

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 $x.\text{compareTo}(y)$, 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 $x.\text{compareTo}(y) \leq 0$. 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 $a[\text{lo}..\text{hi}[$, 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 :

	<code>lo</code>	<code>i</code>	<code>hi</code>
<code>a</code>	...	$\leq v$	$> v$

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

Correction : C'est une petite variante de la recherche dichotomique qui est dans le poly.

```
static int binarySearchRight(E[] a, int lo, int hi, E v) {
    while (lo < hi) { // a[..lo[ <= v < a[hi..[
        int mid = lo + (hi - lo) / 2;
        if (a[mid].compareTo(v) <= 0) lo = mid+1; else hi = mid;
    }
    return lo;
}
```

```

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.

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.

Correction :

i	v	binarySearchRight	j	arraycopy	après a[j]=v
1	12	(a,0,1,12)	1	(a,1,a,2,0)	7,12,1,2,1
2	1	(a,0,2,1)	0	(a,0,a,1,2)	1,7,12,2,1
3	2	(a,0,3,2)	1	(a,1,a,2,2)	1,2,7,12,1
4	1	(a,0,4,1)	1	(a,1,a,2,3)	1,1,2,7,12

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.

Correction : Le nombre de comparaisons est $\log(1) + \log(2) + \dots + \log(N - 1) = O(N \log(N))$.

Ce tri n’est donc pas toujours meilleur que le tri par insertion vu en cours : il est en $O(N \log(N))$ sur un tableau déjà trié, au lieu de $O(N)$. En revanche, il ne fait jamais plus de $O(N \log(N))$ comparaisons, là où le tri par insertion classique peut en faire $O(N^2)$. Si les comparaisons sont coûteuses, il peut donc être avantageux.

Note : il est assez simple de retrouver un temps $O(N)$ sur un tableau déjà trié en rajoutant la ligne

```
if (a[i-1].compareTo(v) <= 0) continue;
```

mais ce n’est pas la question ici.

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`.

Correction : Notons $x \leq_s y$ la relation $x < y$ ou $x \leq y$ et x apparaît avant y dans le tableau initial. Montrons qu'à chaque tour de boucle on a les deux propriétés :

- le segment `a[0..i[` est trié pour \leq_s ;
- pour tout élément x dans `a[0..i[` et tout élément y dans `a[i..a.length[`, si $x \leq y$ alors $x \leq_s y$.

C'est vrai initialement, car `i = 1` et le tableau est dans son état initial. Supposons les deux propriétés pour `i` et insérons la valeur `v = a[i]` à la place `j` donnée par `binarySearchRight`. Par définition de `binarySearchRight`, on se retrouve avec la situation

	0		j		i+1
a	$\leq v$	v	$> v$...	

et donc `a[0..i+1[` est bien trié pour \leq_s . Prenons maintenant un élément x dans `a[0..i+1[` et un élément y dans `a[i+1..a.length[` tels que $x \leq y$. Si x est `v` alors $x \leq_s y$ car `v` apparaît avant y dans le tableau initial (la partie `a[i+1..a.length[` n'a pas été modifiée pour l'instant). Si x n'est pas `v`, alors $x \leq_s y$ par hypothèse sur le tour précédent.

Une fois que l'on sort de la boucle, la première propriété assure que le tableau `a` tout entier est trié pour \leq_s et donc que le tri est stable.

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 $lo < hi \leq a.length$.

Correction : Pas de difficulté. Il faut seulement faire attention à ne pas dépasser la fin du tableau.

```
static int findRun(E[] a, int lo) {
    while (++lo < a.length && a[lo - 1].compareTo(a[lo]) <= 0)
        ;
    return lo;
}
```

Note : on a supposé qu'il y avait au moins un élément ($lo < a.length$), ce qui est nécessaire pour que ce code soit correct.

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.

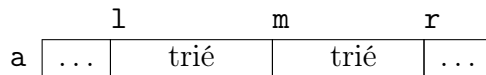
Notre deuxième idée consiste à écrire une fusion plus efficace, en se servant de la méthode `binarySearchRight` de la question 1. On se donne un tableau `a` et trois indices `l`, `m` et `r` délimitant deux segments consécutifs déjà triés par ordre croissant, avec $0 \leq l < m < r \leq a.length$:

```

// 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.



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 $a[m-1]$.

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 $0 \leq l < m < r \leq a.length$. On garantira que `merge2` préserve la position relative des éléments égaux. Indication : on se servira de la méthode `binarySearchRight`.

Correction :

```

static void merge2(E[] a, E[] tmp, int l, int m, int r) {
    int lo = binarySearchRight(a, l, m, a[m]); // a[l..lo[ déjà en place
    int hi = binarySearchRight(a, m, r, a[m-1]); // a[hi..r[ déjà en place
    System.arraycopy(a, lo, tmp, lo, hi - lo);
    merge(tmp, a, lo, m, hi);
}

```

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 $m = \lfloor \frac{l+r}{2} \rfloor$.

Correction : Le nombre de comparaisons est $\log(m-1) + \log(r-1)$ pour les deux recherches dichotomiques, soit $\log(r-1)$ car m est situé au milieu. Dans le pire des cas, il faut ensuite fusionner complètement les deux segments, soit un nombre $O(r-1)$ de comparaisons et donc un total $O(\log(r-1) + r-1) = O(r-1)$. Dans le meilleur des cas, il n'y a aucune fusion à faire et donc le nombre de comparaisons est seulement en $O(\log(r-1))$.

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.

```

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 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.

Correction :

lo	mid	hi	a
0	<code>findRun(0)</code>	2	<code>findRun(2)</code>
4	<code>findRun(4)</code>	7	<code>break</code>
0	<code>findRun(0)</code>	4	<code>findRun(4)</code>
0	<code>findRun(0)</code>	7	<code>return</code>

Question 9 Justifier la terminaison de cet algorithme.

Correction : Il y a deux boucles pour lesquelles il faut justifier la terminaison.

Pour la boucle `for` interne, il faut montrer que `lo` progresse strictement. Comme `lo < n - 1`, il y a au moins deux éléments et le premier appel à `findRun` renvoie donc une valeur `mid` strictement plus grande que `lo`. Puis le second appel à `findRun` (si on n'est pas sorti avant avec `return` ou `break`) renvoie une valeur `hi` strictement plus grande que `mid`. Donc au final `lo < hi` et l'affectation `lo=hi` assure donc la progression.

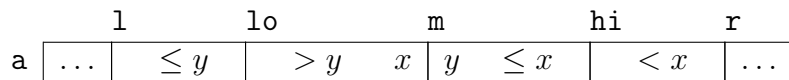
Pour la terminaison de la boucle `while` externe, il faut montrer qu'on finira fatalement par exécuter l'instruction `return` correspondant au cas `mid = n` et `lo = 0`. Pour cela, montrons que le segment initial trié par ordre croissant, démarrant à l'indice 0, augmente strictement à chaque tour de la boucle `while`. Lorsque `lo = 0`, au premier tour de la boucle `for`, la valeur renvoyée par le premier appel à `findRun` est justement la longueur de ce segment. Si la valeur de `mid` n'est pas égal à `n` (sinon, on exécute `return`), on fait

un second appel à `findRun` qui renvoie une valeur strictement plus grande que `mid` (on l'a montré juste avant). Dès lors, la fusion aboutit à un premier segment trié strictement plus grand. On finira donc par avoir `mid = n` pour `lo = 0`.

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

Correction : Commençons par montrer que la méthode `merge` préserve la position relative des éléments égaux. Le test étant écrit sous la forme `a[i].compareTo(a[j]) <= 0`, les éléments de la première moitié sont toujours préférés à ceux de la seconde lorsqu'il y a égalité. Par ailleurs, les éléments d'une même moitié conservent leurs positions relatives (on ne fait qu'avancer dans chaque segment).

Montrons ensuite que `merge2` préserve la position relative des éléments égaux. En notant respectivement x et y les éléments `a[m-1]` et `a[m]`, on a la situation suivante après les deux appels à `binarySearchRight` :

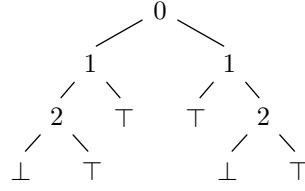


Puis on copie vers `tmp` le segment `a[lo..hi[` et on appelle `merge(tmp, a, lo, m, hi)`. Prenons maintenant deux éléments égaux dans le segment `a[l..r[` tout entier. S'ils sont tous les deux en dehors de `a[lo..hi[` ils n'ont pas été déplacés. S'ils sont tous les deux dans `a[lo..hi[`, leurs positions relatives sont préservées par `merge` (hypothèse). Enfin, si l'un est dans `a[lo..hi[` et l'autre en dehors, leurs positions relatives sont également préservées (car les segments `a[l..lo[` et `a[hi..r[` ne sont pas modifiés).

Montrons enfin qu'à tout instant dans `naturalMergesort`, les positions relatives des éléments égaux sont préservées. Pour cela, montrons que c'est un invariant de la boucle `for` (et donc automatiquement de la boucle `while` car aucun travail n'est fait dans la boucle `while` à l'extérieur de la boucle `for`). Initialement, c'est vrai, car le tableau est dans son état initial. Considérons une itération de la boucle `for`, en supposant les éléments égaux dans leurs positions relatives initiales. Les deux appels à `findRun` ne modifient pas le tableau. L'appel à `merge2` préserve la position relative des éléments égaux dans le segment `a[lo..hi[` et ne modifie pas les autres éléments. Dès lors, la position relative des éléments égaux est préservée sur l'intégralité du tableau, en considérant les neuf cas de figure pour deux éléments se trouvant dans les segments `a[0..lo[`, `a[lo..hi[` ou `a[hi..a.length[`.

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 \perp 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 $i \rightarrow L, R$. L'arbre ci-dessus peut donc également s'écrire sous la forme

$$0 \rightarrow (1 \rightarrow (2 \rightarrow \perp, \top), \top), (1 \rightarrow \top, (2 \rightarrow \perp, \top)). \quad (1)$$

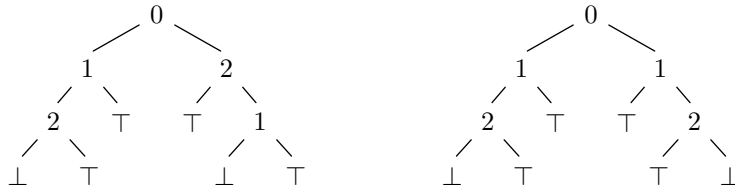
Dans ce sujet, on impose la double propriété suivante sur tout (sous-)arbre combinatoire de la forme $i \rightarrow L, R$: d'une part

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

et d'autre part

$$R \neq \perp. \quad (\text{suppression})$$

Ainsi les deux arbres

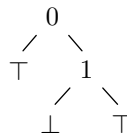


ne correspondent pas à des arbres combinatoires, car celui de gauche ne vérifie pas la condition (ordre) et celui de droite ne vérifie pas la condition (suppression).

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

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

L'interprétation d'un arbre A de la forme $i \rightarrow L, R$ 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 $\{\emptyset, \{0, 1\}\}$.

```

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

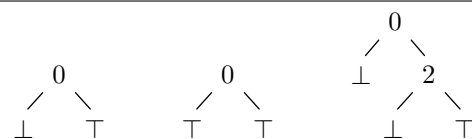
```

FIGURE 4 – Arbres combinatoires en Java.

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

Correction : $\{\{0\}, \{1\}, \{2\}, \{0, 1, 2\}\}$

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

Correction : 

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 \perp , la classe `One` représente \top et la classe `Znode` représente un nœud $i \rightarrow L, R$, 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`.

Correction : On construit le ZDD de bas en haut, avec un nœud pour chaque élément de l'ensemble. Il faut parcourir le tableau de la droite vers la gauche.

```

static ZDD singleton(int [] set) {
    ZDD z = new One();
    for (int i = set.length - 1; i >= 0; i--)
        z = new Znode(set[i], new Zero(), z);
    return z;
}

```

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 $\{0, \dots, n-1\}$. On garantira une complexité $O(n)$.

Correction : On construit le ZDD de bas en haut.

```
static ZDD allSubsets(int n) {
    assert 0 <= n;
    ZDD z = new One();
    while (--n >= 0) z = new Znode(n, z, z);
    return z;
}
```

Question 15 Écrire une méthode boolean `contains(int[] set)` qui prend en argument un ensemble `set` et détermine si cet ensemble appartient à $S(\text{this})$. 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 $S(\text{this})$, en la définissant dans chacune des trois sous-classes.

Correction : On suit l'indication, ce qui donne dans la classe ZDD

```
boolean contains(int[] set) { return contains(set, 0); }
abstract boolean contains(int[] set, int i);
```

Dans la classe Zero, c'est immédiat :

```
boolean contains(int[] set, int i) { return false; }
```

Dans la classe One, il faut s'assurer qu'il ne reste plus d'éléments dans l'ensemble :

```
boolean contains(int[] set, int i) { return i == set.length; }
```

Enfin, dans la classe Znode, on descend à gauche ou à droite, selon que `set[i]` est ou non égal à `this.element`.

```
boolean contains(int[] set, int i) {
    assert i <= set.length;
    if (i == set.length) return this.left.contains(set, i);
    int x = set[i];
    return x == this.element && this.right.contains(set, i+1)
        || x > this.element && this.left.contains(set, i);
}
```

Il faut prendre soin notamment de ne pas accéder au tableau au delà de sa taille.

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 $S(\text{inter}(A_1, A_2)) = S(A_1) \cap S(A_2)$. On l'écrira sous la forme d'équations récursives

$$\begin{aligned} \text{inter}(A_1, \perp) &= \dots \\ \text{inter}(\perp, A_2) &= \dots \\ \text{inter}(\top, \top) &= \dots \\ \text{inter}((i_1 \rightarrow L_1, R_1), \top) &= \dots \\ \text{inter}(\top, (i_2 \rightarrow L_2, R_2)) &= \dots \\ \text{inter}((i_1 \rightarrow L_1, R_1), (i_2 \rightarrow L_2, R_2)) &= \dots \end{aligned}$$

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

Correction :

$$\begin{aligned}
 \text{inter}(A_1, \perp) &= \text{inter}(\perp, A_2) = \perp \\
 \text{inter}(\top, \top) &= \top \\
 \text{inter}((i_1 \rightarrow L_1, R_1), \top) &= \text{inter}(L_1, \top) \\
 \text{inter}(\top, (i_2 \rightarrow L_2, R_2)) &= \text{inter}(\top, L_2) \\
 \text{inter}((i \rightarrow L_1, R_1), (i \rightarrow L_2, R_2)) &= i \rightarrow \text{inter}(L_1, L_2), \text{inter}(R_1, R_2) \\
 &\quad \text{si } \text{inter}(R_1, R_2) \neq \perp, \\
 &\quad \text{inter}(L_1, L_2) \text{ sinon} \\
 \text{inter}((i_1 \rightarrow L_1, R_1), (i_2 \rightarrow L_2, R_2)) &= \text{inter}(L_1, (i \rightarrow L_2, R_2)) \text{ si } i_1 < i_2 \\
 \text{inter}((i_1 \rightarrow L_1, R_1), (i_2 \rightarrow L_2, R_2)) &= \text{inter}((i_1 \rightarrow L_1, R_1) L_2) \text{ si } i_1 > i_2
 \end{aligned}$$

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 $T(\text{allSubsets}(1000)) = 1000$.

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

Correction : La taille minimale est quatre. L'idée est de ne construire qu'une seule fois le sous-arbre qui apparaît deux fois :

```

ZDD z2 = new Znode(2, new Zero(), new One());
ZDD z  = new Znode(0, new Znode (1, z2, new One()),
                  new Znode (1, new One(), z2));

```

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 $O(T(\text{this}))$.

Correction : On écrit une méthode auxiliaire qui prend l'ensemble de type `HashSet<Znode>` en argument :

```

class ZDD {
    int size() {
        HashSet<Znode> nodes = new HashSet<>();
        this.size(nodes);
        return nodes.size();
    }
    void size(HashSet<Znode> nodes) { }
}

```

Elle ne fait rien par défaut et est seulement redéfinie dans la sous-classe `Znode` :

```
class Znode {
    void size(HashSet<Znode> nodes) {
        if (nodes.contains(this)) return; // déjà vu
        nodes.add(this);
        this.left.size(nodes);
        this.right.size(nodes);
    }
}
```

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

Correction : En soi, la question n'est pas difficile car on a

$$\begin{aligned} \text{card}(\perp) &= 0 \\ \text{card}(\top) &= 1 \\ \text{card}(i \rightarrow L, R) &= \text{card}(L) + \text{card}(R) \end{aligned}$$

La difficulté est ici de garantir la complexité $O(T(\mathbf{this}))$. Pour cela, il suffit de mémoïser le calcul, pour ne pas le refaire deux fois sur un même sous-arbre. On peut s'inspirer de ce qu'on vient de faire pour `size` :

```
class Znode {
    int card() {
        HashMap<Znode, Integer> memo = new HashMap<>();
        return this.card(memo);
    }
    abstract int card(HashMap<Znode, Integer> memo);
}
class Zero {
    int card(HashMap<Znode, Integer> memo) { return 0; } }
class Zone {
    int card(HashMap<Znode, Integer> memo) { return 1; } }
class Znode {
    int card(HashMap<Znode, Integer> memo) {
        if (memo.containsKey(this)) return memo.get(this); // déjà calculé
        int c = this.left.card(memo) + this.right.card(memo);
        memo.put(this, c);
        return c;
    }
}
```

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 $O(T(\mathbf{this}) \times T(\mathbf{z}))$. On ne demande pas d'écrire le code Java. Justifier soigneusement la complexité.

Correction : Comme pour la question précédente, on utilise le principe de mémoïsation et stockant toutes les paires d'arbres pour lesquelles on a déjà calculé l'intersection (par exemple dans une table de hachage indexée par une paire d'arbres ou plus simplement dans une table de hachage de type `HashMap<Znode, HashMap<Znode, ZDD> >`). Or, on note que le calcul de `inter(A1, A2)`

— construit au plus un arbre par appel;

— ne s'appelle récursivement que sur des sous-arbres de A_1 et A_2 .

Donc il ne peut construire au plus que $T(A_1) \times T(A_2)$ arbres distincts pour construire $T(\text{inter}(A_1, A_2))$, d'où le résultat.

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

* *
*