

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 Expressions régulières

Dans ce problème, on s'intéresse à la reconnaissance d'une chaîne de caractères par un motif décrit par une *expression régulière*. Ici, les expressions régulières se limitent<sup>1</sup> à des séquences de caractères où

- le caractère `.` est interprété comme « n'importe quel caractère » ;
- chaque caractère peut être suivi d'une étoile (`*`) pour indiquer sa répétition un nombre arbitraire de fois (y compris 0).

Voici quelques exemples d'expressions régulières et leur signification :

exemple	reconnaît les chaînes
<code>.a.</code>	de longueur 3 avec un <code>a</code> comme second caractère
<code>.*hello.*</code>	contenant le mot <code>hello</code>
<code>a*b*</code>	formées de <code>a</code> suivis de <code>b</code>

Formellement, une expression régulière  $r$  est donc une séquence de  $n$  motifs

$$r = m_0 m_1 \cdots m_{n-1}$$

où chaque motif  $m_i$  est de quatre types possibles :

- $c$ , avec  $c$  un caractère autre que `.` et `*`
- $c*$ , avec  $c$  un caractère autre que `.` et `*`
- `.`
- `.*`

Une chaîne  $s$  est *reconnue* par l'expression régulière  $r$  si elle peut être décomposée en  $n$  sous-chaînes

$$s = s_0 s_1 \cdots s_{n-1}$$

où chaque chaîne  $s_i$  est reconnue par le motif  $m_i$ , c'est-à-dire

- si  $m_i = c$ , alors  $s_i$  est la chaîne de longueur 1 réduite à  $c$  ;

---

1. En toute généralité, les expressions régulières incluent d'autres constructions, comme l'alternative ou le parenthésage, mais on ne les considère pas ici.

---

```

class SRE {
    final char c;
    final boolean star;
    final SRE next;

    SRE(char c, boolean star, SRE next) {
        this.c = c;
        this.star = star;
        this.next = next;
    }
}

```

---

FIGURE 1 – Classe Java pour représenter une expression régulière.

- si  $m_i = c^*$ , alors  $s_i$  est une concaténation de caractères  $c$  (éventuellement 0 fois) ;
- si  $m_i = .$ , alors  $s_i$  est une chaîne de longueur 1 ;
- si  $m_i = .*$ , alors  $s_i$  est une chaîne quelconque.

Ainsi, l'expression régulière  $a^*b^*$  reconnaît la chaîne  $aa$  car c'est la concaténation de la chaîne  $aa$  reconnue par le motif  $a^*$  et de la chaîne vide reconnue par le motif  $b^*$ . De même, l'expression régulière  $a^*b^*$  reconnaît la chaîne vide ou encore la chaîne  $abb$ , mais pas la chaîne  $ba$  ni la chaîne  $abc$ .

**Question 1** Donner une expression régulière qui reconnaît les chaînes de longueur au moins 4, commençant par le caractère  $a$  et se terminant par le caractère  $b$ .

**Représentation en Java.** Pour représenter une expression régulière en Java, on se donne la classe `SRE` de la figure 1. Il s'agit d'une liste simplement chaînée (champ `next`), contenant des caractères (champ `c`), chacun étant possiblement marqué d'une étoile (champ `star`). Ainsi, l'expression régulière  $a^*.b$  sera représentée par

```
new SRE('a', true, new SRE('.', false, new SRE('b', false, null)))
```

 (1)

On note que la liste vide, c'est-à-dire `null`, est une expression régulière valable, qui représente une séquence vide de motifs et qui reconnaît donc uniquement la chaîne vide.

**Question 2** Écrire une méthode `static String toString(SRE r)` qui écrit une expression régulière sous la forme d'une chaîne. Sur l'expression régulière (1) ci-dessus, cette méthode doit renvoyer la chaîne `"a*.b"`.

**Question 3** Inversement, écrire une méthode `static SRE ofString(String s)` qui construit une expression régulière à partir de sa description sous la forme d'une chaîne de caractères. Ainsi, `ofString("a*.b")` doit renvoyer l'expression régulière correspondant à (1). On rappelle que `.` et `*` ne sont pas utilisés comme des caractères normaux.

**Reconnaissance d'une chaîne par une expression régulière.** La figure 2 contient le code d'une méthode `matches` (ligne 18) qui détermine si une chaîne de caractères `s` est reconnue par une expression régulière `r`. Cette méthode est définie plus généralement pour le suffixe de `s` qui commence au caractère `i` (ligne 2), sous l'hypothèse  $0 \leq i \leq s.length()$ . On prendra le temps de bien lire et de bien comprendre ce code. Dans la suite, on pourra faire l'hypothèse que cette méthode est correcte.

**Question 4** Détailler les différents appels à la méthode `matches` (ligne 2) dans le calcul de `matches("aa", r)` où `r` est l'expression régulière correspondant à  $a^*a$ . Pour chaque appel, on indiquera la valeur de ses arguments et de son résultat.

```

1 // est-ce que s[i..] est reconnue par r ?
2 static boolean matches(String s, int i, SRE r) {
3     assert s != null && 0 <= i && i <= s.length();
4     if (r == null)
5         return i == s.length();
6     if (r.star && matches(s, i, r.next))
7         return true;
8     if (i == s.length())
9         return false;
10    if (r.c == '.' || r.c == s.charAt(i))
11        if (r.star)
12            return matches(s, i+1, r);
13        else
14            return matches(s, i+1, r.next);
15    return false;
16 }
17
18 static boolean matches(String s, SRE r) {
19     return matches(s, 0, r);
20 }

```

---

FIGURE 2 – Reconnaissance par une expression régulière.

**Question 5** Justifier que la méthode `matches` termine toujours.

**Question 6** Justifier que la méthode `matches` ne provoque jamais d'accès à `s` en dehors des bornes (`charAt` ligne 10), ni d'exception `NullPointerException` (lignes 6, 10, 11 et 14).

**Question 7** Donner un exemple de chaîne `s` de longueur  $N$  et d'expression régulière `r` de longueur  $N$  telles que le calcul de `matches(s, r)` prend un temps proportionnel à  $2^N$ . On demande de justifier le temps de calcul.

```

1 static boolean fastMatches(HashMap<MemoKey, Boolean> memo, String s, int i, SRE r) {
2     MemoKey k = new MemoKey(i, r);
3     if (memo.containsKey(k)) return memo.get(k);
4     boolean b;
5     ... // on calcule le résultat dans b
6     ... // comme dans la méthode matches mais en remplaçant
7     ... // d'une part return ... par b = ...
8     ... // et d'autre part matches(...) par fastMatches(memo, ...)
9     memo.put(k, b);
10    return b;
11 }
12 static boolean fastMatches(String s, SRE r) {
13     return fastMatches(new HashMap<>(), s, 0, r);
14 }
15
16 class MemoKey {
17     final int i;
18     final SRE r;
19
20     MemoKey(int i, SRE r) { this.i = i; this.r = r; }
21
22     @Override
23     public int hashCode() { ... }
24
25     @Override
26     public boolean equals(Object o) { ... }
27 }

```

---

FIGURE 3 – Application du principe de mémorisation.

**Mémorisation.** Pour améliorer la complexité de la méthode `matches`, on peut utiliser le principe de mémorisation. La figure 3 contient le code Java d'une méthode `fastMatches` qui réalise cette idée. La mémorisation utilise une table de hachage `memo` qui est créée ligne 13 puis passée en argument à la méthode récursive `fastMatches` de la ligne 1. Une clé dans cette table est une paire  $(i, r)$ , représentant des arguments de la méthode `fastMatches`. Une telle paire est réalisée par la classe `MemoKey` (lignes 16–27).

**Question 8** Justifier l'intérêt de la mémorisation, en exhibant une chaîne `s` et une expression régulière `r` pour lesquelles l'exécution de `matches(s, r)` amène à refaire au moins deux fois le même calcul, c'est-à-dire au moins deux appels à `matches` avec exactement les mêmes arguments.

**Question 9** Donner un code Java pour les méthodes `hashCode` et `equals` de la classe `MemoKey` (respectivement lignes 23 et 26).

**Question 10** Donner la complexité du calcul de `fastMatches(s, r)` en fonction de la longueur  $N$  de la chaîne `s` et de la longueur  $M$  de l'expression régulière `r`.

---

```

class Graph {
    final int V; // les sommets sont 0,1,...,V-1
    Graph(int V) // construit un graphe vide
    void addEdge(int u, int v) // ajoute un arc u->v
    LinkedList<Integer> adj(int v) // renvoie les voisins de v
}

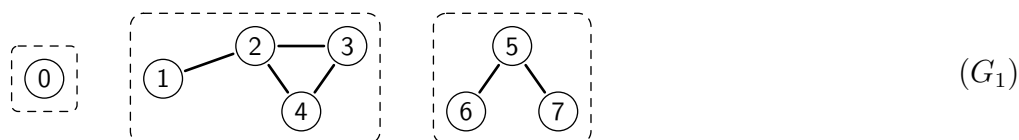
```

FIGURE 4 – Structure de graphe.

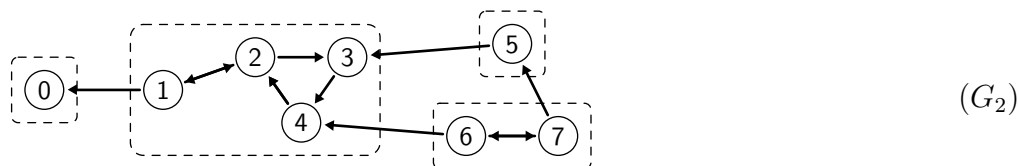
## 2 Composantes fortement connexes

Dans ce problème, on s'intéresse à la connectivité dans un graphe. Si  $u$  et  $v$  sont deux sommets d'un graphe, on dit qu'ils sont *connectés* s'il existe un chemin de  $u$  à  $v$  ou un chemin de  $v$  à  $u$ , et qu'ils sont *fortement connectés* s'il existe un chemin de  $u$  à  $v$  et un chemin de  $v$  à  $u$ . Pour un graphe non orienté, les deux notions coïncident.

Une *composante fortement connexe* est un sous-ensemble de sommets qui sont tous fortement connectés deux à deux, et qui est maximal pour l'inclusion. Voici un exemple de graphe non orienté, avec trois composantes fortement connexes (identifiées par des pointillés) :



Voici un exemple de graphe orienté, avec quatre composantes fortement connexes (identifiées par des pointillés) :



Notre objectif est de calculer les composantes fortement connexes d'un graphe. La figure 4 décrit (partiellement) une classe `Graph` pour des graphes dont les sommets sont des entiers. Si  $g$  est un objet de cette classe, alors les sommets sont les entiers  $0, 1, \dots, g.V - 1$ . Si  $0 \leq v < g.V$ , les voisins de  $v$  sont obtenus avec `g.adj(v)`, par ordre croissant, et on peut en particulier les parcourir avec une boucle `for (int w: g.adj(v))`.

On se propose de calculer les composantes fortement connexes sous la forme d'une méthode

```
int[] components(Graph g)
```

qui renvoie un tableau de taille `g.V` donnant, pour chaque sommet, le numéro de sa composante fortement connexe. S'il y a  $N$  composantes, elles sont numérotées  $0, 1, \dots, N - 1$ , dans un ordre arbitraire. Dans le cas du graphe  $G_2$  ci-dessus, une réponse possible est le tableau

0	1	1	1	1	2	3	3
---	---	---	---	---	---	---	---

mais toute autre permutation de  $0, 1, 2, 3$  dans ce tableau serait également valable.

**Question 11** Donner les composantes fortement connexes du graphe suivant :



```

1  static int nc;
2  static int[] num;
3
4  static int[] components(Graph g) {
5      num = new int[g.V];
6      nc = 0;
7      boolean[] visited = new boolean[g.V];
8      for (int v = 0; v < g.V; v++) {
9          if (!visited[v]) {
10             dfs(visited, g, v);
11             nc++;
12         }
13     }
14     return num;
15 }
16
17 static void dfs(boolean[] visited, Graph g, int v) {
18     if (visited[v]) return;
19     visited[v] = true;
20     num[v] = nc;
21     for (int w: g.adj(v))
22         dfs(visited, g, w);
23 }

```

---

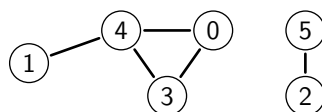
FIGURE 5 – Calcul des composantes d’un graphe non orienté avec un parcours en profondeur.

**Graphes non orientés.** On commence par le cas d’un graphe non orienté, pour lequel le problème est plus simple.

**Question 12** Proposer un algorithme de calcul des composantes fortement connexes utilisant la structure *union-find* vue en cours, et donner la complexité de cet algorithme. On ne demande pas d’écrire le code Java.

Une autre solution consiste à utiliser un parcours en profondeur. La figure 5 contient un code qui réalise cette idée. L’entier `nc` (ligne 1) compte les composantes et le tableau `num` (ligne 2) est la numérotation qui sera renvoyée au final (ligne 14). Le tableau `visited` (ligne 7) marque les sommets déjà visités par le parcours en profondeur. La boucle ligne 8 lance le parcours en profondeur sur tous les sommets qui ne sont pas déjà marqués. Le parcours en profondeur est réalisé par la méthode récursive `dfs` ligne 17. Il est tout à fait analogue à celui vu en cours. Le seul ajout est la ligne 20, qui affecte au sommet `v` le numéro de sa composante.

**Question 13** Donner la séquence des appels à la méthode `dfs` lorsque la méthode `components` (ligne 4) est appelée sur le graphe suivant :



Pour chaque appel à `dfs`, on indiquera uniquement la valeur de `v`. Donner également le tableau renvoyé par la méthode `components` au final.

```

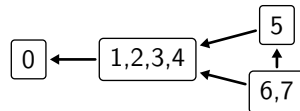
1  static LinkedList<Integer> order;
2
3  static LinkedList<Integer> postOrder(Graph g) {
4      order = new LinkedList<>();
5      boolean[] visited = new boolean[g.V];
6      for (int v = 0; v < g.V; v++)
7          dfsPost(visited, g, v);
8      return order;
9  }
10
11 static void dfsPost(boolean[] visited, Graph g, int v) {
12     if (visited[v]) return;
13     visited[v] = true;
14     for (int w: g.adj(v))
15         dfsPost(visited, g, w);
16     order.addFirst(v);
17 }

```

FIGURE 6 – Calcul de l'ordre postfixe.

**Graphes orientés.** Calculer les composantes fortement connexes d'un graphe orienté est plus difficile. L'idée reste d'utiliser le code de la figure 5, mais en parcourant les sommets à la ligne 8 dans un ordre différent. Déterminer cet ordre est l'objet des questions qui viennent.

**Graphe des composantes.** Pour un graphe  $G$ , on peut construire un graphe orienté  $C(G)$ , appelé graphe des composantes de  $G$ , de la manière suivante : les sommets de  $C(G)$  sont les composantes fortement connexes de  $G$  (donc des ensembles de sommets) et il y a un arc entre une composante  $C_1$  et une composante  $C_2$  si et seulement s'il existe un arc entre un sommet de  $C_1$  et un sommet de  $C_2$ . Ainsi, le graphe des composantes du graphe  $G_2$  donné plus haut est le suivant :



**Question 14** Montrer que  $C(G)$  ne contient pas de cycle.

**Graphe miroir.** Pour un graphe  $G$ , son graphe miroir  $G^R$  est un graphe qui a les mêmes sommets que  $G$  et des arcs inversés, c'est-à-dire que  $u \rightarrow v$  est un arc de  $G^R$  si et seulement si  $v \rightarrow u$  est un arc de  $G$ .

**Question 15** Donner le code Java d'une méthode `static Graph reverse(Graph g)` qui renvoie le graphe miroir du graphe  $g$ .

**Ordre postfixe.** Lorsque l'on réalise le parcours en profondeur d'un graphe, on peut noter dans quel ordre se terminent les visites des sommets. La figure 6 contient le code Java d'une méthode `postOrder` qui réalise cette idée. C'est exactement un parcours en profondeur, où le seul ajout est la ligne 16 qui ajoute le sommet  $v$  au *début* de la liste `order` une fois que sa visite est terminée. Cette liste est renvoyée une fois que le parcours en profondeur a visité tous les sommets (ligne 8). Ainsi, pour le graphe  $G_2$ , la liste renvoyée par `postOrder` est 6, 7, 5, 1, 2, 3, 4, 0.

**Question 16** Quelle est la liste renvoyée par `postOrder` sur le graphe  $G_2^R$  ?

**Question 17** Montrer que, sur un graphe *acyclique*, la liste renvoyée par `postOrder` définit un *tri topologique* du graphe, c'est-à-dire que pour tout arc  $u \rightarrow v$ , le sommet  $u$  apparaît *avant* le sommet  $v$  dans la liste.

**Algorithme de Kosaraju–Sharir.** Pour calculer les composantes fortement connexes d'un graphe orienté, on peut utiliser l'algorithme de Kosaraju–Sharir. Son code est celui de la figure 5, où le seul changement consiste à remplacer la ligne 8 par la ligne suivante :

```
for (int v: postOrder(reverse(g))) {
```

Dit autrement, on ne parcourt pas les sommets de  $g$  dans l'ordre croissant, mais dans l'ordre postfixe du graphe miroir de  $g$ . Tout le reste est inchangé.

**Question 18** Dérouler l'algorithme de Kosaraju–Sharir sur le graphe  $G_3$  de la question 11.

**Question 19** Donner la complexité en temps de l'algorithme de Kosaraju–Sharir, en fonction du nombre  $V$  de sommets et du nombre  $E$  d'arcs. Est-elle optimale ?

**Question 20** Montrer la correction de l'algorithme de Kosaraju–Sharir.

---



## A Bibliothèque standard Java

```
class LinkedList<E>
    void add(E e)           ajoute l'élément e à la fin de la liste
    void addFirst(E e)     ajoute l'élément e au début de la liste
    On peut parcourir tous les éléments d'une liste l avec la construction
        for (E x: l) ...

class String
    int length()          la longueur de la chaîne
    char charAt(int i)    le caractère d'indice i, pour  $0 \leq i < \text{length}()$ 

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

        *   *
        *

```