X2024 — CSC_41011
Contrôle classant / 12 novembre 2025 / 9h–12h

Polycopié et notes personnelles sont autorisés. 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.

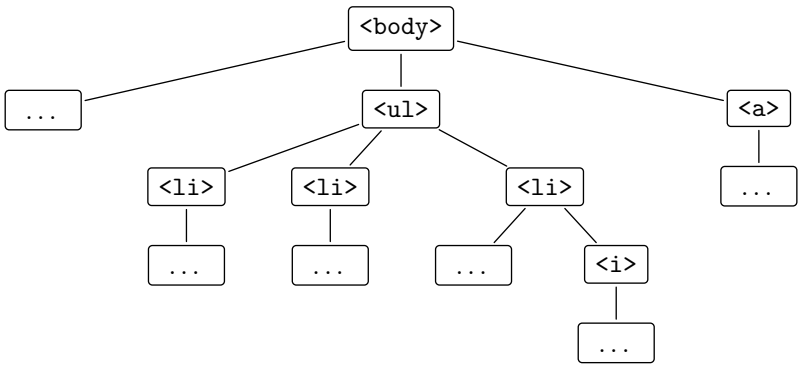
Les questions de code n'appellent aucune justification. *Une indication du nombre de lignes attendues est donnée entre parenthèses. Quelques éléments de la bibliothèque standard Java sont rappelés à la fin du sujet. Ils ne sont pas forcément tous nécessaires. Vous pouvez utiliser librement tout autre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

1 Langages à balises

Les langages à balises comme HTML permettent de décrire des documents structurés sous une forme textuelle, à l'aide de *balises* ouvrantes et fermantes comme `<body>` et `</body>`. Toute balise ouvrante `<x>` est associée à une balise fermante `</x>` située plus loin. Les balises sont *bien parenthésées*, en ce sens qu'une balise `<y>` ouverte à l'intérieur d'une balise `<x>` sera fermée (par `</y>`) avant que la balise `<x>` ne soit fermée (par `</x>`). Entre les balises se trouvent des morceaux de texte arbitraires. Voici un exemple de document structuré :

```
<body>
  ...
  <ul><li> ... </li><li> ... </li><li> ... <i> ... </i></li></ul>
  <a> ... </a>
</body>
```

Les morceaux de texte sans balise sont notés « `...` » et on s'est autorisé des retours chariot entre les balises pour la lisibilité. Un tel document structuré peut être vu comme un *arbre* :



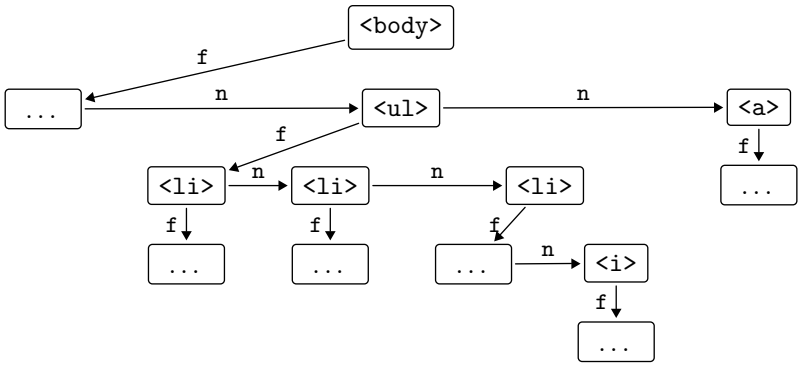
Les nœuds internes correspondent à des balises et les feuilles à des morceaux de texte. Dans ce problème, on étudie une structure de données Java pour représenter de tels arbres. La figure 1 contient une classe `Node` pour les nœuds de nos arbres. Le champ `text` contient le nom de la balise

```
class Node {
  final String text;
  Node first, next;
  int pre, post, level;

  Node(String text) { this.text = text; }
}
```

FIGURE 1 – Représentation des arbres.

(sans les < >) pour un nœud interne et le morceau de texte pour une feuille. Le champ `first` contient un pointeur vers le premier sous-arbre, lorsque le nœud est interne, et vaut `null` pour une feuille. Le champ `next` contient un pointeur vers le sous-arbre suivant lorsque le nœud parent contient plusieurs sous-arbres. Le rôle des champs entiers `pre`, `post` et `level` sera expliqué plus tard. Avec cette représentation, voici comment l'arbre donné en exemple plus haut est représenté par 13 objets de la classe `Node`,



où un pointeur `first` non nul est noté $\xrightarrow{f}$ et un pointeur `next` non nul est noté $\xrightarrow{n}$. On note que, si une feuille n'a pas de pointeur `first`, elle peut en revanche avoir un pointeur `next`, comme on le voit ci-dessus à deux endroits. Dans tout ce problème, on suppose que toute balise contient au moins un sous-arbre (les nœuds avec `first` $\neq$ `null` sont donc exactement les balises) et que la racine de l'arbre a toujours un champ `next` qui vaut `null`.

On dit qu'un nœud x est le *parent* d'un nœud y si on peut aller du nœud x au nœud y en suivant exactement un pointeur `first` puis un nombre arbitraire de pointeurs `next` (y compris zéro). On dit qu'un nœud z est le *descendant* d'un nœud x si $z = x$ ou si z est le descendant d'un nœud y avec x le parent de y. En particulier, un nœud est son propre descendant.

Question 1 Écrire une méthode `void addLast(Node n)` dans la classe `Node` qui ajoute le nœud `n` à la fin de la liste des sous-arbres du nœud `this`. (Si le nœud `this` n'a aucun sous-arbre, alors `n` est ajouté comme son seul sous-arbre.) On suppose `n.next` = `null`. (4 lignes de code)

Correction :

```
void addLast(XMLnode n) {
  if (this.first == null) { this.first = n; return; }
  Node p = this.first;
  while (p.next != null) p = p.next;
  p.next = n;
}
```

Question 2 Écrire une méthode *récursive* `static void print(Node n)` qui imprime la forme textuelle de l'arbre de racine `n` (c'est-à-dire qui imprime la chaîne `<body>...<ul><li>...</li>` etc. sur notre exemple). Le contenu de `n.next` doit être ignoré. (6 lignes de code)

Correction :

```
static void print(Node n) {
  if (n == null) return;
  if (n.first == null) { System.out.print(n.text); return; }
  System.out.print("<" + n.text + ">");
  for (Node c = n.first; c != null; c = c.next)
    print(c);
  System.out.print("</" + n.text + ">");
}
```

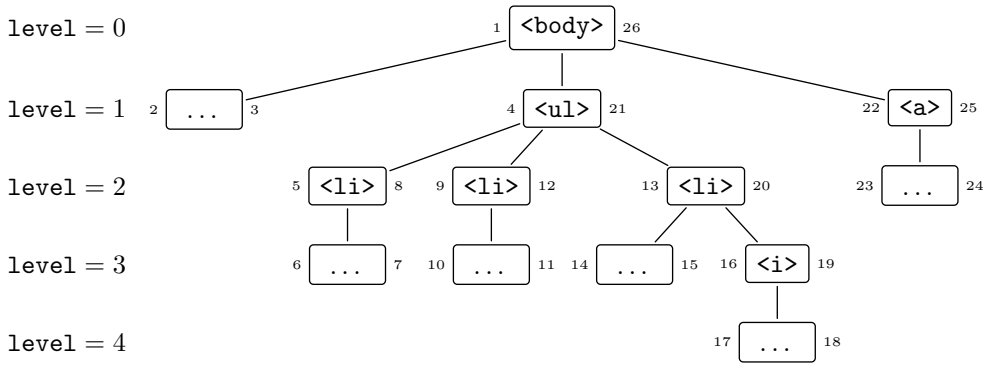
Question 3 Écrire une méthode *non récursive* `static int size(Node n)` qui renvoie le nombre de nœuds de l'arbre de racine `n`. Le contenu de `n.next` doit être ignoré. La complexité doit être proportionnelle au résultat, et il est demandé de la justifier. Indication : Utiliser une pile contenant des nœuds dont il faut calculer la taille. (10 lignes de code)

Correction :

```
int treeSize(Node n) { // suppose n != null
  int count = 1;
  Stack<XMLnode> st = new Stack<>();
  st.push(n.first);
  while (!st.isEmpty()) {
    Node n = st.pop();
    if (n == null) continue;
    count++;
    st.push(n.next);
    st.push(n.first);
  }
  return count;
}
```

Les opérations `push` et `pop` s'exécutent en temps constant (amorti) et la complexité est donc clairement en temps proportionnel au résultat car `count` est incrémenté à chaque tour de boucle.

Numérotation des nœuds de l'arbre. Pour les besoins des questions suivantes, nous allons associer trois entiers à chaque nœud : sa profondeur dans l'arbre (champ `level`) et deux entiers (champs `pre` et `post`) obtenus par un *parcours en profondeur*. Dans ce parcours, les entiers `pre` et `post` sont attribués dans l'ordre croissant à partir de 1. Pour chaque nœud, on commence par lui attribuer le prochain entier dans le champ `pre`, puis on parcourt ses sous-arbres, puis on lui attribue le prochain entier dans le champ `post`. Sur l'arbre pris en exemple, on obtient la numérotation suivante



où le champ `pre` est indiqué à gauche de chaque nœud et le champ `post` à droite. Le champ `level` de chaque nœud est indiqué sur la gauche.

Question 4 Écrire une méthode `int num(int p, int lvl)` dans la classe `Node` qui réalise la numérotation du sous-arbre `this` à partir de l'entier `p` et de la profondeur `lvl`, en donnant des valeurs aux trois champs `pre`, `post` et `level` de tous les nœuds descendants de `this`. Indication : procéder récursivement et renvoyer le prochain entier disponible pour la suite de la numérotation. Ainsi, si `body` désigne la racine de notre arbre, l'appel `body.num(1, 0)` doit donner la numérotation illustrée ci-dessus et renvoyer 27. (6 lignes de code)

Correction :

```
int num(int p, int l) {
  this.pre = p++;
  this.level = l;
  for (XMLnode n = this.first; n != null; n = n.next)
    p = n.num(p, l+1);
  this.post = p++;
  return p;
}
```

Question 5 Montrer qu'un nœud y est le descendant d'un nœud x si et seulement si on a les inégalités $x.\texttt{pre} \leq y.\texttt{pre}$ et $y.\texttt{post} \leq x.\texttt{post}$.

Correction : Si le nœud y est le descendant du nœud x, alors le parcours du nœud y sera commencé après celui de x et terminé avant, par propriété du parcours en profondeur. Il s'en suit les deux inégalités.

Question 6 Donner de même une formule permettant de déterminer en temps constant si le nœud x est le parent du nœud y. On ne demande pas de justification.

Correction : Il suffit d'ajouter une condition sur les profondeurs :

$$x.\texttt{pre} < y.\texttt{pre} \wedge y.\texttt{post} < x.\texttt{post} \wedge y.\texttt{level} = x.\texttt{level} + 1$$

Question 7 Soit x un nœud de l'arbre. Montrer que la taille du sous-arbre de racine x est égale à $(x.\texttt{post} - x.\texttt{pre} + 1)/2$.

Correction : On montre la propriété

$$\texttt{post} = \texttt{pre} + 2 \times \texttt{size} - 1$$

par récurrence sur la structure de l'arbre. Pour une feuille, on a $\texttt{size} = 1$ et $\texttt{post} = \texttt{pre} + 1$. Sinon, pour un nœud interne ayant k sous-arbres $T_1, T_2, \ldots, T_k$, on a la propriété vérifiée pour chacun par hypothèse de récurrence, et par ailleurs

- $T_1.\texttt{pre} = \texttt{pre} + 1$
- $T_{i+1}.\texttt{pre} = T_i.\texttt{post} + 1$ pour tout $i < k$
- $\texttt{post} = T_k.\texttt{post} + 1$

D'où

$$\begin{aligned} \texttt{post} &= \texttt{pre} + 2 \times (|T_1| + |T_2| + \cdots + |T_k|) + 2 - 1 \\ &= \texttt{pre} + 2 \times \texttt{size} - 1 \end{aligned}$$

Indexation et requêtes. On souhaite pouvoir effectuer efficacement des requêtes sur notre arbre de la forme « trouver tous les nœuds qui sont des balises `<foo>` qui se trouvent être des descendants de balises `<bar>` ». Pour cela, on se donne un dictionnaire

```
HashMap<String, Vector<Node>> byPre = new HashMap<>();
```

qui associe à chaque nom de balise l'ensemble des nœuds internes de l'arbre portant exactement ce nom de balise. Cet ensemble est représenté par un tableau redimensionnable (`Vector`) *trié par ordre croissant* de l'entier contenu dans le champ `pre`.

Question 8 Expliquer comment remplir le dictionnaire `byPre` ci-dessus avec une complexité linéaire en la taille de l'arbre. On ne demande pas d'écrire le code Java. Mais on demande de justifier soigneusement la complexité de l'algorithme proposé.

Correction : On parcourt l'arbre en profondeur et, pour chaque nœud,

1. s'il n'a pas encore d'entrée dans le dictionnaire, on l'y ajoute ; c'est de temps constant (chercher/ajouter dans une table de hachage + créer un vecteur vide) ;
2. on l'ajoute à la fin du vecteur lui correspondant dans le dictionnaire ; c'est également de temps constant.

La complexité totale est donc bien linéaire.

Vu que l'on parcourt l'arbre en profondeur, les nœuds seront bien visités par ordre croissant du champ `pre`, et les vecteurs donc bien triés au final.

Question 9 Écrire une méthode `Vector<Node> queryBelow(String a, String b)` qui renvoie tous les nœuds de l'arbre correspondant à des balises de nom `b` se trouvant être des descendants de balises de nom `a`. On essayera d'exploiter au mieux le dictionnaire `byPre` d'une part, et le caractère trié des vecteurs qu'il contient d'autre part.

Indication : On pourra supposer que les valeurs de type `Node` sont comparées selon la valeur du champ `pre`, et utiliser alors la méthode `Collections.binarySearch(v, x)` qui recherche dans un

vecteur `v` de nœuds, supposé trié par ordre croisant, la position i du premier nœud qui a la même valeur de champ `pre` que le noeud `x`. S'il n'y a pas de tel nœud, la valeur renvoyée est $-i-1$ avec i la position où il faudrait insérer `x` dans `v` pour le maintenir trié. (8 lignes de code)

Correction : On fait une boucle sur les `a` et, pour chacun, on ne considère que les `b` qui peuvent être leur descendants :

```
Vector<Node> query(String a, String b) {
  Vector<Node> result = new Vector<>();
  Vector<Node> bPre = byPre.get(b);
  for (Node n: byPre.get(a)) {
    int i = Collections.binarySearch(bPre, n));
    if (i < 0) i = -i - 1;
    for (int j = i; j < bPre.size() && bPre.get(j).pre < n.post; j++)
      result.add(bPre.get(j));
  }
  return result;
}
```

Analyse syntaxique. Dans cette dernière question, on souhaite réaliser l'analyse d'une chaîne de caractères contenant des balises, pour construire l'arbre correspondant. Ainsi, la chaîne de caractères

```
<body>...<ul><li>...</li><li>...</li><li>...<i>...</i></li></ul><a>...</a></body>
```

doit être analysée avec succès pour construire l'arbre utilisé plus haut en exemple. On suppose qu'une première partie du travail a déjà été effectuée pour découper la chaîne selon les balises, sous la forme d'une liste d'éléments de la classe `Token` suivante :

```
enum Kind { Text, Open, Close }
class Token {
  final String text;
  final Kind   kind;
}
```

Un élément est un morceau de texte sans balises (`Text`), une balise ouvrante (`Open`) ou une balise fermante (`Close`). Pour une balise ouvrante ou fermante, le champ `text` contient son nom (sans les `< </ >`).

Question 10 Écrire une méthode `Node parse(LinkedList<Token> input)` qui reçoit en entrée une liste d'éléments et renvoie l'arbre correspondant, s'il existe, et échoue en levant une exception (par exemple avec `throw new Error()`) si les balises ne sont pas correctement parenthésées. Indication : Utiliser une pile contenant les nœuds internes depuis la racine jusqu'à la position courante. (17 lignes de code)

Correction :

```
static Node parse(LinkedList<Token> input) {
   Node root = new Node("root");
   Stack<Node> st = new Stack<>();
   st.push(root);
```

```
  for (Token t: input) {
    switch (t.kind) {
      case Open:
        Node n = new Node(t.text);
        st.peek().addLast(n);
        st.push(n);
        break;
      case Close:
        if (!t.text.equals(st.peek().text)) throw new Error("balises mal parenthésé
        st.pop();
        break;
      case Text:
        st.peek().addLast(new Node(t.text));
        break;
    }
  }
  if (st.size() != 1) throw new Error("balise non fermée");
  return root.first;
}
```

```
class Edge implements Comparable<Edge> {
  final int src, dst;     // un arc entre src et dst
  final double weight;    // le poids de cet arc
}

class Graph {
  final int N;            // les sommets sont les entiers 0,1,...,N-1
  LinkedList<Edge>[] adj; // un tableau de taille N
                          // adj[i] est la liste des arcs issus de i
}
```

FIGURE 2 – Représentation des graphes.

2 Algorithme de Prim

Dans ce problème, on s'intéresse à des graphes *non orientés* dont les arcs sont étiquetés par des *poids*. On ne considère que des graphes *connexes*, c'est-à-dire où il existe toujours un chemin entre deux sommets donnés, et *sans boucles*, c'est-à-dire sans arc $a - a$.

En Java, on choisit de représenter de tels graphes à l'aide de deux classes, `Edge` et `Graph`, données figure 2. Les sommets d'un graphe sont représentés par les entiers 0,1,...,`N` -1 où `N` est le nombre de sommets, donné par le champ `N`. La structure du graphe est donnée par le tableau `adj`, où `adj[`i`]` est la liste des arcs issus du sommet i. Un arc est décrit par un objet de la classe `Edge`, où les champs `src` et `dst` sont les deux extrémités de l'arc et où le champ `weight` est le poids de l'arc. On note `src` $\overset{\texttt{weight}}{\longrightarrow}$ `dst` un tel objet. Les poids sont ici des nombres flottants, de type `double`. Les arcs sont comparés selon leur poids (la méthode `compareTo`, évidente, est omise).

Le graphe étant non orienté, un arc $a - b$ entre les sommets a et b apparaît à la fois comme un arc issu de a, c'est-à-dire un objet e de type `Edge` dans la liste `adj[`a`]`, avec e`.src` $= a$ et e`.dst` $= b$, et comme un arc issu de b, c'est-à-dire un *autre* objet e' de type `Edge` dans la liste `adj[`b`]`, avec e'`.src` $= b$ et e'`.dst` $= a$. Ces deux objets représentent le *même arc*. En particulier, on a donc e`.weight` $= e'$`.weight`.

Arbre couvrant minimal. Étant donné un graphe G, un *arbre couvrant minimal* de G est un sous-ensemble T d'arcs de G tel que

1. T est connexe et sans cycle (c'est pourquoi on parle d'*arbre*) ;
2. chaque sommet de G est l'extrémité d'au moins un arc de T (c'est pourquoi on dit que T *couvre* tous les sommets de G) ;
3. la somme des poids des arcs de T est minimale.

Voici par exemple un graphe de six sommets à gauche et un arbre couvrant minimal de ce graphe à droite (dont le poids total vaut 12).

Notre objectif dans ce problème est de calculer un arbre couvrant minimal. Le polycopié contient une solution (l'algorithme de Kruskal, qu'il n'est pas nécessaire de lire) et nous en étudions ici une autre : l'algorithme de Prim.

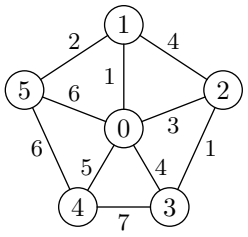
La figure 3 contient le code Java de cet algorithme. On utilise d'une part un tableau de booléens `marked` pour marquer les sommets déjà atteints (ligne 3) et d'autre part une file de priorité `pq` contenant des arcs (ligne 4). Initialement, le sommet 0 est marqué et ses arcs sortants sont ajoutés

```
1  static LinkedList<Edge> prim(Graph g) {
2    LinkedList<Edge> result = new LinkedList<>();
3    boolean marked[] = new boolean[g.N];
4    PriorityQueue<Edge> pq = new PriorityQueue<>();
5    pq.addAll(g.adj[0]); marked[0] = true;
6    while (!pq.isEmpty()) {
7      Edge e = pq.remove();
8      int v = e.dst;
9      if (marked[v]) continue;
10     marked[v] = true;
11     result.add(e);
12     for (Edge ev: g.adj[v])
13       if (!marked[ev.dst])
14         pq.add(ev);
15   }
16   return result;
17 }
```

FIGURE 3 – Code de l'algorithme de Prim.

à la file de priorité (ligne 5). Puis une boucle `while` examine tour à tour les arcs sortants de la file de priorité (lignes 6–15). Pour un arc `e` (ligne 7) arrivant sur le sommet `v` (ligne 8), on commence par regarder si le sommet est déjà marqué ; si oui, on ne fait rien (ligne 9). Sinon, on marque ce sommet (ligne 10), on ajoute l'arc au résultat (ligne 11) puis on ajoute les arcs sortants de `v` à la file de priorité (lignes 12–14). (La ligne 13 évite d'ajouter inutilement à la file de priorité un arc menant à un sommet déjà marqué, même si ce ne serait pas incorrect.)

Question 11 Décrire les itérations successives de la boucle `while` de la méthode `prim` sur le graphe suivant



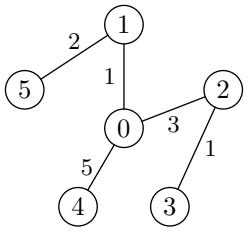
sous la forme d'un tableau

étape	arc $u \xrightarrow{w} v$	action
1	$0 \xrightarrow{1} 1$	ajouté
2	...	...
⋮	⋮	⋮

indiquant, pour chaque arc $u \xrightarrow{w} v$ retiré de la file de priorité, l'action qui est effectuée : arc ignoré (ligne 9) ou arc ajouté au résultat (lignes 10–14). Dessiner ensuite l'arbre couvrant renvoyé au final par la méthode `prim`.

Correction :

étape	arc u $\xrightarrow{w}$ v	action
1	$0 \xrightarrow{1} 1$	ajouté
2	$1 \xrightarrow{2} 5$	ajouté
3	$0 \xrightarrow{3} 2$	ajouté
4	$2 \xrightarrow{1} 3$	ajouté
5	$0 \xrightarrow{4} 3$	ignoré
6	$1 \xrightarrow{4} 2$	ignoré
7	$0 \xrightarrow{5} 4$	ajouté
8	$0 \xrightarrow{6} 5$	ignoré
9	$5 \xrightarrow{6} 4$	ignoré
10	$3 \xrightarrow{7} 4$	ignoré



Question 12 En supposant que l'algorithme de Prim fonctionne correctement sur un graphe dont tous les arcs ont un poids positif ou nul, montrer qu'il fonctionne également correctement sur un graphe dont certains arcs ont un poids négatif.

Correction : L'algorithme de Prim ne fait que *comparer entre eux* les poids des arcs (par l'intermédiaire de la file de priorité).
Dit autrement, l'algorithme de Prim se comporte exactement de la même façon sur un graphe où on aurait augmenté tous les poids de la même valeur, c'est-à-dire remplacé tout arc $x \xrightarrow{w} y$ par $x \xrightarrow{w+d} y$. On a donc le même résultat sur un graphe avec des poids négatifs que sur un graphe qu'on obtiendrait en les rendant tous positifs ou nuls en ajoutant $-m$ à tous les poids, avec $m < 0$ le plus petit poids négatifs, et dans les deux cas le résultat est minimal car la somme des poids d'un arbre couvrant ne diffère que de $(N-1)m$.

Question 13 Justifier que la méthode `prim` termine toujours, en exhibant une quantité qui décroît strictement à chaque itération de la boucle `while`.

Correction : La quantité

$$(\texttt{g.N} - \text{nombre de sommets marqués, taille de } \texttt{pq})$$

est un variant de boucle pour l'ordre lexicographique. En effet,

- soit le sommet `v` est déjà marqué, et dans ce cas la première composante est inchangée et la seconde décroît ;
- soit le nombre de sommets marqués augmente strictement et la première composante décroît donc.

Question 14 Montrer que la complexité en espace de la méthode `prim` peut être asymptotiquement proportionnelle à N^2, en exhibant pour toute valeur de N un graphe pour lequel une telle complexité est atteinte.

Correction : Considérons un graphe complet (tout sommet est relié à tout autre), où tous les arcs sont $i \xrightarrow{|j-i|} j$ pour $i \neq j$. Alors les sommets vont être marqués dans l'ordre 0,1,2,. . ..

Lorsque le sommet i est marqué, tous les arcs $i \xrightarrow{j-i} j$ pour $j > i$ sont ajoutés à la file de priorité. Après $N/2$ sommets marqués, on a donc

$$\begin{aligned} \text{taille de } \texttt{pq} &= -N/2 + \sum_{i<N/2} (N - i - 1) \\ &= N^2/2 + 2N + 1 \\ &\sim N^2/2 \end{aligned}$$

où le premier terme correspond aux $N/2$ arcs qui ont été retirés de la file de priorité.

Question 15 Donner la complexité en temps de la méthode `prim` dans le pire des cas, en fonction du nombre N de sommets et du nombre E d'arcs.

Correction : Chaque arc est ajouté à la file au plus une fois, lorsque son sommet d'origine est marqué. Il y a donc au plus $\mathcal{O}(E)$ tours de boucle et une file de priorité contenant au plus $\mathcal{O}(E)$ éléments. Chaque opération sur la file de priorité coûte donc au plus $\mathcal{O}(\log E)$. La complexité *totale* des lignes 12–14 est donc $\mathcal{O}(E \log E)$, de même que la complexité du reste des opérations de la boucle.
Soit un total $\mathcal{O}(E \log E)$, c'est-à-dire $\mathcal{O}(E \log N)$.

Question 16 Montrer que la propriété « la liste `result` est un arbre, couvrant exactement les sommets marqués dans `marked` » est un invariant de la boucle `while`. Il pourra être nécessaire de renforcer cet invariant.

Correction : Il faut renforcer la propriété en ajoutant que « pour tout arc dans `pq`, son origine (`src`) est marquée ».
La propriété est vraie initialement, car seul le sommet 0 est marqué, la liste `result` est vide et tous les arcs dans `pq` ont pour origine 0.
Supposons la propriété vraie et considérons une itération de la boucle.

- Si le sommet `v` est déjà marqué, alors on passe à l'itération suivante et la propriété reste vraie car `marked` et `result` restent inchangés, et la file a seulement perdu un élément.
- Sinon, le sommet `v` est marqué et l'arc `e` est ajouté à `result`.
  - Comme l'origine de `e` est marquée, `result` reste connexe.
  - Comme le sommet `v` n'était pas marqué, alors il n'était pas dans l'arbre `result` et on ne crée donc pas de cycle. L'ensemble `result` reste un arbre, et il couvre maintenant le sommet `v`.
  - Les arcs ajoutés à la file de priorité ont pour origine `v`, qui est marqué.

Question 17 Déduire de la question précédente que la liste renvoyée par la méthode `prim` est bien un arbre couvrant du graphe `g`.

Correction : Il suffit de montrer que tous les sommets de `g` sont marqués lorsque l'on sort de la boucle. Supposons qu'un sommet v ne soit pas marqué. Comme `g` est connexe, il existe un chemin du sommet 0 (marqué) au sommet v (non marqué). Ce chemin sort de

l'ensemble des sommets marqués par un arc $s \to t$ avec s marqué et t non marqué. Mais lorsque le sommet s a été marqué, cet arc a été ajouté à la file et a donc été examiné plus tard, ce qui contredit le fait que t ne soit pas marqué.

Question 18 Soit T un arbre couvrant minimal d'un graphe G et soit $x \xrightarrow{w} y$ un arc de T. Soit un arc $x' \xrightarrow{w'} y'$ de G avec un chemin de x à x' intégralement contenu dans T et de même un chemin de y' à y intégralement contenu dans T. Montrer qu'alors $w' \geq w$.

Correction : Si l'arc $x' \xrightarrow{w'} y'$ est l'arc $x \xrightarrow{w} y$, c'est évident (car $w' = w$).

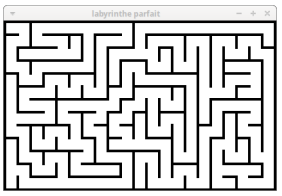
Sinon, l'arc $x' \xrightarrow{w'} y'$ ne peut pas être contenu dans T, sans quoi il y aurait un cycle dans T passant par x et y. Si on avait $w' < w$, alors $T' = T \backslash \{x \xrightarrow{w} y\} \cup \{x' \xrightarrow{w'} y'\}$ serait un arbre couvrant de G de poids total plus petit que T, contradiction.

Question 19 Déduire de la question précédente un algorithme pour vérifier qu'un arbre couvrant (une liste d'arcs) est minimal. On ne demande pas d'écrire le code Java. On ne demande pas non plus de vérifier qu'il s'agit d'un arbre et qu'il est couvrant, mais seulement de vérifier sa minimalité en supposant qu'on a déjà vérifié qu'il s'agit d'un arbre couvrant. Indication : On pourra supposer donnée une structure *union-find*.

Correction : On donne ici la réponse en Java (même si seul l'algorithme était demandé) :

```
static boolean isMinimal(Graph g, List<Edge> st) {
  for (Edge e: st) {
    // les deux classes obtenues avec tous les arcs de ‘st‘ sauf ‘e‘
    UnionFind uf = new UnionFind(g.V);
    for (Edge f: st)
      if (f != e)
        uf.union(f.src, f.dst);
    // tout arc ‘f‘ qui les relie ne peut être meilleur que ‘e‘
    for (Edge f: g.allEdges())
      if (!uf.sameClass(f.src, f.dst) && f.weight < e.weight)
        return false;
  }
  return true;
}
```

Question 20 Proposer un algorithme pour construire un labyrinthe parfait sur une grille rectangulaire $N \times M$ à partir de la méthode `prim`. Voici un exemple de taille 21×13 :



On rappelle qu'un labyrinthe parfait est un labyrinthe où toute paire de cases est reliée par un unique chemin.

Correction : On construit un graphe dont les sommets sont les paires (i, j), avec $0 \leq i < M$ et $0 \leq j < N$ et où chaque sommet est relié à ses quatre voisins (possiblement moins de quatre sur les bords) avec un arc *de poids aléatoire* (par exemple tiré entre 0 et 1).

On calcule alors un arbre couvrant de ce graphe. Par définition, il couvre tous les sommets et deux sommets sont reliés par un et un seul chemin. On a donc bien un labyrinthe parfait.

A Bibliothèque standard Java

`class LinkedList<E>`	liste chaînée dont les éléments sont de type `E`
`void add(E e)`	ajoute l'élément `e` à la fin de la liste
`E removeFirst()`	supprime et renvoie le premier élément de la liste

On peut parcourir tous les éléments d'une liste `l` avec la construction

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

`class PriorityQueue<E>`	une file de priorité dont les éléments sont de type `E` (`E` doit implémenter l'interface `Comparable<E>`)
`void add(E x)`	ajoute l'élément `x`
`void addAll(LinkedList<E> l)`	ajoute tous les éléments de `l`
`E remove()`	supprime et renvoie le plus petit élément de la file
`boolean isEmpty()`	renvoie `true` si et seulement si la file est vide

Les opérations `add` et `remove` ont une complexité en $\mathcal{O}(\log N)$ où N est la taille de la file de priorité. L'opération `addAll` a la même complexité qu'une itération de `add` sur tous les éléments de `l`.

`class Stack<E>`	une pile dont les éléments sont de type `E`
`Stack()`	renvoie une nouvelle pile, vide
`boolean isEmpty()`	renvoie `true` si et seulement si la pile est vide
`void push(E x)`	ajoute l'élément `x` au sommet de la pile
`E pop()`	supprime et renvoie l'élément au sommet de la pile lève `EmptyStackException` si la pile est vide

Les trois opérations `isEmpty`, `push` et `pop` s'exécutent en temps constant (amorti).

`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 HashMap<K, V>`	un dictionnaire dont les clés sont de type `K` et les valeurs sont 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)

Les trois opérations `containsKey`, `get` et `put` s'exécutent en temps constant (amorti).

* *
*