Informatique PT. 2024 - 2025

Mathématiques & Informatique - F. Gaunard
http://frederic.gaunard.com
Lycée Voltaire, Paris 11e.

1 Travaux Pratiques : Révisions d'algorithmique

Exercice 1. *Solution*

Donner une fonction récursive qui détermine si un *flottant* appartient à une liste **triée**, en effectuant une recherche par dichotomie.

```
def recherche_flottant(x, L):
    if len(L)==0 :
        return False
    else :
        p=len(L)//2
        if L[p]==x :
            return True
        if x< L[p] :
            return recherche_flottant(x, L[0:p])
        return recherche_flottant(x, L[p+1, len(L)])
```

Dans ce cas, si la liste d'entrée L est triée et de longueur $n = 2^p$, on obtient une complexité en $O(p) = O(\log_2(n))$. En effet, l'algorithme consiste à couper la liste en deux à chaque étape (à peu de choses près...), si bien qu'au bout de p étapes on s'est ramené à une liste de longueur $2^0 = 1$, et l'algorithme renvoie alors le résultat. D'où une complexité en $O(p)$.

Exercice 2. *Solution*

Écrire une fonction d'en-tête `def Doublon(L)` qui prend en entrée une liste `L` et qui renvoie `True` si elle contient deux fois un même élément, et `False` sinon.
Quelle est la complexité de votre fonction?

```
def Doublon(L):
    n=len(L)
    for i in range(n):
        for j in range(i+1, n):
            if L[j]==L[i]:
                return True
    return False
```

☞ On y utilise le fait que, lors de l'appel d'une fonction, l'exécution de la commande `return` interrompt le programme.

Pour chaque valeur de $i \in [\![0, n-1]\!]$, on effectue $n-i-1$ comparaisons, donc le nombre total de comparaisons effectuées est

$$\sum_{i=0}^{n-1}(n-i-1) = \sum_{j=0}^{n-1} j = \frac{(n-1)n}{2} = O(n^2)$$

comparaisons au total (on a ainsi une complexité quadratique).

Exercice 3. ***Solution*** **Liste des nombres premiers**

On rappelle que la commande `a//b` renvoie le quotient dans la division euclidienne de a par b, et `a%b` le reste.

1. Écrire une fonction d'en-tête `def diviseurs(n)` qui prend en entrée un entier positif n et renvoie la liste des diviseurs positifs de n.

```
def diviseurs(n): # fonction en O(n)
    L=[]
    for k in range(1, n+1):
        if n%k==0 :
            L.append(k)
    return L
```

2. Quelle est la complexité de votre fonction? Pouvez-vous donner une fonction en $O(\sqrt{n})$?

   L'algorithme précédent réalise exactement n calculs de restes de divisions euclidiennes, n comparaisons et au plus n ajouts d'un élément dans une liste. On peut donc penser que son temps d'exécution va être proportionnel à n.

   On peut améliorer la vitesse d'exécution de cet algorithme en remarquant que si $n = pq$ avec $p \geqslant \sqrt{n}$, alors q est un diviseur de n et $q \leqslant \sqrt{n}$. Il suffit donc de chercher chaque diviseur $q \leq \sqrt{n}$, puis d'ajouter q et $p = n/q$ dans la liste des diviseurs (il faut encore faire attention au cas où n est un carré, et ne pas ajouter deux fois sa racine carrée à la liste des diviseurs). On obtient ainsi un algorithme en $O(\sqrt{n})$.

```
def diviseurs2(n): # fonction en O(sqrt(n))
    L=[]
    k=1
    while k**2 < n :
        if n%k == 0 :
            L.append(k)
            L.append(n//k)
        k=k+1
    if k**2==n :
        L.append(k)
    return L
```

3. En déduire une fonction `def premier` qui prend en entrée un entier n puis renvoie `True` s'il est premier, et `False` sinon.

```
def premier(n):
    return len(diviseurs2(n))==2
```

4. Afficher à l'écran la liste des nombres premiers inférieurs à 1000. Combien y en a-t-il?

   La fonction suivante permet d'afficher la liste des nombres premiers inférieurs à N entré en argument.

```
def liste_nombres_premiers(N):
    L=[]
    for k in range(1, N+1):
        if premier(k) :
            L.append(k)
    return L
```

   Avec $N = 1000$ et une commande`len` sur la résultat en sortie, on a la réponse à la question posée.

```
>>> print(len(liste_nombres_premiers(1000)))
168
```

Exercice 4. *Solution* — **Algorithme d'Euclide**

Implémenter l'algorithme d'Euclide, qui calcule le plus grand commun diviseur de deux entiers a et b non tous deux nuls, en se basant sur les propriétés suivantes :

i. $\text{pgcd}(a, 0) = a$,
ii. $\text{pgcd}(a, b) = \text{pgcd}(b, r)$ où r est le reste de la division euclidienne de a par $b \neq 0$.

```
def euclide(a, b):
    if b==0:
        return a
    return euclide(b, a%b)
```

Exercice 5. *Solution* — **Plus grande somme d'une sous-liste**

On considère une liste `L` d'entiers. On veut écrire un programme la plus grande valeur possible de la somme des termes consécutifs d'une sous-liste de L.

Par exemple, on devra avoir l'affichage suivant lors de l'exécution ci-dessous.

```
>>> recherche_s_liste([2, -2, -3, 2, 3, -4, 2, 3, -1])
6
```

On commence par initialiser une valeur maximale égale au premier terme de la liste. Puis partant de chaque terme on calcule toutes les sommes consécutives correspondant aux sous-listes successives. Si une somme calculée est supérieure à la valeur du max, cette valeur est actualisée.

```
def recherche_s_liste(L):
    longueur = len(L)
    max=L[0] # initialisation du max premier terme de la liste
    for k in range(longueur):
        s=L[k] # somme initialisée
        if s > max :
            max = s
        for j in range(k+1, longueur): # on parcourt les termes qui suivent L[k]
            s=s+L[j] # on actualise la somme consécutive
            if s > max : # si cette somme dépasse le max actuel
                max=s # elle devient le max
    return max
```

Exercice 6. *Solution* — **Suite et codage de Fibonacci avec le théorème de Zeckendorf**

On rappelle que la suite de Fibonacci (F_n) est la suite de nombre réels définis par

$$F_0 = 0,\ F_1 = 1, \quad \text{et} \quad \forall n \in \mathbb{N},\ F_{n+2} = F_{n+1} + F_n.$$

Les premiers termes de la suite sont $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...$

Théorème 1. — **Théorème de Zeckendorf**

Le **Théorème de Zeckendorf** dit la chose suivante.
Pour tout entier non nul n, il existe un unique entier k et un unique $k-$uplet d'entiers $(c_1, .., c_k)$ vérifiant

- ✗ $c_1 \geq 2$
- ✗ $c_i + 1 < c_{i+1}$

tels que

$$n = \sum_{i=1}^{k} F_{c_i}$$

Cette décomposition de n en somme de nombres de la suite de Fibonacci est appelée *décomposition de Zeckendorf de* n.

Par exemple $17 = 13 + 3 + 1 = F_7 + F_4 + F_2$ donc $k = 3$ et $\{c_1, c_2, c_3\} = \{2, 4, 7\}$.
Le but de cet exercice est d'écrire un programme qui renvoie cette décomposition pour un nombre n pris en argument.

1. Écrire une fonction récursive `Fibonacci(k):` qui renvoie le terme F_k de la suite (F_n).

Question classique de l'écriture d'une fonction récursive.

```
def Fibonacci(k):
    if k==0 :
        return 0
    if k==1  :
        return 1
    return Fibonacci(k-1)+Fibonacci(k-2)
```

2. Écrire une fonction récursive `Zeckendorf(n):` qui renvoie la décomposition de Zeckendorf de n.
On n'oubliera pas de réordonner la liste pour que les termes soient dans l'ordre croissant.

On va commencer par écrire une fonction qui renvoie la liste des nombres de Fibonacci inférieurs à un flottant x pris en argument.

```
def liste_fibo(n):
    k=0
    L=[]
    while Fibonacci(k)<=n :
        L.append(Fibonacci(k))
        k=k+1
    return L
```

L'idée est alors la suivante ; si le nombre à décomposer est déjà un nombre de Fibonacci (donc présent dans la liste des nombres de Fibonacci qui lui sont inférieurs), la décomposition ne contiendra que l'indice de ce nombre (dans la suite des nombre de Fibonacci).
Sinon, on lui soustrait le plus grand nombre de Fibonacci qui lui est inférieur, et on répète le processus. Cela donne la fonction suivante :

```
def Zeckendorf(n) :
    L=liste_fibo(n)
    Z=[]
    f=len(L)-1
    Z.append(f)
    if n in L :
        return Z
    else :
        return Z+Zeckendorf(n-L[f])
```

On remet les termes dans l'ordre croissant (ils sont dans l'ordre décroissant vu qu'on part de la droite) en écrivant d'abord une fonction qui inverse l'ordre des éléments d'une liste, en utilisant une énumération par la droite

```
def inversion(L):
    inv=[]
    for k in range(1, len(L)+1):
        inv.append(L[-k])
    return inv

def Zeckendorf_ordre_croissant(n):
    return inversion(Zeckendorf(n))
```

☞ On peut même tester que ça marche en vérifiant qu'on retombe sur le nombre de départ en additionnant les nombres de Fibonacci dont les indices sont dans la décomposition :

```
def test_Z(n):
    L=Zeckendorf(n)
    return somme([Fibonacci(k) for k in L])==n
```

Un caractère est représenté en ASCII par un nombre. Par exemple, le caractère « A » est représenté par le nombre 65, « B » par 66, etc.. La commande `ord(c)` renvoie la valeur du nombre qui code le caractère `c` en ASCII. Réciproquement, la commande `chr(n)` renvoie le caractère codé par n.

Le théorème de Zeckendorf permet alors d'effectuer le *codage de Fibonacci* d'un nombre et donc d'un caractère. Plus précisément, connaissant la décomposition $Z_n = \{c_1, .., c_k\}$ de Zeckendorf d'un entier n, on peut écrire

$$n = \sum_{i=1}^{k} \alpha_i F_{c_i}, \qquad \text{où} \qquad \alpha_i = \begin{cases} 1, & \text{si } c_i \in Z_n \\ 0, & \text{sinon.} \end{cases}$$

ce qui va permettre de coder chaque caractère avec une suite de 0 et de 1.
Le théorème assure notamment que le codage ne peut pas comporter deux 1 côte-à-côte (il n'y a jamais deux nombres de Fibonacci consécutifs dans la décomposition). Ainsi, pour permettre de détacher les lettres en identifiant la fin du codage d'un caractère, on rajoute un 1 à la fin de la décomposition.
Par exemple, comme $65 = F_3 + F_6 + F_{10}$, le caractère « A » est codé par 000100100011.

3. Écrire des fonctions de codage et de décodage (selon le code de Fibonacci) d'une chaîne de caractères.

Il faut commencer par une fonction qui permet de coder un caractère. On récupère la valeur numérique en ASCII du caractère, on fait sa décomposition de Zeckendorf puis si les indices successifs de tous les nombres de Fibonacci sont dans la décomposition on ajoute un $'1'$ à la chaîne de caractère, sinon un $'0'$, le tout par concaténations successives. On rajoute un $'1'$ à la fin.

```
def codage_caractere_fib(c):
    n=ord(c)
    Z=Zeckendorf(n)
    L=liste_fibo(n)
    l=len(L)
    codage=[]
    for k in range(l):
        if k in Z :
            codage.append('1')
        else :
            codage.append('0')
    codage.append('1')
    code=''
    for x in codage :
        code=code+x
    return code
```

On peut ensuite écrire une fonction pour coder un mot ou un message, où l'on parcourt tous les termes de la suite de caractère à coder.

```
def codage_fibonacci(mot):
    codage=''
    for x in mot :
        codage=codage+codage_caractere_fib(x)
    return codage
```

Pour décoder, il faut commencer à décoder un caractère. Chaque $'1'$ (à part le dernier) en position k traduit la présence du $k-$ième nombre de Fibonacci dans la décomposition de Zeckendorf. On les additionne pour trouver le nombre auquel correspond le caractère qu'on retrouve avec la commande `chr()`.

```
def decodage_caractere_fib(code):
    x=0
    for k in range(len(code)-1):
        if code[k]=='1':
            x=x+Fibonacci(k)
    return chr(x)
```

Il faut ensuite découper le message. Dès qu'on a deux 11 à la suite, c'est la fin d'un caractère.

```
def decodage_fibonacci(message):
    n=len(message)
    solution=''
    i=0
    l=1
    while i<n :
        j=i
        while message[j] !='1' or message[j+1] !='1' :
            j=j+1
        solution=solution+decodage_caractere_fib(message[i:j+2])
        i=j+2
    return solution
```

4. Décoder le message.

```
>>> decodage_fibonacci('00101010001100101010000110000
10101100000010101100101000101100001010110010101000
01100000100100110010010010011000010101100000100100
11000101001001100010000100110010101000011001010001 0
0110000101011001010101 1')
'Ce TP est super !'
```

Exercice 7. ***Solution*** **Matrices stochastiques**

On dit qu'une matrice carrée $M \in \mathcal{M}_n(\mathbb{R})$ est *stochastique* si tous ses coefficients sont positifs ou nuls et si, la somme de chacune de ses lignes vaut 1.
Écrire une fonction `def test_stochastique(M)` qui prend en argument une matrice M (implémentée sous la forme d'un tableau `ndarray` 2D) et qui renvoie `True` ou `False` selon que celle-ci est stochastique ou non.

On pourra commencer par écrire une fonction récursive `somme(L)` qui renvoie la somme des termes d'une liste.

Sans difficulté pour la fonction récursive.

```
def somme(L):
    if len(L)==0 :
        return 0
    else :
        return L[0] + somme(L[1:len(L)])
```

Ensuite, on renvoie `False` dès qu'on tombe sur un terme négatif ou sur une ligne dont la somme ne vaut pas 1. Sinon, on renvoie `True`.

```
def test_stochastique(M):
    n=len(M)
    for i in range(n):
        if somme(M[i]) != 1 :
            return False
        for j in range(n):
            if M[i][j]<0 :
                return False
    return True
```

Exercice 8. ***Solution*** **Tri à bulles ou *bubble sort***

L'algorithme de tri à bulles n'est, en pratique, guère utilisé. C'est néanmoins un exemple classique, car il est particulièrement formateur. Le principe sous-jacent est de faire remonter progressivement vers la droite les plus grands termes de la liste considérée, à la manière des plus grosses bulles d'air qui sont les premières à remonter à la surface d'un liquide.

Plus précisément, le principe est le suivant:

i. on parcourt les éléments k premiers éléments de la liste (en commençant avec k qui vaut la longueur de la liste), et à chaque fois qu'un élément est plus grand que le suivant, on les échange (à l'issue de cette opération, le plus grand élément est alors en dernière position);
ii. on recommence l'opération précédente, en ne considérant pas le dernier élément de la liste, puis en ne considérant pas les deux derniers éléments etc... jusqu'à ce que la liste soit entièrement triée.

1. Écrire une procédure Python qui implémente le tri à bulles en place. On commencera par écrire une procédure `remonte` qui implémente l'étape *i.*

```
def remonte(L, k) :
    i=0
    while i < k-1 :
        if L[i] > L[i+1] :
            L[i+1], L[i] = L[i], L[i+1]
        i=i+1

def tri_a_bulles(L) :
    n=len(L)
    for k in range(n):
        remonte(L, n-k)
    return L
```

2. Combien de comparaisons sont nécessaires lors de l'exécution de cet algorithme? Quelle est la complexité du tri à bulles?

Soit n la longueur de la liste d'entrée. L'exécution de `remonte(L, n-k)` implique $n-k-1$ comparaisons, et comme on appelle cette fonction pour k allant de 0 à $n-1$, cela donne

$$\sum_{k=0}^{n-1}(n-k-1) = \sum_{j=0}^{n-1} j = O(n^2)$$

comparaisons au total. On a ainsi une complexité quadratique, en $O(n^2)$.

3. Modifier la procédure pour obtenir une complexité en $O(n)$ dans le meilleur des cas.

Le meilleur des cas est celui où la liste est déjà triée. Pour le moment, notre procédure a toujours une complexité quadratique dans ce cas. L'idée est, lorsque l'on appelle la fonction `remonte`, de vérifier si celle-ci effectue ou pas des permutations de termes: si non, cela signifie que la liste est triée, et on peut donc s'arrêter. On utilise pour ce faire une variable `test` que l'on passe à `True` lorsque l'on effectue une permutation de termes.

```
def remonte2(L, k) :
    i=0
    test=False
    while i < k-1 :
        if L[i] > L[i+1] :
            L[i+1], L[i] = L[i], L[i+1]
            test=True
        i=i+1
    return test

# si remonte2 renvoie False, la liste est déjà triée

def tri_a_bulles2(L) :
    n=len(L)
    k=0
    test=True
```

```
    while k<n and test :
        remonte2(L, n-k)
        k=k+1
    return L
```

4. Application: en déduire un programme calculant la médiane d'une série de données contenue dans une liste.

   Il suffit de trier la liste (sans la modifier, elle pourrait resservir dans la suite), puis d'extraire la médiane, en distinguant les cas où la longueur de la série est paire ou impaire (attention au fait que les termes de la liste sont indexés de 0 à $n-1$). C'est une application classique des algorithmes de tri.

```
def mediane(L):
    tri_a_bulles2(L)
    n=len(L)
    if n%2==1:
        return L[n//2]
    return (L[n//2-1]+L[n//2])/2
```

Exercice 9. *Solution* **Tri par dénombrement ou *counting sort***

Soit $N \in \mathbb{N}^*$. On cherche à trier une liste `L` d'entiers positifs strictement inférieurs à N.
Certaines valeurs peuvent apparaître plusieurs fois dans la liste initiale, et devront donc apparaître autant de fois dans la liste triée.

1. Écrire une fonction `Comptage`, d'arguments `L` et `N`, renvoyant une liste `P` de longueur N dont le k-ième élément est le nombre d'occurrences de l'entier k dans `L`.

   On pré-remplit une liste de N zéros correspondant aux occurrences de chaque entier. On remplit, en parcourant la liste, directement cette liste des *effectifs*.

```
def Comptage(L, N):
    P=[0]*N
    for k in range(len(L)):
        P[L[k]]+=1
    return P
```

2. En utilisant la fonction `Comptage`, écrire une fonction `Tri_par_denombrement`, d'arguments `L` et `N`, renvoyant une liste correspondant au tri de `L`.

   En partant d'une liste vide, on concatène un nombre de copies de chaque entiers selon l'effectif correspondant dans la liste de comptage, jusqu'à avoir une liste de la longueur de la liste initiale.

```
def Tri_par_denombrement(L, N):
    P=Comptage(L, N)
    n=len(L)
    T=[]
    k=0
    while len(T)<n :
        T=T+[k]*P[k] # on rajoute le bon nombre d'occurrences de chaque valeur
        k=k+1
    return T
```

3. Quelle est la complexité de cette fonction de tri (en fonction de la longueur n de `L`)? Commenter.

   On fait toujours N tours de boucle dans la fonction de comptage, puis, au pire (si les valeurs sont toutes différentes), n tours dans la boucle `while`. On a donc une complexité en $O(n+N)$, c'est à dire linéaire, ce qui est le tri le plus rapide (on troque, en quelque sorte, du temps de calcul contre de la mémoire).
   Il nécessite quand même une liste de valeurs entières et donc on connait les bornes min et max...