

COMPOSITION d'inf 411
en date du 16/11/2021

2506

INF411

Question 1: "a...*b" convient

Question 2: static String toString(SRE r) {

```
String res = "";  
while (r != null) {  
    res = res + r.c;  
    if (r.star) res = res + "*";  
    r = r.next;  
}  
return res;  
}
```

Question 3: static SRE of String(String s) {

~~SRE res;~~

~~for (int i = 0; i < s.length(); i++) {~~

~~SRE res;~~

```
if (s == "") return res;  
res = new SRE(s.charAt(0), false, null);  
int i = 1;  
int n = s.length();  
char c;  
while (i < n) {
```

```
    c = s.charAt(i);  
    if (c == "*") res.star = true;
```

```
    else {
```

```
        res.next = new SRE(c, false, null);
```

```
    }
```

```
    i++;
```

```
return res;
```

NOTE

N°

1

3

NE PAS ÉCRIRE DANS CE CADRE

Ques. 4: ① matches("aa", i=0, r)

② test dans le 2^{ème} "if": ~~set~~.star = true

puis matches(s, i=0, r.next)

③ test dans le 4^{ème} "If" $r.c == s.charAt(i)$

puis $r.star = false$ donc:
appel à matches(s, i+1=1, r.next=null)

④ test dans le 1^{ère} "If": $r = null$ donc:

On obtient le booléen $(i == s.length)$

soit $(1 == 2)$

c'est-à-dire false

on obtient false

la condition ② matches(s, i=0, r.next) en test
donc pas vérifiée

test dans le 4^{ème} "If": on a bien $r.c == s.charAt(i)$

Test du "If" imbriqué: on a bien $(r.star)$

⑤ return matches(s, i+1, r)

Test dans le 2^{ème} "If":

$(r.star) = true$ et:

⑥ matches(s, i+1=1, r.next)

1^{ère} "If": $r.c == s.charAt(i)$

vérifié

et $r.star = false$

donc

⑦ matches(s, 2, null)

1^{ère} "If": $r == null$ donc return...

NE PAS ÉCRIRE DANS CE CADRE

1^{er} If: $r == \text{null}$ donc $\text{return } i = s.\text{length}$ soit $(2=2)$ c'est True
et on a un ⑤ : c'est donc vrai (matches(s, 1, r))

On obtient donc vrai en réponse de l'algorithme.

Question 5: La méthode `matches` termine toujours, car à chaque appel récuratif ~~elle s'appelle~~ qui doit fournir le résultat de l'algorithme (lignes 12 et 14), l'indice est incrémenté de $+1$. Or, on a la condition $i \leq s.\text{length}()$ (testée au début).

Question 6: D'une part, la condition $0 \leq i \leq s.\text{length}()$ est testée dès l'entrée de l'algorithme. D'autre part, la condition $i == s.\text{length}()$ est testée dans la 3^e branche "If" (lignes 8 et 9) avant que puisse être fait appel à `s.charAt(i)`. Quand cette ^{valeur} condition est ~~testée~~ appelée, on a bien $i \leq s.\text{length}()$.

Par ailleurs, il ne peut y avoir d'exception `NullPointerException` car l'éventualité est écartée dès la première branche If: (lignes 4 et 5):

```
if (r == null)
```

```
return i == s.length();
```

Question 7: Pour avoir un tel temps, on veut que chaque appel à `matches` fasse lui-même à nouveau 2 fois appel à `matches`. ~~Subant~~

~~qui à chaque appel~~ Il faut donc que la condition de la 2^{ème} branche If soit testée à chaque fois: `r.charAt(i) && matches(s, i, r.next)`

On veut donc une expression régulière r de la forme:

$a^* a^* a^* \dots a^*$

N itérations de a^*

et une chaîne: $aa \dots a$

N itérations de a

NE PAS ÉCRIRE DANS CE CADRE

Question 8: Dans l'exemple de la qu. 4, il est fait deux fois appel à la fonction `matches` avec les arguments $i = 1$ et

l'expression régulière r next correspondant à "a".
L'exemple de la question 7) contient aussi forcément car il y a plus $N \times M$ appels différents à `matches`, ce qui est plus petit que 2^N pour N assez grand.
(avec M la longueur de r)

Question 9:

```
public int hashCode () {
```

```
    int h = 0;
```

```
    String s = toString(this.r);
```

```
    for (int j = 0; j < toString(this.r).length(); j++)
```

```
        h = s.charAt(j) + 13 * h;
```

```
    h = i + 13 * h;
```

```
    return (h & 0x7fffffff) % M;
```

```
}
```

(où on a pris M la longueur de l'expression régulière r)

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

```
    MemoKey m = (MemoKey) o;
```

```
    return this.i.equals(m.i) && this.r.equals(m.r);
```

```
}
```

Question 10:

La complexité du calcul est contrôlée par le cardinal du nombre

$\{0, i+1\}$ maximal d'appels différents que l'on peut faire à `matches`. Elle est donc en $O(NM)$.

COMPOSITION d'Inf 411
en date du 16/11/2021



II Composantes fortement connexes

Question 11 :

(1, 5, 2, 4)

est une composante fortement connexe.

(3)

et (6)

aussi.

NOTE

N°

2

3

NE PAS ÉCRIRE DANS CE CADRE

Question 12: (Pour chaque élément i , on évalue sa valeur avec l'opération $\text{find}(i)$ et on assigne à la i^{e} ligne du tableau renvoyé par composants cette valeur)

Mais avant cela, on parcourt les N éléments $i \in [0, N-1]$
et pour chaque élément j auquel i est lié, on effectue l'opération $\text{union}(i, j)$

On obtient donc une complexité en $O(N^2)$.

Question 13:

① $\text{dfs}(\text{visited}, g, v=0)$

état de mem

0	0		0	0	

② $\text{dfs}(\text{visited}, g, v=2)$

→ tableau renvoyé par composants au final

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

Question 14: Si $C(G)$ contient un cycle alors pour n'importe quels points A et B visités dans deux sommets différents ^{de ce cycle}, il existerait un chemin de A vers B et un chemin de B vers A : c'est absurde car alors A et B devraient faire partie de la même composante fortement connexe.

Question 15:

static Graph reverse (Graph g)

Graph miroir = new Graph($g \circ V$)

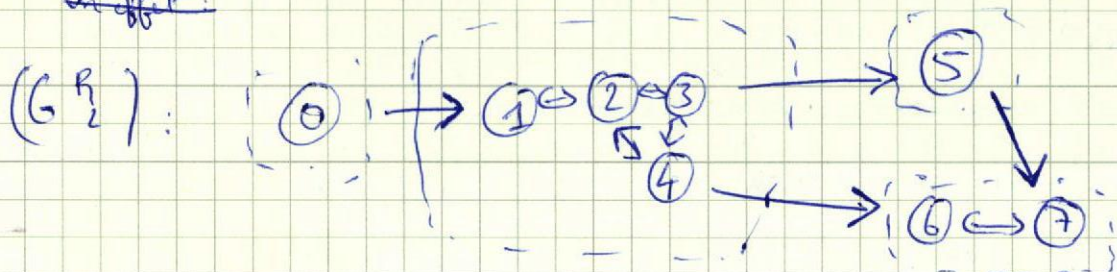


NE PAS ÉCRIRE DANS CE CADRE

```
for (int i = 0; i < g.V; i++)
    for (int j : g.adj(i))
        miroir.addEdge(j, i);
return miroir;
```

Question 16: 0 1 5 7 6 4 3 2 4

En effet:



dfsPost (visited, g, v = 0)

appelle récursivement dfsPost pour v = 1

qui appelle récursivement dfsPost pour v = 2

On a donc les schéma d'appels récursifs etc...

suivant: 0 → 1 → 2 → 3 → 4 → 6 → 7

→ 5.

On a donc:

order =

0 1 2 3 5 4 6 7

Question 17:

dans le cas où visited[u] est false

Supposons que dfs(visited, g, u) est appelé: il va alors appeler

récursivement dfs(visited, g, v). Il y a 2 cas de figure:

- soit visited[v] = true: auquel cas v a déjà été ajouté à l'orden et sera donc affiché à la droite de u.
- soit visited[v] = false: dans ce cas, v est rajouté à l'orden et sera affiché à droite de u aussi.

NE PAS ÉCRIRE DANS CE CADRE

dans ces deux cas, u apparaît ^{à gauche de} avant v dans $order$.

~~Supposons maintenant que $visited[u]$ est True : alors supposons par l'absurde que à ce moment là, $visited[v]$ est False. cela signifie que u a été visité avant v , et donc alors il~~

~~Supposons maintenant~~ ~~par l'absurde~~ que v est visité avant u : comme le graphe est acyclique, il n'existe pas de chemin qui relie v à u , il va donc être affiché à la droite de u également car il ne va pas faire d'appel récursif à $dfsPost$ avec l'argument u .

Dans tous les cas, on a donc bien u qui apparaît avant v dans la liste.

Question 18 : ordre postfixe de G_3^R : 1 3 2 5 4 0

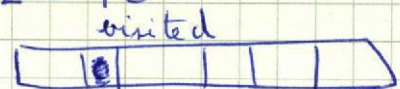
$mc = 0$

$dfs(visited, g, 1)$

↳ $dfs(visited, g, 4)$

↳ $dfs(visited, g, 2)$

↳ $dfs(visited, g, 5)$



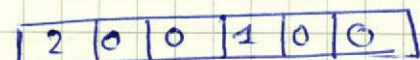
$mc = 1$

$dfs(visited, g, 3)$



$mc = 2$

$dfs(visited, g, 0)$



COMPOSITION d'inf 6,11
en date du 16/11/2021

Question 19:

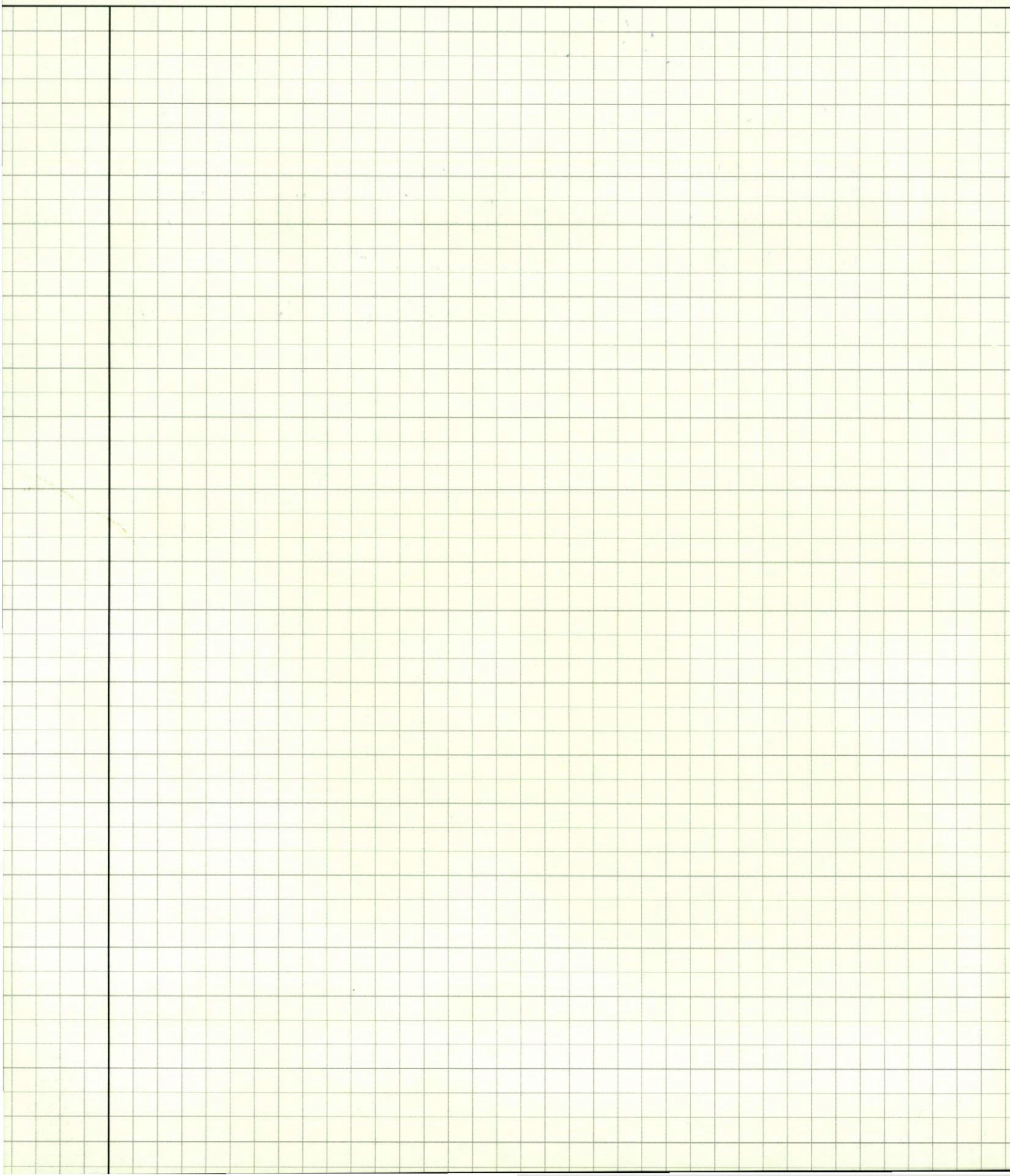
Grâce au tableau visited, comme on le voit à la question précédente, le nombre d'appels ^{à dfs} est ^{limité par} ~~exactement égal~~ à V (appels récursifs inclus). De même pour le calcul de Order.

On a donc une complexité temporelle en $O(V)$

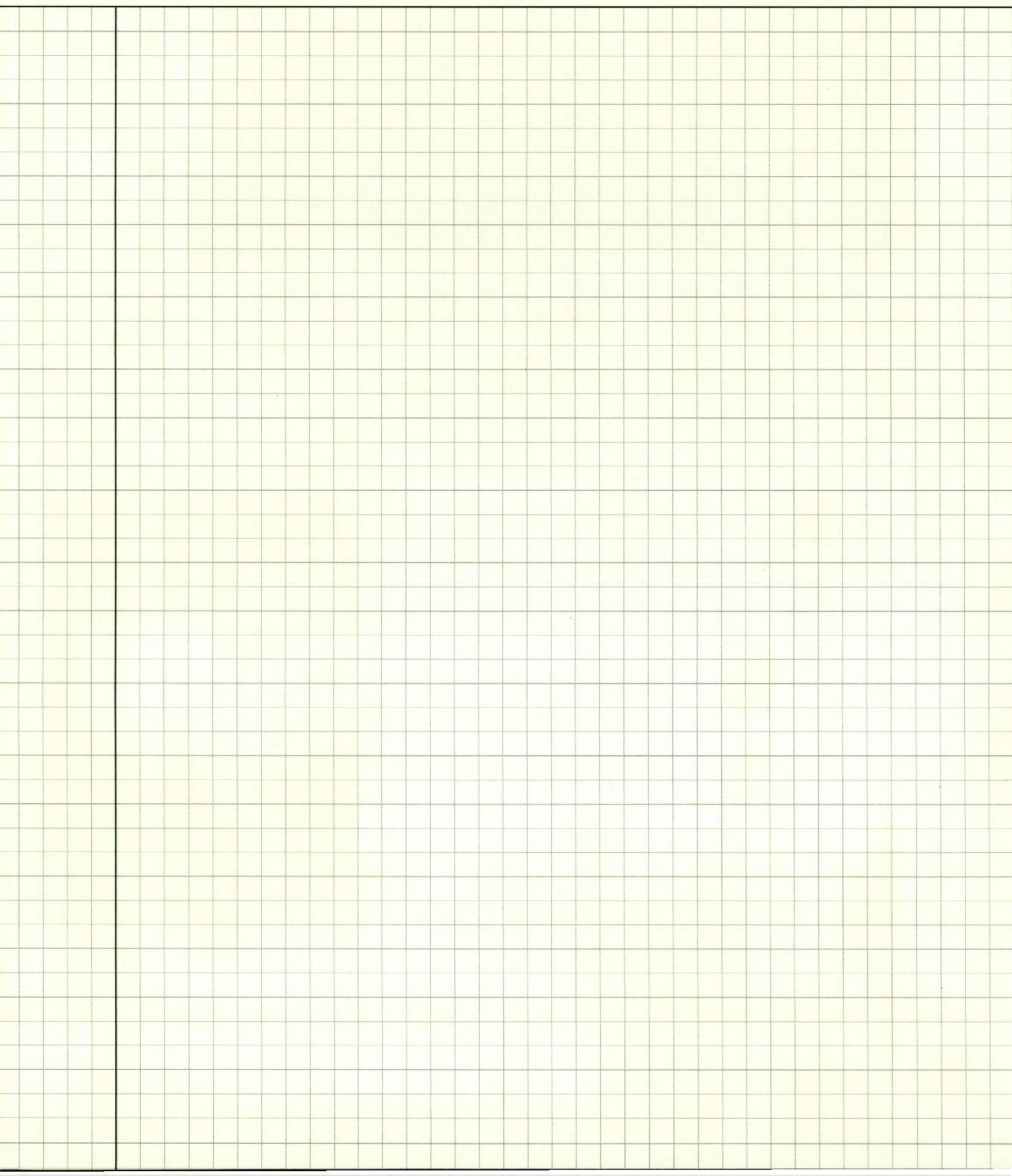
NOTE

N°
2
3

NE PAS ÉCRIRE DANS CE CADRE



NE PAS ÉCRIRE DANS CE CADRE



NE PAS ÉCRIRE DANS CE CADRE