Option Informatique

Le sujet est constitué de deux problèmes indépendants.

Problème n° 1 : suite de Lucas

Rappels et notations. Pour x un nombre réel, il existe un plus grand entier inférieur ou égal à x, appelé *plancher* de x ou *partie entière* de x. Ce nombre entier est noté $\lfloor x \rfloor$. Il existe un plus petit entier supérieur ou égal à x, appelé *plafond* de x.

Les fonctions `ceil` et `floor` du module `math` de Python calculent respectivement le plafond et le plancher d'un nombre flottant x : Ainsi `ceil(1.24)` vaut 2 et `floor(1.24)` vaut 1.

Soit a un nombre réel strictement positif. On désigne par $\log_a$ la fonction logarithme en base a : si x est un nombre réel strictement positif,

$$\log_a x = \frac{\ln x}{\ln a},$$

où ln désigne la fonction logarithme néperien.

Dans ce problème, on étudie plusieurs algorithmes de calcul des nombres de Lucas.

Ces nombres sont les termes de la suite $(L_n)_{n\geqslant 0}$ définie par les relations suivantes :

$$\begin{cases} L_0 = 2, \\ L_1 = 1, \\ L_n = L_{n-1} + L_{n-2} \text{ pour } n \geqslant 2. \end{cases}$$

I. Calculer L_2, L_3, L_4, L_5, L_6 et L_7.

II. Montrer que pour tout $n \geqslant 0$, L_n est un entier naturel.

III. On considère l'équation $x^2 - x - 1 = 0$.

1. Montrer que cette équation possède deux solutions, l'une positive que l'on note ϕ, l'autre négative que l'on note $\hat{\phi}$. Des valeurs approchées à 10^{-4} près de ces deux nombres sont $\phi \approx 1{,}6180$ et $\hat{\phi} \approx -0{,}6180$.

2. Justifier que

$$\phi + 1 = \phi^2, \qquad \hat{\phi} + 1 = \hat{\phi}^2, \qquad \phi + \hat{\phi} = 1, \qquad \phi\hat{\phi} = -1.$$

IV. Montrer que pour tout entier naturel n, $L_n = \phi^n + \hat{\phi}^n$. On pourra raisonner par récurrence.

V. On donne cette valeur approchée du logarithme en base 10 de ϕ :

$$\log_{10} \phi = \frac{\ln \phi}{\ln 10} \approx 0{,}2090.$$

Montrer que pour tout entier naturel p,

$$n \geqslant 5p \Longrightarrow L_n \geqslant 10^p.$$

VI. **1.** On propose la fonction suivante pour calculer le n-ième nombre de Lucas. On rappelle que ** est l'opérateur Python d'élévation à la puissance.

```
0 from math import *
1
2 def lucas1(n):
3     if n == 0:
4         return 2
5     phi = (1+sqrt(5))/2
6     phi2 = (1-sqrt(5))/2
7     return phi**n + phi2**n
```

L'évaluation de `[lucas1(n) for n in range(8)]` renvoie la liste

`[2, 1.0, 3.0, 4.0, 7.000000000000001, 11.000000000000002, 18.000000000000004, 29.000000000000007]`.

Pourquoi ne s'agit-il pas d'une liste d'entiers ?

2. On propose maintenant la fonction suivante pour calculer le n-ième nombre de Lucas.

```
0 from math import *
1
2 def lucas2(n):
3     if n == 0:
4         return 2
5     phi = (1+sqrt(5))/2
6     if n%2 == 0:
7         return ceil(phi**n)
8     else:
9         return floor(phi**n)
```

L'évaluation de `[lucas2(n) for n in range(8)]` renvoie la liste

`[2, 1, 3, 4, 7, 11, 18, 29]`.

Expliquer le choix de ces fonctions en lignes 6 à 9 et démontrer que, si les calculs en flottants sont exacts, `lucas2(n)` calcule bien L_n.

3. Un calcul exact montre que $L_{36} = 33385282$, mais `lucas2(36)` renvoie la valeur 33385283. Comment expliquez-vous cela ?

VII. On souhaite écrire une nouvelle fonction de calcul des termes de la suite de Lucas. Pour éviter tout problème lié au calcul avec des flottants, on souhaite ne travailler qu'avec des entiers, sur lesquels Python calcule de manière exacte.

1. Recopier et compléter la fonction suivante, qui renvoie la valeur de L_n.

```
0 def lucas3(n):
1     if n == 0:
2         return 2
3     if n == 1:
4         return 1
5     a,b = (2,1)
6     for i in range(n):
7         a,b = (..., ...)
8     return ...
```

2. Déterminer un invariant de boucle qui précise, en fonction de la valeur de `i`, les valeurs des variables `a` et `b` avant et après l'exécution de la ligne 7 et en déduire que `lucas3` renvoie un résultat exact.

3. Pour $n \geqslant 2$, combien d'additions sont effectuées par `lucas3(n)` ?

VIII. **1.** Démontrer que, pour tout entier $n \geqslant 1$,

$$L_n^2 - L_{n+1}L_{n-1} = 5\,(-1)^n.$$

2. Démontrer que, pour tout entier $n \geqslant 1$,

$$\begin{cases} L_{2n} &= L_n^2 - 2(-1)^n, \\ L_{2n+1} &= L_n L_{n+1} - (-1)^n. \end{cases}$$

IX. **1.** Si k est un entier, que calcule l'expression Python suivante : `1 - 2*(k % 2)` ?

On rappelle que l'opérateur `%` calcule le reste dans la division entière : `7 % 3` vaut 1 ; `8 % 3` vaut 2 et `9 % 3` vaut 0.

2. Recopier et compléter la fonction récursive suivante, de sorte que `lucas4(n)` renvoie le couple (L_n, L_{n+1}).

```
0  def lucas4(n):
1      if n == 0:
2          return (2,1)
3      if n == 1:
4          return (1,3)
5      k = n // 2
6      u = 1 - 2*(k % 2)
7      a,b = lucas4(...)
8      if n % 2 == 0:
9          return (..., ...)
10     else:
11         return (..., ...)
```

3. Pour tout entier $n \geqslant 2$, exprimer en fonction de n le nombre d'appels récursifs que réalise `lucas4(n)`.

X. Pour n un entier naturel, on considère le vecteur colonne de $\mathbb{R}^2$ défini par

$$V_n = \begin{pmatrix} L_n \\ L_{n+1} \end{pmatrix}.$$

On considère également la matrice $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Pour tout entier $n \geqslant 1$, exprimer V_n en fonction de A et de V_{n-1}, puis exprimer V_n en fonction de A, de n et de V_0. On représente dans la suite une matrice carrée $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ par la liste de deux listes

`[[a,b], [c,d]]`.

Tournez la page S.V.P.

XI. **1.** Écrire en Python une fonction `prodMat(M1,M2)` qui prend en arguments deux matrices 2×2 représentées par des listes comme indiqué ci-dessus, et qui renvoie la liste qui représente la matrice produit $M1 \times M2$.

2. Combien l'appel `prodMat(M1,M2)` effectue-t-il d'additions ? de multiplications ?

3. On considère la fonction (naïve) d'élévation d'une matrice à une puissance entière suivante.

```
0 def puissanceMat(M,p):
1     R = [[1,0],[0,1]]
2     for i in range(p):
3         R = prodMat(R,M)
4     return R
```

Combien l'appel `puissanceMat(A,p)` réalise-t-il d'appels à `prodMat` en fonction de p ?

XII. L'algorithme d'exponentiation rapide repose sur la remarque que

$$a^{2p+1} = (a^p)^2 a = b \times b \times a,$$

où $b = a^p$, quand $a^{2p} = b \times b$. Selon la parité de p, il faut donc 1 ou 2 multiplications de plus pour calculer a^{2p+1} ou a^{2p} connaissant a^p. Ainsi, pour calculer a^{122}, par exemple :

$$\begin{aligned} a^{122} &= (a^{61})^2 \\ &= (a.(a^{30})^2)^2 \\ &= (a.((a^{15})^2)^2)^2 \\ &= (a.((a.(a^7)^2)^2)^2)^2 \\ &= (a.((a.(a.(a^3)^2)^2)^2)^2)^2 \\ &= (a.((a.(a.(a.a^2)^2)^2)^2)^2)^2. \end{aligned}$$

Ainsi, on se contente de 10 multiplications.

1. On propose la fonction suivante qui implémente l'exponentiation rapide pour les matrices 2×2.

```
0 def puissanceMatRapide(M,n):
1     R = [[1,0], [0,1]]
2     P = M
3     while n > 0:
4         if n%2 == 1:
5             R = prodMat(R,P)
6         P = prodMat(P,P)
7         n = n // 2
8     return R
```

Montrer que par cette méthode, le nombre d'appel à la fonction `prodMat` est majoré par $2 + 2\lfloor \log_2 p \rfloor$.

2. Exprimer en fonction de M et de i la valeur de la matrice P, après la ième itération de la boucle `while`.

3. Soit $n = \overline{c_p \dots c_0}$ l'écriture de n en base 2. Montrer que, pour tout entier i compris entre 1 et p, la valeur de l'entier n après la ième itération de la boucle `while` est donée par

$$n = \sum_{j=i}^{p} c_j 2^{j-i}.$$

4. Montrer que pour tout entier i compris entre 1 et p, la valeur de la matrice R après la ième itération de la boucle `while` est donnée par $R = M^k$, avec

$$k = \sum_{j=0}^{i-1} c_j 2^j.$$

5. Exprimer le nombre d'exécutions de la boucle `while` en fonction de n puis démontrer la correction de l'algorithme proposé, c'est-à-dire que `puissanceMatRapide(M,n)` calcule effectivement la puissance désirée.

XIII. Proposer le code d'une fonction `lucas5(n)` qui retourne la valeur du nombre de Lucas L_n en utilisant la fonction `puissanceMatRapide`.

Tournez la page S.V.P.

Problème n° 2 : emplois du temps et graphes d'intervalles

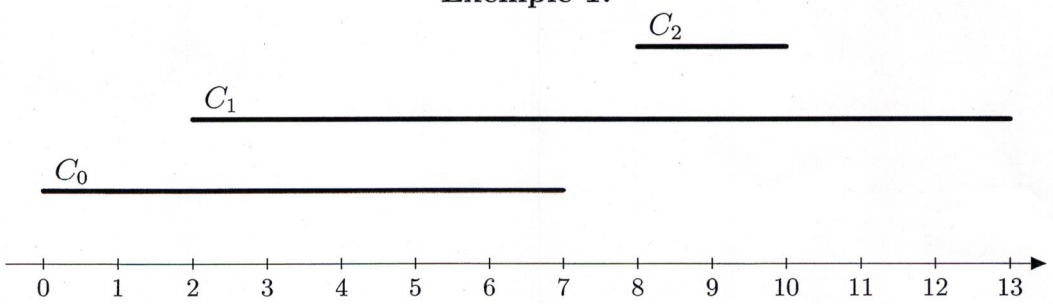
Dans ce problème, on s'intéresse à l'allocation des salles d'un lycée à partir des horaires des cours.

La donnée du problème est une liste de cours. Un cours est simplement représenté par un intervalle [`deb`, `fin`[, où `deb` est l'instant de début du cours et `fin` est l'instant de fin du cours. Les instants peuvent être décomptés en heures ou en minutes au fil de la journée selon les besoins ; dans ce sujet, les instants sont des nombres entiers et l'unité de temps n'est pas précisée.

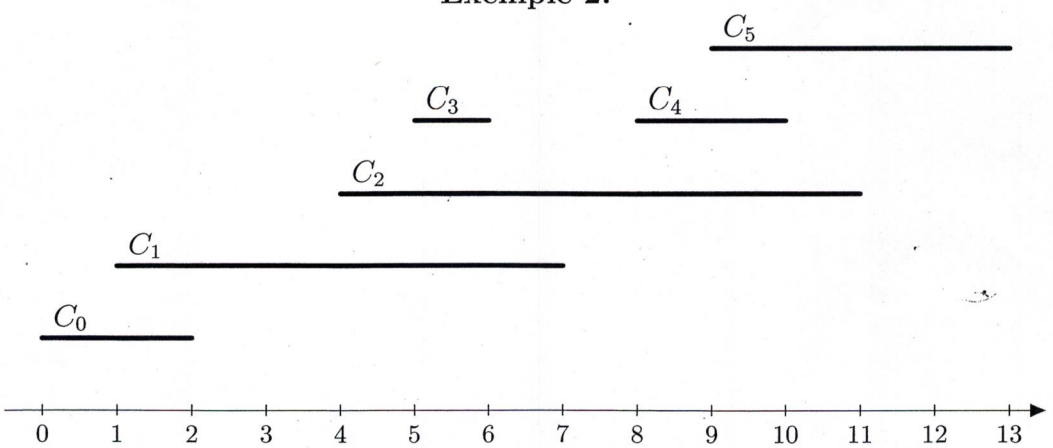
À chaque cours, on doit allouer une salle. Deux cours peuvent se voir allouer la même salle, uniquement si leurs intervalles sont disjoints. En particuliers, deux cours représentés par les intervalles $[4, 6[$ et $[6, 8[$ peuvent se voir allouer la même salle. L'objectif est d'utiliser un minimum de salles.

On présente ci-dessous, deux instances de ce problème.

Exemple 1.



Exemple 2.



Dans l'exemple 1, le cours C_2 est représenté par l'intervalle $[8, 10[$.

I. Pour l'allocation des salles, on parcourt simplement les cours par instants de début croissants et on alloue systématiquement la salle disponible avec le numéro le plus petit.

1. Décrire, sans justification, les allocations ainsi obtenues pour chacun des exemples précédents.
2. Déterminer le nombre minimal de salles nécessaires pour l'allocation dans les deux exemples précédents.

On admet pour la suite que le nombre optimal de salles nécessaire pour une liste de cours est le nombre maximal d'intervalles s'intersectant mutuellement en un instant donné.

II. On s'intéresse dans cette question à une première modélisation du problème. Le cours représenté par la liste Python `[d,f]` se déroule sur l'intervalle mathématique $[d, f[$.

1. Déterminer les listes d'intervalles représentant les exemples 1 et 2.
2. Compléter la définition suivante :

```
def insere(l,elt):
```

où `l` est une liste d'entiers triée dans l'ordre croissant et `elt` est un entier, et qui renvoie une liste triée obtenue par insertion à sa place de `elt` dans `l`. On notera que la liste `l` peut être vide.

On suppose pour la suite que l'on dispose d'une fonction similaire `insereBis` qui opère sur des listes de listes plutôt que sur des listes d'entiers, le critère de tri étant l'ordre des premiers éléments de chaque sous-liste. Plus précisément, la fonction `insereBis` a pour en-tête `def insereBis(LL,li):` et a pour effet d'insérer une liste `li` à la place adéquate dans la liste de listes `LL`.

III. Pour automatiser l'allocation des salles, on va utiliser une autre modélisation. L'intervalle `[deb,fin[` représentant le cours numéro i va être représenté par deux événements, modélisés par des listes de longueur 3 : `[deb,i,0]` et `[fin,i,1]`. Une liste d'intervalles pourra ainsi être représentée par une liste d'événements, c'est-à -dire une liste de listes de longueur 3 de la forme `[instant,num,0 ou 1]`.

1. Compléter la définition

```
def traduit(liste_intervalles):
```

qui prend en argument une liste d'intervalles représentés par des listes de longueur 2 et qui renvoie une liste d'événements (listes de longueur 3) correspondante. Notons que pour n intervalles, on obtient $2n$ événements.

2. Pour l'efficacité des algorithmes de résolution, on va travailler sur une liste d'événements triée par instants croissants. On appellera *agenda* toute liste d'événements triés par instants croissants. Quels sont les agendas correspondant aux exemples 1 et 2 ?
3. Compléter la définition

```
def agenda(liste_evt):
```

qui prend en argument une liste d'événements (listes de longueur 3) obtenue par un appel à la fonction traduit et qui renvoie l'agenda correspondant. On pourra utiliser la fonctions `insereBis`.

Tournez la page S.V.P.

IV. On a demandé à des élèves d'écrire une fonction qui vérifie qu'une liste d'événements donnée contient bien autant de fin que de début, et dans le bon ordre, mais sans tester l'appariemment cours par cours, c'est-à -dire sans considérer les numéros d'intervalles.

1. Parmi les quatres solutions proposées, déterminer (sans justifier) la ou les réponses correctes.

```
0  def valideA(agenda):
1      c = 0
2      for e in agenda:
3          if e[2] == 0: c += 1
4          else: c -= 1
5          if c < 0: return False
6          else : return True
7
8  def valideB(agenda):
9      n,c,i,b = len(agenda),0,0,True
10     while b and (i < n):
11         if agenda[i][2] == 0: c += 1
12         else: c -= 1
13         i += 1
14         b = (c >= 0)
15     return c == 0
16
17 def valideC(agenda):
18     for i in range(len(agenda)):
19         if agenda[i][2] == 0:
20             b = False
21             for j in range(i+1,len(agenda)):
22                 if agenda[j][2] == 1: b = True
23             if not b : return(b)
24     return(True)
25
26 def valideD(agenda):
27     c = 0
28     for e in agenda:
29         c += 1 - 2*e[2]
30         if c < 0: return False
31     return c == 0
```

2. Adapter une des fonctions précédentes pour écrire une fonction `intersection_max` qui à partir d'un agenda calcule le nombre maximal d'intervalles qui s'intersectent mutuellement. Justifier la correction de l'approche.

3. En utilisant les fonctions précédentes, écrire une fonction `nbr_optimal` qui à partir d'une liste d'intervalles (modélisation initiale), calcule le nombre de salles nécessaire.

V. **1.** On a demandé à un élève d'écrire une fonction qui, étant donné une liste de booléens, calcule le plus petit entier i tel que la case d'indice i vaille `True`. La fonction renverra -1 si un tel indice n'existe pas. Corriger sa fonction.

```
0  def plus_petit_vrai(liste):
1      n = len(liste)
2      while liste[i] and (i < n) : i+= 1
3      if i = n: return -1
4      else: return i
```

2. On utilise à chaque instant une liste de booléens pour indiquer si une salle est disponible ou non. En utilisant les fonctions précédentes, compléter la fonction suivante qui étant donnée une liste d'intervalles (listes de longueur 2), calcule une liste d'allocations des salles, toujours en allouant la salle disponible avec le plus petit numéro. Dans cette liste, la case d'indice le numéro du cours contient le numéro de la salle allouée.

```
0  def allocation(liste_intervalles):
1      nb_cours = ...
2      liste = ...
3      nb_salles = ...
4      salles_dispos = [True]*nb_salles
5      alloc = [-1]*nb_cours
6      for l in liste:
7          if l[2] == 0 :
8              alloc[l[1]] = ...
9              salles_dispos[...] = False
10         else :
11             salles_dispos[...] = ...
12     return(alloc)
```