TP 3 : Boucles (II), listes

Rappels sur les listes

- Les éléments d'une liste `L` sont numérotés à partir de 0. On accède à l'élément d'indice i via `L[i]`.
- La longueur d'une liste est obtenue avec la fonction `len`.
- On crée une liste vide à l'aide de `[]`, puis pour lui ajouter des éléments, il est possible d'utiliser `L.append()`.
- Il est possible de définir des listes par compréhension, par exemple `L = [k**2 for k in range(1,10)]`.
- On peut itérer sur les éléments d'une liste `L`, à l'aide de `for e in L :`
  Ou préférer itérer sur les indices de ces éléments : `for i in range(len(L)) : ... L[i]...`

Exercice 1 Écrire une fonction `supprime_doublons` qui prend comme paramètre une liste et retourne une liste contenant les mêmes éléments, mais sans doublons. PD

Exercice 2 Écrire une fonction `somme` qui prend comme paramètre une liste de nombres et retourne la somme de ces nombres. F

Exercice 3 Écrire une fonction `max_avant_min` qui prend comme paramètre une liste `L`, détermine son maximum et son minimum, et retourne `True` si le maximum apparaît dans `L` avant le minimum, et `False` sinon. PD

Tracé basique de graphiques

Nous serons régulièrement amenés cette année à tracer des graphiques, et nous utiliserons à cet effet le module `matplotlib.pyplot`.
L'usage veut qu'on l'importe sous la forme `import matplotlib.pyplot as plt`.
Si on dispose de deux listes `x` et `y` **de même longueur** n contenant respectivement les abscisses et les ordonnées de n points, alors la commande `plt.plot(x,y)` place ces n points sur un graphique, et les relie (par des segments). Donc si les éléments de `x` sont dans l'ordre croissant, cela permet de tracer approximativement le graphe une fonction.
Essayer par exemple le code suivant :

```
1 from math import sin
2 x = [(6*k)/100 for k in range(1,100)]
3 plt.close()
4 plt.plot(x,[sin(t) for t in x])
5 plt.show()
```

On remarquera la présence de la dernière ligne, sans laquelle la figure ne s'affiche pas.
La ligne `plt.close()` n'est pas obligatoire, mais si le code est exécuté alors qu'un graphique était déjà ouvert, le nouveau graphique ne s'affichera pas. Commencer par fermer un éventuel graphique déjà ouvert permet d'éviter ceci.
Essayer le même code, mais en remplaçant `plt.plot` par `plt.scatter`, qui trace un nuage de points.

Exercice 4 Série harmonique PD

Pour $n \in \mathbf{N}^*$, on pose $H_n = \sum_{k=1}^{n} \frac{1}{k}$. On admet que $H_n \underset{n\to+\infty}{\longrightarrow} +\infty$.

1. Écrire un programme qui détermine la plus petite valeur de n telle que $H_n > A$, que vous testerez avec $A = 5, A = 10$ et $A = 15$.

2. Créer une liste `H` qui contient $[H_1, H_2, \dots, H_{1000}]$ (on pourra notamment commencer par déterminer une relation liant H_{n+1} et H_n, puis utiliser une boucle **`for`**).
3. Représenter graphiquement les 1000 premiers termes de la suite $(H_n)_{n\geqslant 1}$.
   Puis, sur un même graphique, tracer la fonction ln à l'aide de la commande
   `plt.plot(range(1,1001),[log(k) for k in range(1,1001)])`
   Commenter le graphique obtenu.

▶ La suite de Syracuse, ou suite de Collatz

AD

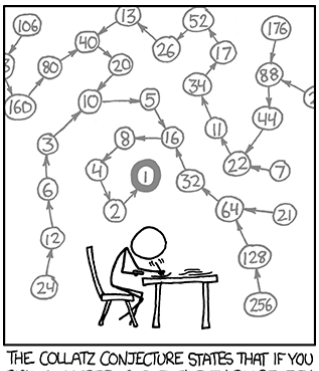
Exercice 5 Nous allons nous intéresser dans ce problème à une conjecture célèbre, connue sous différents noms : conjecture de Syracuse (du nom de l'université américaine où le problème, pourtant découvert en Suisse, se popularisa), conjecture de Collatz (du nom de son découvreur) ou conjecture $3x+1$.
Le principe est simple : étant donné un entier $a \in \mathbf{N}^*$, on appelle suite de Syracuse associée à a la suite $(u_n)_{n\in\mathbf{N}}$ définie par

$$u_0 = a \text{ et } \forall n \in \mathbf{N},\ u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

1. Écrire une fonction `syracuse(a,n)` qui calcule le $n^{\text{ème}}$ terme de la suite de Syracuse associée à a.
2. Modifier cette fonction pour qu'elle affiche sur un graphique les n premiers termes de la suite de Syracuse associée à a.
   Essayer cette fonction avec des «petites» valeurs de a (disons inférieures à 30) et $n = 30$.
   Quelle conjecture pouvez-vous émettre au sujet du comportement des suites de Syracuse ?

Il est facile de voir que si une suite de Syracuse contient le terme 4, alors les termes suivants sont $2, 1, 4, 2, 1, 4, \dots$, le cycle se répétant à l'infini.
La conjecture de Syracuse (non prouvée à ce jour) est que toute suite de Syracuse aboutit à ce cycle.
Autrement dit, que pour tout $a \in \mathbf{N}^*$, si (u_n) est la suite de Syracuse associée à a, alors il existe $n \in \mathbf{N}$ tel que $u_n = 1$.

3. Écrire une fonction `premier_un(a)` qui renvoie la plus petite valeur de n pour laquelle la suite de Syracuse (u_n) associée à a vaut 1.
4. Vérifier que la conjecture de Syracuse est vraie pour $a \in [\![2, 100\,000]\!]$.



À ce jour, on ne sait toujours pas si cette conjecture est vraie, même si elle été vérifiée numériquement pour tous les entiers inférieur à 10^{18}.
Bien que formulée dans les années 30, cette conjecture a connu son heure de gloire dans les années 60 dans les universités américaines. En pleine guerre froide, le bruit courait qu'il s'agissait alors d'un complot soviétique visant à détourner les chercheurs américains de leurs travaux.
De grosses avancées sur cette conjecture ont été publiées en 2019 par Terry Tao (médaille Fields 2006), mais le problème n'est pas clos pour autant.

5. On appelle *altitude maximale* d'un entier a la valeur maximale prise par la suite de Syracuse associée à a.
   Écrire une fonction `altitude_max` qui prend comme paramètre un entier a et retourne son altitude maximale.
6. On dit qu'un entier a est un `record` si son altitude maximale est strictement supérieure aux altitudes maximales de tous les entiers strictement inférieurs à a.
   Écrire une fonction `record`, qui prend comme paramètre un entier n et retourne une liste contenant tous les records inférieurs ou égaux à n.

7. Le *temps de vol* d'un entier a est le nombre de termes de la suite de Syracuse de premier terme a qu'il faut calculer avant de rencontrer un 1 (autrement dit, c'est le plus petit n tel que $u_n = 1$).
   Tracer sur un graphique les temps de vol des entiers de 1 à 10 000 (on préférera `plt.scatter` à `plt.plot`).
   Apprécier la beauté du graphique obtenu, sans chercher à en tirer de conclusion.

▶ Pour les plus rapides

Exercice 6 Le triangle de Pascal

AD

On rappelle que $\binom{n}{0} = 1$ et que pour $k \geqslant 1$, $\binom{n}{k} + \binom{n}{k-1} = \binom{n+1}{k}$ (identité de Pascal).

1. Écrire une fonction `Pascal` qui prend comme paramètre un entier n et retourne une liste de listes qui correspond aux $n+1$ premières lignes du triangle de Pascal.
   Autrement dit, on souhaite que pour $i \in [\![0, n]\!]$ et $j \leqslant i$, `L[i][j]` soit égal à $\binom{i}{j}$.
   Ainsi, `Pascal(3)` doit retourner `[[1],[1,1],[1,2,1],[1,3,3,1]]`.
2. Une solution qui utilise la fonction `factorial` du module `math`, consiste en la ligne suivante :
   ```
   [[factorial(i)//(factorial(j)*factorial(i-j)) for j in range(i+1)] for i in range(n+1)]
   ```
   Expliquer pourquoi cette unique ligne de commande répond bien à la question.
3. *Question facultative* : pour $n = 400$, comparer le temps de calcul des deux méthodes. Commenter en essayant d'estimer approximativement le nombre de calculs élémentaires (sommes et produits de deux nombres) que nécessite chacune des deux méthodes.

Exercice 7 Suite croissante de longueur maximale

D

Écrire une fonction qui prend comme paramètre une liste `L` formée de nombres et retourne une liste formée de termes consécutifs de `L`, croissante et de longueur maximale parmi les telles suites.

Exercice 8 Combien de dimanches ?

D

Sachant que le 1er janvier 1900 était un lundi, combien de fois au XXème siècle (1er janvier 1901–31 décembre 2000) un mois a-t-il commencé par un dimanche ?

Exercice 9 Amélioration de Syracuse

TD

La fonction `time` du module **`time`** retourne le temps en secondes depuis une date fixe (qui dépend du système d'exploitation).
Pour obtenir le temps d'exécution d'un programme, il suffit d'appeler cette fonction avant l'exécution du programme, de l'appeler une seconde fois après l'exécution et de calculer la différence de ces deux temps.
Combien de temps faut-il pour vérifier la conjecture jusqu'à $n = 10^6$?
Proposer des améliorations permettant d'aller plus loin que 10^6 tout en gardant un temps d'exécution inférieur à une minute. À titre indicatif, j'arrive à aller jusqu'à un million en un quart de seconde et jusqu'à 100 millions en moins de 20 secondes.
Quelques pistes de réflexion :

- *il suffit de s'arrêter dès qu'une suite de Syracuse devient inférieure strictement à son terme initial*
- *il suffit de tester la conjecture pour les entiers impairs*
- *sur le même principe, justifier qu'il suffit de tester la conjecture pour les entiers de la forme $4n+3$ (soit seulement un entier sur quatre). Essayer de généraliser.*

Correction des exercices du TP 3

Solution de l'exercice 1

On se propose de parcourir toute la liste, et de stocker dans une nouvelle liste les éléments déjà rencontrés.
Ainsi, pour chaque nouvel élément de la liste de départ, il suffira de vérifier s'il fait déjà partie, ou non de la liste des éléments déjà rencontrés.
Si ce n'est pas le cas, alors on l'ajoutera à cette dernière liste.

```
def supprime_doublons(L) :
        M = []
        for e in L :
                if not(e in M) :
                        M.append(e)
        return M
```

□

Solution de l'exercice 2

```
def somme(L) :
        s = 0
        for i in range(len(L)) :
                s+=L[i]
        return(S)
```

En réalité, il existe déjà une fonction Python qui a le même effet, c'est la fonction nommée sum. □

Solution de l'exercice 3

```
def max_avant_min(L) :
        m = L[0]
        M = L[0]
        im,iM = 0,0
        for i in range(1,len(L)) :
                if L[i]<m :
                        m = L[i]
                        im = i
                elif L[i]>M :
                        M = L[i]
                        iM = i
        return iM<im
```

□

Solution de l'exercice 4

1. On ajoute un à un les termes de la somme jusqu'à dépasser A :

```
A = 5
S = 1
n=1
while S<=A :
        n+=1
        S+=1/n
print(n)
```

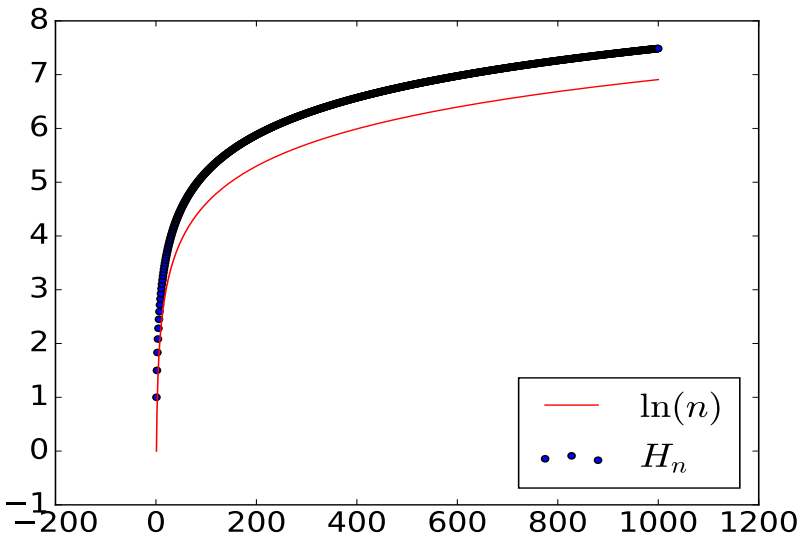
2. La relation demandée est $H_{n+1} = H_n + \dfrac{1}{n+1}$. Et donc

```
L = [1]
for n in range(2,1001) :
        L.append(L[-1]+1/n)
```

3. On utilise donc

```
import matplotlib.pyplot as plt
from math import log
plt.scatter(range(1,1001),L)
plt.plot(range(1,1001),[log(k) for k in range(1,1001)])
plt.show()
```

On obtient alors le graphique suivant



On constate alors que l'écart entre H_n et $\ln(n)$ semble tendre vers une limite finie, plus petite que 1.

□

Pour la culture

Cette limite finie s'appelle la constante γ d'Euler, et on sait peut de choses à son sujet (par exemple on ne sait pas si elle est rationnelle ou non), si ce n'est que

$$\gamma \simeq 0.577.$$

Solution de l'exercice 5

1. Une simple boucle **for** suffit :

```
def syracuse(a,n) :
        u = a
        for i in range(n) :
                if u%2==0 :
                        u=u//2
                else :
                        u = 3*u+1
        return u
```

2. Il s'agit cette fois de garder en mémoire les termes successifs calculés, ce qui se fait bien à l'aide d'une liste :

```
def syracuse_graphique(a,n) :
        u =a
        L = [a]
        for i in range(n) :
                if u%2==0 :
                        u=u//2
                else :
                        u = 3*u+1
                L.append(u)
        plt.plot(range(n+1),L)
        plt.show()
```

Lorsqu'on essaie cette fonction avec $a \leqslant 30$, alors on observe que la suite finit toujours par arriver sur 4, puis sur 2, puis sur 1.
Il est donc raisonnable de supposer que c'est toujours le cas.

Toujours ?

En fait, le cas $a = 27$ ne semble pas terminer par 4, 2, 1, mais il faut insister un peu plus : si on trace les 120 premiers termes de la suite, cela finit par arriver.

```
def premier_un(a) :
        u = a
        n = 0
        while u !=1 :
                if u%2==0 :
                        u=u//2
                else :
                        u = 3*u+1
                n+=1
        return(n)
```

4. On peut se contenter de faire appel à la fonction précédente, pour a dans l'intervalle souhaité.
Si le programme ainsi obtenu termine, c'est donc que toutes les suites de Syracuse auront abouti à 1.
S'il ne se termine pas, ce ne sera une preuve de rien du tout ! En effet, peut-être qu'en attendant suffisamment longtemps et en laissant l'ordinateur calculer suffisamment de termes, on finirait toujours par arriver sur un 1. Mais comme nous ne disposons que d'un temps fini, il faudra bien à un moment prendre la décision d'arrêter le calcul.

```
for i in range(2,100001) :
        premier_un(i)
print('La conjecture est bien vérifiée pour a inférieur à 100 000')
```

```
def altitude_max(a) :
        u = a
        max = a
        while u !=1 :
                if u%2==0 :
                        u=u//2
                else :
                        u = 3*u+1
                        if u>max :
                                max=u
```

□

Solution de l'exercice 6

```
def Pascal(n) :
    L = [[1]]
    for i in range(n) :
        M = [1]
        for j in range(i) :
            M.append(L[i][j] + L[i][j+1])
        M.append(1)
        L.append(M)
    return(L)
```

```
def factorielle(n) :
        u = 1
        for i in range(1,n+1) :
                u*=i
        return u

[[factorielle(i)//(factorielle(j)*factorielle(i-j)) for j in range(i+1)]
for i in range(n+1)]
```

Avec la première méthode, le calcul de la $i^{\text{ème}}$ ligne nécessite $i-1$ additions, soit un nombre total d'opérations de l'ordre de $1+2+\cdots+399 \simeq 400^2/2$.
En revanche, le calcul d'un coefficient de a $i^{\text{ème}}$ ligne à l'aide de la seconde méthode nécessite $2i$ multiplications : i pour calculer $i!$, et j puis $i-j$ pour calculer $j!(i-j)!$. Plus une

division. Soit un total de $2i+1$ opérations. Disons i^2 pour simplifier. Fois i termes sur la ligne, de l'ordre de $2i^2$ opérations.

Et donc le nombre total d'opérations est de l'ordre de $2\left(1^2+2^2+\cdots+400^2\right) = 2\dfrac{400\times 401\times 801}{6} \simeq \dfrac{400^3\times 2}{3}$.

Soit un nombre d'opérations considérablement plus élevé qu'avec la première méthode.

□

Solution de l'exercice 7

```
def croissante(L) :
    max = 1
    debut = 0
    fin = 1
    i = 0
    while i<len(L)-max :
        j = i+1
        while j<len(L) and L[j-1]<=L[j] :
            j+=1
        if j-i>max :
            max = j-i
            debut=i
            fin = j
        i=j
    return L[debut :fin]
```

□

Solution de l'exercice 8

Commençons par chercher quel était le premier jour de 1901 : puisque 1900 n'est pas bissextile, elle a 365 jours. Or, $365 \equiv 1$ [7], donc 1901 commence par un mardi.
Et donc le premier dimanche de 1901 est le 6 janvier.
À partir de là, nous proposons de créer une fonction qui calcule la date de chaque dimanche du XX$^{\text{ème}}$ siècle, et compte combien tombent les premiers du mois.

Remarque

Une petite astuce ici : au XX$^{\text{ème}}$ siècle, les années bissextiles sont exactement les années divisibles par 4 (ce qui n'était par exemple pas vrai au XIX$^{\text{ème}}$).

```
def dimanche_suivant(j,m,a) :
    mois = [31,28,31,30,31,30,31,31,30,31,30,31]
    if a%4==0 :
        mois[1] = 29
    j = (j+7)
    if j>mois[m-1] :
        j = j - mois[m-1]
        m = m+1
    if m>12 :
        m=1
        a +=1
    return (j,m,a)

j=6
m=1
a = 1901
tot=0
while a<2001 :
    (j,m,a) = dimanche_suivant(j,m,a)
    if j==1 :
        tot+=1
if (j,m,a) == (1,1,2001) : #au cas où...
    tot-=1
print('Le XXeme siècle a compté '+ str(tot) + ' mois débutant par un dimanche.')
```

□

Solution de l'exercice 9

Commençant par les améliorations faciles : si on vérifie la conjecture dans l'ordre croissante (pour 1, puis pour 2, puis pour 3, etc) alors il suffit de s'arrêter dès qu'une suite devient strictement inférieure à son terme initial, puisqu'on arrive alors sur une valeur à partir de laquelle on finit par rencontrer le cycle $4 - 2 - 1$.

Si u_n est impair, alors $u_{n+1} = 3u_n + 1$ sera pair, et donc $u_{n+2} = \dfrac{3u_n + 1}{2}$.
On peut, quitte à changer les suites, mais sans changer la convergence, réaliser directement l'opération $u = \dfrac{3u + 1}{2}$ lorsque u est impair.

Enfin, il est clair qu'il suffit de vérifier la conjecture pour les nombres impairs, puisqu'un nombre pair n s'écrit nécessairement sous la forme $n = 2^p(2q + 1)$, et donc après les p divisions par 2 initiales, vérifier la conjecture pour n revient à la vérifier pour $2q + 1$.
Cela a déjà pour effet de diviser par 2 le nombre d'entiers à tester !
De même, pour un entier de la forme $4n + 1$, les premières étapes vont être

$$4n + 1 \to 12n + 4 \to 6n + 2 \to 3n + 1.$$

On aboutit donc nécessairement, pour un entier de la forme $4n + 1$, à un entier strictement inférieur.
Donc il n'est pas utile de vérifie la conjecture pour les $4n + 1$.
En revanche, pour un entier de la forme $4n + 3$, on peut juste affirmer que

$$4n + 3 \to 12n + 10 \to 6n + 5 \to 18n + 16 \to 9n + 8.$$

Et alors, sans information sur la parité de n, on ne peut en dire plus.
Donc il faut bien tester tous les entiers de la forme $4n + 3$. Notons qu'il ne nous reste alors plus qu'un entier sur 4 !

On peut pousser ce principe plus loin, et chercher, à p fixé, les valeurs de $k \in [\![1, 2^p]\!]$, pour lesquelles il est indispensable de tester les $2^p n + k$, $n \in \mathbf{N}$.
Nous proposons à cet effet la fonction suivante :

```
def elimination(p) :
    L = []
    puiss = 2**p
    for i in range(1,puiss,2) :
        d = puiss
        r = i
        while d%2==0 and d>=puiss :
            if r%2==0 :
                d=d//2
                r = r//2
            else :
                r=3*r+1
                d = 3*d
        if d>=puiss :
            L.append(i)
    return(L)
```

qui retourne une liste formée des $k \in [\![1, 2^n]\!]$ qui nous intéressent.
À titre d'exemple `elimination(2)` renvoie `[3]` signifiant qu'il faut tester les $4n + 3$.
De même, `elimination(4)` renvoie `[7,11,15]` ce qui signifie qu'il suffit de tester les $16k + 7$, $16k + 11$ et $16k + 15$. Soit environ un entier sur 5 (18%).
En poussant plus loin, tout en essayant de garder un temps d'exécution raisonnable pour `elimination`, on arrive à se ramener à des entiers de la forme $2^{20} + k$, où k figure dans une liste de 27 320 entiers, soit environ un entier sur 50.
Il est alors aisé de programmer la vérification pour ces nombres :

```
def syracuse_reduite(n) :
    u = n
    while u>=n :
```

Autrement dit
On a éliminé d'office les $4n + 1$.

```
        if u%2==0 :
            u=u//2
        else :
            u = (3*u+1)//2

from time import time

def syracuse_rapide(n,p) :
    t = time()
    L = elimination(p)
    r = n//2**p+1
    for i in range(1,r) :
        l = 2**p*i
        for j in L :
            syracuse_reduite(l+j)
    return time()-t
```

Expérimentalement, on constate que $p = 16$ est à peu près aussi, voire plus rapide que les puissances suivantes, probablement car il s'agit d'une puissance de 2, et que les puissances suivantes ne réduisent que très peu le nombre de cas à traiter.
Cet algorithme me permet de vérifier la conjecture de Syracuse jusqu'à 100 millions en 20 secondes à l'aide de `syracuse_rapide(10**8,16)`.

□