TP d'Informatique du Concours Blanc : corrigé

PTSI Lycée Eiffel

6 juin 2019

I. Exploitation de la relation de récurrence.

1. Comme suggéré par l'énoncé, on va utiliser deux variables u et v qui stockeront à la n-ème étape de calcul les valeurs de u_n et de u_{n+1}. On fait bien attention au fait que, connaissant initialement u_0 et u_1, on n'a besoin que de $n-1$ actualisations des valeurs pour obtenir u_n. Il n'est par ailleurs pas du tout obligatoire de créer une variable temporaire dans la boucle puisqu'on peut très bien effectuer les modifications des valeurs des deux variables simultanément en Python. Par exemple :

```
def fibo(n) :
    u,v=0,1
    for i in range(n-1) :
        u,v=v,u+v
    return v
```

2. Il s'agit cette fois d'ajouter à chaque étape de calcul un nouveau terme à notre liste, obtenu en faisant la somme des deux termes calculés, c'est-à-dire des deux derniers termes de la liste (avant modification). En Python, les commandes du type l[-i] permettent de faire celà très simplement :

```
def fiboliste(n) :
    L=[0,1]
    for i in range(n-1) :
        L.append(L[-1]+L[-2])
    return L
```

3. Il suffit de construire la liste des termes de la suite jusqu'à u_{2n+1} (à l'aide du programme de la question précédente), puis de tester l'égalité pour tous les entiers de 0 jusqu'à n inclus (attention à mettre ce qu'il faut dans le range). Bien sûr, si l'égalité n'est pas vérifiée (ce qui ne devrait pas arriver) pour une certaine valeur de l'indice, on peut sortir de la boucle via un return sans tester les valeurs suivantes. Le return de la dernière ligne ne sera exécuté que si on est arrivé au bout de la boucle sans en sortir, donc si l'égalité est bien vérifiée pour tous les entiers souhaités.

```
def verifiecarres(n) :
    L=fiboliste(2*n+1)
    for i in range(n+1) :
        if L[i]**2+L[i+1]**2 !=L[2*i+1] :
            return False
    return True
```

4. On reprend tout simplement le programme de la première question, en remplaçant la boucle for par un while pour s'arrêter dès qu'on a dépassé la valeur de p. On peut raffiner le programme pour qu'il renvoie la valeur 0 (au lieu de 1) si $p = 0$ (ou même si $p \leqslant 0$).

```
def premierplusgrand(p) :
    u,v=0,1
    while v<p :
        u,v=v,u+v
    return v
```

5. Il suffit d'ajouter au programme précédent une variable comptant le nombre de calculs effectués (variable initialisée à 1 puisque v prend la valeur u_1 en début de programme) :

```
def plusgrandavecindice(p) :
    u,v,n=0,1,1
    while v<p :
        u,v,n=v,u+v,n+1
    return n,v
```

6. L'idée naturelle est de transformer le nombre n en une liste de chiffres, puis de sélectionner les entiers compris entre 0 et 9 qui apparaissent (au moins une fois) dans cette liste. Pour cette dernière tâche, la méthode **count** déjà existante en Python (ou même un simple **in**) ferait le travail mais l'énoncé laisse entendre qu'on n'a pas trop le droit à ça, donc on va reprogrammer une fonction **compte(L,n)** déterminant le nombre d'apparitions de n dans la liste L (en plus, ce n'est pas dur à faire). Pour transformer notre entier décimal en liste de chiffres, plein de possibilités, j'en donne deux : la plus sale consiste à transformer l'entier en chaîne de caractères pour isoler facilement les chiffres ; la seconde consiste à récupérer les chiffres l'un après l'autre (en commençant par le chiffre des unités) en effectuant des divisions euclidiennes par 10 (le chiffres des unités d'un entier étant simplement le reste de sa division euclidienne par 10). En bonus une version récursive élégante de la création de la liste des chiffres par cette dernière méthode (ce dernier programme ne répond pas à la question de l'énoncé !).

```
def compte(L,n) :
    c=0
    for i in L :
        if i==n :
            c=c+1
    return c


def chiffres(n) :
    L=[int(i) for i in str(n)]
    return [i for i in range(10) if compte(L,i)>0]


def chiffresbis(n) :
    L,a=[],n
    while a>0 :
        L.append(a%10)
        a=a//10
```

```
    return [i for i in range(10) if compte(L,i)>0]


def recchiffres(n) :
    if n==0 :
        return []
    return recchiffres(n//10)+[n%10]
```

7. En utilisant les programmes écrits pour les questions précédentes, un nombre contient chacun des chiffres de 0 à 9 si et seulement si chiffres(n) est une liste de longueur 10. On arrête donc le programme dès que cette condition est vérifiée (la liste L est initialisée arbitrairement à [0]).

```
def touschiffres() :
    u,v,n,L=0,1,1,[0]
    while len(L)<10 :
        u,v,n=v,u+v,n+1
        L=chiffres(v)
    return n
```

8. On veut écrire un programme similaire au précédent, mais il faut cette fois connaitre le nombre de fois où chaque chiffre apparait dans l'écriture décimale de v, ce qui empêche de travailler avec chiffres(v) et force à utiliser la liste complète de tous les chiffres de v. Notre nombre contient au moins p fois chaque chiffre entre 0 et 9 si le minimum du nombre d'apparitions de chaque chiffre dans le liste complète des chiffres de v est au moins égal à v. Du coup, une fonction déterminant le minimum d'une liste est utile, et comme on n'a pas le droit d'utiliser directement **min**, on en reprogramme un. On réutilise par ailleurs la fonction compte écrite pour la question 6.

```
def mini(L) :
    a=L[0]
    for i in L[1 :] :
        if i<a :
            a=i
    return a


def touschiffresplusieursfois(p) :
    u,v,n,L=0,1,1,[0]
    while mini(L)<p :
        u,v,n=v,u+v,n+1
        liste=[int(i) for i in str(v)]
        L=[compte(liste,i) for i in range(10)]
    return n
```

II. Exploitation d'une formule explicite.

1. On peut utiliser la commande Python **int** pour obtenir la partie entière du nombre x, et distinguer deux cas selon que la distance entre x et sa partie entière est ou non supérieure à

$\frac{1}{2}$. Le programme ci-dessous ne fonctionne que si $x \geqslant 0$ (ce qui suffira pour la suite) mais on peut facilement l'adapter pour prendre aussi en compte les nombres négatifs.

```
def plusprocheentier(x) :
    if x-int(x)<0.5 :
        return int(x)
    return int(x)+1
```

2. Il suffit tout bêtement d'appliquer la formule donnée. On a ici utilisé la fonction **sqrt** mais les résultats sont exactement les mêmes avec des puissances.

```
from math import sqrt
def fibobis(n) :
    phi=(1+sqrt(5))/2
    return plusprocheentier(phi**n/sqrt(5))
```

3. On peut simplement faire augmenter un compteur jusqu'à ce que les fonction fibobis et fibo donnent des résultats différents (ce n'est pas optimal car on va refaire beaucoup de calculs déjà faits à chaque nouvel appel des fonctions, mais ce n'est pas bien grave ici, la réponse arrivera de toute façon très rapidement. Pour ceux qui se demandent pourquoi on observe ces différences entre les valeurs calculées par les deux foncions, elles sont tout simplement dues à des erreurs d'arrondi dans les calculs de la fonction fibobis (la fonction fibo, elle, donne toujours le bon résultat). En gros, les calculs sur les nombres flottants en Python permettent d'effectuer des calculs avec 15 chiffres significatifs exacts (ce qui est normal pour des flottants codés sur 8 octets avec 52 bits de mantisse). Comme les valeurs de u_n atteignent les quinze chiffres pour $n = 70$, il est normal qu'on commence à observer des erreurs aux alentours de cette valeur de n. Remarquez l'initialisation du compteur à $n = 1$, en effet notre fonction fibo ne donne pas la bonne valeur pour $n = 0$...

```
n=1
while fibobis(n)==fibo(n) :
    n=n+1
print(n)
```

4. On reprend un programme du même type que pour la toute première question du sujet, mais en plus de stocker deux valeurs successives de la suite (u_n) (ici dans les variables nommées u et v), on stocke aussi deux valeurs successives de la suite (v_n) (dans des variables nommées z et t). On actualise ces valeurs jusqu'à obtenir un écart inférieur à e pour la distance entre z et t, ce qui d'après les résultats rappelés dans l'énoncé implique que z (ou t) sera une valeur approchée de φ à e près.

```
def approxor(e) :
    u,v,t,z=1,2,1,2
    while abs(t-z)>e :
        u,v=v,u+v
        t,z=z,float(v)/u
    return z
```

III. Exploitation du calcul matriciel.

1. On applique très bourrinement la formule de produit de deux matrices à deux lignes et deux colonnes (on a donné des noms aux coefficients des deux matrices dans le programme ci-dessus,

mais on peut en écrire salement les formule directement dans le return pour economiser des variables.

```
def produitmat(L1,L2) :
    a,b,c,d=L1[0][0],L1[0][1],L1[1][0],L1[1][1]
    e,f,g,h=L2[0][0],L2[0][1],L2[1][0],L2[1][1]
    return [[a*e+b*g,a*f+b*h],[c*e+d*g,c*f+d*h]]
```

2. On sépare le cas $n = 0$ (on retourne les liste correspondant à la matrice identité dans ce cas particulier), et on fait une classique boucle for sinon en utilisant le programme précédent. On peut aussi écrire un programme récursif (ça n'a pas grand intérêt ici), ou même si on veut optimiser les calculs réfléchir à une façon d'effectuer le moins de produits matriciels possible, ce qu'on n'a pas fait ici (par exemple, notre programme effectuera sept produits matriciels pour calculer A^8 alors que trois produits suffisent en exploitant $A^8 = ((A^2)^2)^2$; les plus curieux se renseigneront sur l'exponentiation rapide pour améliorer ce programme).

```
def puissancemat(M,n) :
    if n==0 :
        return [[1,0],[0,1]]
    res=M[ :]
    for i in range(n-1)
        res=produitmat(res,M)
    return res
```

3. Encore un programme facile, il suffit de calculer A^{n-1} et de retourner son coefficient deuxième ligne deuxième colonne :

```
def fibomat(n) :
    A=[[0,1],[1,1]]
    B=puissancemat(A,n-1)
    return B[1][1]
```

IV. Exploitation des coefficients binômiaux.

1. C'est normalement une question de cours. On donne ici les deux versions, avec une boucle et avec la récursivité :

```
def factorielle(n) :
    a=1
    for i in range(2,n+1) :
        a=a*i
    return a


def recfact(n) :
    if n==0 :
        return 1
    return n*recfact(n-1)
```

2. Encore une question triviale, où on se contente de recopier une formule en exploitant le programme précédent (les plus observateurs noteront quand même qu'il semblerait que j'aie oublié la condition $n < 0$ dans mon programme ; c'est bien sûr faux : si $n < 0$ on a de toute façon $k < 0$ ou $n < k$, et la fonction renverra bien 0 comme prévu) :

```
def binom(n,k) :
    if n<k or k<0 :
        return 0
    return (recfact(n)/(recfact(k)*recfact(n-k)))
```

3. Il s'agit encore une fois essentiellement de recopier une formule, mais on a tout de même un léger souci pour le calcul de somme dans la mesure où l'énoncé nous interdit la commande **sum**. Pas grave, on la reprogramme rapidement. Notez qu'une partie encière peut se calculer directement comme quotient d'une division euclidienne.

```
def somme(L) :
    a=0
    for i in L :
        a=a+i
    return a


def fibobino(n) :
    p=(n-1)//2
    return somme([binom(n-1-k,k) for k in range(p+1)])
```

4. Ah, enfin une question non triviale dans ce sujet ! Le programme proposé manipule une seule liste, en lui adjoignant un 1 à chaque étape (les deux coefficients binômiaux extrêmes sont de toute façon toujours égaux à 1) puis en actualisant les coefficients intermédiaires de droite à gauche (d'où la deuxième boucle for à pas négatif), de façon à ne pas écraser trop tôt des valeurs qui vont resservir pour le calcul suivant. On peut bien sûr aussi utiliser des variables flottantes temporaires pour effectuer les calculs, voire même une liste temporaire pour garder tous les coefficients sous la main pendant les calculs.

```
def listebino(n) :
    L=[1]
    for i in range(n) :
        L.append(1)
        for j in range(i,0,-1) :
            L[j]=L[j]+L[j-1]
    return L
```