X2020 — INF411
Contrôle classant / 16 novembre 2021 / 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 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 [1] 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
`.a.`	de longueur 3 avec un `a` comme second caractère
`.*hello.*`	contenant le mot `hello`
`a*b*`	formées de `a` suivis de `b`

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

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

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`.

Correction : `a...*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"`.

Correction : Pas de difficulté ici : on parcourt la liste et on accumule les caractères et les étoiles. On le fait efficacement avec `StringBuffer` :

```
static String toString(SRE r) {
  StringBuffer sb = new StringBuffer();
  for (; r != null; r = r.next) {
    sb.append(r.c);
    if (r.star) sb.append('*');
  }
  return sb.toString();
}
```

On pourrait le faire avec des concaténations de chaînes, comme cela

```
static String toString(SRE r) {
  String s = "";
  for (; r != null; r = r.next) {
    s += r.c;
    if (r.star) s += '*';
  }
  return s;
}
```

mais la complexité serait alors moins bonne (quadratique).

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.

Correction :

```
static SRE ofString(String s) {
  SRE r = null;
  for (int i = s.length() - 1; i >= 0; i--) {
    boolean star = s.charAt(i) == '*';
    if (star) i--;
    if (i < 0 || s.charAt(i) == '*') throw new Error("bad RE");
    r = new SRE(s.charAt(i), star, r);
  }
  return r;
}
```

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.

Correction :

```
matches s="aa" i=0 r=a*a     => true
  matches s="aa" i=0 r=a     => false
    matches s="aa" i=1 r=    => false
  matches s="aa" i=1 r=a*a   => true
    matches s="aa" i=1 r=a   => true
      matches s="aa" i=2 r=  => true
```

```
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.

Correction : On constate que soit l'indice `i` augmente strictement (lignes 12 et 14), soit la longueur de `r` diminue strictement (lignes 6 et 14).
Plus précisément, la grandeur (`s.length()` $-$ `i`, $|$`r`$|$) diminue strictement pour l'ordre lexicographique à chaque appel récursif, ce qui assure la terminaison.

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).

Correction : Pour ce qui est de `charAt`, on a fait l'hypothèse que $0 \leq$ `i` $\leq$ `s.length()`, qui plus est vérifiée par la ligne 3. Par ailleurs, le test ligne 8 assure que `i` $<$ `s.length()` au-delà de la ligne 9. L'accès est donc bien valide.
Pour ce qui est de `NullPointerException`, c'est facile : les lignes 4–5 assurent que `r` n'est pas `null` au-delà de la ligne 5.

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.

Correction : On peut prendre par exemple

$$\texttt{s} \;=\; \texttt{aa}\cdots\texttt{aa}$$
$$\texttt{r} \;=\; \texttt{a*a*}\cdots\texttt{a*b}$$

On note que tout appel à `matches` sur un suffixe de `s` et de `r` va renvoyer `false`.

Soit C_n le temps minimal de calcul de `matches` sur un suffixe de `s` de longueur au moins n et un suffixe de `r` de longueur au moins n. On a $C_0, C_1 \geq 1$. Pour $n \geq 2$, l'appel ligne 6 est effectué car `r.star` et vrai. Il va coûter au moins C_{n-1} et renvoyer `false`. Du coup, on va également réaliser l'appel ligne 12, qui va également coûter au moins C_{n-1}. Du coup,

$$C_n \geq 1 + 2C_{n-1}$$

ce que l'on peut réécrie

$$C_n + 1 \geq 2(C_{n-1} + 1)$$

pour en déduire $C_n + 1 \geq 2^{n+1}$. Le temps de calcul total est au moins C_N, et donc bien exponentiel en N.

Mémoïsation. Pour améliorer la complexité de la méthode `matches`, on peut utiliser le principe de mémoïsation. La figure 3 contient le code Java d'une méthode `fastMatches` qui réalise cette idée. La mémoïsation 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 $(\texttt{i}, \texttt{r})$, 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 mémoïsation, 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.

Correction : Si on reprend l'exemple de la question 7, alors on va avoir une exécution de la forme

```
matches(s, 0, r)
  matches(s, 0, r.next) ligne 6
    matches(s, 0, r.next.next) ligne 6
      ...
    matches(s, 1, r.next) ligne 12  <----------+
      ...                                      |
  matches(s, 1, r) ligne 12                    |
    matches(s, 1, r.next) ligne 6   <----------+
      ...
```

car tous les appels à `matches` renvoient `false`. On a en particulier deux appels identiques `matches(s, 1, r.next)`.

Le phénomène va d'ailleurs se répéter à chaque niveau, ce qui est une autre façon de justifier la complexité exponentielle exhibée à la question 7.

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

Correction : On peut par exemple proposer ceci :

```
public int hashCode() {
  return 31 * this.c + this.r.hashCode();
}
public boolean equals(Object o) {
  MemoKey that = (MemoKey)o;
  return this.c == that.c && this.r == that.r;
}
```

Pour le champ `r`, on se contente d'une égalité physique dans `equals`, car un suffixe de `r` ne peut être structurellement égal à un autre suffixe de `r` que s'il s'agit exactement du même. Dès lors, on se sert de la méthode `hashCode` de `Object` dans `hashCode`.

On note qu'avec ces définitions, ces deux méthodes sont en $O(1)$.

```
static boolean fastMatches(HashMap<MemoKey, Boolean> memo, String s, int i, SRE r) {
  MemoKey k = new MemoKey(i, r);
  if (memo.containsKey(k)) return memo.get(k);
  boolean b;
  ... // on calcule le résultat dans b
  ... // comme dans la méthode matches mais en remplaçant
  ... // d'une part return ... par b = ...
  ... // et d'autre part matches(...) par fastMatches(memo, ...)
  memo.put(k, b);
  return b;
}
static boolean fastMatches(String s, SRE r) {
  return fastMatches(new HashMap<>(), s, 0, r);
}

class MemoKey {
  final int i;
  final SRE r;

  MemoKey(int i, SRE r) { this.i = i; this.r = r; }

  @Override
  public int hashCode() { ... }

  @Override
  public boolean equals(Object o) { ... }
}
```

FIGURE 3 – Application du principe de mémoïsation.

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`.

Correction : Les paires (`i`, `r`) sont au nombre de $(N+1) \times M$. Pour chacune, le calcul est fait au plus une seule fois. Les opérations en dehors des appels récursifs étant en nombre borné (pas de boucle et des accès à la table de hachage en temps constant), la complexité est donc en $O(NM)$.

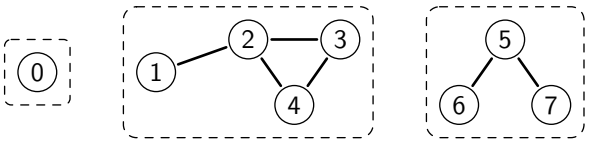
```
class Graph {
  final int V;                    // les sommets sont 0,1,...,V-1
  Graph(int V)                    // construit un graphe vide
  void addEdge(int u, int v)      // ajoute un arc u->v
  LinkedList<Integer> adj(int v) // renvoie les voisins de v
}
```

FIGURE 4 – Structure de graphe.

2 Composantes fortement connexes

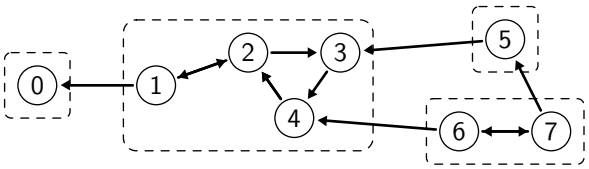
Dans ce problème, on s'intéresse à la connectivité dans un graphe. Si u et v sont deux sommets d'un graphe, on dit qu'ils sont *connectés* s'il existe un chemin de u à v ou un chemin de v à u, et qu'ils sont *fortement connectés* s'il existe un chemin de u à v *et* un chemin de v à u. Pour un graphe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



(G_1)

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



(G_2)

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 $0, 1, \ldots,$ `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

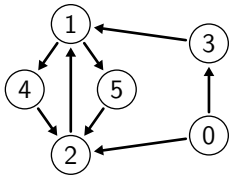
```
int[] components(Graph g)
```

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 $0, 1, \ldots, N-1$, 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 :



(G_3)

Correction : $\{1, 2, 4, 5\},\ \{3\},\ \{0\}$

```
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.

Correction : On se donne une structure union-find dont les élément sont les V sommets. On parcourt chaque arc $u \to v$ du graphe et on fait l'union des classes de u et v, avec l'invariant suivant :

> deux sommets dans la même classe sont connectés

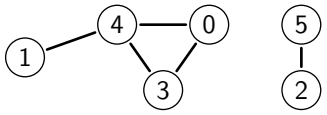
Ainsi, au final, chaque classe correspond exactement aux sommets qui sont connectés, c'est-à-dire à une composante, car un arc qui relierait deux sommets de deux classes disjointes aurait conduit à la réunion de ces deux classes.

La complexité est $O(V + E\alpha(V))$, que l'on peut considérer comme $O(V + E)$ en pratique.

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.

Correction : On a la séquence suivante :

```
dfs(0)
  dfs(3)
    dfs(0) // déjà vu
    dfs(4)
      dfs(0) // déjà vu
      dfs(1)
        dfs(4) // déjà vu
      dfs(3) // déjà vu
  dfs(4) // déjà vu
dfs(2)
  dfs(5)
    dfs(2) // déjà vu
```

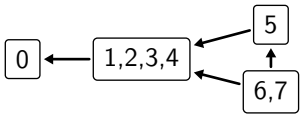
Attention à ne pas oublier les appels effectués sur des sommets déjà visités, même s'ils terminent immédiatement (`return` ligne 18).

Le tableau renvoyé est

```
  0   1   2   3   4   5
+---+---+---+---+---+---+
| 0 | 0 | 1 | 0 | 0 | 1 |
+---+---+---+---+---+---+
```

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.

Correction : Supposons l'existence d'un cycle dans $C(G)$ et soit deux sommets u et v dans deux composantes distinctes de ce cycle, respectivement appelées U et V. Le cycle sort de U par un certain sommet u^o et y entre par un sommet u^i. De même, le cycle sort de V par un sommet v^o et y entre par un sommet v^i. Or, u, u^o, u^i sont fortement connectés et v, v^o, v^i sont fortement connectés. On en déduit que u et v sont fortement connectés, ce qui contredit le fait qu'il ne sont pas dans la même composante.

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 $u \to v$ est un arc de G^R si et seulement si $v \to u$ 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`.

Correction : Pas de difficulté : pour chaque arc `v->w`, on crée un arc `w->v`.

```
static Graph reverse(Graph g) {
  Graph gr = new Graph(g.V);
  for (int v = 0; v < g.V; v++)
    for (int w: g.adj(v))
      gr.addEdge(w, v);
  return gr;
}
```

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 ?

Correction : $0, 1, 2, 4, 3, 5, 7, 6$

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 $u \to v$, le sommet u apparaît *avant* le sommet v dans la liste.

Correction : Cela revient à montrer que `dfs(u)` termine *après* `dfs(v)`. Considérons le moment où `dfs(u)` est appelé pour la première fois.

— Si l'appel à `dfs(v)` a déjà été fait et est déjà terminé, alors `v` est déjà dans la liste et donc `u` se retrouvera bien avant.

```
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.

— Si l'appel à `dfs(v)` est déclenché par cet appel à `dfs(u)`, directement ou indirectement, alors l'appel à `dfs(v)` terminera avant l'appel à `dfs(u)`, et là encore la propriété voulue sera établie.
— Enfin, le dernier cas de figure est un appel `dfs(u)` déclenché, directement ou indirectement, par un appel à `dfs(v)`. Mais dans ce cas, il existe un chemin de `v` à `u`, et donc un cycle, ce qui est impossible.

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.

Correction : La liste renvoyée par `postorder(reverse(G3))` est $1, 3, 2, 5, 4, 0$.

On commence donc par appeler `dfs` sur le sommet 1, qui visite les sommets $1, 4, 2, 5$. C'est la première composante $\{1, 4, 2, 5\}$.

Puis, on appelle `dfs` sur 3, qui visite uniquement 3. C'est la deuxième composante $\{3\}$.

Puis, on appelle `dfs` sur les sommets 2, 5 et 4, ce qui n'a aucun effet car ils ont déjà été visités.

Enfin, on appelle `dfs` sur le sommet 0, qui visite uniquement le sommet 0. C'est la troisième composante $\{0\}$.

On retrouve bien le résultat 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 ?

Correction : Le calcul de G^R est en $O(V + E)$.

Le calcul de `postOrder` est également en $O(V + E)$, car il s'agit d'un parcours en profondeur et `addFirst` s'exécute en temps constant.

Enfin, le calcul de `components` est toujours en $O(V+E)$, car là aussi il s'agit d'un parcours en profondeur.

On a donc une complexité totale en $O(V + E)$, ce qui est optimal car on est obligé de considérer tout le graphe.

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

Correction : Montrons la correction par récurrence sur le nombre de composantes fortement connexes identifiées par l'algorithme. Supposons que l'algorithme a correctement identifié les n premières composantes fortement connexes. En particulier, tous les sommets et seuls les sommets de ces composantes sont marqués dans `visited`.

Soit alors `v` le prochain sommet non visité sur lequel `dfs` est appelée et C la composante de `v`.

— Les sommets de C ne sont pas marqués (par HR) et sont donc visités car accessibles à partir de `v` (par définition de C).

— Les sommets des autres composantes se sont pas visités, car

  — Les sommets des n premières composantes sont déjà marqués et ne sont donc pas visités.

  — Si C' est une autre composante (que les $n + 1$ premières), alors il ne peut exister d'arc $x \to y$ allant de C à C'. En effet, dans le cas contraire, l'arc $y \to x$ existe dans le graphe miroir et y apparaît alors *avant* x dans la liste `postOrder(reverse(g))`, ce qui contredit le choix de `v`.

  Justifions cette dernière affirmation : si `dfsPost` est appelée sur un sommet de C avant d'être appelée sur y, alors tous les sommets de C sont visités et ajoutés à la liste avant y, car il n'y a pas de chemin depuis l'un d'eux vers y (sans quoi y serait dans C). Sinon, c'est que `dfsPost` est appelée sur y avant tout sommet de C mais alors tous les sommets de C seront visités avant que la visite de y ne termine, car il sont tous accessibles par l'arc de $y \to x$.

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`	le type des chaînes de caractères
`int length()`	la longueur de la chaîne
`char charAt(int i)`	le caractère d'indice `i`, pour $0 \leq$ `i` $<$ `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

* *
*