X2023 — CSC_41011
Contrôle classant / 13 novembre 2024 / 9h–12h

Tous documents de cours autorisés (polycopié, notes personnelles). Le dictionnaire papier est autorisé pour les FUI et FUI-FF. Les calculatrices, ordinateurs, tablettes et téléphones portables sont interdits.

L'énoncé est composé de deux problèmes indépendants, que vous pouvez traiter dans n'importe quel ordre (en revanche, merci de clairement numéroter les réponses). Vous pouvez, quand vous traitez une question, considérer comme déjà traitées les questions précédentes du même problème.

Le correcteur prêtera attention à la qualité de la rédaction et vous remercie d'avance d'écrire lisiblement. Merci également de numéroter vos copies en indiquant leur nombre total.

Les questions de code n'appellent aucune justification. Une indication du nombre de lignes attendues est donnée entre parenthèses. Quelques éléments de la bibliothèque standard Java sont rappelés à la fin du sujet. Ils ne sont pas forcément tous nécessaires. Vous pouvez utiliser librement tout autre élément de la bibliothèque standard Java.

1 Multiensembles et anagrammes

Un multiensemble est une variante d'ensemble où un même élément peut apparaître plusieurs fois. Ainsi, on note $\{\!\{ a, a, a, b, c, c \}\!\}$ le multiensemble qui contient trois occurrences de a, une occurrence de b et deux occurrences de c. Si m est un multiensemble, on note $m(x)$ le nombre d'occurrences de x dans m, appelé sa *multiplicité*. Si m désigne le multiensemble $\{\!\{ a, a, a, b, c, c \}\!\}$, on a donc $m(a) = 3$, $m(b) = 1$ et $m(c) = 2$. Un élément x appartient à m si et seulement si $m(x) > 0$.

Mise en œuvre en Java. La figure 1 contient le code Java d'une classe `MSet<E>` pour représenter des multiensembles dont les éléments sont du type `E`. Un multiensemble est représenté par un objet qui contient deux champs : d'une part une table de hachage `mult` (ligne 2) qui associe à chaque élément du multiensemble sa multiplicité ; et d'autre part un entier `size` (ligne 3) qui stocke le nombre total d'éléments. Le champ `mult` vérifie l'**invariant** suivant :

$$\text{pour tout } (x, n) \in \texttt{mult}, \text{ on a } n > 0 \tag{1}$$

Autrement dit, la table `mult` contient exactement les éléments du multiensemble, avec pour chaque sa multiplicité. Le champ `size` vérifie l'**invariant** suivant :

$$\texttt{size} = \sum_{(x,n)\in\texttt{mult}} n \tag{2}$$

Le code des différentes méthodes doit maintenir ces deux invariants. On note que ces deux invariants sont trivialement vérifiés pour le multiensemble vide renvoyé par le constructeur, la table `mult` étant vide. La méthode `mult` renvoie la multiplicité d'un élément, et 0 s'il n'est pas dans le multiensemble. On utilisera systématiquement cette méthode plutôt que d'accéder directement à la table de hachage `mult`.

Question 1 Écrire le code de la méthode `void remove(E x)` (ligne 13) qui retire une occurrence de l'élément `x` du multiensemble. Si `x` n'apparaît pas dans le multiensemble, cette méthode est sans effet. Attention à bien préserver les invariants. (4 lignes de code)

```java
1 class MSet<E> {
2   private HashMap<E, Integer> mult;
3   private int size;
4
5   MSet() { this.size = 0; this.mult = new HashMap<>(); }
6
7   int size() { return this.size; }
8   int mult(E x) {
9     if (this.mult.containsKey(x)) return this.mult.get(x); else return 0;
10  }
11  boolean contains(E x) { return mult(x) > 0; }
12  void add(E x) { this.mult.put(x, 1 + mult(x)); this.size++; }
13  void remove(E x) { ... }
14
15  // est-ce que ‘this‘ est inclus dans ‘that‘ ?
16  boolean included(MSet<E> that) { ... }
17
18  public boolean equals(Object obj) {
19    MSet<E> that = (MSet<E>)obj;
20    return this.size == that.size && this.included(that);
21  }
22 }
```

FIGURE 1 – Classe `MSet` pour les multiensembles.

Question 2 Justifier le caractère `private` des champs `mult` et `size` (lignes 2 et 3).

Question 3 Écrire le code de la méthode `boolean included(MSet<E> that)` qui détermine si le multiensemble `this` est inclus dans le multiensemble `that`, au sens où la multiplicité de tout élément dans `this` ne dépasse pas celle dans `that`. (4 lignes de code)

Question 4 Justifier le code de la méthode `equals` (lignes 18–20).

Fonction de hachage. On aimerait pouvoir se servir de multiensembles comme clés dans des tables de hachage. (On en verra une application plus loin.) Pour cela, il faut équiper notre classe `MSet` d'une méthode `hashCode` adéquate. On se propose de l'ajouter à la classe `MSet` comme ceci, avec un parcours de toutes les clés de la table `mult` :

```java
  public int hashCode() {
    int h = 0;
    for (E x : this.mult.keySet()) {
      int hx = x.hashCode();
      ...
    }
    return h;
  }
```

Question 5 Voici trois propositions pour la ligne 5 manquante dans le code ci-dessus.

1. `h = h + hx;`
2. `h = h + mult(x) * hx * hx;`
3. `h = 31 * h + mult(x) * hx;`

Pour chacune, indiquer si elle est cohérente avec `equals` et, si oui, si elle est raisonnable au regard des collisions. On pourra par exemple raisonner dans le cas où `E` est la classe `Character`.

Question 6 Écrire une méthode `static MSet<Character> msOfWord(String s)` qui renvoie le multiensemble des caractères constituant la chaîne `s`. La complexité doit être linéaire (amortie) en la taille de `s` et on la justifiera. (5 lignes de code)

Anagrammes. On dit que deux mots sont des *anagrammes* s'ils sont constitués du même multiensemble de lettres. Ainsi, `MATERNER` et `RAREMENT` sont des anagrammes, car constitués des lettres $\{\!\{ \mathtt{A}, \mathtt{E}, \mathtt{E}, \mathtt{M}, \mathtt{N}, \mathtt{R}, \mathtt{R}, \mathtt{T} \}\!\}$. Étant donné un fichier F contenant des mots (par exemple le fichier de votre correcteur orthographique), on se propose d'en trouver tous les anagrammes. Pour cela, on va remplir un dictionnaire `words` de type

```
HashMap< MSet<Character>, Vector<String> >
```

qui associe à un multiensemble de lettres la liste des mots qui lui correspondent. Avec des mots français, on aura donc quelque chose comme cela :

$$\begin{array}{rcl} \{\!\{ \mathtt{A}, \mathtt{C}, \mathtt{E}, \mathtt{N}, \mathtt{R} \}\!\} & \mapsto & [\mathtt{ANCRE}, \mathtt{ECRAN}, \mathtt{NACRE}, \text{etc.}] \\ \{\!\{ \mathtt{A}, \mathtt{E}, \mathtt{E}, \mathtt{M}, \mathtt{N}, \mathtt{R}, \mathtt{R}, \mathtt{T} \}\!\} & \mapsto & [\mathtt{MATERNER}, \mathtt{RAREMENT}, \text{etc.}] \\ & \text{etc.} & \end{array}$$

On construit le dictionnaire `words` de la manière suivante. Pour chaque mot s du fichier F, on commence par calculer son multiensemble de caractères m avec la méthode `msOfWord` (question 6). Si m n'est pas dans `words`, on crée une nouvelle entrée avec une liste contenant l'unique mot s. Sinon, on ajoute s à la fin de la liste associée à m, avec la méthode `add` de la classe `Vector`.

Question 7 Donner la complexité totale de la construction du dictionnaire `words`, en fonction de la taille du fichier F et de tout autre paramètre qui vous paraîtra pertinent. Justifier soigneusement. (On ne demande pas d'écrire le code de la construction de `words`.)

Question 8 Écrire une méthode `void addAll(MSet<E> that)` qui ajoute tous les éléments du multiensemble `that` au multiensemble `this`. (3 lignes de code)

Question 9 Écrire une méthode `MSet<E> diff(MSet<E> that)` qui renvoie un *nouveau* multiensemble où, pour chaque élément, sa multiplicité est la différence de sa multiplicité dans `this` et de sa multiplicité dans `that`. On pourra faire l'hypothèse que `that.included(this)` vaut `true`. (5 lignes de code)

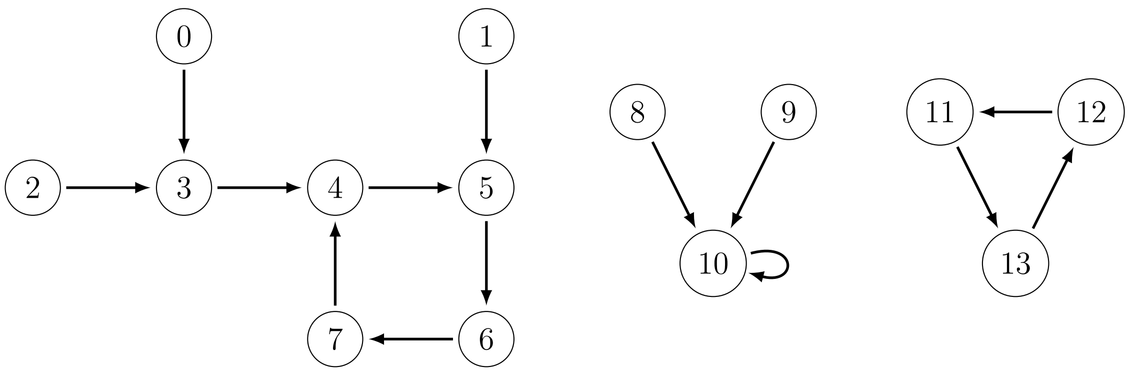
Question 10 Étant donnés deux mots du fichier F, par exemple `MANGER` et `GATEAU`, on peut trouver d'autres paires de mots de F qui en sont des anagrammes : `GAGNE MARTEAU`, `GAGER MANTEAU`, etc. Proposer un algorithme qui prend en entrée deux mots de F et imprime toutes les paires de mots qui en sont des anagrammes. On pourra supposer le dictionnaire `words` construit au préalable. Donner la complexité de votre algorithme en fonction du nombre N d'entrées dans le dictionnaire `words`. Si N est de l'ordre de 10^5, votre solution est-elle réaliste ?

```
class G {
  final int N; // les sommets sont 0,1,...,N-1
  int succ(int v) { ... } // l'unique successeur de v
}
```

FIGURE 2 – Une classe pour les graphes fonctionnels.

2 Graphes fonctionnels

Dans ce problème, on considère des graphes orientés *finis* où, pour chaque sommet, il existe *exactement un arc sortant*. Un tel graphe peut être vu comme la donnée d'un ensemble fini V de sommets et d'une fonction $s : V \to V$ telle que $s(x)$ est l'unique successeur du sommet x (d'où le nom de graphe fonctionnel). Un exemple de graphe fonctionnel est l'ensemble des pages Wikipedia où, pour chaque page, on ne considère que le premier lien vers une autre page (sous l'hypothèse raisonnable qu'il y en a toujours au moins un). Un autre exemple de graphe fonctionnel est un générateur pseudo-aléatoire d'entiers de type `int` en Java, où chaque entier est associé à l'entier suivant dans la séquence pseudo-aléatoire. Voici un exemple de graphe fonctionnel avec 14 sommets :



On note qu'un arc de x à x est possible (comme ici sur le sommet 10) et qu'un cycle peut donc avoir une longueur 1.

Question 11 Soit $G = (V, s)$ un graphe fonctionnel. Montrer que, depuis tout sommet $x \in V$, il existe un chemin de longueur $n \geq 0$ qui mène à un cycle de longueur $m \geq 1$.

Mise en œuvre en Java. Sans perte de généralité, on peut supposer que les sommets sont les entiers $0, 1, \ldots, \texttt{N} - 1$. La figure 2 contient le code Java d'une classe `G` pour représenter un graphe fonctionnel, avec un champ `N` contenant le nombre de sommets et une méthode `succ` donnant l'unique successeur d'un sommet `v`.

Question 12 Quel intérêt y a-t-il à implémenter l'adjacence sous la forme d'une *méthode* `succ` plutôt que sous la forme d'un *tableau* `succ` de taille `N` ?

Question 13 Écrire une méthode `boolean[] dfs(int v)` qui réalise un *parcours en profondeur* à partir du sommet `v` et renvoie un tableau de booléens de taille `N`, indiquant, pour chaque sommet, s'il a été visité par ce parcours. On cherchera à exploiter le caractère fonctionnel du graphe d'une part et à éviter tout débordement de la pile d'appel d'autre part. (3 lignes de code)

```
1  int comp[]; // la composante de chaque sommet (entier >= 1)
2  void compute() {
3    comp = new int[N];
4    int nc = 0;
5    for (int v = 0; v < N; v++) {
6      if (comp[v] > 0) continue;
7      assert (comp[v] == 0);
8      int w = v; do { comp[w] = -1; w = succ(w); } while (comp[w] == 0);
9      if (comp[w] == -1) {
10       nc++; w = v; do { comp[w] = nc; w = succ(w); } while (comp[w] == -1);
11     } else {
12       int u = v; do { comp[u] = comp[w]; u = succ(u); } while (comp[u] == -1);
13     }
14   }
15 }
```

FIGURE 3 – Calcul des composantes.

Composantes. On dit que deux sommets x et y sont dans la même *composante* s'il existe un sommet z (possiblement identique à x ou y) avec un chemin de x à z et un chemin de y à z. Ainsi, le graphe fonctionnel donné en exemple plus haut possède trois composantes, à savoir $\{0, 1, \dots, 7\}$, $\{8, 9, 10\}$ et $\{11, 12, 13\}$. Dans ce qui suit, nous allons chercher à identifier les composantes en les numérotant (à partir de 1) et en associant à chaque sommet le numéro de sa composante.

La figure 3 contient un code Java qui réalise le calcul des composantes. Le tableau `comp` (ligne 1) contient au final le numéro de la composante de chaque sommet, les composantes étant numérotées à partir de 1. La méthode `compute` crée et remplit ce tableau. On rappelle que la boucle `do { b } while (e)` de Java est équivalente à `b; while (e) { b }`. On suggère de **bien prendre le temps de lire et de comprendre ce code**, par exemple en le déroulant entièrement sur le graphe ci-dessus, ce qui facilitera les réponses aux cinq questions suivantes.

Question 14 Pour le graphe donné plus haut en exemple, indiquer l'ordre dans lequel les sommets se voient affecter leur numéro de composante, en donnant pour chaque sommet `v` le numéro de sa composante (`comp[v]`) et la ligne de code (10 ou 12) qui l'a affecté.

Question 15 Expliquer le rôle de chacune des trois boucles `do`/`while` dans la méthode `compute`.

Question 16 Justifier la terminaison de la méthode `compute`, en donnant pour chacune des trois boucles `do`/`while` une grandeur positive ou nulle qui décroît à chaque itération.

Question 17 Montrer que la fonction `compute` s'exécute en temps $O(\texttt{N})$, en supposant que la méthode `succ` s'exécute en temps constant.

Question 18 On aimerait connaître, pour chaque sommet, sa distance au cycle et la longueur de ce dernier, qui existent en vertu de la question 1. (Si un sommet est déjà sur un cycle, alors sa distance au cycle est 0.) Expliquer comment modifier la méthode `compute` pour qu'elle calcule également ces deux quantités, et comment on peut les stocker. Si la complexité reste la même, le justifier ; si la complexité change, donner la nouvelle complexité et la justifier. (On ne demande pas d'écrire le code Java.)

Degré entrant. On rappelle que le degré entrant d'un sommet est le nombre d'arcs vers ce sommet. Ainsi, dans l'exemple ci-dessus, le sommet 2 a un degré entrant 0, le sommet 4 un degré entrant 2 et le sommet 10 un degré entrant 3. Dans un graphe fonctionnel, si le degré sortant vaut toujours 1, le degré entrant peut prendre toute valeur entre 0 et `N`.

Question 19 Proposer un algorithme pour déterminer, pour un entier K donné, les K sommets ayant *le plus grand degré entrant*. En cas d'égalité, on choisit arbitrairement. (Si par exemple $K = 3$ et qu'il y a 5 sommets de plus grand degré entrant, alors tout sous-ensemble de 3 sommets parmi ces 5 convient comme réponse.) Donner la complexité de votre algorithme, temporelle et spatiale, en fonction de `N` et de K. (On ne demande pas d'écrire le code Java.)

Grandes valeurs de `N`. Si la valeur de `N` est très grande, il n'est pas forcément possible d'allouer un espace proportionnel à `N`. On peut encore avoir un objet de la classe `G`, qui ne stocke rien d'autre que `N`, mais en revanche on ne peut plus allouer de tableau de taille `N` comme dans la méthode `compute`.

Question 20 Dans ce contexte, expliquer comment on peut adapter l'algorithme du lièvre et de la tortue vu en cours pour calculer, pour *un* sommet donné, sa distance au cycle et la longueur de ce cycle. La complexité en espace doit être constante.

A Bibliothèque standard Java

`class Vector<E>`	un tableau redimensionnable avec des éléments de type `E`
`Vector<>()`	renvoie un nouveau tableau redimensionnable, vide
`boolean isEmpty()`	renvoie `true` si et seulement si le tableau est vide
`void add(E x)`	ajoute l'élément `x` à la fin du tableau
`void addAll(Collection<E> c)`	ajoute tous les éléments de la collection `c` à la fin du tableau (la collection `c` peut être une `LinkedList` par exemple)

On peut parcourir tous les éléments d'un tableau redimensionnable `v` avec la construction

```
for (E x: v) ...
```

`class String`	le type des chaînes de caractères
`int length()`	la longueur de la chaîne
`char charAt(int i)`	le caractère d'indice `i`, pour $0 \leq$ `i` $<$ `length()`

`class HashMap<K, V>`	un dictionnaire dont les clés sont de type `K` et les valeurs de type `V`
`HashMap<>()`	renvoie un nouveau dictionnaire, vide
`boolean containsKey(K k)`	indique s'il existe une valeur associée à `k`
`V get(K k)`	renvoie la valeur associée à `k`, si elle existe, et `null` sinon
`void put(K k, V v)`	associe la valeur `v` à la clé `k` (en écrasant toute valeur précédemment associée à `k`, le cas échéant)
`void remove(K k)`	supprime l'entrée pour la clé `k`, si elle existe
`Set<K> keySet()`	renvoie l'ensemble de toutes les clés de la table (dans un ordre arbitraire)

On peut parcourir toutes les clés d'un dictionnaire `d` avec la construction

```
for (K k: d.keySet()) ...
```

`Collection<V> values()`	renvoie l'ensemble de toutes les valeurs de la table (dans un ordre arbitraire)

On peut parcourir toutes les valeurs d'un dictionnaire `d` avec la construction

```
for (V v: d.values()) ...
```

*   *
*