Révisions d'algorithmique

Informatique #01

Partie A – Listes

Exercice 1 — échange d'éléments
Écrire une fonction `echange(L, i, j)` qui échange les deux éléments d'indices `i` et `j` d'une liste `L`.

```
In [1]: def echange(L, i, j):
            L[i], L[j] = L[j], L[i]
            return L
In [2]: echange([0, 1, 2, 3, 4, 5], 2, 5)
Out[2]: [0, 1, 5, 3, 4, 2]
```

Exercice 2 — moyenne et variance d'une liste de valeurs
Écrire deux fonctions `moyenne(L)` et `variance(L)` qui retournent respectivement la moyenne et la variance d'une liste de flottants `L` sans utiliser la fonction `sum`.

```
In [3]: def moyenne(L):
            n = len(L)
            s = 0
            for i in range(n):
                s += L[i]
            return s/n
In [4]: moyenne([0, 1, 2, 2, 2, 4, 4])
Out[4]: 2.142857142857143
In [5]: def variance(L):
            L2 = [a**2 for a in L]
            return moyenne(L2) - moyenne(L)**2
In [6]: variance([0, 1, 2, 2, 2, 4, 4])
Out[6]: 1.8367346938775517
```

Exercice 3 — liste croissante
Écrire une fonction `croissante(L)` qui retourne `True` si la liste d'entiers `L` est croissante, `False` sinon.

```
In [7]: def croissante(L):
            n = len(L)
            for i in range(n-1):
                if L[i] > L[i+1]:
                    return False
            return True
In [8]: croissante([0, 1, 2, 3])
Out[8]: True
In [9]: croissante([0, 3, 5, 4, 5])
Out[9]: False
```

Exercice 4 — présence de doublons
Écrire une fonction `doublon(L)` qui détecte la présence d'un doublon dans une liste `L` en utilisant puis sans utiliser la méthode `count`. Déterminer la complexité temporelle des deux fonctions.

```
In [10]: def doublon(L):
             for i in range(len(L)):
                 if L.count(L[i])>1:
                     return True
             return False
In [11]: doublon([0, 1, 2, 2, 1, 4])
Out[11]: True
In [12]: doublon([0, 1, 2, 3, 4])
Out[12]: False
In [13]: def doublon2(L):
             for i in range(len(L)-1):
                 if L[i] in L[i+1:]:
                     return True
             return False
In [14]: doublon2([0, 1, 2, 2, 1, 4])
Out[14]: True
In [15]: doublon2([0, 1, 2, 3, 4])
Out[15]: False
```

Exercice 5 — maximum d'une liste, valeur et occurence
Écrire une fonction `recherchemax(L)` qui retoune la valeur maximum d'une liste

d'entiers L, la position de la première valeur correspondante et son nombre d'occurence.

```
In [16]: def recherchemax(L):
             maxi, pos, occ = L[0], 0, 1
             for i in range(1,len(L)):
                 if L[i] > maxi:
                     maxi, pos, occ = L[i], i, 1
                 else:
                     if L[i] == maxi:
                         occ += 1
             print('Le maximum vaut', maxi, '; il apparaît en i =',
                           pos, '; il est présent', occ, 'fois.')
In [17]: import random
         L = [random.randint(0,9) for k in range(15)]
         print(L)
         recherchemax(L)
[2, 7, 6, 3, 2, 5, 6, 1, 2, 6, 8, 5, 4, 5, 1, 5]
Le maximum vaut 8 ; il apparaît en i = 10 ; il est présent 1 fois.
```

Exercice 6 — distance maximale dans une liste

Écrire une fonction `distancemax(L)` qui retourne le plus grand écart (en valeur absolue) entre deux valeurs d'une liste d'entiers L.

```
In [18]: def distancemax(L):
             n = len(L)
             d = 0
             for i in range(n):
                 for j in range (i, n):
                     if abs(L[i]-L[j]) > d:
                         d = abs(L[i]-L[j])
             return d
In [19]: L = [random.randint(0,30) for k in range(10)]
         print(L)
         distancemax(L)
[27, 7, 29, 20, 18, 22, 2, 29, 3, 12]
Out[19]: 27
In [20]: max(L) - min(L)
Out[20]: 27
```

Partie B – Arithmétique

Exercice 7 — tables de multiplications

Écrire une fonction `tablemult(n)` qui retourne la table de multiplications associée à l'entier n.

```
In [21]: def tablemult(n):
             for i in range(10):
                 print(n,'*',i,' = ',n*i)
In [22]: tablemult(7)
7 * 0  =  0
7 * 1  =  7
7 * 2  =  14
7 * 3  =  21
7 * 4  =  28
7 * 5  =  35
7 * 6  =  42
7 * 7  =  49
7 * 8  =  56
7 * 9  =  63
```

Exercice 8 — années bissextiles

Dans le calendrier grégorien, on appelle année bissextile une année dont le numéro est soit divisible par 4 et non divisible par 100, soit divisible par 400.

1. Écrire une fonction `estbissextile(n)` qui retourne `True` si l'année n est bissextile, `False` sinon.
2. Écrire une fonction `suivbissextile(n)` qui retourne la première année bissextile après l'année n.

```
In [23]: def estbissextile(n):
             if (n % 4 == 0) and (n % 100 != 0):
                 return True
             elif (n % 400 == 0):
                 return True
             else:
                 return False
In [24]: [estbissextile(1900),estbissextile(2000),
             estbissextile(2014), estbissextile(2016)]
Out[24]: [False, True, False, True]
```

```
In [25]: def estbissextile2(n): # En plus compact !
             return ((n % 4 == 0) and (n % 100 != 0))
                                  or (n % 400 == 0)
In [26]: [estbissextile2(1900),estbissextile2(2000),
          estbissextile2(2014), estbissextile2(2016)]
Out[26]: [False, True, False, True]
In [27]: def suivbissextile(n):
             n += 1
             while not(estbissextile2(n)):
                 n += 1
             return n
In [28]: suivbissextile(1896), suivbissextile(2014)
Out[28]: (1904, 2016)
```

Exercice 9 — paiement en euros

On souhaite payer une certaine somme en euros à l'aide (uniquement) de pièces de 2 euros, de 1 euro et de billets de 5 euros. Écrire une fonction `euros` prenant en entrée un entier `n` et renvoyant le nombre de pièces et de billets correspondant.

```
In [29]: def euros(n):
             b = n // 5
             r = n % 5
             p2 = r // 2
             p1 = r % 2
             print(b, 'billet(s) de 5 euros ;',
                         p2, 'pièce(s) de 2 euros ;',
                               p1, 'pièce de 1 euro.')
In [30]: euros(33)
6 billet(s) de 5 euros ; 1 pièce(s) de 2 euros ; 1 pièce de 1 euro.
```

Exercice 10 — diviseurs et nombres parfaits

1. Écrire une fonction `diviseurs(n)` qui retourne la liste des diviseurs entiers positifs d'un entier naturel `n`.
2. Un entier naturel `n` est dit *parfait* si la somme de ses diviseurs vaut 2`n`. Écrire une fonction `parfait(n)` qui détecte si l'entier `n` est parfait ou non.
3. Écrire une fonction `listeparfait(n)` qui retourne la liste des entiers parfaits inférieurs ou égaux à `n`.
4. Retourner la liste des entiers $2^{p-1}(2^p-1)$ inférieurs à 10000 pour 2^p-1 premier. Comparer avec `listeparfait(10000)`.

```
In [31]: def diviseurs(n):
             L = []
             for i in range(1, n+1):
                 if (n % i == 0):
                     L += [i]
             return L
In [32]: def diviseurs2(n):
             return [d for d in range(1, n+1) if (n % d == 0)]
In [33]: diviseurs(24)
Out[33]: [1, 2, 3, 4, 6, 8, 12, 24]
In [34]: diviseurs2(24)
Out[34]: [1, 2, 3, 4, 6, 8, 12, 24]
In [35]: def parfait(n):
             L = diviseurs2(n)
             return (sum(L) == 2*n)
In [36]: parfait(5), parfait(6)
Out[36]: (False, True)
In [37]: def listeparfait(n):
             return [a for a in range(1, n+1) if parfait(a)]
In [38]: listeparfait(10000)
Out[38]: [6, 28, 496, 8128]
In [39]: def estpremier(n): # méthode barbare
             return len(diviseurs(n)) == 2
In [40]: estpremier(13)
Out[40]: True
In [41]: [2**(p-1)*(2**p-1) for p in range(1,8) if estpremier(2**p-1)]
Out[41]: [6, 28, 496, 8128]
```

Exercice 11 — nombres premiers et crible d'Eratosthène

1. Écrire une fonction `estpremier(n)` qui détecte la primalité d'un entier `n` donné.
2. En déduire une fonction `listepremier(n)` qui retourne une liste des

nombres premiers compris entre 2 et n.

3. Écrire, sans utiliser les fonctions précédentes, une fonction `eratosthene(n)` qui retourne une liste de tous les nombres premiers compris entre 1 et n à l'aide de la méthode du crible d'Ératosthène.

```
In [42]: def estpremier(n):
             for i in range(2,n):
                 if (n % i == 0):
                     return False
             return True
In [43]: def listepremier(n):
             return [i for i in range(2,n) if estpremier(i)]
In [44]: listepremier(50)
Out[44]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
In [45]: def eratosthene(n):
             L = [True]*(n+1) # Les entiers entre 0 et n
                              sont supposés premiers par défaut
             L[0] = L[1] = False # 0 et 1 ne sont pas premiers
             for i in range(2, n+1): # on parcourt la liste
                 if L[i] == True: # si l'entier est premier,
                                  ses multiples ne le sont pas
                     for j in range(2*i, n+1, i): # on les élimine
                         L[j] = False
             L2 = [i for i in range(n+1) if L[i] == True]
             return L2 # on a converti L en liste d'entiers premiers
In [46]: print(eratosthene(40))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

Exercice 12 — algorithme d'Euclide

Écrire une fonction `euclide(a, b)` qui retoune le pgcd de deux entiers naturels a et b à l'aide de l'algorithme d'Euclide.

```
In [47]: def euclide(a, b):
             while b != 0:
                 print((a, b))
                 a, b =  b, a % b
             return a
In [48]: euclide(29029, 6090)
(29029, 6090)
(6090, 4669)
(4669, 1421)
(1421, 406)
(406, 203)
Out[48]: 203
```

Partie C – Chaînes de caractères

Exercice 13 — palindromes

Écrire une fonction `palindrome(mot)` qui détecte la présence d'un palindrome.

```
In [49]: def palindrome(mot):
             L = list(mot)
             L2 = L[:]
             L2.reverse()
             return L == L2
In [50]: palindrome('tot'), palindrome('tard')
Out[50]: (True, False)
In [52]: palindrome('eluparcettecrapule')
Out[52]: True
```

Exercice 14 — somme des carrés des chiffres d'un nombre

1. Écrire une fonction `decompose(n)` qui prend en entrée un entier n et retourne une liste constituée de ses chiffres.
2. En déduire une fonction `carres(n)` qui détermine la somme des carrés des chiffres d'un entier n donné.

```
In [53]: def decompose(n):
             return [int(a) for a in list(str(n))]
In [54]: decompose(1234)
Out[54]: [1, 2, 3, 4]
In [55]: def carres(n):
             L = decompose(n)
             return sum([a**2 for a in L])
In [56]: carres(1234)
Out[56]: 30
```

Exercice 15 — recherche de chaînes de caractères

1. Écrire une fonction `rech1(mot, phrase)` qui retourne `True` si le mot `mot` est présent dans la phrase `phrase` et précise la position de celui-ci; `False` sinon.
   On pourra utiliser les méthodes relatives aux chaînes de caractères.
2. Écrire une fonction `position(mot, phrase, i)` qui retourne `True` si le mot est présent dans la phrase, en position `i`, SANS utiliser de méthode.
3. En déduire une fonction `rech2(mot, phrase)` renvoyant le même résultat que la fonction `rech1` SANS utiliser de méthode.
4. Pour finir, écrire enfin une fonction `rech3(mot, phrase)` qui renvoie une liste vide si le mot n'est pas contenu dans la phrase ou une liste précisant la/les position(s) du mot dans la phrase.

```
In [57]: def rech1(mot, phrase):
             if mot in phrase:
                 return (True, phrase.index(mot))
             return (False, None)
In [58]: rech1('soir', 'accessoire')
Out[58]: (True, 5)
In [59]: rech1('matin', 'matignon')
Out[59]: (False, None)
In [60]: def position(mot, phrase, i):
             n, N = len(mot), len(phrase)
             k = 0
             while (i < N-n) and (k < n) and (mot[k] == phrase[i+k]):
                 k += 1
             return (k == n)
In [61]: position('soir', 'accessoire', 10)
Out[61]: False
In [62]: def rech2(mot, phrase):
             n, N = len(mot), len(phrase)
             for i in range(N - n):
                 if position(mot, phrase, i):
                     return (True, i)
             return (False, None)
```

```
In [63]: rech2('soir', 'accessoire')
Out[63]: (True, 5)
In [64]: rech2('matin', 'matignon')
Out[64]: (False, None)
In [65]: def rech3(mot, phrase):
             n, N = len(mot), len(phrase)
             return [i for i in range(N - n)
                             if position(mot, phrase, i)]
In [66]: rech3('soir', 'accessoiresoire')
Out[66]: [5, 10]
In [67]: rech3('matin', 'unmatinamatignon')
Out[67]: [2]
```

Exercice 16 — séquence nucléotidique

Une séquence d'ADN est une suite de 4 nucléotides désignés par les lettres : A pour l'adénine, G pour la guanine, T pour la thymine et C pour la cytosine.

1. Écrire une fonction `sequence(n)` qui retourne une séquence aléatoire d'ADN de n caractères.
2. Écrire une fonction `recherche(mot,phrase)` qui détecte la présence du mot `mot` dans la séquence d'ADN `phrase`.

```
In [68]: import random
         def sequence(n):
             Dico = ['A', 'G', 'T', 'C']
             L = [random.randint(0,3) for i in range(n)]
             return [Dico[a] for a in L]
In [69]: sequence(10)
Out[69]: ['C', 'A', 'A', 'G', 'T', 'T', 'G', 'C', 'G', 'T']
In [70]: ''.join(sequence(80))
Out[70]: 'CAATATAAGTTCATCTGAGGTAACATGCAGACTATTT
         CAGCGACGCATCCGTTCTAGGAAAAGGTGCTCTTCCTGGTCTA'
```

```
In [71]: def recherche(mot, phrase):
             n, N = len(mot), len(phrase)
             for i in range(N - n):
                 j = 0
                 while (j < n) and (phrase[i+j] == mot[j]):
                     j += 1
                 if j == n:
                     print(i)
                     return True
             return False
In [72]: phrase = sequence(10)
         print(''.join(phrase))
         recherche('TAC',phrase)
GCGCAGCGTG
Out[72]: False
In [73]: phrase = sequence(150)
         print(''.join(phrase))
         recherche('TAC',phrase)
ATAGACGCGAGCTTGGTCAGAGATTTTGAATCCCAAAGTCTAAGAGATGGCAGGC
          CGAGACATAACAATTCCACATACATTCATCCCGCTTAATAACATAAACGGA
                    CGCTAATGGAGCGGCAGCGCCATCGTACAAATACCGCGAGGGTT
75
Out[73]: True
```

Exercice 17 — anagrammes

Écrire une fonction `anagramme(mot1, mot2)` qui renvoie `True` si `mot1` et `mot2` sont des anagrammes, `False` sinon.

On pourra utiliser la méthode `sort` sur des listes bien choisies.

```
In [74]: def anagramme(mot1, mot2):
             L1 = list(mot1)
             L2 = list(mot2)
             L1.sort()
             L2.sort()
             return L1 == L2
In [75]: anagramme('chat', 'tache')
Out[75]: False
In [76]: anagramme('romain','marion')
Out[76]: True
```

Partie D – Miscellaneous

Exercice 18 — calcul de $\sum_{k=a}^{b} f(k)$

Écrire une fonction `somme(f, a, b)` qui retourne la valeur de $\sum_{k=a}^{b} f(k)$.

```
In [77]: def somme(f, a, b):
             s = 0
             for k in range(a,b+1):
                 s += f(k)
             return s
In [78]: somme(lambda x: x**2, 1, 5)
Out[78]: 55
```

Exercice 19 — série harmonique

Trouver n tel que $\sum_{k=1}^{n} \frac{1}{k} > 10$.

```
In [79]: n, s = 0, 0
         while s <= 10:
             n += 1
             s += 1/n
         print('On trouve n =', n, 'et s =', s)
On trouve n = 12367 et s = 10.000043008275778
```

Exercice 20 — évaluation d'un polynôme et méthode de Horner (☕)

On représente un polynôme $P = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ par sa liste de coefficients :

a_0	a_1	a_2	$\cdots$	$\cdots$	a_n

1. Écrire une fonction `eval1(P, a)` qui permet pour un polynôme `P` et un réel `a` donnés, de calculer le réel `P(a)`.
2. Écrire une fonction `eval2(P, a)` qui fait de même sans utiliser l'opérateur `**`. Combien de multiplications sont-elles effectuées ?

3. Écrire une fonction `horner(P, a)` qui calcule `P(a)` en se basant sur le principe suivant :

$$P(x) = a_0 + x \times (a_1 + x \times (a_2 + x \times \cdots (a_{n-1} + a_n \cdot x)))$$

Combien de multiplications sont cette fois-ci nécessaires ?

```
In [86]: def eval1(P, a):
             s = 0
             for i, coeff in enumerate(P):
                 s += coeff*a**i
             return s
In [87]: def eval2(P, a):
             s, puis = 0, 1
             for coeff in P:
                 s += coeff*puis
                 puis *= a
             return s
In [88]: P = [2, 1, 5] # P = 5*X**2+X+2
         a = 2
In [89]: print(eval1(P, a))
         print(eval2(P, a))
24
24
In [90]: def horner(P, a):
             val = 0
             for coeff in reversed(P):
                 val = val * a + coeff
             return val
In [91]: horner(P, a)
Out[91]: 24
```