

CORRECTION DU TD IPT² N° 1: RÉVISIONS 1/2

RAPPELS DE PROGRAMMATION ÉLÉMENTAIRE: STRUCTURES CONDITIONNELLES, BOUCLES, STRUCTURES DE DONNÉES

Procédures de bases - structures conditionnelles

EXERCICE N°1: Nombres parfaits

Listing 1:

```
1 ### Nombres parfaits ###
2 import numpy as np
3 def parfait(n):
4     sommediviseurs=1
5     for diviseur in range(2,n//2+1):#on balaie les diviseurs testés
6         de 2 à n//2+1
7         if n%diviseur==0:
8             sommediviseurs=sommediviseurs+diviseur
9     if n==sommediviseurs:
10        print u"Le nombre",n,u"est parfait!"
11    else:
12        print u"Le nombre",n,u"n'est pas parfait!"
13 n=0.0
14 while type(n)<>int or n<1:
15     n=input("Entrer un entier naturel supérieur ou égal à 1:")
16 parfait(n)
```

EXERCICE N°2: Suite ordonnée des nombres à petits diviseurs

Listing 2:

```
1 def suiteordo(N):
2     listefinale=[]
3     for nombre in range(1,N+1,1):
4         p,q,r=0,0,0
5         while nombre%2**p==0:
6             p=p+1
7         while nombre%3**q==0:
```

```
8         q=q+1
9         while nombre%5**r==0:
10            r=r+1
11         if nombre==2**(p-1)*3**(q-1)*5**(r-1):
12            listefinale.append([nombre,(p-1,q-1,r-1)])
13         print(u"La liste ordonnée est:"), listefinale
14
15 N=0.0
16 while (type(N)<>int) or (N<0):
17     N=input("Entrer un nombre entier positif:")
18 suiteordo(N)
```

EXERCICE N°3: Résolution d'une énigme par force brute

- ① On propose 3 variantes pour la fonction listecorrecte(L):

Listing 3:

```
1 def listecorrecte1(L):
2     if len(L)!=9:
3         return len(L)==9
4     else:
5         listeverif=[0]*9
6         for nb in L:
7             if nb>0 and nb
8             <10:
9                 listeverif[nb
10                -1]=1
11         return not(0 in
12        listeverif)
```

Listing 4:

```
1 import copy
2 def listecorrecte2(L):
3     L1=copy.deepcopy(L)
4     if len(L)!=9:
5         return len(L)==9
6     else:
7         L1.sort()
8         return L1
9     ==[1,2,3,4,5,6,7,8,9]
```

Listing 5:

```
1 def listecorrecte3(L):
2     if len(L)!=9:
3         return len(L)==9
4     else:
```

```

5     listeconstr=[]
6     for nb in L:
7         if nb>0 and nb<10 and not(nb in listeconstr):
8             listeconstr.append(nb)
9     return listeconstr==L

```

- ② On doit préalablement charger les modules `sympy` et `sympy.solvers`;

Il faut ensuite déclarer les noms de variables comme des symboles afin que python ne tente pas de renvoyer leur valeur. Enfin on lance la résolution pour obtenir le set d'équations définissant les expressions de A, B, C, D . Cela donne:

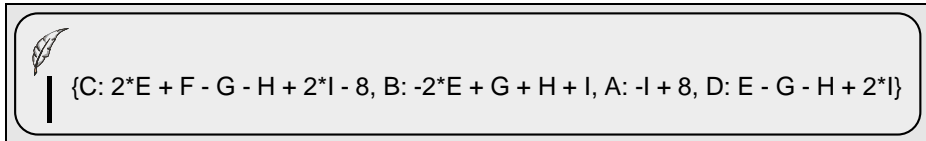
Listing 6:

```

1 from sympy import symbols
2 from sympy.solvers import solve
3 # création des noms des symboles
4 A,B,C,D,E,F,G,H,I=symbols("A:I")
5
6 # Résolution du système d'équations
7 print solve([A+B+C-2*I-F,D+E+F-A-C-I,G+H+I-2*E-B,A+I-8],A,B,C,D
8             )

```

On obtient le résultat suivant:



{C: $2 \cdot E + F - G - H + 2 \cdot I - 8$, B: $-2 \cdot E + G + H + I$, A: $-I + 8$, D: $E - G - H + 2 \cdot I$ }

- ③ On propose le code complété suivant:

Listing 7:

```

1 import time as t
2 debut=t.time()
3 ##### Préparation de l'affichage des solutions #####
4 s="A={ },B={ },C={ },D={ },E={ },F={ },G={ },H={ },I={ }"
5 nbessais = nbsol = 0
6 for E in range(1,10):
7     for F in range(1,10):
8         for G in range(1,10):
9             for H in range(1,10):
10                 for I in range(1,10):
11                     nbessais+=1

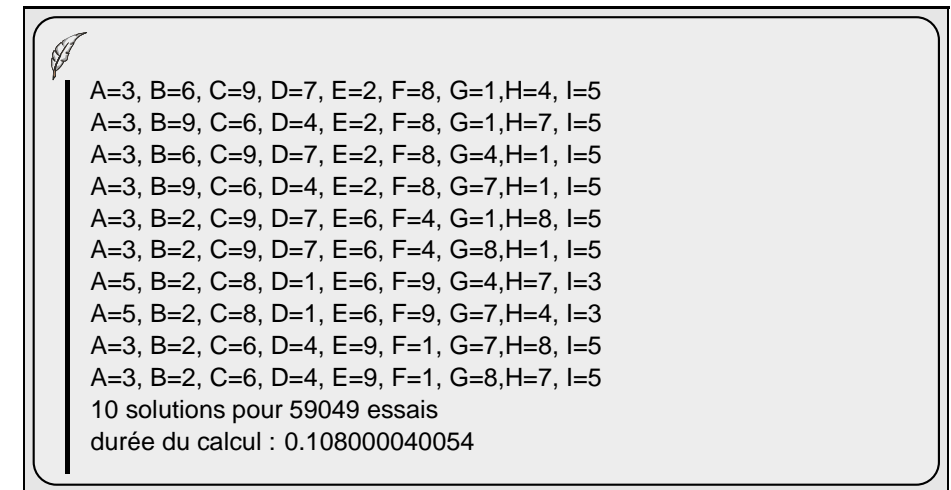
```

```

12 A=8-I
13 B=-2*E+G+H+I
14 C=2*E+F-G-H+2*I-8
15 D=E-G-H+2*I
16 if listecorrecte1([A,B,C,D,E,F,G,H,I]):
17     nbsol+=1
18     print(s.format(A,B,C,D,E,F,G,H,I))
19 print(' {} solutions pour {} essais'.format(nbsol,nbessais))
20 print(u"durée du calcul:", t.time()-debut)

```

La sortie donne:



A=3, B=6, C=9, D=7, E=2, F=8, G=1, H=4, I=5
A=3, B=9, C=6, D=4, E=2, F=8, G=1, H=7, I=5
A=3, B=6, C=9, D=7, E=2, F=8, G=4, H=1, I=5
A=3, B=9, C=6, D=4, E=2, F=8, G=7, H=1, I=5
A=3, B=2, C=9, D=7, E=6, F=4, G=1, H=8, I=5
A=3, B=2, C=9, D=7, E=6, F=4, G=8, H=1, I=5
A=5, B=2, C=8, D=1, E=6, F=9, G=4, H=7, I=3
A=5, B=2, C=8, D=1, E=6, F=9, G=7, H=4, I=3
A=3, B=2, C=6, D=4, E=9, F=1, G=7, H=8, I=5
A=3, B=2, C=6, D=4, E=9, F=1, G=8, H=7, I=5
10 solutions pour 59049 essais
durée du calcul : 0.108000040054

Manipulations de base sur les chaines

EXERCICE N°4:

Vérification d'un palindrome

On propose le code suivant:

Listing 8:

```

1 def palindrome(ch1):
2     ch2=""
3     for i in range(-1,-len(ch1)-1,-1):
4         print ch1[i]
5         ch2=ch2+ch1[i]
6     if ch2!=ch1:
7         return False
8     return True

```

On peut également faire appel à la méthode `.reverse()` qui permet d'inverser l'ordre des éléments d'une liste. Il faut donc préalablement transformer la chaîne en liste de caractères:

Listing 9:

```
1 def palindrome_reverse(ch1):
2     liste1=[]
3     for carac in ch1:
4         liste1.append(carac)
5     listecop=deepcopy(liste1)
6     liste1.reverse()
7     if listecop==liste1:
8         return True
9     return False
```

EXERCICE N°5: Recherche d'acides aminés dans une chaîne d'ADN

- 1 Fonction valide(seq):

Listing 10:

```
1 def valide(seq):
2     ret=len(seq)!=0 #initialise le renvoi à True s'il y a une
3     séquence non nulle
4     for c in seq:
5         if not((c == 'a') or (c == 't') or (c == 'g') or (c ==
6         'c')):
7             ret=False
8     return ret
```

- 2 Fonction de saisie valide:

Listing 11:

```
1 def saisie(chaine):
2     seq=""
3     while not valide(seq):
4         seq=input(u"Faire une saisie valide:")
5     return seq
```

- 3 Fonction proportion

Listing 12:

```
1 def proportion(chaine,sequence):
```

```
2     compteur=0
3     i=0
4     while i<len(chaine) and (i+len(sequence)<=len(chaine)): #
5         vérifie la longueur de chaîne restante
6         if chaine[i:i+len(sequence)]==sequence:
7             compteur=compteur+1
8             i=i+len(sequence)
9         else:
10            i=i+1
11    return 100*compteur/(len(chaine)-len(chaine)%len(sequence)),compteur
```

Enfin le programme principal

Listing 13:

```
1 chaine=saisie(u"introduire la chaîne d'ADN")
2 sequence=saisie(u"introduire la séquence")
3 print u"la proportion (en %) et le nombre d'occurrences de la
4     séquence dans la chaîne d'ADN sont:",proportion(chaine,
5     sequence)
```

EXERCICE N°6: Recherche dans un texte

- 1 On propose la fonction `caracmaj(c)` suivante, précédée d'une boucle `while` destinée à trouver l'indice de la première majuscule "A" dans le tableau ASCII (utile pour la suite):

Listing 14:

```
1 def caracmaj(c):
2     if ord(c) in range(ord("A"),ord("A")+26):
3         return c
4     else:
5         return chr(0)
```

La fonction `caracmaj(c)` vérifie simplement si le code ASCII du caractère `c` se trouve dans l'intervalle des codes correspondant aux majuscules de A à Z, et le cas échéant renvoie le caractère, ou bien ne renvoie rien du tout (ie le code ASCII 0 par la commande `chr(0)`) dans le cas contraire.

Autre proposition:

Listing 15:

```
1 def caracmaj(c):
2     inf, sup = ord("A"), ord("Z")
3     if (ord(c) >= inf) and (ord(c) <= sup):
4         return c
5     else:
6         return chr(0)
```

Listing 16:

```
1 def compte(s, c):
2     n = 0
3     for car in s:
4         if car == c:
5             n += 1
6     return n
```

- ③ On peut en effet exploiter la fonction `compte(s, c)` dans la fonction `nb_lettres(s)` qui recense la fréquence de toutes les lettres majuscules de l'alphabet:

Listing 17:

```
1 def nb_lettres(s):
2     res = []
3     for p in range(ord("A"), ord("A") + 26):
4         res = res + [compte(s, chr(p))]
5     return res
```

- ④ On constate que la chaîne `s` est parcourue à chaque appel de la fonction `compte(s, c)`, soit **26 fois au total**.
- ⑤ On propose la fonction optimisée `nb_lettres_opt(s)` suivante qui exploite avantageusement la fonction `caracmaj(c)` définie plus haut:

Listing 18:

```
1 def nb_lettres_opt(s):
2     res = [0 for p in range(26)]
3     for car in s:
4         if caracmaj(car) != chr(0):
5             res[ord(caracmaj(car)) - ord("A")] += 1
6     return res
```

EXERCICE N°7:

Découpage et recensement des mots dans un texte

- ① On propose la fonction suivante qui traite tous les cas possibles:

Listing 19: Sources_Python/mot_suivant.py

```
1 def mot_suivant(expression, i):
2     n = len(expression)
3     mot = ""
4     if i >= n:
5         return "indice_i_incorrect"
6     else:
7         ind = i #initialisation indice courant
8         while ind != n and expression[ind] != " ": #tant que pas
9             #bout de mot et pas en bout expression
10            mot = mot + expression[ind] # on ajoute le caractÃre
11            suivant au mot
12            ind += 1 #incrÃment de l'indice courant
13            if ind == n: #teste si seul le premier mot existe
14                return (mot, n) #alors on renvoie le mot et sa
15            longueur
16            else: #sinon
17                while ind < n and expression[ind] == " ": #tant qu'on n
18                'est pas en fin de liste et au prochain mot
19                    ind += 1 #on avance
20            return (mot, ind)
21 ch = "La nuit est la plus belle"
22 print(mot_suivant(ch, 9))
```

- ② mots exploite évidemment la fonction `mot_suivant`. On notera la structure conditionnelle en ligne 7 qui permet d'éviter l'inclusion d'un mot vide dans l'hypothèse d'un espace en tête de `expression`:

Listing 20: Sources_Python/mots.py

```
1 def mots(expression):
2     n = len(expression)
3     k = 0
4     listemots = []
5     while k < n:
6         (suivant, ind) = mot_suivant(expression, k)
7         if suivant != '':
8             listemots += [suivant]
9         k = ind
10    return listemots, "Nombre de mots:", len(listemots)
```

- ③ On inclut cette fois le caractère d'espace et tous les signes de ponctuation dans une liste, et l'on teste si le caractère analysé est présent dans cette liste, plutôt que de se limiter comme ci-dessus à tester s'il s'agit d'un simple espace:

Listing 21: Sources_Python/mots_punctuation.py

```
1 s=[" ", ",", ";", ".", "!", "?"]
2 def mot_suitant(expression, i, s):
3     n=len(expression)
4     mot=""
5     if i>=n:
6         return "indice_à_incorrect"
7     else:
8         ind=i #initialisation indice courant
9         while ind!=n and expression[ind] not in s: #tant que
10             pas bout de mot et pas en bout expression
11             mot=mot+expression[ind] # on ajoute le caractÃ`re
12             suivant au mot
13             ind+=1 #incrÃ©ment de l'indice courant
14             if ind==n: #teste si seul le premier mot existe
15                 return (mot,n) #alors on renvoie le mot et sa
16             longueur
17             else: #sinon
18                 while ind<n and expression[ind] in s: #tant qu'on n
19                     'est pas en fin de liste et au prochain mot
20                     ind+=1 #on avance
21                 return (mot, ind)
```

Manipulations de base sur les listes

EXERCICE N°8: Le crible d'Eratostène

- ① Script possible:

Listing 22:

```
1 import numpy as np
2 N=0.0
3 while (type(N)<>int) or (N<2):
4     N=input("Entrez un entier N: ")
5 liste=range(2,N+1)
6 listefinale=[]
7 while liste <>[]:
8     listefinale=listefinale+liste[0:1]
```

```
9 for p in liste[1::]:# va examiner du second au dernier
10     élément de liste son caractère multiple du premier element
11     if p%liste[0]==0:
12         liste.remove(p)
13 print("La liste des nombres premiers entre 2 et ",N, ("est: "),
14     listefinale
```

- ② On ajoute un compteur dans le script précédent:

Listing 23:

```
1 import numpy as np
2 N=0.0
3 while (type(N)<>int) or (N<2):
4     N=input("Entrez un entier N: ")
5 liste=range(2,N+1)
6 listefinale=[]
7 comp=0
8 while liste <>[]:
9     listefinale=listefinale+liste[0:1]
10    for p in liste[1::]:# va examiner du second au dernier
11        élément de liste son caractère multiple du premier element
12        if p%liste[0]==0:
13            liste.remove(p)
14            comp+=1
15        liste.remove(liste[0])
16 print("La liste des nombres premiers entre 2 et ",N, ("est: "),
17     listefinale
18 print("Compteur en fonction de N=",N, ":", comp
```

EXERCICE N°9: Recherche de répétitions dans une liste

- ① On propose l'implémentation suivante en Python:

Listing 24:

```
1 def nombreZeros(t, i):
2     if t[i]==1:
3         return 0
4     else:
5         nb0=0
6         k=i
```

```

7         while k<len(t) and t[k]==0:
8             nb0+=1
9             k+=1
10    return nb0

```

② Méthode exploitant la fonction nombreZeros(t,i):

- on initialise un compteur comp à 0
- on initialise un indice i à 0
- on lance une boucle conditionnelle while qui se poursuit tant que $i < \text{len}(t) - 1$, et qui vérifie si `nombreZeros(t,i)` renvoie une valeur supérieure à comp et remplace comp par cette valeur le cas échéant; enfin si il y a des 0 contigus, on incrémente i de la valeur retournée par `nombreZeros(t,i)` sinon on incrémente simplement de 1, puis la boucle se poursuit.

On propose l'implémentation suivante très naïve:

Listing 25:

```

1 def nombreZerosMax(t):
2     comp=0
3     i=0
4     while i<len(t)-1:
5         if nombreZeros(t,i)>comp:
6             comp=nombreZeros(t,i)
7         if nombreZeros(t,i)>1:
8             i=i+nombreZeros(t,i)-1
9         else:
10            i+=1
11    return comp

```

et une version de meilleure complexité qui limite le nombre d'appels à `nombreZeros(t,i)`:

Listing 26:

```

1 def nombreZerosMax(t):
2     comp=0
3     i=0
4     while i<len(t)-1:
5         nombreZeroscontigus=nombreZeros(t,i)
6         if nombreZeroscontigus>comp:
7             comp=nombreZeroscontigus
8         if nombreZeroscontigus>1:
9             i=i+nombreZeroscontigus-1
10    else:

```

```

11        i+=1
12    return comp

```

Procédés aléatoires

EXERCICE N°10:

Marche auto-évitante (d'après CCMP 2021)

- ① Cette question est simple: on implémente par exemple une liste contenant les positions des 4 points voisins de p sur la grille (attention: les déplacements en diagonale sont interdits puisque la distance entre deux points consécutifs est obligatoirement de 1), puis on vérifie pour chacun d'entre eux s'il n'est pas déjà dans la liste atteints avant de l'ajouter en queue de la liste possibles:

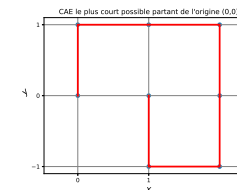
Listing 27: Sources_Python/Positions_possibles.py

```

1 def positions_possibles(p, atteints):
2     possibles=[]
3     P=[[p[0]+1,p[1]], [p[0]-1,p[1]], [p[0],p[1]+1], [p[0],p[1]-1]]
4     for pos in P:
5         if pos not in atteints:
6             possibles.append(pos)
7     return possibles

```

- ② Propositions de CAE le plus court possible:



Les autres CAE sont obtenus par rotation de $\pi/2$ de ce chemin, puis symétrie par rapport à un plan (fixe) pour chacun d'entre-eux, soit 8 chemins au total.

- ③ On propose enfin le code suivant pour `genere_chemin_naif(n)`:

Listing 28: Sources_Python/genere_chemin_naif.py

```

1 import random as rd
2 def genere_chemin_naif(n):
3     chemin=[[0,0]]
4     k=0
5     while k<n+1:
6         liste_points=positions_possibles(chemin[-1],chemin)
7         if liste_points==[]:

```

```

8         return None
9     else :
10         k+=1
11         chemin.append(rd.choice(liste_points))
12     return chemin

```

EXERCICE N°11: Le paradoxe des anniversaires

- ❶ On recherche la probabilité qu'au moins deux personnes parmi les N convives soient nées le même jour, soit le complément à 1 que toutes les personnes soient nées un jour différent dont la probabilité se calcule facilement:
- nombre total de possibilités de jour de naissance pour N convives: pour le premier: 365, pour le second: 365 etc... soit: $\Omega = 365^N$
 - nombre de possibilités que les N convives soient tous nés un jour différent: pour le premier: 365, pour le second 364, pour le $N^{ième}$ et dernier $365 - N + 1$ soit

$$\Omega_{\neq} = 365 \times 364 \times \dots \times (365 - N + 1) = \frac{365!}{(365 - N)!}$$

La probabilité recherchée est donc:

$$P_{\text{même}} = 1 - P_{\neq} = 1 - \frac{\Omega_{\neq}}{\Omega} = 1 - \frac{365!}{(365 - N)! \times 365^N}$$

- ❷ Script Python du calcul de probabilité:
Si l'on tente le calcul à partir de la relation obtenue ci-dessus, c'est à dire en écrivant:

Listing 29:

```

1 def fact(n):
2     res=1
3     while n>0:
4         res=res*n
5         n-=1
6     return res
7
8 def para_anniversaires(N):
9     return 100*(1 - float(fact(365)) / float(fact(365-N)*365**N))

```

Lorsque l'on exécute ce script par exemple pour $N = 100$, Python renvoie alors un message d'erreur signalant qu'il est incapable de convertir des entiers si longs en flottants:



Traceback (most recent call last):
 return 100*(1-float(fact(365))/float(fact(365-N)*365**N))
 OverflowError: long int too large to convert to float

On peut alors procéder en formulant le calcul demandé par un procédé itératif évitant notamment de passer par le calcul d'une fraction à grands nombres:

$$P_{\neq} = \frac{(365 - N + 1)}{365^N} = \prod_{i=1}^{N-1} \frac{365 - i}{365}$$

Listing 30:

```

1 def para_anniversaires_bis(N):
2     p=float(1) #on peut aussi écrire p=1. pour la conversion en
3     flottant
4     for i in range(1,N):
5         p=p*(365-i)/365
6     return 1-p

```

Pour $N = 100$ soit la centaine d'invités, ce script renvoie 0,999999692751, résultat plutôt différent de ce que dicte l'intuition, d'où l'appellation de *paradoxe des anniversaires*.

EXERCICE N°12: Méthodes de Monte-Carlo

- ❶
- On entre une valeur de précision de calcul
 - On initialise les compteurs N_{int} et N
 - Tant que la précision n'est pas atteinte sur l'évaluation numérique de π , on itère la suite:
 - tirage des coordonnées x et y d'un point M comprise entre 0 et 1.
 - si le point M tombe dans un cercle de centre $C(0.5, 0.5)$ et de rayon $R = 0.5$ on incrémente le compteur intérieur N_{int} , ainsi que le compteur total N .
 - sinon (il tombe forcément dans le carré de côté 1 et de centre C), on incrémente seulement le compteur total N

- A chaque itération, on évalue une valeur expérimentale de π stockée dans `resexp` en posant que le rapport N/N_{int} tend vers le rapport des surfaces S_{cercle}/S_{carre} ; la relation donnée correspond bien à l'évaluation de π (à faire à la main!)

② Version 3D de cet algorithme:

Listing 31:

```
1 from random import *
2 import numpy as np
3 pi=np.pi
4 erreur=0
5 while not( erreur >= 1e-9):
6     erreur=input(u" Entrer la précision désirée pour ce calcul (e>1E-9): ")
7 Nint=0
8 N=0
9 resexp=0
10 while abs(pi-resexp)>erreur:
11     x=random()
12     y=random()
13     z=random()
14     if np.sqrt((x-0.5)**2+(y-0.5)**2+(z-0.5)**2) <= 0.5:
15         Nint=Nint+1
16         N=N+1
17         resexp=3*Nint/(4*0.5**3*N)
18     else:
19         N=N+1
20         resexp=3*Nint/(4*0.5**3*N)
21     print(resexp)
22
23 print(u"La valeur approchée recherchée est: "), resexp
```

Arithmétique

EXERCICE N°13:

Structure du code INSEE

On propose le code suivant qui manipule de bout en bout le code INSEE comme un nombre entier long:

Listing 32:

```
1 import math as m
2 INSEE=1.0
```

```
3 while not( type(INSEE)==long and int(1+m.floor(m.log10(INSEE)))==13
4         and (int(m.floor(INSEE/1E12))==1 or int(m.floor(INSEE/1E12))
5         ==2)):
6     INSEE=input(u" Entrer un numéro INSEE à 13 chiffres: ")
7     INSEE=long(INSEE)
8     print(u"La clé du numéro INSEE est: "), int(97-INSEE%97)
```

On propose également la variante suivante, qui cette fois convertit le code INSEE en chaîne de caractère pour en analyser la structure:

Listing 33:

```
1 INSEE="1.0"
2 while not( type(INSEE)==str and len(INSEE)==13 and (INSEE[0]=="1" or
3         INSEE[0]=="2")):
4     INSEE=str(input(u" Entrer un numéro INSEE à 13 chiffres: "))
5     INSEE=long(INSEE)
6     print(u"La clé du numéro INSEE est: "), int(97-INSEE%97)
```

EXERCICE N°14:

Vérification des codes barres

- D'après la définition donnée, la clé qui est le chiffre a_{13} (le dernier!) du code barre correspond simplement au complément à 10 du nombre défini par:

$$3 \sum_{k=1}^6 a_{2k} + \sum_{k=0}^5 a_{2k+1}$$

ainsi:

$$cle = \left(10 - \left(3 \sum_{k=1}^6 a_{2k} + \sum_{k=0}^5 a_{2k+1} \right) \% 10 \right) \% 10$$

On propose un premier script manipulant le code barre en tant que nombre:

Listing 34:

```
1 import math as m
2 CODE=1.0
3 while not( type(CODE)==long and int(1+m.floor(m.log10(CODE)))
4         ==12):
5     CODE=long(input(u" Entrer les douze chiffres 'produit' du
6         code barre: "))
7     CODEp=CODE
```



```

6 i=1
7 cp=0
8 ci=0
9 while CODEp>0:
10     r=CODEp%10
11     CODEp=CODEp//10
12     if i%2==0:
13         cp =cp+r
14         print("somme_paire_au_rang_",i,":",cp)
15     else:
16         ci=ci+r
17         print("somme_impaire_au_rang_",i,":",ci)
18     i+=1
19 cle=(10-(3*cp+ci)%10)%10
20 print(u"La clé du code barre est:",cle)
21 print(u"Le code barre est:",long(cle*1E12+CODE))

```

et un second manipulant les digits du code barre sous forme de chaîne de caractères:

Listing 35:

```

1 while not(type(CODE)==long and int(1+m.floor(m.log10(CODE)))
  ==12):
2     CODE=long(input(u"Entrer les douze chiffres 'produit' du
  code barre:"))
3 CODEp=str(CODE)
4 cp=0
5 ci=0
6 for i in range(11,-1,-1):
7     if i%2==0:
8         cp=cp+int(CODEp[i])
9     else:
10        ci=ci+int(CODEp[i])
11 cle=(10-(3*cp+ci)%10)%10
12 print(u"La clé du code barre est:",cle)
13 print(u"Le code barre est:",long(cle*1E12+CODE))

```

② Script de vérification d'un code barre:

Listing 36:

```

1 import math as m
2 CODE=1.0
3 while not(type(CODE)==long and int(1+m.floor(m.log10(CODE)))
  ==13):
4     CODE=long(input(u"Entrer les treize chiffres du code barre:
  "))

```

```

5 CODEp=CODE
6 i=1
7 cp=0
8 ci=0
9 while CODEp>0:
10     r=CODEp%10
11     CODEp=CODEp//10
12     if i%2==0:
13         cp =cp+r
14         print("somme_paire_au_rang_",i,":",cp)
15     else:
16         ci=ci+r
17         print("somme_impaire_au_rang_",i,":",ci)
18     i+=1
19 if (3*cp+ci)%10==0:
20     print(u"Le code barre est valide")
21 else:
22     print(u"Le code barre est invalide")

```

EXERCICE N°15:

Technique du hachage des chaînes

① Fonction naïve:

Listing 37:

```

1 import time as t
2 def chaine_entier(ch):
3     res=0
4     i=1
5     for c in ch:
6         res=res+ord(c)*256**((len(ch)-i))
7         i=i+1
8     return res
9 a=t.time()
10 print(chaine_entier("abcdefghijklmnopqrstuvwxyz"*3))
11 b=t.time()
12 print(u'Le temps d'execution est: ',b-a,' secondes')

```

② Exploitation du schéma de Horner; on remarque que l'entier recherché est égal à la valeur d'un polynôme $P(x)$ dont les coefficients correspondent aux codes ASCII des caractères de la clé (du premier au dernier caractère) en $x = 256$; on calcule donc $P(256)$.

Listing 38:

```
1 def chaine_entier_Horner(ch):
2     res=ord(ch[0])*256+ord(ch[1])
3     for i in range(2,len(ch)):
4         res=res*256+ord(ch[i])
5     return res
6
7
8 a=float(t.time())
9 print(chaine_entier_Horner("abcdefghijklmnopqrstuvwxy" * 3))
10 b=float(t.time())
11 print(u"Le temps d'execution est :", b-a, "secondes")
```

- ③ On peut par exemple proposer le script suivant:

Listing 39:

```
1 def entier_chaine(int):
2     res=""
3     while int%256!=0:
4         res=chr(int%256)+res
5         int=int//256
6     return res
```

- ④ a. Fonction de hachage élémentaire $h(ch)$ et exploitation sur les chaînes pouet, chariot, haricot:

Listing 40:

```
1 def h(ch):
2     return chaine_entier_Horner(ch)%255
3 print(h("pouet"))
4 print(h("chariot"))
5 print(h("haricot"))
```

- b. Le code script précédent appliqué aux chaînes "pouet", "chariot", "haricot" renvoie dans cet ordre:

47
236
236

On constate donc que la fonction de hachage choisie (qui est un simple "modulo 255") est trop simple et conduit au même entier pour toutes les clés *anagrammes* du dictionnaire. Elle est donc inutilisable.

- ⑤ L'indice obtenu après hachage de l'entier "long" peut s'écrire:

$$(a_n \cdot 256^n + a_{n-1} \cdot 256^{n-1} + \dots + a_1 \cdot 256 + a_0 \cdot 256^0)[255]$$

$$= (a_n + a_{n-1} + \dots + a_1 + a_0)[255]$$

Ainsi, l'indice renvoyé par la fonction de hachage est indépendant de l'ordre du "set" des coefficients $(a_n, a_{n-1}, a_{n-2}, \dots, a_0)$, et sera donc le même pour des mots anagrammes.

- ⑥ Script de vérification de validité du dictionnaire pour $h(ch)$:

NB: cette procédure ne fonctionne que dans le cas d'un dictionnaire dont les clés sont des chaînes.

Listing 41:

```
1 def dico_valide(dict):
2     res=True
3     listecles=dict.keys()
4     for i in range(len(listecles)):
5         for j in range(i+1,len(listecles)):
6             if h(listecles[i])==h(listecles[j]):
7                 res=False
8     return res
9
```

Enfin, on propose cette variante qui évite un balayage total du dictionnaire par l'emploi de boucles conditionnelles:

Listing 42:

```
1 def dico_valide_bis(dict):
2     res=True
3     listecles=dict.keys()
4     i=0
5     while res and i<len(listecles)-1:
6         j=i+1
7         while res and j<len(listecles)-1:
8             res=h(listecles[i])!=h(listecles[j])
9             j=j+1
10        i+=1
11    return res
```

Enfin, on propose dernière variante qui exploite une liste de 256 entiers (d'indice $\in [0, 255]$), tous initialisés à 0. Cette liste va recueillir, à chaque valeur d'indice, le nombre d'occurrences du hachage de clé qui a justement donné la valeur de cet indice (liste de fréquences); pour assurer l'absence d'anagrammes, aucun total d'occurrences ne doit dépasser 1: on teste donc cela directement dans la boucle conditionnelle de balayage de la liste des clés; la complexité est ainsi nettement réduite par rapport aux scripts proposés plus haut:

Listing 43:

```

1 def dico_valide_ter(dict):
2     res=True
3     listecles=dict.keys()
4     tab=[0]*256
5     i=0
6     while res and i<len(listecles):
7         if tab[h(listecles[i])]==0:
8             tab[h(listecles[i])]+=1
9             i+=1
10        else:
11            res=False
12    return res

```