I Rappels de cours

I.1 Fonctions et procédures

```
def nom_de_la_fonction(parametre1,parametre2):
    # code qui calcule un/des résultat(s)
    return resultat #return resultat1,resultat2 pour un couple de résultats;
    return [resultat1,resultat2] pour un tableau de résultats
```

```
def nom_de_la_fonction(): #fonction sans paramètre
    # calcul du résultat
    # fonction sans résultat renvoyé (procédure)
```

☞ Ne pas confondre `print` (on affiche et on oublie) et `return` (on peut utiliser le résultat par la suite).

I.2 Données

- Types simples : `integer` (entier), `boolean` (booléens), `float` (réels approchés en virgule flottante), `complex` (complexes), `char` (caractère).
- Types composés :
  - `list` : liste, modifiable, délimitée par des crochets, séparation par des virgules, *par ex. :* `[1,2,3.0,"a"]` ,
  - `string` : chaîne de caractères, non mutable, délimitée par ' ou ", pas de séparateur entre les caractères, *par ex. :* `"123a"` ,
  - `tuple` : liste, non mutable, délimitée par des parenthèses, séparation par des virgules, *par ex. :* `(1,2,3)`,
  - `array numpy` : cf. TD ultérieur

  ☞ **Les n éléments des types composés précédents sont numérotés de 0 à $n-1$.**
  - `set` : ensemble d'éléments distincts, non accessibles par indice, mais par nom ; non mutable, délimité par des accolades, *par ex. :* `{'1','2','3','4'}` ;
  - `dict` : ensemble d'éléments, non accessibles par indice, mais par le nom de leur clé (noms distincts) ; non mutable, délimitée par des accolades, séparation par des virgules *par ex. :* `{'1':'Roger','2':'Jeanne','3':'Robert','4':'Jeanine'}` ;
- Opérations usuelles :
  - `x+y` : somme pour les nombres entiers ou flottants, *par ex. :* `1+2` → `3`
    mais concaténation pour les chaînes de caractères ou les tableaux,
    par ex. :`'1'+'2'` → `'12'`
  - `x*y` : produit pour les nombres entiers ou flottants, mais aussi un signe de concaténations multiples : *par ex. :* `'ar'*2` → `'arar'`
  - division : `x/y` division (résultat réel) ; `x//y` division euclidienne (résultat entier) ; `x%y` reste de la division euclidienne,
    par ex. : `6/2` → `3.0` ; `7//2` → `3` ; `12%5` → `2`
  - puissance `x**y`, *par ex. :* `2**5` → `32`
- Conversions (quand c'est possible), *par ex. :*
  `float(3)` → `3.0` ; `int(3.5)` → `3` ; `list("abc")` → `["a","b","c"]` ; `boolean(0)` → `False`
- Manipulation des types composés :
  - nombre d'éléments : `len(x)` (en anglais *length* signifie "longueur") ;

- appel de l'élément d'indice i (qui est le $i+1$-ème élément) : `x[i]` ;
- appel du dernier élément : `x[-1]` ou `x[len(x)-1]` ;
  ☞ « `x[len(x)]` » provoque toujours l'erreur « `out of range` ».
- (*slicing*) extraction d'une tranche de i (inclus) à j (exclu) : `x[i:j]` ;
- (*slicing*) extraction à partir du début (ou jusqu'à la fin) : `x[:j]` (ou `x[i:]`) ;

- **L'affectation `x=`**$[\cdots]$: en général, création d'une variable x qui contient la valeur donnée par $[\cdots]$, **sauf deux cas particuliers importants :**
  - une variable `x` existait déjà : elle est "remplacée" par la nouvelle valeur.
    Exemple : `x=x+k` (incrémentation, ie. on ajoute `k` à `x`) se code aussi par `x+=k`.
    ☞ Faire attention à tenir compte de la portée de la variable, *par exemple :*

```
def f(x):
    y = x
    return y
y = 2
z = f(12)
print(y,z)
```

    → affiche `2, 12` (et non `12, 12` ; ce n'est pas le même `y` dans et hors de `f`).
  - `x=y`, où `y` est une variable d'un type composé : il n'est pas créée de variable `x` mais le contenu de la variable `y` a indifféremment les deux noms `x` et `y` ;
    Analogue avec `x=[y,y]` ou `x=[y]*2`, avec `y` une variable d'un type composé (toute modification d'un élément d'une liste parmi `x[0], x[1]` ou `y` modifie les autres).
  - Affectations simultanées (pour alléger le code) :

```
x, y = a, b       #plusieurs affectations sur une seule ligne
[x, y] = point    #point est une liste de 2 éléments
x, y = y, x       #échange les valeurs de x et y
```

I.3 Structures conditionnelles

```
if condition:
    #code à exécuter si la condition est réalisée

if condition1:
    #code1, à exécuter si condition1 est réalisée
elif condition2:
    #code2, à exécuter si condition1 n'est pas réalisée et si la condition2
    est réalisée
else:
    #code3, à exécuter si aucune des conditions précédentes n'est réalisée
```

- `condition1` et `condition2` ne sont pas forcément incompatibles (condition 2 peut être une sous-condition de condition1) ; mais une seule portion de code est exécutée.
- `condition` est un booléen (à valeurs dans `True`, `False`) obtenu (ou non) par un calcul ou une fonction.
  — Opérations sur tous les types, qui donnent un résultat booléen : `==` ; `!=` ; `>` ; `<` ; `<=` ; `>=`.
  — Opérations entre booléens avec un résultat booléen : `and`/`&` ; `or` ; `not`.
  Exemple classique : `x%2 == 0` → `True` si `x` est pair, `False` sinon
  ⇝ l'évaluation paresseuse permet d'éviter des erreurs d'exécution
  par ex. : `i<len(L) and L[i]==12` → `False` (si `i` vaut `len(L)` ; pas d'erreur).

I.4 Boucles

```
while condition:
    #code à effectuer tant que la condition est réalisée

for i in range(n):
    #code à effectuer n fois
    #si nécessaire on peut utiliser la valeur i qui va de 0 à n-1
```

La boucle `for` est une boucle `while` particulière avec une écriture simplifiée :

```
#équivalent avec while de la boucle for ci-dessus (très maladroit; à éviter)
i=0
while i<n:
    #code à effectuer n fois
    i=i+1
```

Cas particulier d'une boucle `for` interrompue (dans une fonction) :

```
def mafonction(...):
    for i in range(n):
        if condition:
            ...
            return resultat1 #return termine la fonction donc interrompt la
    boucle
    return resultat2 #si condition jamais réalisée, toute la boucle est
    effectuée

#peut remplacer :
def mafonction(...):
        ...
    while i<n and not(condition):
        ...
if condition:
    return resultat1
return resultat2
```

☞ Dans une boucle `while`, il faut que les paramètres de la condition soient modifiés par le code de la boucle ; sinon la boucle ne se termine jamais.

☞ **Il ne faut jamais** modifier la variable `i` de la boucle `for` à l'intérieur de la boucle `for` :

- Extension des itérateurs pour les boucles
  - `i in range(n)` : `i` parcourt les entiers de 0 à $n-1$;
  - `i in range(n,p)` : `i` parcourt les entiers de n à $p-1$;
  - `i in range(n,p,q)` : valeurs de n (compris) à p (non compris) avec un pas de q ie. `i` parcourt les entiers n, $n+q$, $n+2q$ … $n+kq$ avec $n+kq < p \leq n+(k+1)q$.
  - `e in t` : `e` parcourt tous les éléments du tableau `t` (qui ne sont pas forcément des entiers).

```
for i in range(len(L)):
    #calcul sur L[i] (mais pas sur i)
#revient au même que :
for e in L:
    #calcul sur e
```

I.5 Construction d'une liste (d'un tableau)

- À l'aide d'une boucle avec la méthode `append` :

```
L = [] # initialisation
for i in range(n) :
    L.append(fonction(i)) # ajoute le résultat de fonction(i) à la fin de L
```

- Avec l'écriture en compréhension :

```
L = [fonction(i) for i in range(n)]
```

- À l'aide d'une boucle par concaténation (**à éviter** pour des questions de performance) :

```
L = [] # initialisation
for i in range(n) :
    L = L + [fonction(i)]  #cette instruction permet de créer L à partir de la fin
```

- Cas particulier d'une copie de tableau :

```
copie = []
for i in range(len(original)):
     copie.append(original[i])

#ou en compréhension
copie = [l for l in original]

#ou par slicing
copie = original[:]

#ou avec une fonction dédiée (seule copie "en profondeur" de tableau de tableaux)
import copy
copie = copy.deepcopy(original)
```

I.6 Algorithmes classiques de recherche dans un tableau `t`

- **Recherche si l'un des éléments du tableau vérifie une condition donnée**

```
def recherche1(t):
    for e in t:
        if condition(e): # implicitement == True
            return True # boucle interrompue (while possible aussi)
    return False
```

Exemples : tester la présence d'un élément donné dans un tableau, comparer deux tableaux revient à rechercher s'il existe deux éléments homologues différents.

- **Recherche d'un élément optimal (d'un certain point de vue) dans un tableau :**

```
resultat = t[0] # ou initialiser avec le pire résultat possible
for e in t:
    if e meilleurque resultat:
        resultat = e
```

Exemples : maximum, minimum.

- **Résultat obtenu par agrégation de tous, ou de certains éléments d'un tableau :**

```
resultat = initialisation #resultat si la liste était vide
for condition_sur_j :
    resultat = fonction_de_resultat_et_de_j
```

Exemples : somme, produit, moyenne, nombre des éléments vérifiant une condition donnée.

II Exercices

Exercice 1.

Q1 ***(Création de tableau).*** *Écrire deux façons différentes (avec* `append` *et en compréhension) une fonction* `zero_un(n)` *qui renvoie la liste formée de* `n` *nombres qui valent alternativement* `0` *puis* `1`*. Par exemple :* `zero_un(7)`$\rightarrow$`[0,1,0,1,0,1,0]`

Q2 ***(Test d'appartenance).*** *Écrire une fonction* `est_element(t,a)` *qui renvoie* `True` *si* `a` *est élément du tableau* `t`*, et* `False` *sinon.*

Q3 ***(Compteur).***
Écrire une fonction `NB(L,M)` *qui renvoie le nombre d'éléments de* `L` *qui appartiennent à* `M`*.*

Exercice 2. (construction de listes de listes)

Q4 *Écrire une fonction* `listenulle(n,p)` *qui renvoie la liste* `[l1,l2,...,ln]` *où chaque* `li` *est une liste de* p *zéros.*
Par exemple : `listenulle(3,2)` $\rightarrow$ `[[0,0],[0,0],[0,0]]`
Indiquer comment utiliser cette fonction afin de créer un tableau de taille 11×11 *dont toutes les cases sont remplies de zéros sauf celle du centre qui vaut* 1.

Q5 *Écrire une fonction* `L2(l)`*, d'argument une liste* `l` *d'entiers de longueur* `n`*, qui renvoie la liste* `[l0,l1,...,ln-1]` *où chaque* `li` *est une liste de* `l[i]` *zéros.*
Par exemple : `L2([3,2,4])` $\rightarrow$ `[[0,0,0],[0,0],[0,0,0,0]]`.

Q6 *Écrire une fonction* `A(n,a,b)`*, où* `n` *est un entier positif, qui renvoie le tableau* $n \times n$ *dont toutes les cases contiennent* `0` *sauf celles de la diagonale qui valent* `1` *et celles de la première ligne et de la première colonne qui valent alternativement* `b` *puis* `a`*.*
Exemple : `A(5,8,3)` $\rightarrow$ `[[1,3,8,3,8],[3,1,0,0,0],[8,0,1,0,0],[3,0,0,1,0],[8,0,0,0,1]]`

1	3	8	3	8
3	1	0	0	0
8	0	1	0	0
3	0	0	1	0
8	0	0	0	1

Q7 *Écrire une fonction* `T(L,n)`*, où* `L=[a,b,c]` *est une liste de trois nombres et* `n` *est un entier supérieur à 2, qui renvoie une matrice carrée d'ordre* `n` *avec* `b` *sur la diagonale, des* `a` *au dessus de la diagonale et des* `c` *en dessous de la diagonale.*

Exercice 3. ***(Suite récurrentes).***

Soit la suite de Syracuse définie par $u_0 \in \mathbb{N}^*$ *et :* $\forall n \geq 0,\ u_{n+1} = \begin{cases} 3u_n + 1 & \text{si } u_n \text{ impair} \\ \frac{u_n}{2} & \text{si } u_n \text{ pair.} \end{cases}$

Q8 *Écrire une fonction* `suite(u0,n)` *qui calcule* u_n*.*

Q9 *Écrire une fonction* `somme(u0,n)` *qui calcule* $u_0 + \cdots + u_n$*.*

Q10 *Écrire une fonction* `prod(u0,n)` *qui calcule le produit des termes* u_i *de la suite dont la valeur est paire et tels que* $0 \leq i \leq n$*.*

Q11 *On définit la suite* (u_n) *par* $u_0 = u_1 = 1$ *et* $u_{n+2} = u_{n+1} + u_n$*.*
Écrire une fonction `fibo(n)` *qui calcule* u_n*.*

Exercice 4.

Q12 ***(maximum)*** *Les sommes partielles d'une liste* `L` *de flottants sont les* $S_k = \sum_{i=0}^{n} L[i]$ *(*$0 \leq k <$`len(L)`*). Écrire une fonction* `max_SP(L)` *qui renvoie la plus grande somme partielle de la liste* `L`*.*

Q13 *Écrire une fonction* `max_somme_ligne(L)`*, où* `L` *est une liste de listes d'entiers, qui renvoie la plus grande somme des éléments d'une liste* `L[i]`*.*
Par exemple : `max_somme_ligne([[0,0,0,9,0,0],[3,7,0],[2,1,1,1]]` $\to$ `10`*.*

III Exercices plus difficiles

Exercice 5.

Q14 *Écrire une fonction* `bissextile(n)` *qui renvoie* `True` *si l'année* `n` *est bissextile (i.e. divisible par 4 mais pas par 100, ou divisible par 400) ou* `False` *sinon.*

Q15 *Écrire une fonction* `jourannee(jour, mois, annee)` *qui renvoie le nombre de jours écoulés depuis le début de l'année* `annee` *jusqu'à la date (incluse) donnée par* `[jour, mois, annee]`*.*
Par exemple : `jourannee([2, 3, 2004])`$\to$ `62` *et* `jourannee([2, 3, 2003])`$\to$ `61`

Q16 *On a compté 1446 pleines lunes entre le 15 janvier 1900 et le 14 novembre 2016. Programmer un script qui calcule la période de révolution de la Lune.*

Exercice 6. *On appelle « renversé » d'un entier le nombre obtenu en écrivant les chiffres de l'entier de départ à l'envers. Par exemple le renversé de 563 est 365. Un entier palindrome est un entier qui est égal à son renversé, par exemple 121 ou 1331.*

Q17 *Écrire une fonction* `estPalindrome(n)` *qui renvoie* `True` *si* `n` *est un palindrome ou* `False` *sinon. On pourra remarquer que :* `str(123)`$\to$ `'123'`

Q18 *Écrire une fonction* `verlan(n)` *qui renvoie l'entier obtenu en écrivant les chiffres de l'entier* `n` *à l'envers.* Par exemple : `verlan(563)`$\to$ `365`

Q19 *On note $r(n)$ la somme de n et de son renversé. Écrire une fonction* `pal(n,N)` *qui renvoie la liste* `[n,r(n),r(r(n)),...,`r^k`(n)]` *où k est le plus petit entier inférieur ou égal à* `N` *tel que* r^k`(n)` *soit un palindrome, s'il en existe un, et qui renvoie* `False` *s'il n'en existe pas.*
Par exemple : `pal(64,50)`$\to$`[64,110,121]` *et* `pal(98,20)`$\to$ `False`

IV Pour ceux qui ont déjà fini

Exercice 7. (problème de Flavius Joseph)

Q20 *On veut simuler le jeu suivant :*

- *on dispose en cercle n jetons numérotés de 0 à $n-1$;*
- *on commence par retirer le jeton 1, puis, un jeton sur deux en parcourant le cercle jusqu'à ce qu'il ne reste qu'un jeton.*

On représente la situation du jeu à un moment donné par une liste `L` *de* `n` *booléens (*`False` *si le jeton est présent et* `True` *sinon).*

(a) *Écrire une fonction* `suivant(L,p)`*, où* `L` *est la position du jeu à un moment donné, qui renvoie le rang du premier jeton présent dans la liste* `L` *en partant du rang* `p+1`*.*

(b) *En déduire une fonction* `reste(n)` *qui renvoie le rang du jeton restant à la fin du jeu.*

Q21 *Émettre une conjecture sur le numéro restant en fonction de n et la vérifier pour $n = 256, 512, 1024\ldots$*

Exercice 8.

Q22 *Écrire une fonction* `escargot(L)`*, d'argument un tableau* `L` *carré d'ordre* `n` *impair, à coefficients numériques, et qui renvoie la liste des coefficients de* `L` *dans l'ordre de parcours de l'escargot.* Par exemple :
`[[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15],[16,17,18,19,20],[21,22,23,24,25]]`
$\to$ `[1,2,3,4,5,10,15,20,25,24,23,22,21,16,11,6,7,8,9,14,19,18,17,12,13]`

Exercice 9.

Q23 *Écrire une fonction* `melange(L)`*, où* `L` *est une liste de longueur paire, qui renvoie la liste obtenue en mélangeant* `L` *à la manière d'un jeu de cartes : on coupe le jeu en deux tas égaux puis on mélange les tas en intercalant une carte du second tas après chaque carte du premier tas.* Par exemple : `melange([1,2,3,4,5,6])` → `[1,4,2,5,3,6]`

Q24 Pour les 5/2 : *Écrire une deuxième version de la fonction* `melange` *en utilisant uniquement les manipulations de piles (méthodes* `append` *et* `pop`*) — i.e. sans utiliser d'extraction de terme d'une liste (du type* `L[i]` *ou* `L[i:j]`*).*

Q25 *Écrire une fonction* `est_liste(L,l)`*, où* `L` *et* `l` *sont des listes, qui renvoie* `True` *si les éléments de* `l` *sont des éléments consécutifs de* `L` *et qui renvoie* `False` *sinon.*
Par exemple : `est_liste([1,3,5,7,9,11,13],[5,7,9])` → `True`
`est_liste([1,3,5,7,9,11,13],[5,7,11])` → `False`

Q26 *Écrire une fonction* `est_liste2(L,l)`*, où* `L` *et* `l` *sont des listes, qui renvoie* `True` *si les éléments de* `l` *sont des éléments de* `L`*, non nécessairement consécutifs, mais rangés dans le même ordre que celui de* `L`*, et qui renvoie* `False` *sinon.*
Par exemple : `est_liste2([1,3,5,7,9,11,13],[5,7,11])` → `True`
`est_liste2([1,3,5,7,9,11,13],[5,1,7])` → `False`

Exercice 10.

Q27 **(Codage)**. *Écrire une fonction* `code(L,n)`*, où* `L` *est une liste formée d'éléments distincts de* $[\![0, n-1]\!]$*, qui renvoie la liste* `LC` *de longueur* `n` *définie par* `LC[i]=1` *si* `i` *est élément de* `L` *et sinon* `LC[i]=0`.Par exemple : `code([1,3,4],5)` → `[0,1,0,1,1]`
Peut-on obtenir une fonction de complexité $O(n)$ *?*

Q28 *Écrire une fonction* `compare(L1,L2,n)` *qui renvoie* `True` *si les deux listes* `L1` *et* `L2`*, chacune formée d'éléments distincts de* $[\![0, n-1]\!]$*, ont exactement les mêmes éléments.*
Attention : la comparaison `L1==L2` ne convient pas car les éléments ne sont pas forcément dans le même ordre.

Q29 **(Décodage)** *Écrire une fonction* `decode_croissant(LC)` *qui renvoie la liste* `L` *des entiers distincts compris entre* 0 *et* $n-1$ *dans l'ordre croissant dont le code est* `LC`*.*

Q30 *En déduire une fonction* `intersection(L1,L2,n)` *qui renvoie une liste contenant exactement les éléments communs à* `L1` *et* `L2`*, triés dans l'ordre croissant.*

Exercice 11.

Q31 *Écrire une fonction* `F` *donnant le nombre de chiffres en base* 10 *d'un entier :* `F(9375)` → `4`*.*

Q32 *Écrire une fonction* `QueDesUns(n)` *qui renvoie le plus petit* N *multiple de* n *qui s'écrit en base* 10 *avec uniquement des 1.*
On admettra que, lorsque n impair et non multiple de 5, il existe toujours un tel N.

Exercice 12. *On appelle* diviseurs propres *d'un entier* $n \in \mathbb{N}^*$ *les entiers naturels différents de* n *qui divisent* n *(y compris* 1*).*

Q33 *Écrire une fonction* `SDP(n)` *qui renvoie la somme des diviseurs propres de l'entier* `n`*.*

Q34 *Un nombre est dit* parfait *si et seulement s'il est égal à la somme de ses diviseurs propres. Écrire une fonction* `parfaits(n)` *qui renvoie la liste des entiers parfaits inférieurs ou égaux à* `n`*.*

Q35 *Deux entiers* `p,q` *sont dits amicaux si et seulement si l'un est égal à la somme des diviseurs propres de l'autre et inversement. Écrire une fonction* `amicaux(k)` *qui renvoie la liste des listes* `[p,q]` *d'entiers amicaux vérifiant* $1 \leq$ `p`$<$`q`$\leq k$*.*

Corrigé

```python
###Q1###
def zero_un(n):
    t = []
    for i in range(n):
        t.append(i%2)
    return t

#ou
def zero_un2(n):
    t = [i%2 for i in range(n)]
    return t

#ou (version non optimisée)
def zero_un3(n):
    t = []
    for i in range(n):
        if i%2==0:
            t.append(0)
        else:
            t.append(1)
    return t

###Q2###
def est_element(t,a):
    """ parcours par les indices"""
    for i in range(len(t)):
        if t[i]==a:
            return True  #boucle interrompue
    return False

def est_element2(t,a):
    """ parcours par les elements"""
    return a in t

#les constantes True et False s'écrivent avec une majuscule

###Q3###
def NB(L,M):
    total = 0
    for i in range(len(L)):
        if est_element(M,L[i]):
            total = total + 1
    return total

def NB(L,M):
    total = 0
    for x in L:
        if est_element(M,x)==True:
            total += 1
    return total

#####Q4###
#Premier essai (incorrect)
def lignenulle(p):
    ligne = []
    for i in range(p):
        ligne.append(0)
    return ligne

```

```python
def listenullefaux(n,p):
    """ avec append"""
    ligne = lignenulle(p)
    tableau = []
    for i in range(n):
        tableau.append(ligne) #Attention, il s'agit à chaque fois de la même référence de ligne
    return tableau

T=listenullefaux(3,2)
T[0][1]=5 #modifie toutes les lignes

#plutôt
def lignenulle(p):
    ligne = []
    for i in range(p):
        ligne.append(0)
    return ligne

def listenulle(n,p):
    """ avec append"""
    tableau = []
    for i in range(n):
        tableau.append(lignenulle(p))
    return tableau

#ou
def listenulle1(n,p):
    """ par comprehension """
    return [[0 for i in range(p)] for i in range(n)]

#ou
def listenulle2(n,p): #exotique (inutilement complexe)
    res = [ [] for i in range(n)]
    for i in range(n):
        for j in range(p):
            res[i].append(0)
    return res

#####Q5###
#avec la fonction auxiliaire lignenulle
def L2(l):
    t = []
    for i in range (len(l)):
        t.append(lignenulle(l[i]))
    return t

#sans la fonction auxiliaire lignenulle (moins lisible)
def L2a(l):
    """ avec append"""
    tableau = []
    for i in range(len(l)):
        #formation d'une ligne de zéros
        ligne = []
        for i in range(l[i]):
            ligne.append(0)
        #utilisation de la ligne de zéros
        tableau.append(ligne)
    return tableau

#ou Pythonesque
```

```python
def L2b(l):
    """ par compréhension """
    return [[0 for i in range(l[i])] for i in range(len(l))]

#voire
def L2c(l):
    return [[0 for j in range(x)] for x in l]

#####Q6###
def A(n,a,b):
    '''optimisée en terme de complexité'''
    t = listenulle(n,n)
    for i in range(n):
        t[i][i] = 1
    for i in range(1,n):
        if i%2 == 0:
            t[0][i],t[i][0] = a,a
        else:
            t[0][i],t[i][0] = b,b
    return t

#ou
def A2(n,a,b):
    t = listenulle(n,n)
    for i in range (n):
        t[i][i] = 1
    #il faut surtout éviter une double boucle
    for j in range(1,n,2): #boucle avec un pas de 2
        t[0][j] = b
        t[j][0] = b
    for j in range(2,n,2): #boucle avec un pas de 2
        t[0][j] = a
        t[j][0] = a
    return t

#####Q7###
def T(L,n):
    a,b,c = L #alimentation des variables à partir de la liste
    m = listenulle(n,n)
    for i in range(n):
        for j in range(n): # parcours de tous les couples (i,j)
            if i == j:
                m[i][j] = b
            elif i < j:
                m[i][j] = a
            else:
                m[i][j] = c
    return m

#####Q8###
def suite(u0, n):
    u = u0
    for i in range(n):
        # a%b est le reste de de la division euclidienne de a par b
        if u%2 == 0:
            u = u//2 #// pour obtenir un entier et non un float
        else:
            u = 3 * u + 1
    return u
# ne pas oublier le ":" dans les structures de programmation
# attention à l'indentation
```

```python

#####Q9###
#Premier essai (complexité élevée)
def sommelourde(u0,n):
    """ version avec appel de suite"""
    S = 0
    for i in range(n+1):
        S = S + suite(u0,i) #Complexite O(i)
    return S #Complexite O(n**2)

#meilleure complexité - en O(n) - que l'utilisation de suite(u0,n) :
def somme(u0,n):
    u,S = u0,u0
    for i in range(n):
        if u%2==0:
            u=u//2
        else:
            u=3*u+1
        S += u #écriture abrégée de S=S+u (inspirée du langage C++)
    return S

#####Q10###
#Premier essai (complexité élevée)
def prodlourde(u0,n):
    produit = 1
    for i in range(n+1):
        u = suite(u0,i) #on le calcule ici car on va l'utiliser deux fois
        if u%2 == 0:
           produit = produit * u
    return produit

#meilleure complexité - en O(n) - que l'utilisation de suite(u0,n) :
def prod(u0,n):
    u = u0
    produit = 1
    for i in range(n+1):
        if u%2 == 0:
            produit *= u
        if u%2 == 0:
            u = u//2
        else:
            u = 3 * u + 1
    return produit
#meilleure complexité que l'utilisation de suite(u0,n) suivante :

#####Q11###
def fibo(n):
    u0,u1,u2 = 1,1,1 #u2 est alimenté à 1 pour le cas où n==0 ou n==1
    for i in range(n-1): #quand n==0 ou n==1: boucle non exécutée
        u2 = u1 + u0
        u0 = u1 #on fait glisser les valeurs de u0 et u1
        u1 = u2
    return u2

#variante avec deux variables
def fibo2(n):
    u,v = 1,1 #deux premiers termes
    for i in range(n-1):
        u,v = v,u+v
    return v

```

```python
#variant à n'utiliser que si il y a un intérêt à stocker toutes les valeurs de la suite (espace mémoire important)
def fibo3(n):
    L=[1,1]
    for i in range(n-1):
        L.append(L[i]+L[i+1])
    return L[n],L

#####Q12###
def max_SP(L):
    """ renvoie le max des sommes partielles; renvoie -inf lorsque L est vide """
    somme,maximum = 0,float('-inf')
    for i in range(len(L)):
        somme = somme + L[i]
        if somme > maximum:
            maximum = somme
    return maximum

#ou
def max_SP2(L):
    """ renvoie le max des sommes partielles; renvoie -inf lorsque L est vide """
    somme,maximum = 0,float('-inf')
    for x in L:
        somme = somme + x
        if somme > maximum:
            maximum = somme
    return maximum

#####Q13###
def somme_ligne(l):
    '''somme des éléments d'une ligne; l est supposé non vide'''
    s = 0
    for j in range(len(l)):
        s = s + l[j]
    return s

def max_somme_ligne(L):
    '''maximum des sommes des éléments d'une ligne; renvoie -inf lorsque L est vide'''
    if L == []:
        return float('-inf')
    else :
        somme = 0
        maximum = somme
        for i in range(len(L)):
            somme = somme_ligne(L[i]) #variable utilisée deux fois
            if  somme > maximum :
                maximum = somme
        return maximum

###Q14###
def bissextile(n):
    return n%400==0 or (n%4==0 and n%100!=0)

###Q15###
def jourannee(L):
    """jour numerote de 1 a 31, mois de 1 a 12"""
    nbJoursMois=(31,28,31,30,31,30,31,31,30,31,30,31) #constante

```

```
299     jour,mois,annee = L
300     #variable totalisant le nombre de jours depuis le début de l'année,
        initialisée au nombre de jours du mois courant
301     nbJours = jour
302 
303     #on ajoute les jours des mois précédant le mois donné
304     for i in range(mois-1):
305         nbJours += nbJoursMois[i]
306 
307     #on ajoute 1 jour pour les années bissextiles quand le mois est au moins
        celui de février
308     if mois >= 2:
309         nbJours += bissextile(annee) #True==1 et False ==0
310 
311     return nbJours
312 
313 
314 #ou, avec fonction auxiliaire qui donne le nb de jours des mois
315 def nbJoursMois(annee,i):
316     nb = (31,28,31,30,31,30,31,31,30,31,30,31)
317     total = nb[i]
318     if bissextile(annee) and i==1:
319         total = 29
320     return total
321 
322 def jourannee2(L):
323     jour,mois,annee = L
324     nbJours = jour
325     for i in range (mois-1):
326         nbJours = nbJours + nbJoursMois(annee,i)
327     return nbJours
328 
329 ###Q16###
330 def jourspasses(j1,m1,a1,j2,m2,a2):
331     '''envoie le nombre de jours écoulés entre j1,m1,a1 et j2,m2,a2'''
332     #nombre de jours dans la dernière année :
333     j = jourannee([j2,m2,a2])
334     #ajout du nombre de jours dans la première année :
335     j = j + (365-jourannee([j1,m1,a1])) + bissextile(a1)
336     #ajout nombre de jours des autres années :
337     for i in range (a1+1,a2):
338         j = j + 365 + bissextile(i)
339     return j
340 
341 # on suppose que 1446 pleines lunes ont été observées entre le 15 janvier
    1900 et le 14 novembre 2016
342 print(jourspasses(15,1,1900,14,11,2016)//1446)
343 
344 #ou calcul "en direct"
345 
346 N=0
347 for i in range(1900,2016):
348     N = N + 365 + bissextile(i)
349 # entre le 15 janvier 1900 et le premier 31 decembre 2015
350 N=N-14
351 # entre le 15 janvier 1900 et le premier 13 novembre 2016
352 N=N+jourannee([13,11,2016])
353 print(N//1446)
354 
355 ###Q17###
356 def est_palindrome(n):
```

```
357     '''on suppose n entier naturel'''
358     s = str(n) # conversion en chaîne de caractères
359     for i in range(len(s)//2):
360         if s[i] != s[len(s)-1-i]:
361             return False
362     return True
363 
364 ###Q18###
365 def verlan(n):
366     '''on suppose n entier naturel'''
367     s = '' #chaîne vide
368     N = str(n)
369     for i in range(len(N)):
370         s += N[len(N)-1-i]
371     return int(s)
372 
373 def verlan2(n):
374     '''on suppose n entier naturel'''
375     return int(str(n)[::-1])
376 
377 ###Q19###
378 def r(n):
379     '''on suppose n entier naturel'''
380     return n + verlan(n)
381 
382 def pal(n,N):
383     L = [n]
384     for i in range(N):
385         if est_palindrome(n):
386             return L
387         n = r(n)
388         L.append(n)
389     return False
```