X2016 — INF411
Contrôle classant / 23 octobre 2017 / 14h–17h

Tous documents de cours autorisés (polycopié, transparents, notes personnelles). Le dictionnaire papier est autorisé pour les EV2 et EV3. Les ordinateurs, Ipad, dictionnaires électroniques, les tablettes et téléphones portables sont interdits.

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

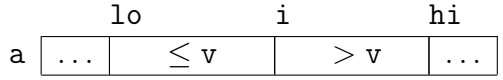
Le correcteur prêtera attention à la qualité de la rédaction et vous remercie d'avance d'écrire lisiblement. Merci également de numéroter vos copies en indiquant leur nombre total.

Quelques éléments de la bibliothèque standard Java sont rappelés à la fin du sujet. Vous pouvez utiliser librement tout autre élément de la bibliothèque standard Java.

1 Trier naturellement

Dans ce problème, on cherche à améliorer le tri par insertion et le tri fusion vus en cours. On considère que l'on trie des éléments d'une classe `E` donnée qui implémente l'interface `Comparable<E>`. On rappelle qu'on peut alors comparer deux éléments x et y de type `E` avec le résultat de x`.compareTo(`y`)`, de type `int`. Cet entier est strictement négatif si $x < y$, nul si $x = y$ et strictement positif si $x > y$. En dehors du code Java, on s'autorisera à écrire les comparaisons sous la forme $x \leq y$ plutôt que x`.compareTo(`y`)` ≤ 0. On note $a[l..h[$ la portion du tableau a comprise entre les indices l inclus et h exclus. Dans la suite, quand on dit qu'un segment $a[l..h[$ du tableau a est trié par ordre croissant, on l'entend au sens large, c'est-à-dire $a[i] \leq a[j]$ pour tous i et j tels que $l \leq i \leq j < r$.

Question 1 Écrire une méthode statique `int binarySearchRight(E[] a, int lo, int hi, E v)` qui cherche dans le segment `a[lo..hi[`, supposé trié par ordre croissant, la position la plus à droite où la valeur `v` doit être insérée pour conserver un ordre croissant, c'est-à-dire l'entier `i` tel que l'on ait la situation suivante :



On suppose $0 \leq$ `lo` $<$ `hi` $\leq$ `a.length` et on doit notamment assurer `lo` $\leq$ `i` $\leq$ `hi`. Le cas `i` $=$ `lo` (resp. `i` $=$ `hi`) correspond à une valeur `v` strictement inférieure (resp. supérieure ou égale) à toutes les valeurs de `a[lo..hi[`. On garantira une complexité $O(\log($`hi` $-$ `lo`$))$.

Tri par insertion dichotomique. La figure 1 contient le code d'une méthode `binarySort` qui réalise un tri par insertion du tableau `a` en se servant à chaque étape de la méthode `binarySearchRight` pour trouver le point d'insertion. (Si besoin, la documentation de `System.arraycopy` est donnée à la fin du sujet.)

Question 2 Illustrer le fonctionnement de cet algorithme sur le tri du tableau d'entiers

7	12	1	2	1

On indiquera les différents appels à `binarySearchRight` et `arraycopy` avec leurs arguments et leurs résultats.

```
static void binarySort(E[] a) {
  for (int i = 1; i < a.length; i++) {
    E v = a[i];
    int j = binarySearchRight(a, 0, i, v);
    System.arraycopy(a, j, a, j + 1, i - j);
    a[j] = v;
  }
}
```

FIGURE 1 – Tri par insertion dichotomique.

```
// fusionne a1[l..m[ et a1[m..r[ dans a2[l..r[
static void merge(E[] a1, E[] a2, int l, int m, int r) {
  assert l <= m && m <= r;
  int i = l, j = m;
  for (int k = l; k < r; k++)
    if (i < m && (j == r || a1[i].compareTo(a1[j]) <= 0))
      a2[k] = a1[i++];
    else
      a2[k] = a1[j++];
}
```

FIGURE 2 – Fusion.

Question 3 Donner un ordre de grandeur du nombre d'appels à `compareTo` effectués par `binarySort`. Comparer ce tri au tri par insertion (vu en cours) dans le pire des cas et dans le meilleur des cas.

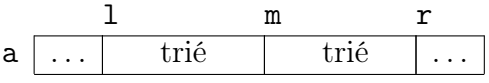
Question 4 Justifier que la méthode `binarySort` réalise un tri *stable*. On rappelle qu'un tri stable est un tri qui préserve les positions relatives des éléments égaux au sens de `compareTo`.

Tri fusion naturel. On cherche maintenant à améliorer le tri fusion. Notre première idée consiste à tirer partie des segments déjà triés (appelés *runs* en anglais).

Question 5 Écrire une méthode `int findRun(E[] a, int lo)` qui renvoie le plus grand entier `hi` tel que `a[lo..hi[` est trié par ordre croissant. On suppose $0 \leq$ `lo` $<$ `a.length`. On doit garantir `lo` $<$ `hi` $\leq$ `a.length`.

Fusion. On commence par se donner une méthode `void merge(E[] a1, E[] a2, int l, int m, int r)` qui fusionne les segments `a1[l..m[` et `a1[m..r[`, supposés triés par ordre croissant, dans `a2[l..r[`. Son code est donné figure 2. Elle est identique à celle vue en cours.

Notre deuxième idée consiste à écrire une fusion plus efficace, en se servant de la méthode `binarySearchRight` de la question 1. On se donne un tableau `a` et trois indices `l`, `m` et `r` délimitant deux segments consécutifs déjà triés par ordre croissant, avec $0 \leq$ `l` $<$ `m` $<$ `r` $\leq$ `a.length` :



Plutôt que de copier tous les éléments de `a[l..r[` dans un tableau auxiliaire puis d'appeler `merge`, on peut noter que certains éléments au début du premier segment sont déjà à leur place car inférieurs ou égaux à `a[m]`. De même, certains éléments à la fin du second segment sont déjà à leur place car strictement supérieurs à `a[m-1]`.

```
static void naturalMergesort(E[] a) {
  int n = a.length;
  if (n <= 1)
    return;
  E[] tmp = new E[n];
  while (true) {
    for (int lo = 0; lo < n - 1;) {
      int mid = findRun(a, lo);
      if (mid == n) {
        if (lo == 0)
          return;
        break;
      }
      int hi = findRun(a, mid);
      merge2(a, tmp, lo, mid, hi);
      lo = hi;
    }
  }
}
```

FIGURE 3 – Tri fusion naturel.

Question 6 Écrire une méthode `void merge2(E[] a, E[] tmp, int l, int m, int r)` qui fusionne les segments `a[l..m[` et `a[m..r[`, supposés triés par ordre croissant, dans `a[l..r[`, en se servant du tableau auxiliaire `tmp` et de la méthode `merge` uniquement pour les éléments ayant vraiment besoin d'être fusionnés. On suppose $0 \leq \texttt{l} < \texttt{m} < \texttt{r} \leq \texttt{a.length}$. On garantira que `merge2` préserve la position relative des éléments égaux. Indication : on se servira de la méthode `binarySearchRight`.

Question 7 Donner un ordre de grandeur du nombre d'appels à `compareTo` effectués par `merge2` dans le pire des cas et dans le meilleur des cas. On supposera $\texttt{m} = \lfloor \frac{\texttt{l}+\texttt{r}}{2} \rfloor$.

Tri fusion naturel. Des deux méthodes `findRun` et `merge2` on déduit un algorithme de tri fusion, dit tri fusion naturel, dont le code est donné figure 3.

Question 8 Illustrer le fonctionnement de cet algorithme sur le tri du tableau d'entiers

7	12	1	2	1	8	16

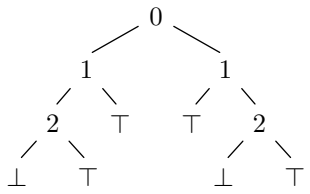
On indiquera les différents appels à `findRun` et `merge2` avec leurs arguments et leurs résultats.

Question 9 Justifier la terminaison de cet algorithme.

Question 10 Justifier que la méthode `naturalMergesort` réalise un tri stable (voir la question 4 pour la définition).

2 Arbres combinatoires

Dans cette partie, on étudie les arbres combinatoires (en anglais ZDD pour *Zero-suppressed Decision Diagram*), une structure de données pour représenter des ensembles d'ensembles d'entiers. Un arbre combinatoire est un arbre binaire dont les nœuds sont étiquetés par des entiers et les feuilles par $\bot$ ou $\top$. Voici un exemple d'arbre combinatoire :



Un nœud étiqueté par i, de sous-arbre gauche L et de sous-arbre droit R sera noté $i \to L, R$. L'arbre ci-dessus peut donc également s'écrire sous la forme

$$0 \to (1 \to (2 \to \bot, \top), \top), (1 \to \top, (2 \to \bot, \top)). \tag{1}$$

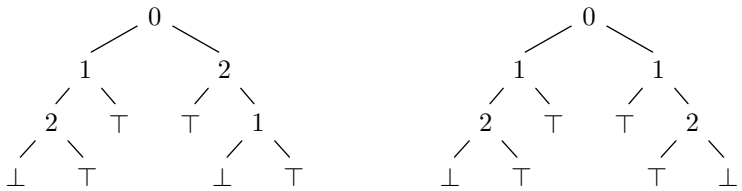
Dans ce sujet, on impose la double propriété suivante sur tout (sous-)arbre combinatoire de la forme $i \to L, R$: d'une part

$$L \text{ et } R \text{ ne contiennent pas d'élément } j \text{ avec } j \leq i \tag{ordre}$$

et d'autre part

$$R \neq \bot. \tag{suppression}$$

Ainsi les deux arbres

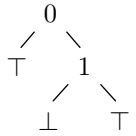


ne correspondent pas à des arbres combinatoires, car celui de gauche ne vérifie pas la condition (ordre) et celui de droite ne vérifie pas la condition (suppression).

À tout arbre combinatoire A on associe un ensemble d'ensembles d'entiers, noté $S(A)$, défini par

$$\begin{aligned} S(\bot) &= \emptyset \\ S(\top) &= \{\emptyset\} \\ S(i \to L, R) &= S(L) \cup \{\{i\} \cup s \mid s \in S(R)\} \end{aligned}$$

L'interprétation d'un arbre A de la forme $i \to L, R$ est donc la suivante : i est le plus petit élément appartenant à au moins un ensemble de $S(A)$, L est le sous-ensemble de $S(A)$ des ensembles qui ne contiennent pas i, et R est le sous-ensemble de $S(A)$ des ensembles qui contiennent i auxquels on a enlevé i. Ainsi, l'arbre



est interprété comme l'ensemble $\{\emptyset, \{0, 1\}\}$.

Question 11 Donner l'ensemble défini par l'arbre combinatoire de l'exemple (1).

```
abstract class ZDD      { ... }
class Zero  extends ZDD { ... }
class One   extends ZDD { ... }
class Znode extends ZDD { int element; ZDD left, right; ... }
```

FIGURE 4 – Arbres combinatoires en Java.

Question 12 Donner les trois arbres combinatoires correspondant aux trois ensembles $\{\{0\}\}$, $\{\emptyset, \{0\}\}$ et $\{\{0, 2\}\}$.

Représentation en Java. Pour représenter les arbres combinatoires en Java, on utilise une classe abstraite et des sous-classes, comme pour la structure de cordes vue en cours. La figure 4 contient la déclaration de quatre classes : une classe abstraite `ZDD` et trois sous-classes `Zero`, `One` et `Znode`. La classe `Zero` représente $\bot$, la classe `One` représente $\top$ et la classe `Znode` représente un nœud $i \to L, R$, les champs `element`, `left` et `right` contenant respectivement les valeurs de i, L et R. Les constructeurs, évidents, sont omis.

Dans tout ce qui suit, on représente en Java un ensemble d'entiers par un tableau de type `int[]` trié par ordre croissant.

Question 13 Écrire une méthode statique `ZDD singleton(int[] set)` qui prend en argument un ensemble `set` et renvoie l'arbre combinatoire qui représente le singleton $\{X\}$ où X est l'ensemble représenté par `set`.

Question 14 Écrire une méthode statique `ZDD allSubsets(int n)` qui prend en argument un entier `n`, avec $0 \leq$ `n`, et renvoie l'arbre combinatoire qui représente toutes les parties de $\{0, \ldots,$ `n`$-1\}$. On garantira une complexité $O($`n`$)$.

Question 15 Écrire une méthode `boolean contains(int[] set)` qui prend en argument un ensemble `set` et détermine si cet ensemble appartient à $S($`this`$)$. Indication : on pourra commencer par écrire une méthode plus générale `boolean contains(int[] set, int i)` qui détermine si le sous-ensemble des éléments de `set` aux indices supérieurs ou égaux à `i` appartient à $S($`this`$)$, en la définissant dans chacune des trois sous-classes.

Question 16 Proposer un algorithme pour calculer l'intersection `inter`(A_1, A_2) de deux arbres combinatoires A_1 et A_2, c'est-à-dire un arbre combinatoire tel que $S($`inter`$(A_1, A_2)) = S(A_1) \cap S(A_2)$. On l'écrira sous la forme d'équations récursives

$$\begin{aligned}
\texttt{inter}(A_1, \bot) &= \ldots \\
\texttt{inter}(\bot,\ A_2) &= \ldots \\
\texttt{inter}(\top,\ \top) &= \ldots \\
\texttt{inter}((i_1 \to L_1, R_1),\ \top) &= \ldots \\
\texttt{inter}(\top,\ (i_2 \to L_2, R_2)) &= \ldots \\
\texttt{inter}((i_1 \to L_1, R_1),\ (i_2 \to L_2, R_2)) &= \ldots
\end{aligned}$$

en prenant soin de garantir la terminaison et le respect des conditions (ordre) et (suppression). On ne demande pas d'écrire le code Java.

Taille d'un arbre combinatoire. On définit la taille d'un arbre combinatoire A de type `ZDD`, notée $T(A)$, comme le nombre d'objets *distincts* de la classe `Znode` qui le composent. Ainsi, si vous avez correctement programmé la méthode `allSubsets` de la question 14, vous devez avoir $T(\texttt{allSubsets(1000)}) = 1000$.

Question 17 Quelle est la taille minimale d'un arbre combinatoire de type `ZDD` pour l'exemple (1) ? Justifier.

Question 18 Écrire une méthode `int size()` qui calcule la taille de l'arbre combinatoire `this`. Indication : on pourra écrire un parcours de l'arbre et se servir d'un ensemble de type `HashSet<Znode>` dans lequel on collecte tous les objets de type `Znode` rencontrés. On garantira une complexité $O(T(\texttt{this}))$.

Question 19 Écrire une méthode `int card()` qui calcule le cardinal de l'ensemble représenté par l'arbre combinatoire `this`. On garantira une complexité $O(T(\texttt{this}))$. Indication : utiliser le principe de mémoïsation.

Question 20 Expliquer comment implémenter une méthode `ZDD inter(ZDD z)` qui réalise le calcul de l'intersection de la question 16 avec la complexité $O(T(\texttt{this}) \times T(\texttt{z}))$. On ne demande pas d'écrire le code Java. Justifier soigneusement la complexité.

A Bibliothèque standard Java

`System.arraycopy(E[] a1, int i1, E[] a2, int i2, int len)`
copie `a1`[`i1`..`i1+len`[dans `a2`[`i2`..`i2+len`[
suppose $0 \leq$ `i1` $\leq$ `i1+len` $\leq$ `a1.length` et $0 \leq$ `i2` $\leq$ `i2+len` $\leq$ `a2.length`
fonctionne correctement même lorsque `a1` et `a2` sont le même tableau
peut être appelée avec `len` $= 0$ (et dans ce cas ne fait rien)

`class HashSet<E>`

`void add(E x)`	ajoute l'élément `x`
`boolean contains(E x)`	indique la présence de l'élément `x`
`int size()`	renvoie le cardinal de l'ensemble

`class HashMap<K, V>`

`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)
`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

* *
*