Pratique de la programmation orientée-objet
Examen intermédiaire

13 avril 2016

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 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 Rotation de chaîne [39 points]

Pour mémoire, la transformée de Burrows-Wheeler d'une chaîne s'obtient en :

1. calculant toutes les rotations de cette chaîne,
2. triant ces rotations par ordre alphabétique,
3. extrayant la dernière *colonne* des rotation triées, et la position de la chaîne originale.

Si les rotations de la chaîne sont effectivement construites, leur stockage consomme $O(n^2)$ unités de mémoire, où n est la longueur de la chaîne. Cela peut être considérable si la chaîne est longue, p.ex. le texte complet d'un livre.

Pour représenter les rotations de manière plus compacte, il est possible d'utiliser le concept de vue. L'idée est de définir une classe offrant une vue d'une chaîne après rotation, sans réellement construire la rotation en question. Cette vue est spécifiée par la chaîne (avant rotation) et l'index du premier caractère de la rotation dans cette chaîne.

Par exemple, l'extrait de programme ci-dessous montre comment utiliser une telle classe (`RotatedString`) pour représenter toutes les rotations de la chaîne `EPFL`.

```
String s = ”EPFL”;
RotatedString r0 = new RotatedString(s, 0);  // EPFL
RotatedString r1 = new RotatedString(s, 1);  // PFLE
RotatedString r2 = new RotatedString(s, 2);  // FLEP
RotatedString r3 = new RotatedString(s, 3);  // LEPF
```

Le but de cet exercice est de compléter la définition de la classe `RotatedString` ci-dessous, en fonction des instructions données plus bas.

```
public final class RotatedString
    implements Comparable<RotatedString> {
  private final String string; // chaîne avant rotation
  private final int index;     // index du premier car. de la rotation

  public RotatedString(String string, int index) { à faire }
  public String string() { à faire }
  public int index() { à faire }
  public char charAt(int i) { à faire }

  @Override
  public int compareTo(RotatedString that) { à faire }
  // autres redéfinitions (equals, hashCode, toString)
}
```

Partie 1 [5 points] Ecrivez le constructeur et les deux accesseurs, `string` et `index`. Le constructeur doit lever une `IndexOutOfBoundsException` si l'index est invalide pour la chaîne, c-à-d négatif ou supérieur ou égal à sa longueur.

Notez que la méthode `string` retourne la chaîne *avant* rotation, c'est-à-dire celle passée au constructeur.

Réponse :

Partie 2 [4 points] Ecrivez la méthode `charAt`, qui retourne le caractère à l'index donné de la chaîne *après* rotation, ou lève une `IndexOutOfBoundsException` si l'index est invalide. Par exemple `r2.charAt(0)` retourne le caractère `F`, étant donné les définitions plus haut.

Notez que votre méthode `charAt` ne doit pas appeler la méthode `toString` définie plus bas.

Réponse :

Partie 3 [8 points] Ecrivez les méthodes `equals` et `hashCode`, qui redéfinissent celles de la classe `Object`.

La méthode `equals` doit retourner vrai si et seulement si l'objet qu'elle reçoit est aussi une instance de `RotatedString` et les chaînes représentées par les deux rotations sont identiques, après rotation, caractère par caractère.

La méthode `hashCode` doit calculer son résultat en utilisant la même formule que celle utilisée par la classe `String` en Java, à savoir :

$$h = 31^{n-1}c_0 + 31^{n-2}c_1 + \cdots + 31\,c_{n-2} + c_{n-1}$$

où h est la valeur de hachage, c_i est le i^e caractère de la chaîne *après* rotation et n est la longueur de la chaîne. Souvenez-vous qu'en Java les caractères sont des entiers, et peuvent être manipulés comme tels.

Notez que vos méthodes `equals` et `hashCode` peuvent appeler n'importe quelle autre méthode de la classe `RotatedString`, à l'exception de `toString`.

Réponse :

Partie 4 [12 points] Ecrivez la méthode `compareTo`, qui compare deux chaînes après rotation, en ordre lexicographique, et retourne un entier négatif si la première est strictement plus petite que la seconde, zéro si elles sont égales, et un entier positif sinon.

Pour mémoire, l'ordre lexicographique sur les chaînes est défini ainsi :

- si les deux chaînes sont de mêmes longueurs et leurs caractères sont égaux deux à deux, alors les chaînes sont égales,

- sinon, si l'une des deux chaînes est un préfixe (strict) de l'autre, alors la plus courte est plus petite que l'autre,
- sinon, les chaînes diffèrent à un index donné, et la chaîne dont le caractère à cet index est le plus petit est plus petite que l'autre.

Par exemple, étant donné les définitions plus haut, `r1.compareTo(r2)` retourne un entier positif, car les chaînes diffèrent en leur premier caractère, et `P` est plus grand que `F`.

Pour comparer les caractères entre eux, utilisez la méthode statique `compare` de `Character`, et pour comparer les longueurs entre elles, utilisez la méthode statique `compare` de `Integer`. Chacune de ces méthodes prend deux arguments (caractères ou entiers `int`) et retourne un entier négatif si le premier est strictement plus petit que le second, zéro s'ils sont égaux, et un entier positif sinon.

Notez que votre méthode `compareTo` ne doit pas appeler la méthode `toString` définie plus bas.

Réponse :

Partie 5 [6 points] Ecrivez la méthode `toString`, qui redéfinit celle de la classe `Object` et retourne la chaîne après rotation. Par exemple, étant donné les définitions plus haut, `r2.toString()` retourne la chaîne `FLEP`.

Réponse :

Partie 6 [4 points] Pour simplifier la méthode `hashCode`, il est tentant de la définir ainsi :

```
@Override
int hashCode() { return string.hashCode(); }
```

Quel est le problème de cette définition ? Justifiez votre réponse.

Réponse :

2 Collisions de hachage [18 points]

Une manière d'évaluer la qualité d'une fonction de hachage est de déterminer si elle produit un grand nombre de collisions de hachage pour des valeurs typiques. Pour mémoire, une collision de hachage se produit lorsque deux objets différents ont la même valeur de hachage. En Java, cela est vrai lorsque la condition suivante est satisfaite pour deux objets `o1` et `o2` :

```
!o1.equals(o2) && o1.hashCode() == o2.hashCode()
```

Par exemple, la fonction de hachage de la classe `String` en Java (donnée dans l'exercice précédent) produit une collision de hachage pour les mots *gué* et *gym*, les deux ayant la valeur de hachage 102 843.

Pour trouver la totalité des collisions de hachage produites en Java par la méthode `hashCode` de la classe `String` dans une collection de chaînes, écrivez le corps de la méthode suivante :

```
List<Set<String>> hashCollisions(Collection<String> strings)
```

Cette méthode retourne une liste contenant tous les ensembles de plus d'une chaîne de la collection donnée ayant la même valeur de hachage. Par exemple, sachant que le mot *col* a la valeur de hachage 98 688 en Java, l'appel

```
hashCollisions(Arrays.asList("gym", "col", "gué"));
```

retourne une liste contenant un seul ensemble, dont les éléments sont les chaînes *gym* et *gué*.

Votre mise en œuvre de `hashCollisions` doit avoir une complexité de $O(n)$ où n est le nombre de chaînes dans la collection fournie. La liste d'ensembles qu'elle retourne doit satisfaire les conditions suivantes :

1. tous les ensembles contiennent au moins deux éléments,
2. tous les éléments d'un ensemble ont la même valeur de hachage,
3. aucune paire d'éléments provenant de deux ensembles différents n'a la même valeur de hachage,
4. si deux éléments de la collection fournie ont la même valeur de hachage, ils appartiennent tous deux à l'un des ensembles.

Réponse :

(suite au verso)

(suite)

3 Tri par base [18 points]

Le tri par base (*radix sort*) est une technique de tri qui permet de trier efficacement des listes d'entiers. Elle fonctionne par itération sur les chiffres des entiers à trier, du moins significatif (celui des unités) au plus significatif. A chaque itération, les éléments de la liste sont triés en fonction du chiffre courant, en préservant leur ordre relatif. Le résultat final est une liste triée.

Par exemple, la liste [27, 71, 17, 1, 77, 11, 24] peut être triée de cette manière en deux itérations, car son plus grand élément a deux chiffres (en base 10).

Dans un premier temps, ses éléments sont triés par leur chiffre le moins significatif (les unités), en préservant leur ordre relatif. Cela se fait aisément en partitionnant la liste en fonction du dernier chiffre, pour obtenir :

- les éléments se terminant en 1 : [71, 1, 11],
- les éléments se terminant en 4 : [24],
- les éléments se terminant en 7 : [27, 17, 77].

La concaténation de ces sous-listes dans l'ordre donne [71, 1, 11, 24, 27, 17, 77]. Cette liste partiellement triée est triée une seconde fois en utilisant la même technique, mais cette fois en fonction du chiffre le plus significatif de ses éléments. Cela donne :

- les éléments commençant par 0 : [1],
- les éléments commençant par 1 : [11, 17],
- les éléments commençant par 2 : [24, 27],
- les éléments commençant par 7 : [71, 77].

La concaténation de ces sous-listes produit la liste finale triée : [1, 11, 17, 24, 27, 71, 77].

Partie 1 [14 points] Ecrivez le corps de la méthode `sort` ci-dessous, qui trie la liste d'entiers qu'on lui passe par ordre croissant, selon la technique décrite ci-dessus.

```
void sort(List<Integer> ints)
```

Cette méthode peut supposer que tous les nombres de la liste reçue sont positifs. Elle doit directement modifier la liste reçue en argument plutôt que d'en retourner une nouvelle.

Rappel : en Java, la plus grande valeur de type `int` est 2 147 483 647 et comporte dix chiffres (en base 10).

Réponse :

(suite au verso)

(suite)

Partie 2 [4 points] La technique du tri par base a été illustrée plus haut au moyen de la base 10, mais n'importe quelle base convient. Bien entendu, lorsque la base augmente, le nombre de chiffres différents augmente également, mais la longueur des nombres diminue. Dès lors, une augmentation de la base se traduit par une augmentation du nombre de sous-listes dans les partitions successives, mais une réduction du nombre d'itérations.

Décrivez les modifications à apporter à votre méthode de tri pour qu'elle utilise la base 256 plutôt que la base 10.

Réponse :

Formulaire

Ce formulaire présente toutes les parties de la bibliothèque Java nécessaires à cet examen. De nombreuses méthodes inutiles ont été omises pour alléger la présentation.

Classe `StringBuilder`

La classe `java.lang.StringBuilder` est un bâtisseur pour les chaînes de caractères.

```
public class StringBuilder {
  // Ajoute les caractères de la chaîne s compris entre l'index f (inclusif)
  // et t (exclusif) à la chaîne en cours de construction et retourne this.
  public StringBuilder append(String s, int f, int t);

  // Retourne une nouvelle chaîne avec le contenu ajouté jusqu'à présent.
  public String toString();
}
```

Classe `String`

La classe `java.lang.String` représente les chaînes de caractères.

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

  // Retourne le caractère à l'index i.
  public char charAt(int i);
}
```

Classe `Math`

La classe `Math`, non instanciable, contient des méthodes statiques représentant des fonctions mathématiques courantes.

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

(suite au verso)

Interface `Collection`

L'interface `java.util.Collection` représente une collection de valeurs. Elle est implémentée, entre autres, par les classes `ArrayList`, `LinkedList`, `HashSet` et `TreeSet`.

```
public interface Collection<E> {
  // Retourne la taille (nombre d'éléments) de la collection.
  public int size();

  // Vide la collection en supprimant tous ses éléments.
  public void clear();

  // Ajoute l'élément e à la collection et retourne vrai ssi elle a été modifiée
  // en conséquence.
  public boolean add(E e);

  // Ajoute tous les éléments de c à la collection et retourne vrai ssi elle a été
  // modifiée en conséquence.
  public boolean addAll(Collection<E> c);
}
```

Les classes implémentant cette interface offrent toutes un constructeur de copie (c-à-d un constructeur qui prend une valeur de type `Collection<E>` en argument et l'utilise pour initialiser la collection construite).

Interface `Map`

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

```
public interface Map<K, V> {
  // Retourne vrai ssi la table contient la clef key.
  public boolean containsKey(K key);

  // Associe la valeur value à la clef key. Retourne la valeur qui était associée
  // à la clef, ou null s'il n'y en avait pas.
  public V put(K key, V value);

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

  // Si la clef key n'est pas présente dans la table, appelle la fonction f en la lui
  // passant, associe son résultat à key dans la table et le retourne.
  // Sinon, retourne la valeur associée à la clef.
  public V computeIfAbsent(K key, Function<K,V> f);

  // Retourne une vue sur la collection des valeurs.
  public Collection<V> values();
}
```