Devoir surveillé 1 : corrigé
option Info

Master 1 MEEF Maths 2017/2018

Convention Dans ce sujet, l'écriture « `machine à écrire` » représente du code Python.

Problème 1. La spirale d'Ulam

1.1 Dessin d'une spirale

Question 1.1.

(a)
```
from turtle import *
right(90)
forward(20)
left(90)
forward(20)
left(90)
forward(40)
left(90)
forward(40)
left(90)
forward(40)
```

(b) On montre par récurrence sur n que $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. Le résultat est évident pour $n = 1$. Si on suppose le résultat pour une valeur n, on a $\sum_{i=1}^{n+1} i = \frac{n(n+1)}{2} + n + 1 = \frac{(n^2+n)+(2n+2)}{2} = \frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2}$.
On en déduit que $n + \sum_{i=1}^{n}(i+i) = n + n(n+1) = (n+1) - 1 + n(n+1) = (n+1)^2 - 1$.
La somme calculée est le nombre de pas effectués par la tortue dans la spirale $n \times n$. En effet, la spirale est constituée de deux côtés de longueur i pour i de 1 à n, puis un dernier côté de longueur n. L'égalité signifie que la spirale remplit exactement un carré $(n+1) \times (n+1)$: le nombre de pas est le nombre de cases de ce carré moins 1.

(c)
```
def spirale(n):
    right(90)
    for i in range(1, n+1):
        forward(20*i)
        left(90)
        forward(20*i)
        left(90)
    forward(20*n)
```

Question 1.2.

(a) Le point de départ à pour coordonnées $(\lfloor \frac{n-1}{2} \rfloor, \lfloor \frac{n-1}{2} \rfloor)$.

(b)
```
def table_spirale(n):
    t = [[0] * n for _ in range(n)]
    k = 0
    for a,b in coordonnees_spirale(n):
        t[b][a] = k
        k += 1
    return t
```

Question 1.3.

(a) Il y a deux possibilités : la diagonale $47 - 23 - 7 - 5 - 17 - 37$ ou la diagonale $31 - 13 - 3 - 5 - 19 - 41$.

(b)
```
def alignes(a,b,c):
    xa, ya = a
    xb, yb = b
    xc, yc = c
    return (xa-xb) * (ya-yc) == (xa-xc) * (ya-yb)
```

(c)
```
def liste_alignes(L):
    if len(L) < 3: return True # Deux points (ou moins...) sont toujours ali
    a, b = L[:2]
    for c in L[2:]:
        if not alignes(a,b,c):
            return False
    return True
```

1.2 Nombre de diviseurs

Question 1.4.

(a) Les entiers 0 et 1 n'ont aucun diviseur strict. Les nombres ayant un unique diviseur strict sont les nombres premiers.

(b) Le nombre de passages dans la boucle est exactement $n-1$ car l'indice k augmente de 1 à chaque étape. La complexité est donc $O(n)$.

(c) Commençons par remarquer qu'un diviseur strict de n est nécessairement $\leqslant n/2$. On peut donc modifier le test du while en écrivant `while k <= n//2`.

Mais on peut faire beaucoup mieux : si d divise n, alors n/d divise n également, et l'un de ceux deux entiers est $\leqslant \sqrt{n}$. Il suffit donc de tester les diviseurs jusqu'à $\sqrt{n}$, et d'ajouter 2 au compteur à chaque fois, en faisant attention tout de même à ne pas compter deux fois $\sqrt{n}$ quand n est un carré parfait. On obtient l'algorithme suivant, de complexité $O(\sqrt{n})$:

```
def nombre_diviseurs2(n):
    if n < 2: return 0
    c = 1 # 1 est toujours diviseur
```

```
    k = 2
    while k*k < n: # k < racine_carree(n)
        if n % k == 0:
            c += 2
        k += 1
    if k*k == n:
        c += 1
    return c
```

(d)
```
def est_parfait(n):
    if n<2: return False
    somme = 1
    k = 2
    while k*k < n:
        if n % k == 0:
            somme += k + n//k
        k += 1
    if k*k == n:
        somme += k
    return somme == n
```

Question 1.5. Crible d'Eratosthène

(a) Encore une fois il suffit de s'arrêter à $\lfloor\sqrt{n}\rfloor$, pour les mêmes raisons.

(b)
```
def crible(n):
    """
    Algo : on initialise un tableau à True (tous les
    nombres sont potentiellement premiers), puis pour
    chaque nombre non encore barré, on barre ces multiples
    en commencant par son carré. On construit à la fin la
    liste des nombres non barrés.
    """
    L = [True] * (n+1) # nombres potentiellement premiers
    L[0] = L[1] = False # 0, 1 non premiers
    k = 2
    while k*k <= n:
        if L[k]: # k non barré
            j = k*k # tous les multiples inférieurs ont déjà été barrés
            while j <= n:
                L[j] = False # j est un multiple de k
                j += k
        k += 1
    return [p for p in range(2, n+1) if L[p]]
```

(c)
```
def crible_etendu(n):
    """
    Pour le crible étendu, on initialise le tableau à 1
    (les entiers sont tous divisibles par 1) sauf pour 0 et
    1, puis on met à jour en regardant tous les multiples de
    chaque entier k.
    """
    C = [1] * (n+1) # les nombres ont au moins un diviseur strict
    C[0] = C[1] = 0 # sauf 0 et 1
    k = 2
    while 2*k <= n:
        j = 2*k
        while j <= n:
            C[j] += 1
            j += k
        k += 1
    return C
```

(d) La boucle `while` externe de l'algorithme `crible_etendu` parcourt les entiers k de 2 à $\lfloor n/2 \rfloor$. Puis la boucle interne va de $j = 2k$ à $j = n$ (au maximum), de k en k. Donc j/k va de 2 à $\lfloor n/k \rfloor$, c'est-à-dire que la boucle est effectuée au plus n/k fois. Au total, la boucle interne est donc parcourue $\sum_{k=2}^{\lfloor n/2 \rfloor} \lfloor n/k \rfloor \leqslant n \times (H_{n/2} - 1)$ où $H_n = \sum_1^n 1/k$ est le n-ème nombre harmonique et la majoration est obtenue en supprimant les parties entières.

Pour finir, on sait que $H_n \sim \ln(n)$, ou plus simplement $H_n \leqslant 1 + \ln(n)$ (on peut montrer ce dernier résultat par récurrence sur n).

Ainsi, la complexité de l'algorithme est bornée par $O(n \log n)$.

En utilisant `nombre_diviseurs`, on ferait n appels à un algorithme linéaire, on serait donc sur du $\sum_1^n k = O(n^2)$. Remarquons qu'utiliser `nombre_diviseurs2` suffirait déjà à se ramener à $O(n\sqrt{n})$.

1.3 Spirale d'Ulam

Question 1.6.
```
def ulam(n):
    s = table_spirale(n)
    u = [[0] * n for _ in range(n)]
    C = crible_etendu(n*n-1)
    for i in range(n):
        for j in range(n):
            u[i][j] = C[s[i][j]]
    return u
```

D'après un corrigé de B. Grenet (U. Montpellier).

Problème 2. Quelques systèmes de chiffrement

2.1 Chiffrement à clef secrète

Question 2.1. Chiffrement par décalage

(a) ALGORITHME est chiffré en FQLTWNYMRJ et RFYMX est déchiffré en MATHS.

(b) On peut décaler de n'importe quelle valeur entre 0 et 25, soit 26 clefs possibles.

(c) Notez l'opérateur `join` qui concatène efficacement toutes les chaînes d'une liste. La chaîne à gauche de `join` sert de séparateur entre les caînes à concaténer, c'est pourquoi la plupart du temps on mettra une chaîne vide.

```
def chiffre_decalage(clef, message):
    return "".join(lettre((numero(m) + clef) % 26) for m in message)

def dechiffre_decalage(clef, chiffre):
    return "".join(lettre((numero(c) - clef) % 26) for c in chiffre)
```

Question 2.2. Chiffrement de Vigenère

(a) Le chiffré de RECURSIVITE avec la clef CAFE est TEHYTSNZKTJ. Le message clair correspondant à KTJVCTNSP avec la même clef est ITERATION.

(b)
```
def chiffre_Vigenere(clef, message):
    n = len(clef)
    i = 0
    chiffre = ""
    for m in message:
        decalage = numero(clef[i])
        chiffre += lettre((numero(m) + decalage) % 26)
        i = (i + 1) % n
    return chiffre

def dechiffre_Vigenere(clef, chiffre):
    n = len(clef)
    return "".join(
        lettre((numero(chiffre[i]) - numero(clef[i%n])) % 26)
            for i in range(len(chiffre)))
```

(c) Il y a pour chaque lettre 26 possibilités. Le nombre de clefs de longueur n est donc 26^n. Ainsi, les clefs de longueur 4 à 10 sont au nombre de $26^4 + 26^5 + \ldots + 26^{10} = \sum_{i\leq 10} 26^i - \sum_{i\leq 3} 26^i$, ce qui vaut $N = \frac{26^{11}-26^4}{25}$. Pour estimer le nombre de chiffre en base 10 de N, on calcule approximativement $\log_{10} N$. On a $\log N = \log(26^{11} - 26^4) - \log 25$. Or $26^{11} - 26^4$ est très proche de 26^{11}, et $\log 26^{11} = 11 \log 26 = 11(\log 13 + \log 2)$. Et $\log 25 = 2 \log 5$. En lisant les valeurs sur la courbe, on a $\log 2 \simeq 0,3$, $\log 5 \simeq 0,7$ et $\log 13 \simeq 1,1$. On obtient donc $\log N \simeq 14$. On en déduit que N possède 14 chiffres en base 10. Si on calcule précisément, on obtient $N \simeq 1,5 \cdot 10^{14}$.

Question 2.3. Cryptanalyse du chiffrement par décalage

```
def cryptanalyse_decalage(chiffre):
    # comptage des caractères
    occurrences = [0] * 26
    for c in chiffre:
        occurrences[numero(c)] += 1
    # détermination du code de E
    imax = 0
    M = 0
    for i in range(len(occurrences)):
        if occurrences[i] > M:
            imax = i
            M = occurrences[i]
    decalage = imax - numero('E')
    return dechiffre_decalage(decalage, chiffre)
```

Question 2.4. Cryptanalyse du système de Vigenère

(a)

```
def motifs(chiffre):
    d = {}
    for i in range(len(chiffre)-3):
        motif = chiffre[i:i+3]
        if motif not in d:
            L = [i]
            for j in range(i+1, len(chiffre)-3):
                if chiffre[j:j+3] == motif:
                    L.append(j)
            if len(L) > 1:
                d[motif] = L
    return d
```

(b)

```
def cryptanalyse_Vigenere(chiffre):
    motif = motifs(chiffre)
    # détermination de la longueur de la clé
    g = 0
    for m in motif:
        positions = motif[m]
        differences = [p-positions[0] for p in positions[1:]]
        for d in differences:
            g = fractions.gcd(g, d)
    clef = ""
    for i in range(g):
        # les caractères aux positions égales à i modulo g
        # sont tous chiffrés avec le meme décalage,
        # que l'on trouve par l'approche fréquentielle
        occurrences = [0] * 26
```

```
        for j in range(i, len(chiffre), g):
            occurrences[numero(chiffre[j])] += 1
        jmax = 0
        M = 0
        for j in range(len(occurrences)):
            if occurrences[j] > M:
                jmax = j
                M = occurrences[j]
        clef += lettre(jmax - numero('E'))
    return dechiffre_Vigenere(clef, chiffre)
```

Petite remarque : quand on calcule les PGCD, on risque toujours de tomber sur 1 à un moment donné, si un motif apparaît deux fois à une distance qui n'est pas un multiple de la longueur de la clef. Ces deux motifs ne signifient en fait rien sur le message chiffré.
Une façon simple de contourner ce problème (mais qui ne marche pas à tous les coups non plus) est d'ignorer les distances dès qu'elles font tomber sur un PGCD égal à 1. On remplace donc la ligne `g = fractions.gcd(g, d)` par

```
gg = fractions.gcd(g, d)
if gg > 1: g = gg
```

2.2 Chiffrement à clef publique

Question 2.5. Problème du sac-à-dos

(a)

```
def sacados(a, S):
# plusieurs cas de base (possible/impossible)
if len(a) == 1:
    if a[0] == S:
        return (1,)
    else:
        return (0,)
if S == 0:
    return (0,) * len(a)
if S == a[0]:
    return (1,) + (0,) * (len(a)-1)
# et les cas récursifs
if S < a[0]:
    return (0,) + sacados(a[1:], S)
x = sacados(a[1:], S-a[0])
if sum(x) > 0:
    return (1,) + x
x = sacados(a[1:], S)
return (0,) + x
```

(b) L'algorithme a une complexité en 2^n où n est le nombre d'éléments. C'est une complexité exponentielle, ce qui est considéré comme peu efficace. On peut donc dire que le problème est difficile.

Question 2.6. *n*-uplets super-croissants

(a) On part par exemple de 1 et on croit le moins vite possible : 1,2,4,8,16,32,64,128,256,512.

(b)
```
def supercroissant(a):
    somme = a[0]
    for i in range(1, len(a)):
        if a[i] <= somme:
            return False
        somme += a[i]
    return True
```

(c) Supposons que l'entier i_{max} n'appartienne pas à I : alors I est inclus dans $\{1, \dots, i_{max} - 1\}$ (car tous les a_i pour $i > i_{max}$ sont supérieurs à S).
Comme le n-uplet est super-croissant, $\sum_{i<i_{max}} a_i < a_{i_{max}}$. Mais comme $a_{i_{max}} \leq S$, on a $\sum_{i \in I} a_i \leq \sum_{i<i_{max}} a_i < a_{i_{max}} \leq S$. Cela contredit le fait que $\sum_{i \in I} a_i = S$.

(d) Il suffit donc de chercher le plus grand entier du n-uplet qui est inférieur à S, puis de diminuer S de sa valeur ; en itérant, on obtient une solution si et seulement s'il en existe une. La complexité de l'algorithme est linéaire en n puisque la valeur de i diminue de 1 à chaque passage dans la boucle. La complexité est donc exponentiellement meilleure que celle du cas général.

```
def sacados_supercroissant(a, S):
    i = len(a) - 1
    t = tuple()
    while i >= 0:
        while a[i] > S and i >= 0:
            i -= 1
            t = (0,) + t
        if i > 0:
            S -= a[i]
            t = (1,) + t
        i -= 1
    return t
```

Question 2.7. Chiffrement de Merkle et Hellman

(a)
```
def supercroissant_aleatoire(n, M):
    t = [0] * n
    t[0] = random.randint(1, M)
    somme = t[0]
    for i in range(1, n):
        t[i] = random.randint(somme + 1, M*somme)
        somme += t[i]
    return tuple(t)
```

(b)
```
def clefs_MH(n, M):
    a = supercroissant_aleatoire(n, M)
```

```
    s = sum(a)
    u = random.randint(s+1, M*s)
    d = 0
    while d != 1:
        v = random.randint(1, u-1)
        d = fractions.gcd(u, v)
    return (tuple((v * aa) % u for aa in a), (u, v))
```

(c)
```
def chiffre_MH(pk, message):
    return sum(pk[i]*m[i] for i in range(len(pk)))
```

Question 2.8. Inverse modulaire

```
def euclide_etendu(u, v):
    if v == 0:
        return (u, 1, 0)
    q = u // v
    r = u % v
    (d, s1, t1) = euclide_etendu(v, r)
    return (d, t1, s1 - t1*q)
```

Attention, ici (d, s, t) dénote le triplet renvoyé par `euclide_etendu(u,v)`. On ne sait rien de ces nombres, et on cherche justement à prouver leurs propriétés.

(a) Montrons d'abord l'égalité mentionnée en indication. Pour cela, on démontre que tout diviseur commun à x et y divise $(x \bmod y)$, et que tout diviseur commmun à y et $(x \bmod y)$ divise également x. Cela suffira à conclure puisque le PGCD fait partie des diviseurs communs. Soit $x = qy + r$ la division euclidienne de x par y (donc $r = x \bmod y$). Soit d un diviseur commun à x et y. On peut écrire $x = dx'$ et $y = dy'$, donc $dx' = qdy' + r$. Autrement dit, $r = dx' - qdy'$ et d divise donc r. De même, soit d un diviseur commun à y et r. On écrit $y = dy'$ et $r = dr'$, d'où $x = qdy' + dr'$ et d divise donc x.

Le résultat est alors démontré par une récurrence triviale : le cas de base est correct (le PGCD de u et 0 est bien u) ; et le cas récursif correspond directement à l'indication.

(b) On effectue une récurrence sur v. Le cas de base est le cas $v = 0$: on a bien $u = 1 \times u + 0 \times v$. Supposons donc que l'appel récursif `euclide_etendu(v, r)` (où r est bien $< v$) renvoie (d, s_1, t_1) tels que $d = s_1 v + t_1 r$. Il reste à démontrer que le triplet renvoyé par l'algorithme est correct, c'est-à-dire que $d = t_1 u + (s_1 - t_1 q)v$. C'est un simple calcul en utilisant le fait que $u = vq + r$:

$t_1 u + (s_1 - t_1 q)v = t_1(vq + r) + (s_1 - t_1 q)v = t_1 vq + t_1 r + s_1 v - t_1 vq = t_1 r + s_1 v = d$ par hypothèse de récurrence.

(c) s et t sont les coefficients de Bézout de u et v.

(d)
```
def inverse_mod(v, u):
    d, s, t = euclide_etendu(u, v)
    if d != 1:
        raise ValueError("v n'est pas inversible modulo u.")
    return t % u
```

Question 2.9. Déchiffrement du système de Merkle et Hellman

(a) Si c est obtenu en chiffrant m avec la clef pk, alors $c = \sum_i m_i k_i$ par définition. On cherche donc à inverser cette équation, et exprimer m en fonction de c. En prenant les notations du sujet, et en travaillant modulo u, on a $c' = wc = w\sum_i k_i m_i$, et donc $c' = \sum_i (wk_i)m_i$. Comme $a_i = wk_i$ (toujours modulo u), on obtient que $c' = \sum_i a_i m_i$. On retrouve donc bien les m_i en résolvant le sac-à-dos d'entrées $(a_1, \ldots, a_n)$ et c'.

(b) $73 \times 70 = 5110 = 39 \times 131 + 1$, ce qui signifie bien que 73 est l'inverse de 70 modulo 131. On calcule donc les a_i (par produit des k_i avec w modulo u) qui valent ici (2,4,8,16,32,64) : c'est bien un n-uplet super-croissant. On résout le sac à dos avec $c' = 67\text{Ö}73 \bmod 131 = 44$ (facile vu le n-uplet), ce qui donne le message $(0, 1, 1, 0, 1, 0)$.

(c)
```
def dechiffre_MH(pk, sk, chiffre):
    u, v = sk
    w = inverse_mod(v, u)
    c = (w*chiffre) % u
    a = tuple((w*k)%u for k in pk)
    return sacados_supercroissant(a, c)
```