Théorie et Pratique de la Programmation
Examen final

31 mai 2013

Indications :
- l'examen dure de 11h15 à 13h00,
- indiquez votre nom, prénom et numéro SCIPER ci-dessous et sur toutes les éventuelles feuilles additionnelles que vous rendriez,
- placez votre carte d'étudiant sur la table.

Bon travail !

Nom : _____

Prénom : _____

SCIPER : _____

Flots compressés [10 points]

Le codage par plages (*run-length encoding* en anglais) est une technique de compression extrêmement simple mais néanmoins efficace dans certaines situations spécifiques. Elle tire parti des séquences de valeurs identiques dans les données à compresser en représentant chaque séquence de n occurrences d'une même valeur v par le nombre $n - 1$ suivi de la valeur v elle-même.

Soit par exemple la séquence de 12 octets suivante :

```
67 79 79 79 79 79 79 79 79 79 79 76
```

Cette séquence étant composée d'une occurrence de la valeur `67`, suivie de 10 occurrences de la valeur `79` et finalement d'une occurrence de la valeur `76`, sa version encodée par plage est la séquence d'octets suivante (les octets représentant les nombres d'occurrences sont grisés pour faciliter la compréhension) :

```
0 67 9 79 0 76
```

Le premier octet encodé est `0`, ce qui indique que $0 + 1 = 1$ occurrence de l'octet qui suit (à savoir `67`) apparaît initialement dans les données originales. L'octet suivant, `9`, indique que $9 + 1 = 10$ occurrences de l'octet qui suit (`79`) suivent dans les données originales. Et ainsi de suite.

Partie 1 [1 point] On désire offrir la possibilité de décoder un flot d'octets Java (de type `java.io.InputStream`) encodé au moyen de la technique décrite ci-dessus. Quel patron de conception peut-on avantageusement utiliser dans cette situation ?
Réponse : _____

Partie 2 [9 points] Appliquez le patron que vous avez identifié ci-dessus afin d'écrire une sous-classe instanciable de `InputStream` qui, étant donné un flot de données codées par plage, produit la version décodée de ces données.

Comme d'habitude, vous devez pour ce faire redéfinir les méthodes `read` et `close` héritées de `InputStream`. Pensez à lever l'exception `IOError` si le flot compressé est invalide !

Réponse :

(suite à la page suivante)

(suite de la page précédente)

Itérateurs [12 points]

Etant donnés deux itérateurs sur des entiers, il peut être utile d'obtenir un itérateur produisant la somme des entiers produits par chacun de ces itérateurs.

Pour ce faire, on peut définir une classe `AddingIterator` qui implémente l'interface `Iterator<Integer>` et dont le constructeur prend deux itérateurs en arguments. Les entiers produits par `AddingIterator` sont la somme des entiers produits individuellement par ces deux itérateurs. Dès que l'un d'entre-eux ne produit plus d'entiers, `AddingIterator` n'en produit plus non plus.

Une fois écrite, la classe `AddingIterator` peut s'utiliser ainsi :

```
Iterator<Integer> i1 =
  Arrays.asList(1, 2, 3, 4, 5).iterator();
Iterator<Integer> i2 =
  Arrays.asList(6, 7, 8, 9).iterator();
Iterator<Integer> iSum = new AddingIterator(i1, i2);
```

L'itérateur `iSum` ainsi défini produit les entiers 7, 9, 11 et 13, puisque $1 + 6 = 7$, $2 + 7 = 9$, $3 + 8 = 11$, $4 + 9 = 13$. L'entier 5 que l'itérateur `i1` pourrait encore produire est ignoré car l'itérateur `i2` ne peut produire que quatre entiers.

Partie 1 [1 point] Donnez le nom du patron de conception utilisé par la classe `AddingIterator`.
Réponse : _____

Partie 2 [2 points] Admettons qu'en plus des deux itérateurs `i1` et `i2` définis ci-dessus on en ait un troisième, `i3`, défini ainsi :

```
Iterator<Integer> i3 = Arrays.asList(3, 2, 1).iterator();
```

Peut-on, en utilisant uniquement la classe `AddingIterator` décrite ci-dessus, combiner les trois itérateurs `i1`, `i2` et `i3` en un itérateur unique `iSum3` dont les éléments soient la somme des éléments de ces itérateurs ?

Si oui, écrivez le code qui permet de construire cet itérateur `iSum3`. Si non, expliquez pourquoi cela n'est pas possible.
Réponse : _____

Partie 3 [1 point] Indépendemment de votre réponse à la question de la partie 2, donnez — dans l'ordre — les entiers qui devraient être produits par l'itérateur `iSum3`.
Réponse : _____

Partie 4 [6 points] Ecrivez la classe `AddingIterator`. Sa méthode `remove` doit simplement lever l'exception `UnsupportedOperationException`.

Réponse :

Partie 5 [2 points] L'addition n'est pas la seule opération que l'on peut imaginer vouloir utiliser pour combiner les éléments de deux itérateurs sur des entiers. Les autres opérations élémentaires (soustraction, multiplication, division entière, etc.) pourraient également être utiles, et plus généralement n'importe quelle fonction binaire sur les entiers.

Nommez le patron de conception que l'on devrait utiliser pour généraliser l'itérateur combiné `AddingIterator` à une opération de combinaison quelconque.

Réponse : _____

Vecteurs creux [9 points]

Un *vecteur creux* est un vecteur dont la plupart des composantes sont nulles. De tels vecteurs peuvent être modélisés par la classe instanciable et immutable `SparseVector`, dont le squelette est donné ci-dessous.

```
public final class SparseVector {
  private final Map<Integer, Double> nonZeroElems;
  private final int size;

  public SparseVector(Map<Integer, Double> nonZeroElems,
                      int size) { à faire }
  public int size() { à faire }
  public double get(int index) { à faire }
}
```

Le champ `nonZeroEntries` est une table associative associant la valeur des éléments non nuls du vecteur à leur indice (qui débute à 0 !), tandis que `size` contient la taille totale du vecteur.

Par exemple, le vecteur creux $\vec{x}$ de 50 éléments dont tous les éléments x_0 à x_{49} sont nuls sauf x_5 qui vaut 1.2 et x_{22} qui vaut 1.5 est représenté par une instance de `SparseVector` dont le champ `nonZeroElems` est une table associant la valeur 1.2 à la clef 5 et la valeur 1.5 à la clef 22, et dont le champ `size` vaut 50.

Partie 1 [5 points] Ecrivez le corps du constructeur, qui prend en arguments une table associative spécifiant les éléments non nuls du vecteur, et une taille. Ce constructeur doit lever l'exception `IllegalArgumentException` si la taille reçue est invalide (c-à-d strictement négative) ou si l'un des indices de la table `nonZeroElems` est invalide, c-à-d négatif ou supérieur ou égal à `size`.

Notez qu'il n'est *pas* demandé de lever une exception si l'une des valeurs de la table `nonZeroElems` est nulle.

N'oubliez pas que la classe `SparseVector` est immutable !

Réponse :

Partie 2 [1 point] Ecrivez le corps de la méthode `size`.

Réponse :

Partie 3 [3 points] Ecrivez le corps de la méthode `get`, qui retourne l'élément d'indice donné ou lève l'exception `IllegalArgumentException` si celui-ci est invalide. Souvenez-vous que les indices commencent à 0 !

Réponse :

Formulaire

Ce formulaire présente toutes les parties de la bibliothèque standard Java dont vous avez besoin pour cet examen. Notez que de nombreuses méthodes inutiles ont été omises afin de ne pas encombrer la présentation.

Interface `Map`

L'interface `java.util.Map` représente les tables associatives. Elle est implémentée, entre autres, par la classe `HashMap`.

```
interface Map<K,V> {
  // Retourne la valeur associée à key, ou null s'il n'y en a aucune.
  V get(K key);

  // Retourne vrai si et seulement si la table contient la clef key.
  boolean containsKey(K key);

  // Retourne l'ensemble des clefs de la table.
  Set<K> keySet();
}
```

Les classes qui implémentent cette interface offrent toutes un constructeur de copie (c-à-d un constructeur qui prend une valeur de type `Map<K, V>` en argument et l'utilise pour initialiser la table associative construite).

Interface `Set`

L'interface `java.util.Set` représente les ensembles.

```
interface Set<E> extends Iterable<E> {
  // Retourne un itérateur sur les éléments de l'ensemble.
  Iterator<E> iterator();
}
```

Interface `Iterator`

L'interface `java.util.Iterator` représente les itérateurs.

```
interface Iterator<E> {
  // Retourne vrai ssi cet itérateur peut encore livrer des éléments.
  boolean hasNext();

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

  // Supprime la valeur retournée par le dernier appel à next.
  void remove();
}
```

Classe `InputStream`

La classe héritable `java.io.InputStream` représente les flots d'entrée.

```
abstract class InputStream {
  // Lit et retourne le prochain octet du flot, sous la forme d'un entier
  // compris entre 0 et 255 (inclus), ou -1 si la fin du flot a été atteinte.
  // Lève l'exception IOException en cas d'erreur.
  abstract int read() throws IOException;

  // Ferme le flot.
  void close() throws IOException;
}
```