Cette épreuve est constituée de deux problèmes indépendants.

On rappelle que tous les programmes demandés doivent être rédigés dans le langage Python.

# Problème 1 : base 3 équilibrée

**Notation**. $\mathbb{N}^*$ désigne l'ensemble des entiers naturels non nuls.

En informatique, on utilise traditionnellement la base 2 pour représenter les entiers. Ce choix est dicté par la technologie utilisée par les ordinateurs actuels. Dans les années 60, d'autres technologies avaient été envisagées, dont l'utilisation d'une base de numération appelée base 3 équilibrée.

En base 3 classique, on écrit les nombres à l'aide de trois chiffres : 0, 1 et 2 : par exemple, $23 = \overline{212}^3 = 2 \times 3^2 + 1 \times 3 + 2$, $46 = \overline{1201}^3 = 1 \times 3^3 + 2 \times 3^2 + 0 \times 3 + 1$.

En base 3 équilibrée, les « chiffres » utilisés sont 0, 1 et $-1$ :

$23 = \overline{10(-1)(-1)}^e = 1 \times 3^3 + 0 \times 3^2 - 1 \times 3 - 1$, $46 = \overline{1(-1)(-1)01}^e = 1 \times 3^4 - 1 \times 3^3 - 1 \times 3^2 + 0 \times 3 + 1$.

Dans toute la suite, on s'intéresse à l'écriture en base 3 équilibrée des entiers naturels **non nuls**. On note $C = \{-1, 0, 1\}$ l'ensemble des chiffres utilisés dans l'écriture en base 3 équilibrée.

L'écriture en base 3 équilibrée d'un entier naturel **non nul** sera notée comme ci-dessus, par la suite des chiffres surmontée d'un trait avec un exposant $e$. Si $a_0$, $a_1$, …, $a_{p-1}$ sont dans $C$, on a donc :

$$\overline{a_0 a_1 \dots a_{p-1}}^e = \sum_{k=0}^{p-1} a_k 3^{p-1-k}.$$

L'écriture en base 3 équilibrée d'un entier naturel non nul, lue de gauche à droite, commence toujours par le chiffre 1.

## Partie I – généralités

**Question 1.**

**1.a** Donner la valeur de $\overline{1(-1)0(-1)}^e$, de $\overline{1111}^e$ et de $\overline{1(-1)(-1)(-1)(-1)}^e$.

$\overline{1(-1)0(-1)}^e = 27 - 9 - 1 = 17$, $\overline{1111}^e = 27 + 9 + 3 + 1 = 40$ et $\overline{1(-1)(-1)(-1)(-1)}^e = 81 - 27 - 9 - 3 - 1 = 41$.

**1.b** Écrire en base 3 équilibrée les entiers de 1 à 9.

$1 = \overline{1}^e$, $2 = \overline{1(-1)}^e$, $3 = \overline{10}^e$, $4 = \overline{11}^e$, $5 = \overline{1(-1)(-1)}^e$, $6 = \overline{1(-1)0}^e$, $7 = \overline{1(-1)1}^e$, $8 = \overline{10(-1)}^e$, $9 = \overline{100}^e$.

**Question 2.** Soit $k$ un entier naturel.

Déterminer la valeur de $A_k = \overline{1\underbrace{11\dots1}_{k \text{ chiffres}}}^e$ et de $B_k = \overline{1\underbrace{(-1)(-1)\dots(-1)}_{k \text{ chiffres}}}^e$.

(On aura bien noté que ces deux écritures possèdent $k+1$ chiffres au total.)

$A_k = \sum_{j=0}^{k} 3^j = \frac{3^{k+1}-1}{2}$ et $B_k = 3^k - A_{k-1} = 3^k - \frac{3^k-1}{2} = \frac{3^k+1}{2}$.

**Question 3.** On représente l'écriture en base 3 équilibrée $\overline{a_0 \dots a_{p-1}}^e$ par la liste Python $[a_{p-1}, a_{p-2}, \dots, a_0]$ (attention, les chiffres sont écrits dans la liste en lisant de droite à gauche l'écriture en base 3 équilibrée, le dernier d'entre eux est donc toujours le chiffre 1).

Écrire une fonction `valeur(L)` qui prend en argument une liste de chiffres et renvoie l'entier naturel non nul dont c'est une écriture en base 3 équilibrée.

Par exemple `valeur([1,0,-1,-1,1])` renvoie l'entier 46.

```python
def valeur(L):
    s = 0
    a = 1
    for k in range(len(L)):
        if L[k] == 1: s = s + a
        elif L[k] == -1: s = s - a
        a = 3*a
    return s
```

## Partie II – existence et unicité de l'écriture en base 3 équilibrée

**Question 4.** On veut montrer que tout entier naturel non nul $n$ admet une écriture en base 3 équilibrée.

Soit $n \in \mathbb{N}^*$. On note $q$ et $r$ le quotient et le reste dans la division euclidienne de $n$ par 3 : $n = 3q + r$ et $r \in \{0, 1, 2\}$.

**4.a** On suppose que $q$ est non nul et admet une écriture en base 3 équilibrée $q = \overline{a_0 \dots a_{p-1}}^e$. Déterminer une écriture en base 3 équilibrée de $n$ dans le cas où $r = 0$ ou $r = 1$.

$n = 3q + r = 3\overline{a_0 \dots a_{p-1}}^e + r = \overline{a_0 \dots a_{p-1}0}^e + r = \overline{a_0 \dots a_{p-1}r}^e$.

**4.b** On suppose que $q$ est non nul et que $q + 1$ admet une écriture en base 3 équilibrée $q + 1 = \overline{b_0 \dots b_{p-1}}^e$. Déterminer une écriture en base 3 équilibrée de $n$ dans le cas où $r = 2$.

$n = 3q + 2 = 3(q + 1) - 1 = 3\overline{b_0 \dots b_{p-1}}^e - 1 = \overline{b_0 \dots b_{p-1}0}^e - 1 = \overline{b_0 \dots b_{p-1}(-1)}^e$.

**4.c** Montrer par récurrence que tout entier naturel non nul admet une écriture en base 3 équilibrée.

Soit $\mathscr{P}_n$ la propriété : tout entier naturel au plus égal à $n$ admet une écriture en base 3 équilibrée. La question 1.b montre que $\mathscr{P}_9$ est vraie.

Soit $n \geqslant 10$. On suppose $\mathscr{P}_{n-1}$ vraie, on veut montrer $\mathscr{P}_n$ est vraie, c'est-à-dire que $n$ admet également une écriture en base 3 équilibrée. On effectue la division euclidienne par 3, comme suggéré par l'énoncé : $n = 3q + r$ avec $r \in \{0, 1, 2\}$.

On a alors $q < q + 1 = \dfrac{n - r}{3} + 1 \leqslant \dfrac{n}{3} + 1 \leqslant n - 1$ puisque $(n - 1) - \left(\dfrac{n}{3} + 1\right) = \dfrac{2n}{3} - 2 \geqslant \dfrac{20}{3} - 2 \geqslant 0$. D'après l'hypothèse de récurrence, $q$ et $q + 1$ admettent une écriture en base 3 équilibrée, et les questions 4.a et 4.b montrent qu'il en est de même pour $n$.

La récurrence s'enclenche.

**Question 5.** On souhaite écrire en Python une fonction incremente(L) qui prend en argument une liste L de chiffres représentant une écriture en base 3 équilibrée d'un entier naturel non nul $n$ et qui renvoie une liste représentant une écriture en base 3 équilibrée de $n + 1$.

**5.a** On propose la fonction récursive incrementeR(L) suivante.

Expliquer la ligne 9. Pourquoi a-t-il fallu écrire les lignes 1 et 2 ?

```python
0 def incrementeR(L):
1     if len(L)==0:
2         return [1]
3     elif L[0]==0:
4         return [1]+L[1:]
5     elif L[0]==-1:
6         return [0]+L[1:]
7     else:
8         # ici L[0]==1
9         return [-1]+incrementeR(L[1:])
```

On est dans le cas où $n = 1 + 3q$ où $q$ admet une écriture en base 3 équilibrée représentée par la liste L[1:]. Alors $n + 1 = 2 + 3q = -1 + 3(q + 1)$ : le chiffre de poids faible est $-1$ et la suite de l'écriture est celle de $q + 1$.

Dans le cas où L est [1,1,...,1] le dernier appel récursif se fera avec une liste vide en argument : c'est cela qui impose d'écrire les lignes 1 et 2.

**5.b** On souhaite écrire une version non récursive incremente(L) de cette fonction.

Recopier et compléter le script suivant pour répondre à cette question :

```
def incremente(L):
    p = len(L)
    M = [] # liste finale
    k = 0
    while k < p and L[k] == 1:
        M.append(-1)
        k += 1
    if k == p:
        M.append(...)
    elif L[k] == 0:
        M.append(...)
        M = M + L[k+1:]
    elif L[k] == -1:
        ...
        ...
    return M
```

```
def incremente(L):
    p = len(L)
    M = [] # liste finale
    k = 0
    while k < p and L[k] == 1:
        M.append(-1)
        k += 1
    if k == p:
        M.append(1)
    elif L[k] == 0:
        M.append(1)
        M = M + L[k+1:]
    elif L[k] == -1:
        M.append(0)
        M = M + L[k+1:]
    return M
```

**Question 6.** On souhaite montrer que l'écriture en base 3 équilibrée d'un entier naturel $n$ non nul est unique.

**6.a** Soit $a_0, \dots, a_{p-1}$ des chiffres de $C$ (avec $a_0 = 1$ et $p \geqslant 1$). Quel est le reste dans la division euclidienne par 3 du nombre $\overline{a_0 a_1 \dots a_{p-1}}^e$ ?

$\overline{a_0 \dots a_{p-1}}^e = \displaystyle\sum_{k=0}^{p-1} a_k 3^{p-1-k} = a_{p-1} + 3\sum_{k=0}^{p-2} a_k 3^{p-2-k} \equiv a_{p-1}[3]$. Donc le reste dans la division euclidienne cherché est égal à $a_{p-1}$ si $a_{p-1} \in \{0,1\}$ et à 2 si $a_{p-1} = -1$.

**6.b** En déduire que l'écriture en base 3 équilibrée d'un entier naturel $n$ non nul est unique.

On raisonne par récurrence sur $n$.

D'après la question 2, pour $k \geqslant 1$, $B_k \geqslant \dfrac{3+1}{2} = 2$ donc toute écriture comportant au moins 2 chiffres est celle d'un nombre au moins égal à 2. Ainsi $n = 1$ n'a qu'une unique écriture : $1 = \overline{1}^e$.

Soit $n \geqslant 1$. On suppose qu'on a prouvé que tout entier non nul au plus égal à $n$ admet une unique écriture en base 3 équilibrée, et on veut montrer qu'il en est de même de $n+1$.

La question 6.a montre qu'une écriture en base 3 équilibrée de $n+1$ se termine nécessairement par le chiffre $a_p = 0$, 1 ou $-1$ selon le reste dans la division de $n+1$ par 3. Mais alors les chiffres précédents sont ceux d'une écriture d'un entier $n' = \dfrac{n+1-a_p}{3} \leqslant \dfrac{n+2}{3} \leqslant n$ : il y a unicité par hypothèse de récurrence.

La récurrence s'enclenche donc bien.

**Question 7.** Écrire une fonction `base3e(n)` qui prend en argument un entier naturel $n$ non nul et renvoie la liste `L` de chiffres qui correspond à l'écriture en base 3 équilibrée de $n$. Par exemple `base3e(46)` renvoie `[1,0,-1,-1,1]`).

On propose la fonction suivante.

```
def base3e(n):
    M = [] # liste finale
    while n>0:
        r = n % 3 # reste dans la division par 3
        if r<2:
            M.append(r)
            n = n // 3
        else:      # cas r=2
            M.append(-1)
            n = (n+1) // 3
    return M
```

## Partie III – chemins de Delannoy

On considère le plan euclidien $P$ rapporté à un repère orthonormé.

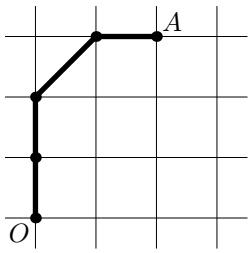
Un chemin de Delannoy du plan $P$ est un arc suivant une ligne brisée partant de l'origine $O$ et arrivant en un point $A$ de coordonnées $(a, b)$ où $a$ et $b$ sont deux entiers naturels, constituée d'arcs à choisir parmi trois types : un arc horizontal (associé au vecteur $(1, 0)$), un arc vertical (associé au vecteur $(0, 1)$) ou un arc diagonal (associé au vecteur $(1, 1)$). Dans le parcours d'un chemin de Delannoy, on va donc toujours vers la droite, vers le haut, ou en diagonale vers la droite et vers le haut en même temps.

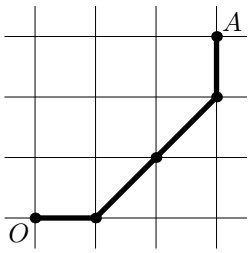
On associe à tout chemin de Delannoy un entier naturel non nul $n$ défini par son écriture en base 3 équilibrée de la façon suivante : le premier chiffre de cette écriture est toujours égal à 1 (comme pour toute écriture en base 3 équilibrée), les chiffres suivants caractérisent les arcs consécutifs du chemin, le chiffre 1 est associé à un segment horizontal, le chiffre $-1$ à un arc vertical, et le chiffre 0 à un arc diagonal. Inversement à tout entier naturel non nul on peut associer le chemin de Delannoy qui est associé à son écriture en base 3 équilibrée.

On fera attention que le premier chiffre de l'écriture en base 3 équilibrée, toujours égal à 1, n'est pas pris en compte dans la construction du chemin.

Remarque : $n = 1 = \overline{1}^e$ est l'entier associé à un chemin de Delannoy de longueur nulle.

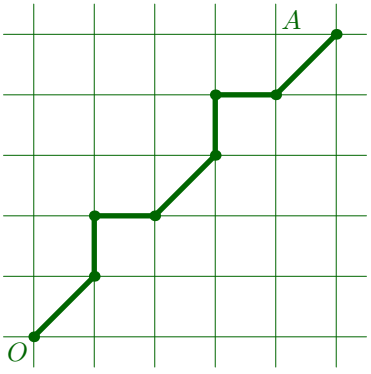
Par exemple, les entiers associés aux deux chemins de Delannoy ci-dessous sont respectivement $46 = \overline{1(-1)(-1)01}^e$ pour la figure de gauche et $107 = \overline{1100(-1)}^e$ pour la figure de droite.




**Question 8.**

**8.a** Dessiner le chemin de Delannoy associé à l'entier 2019.



**8.b** On note $(a, b)$ les coordonnées entières du point d'arrivée d'un chemin de Delannoy. Écrire la fonction `arrivee(n)` qui renvoie ce couple de coordonnées quand on lui donne en argument l'entier non nul $n$ associé au chemin de Delannoy. Par exemple `arrivee(46)` renvoie le couple `(2, 3)` et `arrivee(107)` renvoie le couple `(3, 3)`.

On pourra s'inspirer de la fonction `base3e`.

```
def arrivee(n):
    (a,b) = (0,0)
    while n>1:  # on ne tient pas compte du premier chiffre (toujours 1)
        r = n % 3
        if r == 0:
            (a,b) = (a+1,b+1)
            n = n // 3
        elif r == 1:
            (a,b) = (a+1,b)
            n = n // 3
        else:  # cas r=2
            (a,b) = (a,b+1)
            n = (n+1) // 3
    return (a,b)
```

**Question 9.**

Soit $a$ et $b$ deux entiers naturels, et $A$ le point de coordonnées $(a, b)$. Il existe plusieurs chemins de Delannoy allant de l'origine au point $A$, chacun d'eux étant associé à un entier naturel. Soit $N(a, b)$ l'ensemble de ces entiers.

**9.a** Soit $a \in \mathbb{N}^*$. Déterminer $\min N(a, a)$ et $\max N(a, a)$.

On a vu à la question 2 que les entiers dont l'écriture en base 3 équilibrée comporte exactement $k$ chiffres après le 1 initial sont compris au sens large entre $B_k = \dfrac{3^k+1}{2}$ et $A_k = \dfrac{3^{k+1}-1}{2}$. Comme $B_{k+1} = A_k + 1$, on peut en déduire que l'ensemble des entiers dont l'écriture en base 3 équilibrée comporte exactement $k$ chiffres après le 1 initial sont exactement les entiers de l'intervalle $\left[\!\left[\dfrac{3^k+1}{2}, \dfrac{3^{k+1}-1}{2}\right]\!\right]$. (On peut aussi observer qu'avec $k$ chiffres après le 1 initial, on obtient $3^k$ nombres différents, grâce à l'unicité de l'écriture, et que justement $A_k - B_k + 1 = 3^k$.)

Soit $A(a, a)$. Il existe un unique chemin de $O$ à $A$ composé de $a$ arcs : tous ses arcs sont du type diagonal. Il correspond à l'entier $n = 1\underbrace{0\ldots0}_{a \text{ zéros}} = 3^a$. Tout autre chemin de $O$ à $A$ comporte au moins $a + 1$ arcs et correspond donc à un entier supérieur. Ainsi $\min N(a, a) = 3^a$.

On vérifie facilement que quels que soient les chiffres représentés par $*$ :

$$\overline{1\underbrace{*\ldots*}_{p \text{ chiffres}}}^e \geqslant 3^p > \overline{0\underbrace{*\ldots*}_{p \text{ chiffres}}}^e \geqslant \overline{0\underbrace{-1\ldots-1}_{p \text{ chiffres}}}^e = \frac{-3^p+1}{2} > \overline{-1\underbrace{*\ldots*}_{p \text{ chiffres}}}^e.$$

Ainsi $\max N(a, a)$ sera obtenu en maximisant le nombre de chiffres 1 en début d'écriture en base 3 équilibrée : on obtient donc

$$\max N(a, a) = \overline{1\underbrace{11\ldots1}_{a \text{ chiffres}}\underbrace{-1-1\ldots-1}_{a \text{ chiffres}}}^e = 3^{2a} + \frac{(3^a-1)^2}{2}.$$

**9.b** On suppose que les entiers $a$ et $b$ vérifient $0 < a < b$. Déterminer $\min N(a, b)$ et $\max N(a, b)$.

Un raisonnement analogue conduit à

$$\min N(a, b) = \overline{1\underbrace{-1-1\ldots-1}_{(b-a) \text{ chiffres}}\underbrace{00\ldots0}_{a \text{ chiffres}}}^e = \frac{3^a+3^b}{2},$$

$$\max N(a, b) = \overline{1\underbrace{11\ldots1}_{(a \text{ chiffres}}\underbrace{-1-1\ldots-1}_{b \text{ chiffres}}}^e = \frac{3^{a+b+1} - 2\,.\,3^b + 1}{2}.$$

# Problème 2 : compilation et algorithmes

**Notation**. Pour $m$ et $n$ deux entiers naturels, $[\![m, n]\!]$ désigne l'ensemble des entiers $k$ tels que $m \leqslant k \leqslant n$.

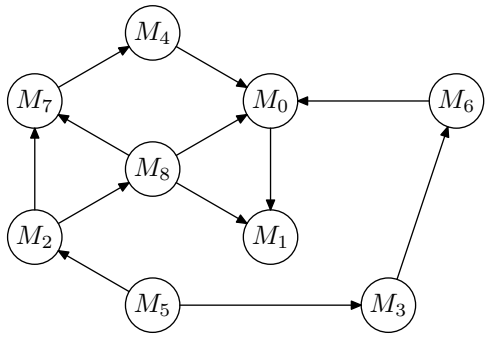
## Partie A – ordre topologique sur un graphe

Lors de l'écriture d'un programme complexe, on le découpe souvent en modules distincts, mais dépendants les uns des autres. Le compilateur a besoin de connaître l'ordre dans lequel compiler ces modules. Pour ce faire, on utilise un graphe de dépendances, que nous décrivons brièvement ici.

On suppose qu'on dispose de $n$ modules numérotés $M_0$, $M_1$, …, $M_{n-1}$ et on représente leurs relations de dépendance à l'aide d'un graphe $\Gamma$ dont les sommets sont les modules, et dont une arête allant du module $M_i$ au module $M_j$ indique que le module $M_i$ doit être compilé avant le module $M_j$.

Par exemple, avec $n = 9$, aux contraintes du tableau de gauche ci-dessous, on peut associer le graphe $\Gamma$ de droite.

| le module | est compilé APRÈS les modules |
|---|---|
| $M_0$ | $M_4$, $M_6$, $M_8$ |
| $M_1$ | $M_0$, $M_8$ |
| $M_2$ | $M_5$ |
| $M_3$ | $M_5$ |
| $M_4$ | $M_7$ |
| $M_5$ | – |
| $M_6$ | $M_3$ |
| $M_7$ | $M_2$, $M_8$ |
| $M_8$ | $M_2$ |

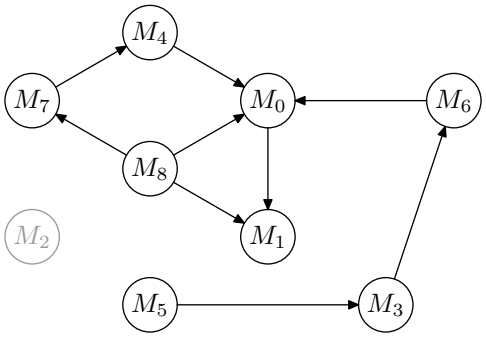



On aura besoin de travailler sur des sous-graphes $G$ du graphe $\Gamma$. C'est pourquoi, pour toute la suite, on suppose fixé l'entier $n$ (qu'on pourra supposer supérieur à 2), et les graphes considérés n'utiliseront que quelques-uns des $n$ sommets $M_0$, $M_1$, …, $M_{n-1}$. Pour simplifier la rédaction, on pourra utiliser la notation suivante : si $s$ est un sommet d'un graphe $G$, on note $[s]$ son numéro, de sorte que $s = M_{[s]}$.

On représente un graphe $G$ dont les sommets sont pris parmi les $n$ sommets listés ci-dessus par une liste G=[App,Succ,p] où App est une liste de $n$ valeurs booléennes (c'est-à-dire True ou False) ; Succ est une liste de $n$ listes ; $p$ est un entier. Plus précisément :

- pour $i \in [\![0, n-1]\!]$, App[i] est True si et seulement si le module $M_i$ est présent dans le graphe ;
- $p$ est le nombre de sommets du graphe considéré, donc le nombre d'occurrences de True dans la liste App ;
- pour $i \in [\![0, n-1]\!]$, Succ[i] est la liste vide si $M_i$ n'est pas présent dans le graphe, et sinon Succ[i] est la liste (éventuellement vide) des numéros $j$ des sommets $M_j$ tels qu'il y a un arc de $M_i$ vers $M_j$.

Par exemple, le graphe $G$ de la figure suivante (obtenu en supprimant dans $\Gamma$ le sommet $M_2$) :




est représenté par la liste G=[App,Succ,p] où

- App = [True,True,False,True,True,True,True,True,True] ;
- Succ = [ [ 1 ], [], [], [ 6 ], [ 0 ], [ 3 ], [ 0 ], [ 4 ], [ 0, 1, 7 ] ] ;
- p=8.

Dans toute la suite, on suppose que la valeur $n$, qui est fixée, est stockée dans une variable globale n. Les variables i et j désignent des numéros de sommets, donc des entiers de l'intervalle $[\![0, n-1]\!]$.

**Question 10.**

Écrire une fonction mem(i,G) qui prend en argument un entier $i$ et la représentation G d'un graphe (telle qu'elle est décrite ci-dessus) et renvoie le booléen indiquant si le sommet de numéro $i$ est dans le graphe.

```
def mem(i,G):
  return G[0][i]
```

**Question 11.**

Écrire une fonction pred(i,G) qui renvoie la liste, éventuellement vide, des numéros $j$ des sommets du graphe représenté par G tels que l'arc qui va de $M_j$ à $M_i$ soit présent dans le graphe.

Par exemple, pour le graphe de la figure ci-dessus, pred(0,G) renvoie la liste [ 4, 6, 8 ].

```
def pred(i,G):
  predecesseurs = []
  for j in range(n):
    if i in G[1][j]:
      predecesseurs.append(j)
  return predecesseurs
```

**Question 12.**

Écrire une fonction sansSuccesseur(G) qui renvoie le numéro $i$ d'un sommet $M_i$ du graphe qui n'a pas de successeur, s'il en existe, ou qui renvoie $-1$ s'il n'y a pas de tel sommet. Dans l'exemple du graphe ci-dessus, sansSuccesseur(G) renvoie 1.

```
def sansSuccesseur(G):
    i = 0
    while i < n and (len(G[1][i]) > 0 or not G[0][i]):
        i += 1
    if i < n:
        return i
    else:
        return -1
```

**Question 13.**

Compléter la fonction suivante suppr(i,G) qui renvoie **un nouveau graphe** H obtenu à partir de G en supprimant le sommet de numéro i et toutes les flèches qui lui sont reliées.

```
def suppr(i,G):
    H = copy.deepcopy(G)
    H[2] = H[2] - 1
    ...
     .
     .
     .
    ...
    return H
```

L'appel H = copy.deepcopy(G) permet d'instancier la variable H avec une copie de G.

```
def suppr(i,G):
    H = copy.deepcopy(G)
    H[2] = H[2] - 1
    H[0][i] = False
    H[1][i] = []
    for j in range(n):
        if i in H[1][j]: H[1][j].remove(i)
    return H
```

Étant donné un graphe de dépendances $G$ possédant $p$ sommets (avec $p \leqslant n$), un ordre topologique sur le graphe est la donnée d'une renumérotation des sommets du graphe qui respecte les arcs du graphe, c'est-à-dire une fonction $N$ bijective, définie sur l'ensemble des numéros des sommets présents dans le graphe $G$, à valeurs dans $[\![0, p-1]\!]$ telle que s'il existe un arc du sommet $M_i$ vers le sommet $M_j$ on a $N(i) < N(j)$.

Par exemple, pour le graphe de la dernière figure ci-dessus, il existe un ordre topologique, défini par $N(0) = 6$, $N(1) = 7$, $N(3) = 1$, $N(4) = 5$, $N(5) = 0$, $N(6) = 2$, $N(7) = 4$ et $N(8) = 3$.

Un chemin dans un graphe est une suite d'au moins deux sommets reliés par des arcs consécutifs. Un cycle est un chemin qui retourne au sommet de départ.

**Question 14.**

Dans cette question, on considère un graphe orienté $G$ possédant $p$ sommets (avec $p \leqslant n$) qui sont pris parmi $M_0$, $M_1$, …, $M_{n-1}$. On suppose que chaque sommet admet au moins un successeur.

**14.a** On effectue une promenade dans le graphe : on choisit un sommet $s_0$, puis un successeur $s_1$ de $s_0$, puis un successeur $s_2$ de $s_1$, etc. On peut ainsi obtenir une suite $s_0$, $s_1$, …, $s_n$. Montrer qu'il existe deux indices $i$ et $j$ tels que $0 \leqslant i < j \leqslant n$ et $s_i = s_j$.

Les $s_i$ forment une famille de $n+1$ objets à choisir parmi les $n$ sommets : au moins deux d'entre eux sont égaux.

**14.b** Justifier qu'il existe au moins un cycle dans ce graphe.

Avec les notations de l'énoncé, $(s_i, s_{i+1}, \ldots, s_j = s_i)$ forme un cycle dans le graphe.

**14.c** Montrer qu'il est impossible de trouver un ordre topologique sur ce graphe.

S'il existait un ordre topologique $N$ sur ce graphe, en notant $[s_i]$ le numéro du sommet $s_i$, on aurait

$$N([s_i]) < N([s_{i+1}]) < \cdots < N([s_{j-1}]) < N([s_j]) = N([s_i]),$$

ce qui est absurde.

**Question 15.**

Soit $G$ un graphe ayant $p$ sommets (avec $p \leqslant n$) parmi $M_0$, …, $M_{n-1}$. On suppose que $G$ admet au moins un sommet $s$ sans successeur. On appelle $H$ le graphe obtenu à partir de $G$ en supprimant le sommet $s$ et tous les arcs reliés à $s$. Montrer que si $H$ possède un ordre topologique $N_H$ à valeurs dans $[\![0, p-2]\!]$, on peut l'étendre à un ordre topologique $N$ sur $G$ en posant $N([s]) = p-1$ et $N([s']) = N_H([s'])$ pour tous les sommets $s'$ de $H$.

Que dire ? c'est assez évident.

**Question 16.**

On peut déduire de cette étude un algorithme de détermination d'un ordre topologique $N$ sur un graphe $G$, quand il en existe. Un tel ordre topologique est représenté par une liste `L` à $n$ éléments : si le sommet $M_i$ n'est pas dans le graphe $G$, on a `L[i]` $= -1$ ; si $M_i$ est dans $G$, `L[i]` contient la valeur $N(i)$.

**16.a** Recopier et compléter la fonction suivante pour répondre à la question : `ordreTopologique(G)` renvoie `None` s'il n'existe pas d'ordre topologique sur le graphe `G` et une liste `L` qui représente un ordre topologique s'il en existe un. La fonction `parcoursReussi` peut procéder à des appels récursifs.

```
def ordreTopologique(G):
    n = len(G[0])
    L = [-1] * n

    def parcoursReussi(G):
        p = G[2]
        if p != 0:
            s = sansSuccesseur(G)
            if s == -1:
                return ...
            else:
                ...
                ...
                return ...
        else:
```

```
            return ...


    b = parcoursReussi(G)
    if b:
        return L
    else:
        return None
```

```
def ordreTopologique(G):
    n = len(G[0])
    L = [-1] * n

    def parcoursReussi(G):
        p = G[2]
        if p != 0:
            s = sansSuccesseur(G)
            if s == -1:
                return False
            else:
                H = suppr(s,G)
                L[s] = p-1
                return parcoursReussi(H)
        else:
            return True


    b = parcoursReussi(G)
    if b:
        return L
    else:
        return None
```

**16.b** Prouver que l'algorithme termine toujours.

Pour prouver la terminaison, il suffit d'observer que la taille (p=G[2]) du graphe passé en argument à parcoursReussi décroît strictement à chaque appel récursif, et qu'il n'y a pas d'appel récursif quand elle descend à 0.

## Partie B – allocation de registres

Quand le code est en phase finale de compilation, on peut supposer, pour simplifier, qu'il se présente comme une succession d'affectations de variables ou d'utilisation de ces variables à l'aide de fonctions simples (opérations arithmétiques ou logiques). L'accès à la mémoire étant plus coûteux en temps que le calcul lui-même, il est préférable de conserver le plus possible de résultats intermédiaires dans les registres du processeur plutôt que dans la mémoire vive. Mais le processeur n'a qu'un nombre limité de registres : il s'agit donc de savoir si tous les calculs peuvent être menés en se confinant à cet espace limité.

On se limite ici à un modèle extrêmement simplifié, où le code du programme ne comporte que des affectations de variables x = ... ou des appels de fonctions f(a,b ,...) , comme le code Python suivant :

```
0 d = 1
1 b = 2 * d
2 a = 3
3 d = a * b
4 print(d)
5 c = 2 * a - b
6 print(a)
7 d = c + b
8 b = 2 * a
9 print(c,d)
```

Comme un programme ne peut utiliser qu'un nombre fini de variables, on suppose dans toute la suite qu'on dispose d'une numérotation des variables, ce qui permet de représenter une variable par un entier naturel. Dans l'exemple ci-dessus, on supposera que a est numérotée 0, b est numérotée 1, etc.

Une affectation de variable est représentée par un couple (i, [j,k,...]) constitué du numéro de la variable qui figure à gauche du symbole = et de la liste des numéros des variables qui figurent à droite. Par exemple, l'affectation c = 2 * a − b est représentée par le couple (2, [0,1]).

Un appel de fonction est représenté par un couple (None, [j,k,...]) dont le deuxième élément est la liste des numéros des variables utilisées dans l'appel de fonction. Par exemple, l'appel **print**(c,d) est représenté par le couple (None, [2,3]).

Les constantes apparaissant dans les différentes instructions ne sont pas prises en compte dans cette représentation.

Le programme précédent est ainsi représenté par la liste de couples suivante :

prog = [(3,[]), (1,[3]), (0,[]), (3,[0,1]), (None,[3]), (2,[0,1]), (None,[0]), (3,[1,2]), (1,[0]), (None,[2,3])]

On cherche à savoir à quels moments il est utile de conserver dans les registres du processeur les valeurs des différentes variables.

Dans l'exemple du programme ci-dessus, si on s'intéresse à la variable d, on observe que :
- elle est initialisée en ligne 0, et cette valeur est utilisée en ligne 1 ;
- elle est modifiée en ligne 3, et la nouvelle valeur est utilisée en ligne 4 ;
- elle est modifiée en ligne 7, et la nouvelle valeur est utilisée en ligne 9.

La variable d a donc trois périodes de vie, qu'on peut représenter par les intervalles $]0,1]$, $]3,4]$ et $]7,9]$.

La variable a a une seule période de vie : l'intervalle $]2,8]$ ; la variable b n'a également qu'une période de vie : l'intervalle $]1,7]$. En effet, b est modifiée en ligne 8 mais sa valeur n'est pas utilisée dans la suite, donc on ne tient pas compte de l'intervalle $]8,9]$.

On dit qu'une variable est vivante pour chaque instruction d'une période de vie, et morte pour les autres.

L'algorithme suivant permet de déterminer pour chaque ligne de code si chaque variable est morte ou vivante.

- on commence par indiquer que chaque variable est morte ;
- pour chaque ligne de code, **en partant du bas** :
  - s'il s'agit d'un appel de fonction, chaque variable utilisée est marquée vivante ;
  - s'il s'agit d'une affectation, la variable affectée est marquée morte sauf si elle figure également à droite du symbole =, et toutes les variables à droite du symbole = sont marquées vivantes.

**Question 17.**

Recopier et compléter les lignes 0 à 4 du tableau suivant qui indique l'état de chaque variable pour chaque instruction du programme ci-dessus. (Un V indique une variable vivante, un M une variable morte.)

| ligne | a | b | c | d |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | V | V | M | M |
| 6 | V | V | V | M |
| 7 | V | V | V | M |
| 8 | V | M | V | V |
| 9 | M | M | V | V |

| ligne | a | b | c | d |
|---|---|---|---|---|
| 0 | M | M | M | M |
| 1 | M | M | M | V |
| 2 | M | V | M | M |
| 3 | V | V | M | M |
| 4 | V | V | M | V |
| 5 | V | V | M | M |
| 6 | V | V | V | M |
| 7 | V | V | V | M |
| 8 | V | M | V | V |
| 9 | M | M | V | V |

**Question 18.** Comment interpréter le cas où en ligne 0 du tableau on trouve une variable vivante ? Le programme est-il exécutable ?

Si une variable est vivante en ligne 0, c'est que sa valeur est nécessaire pour un calcul, alors qu'elle n'a pas été initialisée : le programme n'est donc pas exécutable.

C'est par exemple le cas d'un programme qui commencerait par l'instruction **print**(b,c) alors que les deux variables n'ont jamais été affectées.

**Question 19.**

Pour un programme comportant $n$ instructions et utilisant $p$ variables numérotées de 0 à $p-1$, le tableau qui indique l'état de chaque variable pour chaque instruction peut être représenté par une liste de listes vie telle que pour la ligne $i$ (avec $0 \leqslant i < n$) et la variable $v$ (avec $0 \leqslant v < p$), vie[i][v] est True si la variable est vivante et False si la variable est morte.

Écrire une fonction determineVie(prog,p) qui prend en argument un programme et le nombre $p$ de variables à considérer et qui renvoie le tableau vie correspondant.

```
def determineVie(prog,p):
    n = len(prog)
    vie = [ [False]*p for i in range(n) ]
    for j in range(n):
        i = n - 1 - j # i varie de n-1 a 0
        v,args = prog[i]
        if i < n-1:
            for j in range(p):
                vie[i][j] = vie[i+1][j] # on recopie l'etat courant des variables
        if v != None:
            vie[i][v] = False
        for x in args :
            vie[i][x] = True
    return vie
```

**Question 20.**

Recopier et compléter la fonction suivante periodesVie(vie,v) qui prend en arguments la table vie obtenue par la fonction determineVie et le numéro d'une variable et qui renvoie la liste des périodes de vie de cette variable. Ainsi, pour l'exemple du programme considéré, et la variable d, la fonction periodesVie renvoie la liste [(7, 9), (3, 4), (0, 1)] (l'ordre n'a pas d'importance).

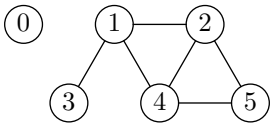
```
def periodesVie(vie,v):
    n = len(vie)
    periodes = []
    i = n-1
    encours = False
    while i>=0:
        if vie[i][v]:
            if not encours:
                ...
                ...
        else:
            if ...:
                ...
                ...
        i -= 1
    return periodes
```

```
def periodesVie(vie,v):
    n = len(vie)
    periodes = []
    i = n-1
    encours = False
    while i>=0:
        if vie[i][v]:
            if not encours:
                encours = True
                fin = i
        else:
            if encours:
                encours = False
                periodes.append((i,fin))
        i -= 1
    return periodes
```

## Partie C – graphe de coexistence

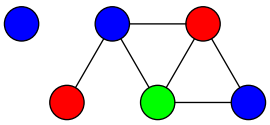
Des techniques analogues à celle présentée dans la partie B permettent de construire un graphe de coexistence des variables occupant des registres. Il s'agit d'un graphe non orienté tel que deux sommets reliés dans ce graphe correspondent à deux variables vivantes au même moment, ce qui oblige à les ranger dans des registres différents.

La figure ci-dessous montre un exemple de graphe de coexistence : les variables numérotées 1, 2, 4 doivent être rangées dans trois registres distincts. Mais les variables numérotées 1 et 5 peuvent être rangées dans le même registre, puisque les sommets correspondants ne sont pas reliés.




Pour déterminer le nombre de registres nécessaire, il suffit de colorier le graphe de sorte que deux sommets adjacents soient de couleurs différentes, en utilisant le moins de couleurs possible. Ce nombre minimal de couleur est le nombre cherché de registres.

Dans l'exemple du graphe ci-dessus, trois couleurs suffisent.



**Question 21.**

Le problème général de la détermination du nombre minimal de couleurs nécessaire à la coloration d'un graphe est connu pour être un problème $NP$-complet. Que signifie l'expression « ce problème est dans la classe $NP$ » ? et que signifie l'adjonction de l'adjectif « complet » ?

Un problème est dans la classe $NP$ s'il est décidé par une machine de Turing non déterministe en temps polynomial. Autrement dit, on peut tester une proposition de solution et dire en temps polynomial si elle est réellement ou non solution. Par exemple, pour la coloration d'un graphe, étant donné un graphe colorié il est facile et rapide de vérifier que deux sommets adjacents n'ont pas la même couleur.

Un problème de la classe $NP$ est $NP$-complet si tout problème $NP$ se ramène à celui-ci en temps polynomial. Autrement dit, il n'y a pas de problème plus difficile que lui. Ces réductions d'un problème à un autre sont souvent difficiles à trouver !

Dans la suite, un graphe non orienté possédant $n$ sommets numérotés de 0 à $n-1$ est représenté par une liste `G` de $n$ listes : `G[i]` est la liste (éventuellement vide) des sommets reliés au sommet `i`. Autrement dit, `G[i]` est la liste des voisins du sommet `i`.

L'exemple du graphe précédent est représenté par la liste `G=[ [], [2,3,4], [1,4,5], [1], [1,2,5], [2,4] ]`.

Une coloration du graphe est représentée par une liste donnant le numéro de la couleur de chaque sommet. Dans l'exemple ci-dessus, si bleu est numéro 0, rouge numéro 1 et vert numéro 2, la coloration proposée est représentée par la liste couleur = [0,0,1,1,2,0] .

**Question 22.**

Écrire une fonction bonnesCouleurs(G,couleur) qui prend en argument un graphe G et une coloration couleur de ses sommets et qui renvoie True si et seulement si la coloration est correcte, c'est-à-dire que deux sommets adjacents n'ont jamais la même couleur.

```
def bonnesCouleurs(G,couleur):
    n = len(G)
    for i in range(n):
        c = couleur[i]
        for voisin in G[i]:
            if c == couleur[voisin]:
                return False
    return True
```

**Question 23.**

Il est facile d'écrire un algorithme qui permet de déterminer une coloration correcte d'un graphe, mais sans garantir qu'on utilise le nombre minimal de couleurs.

Par exemple, en partant d'un graphe dont aucun sommet n'est colorié.

1. choisir la première couleur disponible ;
2. tant qu'il existe un sommet non colorié répéter les étapes 3 à 6 ;
3. choisir un sommet $s$ non colorié, et le colorier avec la couleur courante ;
4. répéter l'étape 5 pour tout sommet $t$ non colorié ;
5. si $t$ n'est adjacent à aucun sommet colorié avec la couleur courante, le colorier avec la couleur courante ;
6. choisir une nouvelle couleur ;
7. fin de l'algorithme.

Écrire une fonction coloriage(G) qui prend en argument un graphe et renvoie à l'aide de l'algorithme ci-dessus une coloration possible de ses sommets. On pourra utiliser (sans la recopier) la fonction auxiliaire suivante qui vérifie qu'aucun sommet de la liste voisins n'est de la couleur c.

```
def coloriable(voisins,couleur,c):
    for v in voisins:
        if couleur[v] == c: return False
    return True
```

```
def coloriable(voisins,couleur,c):
    for v in voisins:
        if couleur[v] == c: return False
    return True

def coloriage(G):
    n = len(G)
    couleur = [None] * n
    c = 0          # couleur courante
    incolores = n # nombre de sommets non colories
    while incolores > 0:
        s = 0
        while couleur[s] != None: s += 1
        couleur[s] = c
        incolores -= 1
        t = 0
```

```
        for t in range(n):
            if couleur[t]==None:
                if coloriable(G[t],couleur,c):
                    couleur[t] = c
                    incolores -= 1
        c += 1
    return couleur
```