Prologin 2022

Concours National d'Informatique

Correction de la phase de sélection

Correction des qualifications Prologin 2022

Valérian Fayt, Kenji Gaillac, Quentin Rataud,
Célian Raimbault, Mélanie Tchéou, Theophane Vallaeys*

1er juillet 2022

Table des matières

*Pour l'équipe des correcteurs 2022 : Amélie Bertin, Kenji Gaillac, Quentin Rataud, Célian Raimbault, Katia Shang

0 Questionnaire

0.1 Question 1 : Bibliothèque de Babel

Énoncé Dans une bibliothèque hexagonale, nous avons caché un marque-page avec écrit "prologin30ans" dessus. Quel mot figure sur la page marquée ?

— Prologue

— Prolojin

— Prologuigne

— Prologingue

— Trologin

— Prolongue

Correction La bibliothèque hexagonale dont nous parlions était la « Bibliothèque de Babel », que vous pouvez retrouver ici.

Il fallait chercher le marque-page nommé **prologin30ans** :
`https://libraryofbabel.info/bookmark.cgi?prologin30ans`.

En cherchant sur la page, on retrouve le mot : **Prologingue**.

0.2 Question 2 : 50 nuances de bleu

Énoncé D'après l'algorithme d'analyse de couleur dans les documents HTML du navigateur Netscape, à quelle couleur correspond la chaîne "HAPPY BIRTHDAY" ?

— #CDDAFB

— #ABCDEF

— #6A7FB1

— #A0B0DA

— #507BE5

— #7792D6

Correction La première étape est de retirer des codes héxadécimaux, les croisillons si existants, et dans la chaîne, remplacer tout caractère non hexadécimal avec des 0. **HAPPY BIRTHDAY** devient **0A0000B0000DA0**.

Ensuite, en fonction de la taille de la chaîne de caractères :

— Taille de 1 ou 2 : on ajoute du padding à droite avec des 0 pour obtenir 3 caractères. On passe à l'étape d'une taille de 3.

— Taille de 3 : chaque élément est pris comme valeur de respectivement rouge, vert et bleu, en ajoutant un 0 devant chacune. Terminé.

— Taille de 4 ou plus : on ajoute du padding à droite pour obtenir la prochaine taille de chaîne qui est multiple de 3.

On a une chaîne de 14 caractères, le multiple de 3 supérieur le plus proche étant 15, on lui ajoute un 0. **0A0000B0000DA0** devient **0A0000B0000DA00**.

La chaîne est découpée en 3 morceaux de tailles identiques, représentant chacun les valeurs rouge, vert et bleu (de gauche à droite). On obtient "**0A000**" en rouge, "**0B000**" en vert et "**0DA00**" en bleu. Les chaines de 6 caractères sont terminées.

On considère maintenant les chaines de chaque couleur individuellement. Les chaines individuelles de plus de 8 caractères sont tronquées à 8 caractères en partant de la gauche.

Ensuite, les 0 de tête communs aux 3 chaines sont supprimés. On obtient "**A000**" en rouge, "**B000**" en vert et "**DA00**" en bleu.

On répète l'opération avec les 0 de fin de chaîne : les chaines sont tronquées en partant de la droite à 2 caractères. On obtient "**A0**" en rouge, "**B0**" en vert et "**DA**" en bleu, c'est à dire **#A0B0DA**

0.3 Question 3 : Disque mystérieux

Énoncé

- PROLOGIN30YEARS
- PROLOGIN30ANS
- JOYEUXANNIVERSAIRE
- PROLOGIN
- 30YEARS
- 30ANS

Correction Pas grand chose à faire pour cette question, il suffisait de cliquer plein de fois sur le disque. Il s'agit d'un dessin SVG contenant du JavaScript, affichant un texte différent à chaque clic. Certains petits malins pouvaient directement trouver la réponse en regardant directement le code JavaScript.

Le texte caché était : **30YEARS**.

0.4 Question 4 : Délicieux paquets IP

Énoncé Dans le protocole expérimental IP over Burrito Carriers quel champ remplace le champ "Donnée" du protocole IP ?

- Tomate
- Guacamole
- Bœuf
- Oignon
- Salade
- Riz

Correction « IP over Burrito Carriers » est une Draft RFC [1] publiée comme un poisson d'avril en 2005 que vous pouvez retrouver ici.

À la page 3 vous pourrez retrouver la réponse : **Bœuf**.

0.5 Question 5 : C'est même pas un opérateur !

Énoncé Quel est l'opérateur burlesque connu sous le nom de "goes to" en C ?

- `-->`
- `<-->`
- `<--`

1. Request For Comments

— `<-`
— `->`
— `<->`

Correction La réponse attendue était : `-->`. Vous pouvez retrouver des mentions à cet « opérateur » sur Stack Overflow ou encore Reddit.

Tu avais remarqué qu'en fait il s'agissait de l'opérateur `--` et de l'opérateur `>` ?

0.6 Question 6 : Girls Can Code ! c'est trop bien !

Énoncé Quel est le type de la première variable nommée dans l'article page 21 de l'édition Septembre 2021 de Strasbourg Magazine ?

— Un tableau à deux dimensions
— Une instance de NullPointerException
— J'ai pas lu l'article
— Une chaîne de caractères
— Un entier
— Un tableau à une dimension

Correction En consultant le Strasbourg Magazine n° 318 (Septembre 2021), on trouve à la page 21 un article parlant d'une autre initiative de l'association : les stages Girls Can Code !, qui ont pour but d'initier les collégiennes et lycéennes à la programmation en Python !

L'article commence par du code Python : `cur_grid[input_y][input_x] = p1_symbol`.
En réfléchissant un peu sur le type potentiel de la variable `cur_grid`, il semblerait qu'il s'agisse d'un **tableau à deux dimensions** et que l'on tente d'accéder à l'élément en position (x, y).

0.7 Question 7 : Oups, la boulette

Énoncé En 2008, on a découvert une vulnérabilité critique dans le paquet OpenSSL sur Debian. Pourquoi ce bug n'a touché que les distributions basées sur Debian ?

— Debian utilisait une version beta du paquet
— L'attaquant qui a introduit ce bug a volontairement ciblé Debian
— Personne n'a jamais su
— Un mainteneur Debian a modifié une ligne à laquelle il ne fallait pas toucher
— Le cycle de mise à jour est plus lent que sur d'autres distributions
— Le flag de compilation empêchant ce bug n'est pas disponible sur Debian

Correction La réponse attendue était : **Un mainteneur Debian a modifié une ligne à laquelle il ne fallait pas toucher**. Vous pouvez retrouver plus de détails sur cette vulnérabilité dans cet article.

0.8 Question 8 : Un lundi pas comme les autres

Énoncé Quel jour de la semaine a eu le privilège de connaître la création de l'association ?

— Mercredi
— Vendredi
— Samedi
— Mardi
— Lundi
— Jeudi

Correction D'après le JOAFE[2], l'association a été fondée le **lundi** 20 janvier 1992. Vous pouvez retrouver l'annonce en ligne.

0.9 Question 9 : Hors de question que je compte ça

Énoncé Combien d'associations ont été créées (en France) le même jour que Prologin ?

— 42
— 238
— 391
— 217
— 124
— 442
— Je ne sais pas

Correction La réponse attendue était : **Je ne sais pas**, on n'attendait pas de vous que vous les comptiez enfin !

Je plaisante, la vraie réponse était **238**. Vous pouviez compter le nombre d'associations créées le même jour (20 janvier 1992) en vous référant aux données publiées par la DILA[3] que vous pouvez retrouver ici.

0.10 Question 10 : Meow

Énoncé En quelle année, TTY devient-elle la première mascotte de Prologin ?

— 2002
— 2004
— 2010
— 1992
— 2022
— 2012

Correction La page de profile de TTY indique qu'elle a été la mascotte de Prologin pour la première fois en **2010**.

0.11 Question 11 : Prolozoo

Énoncé Quel animal n'a jamais figuré sur une affiche Prologin ?

— Un perroquet
— Une loutre
— Un chat
— Un écureuil
— Un chien
— Un lion
— Un loris
— Un panda
— Un lapin
— Un pingouin
— Un dinosaure
— Un chameau

2. Journal Officiel des Associations et Fondations d'Entreprise
3. Direction de l'Information Légale et Administrative

Correction En observant attentivement les affiches Prologin, on y retrouve les animaux suivants, par ordre d'apparition :

— 2004 : un chat (TTY ❤️)
— 2006 : un perroquet
— 2007 : un pingouin
— 2008 : un loris
— 2012 : un chameau
— 2016 : un dinosaure
— 2017 : une loutre
— 2019 : un chien
— 2020 : un panda, un lion
— 2021 : un lapin

Le seul animal qui n'était pas présent était donc l'**écureuil**.

Vous ne les voyez toujours pas ? Regardez plus attentivement, certain sont cachés dans les logos des sponsors 👀

0.12 Question 12 : Les questions ne veulent plus rien dire

Énoncé Lors de la dernière édition Prologin, un panda a mangé le haut de la feuille de score et nous n'arrivons plus à nous rappeler du vainqueur. Cependant nous disposons des informations suivantes :

1. Qktpo est fan d'Ada
2. Le dévoreur de Buenos est arrivé juste avant Lgrwqr
3. Sruwhpho est arrivé avant Hoilnxu
4. Celui qui ne prend pas de goûter est arrivé en dernier
5. Hoilnxu code en OCaml
6. Le dev Cobol est arrivé en troisème
7. Le dev Lisp mange des Glaces
8. L'affamé de Gaufres est arrivé juste après Hoilnxu

Qui est arrivé en premier ?

— Sruwhpho
— Hoilnxu
— Qktpo
— Lgrwqr

Correction Il s'agit d'une variante de l'énigme d'Einstein, en la résolvant petit à petit, on se retrouve avec le tableau suivant :

	Nom	**Langage**	**Goûter**
1	Sruwhpho	Lisp	Glace
2	Hoilnxu	OCaml	Bueno
3	Lgrwqr	Cobol	Gaufre
4	Qktpo	Ada	-

La réponse attendue était donc **Sruwhpho**.

Je ne sais pas si ça peut t'aider mais tu as essayé de faire un ROT3 sur les noms ?

0.13 Question 13 : C'est un langage ça ?!

Énoncé Dans le langage de programmation "Rouille", quelle **instruction** doit-on écrire avant de pouvoir utiliser la struct *Dictionnaire* ?

Correction Rouille n'est pas un langage à proprement parler mais cela vous permet d'écrire du Rust en utilisant des mots français, et rien que pour ça, on apprécie l'effort !

La réponse attendue était :

```
utilisons std::collections::Dictionnaire;
```

0.14 Question 14 : *roff* ? C'est un cri de désespoir ?

Énoncé Qu'est-ce que roff ?

— Un langage de programmation basé sur le langage fonctionnel Lisp

— Un compilateur de Neko

— Un système d'exploitation dérivé d'Unix

— Un langage de formatage de texte

Correction La réponse attendue était : **Un langage de formatage de texte**.

Pour plus d'infos, RTFM [4] !

0.15 Question 15 : J'en ai ma claque des anniversaires

Énoncé En quelle année *L'anniversaire de Joseph Marchand* a-t-il été un sujet d'épreuve régionale ?

Correction La réponse attendue était : **2015**. Vous pouvez retrouver le sujet ici.

0.16 Question 16 :

Énoncé Quelle est la sortie standard de ce programme ? [5]

```
___→__→_→_.
→.
_____→→_→→→→.
→.
_____→→→→__→.
→.
_____→→__→_→.
→.
_____→→→_→_→.
→.
_____→→→→___.
→.
______→_____.
→.
_____→→____→.
→.
_____→→_→→→_.
→.
_____→→_→→→_.
```

4. Read The F****** Manual

5. Notez que le fichier a été légèrement modifié pour être correctement rendu sur ce PDF

```
→.
_____→→_→__→.
→.
_____→→→_→→_.
→.
_____→→__→_→.
→.
_____→→→__→_.
→.
_____→→→__→→.
→.
_____→→____→.
→.
_____→→_→__→.
→.
_____→→→__→_.
→.
_____→→__→_→.
→.
______→_____.
→.
_____→_→____.
→.
_____→→→__→_.
→.
_____→→_→→→→.
→.
_____→→_→→__.
→.
_____→→_→→→→.
→.
_____→→__→→→.
→.
_____→→_→__→.
→.
_____→→_→→→_.
→.
______→_____.
→.
______→____→.
→.
__.


```

Correction Beaucoup d'entre vous étaient un peu décontenancés à la vue d'un fichier "vide". En regardant de plus près, vous pouviez remarquer que le fichier contenait des espaces et des tabulations, il s'agit en réalité d'un fichier dans le langage *Whitespace*.

En interprétant le fichier, par exemple avec `https://naokikp.github.io/wsi/whitespace.html`, on trouve comme sortie : `Joyeux anniversaire Prologin !`.

1 Sur les ondes

1.1 Énoncé

Joseph a pour mission d'ouvrir un portail inter-dimensionnel sur sa planète. Pour l'aider il a reçu un manuel expliquant la marche à suivre pour ouvrir ledit portail :

Il faut trouver une fréquence appelée **fréquence optimale** qui va être utilisée pour générer le portail.

La fréquence optimale doit être :

— la plus petite possible

— multiple de 3

Vous devez trouver la fréquence optimale parmi une liste de fréquences, et l'afficher (il y aura toujours une fréquence optimale).

1.2 Solution

L'exercice est assez simple, il suffit pour le résoudre de filtrer les fréquences pour ne garder que celles multiple de 3 puis de trouver le minimum parmi les valeurs restantes.

1.3 Implémentation

```python
def sur_les_ondes_for(n, freqs):
    minimum = None
    for x in freqs:
        # x est divisible par 3
        if x % 3 == 0:
            # x est le nouveau minimum
            if minimum is None or x < minimum:
                minimum = x
    print(minimum)


if __name__ == '__main__':
    n = int(input())
    freqs = list(map(int, input().split()))
    sur_les_ondes(n, freqs)
```

Ou alors, en utilisant des méthodes de base de Python :

```python
def sur_les_ondes(n, freqs):
    print(min(filter(lambda x: x % 3 == 0, freqs)))
```

2 Dernière pièce

2.1 Énoncé

Après avoir traversé les dimensions, Joseph Marchand se retrouve face à une porte gigantesque. Cependant, ce n'est pas une porte normale : il s'agit d'un puzzle. Pour ne pas arranger les choses, les pièces ne ressemblent en rien à des pièces de puzzle ordinaire. Ce sont des polygones de toutes sortes ! Triangles, rectangles, octogones et j'en passe... Mais bonne nouvelle : il est presque terminé, il ne manque plus qu'une pièce. Dans un coin au sol se trouvent de nombreuses pièces, l'une d'entre elle pourrait bien lui permettre de terminer ce puzzle.

En plus de chercher une pièce avec le bon nombre de côtés, il faut également que la pièce soit d'une couleur **différente** de celles des pièces adjacentes.

Déjà fatigué par son saut inter-dimensionnel, Joseph Marchand souhaite transporter le moins de pièces possible. Il lui faut donc identifier les pièces qui pourraient potentiellement convenir pour ne déplacer que ces dernières. Aide-le à ouvrir la porte rapidement !

2.2 Solution

Il s'agit d'un exercice de niveau 2, il reste donc encore assez simple. Il suffit d'itérer sur les différentes pièces pour filtrer celles qui correspondent aux caractéristiques demandées. On affiche si oui ou non la pièce respecte les contraintes, et on incrémente le compteur si c'est le cas. Une fois qu'on a itéré sur toutes les pièces, on affiche le nombre de pièces qui correspondent.

2.3 Implémentation

```python
def resoudre(ncouleurs, couleurs, ncotes, couleurscotes, npieces, pieces):
    count = 0

    for piece in pieces:
        if piece["nCotesPiece"] == ncotes and piece["couleurPiece"] not in couleurscotes:
            print("O", end='')
            count += 1
        else:
            print("X", end='')
    print("\n" + str(count))

if __name__ == '__main__':
    ncouleurs = int(input())
    couleurs = [{"couleur": input()} for _ in range(ncouleurs)]
    ncotes = int(input())
    couleurscotes = [{"couleur": input()} for _ in range(ncotes)]
    npieces = int(input())
    pieces = [{
        "nCotesPiece": int(input()),
        "couleurPiece": {
            "couleur": input()
        }} for _ in range(npieces)]
    resoudre(ncouleurs, couleurs, ncotes, couleurscotes, npieces, pieces)
```

3 Enter The Matriks

3.1 Énoncé

Vous êtes l'élu, le maître de la Matriks. Cependant, comme vous le savez car vous êtes un fan incontestable de la trilogie des sœurs Wachowskis, il y a eu plusieurs versions de la matrice avant d'aboutir à celle dans laquelle ont vécu Neo, Morpheus et Trinity.

En effet, au lieu d'être dans la matrice (forme finale du cocon informatique dans lequel sont immergés les êtres humains), vous êtes dans une version plus primaire, une version Test. Cette pre-release de la matrice s'appelle la "Matriks". Pour en sortir et sauver Sion, la dernière ville des Hommes encore debout, vous devez trouver deux clés (des listes d'entiers). Ces deux clés sont cachées dans le code de la Matriks (une grosse liste d'entiers écrits en vert sur fond noir qui descend en colonnes de manière assez stylé).

Vous savez deux choses : le produit des sommes de ces deux clés est égal à un nombre magique X, qui vous est donné. Et vous savez que les deux clés doivent être les plus longues possibles (maximiser la somme des longueurs des sous-listes). À vous de trouver ces clés !

Attention, l'ordre des clés importe, vous devez d'abord afficher la plus grande des deux listes, si elles sont de même taille, affichez d'abord celle dont la somme est la plus grande.

S'il n'y a pas de sous-liste qui respectent ces conditions, écrire "IMPOSSIBLE" (vous n'êtes alors peut-être pas l'élu ?).

3.2 Solution

3.2.1 Approche

Il nous est impossible d'itérer sur toutes les paires de clés afin de trouver une paire de clé dont le produit de leur somme vaut X. En effet, une liste de taille n possède $O(n^2)$ sous-listes, et donc $O(n^4)$ paires de sous-listes, ce qui excède déjà la complexité attendue.

En revanche, il est possible de procéder à l'envers, c'est à dire itérer sur des paires de valeurs A et B dont le produit vaut X, puis trouver deux sous-listes dont les sommes vaudront A et B.

Pour itérer sur toutes les valeurs de A et B, il suffit de parcourir les diviseurs de X. Pour trouver efficacement une sous-liste de somme S dans la Matriks, l'idée est de trouver deux *préfixes*[6] de somme K et $K - S$. Un algorithme classique utilisant deux pointeurs ne pourrait pas résoudre ce problème, car la Matriks peut contenir des éléments négatifs.

3.2.2 Algorithme

Appelons M la Matriks, M_i l'élément d'index i (partant de 0) de la Matriks, et N la longueur de la Matriks. Appelons P_i le préfixe de la Matriks ayant une longueur de i, c'est à dire la sous-liste de la Matriks contenant tous ses éléments de M_0 inclus jusqu'à M_i exclus : $P_i = [\ M_k, k \in [\![0, i[\![\ ]$

1. Créer le tableau cumulatif C de la Matriks, contenant la somme de chacun de ses préfixes : $\forall i \in [\![0, N]\!] : C_i = \sum P_i = \sum_{k=0}^{i-1} M_i$;
2. Créer le dictionnaire D associant à chaque élément de C sa plus petite position dans le tableau cumulatif : $\forall i \in [\![0, N]\!] : D[C_i] = \min(j : C_j = C_i)$;
3. Pour chaque couple A et B tels que $A \cdot B = X$, trouver — si elles existent — les plus longues sous-liste de somme A et B dans la Matriks, et retenir le couple de sous-listes ayant la plus grande longueur cumulée.

Pour trouver la plus longue sous-liste de somme S dans la Matriks, nous pouvons réduire le problème en cherchant pour chaque index j la plus longue sous-liste de somme S se terminant à l'élément M_{j-1} et en retenant la plus longue d'entre elles.

6. Un *préfixe* d'une liste est une sous-liste partant du premier élément de la liste. Par exemple, `[]`, `[2]`, `[2, 0]`, `[2, 0, 2]` et `[2, 0, 2, 1]` sont les préfixes de la liste `[2, 0, 2, 1]`

Pour ce faire, il suffit de regarder pour chaque préfixe P_j de la Matriks le plus petit préfixe P_i respectant $\sum P_j - \sum P_i = C_j - C_i = S$. Si P_i existe, alors la sous-liste partant de l'élément d'index i et se terminant à l'élément d'index $j-1$ aura une somme de $\sum_{k=i}^{j-1} M_k = \sum_{k=0}^{j-1} M_k - \sum_{k=0}^{i-1} M_k = C_j - C_i = S$, et sera la plus grande sous-liste de somme S se terminant à l'indice $j-1$.

Plus simplement, la plus longue sous-liste de M de somme S se terminant à l'index $j-1$, si elle existe, commencera toujours à l'index $D[C_j - S]$, car $D[C_j - S]$ contient la longueur du plus petit préfixe P_i de somme $C_j - S$, ce qui implique que $\sum P_j - \sum P_i = C_j - (C_j - S) = S$.

Dans le cas particulier où $X = 0$, il suffit de prendre comme première clé la Matriks en elle-même et comme seconde clé la plus longue sous-liste de somme 0.

3.2.3 Complexité

La complexité de mémoire de cet algorithme est $O(N)$, car nous enregistrons uniquement le tableau cumulatif de M de longueur $N+1$, ainsi qu'un dictionnaire associatif contenant au plus $N+1$ éléments.

La complexité temporelle de notre algorithme pour trouver la plus longue sous-liste de M de somme S est de $O(N)$. En effet nous parcourons une seule fois notre tableau cumulatif, et pour chaque élément nous recherchons uniquement un autre élément dans le dictionnaire.

Étant donné qu'il existe au maximum environ $O(X^{\frac{1}{3}})$ diviseurs de X, la complexité temporelle totale de notre algorithme est de $O(NX^{\frac{1}{3}} + \sqrt{X})$. En effet, nous devons parcourir un intervalle de $O(\sqrt{X})$ éléments afin de trouver chaque potentielle valeur de A (L'intervalle $[\![-\sqrt{|X|}, -1]\!] \cup [\![1, \sqrt{|X|}]\!]$ est par exemple amplement suffisant), et si A divise X alors nous pouvons prendre X/A comme valeur pour B, et par conséquent avoir toutes les paires A, B respectant $A \cdot B = X$. Nous devons appliquer notre algorithme de complexité $O(N)$ pour chaque paire A, B valide, soit une complexité de $O(NX^{\frac{1}{3}} + \sqrt{X})$. La création du tableau cumulatif et du dictionnaire ayant une complexité de $O(N)$, cela n'impacte pas la complexité globale du problème.

3.3 Implémentation

Voici une proposition d'implémentation :

```python
from math import sqrt

def resoudre(x, n, l):

    class Cle:
        def __init__(self, start, end, somme):
            self.start = start
            self.end = end
            self.longueur = end-start
            self.somme = somme

        def __repr__(self):
            return " ".join(map(str, l[self.start:self.end]))

        def __eq__(self, other):
            return self.start == other.start and self.end == other.end

        def __lt__(self, other):
            # Les clés sont triées d'abord par leur longueur puis par leur somme
            return (self.longueur < other.longueur
                    or (self.longueur == other.longueur
                        and self.somme < other.somme))
```

```

        def __bool__(self):
            # Une clé impossible est représentée par une longueur nulle
            return self.longueur > 0

    # Création du tableau cumulatif C
    cumul = [0]
    somme = 0
    for elt in l:
        somme += elt
        cumul.append(somme)

    # Création du dictionnaire associatif D
    hashmap = {}
    for i, somme in enumerate(cumul):
        if not somme in hashmap:
            hashmap[somme] = i

    def sous_tableau(s):
        """
        Retourne la plus longue clé de somme s
        """
        longueur = 0
        start = 0
        end = 0
        for j, somme in enumerate(cumul):
            if somme-s in hashmap:
                i = hashmap[somme-s]
                if j-i > longueur:
                    start = i
                    end = j
                    longueur = j-i
        return Cle(start, end, s)


    # Première clé
    cle1 = Cle(0, 0, 0)
    # Deuxième clé
    cle2 = Cle(0, 0, 0)

    if x == 0:
        cle1 = Cle(0, n, somme)
        cle2 = sous_tableau(0)
    else:
        for a in range(-int(sqrt(abs(x))), int(sqrt(abs(x)))+1):
            if a == 0:
                continue
            if x % a == 0:
                b = x // a
                clea = sous_tableau(a)
                cleb = sous_tableau(b)
                if (clea and cleb and
                    clea.longueur + cleb.longueur > cle1.longueur + cle2.longueur):
```

```
                            cle1 = max(clea, cleb)
                            cle2 = min(clea, cleb)

    if cle1 and cle2:
        print(cle1)
        print(cle2)
    else:
        print("IMPOSSIBLE")

if __name__ == '__main__':
    x = int(input())
    n = int(input())
    l = list(map(int, input().split()))
    resoudre(x, n, l)
```

4 Fuite de clavier

4.1 Énoncé

Joseph Marchand est un petit brigand. Il a installé sur l'ordinateur de son amie Alice un keylogger (un logiciel espion qui capture les entrées du clavier) en dépit des réglementations en vigueur qui condamnent fermement ce genre de pratique.

Il essaie en effet de récupérer le mot de passe Prologin d'Alice ! Malheureusement pour lui, le keylogger a enregistré toutes les frappes sur le clavier, sans distinction de s'il s'agissait d'un mot de passe ou non.

Joseph se retrouve donc avec une suite de caractères composé de lettres minuscules, majuscules, de nombres et de caractères spéciaux.

Il sait juste que le mot de passe d'Alice répond aux exigences de sécurité suivantes :

- Contenir au moins une minuscule (`a-z`)
- Contenir au moins une majuscule (`A-Z`)
- Contenir au moins un nombre (`0-9`),
- Contenir au moins un caractère spécial (`!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`),
- La taille du mot de passe est de k caractères.

Aidez Joseph à savoir combien de chaines de **k** caractères pourraient être le mot de passe d'Alice.

4.2 Solution

Le but est de savoir combien de sous-chaines de taille **k** caractères répondent aux critères. Cet exercice peut être résolu en utilisant une fenêtre coulissante contenant **k** éléments.

Il y a deux étapes à suivre :

1. Initialiser la fenêtre
2. Faire coulisser (mettre à jour) la fenêtre sur toute la chaîne.

4.2.1 Initialisation

La fenêtre doit prendre en compte 4 valeurs :

- Le compteur de lettres minuscules ('lower')
- Le compteur de lettres majuscules ('upper')
- Le compteur de nombres ('digits')
- Le compteur des caractères spéciaux, ceux qui ne vont dans aucune autre catégorie ('special')

Nous pouvons créer une fonction associant à un caractère sa catégorie (**get_class** dans l'implémentation), puis une autre fonction pour déterminer si la fenêtre est valide, c'est à dire qu'elle contient un mot de passe correct (**is_valid** dans l'implémentation).

Pour évaluer la première fenêtre, il faut incrémenter le compteur de chaque catégorie des **k** premiers caractères.

Un autre compteur est créé contenant le nombre de mots de passe corrects (**n_valid** dans l'implémentation). Ensuite, si la première fenêtre est valide, nous pouvons incrémenter ce compteur.

4.2.2 Mise à jour

Il ne reste plus qu'à faire coulisser la fenêtre sur les $n - k$ prochains caractères. Pour cela, nous allons itérer dans toutes les fenêtres existantes, de gauche à droite puis prendre en compte seulement le caractère ne faisant plus partie de la dernière fenêtre (**last**) ainsi que le nouveau caractère, le plus à droite de la fenêtre (**new**).

A présent, nous pouvons décrémenter le compteur de la classe de **last** et incrémenter celui de la classe de **new**.

La fenêtre est à jour, nous pouvons tester si elle est valide via **is_valid** pour incrémenter **n_valid** si besoin.

Le résultat est **n_valid**.

4.2.3 Complexité

Les fonctions **get_class** et **is_valid** ont un temps et une mémoire constante.

Initialiser la fenêtre a une complexité de $O(k)$ en temps ce qui est inclus dans $O(n)$ puisque $k \leq n$.

Mettre à jour la fenêtre a une complexité de $O(n)$ en temps (la boucle ne contient que des opérations de temps constant).

La complexité totale est alors de $O(n)$ en temps et $O(1)$ en mémoire si nous ne comptons pas les entrées.

4.3 Implémentation

```python
import string

if n < k:
    print(0)
    return

chars = {
    'lower': 0,
    'upper': 0,
    'digit': 0,
    'special': 0,
}

def get_class(c):
    if c in string.ascii_lowercase:
        return 'lower'
    elif c in string.ascii_uppercase:
        return 'upper'
    elif c in string.digits:
        return 'digit'
    else:
        return 'special'

def is_valid(chars):
    return (
        chars['lower'] >= 1 and chars['upper'] >= 1 and
        chars['digit'] >= 1 and chars['special'] >= 1
    )

n_valid = 0

# Characters within chaine[0 : k]
for c in chaine[:k]:
    chars[get_class(c)] += 1

# chaine[:k] is valid
if is_valid(chars):
    n_valid += 1

# Test chaine[i : i + k]
for last, new in zip(chaine[:n - k], chaine[k : n]):
    chars[get_class(last)] -= 1
    chars[get_class(new)] += 1

    # chaine[i : i + k] is valid (i is the loop index)
    if is_valid(chars):
        n_valid += 1
```

5 Coffre-fort

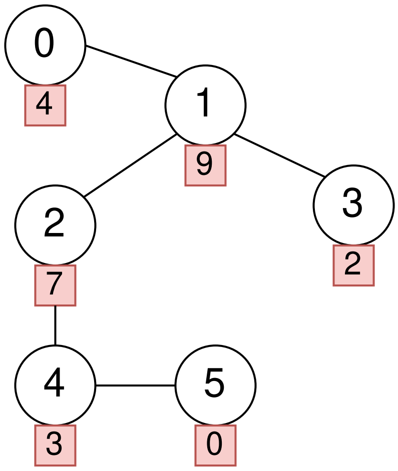
5.1 Énoncé

Joseph Marchand doit ouvrir un coffre électronique sécurisé.

Le coffre contient un circuit complexe, décrit sous la forme d'un ensemble de puces, reliées entre elles par des fils. Il existe toujours exactement un chemin entre chaque paire de puces. Chaque puce envoie un signal s_i au verrou lorsqu'elle est activée. Joseph Marchand tente de passer outre la sécurité du coffre et a besoin de votre aide pour calculer rapidement les effets de ses manipulations.

Il veut régulièrement connecter les deux bornes d'un générateur électrique à deux puces a et b du circuit, et se demande quel signal est envoyé au verrou du coffre suite à cette opération. Une puce c est activée s'il existe un chemin de a vers b passant par c, qui ne repasse pas deux fois par le même fil. Le signal reçu par le verrou est alors le produit du signal envoyé par chaque puce activée, modulo 1671404011 (car on ne peut pas représenter de trop grands nombres par un signal électrique).

5.1.1 Exemple



5.2 Solution

5.2.1 Stratégie

Pour pouvoir répondre aux requêtes, il est nécessaire d'avoir une manière de trouver et parcourir l'unique chemin entre deux puces. Seulement, il n'est pas envisageable de les parcourir intégralement à chaque requête. En effet, un chemin est constitué d'$O(N)$ puces, et parcourir chaque chemin pour chaque requête résulterait en une complexité temporelle de $O(R \times N)$, ce qui excéderait déjà les limites de performance au vu des contraintes.

Une autre stratégie envisageable aurait été d'enraciner l'arbre depuis une puce arbitraire r, et de pré-calculer pour chaque puce le produit des signaux du chemin le reliant à la racine à la manière d'un tableau préfixe. Ensuite, un algorithme de plus petit ancêtre commun permettrait de diviser le chemin entre deux puces en deux plus petits chemins, reliant les deux puces a et b à leur plus petit ancêtre commun c. Enfin, il aurait pu être possible de calculer le produit des signaux entre a (ou b) et c en divisant le produit des signaux entre a et r (précédemment calculé) avec le produit des signaux entre c et r (également précédemment calculé). Ainsi, il aurait été possible de calculer le produit des signaux entre a et c, ainsi qu'entre c et b, et le produit de ces deux valeurs aurait pu donner le résultat attendu. Cependant, sous contrainte du modulo, il serait impossible d'effectuer la division entre deux valeurs, car le modulo donné n'est pas premier et n'est donc pas inversible.

Nous sommes donc contraints de trouver la réponse sans tableau préfixe, car nous ne pouvons pas effectuer de division. L'idée est alors de modifier l'algorithme du plus petit ancêtre commun afin d'enregistrer en plus les produits des signaux entre chaque puce et son 2^k-ème ancêtre (exclu). Ainsi, le produit des signaux entre a et b peut se trouver en calculant, depuis les valeurs pré-calculées, les produits des signaux entre a et c (exclu), ainsi qu'entre b et c, puis en multipliant le produit de ces deux valeurs par le signal envoyé par c.

5.2.2 Plus petit ancêtre commun

L'algorithme du plus petit ancêtre commun est un algorithme permettant, après un pré-traitement d'une complexité temporelle de $O(N \log N)$, de trouver en une complexité de $O(\log N)$ l'ancêtre commun à deux puces a et b qui soit le plus éloigné de la racine d'un arbre.

Le principe est le suivant : pour chaque puce, on pré-calcule son 2^k-ème ancêtre, pour tout k entre 0 et $\log_2(N)$, ainsi que sa profondeur dans l'arbre. Dans notre cas, nous pré-calculerons également le produit des signaux situés sur le chemin entre la puce et son 2^k-ème ancêtre (exclu). On considère l'ancêtre de la racine comme étant la racine elle-même. Depuis cette information, on peut calculer en une complexité temporelle de $O(\log K)$ le K-ème ancêtre de toute puce. Pour ce faire, il suffit de parcourir la décomposition sous forme de puissances de 2 de K (forme binaire), et pour chaque puissance i se déplacer de la puce actuelle à son i-ème ancêtre.

La première étape du traitement d'une requête consiste à remonter de la puce la plus profonde jusqu'à une puce située à la même profondeur. Supposons la puce a plus profonde que b sans perte de généralité (il n'y a rien à faire si les deux puces sont déjà de la même profondeur). Appelons $P(x)$ la profondeur d'une puce x. Le plus petit ancêtre commun entre a et b est identique au plus petit ancêtre commun entre le $P(a) - P(b)$-ème ancêtre de a et b. L'avantage de considérer le $P(a) - P(b)$-ème ancêtre de a plutôt que a en elle-même est qu'ainsi, les deux puces auront la même profondeur, et seront donc situées à la même distance de leur plus petit ancêtre commun. Nous enregistrons également le produit des signaux entre a et son $P(a) - P(b)$-ème ancêtre (exclu), avec lequel nous multiplierons notre résultat final.

Maintenant, il nous reste à trouver le plus petit ancêtre commun de deux puces situées sur la même profondeur. Pour cela, pour chaque 2^k à compter de $k = \log_2(N)$, si le 2^k-ème ancêtre de a est différent de celui de b, cela signifie que leur ancêtre commun est situé plus loin d'eux. On déplace alors nos deux puces de 2^k puces en arrière, en enregistrant le produit des signaux croisés au passage. Faire cela jusqu'à $k = 0$ nous assure d'avoir a et b qui sont, au pire, fils directs de c, la puce recherchée. Depuis cette situation, la résolution est triviale.

5.3 Implémentation

```python
from sys import setrecursionlimit
setrecursionlimit(10**6)

MOD = 1671404011

def calculer_signaux(n, m, r, signaux, fils, questions):
    """
    :param n: nombre de puces
    :type n: int
    :param m: nombre de fils
    :type m: int
    :param r: nombre de questions
    :type r: int
    :param signaux: liste des signaux
    :type signaux: list[int]
    :param fils: liste des fils entre les puces
    :type fils: list[dict["puce1": int, "puce2": int]]
    :param questions: liste des questions
    :type questions: list[dict["puce a": int, "puce b": int]]
    """
    adj = [[] for _ in range(n)] # Liste d'adjacence de l'arbre
    for fil in fils:
        a = fil['puce1']
        b = fil['puce2']
        adj[a].append(b)
        adj[b].append(a)

    # On considère la puce 0 comme racine

    # ancetre[k][i] est un tuple contenant le 2^k-ieme ancetre de la puce k,
    # ainsi que le produit des signaux entre la puce k et son 2^k-ieme ancetre
    # exclu
    ancetre = [[None] * n]

    # profondeur[i] contient la profondeur du puce i
    profondeur = [0] * n

    def init_ancetre(racine, parent, profondeur_racine):
        # Parcours en profondeur de l'arbre pour l'enraciner.
        # Initialise le premier niveau du tableau ancetre ainsi que le tableau
        # profondeur.
        profondeur[racine] = profondeur_racine
        for enfant in adj[racine]:
            if enfant == parent:
                continue
            ancetre[0][enfant] = (racine, signaux[enfant])
            init_ancetre(enfant, racine, profondeur_racine + 1)

    init_ancetre(0, -1, 0)
    ancetre[0][0] = (0, 1)

    # Remplissage du tableau ancetre
```

```
    for k in range(n.bit_length()): # ~log_2(n) niveaux sont suffisants
        niveau = []
        for i in range(n):
            # Le 2^k-ieme ancetre de i est le 2^(k-1)e ancêtre
            # de son 2^(k-1)e ancêtre
            puceA, prodA = ancetre[-1][i]
            puceB, prodB = ancetre[-1][puceA]
            niveau.append((puceB, prodA * prodB % MOD))
        ancetre.append(niveau)

    def k_ancetre(i, k):
        # Retourne le k-ième ancêtre de i ainsi que le produit des signaux entre
        # i et son k-ième ancêtre
        produit = 1
        j = 0
        while k > 0:
            if k % 2 == 1:
                i, prod = ancetre[j][i]
                produit = produit * prod % MOD
            k //= 2
            j += 1
        return i, produit

    def lca(a, b):
        # Retourne le plus petit ancêtre commun entre a et b, ainsi que le
        # produit des signaux entre a et b (inclus)

        produit = 1
        # Premier saut pour mettre a et b à la même profondeur
        if profondeur[a] > profondeur[b]:
            a, produit = k_ancetre(a, profondeur[a] - profondeur[b])
        if profondeur[b] > profondeur[a]:
            b, produit = k_ancetre(b, profondeur[b] - profondeur[a])

        # On fait tout les sauts de taille 2^k possibles (par taille
        # décroissante) tant qu'on n'arrive pas sur la même puce
        for j in range(n.bit_length(), -1, -1):
            sautA, prodA = ancetre[j][a]
            sautB, prodB = ancetre[j][b]
            if sautA != sautB:
                a = sautA
                b = sautB
                produit = produit * prodA * prodB % MOD

        # On fait le dernier saut si a != b
        if a != b:
            produit = produit * signaux[a] * signaux[b] % MOD
            a = b = ancetre[0][a][0]

        # On comptabilise le signal racine
        produit = produit * signaux[a] % MOD
        return a, produit

```

```python
    for question in questions:
        a = question['puce a']
        b = question['puce b']
        racine, produit = lca(a, b)
        print(produit)

if __name__ == '__main__':
    n = int(input())
    m = int(input())
    r = int(input())
    signaux = list(map(int, input().split()))
    fils = [
        dict(zip(("puce1", "puce2"), map(int, input().split())))
        for _ in range(m)
        ]
    questions = [
        dict(zip(("puce a", "puce b"), map(int, input().split())))
        for _ in range(r)
        ]
    calculer_signaux(n, m, r, signaux, fils, questions)
```

6 Coffre électronique

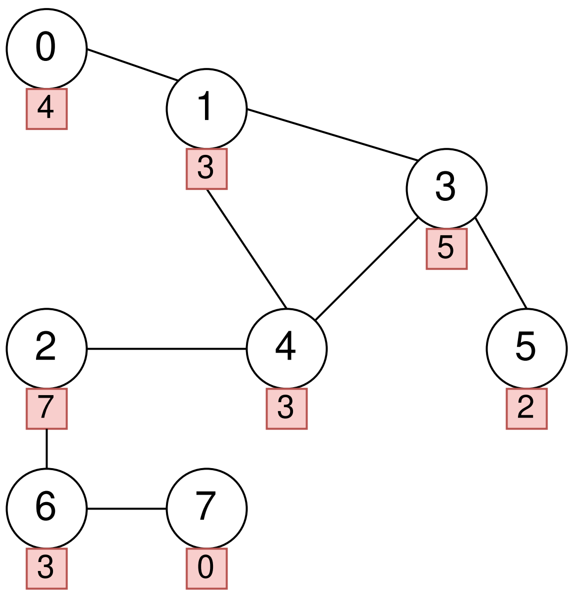
6.1 Énoncé

Joseph Marchand s'est rendu compte que le coffre qu'il cherche à ouvrir est bien trop sécurisé pour qu'il puisse deviner la combinaison d'une façon aussi simple ! Cependant tout espoir n'est pas perdu, car il a réussi à modifier le circuit. Il a rajouté un certain nombre de cables électriques, et est capable de changer la valeur d'une puce à volonté.

Le coffre contient un circuit complexe, décrit sous la forme d'un ensemble de puces, reliées entre elles par des fils. Il existe au moins un chemin entre chaque paire de puces (depuis que Joesph à rajouté des fils, il peut y en avoir plusieurs). Chaque puce envoie un signal s_i au verrou lorsqu'elle est activée. Joseph Marchand tente de passer outre la sécurité du coffre et a besoin de votre aide pour calculer rapidement les effets de ses manipulations.

Il veut régulièrement modifier la valeur s_i contenue par une puce, puis mesurer l'effet que cela à sur le circuit en connectant les deux bornes d'un générateur électrique à deux puces a et b du circuit. Il se demande alors quel signal est envoyé au verrou du coffre suite à cette opération. Une puce c est activée s'il existe un chemin de a vers b passant par c, qui ne repasse pas deux fois par le même fil (mais peut passer deux fois par la même puce). Le signal reçu par le verrou est alors le produit du signal envoyé par chaque puce activée, modulo 1671404011 (car on ne peut pas représenter de trop grands nombres par un signal électrique).

6.1.1 Exemple



Ce circuit est l'état initial du circuit dans le premier exemple d'entrée. Les valeurs des puces 2, 3 et 5 vont cependant changer.

6.2 Solution

Cet exercice ressemble au précédent, qui est une introduction à celui-ci. Deux modifications viennent compliquer le problème : ce n'est plus un arbre, mais un graphe quelconque, et il y a des requêtes de modifications.

6.2.1 Transformer le graphe en arbre

La première difficulté n'en est pas une, mais illustre qu'il est parfois possible de se ramener à un problème plus simple. En effet, un arbre est notamment caractérisé par l'absence de cycles. Prenons un tel cycle de nœuds $c_1, \ldots, c_k$. On va montrer que pour tout couple (a, b) de puces activées, si une puce du cycle est activée, elles le sont toutes.

En effet, si on a un tel chemin entre a et b passant par c_1, on a un chemin $a = x_1, \ldots, x_m = c_1$ (éventuellement avec $m = 1$) de a vers c_1, qui ne passe par aucune arête du cycle, puis de c_1 vers c_i dont les arêtes peuvent appartenir au cycle ou non, et enfin de c_i vers b que l'on note $c_i = y_1, \ldots, y_l = b$, et qui ne passe pas par le cycle. Alors on a les chemins $x_1, \ldots, x_m, c_2, \ldots, c_{i-1}, y_1, \ldots, y_l$ et $x_1, \ldots, x_m, c_k, \ldots, c_{i+1}, y_1, \ldots, y_l$ qui vont donc activer tous les nœuds du cycle.

Nous pouvons donc contracter tous les cycles du graphe en un seul "gros" nœud, nous donnant un arbre des cycles du graphe. Une manière simple de le faire est de parcourir le graphe avec un DFS et d'utiliser un Union-Find pour fusionner les cycles : lorsque l'on arrive sur un nœud, on le marque "en cours", et on retire cette marque une fois son exploration finie. Si un de ses voisins autre que son parent est également en cours, c'est qu'on a trouvé un cycle : il se trouve parmi ses ancêtres dans l'arbre d'exploration du DFS. On fusionne alors tous les nœuds entre le nœud actuel et le nœud ancêtre (les deux extrémités incluses).

Dans la suite, on considérera cet arbre de nœuds fusionné. Mettre à jour la valeur d'un nœud de l'arbre est très facile, compte donné une modification sur un nœud du graphe initial : on construit pour chaque nœud i de l'arbre un arbre binaire T_i de produit, où les nœuds du graphe fusionnés ensemble sont représentés par les feuilles. Ainsi si T_i correspond à une fusion de n_i nœuds différents, il est possible lors d'une modification d'obtenir la nouvelle valeur de i en $O(\log(n_i))$. Il suffit de mettre à jour la feuille affectée, et de remonter T_i en recalculant le produit dans chaque nœud parent. On va donc abstraire les requêtes comme des requêtes sur un arbre, et plus un graphe quelconque.

6.2.2 Chemins lourds

Nous avons donc un arbre et des requêtes de modification, et on veut calculer le produit entre deux nœuds. En utilisant les idées de l'exercice précédent, on sait qu'il suffit de savoir calculer le produit entre un nœud et l'un de ses ancêtres, ce qui à l'aide du plus petit ancêtre commun, nous permet d'avoir la réponse. Reprenons l'algorithme bourrin : si on a une requête entre x et un ancêtre y, il suffit de calculer le produit en remontant arc par arc de x vers y. Les requêtes de modifications sont alors très simples : comme on recalcule à chaque fois, il suffit de modifier la valeur d'un nœud. Mais chaque requête peut être lente : si l'arbre n'est qu'une longue chaîne $a_1, \ldots, a_N$, cela peut prendre $O(N)$ pour chacune.

Le cas d'une longue chaîne semble plus simple que le problème avec un arbre complet, mais nous pose problème : analysons-le. Cela revient à avoir un tableau de valeurs $a_1, \ldots, a_N$, et on veut pouvoir modifier des valeurs n'importe où, mais aussi calculer le produit entre deux valeurs. Et nous avons déjà la solution ! L'arbre binaire utilisé lors de la section précédente permet également de répondre à ce problème en $O(\log(N))$, en ajoutant une fonction récursive pour calculer le produit entre deux cases arbitraires plutôt que tout le graphe.

Cela nous donne alors une idée : tenter de poser ces arbres binaires sur le graphe, ou du moins sur les branches les plus longues. On va donc introduire le concept de **décomposition en chemins lourds**. L'idée est d'avoir les chemins les *plus longs (en réalité, ce ne sont pas toujours les plus longs, mais ce sont les plus* lourds tels qu'expliqué ensuite) qui seront des sortes d'"autoroutes", où les calculs seront réalisés par des arbres binaires. Pour construire le premier chemin lourd, on part du nœud racine x_1 et on sélectionne le nœud x_2 ayant le plus grand sous-arbre (ou le plus profond, cela fonctionne aussi). On continue ainsi à sélectionner le fils le plus lourd à chaque fois, jusqu'à obtenir une branche $x_1, \ldots, x_k$ avec x_k une feuille. Pour chaque fils de $x_1, \ldots, x_{k-1}$ qui n'a pas été sélectionné, on va construire un autre chemin lourd partant de ce nœud, et ainsi de suite jusqu'à ce que chaque nœud fasse partie d'un chemin lourd.

La manière la plus simple d'implémenter l'idée ci-dessus est, pour chaque nœud, de sélectionner le fils le plus lourd à l'avance, puis d'utiliser un simple DFS pour construire les chemins en un seul parcours. On associe ensuite à chaque chemin lourd un arbre binaire, et à chaque nœud l'identifiant de cet arbre, ainsi que sa position dans celui-ci, parmi les feuilles. Chaque arbre binaire calcule un produit et effectuer une modification est alors trivial : on modifie la valeur dans l'arbre binaire du chemin lourd du nœud, et on le met à jour en $O(\log(N))$.

Il reste alors à utiliser cela pour calculer un produit de x vers y. On va reprendre notre algorithme consistant à remonter l'arbre arête par arête, mais en utilisant les chemins lourds. Supposons que l'on se trouve, en se déplaçant selon ce glouton, à un nœud a_i, descendant de y. Il y a alors 3 cas pour trouver le prochain nœud, a_{i+1} :

— Cas 1 : a_i est le premier nœud de son chemin lourd. On remonte alors vers son parent en $O(1)$, et on utilise sa valeur pour le produit.

— Cas 2 : y et a_i sont dans le même chemin lourd. On calcule alors le produit entre a_i et y en $O(\log(N))$.

— Cas 3 : y n'est pas dans le chemin lourd de a_i. On remonte au nœud le plus haut de ce chemin (qui sera a_{i+1}) et on calcule le produit entre a_i et a_{i+1} en $O(\log(N))$.

6.2.3 Complexité

Soit N le nombre de nœuds du graphe et R le nombre de requêtes. Pour calculer la complexité, il faut savoir combien de fois chacun des cas 1, 2 et 3 peuvent arriver lors d'une requête.

Trivialement, le cas 2 n'arrive qu'une fois, car il mène directement à la réponse. Les cas 1 et 3 alternent et arrivent le même nombre de fois, à une constante près. On va alors montrer qu'avec les chemins lourds, le cas 1 n'arrive qu'au plus $\log_2(N)$ fois. Soit $a_1, \dots, a_k$ l'ensemble des nœuds rencontrés dans le cas 1. a_k n'est pas le fils lourd de son propre parent, donc le sous-arbre de a_k est de taille $\leqslant \frac{N}{2}$. Par le même raisonnement, le sous-arbre de a_{k-1} est de taille $\leqslant \frac{N}{4}$, et récursivement celui de a_1 est de taille $\leqslant \frac{N}{2^k}$, mais de taille au moins 1. Ainsi, $1 \leqslant \frac{N}{2^k}$ donc $k \leq \log_2(N)$, d'où la conclusion.

La complexité de l'algorithme est donc de $O(R \log^2(N))$.

6.3 Implémentation

```python
import sys
sys.setrecursionlimit(10**9)

lca_cur_pos = 0

def main():
    MOD = 17 * 9697 * 10139
    LCA_LOG = 18

    nb_nodes = int(input())
    nb_edges = int(input())
    nb_queries = int(input())

    node_signal = list(map(int, input().split()))
    cycle_neighbors = [[] for _ in range(nb_nodes)]
    neighbors = [[] for _ in range(nb_nodes)]
    compo = [-1] * nb_nodes
    compo_size = [1] * nb_nodes
    state = [-1] * nb_nodes
    heavy_child = [0] * nb_nodes
    tree_depth = [0] * nb_nodes
    tree_parent = [0] * nb_nodes
    heavy_parent = [0] * nb_nodes

    lca_min_left = [[0] * LCA_LOG for _ in range(nb_nodes*2)]
    lca_min_right = [[0] * LCA_LOG for _ in range(nb_nodes*2)]
    node_lca_pos = [0] * nb_nodes
    root = None

    i_leaf_of_node = [0] * nb_nodes
    i_leaf_of_signal = [0] * nb_nodes

    segtrees = [[0, []] for _ in range(nb_nodes)] # taille_reelle, noeuds_internes
    compo_sig_tree = [[0, []] for _ in range(nb_nodes)]

    for _ in range(nb_edges):
        node_left, node_right = map(int, sys.stdin.readline().split())
        cycle_neighbors[node_left].append(node_right)
        cycle_neighbors[node_right].append(node_left)

    # Arbres binaires

    def heavy_segtree_build(tree, size, right_node):
        # Initialistion d'un arbre binaire sur un chemin lourd avec au moins 'size'
        # feuilles, le long d'une branche se terminant au noeud 'right_node'
        leaves = []
        while size:
            leaves.append(right_node)
            right_node = tree_parent[right_node]
            size -= 1
        leaves = leaves[::-1]
        for i, i_node in enumerate(leaves):
```

```
            i_leaf_of_node[i_node] = i

        segtree_build(tree, [node_signal[l] for l in leaves])

    def segtree_build(tree, leaves):
        # Initilisation d'un arbre binaire, avec les valeurs de ses feuilles
        real_size = 1
        while real_size < len(leaves):
            real_size *= 2
        tree[:] = [1] * (real_size * 2)

        for i, v in enumerate(leaves):
            tree[real_size + i] = v

        for i in range(real_size-1, 0, -1):
            tree[i] = (tree[i*2] * tree[i*2+1]) % MOD

    def segtree_query_prod(tree, deb, fin):
        # Produit sur un intervalle dans l'arbre binaire
        expl = [(1, 0, len(tree)//2)]
        total = 1
        while expl:
            node, sub_deb, sub_fin = expl.pop()
            if sub_fin <= deb or sub_deb >= fin:
                continue
            elif deb <= sub_deb and sub_fin <= fin:
                total = (total * tree[node])%MOD
            else:
                expl.append((node*2, sub_deb, (sub_deb + sub_fin) // 2))
                expl.append((node*2+1, (sub_deb + sub_fin) // 2, sub_fin))
        return total

    def segtree_set(tree, i_leaf, set_sig):
        # Modification d'une valeur dans l'arbre binaire
        node = len(tree)//2 + i_leaf
        tree[node] = set_sig % MOD
        while node > 1:
            node //= 2
            tree[node] = (tree[node*2] * tree[node*2+1])%MOD

    # Union-find

    def uf_find(x):
        p = [x]
        if compo[x] != -1:
            while compo[p[-1]] != -1:
                p.append(compo[p[-1]])
            for y in p[:-1]:
                compo[y] = p[-1]
        return p[-1]

    def uf_union(x, y):
        x, y = uf_find(x), uf_find(y)
```

```python
        if x != y:
            if compo_size[x] < compo_size[y]:
                x, y = y, x
            elif compo_size[x] == compo_size[y]:
                compo_size[x] += 1
            compo[y] = x

    # Fusion de composantes cycliques

    merge_compos_stack = [[0, -1, 0]]
    compo_stack = []
    def merge_compos(node, parent):
        while merge_compos_stack:
            node, parent, step = merge_compos_stack.pop()

            if step == 0:
                if state[node] == -1:
                    compo_stack.append(node)
                    state[node] = 0

                    merge_compos_stack.append([node, parent, 1])
                    for v in cycle_neighbors[node]:
                        if v != parent:
                            merge_compos_stack.append([v, node, 0])
                elif state[node] == 0:
                    while uf_find(parent) != uf_find(node):
                        compo_stack.pop()
                        uf_union(parent, compo_stack[-1])

            else:
                state[node] = 1
                if compo_stack[-1] == node:
                    compo_stack.pop()

    def assign_signal_in_compos():
        signals_in = [[] for _ in range(nb_nodes)]
        for i_node in range(nb_nodes):
            i_leaf_of_signal[i_node] = len(signals_in[uf_find(i_node)])
            signals_in[uf_find(i_node)].append(node_signal[i_node])

        for i_node in range(nb_nodes):
            if signals_in[i_node]:
                segtree_build(compo_sig_tree[i_node], signals_in[i_node])
                node_signal[i_node] = compo_sig_tree[i_node][1]

    # Construction des chemins lourds

    def find_heavy_child(node, parent, node_depth):
        state[node] = 0
        tree_depth[node] = node_depth
        tree_parent[node] = parent
        node_children = []
        max_child_depth = node_depth
```

```
        heavy_child[node] = -1

        for v in neighbors[node]:
            if state[v] == -1:
                depth = find_heavy_child(v, node, node_depth+1)
                node_children.append(v)
                if depth > max_child_depth:
                    max_child_depth = depth
                    heavy_child[node] = v
        neighbors[node] = node_children
        return max_child_depth

    def build_heavy_paths(node):
        if node == root or heavy_child[tree_parent[node]] != node:
            # Dans le cas où on n'est pas un fils lourd, il faut construire son propre
            # chemin lourd
            heavy_parent[node] = node
            right_node = node
            while heavy_child[right_node] != -1: # Trouver tous les fils lourds
                right_node = heavy_child[right_node]
                heavy_parent[right_node] = node
            # Construire l'arbre binaire sur le chemin lourds à partir du noeud
            heavy_segtree_build(segtrees[node],
                tree_depth[right_node] - tree_depth[node]+1, right_node)

        for v in neighbors[node]:
            build_heavy_paths(v)

    # Remonter l'arbre via des arrêts et chemins de poid lourd jusqu'à une certaine
    # profondeur

    def get_lock_signal(node, top_depth):
        lock_signal = 1
        while node != -1 and tree_depth[node] >= top_depth:
            h_parent = heavy_parent[node]
            if top_depth <= tree_depth[h_parent]:
                # Si on n'est pas sur le même chemin lourd que l'objectif, on le remonte
                # complètement, et on remonte ensuite jusqu'au parent
                # (cas 1 et 3 ensembles)
                lock_signal = (lock_signal * segtree_query_prod(segtrees[h_parent], 0,
                    tree_depth[node] - tree_depth[h_parent] + 1)) % MOD
                node = tree_parent[h_parent]
            else:
                # Cas 2 : on remonte jusqu'au noeud ancêtre objectif
                lock_signal = (lock_signal * segtree_query_prod(segtrees[h_parent],
                    top_depth - tree_depth[h_parent],
                    tree_depth[node] - tree_depth[h_parent]+1)) % MOD
                break
        return lock_signal

    # Plus petit ancêtre commun (construction de la structure)
    def buildLcaTour(node):
        global lca_cur_pos
```

```python
        lca_min_left[lca_cur_pos][0] = tree_depth[node]
        lca_min_right[lca_cur_pos][0] = tree_depth[node]
        node_lca_pos[node] = lca_cur_pos
        lca_cur_pos += 1
        for v in neighbors[node]:
            buildLcaTour(v)
            lca_min_left[lca_cur_pos][0] = tree_depth[node]
            lca_min_right[lca_cur_pos][0] = tree_depth[node]
            lca_cur_pos += 1


    # ===== MAIN =====

    # Transformer le graphe en arbre
    merge_compos(0, -1) # On fusionne les composantes

    # Puis on réassigne les arrêtes
    for i_node in range(nb_nodes):
        for v in cycle_neighbors[i_node]:
            a, b = uf_find(i_node), uf_find(v)
            if a != b:
                neighbors[a].append(b)

    assign_signal_in_compos() # Et on assigne chaque noeud du graphe à un noeud de l'arbre
    root = uf_find(0) # On enracine l'arbre

    # Chemins lourds
    state[:] = [-1]*nb_nodes
    find_heavy_child(root, -1, 0) # Trouver l'enfant le plus lourd pour chaque noeud
    # Et construire les chemins lourds et les arbres binaires associés
    build_heavy_paths(root)

    # Construction de la structure pour le plus petit ancêtre commun
    buildLcaTour(root)
    for i in range(lca_cur_pos):
        s = 1
        while 1 << s <= i+1:
            lca_min_left[i][s] = min(lca_min_left[i][s-1],
                lca_min_left[i-(1 << (s-1))][s-1])
            s += 1
    for i in range(lca_cur_pos-1, -1, -1):
        s = 1
        while 1 << s <= lca_cur_pos - i:
            lca_min_right[i][s] = min(lca_min_right[i][s-1],
                lca_min_right[i + (1 << (s-1))][s-1])
            s += 1


    # Requêtes
    for _ in range(nb_queries):
        # Lecture de l'entrée et conversion des identifiants des noeuds du graphe
        # vers ceux de l'arbre
        set_at, set_signal, req_start, req_end = map(int, sys.stdin.readline().split())
```

```python
        change_compo = uf_find(set_at)
        req_start, req_end = uf_find(req_start), uf_find(req_end)

        # Modification de la valeur dans le sous-arbre binaire d'un noeud de l'arbre
        # principal
        segtree_set(compo_sig_tree[change_compo], i_leaf_of_signal[set_at], set_signal)
        node_signal[change_compo] = compo_sig_tree[change_compo][1]
        # Mise à jour de l'arbre binaire associé au chemin lourd du noeud modifié
        segtree_set(segtrees[heavy_parent[change_compo]],
            i_leaf_of_node[change_compo], node_signal[change_compo])

        # Trouver la profondeur le plus petite ancêtre commun
        if node_lca_pos[req_start] > node_lca_pos[req_end]:
            req_start, req_end = req_end, req_start
        s = 0
        while 1 << (s+1) <= node_lca_pos[req_end] - node_lca_pos[req_start]+1:
            s += 1
        top_depth = min(lca_min_right[node_lca_pos[req_start]][s],
            lca_min_left[node_lca_pos[req_end]][s])

        # Effectuer les deux requêtes pour avoir le produit des valeurs sur un chemin
        signal_left = get_lock_signal(req_start, top_depth)
        signal_right = get_lock_signal(req_end, top_depth+1)
        # Affichage
        sys.stdout.write(f"{(signal_left * signal_right)%MOD}\n")

main()
```

⁂

Félicitations à tous les participantes et participants !

Nous avons tenté de rédiger une correction aussi claire que possible. Néanmoins, si vous avez des questions, n'hésitez pas à nous contacter à l'adresse info@prologin.org.