Examen final

CS-108

2024-05-31

1 Mélange de tableau d'entiers [25 points]

La classe `Collections` de la bibliothèque Java offre plusieurs variantes d'une méthode nommée `shuffle` qui permet de mélanger les éléments d'une liste. La version que nous avons utilisée dans le projet a la signature suivante :

```
public static <T> void shuffle(List<T> l, RandomGenerator r)
```

Malheureusement, ni cette méthode ni aucune autre de ses variantes ne permet de mélanger directement un tableau d'entiers primitifs de type `int[]`. Il n'est pas non plus possible de mélanger un tel tableau indirectement en utilisant la méthode `asList` de `Arrays` pour le voir comme une liste, car `asList` ne fonctionne pas avec un tableau dont les éléments sont d'un type primitif. Le but de cet exercice est d'examiner deux solutions à ce problème.

Partie 1 [12 points] Une première manière de mélanger un tableau d'entiers primitifs de type `int[]` consiste à écrire une méthode d'adaptation nommée `intArrayAsList`, similaire à `asList` de `Arrays` mais fonctionnant avec un tableau primitif :

```
static List<Integer> intArrayAsList(int[] array) {
  return new AbstractList<>() {...}
}
```

Une fois définie, cette méthode peut s'utiliser ainsi pour mélanger un tableau de type `int[]` au moyen de la méthode `shuffle` de `Collections` mentionnée plus haut :

```
RandomGenerator rnd = ...;
int[] a = new int[]{1, 2, 3, 4, 5};
Collections.shuffle(intArrayAsList(a), rnd);
```

▷ Complétez la méthode `intArrayAsList` en écrivant le corps de la sous-classe anonyme de `AbstractList` dont elle retourne une instance. Cette sous-classe doit contenir uniquement des redéfinitions des méthodes `size`, `get` et `set` de l'interface `List`, décrite dans le formulaire.

Partie 2 [13 points] Une seconde manière de mélanger un tableau d'entiers primitifs consiste à écrire une version de la méthode `shuffle` similaire à celle de `Collections` mais fonctionnant sur de tels tableaux :

```
static void shuffle(int[] array, RandomGenerator rnd) {...}
```

▷ Écrivez le corps de cette méthode en effectuant le mélange ainsi : parcourez les éléments du tableau du premier jusqu'à l'avant-dernier, et échangez chacun d'entre eux avec un élément d'index supérieur ou égal au leur, choisi aléatoirement.

Le choix aléatoire de l'élément avec lequel échanger le courant peut se faire au moyen de la méthode `nextInt` de `RandomGenerator` qui a la signature suivante :

```
int nextInt(int origin, int bound)
```

et qui retourne, à chaque appel, un nouvel entier aléatoire compris entre `origin` (inclus) et `bound` (exclu).

Notez bien que vous n'avez pas le droit d'utiliser la méthode définie à la partie 1 dans cette partie, ni la méthode `shuffle` de `Collections`.

2 Itérateurs [25 points]

En Java, les itérateurs (interface `Iterator`) et les flots (interface `Stream`) sont relativement similaires. Le but de cet exercice est d'examiner cette similarité en complétant la classe non instanciable `Iterators` ci-dessous, qui fournit des méthodes similaires à celles existant sur les flots, mais pour les itérateurs.

```
public final class Iterators {
  static <T> Iterator<T> concat(Iterator<T> it1, Iterator<T> it2) {...}
  static <T, R> Iterator<R> map(Iterator<T> it, Function<T, R> f) {...}
  static <T extends Comparable<T>> T max(Iterator<T> it) {...}
}
```

Partie 1 [8 points] ▷ Écrivez le corps de la méthode `concat` qui, étant donné deux itérateurs `i1` et `i2`, retourne un itérateur qui les concatène. En d'autres termes, cette méthode retourne un itérateur qui fournit d'abord les éléments de `i1`, puis ceux de `i2`.

Par exemple, l'extrait de programme suivant utilise la méthode `concat` pour obtenir un itérateur produisant les entiers de 1 à 5 :

```
Iterator<Integer> it = Iterators.concat(List.of(1, 2, 3).iterator(),
                                        List.of(4, 5).iterator());
// it produit 1, 2, 3, 4, 5
```

La méthode `remove` de l'itérateur retourné par `concat` doit simplement lever une exception de type `UnsupportedOperationException`. Pour mémoire, c'est ce que fait la mise en œuvre par défaut dans l'interface `Iterator` (voir le formulaire en annexe).

Partie 2 [9 points] ▷ Écrivez le corps de la méthode `map` qui, étant donné un itérateur `it` et une fonction `f`, retourne un itérateur dont chaque élément est le résultat de l'application de la fonction à l'élément correspondant de l'itérateur. Pour mémoire, une fonction est une instance d'une classe qui implémente l'interface fonctionnelle `Function` suivante :

```
public interface Function<T, R> { R apply(T t); }
```

Par exemple, l'extrait de programme suivant utilise la méthode `map` pour transformer un itérateur produisant des nombres en un itérateur produisant la représentation textuelle binaire de ces nombres :

```
Iterator<Integer> it = List.of(1, 2, 3, 4, 5).iterator();
Iterator<String> it2 = Iterators.map(it, Integer::toBinaryString);
// it2 produit "1", "10", "11", "100" et "101"
```

La méthode `remove` de l'itérateur retourné par `map` doit simplement lever une exception de type `UnsupportedOperationException`.

Partie 3 [8 points] ▷ Écrivez le corps de la méthode `max` qui, étant donné un itérateur `it` fournissant des éléments comparables entre eux, retourne le plus grand d'entre eux ; lève une exception de type `NoSuchElementException` si l'itérateur ne peut fournir aucun élément.

Par exemple, l'extrait de programme suivant utilise `max` pour déterminer le plus grand élément d'une liste de nombres :

```
Iterator<Integer> it = List.of(31, 5, 2024).iterator();
int m = Iterators.max(it);
// m vaut 2024
```

3 Comparateur lexicographique [25 points]

Lorsque des chaînes de caractères sont comparées entre elles, par exemple afin d'être triées, elles le sont généralement dans l'ordre **lexicographique**, aussi appelé **ordre du dictionnaire**.

Pour comparer deux chaînes selon cet ordre, on commence par comparer leur premier caractère. Si celui de la première chaîne est strictement plus petit que celui de la seconde chaîne, alors la première chaîne est plus petite que la seconde. Sinon, si le premier caractère de la première chaîne est strictement plus grand que celui de la seconde chaîne, alors la première chaîne est plus grande que la seconde. Finalement, si les deux premiers caractères sont égaux, la comparaison se poursuit avec le second caractère des deux chaînes, et ainsi de suite.

Si tous les caractères des deux chaînes sont égaux deux à deux, et que les deux chaînes ont la même longueur, alors elles sont égales. Sinon, la plus courte des deux est plus petite que l'autre.

Cette notion d'ordre lexicographique peut se généraliser à toute séquence de valeurs qu'il est possible de comparer entre elles. Par exemple, des listes d'entiers peuvent être ordonnées selon l'ordre lexicographique, car les entiers peuvent être comparés entre eux.

Le but de cet exercice est de compléter la définition de la méthode `lexicographic` ci-dessous qui, étant donné un comparateur de valeurs de type `T`, retourne un comparateur de listes de valeurs de type `T` utilisant l'ordre lexicographique.

```
static <T> Comparator<List<T>> lexicographic(Comparator<T> cmp) {...}
```

Par exemple, l'extrait de programme suivant utilise cette méthode pour trier un flot de listes d'entiers en ordre lexicographique :

```
List<List<Integer>> l = Stream.of(
    List.of(3, 1, 6), List.of(2), List.of(3, 1, 5, 6), List.of(2, 0))
  .sorted(lexicographic(Integer::compareTo))
  .toList();
// l vaut [[2], [2, 0], [3, 1, 5, 6], [3, 1, 6]]
```

▷ Écrivez le corps de la méthode `lexicographic`, en vous aidant au besoin de la définition de l'interface `Comparator` donnée en annexe.

4 Flot d'entiers à taille variable [25 points]

Une application désirant lire ou écrire des entiers de 24 bits non signés dans des flots Java (de type `InputStream` ou `OutputStream`) peut le faire en découpant chacun de ces entiers en 3 octets de 8 bits chacun, placés dans un ordre donné dans le flot.

Par exemple, si on note les 24 bits d'un tel entier au moyen des lettres minuscules de `a` à `x`, où `a` est le bit de poids le plus fort, `x` celui de poids le plus faible :

```
abcdefghijklmnopqrstuvwx
```

alors une manière d'écrire une telle valeur dans un flot d'octets consiste à écrire successivement les trois octets suivants : `abcdefgh`, `ijklmnop`, `qrstuvwx`.

Toutefois, si la plupart de ces entiers sont petits, et donc proches de zéro, une manière plus compacte de les représenter consiste à utiliser un encodage à taille variable, similaire à l'encodage UTF-8 utilisé pour les caractères Unicode. Pour cet exercice, nous utiliserons l'encodage suivant :

Entier	Encodage (1 à 4 octets)
`00000000000000000rstuvwx`	`rstuvwx1`
`0000000000klmnopqrstuvwx`	`klmnop10`, `qrstuvwx`
`000defghijklmnopqrstuvwx`	`defgh100`, `ijklmnop`, `qrstuvwx`
`abcdefghijklmnopqrstuvwx`	`00001000`, `abcdefgh`, `ijklmnop`, `qrstuvwx`

Par exemple, la première ligne de cette table signifie qu'un entier de 24 bits dont les 17 bits de poids fort valent 0 est représenté par un seul octet dont les 7 bits de poids fort sont ceux de l'entier (`rstuvwx`) et le bit de poids faible vaut 1 ; la seconde ligne signifie qu'un entier de 24 bits dont les 10 bits de poids fort valent 0 est représenté par deux octets, le premier contenant les 6 de poids fort (`klmnop`) et deux bits constants (`10`), le second contenant les 8 bits de poids faible (`qrstuvwx`) ; et ainsi de suite. Bien entendu, un entier qu'il serait possible d'encoder de plusieurs manières, comme 0, l'est toujours de la manière la plus courte.

Une caractéristique importante de cet encodage est qu'en comptant le nombre de bits 0 consécutifs dans les bits de poids faible du premier octet représentant un entier, on connaît le nombre d'octets supplémentaires à lire pour décoder cet entier.

Le but de cet exercice est de compléter la définition de la classe `VarIntInputStream` ci-dessous, qui représente un flot d'entrée d'entiers de 24 bits (non signés) encodés de la manière décrite plus haut. Cette classe lit les octets représentant ces entiers depuis le flot d'entrée `stream` passé à son constructeur, appelé le flot sous-jacent plus bas.

```
public final class VarIntInputStream implements AutoCloseable {
  private final InputStream stream;
  public VarIntInputStream(InputStream stream) {this.stream = stream;}

  public int readVarInt() throws IOException { ... }

  @Override
  public void close() throws IOException { stream.close(); }
}
```

L'extrait de test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre l'utilisation de cette classe pour lire l'entier 7 — représenté par un unique octet valant 1111 en base 2 — d'un flot.

```
try (InputStream bs = new ByteArrayInputStream(new byte[]{0b1111});
     VarIntInputStream vs = new VarIntInputStream(bs)) {
  assertEquals(7, vs.readVarInt());
}
```

▷ Écrivez le corps de la méthode `readVarInt` qui lit et retourne le prochain entier de 24 bits du flot, ou -1 si aucun octet n'est disponible dans le flot sous-jacent. Lève `IOException` en cas d'erreur d'entrée-sortie ou lorsque les octets lus du flot sous-jacent n'ont pas le bon format — p.ex. si le premier octet d'un entier vaut 0.

Notez que la classe `Integer` possède la méthode suivante, qui permet de compter le nombre de bits successifs valant 0 dans les bits de poids faible de l'entier qu'on lui passe :

```
static int numberOfTrailingZeros(int i)
```

Par exemple, `numberOfTrailingZeros(0b1000)` retourne 3. Cette méthode simplifie l'écriture de `readVarInt`.

5 Bâtisseur de chaînes [25 points]

La classe `StringBuilder` de la bibliothèque Java représente un bâtisseur de chaîne. Le but de cet exercice est de compléter la définition de la classe `SStringBuilder` ci-dessous, qui constitue une version simplifiée d'un tel bâtisseur permettant l'insertion d'un caractère à un index quelconque de la chaîne en cours de construction.

Cette classe représente la chaîne en cours de construction au moyen de ce que l'on nomme un **tableau à trou** (*gap buffer*). Il s'agit d'un tableau de caractères contenant un unique **trou** (*gap*), dont la taille peut aller de 0 à la taille totale du tableau. La chaîne en cours de construction est constituée de tous les caractères du tableau, excepté ceux du trou.

L'insertion d'un caractère à un index donné dans la chaîne en cours de construction se fait en déplaçant le trou pour qu'il commence à cet index, puis en remplaçant son premier caractère par celui à insérer. Si le trou est vide — c.-à-d. de taille 0 — au moment de l'insertion, le contenu du tableau est copié dans un nouveau tableau plus grand afin de créer un trou commençant à l'index d'insertion, après quoi son premier caractère est remplacé par celui à insérer.

Par exemple, imaginons que l'on désire construire la chaîne `EPFL` en insérant les caractères dans l'ordre `P` (à l'index 0), `E` (à l'index 0), `L` (à l'index 2) puis `F` (à l'index 2). Imaginons de plus que le tableau à trou soit initialement vide, mais ait une taille de 3 caractères. Son évolution lors de l'insertion des 3 premiers caractères est donnée dans la table ci-dessous, dans laquelle le symbole × est utilisé pour représenter les caractères faisant partie du trou :

	0	**1**	**2**	**Note**
1	×	×	×	état initial
2	P	×	×	insertion de `P` à l'index 0
3	×	×	P	déplacement du trou à l'index 0
4	E	×	P	insertion de `E` à l'index 0
5	E	P	×	déplacement du trou à l'index 2
6	E	P	L	insertion de `L` à l'index 2

Initialement (ligne 1), le tableau est constitué uniquement d'un trou de taille 3 commençant à l'index 0. L'insertion de premier caractère, `P`, peut donc se faire en remplaçant le premier caractère du trou par `P` (2). Cela fait, le trou commence maintenant à l'index 1, et a une taille de 2. L'insertion du second caractère, `E`, à l'index 0 nécessite tout d'abord de déplacer le trou afin qu'il commence à l'index 0 (3), après quoi son premier caractère peut être remplacé par celui à insérer (4). L'insertion du caractère suivant, `L`, à l'index 2 se passe de manière similaire (5-6).

L'insertion du prochain caractère, `F`, à l'index 2 n'est plus possible dans ce tableau, le trou étant de taille 0. Il faut donc commencer par créer un nouveau tableau plus grand, ici de taille 6, et y recopier les éléments de l'ancien de manière à ce que le trou commence à l'index d'insertion, à savoir 2 (ligne 7 dans la table ci-dessous). Finalement, le caractère `F` peut être inséré en remplacement du premier caractère du trou, dont la taille passe à 2 (8).

	0	**1**	**2**	**3**	**4**	**5**	**Note**
7	E	P	×	×	×	L	copie dans un tableau de taille 6
8	E	P	F	×	×	L	insertion de `F` à l'index 2

La classe `SStringBuilder` ci-dessous, à compléter, représente un bâtisseur de chaîne au moyen d'un tableau à trou. Le tableau (`buffer`) a initialement une taille de 16 caractères, le trou commence à l'index 0 (`gapPos`) et sa taille (`gapSize`) est celle du tableau.

```
public final class SStringBuilder {
  private char[] buffer = new char[16];
  private int gapPos = 0;
  private int gapSize = buffer.length;

```

```
  public int length() { ... }
  public SStringBuilder insert(int i, char c) { ... }
  @Override
  public String toString() { ... }
}
```

L'extrait de test JUnit ci-dessous, qui doit s'exécuter avec succès, illustre l'utilisation de cette classe pour construire la chaîne `EPFL` de la manière décrite plus haut :

```
String b = new SStringBuilder()
  .insert(0, 'P').insert(0, 'E').insert(2, 'L').insert(2, 'F')
  .toString();
assertEquals("EPFL", b);
```

Partie 1 [2 points] ▷ Écrivez le corps de la méthode `length` qui retourne la longueur de la chaîne en cours de construction, c.-à-d. le nombre d'éléments du tableau qui ne font pas partie du trou.

Partie 2 [17 points] ▷ Écrivez le corps de la méthode `insert`, qui insère le caractère `c` à l'index `i` dans la chaîne en cours de construction, puis retourne le bâtisseur lui-même. Lève `IndexOutOfBoundsException` si l'index n'est pas un index d'insertion valide.

Lorsque le trou a une taille de 0, cette méthode doit créer un nouveau tableau dont la taille est égale au double de la taille du tableau actuel, et y copier le contenu du tableau actuel en plaçant le trou à l'index d'insertion.

Lorsque le trou a une taille supérieure à 0 mais que sa position n'est pas égale à celle d'insertion, il doit y être déplacé au moyen d'une unique copie d'un certain nombre d'éléments du tableau. Deux cas sont alors à distinguer :

1. le trou se trouve avant l'index d'insertion `i`, auquel cas il faut copier `i - gapPos` éléments depuis l'index `gapPos + gapSize` vers l'index `gapPos`,
2. le trou se trouve après l'index d'insertion `i`, auquel cas il faut copier `gapPos - i` éléments depuis l'index `i` vers l'index `i + gapSize`.

Toutes ces copies peuvent se faire au moyen de la méthode `arraycopy` de `System` qui, pour mémoire, a la signature suivante :

```
static void arraycopy(Object s, int sI, Object d, int dI, int len)
```

et qui copie `len` éléments du tableau `s`, à partir de l'index `sI`, dans le tableau `d`, à partir de l'index `dI`. Les arguments `s` et `d` peuvent être le même tableau, auquel cas la copie est faite comme si les éléments à copier étaient d'abord copiés dans un tableau temporaire, puis copiés à nouveau depuis lui vers le tableau original.

Partie 3 [6 points] ▷ Écrivez le corps de la méthode `toString`, qui retourne la chaîne en cours de construction, constituée de tous les caractères du tableau, sauf ceux du trou. Aidez-vous pour cela de la méthode `valueOf` de la classe `String` :

```
static String valueOf(char[] array, int offset, int count)
```

qui retourne une chaîne de longueur `count` constituée des caractères du tableau `array` à partir de l'index `offset`.

6 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 `List` représente une liste.

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

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

  // Remplace l'élément à l'index i par v et retourne l'ancien élément, ou lève
  // IndexOutOfBoundsException si cet index est invalide.
  E set(int i, E v);

  // Retourne un itérateur parcourant les éléments de la liste.
  Iterator<E> iterator();
}
```

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();

  // Supprime l'élément retourné par le dernier appel à next.
  // Lève IllegalStateException si next n'a pas encore été appelée, ou si remove
  // a déjà été appelée après le dernier appel à next.
  // La mise en œuvre par défaut lève une UnsupportedOperationException.
  void remove();
}
```

Classe `InputStream`

La classe abstraite `InputStream` représente un flot d'entrée d'octets.

```
public abstract class InputStream {
  // Lit et retourne le prochain octet sous la forme d'une valeur comprise entre 0
  // et 255 (inclus), ou –1 si la fin du flot a été atteinte.
  int read();
}
```

Interface `Comparable`

L'interface `Comparable` est destinée à être implémentée par toutes les classes dont les instances peuvent être comparées (ordonnées) entre elles. De nombreuses classes de la bibliothèque Java l'implémentent, comme p.ex. `Integer`, `String`, etc.

```
public interface Comparable<T> {
  // Compare this et that et retourne un entier négatif si this < that,
  // zéro si this == that, et un entier positif sinon.
  int compareTo(T that);
}
```

Interface `Comparator`

L'interface `Comparator` représente un comparateur, c.-à-d. un objet capable de comparer deux objets d'un type donné.

```
@FunctionalInterface
public interface Comparator<T> {
  // Compare o1 et o2 et retourne un entier négatif si o1 < o2,
  // zéro si o1 == o2, et un entier positif sinon.
  int compare(T o1, T o2);
}
```