



2502

INF411

COMPOSITION d INF 411  
en date du 16/11/2021

## I. Expressions régulières

1)  $a \dots * b$  est une chaîne de longueur au moins 4 commençant par  $a$  et finissant par  $b$ .

La chaîne contient en effet au moins  $a$  puis deux ~~des~~ caractères quelconques puis  $b$  si  $\dots *$  est contenu 0 fois. Sinon, elle contient  $a$  suivi de deux caractères quelconques suivi d'un caractère quelconques répété  $n \geq 1$  fois, puis  $b$ .

2) `String toString(SRE r){`

`String res = "";`

`while (r != null) {`

`if (r.star) {`

`res += r.c.toString();`

`res += "*";`

`} else {`

`res += r.c.toString();`

`}`

`}`

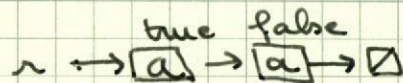
`return res;`

NOTE



## NE PAS ÉCRIRE DANS CE CADRE

```
3) static SRE ofString (String s) {  
    SRE r = null;  
    int i = s.length() - 1;  
    while (i > 0) {  
        if (s.charAt(i) == '*') {  
            r.next = r;  
            r.c = s.charAt(i - 1);  
            r.star = true;  
            i = i - 2;  
        } else {  
            r.next = r;  
            r.c = s.charAt(i);  
            r.star = false;  
            i = i - 1;  
        }  
    }  
    return r;  
}
```



- 4) Si on fait l'appel `matches("aa", r)` on fait :
- (1) un premier appel à `matches("aa", 0, r)`  
on passe le assert et  $r$  n'est pas nul donc on rentre dans le second if.  $r.star == true$  donc on fait
  - (2) un appel à `matches(s, 0, r.next)`  
avec  $s = "aa"$ ,  $i = 0$ ,  $r.next$  est  $\overset{\text{false}}{\boxed{a}} \rightarrow \boxed{\phantom{a}} \rightarrow \boxed{\phantom{a}}$



## NE PAS ÉCRIRE DANS CE CADRE

on repasse l'assert et la première boucle et comme cette fois `r.star` est faux on continue. `i ≠ 2` donc on atteint le `if` suivant. On a bien `r.c = a = s.charAt(i)` donc on entre dans le `if`. Comme `r.star = false`, on fait l'appel :

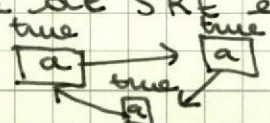
(3) `matches("aa", 1, r.next)` où `r.next` est null on passe le assert et comme `r` est null, on rentre dans le premier `if`. Comme `i = 1 = s.length()` (?) on return true.

Cela entraîne un return true pour le second appel à `matches` (2), on entre donc dans le `if` qui conditionnait le (2) et on return true pour le premier (1).

5) La méthode `matches` termine toujours. Il y a deux possibilités de terminer : soit on arrive au bout de la structure régulière (`r == null`) et alors on return soit true, soit false,

- soit on incrémente `i` un trop grand nombre de fois et on return false via le assert ou via le troisième `if`

La seule chose qui peut poser problème serait d'avoir créé une boucle de SRE étant tous true.

Par exemple :  car on resterait coincés au 3<sup>e</sup> `if`



## NE PAS ÉCRIRE DANS CE CADRE

mais on ne construirait pas un SRF comme ça.

6) `matches` ne provoque jamais d'accès à `s` en dehors des bornes car au début de chaque appel à `matches`, un assert vérifie : que `s` n'est pas nulle et que `i` est bien dans les bornes soit  $0 \leq i \leq s.length$ .

La possibilité d'avoir  $i = s.length$  n'est pas un problème car dans ce cas, car avant de faire appel à `charAt(i)`, `matches` a une condition `if (i == s.length) return false` qui empêche d'atteindre le `charAt(i)`.

Il n'y a pas de `NullPointerException` car avant tous les appels au champ de `r`, la condition `if (r == null)` `return` permet de sortir de la méthode.

7)



COMPOSITION d INF 411  
en date du 16/11/2021

8) La mémorisation a un intérêt car la chaîne :

9) @Override

```
public int hashCode() {  
    int res = 0;  
    res += this.i * 31;  
  
    if (this.r == null) return res;  
    res += this.r.c * 13;  
    res += star ? 1 : 0;  
    SRE current = r;  
    while (current != null) {  
        res += 3;  
        current = current.next;  
    }  
    return res;  
}
```

NOTE



# NE PAS ÉCRIRE DANS CE CADRE

@Override

```
public boolean equals (Object o) {
```

```
    Memokey m = (Memokey o);
```

```
    // ...
```

```
    if (this.i != m.i) return false;
```

```
    if (this.r == null && m.r == null) return true;
```

```
    // ...
```

```
    while (this.r != null || m.r != null) {
```

```
        if (this.r == null || m.r == null) return false;
```

```
        if (this.r.c != m.r.c || this.r.star != m.r.star) return false;
```

```
        this.r = this.r.next;
```

```
        m.r = m.r.next;
```

```
    }
```

```
    return true;
```

```
}
```

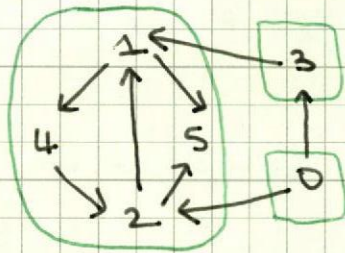
10)



# NE PAS ÉCRIRE DANS CE CADRE

## 2. Composantes fortement connexes

11) Les composantes fortement connexes de ce graphe sont :



12) Pour un tel algorithme, on pourrait :

- créer un tableau union-find de longueur  $q \cdot V$
- parcourir tous les sommets du graphe
- pour chaque sommet parcourir la liste de ses voisins
- pour chaque voisin faire l'union des classes du sommet et du voisin
- une fois que le tableau union est complet, remplacer la valeur dans chaque case  $i$  par  $\text{find}(i)$
- parcourir les cases du tableau pour identifier le nombre de classes différentes et leur représentant
- créer un second tableau  $\text{res}$
- remplacer les cases du tableau  $\text{res}$  tel que : les cases de la classe dont le représentant est le plus petit ont la valeur 1, les cases de la classe dont le représentant est le deuxième plus petit ont la valeur 2, ainsi de suite.
- renvoyer  $\text{res}$

Cet algorithme a une complexité  $O(V^2)$  dans le pire des cas puisqu'on parcourt tous les sommets de  $V$  et pour chaque sommet, tous ses voisins.



# NE PAS ÉCRIRE DANS CE CADRE

13) On commence par :

components(g) : num = new int [6]

mc = 0

visited = new boolean [6]

boucle for de components

- v = 0 . v n'est pas visité

donc on appelle dfs(visited, g, 0)

qui marque visited[0] = true ;

num[0] = 0 ;

boucle for de dfs

[ - w = 3 (voisin de 0)

dfs(visited, g, 3)

visited[3] = true

num[3] = 0

[ boucle for

- dfs(visited, g, 0) on sort

- dfs(visited, g, 4) visited[4] = true  
num[4] = 0

[ boucle for

- dfs(visited, g, 0) on sort

- dfs(visited, g, 1) visited[1] = true  
num[1] = 0

- dfs(visited, g, 3) on sort

on sort

on sort

- u = 4  
dfs(visited, g, 4) on sort  
on sort



COMPOSITION d INF 411

en date du 16/11/2021

$nc = 1$

-  $v = 1$  déjà visité donc on sort

-  $v = 2$  pas visité

appel à  $dfs(visited, g, 2)$  marque  $visited[2] = true$

$num[2] = 1$

boucle for de dfs

-  $w = 5$

$dfs(visited, g, 5)$ ,  $visited[5] = true$ ,  $num[5] = 1$

boucle for

-  $dfs(visited, g, 2)$  on sort

on sort

on sort

-  $v = 3$  à  $5$  déjà faits donc pas de dfs

On retourne :

0	0	1	0	0	1
---	---	---	---	---	---

14) Montrons que  $C(G)$  ne contient pas de cycle.

Supposons que  $C(G)$  est un cycle soit il existe un chemin de longueur  $\geq 2$  allant de  $C_i$  à

$C_i$  :  $C_i \rightarrow C_j \rightarrow C_k \rightarrow \dots \rightarrow C_i$ . Dans ce cas, il

existe un chemin  $C_i \rightarrow C_e$  pour tout  $e$  tel que

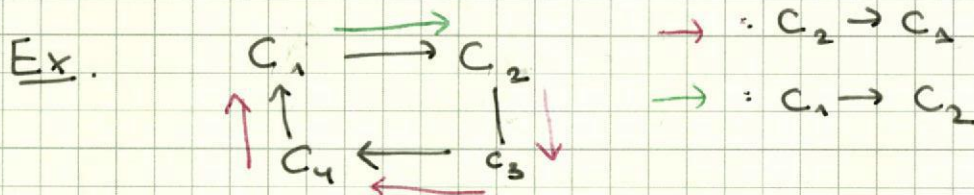
$C_e$  appartient au cycle :  $C_i \rightarrow C_j \rightarrow \dots \rightarrow C_e$  et un

chemin  $C_e \rightarrow C_i$  :  $C_e \rightarrow C_a \rightarrow \dots \rightarrow C_i$ .

NOTE



# NE PAS ÉCRIRE DANS CE CADRE



Or, s'il existe deux chemins  $C_i \rightarrow C_e$  et  $C_e \rightarrow C_i$  alors les sommets des composantes  $C_i$  et  $C_e$  sont fortement connectés. En effet, il existe alors un sommet de  $C_i$  qui a un chemin vers un sommet de  $C_e$  et un sommet de  $C_e$  qui va vers  $C_i$ . Comme au sein des composantes, tous les sommets sont fortement connectés, les sommets de  $C_i$  et  $C_e$  sont ensemble fortement connectés.

Dans ce cas, par définition de  $C(G)$ ,  $C_i$  et  $C_e$  ne devraient pas être deux composantes différentes et il faudrait créer une composante  $C_{i+e}$  contenant tous les sommets de  $C_i$  et  $C_e$ .  $C(G)$  serait faux.

Donc par l'absurde,  $C(G)$  ne contient pas de cycle.

15) 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)) {

addEdge(w,v);

}

}

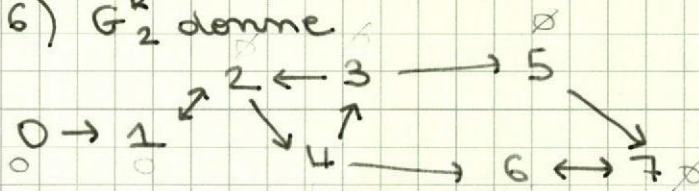
return gR

}



# NE PAS ÉCRIRE DANS CE CADRE

16)  $G_2^R$  donne

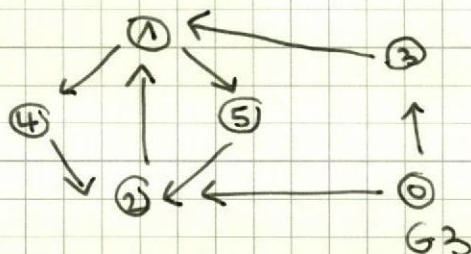


Ainsi, postOrder renvoie : 0 1 2 4 3 5 7 6

17) Pour un graphe acyclique, dfs finit l'exploration de  $v$  avant celle de  $u$  (pour  $u \rightarrow v$ ). En effet, il commence à examiner  $u$ , puis il examine  $v$  (car c'est un voisin de  $u$ ) puis il examine tous les voisins de  $v$  et termine leur <sup>examen</sup> ~~examination~~ \*. Ensuite seulement, il termine l'<sup>examen</sup> ~~examination~~ de  $v$ . Alors, si tous les voisins de  $u$  sont terminés d'examiner il termine l'examen de  $u$ . Donc il termine l'examen de  $v$  avant celui de  $u$ , donc par la définition de postOrder, il remplace  $v$  dans la liste puis ajoute  $u$  en tête de liste. Ainsi, il renvoie une liste où  $u$  est placé avant  $v$ .

\* comme le graphe est acyclique, aucun voisin de  $v$  ou voisins de voisin de  $v$  ou ainsi de suite n'est  $u$  <sup>à moins un chemin  $u \rightarrow u$  par  $u \rightarrow v$  et  $v \rightarrow u$</sup>  (pas de chemin  $v \rightarrow u$ ) donc il n'y a pas de terminaison de  $u$  à ce moment-là.

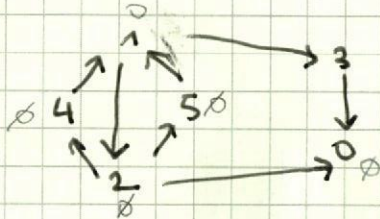
18)





# NE PAS ÉCRIRE DANS CE CADRE

Déterminons le postOrder de  $G_3^R$



postOrder :

1 3 2 5 4 0

② return (pour dfs(1))

- v = 3 dfs(3)

dfs(1) return

①

component (g3)

num = int [6]

nc = 0

visited = boolean [6]

- v = 1

dfs(1)

visited[1] = true

num[1] = 0

for w = 4

dfs(4) visited[4] = true num[4] = 0

for w = 2

dfs(2) visited[2] = true, num[2] = 0

for w = 1

dfs(1) return

return

return

w = 5

dfs(5) visited[5] = true num[5] = 0

for w = 2

dfs(2) return

return