

# CEOI 2026



Jour 1  
(français)

Day 1  
(French)



# Ornithologues

*Limite de temps: 10 s      Limite de mémoire: 512 MiB*

La Société des ornithologues de San Serriffe possède une structure interne curieusement surdimensionnée et en perpétuelle évolution. La Société se compose de  $n$  sections, et chaque membre de la Société appartient à exactement une section. Les sections sont numérotées de 1 à  $n$  et la  $i$ -ème section compte  $m_i$  membres. Ainsi, la Société compte au total  $M = m_1 + m_2 + \dots + m_n$  membres.

Chaque section est dirigée par l'un de ses membres, qui dans ce rôle est appelé l'*officier* de la section. Les officiers sont numérotés de la même manière que les sections, de sorte que (pour chaque  $i = 1, \dots, n$ ) l'officier  $i$  est celui qui dirige la section  $i$ .

De plus, les officiers sont organisés hiérarchiquement selon un système de mentorat : chaque officier sauf un possède un *mentor*, qui est l'officier d'une autre section. Le seul officier sans mentor est le *Président* de la Société. Si l'officier  $a$  est le mentor de l'officier  $b$ , on dit aussi que l'officier  $b$  est un *disciple* de l'officier  $a$ . Aucun officier n'est, directement ou indirectement, son propre mentor ; ainsi, en suivant la suite depuis un officier vers son mentor, puis vers le mentor de son mentor, et ainsi de suite, on finit toujours par atteindre le Président.

Nous définissons l'*influence* d'un officier comme la somme du nombre de membres de sa section et des influences de tous ses disciples (s'il en a). On verra aisément que l'officier ayant la plus grande influence est le Président, dont l'influence vaut toujours  $M$ . Un officier est appelé *officier supérieur* si son influence est  $\geq M/2$ .

Les statuts de la Société stipulent que l'officier supérieur ayant la plus faible influence (parmi tous les officiers supérieurs) fait office de *Trésorier* de la Société.

De temps à autre, un officier (autre que le Président) peut *changer d'allégeance*, de sorte qu'il devient désormais le disciple d'un mentor différent de celui qu'il avait auparavant (à condition que son nouveau mentor ne soit pas l'un de ses disciples, ou l'un des disciples de ses disciples, etc.). De ce fait, il peut arriver que l'influence de certains officiers change et que le rôle de Trésorier échoie à un officier différent de celui d'avant.

## Tâche

Écrivez un programme qui lit la description de l'état initial de la Société et une suite de changements d'allégeance. Votre programme doit afficher qui est le Trésorier dans l'état initial de la Société ainsi qu'après chaque changement d'allégeance.

## Entrée

La première ligne contient deux entiers,  $n$  et  $q$ , séparés par un espace ;  $n$  est le nombre de sections,  $q$  est le nombre de changements d'allégeance.

Les  $n$  lignes suivantes décrivent l'état initial de la Société. La  $i$ -ème de ces lignes contient deux entiers,  $s_i$  et  $m_i$ , séparés par un espace ;  $s_i$  est le mentor de l'officier  $i$  (c'est-à-dire de l'officier dirigeant la section  $i$ ), tandis que  $m_i$  est le nombre de membres de la section  $i$ . La valeur  $s_i = 0$  indique que l'officier  $i$  est le Président de la Société et n'a donc pas de mentor.

Les  $q$  lignes restantes décrivent les changements d'allégeance. La  $j$ -ème de ces lignes contient deux entiers,  $\hat{x}_j$  et  $\hat{z}_j$ , séparés par un espace. La signification de ces entiers est

la suivante. Notons  $t_j$  (pour  $j = 0, \dots, q$ ) le Trésorier après les  $j$  premiers changements d'allégeance (ainsi  $t_0$  est le Trésorier initial avant le premier changement d'allégeance). Alors le  $j$ -ème changement d'allégeance consiste à faire de l'officier  $z_j$  le nouveau mentor de l'officier  $x_j$ , où  $x_j = 1 + ((t_{j-1} + \hat{x}_j) \bmod n)$  et  $z_j = 1 + ((t_{j-1} + \hat{z}_j) \bmod n)$ . Le but de cette représentation des valeurs  $x_j$  et  $z_j$  est de forcer votre programme à traiter les changements d'allégeance dans l'ordre où ils apparaissent.

Les changements d'allégeance dans les données d'entrée seront toujours valides, c'est-à-dire que  $z_j$  ne sera pas égal à  $x_j$ , et  $z_j$  ne sera pas non plus un disciple de  $x_j$ , un disciple d'un disciple de  $x_j$ , etc. Il est toutefois possible que  $z_j$  ait déjà été le mentor de  $x_j$  juste avant le  $j$ -ème changement (de sorte que rien ne change réellement à ce moment-là).

Notez que si votre programme calcule un résultat  $t_j$  erroné à un moment donné, il décodera également de manière incorrecte les entrées suivantes  $\hat{x}_{j+1}, \hat{z}_{j+1}$ , etc., et pourrait planter avec un verdict RTE (erreur d'exécution) au lieu d'un verdict WA (mauvaise réponse), car les entrées mal décodées pourraient être invalides (par exemple, il pourrait obtenir par erreur un  $z_{j+1}$  qui est un disciple de  $x_{j+1}$ ).

### Contraintes

- $1 \leq n \leq 1\,000\,000$
- $1 \leq q \leq 30\,000$
- $1 \leq m_i$  pour chaque  $i = 1, \dots, n$
- $m_1 + m_2 + \dots + m_n \leq 10^9$
- $1 \leq \hat{x}_j \leq n$  et  $1 \leq \hat{z}_j \leq n$  pour chaque  $j = 1, \dots, q$ .

### Sous-tâches

- Sous-tâche 1 (15 points) :  $n \leq 100$
- Sous-tâche 2 (10 points) :  $n \leq 1000$
- Sous-tâche 3 (50 points) :  $n \leq 300\,000$
- Sous-tâche 4 (25 points) : Aucune contrainte supplémentaire.

### Sortie

Affichez les nombres  $t_0, t_1, \dots, t_q$ , chacun sur sa propre ligne, où  $t_j$  est le Trésorier après les  $j$  premiers changements d'allégeance. Naturellement, chaque  $t_j$  doit être un entier de l'intervalle  $1 \leq t_j \leq n$ .

### Exemple

Entrée	Sortie
7 2	2
0 1	2
1 3	3
1 3	
2 3	
2 1	
5 2	
5 1	
3 7	
2 7	

### Commentaire

Initialement, l'officier 2 est le Trésorier (donc  $t_0 = 2$ ). Lors du premier changement d'allégeance, nous lisons  $\hat{x}_1 = 3$  et  $\hat{z}_1 = 7$  et calculons  $x_1 = 1 + ((2+3) \bmod 7) = 6$  et  $z_1 = 1 + ((2+7) \bmod 7) = 3$  ; ainsi, l'officier 3 devient le nouveau mentor de l'officier 6 ; l'officier 2 reste Trésorier (donc  $t_1 = 2$ ). Lors du deuxième changement d'allégeance, nous lisons  $\hat{x}_2 = 2$  et  $\hat{z}_2 = 7$  et calculons  $x_2 = 1 + ((2+2) \bmod 7) = 5$  et  $z_2 = 1 + ((2+7) \bmod 7) = 3$  ; ainsi, l'officier 3 devient le nouveau mentor de l'officier 5 et devient également le nouveau Trésorier (donc  $t_2 = 3$ ).



## DFS

*Limite de temps: 1 s    Limite de mémoire: 256 MiB*

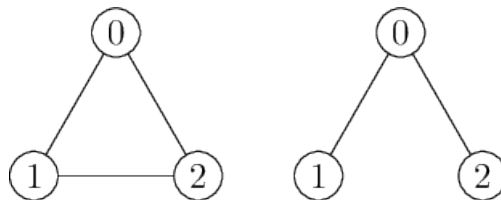
Vous connaissez peut-être déjà le célèbre algorithme DFS (parcours en profondeur) permettant de parcourir un graphe. Dans ce problème, nous ne considérerons que des graphes simples non orientés connexes (sans boucles ni arêtes parallèles) dont les sommets sont numérotés  $0, 1, \dots, n - 1$ , et l'algorithme DFS affichera des profondeurs et des sommets de la manière suivante :

```
DFS(d, v):
    output d/v
    mark vertex v as visited
    W = the list of neighbors of v ordered by increasing numbers
    for each w in W:
        if vertex w has not yet been visited:
            DFS(d + 1, w)
```

Écrivez un programme qui affiche le nombre de graphes différents pour lesquels l'appel  $\text{DFS}(0, n - 1)$  produit le même affichage que celui donné en entrée. Par exemple, l'affichage

```
0/2
1/0
2/1
```

est produit en appelant  $\text{DFS}(0, 2)$  sur l'un ou l'autre des deux graphes simples non orientés connexes à 3 sommets suivants :



### Entrée

L'entrée est l'affichage de l'appel  $\text{DFS}(0, n - 1)$  sur un graphe simple non orienté connexe inconnu à  $n$  sommets. L'entrée est donc constituée de  $n$  lignes au format  $d/v$ , la première ligne étant  $0/n-1$ .

### Contraintes

- $1 \leq n \leq 2 \cdot 10^5$

### Sous-tâches

- Sous-tâche 1 (10 points) :  $n \leq 6$ .
- Sous-tâche 2 (20 points) :  $n \leq 500$ .

- Sous-tâche 3 (20 points) :  $n \leq 10^4$ .
- Sous-tâche 4 (10 points) : Pour chaque  $i \in \{2, \dots, n\}$ , la  $i$ -ème ligne d'entrée est  $i-1/i-2$ .
- Sous-tâche 5 (20 points) : Pour chaque  $i \in \{2, \dots, n\}$ , la  $i$ -ème ligne d'entrée est  $i-1/v$  pour un certain  $v \in \{0, \dots, n-2\}$ .
- Sous-tâche 6 (20 points) : Aucune contrainte supplémentaire.

## Sortie

Affichez le nombre de graphes différents ayant la propriété requise. Comme ce nombre peut être très grand, affichez le résultat modulo 1 000 000 007.

## Exemple

Entrée	Sortie
0/2	2
1/0	
2/1	

# Chasse au trésor

*Limite de temps: 8 s      Limite de mémoire: 256 MiB*

Vous recherchez un trésor perdu depuis longtemps sur un immense champ de cases  $N \times N$  à l'aide d'une boussole magique. Il y a au total  $K \leq 3$  coffres au trésor cachés dans diverses cases distinctes du champ. Votre but est simple : les trouver tous !

Lorsque vous posez la boussole sur l'une des cases, elle trouve les plus courts chemins vers un coffre au trésor en utilisant uniquement des déplacements vers la gauche, le haut, la droite et le bas (c'est-à-dire un coffre au trésor le plus proche selon la distance de Manhattan). Elle indique ensuite la direction du premier déplacement. Si plusieurs premiers déplacements valides mènent à un coffre au trésor le plus proche (ou à plusieurs d'entre eux), la boussole les retourne tous. Si la case contient un trésor, la boussole l'indique à la place.

Chaque fois que vous trouvez un coffre au trésor, vous en videz le contenu, mais vous ne pouvez pas retirer le coffre – il est trop lourd. Votre boussole ne sait pas si un coffre est plein ou vide. Elle indique le chemin vers un coffre le plus proche, qu'il soit vide ou plein.

Essayez de trouver l'emplacement de tous les coffres au trésor en un minimum de requêtes de la boussole.

## Tâche

Ceci est une tâche interactive. Dans chaque cas de test (c'est-à-dire chaque exécution de votre programme), votre programme devra résoudre plusieurs chasses au trésor. Vous devez interagir avec le grader en utilisant une bibliothèque fournie par les organisateurs. Cette bibliothèque contient les déclarations suivantes :

- **void** *NextHunt*(**int** &*N*, **int** &*K*) — appelez cette fonction pour démarrer la prochaine chasse au trésor. Elle place la taille de la grille dans la variable *N* et le nombre de trésors dans *K*. S'il n'y a plus de chasses au trésor à résoudre dans l'exécution en cours, la fonction fixe *N* et *K* à  $-1$  ; dans ce cas, vous devez alors terminer votre programme avec le code de sortie 0. Notez que vous pouvez appeler cette fonction avant d'avoir trouvé tous les trésors de la chasse en cours, par exemple si vous essayez de ne résoudre que pour des points partiels.
- **enum** { *TREASURE* = 0, *DIR\_RIGHT* = 1, *DIR\_UP* = 2, *DIR\_LEFT* = 4, *DIR\_DOWN* = 8 }; — ce sont les constantes utilisées dans les valeurs de retour de la fonction *Query* (voir ci-dessous).
- **int** *Query*(**int** *x*, **int** *y*) — si la case aux coordonnées (*x*, *y*) contient un trésor, cette fonction retourne *TREASURE*. Sinon, elle retourne la somme d'une ou plusieurs des constantes *DIR\_RIGHT*, *DIR\_UP*, *DIR\_LEFT* et *DIR\_DOWN*, indiquant quels déplacements la boussole retourne lorsqu'elle est posée sur la case (*x*, *y*). Les coordonnées *x* et *y* doivent être des entiers de 0 à  $N - 1$ . (Remarque : dans cette tâche, les coordonnées *y* augmentent du haut vers le bas.)

Après que *NextHunt* a retourné  $N = K = -1$ , votre programme ne doit plus appeler *NextHunt* ni *Query*. Il ne doit pas non plus appeler *Query* avant le premier appel à

*NextHunt*. Si votre programme ne respecte pas ces contraintes, ou s'il effectue plus de 1000 requêtes au sein d'une même chasse au trésor, ou s'il fournit des valeurs de  $x$  et/ou  $y$  hors de la plage autorisée lors de l'appel à *Query*, la bibliothèque terminera votre programme et retournera un run-time-error (RTE) pour le cas de test en cours.

Un trésor est considéré comme trouvé si vous avez interrogé au moins une fois la case qui le contient. Si votre programme appelle *NextHunt* avant d'avoir trouvé tous les trésors de la chasse en cours, cela n'est pas traité comme une erreur, mais cela affectera votre score (plus de détails sur le calcul du score ci-dessous).

Pour accéder à la bibliothèque, votre programme doit inclure le fichier d'en-tête `treasurehuntlib.h` :

```
#include "treasurehuntlib.h"
```

Vous pouvez télécharger ce fichier d'en-tête ici : `treasurehuntlib.h`.

Pour vous aider à développer votre solution, une implémentation simple de la bibliothèque est disponible ici : `treasurehuntlib-public.cpp`. Pour la compiler avec votre programme, ajoutez simplement son nom comme paramètre au compilateur, par exemple :

```
g++ foo.cpp treasurehuntlib-public.cpp
```

si `foo.cpp` est le nom du fichier contenant votre solution.

L'implémentation dans `treasurehuntlib-public.cpp` prend en charge, en plus des déclarations ci-dessus, une autre fonction appelée `void InitFromFile(const char *fileName)`, qui lit une liste de chasses au trésor depuis un fichier et vous permet de jouer avec celles-ci plutôt qu'avec ses propres chasses au trésor générées aléatoirement. Vous trouverez plus de détails à l'intérieur de `treasurehuntlib-public.cpp` lui-même.

Sur le serveur d'évaluation, une implémentation différente de la bibliothèque sera utilisée, vous ne devez donc faire aucune supposition sur le fonctionnement exact de l'implémentation. Vous pouvez toutefois supposer qu'elle ne pollue pas l'espace de noms global avec d'autres déclarations que celles listées (*NextHunt*, *Query* et les cinq constantes).

Votre code ne doit pas lire depuis l'entrée standard ni écrire sur la sortie standard, car celles-ci seront utilisées par notre implémentation de la bibliothèque sur le serveur d'évaluation pour communiquer avec le reste de l'environnement d'évaluation.

## Entrée

### Contraintes

- $2 \leq N \leq 10^6$
- $1 \leq K \leq 3$
- Au sein d'une même exécution de votre programme, il y aura au plus 100 000 chasses au trésor.
- Au sein d'une même chasse au trésor, vous pouvez effectuer au plus 1000 requêtes.
- Le système de correction n'est pas adaptatif.

### Sous-tâches

- Sous-tâche 1 (10 points)  $K = 1$
- Sous-tâche 2 (30 points)  $K = 2$
- Sous-tâche 3 (60 points)  $K = 3$ .

### Calcul du score

Une sous-tâche peut être constituée de plusieurs cas de test (plusieurs exécutions de votre programme), et chaque cas de test peut être constitué de plusieurs chasses au trésor. Pour le calcul du score, toutes les chasses au trésor d'une sous-tâche donnée sont évaluées ensemble, indépendamment de la manière dont elles ont été initialement réparties entre les cas de test. Pour la  $i$ -ème chasse au trésor, notons la taille de la grille par  $N_i \times N_i$ , le nombre de trésors par  $K_i$ , le nombre de requêtes effectuées par votre programme par  $Q_i$  et le nombre de trésors trouvés par votre programme par  $F_i$ . Notons de plus par  $S$  le nombre total de points disponibles pour cette sous-tâche. Alors le nombre de points attribués à votre programme pour cette sous-tâche est :

- Si votre programme a toujours trouvé tous les trésors (c'est-à-dire si  $F_i = K_i$  pour tout  $i$ ), son score dépend de  $t_i = \frac{Q_i}{\lceil \log_2 N_i \rceil}$  :

$$\frac{S}{2} + \frac{S}{2} \cdot \min_i f(t_i) \text{ points, où } f(t_i) = \begin{cases} 1, & t_i \leq 11 \\ 1 - (t_i - 11)/9, & 11 \leq t_i \leq 20 \\ 0, & t_i \geq 20. \end{cases}$$

- Si votre programme n'a pas toujours trouvé tous les trésors (c'est-à-dire s'il existe un  $i$  tel que  $F_i < K_i$ ), il reçoit

$$\frac{S}{2} \cdot \min_i \frac{F_i}{K_i} \text{ points.}$$

Autrement dit, vous obtenez la moitié des points pour avoir trouvé tous les trésors, et l'autre moitié pour les avoir trouvés en un minimum de requêtes. Pour un score parfait, votre solution devrait trouver tous les trésors en au plus  $11 \lceil \log_2 N_i \rceil$  requêtes. Entre  $11 \lceil \log_2 N_i \rceil$  et  $20 \lceil \log_2 N_i \rceil$  requêtes, le score décroît linéairement ; et si votre solution effectue plus de  $20 \lceil \log_2 N_i \rceil$  requêtes, elle n'obtiendra que la première moitié des points, pour avoir trouvé tous les trésors. (Les symboles  $\lceil \cdot \rceil$  signifient que la valeur de  $\log_2 N_i$  est arrondie à l'entier supérieur.)

Si les formules ci-dessus donnent un nombre non entier de points pour la sous-tâche, il sera arrondi à l'entier le plus proche.

Si votre programme a une erreur d'exécution ou ne respecte pas le protocole d'utilisation de la bibliothèque mentionné ci-dessus, il obtiendra 0 point pour toute la sous-tâche. Pour recevoir des points partiels pour la découverte d'un sous-ensemble des trésors, votre programme doit donc terminer la recherche proprement en appelant *NextHunt*.

**Exemple**

Appel	Valeur de retour
NextHunt(N, K)	$N = 4, K = 1$
Query(2, 0)	$DIR\_DOWN + DIR\_RIGHT = 9$
Query(3, 1)	$DIR\_DOWN$
Query(3, 2)	$TREASURE$
NextHunt(N, K)	$N = -1, K = -1$