CS-107 : Mini-Project 1
Data compression : the ”Quite Ok Image” format

Hamza Remmal, Barbara Jobstmann, Jamila Sam

-

version 1.7

Contents

This document uses colors and clickable links. It's advised to view it in digital format.

1 Introduction

Whether it is memory or computing speed, a computer's resources are relatively limited. As a future engineer, one of your priorities should be to properly manage these resources.

During this project, you will be introduced into the world of files. This will include knowing how a file that contains an image is structured and how a computer program manages to extract the data it contains. Two image formats will be used for this project, the famous PNG (Portable Network Graphics) format and the QOI (Quite Ok Image) format created on November 24, 2021 by Dominic Szablewski. The latter is a very good example of a compromise between computing speed and memory management. In the next section you will find a short description of the project structure.

Section 8 as well as the slides presented in class contain explanation that it is useful to read before starting the project.

2 The structure and provided code

2.1 Structure

The project is divided into three stages:

- Utility functions: during this first step, we will create a few functions whose role is to make the following steps easier.
- QOI encoder: secondly, we will set up the structure of an encoder for the "Quite Ok Image" format.
- QOI decoder: what is an encoder without its decoder? We are therefore going to finalize the tool by adding the decoder of the "Quite Ok Image" format.

2.2 Code provided

To access the code provided, carefully follow the instructions given in the general description of the project.

The provided archive has the following structure :

```
│
├── references
│   ├── *.png
│   ├── *.qoi
│   └── *.txt
└── src
    └── cs107
        ├── ArrayUtils.java
        ├── Diff.java
        ├── Helper.java
        ├── Hexdump.java
        ├── Main.java
        ├── QOIDecoder.java
        ├── QOIEncoder.java
        ├── QOISpecification.java
        ├── SignatureChecks.java
        └── Submit.java (provided later)
```

You will find here the project's javadoc documentation. Familiarize yourself with the material provided throughout this documentation (you are not expected to understand fully the details of the provided code).

- All mandatory code will need to be written in the files `ArrayUtils.java`, `QOIDecoder.java` and `QOIEncoder.java`.
- The headers of the methods to implement are provided and **should not be changed**.
- The file `Submit.java` will be used to submit your project. It is not provided yet and you will have to store it as shown above. Once provided, **be careful not to corrupt it!**.
- The provided file `SignatureChecks.java` gives the set of signatures which should not be changed. It serves as a control tool during submissions. It will make it possible to use all the methods required **without testing its operation**[a]. Check that this program **compiles** successfully, before submitting.

[a]This verifies that your signatures are correct and that your project will not be rejected for submission.

2.2.1 Advanced code

Some functions necessary for the development of the project, such as those related to the manipulation of files or images, are too advanced for this course. They are therefore coded for you. For this purpose, a utility file `Helper.java` is provided to you containing all the necessary functions. This file contains:

- The definition of a new class `Image` (usable with the name `Helper.Image` from your code) which will encapsulate the information needed to encode/decode an image. You can refer to the javadoc for how to handle objects of type `Helper.Image` and for more details about the functions described below.
- A function to create a new image. The arguments will be explained later.

```
public static Image generateImage(int[][] data, byte channels, byte
    colorSpace)
```

- A function that extracts the content of a PNG image. It takes the relative path to the file as an argument and returns a new image.

```
public static Image readImage(String path)
```

- A function to create a new image in PNG format. It takes as argument the file path and the corresponding image. The image is stored in the path inside the `res/` folder.

```
public static void writeImage(String path, Image image)
```

- A method to be able to read the contents of a file whose access path is given as an argument.

```
public static byte[] read(String path)
```

- A function `Helper::write` that creates a new file, which takes as parameter the file's path and content. Similarly to `Helper::writeImage`, the file is stored inside the `res/` folder.

```
public static void write(String path, byte[] content)
```

- A function that terminates the program using Java's exception system. It takes two parameters: The message format (fmt) and the objects needed to format our message. (Formatting works quite similarly to `String::format`)

```
public static<T> T fail(String fmt, Object ... params)
```

The use of these methods in the appropriate contexts will of course be described in the statement (for the moment it is just a matter of taking note of their existence).

2.3 Tests

Important: It is your responsibility to verify the correct behavior of your program. The provided file `Main.java`, partially written, will be used to test your developments in a simple way. **It is up to you to complete `Main.java` by invoking the methods you implemented in an adequate way and to check the absence of errors in your program**.

When correcting your mini-project, we will use automated tests, which will pass randomly generated input data to the various functions of your program. **There will, therefore, be tests verifying these *special cases*.** Thus, it is important that your program correctly handles any *valid* input data.

With regard to managing error cases, it is necessary to test the input parameters of functions; eg, checking that an array is not null, and/or that it has the correct dimension, etc. These tests generally make debugging easier, and help you reason about the behavior of a function. **We will assume that the function arguments are valid** (except exceptions explicitly mentioned).

To guarantee this hypothesis, we invite you to use the Java assertions[1]. An assertion is written in the form:

```
assert expr;
```

where `expr` is a boolean expression. If the expression is false, then the program throws an error and stops, otherwise it continues normally. For example, to verify that a method's input parameter `key` is not `null`, you can write `assert key != null;` at the beginning of the method body.

Assertions must be enabled to work. This is done by running the program with the "-ea" option: [instructions for IntelliJ] (or [instructions for Eclipse])[Clickable Links].

[1]We will have the opportunity to come back to them in detail, but their use is intuitive enough that we can already use them.

Here is a summary of the main instructions/indications to follow for the coding of the project:

- Method parameters will be considered error-free unless explicitly stated otherwise.
- The headers of the provided methods must remain unchanged: the file `SignatureCheck.java` must therefore not contain **compilation** errors.
- Apart from the given methods, you are free to define any additional method that seems relevant to you. **Modularize and try to produce clean code!**
- It is up to you to verify the correct behavior of your program. Nevertheless, we provide the file `Main.java`, illustrating how to invoke your methods to test them. The examples of tests thus provided are non-exhaustive and you are authorized/encouraged to modify `Main.java` to do more checks. Your efforts in terms of testing will be rewarded.
- Your code must respect the usual naming conventions.
- The project will be coded without the use of external libraries. If you have doubts about the use of such or such library, ask us a question and above all pay attention to the alternatives that IntelliJ (or Eclipse) offers you to import on your machine.
- Your project **should not be stored on a public repository** (like public github). For those of you familiar with git, this may also be useful: https://gitlab.epfl.ch/.
- Use the forum Ed. Before asking your question, check that there are no similar questions. This makes it easier to use the forum and not get lost in the questions. Also respect the structure of the forum: **Use the Mini-Project 1 category**.
- You will find in the file **Main.java** all examples used in the writing of this document. Remember to run them one by one to "verify" the behavior of your code.

3 Debugging your code

As you will see in the next sections, this project manipulates several `byte` arrays (`byte[]`). To help you debug and validate your code, 2 files (`Hexdump.java` and `Diff.java`) are available to you. These last two utilities simulate the use of 2 Unix commands (**hexdump** and **Diff**).

3.1 Hexdump

Utility `Hexdump` has the role of displaying on the terminal all the bytes contained in your table.

To be able to use it, 3 methods have been made available to you.

The first method takes as argument an array of `byte` (`byte[]`) and displays its contents on the terminal. Naturally, its signature is:

```
public static void hexdump(byte[] binary)
```

As you progress through the project, you will start handling large arrays. This makes using the above method complicated. For this reason, the utility `Hexdump` provides 2 overloads of `Hexdump::hexdump`. The first, has the following signature:

```
public static void hexdump(byte[] binary, int start_address)
```

and starts the display from a certain address (index in the table). The second makes it possible to display a part of the table. This is done by specifying the starting address and the ending address. Its signature is as follows:

```
public static void hexdump(byte[] binary, int start_address, int
    end_address)
```

Hints You will use the utility `Hexdump` to inspect the contents of your files as well as your tables. For this, you will need to have understood the structure of your files as well as the values that your file must contain. For example, when parsing a 'QOI' file, one expects to have the magic number present at the beginning of the file. If this is not the case, inspect the code that deals with the header and verify that no error has been made. If you have a reference and you want to compare your file with, you can use the next utility, `Diff` .

3.2 Diff

As its name suggests, this utility allows you to find the difference between the content of 2 files and display it on the terminal. This can be done using a single method (`Diff::diff`) whose signature is:

```
public static void diff(String file_1, String file_2)
```

This method takes as parameters the path to 2 files. These can be of different sizes. If so, a **WARNING** is displayed on the console and the utility compares only the first part of the files. Finally, if the two files have the same content, the method `Diff::diff` displays a **WARNING** followed by the message: "The two inputs have the same content".

3.3 QOI Viewer

To help you view a file in the "Quite Ok Image" format, you can install the following plugin from the Jetbrains Marketplace (here). To do so, follow the instructions below:

1. Open Intellij. (If one of your projects opens, you can close it by pressing *File -> Close Project* at the top of your window)
2. On the left of your window, select *plugins*.
3. In the section *Marketplace*, search for the plugin **QOI Support** and install it.
4. As soon as the installation is complete, restart Intellij. The plugin should be active.

By clicking on a file in format `.qoi`, you will then be able to clearly view the image it encodes.

If you are working with Eclipse, you can use this ”QOI viewer” (just drag your .qoi files into the viewer window).

4 Task 1: Implementation of utility functions: ArrayUtils

As explained in the supplements (section 8.3), the images to be processed are broken down in the channels (R,G,B and A) and are linearized. This facilitates and optimizes the writing of the encoder (step 2) and the decoder (step 3). You are asked to complete the file `ArrayUtils.java` which will provide a number of utility methods to perform this processing. This file already contains the signatures, so all you have to do is complete the body of the methods. You can of course add more methods if you wish, but **do not delete any**.

Throughout the project, you will manipulate bytes using bitwise operators (see the project's presentation slides and section 8.1).

4.1 Methods for testing equality

The purpose of this section is to equip our "toolbox" with methods that will allow us to test the equality between the contents of certain arrays.

4.1.1 Methods `ArrayUtils::equals`

As their name suggests, the purpose of these two methods is to test the equality between the contents of two arrays. They have as signatures:

```
public static boolean equals(byte[] a1, byte[] a2)
```

```
public static boolean equals(byte[][] a1, byte[][] a2)
```

These two methods work in a similar way. If the contents of the two arrays are identical, the value `true` is returned, otherwise the value `false` is returned.

How to handle the case of `null`? If both parameters of your method contain the special reference `null`, the return value must be `true`. We adopt the semantics that since the two references are identical, the two arrays are equal even though they have no content. On the other hand, if only one of the parameters is `null`, the program should terminate using the assertion mechanism previously described.

Your implementation of `ArrayUtils::equals` must be done using the basic structures of Java. You therefore **do not have the right to use predefined methods of the Java API**.

4.1.2 Other methods...?

The methods proposed above allow only to perform the equality test between two arrays of `byte` or `byte[]`. If later you find it necessary, you can override the definition of `ArrayUtils::equals` keeping the same name for the method, but changing the type in the method signature.

4.2 Managing integers

The goal of the next three methods is to allow converting integers into an array and vice versa.

4.2.1 Method `ArrayUtils::wrap`

The first method is to "wrap" a `byte` in a table. The expected behavior is illustrated by the provided example.

```
// ArrayUtils::wrap's signature
public static byte[] wrap(byte value)
// Example
byte a = 1;
byte[] wrapped_a = ArrayUtils.wrap(a); // wrapped_a = [1]
```

4.2.2 Methods `ArrayUtils::toInt` and`ArrayUtils::fromInt`

It will be useful later on to be able to encode the data either in the form of a table of `byte` or in the more concise form of an integer (`int`). As seen in the course, the type `int` is 4 times larger than type `byte`. To pass from one to the other, we will have to adopt a convention on the order of the bytes forming the integer.

These conventions are known as "endianness". For this project, the format for handling integers will be **big endian** (most significant byte first, see examples given below).

You are required to code the two methods below for handling integers. The first one (`ArrayUtis::toInt`) takes as parameter an array of `byte` and returns the corresponding integer. The second (`ArrayUtils::fromInt`) takes an integer as a parameter and returns `byte`.

You will code these methods using **bitwise operators** (see the project presentation slides and sections 8.1 et 8.4). The examples provided illustrate the expected behaviors. **`ArrayUtils::toInt`** :

```
// ArrayUtils::toInt's signature
public static int toInt(byte[] bytes)
// Example
byte[] array = {(byte) 0x7B, (byte) 8, (byte) 4, (byte) 7};
      // same as : byte[] array = {123, 8, 4, 7};
int value = ArrayUtils.toInt(array); // value = 0x7B_08_04_07 / 2064122887
```

`ArrayUtils::fromInt` :

```
// ArrayUtils::fromInt's signature
public static byte[] fromInt(int value)
// Example
int value = 0xBC614E; // 12345678
byte[] array = ArrayUtils.fromInt(value); // array = [0x00, 0xBC, 0x61,
   0x4E]
                                          // array = [0, -68, 97, 78]
```

For the method `ArrayUtils::toInt` you need to verify, using assertions, that the following conditions hold:

- the table `bytes` is not equal to `null` ;
- and the size of the table is 4.

4.3 Concatenation methods

Now that we can test array equality and manipulate integers, we will be able to implement useful methods for managing arrays. One of these operations is **concatenation**.

4.3.1 Method `ArrayUtils::concat`

You are first asked to code, using the principle of *overloading*, two versions of a method `concat` one of which takes as argument a number of arrays of `byte` and the other a number of `byte` numbers. The expected behaviors are illustrated by the examples provided.

```
// ArrayUtils::concat's signature
public static byte[] concat(byte[] ... tabs)
// Example
byte[] tab1 = new byte[]{1, 2, 3};
byte[] tab2 = new byte[0];
byte[] tab3 = new byte[]{4};
// tab = [1, 2, 3, 4]
byte[] tab = ArrayUtils.concat(tab1, tab2, tab3);
```

```
// ArrayUtils::concat's signature
public static byte[] concat(byte ... bytes)
// Example
byte[] tab = ArrayUtils.concat((byte)1, (byte)2, (byte)3, (byte)4);
// tab = [1, 2, 3, 4]
```

These methods `ArrayUtils::concat` use the notion of *ellipsis*, which allows us to concatenate as much data as we want and makes it easier to use.

For `ArrayUtils::concat` you need to verify, using assertions, that none of the objects `tabs`, `tabs[i]` and `bytes` are equal to `null`

4.3.2 Method `ArrayUtils::extract`

Secondly, we are interested in extracting one array from another. For this, you are asked to implement the method `ArrayUtils::extract` which takes as parameter an array, the position at which the extraction will start and the size of the array to extract. The expected general behavior is illustrated by the provided example.

```
// ArrayUtils::extract's signature
public static byte[] extract(byte[] input, int start, int length)
// Example
byte[] tab = new byte[]{1, 2, 3, 4, 5, 6, 7, 8};
// extracted = [3, 4, 5, 6, 7]
byte[] extracted = ArrayUtils.extract(tab, 2, 5);
```

For `ArrayUtils::extract` you need to verify, using assertions:

- that `input` is not equal to `null` ;
- that `start` is a valid index value for `input` ;
- that `length` is positive or zero;
- and that `start+length` is less than or equal to `input.length`.

4.3.3 Method `ArrayUtils::partition`

The project will need to be able to partition arrays into sub-arrays of various sizes. You are asked for this to code the method `ArrayUtils::partition` which takes as parameter the array to be partitioned and the respective sizes of each partition in the final array. The example provided illustrates the general expected behavior.

```java
// ArrayUtils::partition's signature
public static byte[][] partition(byte[] input, int ... sizes)
// Example
byte[] tab = new byte[]{1, 2, 3, 4, 5, 6, 7, 8, 9};
// partitions = [[1, 2, 3], [4], [5, 6], [7], [8, 9]]
byte[][] partitions = ArrayUtils.partition(tab, 3, 1 ,2, 1, 2);
```

For `ArrayUtils::partition` you need to verify, using assertions:

- that `input` and `sizes` are not equal to `null`;
- and that the sum of the integers of `sizes` are equal to the length of `input`.

4.4 Formatting Methods

In the next stage of the project (`QOIEncoder`), you will have to process the pixels of an image one by one. To make your next tasks easier, you will have to implement two methods that will allow you to change the structure of an array, while preserving the information it contains.

The first method will have the role of separating the different color channels of an image (to be able to process the R,G,B and A components of its pixels separately). The second will make it possible to do the opposite operation: namely to restore the usual format of an image in the form of a two-dimensional array of pixels, starting from the representation produced by the first method.

4.4.1 Method `ArrayUtils::imageToChannels`

Consider an image encoded as a two-dimensional array `int[][] input`. Each of the integers encodes a *pixel* which is made up of **4 bytes**. These represent the intensity value of each R,G,B and A channel of a given pixel. The method `ArrayUtils::imageToChannels` that you are asked to code here aims to create a new two-dimensional array, of type `byte[][]` so that each column of this array is an *one-dimensional* array. Each of these one-dimensional arrays will contain the set of bytes of a given channel (for example, the set of bytes of the channel R of all the pixels of the array `input`). Each column represents a channel.

Let the input image be:

```
input = { {p[0][0], ....., 	p[0][w-1]},
          {p[1][0], ....., 	p[1][w-1]},
          ....
          {p[h-1][0], ....., 	p[h-1][w-1]}
        }
// p[x][y] is the int value of a pixel
```

The table created will be:

```
        { {r[0][0],      g[0][0],     b[0][0],      a[0][0]},
          {r[0][1],      g[0][1],     b[0][1],      a[0][1]},
         ...
          {r[0][w-1],    g[0][w-1],   b[0][w-1],    a[0][w-1]}
          {r[1][0],      g[1][0],     b[1][0],      a[1][0]}
          ....
          {r[1][w-1],    g[1][w-1],   b[1][w-1],    a[1][w-1]}
          ...
          {r[h-1][0],    g[h-1][0],   b[h-1][0],    a[h-1][0]}}
          ....
          {r[h-1][w-1], g[h-1][w-1],b[h-1][w-1], a[h-1][w-1]}
       };
//r[x][y] is the value of p[x][y]'s R component
//g[x][y] is the value of p[x][y]'s G component
//b[x][y] is the value of p[x][y]'s B component
//a[x][y] is the value of p[x][y]'s A component
```

The data in the input array is therefore linearized (we have ”broken” the two-dimensional structure for the data of the pixels themselves). All this boils down to coding a method whose signature is:

```java
  // ArrayUtils::imageToChannels signature
  public static byte[][] imageToChannels(int[][] input)
  // Example
  int[][] input = {{1, 2, 3, 4, 5},
                   {6, 7, 8, 9 ,10},
                   {11, 12, 13, 14, 15}};
  // output = [[0, 0,  1, 0], [0, 0,  2, 0], [0, 0,  3, 0],
  //           [0, 0,  4, 0], [0, 0,  5, 0], [0, 0,  6, 0],
  //           [0, 0,  7, 0], [0, 0,  8, 0], [0, 0,  9, 0],
  //           [0, 0, 10, 0], [0, 0, 11, 0], [0, 0, 12, 0],
  //           [0, 0, 13, 0], [0, 0, 14, 0], [0, 0, 15, 0]]
  byte[][] output = ArrayUtils.imageToChannels(input);
```

You will notice that the file provided `QOISpecification.java` provides useful constants `r`, `g`, `b` and `a`. **It will be necessary to respect the order in which you store the columns, as suggested by the examples provided**.

For `ArrayUtils::imageToChannels` you need to check the following conditions:

- that `input` and `input[i]` are not equal to `null`;
- that all the lines of `input` are the same size.

4.4.2 Method `ArrayUtils::channelsToImage`

In a complementary manner, a method will have to be implemented which has the role of restoring the 2-dimensional structure of an image whose R,G,B and A channels have been decomposed. It will take an array as parameter `byte[][]` (having the same structure as the array returned by`ArrayUtils::image_to_channels`), but also the length and width of the array to create.

Unlike other operations, the method `ArrayUtils::channelsToImage` cannot be limited to an array of `bytes` to reconstruct the 2-dimensional structure of the image. These dimensions cannot be inferred based solely on this information. The additional parameters therefore make it possible to specify the dimensions of the array to be constructed.

```
// ArrayUtils::channelsToImage's signature
public static int[][] channelsToImage(byte[][] input, int height, int
    width)
// Example
byte[][] formatted_input = {
        {0, 0,  1, 0}, {0, 0,  2, 0}, {0, 0,  3, 0},
        {0, 0,  4, 0}, {0, 0,  5, 0}, {0, 0,  6, 0},
        {0, 0,  7, 0}, {0, 0,  8, 0}, {0, 0,  9, 0},
        {0, 0, 10, 0}, {0, 0, 11, 0}, {0, 0, 12, 0},
        {0, 0, 13, 0}, {0, 0, 14, 0}, {0, 0, 15, 0}};
// output = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]
int[][]  output = ArrayUtils.channelsToImage(formatted_input, 3, 5);
```

For`ArrayUtils::channelsToImage` you need to verify that the following conditions hold:

- `input` and `input[i]` are not equal to `null`;
- than the size of `input[i]` is 4;
- and that the size of `input` is equal to `height*width`;

4.5 Tests

To test the behavior of your utilities, use the program `Main.java`. You can invoke any methods useful for testing. We provide some sample methods that you can use to learn how to implement more testing functions. These examples are not exhaustive. **Complete them** according to what seems necessary for you so that you test all the methods implemented in this first part of the project.

5 Task 2: An encoder for the "Quite Ok Image" format

Now that we are well equipped with the utilities of `ArrayUtils`, we can get into the nitty-gritty and start implementing the *encoder*.

From an image (of type `Helper.Image`), the goal is to produce a compressed version, according to the "Quite Ok Image" ("QOI") format. According to this format, the result will be an array of `byte`s consisting of:

- of a set of `byte`s representing a *header* ;
- of a set of `byte`s encoding the compressed data ;
- and a set of `byte`s representing a *signature* and which mark the end of a "Quite Ok Image" file.

First, we'll start by creating the header. We will then concern ourselves with the encoding of the data that constitute the image. We will end by adding the *signature* and combining it all.

It is very important to test your code method by method as you progress. Refer to sections 3 and 5.6 for essential information on this subject. **In particular, take note of the various small data files made available to you to facilitate debugging** (as described in 5.6).

5.1 Structure of a "Quite Ok Image" file

First of all, here are some details on the three parts that make up a file in the format **Quit Ok Image**. The first part, the *header* of the file, contains what are called **metadata**. This is additional information about the data contained in the file. The second part includes the data relating to the image, i.e. **the color of each pixel**. Finally, the file ends with a series of bytes that correspond to an end-of-file marker (**EOF**, "End-Of-File").

Header
Image content
EOF (End-Of-File)

5.2 File Header

Most file formats or protocols use a *header*. This part contains additional information such as the size of the image or the compression algorithm used.

For the format **Quit Ok Image**, the header consists of 5 components.

Magic number	4 bytes
Image width	4 bytes
Image height	4 bytes
Number of channels	1 byte
Color space	1 byte

In digital imaging, a color space is an abstract notion that allows the construction of a color palette. This is made from a 3-component base and each color is thus associated with a coordinate in this base. Understanding this concept is not essential for the smooth running of the project. If you are curious for more, this article describes the concept in more detail.

5.2.1 Magic Number

The *magic number* is a sequence of `byte`s which identify a protocol or file format. Many formats familiar to you use such a number. Here are some examples :

Formats or Protocols	Magic Number
JVM (*.class)	0xCAFEBABE
PNG	0x89504E470D0A1A0A
ZIP	'P' 'K'

For **Quite Ok Image**, the magic number consists of 4 bytes.

Formats or Protocols	Magic Number
Quite Ok Image	'q' 'o' 'i' 'f'

5.2.2 Method `QOIEncoder::qoiHeader`

To be able to create the structure of the file header, you are asked to code the method `QOIEncoder::qoiHeader`. This method must take an image and extract the information necessary to form the header for the version **QOI** of the image.

```
// QOIEncoder::qoiHeader's signature
public static byte[] qoiHeader(Image image)
// Example
Image image = Helper.generate_image(new int[32][64], QOISpecification.RGB,
    QOISpecification.sRGB);
// header = [113, 111, 105, 102, 0, 0, 0, 64, 0, 0, 0, 32, 3, 0]
byte[] header = QOIEncoder.qoiHeader(image);
```

The conditions to check for`QOIEncoder::qoiHeader` are:

- that **image** is not `null` ;
- that the number of channels encoding the image does not differ from the values of the constants `QOISpecification.RGB` and `QOISpecification.RGBA` ;
- that the value encoding the color space does not differ from the values `QOISpecification.sRGB` and `QOISpecification.ALL`.

The documentation of `Helper.Image` is at your disposal (see javadoc). Functions associated with data of this type are used with dot notation (the same as what you do for functions of type`String`). For example, if `image` is of type `Helper.Image`:

- `image.data()` : returns the array of pixels associated with the image (`int[][]`) ;
- `image.channels()` : returns the number of channels associated with the image ;
- `image.color_space()` : returns the integer corresponding to the color space.

Hints :
Make sure to use the utility methods already coded. You have available in the file `QOISpecification.java` the constants `RGBA` and `RGB` for the values of the number of channels as well as the constants `sRGB` and `ALL` for values corresponding to color spaces.

5.3 "Quite Ok Image" Compression Algorithm

Now we can move on to the compression algorithm **Quite Ok Image**. The latter is based on the resemblance between pixels. To benefit from a high compression rate, the designer of the format (Dominic Szablewski) uses **six** different types of encodings, each of them brings a benefit to the overall algorithm. In the next sections, we will give a brief description of each of these 6 types and how we will handle them in this project.

5.3.1 Encoding *QOI_PO_RGB*

This encoding is the simplest. It has the structure:

QOI_OP_RGB										
byte[0]								byte[1]	byte[2]	byte[3]
7	6	5	4	3	2	1	0	7 .. 0	7 .. 0	7 ..0
QOI_OP_RGB TAG								RED	GREEN	BLUE

To encode a pixel in the format *QOI_PO_RGB*, you will have to implement the method `QOIEncoder::qoiOpRGB`, whose signature is as follows:

```
// QOIEncoder::qoiOpRGB's signature
public static byte[] qoiOpRGB(byte[] pixel)
// Example
byte[] pixel = {100, 0, 55, 0};  // order is R,G,B,A
// encoding = [-2, 100, 0, 55]
byte[] encoding = QOIEncoder.qoiOpRGB(pixel);
```

The latter takes a pixel as a parameter and returns the corrsponding encoding *QOI_PO_RGB*.

For `QOIEncoder::qoiOpRGB` you need to ensure that `pixel` is of size 4.

Make sure to use the already described utility functions. The constant `QOI_OP_RGB_TAG` which corresponds to the "tag" specific to this type of blocks (`0b11_11_11_10`), is provided in the file `QOISpecification.java`.

5.3.2 Encoding *QOI_PO_RGBA*

Quite similar to encoding *QOI_PO_RGB*, encoding *QOI_PO_RGBA* additionally contains the alpha channel. The memory structure is as follows:

QOI_OP_RGBA											
byte[0]								byte[1]	byte[2]	byte[3]	byte[4]
7	6	5	4	3	2	1	0	7 .. 0	7 .. 0	7 ..0	7 .. 0
QOI_OP_RGBA TAG								RED	GREEN	BLUE	ALPHA

Similar to `QOIEncoder::qoiOpRGB` and adding the necessary modifications, you will have to implement the method `QOIEncoder::qoiOpRGBA` whose signature is:

```
// QOIEncoder::qoiOpRGBA's signature
public static byte[] qoiOpRGBA(byte[] pixel)
// Example
byte[] pixel = {100, 0, 55, 73}; // order R,G,B,A
// encoding = [-1, 100, 0, 55, 73]
byte[] encoding = QOIEncoder.qoiOpRGBA(pixel);
```

For `QOIEncoder::qoiOpRGBA` you need to ensure that `pixel` is of size 4.

5.3.3 Encoding *QOI_PO_INDEX*

As you will see in the next section, the compression algorithm uses a **hash table** (see also sections 8.2 and 8.3 in the appendix). To be able to save space in memory, the encoding *QOI_OP_INDEX* allows to encode an index used in the hash table in the following form :

QOI_OP_INDEX							
byte[0]							
7	6	5	4	3	2	1	0
QOI_OP_INDEX_TAG		INDEX					

By construction (according to the algorithm), the encoded index cannot exceed the value 64 (not included). The constant `QOI_OP_INDEX` is the specific "tag" for this type of blocs (`0b00_00_00_00`).

Concretely, you will code the method `QOIEncoder::qoiOpIndex` whose signature is:

```
// QOIEncoder::qoiOpIndex's signature
public static byte[] qoiOpIndex(byte index)
// Example
byte index = 43;
// encoding = [43]
byte[] encoding = QOIEncoder.qoiOpIndex(index);
```

This method takes as a parameter an index (key) in a hash table (i.e. `INDEX` in the schema above) and returns a `byte` (wrapped in an array) and calculated as explained by the diagram and the example.

For `QOIEncoder::qoiOpIndex` you need to ensure that the index is valid for the algorithm (non-negative and worth at most 63).

5.3.4 Encoding *QOI_OP_DIFF*

In most common images, adjacent pixels very often look alike. A pixel may even differ from the one preceding it only by a few small color units (a little more red or a little less blue, for example). For these specific cases, the "QOI" format uses the *QOI_OP_DIFF* encoding. The latter encodes the variation of the values of each channel between two consecutive pixels. At the time of decoding, we will therefore only need to know the value of the previous pixel and the variations to be able to find the next pixel.

QOI_OP_DIFF							
byte[0]							
7	6	5	4	3	2	1	0
QOI_OP_DIFF_TAG		dr		dg		db	

The constant `QOI_OP_DIFF` which corresponds to the "tag" specific to this type of block is equal to `0b01_00_00_00`. To form this encoding, just pass as parameters to the method `QOIEncoder::qoiOpDiff`, the variations of each channel.

The notation *dr* designates the variation of the "red" channel, *dg* that of the "green" channel and *db* that of the "blue" channel. The variations above are calculated using the following formula:

```
d#' = current_pixel[#] - previous_pixel[#]
d# = d#'+ 2
```
(see explanation below)

The signature of the method to be coded is:

```java
/* QOIEncoder::qoiOpDiff's signature, diff = {dr',dg',db'} */
public static byte[] qoiOpDiff(byte[] diff)
// Example
byte[] diff = {-2, -1, 0};
// encoding = [70]
byte[] encoding = QOIEncoder.qoiOpDiff(diff);
```

To be encoded using `QOIEncoder::qoiOpDiff`, the variations in the channels will have to respect certain constraints.
By construction, the variations of each channel cannot exceed a certain limit. This limit mainly stems from the size of *d#'* in the final encoding. For **QOI_OP_DIFF**, each variation is stored on 2 bits; which finally makes it possible to store up to 4 different values. The format therefore defined as a constraint:

$$-3 < d\#' < 2 \Rightarrow d\#' \in \{-2, -1, 0, 1\}$$

The `qoiOpDiff` method will ensure that the values of the variations passed as parameters are stored with an offset of 2 in the constructed block (ie it adds 2 to each of them). Adding these offsets is a simple simplifying trick (this is to ensure that the manipulated numbers remain positive, which simplifies their manipulation). Operations on difference values are done in **modular arithmetic** (see section 8.4). It is therefore necessary to convert the result of the calculations (addition of the offset) into `byte`.

The presumed correct assumptions (to be verified by assertions) for `QOIEncoder::qoiOpDiff` are :

- that `diff` is not `null` ;
- that `diff` contains 3 bytes (the variation for each of the R,G and B channels) ;
- that the content of `diff` respects the constraints provided for the format `QOIEncoder::qoiOpDiff`; ie the variations of each channel must respect the limit defined above.

5.3.5 Encoding *QOI_OP_LUMA*

As in the previous section, the *QOI_OP_LUMA* encoding is based on the similarity of two adjacent pixels. The only difference between the two encodings appears at the level of the constraints applied to the variations.

<table>
<tr><td colspan="16">QOI_OP_LUMA</td></tr>
<tr><td colspan="8">byte[0]</td><td colspan="8">byte[1]</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="2">QOI_OP_LUMA_TAG</td><td colspan="6">dg</td><td colspan="4">dr - dg</td><td colspan="4">db - dg</td></tr>
</table>

The constant `QOI_OP_LUMA` which corresponds to the ”tag” specific to this type of block is worth `0b10_00_00_00`. Similar to `QOIEncoder::qoiOpDiff`, the method shall take as parameter the variation in each of the channels and has as signature:

```java
// QOIEncoder::qoiOpLuma's signature: diff={dr',dg', db'}
public static byte[] qoiOpLuma(byte[] diff)
// Example
byte[] diff = {19, 27, 20};
// encoding = [-69, 1]
byte[] encoding = QOIEncoder.qoiOpLuma(diff);
```

To be encoded using `QOIEncoder::qoiOpLuma`, the variations in the channels will have to respect certain constraints.
The latter are necessary because of the size of each variation in the final encoding.

$$-33 < dg' < 32$$
$$-9 < dr' - dg' < 8$$
$$-9 < db' - dg' < 8$$

Finally, the value of *dg’* should be stored with an offset of 32. The values of *dr’* - *dg’* and *db’* - *dg’*, on the other hand, should be stored with an offset of 8.

The presumed correct assumptions (to be verified by assertions) for `QOIEncoder::qoiOpLuma` are:

- that `diff` is not `null`;
- that `diff` contains 3 `bytes`;
- and that the content of `diff` respects the constraints of the format `QOIEncoder::qoiOpLuma`; i.e. the conditions mentioned above.

5.3.6 Encoding *QOI_OP_RUN*

Finally, the *QOI_OP_RUN* encoding encodes the number of successive repetitions of a pixel in an image. This last type of block is also subject to certain constraints.

QOI_OP_RUN							
byte[0]							
7	6	5	4	3	2	1	0
QOI_OP_RUN_TAG		count					

The "tag" `QOI_OP_RUN` is `0b11_00_00_00`.

For the encoder to save space, the number of repetitions must not be equal to zero (repeating 0 times a pixel does not bring any benefit). Similar to *QOI_OP_DIFF* and *QOI_OP_LUMA*, the repeat count should be stored with an offset of -1 performed by the `qoiOpRun` method. However, a problem may arise. Indeed, if the value of *count* is equal to 64 or 63 (before applying the offset), the encoding of *QOI_OP_RUN* then corresponds to the value of certain "tags" (specifically to *QOI_OP_RGB_TAG* and *QOI_OP_RGBA_TAG*). This should be avoided to avoid ambiguities that complicate decoding.

For this, these values are considered forbidden and the value of `count` will be worth at most 62.

Thus, if for example 113 consecutive pixels are equal, the encoding algorithm will use two *QOI_OP_RUN* blocks, one with a `count` equal to `62` and the other with a `count` equal to `50` (the first pixel of the sequence will be encoded with another block than *QOI_OP_RUN*).

The method `QOIEncoder::qoiOpRun` will take as parameter the number of repetitions of a pixel and will have as signature:

```
// QOIEncoder::qoiOpRun's signature
public static byte[] qoiOpRun(byte count)
// Example
byte count = 41;
// encoding = [-24]
byte[] encoding = QOIEncoder.qoiOpRun(count);
```

It can be assumed for `QOIEncoder::qoiOpRun` that `count` is between 1 and 62 (inclusive).

5.4 Global encoding of the image

Now that we have the 6 useful compression methods, it is necessary to make them cooperate in an adequate way to be able to exploit their potentials to the maximum. For this, we offer you a simple and basic description of the algorithm. You are free to implement it as you see fit.

The "Quite Ok Image" algorithm:

In this algorithm, you will have to process your images in a linear way, ie the pixels will be processed consecutively and each pixel will be processed only once.

Step 1 (Initialization) :

Before starting to process your image, you will have to define the execution environment of your algorithm, that is to set up a set of required *local variables*.

The first of them is used to memorize the value of the *previous pixel* (`byte[]`), it will need to be initialized using the pixel defined in **`QOISpecification.START_PIXEL`**.

You will also define a *hash table* (`byte[64][4]`). It should be empty at first and is intended to store pixels processed using the predefined hash function (**`QOISpecification::hash`**). This table will be used to form type encodings *QOI_OP_INDEX*.

Finally you will have to define a *counter* (`int`), whose main use is to form the encoding blocks *QOI_OP_RUN*. Other variables can be added to these as you see fit.

Step 2 (Pixel Processing) :

Processing of the pixels can then begin. It is a question of going through the pixels one by one and, along the way, trying to build the different possible blocks in the descending order of priority according to the types of blocks:

- number of consecutive similar pixels (*QOI_OP_RUN*);
- index of the current pixel in the hash table (*QOI_OP_INDEX*);
- difference between 2 pixels (*QOI_OP_DIFF*)
- difference between 2 pixels (*QOI_OP_LUMA*)
- full pixel (*QOI_OP_RGB* or *QOI_OP_RGBA*)

More precisely : For each pixel :

1. **If** pixel == previous, increase counter and check if you need to create a *QOI_OP_RUN* entry (counter reaches the limit 62 or the pixel is the last pixel), then move to the next pixel, **else** check if you need to create a *QOI_OP_RUN*, then continue with Step 2.

2. **If** pixel is in hash table (the pixel stored at its hash position has the same value) , create *QOI_OP_INDEX* entry and move to next pixel, **else** add pixel into hash table, and continue with Step 3.

3. **If** alpha channel is the same and difference between pixel and previous pixel is small[a], create *QOI_OP_DIFF* entry and move to next pixel, **else** continue with Step 4.

4. **If** alpha channel is the same and difference between pixel and previous pixel is similar[b], create (*QOI_OP_LUMA* entry and move to next pixel, **else** continue with Step 5.

5. **If** alpha channel is the same but difference is not small nor similar, create *QOI_OP_RGB* entry and move to next pixel, **else** continue with Step 6.

6. **If** alpha channel is different, create *QOI_OP_RGBA* entry and move to next pixel.

[a] according to the criteria checked by the assertions for this type of blocs
[b] according to the criteria checked by the assertions for this type of blocs

You will therefore have to implement the algorithm in the body of the methodQOIEncoder::encodeData.

```
// QOIEncoder::encodeData's signature
public static byte[] encodeData(byte[][] image)
```

This function takes as a parameter a two-dimensional array whose format is that resulting from the application of ArrayUtils.image_to_channels (decomposition into channels and linearization, review if necessary the section4.4.1).

It returns an array of byte (byte[]) which represents the codification of the pixels (of the content of the image, “data”) in the **Quite Ok Image** format (header and signature not included).

The presumed correct assumptions (to be verified by assertions) for QOIEncoder::encodeData are:

- that picture and image[i] are not equal to null,
- image[i] is size 4.

Warning:

- The differences between pixels must be computed with ”wrap arounds” (see 8.4) ;
- For performance reasons, it is not recommended to use the method concat each time a block is encoded. It is best to log encoded blocks in a ArrayList whose elements will only be concatenated once, at once, at the end of the encoding.

5.5 Creating File Content

As you learned in the section 5.1, a **Quite Ok Image** file is composed of three parts (header, content/data and signature). It is now a question of finalizing the encoder so as to be able to create the complete representation of an image in the **Quit Ok Image** format. You are asked to complete the method QOIEncoder::qoiFile which takes as parameter a java image (of type Helper.Image) and returns an array of bytes (byte[]) in the **Quite Ok Image** format:

```
// QOIEncoder::qoiFile's signature
public static byte[] qoiFile(Helper.Image image)
```

Attention, this method should not create the file in the disk, it is the role of Helper::write (see examples of use in the program Main.java provided). You will notice that the signature is provided through the constant *QOI_EOF*.
The presumed verified hypothesis for QOIEncoder::qoiFile is that image is not equal to null.

5.6 Testing

To test the behavior of your encoder, use the program Main.java. The examples provided therein are not exhaustive. **Complete them** according to what seems judicious to you to test all the methods implemented in this second part of the project.

Note that the code provided also allows you to scrutinize your data and compare it to reference data provided in the directory `references`.

If you have equipped yourself to view the files `.qoi` (see section 3.3), the images in your files `.qoi` should be identical to those of `.png` correspondents.

Visual image comparison is not enough to flush out errors easily. You can then use other utilities.

For example, to verify the method compute the header, you can write in `Main`:

```
Hexdump.hexdump(QOIEncoder.qoiHeader(Helper.read_image("references/beach.png")));
```

Which should display in the console the so-called "dump":

```
==========================================
000000 : 71 6F 69 66 00 00 06 40 00 00 | qoif...@.. |
00000A : 04 B0 04 00 |  ... |
==========================================
```

This is the value (in hexadecimal base) of the `byte`s as calculated by your method `qoiHeader` on the image `res/beach.png`. If you open the latter, you will see that it is 1600x1200 in size. The first 4 `bytes` `71 6F 69 66` correspond to the unicode encoding of the letters q, o, i and f, the 4 bytes `00 00 06 40` correspond to the value 1600, the 4 `byte`s next to the value 1200, the penultimate `byte` is 4 (4 channels) and the last `byte` is 0 (color space code). (Use converters such as this one or this one to verify).

A useful little trick for debugging is to use the facility offered by IntelliJ to display the values in hexadecimal or binary format: in the debugger, right-click on an integer value then `View as > Hex` or `View as > binary`.

To make your task easier, we also provide you with very basic little images (`qoi_op_*.png/.qoi`) which will be encoded rather by specific types of blocks (the filenames indicate which types of blocks it is). You also have a small image `qoi_encode_test.png` whose encoding is done by using once each of the 6 possible blocks (`QOI_OP_RGBA, QOI_OP_RGB, QOI_OP_DIFF, QOI_OP_LUMA, QOI_OP_INDEX` and `QOI_OP_RUN`).

The folder `reference` contains the files `.qoi` which you are supposed to get for these files as well as those for larger images. This folder also contains the reference "dumps" supposed to be obtained for the smallest files (given in files `.._dump.txt`). You can compare them visually to detect anomalies. For example, you can run:

```
Hexdump.hexdump(QOIEncoder.qoiFile(Helper.read_image("references/qoi_op_run.png")));
```

and visually compare what you get with the contents of the file `references/qoi_op_run_dump.txt`. Note that the information in the reference dump, stripped of the header and signature parts, can help you debug the intermediate encoding methods.

You can also "purge" the signature and the header of an encoded data by proceeding as follows :

```
var encoding = QOIEncoder.qoiFile(image);
Hexdump.hexdump(encoding, QOISpecification.HEADER_SIZE, encoding.length -
   QOISpecification.QOI_EOF.length);
```

You can finally, using the `diff` utility, compare your result (produced in the `res/` folder) with the expected one provided in the `references/ folder`. For instance:

```
Diff.diff("references/qoi_op_rgba.qoi", "res/qoi_op_rgba.qoi");
```

6 Task 3: A decoder for the "Quite Ok Image" format

As with every protocol or format, knowing how to encode is not enough. The reverse and complementary process of decoding is equally important.

You saw in the previous step different ways to encode the pixels of an image. For each of them, with one exception, you will now have to program its "decoding" counterpart. Clearly, you will have to write a function to decode a header block, a function to decode a QOI_OP_RGB type block, etc.

6.1 Header decoding

The header of a **Quite Ok Image** file contains data that is crucial for decoding. To be able to extract them, you are asked to implement the `QOIDecoder::decodeHeader` method.

```
// QOIDecoder::decodeHeader's signature
public static int[] decodeHeader(byte[] header)
// Example
byte[] header = {'q', 'o', 'i', 'f', 0, 0, 0, 64, 0, 0, 0, 32, 3, 0};
// decoded = [64, 32, 3, 0]
int[] decoded = QOIDecoder.decodeHeader(header);
```

Consider using what you coded in `ArrayUtils`. `HEADER_SIZE` is provided in `QOISpecification.java` and is the expected size for a header block in QOI format.

The presumed correct assumptions (to be verified by assertions) in `QOIDecoder::decodeHeader` are:

- that `header` is not `null`;
- that the block size `header` conforms to the specification (equal to the provided constant `HEADER_SIZE`);
- that the first 4 `byte` are equal to the constant `QOI_MAGIC`;
- that the number of channels is `RGB` or `RGBA` (constants provided);
- that the color space is `ALL` or `sRGB` (constants provided).

6.2 QOI decompression algorithm

The role of the decompression algorithm is to recover the pixels from their encoded representation. Each type of block in the QOI encoding can be decoded using a specific function. It is now these functions that you are asked to program. These are the functions described below.

The "**Tag**" part of each block is what will let you know which function to invoke in the overall algorithm.

6.2.1 Decoding *QOI_OP_RGB*

As with the encoder, the *QOI_OP_RGB* format is the easiest to decode. The function in charge of decoding this type of block is to be coded as follows:

```
//QOIDecoder::decodeQoiOpRGB's signature
public static int decodeQoiOpRGB(byte[][] buffer, byte[] input, byte alpha,
   int position, int idx)
// Example
byte[][] buffer = new byte[2][4];
// buffer = [[0, 0, 0, 0], [0, 0, 0, 0]]
byte[] input = {0, 0, 0, -2, 100, 0, 55, 8, 0, 0, 0};
byte alpha = 34;
int position = 0;
int idx = 4;
// returned_value = 3
int returned_value = QOIDecoder.decodeQoiOpRGB(buffer, input, alpha,
   position, idx);
// buffer = [[100, 0, 55, 34], [0, 0, 0, 0]]
```

You will have noticed that the `QOIDecoder::decodeQoiOpRGB` signature is a bit more complex than the `QOIEncoder::qoiOpRGB` signature. The suggested settings are actually intended to make it easier to integrate this feature into the overall decoding algorithm:

- `buffer` is the decoded image being built;
- `input` is the (complete) data to decode;
- `position` is the position to write decoded pixels to `buffer`;
- `idx` is the position to read data from `input`.

As the alpha channel value is unknown, it must be in parameter (it can thus be specified when calling the function). The return value is the number of `byte`s consumed in `input`.

The presumed correct assumptions (to be verified by assertions) for `QOIDecoder::decodeQoiOpRGB` are:

- than `buffer` and `input` are not equal to the special reference `null`;
- than `position` points to a valid location of `buffer`;
- than `idx` points to a valid location of `input`;
- than `input` contains enough data to recover the pixel.

6.2.2 Decoding *QOI_OP_RGBA*

Similar to `QOIDecoder::decodeQoiOpRGB`, the `QOIDecoder::decodeQoiOpRGBA` method takes care of decoding the format *QOI_OP_RGBA*.

```
//QOIDecoder::decodeQoiOpRGBA's signature
public static int decodeQoiOpRGBA(byte[][] buffer, byte[] input, int
    position, int idx)
// Example
// buffer = [[0, 0, 0, 0], [0, 0, 0, 0]]
byte[][] buffer = new byte[2][4];
byte[] input = {0, 0, 0, -2, 100, 0, 55, 8, 0, 0, 0};
int position = 0;
int idx = 3;
int returned_value = QOIDecoder.decodeQoiOpRGBA(buffer, input, position,
    idx);
// buffer = [[-2, 100, 0, 55], [0, 0, 0, 0]], returned_value = 4
```

The role of this method is very similar to that of `QOIDecoder::decode_qoi_op_rgb`, so we won't describe it.

The presumed correct assumptions (to be verified by assertions) for `QOIDecoder::decodeQoiOpRGBA` are:

- than `buffer` and `input` are not equal to the special reference `null` ;
- than `idx` points to a valid location of `input` ;
- than `position` points to a valid location of `buffer` ;
- than `input` contains enough data to recover the pixel.

6.2.3 Decoding *QOI_OP_INDEX*

No method will be dedicated to decoding *QOI_OP_INDEX*, it will be up to you to manage it properly when implementing the global algorithm.

6.2.4 Decoding *QOI_OP_DIFF*

As described in section 5.3.4, we only need to know the previous pixel as well as the variations in each channel to reconstruct the next pixel. The method responsible for this processing and that you need to code is:

```
// QOIDecoder::decodeQoiOpDiff's signature
public static byte[] decodeQoiOpDiff(byte[] previous_pixel, byte chunk)
// Example
byte[] previous_pixel = {23 , 117, -4, 7};
byte chunk = (byte) 0b01_11_11_11;
var current_pixel = QOIDecoder.decodeQoiOpDiff(previous_pixel, chunk);
// current_pixel = [24, 118, -3, 7]
```

The presumed correct assumptions (to be verified by assertions) for `QOIDecoder::decodeQoiOpDiff` are:

- than `previous_pixel` is not equal to the special reference `null` ;
- than the size of `previous_pixel` is equal to 4;
- that the "tag" of `chunk` has the value *QOI_OP_DIFF_TAG* .

6.2.5 Decoding *QOI_OP_LUMA*

Similar to the previous part, the method `QOIDecoder::decodeQoiOpLuma` will have the role of decoding the part *QOI_OP_LUMA*.

```
public static byte[] decodeQoiOpLuma(byte[] previous_pixel, byte[] data)
// Example
byte[] previous_pixel = {23, 117, -4, 7};
byte[] chunk = {(byte) 0b10_10_01_01, (byte) 0b11_00_11_01};
byte[] current_pixel = QOIDecoder.decodeQoiOpLuma(previous_pixel, chunk);
// current_pixel = [32, 122, 6, 7]
```

It works similar to `QOIDecoder::decodeQoiOpDiff`.

The presumed correct assumptions (to be verified by assertions) for `QOIDecoder::decodeQoiOpLuma` are:

- than `previous_pixel` and `data` differ from the special reference `null` ;
- than the size of `previous_pixel` is equal to 4;
- that the "tag" of `data` is equal to *QOI_OP_LUMA_TAG*.

6.2.6 Decoding *QOI_OP_RUN*

Finally, the `QOIDecoder::decodeQoiOpRun` method decodes the *QOI_OP_RUN* type encoding. It takes as parameter the buffer to fill, the pixel to reproduce, the encoding to decode and finally, the position from which to start writing in the buffer. The returned value corresponds to the number of pixels recovered in buffer, minus 1.

```
public static int decodeQoiOpRun(byte[][] buffer, byte[] pixel, byte
    chunk, int position)
// Example
byte[][] buffer = new byte[6][4];    // Array is full of zeros
byte[] pixel = {1, 2, 3, 4};
byte chunk = -61;
int position = 1;
int returned_value = QOIDecoder.decodeQoiOpRun(buffer, pixel, chunk,
    position);
// buffer = [[0, 0, 0, 0], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1,
    2, 3, 4], [0, 0, 0, 0]]
// returned_value = 3
```

The presumed correct assumptions (to be verified by assertions) for `QOIDecoder::decodeQoiOpRun` are:

- than `buffer` not equal to the special reference `null` ;
- than `position` points to a valid location of `buffer` ;
- than `pixel` not equal to the special reference `null` ;
- than `pixel` consists of 4 channels;
- than `buffer` contains enough space to recover the pixel.

6.3 Global data decoding

The decoding methods of the different types of blocks must now contribute to building the decoded representation of the data. Quite analogous to `QOIEncoder::encode_data`, you are now asked to implement the method:

```
// QOIDecoder::decode_data's signature
public static byte[][] decode_data(byte[] data, int width, int height)
```

The latter will take as a parameter the "data" part of a "Quite Ok Image" file and will proceed with the decoding to form the pixels. Warning, the format of the array returned does not yet correspond to that of an image of size `width x height`. The result is again linearized and broken down according to the 4 channels (R,G,B and A), ie the returned array will have the size: `tab[width * height][4]`. Each `tab[pos]` is a pixel (array of 4 `bytes`).

"Quite Ok Image" decompression algorithm:

This algorithm has the role of decoding the "data" part of a QOI file. It allows to reconstruct the image in the same format as the input of (`QOIEncoder::encode_data`).

Step 1 (Initialization):

Similar to the compression algorithm, you will have to define a set of variables forming the execution environment of the algorithm. These variables are similar to those defined in the compression algorithm (previous pixel, hash table etc.) The starting environment will need to be initialized using the predefined values provided in `QOISpecification` (START_PIXEL, QOI_OP_RGB_TAG, ...). Other variables can be added to your environment if you wish. The hash table is used here to store the value of the decoded pixels.

Step 2 (Data Processing):

The data to be decoded are traversed according to an index `idx` which is incremented such that it is always placed at the start of a new block. In this configuration, the entire `byte` at position `idx` can be a **Tag** (that of blocks of type `QOI_OP_RGB` or `QOI_OP_RGBA`). You must then call the corresponding decoding methods and use them appropriately so that `idx` is positioned at the beginning of the next block.

If the entire `byte` is not a **Tag**, its first 2 bits must be observed to determine which **Tag** it is. Once again, depending on the value of the latter, it is a question of calling the appropriate decoding method by updating each time the progression index in the data to be decoded.

As progress is made, the decoded pixels are stored in the hash table at the position assigned to them according to the hash function. This is what will make it possible to find the pixel value of blocks of type `QOI_OP_INDEX`.

Not all decoding functions are defined, and it is up to you to define the missing code to complete your algorithm. After invoking any decoding method (which results in the decoding of a pixel), you will have to update the useful variables (the hash table, the previous pixel, etc).

The assumptions to check for `QOIDecoder::decode_data` are:

- that `data` differs from the special reference `null` ;
- than `width` and `height` are positive values;
- than `data` contains enough data to decode the base image.

6.4 Creating the java Image

Now that all of our methods have been implemented, we still have to group them to be able to use our decoder. To do this, you will have to implement the `QOIDecoder::decode_qoi_file` method. It takes as parameter the content of the file and returns a new image.

```
// QOIDecoder::decode_qoi_file's signature
public static Image decode_qoi_file(byte[] content)
```

As a reminder, the role of the `Helper::generate_image` method is to create a new image of type `Image`

> The assumptions to check for `QOIDecoder::decode_qoi_file` are:
> - that `content` differs from `null` ;
> - that the end of file signature (`QOISpecification.QOI_EOF`) is not corrupted.

6.5 Tests

To test the behavior of your decoder, use the `Main.java` program. The examples provided therein are not exhaustive. **Complete them** as you see fit to test all the methods implemented in this last part of the project.

7 Extensions

It is allowed to supplement the code with free extensions. If you have an idea for an extension, please submit it to us via a private post on the forum Ed of the course, for validation. Extensions can compensate for points lost on the mandatory parts, but the overall score of the project remains capped at 6.

8 Theoretical supplement

8.1 ARGB representation

The representation ARGB of a pixel using an integer is done in the following way:
The color is broken down into 4 components with a value between 0 and 255:

- **Alpha** is the opacity of the pixel. 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 colors 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 green light intensity value of the pixel.
- **Blue** is the blue light intensity value of the pixel.

Considering that the least significant bit of an integer has index 0 and the most significant bit has 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 16 to 8 the green value and bits 7 to 0 the blue value. For example, if you want to represent a red pixel in RGBA, you need the alpha and red component to be a maximum of 255. Which 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 convenient way to represent a color in Java is to use hexadecimal (more concise) notation. For example to set the red pixel we can write:

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

Bitwise operators make it easy to extract channel values from the integer representation (see example on next page).

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

8.2 Hash table

A hash table is a data structure that allows you to create an associative key-value type structure: the table contains a set of key-value pairs. The notion of key generalizes the notion of index as you know it in an ordinary table. A key can indeed be any object.

To efficiently access the entry corresponding to a given key in the array, a so-called *hash* function is used.

Hash: Hash is the transformation of any object (key) into an integer. This operation is the basis of how hash tables work.

Collision problem: One of the problems with hashing is the ***collision***. This phenomenon occurs when two different objects produce the same integer when hashing. Several solutions exist when implementing a hash table to solve this problem and the programmer is free to choose which one to use according to his needs.

In the project Java offers predefined data structures to implement the hash table concept. This is beyond the scope of this course and you will have the opportunity to explore this material in detail next semester. For this project, we retain the principle of hashing, but applied to a classic array: the index at which to store a piece of data is calculated by a hash function applied to this piece of data. The hash function is provided.

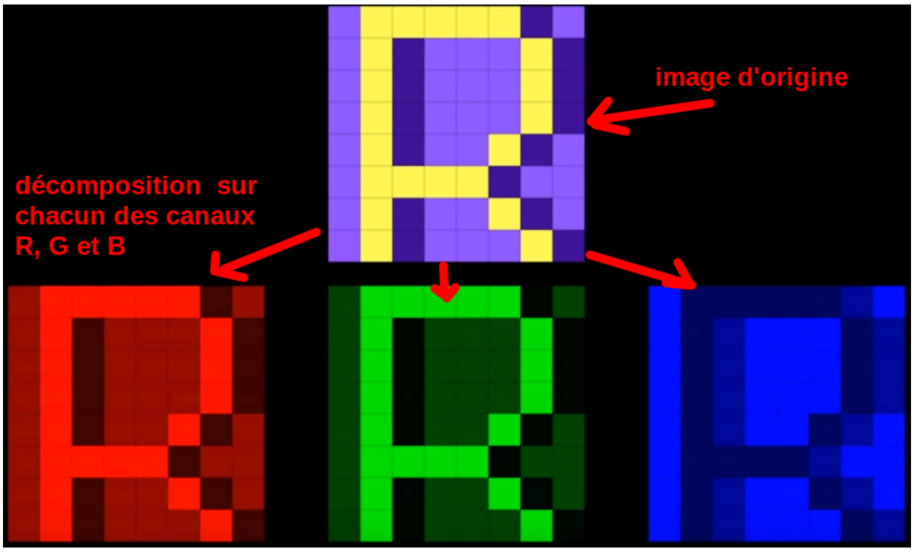
8.3 The “Quite Ok Image” format

An image can be represented as a two-dimensional array of pixels, ie an array of integer values in RGBA format. When the image is large in size, explicitly storing each pixel can quickly become expensive in terms of space.

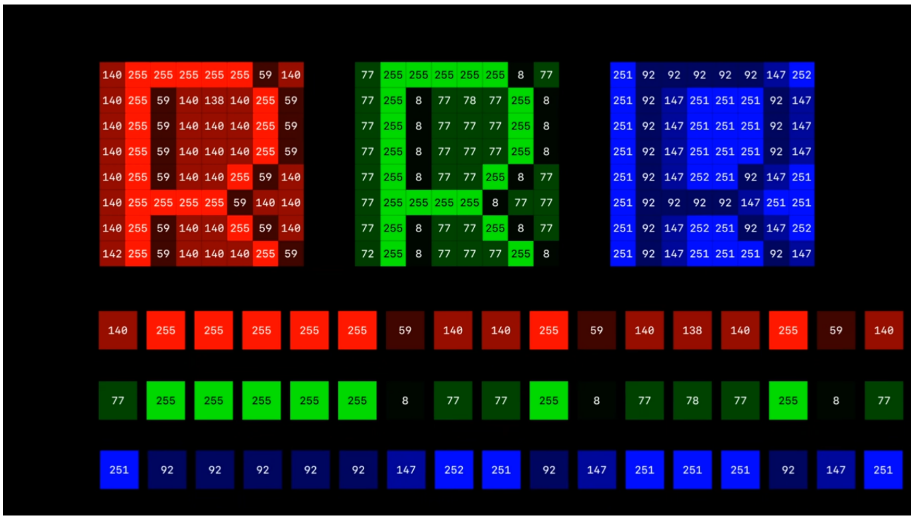
The “Quite Ok Image” format aims to encode the information contained in an image in a more economical way. It is based on the idea that very often in an image, neighboring pixels are identical or very close to each other in terms of color.

Below are some of the fundamental ideas behind the encoding algorithm. The aim here is to give you a general idea, the implementation details will be provided in the description of the tasks to be carried out for the project. Also refer to the slides shown in class.

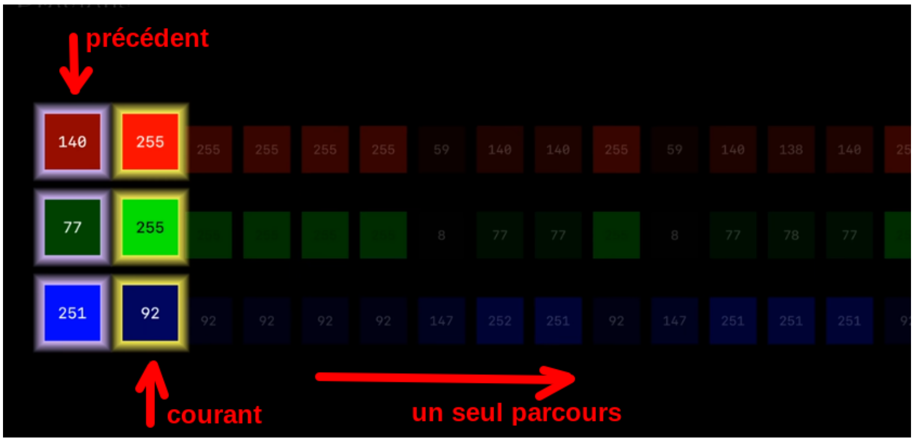
Image decomposition For reasons of optimization and simplification of the data path, each image is first decomposed, pixel by pixel, according to the R, G, B and A channels (here to simplify the decomposition only according to R, G and B is shown):



Each of the two-dimensional arrays corresponding to a channel is then "linearized" so that it can then be traversed as a one-dimensional array (the rows are simply placed end to end):

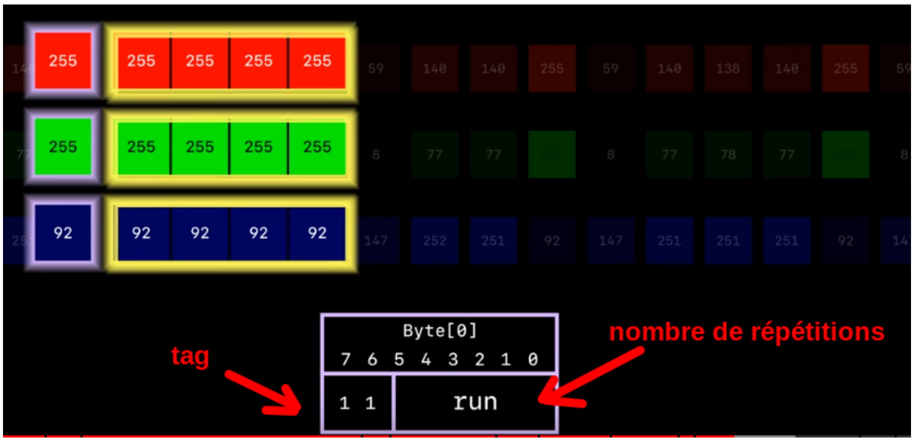


Encoding The encoding algorithm traverses the arrays in a single pass from left to right, memorizing at all times the current index and the previous index



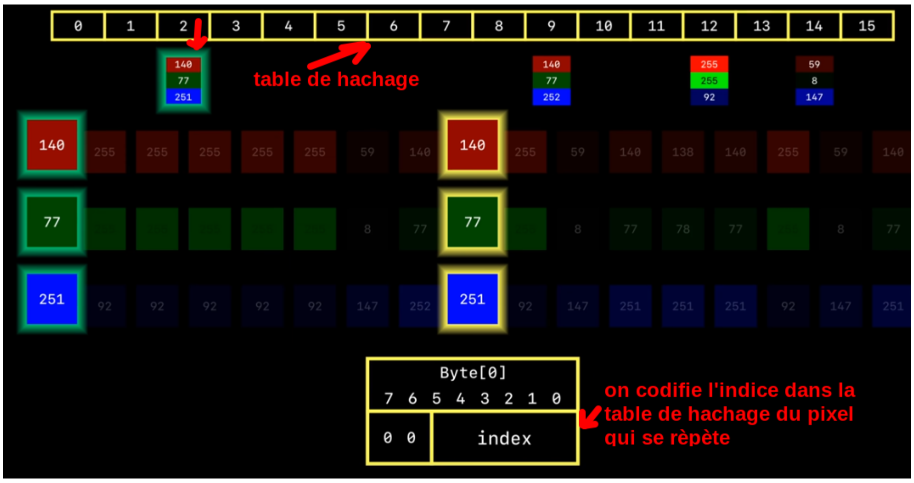
Depending on the value of the pixels at these indices, it is decided to encode the information by means of various types of blocks. There are 6 different types of blocks, described in the QOI format specification.

For example the block `QOI_OP_RUN` is used to code the number of repetitions of an identical pixel:



So instead of storing n times the same pixel we will only store the number n (plus the minimum useful information to be able to decode properly afterwards). Each type of block is indeed identified by a *tag* which is specific to it (which will allow decoding).

During encoding, the last 64 pixels encountered are also stored in a *hash table*. If a pixel stored in the table happens to appear another time in the image, this pixel will not be encoded explicitly, but through a "QOI_OP_INDEX" block which stores the index of the pixel in the hash table; which is much more concise:



The index of the pixel in the hash table is given by a particular hash function and only the pixels which do not cause a collision are encoded by means of a block of type "QOI_OP_INDEX" (the others will be encoded using other types of blocks).

The images used above are taken from this video: https://www.youtube.com/watch?v=EFUYNoFRHQI. You will find there from about minute 23', a complete general presentation of the operating principle of encoding in "Quite Ok Image" format (in English).

The useful explanations for programming the encoding/decoding in QOI format are given in the statement as well as in the course slides.

8.4 Signed and unsigned integers

In memory, an integer is just a sequence of bits. Its actual value depends entirely on how its component bits are interpreted. There are several ways to interpret a sequence of bits, each of them having their own benefits and the most well known being: "2's complement", "1's complement", "sign and magnitude" and "offset binary".

In java, integers are all encoded as *two's complement*. That is to say that the integers in java can be negative and that by adding 1 to the largest representable value we obtain 0 and that by subtracting 1 from the smallest possible value we obtain the largest possible value. Unlike other programming languages, such as C, it is not possible to disable this interpretation which may involve some difficulties.
For this project, this translates into an intrinsic limitation of the compression tool you are going to develop. Indeed, if the image is very large (type `int` insufficient), the header of the "Quite Ok Image" file may include a negative number for information on the size. This limitation is deliberately accepted for simplification.

Here is a summary of important things to remember when coding the project regarding integer representation:

- ARGB format channels take unsigned integer values between 0 and 255 (which requires 1 byte);
- integer type `byte` is single-byte encoded, but its values are still considered signed in Java and they are represented as two's complement;
- in two's complement addition and subtraction on binary numbers are done in modular arithmetic ("wrap around"), i.e. they are done modulo the maximum number that can be represented and that there is no overflow. Some encodings you implement require calculating subtractions between channel values (numbers between 0 and 255) and adding *offsets* to these values: **it is expected that these additions/subtractions are done in modular arithmetic**. A trick to obtain the values in modular

arithmetic of additions and subtractions between binary numbers between 0 and 255 and to convert them into `byte`:

```
byte b = (byte)255;
System.out.println(b);
// - (2^8 - 255) = -1
b = (byte)248;
System.out.println(b);
// - (2^8 -248) = -8
// Modular arithmetic (``wrap around'')
b = (byte)(255 + 1);
System.out.println(b);
// 0b = (byte)-255;
System.out.println(b);
// +1 byte b1 = 127;
byte b2 = -128;
b = (byte)(b2-b1);
System.out.println(b);
// +1
```

For the 2's complement representation, you can watch the video «Représentation binaire des nombres entiers» by Olivier Lévêque (`https://tube.switch.ch/videos/JWKOU3kEA1`) or the video "Représentation des entiers en binaire" by Ronan Boulic (`https://youtu.be/a5gLSc0tbjI`).

In java to convert an integer to its unsigned value you can use different simple processes (see for example this).