Examen final

CS-108

2025-05-30

1 Stéganographie [20 points]

Dans la série sur la stéganographie, nous avons vu comment cacher un message dans une image. Le but de cet exercice est d'adapter cette idée pour cacher un message dans un texte. Pour cela, nous pouvons utiliser le fait que la norme Unicode comporte plusieurs caractères représentant différentes sortes d'espaces, parmi lesquels on trouve :

1. l'espace « normale », dont le code Unicode est 20_{16},
2. l'espace insécable (en anglais : *non-breaking space*), dont le code Unicode est $A0_{16}$,
3. les deux tiers de cadratin (en anglais : *en space*), dont le code Unicode est 2002_{16}, et
4. le cadratin (en anglais : *em space*), dont le code Unicode est 2003_{16}.

Ces différentes sortes d'espaces jouent un rôle important en typographie mais sont souvent difficiles ou impossibles à distinguer visuellement. On peut donc les utiliser pour cacher un message dans un texte en remplaçant les espaces de ce texte par l'une des quatre espaces ci-dessus, ce qui permet d'encoder 2 bits à la fois. C'est le but de la classe `SpaceSteganographer` ci-dessous, à compléter dans le cadre de cet exercice :

```
public final class SpaceSteganographer {
  private SpaceSteganographer() {}

  private static final List<Character> SPACES =
    List.of('\u0020', '\u00A0', '\u2002', '\u2003');

  public static String encode(String substrate,
                              byte[] message) { /* à faire */ }
}
```

Pour mémoire, en Java, la notation `\uXXXX` représente le caractère dont le code Unicode est `XXXX` en hexadécimal (base 16). La liste `SPACES` contient donc les quatre espaces mentionnées plus haut, dans le même ordre.

Partie 1 [20 points] Écrivez le corps de la méthode `encode` qui retourne une chaîne dont le contenu est identique à celui de la chaîne `substrate`, si ce n'est que toutes les espaces d'une des quatre sortes mentionnées plus haut y ont été éventuellement remplacées par d'autres, dans le but de stocker le message donné. Chaque espace représente 2 bits du message, l'espace normale représentant `00`, l'insécable, `01`, les deux tiers de cadratin, `10` et le cadratin, `11`. Les octets sont stockés dans l'ordre, des bits de poids fort aux bits de poids faible. En d'autres termes, la première espace sert à encoder les deux bits de poids le plus fort du premier octet du message.

Si la chaîne `substrate` ne contient pas assez d'espaces pour y stocker la totalité du message, `encode` lève une `IllegalArgumentException` ; si elle en contient plus que nécessaire, alors les espaces inutilisées sont toutes remplacées par des espaces normales.

Le test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre le fonctionnement de cette méthode.

```
byte[] message = new byte[]{ (byte) 0b11_10_00_01 };
String substrate = "To be or not to be";
String actual = SpaceSteganographer.encode(substrate, message);
String expected = "To\u2003be\u2002or\u0020not\u00A0to\u0020be";
assertEquals(expected, actual);
```

2 Échantillonnage [20 points]

En informatique, il est parfois utile d'obtenir un **échantillon aléatoire** (*random sample*) d'une séquence de valeurs. Par exemple, si on dispose d'un fichier contenant un grand nombre de mesures de température, on peut vouloir en choisir aléatoirement quelques-unes et les afficher, dans le but d'avoir une idée des valeurs rencontrées.

Une technique permettant d'obtenir un échantillon aléatoire de taille n d'une séquence de valeurs $v_0, v_1, \ldots, v_m$ consiste à utiliser un tableau de taille n dans lequel on place tout d'abord les éléments $v_0, \ldots, v_{n-1}$. Ensuite, pour chaque élément v_i d'index $i \geq n$, on tire un nombre aléatoire j compris entre 0 (inclus) et i (inclus). Si $j < n$, alors on stocke l'élément v_i à l'index j dans le tableau, sinon on l'ignore. Lorsque tous les éléments de la séquence ont été traités de la sorte, le tableau contient l'échantillon désiré.

Le but de la classe `Sampler` ci-dessous, à compléter dans le cadre de cet exercice, est d'utiliser cette technique pour obtenir un échantillon aléatoire d'une séquence de valeurs de type `T`. Cette séquence n'est toutefois pas fournie en une fois, par exemple sous la forme d'une liste, mais ses éléments sont passés l'un après l'autre à la méthode `accept`.

```
public final class Sampler<T> {
  private final RandomGenerator rng;
  private int count;
  private final T[] samples;

  public Sampler(RandomGenerator rng, int size) { /* à faire */ }
  public void accept(T value) { /* à faire */ }
  public List<T> samples() { /* à faire */}
}
```

L'attribut `rng` contient le générateur aléatoire à utiliser pour tirer des nombre aléatoires, au moyen de la technique décrite plus bas. L'attribut `count` contient le nombre actuel d'éléments de la séquence, qui est simplement le nombre de fois que la méthode `accept` a été appelée. Finalement, `samples` est le tableau destiné à contenir l'échantillon.

Partie 1 [4 points] Écrivez le corps du constructeur, qui prend en arguments le générateur aléatoire et la taille de l'échantillon et les utilise pour initialiser les attributs de l'instance créée, ou lève une `IllegalArgumentException` si la taille est négative.

Partie 2 [10 points] Écrivez le corps de la méthode `accept`, qui reçoit le prochain élément de la séquence et le traite de la manière décrite plus haut.

Pour obtenir un nombre aléatoire, utilisez la méthode `nextInt` de l'attribut `rng`, qui prend en argument une borne $b > 0$ et retourne un entier aléatoire compris entre 0 (inclus) et b (exclu).

Partie 3 [6 points] Écrivez le corps de la méthode `samples`, qui retourne une liste immuable (!) contenant l'échantillon aléatoire de la séquence des valeurs passées à `accept`. Votre mise en œuvre doit gérer correctement le cas où la séquence a une taille inférieure à celle de l'échantillon désiré, en retournant dans ce cas une liste contenant la totalité des éléments de la séquence.

3 Images textuelles [25 points]

Une **image textuelle** est une grille bidimensionnelle infinie composée de caractères. Une telle image est similaire à ce que nous avons appelé une image continue dans les séries d'exercices, à la différence près qu'une image textuelle est discrète et composée de caractères plutôt que de points colorés.

L'interface fonctionnelle `TextImage` ci-dessous représente une telle image sous la forme d'une fonction qui, étant donné un index de ligne (`row`) et un index de colonne (`column`) dans la grille, retourne le caractère de l'image à cette position. Notez que ces index peuvent être quelconques, y compris négatifs.

```
public interface TextImage { char charAt(int row, int column); }
```

Par exemple, la valeur `CHESSBOARD` suivante représente un échiquier, c.-à-d. une image textuelle dont les caractères sont alternativement des dièses (`#`) et des espaces :

```
TextImage CHESSBOARD = (r, c) -> ((r + c) & 1) == 0 ? '#' : ' ';
```

En affichant une portion rectangulaire de cette image au moyen de cet extrait de programme :

```
for (int row = 0; row < 4; row += 1) {
  for (int col = 0; col < 50; col += 1)
    System.out.print(CHESSBOARD.charAt(row, col));
  System.out.println();
}
```

on obtient :

```
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
 # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
 # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
```

Le but de cet exercice est d'augmenter l'interface `TextImage` en lui ajoutant des méthodes permettant d'obtenir différentes images, ou d'en transformer des existantes.

Partie 1 [15 points] Ajoutez à l'interface `TextImage` une méthode statique `rectangle` prenant en arguments une largeur $w \geq 2$ et une hauteur $h \geq 2$ et retournant l'image textuelle d'un rectangle dont les quatre coins se trouvent aux index $(0,0)$, $(0,w-1)$, $(h-1,w-1)$ et $(h-1,0)$. Cette image doit être composée uniquement d'espaces, sauf sur le pourtour du rectangle qui doit être représenté au moyen de caractères plus (`+`), moins (`-`) et barre verticale (`|`). Par exemple, l'image :

```
TextImage.rectangle(20, 4)
```

affichée au moyen de l'extrait de programme plus haut doit donner :

```
+------------------+
|                  |
|                  |
+------------------+
```

Votre méthode doit vérifier que la largeur et la hauteur reçues sont bien supérieures ou égales à 2 et lever une `IllegalArgumentException` dans le cas contraire.

Partie 2 [4 points] Ajoutez à l'interface `TextImage` une méthode par défaut `over` prenant en argument une image textuelle et retournant la superposition de l'image du récepteur (`this`) au-dessus de celle passée en argument. Cette superposition est effectuée en considérant que les espaces de l'image du dessus sont transparentes et que l'image du dessous est donc visible à travers elles. Par exemple, l'image :

```
TextImage.rectangle(20, 4).over(TextImage.rectangle(25, 3))
```

affichée au moyen de l'extrait de programme plus haut doit donner :

```
+------------------+----+
|                  |    |
|------------------|----+
+------------------+
```

Partie 3 [6 points] Ajoutez à l'interface `TextImage` une méthode par défaut `translated` prenant en arguments un déplacement en lignes et un déplacement en colonnes et retournant une image identique à celle du récepteur mais translatée selon ces deux déplacements. Par exemple, l'image :

```
TextImage.rectangle(20, 4)
  .over(TextImage.rectangle(10, 4).translated(1, 15))
```

affichée au moyen de l'extrait de programme plus haut doit donner :

```
+------------------+
|               +---|----+
|               |   |    |
+------------------+    |
```

4 Arbre préfixe [60 points]

Dans le mini-projet sur la compression LZW, nous avons utilisé une table associative de type `Map<List<Integer>, Integer>` pour représenter le dictionnaire d'encodage. Pour mémoire, ce dictionnaire associe leur code à des séquences d'octets, représentées ici par des valeurs de type `List<Integer>`.

Plutôt que d'utiliser une table associative pour représenter ce dictionnaire d'encodage, il est possible d'utiliser un **arbre préfixe** (*prefix tree* ou *trie* en anglais), qui permet d'associer des valeurs à des séquences de « caractères » (au sens large) provenant d'un alphabet donné.

Un arbre préfixe est constitué de **nœuds** (*nodes*), et chaque nœud possède une éventuelle valeur associée et autant d'**enfants** (*children*) — d'autres nœuds, éventuellement inexistants — qu'il y a de caractères dans l'alphabet. La valeur associée à un nœud est celle que l'arbre préfixe associe à la séquence de caractères correspondant au chemin menant de la **racine** (*root*) de l'arbre au nœud en question. La racine est simplement le nœud de départ de l'arbre préfixe, qui correspond donc à une séquence de caractères vides.

Par exemple, si un compresseur LZW utilise l'alphabet {a,b,n} et que son dictionnaire d'encodage est le suivant :

chaîne (clef)	`a`	`b`	`n`	`na`	`an`	`baba`
index (valeur)	0	1	2	3	4	5

alors l'arbre préfixe correspondant est celui de la figure ci-dessous, où les nœuds sont représentés par des rectangles, la racine est grisée, et l'éventuelle valeur associée à un nœud apparaît en son centre. Les flèches menant d'un nœud à l'un de ses enfants sont étiquetées avec le caractère de l'alphabet auquel l'enfant correspond.

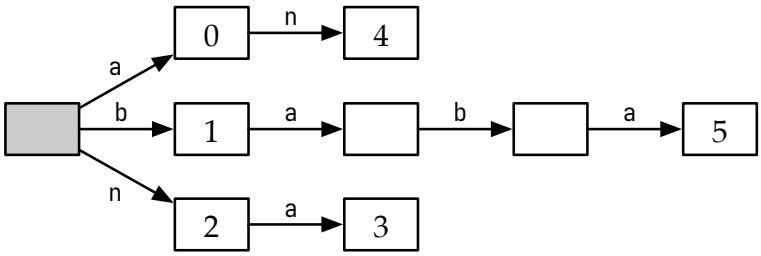

Fig. 1 : Arbre préfixe

Par exemple, pour retrouver la valeur associée à la chaîne `an` dans cet arbre, il suffit de partir de la racine, puis de se déplacer sur l'enfant associé au caractère `a`, puis, de là, sur celui associé au caractère `n` et de consulter sa valeur associée, 4. Si l'on cherche la valeur associée à la chaîne `ba`, on constate qu'il n'y en a aucune car le nœud sur lequel on termine le parcours n'a pas de valeur associée. De même, si l'on cherche la valeur associée à la chaîne `bn`, on constate qu'il n'y en a aucune car l'enfant de la racine correspondant à `b` ne possède aucun enfant correspondant à `n`.

La classe `ByteIntTrie` ci-dessous, à compléter dans le cadre de cet exercice, représente un arbre préfixe associant des valeurs de type `int` à des séquences d'octets de type `byte[]`. L'alphabet utilisé ici est donc celui des octets, et un « caractère » est un entier compris entre 0 (inclus) et 255 (inclus). La racine de l'arbre est stockée dans l'attribut `root`, et sa taille — le nombre d'associations clef/valeur — l'est dans l'attribut `size`.

```
public final class ByteIntTrie {
  private final Node root = new Node();
  private int size = 0;

  public int size() { return size; }
  public int get(byte[] key) { /* à faire */ }
  public int put(byte[] key, int value) { /* à faire */ }
  public void forEach(BiConsumer<byte[], Integer> consumer);
  public Iterator<KeyValue> iterator();

  public record KeyValue(byte[] key, int value) { }

  private static final class Node {
    static final int NO_VALUE = Integer.MIN_VALUE;

    final Node[] children = new Node[256];
    int value = NO_VALUE;
  }
}
```

L'enregistrement imbriqué `KeyValue` représente une paire clef/valeur, et est utilisé par la méthode `iterator`.

La classe imbriquée `Node` représente un nœud, dont les 256 fils sont stockés dans l'attribut `children`. Ce tableau ne contient initialement que des valeurs `null`, ce qui signifie que le nœud ne possède aucun enfant. L'éventuelle valeur associée au nœud est stockée dans l'attribut `value`, et la constante `NO_VALUE` indique son absence.

Partie 1 [10 points] Écrivez le corps de la méthode `get` qui retourne la valeur associée à la clef (séquence d'octets) `key`, ou `NO_VALUE` si aucune valeur n'est associée à cette clef.

Partie 2 [14 points] Écrivez le corps de la méthode `put` qui associe à la clef `key` la valeur `value`, ou lève une `IllegalArgumentException` si `value` vaut `NO_VALUE`. Si la clef est déjà associée à une valeur, la nouvelle valeur remplace simplement l'ancienne.

Partie 3 [16 points] Écrivez le corps de la méthode `forEach` qui appelle la méthode `accept` du consommateur passé en argument avec toutes les paires clef/valeur contenues dans l'arbre préfixe, dans un ordre quelconque. Pour mémoire, l'interface fonctionnelle `BiConsumer` est définie ainsi :

```
public interface BiConsumer<T,U> { void accept(T t, U u); }
```

Pour écrire la méthode `forEach`, vous pouvez vous aider de l'enregistrement `KeyNode` ci-dessous, qui représente une paire clef/nœud — à ne pas confondre avec `KeyValue` qui représente une paire clef/valeur :

```
record KeyNode(byte[] key, Node node) { }
```

Le parcours de l'arbre peut se faire au moyen d'une pile de telles paires, initialisée avec celle correspondant à la racine. Tant que cette pile n'est pas vide, l'élément au sommet peut en être retiré, le consommateur appelé avec la clef et la valeur associée au nœud, puis les paires correspondant aux enfants du nœud ajoutées à la pile.

Partie 4 [20 points] Écrivez le corps de la méthode `iterator`, qui retourne un itérateur permettant de parcourir les paires clef/valeur de l'arbre préfixe. Cet itérateur ne doit redéfinir que les méthodes `hasNext` et `next` de `Iterator`, en omettant la méthode `remove`.

Pour écrire cet itérateur, inspirez-vous de la manière dont la méthode `forEach` parcourt les nœuds de l'arbre.

5 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.

Classe `String`

La classe `String`, immuable, représente une chaîne de caractères.

```
public final class String {
  // Retourne la longueur de la chaîne.
  public int length();

  // Retourne le caractère à l'index donné, ou lève IndexOutOfBoundsException
  // si cet index est invalide.
  public char charAt(int index);
}
```

Classe `StringBuilder`

La classe `StringBuilder` représente un bâtisseur de chaînes de caractères.

```
public final class StringBuilder {
  // Ajoute le caractère donné à la chaîne en cours de construction.
  public void append(char c);

  // Retourne la chaîne en cours de construction.
  public String toString();
}
```

Classe `Arrays`

La classe `Arrays`, non instanciable, contient des méthodes statiques de manipulation de tableaux.

```
public class Arrays {
  // Retourne une copie du tableau original de longeur newLength.
  // Les éventuels nouveaux éléments valent 0.
  static byte[] copyOf(byte[] original, int newLength);
}
```

Classe `Byte`

La classe `Byte` sert de classe d'emballage pour le type primitif `byte` et fournit aussi certaines méthodes statiques manipulant des valeurs de ce même type.

```
public class Byte {
  // Convertit l'octet donné en entier de type int, en l'interprétant de manière non signée.
  static int toUnsignedInt(byte b);
}
```

Classe `Math`

La classe `Math`, non instanciable, contient des méthodes statiques permettant d'effectuer différents calculs.

```
public class Math {
  // Retourne le minimum des deux arguments.
  static int min(int x, int y);
}
```

Interface `List`

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

```
public interface List<E> {
  // Retourne une liste immuable contenant les éléments donnés.
  static <E> List<E> of(E... elements);

  // Retourne une copie immuable de la liste donnée.
  static <E> List<E> copyOf(List<E> l);

  // Retourne vrai ssi la liste est vide.
  boolean isEmpty();

  // Retourne vrai ssi la liste contient l'élément donné.
  boolean contains(E e);

  // Retourne l'élément à l'index donné, ou lève IndexOutOfBoundsException
  // si cet index est invalide.
  E get(int index);

  // Ajoute l'élément donné à la fin de la liste.
  void addLast(E e);

  // Supprime et retourne l'élément en fin de liste, ou lève NoSuchElementException
  // si la liste est vide.
  E removeLast();
}
```

Interface `Iterator`

L'interface `Iterator` représente un itérateur, c.-à-d. un objet permettant de parcourir les éléments d'une collection.

```
public interface Iterator<E> {
  // Retourne vrai ssi l'itérateur a encore un élément à fournir.
  boolean hasNext();

  // Retourne le prochain élément, ou lève NoSuchElementException s'il n'y en a plus.
  E next();
}
```