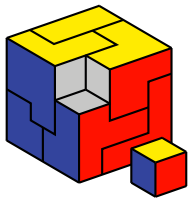
Prologin 2021



Concours National d'Informatique

Correction de la phase de sélection

Correction des qualifications Prologin 2021

Maya El-Gemayel, Valérian Fayt, Aurélien Rebourg, Joël Felderhoff
Milan Gonzalez-Thauvin, Simon Meinhard, Valentin Seux*

24 janvier 2021

Table des matières

*Pour l'équipe des correcteurs 2021 : Kenji Gaillac, Valérian Fayt, Maya El-Gemayel, Valentin Seux

0 Questionnaire

0.1 Question 1 : Malbolge

Énoncé Quelle est la valeur de retour de ce programme en Malbolge ?

```
D'``M#onI;G987g5utt1*MonJl[#F!h}CeS!?a|{)sr8pXn4rqpohg-ediba'eGcba`Y}@\

[TSXQuUTSLKoO10LKJCgAFED=B;_?!7<5:981U/u3,+*N(-&%*#G'&feB"!~w=^]

yxq7otsrk1ingfkd*)J`e^]#a`BA]\[TxRWVUTMqQP2HlLKJCgGFE>&<`#?8\<;43W16/4-,Pq).'&

%$Hih~}${Ay~w=^tyrwvutm3qponmfN+ihgfeG$o
```

Correction Ce programme retourne : **Toujours les meilleurs langages !**

Il suffisait de trouver un interpréteur de Malbolge et de lui donner la chaine de caractères en entrée, par exemple celui-ci.

0.2 Question 2 : Filtre de Bloom

Énoncé Pour un ensemble S contenant un élément X, si l'on cherche à savoir si X est présent dans S en passant par un filtre de Bloom, quelle valeur booléenne est renvoyée par le filtre ?

1. True
2. False
3. True ou False

Correction La réponse attendue était : **True ou False**.

Plus d'informations disponibles sur la page Wikipédia associée.

0.3 Question 3 : FSFE or not FSFE ?

Énoncé Lequel de ces projets n'est pas mené par la Free Software Foundation Europe ?

1. Public Money ? Public Code !
2. Certbot
3. Save Code Share
4. DRM.info

Correction La réponse attendue était **Certbot**.

Retrouvez toutes les activités de la FSFE sur leur site (en anglais).

0.4 Question 4 : Token JWT

Énoncé Quelle est la valeur contenue dans le payload de ce token JWT ?

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJyw6lwb25zZSI6ImplX3N1aXNfcHJvbG9naWlpbiJ9.

VHoYZL8-ePJEb0mbSmSzcPhxSdD2mTk0OEwCsRz8c68
```

Correction Le contenu du token est : **je_suis_prologiiin**.

Vous pouvez tester vous-même et en apprendre plus sur le site https ://jwt.io/.

0.5 Question 5 : Bitwise

Énoncé Quelle est la valeur maximale que peut prendre i pour que l'opération bitwise suivante donne 1 ?

```
    ((1 << (1 << 3 | (i >> 3) | 7)) ^ i) >> 15
```

Correction Cette question a suscitée beaucoup d'interrogations : il fallait répondre **127**.

En testant rapidement en Python :

```
f = lambda i: ((1 << (1 << 3 | (i >> 3) | 7)) ^ i) >> 15
n = 0
while (f(n) == 1)
    n += 1
```

On remarque que la boucle s'arrête quand $n = 128$, donc $f(128) \neq 1$. Le résultat est donc $n = 127$.

Il fallait partir du principe qu'il n'y avait pas d'overflow.

0.6 Question 6 : MarioLANG

Énoncé Quel est la sortie du programme suivant, écrit en MarioLANG ?

```
                 -(+++++ +++++)<
                ===============" 
    +++>+++++>              [! ([!)). ++.---.!|
=========="==========#=|||======#===#==========#|
                        (+)+.++ ))<          |> ---.+++. !|
             ==============="          |"==========#|
                   >
       !  ![    )++++++++ <   .+++++   .++    .-------- <|
        =#==#=============================================="|
            > (.((.)).((----.
            "================
```

Correction Ce programme retourne : **PROLOGIN4EVER**.

Vous pouvez le tester ici. Une version que du programme que vous pouvez copier (avec les espaces en début de ligne) se trouve ici.

0.7 Question 7 : Coupe d'un tore

Énoncé En combien de morceaux maximum peut-on couper un tore, sans le bouger, et avec seulement 3 coupes planaires ?

1. 3
2. 42
3. 8
4. 13

Correction La réponse attendue était : **13**.

Vous pourrez trouver plus d'informations ici, qui donne la formule suivante pour la coupe d'un tore en n coupes planaires : $(n^3 + 3n^2 + 8n)/6$

0.8 Question 8 : Résolution d'un labyrinthe

Énoncé Quel auteur français a donné son nom à un algorithme de résolution de labyrinthe ?

1. Edsger Dijkstra
2. Jonathan Pledge
3. Charles Pierre Trémaux
4. Clémentine Mélois

Correction La réponse attendue est **Charles Pierre Trémaux**, voici le lien wikipédia.

1 Message des dieux

1.1 Énoncé

La guerre entre athéniens et spartiates fait rage. Général des armées, vous vous retrouvez dans une situation difficile. Votre ville est assiégée, et vous savez qu'un prochain assaut pourrait bien être le dernier. Votre honneur - et votre vie - est en ligne de mire : Athènes ne doit pas tomber !

Vous savez qu'il y a différents endroits d'où l'attaque pourrait provenir, mais vous ne disposez pas d'assez d'hommes pour tous les protéger en même temps. Il vous faut impérativement des informations sur les troupes adverses pour éviter une issue fatale.

Heureusement, Hermès, messager des dieux, vous fait parvenir un message. La déesse de la sagesse, Athéna en personne, lui a communiqué des informations capitales, qui pourraient bien retourner le cours de la bataille : l'endroit d'où les forces ennemies vont surgir ! Hélas, afin d'éviter des fuites, elle a dû recourir à un code secret : le message déchiffré sera une lettre qui sera associée à la future position des spartiates.

Vous devez le déchiffrer avec très peu d'informations : vous savez uniquement qu'il s'agit d'une liste de nombres. Le premier nombre de cette séquence représente une lettre majuscule, que vous pourriez retrouver grâce à sa valeur dans la table ascii. Si ce n'est pas une lettre, le message a été corrompu. Sinon, le reste de la séquence, si existant, est les corrections apportées à la lettre.

Par exemple, si la séquence est composée de deux nombres, un nombre représentant la lettre A et le nombre 5, votre message sera F.

Faites vite : le destin d'Athènes est entre vos mains.

1.2 Solution

La solution est composée de plusieurs étapes. La première est de voir le premier caractère ASCII de la séquence d'entiers donnée.

Si ce premier caractère n'appartient pas à la séquence de lettres allant de 'A' à 'Z', on retourne le résultat du message corrompu : une espace.

Sinon il faut continuer à regarder le reste de la séquence, en additionnant les entiers pour avancer dans la liste de 'A' à 'Z'. Il faut cependant faire attention à ne pas dépasser la fin de l'alphabet et donc revenir au début lorsque l'on dépasse la fin. Pour cela, on peut utiliser l'opération modulo (modulo 26 puisque l'alphabet contient 26 lettres).

```
/// \param t taille de la séquence
/// \param s liste des entiers qu'Hermès a donné
char trouverlettre(int t, int* s) {
    char c = s[0];
    if (!(c >= 'A' && c <= 'Z'))
        return ' ';
    c = c - 'A';
    for (int i = 1; i < t; i++)
        c = (c + (s[i] % 26)) % 26;
    return c + 'A';
}
```

2 Généalogie Divine

2.1 Énoncé

En fouillant dans son grenier, Joseph Marchand est tombé sur un ancien jeu de carte. Disposant de temps libre pendant son confinement, Joseph décide de lancer une soirée jeu de société en famille.

Le jeu est similaire à celui des '7 familles', à ceci près qu'il concerne les divinités grecques. Comme dans le jeu des '7 familles', chaque joueur doit rassembler suffisamment de cartes pour compléter un ensemble donné. L'arbre généalogique des divinités grecques est toutefois un peu complexe, donc certains membres apparaissent dans le jeu dans différentes familles.

Joseph est un peu perdu pendant la partie et a du mal à se retrouver dans toutes ces familles entremêlées ! Le but est de l'aider à trouver le plus petit nombre de cartes manquantes pour compléter une famille et ainsi gagner la partie.

2.2 Solution

Pour résoudre ce problème, il faut itérer sur les cartes de chaque famille et déterminer le nombre de cartes manquantes en fonction de la main de Joseph.

Pour chacune des cartes qui sont dans la main de Joseph on décrémente un compteur, dont on se souviendra de la valeur pour la prochaine itération. On pourra donc garder seulement le plus petit nombre de cartes manquantes, qu'on affiche ensuite.

```python
def genealogie_divine(familles, cartes):
    min = len(familles[0])

    for family in familles:
        missing = len(family)

        for card in cartes:
            for c in family:
                if c == card:
                    missing -= 1

        if missing < min:
            min = missing

    print(min)

if __name__ == '__main__':
    f = int(input())
    m = int(input())
    familles = [list(input()) for _ in range(f)]
    c = int(input())
    cartes = list(input())
    genealogie_divine(familles, cartes)
```

3 Labyrinthe Démoniaque

3.1 Énoncé

C'est lors d'un voyage en Grèce que Joseph Marchand eut l'idée de recréer le labyrinthe du minotaure, mais en changeant quelque peu son principe. Son labyrinthe grandeur nature est composé d'une grille de N x M cases. Chaque case renferme une divinité grecque qui attaque le visiteur en lui prenant une part plus ou moins grande de son âme.

Sur la carte vue du dessus on constate que le visiteur démarre sur une des cases en haut et doit rejoindre une des cases en bas. On remarque également que chaque case permet de rejoindre les 3 cases voisines sur la ligne du dessous, menant ainsi vers la sortie après N pas. Heureusement, Joseph donne cette carte, permettant au visiteur d'être informé des divinités présentes dans chaque case ainsi que de leur puissance.

L'objectif est d'arriver à sortir du labyrinthe en conservant un maximum de son âme pour ne pas être transformé en minotaure et rester enfermé à jamais. Comment un visiteur pourrait trouver le meilleur chemin afin de sortir du labyrinthe ?

3.2 Solution

Pour résoudre ce problème, plusieurs solutions existent. C'est fondamentalement un algorithme de plus court chemin auquel on rajoute quelques subtilités comme le fait de retracer le chemin et de vérifier la valeur de sortie.

Nous allons nous concentrer sur une solution optimisée utilisant la programmation dynamique.

L'objectif est simple : on crée un nouveau tableau de même taille que le plateau original. Ensuite on remplace les valeurs de chaque case par la plus petite valeur des 3 cases en dessous. En faisant ça pour toutes les cases, on fait en fait la somme de toutes les cases ayant les plus petites valeurs depuis la première ligne.

Vous remarquerez que cela nous permet d'avoir, à la dernière ligne le meilleur résultat possible pour chaque case. Néanmoins, une fois qu'on a trouvé la valeur de la sortie du meilleur chemin, il faut reconstruire la liste des cases qui ont permis d'arriver là. Soit on enregistre pour chaque case, son père puis on remonte de père en père pour recontruire le chemin original, soit on détermine les pères en regardant les minima des 3 valeurs au dessus de la case puis en itérant dessus.

Il suffit ensuite d'afficher le résultat.

3.3 Complexité

Notons n la longueur du labyrinthe, et m sa largeur.

La complexité mémoire est de l'ordre de la taille du tableau initial, que l'on copie au début, à savoir la largeur multipliée par la longueur : $O(n \cdot m)$.

La complexité en temps est, elle, de $O(n \cdot m + n + m) = O(n \cdot m)$. On passe en effet une fois sur chaque case du tableau.

```
1  def labyrinthe_demonique(a, n, m, plateau):
2      best = [[(plateau[i][j], -1) for j in range(m)] for i in range(n)]
3
4      for i in range(1, n):
5          for j in range(m):
6              val = val2 = val3 = best[i - 1][j][0] + plateau[i][j]
7              if j > 0:
8                  val2 = best[i - 1][j - 1][0] + plateau[i][j]
9              if j + 1 < m:
10                 val3 = best[i - 1][j + 1][0] + plateau[i][j]
11             # set best value and parent index
12             mini = min(val, val2, val3)
13             if mini == val:
```

```
                best[i][j] = (mini, j)
            elif mini == val2:
                best[i][j] = (mini, j - 1)
            elif mini == val3:
                best[i][j] = (mini, j + 1)

    min_index = 0
    # trouver l'index de la sortie du meilleur chemin
    for i in range(1, m):
        if best[n - 1][i] < best[n - 1][min_index]:
            min_index = i
    # vérifier que le chemin est possible
    if best[n - 1][min_index][0] > a:
        print("IMPOSSIBLE")
        return
    last = min_index
    # reconstruire le chemin a l'aide des parents
    chemin = [last]
    for i in range(1, n):
        last = best[n - i][last][1]
        chemin.insert(0, last)
    for x in chemin:
        print(x, end=" ")


a = int(input())
n = int(input())
m = int(input())
plateau = [list(map(int, input().split())) for _ in range(n)]
labyrinthe_demonique(a, n, m, plateau)
```

4 Exercice 4 : Bataille d'eau

4.1 Énoncé

Pour que les habitants de la ville de Joseph aient de l'eau potable, le réseau d'aqueduc de la ville se doit d'être efficace. Il y a des années, les villes de Rome et Tivoli ont été reliées pour permettre une meilleure redistribution de l'eau.

Depuis quelques mois, les habitants de Rome ne sont plus satisfaits de la qualité de l'eau qui provient de Tivoli. Le responsable des aqueducs vous implore de l'aider et de trouver une solution à ce problème !

Après une intense réflexion, vous vous dites que la solution la plus simple est de séparer les aqueducs des deux villes. Bien entendu, Rome et Tivoli ne sont pas les seules à être interconnectées avec des aqueducs !

4.2 Représentation des données

L'exercice nous demande de représenter un réseau de villes reliées par des aqueducs à doubles sens. La structure de données à utiliser est donc un graphe non-orienté.

Il faut ensuite regarder les données en entrée de l'exercice pour savoir la manière de représenter en mémoire le graphe.

Le nombre de sommet n'est pas trop haut (1000 au maximum) et il peut y avoir beaucoup d'arêtes ($10^6 = 1000 \times 1000$), on va donc privilégier une structure de graphe dense en matrice d'adjacence, c'est-à-dire qu'on va utiliser un tableau à deux dimensions contenant des booléens, où $tableau[i][j]$ est vrai si il existe un lien entre la ville i et la ville j.

4.3 Combien d'aqueducs à couper ?

Le problème qu'on a ici à résoudre est un problème connu en théorie des graphes, dont le nom est **min-cut**. Il existe différents algorithmes pour résoudre ce problème, nous allons ici nous concentrer sur une version simplifiée de l'algorithme de Ford-Fulkerson. Il consiste à regarde ce problème comme un problème de **flot**.

On donne en entrée à l'algorithme le sommet de départ (Rome), le sommet d'arrivée (Tivoli) et le graphe G. L'idée est d'essayer de voir comment faire couler le plus d'eau possible entre Rome et Tivoli. On va commencer par considérer que tous les aqueducs ne contiennent pas d'eau, et peuvent faire passer une unité d'eau de poids 1. Donc pour chaque aqueduc de la ville i vers la ville j, on va initialiser $capacit[i][j] = capacit[j][i] = 1$.

Ensuite, on va essayer de trouver un chemin qui emprunte uniquement des aqueducs dont la capacité n'est pas 0 (on "simule" une coulée d'eau de Rome à Tivoli). On appelle ces chemins des **chemins augmentants**. On peut faire ça grâce à un algorithme de parcours en largeur.

Lorsqu'on a trouvé un chemin augmentant, on va le remplir d'eau : on regarde quelle est la plus petite capacité des aqueducs sur le chemin (on la note a), et on va "faire couler" cette capacité sur tout le chemin. Alors si un aqueduc est parcouru dans le sens $i \to j$, la $capacit[i][j]$ est **diminuée** de a (de l'eau coule dans ce sens), et $capacit[j][i]$ est **augmentée** de a (de l'eau coule dans l'autre sens).

À ce stade, il y a deux possibilités. Soit il n'est plus possible d'aller de Rome à Tivoli par un chemin augmentant (le réseau entre Rome et Tivoli est saturé), soit c'est encore possible. Dans le second cas, on recommence en mettant à jour le réseau au fur et à mesure. Dans le premier cas, on a trouvé la quantité maximale d'eau qui peut couler entre Rome et Tivoli. C'est le nombre d'aqueducs à supprimer.

4.4 Trouver où couper

On a trouvé combien d'aqueducs il faut couper, mais reste encore à trouver lesquels.

Cela peut se faire de la manière suivante : on va regarder quelles sont les villes accessibles depuis Rome **en ne passant que par des aqueducs non remplis d'eau**. Cela va séparer le graphe des villes en deux : une partie "Rome" et une partie "Tivoli". Pour les séparer, il faut simplement trouver

les aqueducs remplis d'eau et couper ceux qui ont une extrémité dans la partie "Rome" du graphe et l'autre extrémité dans la partie Tivoli.

Il est possible de démontrer que ces arêtes sont exactement celles qu'il faut couper pour isoler Rome de Tivoli.

4.5 Récapitulatif

In fine, l'algorithme est le suivant :

1. On récupère le graphe G.
2. On initialise *capacit* comme un tableau avec $capacit[i][j] = 0$ si $i \to j$ n'est pas dans G et 1 sinon.
3. On initialise *nombre_a_couper* à 0.
4. Tant qu'il existe un chemin augmentant c entre Rome et Tivoli
5. (a) $nombre\_a\_couper++$.
   (b) On calcule la capacité minimale a d'une arête de c.
   (c) Pour chaque arête $i \to j$ dans c, on fait $capacit[i][j] -= a$ et $capacit[j][i] += a$.
6. On affiche *nombre_a_couper*
7. On initialise *villes_accessibles* à *Rome*.
8. On parcourt le graphe G en partant de *Rome* en rajoutant toutes les villes rencontrées à *villes_accessibles* en ne prenant que des arêtes telles que $capacit[i][j] \neq 0$.
9. On parcourt toutes les arêtes telles que $capacit[i][j] = 0$, et on affiche celles qui ont une extrémité appartenant à *villes_accessibles* et l'autre n'y appartenant pas.

4.6 Complexité temps/mémoire

Dans la suite, on note n le nombre de villes, m le nombre d'aqueducs, et f le nombre d'aqueducs à couper.

L'algorithme présenté ne demande pas plus de mémoire que la taille du graphe, qui est de $O(n^2)$.

Le coût en temps de cet algorithme réside essentiellement dans le calcul du flot optimal. À chaque fois qu'on lance une recherche de chemin augmentant, on fait un parcours en largeur du graphe. Ce parcours coûte $O(n)$ opérations car on parcourt au pire toutes les arêtes du graphe. On va chercher autant de chemin qu'il y a d'aqueducs à couper, le coût total de calcul du flot est donc $O(m \cdot f)$.

Le coût en temps pour trouver les aqueducs à couper est de $O(m)$.

Coût final de l'algorithme en temps $O(f \cdot m + n^2 + m) = O(f \cdot m + n^2)$.

4.7 Proposition d'implémentation

```
1  INF = 10000000
2
3
4  def trouver_chemin_augmentant(graphe, noeud_depart, noeud_cible, capacites_actuelles):
5      """
6      Retourne un chemin augmentant entre noeud_depart et noeud_cible. Si ce n'est pas possible,
7      """
8      noeud_precedent = [None] * len(graphe)
9      pile = [noeud_depart]
10     noeud_precedent[noeud_depart] = noeud_depart
11
12     while len(pile) > 0:
13         noeud_courant = pile.pop()
14         for voisin in graphe[noeud_courant]:
15             if (
```

```
                noeud_precedent[voisin] is None
                and capacites_actuelles[noeud_courant][voisin] > 0
            ):
                noeud_precedent[voisin] = noeud_courant
                pile.append(voisin)

    if noeud_precedent[noeud_cible] is None:
        return []

    noeud_courant = noeud_cible
    chemin_augmentant = [noeud_cible]
    while noeud_precedent[noeud_courant] != noeud_courant:
        noeud_courant = noeud_precedent[noeud_courant]
        chemin_augmentant.append(noeud_courant)
    chemin_augmentant.reverse()
    return chemin_augmentant


def ford_fulkerson(graphe, noeud_depart, noeud_cible):
    """
    Calcule le flot-max du graphe et le renvoie
    """

    capacites_actuelles = [[0] * len(graphe) for _ in range(len(graphe))]
    flot = [[0] * len(graphe) for _ in range(len(graphe))]
    for noeud in range(len(graphe)):
        for voisin in graphe[noeud]:
            capacites_actuelles[noeud][voisin] = 1
            capacites_actuelles[voisin][noeud] = 1

    chemin_augmentant = trouver_chemin_augmentant(
        graphe, noeud_depart, noeud_cible, capacites_actuelles
    )
    while chemin_augmentant != []:
        capacite_mini = INF
        index_mini = -1
        for i in range(len(chemin_augmentant) - 1):
            capacite_arete = capacites_actuelles[chemin_augmentant[i]][
                chemin_augmentant[i + 1]
            ]
            if capacite_arete < capacite_mini:
                capacite_mini = capacite_arrete
                index_mini = i

        for i in range(len(chemin_augmentant) - 1):
            u = chemin_augmentant[i]
            v = chemin_augmentant[i + 1]
            flot[u][v] += capacite_mini
            capacites_actuelles[u][v] -= capacite_mini

            flot[v][u] -= capacite_mini
            capacites_actuelles[v][u] += capacite_mini
        chemin_augmentant = trouver_chemin_augmentant(
```

```
            graphe, noeud_depart, noeud_cible, capacites_actuelles
        )
    return flot


def trouver_aretes_a_couper(graphe, noeud_depart, noeud_cible):
    """
    Calcule les arêtes à couper pour isoler noeud_depart et noeud_cible
    """

    flot = ford_fulkerson(graphe, noeud_depart, noeud_cible)

    atteignable_depuis_depart = [False] * len(graphe)
    pile = [noeud_depart]
    atteignable_depuis_depart[noeud_depart] = True

    while len(pile) > 0:
        noeud_actuel = pile.pop()
        for voisin in graphe[noeud_actuel]:
            if (
                not atteignable_depuis_depart[voisin]
                and flot[noeud_actuel][voisin] != 1
            ):
                atteignable_depuis_depart[voisin] = True
                pile.append(voisin)

    aretes_a_couper = []
    for noeud in range(len(graphe)):
        if not atteignable_depuis_depart[noeud]:
            continue
        for voisin in graphe[noeud]:
            if not atteignable_depuis_depart[voisin]:
                aretes_a_couper.append((noeud, voisin))
    return aretes_a_couper


if __name__ == "__main__":
    n = int(input())
    m = int(input())

    graphe = [[] for _ in range(n)]

    for _ in range(m):
        a, b = map(int, input().split(" "))
        graphe[a].append(b)
        graphe[b].append(a)

    H = int(input())
    T = int(input())

    aretes_a_couper = trouver_aretes_a_couper(graphe, H, T)
    print(len(aretes_a_couper))
    for arete in aretes_a_couper:
```

```
122             print(f"{arete[0]} {arete[1]}")
```

			$(4,4)$
		$(3,3)$	$(3,4)$
	$(2,2)$	$(2,3)$	$(2,4)$
$(1,1)$	$(1,2)$	$(1,3)$	$(1,4)$

TABLE 1 – Le labyrinthe pour $n = 4$, les cases avec des coordonnées sont accessibles

5 Oracle de Delphes

5.1 Énoncé

Vous trouvez le concept de faire un concours d'informatique avec trois étapes et plusieurs centaines de participants ridicule pour déterminer un seul gagnant. Vous faites donc appel à l'oracle de Delphes pour connaître le meilleur participant qui gagnerait Prologin 2021.

Pour parvenir à l'oracle de Delphes, vous devez traverser en premier le labyrinthe de Minos, où le Minotaure veille que personne ne passe. Le labyrinthe est composé de n rangées et n colonnes, pour un total de n^2 chambres. Vous commencez à la chambre $(1, 1)$, et la sortie du labyrinthe se situe à la chambre (n, n). Heureusement, le Minotaure, après plusieurs millénaires souffre d'arthropathie chronique dégénérative et ne se déplace que quand son odorat détecte un intrus dans le labyrinthe. Doté des capacités divines de Posèidon, le Minotaure, une fois qu'il vous détecte peut se déplacer à une vitesse infinie et il vous sera impossible de vous échapper. Le Minotaure est dans la chambre $(n, 1)$ et sont odorat peut vous détecter si vous êtes dans la chambre (x, y) et $|x - n| + |y - 1| < n - 1$. De plus, craignant le regard cuisant d'Hélios, le dieu du soleil, vous ne pouvez seulement aller de la chambre (x, y) à la chambre $(x + 1, y)$ ou la chambre $(x, y + 1)$. Sinon, vous êtes cuits au sens littéral du terme.

Une fois passé à travers le labyrinthe de Minos, vous terminez assez aisément votre trajet. Arrivé à Delphes, la Pythie, depuis peu passionée d'algorithmique et de mathématiques, décide de vous aider pour votre quête de déterminer qui va gagner l'édition 2021 du concours Prologin à condition que vous l'aidiez pour un problème qu'elle tente de résoudre.

En effet, Pythie vous demande de calculer en premier le nombre de chemins k par lesquels vous auriez pu traverser le labyrinthe de Minos, puis ensuite $\sum_{1 \leq i \leq k} \sum_{1 \leq j \leq k} \operatorname{lcm}(i, j)$, où $\operatorname{lcm}(i, j)$ désigne le plus petit commun multiple des entiers positifs i et j.

Sauriez-vous répondre à Pythie pour pouvoir demander son oracle ?

5.2 Solution, trouver le nombre de chemins

Pour ce genre de problèmes, il est souvent utile de commencer par une visualisation. Dans la Table 1 (pour $n = 4$), on note les coordonnées de chaque case accessible, où on ne se fait pas repérer par le Minotaure.

5.2.1 Approche brute-force

Un algorithme récursif permet de facilement compter les chemins. Pour trouver l'Algorithme 1, on remarque qu'on peut passer aux cases $(x + 1, y)$ et $(x, y + 1)$ de la case (x, y).

Une implémentation typique de cet algorithme résulte aux temps d'éxécution de la table 1. On remarque que le temps d'éxécution explose !

Largeur du labyrinthe ($= n$)	2	3	4	5	6	7	8	9	10
Nombre de chemins ($= k$)	1	2	5	1.4×10^1	4.2×10^1	1.3×10^2	4.3×10^2	1.4×10^3	4.9×10^3
Temps d'éxécution en µs ($= 10^{-6}$ secondes)	0	0	0	0	1	5	1.7×10^1	4.4×10^1	1.3×10^2
	11	12	13	14	15	16	17	18	19
	1.7×10^4	5.9×10^4	2.1×10^5	7.4×10^5	2.7×10^6	9.7×10^6	3.5×10^7	1.3×10^8	4.8×10^8
	4.5×10^2	1.6×10^3	7.4×10^3	2.3×10^4	8.9×10^4	2.7×10^5	8.7×10^5	3.1×10^6	1.2×10^7
	20	21	22	23	24	25	26	27	28
	1.8×10^9	6.6×10^9	2.4×10^{10}	9.1×10^{10}	3.4×10^{11}	1.3×10^{12}	4.9×10^{12}	1.8×10^{13}	7.0×10^{13}
	4.6×10^7	1.7×10^8	6.6×10^8	?	?	?	?	?	?

FIGURE 1 – Temps d'éxécution

Algorithme 1 Compter les chemins (lent)

```
1 : procedure SlowCount(n, i = 1, j = 1)
2 :     if Cell (i, j) is not in the maze or the Minotaur can detect this cell then
3 :         return 0
4 :     else if i = n and j = n then
5 :         return 1
6 :     else
7 :         return SlowCount(n, i + 1, j) + SlowCount(n, i, j + 1)
8 :     end if
9 : end procedure
```

		$(3,3)$
	$(2,2)$	$(2,3)$
$(1,1)$	$(1,2)$	$(1,3)$

TABLE 2 – Le labyrinthe pour $n = 3$, les cases avec des coordonnées sont accessibles

Il est facile de voir que le temps de notre algorithme est de l'ordre de $O(n \cdot k)$ [1]. En effet, la longueur de chaque chemin est $O(n)$, et il y a k chemins. Déjà avec $n = 22$, on a besoin d'une dizaine de minutes. Un grossier calcul en utilisant la borne supérieur pour la complexité temporelle permet de trouver que le temps requis pour calculer k pour $n = 28$ est environ 3000 fois plus élevé, vers les 500 heures (soit environ 20 jours).

5.2.2 Un algorithme rapide

Bien que la phase de qualifications dure plus longtemps que 20 jours, la limite de temps est bien inférieure [2]. En fait, on remarque qu'on appelle la fonction plusieurs fois avec les mêmes arguments, on recalcule donc plusieurs fois le nombre de chemins de la case décrite par les arguments jusqu'à la sortie du labyrinthe, la case (n, n). Dans la Table 2 avec $n = 3$, on appelle la fonction avec les arguments $i = 2$ et $j = 3$ deux fois, une fois depuis le chemin $(1,1) \to (1,2) \to (1,3) \to (2,3) \to (3,3)$ et une fois depuis le chemin $(1,1) \to (1,2) \to (2,2) \to (2,3) \to (3,3)$.

Pour combler ce défaut, on peut simplement sauvegarder les calculs déjà faits, ce qu'on appelle *mémoization*. On obtient alors l'Algorithme 2.

Algorithme 2 Compter les chemins (rapide, récursif)

```
1 : procedure RecursiveCount(n, i = 1, j = 1)
2 :     if Cell (i, j) is not in the maze or the Minotaur can detect this cell then
3 :         return 0
4 :     else if i = n and j = n then
5 :         return 1
6 :     else
7 :         if a result for cell (i, j) is stored then
8 :             return the stored result
9 :         else
10 :            return SlowCount(n, i + 1, j) + SlowCount(n, i, j + 1) and store the result
11 :        end if
12 :    end if
13 : end procedure
```

Pour stocker les résultats partiels, on peut par exemple utiliser une table associative, ou en plus

1. Par un argument combinatoire, il est simple à voir que $k \leq 2^{2(n-1)}$ (des entiers avec 64 bits suffisent donc largement).

2. Pour la deuxième partie du problème, du hardcodage est nécessaire à notre connaissance. Cependant, ici le but est de trouver un algorithme qui résout la première partie sans hardcodage.

élégant simplement un tableau aux mêmes dimensions que le labyrinthe. Avec cet algorithme, on fait un appel récursif au plus une fois par case. Comme il y a n^2 cases et que le travail pour chaque case (si on ignore les appels récursifs) est constant, la complexité est $O(n^2)$. Mesurer le temps d'éxécution confirme ce résultat théorique, on a besoin de moins de une milliseconde pour calculer k pour $n = 28$. Même si l'algorithme d'avant est plus que suffisant pour cette partie, il est parfois utile d'exprimer un tel algorithme de manière itérative et non récursive. Cela permet de gratter du temps [3], de la mémoire [4] et d'avoir un algorithme qui est encore plus simple à écrire. Pour écrire cet algorithme en itératif, on observe que pour chaque case (x, y), il faut connaître la valeur des cases $(x + 1, y)$ et $(x, y + 1)$ avant. Avec un ordre de visite des cases astucieux, l'Algorithme 3 est essentiellement composé de deux boucles `for` !

Algorithme 3 Compter les chemins (rapide, itératif)

```
1 : procedure IterativeCount(n)
2 :     dp ← initialized by a n × n array containing 0's
3 :     dp[n, n] ← 1
4 :     for i = n, n − 1, ··· , 1 do
5 :         for j = n, n − 1, ··· , 1 do
6 :             if the Minotaur can not detect cell (i, j) then
7 :                 if cell (i + 1, j) is in the maze then
8 :                     dp[i, j] ← dp[i, j] + dp[i + 1, j]
9 :                 end if
10 :                if cell (i, j + 1) is in the maze then
11 :                    dp[i, j] ← dp[i, j] + dp[i, j + 1]
12 :                end if
13 :            end if
14 :        end for
15 :    end for
16 : end procedure
```

L'algorithme précédent (la version itérative et la version récursive) est un exemple d'un *algorithme dynamique*, un algorithme qui décompose le problème en sous-problèmes [5] et utilise une certaine structure d'optimalité pour les sous-problèmes [6] pour trouver le résultat. Des ressources utiles pour apprendre la programmation dynamique sont un article sur Codeforces, un article d'un des organisateurs de Prologin, et une série de vidéos. Pour la fin de cette partie, nous remarquons qu'il est possible d'exprimer cet algorithme de manière encore plus élégante, en itérant de 1 à n et en faisant quelques modifications nécessaires [7] (lesquelles ?).

5.2.3 Détour par les nombres de Catalan

Pour finir, regardons plus en détail la suite du nombre de chemins k en fonction de n. En tant que bon informaticien, un bon réflexe est de regarder si une telle suite est connue. Le site oeis permet de voir que la suite s'appelle la suite de Catalan, et qu'il existe beaucoup de manières équivalentes de la définir. Un exemple assez classique est le nombre de parenthésages valides de longueur $2n$ (voir Wikipedia par exemple).

On finit cette section par trouver une formule pour le nombre de chemins valides dans le labyrinthe (et donc aussi pour les nombres de Catalan).

3. En général seulement une constante. Ainsi, si la complexité temporelle de l'algorithme est trop mauvaise, passer en itératif ne sert à rien !

4. En général, seulement la mémoire nécessaire pour le *call stack*, qui peut être optimisée dans certains cas.

5. Ici, le nombre de chemins de chaque case (x, y) à (n, n).

6. Le nombre de chemins de la case (x, y) à (n, n) est le nombre de chemins de la case $(x+1, y)$ à (n, n) plus le nombre de chemins de la case $(x, y + 1)$ à la case (n, n).

7. Pour des raisons didactiques (suivre exactement ce que l'algorithme récursif fait), l'algorithme a été présenté en itérant de n à 1.

Lemme 5.1. *Le nombre de chemins valides dans le labyrinthe est* $\frac{1}{n}\binom{2(n-1)}{n-1}$.

Pour montrer cette formule, on commence par dénombrer une autre quantité.

Lemme 5.2. *Le nombre de chemins du coin gauche inférieur au coin opposé (droit supérieur) dans une grille de* $(n+1)\times(m+1)$ *cases, où pour chaque déplacement, les seuls mouvements possibles sont une case vers la droite ou une case vers le haut est* $\binom{n+m}{n}$.

Démonstration. Il se trouve qu'il y a exactement n déplacements vers le haut et m déplacements vers la droite à faire. Ainsi, il suffit de compter le nombre de manières de choisir n nombres différents parmi $\{1, 2, \cdots, n+m\}$, sans répétition et sans prendre en compte l'ordre. Le nombre de chemins est donc $\binom{n+m}{n}$ [8]. □

Maintenant, on peut compter le nombre de chemins valides dans le labyrinthe par le nombre de chemins total (en ignorant si le Minotaure peut repérer le héros), moins le nombre de chemins où le Minotaure peut repérer le héros. D'après le lemme intermédiaire, la première quantité est $\binom{2(n-1)}{n-1}$. L'astuce pour trouver le terme à soustraire est de construire une bijection entre ces chemins et notre problème précédent. En effet, on peut montrer qu'il y a une bijection entre ces chemins "invalides" et les chemins dans une grille $(n+1)\times(n-1)$. Ainsi la deuxième quantité est égale à $\binom{(n+1-1)+(n-1-1)}{n} = \binom{2(n-1)}{n}$. On trouve donc que le résultat est $\binom{2(n-1)}{n-1} - \binom{2(n-1)}{n}$.

On remarque qu'au lieu de l'algorithme précédent, on peut simplement évaluer deux coefficient binomiaux pour trouver le résultat ! Comment peut-on calculer ces coefficient ? Une manière consiste de calculer les factorielles dans la définition. Sans avoir accès à des entiers à taille dynamique, on obtient très rapidement un overflow. Aussi, si on désire calculer ce coefficient modulo un nombre, alors il est nécessaire de calculer l'inverse multiplicatif d'une factorielle, ce qui n'est pas forcément évident. De plus un tel inverse n'existe pas forcément. Une autre manière de calculer ces coefficient est d'exploiter la relation de Pascal, $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$. Calculer un coefficient binomial modulo un nombre est donc trivial. Calculer le coefficient de manière récursive avec cette relation nous donne un algorithme qui explose comme dans pour l'algorithme naïf. Comme pour les deux variations de l'algorithme dynamique d'avant, on peut facilement calculer un coefficient binomial $\binom{n}{k}$ en $O(nk)$.

On finit par utiliser l'approximation de Stirling pour avoir une idée de comment croît cette suite. Une variante de l'approximation de Stirling nous donne que $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + O\left(\frac{1}{n}\right)\right)$. En utilisant notre formule pour le nombre de chemins valides, on obtient donc que $k = \Theta\left(\frac{4^n}{n^{3/2}}\right)$.

5.3 Solution, calculer la somme des plus petits communs multiples

Cette partie a été inspirée par la correction d'un problème de Project Euler (accessible seulement après avoir résolu le problème [9]) et d'un exercice de Mathraining.be, jusqu'à ce que l'auteur découvre un article sur Codeforces qui donne directement un algorithme en $O\left(k^{3/4}\right)$ [10].

Nous sommes conscients qu'il existe aussi des algorithmes légèrement plus rapides, l'algorithme présenté dans le lien est cependant assez rapide pour pouvoir précalculer toutes les sommes dans l'espace d'au plus plusieurs jours.

Par manque de temps, cette section sera écrite au cours du temps, et l'algorithme présenté dans l'article de Codeforces et des autres solutions au problème seront présentés ici.

***

Félicitations à tous les participantes et participants !

8. De manière équivalente, on aurait pu raisonner avec m, et on aurait obtenu $\binom{n+m}{m}$ qui est bien égal au résultat puisque $\binom{n+m}{n} = \binom{n+m}{(n+m)-n} = \binom{n+m}{m}$.

9. indice : indicatrice d'Euler

10. Pour être exact, on utilise le modèle de calcul Real RAM Model avec des opérations arithmétiques de base en temps constant

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.