

COMPOSITION de INF 411

en date du 16/11/2021

2507

INF411

I Expressions régulières

1) a^*b est une expression régulière qui reconnaît les chaînes de longueur au moins 1 commençant par a et finissant par b .

```
2) static String toString (SRE r) {  
    StringBuffer s = new StringBuffer("[");  
    if (r == null) : s;  
    while (r != null) {  
        s.append(r.char);  
        if (r.star == true) s += "*";  
        r = r.next;  
    }  
    return s.append("]").toString();  
}
```

```
3) static SRE ofString (String s) {  
    if (s.charAt(s.length()-1) == '*') {  
        int p = 1;  
        SRE next = new SRE (s.charAt(s.length()-2), true, null);  
    }  
    else { SRE next = new SRE (s.charAt(s.length()-1), false, null);  
        int p = 0; }  
}
```

NOTE

N°

1

2

NE PAS ÉCRIRE DANS CE CADRE

```
for (int i = s.length() - (2+p), i >= 0, i--) {  
    if (!s.charAt(i).equals('*')) {  
        if (s.charAt(i+1).equals('*')) {  
            boolean star = true;  
        } else { boolean star = false }  
        SRE pred = new SRE (s.charAt(i), star, next)  
        next = pred  
    }  
    return next  
}
```

4) 1^{er} appel ligne 2 : matches (s, 0, r) où r correspond à axa

→ comme r.star = true, on regarde matches (s, 0, r.next) où r.next correspond à a.

→ comme r.e == s.charAt(0) et r.star == false, on renvoie matches (s, 1, null)

→ Comme r = null et i = s.length(), on renvoie true.

5) la méthode matches renvoie à chaque appel : so

- soit un booléen et elle se termine

- soit matches (s, i+1, r)

- soit matches (s, i+1, r.next)

NE PAS ÉCRIRE DANS CE CADRE

- soit `matches(s, i, r.next)`

Dans les cas où elle ne se termine pas, soit i augmente et `s.length() - i` diminue ; soit r devient `r.next`.

Or, `s.length - i` est une suite décroissante minorée par 0 et l'appel à `r.next` se termine forcément par un null

Un des deux éléments finira soit en `r=null` soit en `i==s.length()` et la méthode se terminera.

6) On a vérifié à la ligne 4 que `r` n'était pas null ce qui explique que les appels à `r.star`, `r.c` et `r.next` ne provoquent pas de `NullPointerException` aux lignes 6, 10, 11, 14.

7) Pour une chaîne `s = abcde...` (N fois les lettres dans l'ordre alphabétique) et `r` correspond à `a*b*c*d*...x*a*` (dernière lettre différente de celle en `s`)

→ Pour chaque lettre, `r.star = true` et on regarde `matches(s, i, r.next)` pour lequel `r.next.star = true`, on regarde alors `matches(s, i, r.next.next)` et ainsi de suite.

→ le dernier de la chaîne `r=null` renvoie false (car $0 \neq s.length()$)

→ on appelle alors `matches(s, i, r)` jusqu'à la fin.

À chaque fois, il y a en 2 appels, d'où $O(2^N)$ est la complexité

8)

```

g) public int hashCode () {
    int h = 233 * this.s.length();
    for (int i = 0; i < s.length(); i++) {
        h = r.charAt(i) + 19 * h;
    }
    return (h & 0x7fffffff) ^ M;
}

```

où on a défini M comme final private static $M = i * r.c.length()$

```

public boolean equals (Object o)
{
    DemosKey m = (DemosKey) o;
    return (m.i == this.i) && [ this.r.toString().equals(m.r.toString()) ]
}

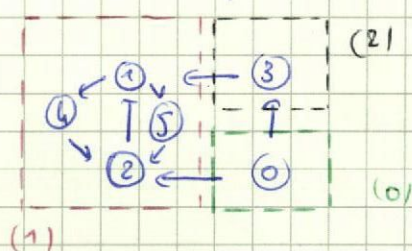
```

(a)

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

II Composantes fortement connexes

11) Les composantes fortement connexes sont :



soit le tableau $[0, 1, 1, 2, 1, 1]$

12) On parcourt la liste des sommets, considère comme étant chacun une classe et étant eux-même le représentant de cette classe.

→ à chaque sommet parcouru, on regarde s'il y a un chemin à un sommet déjà parcouru. Si c'est le cas, on réunit les classes et le représentant de cette classe est celui du sommet déjà parcouru.

→ On renvoie le tableau des représentants de chaque sommet.

~~Dans le pire des cas, on parcourt 2 fois tous les~~
la complexité est en $O(N^2)$.

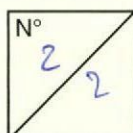
NOTE

13) 1^{er} appel dfs ($v=0$)

↳ dfs ($v=3$)

↳ dfs ($v=0$)

↳ dfs ($v=4$)



NE PAS ÉCRIRE DANS CE CADRE

puis

$\text{dfs}(v=0)$

$\text{dfs}(v=1)$

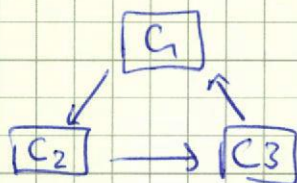
à la ^{1ère} étape de la boucle for visited = [true, true, false, true, true, false]
fin de

L'appel suivant est $\text{dfs}(v=2)$

↳ $\text{dfs}(v=5)$

↳ $\text{dfs}(v=2)$

14) Supposons que $C(G)$ contient un cycle :



alors il existe : un sommet de C_1 qui pointe vers un sommet de C_2

- un sommet de C_2 ————— C_3

- ————— C_3 ————— C_1

Donc à partir d'un sommet de C_1 , il y a un chemin vers C_3 et
réciproquement. C_1 et C_3 étant fortement connexes, ils constituent ^{ensemble}
un même groupe fortement connexe.

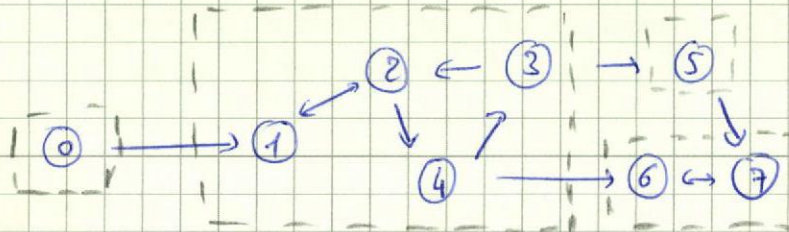
Aburde car cela contredit la $C(G)$ de départ.

NE PAS ÉCRIRE DANS CE CADRE

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

16)

G_n



la liste renvoyée par postorder [6, 7, 5, 3, 4, 2, 1, 0]

17) Pour un graphe acyclique, si $u \rightarrow v$ l'appel de u par postorder entraînera l'appel de v qui sera ajouté à order, puis u sera à son tour ajouté (en première position) à order.

18) L'ordre postfixe de G_n est [0, 5, 4, 2, 3, 1]

• 1^{er} appel: $v=0$ [true, false, false, false, false, false]

↳ $v=2$ (1^{er} voisin)

↳ $v=1$

↳ $v=4 \rightarrow v=2$

↳ $v=5 \rightarrow v=2$

[true, true, true, true, false, true]

↳ $v=3$

NE PAS ÉCRIRE DANS CE CADRE