



2505

INF411

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

## 1. Expressions régulières

Question 1: l'expression suivante reconnaît les chaînes de longueur au moins 4, commençant par a et se terminant par b :

$a \dots * b$

Question 2:

```
static String toString (SRE r) {  
    SRE sre = r ;  
    String s = "" ;  
    while (sre != null) {  
        s += sre.char ;  
        if (sre.star) {  
            s += "*" ;  
        }  
        sre = sre.next ;  
    }  
    return s ;  
}
```

NOTE



## NE PAS ÉCRIRE DANS CE CADRE

Question 3:

on utilise une méthode auxiliaire :

```
static SRÉ of String (String s, int i) {  
    if (i >= s.length()) {  
        return null;  
    }  
    if (i+1 < s.length() && s.charAt(i+1) == '*') {  
        return new SRÉ (s.charAt(i), true, of String (s, i+2));  
    }  
    else {  
        return new SRÉ (s.charAt(i), false, of String (s, i+1));  
    }  
}
```

il vaut alors :

```
static SRÉ of String (String s) {  
    return of String (s, 0);  
}
```

Question 4:

le premier appel à matches ("aa", 0, n) tombe dans la 4<sup>e</sup> condition : on a n.c == "aa".charAt(0) = 'a' comme n.star, on appelle matches ("aa", 1, n) cette fois-ci, on tombe dans la 2<sup>e</sup> condition : on a n.star et matches ("aa", 1, n.next)



## NE PAS ÉCRIRE DANS CE CADRE

en effet, l'appel à `matches("aa", 1, n.next)` tombe dans la 4<sup>e</sup> condition : on a `n.next.c == "aa"`, donc `ht(i) == 'a'` comme `!n.next.stor`, on appelle `matches("aa", 2, n.next.next)` ce dernier appel tombe dans la 1<sup>ère</sup> condition, car `n.next.next == null` comme `l == "aa"`, `length()`, on retourne `true` en conclusion, `matches("aa", n)` donne `true`

### Question 5:

pour justifier la terminaison de `matches`, qui utilise une méthode auxiliaire récursive, on utilise un invariant de boucle. Lorsque on regarde la méthode auxiliaire, on constate qu'il y a 4 conditions les 3 1<sup>ères</sup> permettent de renvoyer `true` / `false` immédiatement (tests triviaux) la 4<sup>e</sup> donne lieu à un appel récursif, mais avec `i+1` à la place de `i` ainsi, si  $i \uparrow$ , on arrivera tôt ou tard à une des 3 1<sup>ères</sup> conditions. Toutefois, dans la 2<sup>e</sup> condition, on appelle `matches(i, n.next)` cette fois, on n'a pas  $i \leftarrow i+1$ , mais  $n \leftarrow n.next$



# NE PAS ÉCRIRE DANS CE CADRE

ainsi, dans tous les cas on avance (dans  $s$  ou dans  $n$ ), ce qui nous assure que l'algorithme termine

## Question 6:

il s'agit de justifier qu'on n'aura pas  $i \geq s.length()$  à la ligne 10 ni  $n == null$  aux lignes 6, 10, 11 et 14

- le 1<sup>er</sup> point est assuré par la ligne 8, où la condition  $i \geq s.length()$  donne false

- le 2<sup>nd</sup> point est assuré par la ligne 4 avec le test  $n == null$

→ ainsi, on n'aura pas d'accès à  $s$  en dehors des bornes ni de NullPointer Exception

## Question 7:

on veut taper à chaque fois dans la 1<sup>ère</sup> sous condition de la 4<sup>e</sup> condition de matches il faut donc à chaque étape :

$p.c = '.'$  ou  $n.c = s.charAt(i)$

$\downarrow$   $n.star$

ainsi, on multipliera le nombre d'appels par 2 à chaque étape, d'où un temps en  $2^n$

ainsi, une expression de la forme suivante devrait faire l'affaire :  $s = 'aa...a'$  et  $n = a^* a^* \dots a^* b$



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

Question 8 :

mémorisation = ne pas calculer 2 fois la même chose  
on reprend l'exemple précédent (Question 7) :

$s = "aaa"$  et  $n = a * a * \dots * a * b$

à chaque étape on effectue le test de la  
2<sup>e</sup> condition :  $\text{if } (n.\text{str} \&\& \text{matches}(s, i, n, \text{rest}))$

il échoue, donc on effectue aussi le test  
de la 1<sup>ère</sup> sous condition de la 4<sup>e</sup> condition,

qui donne lieu à l'appel récursif  $\text{matches}(s, i+1, n)$

à chaque étape, on effectue deux fois le  
même calcul

en "sauvegardant" les résultats, la mémorisation permet  
d'éviter ce problème

NOTE

N° 9  
4



# NE PAS ÉCRIRE DANS CE CADRE

Question 9:

```
public boolean equals (Object o) {  
    MemoKey that = (MemoKey) o;  
    if (this.i != that.i) {  
        return false;  
    }  
    Sak n1 = this.n;  
    Sak n2 = that.n;  
    while (n1 != null && n2 != null) {  
        if (n1.c != n2.c || n1.stor != n2.stor) {  
            return false;  
        }  
        n1 = n1.next;  
        n2 = n2.next;  
    }  
    if ((n1 == null && n2 != null) || (n1 != null && n2 == null)) {  
        return false;  
    }  
    return true;  
}
```



## NE PAS ÉCRIRE DANS CE CADRE

```
public int hashCode () {  
    int sum = 0 ;  
    for ( n = this.n ;  
        while (n != null) {  
        if (n.start) {  
            sum += 10 * n.c ;  
        }  
        else {  
            sum += n.c ;  
        }  
        n = n.next ;  
    }  
    return this.i * sum ;  
}
```

en effet, les caractères codent des entiers

### Question 10 :

avec la récursion, chaque calcul est effectué au plus 1 fois  
comme on avance dans  $n$  ou dans  $n$  à chaque étape  
(ce qui assure que l'algorithme termine), on retombe  
sur une complexité en temps linéaire en  $O(N+K)$   
en revanche, on a une complexité en espace du fait



NE PAS ÉCRIRE DANS CE CADRE

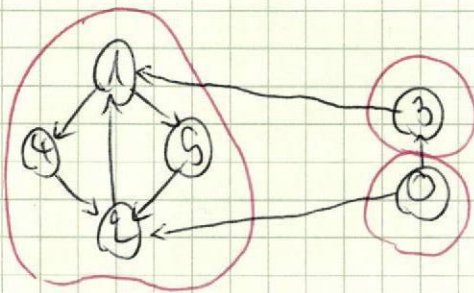
du stockage, mais ce n'est pas un problème



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

## 2. Composantes fortement connexes

Question 11:



on a 3 composantes fortement connexes, ce qui donne le tableau suivant :

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

Question 12:

pour chaque sommet  $i$  de  $0$  à  $V-1$ , on considère ses voisins et on fait sommet.union(besin)  
 à la fin, on a bien nos  $N$  classes disjointes (composantes fortement connexes)

l'appel à find nous donne un représentant de la composante

toutefois, pour construire le tableau composants, il faudra bien numéroter les classes de  $0$  à  $N-1$  (on ne peut pas se contenter de prendre le représentant)

NOTE

N°  
 3  
 4



# NE PAS ÉCRIRE DANS CE CADRE

et l'algorithme est naïf : complexité en  $O(V)$  dans le meilleur cas et en  $O(V^2)$  dans le pire cas

Question 13:

séquence des appels à dfs :

$$\begin{cases} \text{dfs}(\text{visited}, z, 0) \\ 1 \text{ est déjà dans visited} \\ \text{dfs}(\text{visited}, z, 1) \\ 3, 4, 5 \text{ sont déjà dans visited} \end{cases}$$

tableau composants :

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

Question 14:

supposons (par l'absurde) que  $C(G)$  contient un cycle alors on peut trouver un chemin de longueur au moins 2 entre une composante et elle-même tous les sommets des composantes se trouvant sur ce cycle sont fortement connectés, i.e. appartenant à une même composante connexe ceci implique que la composante considérée n'est pas maximale pour l'inclusion — contradiction ! ainsi,  $C(G)$  est acyclique



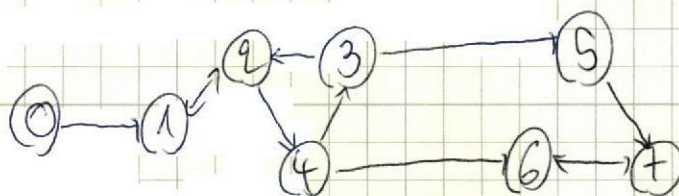
# NE PAS ÉCRIRE DANS CE CADRE

Question 15:

```
static Graph reverse (Graph g) {
    Graph gn = new Graph (g.V);
    for (int i = 0; i < g.V; i++) {
        for (int j = g.adj(i)) {
            gn.addEdge(j, i);
        }
    }
    return gn;
}
```

Question 16:

graphe  $G_a$ :



la liste renvoyée par postOrder est la suivante:  
 0, 1, 2, 4, 3, 5, 7, 6  
 en effet, l'appel à dfsPost (visited, g, 0) entraîne récursivement  
 les appels à dfsPost (visited, g, v) pour  $v = 1, 2, 3, 4, 5, 6, 7$



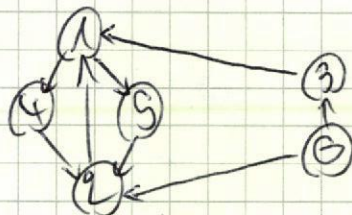
# NE PAS ÉCRIRE DANS CE CADRE

## Question 17:

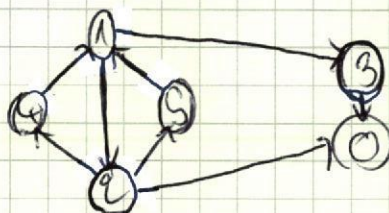
supposons le graphe acyclique  
 alors il n'existe pas de chemin de longueur au  
 moins 2 entre un sommet du graphe et lui-même  
 lors de l'exécution de post Order, on  
 déclenchera l'appel à dfs Post sur les sommets du  
 graphe, puis sur leurs voisins non visités et ainsi  
 de suite récursivement  
 pour tout arc  $u \rightarrow v$ , le sommet  $u$  sera  
 visité avant le sommet  $v$   
 ainsi, dfs Post (visited,  $g$ ,  $u$ ) précède dfs Post (visited,  $g$ ,  $v$ )  
 donc order.add First ( $v$ ) précède order.add First ( $u$ )  
 donc  $u$  apparaît avant  $v$  dans la liste renvoyée par  
 post Order, d'où l'idée de tri topologique du graphe

## Question 18:

$G_3$ :



$G_3^R$ :





COMPOSITION d

10/4/11

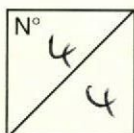
en date du

16/11/2011

liste nouvelle par post Order (reverse ( $G_3$ )) :

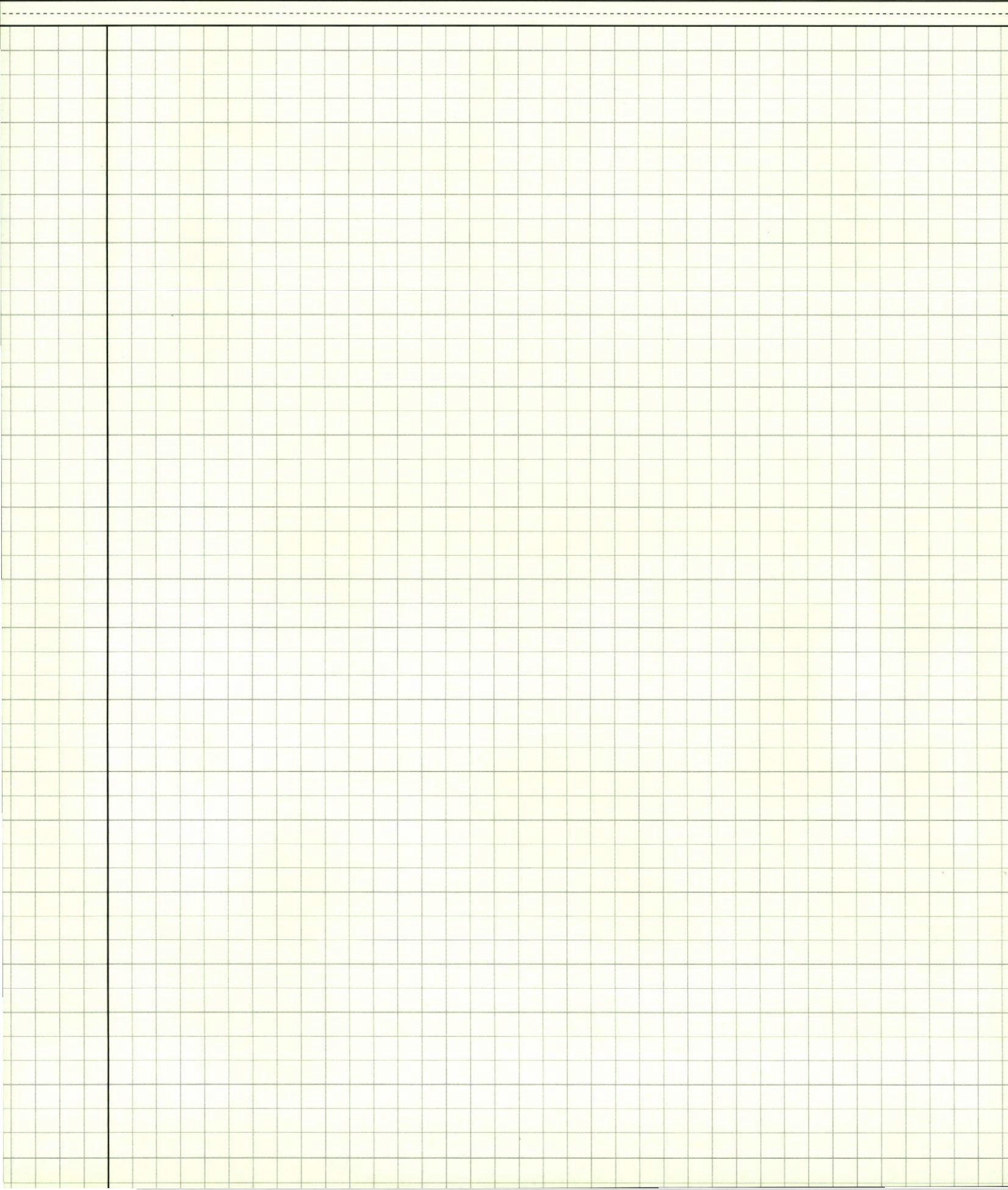
1, 3, 2, 5, 4, 0

NOTE



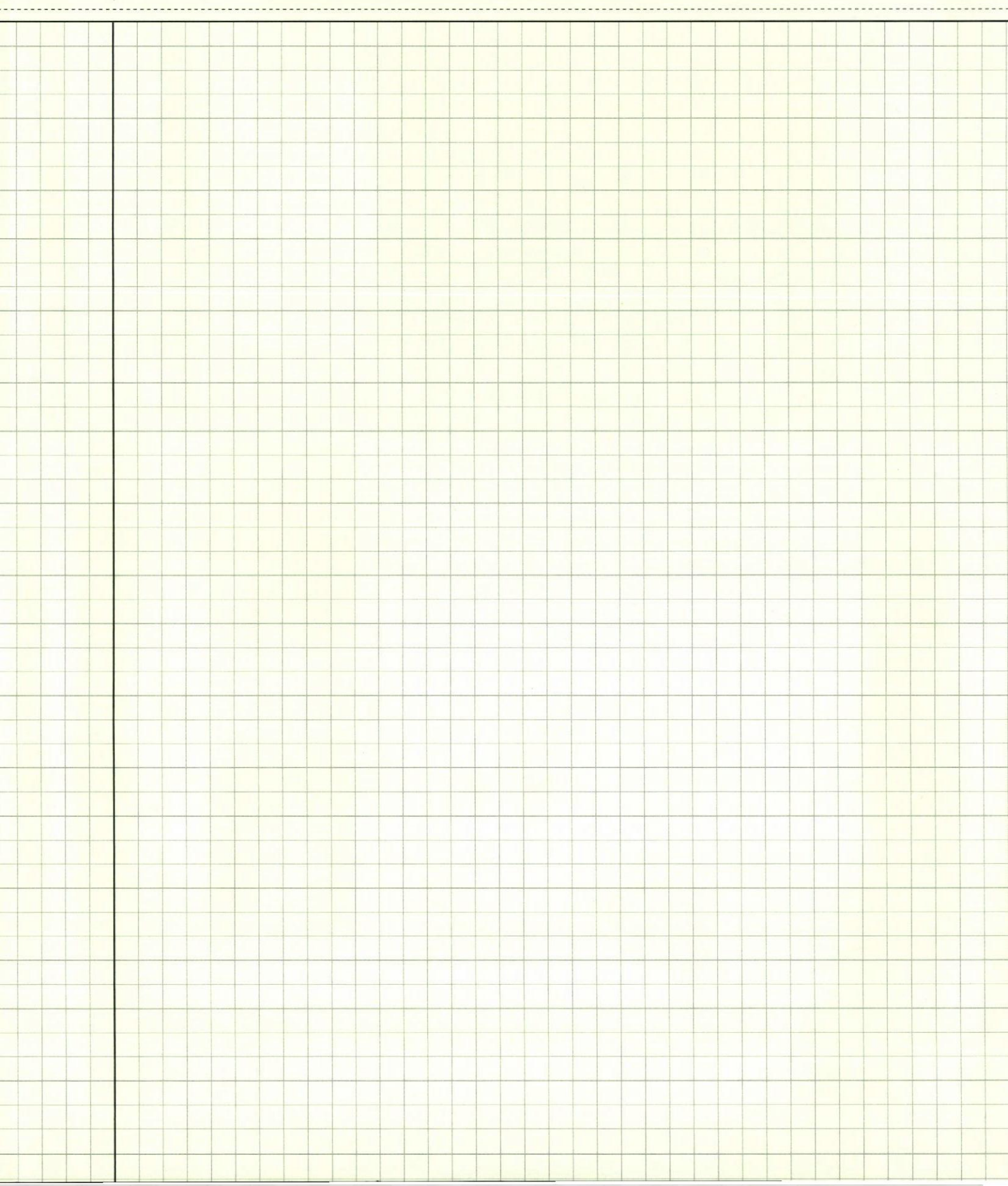


NE PAS ÉCRIRE DANS CE CADRE





NE PAS ÉCRIRE DANS CE CADRE





NE PAS ÉCRIRE DANS CE CADRE