

Exercice 1

1 $\alpha \dots * 6$

2

```
static String toString(SRE r) {  
    String str = "";  
    return toString(r, str);  
}
```

```
static String toString(SRE r, String str) {  
    if (r == null)  
        return str;  
    str = str + c;  
    if (star)  
        str = str + "*";  
    return toString(r.next, str);  
}
```

NOTE

NE PAS ÉCRIRE DANS CE CADRE

[3]

```

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

```


NE PAS ÉCRIRE DANS CE CADRE

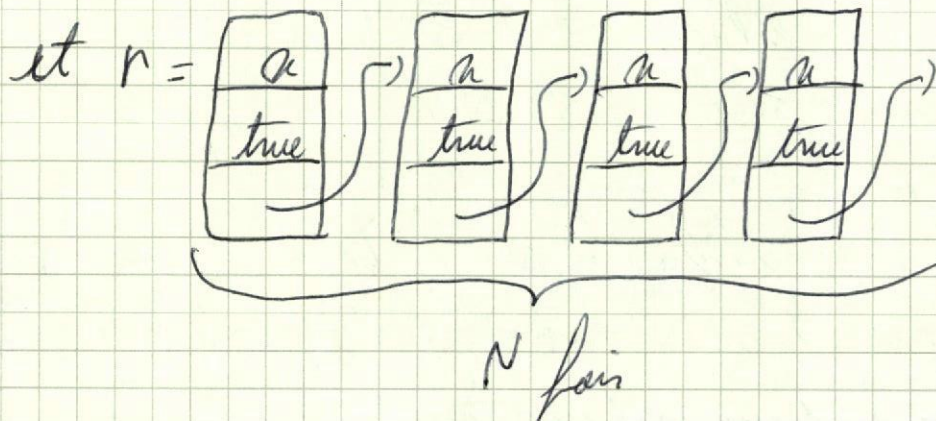
[4] D'abord, nous avons un premier appel :
return matcher(s, 0, n); (ligne 19)
qui engendre le test matcher(s, 0, n.next) (ligne 6)
qui retourne le résultat de matcher(s, 1, null) (ligne 14)
qui retourne false (test de la ligne 5)
Puis, on retourne matcher(s, 1, n) (ligne 12)
qui appelle le test matcher(s, 1, n.next) (ligne 6)
qui retourne matcher(s, 2, null) (ligne 14)
qui retourne true (test de la ligne 5)
Enfin, nous obtenons return true; (ligne 7)

[5] Le seul endroit où l'on risquerait de tomber dans une boucle est le test de la ligne 6. Cependant, nous appelons matcher avec l'argument n.next, et comme les données sont finies, à un moment n.next est null, et le test de la ligne 4 nous ferait sortir de la boucle.

[6] Le Null Pointer Exception est impossible grâce au test de la ligne 4.
En outre, on n'accède jamais s en dehors de ses bornes grâce à la commande next de la ligne 3.

NE PAS ÉCRIRE DANS CE CADRE

⑦ Nous obtenons ce temps de calcul pour
 $S = \underbrace{a a a a a \dots}_{N \text{ fois}}$



Le temps de calcul est proportionnel à 2^N car pour chaque nouveau a , on ajoute un appel au test $\text{match}(a, i, r.\text{next})$ qui retourne `false`, mais double le nombre d'appels à match .

COMPOSITION d INF411
en date du 16/11/21

8

NOTE

N°
2/4

9

```
@Override
public int hashCode() {
    return 5003 * i + v.c.hashCode();
}
```

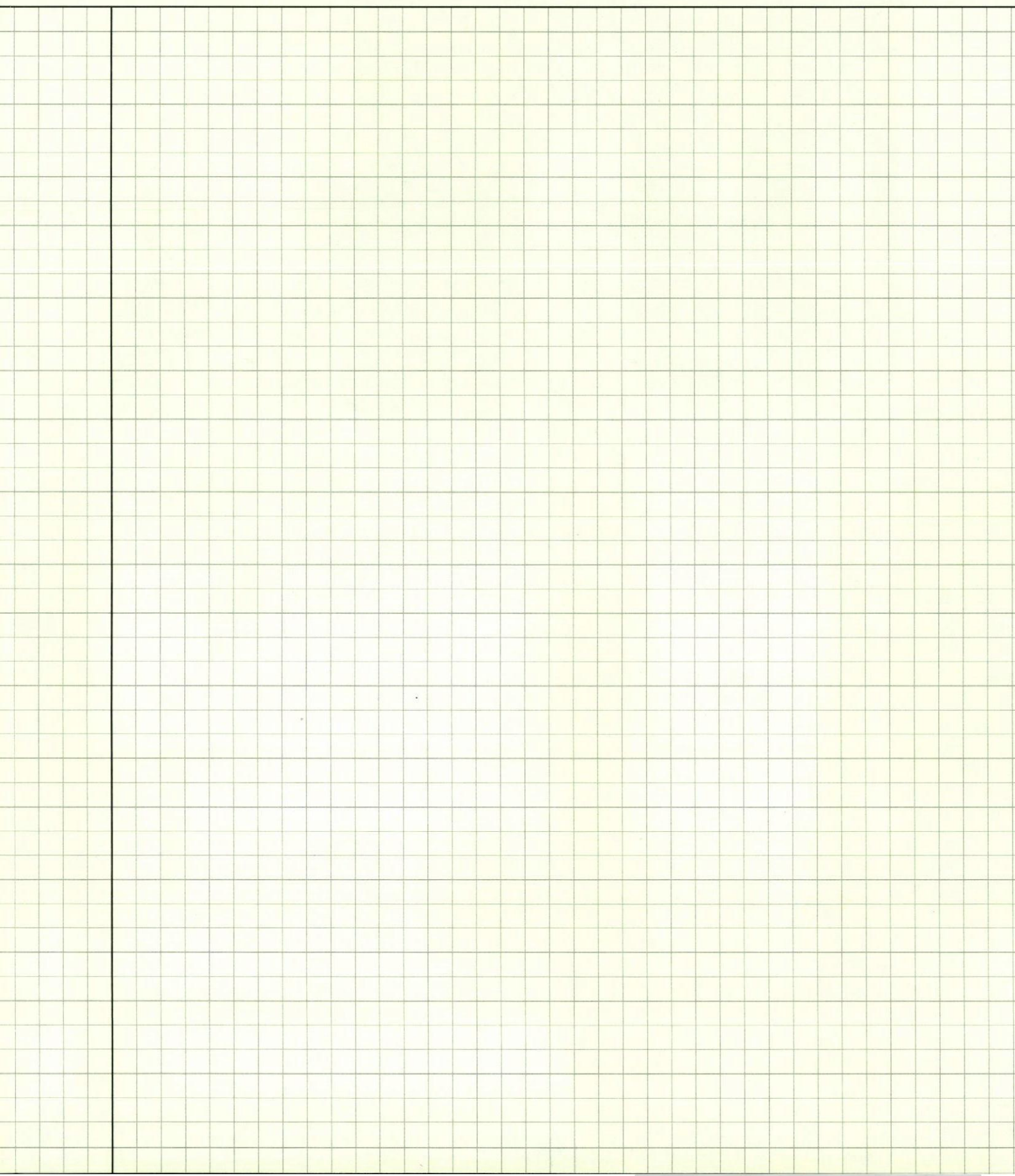
10

```
@Override
public boolean equals(Object o) {
    MemoKey that = (MemoKey) o;
    return (this.i == that.i &&
            this.v.c.equals(that.v.c) &&
            this.v.stor == that.v.stor);
}
```

10

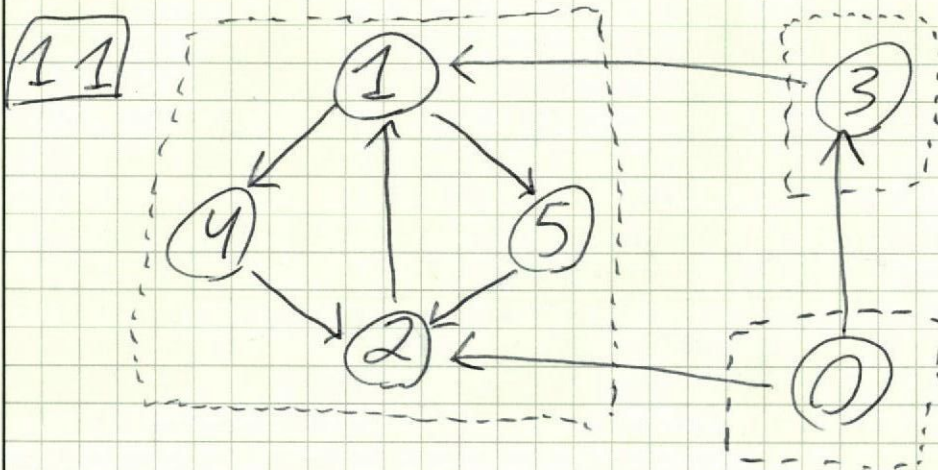
Nous obtenons une complexité $O(NM)$
grâce à la mémorisation

NE PAS ÉCRIRE DANS CE CADRE



NE PAS ÉCRIRE DANS CE CADRE

Exercice 2



Les composantes connexes sont identifiées par des pointillés, on pourrait donner en réponse le tableau suivant :

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

12 Une stratégie simple serait de parcourir les sommets V du graphe G , puis parcourir les voisins V' de V , en utilisant la méthode union (V, V')

La complexité de cet algorithme est quadratique sur le nombre de sommets.

NOTE

NE PAS ÉCRIRE DANS CE CADRE

13 On a une première série d'appels avec $MC = 0$:

$dfs(reinited, q, 0)$	(ligne 10)
$dfs(reinited, q, 3)$	(ligne 22)
$dfs(reinited, q, 0)$	(" ")
$dfs(reinited, q, 4)$	(" ")
$dfs(reinited, q, 0)$	(" ")
$dfs(reinited, q, 1)$	(" ")
$dfs(reinited, q, 4)$	(" ")
$dfs(reinited, q, 3)$	(" ")
$dfs(reinited, q, 4)$	(" ")

Puis, une deuxième série avec $MC = 1$

$dfs(reinited, q, 2)$	(ligne 10)
$dfs(reinited, q, 5)$	(ligne 22)
$dfs(reinited, q, 2)$	(" ")

Enfin, composants renvoie $(0, 0, 1, 0, 0, 1)$

NE PAS ÉCRIRE DANS CE CADRE

14 Supposons que $C(G)$ contient un cycle.
 Alors, il existe un chemin partant de $v_1 \in C_1 \in C(G)$
 passant par $v_2 \in C_2 \in C(G)$ et qui retourne à v_1 .
 Or $C_1 \neq C_2$, cependant peut être que'il existe
 un chemin de v_1 vers v_2 et un chemin de v_2
 vers v_1 , v_1 et v_2 sont alors dans la même
 composante connexe : $C_1 = C_2$ ce qui est absurde.
 Alors, $C(G)$ ne contient pas de cycle.

15

```

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


16 N'ayant pas trouvé le graphe G^R sur le sujet, je suppose que la question porte sur le graphe G , dans ce cas, le tableau renvoyé est :

~~$(0, 3, 2, 1, 5, 4)$~~

17

16 Le tableau renvoyé est :

$(0, 1, 2, 4, 3, 5, 7, 6)$

17

Soit G acyclique tel que $\exists u, v \in G, u \rightarrow v$
 alors $\neg v \rightarrow u$, et il n'y a pas de chemin de v
 vers u car G est acyclique.

Dans G^R , on a l'inverse, $v \rightarrow u$ et on n'a pas de
 chemin de u vers v . Alors u apparaît forcément
 avant v car dans le parcours de Post Order
 v ne sera renvoyé qu'après avoir parcouru tous
 ses voisins. Post Order aura alors déjà parcouru et
 renvoyé u .

COMPOSITION d INF411

en date du 16/11/21

(18) 2) d'abord, postOrder(reverse(α)) renvoie le tableau suivant: (1, 3, 2, 5, 4, 0)

Ce sera l'ordre dans lequel on déroule l'algorithme:

$nc = 0$

dfs(visited, α , 1)

dfs(visited, α , 4)

dfs(visited, α , 2)

dfs(visited, α , 1)

dfs(visited, α , 5)

dfs(visited, α , 2)

$nc = 1$

dfs(visited, α , 3)

dfs(visited, α , 1)

$nc = 2$

dfs(visited, α , 0)

dfs(visited, α , 2)

dfs(visited, α , 3)

Le tableau renvoyé est: (2, 0, 0, 1, 0, 0)

NOTE

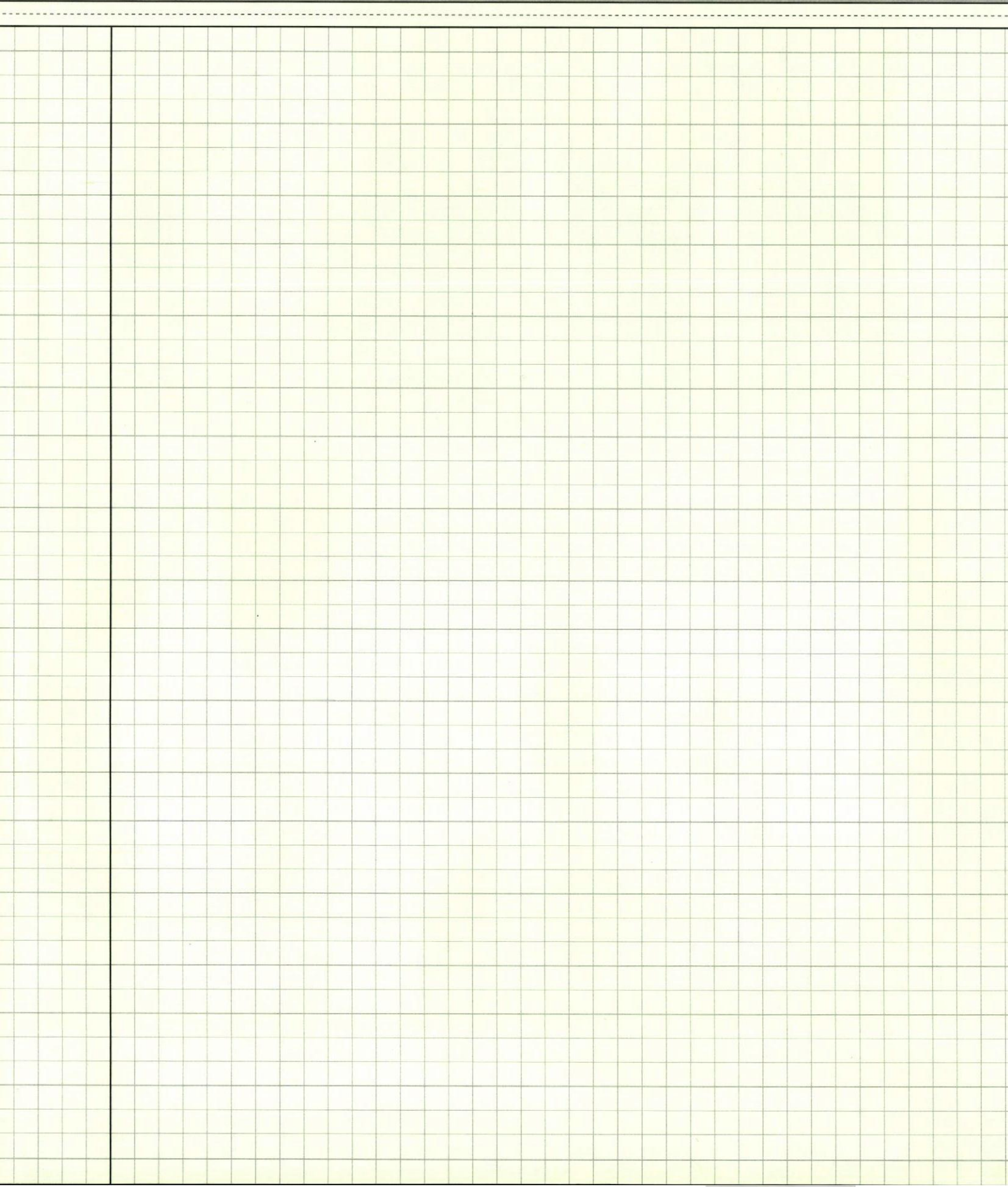
[29]

L'algorithme de Kerasaju-Shamir a une complexité en temps proportionnelle au nombre E d'arcs de G et plus du nombre V de sommets de $C(G)$.

En reportant le pire cas, $\text{card}(G) = \text{card}(C(G))$ on obtient la complexité $O(E+V)$

Or, nous avons besoin de parcourir G , c'est optimal.

NE PAS ÉCRIRE DANS CE CADRE



NE PAS ÉCRIRE DANS CE CADRE