Pratique de la programmation orientée-objet
Examen intermédiaire

19 avril 2023

Indications :

— l'examen dure de 13h15 à 15h00,

— indiquez votre nom, prénom et numéro SCIPER ci-dessous et sur toutes les éventuelles feuilles additionnelles que vous rendriez,

— placez votre carte d'étudiant·e sur la table.

Seuls les documents suivants sont autorisés, en version papier uniquement :

— les notes de cours,

— les énoncés et les corrigés des séries d'exercices,

— une feuille de résumé de format A4 au maximum, ne couvrant que le cours.

Aucun document concernant le projet n'est autorisé !

Bon travail !

Nom : _____

Prénom : _____

SCIPER : _____

1 Compression de vecteur de bits [25 points]

Dans la version 19 de Java, la méthode suivante a été ajoutée à la classe `Integer` :

```
public static int compress(int i, int mask) { ... }
```

Cette méthode extrait les bits de `i` dont le bit correspondant de `mask` vaut 1, et les place dans les bits de poids faible du résultat, en préservant leur ordre. Les éventuels bits restants du résultat valent tous 0.

Par exemple, appliquée aux argument `i` et `mask` suivants — où les 32 bits de `i`, qui valent bien entendu chacun 0 ou 1, sont représentés par des lettres pour faciliter la compréhension :

```
   i = abcdefghijklmnopqrstuvwxyzABCDEF
mask = 10000011100110000000100000001101
```

la méthode `compress` retourne :

```
00000000000000000000aghilmuCDF
```

Le but de cet exercice est d'écrire la méthode `compress` et de l'utiliser pour résoudre un problème rencontré dans le cadre du projet Javions.

Partie 1 [10 points] Écrivez le corps de la méthode `compress`. Votre mise en œuvre doit manipuler exclusivement des valeurs de type **`int`** — au moyen, par exemple, des opérateurs bit à bit ou de décalage — et ne peut utiliser ni collections, ni tableaux, ni chaînes de caractères.

Réponse :

Partie 2 [10 points] Parmi les propriétés suivantes de la méthode `compress`, cochez celles qui sont vraies pour toutes valeurs `i`, `m` et `n` de type **`int`**. Vous pouvez faire l'hypothèse que $0 \leq \texttt{n} < 32$.

- ☐ `compress(i, i) == i`,
- ☐ `compress(i, m) == compress(i & m, m)`,
- ☐ `compress(i, -1) == i`,
- ☐ `compress(i, -1 << n) == i >> n`,
- ☐ `compress(i, 1 << n) == (i >> n) & 1`.

Partie 3 [5 points] Utilisez la méthode `compress` pour écrire une méthode permettant de « démêler » les 12 bits de l'attribut `ALT` (altitude) d'un message ADS-B de positionnement en vol dont le bit Q vaut 0. Pour mémoire, ces 12 bits sont initialement ordonnés ainsi :

$$C_1 \quad A_1 \quad C_2 \quad A_2 \quad C_4 \quad A_4 \quad B_1 \quad D_1 \quad B_2 \quad D_2 \quad B_4 \quad D_4$$

et le but du démêlage est de les réordonner ainsi :

$$D_1 \quad D_2 \quad D_4 \quad A_1 \quad A_2 \quad A_4 \quad B_1 \quad B_2 \quad B_4 \quad C_1 \quad C_2 \quad C_4$$

Votre méthode doit impérativement utiliser `compress` pour extraire les bits à démêler, mais ne peut pas l'appeler plus de trois fois, ni contenir de boucle.

Réponse :

2 Correction d'erreurs [25 points]

Dans le projet Javions, un contrôle de redondance cyclique de 24 bits (CRC24) est utilisé pour tenter de détecter les messages ADS-B corrompus. Pour ce faire, le CRC24 des 112 bits d'un message complet — composé de 88 bits de données suivis de leur CRC24 — est calculé, et le message est considéré comme valide si et seulement si ce CRC vaut 0.

Lorsque ce CRC ne vaut pas 0, il se trouve que sa valeur permet de savoir si, en inversant *un seul* bit du message complet, on peut en obtenir un valide — c.-à-d. dont le CRC vaut 0. Cette propriété peut être utilisée pour corriger les erreurs de transmission n'affectant qu'un seul bit d'un message.

En effet, le calcul du CRC a la caractéristique importante suivante : en prenant un message ADS-B valide (dont le CRC vaut 0) et en inversant son bit d'index n, on obtient un nouveau message dont le CRC — non nul — ne dépend que de n, et pas de la valeur des bits du message initial !

Par exemple, en inversant le premier bit d'un message ADS-B *quelconque* mais valide, on en obtient un dont le CRC vaut toujours 1491358 ; de même, en inversant le second bit d'un tel message, on en obtient un dont le CRC vaut toujours 745679 ; et ainsi de suite, l'inversion de chacun des 112 bits produisant un CRC non nul différent, nommé le **syndrome** de ce bit.

Dès lors, à la réception d'un message ADS-B, si l'on constate par exemple que son CRC est égal au syndrome du premier bit (1491358), alors on sait qu'en inversant ce bit, on obtiendra un message valide. On peut donc faire l'hypothèse que ce bit a été corrompu lors de la transmission, l'inverser, et traiter ce message corrigé comme valide.

Pour pouvoir ainsi détecter et corriger efficacement les erreurs affectant un seul bit d'un message ADS-B, on peut construire en avance ce que l'on nomme la **table des syndromes**, qui associe à chaque syndrome d'un des 112 bits d'un message, l'index de ce bit. Étant donné ce qui a été dit plus haut, cette table contiendra une entrée associant l'index 0 au syndrome 1491358, une autre associant l'index 1 au syndrome 745679, et ainsi de suite.

Le but de cet exercice est de compléter la définition de la classe `OneBitErrorDetector` ci-dessous, qui permet de détecter et corriger les erreurs de 1 bit dans les messages ADS-B.

```
public final class OneBitErrorDetector {
  // à faire (attributs)
  public static void flipBit(byte[] bytes, int bitIndex) { /* à faire */ }
  public OneBitErrorDetector(Crc24 crc24) { /* à faire */ }
  public int indexOfBitToFlip(int messageCrc) { /* à faire */ }
}
```

La valeur `crc24` passée au constructeur est un calculateur de CRC24 du même type de celui du projet. Pour mémoire, sa seule méthode publique est :

```
public int crc(byte[] bytes) { ... }
```

qui retourne le CRC24 du tableau d'octets passé en argument.

Partie 1 [2 points] Écrivez la déclaration de l'unique attribut de `OneBitErrorDetector`, qui contient la table des syndromes.

Réponse :

Partie 2 [4 points] Écrivez le corps de la méthode statique `flipBit`, qui inverse la valeur du bit d'index `bitIndex` du tableau d'octets `bytes`.

Comme lors du calcul du CRC, les bits du tableau d'octets sont numérotés de gauche à droite, c.-à-d. que le bit d'index 0 du tableau correspond au bit d'index 7 (celui de poids le plus fort) de l'octet d'index 0 du tableau, tandis que le bit d'index 7 du tableau correspond au bit d'index 0 de ce même octet. Le bit d'index 8 du tableau correspond au bit d'index 7 de l'octet d'index 1 du tableau. Et ainsi de suite.

Réponse :

Partie 3 [15 points] Écrivez le corps du constructeur, qui crée la table des syndromes et la stocke dans l'attribut correspondant. Le calcul des syndromes est fait avec l'instance de `Crc24` passée en argument.

Notez que le CRC d'un tableau ne contenant que des 0 vaut 0, ce qui implique que le syndrome d'un bit donné peut être déterminé en calculant le CRC d'un tableau dont seul ce bit vaut 1.

Réponse :

Réponse (suite) :

Partie 4 [4 points] Écrivez le corps de la méthode `indexOfBitToFlip`, qui retourne l'index du bit d'un message ADS-B à inverser pour en obtenir un qui soit valide, sachant que son CRC est égal à la valeur passée en argument. Si cette valeur ne correspond à aucun syndrome, -1 est retourné.

Réponse :

3 Bourrage d'octets [25 points]

Le terme **bourrage d'octets** (*byte stuffing*) désigne les techniques permettant de transformer une séquence d'octets en une autre, potentiellement plus longue, de sorte à ce que :

— la séquence transformée ne contienne aucun octet nul (c.-à-d. égal à 0), même si la séquence originale en contient,

— la séquence originale puisse être reconstruite à partir de la séquence transformée,

— la séquence transformée ne soit pas beaucoup plus longue que la séquence originale.

L'intérêt de cette transformation tient au fait que la séquence transformée ne contient aucun octet nul, ce qui permet d'utiliser cette valeur dans différents buts. Par exemple, si on désire stocker dans un fichier plusieurs séquences d'octets de longueur variable, on peut les séparer les unes des autres au moyen d'un octet nul. Pour garantir que les séquences elles-mêmes ne contiennent pas un tel octet, on peut les transformer par bourrage d'octets avant de les stocker dans le fichier.

COBS (*consistent overhead byte stuffing*) est une technique de bourrage d'octets. Elle consiste à tout d'abord augmenter la séquence originale en lui ajoutant un octet nul, puis à la découper en un nombre minimum de blocs successifs, de manière à ce que chaque bloc soit constitué :

— soit de 254 octets non nuls,

— soit de 0 à 253 octets non nuls suivis d'un octet nul.

Cela fait, chacun des blocs est représenté au moyen d'une séquence d'octets obtenue ainsi :

— si le bloc est constitué de 254 octets non nuls, la séquence consiste en un octet d'en-tête valant 255, suivi des 254 octets non nuls du bloc,

— si le bloc est constitué de n octets non nuls ($0 \leq n \leq 253$) suivi d'un octet nul, la séquence consiste en un octet d'en-tête valant $n+1$, suivi des n octets non nuls du bloc.

La concaténation des séquences représentant les différents blocs constitue la séquence d'octets transformée. Par construction, cette séquence ne contient aucun octet nul, et il n'est pas difficile de voir que la séquence originale peut en être reconstruite.

Cette technique peut être illustrée en l'appliquant à la séquence de huit octets [61, 62, 63, 0, 65, 0, 0, 68]. Après avoir augmenté cette séquence d'un octet nul, on la découpe en quatre groupes qui sont [61, 62, 63, 0], [65, 0], [0] et [68, 0]. Ces groupes contiennent tous moins de 254 octets non nuls suivis d'un octet nul et sont donc tous représentés au moyen d'un octet d'en-tête donnant leur longueur totale, suivi de leurs éventuels octets non nuls. La séquence transformée est donc [**4**, 61, 62, 63, **2**, 65, **1**, **2**, 68], où les octets d'en-tête sont en gras pour faciliter leur identification.

La classe ci-dessous, à compléter, possède une méthode permettant de faire du bourrage d'octets au moyen de la technique COBS.

```
public final class COBS {
  private COBS() {}
  public static List<Byte> stuff(List<Byte> bytes) { /* à faire */ }
}
```

Partie 1 [25 points] Écrivez le corps de la méthode `stuff` de la classe `COBS`, qui retourne la version transformée de la séquence d'octets donnée. Notez que dans un soucis de simplicité, les séquences d'octets sont représentées par des listes.

Réfléchissez bien avant d'écrire le code, qui n'est pas très long, mais subtile ! N'hésitez pas à utiliser une liste auxiliaire dans laquelle vous placez les octets du bloc en cours de construction si cela vous simplifie la tâche.

Réponse :

Formulaire

Ce formulaire présente toutes les parties de la bibliothèque Java nécessaires à cet examen. De nombreuses méthodes ont été omises et les types parfois simplifiés.

Interface `List`

L'interface `java.util.List` représente les listes. Elle est entre autres implémentée par la classe `ArrayList`, qui possède un constructeur de copie prenant une collection en argument.

L'interface `List` étend l'interface `Iterable`, donc ses éléments peuvent être parcourus au moyen d'un itérateur ou d'une boucle *for-each*.

```
public interface List<E> {
  // Retourne le nombre d'éléments contenus dans la liste.
  int size();

  // Ajoute l'élément e à la fin de la liste et retourne true.
  boolean add(E e);

  // Ajoute tous les éléments de c à la fin de la liste et retourne true.
  boolean addAll(Collection<E> c);

  // Efface la totalité des éléments de la liste.
  void clear();
}
```

Interface `Map`

L'interface `java.util.Map` représente les tables associatives. Elle est implémentée, entre autres, par les classes `HashMap` et `TreeMap`. Ces classes possèdent un constructeur de copie prenant une table associative en argument.

```
public interface Map<K, V> {
  // Retourne le nombre de paires clef/valeur présentes dans la table.
  int size();

  // Retourne vrai ssi la table contient la clef k.
  boolean containsKey(K k);

  // Associe la valeur v à la clef k.
  V put(K k, V v);

  // Retourne la valeur associée à k, ou null s'il n'y en a aucune.
  V get(K k);

  // Retourne la valeur associée à k, ou d s'il n'y en a aucune.
  V getOrDefault(K k, V d);
}
```