Pratique de la programmation orientée-objet
Examen final

7 août 2020

Indications :

— l'examen dure de 16h15 à 20h00,

— 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 Ensembles définis par compréhension [18 points]

En mathématiques, il est fréquent de définir un ensemble « par compréhension », c-à-d en spécifiant les conditions qu'un élément doit satisfaire pour appartenir à cet ensemble. Par exemple, l'ensemble Y des entiers strictement positifs mais inférieurs ou égaux à 2020 peut se définir par compréhension ainsi :

$$Y = \{n \in \mathbb{Z} \mid 0 < n \leq 2020\}$$

Il est possible de définir des ensembles de manière très similaire en Java, en introduisant une interface fonctionnelle, nommée p.ex. `FunSet`, pour ce nouveau type d'ensembles. L'unique méthode abstraite de cette interface est `contains`, qui détermine si un élément appartient à l'ensemble ou non :

```
@FunctionalInterface
public interface FunSet<E> {
  boolean contains(E elem);
}
```

Grâce à cette interface, l'ensemble Y ci-dessus peut être défini au moyen d'une lambda, et la notation rappelle agréablement la notation mathématique :

```
FunSet<Integer> y = n -> 0 < n && n <= 2020;
```

Déterminer si un entier appartient à l'ensemble peut se faire en appelant sa méthode `contains` :

```
System.out.println(y.contains(1995)); // imprime true
System.out.println(y.contains(2050)); // imprime false
```

Le but de cet exercice est de compléter la définition de `FunSet` en y ajoutant des méthodes statiques pour créer des ensembles et des méthodes par défaut pour les manipuler.

Partie 1 [9 points] Ajoutez à `FunSet` les définitions surchargées suivantes d'une méthode statique nommée `of` :

1. une ne prenant aucun argument et retournant un ensemble `FunSet` vide,
2. une prenant un seul argument et retournant un ensemble `FunSet` contenant uniquement cet élément,
3. une prenant un ensemble Java « standard » — c'est-à-dire une valeur de type `java.util.Set<E>` pour un type `E` quelconque — en argument et retournant un ensemble `FunSet` contenant exactement les mêmes éléments que lui.

Par exemple, la seconde variante de cette méthode devrait être utilisable ainsi pour créer un ensemble ne contenant que l'entier 2020 :

```
FunSet<Integer> annusHorribilis = FunSet.of(2020);
```

Notez que toutes ces méthodes sont très courtes, mais faites attention à leur donner au besoin le(s) bon(s) paramètre(s) de type.

Réponse :

Partie 2 [9 points] Ajoutez à `FunSet` les méthodes par défaut suivantes :

1. `complement`, qui retourne le complément de l'ensemble auquel on l'applique,
2. `union`, qui retourne l'ensemble `FunSet` résultant de l'union du récepteur (`this`) et d'un second ensemble `FunSet` passé en argument,
3. `intersection`, qui retourne l'ensemble `FunSet` résultant de l'intersection du récepteur (`this`) et d'un second ensemble `FunSet` passé en argument,
4. `difference`, qui retourne l'ensemble `FunSet` résultant de la différence entre le récepteur (`this`) et un second ensemble `FunSet` passé en argument.

Par exemple, la dernière de ces méthodes devrait être utilisable ainsi pour obtenir un ensemble équivalent à $\{n \in \mathbb{Z} \mid 0 < n \leq 2019\}$:

```
FunSet<Integer> goodYears = y.difference(annusHorribilis);
```

Réponse :

2 Matrices [75 points]

En mathématiques, une *matrice* $m \times n$, nommée A ci-dessous, est un tableau bi-dimensionnel de nombres $a_{0,0}$, $a_{0,1}$, ... appelés ses *coefficients*. Ceux-ci sont organisés en m lignes de n colonnes chacune, ainsi :

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix}$$

Notez que les coefficients sont numérotés à partir de 0, comme habituellement en informatique, et pas à partir de 1, comme habituellement en mathématiques.

Bien qu'ils soient généralement organisés en deux dimensions comme ci-dessus, les coefficients d'une matrice peuvent également être organisés en une seule dimension, dans l'ordre nommé *row-major* en anglais. Dans cet ordre, les coefficients sont organisés linéairement, une ligne après l'autre, ainsi :

$$a_{0,0} \quad a_{0,1} \quad \ldots \quad a_{0,n-1} \quad a_{1,0} \quad a_{1,1} \quad \ldots \quad a_{1,n-1} \quad \ldots \quad a_{m-1,0} \quad a_{m-1,1} \quad \ldots \quad a_{m-1,n-1}$$

L'interface `Matrix` ci-dessous représente une matrice en Java, et l'interface imbriquée `Builder` représente un bâtisseur de matrice. Le but de cet exercice est de compléter l'interface `Matrix` et d'écrire plusieurs classes l'implémentant, de même qu'une classe implémentant l'interface `Builder`.

```
public interface Matrix {
  public abstract int rows();
  public abstract int columns();

  public abstract double getRowMajor(int i);
  public default double get(int r, int c) { à faire }

  public default Matrix transposed() {
    return new TransposedMatrix(this);
  }

  public static interface Builder {
    public abstract Builder set(int r, int c, double v);
    public abstract Matrix build();
  }
}
```

Les méthodes `rows` et `columns` retournent respectivement le nombre de lignes et de colonnes de la matrice.

La méthode `getRowMajor` retourne le coefficient à l'index `i` dans l'ordre *row-major*. La méthode `get` retourne le coefficient à la ligne `r` et à la colonne `c`. Ces deux méthodes lèvent `IndexOutOfBoundsException` si l'un des index qu'elles reçoivent est invalide.

La méthode `transposed` est décrite plus loin.

Partie 1 [6 points] Écrivez le corps de la méthode `get`, qui prend un index de ligne `r` et un index de colonne `c` et retourne le coefficient à cette position, ou lève `IndexOutOfBoundsException` (!) si l'un de ces index est invalide.

Souvenez-vous que les index commencent à 0, et notez que cette méthode doit être exprimée en termes d'autres méthodes de l'interface `Matrix` — `rows`, `columns` et `getRowMajor`.

Réponse :

Partie 2 [11 points] Écrivez la classe `DenseMatrix`, instanciable et immuable, implémentant l'interface `Matrix` et représentant une matrice dense, c-à-d une dont peu de coefficients (voir aucun) sont nuls.

Le constructeur de cette classe doit prendre trois arguments qui sont, dans l'ordre, le nombre de lignes de la matrice, son nombre de colonnes, et le tableau de ses coefficients en ordre *row-major*. Par exemple, le code suivant :

```
new DenseMatrix(2, 3, new double[]{1.2, 3, 4, 5, 6.7, 8})
```

doit créer une instance de `DenseMatrix` représentant la matrice suivante :

$$\begin{pmatrix} 1.2 & 3 & 4 \\ 5 & 6.7 & 8 \end{pmatrix}$$

Le constructeur doit lever une `IllegalArgumentException` si le nombre de lignes ou de colonnes n'est pas strictement positif, ou si le tableau n'a pas la bonne longueur.

En plus du constructeur, `DenseMatrix` ne doit contenir que les mises en œuvre des méthodes abstraites de `Matrix` — `rows`, `columns` et `getRowMajor` —, ainsi que les attributs qui leur sont nécessaires. Souvenez-vous que `getRowMajor` doit lever une `IndexOutOfBoundsException` (!) si l'index qu'elle reçoit est invalide.

Réponse :

Partie 3 [18 points] Écrivez la classe `SparseMatrix`, instanciable et immuable, implémentant l'interface `Matrix` et représentant une matrice creuse, c-à-d une dont de nombreux coefficients sont nuls. Afin d'économiser de la mémoire, cette classe ne stocke que les coefficients non nuls de la matrice qu'elle représente.

Le constructeur de `SparseMatrix` doit prendre quatre arguments qui sont, dans l'ordre, le nombre de lignes de la matrice, son nombre de colonnes, le tableau des index *row-major* des coefficients non nuls (triés par ordre croissant, sans doublons), et le tableau de ces coefficients non nuls. Par exemple, le code suivant :

```
new SparseMatrix(2, 3, new int[]{0, 5}, new double[]{1.2, 3})
```

doit créer une instance de `SparseMatrix` représentant la matrice suivante :

$$\begin{pmatrix} 1.2 & 0 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

Le constructeur doit lever une `IllegalArgumentException` si le nombre de lignes ou de colonnes n'est pas strictement positif, si le tableau des index n'est pas trié ou contient un index invalide ou dupliqué, si le tableau des coefficients non nuls contient un 0, ou si les deux tableaux n'ont pas la même longueur.

En dehors de ce constructeur, `SparseMatrix` ne doit contenir que les mises en œuvre des méthodes abstraites de `Matrix`, ainsi que les attributs qui leur sont nécessaires.

Pour des raisons de performances, votre mise en œuvre de `getRowMajor` doit utiliser la méthode statique `binarySearch` de la classe `Arrays` pour rechercher l'index dans le tableau des coefficients non nuls. Cette méthode est décrite dans le formulaire à la fin de l'examen.

Réponse :

Partie 4 [13 points] Écrivez la classe `TransposedMatrix`, instanciable et immuable, implémentant l'interface `Matrix` et représentant la transposée d'une matrice.

Pour mémoire, la transposée d'une matrice $m \times n$ nommée A se note A^T et est une matrice $n \times m$ dont les coefficients $b_{i,j}$ sont donnés par : $b_{i,j} = a_{j,i}$

Pour des raisons de performances, `TransposedMatrix` ne doit pas stocker les coefficients ou le nombre de lignes ou de colonnes de la matrice transposée, mais uniquement une référence vers la matrice originale passée à son constructeur — voir la méthode par défaut `transposed` de l'interface `Matrix` plus haut.

Comme les autres mises en œuvre de `Matrix`, `TransposedMatrix` doit contenir des mises en œuvre des méthodes abstraites de `Matrix`. D'autre part, elle doit redéfinir la méthode `transpose` afin de tirer parti du fait que la transposée de la transposée d'une matrice est la matrice elle-même, c-à-d que :

$$\left(A^T\right)^T = A$$

Réponse :

Partie 5 [1 point] Quel est le nom du patron de conception utilisé par la classe `TransposedMatrix` ? _____

Partie 6 [26 points] Écrivez la classe instanciable `MatrixBuilder`, qui implémente l'interface `Matrix.Builder` et représente un bâtisseur de matrice.

Le constructeur de `MatrixBuilder` doit prendre exactement deux arguments qui sont, dans l'ordre, le nombre de lignes et le nombre de colonnes de la matrice à bâtir. Il doit lever une `IllegalArgumentException` s'ils ne sont pas les deux strictement positifs. Initialement, tous les coefficients de la matrice doivent valoir 0.

La méthode `set` doit attribuer la valeur `v` au coefficient se trouvant à la ligne `r` et à la colonne `c` de la matrice en cours de construction, ou lever une exception de type `IndexOutOfBoundsException` (!) si l'un des index est invalide. Elle doit retourner le bâtisseur lui-même.

Finalement, la méthode `build` doit retourner une nouvelle matrice avec les coefficients stockés actuellement dans le bâtisseur, qui doit être une instance de `DenseMatrix` si strictement plus de 50% des coefficients sont différents de 0, et une instance de `SparseMatrix` sinon. Par exemple, le code suivant :

```
Matrix m = new MatrixBuilder(2, 3)
  .set(1, 2, 3)
  .set(0, 0, 1.2)
  .build();
```

doit construire une matrice identique à celle de la partie 3. En particulier, il doit aussi s'agir d'une instance de `SparseMatrix`.

Réponse :

Réponse (suite) :

3 Nombres à virgule flottante [23 points]

En Java, un nombre à virgule flottante de type `float` est formé de trois composantes encodées chacune au moyen d'un nombre fixe de bits : le signe s (1 bit), l'exposant e (8 bits) et la mantisse (ou significande) m (23 bits). Ces trois composantes sont empaquetées dans une valeur unique de 32 bits, ainsi :

$$\underbrace{b_{31}}_{s}\ \underbrace{b_{30}\, b_{29} \ldots b_{24}\, b_{23}}_{e}\ \underbrace{b_{22}\, b_{21} \ldots b_1\, b_0}_{m}$$

où b_i représente le bit d'index i. Le nombre n représenté par ces composantes est :

$$n = (-1)^s \times 1.m \times 2^{e-127}$$

Par exemple, le nombre de type `float` dont la valeur est -2.5 est représenté par les 32 bits suivants (pour faciliter la lecture, des espaces fines ont été insérées entre chacune des composantes) :

`1 10000000 01000000000000000000000`

car :

$$-1^1 \times 1.01000000000000000000000_2 \times 2^{10000000_2 - 127} = -1 \times 1.25 \times 2^1 = -2.5$$

(Souvenez-vous que lorsqu'un nombre n est écrit n_2, cela signifie qu'il est représenté en base 2, sinon il est en base 10. Ainsi, $10000000_2 = 128$.)

Certaines valeurs des différentes composantes sont réservées pour représenter des « nombres » spéciaux :

— si $e = m = 0$, le nombre représenté est $+0$ si $s = 0$, -0 sinon,

— si $e = 11111111_2$ et $m = 0$, le nombre représenté est $+\infty$ si $s = 0$, $-\infty$ sinon,

— si $e = 11111111_2$ et $m \neq 0$, la valeur représente une erreur et pas un nombre (*not a number* en anglais, abrégé NaN).

Notez que cela implique qu'il y a deux valeurs différentes pour 0, une positive et une négative, ainsi qu'un très grand nombre de valeurs NaN. Ces dernières sont produites en cas d'erreur, p.ex. lorsqu'on essaie de diviser 0 par lui-même.

Il est possible d'obtenir les 32 bits représentant une valeur de type `float` sous la forme d'une valeur de type `int` au moyen de la méthode `floatToIntBits` de la classe `Float`. Par exemple, l'expression suivante :

```
Integer.toBinaryString(Float.floatToIntBits(-2.5f))
```

retourne la chaîne `11000000001000000000000000000000`, comme attendu.

L'opération inverse peut être effectuée au moyen de la méthode `intBitsToFloat` de la classe `Float`. Par exemple, l'expression suivante :

```
Float.intBitsToFloat(0b11000000001000000000000000000000)
```

retourne la valeur `-2.5`, de type `float`, comme attendu.

Le but de cet exercice est d'écrire plusieurs méthodes travaillant sur des valeurs de type `float` en manipulant directement les bits de leur représentation.

Partie 1 [12 points] Écrivez trois méthodes nommées `isZero`, `isInfinite` et `isNaN` qui prennent un seul argument de type `float` et retournent vrai (`true`) si et seulement si cette valeur est ± 0 (`isZero`), $\pm\infty$ (`isInfinite`) ou une valeur NaN quelconque (`isNaN`).

La seule opération que ces méthodes ont le droit d'effectuer sur leur argument est d'appeler la méthode `floatToIntBits` dessus, après quoi elles ne doivent plus l'utiliser et travailler uniquement sur la valeur retournée par cet appel. Par exemple, `isZero` doit avoir la forme suivante :

```
boolean isZero(float f) {
  int i = floatToIntBits(f);
  // reste du code, qui n'utilise jamais f
}
```

Suggestion : utilisez les opérateurs bit à bit (`~`, `&`, `|`, `^`, `<<`, `>>` et `>>>`) pour analyser les bits de `i`.

Réponse :

Partie 2 [5 points] Écrivez une méthode nommée `abs` qui prend un seul argument de type `float` et retourne sa valeur absolue. Si cet argument est un NaN, il doit être retourné tel quel.

Comme les méthodes de la partie 1, votre méthode `abs` doit travailler directement sur la représentation de son argument sous la forme d'une valeur de 32 bits, et n'a pas le droit d'effectuer d'opérations en virgule flottante. Elle peut par contre appeler `floatToIntBits`, `intBitsToFloat` ou des méthodes de la partie 1.

Réponse :

Partie 3 [6 points] Écrivez une méthode nommée `signum` qui prend un seul argument de type `float` et retourne la fonction signe de cet argument, à savoir :

— l'argument lui-même s'il s'agit d'un NaN ou de ± 0,

— sinon, si cet argument est négatif, la valeur de type `float` représentant -1,

— sinon, la valeur de type `float` représentant $+1$.

Une fois encore, cette méthode doit travailler directement sur la représentation de son argument sous la forme d'une valeur de 32 bits, et son résultat doit être calculé au moyen de `intBitsToFloat`. Elle ne peut ni utiliser des constantes de type `float` ou `double` (p.ex. `1f` ou `1.0`), ni dépendre du fait que Java convertit automatiquement les valeurs de type `int` en `float` (p.ex. `return 1` est interdit).

Réponse :

4 Partition d'un ensemble [34 points]

En mathématiques, une *partition* d'un ensemble est une répartition de ses éléments dans différents sous-ensembles telle que chaque élément appartient à un sous-ensemble exactement.

Par exemple, l'ensemble de toutes les villes italiennes peut être partitionné de sorte à ce que deux villes appartiennent au même sous-ensemble si et seulement si elles sont reliées par la route. Dans cette partition, Milano, Firenze, Roma, Napoli et Venezia, qui sont toutes sur le continent, appartiennent à un premier sous-ensemble ; Palermo et Siracusa, toutes deux sur l'île de Sicile, appartiennent à un second sous-ensemble ; et Cagliari et Olbia, toutes deux sur l'île de Sardaigne, appartiennent à un troisième sous-ensemble.

Une caractéristique d'une partition est que, comme chaque élément appartient à exactement un sous-ensemble, on peut identifier un sous-ensemble en choisissant arbitrairement un de ses éléments comme *représentant*. Pour les villes italiennes, le représentant du premier sous-ensemble pourrait par exemple être Roma, celui du second Palermo, et celui du troisième Olbia. Cela dit, n'importe quel autre choix conviendrait, tant et aussi longtemps qu'un élément exactement est choisi comme représentant du sous-ensemble qui le contient.

Le but de cet exercice est d'écrire une classe Java pour représenter une partition d'un ensemble. Cette classe ne doit offrir que deux opérations, mais elles doivent être efficaces :

1. déterminer si deux éléments appartiennent au même sous-ensemble,

2. joindre deux sous-ensembles en un seul.

Dans notre exemple, la première opération serait utile pour savoir si deux villes italiennes sont reliées par la route, tandis que la seconde pourrait être utile pour mettre à jour la partition si un pont reliant le continent à la Sicile était construit.

Une manière de représenter une partition consiste à lier les éléments entre eux de sorte à ce qu'en suivant ces liens en partant de n'importe quel élément, on finisse par arriver au représentant de l'ensemble auquel cet élément appartient. La figure 1 illustre comment cette technique pourrait être utilisée pour représenter la partition des villes italiennes, en faisant l'hypothèse que les représentants des sous-ensembles sont ceux mentionnés plus haut. Les flèches symbolisent les liens entre les éléments, et les représentants se distinguent par le fait qu'aucune flèche ne part d'eux.

Avec une telle représentation, le représentant d'un sous-ensemble peut être obtenu simplement en suivant les liens. Cela permet la mise en œuvre efficace des deux opérations mentionnées précédemment :

1. déterminer si deux éléments appartiennent au même sous-ensemble se fait en cherchant leurs représentants puis en regardant s'ils sont égaux,

2. joindre deux sous-ensembles se fait en ajoutant un unique lien allant de l'un à l'autre de leurs représentants.

Par exemple, dans la figure 1, déterminer si Palermo et Napoli sont connectées par la route se fait en cherchant les représentants de leurs sous-ensembles respectifs (Palermo et Roma) et en regardant s'ils sont égaux. Joindre les sous-ensembles des villes continentales et siciliennes se fait en ajoutant un lien allant de Palermo à Roma (ou de Roma à Palermo).

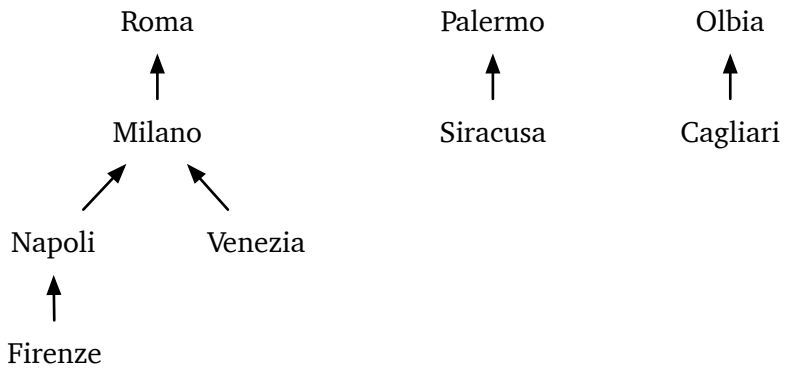

Fig. 1 : Une représentation possible de la partition des villes italiennes

Partie 1 [18 points] Écrivez la classe `Partition`, instanciable et générique, représentant une partition au moyen de la technique décrite ci-dessus. À sa création, la partition doit être maximale, dans le sens où chaque élément doit se trouver dans son propre sous-ensemble, dont il est le représentant. En d'autres termes, aucun lien ne doit initialement exister entre deux éléments, ces liens étant créés au moyen de la méthode `join` décrite ci-dessous.

Votre classe `Partition` doit offrir deux méthodes publiques : `inSameSubset`, qui prend deux éléments et retourne vrai s'ils appartiennent au même sous-ensemble de la partition, et `join`, qui prend deux éléments et modifie la partition de manière à ce que leurs sous-ensembles soient joints, et ne retourne rien (`void`).

En plus de ces deux méthodes publiques, votre classe doit également contenir une méthode privée nommée `representative` retournant le représentant du sous-ensemble contenant l'élément qu'on lui passe en argument. Cette méthode simplifiera votre code et est nécessaire pour la partie 2.

Le test JUnit suivant, qui doit s'exécuter avec succès, illustre l'utilisation de la classe `Partition` pour créer et interroger la partition des villes italiennes décrites plus haut. Notez que les villes sont représentées simplement par leur nom — une chaîne de caractères.

```
Partition<String> itCities = new Partition<>();

for (String c : List.of("Milano","Roma","Napoli","Venezia"))
  itCities.join("Firenze", c);
itCities.join("Palermo", "Siracusa");
itCities.join("Cagliari", "Olbia");

assertTrue(itCities.inSameSubset("Firenze", "Venezia"));
assertFalse(itCities.inSameSubset("Olbia", "Milano"));
```

Notez que votre mise en œuvre ne doit pas obligatoirement créer la même représentation de la partition que celle de la figure 1. N'importe quelle représentation équivalente est acceptable.

Suggestion : stockez les liens entre les éléments (les flèches de la figure 1) dans une table associative.

Réponse :

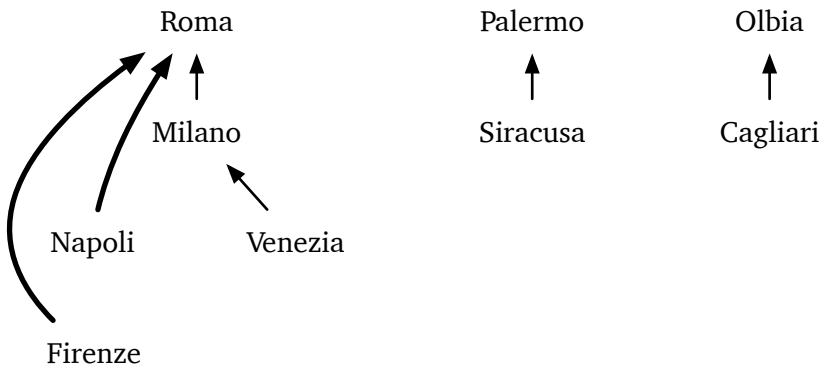

Fig. 2 : Partition de la figure 1 après compression du chemin partant de Firenze

Partie 2 [6 points] La classe `Partition` peut être rendue plus efficace au moyen d'une stratégie nommée *compression des chemins*. L'idée est que chaque fois que le représentant d'un élément est calculé en suivant les liens menant à lui, les liens de *tous* les éléments sur le chemin allant de l'élément à son représentant sont mis à jour pour faire directement référence au représentant.

Par exemple, étant donné la partition représentée à la figure 1, si le représentant de Firenze est calculé, alors les liens de tous les éléments situés sur le chemin allant de Firenze à Roma (c-à-d ceux de Firenze, Napoli et Milano) sont mis à jour pour référencer directement Roma. Cela est illustré à la figure 2, sur laquelle les liens ayant changé par rapport à la figure 1 sont en gras.

Montrez le code à ajouter à votre méthode `representative` pour que, à chaque appel, elle compresse le chemin partant de l'élément qui lui est donné en argument.

Réponse :

Partie 3 [10 points] Une seconde stratégie permettant d'améliorer l'efficacité de la classe `Partition` est de choisir judicieusement le nouveau représentant lorsque deux sous-ensembles sont joints.

Pour cela, on assigne à chaque élément dans la partition un entier non négatif nommé son *rang*, qui vaut initialement 0. Lorsque deux sous-ensembles sont joints, le rang de leurs représentants détermine lequel des deux sert de nouveau représentant, de la manière suivante :

— si les deux représentants ont un rang différent, celui de plus haut rang est choisi comme représentant du sous-ensemble joint, et aucun des rangs ne change,

— sinon, un des deux représentants — peu importe lequel — est choisi comme représentant du sous-ensemble joint, et son rang est incrémenté.

Montrez le code à ajouter à votre méthode `join` pour qu'elle détermine lequel des deux représentants existants utiliser pour le sous-ensemble joint en fonction de leurs rangs. Au besoin, vous pouvez ajouter un attribut à votre classe.

Suggestion : stockez le rang des éléments dans une autre table associative.

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.

Classe `Arrays`

La classe `java.util.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 int[] copyOf(int[] original, int newLength);
  static double[] copyOf(double[] original, int newLength);

  // Retourne l'index de l'élément key dans le tableau array, ou une valeur
  // (strictement) négative s'il ne s'y trouve pas.
  // La recherche est faite par dichotomie, en O(log n), et le tableau array
  // doit donc être trié par ordre croissant.
  static int binarySearch(int[] array, int key);
}
```

Interface `Set`

L'interface `java.util.Set` représente un ensemble. Elle est implémentée, entre autres, par les classes `HashSet` et `TreeSet`.

```
public interface Set<E> extends Iterable<E> {
  // Retourne vrai si et seulement si l'ensemble contient l'élément element.
  boolean contains(E element);
}
```

Interface `Map`

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

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

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

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

  // Associe la valeur value à la clef key.
  V put(K key, V value);
}
```

Classe `Integer`

La classe `java.lang.Integer` contient entre autres des méthodes statiques travaillant sur les entiers de type `int`.

```
public class Integer {
  // Retourne la représentation textuelle en base 2 de l'entier i.
  static String toBinaryString(int i);
}
```

Classe `Float`

La classe `java.lang.Float` contient entre autres des méthodes statiques travaillant sur les nombres à virgule flottante de type `float`.

```
public class Float {
  // Retourne la valeur de type int composée des mêmes 32 bits que f.
  static int floatToIntBits(float f);

  // Retourne la valeur de type float composée des mêmes 32 bits que i.
  static float intBitsToFloat(int i);
}
```