negative, and that adding 1 to the largest possible value gives the smallest possible value, and subtracting 1 from the smallest possible value gives the largest possible value. Unlike other programming languages, such as C, it is not possible to disable this interpretation, which can cause certain difficulties.

Here's a summary of the important points to remember when coding the project for integer representation:

- ARGB format channels take as values unsigned integers between 0 and 255 (which requires one byte).
- The `byte` integer type is encoded on one byte, but its values are always considered signed in Java and are represented in two's complement.
- In a `byte`, the most significant bit (the one on the far left) indicates the sign. When you convert a number larger than the maximum limit, parts of it will be truncated. An `int` corresponds to 32 bits. Imagine you have an `int` worth 255 (11111111). Add 1 and you get 256 (100000000). If you convert it to `byte`, you'll lose all the bits beyond the 8th from the right, so you'll get 00000000, which is 0 in `byte`.
- In 2's complement, addition and subtraction on binary numbers are performed in modular arithmetic ("wrap around"), i.e., they are performed modulo the maximum representable number and there is no overflow. A trick for obtaining modular arithmetic values of additions and subtractions between binary numbers between 0 and 255 is to convert them to `byte`:

```
byte b = (byte)255;
System.out.println(b); // - (2^7 - (255-128)) = -1
b = (byte)248;
System.out.println(b); // - (2^7 - (248-128)) = -8

// "wrap around"
b = (byte)(255 + 1);
System.out.println(b);  // 0
b = (byte)-255;
System.out.println(b);  // +1
```

```
byte b1 = 127;
byte b2 = -128;
b = (byte)(b2-b1);
System.out.println(b);  // +1
```

- The Java method `int Byte.toUnsignedInt(byte x)` gives the value in unsigned representation of the `byte x`:

```
byte b = -1;
System.out.println(Byte.toUnsignedInt(b); // 255

b = 127;
System.out.println(Byte.toUnsignedInt(b); // 127
```

- For the 2's complement representation, you can watch the video "Binary representation of integers" by Olivier Lévêque or the video "Représentation des entiers en binaire" by Ronan Boulic.
- To convert an integer to its unsigned value in Java, you can use a number of simple procedures (for example, this).

8.3 Random number generators

On several occasions in this project, it is necessary to use a random number generator. This is typically the case for methods such as "One-time pad" and "CBC". Such a generator can simply be created in Java as follows:

```
import java.util.Random;

Random r = new Random();
```

To use a generator declared in this way, simply invoke the `r.nextInt(int i)` method, which produces a random number between 0 and `i` (excluded).

The numbers generated are in fact "pseudo-random": the algorithms used by generators use what is known as a "seed". From the same seed, the same sequence of numbers is always obtained (so it's not completely random in reality).

During development, methods using random numbers are difficult to test, as their behavior varies from run to run. To get around this problem, it is possible to ensure that the "random" sequence is always the same by setting the "seed". This can be done in Java by declaring the generator as follows:

```
import java.util.Random;

Random r = new Random(seed);
```

The `seed` is a `long` value. This type models larger integers than `int` (allowing much larger integer values to be modeled).

The value given to `seed` is of no importance in our context. The call for a seed worth 2 is written as follows:

```
Random r = new Random(2l); // the lowercase 'L' after the 2 indicates the
    type long
```

In our case, you don't need any large numbers, you can use an integer of your choice (0, 1, 2 etc.).

References

[1] Trappe, Wade and Washington, Lawrence C. *Introduction to Cryptography with Coding Theory (2nd Edition)*. Prentice-Hall, Inc., 2005

[2] https://www.eiron.net/thesis

[3] https://www.ojp.gov/ncjrs/virtual-library/abstracts/overview-steganography-computer-forensics-examiner