Exercices - Capes 2018 - première épreuve option Informatique : corrigé

Avertissement : Ceci n'est pas une correction *in extenso* du problème de capes. Il s'agit plutôt d'une lecture personnelle des questions, avec des indications, des idées de preuve, des mises en garde d'erreurs à éviter. Ce n'est surtout pas une correction modèle à reproduire... Pour signaler toute erreur, merci d'écrire à `devgeolabo@gmail.com`

Problème 1

I.

II. Comment bien rédiger cette question ? Par récurrence double ? C'est parti ! On démontre par récurrence double que pour tout $n \in \mathbb{N}$, la propriété $\mathcal{P}(n)$ ="L_n est un entier naturel" est vraie.
Initialisation : $\mathcal{P}(0)$ et $\mathcal{P}(1)$ sont vraies.
Hérédité : Soit $n \in \mathbb{N}$ tel que $\mathcal{P}(n)$ et $\mathcal{P}(n+1)$ sont vraies. Alors on a $L_{n+2} = L_{n+1} + L_n$; la somme de deux entiers naturels étant un entier naturel, $\mathcal{P}(n+2)$ est vraie.
Conclusion : Par le principe de récurrence double, $\mathcal{P}(n)$ est vraie pour tout entier n.

III.1. Le discriminant de ce polynôme du second degré est $5 > 0$. L'équation admet donc deux racines,

$$\phi = \frac{1+\sqrt{5}}{2} \text{ et } \hat{\phi} = \frac{1-\sqrt{5}}{2}.$$

Clairement, $\phi > 0$ et puisque $\sqrt{5} > 2 > 1$, $\hat{\phi} < 0$.

III.2. On a

$$\phi^2 = \frac{1+2\sqrt{5}+5}{4} = \frac{3+\sqrt{5}}{2} = 1+\phi.$$

Les autres propriétés se démontrent exactement de la même façon, en n'utilisant que des arguments de collège !

IV. Comme à la question II., il faut faire une récurrence double. Démontrons donc par récurrence double que pour tout $n \in \mathbb{N}$, la propriété $\mathcal{P}(n)$ ="$L_n = \phi^n + \hat{\phi}^n$" est vérifiée.
Initialisation : On a bien $L_0 = 2 = \phi^0 + \hat{\phi}^0$. La propriété $\mathcal{P}(0)$ est vraie. De plus, on a aussi $\phi + \hat{\phi} = 1 = L_1$. La propriété $\mathcal{P}(1)$ est vraie.
Hérédité : Soit $n \in \mathbb{N}$ tel que $\mathcal{P}(n)$ et $\mathcal{P}(n+1)$ sont vraies. Alors,

$$\begin{aligned} L_{n+2} &= L_{n+1} + L_n \\ &= \phi^{n+1} + \hat{\phi}^{n+1} + \phi^n + \hat{\phi}^n \\ &= \phi^n(\phi+1) + \hat{\phi}^n(\hat{\phi}+1) \\ &= \phi^n\phi^2 + \hat{\phi}^n\hat{\phi}^2 \\ &= \phi^{n+2} + \hat{\phi}^{n+2}. \end{aligned}$$

La propriété $\mathcal{P}(n+2)$ est donc vraie.
Conclusion : par le principe de récurrence double, $\mathcal{P}(n)$ est vraie pour tout entier n.

V. Si $n \geq 5p$, alors $n\log_{10}(\phi) \geq 5 \times p \times 0,2 = p$. Ainsi,

$$n\ln\phi \geq p\ln 10 \implies \ln(\phi^n) \geq \ln(10^p) \implies \phi^n \geq 10^p$$

par croissance de la fonction logarithme népérien.

VI.1. On ne trouve pas des entiers, car puisqu'on utilise $\sqrt{5}$, Python va réaliser des calculs en flottants. Les calculs en flottants ne sont pas exacts, ce qui fait que, dès $n = 5$, on ne trouve plus un entier (mais presque un entier !).

VI.2. Si n est pair, alors $\hat{\phi}^n \geq 0$. En revanche, si n est impair, on a $\hat{\phi}^n \leq 0$. De plus, pour tout entier naturel n, $|\hat{\phi}^n| < 1$. Ainsi, pour n pair, $L_n = \phi^n + \hat{\phi}^n$ est bien le plus petit entier (strictement) plus grand que ϕ^n, que l'on obtient par la fonction `ceil`, et si n est impair, L_n est le plus grand entier (strictement) plus petit que ϕ^n, que l'on obtient par la fonction `floor`. Ceci justifie les lignes 6 à 9.

VI.3. Le problème est que ϕ^{36} est désormais très très proche de l'entier L_{36} -rappelons que la suite $(\hat{\phi}^n)$ converge vers 0. Il en est tellement proche que, avec les erreurs d'arrondis qui se propagent quand on calcule des puissances, la valeur approché que Python calcule pour ϕ^{36} est supérieure stricte à L_{36}. Ce qui fait que l'arrondi par excès sera non L_{36}, mais $L_{36} + 1$.

VII.1.

```
def lucas3(n):
    if n==0:
        return 2
    if n==1:
        return 1
    (a,b)=(2,1)
    for i in range(n):
        (a,b)=(b,a+b)
    return a
```

La solution proposée par l'énoncé n'est d'ailleurs pas optimale. Il aurait mieux valu effectuer une fois de moins la boucle et retourner b, comme ci-dessous :

```
def lucas3(n):
    if n==0:
        return 2
    if n==1:
        return 1
    (a,b)=(2,1)
    for i in range(n-1):
        (a,b)=(b,a+b)
    return b
```

VII.2. L'invariant de boucle à considérer est : au début de chaque exécution de la boucle, alors $(a, b) = (L_i, L_{i+1})$. On peut démontrer très facilement par récurrence que cette propriété est vraie à chaque itération.

VII.3. La boucle est effectuée n fois. A chaque exécution de la boucle, on effectue une addition. Donc, au total, on effectue n additions.

VIII.1. Il y a deux méthodes naturelles pour résoudre cette question. On peut procéder par une récurrence double, ou bien revenir à l'expression de L_n sous la forme $\phi^n + \hat{\phi}^n$, puis utiliser

les propriétés sur ϕ et $\hat{\phi}$ démontrées à la question III.2. Je choisis cette deuxième méthode, sans savoir si elle est la plus simple. On a d'une part

$$L_n^2 = (\phi^n + \hat{\phi}^n)^2 = \phi^{2n} + 2\left(\phi\hat{\phi}\right)^2 + \hat{\phi}^{2n} = \phi^{2n} + \hat{\phi}^{2n} + 2 \times (-1)^n$$

et d'autre part

$$\begin{aligned} L_{n+1}L_{n-1} &= (\phi^{n+1} + \hat{\phi}^{n+1})(\phi^{n-1} + \hat{\phi}^{n-1}) \\ &= \phi^{2n} + \hat{\phi}^{2n} + \phi^2\phi^{n-1}\hat{\phi}^{n-1} + \hat{\phi}^2\phi^{n-1}\hat{\phi}^{n-1} \\ &= \phi^{2n} + \hat{\phi}^{2n} + (\phi^2 + \hat{\phi}^2)(-1)^{n-1}. \end{aligned}$$

Maintenant,

$$\phi^2 + \hat{\phi}^2 = 2 + \phi + \hat{\phi} = 3.$$

Je crois que maintenant tout le monde est capable de finir !

VIII.2. La même méthode fonctionne exactement de la même façon (avec même des calculs plus faciles). Je vous laisse le soin de rédiger cela.

IX.1. On distingue deux cas :
— Si k est pair, alors $k\%2$ vaut 0, et l'expression vaut 1.
— Si k est impair, alors $k\%2$ vaut 2, et l'expression vaut -1.
Finalement, l'expression Python calcule $(-1)^k$.

IX.2. Voici une fonction qui convient.

```
def lucas4(n):
    if n==0:
        return(2,1)
    if n==1:
        return(1,3)
    k=n//2
    u=1-2*(k%2)
    a,b=lucas4(k)
    if n%2==0:
        return (a*a-2*u,a*b-u)
    else:
        return (a*b-u,b*b+2*u)
```

Un petit mot d'explication. Si n est pair, n s'écrit $n = 2k$ et $a, b = L_k, L_{k+1}$. On doit retourner L_{2k}, L_{2k+1}, ce que l'on fait avec les formules de la question précédente. Si n est impair, alors n s'écrit $2k+1$ et $a, b = L_k, L_{k+1}$. On doit retourner L_{2k+1}, L_{2k+2} et on écrit la formule donnant L_{2k+2} sous la forme $L_{2k+2} = L_{k+1}^2 - 2 \times (-1)^{k+1} = L_{k+1}^2 + 2 \times (-1)^k$.

IX.3. Démontrons par récurrence sur $p \geq 1$ que si n vérifie $2^p \leq n < 2^{p+1}$, alors le nombre d'appels récursifs que réalise `lucas4(n)` est exactement égal à p.
Initialisation : si $p = 1$, soit $n = 2$ ou $n = 3$, alors $k = 1$, et la fonction `lucas4(n)` appelle récursivement `lucas4(1)` qui ne fait plus d'appels récursifs.
Hérédité : soit $p \geq 2$ tel que la propriété soit vraie au rang p. Soit n tel que $2^{p+1} \leq n < 2^{p+2}$. Alors le k calculé dans la fonction vérifie $2^p \leq k < 2^{p+1}$. De plus, `lucas4(n)`

appelle récursivement `lucas4(k)`. Par hypothèse de récurrence, cette dernière fonction fera p appels récursifs. La fonction `lucas4(n)` fera donc en tout $p+1$ appels récursifs, et la propriété est vraie au rang $p+1$.
Par le principe de récurrence, on a donc démontré que si p est l'entier tel que $2^p \leq n < 2^{p+1}$, alors la fonction `lucas4(n)` fera p appels récursifs. On rappelle que p est égal à $\lfloor \log_2 n \rfloor$.

X. Les formules du produit matriciel nous disent que $V_n = AV_{n-1}$. Par récurrence (ou parce que l'on a une suite géométrique ?), $V_n = A^n V_0$.

XI.1. Voici une possibilité, sans doute pas la meilleure !

```
def prodMath(M1,M2):
    M=[]
    L1=[]
    L2=[]
    L1.append(M1[0][0]*M2[0][0]+M1[0][1]*M2[1][0])
    L1.append(M1[0][0]*M2[0][1]+M1[0][1]*M2[1][1])
    L2.append(M1[1][0]*M2[0][0]+M1[1][1]*M2[1][0])
    L2.append(M1[1][0]*M2[0][1]+M1[1][1]*M2[1][1])
    M.append(L1)
    M.append(L2)
    return M
```

XI.2. Il suffit de compter : 8 multiplications et 4 additions !

XI.3. La fonction `prodMath` est appelée une fois à chaque exécution de la boucle. La boucle est effectuée p fois, donc la fonction `prodMath` est appelée p fois.

XII.1. Cette question est très mal formulée ! Il faudrait mettre “appels” au pluriel, et on ne sait pas ce qu'est p. Je comprends la question sous la forme suivante : lorsqu'on appelle la fonction `puissanceMatRapide(M,n)`, alors le nombre d'appels à la fonction `prodMat` est majoré par $2+2\lfloor \log_2 n \rfloor$. La démonstration est alors très proche de la démonstration que nous avons réalisée à la question IX.3. Toutefois, je vais écrire une petite variante, avec une hypothèse de récurrence plus simple, mais il va falloir rédiger une récurrence forte. Pour $n \geq 1$, on note $\mathcal{P}(n)$ la propriété suivante : “ le nombre d'appels à la fonction `prodMat` dans la fonction `puissanceMatRapide(M,n)` est inférieur ou égal à $2+2\lfloor \log_2 n \rfloor$”.
Initialisation : `puissanceMetRapide(M,1)` appelle `prodMat` deux fois, et $2 = 2+2\lfloor \log_2 1 \rfloor$.
Hérédité : Soit $n \geq 2$. On suppose que $\mathcal{P}(k)$ est vraie pour tout $k \leq n-1$ et on va prouver que $\mathcal{P}(n)$ est vraie. On distingue alors deux cas :

— ou bien n est pair : dans ce cas, `ProdMat` est appelée une fois, et on appelle ensuite cette fonction autant de fois que dans un appel de `puissanceMatRapide(M,n/2)`. Le nombre d'appels total est donc majoré par $1+2+2\lfloor \log_2(n/2) \rfloor$. Mais $\log_2(n/2) = \log_2(n) - 1$ et donc le nombre d'appels est majoré par $1+2+2\lfloor \log_2(n) \rfloor - 2 \leq 2+2\lfloor \log_2(n) \rfloor$.

— ou bien n est impair : dans ce cas, `ProdMat` est appelée deux fois, et on appelle ensuite cette fonction autant de fois que dans un appel à `puissanceMatRapide(M,(n-1)/2)` Le nombre d'appels total est donc majoré par

$$2+2+2\lfloor \log_2((n-1)/2) \rfloor \leq 4+2\lfloor \log_2(n-1) \rfloor - 2 \leq 2+2\lfloor \log_2(n) \rfloor.$$

Dans tous les cas, on a prouvé que $\mathcal{P}(n)$ est vérifiée.
Conclusion : par le principe de récurrence fort, $\mathcal{P}(n)$ est vrai pour tout $n \geq 1$.

XII.2. A chaque itération de la boucle `while`, la valeur de P est changée en P^2. On part donc de $P = M$, après une itération, on a $P = M^2$, après deux itérations, $P = M^4$ et plus généralement après i itérations, on a $P = M^{2^i}$.

XII.3. Démontrons par récurrence sur $i \in \{0, \dots, p\}$ que la valeur de l'entier n après la i-ème itération de la boucle `while` est égale à $n = \sum_{j=i}^{p} c_j 2^{j-i}$ (on sous-entend que pour $i = 0$, la valeur de n est la valeur initiale).
Initialisation : pour $i = 0$, le résultat est vrai (l'égalité $n = \sum_{j=0}^{p} c_j 2^j$ correspond à l'écriture en base 2 de n).
Hérédité : soit $i \in \{1, \dots, p-1\}$ tel que la propriété est vraie au rang i. Alors, en sortant de la i-ème itération de la boucle `while`, on a $n \geq c_p 2^{p-i} \geq c_p = 1$. On exécute bien encore une fois la boucle while. La nouvelle valeur prise par n est alors

$$\frac{\sum_{j=i+1}^{p} c_j 2^{j-i}}{2} = \sum_{j=i+1}^{p} c_j 2^{j-(i+1)}.$$

La propriété est donc vérifiée au rang $i+1$.
En conclusion, par le principe de récurrence (finie), on a démontré que la propriété est vraie pour tout i allant de 1 à p.

XII.4. Il faut encore faire une récurrence finie, très semblable à la question précédente, et qui utilise aussi de façon cruciale la question XII.2. Le point clé, dans l'hérédité, est de remarquer que, si avant l'exécution de la $i+1$-ème itération de la boucle, on a $R = M^k$, alors, utilisant que après la i-ème itération de la boucle, on a $n = \sum_{j=i}^{p} c_j 2^{j-i}$ (et donc n est pair si et seulement si $c_i = 1$) et $P^i = M^{2^i}$, alors $R = M^\ell$ où

$$\ell = k + c_i 2^i.$$

Les détails sont laissés au lecteur !

XII.5. On a déjà observé à la question XII.3. qu'on allait exécuter au moins p fois la boucle `while` et qu'après la p-ème itération, on a $n = c_p = 1$. On exécute alors encore une fois la boucle (et donc on exécute $p+1 = \lfloor \log_2 n \rfloor + 1$ fois la boucle), et on ressort (cf la question XII.4. dont le résultat est valable jusque $p+1$ et pas seulement p) avec $R = M^n$. L'algorithme est correct ! Cette question XII. est certainement plus compliquée que le reste du problème !

XIII. En faisant le bilan des questions précédentes, on peut utiliser la fonction suivante (remarquons qu'il suffit de calculer A^{n-1} pour en déduire (L_{n-1}, L_n)).

```
def lucas5(n):
    if n==0:
        return 2
    if n==1:
        return 1
    A=[ [0,1], [1,1]]
    M=puissanceMatRapide(A,n-1)
    return M[1][0]*2+M[1][1]*1
```

Problème 2

I.1. Exemple 1 : C_0 salle 0, C_1 salle 1, C_2 salle 0.
Exemple 2 : C_0 salle 0, C_1 salle 1, C_2 salle 0, C_3 salle 2, C_4 salle 1, C_5 salle 2.

I.2. Il suffit de deux salles pour le premier exemple et de trois salles pour le deuxième. Ces nombres sont minimaux. Dans le premier exemple, les cours C_0 et C_1 ont une intersection non vide. Dans le deuxième exemple, les cours C_1, C_2 et C_3 ont aussi une intersection (mutuelle) non vide. Il faut au moins trois salles pour ces trois cours.

II.1. Exemple 1 : $[0, 7[$, $[2, 13[$ et $[8, 10[$.
Exemple 2 :$[0, 2[$, $[1, 7[$, $[4, 11[$, $[5, 6[$, $[8, 10[$, $[9, 13[$.

II.2.
```
def insere(l,elt):
    m=l.copy()
    if (len(m)==0):
        m.append(elt)
        return m
    if (elt>m[len(m)-1]):
        m.append(elt)
        return m
    i=0
    while (m[i]<elt):
        i+=1
    m.insert(i,elt)
    return m
```

Dans la fonction, on commence par tester si la liste est vide. Si ce n'est pas le cas, on regarde si l'élément est plus grand que tous les éléments de la liste : si c'est le cas, on l'ajoute à la fin. Sinon, on parcourt tous les éléments de la liste jusqu'à trouver la bonne place. Remarquons que la boucle se termine car, si on exécute la boucle, on sait que `elt` est inférieur ou égal au dernier élément de la liste. On réalise au début de la fonction une copie de la liste l pour ne pas modifier celle-ci.

III.1. Le plus difficile est de comprendre ce que l'on doit faire !

```
def traduit(liste_intervalles):
    liste_evts=[]
    for i in range(len(liste_intervalles)):
        liste_evts.append([liste_intervalles[i][0],i,0])
        liste_evts.append([liste_intervalles[i][1],i,1])
    return liste_evts
```

III.2. Je ne le fais que pour l'exemple 1. L'agenda correspondant est la liste

$$[0, 0, 0], [2, 1, 0], [7, 0, 1], [8, 2, 0], [10, 2, 1], [13, 1, 1]$$

III.3. Il s'agit d'un simple tri par insertion.

```
def agenda(liste_evt):
    agenda=[]
```

```
    for evt in liste_evt:
        agenda=insereBis(agenda,evt)
    return agenda
```

IV.1. Les fonctions `valideA` et `valideC` ne fonctionnent pas. La fonction `valideA` renverra vraie si la liste ne contient que des débuts de cours, et la fonction `valideC` renverra vrai si la liste ne contient que des fins de cours. Les deux autres fonctions sont (semblent ?) correctes.

IV.2. Il suffit de compter le nombre de cours qui ont déjà commencé, mais ne se sont pas encore terminés. Dans les diverses fonctions proposées, ce nombre est c, et on cherche donc la valeur maximale prise par c. Ainsi, en modifiant `valideD`, on peut écrire

```
def intersection_max(agenda):
    c = 0
    d = 0
    for e in agenda :
        c += 1 - 2*e [2]
        if c >d :
            d=c
    return d
```

V.1. Plusieurs erreurs dans ce programme :
- la variable i n'est pas initialisée ;
- le test est effectué avec une mauvaise condition : on va plus loin tant que Liste [i] est faux ;
- si la liste ne contient que des False, il y a débordement : on va tester Liste[n] qui n'existe pas. On peut résoudre facilement ce problème en permutant l'ordre des tests dans la boucle while ;
- le test du if doit s'écrire avec le symbole ==.

Voici une version corrigée (enfin, j'espère) :

```
def plus_petit_vrai(liste):
    n=len(liste)
    i=0
    while (i<n) and not(liste[i]):
        i+=1
    if i==n:
        return -1
    else:
        return i
```

V.2. Et voila !

```
def allocation ( liste_intervalles ):
    nb_cours = len(liste_intervalles)
    liste = agenda(traduit(liste_intervalles))
    nb_salles = intersection_max(liste)
    salles_dispos = [ True ]* nb_salles
```

```
    alloc = [ -1]* nb_cours
    for l in liste :
        if l [2] == 0 :
            alloc [l [1]] = plus_petit_vrai(salles_dispos)
            salles_dispos [alloc[l[1]]] = False
        else :
            salles_dispos [alloc[l[1]]] = True
    return ( alloc )
```