CS-107 : Mini-projet 1
Génération de codes QR

L. Vandenberghe, B. Jobstmann, J. Sam

Version 1.2

Table des matières

1 Présentation

Ce document utilise des couleurs et contient des liens cliquables. Il est préférable de le visualiser en format numérique.

FIG. 1: Exemple de QR Code

Le but de ce projet est de créer un **Quick Response Code** (QR code) à partir d'une chaîne de caractères donnée par l'utilisateur. Un QR code est un type particulier de code barre en 2 dimensions (FIG. 1) permettant d'encoder divers types de données tels que des nombres, des lettres, des caractères ASCII etc. Ils peuvent, de ce fait, être utilisés pour coder de façon compacte toutes sortes d'informations utiles comme des liens internet par exemple.

Les QR codes sont omniprésents dans la vie de tous les jours car ils sont utilisables de façon efficace dans de multiples contextes : conditionnement de produits, applications en téléphonie mobile, envois de messages, payements dématérialisés, accès/connexion facilité(e) à des pages internet, à des réseaux sans fils, pour faire de la publicité et bien d'autres choses. Ils sont d'autant plus attractifs qu'ils peuvent aussi contenir des éléments de correction automatique d'erreurs. Un message peut ainsi être décodé même si le QR code qui l'englobe est partiellement endommagé.

Comme montré ci-dessus (FIG. 1), un QR code n'est autre qu'un tableau bi-dimensionnel de petit carrés, généralement noir ou blanc, que l'on appellera « module ». Un module peut-être représenté graphiquement par un simple *pixel* (point dans une image)[1], lequel sera codifié par un code couleur (voir annexe C).

Le plus petit QR code possible est de taille 21x21 modules, le plus grand, est de taille 177x177. Les différentes tailles possibles de QR codes sont référencées au travers de *versions*, allant de 1 à 40. Un QR code de version 1 est de taille 21x21 modules. Chaque version augmente ensuite sa largeur et hauteur de 4 modules par rapport à la version précédente. Par exemple, un QR code de version 2 est de taille 25x25 modules et ainsi de suite. **Ce projet se contentera uniquement**

1. rien n'empêche cependant de représenter graphiquement un module au moyen de plusieurs pixels

d'utiliser les versions 1 à 4. Ci-dessus (Fig. 1), la version utilisée est la version 3 (29x29 modules).

Un QR code est aussi caractérisé par son *niveau de correction*. Plus ce niveau est élevé plus, plus il est probable pour un lecteur automatique de le décoder avec succès, même si certaines de ses parties ne sont pas lisibles. Ces niveaux sont L (Low), M (Medium), Q (Quartile) et H (High). Un niveau de correction L permet à un QR code d'être lu même si 7% des données sont effacées. Ensuite, M permet une correction de 15%, Q de 25% et H de 30%. Il est à noter que le niveau de correction a une contrepartie : plus il est élevé, moins de données peuvent être encodées au sein du QR code. **Dans le cadre de ce projet, nous utiliserons uniquement le niveau de correction L.**

Enfin, le message à encoder dans un QR code est naturellement converti, d'une façon ou d'une autre, dans un format binaire (0/1, vrai/faux). Le message peut ainsi être représenté au moyen de modules à placer dans le QR code : par exemple, un module noir pour « vrai » et un module blanc pour « faux ». La façon de réaliser cette conversion en format binaire dépend du « mode de codage » des caractères employés dans le message. Les trois principaux modes de codages sont : (1) *« numeric mode »* qui permet d'encoder une suite de chiffres de 0 à 9, (2) *« alphanumeric mode »* pouvant encoder les 24 lettres de l'alphabet ainsi que les chiffres de 0 à 9, (3) *« byte mode »* qui code une chaîne de caractères avec sa représentation en octets. Un octet étant un nombre compris entre 0 et 255 représenté sur 8 bits.

Le mode de codage utilisé dans un QR code dépend de son utilisation. Les modes plus limités dans l'expression, comme le mode numérique, permettent une plus grande capacité, c.-à-d. des messages plus longs, que les modes plus évolués comme le « byte mode ». Dans ce projet, **nous allons utiliser le « byte mode » (selon la norme internationale ISO-8859-1)**[2], car il offre un champs d'expression plus complet que l'alphanumérique pour les messages, notamment avec l'ajout d'accents.

Une fois le mode de codage et le niveau de correction fixés, le nombre de données pouvant être encodées dans un QR code dépend de la version. Le moyen de déterminer la capacité d'un QR code sera détaillé plus tard.

La première étape de ce projet consistera à convertir une chaîne de caractères, codée en « byte mode » selon la norme ISO-8859-1 en une séquence de bits équivalente. Ce sujet sera traité dans la Section 3.

Un QR code ne contient pas que le message codé (ce que l'on appellera la « donnée » dans ce qui suit). Certaines de ses parties servent à sa bonne lisibilité. En particulier, certains motifs doivent être placés sur la matrice à des endroits précis (trois motifs occupant des coins). À cela s'ajoute une séquence de bits comportant les informations relatives au format utilisé par le QR code. Ceci fait l'objet de la Section 4.

Enfin, les données du QR code sont placées sur la matrice finale selon un certain algorithme de parcours. Il se peut cependant que des combinaisons de modules ainsi placés rendent difficile la lecture du QR code. Pour éviter cela, il est important d'avoir ensuite recours à une technique dite de « masquage » des modules. Ceci est détaillé dans la Section 5.

Au vu des temps impartis, la description du projet est délibérément allégée et ne détaille pas la construction d'un QR code dans son intégralité. Si vous êtes intéressés à apprendre plus en profondeur le fonctionnement d'un QR code, nous vous conseillons de jeter un coup d'oeil au

2. https ://fr.wikipedia.org/wiki/ISO/CEI_8859-1

tutoriel sur le site https ://www.thonky.com/qr-code-tutorial/introduction.

2 Structure et code fourni

2.1 Structure

Le projet est divisé en trois étapes :

1. **Codage des données** – Conversion d'une chaîne de caractères en une séquence de booléens à placer sur la matrice du QR code.
2. **Construction de la matrice** – Placements des motifs et des bits d'information sur la matrice du QR code.
3. **Placement des données** – Placements de données et masquage des modules.

Vous disposez de deux fichiers à compléter :

- `DataEncoding.java` pour la partie 1 ;
- `MatrixConstruction.java` pour la partie 2 et 3 ;

Les entêtes des méthodes à implémenter sont fournies et **ne doivent pas être modifiées**. La partie 1 : « Codage des données » est indépendante de la partie 2 : « Construction de la matrice », vous pouvez donc faire ces 2 parties dans l'ordre que vous souhaitez. La partie 3 : « Placement des données », doit être complétée une fois les 2 parties précédentes fonctionnelles et correctes.

Le fichier fourni `SignatureChecks.java` donne l'ensemble des signatures à ne pas changer. Il sert notamment d'outil de contrôle lors des soumissions. Il permettra de faire appel à toutes les méthodes requises **sans en tester le fonctionnement**[3]. Vérifiez que ce programme compile bien, avant de soumettre.

Trois dossiers sources, `images`, `test` et `reedsolomon`, sont également fournis. `images` contient des matrices témoins résultant de plusieurs étapes du projet, `test` contient des tests sur les différentes parties que l'on vous demande d'implémenter et `reedsolomon` du matériel pour assurer le niveau de correction du QR code.

2.2 Code fourni

Dans ce projet, chaque QR code possède plusieurs caractéristiques dépendantes de sa version, du masque qui lui est appliqué et de son niveau de correction. Les caractéristiques nécessaires pour ce projet vous seront données dans l'énoncé. Cependant afin de vous simplifier la tâche, nous mettons à disposition un fichier `QRCodeInfos.java` contenant les méthodes retournant les différentes caractéristiques d'un QR code. L'utilisation des méthodes dans ce fichier sera décrite au fur et à mesure de votre progression.

La manipulation de fichiers et de fenêtres étant fastidieuse et trop avancée pour ce cours, un fichier `Helpers.java`, est mis à disposition pour vous aider à visualiser le comportement de vos

3. Cela permet de vérifier que vos signatures sont correctes et que votre projet ne sera pas rejeté à la soumission.

méthodes et d'afficher vos résultats :

- La fonction `show(int[][] matrix, int scale)` permet d'afficher dans une fenêtre la matrice donnée en argument. Celle-ci doit être un tableau de pixels suivant la norme ARGB. (voir Section C). Il est possible d'afficher des pixels de différentes couleurs. L'argument `scale` représente l'agrandissement des modules du QR code. Chaque module sera affiché avec une taille de $scale \times scale$ pixels sur l'écran.
- Les deux fonctions suivantes permettent de lire et écrire des images en format PNG :
  `readMatrix(String filename)`
  `writeMatrix(String filename, int[][] matrix)`
- La fonction `compare(int [][] matrix,String filename)` permet de comparer deux QR codes en affichant leurs différences. Les QR codes comparés sont celui stocké dans `matrix` et celui stocké dans le fichier de nom `filename` du dossier fourni `images/`. La méthode retourne `true` si les deux QR codes sont identiques. L'affichage ne se fait que si les deux matrices ont la même taille. Les modules identiques s'affichent alors en vert, ceux différents en rouge.

2.3 Tests

Important : il vous incombe de **vérifier à chaque étape que vous produisez bel et bien des données correctes** avant de passer à l'étape suivante qui va utiliser ces données. Pour ce qui est de la gestion des cas d'erreurs, il est d'usage de tester les paramètres d'entrée des fonctions ; e.g., vérifier qu'un tableau n'est pas nul, et/ou est de la bonne dimension, etc. Ces tests facilitent généralement le débuggage, et vous aident à raisonner quant au comportement d'une fonction. Par simplification, nous supposerons cependant que les arguments des fonctions sont valides, sauf pour certaines exceptions qui vous seront précisées.

Certains tests sont fournis dans le dossier source `test`. Il peuvent dépendre des images en `.png` placées dans le dossier `images`, veuillez donc prendre garde à ne pas modifier ou effacer les fichiers de ce dossier. Vous êtes libre cependant d'en rajouter. Mais ceux-ci ne couvrent pas l'intégralité des erreurs possibles. Libre à vous de compléter ces tests ou manipuler vos méthodes pour vérifier la présence de bugs dans votre programme.

Voici un résumé des consignes/indications principales à respecter pour le codage du projet :

- Les paramètres des méthodes seront considérés comme exempts d'erreur, sauf mention explicite du contraire.
- Les entêtes des méthodes fournies doivent rester inchangées : le fichier `SignatureCheck.java` ne doit donc pas comporter de fautes de compilation au moment du rendu.
- (suite sur la page suivante ...)

- En dehors des méthodes imposées, libre à vous de définir toute méthode supplémentaire qui vous sembles pertinente. Modularisez et tentez de produire un code propre !
- La vérification du comportement correct de votre programme vous incombe. Néanmoins, nous fournissons des tests à exécuter ainsi que le fichier `Main.java`, vous donnant un exemple de comment appeler vos méthodes en guise de vérification. Les tests fournis sont non exhaustifs et vous êtes autorisé/encouragé à modifier `Main.java` pour faire d'avantage de vérifications.
- Votre code devra respecter les conventions usuelles de nommage.
- Le projet sera codé sans le recours à des librairie externes (sauf bonus éventuels). Si vous avez des doutes sur l'utilisation de telle ou telle librairie posez-nous la question et surtout faites attention aux alternatives que Eclipse vous propose d'importer sur votre machine.
- Votre projet **ne doit pas être stocké sur un dépôt publique** (de type github). Pour ceux d'entre vous familier avec git, nous recommandons l'utilisation de GitLab : https ://gitlab.epfl.ch/.

3 Codage des données

Un QR code est amené à encoder une séquence de caractères alors qu'il ne peut contenir que des modules noirs ou blancs. Il est donc nécessaire de transformer la chaîne de caractères en une séquence binaire équivalente. Le QR code doit également comporter des informations décrivant la taille et le type des données du message pour en permettre le décodage. Enfin, selon les spécifications usuelles, un QR code ne peut pas contenir de région vide. Il est donc généralement nécessaire d'y ajouter des données factices jusqu'à obtenir le remplissage complet.

Le but de cette première partie est donc de suivre ces spécifications pour transformer un message en données pouvant être placées et lues dans un QR code.

Cette partie va nécessiter de nombreuses manipulations sur des bits. Il est donc important d'être à l'aise avoir les opérations binaires en java. Voir Annexe A.

Comme mentionné dans l'introduction, nous allons utiliser le « byte mode » pour l'encodage des caractères. Ceci signifie qu'un caractère du message va être représenté au moyen de 8 bits (un octet ou « byte »). Pour des raisons techniques (expliquées dans l'Annexe B), nous utiliserons cependant le type Java `int` plutôt que le type `byte` pour stocker la valeur de ces octets. Le message encodé est donc une séquence d'octets, stockés dans une séquence d'entiers (`int[]`) compris entre 0 et 255 (intervalle qui est représentable sur 8 bits non-signés).

Afin de convertir un message en une séquence de bits qui peut être utilisée dans un QR Code, vous aurez à réaliser les 5 étapes suivantes :

1. convertir le message en une séquence d'entiers, de taille bornée, en utilisant la norme ISO-8859-1 (Section 3.1) ;
2. ajouter à la séquence ainsi obtenue, les informations qui en permettront le décodage ; à savoir l'indicateur de mode, le nombre de caractères et des bits de terminaison (Section 3.2) ;
3. complèter cette séquence jusqu'à avoir une taille souhaitée (Section 3.3) ;
4. y ajouter des informations de correction d'erreurs ;(Section 3.4)
5. et enfin, la convertir en séquence binaire.(Section 3.5)

3.1 Encodage d'une chaine de caractères

Commencez par implémenter la méthode :

`int[] encodeString(String input,int maxLength)`.

qui prend en argument le message `input` ainsi que le nombre maximal d'octets à encoder. Elle retourne la séquence d'octets représentant le message encodé en utilisant la norme ISO-8859-1. Ces octets seront stockés dans des entiers. La borne `maxLength` est ce qui permettra de tronquer le message en fonction des limites imposées par la version de QR code utilisée.

Il vous faudra commencer par convertir la chaine de caractères du type `String` en une séquence de `byte` selon la norme ISO-8859-1. Ceci peut être fait par l'appel :

```
input.getBytes(StandardCharsets.ISO_8859_1)
```

qui retourne un tableau de type `byte[]`. Notez que cette instruction nécessite l'importation `java.nio.charset.StandardCharsets`.

Dans la norme ISO-8859-1, chaque caractère a une codification binaire comprise entre 0 et 255. Supposons que `input` vaille la chaîne `"é"` (constituée pour l'exemple, d'un caractère unique). Dans la codification ISO-8859-1, le caractère `'é'` a pour code `233` qui vaut `1110_1001` en binaire. L'appel à la méthode `getBytes` va produire un tableau contenant l'unique octet `11101001`. Nous ne souhaitons pas interpréter cet octet dans le format signé supposé par Java (ou le 1 de poids le plus fort est un signe négatif, et où le byte `11101001` représente la valeur `-23`). Nous voulons au contraire préserver la valeur binaire `1110_1001` telle qu'elle. Pour pouvoir le faire, nous utilisons l'artifice de la stocker dans un entier où les 8 bits de poids le plus faible valent `1110_1001` et où tous les autres bits valent zéro. Ceci peut être réalisé au moyen d'une instruction telle que : `int myInt = myByte & 0xFF` (Annexe B). Par exemple notre byte `1110_1001` se trouverait ainsi stocké dans un entier :

```
00000000 00000000 00000000 11101001
```

ce qui préserve sa valeur binaire dans la codification ISO-8859-1. Cet entier vaut bien la valeur voulue 233.

Soit `tabByte` le tableau de `byte` obtenu suite à la conversion de `input` selon la nome ISO-8859-1. La méthode `encodeString` doit transformer les `maxLength` premier octets de `tabByte` en entiers selon le schéma décrit ci-dessus : chaque octet est placé dans un entier où il occupe les 8 bits de poids le plus faible, les autres octets de l'entier valant zéro.

Si la séquence d'octets est plus petite que `maxLength` alors tous les bytes seront transformés et la séquence retournée fera la même longueur que la séquence d'octets.

3.2 Ajout d'informations

Implémentez la méthode :

```
int[] addInformations(int[] inputBytes)
```

qui prend en argument une séquence d'octets (stockés dans des entiers) et la complète en y ajoutant les informations qui la caractérisent (mode d'encodage, longueur etc.).

Attention, ce que l'on entend par octet dans ce qui suit est la codification sur 8 bits d'un caractère (il s'agit donc des 8 bits les moins significatifs des entiers qui les stockent).

Pour ajouter les informations caractéristiques, il faut restructurer la séquence d'octets comme suit :

- Les 4 bits de poids fort du premier octet de la séquence déterminent le mode utilisé. L'indicateur du « byte mode » est **'0100'**.
- les 8 bits suivants (4 bits de poids faible du premier octet et 4 bits de poids fort du second octet) déterminent le nombre d'octets dans le message (`inputBytes`).
- Les bits suivants sont les octets contenus dans `inputBytes`
- Les 4 bits de poids faible du dernier octet sont les bits de terminaison '0000'.

Notez que 16 bits sont ainsi ajoutés et que la liste retournée doit faire 2 octets de plus que celle donnée en argument. Notez aussi qu'il ne suffit pas de copier les valeurs de `inputBytes` directement dans la nouvelle séquence car toutes les valeurs ne sont plus parfaitement alignées sur un octet, mais sont à cheval entre 2 octets. Voir tableau ci-dessous sachant que "#i" signifie « byte » d'indice i de `inputBytes`.

octet 0	octet 1	octet 2	...	octet (Taille+1)
0100 0000	0101 1111	1110 1111	...	1110 0000
Indicateur Taille	Taille #0	#0 #1	...	#(Taille-1) Terminaison

Exemple : Nous voulons encoder la séquence $\{254, 254, 254, 254, 254\}$ (Noté que 254 est représenté par 1111_1110 en binaire. La taille de la séquence est 5 (0000_0101 en binaire sur un octet). Alors le premier octet (octet 0) est composé de l'indicateur de mode (0100) et des 4 bits de poids fort de la taille de la séquence (0000). Le premier octet est donc 0100_0000 soit 64. L'octet 1 est composé des 4 bits de poids faibles de la taille (0101) ainsi que des 4 premier bits du premier octet du message (1111). Le second octet est donc 0101_1111 ou 95 en décimal. Les octets 2 à 5 sont 1110_1111 (239) et l'octet 6 est 1110_0000 (224). Donc la séquence finale devrait être $\{64, 95, 239, 239, 239, 239, 224\}$.

3.3 Complétion

Implémentez la méthode :

```
int[] fillSequence(int[] encodedData, int finalLength)
```

qui complète la séquence donnée jusqu'à ce qu'elle fasse la taille `finalLength`. Les octets 11101100 (236 en decimal) et 00010001 (17 en decimal) doivent être ajoutés successivement à la suite des données encodées jusqu'à ce que la totalité des données contiennent exactement `finalLength` octets. Si `finalLength` est plus petit ou égal au nombre d'octets encodés alors la méthode devra retourner `encodedData` sans modifications.

Exemple : Imaginons qu'après avoir ajouté l'indicateur de mode, l'indicateur de taille et les bits de terminaison, l'on obtient la séquence de 10 octets suivante {34, 13, 52, 2, 09, 201, 10, 0, 59, 100}. Imaginons que `finalLength` est 19 (nécessaire pour la version 1), il faut donc ajouter 9 octets avec la valeur 236 et 17, ce qui nous donne une séquence final comme suit : {34, 13, 52, 2, 09, 201, 10, 0, 59, 100, 236, 17, 236, 17, 236, 17, 236, 17, 236}.

3.4 Correction d'erreurs

Une fois que les données sont encodées en « byte mode », il est nécessaire de rajouter les octets de correction d'erreurs. Ces octets permettent de reconstruire des données effacées, modifiées ou mal lues par le lecteur de QR code. Ces octets sont ensuite concaténés à la suite des données principales du QR code. Les octets de correction sont créés à l'aide du code de Reed-Solomon. Celui-ci étant relativement complexe, nous fournissons une méthode permettant de réaliser le traitement voulu :

```
int[] ErrorCorrectionEncoding.encode(int[] data, int n)
```

qui retourne `n` octets de correction pour les données `data`. L'utilisation de cette méthode nécessite l'importation de : `reedsolomon.ErrorCorrectionEncoding`.

Implémentez la méthode :

```
int[] addErrorCorrection(int[] encodedData, int eccLength)
```

qui retourne une liste d'octets comprenant les octets de `encodedData` suivis des `eccLength` octets de correction correspondants (généré au moyen de la méthode fournie).

3.5 Séquence binaire

Nous avons à présent fini d'encoder le message à positionner dans le QR code. Par commodité, il serait cependant plus lisible de disposer de ces données sous forme binaire (noir/blanc).

Implémentez la méthode :

```
boolean[] bytesToBinaryArray(int[] data)!
```

qui prend en argument une séquence d'octets et la convertit une séquence binaire, où `true` représente le chiffre `1` et `false` représente le chiffre `0`.

Exemple Considérez : `data = {170, 15}`, alors, puisque :
$170 = 0b10101010$
$15 = 0b00001111$
la méthode doit retourner :

```
{true,false,true,false,true,false,true,false, // 1 0 1 0 1 0 1 0
false,false,false,false,true,true,true,true}; // 0 0 0 0 1 1 1 1
```

3.6 Assemblage du tout

Dans la méthode

```
boolean[] byteModeEncoding(String input, int version)
```

appelez séquentiellement, avec les arguments adéquats, les méthodes décrites ci-dessus afin de générer une séquence d'octets utilisable par un QR code de version donnée. Cette méthode doit utiliser toutes les méthodes implémentées dans la Section 3

Le nombre maximal de caractères à encoder en ISO-8859-1 selon la version peut être trouvé par la méthode

```
int QRCodeInfos.getMaxInputLength(int version)
```

Le nombre final d'octets de la séquence sans les octets de correction est donné par la méthode

```
int QRCodeInfos.getCodeWordsLength(int version)
```

Le nombre d'octets de correction à génerer en fonction de la version est donné par

```
int QRCodeInfos.getECCLength(int version).
```

3.7 Test

Pour tester cette partie, vous pouvez bien sûr utiliser/augmenter le programme principal fourni, `Main.java`, à votre guise.

Nous mettons également à votre disposition ce que l'on appelle des *tests unitaires*[4]. Celui qui peut être utilisé pour cette partie s'appelle `DataEncodingTest.java` (dans le répertoire `test/qrcode/`.

Pour le lancer dans Eclipse, faites un clic-droit sur `DataEncodingTest.java`, puis « Run as » et « JUnitTest ». Un fenêtre devrait apparaître indiquant que certains tests sont échoués (en rouge ou bleu) et d'autres réussis (en vert).

À ce stade, `DataEncodingTest.java` doit valider tous les tests (vert).

4. vous apprendrez à en écrire vous-même au second semestre

4 Construction de la matrice

Dans cette section, nous nous intéressons à l'initialisation d'une matrice de QR code, indépendamment du message qu'elle est destinée à recevoir. Cette initialisation consiste à construire une matrice de bonne taille et à y placer des motifs permettant la bonne détection du QR code par un lecteur automatique. Il faudra également placer les informations relatives au format utilisé (niveau de correction et masque utilisé).

À la fin de l'étape vous devriez être capable d'afficher une ébauche de QR code tel que Fig. 2.

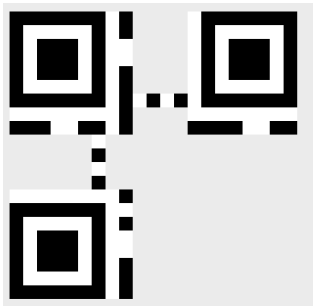


Fig. 2: QR code sans données

Terminologie : Les motifs prédéfinis (« patterns ») se placent toujours selon les mêmes conventions : il s'agit des « Finder patterns », occupant 3 coins du QR code, des « Timing patterns » constitué d'une ligne verticale et une horizontale, des « Alignment patterns » et du « module noir ».

La création d'un QR code prêt à accueillir des données se fera donc en 6 étapes :

1. créer un QR code de taille correcte (Section 4.1) ;
2. y ajouter les « Finder patterns » (Section 4.2) ;
3. y ajouter les « Alignment patterns » (Section 4.3) ;
4. y ajouter les « Timing patterns » (Section 4.4) ;
5. y ajouter le module noir (Section 4.5) ;
6. et y ajouter les informations sur le format (Section 4.6).

Chacun des traitements ci-dessus sera pris en charge par une méthode. Une septième méthode sera utilisée pour assembler le tout (Section 4.7).

Les méthodes décrites dans cette section doivent être codées dans le fichier `MatrixConstruction.java`.

4.1 Représentation et initialisation du QR code

Dans ce projet, nous allons représenter la matrice du QR code comme une image ARGB (voir Annexe C). Une image ARGB en java est représentée par un tableau à 2 dimensions d'entiers (`int`). Chaque entier représente la valeur d'un pixel dans son code couleur ARGB.

La matrice du QR code sera donc une image pour lequel chaque pixel correspond à un module (voir à ce propos le système de coordonnées utilisé pour référencer un pixel dans une image Annexe D). Les modules noirs correspondent aux bits `1` alors que les modules blancs aux bits `0`. Un module blanc sera représenté par un pixel de couleur blanche, et les modules noirs par un pixel de couleur noire. Les pixels doivent avoir une couleur **parfaitement noire et parfaitement blanche**, même si aucune distinction ne se fait à l'oeil nu, les projets ne respectant pas cette spécification seront pénalisés. **De plus, le composant alpha du code ARGB doit avoir la valeur maximale pour les deux types de modules ; soit 255.**

Définissez la méthode

```
int[][] initializeMatrix(int version)
```

dont le rôle est de créer et retourner une matrice de la taille nécessaire selon la version. Afin de connaître la taille pour une version donnée, utilisez la méthode :

```
int QRCodeInfos.getMatrixSize(int version)
```

Celle méthode retourne la largeur (et longueur), en nombre de modules, d'un QR code selon la version souhaitée (rappelons qu'un QR code doit être carré). La méthode `initializeMatrix` doit retourner une matrice vide, c'est à dire une matrice dont les pixels ont tous leurs composants ARGB à zéro, ce sont donc des pixels totalement « transparents ».

Définissez également deux constantes publiques et statiques[5], en haut de la classe (en dehors de toutes méthodes). La première contiendra la valeur ARGB (en `int`) du blanc et la seconde, celle du noir. Nous vous conseillons de nommez ces constantes `W` pour le blanc (« white ») et `B` pour le noir (« black »), mais vous pouvez décider de votre propre appellation. Cela permettra d'exprimer facilement le placement des modules blancs ou noirs.
Il est par ailleurs conseillé de définir d'autres couleurs pour vous aidez dans le débuggage de votre projet.

Complétez la méthode

```
int[][] constructMatrix(int version, int mask)
```

en conformité avec la description de Section 4.7 et exécutez la méthode `main()` de la classe `Main.java`. Le programme devrait fonctionner sans erreur et afficher ... un carré grisé. C'est normal, et il s'agit maintenant de compléter la matrice pour se rapprocher d'une image de QR code.

Chacune des étapes suivantes consiste à coder une méthode. De la même façon que vous l'avez fait pour `initializeMatrix`, il est conseillé de compléter la méthode `constructMatrix` en y appelant chaque méthode nouvellement codée, et ce, au fur et à mesure que vous progresserez. Cela vous permettra de tester votre avancement étape par étape.

4.2 « Finder Patterns »

Le premier motif à placer sont les « Finder patterns ». Un « Finder pattern » consiste en un carré noir de 7x7 comprenant en son centre un carré blanc de 5x5 modules, lui-même comprenant un carré noir de 3x3 (voir Fig. 3). Ces motifs sont toujours placés dans le coin haut-gauche,

5. A l'image de ce que vous faîtes quand vous utilisez un `Scanner` dans les exercices du TP. Nous verrons un peu plus tard ce que `static` signifie.

haut-droit et bas-gauche, peu importe la version.

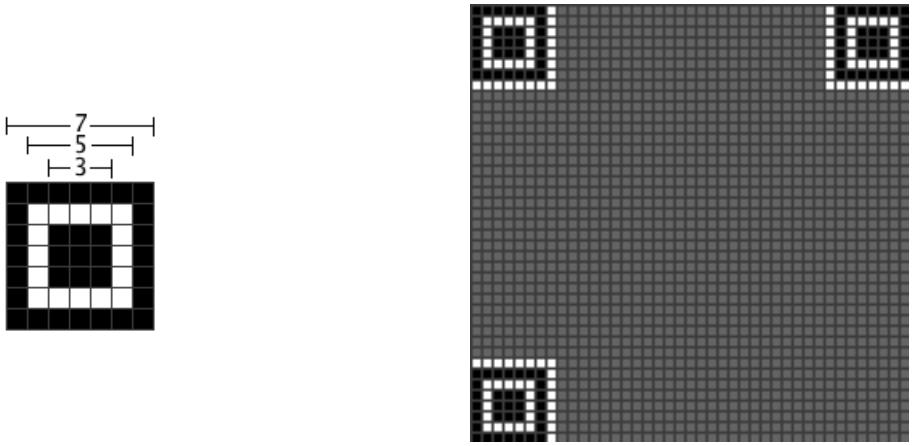


Fig. 3: à gauche le motif « Finder pattern », à droite son placement dans la matrice, avec l'ajout de séparateurs.

Il y a donc trois (3) motifs à placer dans les coins de la matrice telle que montré dans la figure Fig. 3. Il faut ensuite ajouter ce que l'on appelle les « séparateurs ». Il s'agit des lignes/colonnes de modules blancs entourant les bords intérieurs des « Finder Patterns », et que l'on peut également visualiser sur la figure de droite (Fig. 3).

Implémentez cette étape dans la méthode :

`void addFinderPatterns(int[][] matrix)`.

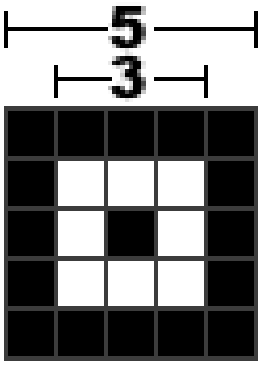
Indications Pour éviter la copie de code, définissez une méthode auxiliaire permettant de placer un motif donné à un endroit donné. Si la fonction est bien conçue elle peut être également utilisée pour les étapes suivantes. Comme indice utile, pensez à définir un motif donnée comme une constante.

4.3 « Alignment patterns »

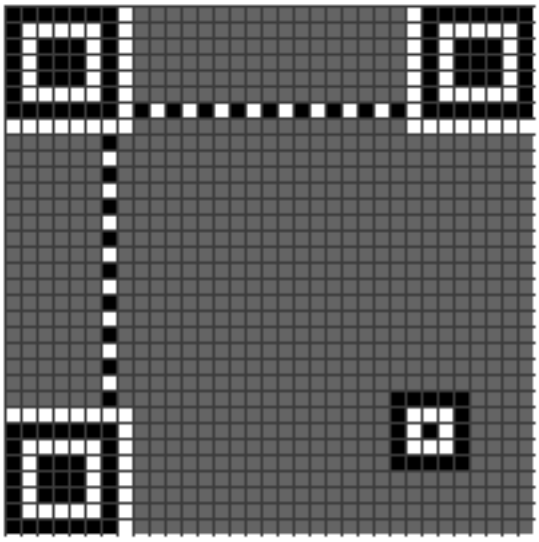
Les QR code de version 2 ou plus possèdent ce que l'on appelle des « Alignment pattern ». Ce sont des motifs de 5x5 modules, tel que celui présenté à gauche dans la figure Fig. 4a. Les versions 2 à 4 ne comportent qu'un seul de ces motifs, placé en bas à droite (voir la partie droite de la figure Fig. 4a) : le centre du motif se situe à 7 pixels des extrémités de la matrice. Les coordonnées du pixel central sont donc (`matrix.length` -7, `matrix.length`-7). Les QR code de version 1, ne doivent pas contenir de « Alignement pattern ».

Implémentez l'ajout de ces patterns dans la méthode :

`void addAlignmentPatterns(int[][] matrix, int version)` qui prend en argument la matrice à remplir ainsi que la version du QR code.



(a) « Alignment pattern »



(b) Positions des motifs

FIG. 4: Motifs « Alignment » et « Timing »

4.4 « Timing patterns »

Dans ce qui suit, la numérotation des lignes et colonnes commence à zéro, comme en Java. Les « Timing Patterns » sont deux lignes, une verticale et une horizontale, qui font alterner un module noir et un module blanc. La ligne horizontale est placée sur la $6ième$ ligne de la matrice et commence par un module noir à la $8ième$ colonne. Elle se termine 8 colonnes avant la fin de la matrice, soit juste au début du séparateur vertical en haut à droite. La ligne verticale est placée sur la $6ième$ colonne et commence par un pixel noir à la $8ième$ ligne. Elle se termine 8 lignes avant la fin de la matrice soit juste avant le séparateur horizontal en bas à gauche.(voir FIG. 4b) Il n'est pas nécessaire de vérifier la présence de superpositions avec un autre motif car ceux-ci coïncident toujours, par calcul, pour avoir des couleurs en adéquation avec à ceux du « Timing Pattern ».

Faites en sorte que l'ajout de ces motifs soit effectué par la méthode :

```
void addTimingPatterns(int[][] matrix)
```

4.5 Ajout du module noir

Tous les QR codes possèdent un module noir situé juste à coté du « Finder Pattern » en bas à gauche du QR code. Plus précisément le module à la position $(8, size - 8)$, où size est la longueur/largeur en module du QR code, doit être noir. Ajoutez le au moyen de la méthode :

```
void addDarkModule(int[][] matrix)
```

À ce stade du développement, vous devriez pouvoir faire apparaître une image similaire à la figure 5 pour un QR code de version 2.

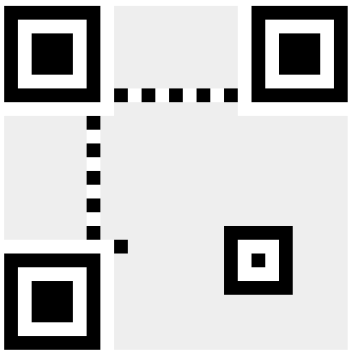


Fig. 5: Motifs d'un QR code de version 2

4.6 Information sur le format

Avant que le QR code ne soit prêt à accueillir ses données, il faut ajouter les informations relatives à son format. Le format d'un QR code est défini par son *niveau de correction* ainsi que le *masque* utilisé sur les données. Le masquage des données est décrit dans la Section 5.1. Il n'est pas nécessaire d'en comprendre le fonctionnement pour le moment. Il suffit de savoir qu'un masque est caractérisé par un entier. C'est cet entier qui sera utilisé pour générer l'information sur le format. Implémentez la méthode :

```
void addFormatInformation(int[][] matrix, int mask)
```

qui prend en argument la matrice du QR code et y ajoute les bits d'informations décrivant son format. Le format du QR code sera, dans le cadre de ce projet, toujours avec un niveau de correction **L**.

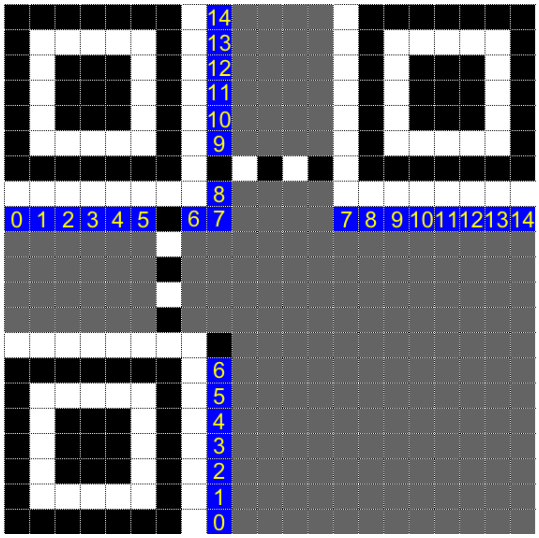
Pour générer la séquence de bits qui représente le format, il est nécessaire d'appliquer une technique dite « code correcteur de Reed-Solomon »[6] appliquée au niveau binaire. La connaissance de cette technique (codée dans le dossier fourni `reedsolomon`) dépasse le cadre de ce projet. Vous utiliserez donc la méthode fournie :

```
boolean[] QRCodeInfos.getFormatSequence(mask)
```

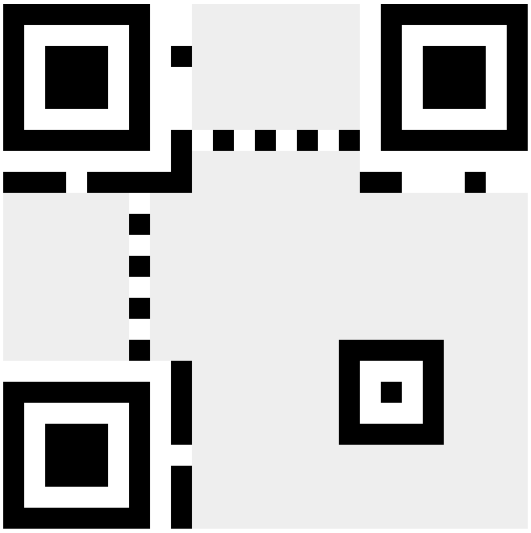
pour calculer la séquence de bits décrivant le format pour un niveau de correction **L** et un masque donné. La séquence retournée est une liste de booléens où `true` représente le `1` et `false` représente un `0`. La liste est ordonnée de telle sorte à ce que le premier booléen soit le bit de poids fort et le dernier celui de poids faible.

La séquence décrivant le format doit être placée deux fois dans le QR code. Une fois autour du « Finder Pattern » en haut à gauche et une fois à droite du « Finder Pattern » en bas à gauche, en continuant en dessous de celui situé en haut à droite du QR code. Le placement est détaillé sur la figure 6a. Le nombre 0 sur la figure fait référence au premier élément de la séquence. En d'autres termes, le bit représenté à l'indice 0 de la séquence doit être placé à la position 0 sur l'image et ainsi de suite.

6. code de Reed-Solomon



(a) Position de l'information concernant le format du QR code



(b) Résultat final de l'étape avec un QR code de version 2 et le masque 0

4.7 Assemblage du tout

Implémentez la méthode :

```
int[][] constructMatrix(int version,int mask)
```

qui retourne la matrices pour un QR code de la bonne version prêt à accueillir les données. Cette méthode appelle séquentiellement les 6 méthodes décrites plus haut **dans l'ordre des étapes données**.

Une fois la méthode implémentée, utilisez le masque 0 et une version 2 pour le moment. Vérifiez que le résultat est conforme à la figure 6b.

4.8 Test

À ce stade, `MatrixConstructionTest01.java` doit valider tous les tests. Si un test n'est pas validé, allez dans `Debug.java`, décommentez le test échoué et exécutez ce programme. Une fenêtre vous donnera des détails sur la raison de l'échec.

5 Placement des données

Il est temps maintenant de combiner le résultat des deux sections précédentes afin d'ajouter notre message sur le QR code.

Les méthodes décrites dans cette section doivent être codées dans le fichier `MatrixConstruction.java`.

5.1 Masquage

Dans tout QR code, il faut en principe appliquer un masque sur les modules de données. Un masque est un motif qui définit quels modules doivent échanger leur couleur. Le masque définit pour chaque module si celui-ci est masqué ou non. Si le module est masqué alors sa couleur doit être changée. Par exemple, si un module de couleur noire est masqué alors il doit être prendre la couleur blanche. S'il n'est pas masqué, sa couleur reste noire.

Le masque est une condition nécessaire pour que le QR code soit lisible (mais nous laisserons la possibilité de ne pas masquer les données). Il sert principalement à limiter le nombre de modules similaires se succédant, ainsi que l'apparition de certains motifs qui rendraient le QR code plus difficilement détectable.

Les spécifications d'un QR code laissent le choix au développeur de choisir parmi les 8 motifs suivants de masque :

Numéro du masque	**Formule :** Si la formule est vraie, masquer le module x = indice de la colonne, y = indice de la ligne
0	$(x + y) \mod 2 == 0$
1	$y \mod 2 == 0$
2	$x \mod 3 == 0$
3	$(x + y) \mod 3 == 0$
4	$(\lfloor y/2 \rfloor + \lfloor x/3 \rfloor) \mod 2 == 0$
5	$((x * y) \mod 2) + ((x * y) \mod 3) == 0$
6	$(((x * y) \mod 2) + ((x * y) \mod 3)) \mod 2 == 0$
7	$(((x + y) \mod 2) + ((x * y) \mod 3)) \mod 2 == 0$

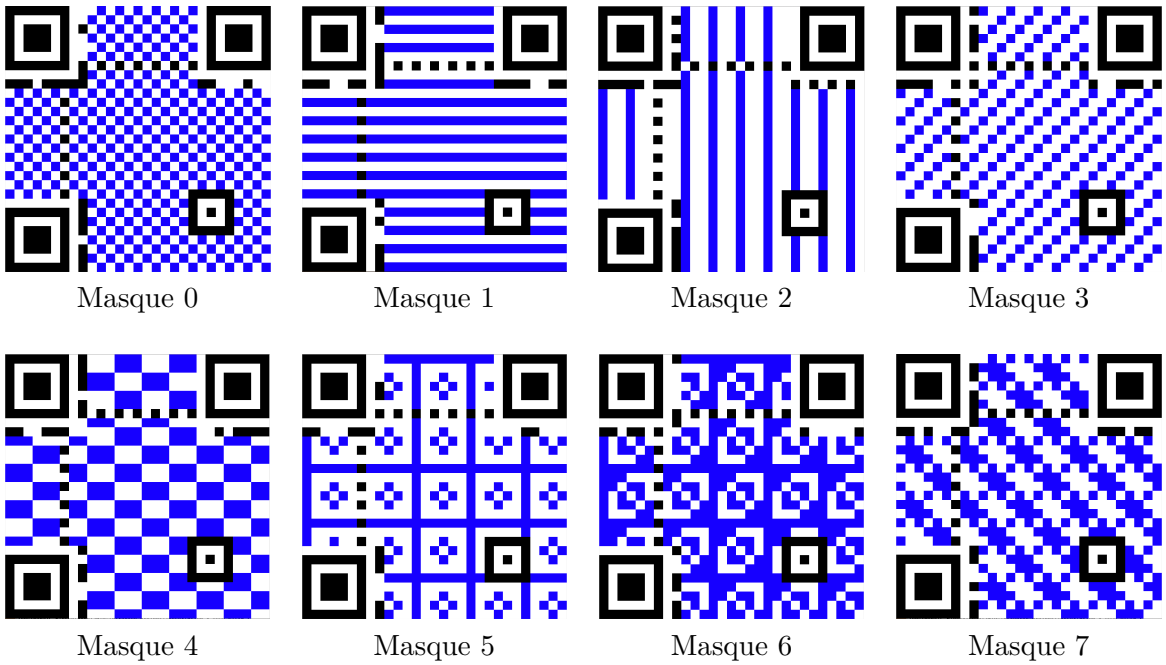


FIG. 7: Représentation des masques. Les modules bleus représentent les modules masqués

Implémentez la méthode :

```
int maskColor(int col, int row, boolean dataBit, int masking)
```

qui retourne la **couleur** en ARGB correspondant au bit de données `dataBit` après application du masque donné en argument. Les paramètres `col` et `row` correspondent à l'indice de la colonne et de la ligne.

Si le numéro du masque n'existe pas, donc n'est pas compris entre 0 et 7, la méthode devra retourner la couleur correspondant au bit de données sans masquage.

Rappel le chiffre binaire `1` est représenté par un module noir s'il n'est pas masqué et le chiffre `0` par un module blanc.(Section 4.1)

5.2 Ordres des bits de données

Il est temps maintenant de s'intéresser à l'algorithme de placement des données.

Chemin général : Les bits de données sont placés selon un chemin bien précis. Le début de ce chemin commence toujours dans le module du coin en bas à droite. Il monte ensuite dans une colonne de deux modules de large, une fois arrivé au sommet du QR code, il redescend sur la colonne suivante, toujours d'une largeur de deux modules et ainsi de suite. Il y a cependant une exception, la colonne comportant le « Timing Pattern » vertical est ignoré, le chemin fait donc un saut de 1 module de plus de large. (Voir FIG. 8)

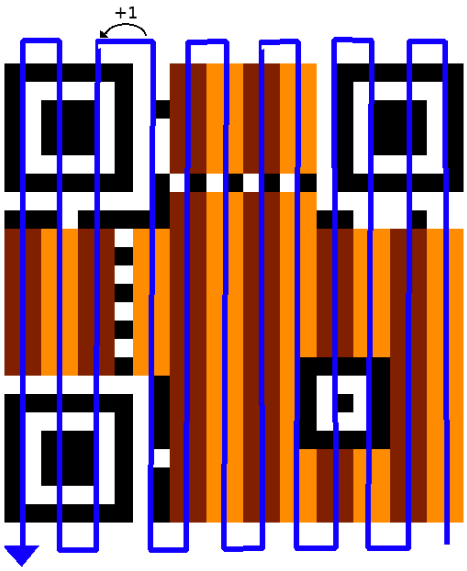


FIG. 8: Chemin pour le placement des bits.
Les modules clairs sont placés de bas en haut, les foncés de haut en bas.

Chemin montant et descendant : La montée et la descente sur la colonne de 2 modules de large se font en zigzag. On commence toujours par placer le bit sur le module de droite de la colonne avant de placer le bit sur le module de gauche. Les images suivantes montrent les mouvements de montée (FIG. 9a) et descente(FIG. 9b) pour l'ordre de placement des bits de données. Une démonstration du placement des modules est présenté en Annexe E, FIG. 12 et FIG. 13.

(a) Chemin montant. (b) Chemin descendant.

FIG. 9: Placement en zigzag sur une colonne large de 2 modules.

Ignorer les motifs : Lors du placement des bits sur les modules, il faut faire très attention à ne pas effacer un module contenant déjà de l'information. Pour cela, nous demandons que votre méthode place les données uniquement sur les modules dont le composant alpha est égal à zéro. Si le composant alpha n'est pas nul, il faut ignorer le module et placer le bit de données sur le prochain module disponible. **Indication :** Composez un chemin qui couvre l'entièreté de la matrice et chevauche les motifs (à la manière du chemin bleu sur la figure 8) mais ajoutez une condition pour que le bit soit placé seulement si le composant alpha du module est 0. Un exemple de rencontre avec un motif est montré en Annexe E, FIG. 14.

Éviter le « Timing pattern » : Comme cité précédemment, il est important de sauter la colonne du « Timing pattern vertical » (6). Attention, il ne faut pas confondre sauter et ignorer. Dans ce cas-là, ce n'est pas seulement que l'on ne place aucun bit sur la colonne, ici le chemin est modifié et « contourne » cette colonne, voir Fig. 8. **Indication :** Vous pouvez implémenter le chemin en 2 parties : l'une avant d'atteindre le « Timing pattern », et la seconde pour la partie restante à gauche de ce motif.

Masquage : Pour déterminer la couleur d'un module, utiliser la méthode :

`int maskColor(int col, int row, boolean dataBit, int masking)` précédemment implémentée.

Complétion : Une fois toutes les données placées sur la matrice, il est possible que tous les modules ne soient pas définis. S'il ne reste plus de bits de données mais qu'il reste des modules libres, ces modules doivent être complétés avec le bit `0`. **Il faut continuer à appliquer le masquage même sur les bits de complétion**.

Implémentation : En utilisant toutes ces informations, implémentez la méthode :

```
void addDataInformation(int[][] matrix, boolean[] data, int mask)
```

qui prend en argument la matrice du QR code pre-remplie avec tous les motifs nécessaire, les données à ajouter et le masque utilisé. Invoquez cette méthode dans

```
int[][] renderQRCodeMatrix(int version, boolean[] data,int mask)
```

pour finaliser le programme. Vous devriez alors avoir un QR code complet et lisible par un lecteur (par exemple sur votre téléphone portable) en lançant le fichier `Main.java`. Le masque à utiliser pour ce projet est le masque `0` (zéro). Assurez-vous que ce soit bien le cas avant d'envoyer votre code. Cependant les méthodes doivent fonctionner avec n'importe quel masque.

Conseil : Prenez le temps de réfléchir pour cette étape. Il y a beaucoup d'informations à gérer et il existe plusieurs façons d'aborder certains problèmes. Réfléchissez surtout à comment formuler de façon adéquate vos boucles. Dans un premier temps, il est sans doute préférable de commencer par compléter tous les modules de données avec des couleurs arbitraires (autres que noir/blanc). Ceci permettra de vérifier que vous ne remplacez pas les modules déjà existants. Testez aussi séparément l'ajout et la complétion des données, plutôt que d'essayer de finir toute cette étape en une traite avant de vérifier le résultat.

5.3 Test

À ce stade, `MatrixConstructionTest02.java` doit valider tous les tests. Si ce n'est pas le cas, utilisez `Debug.java` pour comprendre pourquoi.

Vous devriez à ce stade disposer d'un programme fonctionnel. Amusez vous à encoder vos messages dans plusieurs versions et masques et à les lire à l'aide de votre téléphone[7]/[8]

7. par exemple pour Samsung
8. ou pour Iphone

Vous pouvez également essayer d'évaluer empiriquement l'impact de la correction d'erreur. Ceci peut se faire par exemple en codant une méthode

```
int[][] damage(int[][] qrccode, double errorPercentage)
```

qui construit un QR code endommagé à partir du QR code fourni. Endommager un QR code peut revenir à simplement effaccer certains pixels (les rendre transparents). Le second paramètre donne le pourcentage de pixels endommagés souhaité.

Attention Il faudra cependant prendre garde à ne pas endommager les « Finder patterns », sans quoi il est peu probable que le lecteur automatique de votre téléphone parvienne à un décodage satisfaisant.

Vous devriez pouvoir vérifier que les QR code « proprement » endommagés restent lisibles pour des seuils d'erreur $< 7\%$[9].

9. Notez cependant que cela peut aussi dépendre de la qualité de votre lecteur

6 Pour aller plus loin

Dans cette section, **la partie optionnelle** du projet est décrite. L'implémentation de bonus vous permet de compenser des points éventuellement perdus dans la partie obligatoire, mais la note du projet reste plafonnée à 6. Notez aussi que la mise en oeuvre de la partie bonus implique que **vous fassiez preuve de plus d'indépendance et les assistants vous aideront en principe moins**.

Vous trouverez ci-dessous la description de quelques bonus envisageables. Seul l'un de ces bonus est décrit en détail. Si vous souhaitez implémenter d'autres bonus non décrits en détail, parmi ceux suggérés ou pas, nous vous conseillons de vous référencer au tutoriel en ligne https ://www.thonky.com/qr-code-tutorial/introduction. Ce document vous suggérera des fonctionnalités supplémentaires possibles ou des possibilités de mise en oeuvre.

- Avant de vous lancer dans un bonus non décrit en détail dans l'énoncé, veuillez vous assurer que votre implémentation ne perturbe pas le résultat des parties obligatoires. Il faudra donc créer de nouvelles méthodes indépendantes de celles existantes. Si vous avez le moindre doute sur la validité ou les modalités de mise en oeuvre d'un bonus, n'hésitez pas à consulter un assistant ou poster un message sur le forum.
- En cas de codage de un ou plusieurs bonus, vous créerez un fichier `Readme.md` ou `Readme.txt`, à la racine du projet, où vous décrirez la nature du bonus et comment utiliser ce que vous avez implémenté.
- L'extension détaillée ci-dessous sera codée dans le fichier `MatrixConstrution.java`. Pour d'autres extensions, et si vous n'estimez pas pertinent de placer le code dans les fichiers déjà prévus, vous pouvez utiliser un nouveau fichier nommé `Extensions.java`.

Voici quelques idées possibles :

- Évaluer et choisir le masque de façon plus appropriée (Section 6.1).
- Encoder des données en modes « Numeric » ou « Alphanumeric ».
- Ajouter des fonctionnalités permettant de réordonner les données et la correction d'erreur (nécessaire pour les versions de QR code supérieur à 4).
- Proposer une variante de la méthode `void addAlignmentPatterns(int[][] matrix, int version)` qui fonctionne pour toutes les versions de QR code.

Dans ce qui suit, vous trouverez la description de comment mettre en oeuvre le premier de ces bonus.

6.1 Évaluation d'un QR code

Implémentez la méthode :

```
int evaluate(int[][] matrix)
```

qui retourne le nombre de points de pénalité d'un QR code.

Les points de pénalités sont calculés de la manière suivante :

- Pour chaque ligne et chaque colonne, si il y a plus de 5 modules de la même couleur consécutivement, ajoutez 3 points de pénalité pour les 5 premiers modules plus 1 point par modules supplémentaires. Par exemple, si une ligne du QR code suit cette séquence de modules `{blanc, noir, noir, noir, noir, noir, noir,blanc, blanc}` , on rajoutera 4 points de pénalité pour les 6 modules noirs consécutifs.
- Pour chaque carré 2x2 de modules similaires, ajoutez 3 points de pénalité. Les carrés peuvent se superposer (voir Fig. 10)
- La quatrième règle de pénalité cherche les séquences :

  `{white, white, white, white, black, white, black, black, black, white, black}`
  et
  `{black, white, black, black, black, white, black, white, white, white, white}`

  pour chaque ligne et colonne. Chaque séquence trouvée rajoute 40 points de pénalité à l'évaluation. (En règle générale, il faudrait rajouter une bordure d'une largeur de 1 module blanc autour du QR code avant de calculer la pénalité de cette étape. Mais nous accepterons les projets ne l'ayant pas fait.)
- Les derniers points de pénalité sont calculés avec la formule suivante :
  1. Calculer le nombre de modules total dans la matrice.
  2. Calculer le nombre de modules noirs.
  3. Calculer le pourcentage de modules noirs. $\frac{blackModules}{nModules} * 100$
  4. Déterminer les pourcentages précédent et suivant, multiples de 5. Par exemple, pour le pourcentage 34, le multiple de 5 précédent est 30 et le suivant 35. Si la valeur est déjà un multiple de 5, cette valeur sera prise comme le plus petit pourcentage multiple de 5. Exemple : pour le pourcentage 25, le multiple de 5 précédent est 25 et le suivant 30.
  5. Soutraire 50 aux deux valeurs et prendre la valeur absolue des résultats.
  6. Finalement, prendre la plus petite valeur des 2 résultats précédent et la multiplier par 2. Rajouter celle-ci aux points de pénalité.

Vous pouvez comparer vos résultats avec le site suivant : https ://www.nayuki.io/page/creating-a-qr-code-step-by-step, en prenant en compte le fait que la 4ème règle de pénalité est légèrement différente sur le site (voir cette discussion sur le forum du cours).

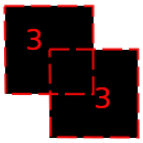


Fig. 10: Pénalité de carré 2x2

6.2 Sélection du masque

Une fois la méthode précédente complétée, implémentez la méthode :

`int findBestMasking(int version, boolean[] data)`

qui retourne l'indice du masque produisant le moins de points de pénalités avec les données pour la version.

Pour cela, construisez une matrice pour chaque masque et utilisez la méthode `int evaluate(int[][] matrix)` pour déterminer le masque ayant le moins de pénalité.

Pour finir, la méthode `int[][] renderQRCodeMatrix(int version, boolean[] data)`(déjà implémentée) vous permettra de construire un QR code doté du masque le plus adapté pour les données.

A Manipulation des bits en java

En java, les valeurs numériques peuvent être exprimées dans des bases différentes. Celle par défaut est la base 10, mais il est aussi possible de définir une valeur en base 2 (binaire) ou 16 (hexadécimale). Pour écrire un nombre en hexadécimal, il faut précéder sa représentation hexadécimale de `0x`, pour écrire un nombre en binaire il faut le précéder de `0b`. Par exemple si vous avez besoin de la suite binaire `1010` il est préférable d'utiliser sa représentation binaire en java `0b1010` plutôt que sa représentation décimale `10`.

Rappel des opérateurs binaires en java

Nom	Symbole	Exemple
OR	`|`	`0b001 | 0b011 ≡ 0b011`
AND	`&`	`0b001 & 0b011 ≡ 0b001`
XOR	`^`	`0b001 ^ 0b011 ≡ 0b010`
NOT	`~`	`~0b011 ≡ 0b100`
décalage droit	`>>`	`0b1000 >> 2 ≡ 0b0010`
décalage gauche	`<<`	`0b1010 << 2 ≡ 0b101000`

B Utilisation d'octets non signés en java

En Java, les octets (`byte`) sont des nombres *signés* utilisant la représentation en complément à 2. Dans cette représentation, le premier bit détermine le signe du nombre. Un `byte` en java possède donc une valeur entre -128 et 127.
De plus, de par le fait de la promotion entière, le type de retour des opérateurs binaires est `int`. Si l'on combine donc deux variables de type `byte` à l'aide d'un opérateur binaire, le résultat sera de type `int` et non `byte`. Ainsi, si l'on souhaite manipuler des octets en tant que tels, il devient nécessaire de convertir tous les résultats d'opérations en `byte`. Le problème que cela pose est que Java fait cette conversion sur la base de la valeur signée. Or, dans le cadre de ce projet nous désirons avoir des octets *non signés*, entre 0 et 255, car c'est ainsi que sont codifié les caractères dans la norme ISO-8859-1. L'utilisation d'octets signés n'est donc pas adaptée pour notre projet et peut résulter en une perte d'information au niveau binaire.

Pour illustrer avec un exemple :

```
byte b = -128; //10000000 en binaire
int i = (int)b; //-128 soit 1111 1111 1111 1111 1111 1111 1000
   0000 en binaire
//les deux résultats ne sont pas équivalents au niveau binaire.
```

Pour cette raison nous n'allons pas utiliser le type Java `byte` mais uniquement le type `int`. Dans la Section 3, les octets qui codifie un caractères seront stockés sous la forme de `int` où les 8 bits de poids faible (les 8 derniers bits) représentent l'octet et les 24 bits de poids fort sont à 0.

Remarque Une manière simple de vérifier que les entiers ont leurs 24 bits de poids fort à 0 est de vérifier que leur valeur est comprise entre 0 et 255.

Exemples : Représentation de l'octet 11111111 sur un entier de 32 bits.

Représentation hexadécimale	00							00	00	FF						
Représentation sur 32 bits	0	0	0	0	0	.	.	.	0	0	1	1	1	1	1	1	1	1
indice	31	30	29	28	27	.	.	.	9	8	7	6	5	4	3	2	1	0

Conversion d'un `byte` en `int` en conservant sa représentation binaire :

```
byte b = -128; //10000000 en binaire
int i = (int)(b & 0xFF); //128 soit 0000 0000 0000 0000 0000 0000
   1000 0000 en binaire
//les deux résultats sont équivalents au niveau binaire.
```

C Représentation ARGB

La représentation ARGB d'un pixel à l'aide d'un entier se fait de la manière suivante :
La couleur est décomposée en 4 composants ayant une valeur comprise entre 0 et 255 :

- **Alpha** est l'opacité du pixel. Si elle est à 0 alors le pixel est « invisible », peu importe ses autres composants. Si elle est entre 1 et 254 alors il est transparent et laisse passer les couleurs de l'image derrière celui-ci. Si sa valeur est 255, alors le pixel est parfaitement opaque.
- **Red** est la valeur d'intensité de la lumière rouge du pixel.
- **Green** est la valeur d'intensité de la lumière verte du pixel.
- **Blue** est la valeur d'intensité de la lumière bleue du pixel.

En considérant que le bit de poids le plus faible d'un entier ait pour indice 0 et que le bit de poids fort ait pour indice 31, alors ces 4 composants sont placés sur un même entier de telle sorte : les bits 31 à 24 définissent la valeur alpha, les bits 23 à 16 la valeur rouge, les bits 16 à 8 la valeur verte et les bits 7 à 0 la valeur bleue. Par exemple, si l'on veut représenter un pixel rouge en ARGB on a besoin que le composant alpha et rouge soit au maximum soit 255. Ce qui nous donne :

Alpha	Rouge	Vert	Bleu
255	255	0	0
0xFF	0xFF	0x00	0x00
11111111	11111111	00000000	00000000
31 24	23 16	15 8	7 0

La manière la plus pratique pour représenter une couleur en Java est d'utiliser l'écriture hexadécimale (plus concise). Par exemple pour définir le pixel rouge nous pouvons écrire :
`int red = 0xFF_FF_00_00;`

D Coordonnées d'un pixel

Le système de coordonnées usuellement utilisé pour représenter une image est particulier. L'origine est située dans le coin en haut à gauche de l'image. L'abscisse et l'ordonnée suivent la représentation montrée par la figure Fig. 11. Comme il est plus intuitif d'accéder à un pixel de l'image au moyen de ses coordonnées dans ce repère, il est d'usage de stocker l'image dans une

matrice Java où les lignes représentent les composantes en `y` et les colonnes celles en `x`. Ainsi si nous souhaitons que le pixel de coordonnées `(13,6)` dans le repère soit rouge, on écrira en Java :

```
matrix[12][5] = red;
```

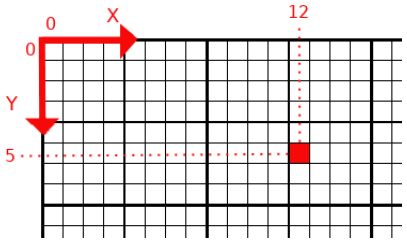
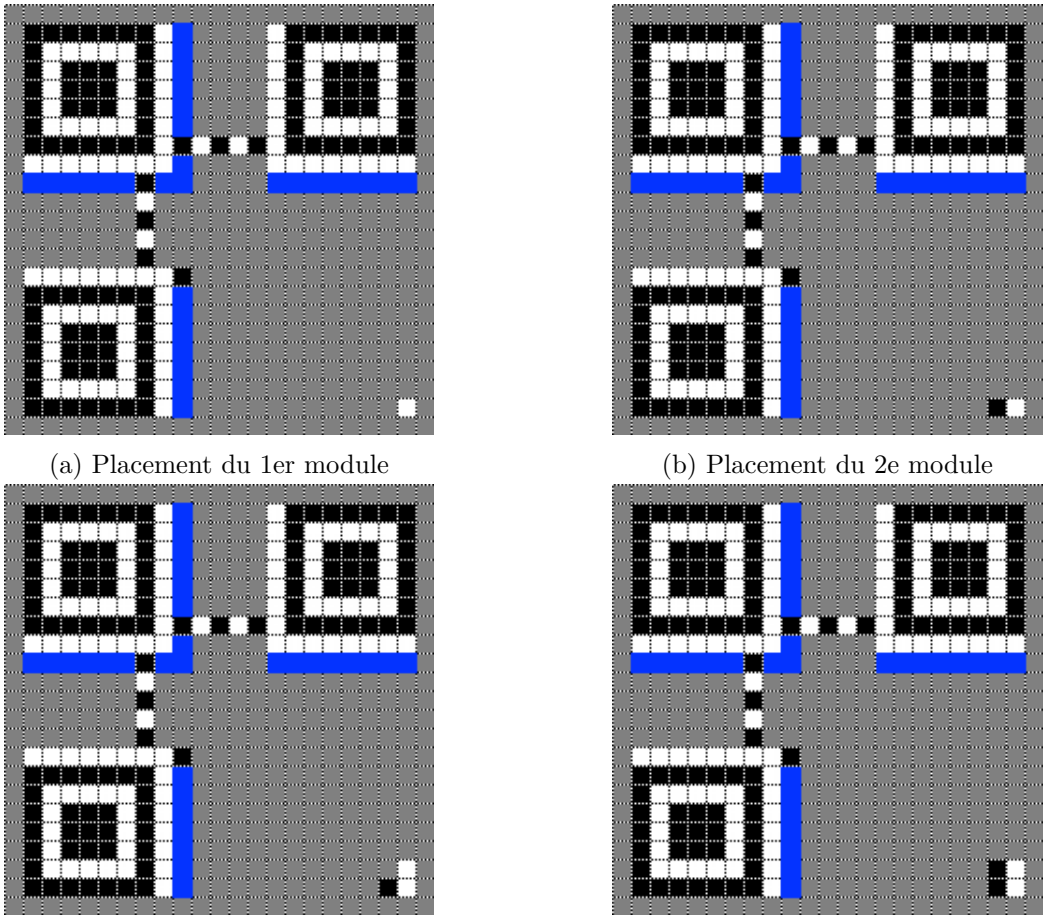


FIG. 11: Coordonnées des pixels dans une image

La classe fournie, `Helpers.java`, est codée selon cette convention.

E Démonstration du placement des modules



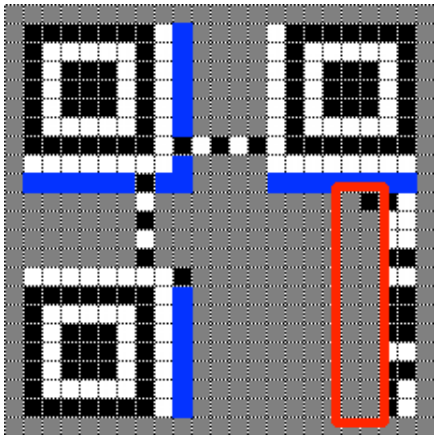
(a) Placement du 1er module

(b) Placement du 2e module

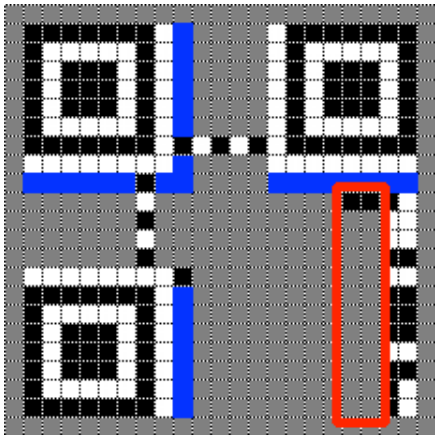
(c) Placement du 3e module

(d) Placement du 4e module

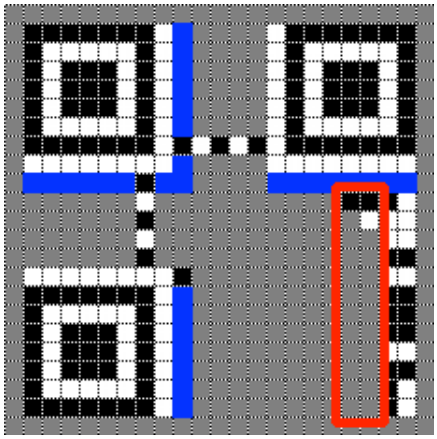
Fig. 12: Placement ascendant des modules



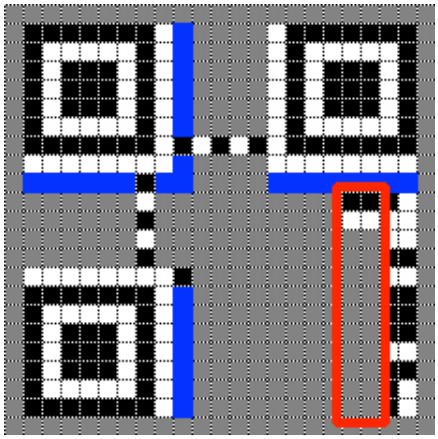
(a) Placement du 1er module



(b) Placement du 2e module

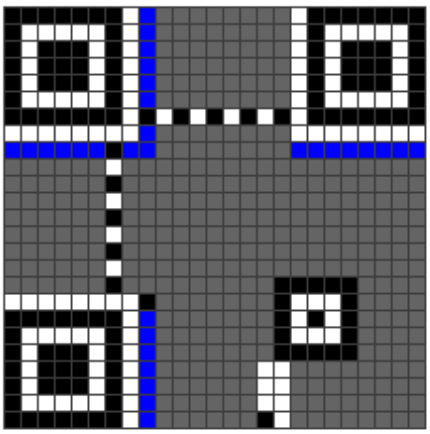


(c) Placement du 3e module

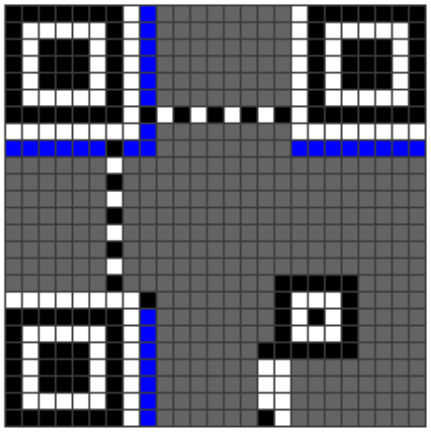


(d) Placement du 4e module

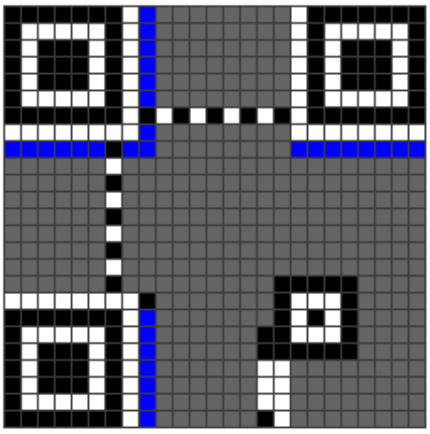
FIG. 13: Placement descendant des modules



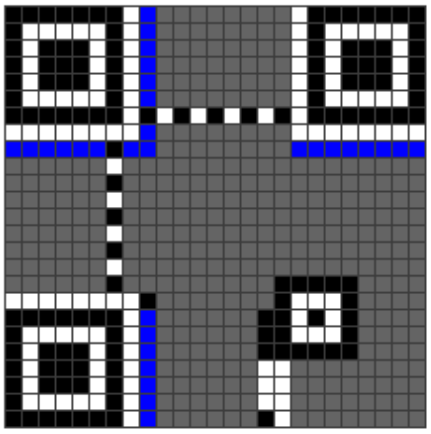
(a) Placement du 1er module



(b) Placement du 2e module



(c) Placement du 3e module



(d) Placement du 4e module

FIG. 14: Placement ascendant des modules lors d'une rencontre avec un motif