

1. L'expression régulière :  $a \dots * b$  reconnaît les chaînes qui
- commencent par  $a$
  - terminent par  $b$
  - ont au moins 2 autres caractères (..) entre les deux, et parfois plus (.\*)

```

2. static String toString(SRE r) {
    String chaine = new String();
    if (r == null) return chaine;
    while (r != null) {
        chaine += r.char;
        if (r.star) chaine += "*";
        r = r.next;
    }
    return chaine;
}

```

```

3. static SRE ofString(String s) {
    int i = 0; SRE e = new SRE(null, false, null);
    while (i < s.length()) {
        e.c = s.charAt(i); i++;
        if (s.charAt(i).equals("*")) {
            e.star = true; i++;
        }
    }
}

```



# NE PAS ÉCRIRE DANS CE CADRE

```

3. static SRE of String (String s) {
    int i = s.length() - 1;
    SRE this = new SRE (null, false, null);
SRE prev = new SRE (null
    while (i >= 0) {
        if (s.charAt(i).equals("*")) {
            this.star = true;
            i--;
        }
        this.c = s.charAt(i);
        SRE prev = new SRE (null, false, this);
        this = prev;
    }
    return this.next;
}

```

4. On appelle `matches("aa", r) =`  
 la méthode appelée `matches("aa", 0, r)`.

( $r \Leftrightarrow "a*a"$   
 ou bien :  
 r: a → a → null  
     true      false

- on a bien  $\begin{cases} "aa" \neq \text{null} \\ 0 \leq 0 \\ 0 \leq "aa".length() = 2. \end{cases}$

$r \neq \text{null}$

$r.star$  est vrai, on vérifie donc `matches("aa", 0, r.next) =`

- on a bien les 3 conditions,  $r \neq \text{null}$ ,  $r.star$  faux,  
 $0 \neq 2$ ,  
 $r.next.c = "a" = s.charAt(0)$  et  $r.next.star$  faux  
 donc on ~~vérifie~~ renvoie  
`matches("aa", 1, r.next.next) =`



## NE PAS ÉCRIRE DANS CE CADRE

- les trois conditions,  $r.next.next == null$  vrai, on renvoie  $1 == "aa".length \rightarrow$  false

on revient ~~à~~ au 1<sup>er</sup> appel,  $matches("aa", 0, r.next)$  est false.

$0 \neq 2$

$r.c = "a" = "aa".charAt(0)$  et  $r.star$  vrai

On renvoie  $matches("aa", 1, r) =$

- 3 conditions OK,  $r \neq null$ ,  $r.star$  vrai donc on évalue  $matches("aa", 1, r.next) =$

- 3 conditions OK,  $r \neq null$ ,  $r.star$  false,  $1 \neq 2$ ,  $r.c = "a" = "aa".charAt(1)$  et  $r.next.star$  false next.

donc on renvoie  $matches("aa", 2, r.next.next)$

- $r.next.next = null$  et  $2 = 2 \rightarrow$  true

on renvoie true

on renvoie true

Par résumer :

$matches("aa", r)$

↳  $matches("aa", 0, r) = true$

↳  $matches("aa", 0, r.next) = false$

↳  $matches("aa", 1, r.next.next) = false$

↳  $matches("aa", 1, r) = true$

↳  $matches("aa", 1, r.next) = true$

↳  $matches("aa", 2, r.next.next) = true$



# NE PAS ÉCRIRE DANS CE CADRE

5. On appelle  $\text{matches}(s, i, r)$ .

À chaque appel fait conséquemment à  $\text{matches}$ , on a forcément incrémentation de  $i$ , passage à  $r.\text{next}$ , ou les deux.

Or, -  $i$  est borné par  $s.\text{length}()$  qui est finie, et lorsque  $i = s.\text{length}()$ , la ~~fonc~~ fonction se termine et renvoie `false`.

-  $r$  a un nombre fini de successeurs  $\text{next}$ , on finit donc par faire appel à un  $r.\text{next}$ . ...  $\text{next} = \text{null}$ ; alors, la fonction se termine et renvoie `true` ou `false`.

Ainsi, même en s'appelant elle-même un maximum de fois,  ~~$(s.\text{length}() - i) \times \text{nombre de successeurs de } r$~~ , la fonction terminera toujours, car elle incrémente toujours une variable à chaque appel.

6. Il est vérifié au début de la fonction que  $0 \leq i \leq s.\text{length}()$  et si  $i = s.\text{length}()$ , la fonction termine avant l'appel à  $\text{charAt}(i)$  ligne 10. Ainsi, ce lors de l'appel à  $\text{charAt}(i)$ , on est certain que  $0 \leq i \leq s.\text{length}() - 1$ .

G, 10, 11 et 14.  $r.c$  et  $r.\text{next}$   
Aux lignes 7, on appelle  $r.\text{star}$ , après avoir vérifié ligne 4 que  $r \neq \text{null}$  et terminé sinon. On n'aura donc jamais de Null Pointer Exception.

8. Pour "aa" et  $r = a \rightarrow a \rightarrow a \rightarrow \text{null}$   

```
graph LR
    r1[a] --> r2[a]
    r2 --> r3[a]
    r3 --> null
    matches1[true] --- r1
    matches2[true] --- r2
    matches3[false] --- r3
```

On doit refaire deux fois le calcul de  $\text{matches}("aa", 1, r.\text{next}.\text{next})$

9. ~~On suppose que les chaînes s ne sont jamais~~

```
public int hashCode() {
    return this.i.hashCode() + this.r.hashCode()*5003;
}
```



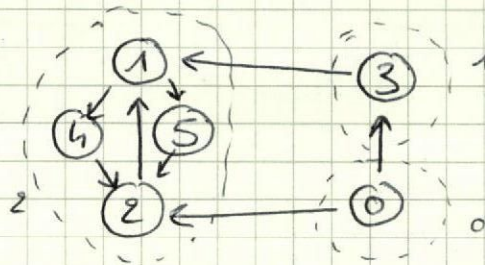
COMPOSITION d INF 411

en date du 16/11/2021

```
public boolean equals (Object o) {  
    Memokey m = new Memokey(o); (Memokey) o;  
    return this.i.equals(m.i) && this.r.equals(m.r);  
}
```

10. Dans le pire des cas, on calcule une fois chaque combinaison de  $i$  et  $r$ , donc sa complexité est  $O(N \times M)$ .

11. 0 2 2 1 2 2



12. On crée une classe UnionFind contenant 2 champs, `link` et `rank`, tous deux des tableaux d'entiers de taille  $V$ .

On initialise en donnant pour tout  $i \in [0, V-1]$ , `link[i] = i` et `rank` un tableau vide de même taille.

On écrit une fonction `find` qui regarde pour un sommet  $i$  le représentant `link[i]` de sa composante, et si ce n'est pas lui le représentant, actualise le champ `link[i]` pour avoir le seul représentant de la composante. Cela permet de rassembler chaque composante sous un unique représentant.

On écrit une fonction `union` qui unit deux composantes en égalisant les valeurs de leurs représentants (on garde celui de rang supérieur, pour optimiser la complexité).

La complexité est en  $O(\log n)$ .

NOTE



# NE PAS ÉCRIRE DANS CE CADRE

13. components (g)

↳ dfs (-, -, 0)

↳ dfs (-, -, 3)

↳ dfs (-, -, 0)  
dfs (-, -, 4)

↳ dfs (-, -, 0)  
dfs (-, -, 4) → dfs (-, -, 4)  
dfs (-, -, 3)

↳ dfs (-, -, 2)

↳ dfs (-, -, 5)

↳ dfs (-, -, 2)

↳ num = 

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

14. Supposons par l'absurde que  $C(G)$  contient un cycle, c'est-à-dire qu'on peut revenir à un même sommet  $S$  par un chemin qui parcourt plusieurs sommets.

Soit  $S'$  un des sommets parcourus par ce chemin.

On a alors des chemins qui vont de  $S$  à  $S'$  et de  $S'$  à  $S$ , donc  $S$  et  $S'$  sont fortement connexes, absurde.

15. ~~static Graph reverse (Graph g) {~~

~~Graph rev = new Graph(g.V);  
for (int v=0; v<g.V; v++) {  
for (int w: g.adj(v)) {  
if (  
rev.addEdge(w, v);  
}~~



# NE PAS ÉCRIRE DANS CE CADRE

```

15. static Graph reverse (Graph g) {
    Graph rev = new Graph (g.V);
    for (int v=0; v < g.V; v++) {
        for (int w : g.adj(v)) {
boolean arc = false;
for (int z : rev.adj(w)) {
    if (z==v) { arc = true; break; }
}
if (!arc) rev.addEdge (w,v);
        }
    }
    return rev;
}

```

16. ~~6, 7, 5, 3, 4, 2, 1, 0~~      0, 1, 2, 4, 3, 5, 7, 6

17. Soit  $g$  un graphe acyclique et  $u, v$  deux sommets tels que  $u \rightarrow v$ .

- \* Supposons que l'algorithme parvient à  $u$  en premier (par un appel de la boucle for de postOrder ou par voisinage avec un autre sommet dans dfsPost).

On a alors l'appel  $\text{dfsPost}(-, -, u)$ , qui dans sa liste d'adjacence va tomber sur  $v$  et appeler  $\text{dfsPost}(-, -, v)$ .

On aura alors l'action  $\text{order.addFirst}(v)$  avant  $\text{order.addFirst}(u)$ , par emboîtement des appels.

- \* Supposons maintenant que  $v$  soit appelé en premier. Comme le graphe est acyclique, il n'existe pas ~~d'autre chemin~~ de chemin  $v \rightarrow u$ , même indirect. Donc  $u$  ne sera appelé qu'après terminaison de  $\text{dfsPost}(-, -, v)$  et ajout de  $v$  à  $\text{order}$ .



# NE PAS ÉCRIRE DANS CE CADRE

Dans les deux cas,  $u$  vient avant  $v$  dans l'ordre à la fin de l'algorithme.

18. L'ordre postfixe donné par postOrder (reverse (63)) est :

~~0 3 1 5 4 2~~     3 1 2 5 4 0

dfs (—, —, 3)

↳ dfs (—, —, 0)

dfs (—, —, 1)

↳ dfs (—, —, 3) x

dfs (—, —, 2)

↳ dfs (—, —, 0) x

↳ dfs (—, —, 4)

↳ dfs (—, —, 1) x

↳ dfs (—, —, 5)

↳ dfs (—, —, 1) x

↳ num = 

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

~~19.~~

20. ~~L'algorithme de Kosaraju-Sharir~~ est le même que

Invariant de boucle : tous les sommets  $i$  dont  $\text{num}[i]$  est égal appartiennent à la même comp. fortement connexe