X2019 — INF411
Contrôle classant / 17 novembre 2020 / 9h–12h

Tous documents de cours autorisés (polycopié, transparents, notes personnelles). Le dictionnaire papier est autorisé pour les FUI et FUI-FF. Les ordinateurs, Ipad, dictionnaires électroniques, les 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.

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 Tableur

Dans ce problème, on considère la programmation d'un *tableur* (en anglais *spreadsheet*). Un tableur est une grille, dont chaque cellule contient une formule définissant sa valeur. Une formule peut faire référence à la valeur d'une autre cellule.

Pour simplifier les choses, on considère ici une grille carrée, de taille $N \times N$, avec des valeurs entières et des formules de deux sortes uniquement : soit une constante, soit la somme des valeurs de deux autres cellules. Voici un exemple d'une grille 3×3 avec les définitions de chaque cellule :

	0	1	2
0	$\stackrel{\text{def}}{=} 1$	$\stackrel{\text{def}}{=} 0$	$\stackrel{\text{def}}{=} 0$
1	$\stackrel{\text{def}}{=} 1$	$\stackrel{\text{def}}{=} C(0,0) + C(0,1)$	$\stackrel{\text{def}}{=} C(0,1) + C(0,2)$
2	$\stackrel{\text{def}}{=} 1$	$\stackrel{\text{def}}{=} C(1,0) + C(1,1)$	$\stackrel{\text{def}}{=} C(1,1) + C(1,2)$

Ici, $C(i,j)$ indique une référence à la valeur de la cellule (i,j), l'indice i désignant la ligne et l'indice j désignant la colonne, avec $0 \leq i, j < N$. Sur cet exemple, un calcul des valeurs de la grille donne le résultat suivant :

	0	1	2
0	1	0	0
1	1	1	0
2	1	2	1

Notre objectif est d'écrire un programme qui réalise le calcul des valeurs. Dans un second temps, nous ajouterons la possibilité de modifier la définition d'une cellule et de mettre alors à jour les valeurs.

Pour programmer un tel tableur en Java, on se donne les classes de la figure 1. La classe `Def` représente une définition donnée à une cellule. C'est une classe abstraite, avec deux sous-classes `Cst` et `Sum`, pour représenter respectivement les deux types de formules. Ainsi, dans notre exemple ci-dessus, la cellule $(0,0)$ a une définition du type `Cst` et la cellule $(1,1)$ a une définition du type `Sum`. La classe `Ref` représente une référence à une cellule, comme une paire de deux entiers `row` et `col` donnant respectivement la ligne et la colonne de la cellule. Cette classe `Ref` est utilisée dans la classe `Sum` pour indiquer les deux cellules dont on fait la somme. La classe `Cell` représente une cellule de la grille, avec

```
// la définition d'une cellule
abstract class Def {
  abstract int eval(Grid g);
}
// une définition est de deux sortes possibles
class Cst extends Def {
  int c;
  int eval(Grid g) { return this.c; }
}
class Sum extends Def {
  Ref r1, r2;
  int eval(Grid g) { return this.r1.eval(g) + this.r2.eval(g); }
}

// une référence à une cellule de la grille
class Ref {
  int row, col;
  int eval(Grid g) { return g.grid[this.row][this.col].eval(g); }
}

// une cellule de la grille
class Cell {
  Def def;   // la définition de cette cellule
  int value; // sa valeur, une fois calculée

  int eval(Grid g) {
    this.value = this.def.eval(g);
    return this.value;
  }
}

// l'ensemble des cellules
class Grid {
  int N;
  Cell[][] grid; // de taille NxN
}
```

Figure 1 – Classes Java représentant le tableur (les constructeurs sont omis).

sa définition (champ `def`) et sa valeur, lorsqu'elle est calculée (champ `value`). Enfin, la classe `Grid` représente une grille, comme un tableau de tableaux `grid` de taille `N` × `N`. Les constructeurs de toutes ces classes sont volontairement omis.

Comme on le voit dans la définition de la classe `Cell`, la méthode `eval` calcule la valeur de la cellule en utilisant la méthode `eval` de la définition, puis stocke le résultat dans le champ `value` de la cellule. Quand on calcule la valeur d'une somme, dans la méthode `eval` de la classe `Sum`, on utilise la méthode `eval` de la classe `Ref`, qui appelle elle-même la méthode `eval` de la classe `Cell`. Pour calculer les valeurs de toutes les cellules, on définit la méthode suivante dans la classe `Grid` :

```
void computeAll() {
  for (int r = 0; r < N; r++)
    for (int c = 0; c < N; c++)
      grid[r][c].eval(this);
}
```

On suppose pour l'instant que la grille ne contient aucune *circularité*, c'est-à-dire que la définition d'aucune cellule ne fait référence à sa propre valeur de façon directe ou indirecte.

Question 1 Combien d'additions sont effectuées pendant le calcul de `computeAll` sur la grille 3×3 donnée en exemple plus haut ?

Correction :
— 1 pour $(1, 1)$;
— 1 pour $(1, 2)$;
— 2 pour $(2, 1)$, car on refait le calcul de $(1, 1)$;
— 3 pour $(2, 2)$, car on refait le calcul de $(1, 1)$ et $(1, 2)$
D'où 7 additions au total.

Question 2 Pour une valeur arbitraire de N, donner un exemple de grille $N \times N$ dont le calcul par `computeAll` demande un nombre d'additions proportionnel à 2^N.

Correction : Une colonne suffit. Sur chaque ligne, on fait l'addition de deux fois la cellule de la ligne précédente.

0	1
1	$\stackrel{\text{def}}{=} C(0, 0) + C(0, 0)$
	$\vdots$
i	$\stackrel{\text{def}}{=} C(i - 1, 0) + C(i - 1, 0)$
	$\vdots$

Ainsi, le calcul de la ligne i demande $2^i - 1$ additions.

Mémoïsation. Pour remédier à cette possible explosion des calculs, on applique le principe de *mémoïsation* : si une valeur a déjà été calculée, on ne la recalcule pas. Pour cela, on ajoute un champ `status` à la classe `Cell`

```
class Cell {
  ...
  int status;
```

dont la valeur vaut soit `TODO` (le calcul reste à faire), soit `DONE` (le calcul est fait).

```
final static int TODO = 0, DONE = 1;
```

On suppose qu'initialement le statut de toutes les cellules vaut `TODO`.

Question 3 Modifier le code de la méthode `eval` de la classe `Cell` pour implémenter le principe de mémoïsation. (Le code de `computeAll`, en revanche, reste inchangé.) Quelle est maintenant la complexité de la méthode `computeAll` ?

Correction :

```
int eval(Grid g) {
  if (this.status == TODO) {
    this.value = this.def.eval(g);
    this.status = DONE;
  }
  return this.value;
}
```

La méthode `computeAll` a maintenant une complexité linéaire en la taille de la grille, c'est-à-dire $O(N^2)$.

Détection de circularité. On suppose maintenant que la grille peut contenir une circularité, c'est-à-dire une cellule dont la définition fait référence à sa propre valeur de façon directe ou indirecte. Ainsi, la grille suivante contient une circularité :

	0	1
0	$\stackrel{\text{def}}{=} 1$	$\stackrel{\text{def}}{=} C(0,0) + C(1,0)$
1	$\stackrel{\text{def}}{=} C(1,1) + C(0,0)$	$\stackrel{\text{def}}{=} C(0,0) + C(0,1)$

Afin de détecter une circularité, on se donne maintenant *trois* valeurs différentes pour le champ `status` d'une cellule

```
final static int TODO = 0, INPROGRESS = 1, DONE = 2;
```

où `INPROGRESS` désigne maintenant une cellule dont le calcul est en cours.

Question 4 Donner un nouveau code pour la méthode `eval` de la classe `Cell`, qui signale maintenant toute circularité rencontrée pendant le calcul (avec `throw new Error("cycle")`). Le code de `computeAll` reste toujours inchangé.

Correction :

```
int eval(Grid g) {
  if (this.status == INPROGRESS)
    throw new Error("cycle");
  if (this.status == TODO) {
    this.status = INPROGRESS;
    this.value = this.def.eval(g);
    this.status = DONE;
  }
  return this.value;
}
```

Calcul des dépendances. On souhaite maintenant pouvoir modifier le contenu d'une cellule, c'est-à-dire pouvoir lui donner une nouvelle définition, à la manière de ce que l'on fait quand on utilise un tableur interactivement. Il faut alors mettre à jour les valeurs. Bien entendu, on pourrait le faire facilement en donnant le statut `TODO` à toutes les cellules puis en appelant la méthode `computeAll`. C'est très inefficace, cependant, car on recalcule toutes les valeurs. On se propose de programmer quelque chose de potentiellement plus efficace. Pour cela, on va calculer des dépendances avant et arrière.

Calcul des dépendances avant. On commence par calculer, pour une formule donnée, les cellules dont elle dépend *directement*. Pour cela, on ajoute à la classe `Def` une méthode `depends` qui renvoie un ensemble de références.

```
abstract class Def {
  ...
  abstract HashSet<Ref> depends();
}
```

Si l'ensemble renvoyé par `depends` contient une référence $C(i, j)$, alors cela signifie que la définition contient une référence à la cellule (i, j). Il est très facile de programmer cette méthode dans chacune des deux sous-classes :

```
class Cst extends Def {
  ...
  HashSet<Ref> depends() { return new HashSet<>(); }
}
class Sum extends Def {
  ...
  HashSet<Ref> depends() {
    HashSet<Ref> s = new HashSet<>(); s.add(r1); s.add(r2); return s; }
}
```

Dans la sous-classe `Cst`, on renvoie un ensemble vide et dans la sous-classe `Sum`, on renvoie un ensemble contenant les deux références `r1` et `r2`.

Question 5 Pour pouvoir utiliser ainsi le type `HashSet<Ref>`, il faut munir la classe `Ref` de méthodes `hashCode` et `equals` adéquates. Proposer un code pour ces deux méthodes.

Correction :

```
public int hashCode() {
  return 5003 * row + col;
}
public boolean equals(Object o) {
  Ref that = (Ref)o;
  return this.row == that.row && this.col == that.col;
}
```

Calcul des dépendances arrière. Lorsqu'une cellule est modifiée, et que sa valeur change, il faut recalculer la valeur des cellules qui en dépendent. Ainsi, dans l'exemple donné en introduction page 6, lorsque la valeur de la cellule $(0,1)$ change, il faut recalculer les valeurs des cellules $(1,1)$ et $(1,2)$ qui en dépendent (et ensuite recalculer les valeurs des cellules $(2,1)$ et $(2,2)$ qui en dépendent à leur tour). Pour retrouver efficacement les cellules qui dépendent d'une cellule donnée, on ajoute un champ `pred` à la classe `Cell` :

```
class Cell {
  ...
  HashSet<Ref> pred;
}
```

Ce champ contient l'ensemble des références aux cellules qui dépendent *directement* de la cellule `this`. Dit autrement, si l'ensemble `pred` contient une référence $C(i,j)$, alors cela signifie que la formule définissant la cellule (i,j) contient une référence à la cellule `this`.

Question 6 Donner les ensembles `pred` pour toutes les cellules de la grille 3×3 de la page 6.

Correction :

	0	1	2
0	$\{C(1,1)\}$	$\{C(1,1), C(1,2)\}$	$\{C(1,2)\}$
1	$\{C(2,1)\}$	$\{C(2,1), C(2,2)\}$	$\{C(2,2)\}$
2	$\emptyset$	$\emptyset$	$\emptyset$

Modification d'une cellule. Pour modifier la définition d'une cellule, on ajoute une méthode `update` dans la classe `Grid` :

```
class Grid { ...
  void update(int r, int c, Def def) {
    ???                             // mettre à jour les champs .pred
    this.grid[r][c].def = def;      // changer la définition
    this.grid[r][c].update(this); // mettre à jour les valeurs
  }
}
```

Cette méthode prend en arguments une ligne `r`, une colonne `c` et une définition `def`, et elle donne à la cellule (`r`, `c`) la définition `def`. Elle commence par mettre à jour les champs `pred`, puis elle change la définition, puis enfin elle met à jour la valeur, en appelant une méthode `update` de la classe `Cell` que l'on écrira plus loin.

Question 7 Donner le code qui manque dans la méthode `update` de la classe `Grid` (ici représenté par `???`).

Correction : Il faut bien penser à supprimer les anciennes dépendances avant d'ajouter les nouvelles :

```
void update(int r, int c, Def def) {
  Ref rc = new Ref(r, c);
  for (Ref dep: this.grid[r][c].def.depends())
    this.grid[dep.row][dep.col].pred.remove(rc);
```

```
    for (Ref dep: def.depends())
      this.grid[dep.row][dep.col].pred.add(rc);
    ...
  }
```

Mise à jour des valeurs. Il ne nous reste plus qu'à mettre à jour les valeurs. On le fait avec une méthode récursive `update` dans la classe `Cell`.

```
class Cell { ...
  int eval(Grid g) {
    if (this.status == INPROGRESS)
      throw new Error("cycle!");
    return this.value;
  }
  void update(Grid g) {
    this.status = INPROGRESS;
    // 1. mettre à jour la valeur de la cellule this
    ???
    // 2. si elle a changé, mettre à jour les valeurs
    //    des cellules qui en dépendent
    ???
  }
}
```

On note que la méthode `eval` est toujours là (c'est nécessaire pour que la méthode `eval` de la classe `Ref` fonctionne toujours) mais que son code est grandement simplifié. En effet, on suppose maintenant que les valeurs sont toujours à jour, sauf si elles sont en cours de calcul. En particulier, on n'utilise plus du tout la méthode `computeAll` et on ne se sert plus du statut `TODO`. Le statut d'une cellule est soit `DONE`, lorsqu'elle est à jour, soit `INPROGRESS` lorsqu'elle est en cours de mise à jour.

Question 8 Donner le code qui manque dans la méthode `update` de la classe `Cell` (ici représenté par `???`).

Correction :

```
void update(Grid g) {
  this.status = INPROGRESS;
  int old = this.value;
  this.value = this.def.eval(g);
  this.status = DONE;
  if (this.value != old) // optim
    for (Ref d: this.pred)
      g.grid[d.row][d.col].update(g);
}
```

Question 9 Montrer que la mise à jour d'une cellule peut entraîner un calcul qui demande un nombre d'additions proportionnel à 2^N. Proposer une solution pour que le calcul impliqué par une mise à jour soit en $O(N^2)$ dans le pire des cas. On ne demande pas d'écrire le code Java correspondant.

Correction : Si on a une dépendance en losange, comme ceci

	$j-1$	j	$j+1$
i		?	
$i+1$	$C(i,j)+\dots$		$C(i,j)+\dots$
$i+2$		$C(i+1,j-1)+C(i+1,j+1)$	

alors la modification de la cellule (i, j) entraîne la mise à jour de la cellule $(i + 1, j - 1)$ et de la cellule $(i + 1, j + 1)$, qui *toutes les deux* entraînent la mise à jour de la cellule $(i + 2, j)$, qui est donc recalculée deux fois. En enchaînant de tels losanges plusieurs fois, on aboutit à un calcul exponentiel.

Bien entendu, une solution triviale pour rester en $O(N^2)$ consisterait à revenir à la solution précédente avec `computeAll`. Plus subtilement, on peut

1. mettre à `TODO` toutes les cellules qui dépendent de la cellule modifiée, récursivement ;
2. modifier la méthode `update` de la classe `Cell` pour qu'elle ne fasse rien sur une cellule de statut `DONE`.

Mais on perd alors la possibilité de s'arrêter lorsqu'une valeur recalculée ne change pas.

Question 10 On souhaite étendre notre langage de formules avec une troisième sorte de formule, de la forme

$$\texttt{if } C(i_1, j_1) = 0 \texttt{ then } C(i_2, j_2) \texttt{ else } C(i_3, j_3)$$

c'est-à-dire une formule permettant de tester si la valeur de la cellule $C(i_1, j_1)$ vaut 0 et, selon le cas, de sélectionner la valeur de la cellule $C(i_2, j_2)$ ou bien celle de la cellule $C(i_3, j_3)$.

Donner tout le code qu'il est nécessaire d'ajouter ou de modifier pour disposer maintenant de cette troisième sorte de formule.

Correction : Il suffit d'ajouter une troisième sous-classe `Ifz` à la classe `Def`, correspondant à ce nouveau type de formule. Cette nouvelle classe s'implémente facilement :

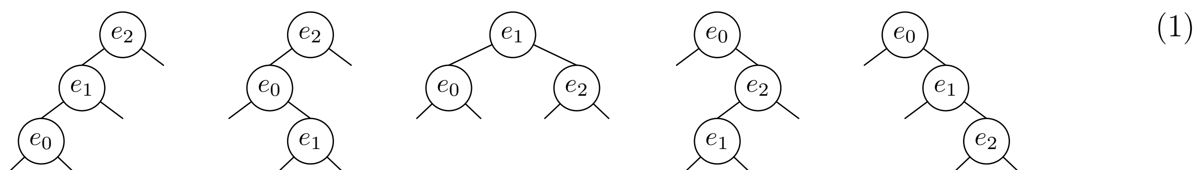
```
class Ifz extends Def {
  Ref r1, r2, r3;
  int eval(Grid g) {
    return this.r1.eval(g) == 0 ? this.r2.eval(g) : this.r3.eval(g); }
  HashSet<Ref> depends() {
    Set<Ref> s = new HashSet<>(); s.add(r1); s.add(r2); s.add(r3); return s; }
}
```

Aucune autre classe n'a besoin d'être modifiée.

---

2 Arbres binaires de recherche optimaux

Dans ce problème, on étudie la construction d'un arbre binaire de recherche pour un ensemble de N éléments donnés $e_0 < e_1 < \cdots < e_{N-1}$. Voici par exemple tous les arbres binaires de recherche contenant trois éléments $e_0 < e_1 < e_2$.



(1)

Notre objectif est de choisir, parmi tous les arbres binaires de recherche possibles pour $e_0 < e_1 < \cdots < e_{N-1}$, celui qui sera optimal dans un sens que nous définirons plus loin. Pour les questions de programmation, on suppose que les éléments sont d'un certain type `E`, non précisé, et qu'un arbre binaire de recherche est du type suivant

```
class BST {
  E   value;
  BST left, right;
  BST(BST left, E value, BST right) {
    this.left = left; this.value = value; this.right = right;
  }
}
```

où la valeur `null` est utilisée pour représenter un arbre vide.

Question 11 Donner une méthode `static BST ofArray(E[] e)` qui reçoit en argument un tableau contenant N éléments $e_0 < e_1 < \cdots < e_{N-1}$ et qui renvoie un arbre binaire de recherche contenant ces éléments et de hauteur $O(\log N)$. La complexité doit être $O(N)$.

Indication : Couper le tableau en deux moitiés égales et procéder récursivement.

Correction : La seule difficulté consiste à ne pas faire d'extraction de sous-tableaux afin de garantir une complexité linéaire. Pour cela, on passe en arguments les indices de la portion de tableau à considérer :

```
static BST ofArray(E[] e) {
  return ofArray(e, 0, e.length);
}
static BST ofArray(E[] e, int lo, int hi) {
  if (lo >= hi) return null;
  int mid = lo + (hi - lo) / 2;
  return new BST(ofArray(e, lo, mid),
                 e[mid],
                 ofArray(e, mid + 1, hi));
}
```

Une notion de coût. Étant donné un arbre binaire de recherche t contenant $e_0 < e_1 < \cdots < e_{N-1}$, la recherche de l'élément e_i a un coût directement proportionnel à la profondeur de e_i dans cet arbre.

Ainsi, lorsque l'on cherche un élément dans l'arbre

(2)

on fait une seule comparaison s'il s'agit de e_2, deux comparaisons s'il s'agit de e_0 et trois comparaisons s'il s'agit de e_1. On suppose par ailleurs que l'on sait estimer la fréquence $f(e_i)$ avec laquelle l'élément e_i sera recherché dans l'arbre t. On définit alors le *coût* de l'arbre t comme

$$\text{coût}(t) \stackrel{\text{def}}{=} \sum_{i=0}^{N-1} f(e_i) \times (\text{profondeur de } e_i \text{ dans } t)$$

avec la convention que la profondeur de la racine est 1. Ainsi, le coût de l'arbre (2) ci-dessus est $2f(e_0) + 3f(e_1) + f(e_2)$. Un arbre binaire de recherche *optimal* pour $e_0 < e_1 < \cdots < e_{N-1}$ est un arbre de coût minimal.

Question 12 En supposant $f(e_0) = 1$, $f(e_1) = 2$ et $f(e_2) = 3$, quels sont, parmi les cinq arbres donnés en (1), ceux qui sont de coût minimal ?

Correction : On trouve respectivement :

$$\begin{array}{rcl} 3f(e_0) + 2f(e_1) + f(e_2) & = & 3 + 4 + 3 = 10 \\ 2f(e_0) + 3f(e_1) + f(e_2) & = & 2 + 6 + 3 = 11 \\ 2f(e_0) + f(e_1) + 2f(e_2) & = & 2 + 2 + 6 = 10 \\ f(e_0) + 3f(e_1) + 2f(e_2) & = & 1 + 6 + 6 = 13 \\ f(e_0) + 2f(e_1) + 3f(e_2) & = & 1 + 4 + 9 = 14 \end{array}$$

Le premier et le troisième sont donc de coût minimal 10.

Question 13 Dans le cas particulier où $f(e_i) = 1$ pour tout $0 \leq i < N$, quelle est la forme d'un arbre optimal pour $e_0 < e_1 < \cdots < e_{N-1}$? L'arbre construit par votre méthode `ofArray` de la question 11 est-il optimal dans ce cas particulier ?

Correction : Soit t un arbre optimal et h sa hauteur, de sorte que les éléments sont situés à des profondeurs $1, 2, \ldots, h$. S'il n'y a pas exactement 2^{p-1} nœuds à la profondeur p, pour $p < h$, alors on pourrait remonter un nœud situé plus bas à cette profondeur et obtenir alors un arbre de coût inférieur. Dès lors, tous les niveaux de l'arbre t sont complets, sauf éventuellement le dernier.

L'arbre renvoyé par `ofArray` a cette propriété. Montrons-le par récurrence forte sur N. Pour $N = 0$, c'est vrai. Pour $N > 0$, on construit récursivement deux arbres avec `ofArray`, avec respectivement $N_1 = \lfloor (N-1)/2 \rfloor$ et $N_2 = \lceil (N-1)/2 \rceil$ éléments. Par hypothèse de récurrence, ces deux arbres sont complets. Si leurs hauteurs diffèrent, on a nécessairement $N_2 = 2^h$ et $N_1 = 2^h - 1$ car $0 \leq N_2 - N_1 \leq 1$, et l'arbre renvoyé par `ofArray` est également complet, et donc optimal.

Calcul du coût. On suppose que la fréquence $f(e)$ d'un élément e est un entier stocké dans un champ `freq` de la classe `E` :

```
class E { int freq; ... }
```

Question 14 Écrire une méthode `static int cost(BST t)` qui renvoie le coût de l'arbre `t`. On garantira une complexité $O(N)$ où N est le nombre de nœuds de l'arbre `t`.

Correction :

```
static int cost(BST t) {
  return cost(1, t);
}
static int cost(int d, BST t) {
  if (t == null) return 0;
  return d * t.value.freq + cost(d+1, t.left)+ cost(d+1, t.right);
}
```

Construction d'un arbre optimal. On cherche maintenant à construire un arbre binaire de recherche optimal pour $e_0 < e_1 < \cdots < e_{N-1}$. Pour $0 \le i \le j \le N$, on note $c(i,j)$ le coût d'un arbre optimal pour les éléments $e_i, e_{i+1}, \ldots, e_{j-1}$ et on pose $w(i,j) = f(e_i) + f(e_{i+1}) + \cdots + f(e_{j-1})$.

Question 15 Montrer que

$$\begin{array}{rcll} c(i,i) & = & 0 & \text{pour } 0 \le i \le N, \\ c(i,j) & = & w(i,j) + \min_{i \le k < j}\left(c(i,k) + c(k+1,j)\right) & \text{pour } 0 \le i < j \le N. \end{array}$$

Correction : Un arbre qui ne contient aucun élément a un coût égal à 0 et donc $c(i,i) = 0$. Pour $i < j$, la racine de l'arbre est e_k pour un certain $i \le k < j$, avec un sous-arbre gauche de coût $c(i,k)$ et un sous-arbre droit de coût $c(k+1,j)$. Dans le calcul du coût $c(i,j)$,

— la racine e_k contribue pour $f(e_k)$;

— chaque élément e_l pour $i \le l < j$ et $l \ne k$ apparaît à une profondeur un de plus que dans le calcul de $c(i,k)$ et $c(k+1,j)$, d'où l'ajout de $f(e_l)$.

D'où l'addition de $w(i,j)$.

Question 16 Déduire de la question précédente une méthode `static BST optimum(E[] e)` qui reçoit en argument un tableau contenant N éléments $e_0 < e_1 < \cdots < e_{N-1}$ et qui renvoie un arbre binaire de recherche optimal pour ces éléments, en temps $O(N^3)$. Quelle est la complexité en espace de votre code ?

Indication : Utiliser trois tableaux pour stocker respectivement la somme $w(i,j)$, le coût $c(i,j)$ et un arbre de coût $c(i,j)$ pour $e_i, \ldots, e_{j-1}$. Remplir ces trois tableaux pour $j - i$ croissant.

Correction :

```
BST optimum(E[] e) {
  int n = e.length;
  BST[][] opt = new BST[n+1][n+1];
  int[][] c   = new int [n+1][n+1];
  int[][] w   = new int [n+1][n+1];
  for (int len = 1; len <= n; len++)
    for (int i = 0; i <= n - len; i++) {
      int j = i + len;
      w[i][j] = w[i][j-1] + e[j-1].freq;
      c[i][j] = w[i][j] + c[i+1][j];
      int kmin = i;
      for (int k = i + 1; k < j; k++) { // la racine est k
        int cr = w[i][j] + c[i][k] + c[k+1][j];
        if (cr < c[i][j]) {
          c[i][j] = cr;
          kmin = k;
        }
      }
      opt[i][j] = new BST(opt[i][kmin], e[kmin], opt[kmin+1][j]);
    }
  return opt[0][n];
}
```

Le code alloue trois tableaux, chacun de taille $O(N^2)$. Par ailleurs, le code alloue $O(N^2)$ objets de la classe `BST` (au plus un par case du tableau `opt`). La complexité en espace est donc $O(N^2)$.

Question 17 Pour $0 \leq i < j \leq N$, on note $r(i,j)$ l'indice de la racine d'un arbre optimal pour les éléments $e_i, \ldots, e_{j-1}$ c'est-à-dire un entier $i \leq k < j$ qui minimise $c(i,k) + c(k+1,j)$. On admet que

$$r(i, j-1) \leq r(i,j) \leq r(i+1,j)$$

dès lors que les fréquences sont positives ou nulles. Montrer comment cela permet d'améliorer la complexité de la fonction `optimum`. On ne demande pas d'écrire le code Java correspondant.

Correction : On peut conserver la valeur trouvée pour $r(i,j)$ dans un quatrième tableau. Avec l'inégalité sur les $r(i,j)$, on peut limiter la recherche de k dans boucle interne de la fonction `optimum` entre $r(i,j-1)$ et $r(i+1,j)$. Pour $d = j - i$ donné, le temps total passé dans les deux boucles internes est alors

$$\begin{aligned} \sum_{\substack{0 \leq i \leq N-d \\ j=i+d}} (r(i+1,j) - r(i,j-1) + 1) &= r(N-d+1, N) - r(0, d-1) + N - d + 1 \\ &< 2N \end{aligned}$$

(somme téléscopique). On en déduit que la complexité est maintenant $O(N^2)$.

A Bibliothèque standard Java

`class HashSet<E>`	un ensemble dont les éléments sont de type `E`
`HashSet()`	renvoie un nouvel ensemble, vide
`void add(E x)`	ajoute l'élément `x`
`void remove(E x)`	supprime l'élément `x`
`boolean contains(E x)`	indique la présence de l'élément `x`
`int size()`	renvoie le cardinal de l'ensemble

On peut parcourir tous les éléments d'un ensemble `s` avec la construction
`for (E x: s) ...`

* *
*