# MAO-MI

***Version « 2017/2018 »***

**Jean-François Burnol**

**12 décembre 2017**

# Table des matières

# Présentation



## 1 Objectifs

Ce cours est conçu pour des étudiants sans expérience préalable en programmation.

Nous aborderons avec le langage de programmation Python3 [2] (voir *Documentation sur Python* (page v)) quelques sujets classiques en algorithmique et arithmétique.

## 2 Environnement de travail

Le cours et les exercices sont consultables par un navigateur web.

Pour faire les exercices il vous faut un environnement de programmation Python. Il y a deux possibilités principales:

**`Idle3`** Un environnement léger qui comporte un éditeur vous permettant d'y écrire le code Python et de le sauvegarder (avec extension `.py`); vous pouvez exécuter le code et le résultat apparaît dans la fenêtre d'exécution. Dans cette dernière la commande `help()` vous permet de faire afficher la documentation correspondante, par exemple `help(print)`.

**`Spyder`** Un environnement intégré beaucoup plus complet qui, dans sa configuration par défaut, comporte une grande fenêtre découpée en plusieurs panneaux : à gauche celui correspondant à l'éditeur de fichiers `.py`, en bas à droite la console d'exécution, en haut à droite le panneau dans lequel on peut faire s'afficher la documentation des modules, classes, méthodes Python.

De plus `Spyder` offre l'utilisation d'une console IPython (« Interactive Python ») qui comporte de nombreux avantages.

2. https://docs.python.org/fr/3/tutorial/index.html

Le défaut de Spyder sur les machines des salles de TPs est le temps de lancement au démarrage. Pour suivre ce cours le choix entre Idle3 ou Spyder est sans importance. Vous pourriez même vous passer complètement de l'un comme de l'autre et utiliser par exemple gedit pour les fichiers .py et une console de commande pour l'exécution, mais cela s'adresse plutôt à ceux qui connaissent déjà.

Idle3 comme Spyder ont une aide en ligne que je conseille de consulter lors de vos premières utilisations (surtout pour Spyder car il y a plusieurs raccourcis claviers at autres « astuces d'utilisation » très utiles).

Vous devriez pouvoir trouver une icône de lancement pour l'un comme pour l'autre. Au pire il devrait être possible de lancer Spyder via la commande /usr/local/anaconda3/bin/spyder & émise dans un Terminal de commandes.

## 2.1 Installation sur son ordinateur personnel

Les feuilles de TPs seront accessibles via internet.

Il est possible que votre ordinateur soit déjà équipé avec Python (et alors probablement aussi Idle), mais attention vérifiez bien qu'il a aussi une installation de Python3 et que c'est Python3 que vous utilisez. (voir *Python3 vs Python2* (page vi)).

Pour l'installation sur votre ordinateur personnel, je recommande Anaconda [3], qui par défaut fait une installation très complète (attention, occupe plusieurs centaines de méga-octets). On peut ensuite la mettre à jour facilement par la commande conda. C'est ce qui est installé sur les machines dans les salles de TPs (mais la commande conda ne peut être utilisée que par les informaticiens gérant les machines).

Après l'installation sur votre ordinateur personnel vous pourrez peut-être lancer le programme Spyder par une icône appropriée ou à défaut en tout cas en ligne de commande (Linux ou Mac) par spyder &; sur Windows il faudra peut-être exécuter quelque chose dans le style Anaconda\Scripts\spyder.bat.

## 2.2 Utilisation en salles de TPs

Lors des premières utilisations que ce soit de Idle3 ou de Spyder, veillez à bien vous assurer que vous comprenez où sont sauvegardés vos fichiers de travail.

Je conseille que vous créiez un répertoire tpsMA0 dédié pour y stocker vos fichiers pendant le semestre. Vous pourrez informer Spyder via Outils::Préférences::Répertoire de travail global qu'il s'agit là de votre répertoire de travail.

En ce qui concerne Spyder vérifiez que le panneau en bas à droite présente bien un volet IPython ; ce volet doit comporter une invite du type :

```
In [1]:
```

C'est différent de l'invite >>> d'une console Python standard. La console IPython est plus puissante.

Tapez votre premier code dans la console IPython du panneau en bas à droite :

3. http://continuum.io/downloads

```
In [1]: print('Bonjour !')
Bonjour !

In [2]: 2**100
Out[2]: 1267650600228229401496703205376

In [3]: for i in range(10):
   ...:     print(2**i, end=', ')
   ...: print(2**10)
   ...:
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024

In [4]:
```

**Important :** Dans les blocs d'instructions, l'indentation est au cœur de la syntaxe Python. Elle sera automatiquement insérée par la console IPython si la ligne précédente se termine par un `:`. Contrairement à d'autres langages il n'y pas de terminateur explicite genre `end` ou `fi` etc… c'est le retour à moins d'indentation qui signale la fin d'un sous-ensemble après un `for` ou autre `while`.

Pour terminer un bloc taper une première fois retour-chariot pour avoir une ligne vide, puis un second retour chariot (touche `Entrée`) qui lancera l'exécution du bloc d'instructions.

# 3 Documentation sur Python

Les bases du langage Python3 sont simples.

## 3.1 Liens utiles

**https://docs.python.org/fr/3/tutorial/index.html** Le tutoriel officiel (en français) pour Python3.

**https://docs.python.org/fr/3/** Site principal de la documentation (partiellement en français) de Python3.

**http://perso.limsi.fr/pointal/python:courspython3** Un livre de Robert Cordeau et Laurent Pointal, librement téléchargeable au format PDF.

**http://inforef.be/swi/python.htm** Site de Gérard Swinnen avec les versions successives de son livre *Programmer avec Python*, que l'on peut également librement télécharger au format PDF.

**http://python.developpez.com/cours/apprendre-python3/** Le livre de Gérard Swinnen consultable en ligne au format HTML.

Les deux références suivantes sont d'une lecture agréable mais les exemples donnés utilisent Python2. Il y a plusieurs différences essentielles entre Python3 et Python2 et il faut bien les connaître, voir la section suivante *Python3 vs Python2* (page vi).

**http://python-prepa.github.io/intro.html** Extrait d'une formation à l'ENS de Paris pour des professeurs de classes préparatoires. La page donne aussi un aperçu de l'environnement de travail `Spyder` que nous utiliserons. (*documente Python2*)

**http://www.fil.univ-lille1.fr/~marvie/python/chapitre1.html** Rédigé par un collègue informaticien Raphaël Marvie de l'université de Lille (*documente Python2*).

De la documentation en anglais :

**https://docs.python.org/3/tutorial/introduction.html** Le tutoriel officiel de Python3 (en anglais).

**https://docs.python.org/3/reference/index.html** La documentation officielle du langage (en anglais).

### 3.2 Python3 vs Python2

Il y a deux variantes de Python, nous utiliserons la plus récente, Python3. Cependant beaucoup de documentation sur internet est encore pour la version antérieure Python2. Voici quelques différences importantes :

1. en Python3 il y a un seul type d'entier, qui n'est pas limité en taille.
2. en Python3 la division de deux entiers donne un nombre « réel ». Par exemple :

```
>>> 10/7
1.4285714285714286
```

Alors qu'en Python2 on aurait obtenu :

```
>>> 10/7
1
```

Il faut utiliser // pour la division entière (quotient euclidien si le diviseur est positif) en Python3 :

```
>>> 10//7, -10//7
(1, -2)
```

---

**Note :** La fonction `divmod(a,b)` donne à la fois la division entière et le reste, en une seule opération.

```
>>> divmod(117,21)
(5, 12)
>>> q, r = divmod(117,21); print(q, r, 21*q+r)
5 12 117
>>> divmod(117,-21) # ici ce n'est pas la division Euclidienne car r < 0 !
(-6, -9)
>>> q, r = divmod(117,-21); print(q, r, -21*q+r)
-6 -9 117
```

---

3. en Python3 `print` est une fonction et s'utilise obligatoirement avec des parenthèses.
4. en Python3 `range(5)` n'est pas une liste explicite `[0, 1, 2, 3, 4]` mais un objet itérateur qui occupe moins de place en mémoire (en Python2 on aurait écrit `xrange(5)`). Pour obtenir en Python3 l'objet de type liste qui est fourni par `range(5)` en Python2, il faut faire `list(range(5))`, ou `[x for x in range(5)]`.

```
>>> range(5)
range(0, 5)
>>> list(range(5))
```

```
[0, 1, 2, 3, 4]
>>> [x for x in range(5)]
[0, 1, 2, 3, 4]
```

*Date de dernière modification* : 06-09-2017 à 21:44:54.

# 1

# Feuille de travaux pratiques 1

*Date de dernière modification* : 24-10-2017 à 14:58:00.



**Note :** Une bonne habitude si vous utilisez `Spyder` est de précéder chacune de vos définitions de fonctions Python par un `# %%` car cela permet à `Spyder` de reconnaître des « blocs d'instructions » que l'on peut exécuter séparément via sa commande intitulée `Exécuter la cellule`.

## 1.1 Algorithme d'Euclide

### 1.1.1 pgcd(a, b)

Rappel: le pgcd d'entiers (éventuellement négatifs) $a_1, \ldots, a_n$ est l'unique entier positif ou nul $d$ tel que l'idéal $a_1\mathbb{Z} + \cdots + a_n\mathbb{Z}$ est $d\mathbb{Z}$. On démontre que $d$ divise chaque $a_k$ et que tout entier $x$ qui divise tous les $a_k$ est un diviseur de $d$, d'où l'appellation de « plus grand commun diviseur ». Le pgcd ne peut être nul que si tous les $a_k$ sont nuls. Dans ce cas très spécial « plus grand » ne colle plus à la réalité.

Voici une implémentation en Python pour le pgcd de deux entiers. C'est le moment de vous assurer que vous avez lu au moins un peu le Tutoriel [4] et donc que vous comprenez la syntaxe utilisée, que je ne vais pas commenter plus ici.

```
def pgcd(a, b):
    """Calcule le PGCD par l'algorithme d'Euclide.

    Le résultat est positif, et nul uniquement si a==b==0. Les arguments
    sont des entiers éventuellement négatifs.

    Exemples
    --------

    >>> pgcd (121, 143)
    11
    >>> pgcd (-1000, -225)
    25
    """
    while b:
        a, b = b, a % b
    return abs(a)
```

### 1.1.2 La division en Python3

Cette fonction *pgcd(a, b)* (page 2) admet des arguments négatifs, il faut donc être au courant de ce que fait `a % b` dans ce cas. En Python la formule suivante est toujours valable:

```
A == (A // B) * B + A % B
```

J'y ai utilisé == et non pas le = puisque le reste de la formule utilise des notations de Python.

Cela ressemble à la formule de la division euclidienne:

$$a = qb + r$$

avec $q$ le quotient et $r$ le reste. Mais en mathématiques la division euclidienne assure $0 \leqslant r < |b|$: le reste est toujours positif ou nul quel que soit le signe de $b$ (non nul).

Lorsque `B > 0` alors en effet Python calcule le quotient `A // B` exactement selon la division euclidienne, *quel que soit le signe de* `A`. Cela est très utile car le modulo `A % B` est donc *périodique* dans la variable `A` (de période `B`):

4. https://docs.python.org/fr/3/tutorial/index.html

```
(A + B) // B == 1 + A // B
(A + B) % B  == A % B
```

*Ce n'est pas le cas dans d'autres langages de programmation qui conviennent au contraire que le signe du modulo est celui de* A. Il est très utile que Python ait fait le choix de la périodicité du modulo.

Pour `B < 0`, Python calcule le quotient `A // B` par la formule `(-A) // (-B)`. Cela veut dire (vérifiez-le) que `A % B` est *l'opposé* du reste dans la division euclidienne de `-A` par `-B`. Ainsi `A % B` est alors nul ou négatif.

Quels que soient les signes de `A` et `B`, on a (**Démontrez-le** !):

```
(A + B) // B == 1 + A // B
(A + B) % B  == A % B
```

Dans la boucle `while` du code ci-dessus, si le premier `b` est `< 0`, tous les `b` (et donc les `a`) ultérieurs resteront donc `< 0` jusqu'à la sortie de boucle déclenchée par le premier `b` nul.

Le `abs(a)` final est donc nécessaire exactement pour ces deux cas:
- le `b` initial est nul, et le `a` initial est négatif strictement,
- le `b` initial est strictement négatif.

On pourrait décider de coder de cette manière:

```
def pgcd(a, b):
    """Calcule le PGCD par l'algorithme d'Euclide.

    Le résultat est positif, et nul uniquement si a==b==0. Les arguments
    sont des entiers éventuellement négatifs.

    Exemple:
    >>> pgcd (121, 143)
    11
    """
    a, b = abs(a), abs(b)
    while b:
        a, b = b, a % b
    return a
```

La boucle `while` ne manipule alors que des entiers positifs.

Question: que se passe-t-il si `0 < a < b` ?

### 1.1.3 bezout(a, b)

Mais nous voulons un algorithme *étendu* qui calculera des coefficients de Bézout, c'est-à-dire `u` et `v` tels que `u * a + v * b == pgcd(a, b)`. Si on remplaçait dès le début de la procédure `a` et `b` par leurs valeurs absolues il nous faudrait garder une trace de leurs signes initiaux, pour ajuster les signes des `u` et `v` obtenus. On va donc plutôt conserver le style du premier code.

```
def bezout(a, b):
    """Renvoie un tuple (d, u, v) avec d le pgcd et u*a+v*b==d.

    Exemples
    --------
```

```
    >>> bezout(121, 143)
    (11, 6, -5)

    >>> bezout(-1000, -225)
    (25, 2, -9)
    """
    u, uu = 1, 0
    v, vv = 0, 1
    while b:                    # tant que b est non nul, itérer.
        q, r = divmod(a, b)     # calcule quotient et reste d'un coup.
        a, b = b, r             # r remplace b, b remplace a.
        u, uu = uu, u - q * uu  # formules mystérieuses.
        v, vv = vv, v - q * vv
    if a < 0:
        a, u, v = -a, -u, -v
    return a, u, v
```

**Exercice**

Cet exercice utilise les notations mathématiques plutôt que celles du langage Python.

1. Notons $A$ et $B$ les valeurs initiales de $a$ et $b$. Montrez qu'après `k` itérations de la boucle `while`, on a (notations mathématiques ici, pas celles de Python, mais $uu$ bien sûr n'est pas le produit de $u$ par $u$ mais une variable notée $uu$):
   — $u \cdot b - uu \cdot a = (-1)^k B$,
   — $v \cdot b - vv \cdot a = (-1)^{k-1} A$,
   — et $u \cdot vv - uu \cdot v = (-1)^k$.
   Indication: on procédera par récurrence en établissant qu'à chaque itération les expressions en question ne font que changer de signe.
2. En déduire que si la boucle `while` se termine au bout de `N` étapes on a, en notant `d` la dernière valeur de `a` à la sortie de la boucle (qui est donc à ce stade seulement au signe près le `pgcd` de $A$ et $B$):
   — $\frac{B}{d} = (-1)^{N-1} uu$,
   — $\frac{A}{d} = (-1)^N vv$.
   Confirmer alors la formule $u \cdot A + v \cdot B = a$ en sortie de boucle.
3. Justifier que les dernières lignes du code permettent de produire comme promis le `pgcd` avec le bon signe, et une identité de Bézout associée.
4. Une fois connue une solution particulière $(u, v)$ comment s'obtiennent toutes les autres solutions de $au + bv = \text{pgcd}(a, b)$ ?

### 1.1.4 nb_etapes_euclide(a, b)

**Exercice**

Implémenter une fonction `nb_etapes_euclide(a, b)` qui renvoie le nombre d'étapes (i.e. le nombre de divisions) dans le calcul du `pgcd` par l'algorithme d'Euclide.

```python
def nb_etapes_euclide(a, b):
    """Renvoie le nombre de divisions lors de l'algorithme d'Euclide.

    Exemples
    --------

    >>> nb_etapes_euclide(123456789123456789, 987654321987654321)
    3
    >>> nb_etapes_euclide(4907536168182147272, 7341095255814111957)
    30
    >>> nb_etapes_euclide(4977239559777587284, 5491444045422016999)
    29
    """
    k = 0                # un compteur d'itérations
    while b:
        a, b = b, a % b
        k += 1           # incrémenter k à chaque itération
    return k
```

### 1.1.5 nb_etapes_moyen_pour_euclide(N, reps)

---

**Exercice**

Implémenter une fonction `nb_etapes_moyen_pour_euclide(N, reps)` qui, en s'aidant de la fonction `random.randrange()`[5], choisit `reps` fois de suite un couple au hasard de deux entier `a` et `b` de `N` chiffres, évalue le nombre d'étapes lors du calcul des `pgcd`, et en fait la moyenne. Puis cette moyenne est normalisée en la divisant par `N` (qui est le nombre de chiffres des entiers aléatoires `a` et `b`).

Ceci donne donc quelque chose qu'on peut appeler le « coût par chiffre décimal » de l'algorithme d'Euclide.

---

```python
def nb_etapes_moyen_pour_euclide(N, reps):
    """Estime le nombre d'étapes « par chiffre » nécessaire pour Euclide.

    Fait reps répétitions et renvoie l'« espérance » du nombre d'étapes,
    divisée par le nombre de chiffres N.
    """
    Min = 10 ** (N - 1)
    Max = 10 * Min
    # note: Python ne procède pas en stockant d'abord en mémoire toutes les
    # valeurs à additionner, puisque nous n'avons pas mis ici les crochets
    # [...] qui créerait une liste avec toutes ces valeurs. Donc on ne doit
    # pas s'inquiéter de l'impact sur la mémoire même si reps est très grand.
    E = sum(nb_etapes_euclide(randrange(Min, Max),
                              randrange(Min, Max)) for i in range(reps))
    return E / (reps * N)
```

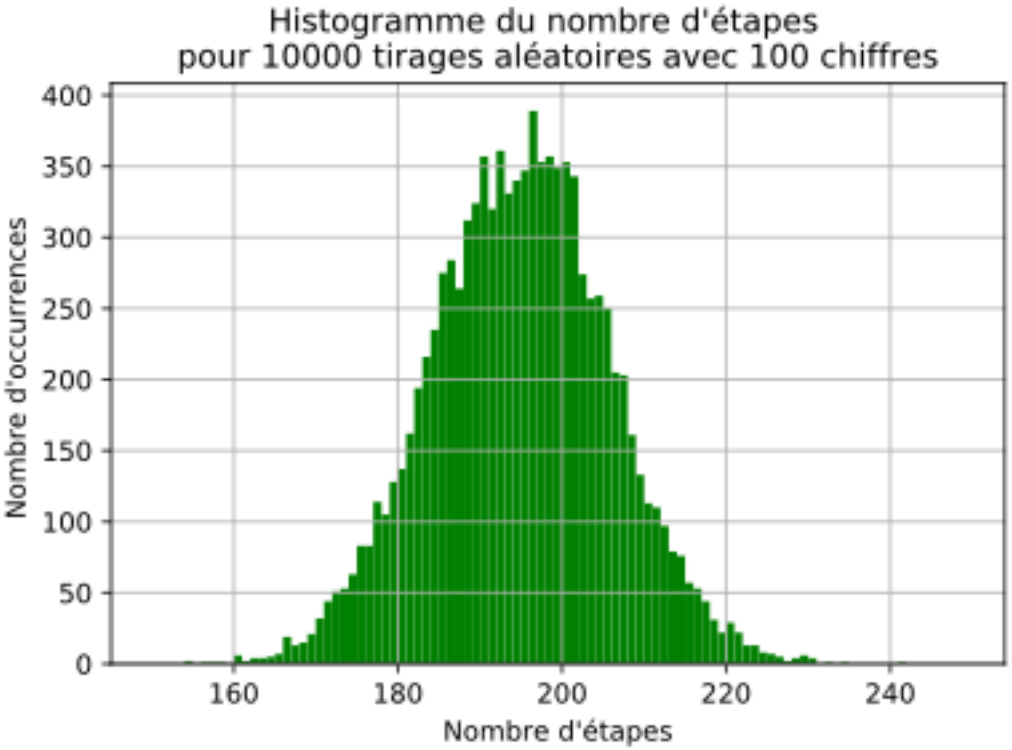
Donne par exemple :

5. https://docs.python.org/3/library/random.html#random.randrange

```
>>> for N in range(15, 50, 5):
...     print(N, nb_etapes_moyen_pour_euclide(N, 1000))
...
15 1.9748666666666668
20 1.95885
25 1.9582
30 1.9556333333333333
35 1.9512
40 1.956475
45 1.9536666666666667
>>> for N in [50, 75, 100]:
...     print(N, nb_etapes_moyen_pour_euclide(N, 100000))
...
50 1.9484474
75 1.9455717333333333
100 1.9443993
```

Il y a un résultat mathématique qui dit en effet que lorsque `N` tend vers l'infini ce « nombre d'étapes nécessaire par chiffre décimal » converge vers $12\frac{(\log 2)(\log 10)}{\pi^2} \approx 1.940540\ldots$.

Cet histogramme suggère une loi-limite « gaussienne » (confirmée par des résultats théoriques établis à partir des années soixante-dix):




## 1.2 Exponentiation rapide

### 1.2.1 `puissance(n, m)`

Dans Python, on peut faire des calculs avec de grands entiers, par exemple :

```
In [1]: 2 ** 100
Out[1]: 1267650600228229401496703205376

In [2]: 3 ** 100
Out[2]: 515377520732011331036461129765621272702107522001
```

Le symbole pour l'exponentiation (i.e. les puissances) est **. Attention, en Python le symbole ^ est utilisé pour le « bitwise xor [6] », c'est-à-dire l'addition modulo 2, bit par bit (i.e. sans retenue) sur les représentations binaires, ce qui n'a absolument rien à voir avec les puissances.

---

**Note :** La fonction `pow()` [7] peut aussi être utilisée :

```
>>> pow(2, 10)
1024
>>> pow(3, 10)
59049
```

Cette fonction admet un troisième argument optionnel `N` pour calculer « modulo `N` ».

```
>>> pow(3, 10, 13)
3
```

---

Bref, imaginons que Python ne connaisse pas **, et qu'on veuille définir une fonction `puissance(n, m)` qui calcule l'entier `n` à la puissance `m`. On peut utiliser une approche récursive suivant l'algorithme :

```
si m est nul renvoyer 1
sinon renvoyer n * puissance(n, m - 1)
```

---

**Exercice**

Faites l'implémentation en Python de cet algorithme. Votre fonction `puissance(n, m)` ne vérifiera pas que `n` et `m` sont bien des entiers positifs, cependant les cas `n=0` ou `m=0` doivent être correctement traités.

---

```
def puissance(n, m):
    """Évalue n à la puissance m.

    Procède par récursion sur m.
    Attention : boucle infinie si m<0 !

    Exemples
    --------

    >>> puissance(3, 0)
    1
    >>> puissance(0, 3)
    0
    >>> puissance(0, 0)
```

6. https://docs.python.org/3/reference/expressions.html#binary-bitwise-operations
7. https://docs.python.org/3/library/functions.html#pow

```
    1
    >>> puissance(3, 100)
    515377520732011331036461129765621272702107522001
    >>> puissance(3, 101)
    1546132562196033993109383389296863818106322566003
    """
    if m == 0:
        return 1  # après la ligne avec return, tout ce qui suit sera ignoré
    return n * puissance(n, m - 1)  # appel récursif
```

Vérifions que `puissance(2, 100)` et `puissance(3, 100)` donnent les bons résultats.

```
In [4]: puissance(2, 10)
Out[4]: 1024

In [5]: puissance(2, 100)
Out[5]: 1267650600228229401496703205376

In [6]: puissance(3, 100)
Out[6]: 515377520732011331036461129765621272702107522001
```

Est-ce aussi correct pour $0^0$ et les $0^m, m > 0$ ?

On voudrait maintenant mesurer l'efficacité d'une telle fonction. Pour cela, dans une console IPython, on dispose de `%timeit`:

```
In [12]: %timeit -n 10000 puissance(2, 100)
10000 loops, best of 3: 22.9 µs per loop

In [13]: %timeit -n 10000 2 ** 100
10000 loops, best of 3: 22.6 ns per loop
```

Bon c'est pas brillant, on est au moins 1000 fois plus lent que la routine native.

Une autre remarque : si `n==1` (ou aussi si `n==0` et j'utilise la notation Python pour les tests d'égalité) il est un peu bête de suivre toute la récursion. Mais si on teste l'égalité `n==1` c'est un peu bête aussi de le faire à chaque étape de la récursion ! donc il faudrait faire deux routines, la seconde, récursive, étant appelée par la première seulement pour $n \neq 0, 1$.

### 1.2.2 puissanceiter(n, m)

---

**Exercice**

Faites une fonction `puissanceiter(n, m)` qui sera non plus récursive mais itérative, c'est-à-dire avec une variable locale et une boucle. Profitez-en pour traiter plus efficacement les cas `n==0` et `n==1` (au prix d'un léger sur-coût pour les autres, nécessairement).

---

```
def puissanceiter(n, m):
    """Évalue n à la puissance m.

    Exemples
    --------
```

```
    >>> puissanceiter(3, 0)
    1
    >>> puissanceiter(0, 3)
    0
    >>> puissanceiter(0, 0)
    1
    >>> puissanceiter(3, 100)
    515377520732011331036461129765621272702107522001
    >>> puissanceiter(3, 101)
    1546132562196033993109383389296863818106322566003
    """
    if n == 0:
        if m == 0:
            return 1
        else:
            return 0
    if n == 1:
        return 1
    p = 1
    while m > 0:
        p = n * p
        m = m - 1  # ce m est ici une variable locale
    return p
```

```
In [18]: puissanceiter(2, 10), puissanceiter(2, 100)
Out[18]: (1024, 1267650600228229401496703205376)

In [19]: puissanceiter(3, 100)==puissance(3, 100)
Out[19]: True
```

Que donne `puissanceiter` du point de vue du temps d'exécution ?

```
In [22]: %timeit -n 10000 puissanceiter(17, 100)
10000 loops, best of 3: 14.2 µs per loop

In [23]: %timeit -n 10000 puissance(17, 100)
10000 loops, best of 3: 21.6 µs per loop
```

On a gagné un peu.

### 1.2.3 puissanceiteravecfor(n, m)

**Exercice**

La solution proposée pour *puissanceiter(n, m)* (page 8) utilise une boucle `while` qui décrémente un compteur. Mais il est plus « pythonesque », et aussi plus efficace, d'utiliser pour ce genre de choses une boucle `for` avec un index parcourant un `range()`. Le faire.

```
def puissanceiteravecfor(n, m):
    """Évalue n à la puissance m.

    Exemples
    --------
```

```
    >>> puissanceiteravecfor(3, 0)
    1
    >>> puissanceiteravecfor(0, 3)
    0
    >>> puissanceiteravecfor(0, 0)
    1
    >>> puissanceiteravecfor(3, 100)
    515377520732011331036461129765621272702107522001
    >>> puissanceiteravecfor(3, 101)
    1546132562196033993109383389296863818106322566003
    """
    if n == 0:
        if m == 0:
            return 1
        else:
            return 0
    if n == 1:
        return 1
    p = 1
    for i in range(m):
        p = n * p  # ou aussi p *= n
    # on aura donc multiplié exactement m fois par n
    return p
```

C'est mieux que notre implémentation précédente :

```
In [92]: %timeit puissanceiteravecfor(2, 100)
100000 loops, best of 3: 8.5 µs per loop

In [93]: %timeit puissanceiter(2, 100)
100000 loops, best of 3: 13.2 µs per loop

In [94]: puissanceiteravecfor(2, 101)==2 ** 101
Out[94]: True
```

Voir aussi plus bas la comparaison d'efficacité entre *fiboiteratif(n)* (page 18) et *fiboiteratifavecfor(n)* (page 20).

### 1.2.4 puissancerap(n, m)

On va maintenant utiliser un algorithme plus sioux :

```
si m est nul renvoyer 1
si m est pair renvoyer puissance(n * n, m // 2)
si m est impair renvoyer n*puissance(n * n, m // 2)
```

**Exercice**

Prouver que cet algorithme calcule $n^m$ et implémenter le de manière récursive avec comme nom `puissancerap` (« rap » pour « rapide »).

```python
def puissancerap(n, m):
    """Exponentiation entière rapide récursive.

    Exemples
    --------

    >>> puissancerap(3, 0)
    1
    >>> puissancerap(0, 3)
    0
    >>> puissancerap(0, 0)
    1
    >>> puissancerap(3, 100)
    515377520732011331036461129765621272702107522001
    >>> puissancerap(3, 101)
    1546132562196033993109383389296863818106322566003
    """
    if m == 0:
        return 1
    if m & 1:
        return n * puissancerap(n * n, m >> 1)
    else:
        return puissancerap(n * n, m >> 1)
```

**Note :** Je rappelle qu'en Python un entier non nul est comme `True` dans un test avec `if`. Donc `if m % 2:` peut être utilisé pour voir si `m` est impair. Mais il est plus efficace d'utiliser comme test `if m & 1:`. Ce `&` est le ET « bitwise », et comme la représentation binaire de `1` a un unique bit non nul, le `m & 1` donne `1` si `m` est impair, `0` sinon.

Comme on doit à la fois tester la parité et diviser par 2, on aurait pu aussi envisager l'emploi de `divmod(m, 2)`.

J'ai utilisé `m >> 1` pour diviser par 2, voir Shifting operations [8] pour les opérateurs >> et <<.

```
In [28]: puissancerap(2, 100)
Out[28]: 1267650600228229401496703205376

In [29]: puissancerap(2, 100)==puissanceiter(2, 100)
Out[29]: True

In [30]: puissancerap(3, 100)==puissance(3, 100)
Out[30]: True
```

On va regarder si on a gagné en rapidité:

```
In [32]: %timeit -n 10000 puissancerap(17, 100)
10000 loops, best of 3: 2.62 µs per loop
```

Bon , c'est pas trop mal. Ça reste infiniment plus lent que le `n ** m` natif, mais c'est mieux que notre premier essai.

8. https://docs.python.org/3/reference/expressions.html#shifting-operations

### 1.2.5 puissancerapiter(n, m)

**Exercice**

Faites maintenant `puissancerapiter` de manière itérative et non plus récursive afin de gagner un peu encore en efficacité.

```
def puissancerapiter(n, m):
    """Évalue n à la puissance m.

    Exemples
    --------

    >>> puissancerapiter(3, 0)
    1
    >>> puissancerapiter(0, 3)
    0
    >>> puissancerapiter(0, 0)
    1
    >>> puissancerapiter(3, 100)
    515377520732011331036461129765621272702107522001
    >>> puissancerapiter(3, 101)
    1546132562196033993109383389296863818106322566003
    """
    if m == 0:
        return 1
    p = 1
    while m > 1:
        if m & 1:   # teste si m est impair
            p *= n  # si oui multiplier p par n
        m = m >> 1  # variantes: m >>= 1 ou m = m // 2 ou m //= 2
        n = n * n   # ou aussi n *= n
    return p * n
```

```
In [35]: puissancerapiter(2, 100)
Out[35]: 1267650600228229401496703205376

In [36]: 2 ** 100
Out[36]: 1267650600228229401496703205376

In [37]: puissancerapiter(17, 100)==puissancerap(17, 100)
Out[37]: True

In [38]: puissancerapiter(17, 99)==puissance(17, 99)
Out[38]: True
```

Le code a l'air de marcher, le comprenez-vous ? Au niveau du temps d'exécution :

```
In [41]: %timeit -n 10000 puissancerapiter(2, 500)
10000 loops, best of 3: 2.47 µs per loop

In [42]: %timeit -n 10000 puissancerap(2, 500)
10000 loops, best of 3: 3.37 µs per loop

In [43]: %timeit -n 10000 puissanceiter(2, 500)
```

```
10000 loops, best of 3: 75.5 µs per loop

In [44]: %timeit -n 10000 puissance(2, 500)
10000 loops, best of 3: 121 µs per loop
```

C'est pas mal, on voit qu'on a gagné par exemple un facteur d'environ 50 sur notre première implémentation, pour `2 ** 500`. Mais le calcul natif est encore cent fois plus rapide :

```
In [45]: %timeit -n 10000 2 ** 500
10000 loops, best of 3: 25.9 ns per loop
```

Comme 2 est peut-être spécial on peut essayer autre chose comme 357:

```
In [46]: 357 ** 1000==puissancerapiter(357, 1000)
Out[46]: True

In [47]: %timeit -n 10000 357 ** 1000
10000 loops, best of 3: 22.6 ns per loop

In [48]: %timeit -n 10000 puissancerapiter(357, 1000)
10000 loops, best of 3: 38.1 µs per loop
```

Bon, en fait c'est pire, on est à nouveau plus de mille fois plus lent, mais avec la première implémentation c'était sûrement pire. Ah, en fait c'était bien pire :

```
In [49]: %timeit -n 1000 puissance(357, 1000)
         (énormément de lignes supprimées)
          File "<ipython-input-3-72f2e00765d6>", line 2, in puissance
    if m==0:

RuntimeError: maximum recursion depth exceeded in comparison
```

Notre `puissance` récursive ne passe pas. Et pour `puissanceiter`:

```
In [50]: %timeit -n 1000 puissanceiter(357, 1000)
1000 loops, best of 3: 379 µs per loop
```

Bon, elle passe, mais est dix fois plus lente que `puissancerapiter`. Cette dernière reste néanmoins, on l'a vu, plus de mille fois plus lente (sur mon ordinateur et avec mon Python) pour le calcul de `357 ** 1000` en comparaison avec l'implémentation native. J'ai essayé de comparer avec :

```
def pythonpuissance(n, m):
    return n ** m
```

pour ajouter un coût lié au passage de paramètres, mais ça n'a rien changé.

### 1.2.6 puissancerapitervariante(n, m)

L'algorithme récursif sous-jacent à de *puissancerap(n, m)* (page 10) était :

```
si m est nul renvoyer 1
si m est pair renvoyer puissance(n * n, m // 2)
si m est impair renvoyer n * puissance(n * n, m // 2)
```

Dans la version itérative *puissancerapiter(n, m)* (page 12) on a exécuté des multiplications suivant la parité de `m`, autrement dit c'est comme si on avait examiné l'écriture binaire de `m` de la droite vers la gauche (des bits les moins significatifs vers les plus significatifs).

---

**Exercice**

Mettre au point une fonction `puissancerapitervariante(n, m)` qui procédera itérativement mais avec l'ordre des opérations gouverné par la succession des bits de l'exposant du plus significatif vers le moins significatif.

```
Ainsi, pour m == 198, qui vaut 11000110 en binaire,
l'algorithme effectuera les opérations :

  p = n           # (initialisation, le 1 dominant)
  p = n * p * p   # (le deuxième 1 en partant de la gauche)
  p = p * p       # (le premier 0)
  p = p * p       # (le deuxième 0)
  p = p * p       # (le troisième 0)
  p = n * p * p   # (etc...)
  p = n * p * p   #
  p = p * p       # (le dernier chiffre binaire est un zéro)

soit encore successivement les valeurs :

  n, n ** 3, n ** 6, n ** 12, n ** 24, n ** 49, n ** 99, n ** 198

correspondant aux exposants (en binaire) :

  1, 11, 110, 1100, 11000, 110001, 1100011, 11000110

La valeur finale est bien le n ** 198 escompté.
```

---

```python
def puissancerapitervariante(n, m):
    """Renvoie n à la puissance m.

    Exponentiation rapide, codée de manière itérative en utilisant
    tacitement les chiffres binaires de m en partant du plus significatif.

    Exemples
    --------

    >>> puissancerapitervariante(3, 0)
    1
    >>> puissancerapitervariante(0, 3)
    0
    >>> puissancerapitervariante(0, 0)
    1
    >>> puissancerapitervariante(3, 100)
    515377520732011331036461129765621272702107522001
    >>> puissancerapitervariante(3, 101)
    1546132562196033993109383389296863818106322566003
    >>> puissancerapitervariante(2, 10)
    1024
    >>> puissancerapitervariante(3, 10)
    59049
```

```
    >>> puissancerapitervariante(7, 50)
    1798465042647412146620280340569649349251249
    """
    # assert type(m)==int and m>=0, "L'exposant doit être un entier positif!"
    if m == 0:
        return 1
    k = m.bit_length()  # donc m a k chiffres en binaire, le premier est un 1
    M = 1 << (k - 1)    # c'est 2**(k-1), plus grande puissance de 2 <= m
    p = n   # au lieu de débuter par p = 1 on part avec p = n directement
    # donc on a déjà tenu compte du chiffre binaire dominant
    m -= M  # et on le retire de m
    k -= 1  # m a un chiffre de moins qu'au départ
    while k > 0:
        # pourquoi pas while m > 0 ? expliquez !
        p *= p      # on remplace p par p au carré car il y a encore
        # au moins un chiffre binaire
        M >>= 1     # et on divise M par 2
        # faut-il multiplier le p actuel par n ?
        # c'est-à-dire, le chiffre binaire qui vient est-il 1 ?
        if m >= M:  # oui
            p *= n
            m -= M  # on retire le 1 binaire dominant
        # et on boucle
        k -= 1  # bien sûr en ayant décrémenté k
        # donc il aurait été plus pythonesque de faire un « for i in range(k) »
    return p
```

### Exercice

Dans le code donné en solution pour l'exercice précédent on aurait pu faire quelque chose comme :

```
def puissancerapitervarvar(n, m):
    """Renvoie n à la puissance m.

    Exponentiation rapide, codée de manière itérative en utilisant
    les chiffres binaires de m à partir du plus significatif.

    Exemples
    --------

    >>> puissancerapitervarvar(3, 0)
    1
    >>> puissancerapitervarvar(0, 3)
    0
    >>> puissancerapitervarvar(0, 0)
    1
    >>> puissancerapitervarvar(3, 100)
    515377520732011331036461129765621272702107522001
    >>> puissancerapitervarvar(3, 101)
    1546132562196033993109383389296863818106322566003
    >>> puissancerapitervarvar(2, 10)
    1024
    >>> puissancerapitervarvar(3, 10)
    59049
    >>> puissancerapitervarvar(7, 50)
    1798465042647412146620280340569649349251249
```

```
    """
    if m == 0:
        return 1
    p = n
    for i in bin(m)[3:]:
        p *= p
        if i == '1':  # ou « if ord(i) == 49: », peut-être plus efficace !
            p *= n
    return p
```

Expliquez. Pour comparer l'efficacité avec la méthode donnée en solution, il faudrait modifier le code pour ne pas faire les opérations de multiplication mais uniquement la boucle, et tester avec de très longs (i.e. grands) exposants `m`. Je n'ai pas fait ces tests.

## 1.3 Fibonacci

### 1.3.1 `fibostupide(n)`

La célèbre suite de Fibonacci est définie par $F_0 = 0, F_1 = 1$, et $\forall n\ :\ F_{n+2} = F_{n+1} + F_n$ (aussi pour $n < 0$, mais nous nous intéresserons principalement aux indices positifs).

**Exercice**

Faites une implémentation récursive `fibostupide(n)` qui calculera stupidement (du point de vue du temps nécessaire pour le calcul) les valeurs de la suite pour les entiers positifs.

```
def fibostupide(n):
    """Fibonacci (stupidement).

    Exemples
    --------

    >>> fibostupide(5)
    5
    >>> fibostupide(10)
    55
    """
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibostupide(n - 1) + fibostupide(n - 2)
```

Vérifions si ça marche :

```
In [13]: for i in range(10):
    ...:     print(fibostupide(i))
    ...:
0
1
1
```

```
2
3
5
8
13
21
34
```

Bon, et au point de vue temps de calcul :

```
In [19]: %timeit -n 100 fibostupide(10)
100 loops, best of 3: 35.4 µs per loop

In [20]: %timeit -n 100 fibostupide(15)
100 loops, best of 3: 348 µs per loop

In [21]: %timeit -n 100 fibostupide(20)
100 loops, best of 3: 3.83 ms per loop

In [22]: %timeit -n 100 fibostupide(25)
100 loops, best of 3: 43.1 ms per loop
```

C'est lamentable ! on a bien travaillé… il semble que si on ajoute 5 à l'indice on multiplie par au moins 10 le temps de calcul. Inutile de dire que même les ordinateurs les plus puissants auraient du mal avec $F_{100}$ et certainement avec $F_{1000}$ !

### 1.3.2 fiborecursif(n)

---

**Exercice**

Faites une implémentation récursive `fiborecursif(n)` moins stupide que la précédente. On utilisera une routine récursive portant sur des *couples* $(F_{n-1}, F_n)$.

---

```python
def fiborecursif_a(n):
    if n == 0:
        return 1, 0
    fibs = fiborecursif_a(n - 1)
    return fibs[1], fibs[0] + fibs[1]
```

```python
def fiborecursif(n):
    """Fibonacci (implémentation récursive).

    Exemples
    --------

    >>> fiborecursif(5)
    5
    >>> fiborecursif(10)
    55
    >>> fiborecursif(100)
    354224848179261915075
    """
    return fiborecursif_a(n)[1]
```

Vérifions si ça marche :

```
In [33]: for i in range(10):
    ...:     print(fiborecursif(i))
    ...:
0
1
1
2
3
5
8
13
21
34
```

Et au niveau du temps de calcul :

```
In [34]: %timeit -n 100 fiborecursif(10)
100 loops, best of 3: 3.42 µs per loop

In [35]: %timeit -n 100 fiborecursif(15)
100 loops, best of 3: 5.05 µs per loop

In [36]: %timeit -n 100 fiborecursif(20)
100 loops, best of 3: 6.79 µs per loop

In [37]: %timeit -n 100 fiborecursif(25)
100 loops, best of 3: 8.35 µs per loop

In [38]: %timeit -n 100 fiborecursif(30)
100 loops, best of 3: 10.1 µs per loop

In [39]: %timeit -n 100 fiborecursif(35)
100 loops, best of 3: 12 µs per loop
```

On a l'impression que c'est linéaire maintenant. Attention, si c'est linéaire en $n$, c'est exponentiel en la taille de l'écriture décimale de $n$ ! mais bref, c'est mieux qu'avant. On peut en profiter :

```
In [40]: fiborecursif(1000)

  (plein de lignes supprimées)
  File "<ipython-input-32-43580ccd4821>", line 2, in fiborecursif_a
    if n==0:

RuntimeError: maximum recursion depth exceeded in comparison
```

ah zut, marche pas.

### 1.3.3 `fiboiteratif(n)`

**Exercice**

Refaites l'implémentation précédente mais sous forme itérative `fiboiteratif(n)`. On pourra

utiliser les possibilités d'affectations simultanées du langage Python.

```
def fiboiteratif(n):
    """Fibonacci (implémentation itérative).

    Exemples
    --------

    >>> fiboiteratif(5)
    5
    >>> fiboiteratif(10)
    55
    """
    if n == 0:
        return 0
    a, b = 0, 1          # F_0 et F_1
    while n > 1:
        a, b = b, a + b  # F_k, F_{k+1} -> F_{k+1}, F_{k+2}
        n -= 1           # n<- n-1 (fait localement à la procédure)
    return b
```

Vérifions si ça marche :

```
In [45]: for i in range(10):
    ...:     print(fiboiteratif(i), end=", ")
    ...: print(fiboiteratif(10))
    ...:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

Bien sûr, si l'objectif avait été d'imprimer une telle suite, il aurait fallu faire une procédure imprimant directement les nombres de Fibonacci déjà calculés, là on recommence à chaque fois depuis le début.

Comme on ne fait plus de récursion, on devrait pouvoir obtenir $F_{1000}$ (j'ai manuellement mis sur plusieurs lignes le résultat pour affichage sur cette page html) :

```
In [46]: fiboiteratif(1000)
Out[46]: 43466557686937456435688527675040625802564660517371780\
04024817290895365554179490518904038798400792551692959225930800\
32263477520968962323987332247116164299644090653318793829896960\
49928516003704476137795166849228875
```

Et pour les temps de calcul :

```
In [47]: %timeit -n 100 fiboiteratif(10)
100 loops, best of 3: 1.41 µs per loop

In [48]: %timeit -n 100 fiboiteratif(15)
100 loops, best of 3: 2.07 µs per loop

In [49]: %timeit -n 100 fiboiteratif(20)
100 loops, best of 3: 2.79 µs per loop

In [50]: %timeit -n 100 fiboiteratif(25)
100 loops, best of 3: 3.49 µs per loop
```

```
In [51]: %timeit -n 100 fiboiteratif(30)
100 loops, best of 3: 4.48 µs per loop

In [52]: %timeit -n 100 fiboiteratif(35)
100 loops, best of 3: 5.11 µs per loop

In [53]: %timeit -n 100 fiboiteratif(1000)
100 loops, best of 3: 164 µs per loop
```

On a gagné aussi en vitesse.

### 1.3.4 fiboiteratifavecfor(n)

**Exercice**

La solution proposée pour *fiboiteratif(n)* (page 18) utilise un `while` avec un compteur `n` qui est décrémenté à chaque exécution du corps de la boucle et dont la valeur initiale est celle passée en argument à la procédure. Il est plus « pythonesque » d'utiliser un objet itérateur `range` avec une boucle `for`. Le faire.

```
def fiboiteratifavecfor(n):
    """Fibonacci (implémentation avec boucle :keyword:`for`).

    Exemples
    --------

    >>> fiboiteratifavecfor(5)
    5
    >>> fiboiteratifavecfor(10)
    55
    >>> fiboiteratifavecfor(100)
    354224848179261915075
    """
    if n == 0:
        return 0
    a, b = 0, 1            # F_0 et F_1
    for i in range(1, n):
        a, b = b, a + b    # F_{i-1}, F_{i} devient F_{i}, F_{i+1}
    # la première valeur de i est 1 et en entrée b = F_1 = 1
    # la dernière valeur de i dans la boucle est n-1, donc
    # le dernier b en entrée sera F_{n-1} et en sortie F_n
    return b
```

Et c'est plus efficace :

```
In [89]: fiboiteratif(1000)==fiboiteratifavecfor(1000)
Out[89]: True

In [90]: %timeit fiboiteratif(1000)
10000 loops, best of 3: 163 µs per loop

In [91]: %timeit fiboiteratifavecfor(1000)
10000 loops, best of 3: 108 µs per loop
```

### 1.3.5 fiborapide(n)

Un peu de mathématiques :

Soit $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Alors pour tout $n$, on a $A^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$.

---

**Exercice**

Démontrer ce Théorème (par récurrence sur $n$ ; faites le aussi pour $n < 0$).

---

Nos idées sur l'exponentiation rapide sur les entiers peuvent s'appliquer ici aux matrices. On ne fait pas créer de `class`[9] mais plus « à la main » en manipulant uniquement des variables $a, b, c$ représentant des matrices symétriques $\begin{pmatrix} a & b \\ b & c \end{pmatrix}$.

---

**Note :** Dans l'algorithme ci-dessous, la relation $a = b + c$ est vérifiée à chaque étape car les matrices seront toutes parmi les $A^n$ ci-dessus.

---

---

**Exercice**

Analyser l'algorithme suivant et montrer qu'il calcule $F_n$ :

```
(note : les variables a, b, c vont correspondre aux entrées de la
 matrice A à la puissance n)

si n vaut 0 renvoyer 0
si n vaut 1 renvoyer 1
sinon poser a=1, b=1, c=0, d=1, e=0, f=1 (=matrice Identité)
répéter jusqu'à ce que n vaille 1:
    si n est impair f<- b*e + c*f, e<-b*d + c*e, d<- e + f (après !)
    n<-n//2
    c<-b**2 + c**2, b<-(a + c)*b (simultanément...), a<-b + c (après !)
renvoyer alors b*d + c*e
```

---

---

**Exercice**

Implémenter l'algorithme précédent sous la forme d'une fonction `fiborapide(n)`.

---

```python
def fiborapide(n):
    """Fibonacci (méthode type exponentiation rapide).

    Exemples
    --------

    >>> fiborapide(5)
    5
    >>> fiborapide(10)
    55
    >>> fiborapide(100)
```

9. https://docs.python.org/3/reference/compound_stmts.html#class

```
    35422484817926191507 5
    """
    if n == 0:
        return 0
    # if n==1:
    #    return 1
    a, b, c, d, e, f = 1, 1, 0, 1, 0, 1
    while n > 1:
        if n & 1:
            f, e = b * e + c * f, b * d + c * e
            d = e + f
        n = n >> 1
        b, c = (a + c) * b, b * b + c * c
        a = b + c
    return b * d + c * e
```

Vérifions si ça marche :

```
In [55]: for i in range(30):
    ...:     print(fiborapide(i), end=", ")
    ...: print(fiborapide(30))
    ...:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,\ # retours à la ligne ajoutés
233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946,\
17711, 28657, 46368, 75025, 121393, 196418, 317811,\
514229, 832040

In [56]: fiborapide(1000)==fiboiteratif(1000)
Out[56]: True
```

On peut remarquer que comme l'algorithme légèrement modifié donne non seulement $F_n$ mais aussi $F_{n+1}$, si on voulait par exemple calculer les $F_n$ pour $n = 10000, 10001, ..., 10100$, une bonne idée serait de l'utiliser pour obtenir d'abord la paire $F_{10000}, F_{10001}$ et de continuer ensuite simplement par des additions suivant la relation de récurrence de base définissant la suite de Fibonacci.

Et au niveau du temps ? ah c'est que j'ai changé d'ordinateur entretemps. Je dois donc re-évaluer aussi pour `fiboiteratif`. Voici ce que ça donne chez moi :

```
In [71]: %timeit -n 100 fiboiteratif(1000)
100 loops, best of 3: 169 µs per loop

In [72]: %timeit -n 100 fiborapide(1000)
100 loops, best of 3: 8.81 µs per loop

In [81]: %timeit -n 100 fiboiteratif(10000)
100 loops, best of 3: 3.31 ms per loop

In [82]: %timeit -n 100 fiborapide(10000)
100 loops, best of 3: 76.9 µs per loop
```

Il n'y a pas photo : pour $n = 1000$, 19 fois plus rapide, pour $n = 10000$, 43 fois plus rapide…

```
In [83]: %timeit -n 100 fiboiteratif(20000)
100 loops, best of 3: 8.92 ms per loop

In [84]: %timeit -n 100 fiborapide(20000)
100 loops, best of 3: 221 µs per loop
```

```
In [85]: 8920/221
Out[85]: 40.36199095022624

In [86]: %timeit -n 100 fiboiteratif(50000)
100 loops, best of 3: 39.1 ms per loop

In [87]: %timeit -n 100 fiborapide(50000)
100 loops, best of 3: 866 µs per loop

In [88]: 39100/866
Out[88]: 45.15011547344111
```

Bien sûr il y a peut-être aussi à ce niveau des détails liés à la gestion de la mémoire (intervention du ramasse-miettes, aka garbage collector, entre autres), mais cela incite néanmoins à poser la question suivante :

---

**Exercice**

On ne prend en compte que les opérations arithmétiques (additions et multiplications). En supposant que l'addition de deux nombres de $k$ et $l$ bits prend de l'ordre de $c_1 \max(k, l)$ secondes, avec $c_1$ une constante, et que la multiplication prend de l'ordre de $c_2 kl$ secondes, estimez à la louche le temps nécessaire pour évaluer $F_n$ par, respectivement `fiboiteratif(n)` et `fiborapide(n)` en fonction de $n$. L'algorithme rapide est-il vraiment, sous ces hypothèses, intrinsèquement plus rapide ?

---

```
désolé, à vous de travailler un peu, je ne peux pas tout faire.
```

---

**Indice :** Quel est le lien entre le logarithme en base 2 d'un entier et le nombre de bits de son écriture binaire ? Quelle est la formule exacte pour les nombres de Fibonacci en fonction du nombre d'or ? Montrer que le nombre de bits de $F_n$ en binaire est approximativement proportionnel à $n$.

---

## 1.4 Temps d'exécution de la multiplication en Python

Les entiers sont représentés intérieurement en binaire. Lorsqu'un entier est imprimé à l'écran ou dans un fichier en décimal, une conversion doit être faite. De même en entrée. Cette conversion, sur les entiers de dizaines de milliers de chiffres, est coûteuse. Elle prend à Python de l'ordre de $O(k^2)$ unités de temps, avec $k$ le nombre de chiffres (en décimal ou en binaire, cela revient au même).

En mathématiques la notation $O(..)$ est une relation de majoration avec des constantes non spécifiés ; par commodité je l'ai utilisée dans ce cas spécifique aussi avec le sens que $k^2$ est non seulement un majorant mais également un minorant du temps d'exécution (à des constantes multiplicatives près ; normalement on devrait aussi dire « pour $k$ suffisamment grand », mais comme $k$ est au moins 1, ça ne change rien, on peut toujours ajuster la constante soit vers le haut soit vers le bas suivant le sens de l'inégalité à rendre valable).

Si l'on fait le produit de deux entiers avec chacun environ $k$ bits, ou $k$ chiffres décimaux, par la méthode usuelle apprise à l'École, cela prend de l'ordre de $k^2$ opérations élémentaires. On va voir

que Python a un algorithme plus efficace : il opère, si j'en crois une source sur internet, en temps $O(k^{1.585})$ (vous trouverez le lien tout en bas de cette feuille).

### 1.4.1 Importation de modules

Notre objectif dans cette section est de vérifier cette affirmation. On va avoir besoin de plusieurs modules supplémentaires :

- `timeit` [10] pour sa fonction `timeit.timeit()` [11] qui permet de mesurer des temps d'exécutions,
- `random` [12] pour `random.randrange()` [13] qui engendre des nombres entiers pseudo-aléatoires dans un intervalle,
- matplotlib [14] pour créer des graphiques, ici avec axes logarithmiques,
- `math` [15] pour appliquer le logarithme `math.log()` [16] sur des données,
- et la librairie scientifique scipy [17], parce qu'on aura besoin de faire une régression linéaire sur ces logarithmes. Cela utilisera `scipy.stats.linregress()` [18] (on aurait pu aussi choisir numpy [19] et sa fonction `numpy.polyfit()` [20]).

```
import math
import matplotlib.pyplot as plt
import timeit
## import numpy
from scipy import stats
from random import randrange
```

### 1.4.2 `tempspourmultiplication(M, N, reps)`

Notre première procédure `tempspourmultiplication(M, N, reps)` accomplit les choses suivantes :

1. pour chaque entier `i` de `M` à `N` choisir au hasard deux nombres avec chacun `i` bits,
2. mesurer le temps `t` pris par Python pour évaluer leur produit, via `timeit.timeit()` [21] avec un nombre de répétitions égal à `reps`,
3. stocker `i` et `t` dans des listes. Puis, faire un graphique `matplotlib.pyplot.loglog()` [22] pour repérer visuellement une loi en puissance `t=c i^a`, autrement dit voir si les points en graphe loglog s'agencent plus ou moins sur une droite de coefficient directeur `a`,
4. appliquer la fonction `scipy.stats.linregress()` [23] sur les logarithmes de nos données précédentes pour obtenir une valeur pour `a`, et examiner si le coefficient de corrélation est raisonnablement proche de 1.

10. https://docs.python.org/3/library/timeit.html#module-timeit
11. https://docs.python.org/3/library/timeit.html#timeit.timeit
12. https://docs.python.org/3/library/random.html#module-random
13. https://docs.python.org/3/library/random.html#random.randrange
14. http://matplotlib.sourceforge.net/
15. https://docs.python.org/3/library/math.html#module-math
16. https://docs.python.org/3/library/math.html#math.log
17. http://docs.scipy.org/doc/scipy/reference/
18. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.linregress.html#scipy.stats.linregress
19. http://docs.scipy.org/doc/numpy/reference/
20. https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html#numpy.polyfit
21. https://docs.python.org/3/library/timeit.html#timeit.timeit
22. https://matplotlib.org/api/_as_gen/matplotlib.pyplot.loglog.html#matplotlib.pyplot.loglog
23. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.linregress.html#scipy.stats.linregress

Afin de rendre la procédure un peu plus performante, nous nous sommes donc arrangés pour ne pas avoir à calculer la longueur des écritures décimales :

1. d'une part la formule $1+\lfloor \log_{10} N \rfloor$ cesse rapidement de fonctionner car nos nombres avec des centaines de chiffres sont trop grands pour la conversion en type float nécessaire au log,
2. la méthode via `len(str(N))` nécessite une conversion en décimal qui, comme je l'ai dit, est coûteuse,
3. et tout cela est idiot car par construction on connaîtra la longueur en binaire, or la longueur en décimal lui est presque exactement proportionnelle. Et une constante de proportionnalité modifiant le $i$ dans une formule $ci^\alpha$ est absorbée dans le $c$, cela ne change rien au $\alpha$ que nous recherchons.

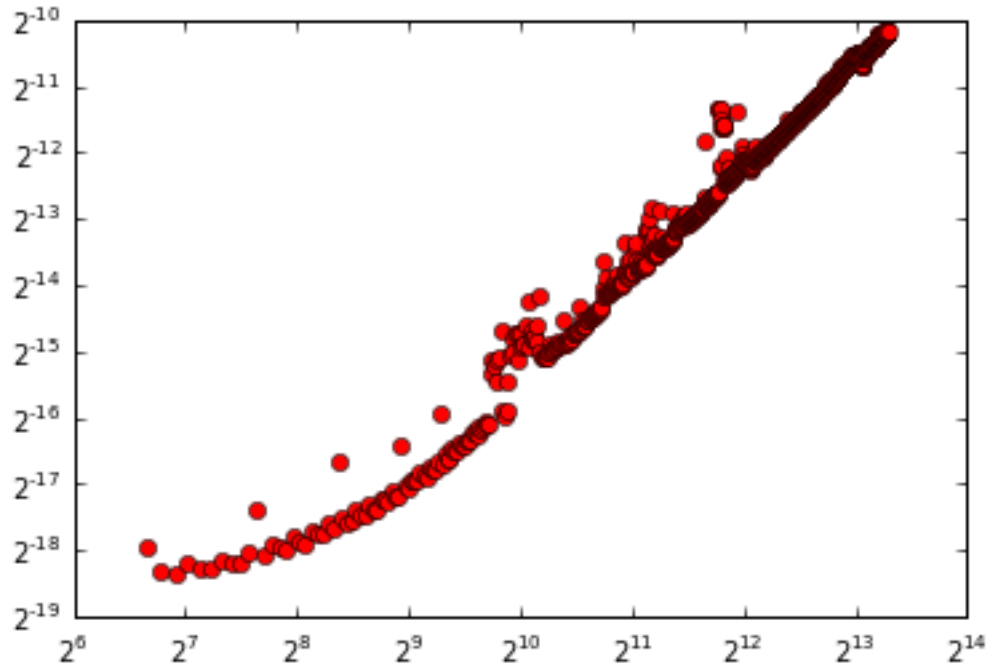
Voici une implémentation :

```
def tempspourmultiplication(M, N, reps=100):
    """Explore le temps de calcul pour les multiplications.

    Évalue pour chaque entier multiple de 10 de 10M à 10N combien de
    temps est pris par la multiplication de deux nombres aléatoires avec
    ce nombre de bits.
    """
    x = 2 ** (10 * M - 1)
    l = []  # on y stockera les longueurs
    t = []  # on y stockera les temps de calcul
    for i in range(M, N + 1):
        y = 2 * x              # on aura toujours y=2**(10*i)
        w = randrange(x, y)  # entier aléatoire avec exactement 10*i bits
        z = randrange(x, y)  # un autre
        # pour utiliser timeit

        def wrapper():
            return w * z

        # l.append(1+math.floor(math.log10(w)))
        #    (restreint trop les M, N possibles)
        # l.append(len(str(w)))
        #    (est coûteux et inutile)
        l.append(10 * i)         # on stocke le nombre de bits pour ce test
        # évaluation du temps de calcul :
        t1 = timeit.timeit(wrapper, number=reps)
        t.append(t1)             # on le stocke
        x = 1024 * x               # pour boucler, 1024=2**10
    # maintenant on fait un plot avec matplotlib
    # J'ai configuré mon Spyder, dans Préférences:iPython:Graphiques,
    # pour faire affichage à l'intérieur de la console IPython
    plt.loglog(l, t, 'ro', basex=2, basey=2)  # ro = red dot
    # plt.show(p)
    #
    # on pourrait faire ceci pour obtenir les coefficients de
    # la droite de régression :
    # print(numpy.polyfit(numpy.log(l), numpy.log(t), 1))
    #
    # mais on va faire avec stats.linregress de scipy :
    # d'abord appliquons le log sur toutes nos données
    llog = list(map(math.log, l))  # il y aurait aussi numpy.log(l) possible
    tlog = list(map(math.log, t))
```

```
    # régression linéaire, voir la doc de scipy.stats.linregress
    # http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.linregress.
↪html
    slope, intercept, r_value, p_value, std_err = stats.linregress(llog, tlog)
    # slope est le coefficient directeur, r_value le coefficient de corrélation
    print(slope, r_value, std_err)
```

Voici ce que ça donne :

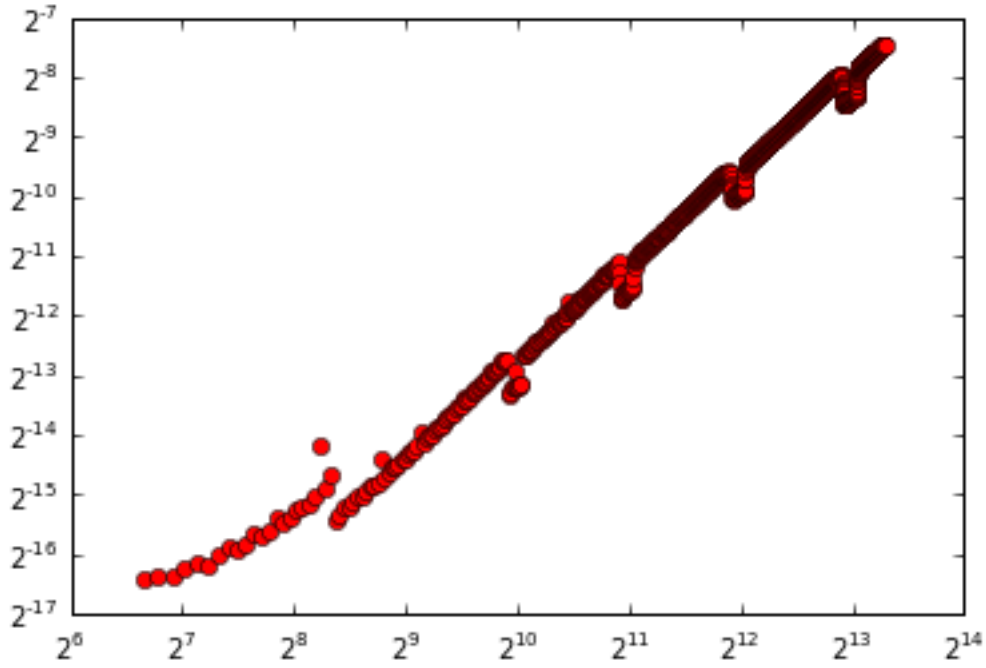
```
In [4]: tempspourmultiplication(10, 1000, 10)
1.49359699075 0.992311461885 0.00592362802229
```




L'impression visuelle est que c'est seulement à partir d'environ $2^9 = 512$ chiffres en binaire, soit environ 155 (disons 200) chiffres en décimal que l'on tombe dans une loi de puissance à peu près respectée (ou encore, on a d'abord un régime pour les « petits » nombres de chiffres, et ensuite un autre à partir d'un certain seuil).

Sur un autre ordinateur, j'ai obtenu (plusieurs fois consécutives) un graphique du type suivant :

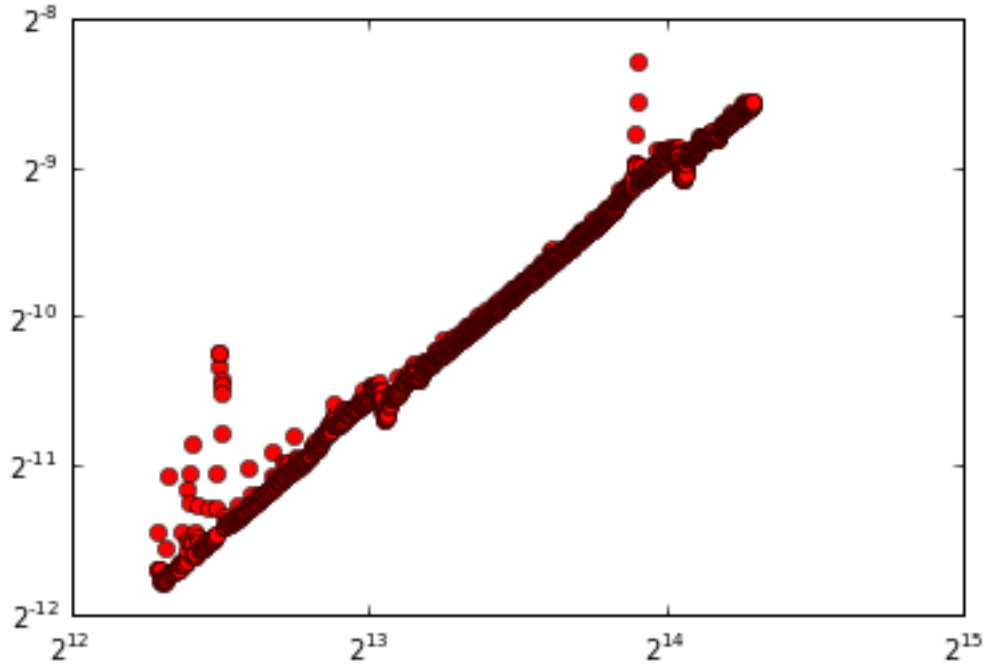
```
In [5]: tempspourmultiplication(10, 1000, 10)
1.5381069965 0.99478525524 0.00501445415698
```




On a l'impression qu'il se passe des choses spéciales à chaque fois que le nombre de bits devient égal à une puissance de 2 (à partir de $2^{10}$) ce qui suggère fortement que Python fait la multiplication par un algorithme qui commence par diviser la représentation binaire en deux morceaux. Il est possible que la fonction de mesure du temps d'exécution soit plus fine sur cet ordinateur et permette de mieux déceler ce phénomène lié au franchissement des nombres de bits égaux à une puissance de deux.

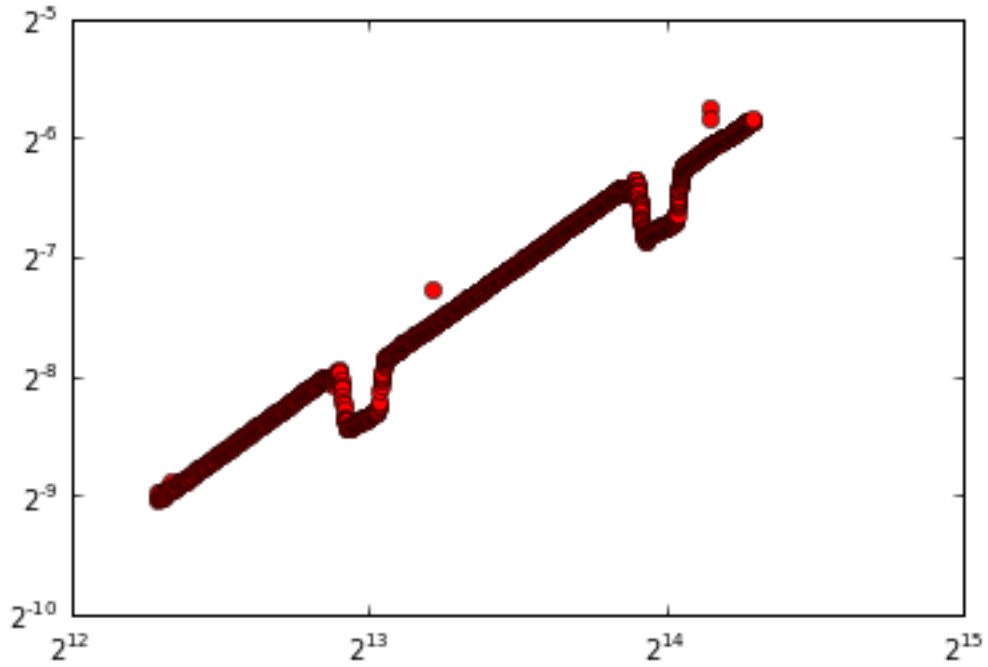
Regardons entre 5000 et 20000 bits binaires (donc grosso modo entre 1500 et 6000 chiffres décimaux). Il faut s'armer d'un peu de patience :

```
In [6]: tempspourmultiplication(500, 2000, 10)
1.58398183737 0.994150914646 0.0044444707783
```



On distingue vaguement sur ce graphique, fait sur le premier ordinateur, le phénomène particulier des puissances de deux, et c'est flagrant sur l'autre ordinateur. J'y ai obtenu trois fois de suite un graphique spectaculaire comme celui qui suit, avec un « trou » pour $i$ (qui est un multiple de 10 entre 5000 et 20000) aux alentours des puissances de deux 8192 et 16384) :

```
In [7]: tempspourmultiplication(500, 2000, 10)
1.55105593679 0.982751936581 0.00753853745876
```




Ma suspicion que ce deuxième ordinateur mesure plus finement les temps d'exécution se renforce ! Mais il est aussi possible que d'autres différences interviennent, ce ne sont pas les mêmes processeurs, donc pas les mêmes compilateurs `C` qui produisent le code exécuté par Python pour son exponentiation d'entiers.

Mais bref, ce qui nous importe c'est l'aspect linéaire du graphique.

### I.4.3 tempspourmultiplicationII(M, N, reps)

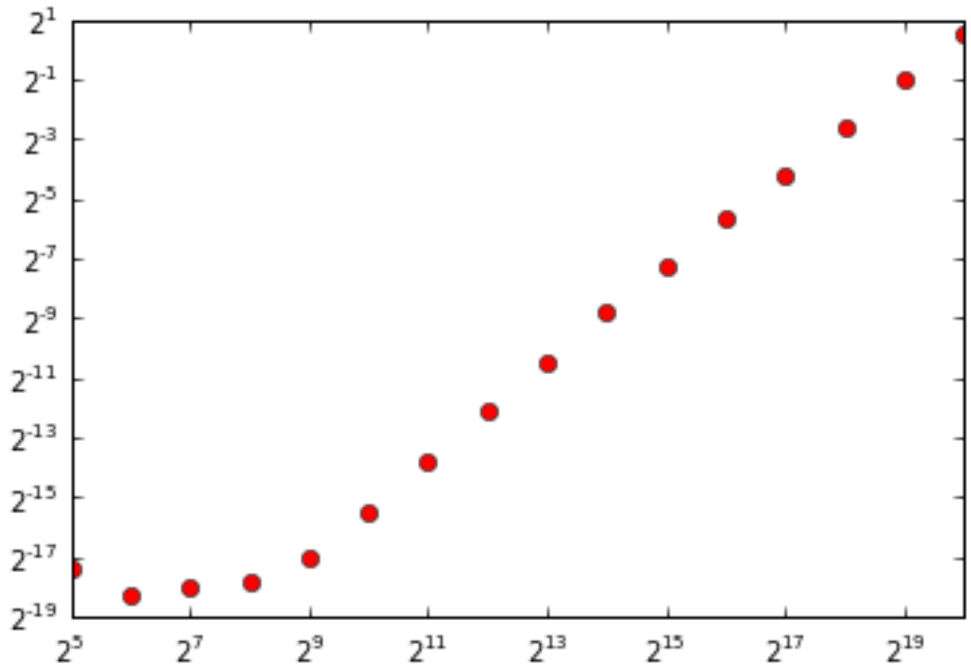
En réalité, Python peut manipuler des nombres beaucoup plus grands. Pour explorer cela j'ai donc fait une deuxième procédure qui carrément double le nombre de bits à chaque itération, afin de regarder ce qui se passe pour les nombres comportant des dizaines voire des centaines de milliers de chiffres :

```
def tempspourmultiplicationII(M, N, reps=10):
    """Explore le temps de calcul pour les multiplications.

    Évalue pour chaque entier de la forme 2 ** i avec i entre M et N
    combien de temps est pris par la multiplication de deux nombres
    aléatoires avec ce nombre de bits.

    Fait un graphique loglog et une régression linéaire sur les données.
    """
    j = 2 ** M
    x = 2 ** (j - 1)  # au début, x a 2**M bits
```

```
    l = []
    t = []
    for i in range(M, N + 1):
        y = 2 * x
        w = randrange(x, y)  # a exactement j = 2**i bits
        z = randrange(x, y)  # idem

        def wrapper():
            return w * z

        l.append(j)  # nos w et z ont j bits
        t1 = timeit.timeit(wrapper, number=reps)
        t.append(t1)
        x = x ** 2   # pour boucler
        j = 2 * j    # deux fois plus de bits
    plt.loglog(l, t, 'ro', basex=2, basey=2)  # ro = red dot
    llog = list(map(math.log, l))
    tlog = list(map(math.log, t))
    slope, intercept, r_value, p_value, std_err = stats.linregress(llog, tlog)
    print(slope, r_value, std_err)
```

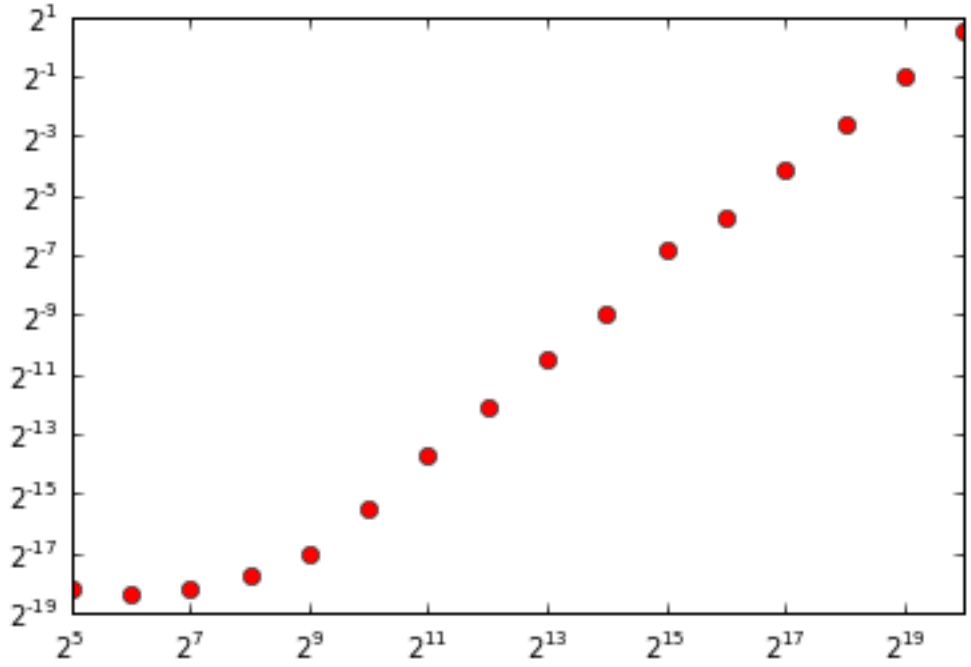
Voici ce que cela donne (il faut attendre un peu que la procédure aboutisse ; si un de vos camarades vous dit attendre depuis plusieurs minutes, essayez d'abord un 15 par exemple à la place de 20) :

```
In [8]: tempspourmultiplicationII(5, 20)
1.35657600862 0.981790253246 0.0701523453644
```




Une deuxième fois :

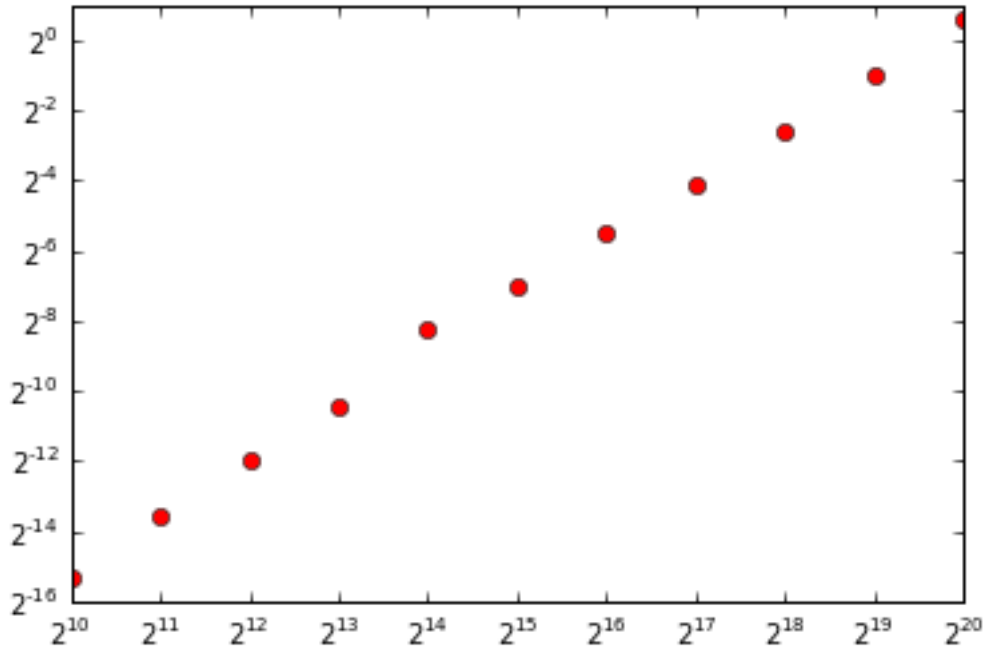
```
In [9]: tempspourmultiplicationII(5, 20)
1.38055880444 0.985905166856 0.0626130661904
```




Sur l'autre ordinateur c'est à peu près pareil, mais il n'y a pas le phénomène que pour $32 = 2^5$ bits ça prend apparemment peut-être plus de temps que pour 64 voire 128 bits. Les deux ordinateurs ont un processeur `64bits`, mais seul le premier a un système d'exploitation `64bits`.

Pour regarder mieux la partie linéaire, on va se restreindre à un nombre de bits au moins $2^{10} = 1024$ (environ 300 chiffres en décimal), et aller encore jusqu'à $2^{20} = 1048576$ bits soit des entiers qui auraient environ 315000 chiffres en écriture décimale. Voici ce que j'obtiens :

```
In [10]: tempspourmultiplicationII(10, 20)
1.58032690609 0.999111859989 0.0222162539383
```




Pour être honnête, j'ai observé d'assez fortes variations en ce qui concerne le coefficient directeur, qui va dans mes essais de moins de 1.5 à plus de 1.6 ; mesurer des temps d'exécution intrin-

sèques sur un ordinateur multi-tâches n'est pas une chose très aisée. De plus on ne teste ici que les nombres de bits égaux à des puissances de deux, et ils semblent constituer les cas les plus favorables.

### 1.4.4 Conclusion

Mais, en conclusion, on a tout de même confirmé que Python multiplie les grands entiers avec un algorithme de multiplication rapide. D'après l'une des référence ci-dessous, Python implémente la méthode de Karatsuba [24], historiquement la première « multiplication rapide ».

http://fr.wikipedia.org/wiki/Algorithme_de_Karatsuba

Donc notre routine *fiborapide(n)* (page 21) est en effet intrinsèquement plus rapide que la routine *fiboiteratif(n)* (page 18) qui ne fait que des additions. Mais les deux sont de toute façon exponentielles en le nombre de bits de $n$ (et par ailleurs quoi qu'en fasse à un moment il faudra bien imprimer le nombre $F_n$ or celui-ci a un nombre de bits proportionnel à $n$, donc exponentiel en le nombre de bits de $n$ lui-même).

On obtient quelques informations sur la multiplication en Python sur le site suivant de la librairie DecInt [25] :

http://home.comcast.net/~casevh/

Voir aussi à ce sujet : http://stackoverflow.com/a/1845764/4184837

Cette librairie fait pour de très grands entiers la multiplication en un temps de calcul en $O(k * \log(k))$ et de plus sa représentation interne permet la conversion vers le décimal (ce qui est nécessaire pour affichage à l'écran ou dans un fichier) en $O(k)$ et non plus en $O(k^2)$ comme c'est le cas en natif dans le langage Python.

Pour les gens intéressés par les librairies de calcul avec des nombres arbitrairement grands, je signale aussi le site du project gmpy [26] et celui de sa documentation [27].

---

*Date de dernière modification* : `24-10-2017 à 14:58:00.`

24. http://fr.wikipedia.org/wiki/Anatoli_Karatsouba
25. http://home.comcast.net/~casevh/
26. https://code.google.com/p/gmpy/
27. https://gmpy2.readthedocs.org/en/latest/

# 2

# Feuille de travaux pratiques 2

*Date de dernière modification* : 03-10-2017 à 15:56:38.



## 2.1 Avant d'aller plus loin

Vous devez maintenant avoir acquis les notions de base du langage Python3, par exemple en ayant suivi certains *des liens précédemment indiqués* (page v) dans la fiche d'introduction à ce module. En particulier, je ne peux pas lire et assimiler à votre place le Tutoriel officiel [28].

28. https://docs.python.org/fr/3/tutorial/index.html

## 2.2 Trier

Nous allons travailler avec des objets de type `list` [29]. Dans la liste des méthodes associées [30] du tutoriel on trouve `list.sort()` [31].

Voici un exemple d'utilisation, les éléments de la liste sont ici des chaînes de caractères :

```
In [1]: L = ['Hashem', 'Atmine', 'Danai', 'Florent', 'Ludovic', 'Kevin',
   ...:      'Mohamad', 'Justine', 'Olivier', 'Victor', 'Adèle',
   ...:      'Vincent', 'Martin', 'Nagham', 'Benoît', 'Ana Isabel',
   ...:      'Nils', 'Maureen', 'Anthony', 'Salim', 'Corentin',
   ...:      'Gautier', 'Antoine', 'Marine', 'Marie', 'Sami', 'Lisa',
   ...:      'Ali']
   ...:

In [2]: type(L), len(L)
Out[2]: (list, 28)

In [3]: M = L[:] # on fait une copie de la liste.

In [4]: M[1], M[5]
Out[4]: ('Atmine', 'Kevin')

In [5]: M.sort() # attention, modifie M elle-même.

In [6]: M
Out[6]:
['Adèle',
 'Ali',
 'Ana Isabel',
 'Anthony',
 'Antoine',
 'Atmine',
 'Benoît',
 'Corentin',
 'Danai',
 'Florent',
 'Gautier',
 'Hashem',
 'Justine',
 'Kevin',
 'Lisa',
 'Ludovic',
 'Marie',
 'Marine',
 'Martin',
 'Maureen',
 'Mohamad',
 'Nagham',
 'Nils',
 'Olivier',
 'Salim',
 'Sami',
 'Victor',
```

29. https://docs.python.org/3/library/stdtypes.html#list
30. https://docs.python.org/3/tutorial/datastructures.html#more-on-lists
31. https://docs.python.org/3/library/stdtypes.html#list.sort

```
 'Vincent']

In [7]: L.index('Benoît'), M.index('Benoît')
Out[7]: (14, 6)

In [8]: M.index('Jean')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-8-fdfda10794cd> in <module>()
----> 1 M.index('Jean')

ValueError: 'Jean' is not in list
```

J'en profite pour faire un aparté sur `try`. Il permet ici de faire une petite fonction qui absorbera silencieusement l'exception qui est déclenchée par la méthode `index` lorsque l'élément ne se trouve pas dans la liste.

```
In [9]: def indicedansliste(liste, candidat):
   ...:     try:
   ...:         i=liste.index(candidat)
   ...:     except ValueError as err:
   ...:         print(err) # commenter cette ligne pour silence.
   ...:         return None
   ...:     else:
   ...:         return i
   ...:

In [10]: indicedansliste(M, 'Jean')
'Jean' is not in list

In [11]: indicedansliste(L, 'Sami'), indicedansliste(M, 'Sami')
Out[11]: (25, 25)
```

Le dernier résultat est une coïncidence !

Nous allons coder nos propres (maladroites, mais on assume) procédures de tri pour des listes, qui n'utiliseront pas la méthode `list.sort()` [32].

### 2.2.1 `estordonnee(liste)`

**Exercice**

Faire une routine `estordonnee(liste)` qui renvoie `True` si la liste est ordonnée de manière croissante, et `False` sinon. On la supposera donc composée uniquement d'objets deux à deux comparables par <. Pour une liste vide, la routine produira `True`.

On implémentera l'algorithme suivant :

```
On teste si liste[i] <= liste[i+1] pour 0 <= i < i + 1 < longueur
Bien sûr on s'arrête dès que ça foire.
```

32. https://docs.python.org/3/library/stdtypes.html#list.sort

```python
def estordonnee(liste):
    """True si la liste est ordonnée, False sinon.

    >>> estordonnee([1, 2, 3])
    True
    >>> estordonnee(['a', 'f', 'z'])
    True
    >>> estordonnee([1, 2, 5, 4])
    False
    """
    for i in range(len(liste)-1):
        if liste[i + 1] < liste[i]:
            return False
    return True
```

### 2.2.2 indicedumin(L)

**Exercice**

Faire une fonction `indicedumin(L)` qui renvoie l'indice de l'élément minimal d'une liste `L`, Si plusieurs indices donnent un élément minimal, renvoyer le premier d'entre eux.

On ne vérifiera pas que la liste `L` est non vide.

Outils suggérés : `len`, boucle `for`, itérateur `range`, expression conditionnelle `if`, `return`. L'idée sera de parcourir la liste avec un indice croissant en mettant à jour à chaque étape l'indice ayant donné jusqu'à présent la plus petite valeur.

```python
def indicedumin(liste):
    """Renvoie l'indice du premier élément minimal.

    Présuppose que liste a au moins un élément.

    >>> indicedumin([2, 5, 1, 4])
    2
    """
    longueur = len(liste)
    # si on voulait gérer aussi liste vide :
    # if l == 0:
    #    return None
    # elif
    if longueur == 1:
        return 0
    vmin = liste[0]  # valeur minimale (candidat)
    imin = 0         # indice de la valeur minimale
    for j in range(1, longueur):
        if liste[j] < vmin:
            imin = j
            vmin = liste[j]
    return imin
```

### 2.2.3 triparselection(L)

**Exercice**

Faire une fonction `triparselection(L)` qui renvoie une nouvelle liste, égale à `L` triée par ordre ascendant. L'algorithme implémenté sera le suivant :

```
Si L est vide ou n'a qu'un seul élement, (presque) rien à faire

Si L a au moins deux éléments, déterminer l'indice d'un élément
   minimal, supprimer l'élément correspondant par L.pop(indice) et
   le stocker dans la nouvelle liste.

Itérer jusqu'à ce que L n'ait plus d'éléments.
```

Attention on demande que la liste d'origine ne soit pas modifiée.

Vous aurez probablement besoin des opérations `pop()` et `append()` (voir les opérations sur les objets modifiables [33] [34]) ainsi que de la fonction *indicedumin(L)* (page 36).

```python
def triparselection(liste):
    """Trie une liste par ordre ascendant.

    Ne modifie pas la liste initiale.

    Opère en identifiant un élément minimal, puis
    en le retirant, puis en itérant.

    >>> triparselection([6, 3, 10, 1, 8, 5])
    [1, 3, 5, 6, 8, 10]
    """
    L = liste[:]      # copie la liste (copie « superficielle »)
    longueur = len(L)
    if longueur < 2:  # liste vide ou singleton
        return L
    M = []            # la future liste triée
    # on pourrait faire avec len(L) > 1
    # mais c'est probablement un tout petit peu
    # plus rapide de tenir à jour son propre compteur
    while longueur > 1:
        imin = indicedumin(L)
        # le pop supprime de l'ancien,
        x = L.pop(imin)
        # et on ajoute à la liste en construction
        M.append(x)
        # décrémenter longueur avant de boucler
        longueur -= 1
    # ne reste plus que le dernier élément :
    M.append(L[0])
    return M
```

On va maintenant procéder à l'humiliation habituelle de vérifier le temps d'exécution. On teste

33. https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types
34. ce lien (le mercredi 28 janvier 2015 à 11:03:27) dit que la syntaxe est `s.pop([i])` mais cela semble être une coquille et la syntaxe qui fontionne est `s.pop(i)`.

sur la liste `N=[1000, 999, 998,..., 1]`:

```
In [15]: N=list(range(1000, 0, -1))

In [16]: len(N)
Out[16]: 1000

In [17]: %timeit -n 10 triparselection(N)
10 loops, best of 3: 70.7 ms per loop

In [17]: %timeit -n 10 sorted(N)
10 loops, best of 3: 23.6 µs per loop
```

On est 3000 fois plus lent. On a utilisé `sorted()` [35] pour ne pas modifier la liste `N`, contrairement à ce que ferait `N.sort()`.

**Exercice**

Combien de comparaisons deux-à-deux sont-elles faites par l'algorithme précédent ?

### 2.2.4 `triparinsertion(L)`

**Exercice**

Faire une fonction `triparinsertion(L)` qui suivra cet algorithme :

```
Si L est vide ou n'a qu'un élément, (presque) rien à faire

Sinon on prend les éléments les uns après les autres, et on insère
le petit nouveau au bon endroit parmi les précédents, déjà triés.
```

On aura donc une liste en construction `M` et on pourra utiliser sa méthode `M.insert(position, objet)` afin d'insérer au bon endroit un nouvel élément.

On parcourera la liste originelle par une première boucle `for`, et on utilisera une boucle secondaire `for` pour déterminer l'endroit où insérer le nouvel élément dans `M`. Après cette insertion on pourra utiliser un `break` pour revenir à la boucle `for` primaire. Voir break and continue statements [36].

```
def triparinsertion(liste):
    """Trie une liste. (renvoie une nouvelle liste)

    Procède en insérant un nouvel élément au bon
    endroit parmi ceux préalablement triés.

    Ne modifie pas la liste initiale.

    >>> triparinsertion([6, 3, 10, 1, 8, 5])
    [1, 3, 5, 6, 8, 10]
    """
```

35. https://docs.python.org/3/library/functions.html#sorted
36. https://docs.python.org/3/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops

```
    longueur = len(liste)
    if longueur < 2:
        M = liste[:]  # on renvoie une copie
        return M
    M = [liste[0]]
    # on va lui ajouter un par un, au bon endroit
    # les éléments de la liste originale.
    for i in range(1, longueur):
        x = liste[i]
        for j in range(0, i):
            if x < M[j]:
                M.insert(j, x)
                break
        else:
            M.append(x)
    return M
```

Mais comparons l'efficacité, d'abord sur notre liste originelle avec des prénoms [37] :

```
In [117]: %timeit -n 1000 triparselection(L)
1000 loops, best of 3: 42.4 µs per loop

In [118]: %timeit -n 1000 triparinsertion(L)
1000 loops, best of 3: 25.5 µs per loop
```

En effet c'est plus rapide. Regardons avec une liste aléatoire de 100 nombres :

```
In [123]: import random

In [124]: U = [random.randrange(1, 10000) for i in range(100)]

In [125]: %timeit -n 100 triparselection(U)
100 loops, best of 3: 482 µs per loop

In [126]: %timeit -n 100 triparinsertion(U)
100 loops, best of 3: 250 µs per loop

In [127]: %timeit -n 100 sorted(U)
100 loops, best of 3: 14.9 µs per loop
```

On a sur cette taille de liste, qui reste modeste, gagné un facteur 2 par rapport à la méthode de tri par sélection. Les spécificités de ce que sont les listes en Python jouent aussi, par exemple ici par rapport à *triparselection(L)* (page 37), on a l'avantage d'éviter de faire des `L.pop(indice)` qui sont probablement coûteux.

Essayons sur la liste des 1000 premiers entiers à l'envers :

```
In [129]: N=list(range(1000, 0, -1))

In [130]: %timeit -n 10 triparselection(N)
10 loops, best of 3: 71 ms per loop

In [131]: %timeit -n 100 triparinsertion(N)
100 loops, best of 3: 1.14 ms per loop
```

37. partout dans cette feuille, les temps ont été évalués avec une liste `L` comportant alors seulement 20 prénoms, je ne les ai pas mis à jour pour cette année.

```
In [132]: %timeit -n 1000 sorted(N)
1000 loops, best of 3: 22.4 µs per loop
```

Au lieu d'être 3000 fois plus lent on n'est plus que 50 fois plus lent, c'est une belle amélioration. Mais c'est de la triche car si vous réfléchissez, vous verrez que la liste `N` est très favorable à `triparinsertion`.

### 2.2.5 `trirapide(L)`

**Exercice**

Faire une fonction `trirapide(L)` qui produira une nouvelle liste, triée, par ordre ascendant, en procédant de manière récursive :

```
Si L est vide ou n'a qu'un élément, on renvoie (une copie de) L.

Sinon on choisit un élément de L, par exemple le premier, ou le
dernier, ou un élément d'indice au milieu, ou encore un élément
pris au hasard, que l'on appelle le pivot. On notera ipivot
l'indice de cet élément et vpivot sa valeur.

On répartit les autres éléments en deux listes,
    Lmoins= ceux < vpivot,
    Lplus = ceux >= vpivot.

Il suffit alors d'appeler de manière récursive la fonction de tri
sur ces deux listes :

         tri(L)=tri(Lmoins)+[pivot]+tri(Lplus)

(en effet la concaténation de listes peut se faire par l'opérateur +)
```

Le choix du pivot a un impact sur l'efficacité de l'algorithme surtout dans les pires cas (comme ceux d'une liste déjà triée ou comportant de nombreuses répétitions). Pour être spécifique vous prendrez le pivot comme étant l'élément d'indice `len(liste)//2`.

```python
def trirapide(liste):
    """Trie par algorithme de type ``QuickSort``.

    Ne modifie pas liste initiale.

    Implémenté de manière récursive. Choisit
    son pivot avec un indice au milieu.

    >>> trirapide([6, 3, 10, 1, 8, 5])
    [1, 3, 5, 6, 8, 10]
    """
    L = liste[:]
    longueur = len(L)
    if longueur < 2:
        return L
    ipivot = longueur // 2
```

```
    # ipivot= 0
    vpivot = L[ipivot]
    Lmoins = []
    Lplus = []
    for i in range(0, ipivot):
        if L[i] < vpivot:
            Lmoins.append(L[i])
        else:
            Lplus.append(L[i])
    # la raison pour deux boucles for est de ne pas
    # avoir à faire un if i == ipivot qui est nécessaire
    # pour éviter de mettre le pivot aussi dans Lplus.
    for i in range(ipivot + 1, longueur):
        if L[i] < vpivot:
            Lmoins.append(L[i])
        else:
            Lplus.append(L[i])
    return trirapide(Lmoins) + [vpivot] + trirapide(Lplus)
```

Voyons si on a gagné en efficacité, soit sur la liste `L` des prénoms, ou la liste `N` des 1000 entiers en ordre décroissant, ou la liste `U` des cent entiers aléatoires.

```
In [151]: %timeit -n 100 triparinsertion(L)
100 loops, best of 3: 27.5 µs per loop

In [152]: %timeit -n 100 triparinsertion(U)
100 loops, best of 3: 248 µs per loop

In [153]: %timeit -n 100 triparinsertion(N)
100 loops, best of 3: 1.14 ms per loop

In [154]: %timeit -n 100 trirapide(L)
100 loops, best of 3: 51.2 µs per loop

In [155]: %timeit -n 100 trirapide(U)
100 loops, best of 3: 335 µs per loop

In [156]: %timeit -n 100 trirapide(N)
100 loops, best of 3: 3.12 ms per loop
```

On est plus lent dans les trois cas ! c'est peut-être causé par les copies de liste faites par notre implémentation du tri rapide (néanmoins il ne s'agit que de copies « shallow », ce ne sont pas les objets de chaque liste qui sont copiés, uniquement des références aux objets). Le calcul des `longueur // 2` a aussi un coût ; mais si on prenait `ipivot=0` alors dans le cas de `N` cela engendrerait une erreur :

```
RuntimeError: maximum recursion depth exceeded in comparison
```

Il est vrai que de toute façon le cas de `N` est spécial, il est particulièrement favorable à `triparinsertion`, mais particulièrement défavorable à l'algorithme récursif si le pivot est toujours choisi en début ou toujours en fin de liste.

Pour une analyse du temps d'exécution il faudrait bien sûr prendre en considération la nature exacte de l'implémentation en Python des objets manipulés.

**quelques informations :**

— https://wiki.python.org/moin/TimeComplexity

On va comparer sur une liste plus grande `V` de 1000 entiers :

```
In [51]: V = [random.randrange(1, 10000) for i in range(1000)]

In [52]: %timeit -n 10 triparinsertion(V)
10 loops, best of 3: 19.1 ms per loop

In [53]: %timeit -n 10 trirapide(V)
10 loops, best of 3: 4.14 ms per loop
```

Ah tout de même. Et avec une liste `W` de 5000 entiers (attention c'est lent pour le premier, faites avec `-n 1` peut-être)

```
In [54]: W = [random.randrange(1, 100000) for i in range(5000)]

In [55]: %timeit -n 10 triparinsertion(W)
10 loops, best of 3: 505 ms per loop

In [56]: %timeit -n 10 trirapide(W)
10 loops, best of 3: 24.7 ms per loop
```

Pour 1000 entiers `trirapide` était 5 fois plus rapide, pour 5000 entiers il est 20 fois plus rapide. Mais il est à son tour près de 16 fois plus lent que la routine native (qui bien sûr n'a pas tout le surcoût du langage interprété) :

```
In [57]: %timeit -n 10 sorted(W)
10 loops, best of 3: 1.59 ms per loop
```

### 2.2.6 `trirapidesurplace(debut, fin, liste)`

**Exercice**

Faire une fonction `trirapidebis(L)` qui cherchera à utiliser moins de mémoire que `trirapide(L)`, en évitant de faire des copies de listes pendant la récursion (je rappelle cependant que les copies par `L[:]` sont « superficielles »).

La récursion sera réalisée par une sous-routine `trirapidesurplace(debut, fin, liste)` qui ne s'occupera que des indices de `debut` à `fin - 1` et modifiera la liste par l'algorithme suivant :

```
Prendre le pivot au milieu, l'échanger avec le dernier,

Examiner en débutant avec l'avant-dernier si < pivot,
si c'est le cas échanger cette valeur avec le premier endroit
à droite des valeurs < pivot déjà trouvées,

Itérer jusqu'à ce que tous les éléments aient été examinés,
puis finalement remettre le pivot entre les deux groupes
ainsi constitués.
```

```
Faire une récursion sur les deux groupes ainsi constitués
à gauche et à droite du pivot.
```

Remarque: il y aura exactement `fin - debut - 1` comparaisons, donc plutôt qu'un `while` on gagnera peut-être du temps avec une boucle `for`.

```python
def trirapidesurplace(debut, fin, liste):
    """Trie par algorithme ``QuickSort`` une partie d'une liste.

    attention : agit « en place ». Renvoie « None ».
    La liste passée en argument est modifiée par des échanges d'éléments.

    Le pivot est pris au milieu.

    >>> L = [6, 3, 10, 1, 8, 5]
    >>> trirapidesurplace(1, 5, L)
    >>> L
    [6, 1, 3, 8, 10, 5]
    """
    # attention, l'indice du dernier élément est fin-1 pas fin
    longueur = fin - debut
    if longueur < 2:
        return None  # rien à faire
    ipivot = debut + (longueur // 2)
    vpivot = liste[ipivot]
    #
    # PREMIÈRE ÉTAPE : PARTITION
    #
    # On commence par échanger le pivot avec la dernière position.
    dernier = fin - 1
    liste[ipivot] = liste[dernier]
    liste[dernier] = vpivot
    #
    # Ensuite on aura un indice i qui sera celui pour insérer une
    # nouvelle valeur < pivot (donc sa valeur finale sera là où remettre
    # le pivot) et un indice j qui parcourera tous les éléments de la
    # droite vers la gauche. On s'arrêtera lorsque j = i - 1
    #
    i = debut
    j = dernier - 1  # au départ j est au moins égal à i
    for k in range(debut, dernier):
        # en effet on sait exactement combien d'étapes il y aura,
        #  (à savoir longueur - 1 = fin - debut - 1 =dernier - debut)
        # c'est donc probablement plus rapide que faire while j >= i :
        x = liste[j]
        if x < vpivot:
            liste[j] = liste[i]  # <-- cette valeur n'a pas encore été examinée
            liste[i] = x
            i += 1
        else:
            j -= 1
    # on remet le pivot entre les deux groupes :
    liste[dernier] = liste[i]
    liste[i] = vpivot
    #
    # DEUXIÈME ÉTAPE : RÉCURSION
```

```
    #
    trirapidesurplace(debut, i, liste)
    trirapidesurplace(i + 1, fin, liste)
    return None
```

```
def trirapidebis(L):
    longueur = len(L)
    M = L[:]
    if longueur >= 2:
        trirapidesurplace(0, longueur, M)
    return M
```

Voyons ce que ça donne :

```
In [58]: %timeit -n 100 trirapide(L)
100 loops, best of 3: 53.6 µs per loop

In [59]: %timeit -n 100 trirapide(U)
100 loops, best of 3: 344 µs per loop

In [60]: %timeit -n 100 trirapide(V)
100 loops, best of 3: 4.13 ms per loop

In [61]: %timeit -n 100 trirapidebis(L)
100 loops, best of 3: 31.4 µs per loop

In [62]: %timeit -n 100 trirapidebis(U)
100 loops, best of 3: 238 µs per loop

In [63]: %timeit -n 100 trirapidebis(V)
100 loops, best of 3: 3.5 ms per loop

In [64]: %timeit -n 100 triparinsertion(L)
100 loops, best of 3: 27.1 µs per loop

In [65]: %timeit -n 100 triparinsertion(U)
100 loops, best of 3: 247 µs per loop
```

C'est pas mal, notre nouvelle implémentation `trirapidebis` est seulement un peu plus lente que `triparinsertion` sur la liste `L` avec 21 éléments (en fait seulement 20 éléments lorsque `%timeit` a été utilisé plus haut) et elle plus rapide qu'elle sur `U` qui a 100 éléments. Voyons finalement comment se comportent nos trois routines sur la liste avec 5000 entrées :

```
In [65]: %timeit -n 10 triparinsertion(W)
10 loops, best of 3: 505 ms per loop

In [66]: %timeit -n 10 trirapide(W)
10 loops, best of 3: 24.2 ms per loop

In [67]: %timeit -n 10 trirapidebis(W)
10 loops, best of 3: 20.9 ms per loop
```

À propos il faudrait peut-être aussi se rassurer en utilisant *estordonnee(liste)* (page 35) :

```
In [134]: estordonnee(trirapidebis(W))
Out[134]: True
```

```
In [135]: estordonnee(trirapide(W))
Out[135]: True

In [136]: estordonnee(triparinsertion(W))
Out[136]: True
```

**à propos des algorithmes de tri :**

— https://fr.wikipedia.org/wiki/Tri_rapide
— https://en.wikipedia.org/wiki/Quicksort
— https://fr.wikipedia.org/wiki/Algorithme_de_tri

## 2.3 Permuter

Comme l'indice en Python du premier élément d'une liste est zéro, nous visualiserons (au moins dans un premier temps) les permutations comme des bijections d'ensembles du type $\{0, 1, ..., N-1\}$ (je préviens que ça pourra perturber certains qui ne sont habitués qu'aux ensembles $\{1, ..., N\}$).

Nous représenterons donc en Python (au moins pour le moment) une permutation par une liste `P` de longueur `N` dont les éléments successifs `P[i]` forment une permutation de $\{0, 1, ..., N-1\}$. À la place d'une liste on aurait pu utiliser un `tuple`[38], mais le type `list`[39] est commode car il définit des objets modifiables : on pourra faire une affectation `P[x]=y` ce qui n'est pas possible avec les `tuple`[40] ou les autres objets non-modifiables comme les `str`[41].

Il y a **deux** interprétations possibles à `P=[2, 0, 1]` comme permutation $\sigma$ de l'ensemble $\{0, 1, 2\}$:

1. $\sigma(0) = 2, \sigma(1) = 0, \sigma(2) = 1$, ou
2. $\sigma(2) = 0, \sigma(0) = 1, \sigma(1) = 2$ qui est la permutation *inverse*.

La première correspond à la notation mathématique traditionnelle $\begin{pmatrix} 0 & 1 & 2 \\ 2 & 0 & 1 \end{pmatrix}$. Une permutation est vue comme une bijection de l'ensemble $\{0, 1, 2\}$, et la notation mathématique traditionnelle représente chaque élément au dessus de son image.

La seconde interprétation voit la permutation comme une modification d'un ordre linéaire : avant on avait $(0, 1, 2)$ après on a $(2, 0, 1)$ donc l'objet en position 2 est passé en position 0, l'objet en position 0 est passé en position 1 et l'objet en position 1 est passé en position 2. C'est tout aussi naturel et invite une interprétation opératorielle : si `L=[x, y, z]` est une liste de trois éléments on a envie de dire que l'action de `P` sur `L` sera de la transformer en une autre liste `M = [z, x, y]`, en imitation de `[0, 1, 2]->[2, 0, 1]`.

On peut rapprocher les deux interprétations. En effet la formule pour `M[i]` est visiblement `M[i] = L[P[i]]`: car `M[0] = z = L[2] = L[P[0]]`, `M[1] = x = L[0] = L[P[1]]`, `M[2] = y = L[1] = L[P[2]]`. Donc si on associe à `P` la permutation $\sigma(i) = P[i]$ (première interprétation) alors l'action de `P` sur une liste `L` que l'on peut noter mathématiquement comme un nuple $(l_0, \dots, l_{n-1})$ est de le transformer en le nuple $(m_0, \dots, m_{n-1})$ avec $m_i = l_{\sigma(i)}$. Cela veut dire qu'on met en position $i$ l'objet qui était situé en position $\sigma(i)$. On rejoint donc l'interprétation de $(2, 0, 1)$ comme une permutation de positions d'objets, mais on ne dit pas *objet en position $i$ passe en position $\tau(i)$*, mais *objet*

38. https://docs.python.org/3/library/stdtypes.html#tuple
39. https://docs.python.org/3/library/stdtypes.html#list
40. https://docs.python.org/3/library/stdtypes.html#tuple
41. https://docs.python.org/3/library/stdtypes.html#str

*maintenant en position $i$ était avant en position $\sigma(i)$*. Donc ici, le nouvel objet en position 0, à savoir 2, était avant en position 2, donc $\sigma(0) = 2$ ce qui correspond à la première interprétation.

Pour récapituler, une liste `P` avec N éléments formant une énumeration de $\{0, 1, ..., N-1\}$ est vue :

1. comme une permutation $\sigma$ de cet ensemble via la formule $\sigma(i) = P[i]$,
2. et comme agissant sur toutes les listes `L` de N objets via la formule `L'[i] = L[P[i]]`.

Si l'on fait agir la liste-permutation `Q` sur `P` on obtient par définition la liste-permutation `R` définie par `R[i]=P[Q[i]]`, ce qui correspond bien à la définition mathématique de la composition d'application $P \circ Q$. Au point de vue notation, on doit donc mettre `Q` à droite de `P`. Pour cette raison on dit que la transformation `L->L'` par la formule ci-dessus est une action *à droite* de `P` sur `L`, et il est naturel de noter `LP` ou peut-être $L^P$ cette action. En effet `L(PQ)` sera égal à l'action de `Q` sur `LP`, donc c'est cohérent : `L(PQ)=(LP)Q`.

### 2.3.1 permute(L, P)

---

**Exercice**

Faire une fonction `permute(L, P)` qui renvoie une nouvelle liste égale à l'action à droite de `P` sur `L`. Ces arguments sont donc deux listes du même nombre `N` d'éléments, la seconde étant supposée être une permutation de $\{0, 1, ..., N-1\}$. Ne pas vérifier la validité des arguments passés à la fonction.

---

Je propose trois implémentations. La première est plus lente que les deux autres, et il semble que la troisième soit légèrement plus efficace que la seconde.

```
def permuteA(L, P):
    """Action à droite de la permutation P sur la liste L.

    Pas de contrôle des arguments.

    >>> P = [4, 1, 0, 3, 2]
    >>> L = ['a', 'b', 'c', 'd', 'e']
    >>> permuteA(L, P)
    ['e', 'b', 'a', 'd', 'c']
    """
    M = []
    for i in range(len(P)):
        M.append(L[P[i]])
    return M
```

```
def permuteB(L, P):
    """Action à droite de la permutation P sur la liste L.

    Pas de contrôle des arguments.

    >>> P = [4, 1, 0, 3, 2]
    >>> L = ['a', 'b', 'c', 'd', 'e']
    >>> permuteB(L, P)
    ['e', 'b', 'a', 'd', 'c']
    """
    M = [None]*len(P)  # initialisation
    for i in range(len(P)):
```

```
        M[i] = L[P[i]]
    return M
```

```
def permuteC(L, P):
    """Action à droite de la permutation P sur la liste L.

    Pas de contrôle des arguments.

    >>> P = [4, 1, 0, 3, 2]
    >>> L = ['a', 'b', 'c', 'd', 'e']
    >>> permuteC(L, P)
    ['e', 'b', 'a', 'd', 'c']
    """
    return [L[P[i]] for i in range(len(P))]
```

La composition de deux permutations étant un cas particulier d'une action à droite, il sera inutile d'écrire aussi une procédure pour l'implémenter.

### 2.3.2 permutesurplace(L, P)

**Exercice**

On veut une fonction `permutesurplace(L, P)` qui remplace `L` par le résultat de l'action à droite de `P` sur `L` (on n'exige pas, techniquement, de travailler uniquement avec `L` on peut allouer d'autres objets en mémoire, donc l'expression « sur place » est un peu exagérée).

Ceci ne fonctionne pas :

```
def permutesurplace(L, P):
    L = permute(L, P)
```

comme le montre cette session :

```
In [53]: P = [1, 0, 2]

In [54]: L = ['x', 'y', 'z']

In [55]: permutesurplace(L, P)

In [56]: L
Out[56]: ['x', 'y', 'z']
```

C'est normal puisque qu'en Python les arguments sont passés « par valeur ». Mais attention, pour une liste, cela ne signifie pas qu'une copie est faite de chacun de ses items : une copie est faite de la référence à l'objet de type liste stocké en mémoire qui est lui-même une collection de références à ses items. Si, en local dans une fonction, via cette référence à la liste on modifie un item, cette modification n'est pas annulée en sortie de la fonction : par exemple `L[0]='truc'` aurait un effet permanent sur une liste passée en argument. La procédure ci-dessus ne fait que re-assigner localement le nom `L` à la nouvelle liste en mémoire créée par `permute(L, P)`. L'objet liste passé en premier argument n'est pas modifié. Voir aussi les opérations sur les types modifiables de genre sequence [42].

42. https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types

```python
def permutesurplace(L, P):
    """Action à droite de la permutation P sur la liste L.

    Modifie le L d'origine.

    >>> P = [4, 1, 0, 3, 2]
    >>> L = ['a', 'b', 'c', 'd', 'e']
    >>> permutesurplace(L, P)
    >>> L
    ['e', 'b', 'a', 'd', 'c']
    """
    # L[:] = permute(L, P)
    # ou même directement :
    L[:] = [L[P[i]] for i in range(len(P))]
```

### 2.3.3 inverseperm(P)

**Exercice**

Faire une fonction `inverseperm(P)` qui renvoie une nouvelle liste-permutation représentant l'inverse de `P`.

Une façon affreuse de procéder serait pour chaque `i` de trouver le `j` tel que `P[j]=i`. Mais bien sûr, vous êtes plus astucieux que cela, n'est-ce pas ?

```python
def inverseperm(P):
    """Renvoie la permutation inverse.

    Ne modifie pas la permutation originelle.

    >>> P = [4, 1, 0, 3, 2]
    >>> inverseperm(P)
    [2, 1, 4, 3, 0]
    """
    Q = [None]*len(P)
    for i in range(len(P)):
        Q[P[i]] = i
    return Q
```

### 2.3.4 estuntriage(perm, liste)

On dira qu'une permutation `perm` réalise un triage d'une liste `liste` si l'action à droite de `perm` sur `liste` la transforme en liste ordonnée. Dans le cas où `liste` a des éléments répétés, la permutation réalisant son triage n'est pas unique, elle le devient si on impose la condition de « stabilité » qui est de ne pas permuter les indices des éléments identiques (ou jugés identiques par la relation de comparaison).

Autrement dit on veut que `perm[0]` soit l'indice du plus petit élément, `perm[1]` celui du second plus petit élément, etc…

**Exercice**

Modifier la routine *estordonnee(liste)* (page 35) pour qu'elle accepte en premier argument une permutation et qu'elle retourne `True` si et seulement si cette permutation est un triage, au sens précédent, de la liste donnée en deuxième argument.

```
def estuntriage(perm, liste):
    """True si la permutation de liste par perm l'ordonne.

    >>> L = [4, 1, 0, 3, 2, 2, 3]
    >>> P = [2, 1, 5, 4, 6, 3, 0]
    >>> estuntriage(P, L)
    True
    >>> P = [2, 1, 4, 5, 6, 3, 0]
    >>> estuntriage(P, L)
    True
    >>> P = [2, 1, 4, 5, 6, 0, 3]
    >>> estuntriage(P, L)
    False
    """
    for i in range(len(liste) - 1):
        if liste[perm[i + 1]] < liste[perm[i]]:
            return False
    return True
```

### 2.3.5 `triageparinsertion(L)`

**Exercice**

Faire une fonction `triageparinsertion(L)` qui, avec comme argument une liste de nombres, ou de chaînes, renverra une permutation `P` réalisant un triage de `L`.

L'idée de l'implémentation sera la suivante : la routine *triparinsertion(L)* (page 38) sera étendue pour mettre à jour progressivement une liste `P` de la manière suivante : à chaque fois qu'un nouvel élement `L[i]` sera inséré dans `M` en position `j`, l'indice `i` sera inséré dans `P` en position `j`.

```
def triageparinsertion(liste):
    """Produit une permutation réalisant le tri croissant.

    Autrement dit, produit une liste P avec P[0] l'indice
    du plus petit élément dans L, puis P[1] l'indice du
    second plus petit élément. L'algorithme est stable,
    au sens où il maintient l'ordre d'éléments identiques.

    >>> L = [4, 1, 0, 3, 2, 2, 3]
    >>> P = triageparinsertion(L)
    >>> P
    [2, 1, 4, 5, 3, 6, 0]
    """
    P = []
    longueur = len(liste)
    if longueur == 0:
        return P
    P = [0]
    if longueur == 1:
```

```
        return P
    M = [liste[0]]
    for i in range(1, longueur):
        x = liste[i]
        for j in range(0, i):
            if x < M[j]:
                M.insert(j, x)
                P.insert(j, i)
                break
        else:
            M.append(x)
            P.append(i)
    return P
```

Exemples :

```
In [62]: L = list(range(10, 0, -1)); print(L)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

In [63]: triageparinsertion(L)
Out[63]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

In [64]: L=[9, 9, 9, 6, 6, 6, 3, 3, 3, 1, 1, 1]

In [65]: triageparinsertion(L)
Out[65]: [9, 10, 11, 6, 7, 8, 3, 4, 5, 0, 1, 2]

In [66]: Z = [random.randrange(1, 100) for i in range(20)]

In [67]: Z
Out[67]: [32, 38, 91, 37, 52, 64, 16, 1, 3, 5, 79, 81, 27, 92, 63, 32, 78, 33, 77, 41]

In [68]: P = triageparinsertion(Z)

In [69]: P
Out[69]: [7, 8, 9, 6, 12, 0, 15, 17, 3, 1, 19, 4, 14, 5, 18, 16, 10, 11, 2, 13]

In [70]: estuntriage(P, Z)
Out[70]: True
```

**Exercice**

Montrer que cet algorithme est « stable » au sens suivant: si des éléments coïncident dans la liste initiale, leurs indices resteront dans le même ordre dans la permutation.

### 2.3.6 Tris décorés

Il faut savoir qu'en Python, la comparaison de deux `tuple` [43] par < procède lexicographiquement :

```
>>> ('a', 100) < ('b', 0)
True
>>> ('a', 0) < ('a', 1)
```

43. https://docs.python.org/3/library/stdtypes.html#tuple

```
True
>>> ('a', 5, 3) < ('a', 5, 2)
False
```

Ceci suggère une méthode générale pour transformer n'importe quel algorithme de tri en un algorithme de triage :

1. Décoration : décorer chaque élément `x` d'une liste par son index `i`, c'est-à-dire faire la liste des `(x, i)`. Ceci fonctionne :

   ```
   >>> M = [(L[i], i) for i in range(len(L))]
   ```

   Mais avec `enumerate()` [44] on a une syntaxe plus commode (et probablement plus efficace) :

   ```
   >>> M = [(x, i) for i, x in enumerate(L)]
   ```

2. Appliquer la fonction de tri sur la liste décorée :

   ```
   >>> N = tri(M)
   ```

3. dé-décorer le résultat pour en déduire la permutation de triage :

   ```
   >>> P = [i for x, i in N]
   ```

**Exercice**

Montrer que ces trois étapes donnent une permutation `P` qui constitue un triage de `L` et de plus que cet algorithme est stable, même si l'algorithme sous-jacent à `tri` ne l'était pas.

### 2.3.7 `triageparinsertionII(L)`

**Exercice**

Faire `triageparinsertionII(L)` par décoration de *triparinsertion(L)* (page 38).

Comparer la vitesse d'exécution sur une liste aléatoire de 1000 nombres.

```
def triageparinsertionII(L):
    """Produit une permutation réalisant le tri croissant.

    Obtenu par décoration de triparinsertion.

    >>> L = [4, 1, 0, 3, 2, 2, 3]
    >>> P = triageparinsertionII(L)
    >>> P
    [2, 1, 4, 5, 3, 6, 0]
    """
    M = [(x, i) for i, x in enumerate(L)]
    return [i for x, i in triparinsertion(M)]
```

44. https://docs.python.org/3/library/functions.html#enumerate

```
In [11]: V = list(range(1000))

In [12]: random.shuffle(V)

In [13]: %timeit triageparinsertion(V)
10 loops, best of 3: 21.2 ms per loop

In [14]: %timeit triageparinsertionII(V)
10 loops, best of 3: 28.2 ms per loop

In [15]: estordonnee(permute(V, triageparinsertionII(V)))
Out[15]: True
```

Il y a une pénalité de temps par rapport à *triageparinsertion(L)* (page 49), c'est normal.

### 2.3.8 triagerapide(L)

**Exercice**

Faire une fonction `triagerapide(L)` à partir de `trirapidebis(L)`. L'algorithme sera le suivant : mettre au point

```
triagerapidesurplace(debut, fin, liste, perm)
```

en ajoutant à *trirapidesurplace(debut, fin, liste)* (page 42) les lignes de code nécessaires pour que toutes les transpositions effectuées soient fidèlement répercutées sur `perm`. Au tout début, `perm` représentera la permutation identité.

Comment se comporte notre procédure en terme de vitesse par rapport à *triageparinsertion(L)* (page 49) ? tester sur des listes aléatoires de 100, 500, 1000 et 5000 nombres.

L'algorithme obtenu est-il stable ?

```
def triagerapidesurplace(debut, fin, liste, perm):
    """Trie par algorithme ``QuickSort`` en conservant une trace des permutations.

    Agit « sur place ». Renvoie « None ». Le pivot est pris au milieu.

    La liste passée en argument est modifiée par des échanges d'éléments.
    De plus la permutation passée en dernier argument est également
    mise à jour.

    >>> L = [6, 3, 10, 1, 8, 5]
    >>> P = list(range(len(L)))
    >>> triagerapidesurplace(0, len(L), L, P)
    >>> L
    [1, 3, 5, 6, 8, 10]
    >>> P
    [3, 1, 5, 0, 4, 2]
    """
    # attention, l'indice du dernier élément est fin - 1 pas fin
    longueur = fin - debut
    if longueur < 2:
        return None  # rien à faire
```

```
    ipivot = debut + (longueur // 2)
    vpivot = liste[ipivot]
    ppivot = perm[ipivot]   # l'emplacement dans la liste originelle
    #
    # PREMIÈRE ÉTAPE : PARTITION
    #
    # On commence par échanger le pivot avec la dernière position.
    dernier = fin - 1
    liste[ipivot] = liste[dernier]
    liste[dernier] = vpivot
    # on accompagne ceci au niveau de la permutation
    perm[ipivot] = perm[dernier]
    perm[dernier] = ppivot
    #
    # Ensuite on aura un indice i qui sera celui pour insérer une
    # nouvelle valeur < pivot (donc sa valeur finale sera là où remettre
    # le pivot) et un indice j qui parcourera tous les éléments de la
    # droite vers la gauche. On s'arrêtera lorsque j = i - 1
    #
    i = debut
    j = dernier - 1  # au départ j est au moins égal à i
    for k in range(debut, dernier):
        # en effet on sait exactement combien d'étapes il y aura,
        #  (à savoir longueur - 1 = fin - debut - 1 =dernier - debut)
        # c'est donc probablement plus rapide que faire while j >= i :
        x = liste[j]
        if x < vpivot:
            liste[j] = liste[i]  # <-- cette valeur n'a pas encore été examinée
            liste[i] = x
            # ne pas oublier notre ami perm
            t = perm[j]
            perm[j] = perm[i]
            perm[i] = t
            # on incrémente i avant de boucler
            i += 1
        else:
            j -= 1
    # on remet le pivot entre les deux groupes:
    liste[dernier] = liste[i]
    liste[i] = vpivot
    # aussi pour perm !
    perm[dernier] = perm[i]
    perm[i] = ppivot
    #
    # DEUXIÈME ÉTAPE : RÉCURSION
    #
    triagerapidesurplace(debut, i, liste, perm)
    triagerapidesurplace(i + 1, fin, liste, perm)
    return None
```

```
def triagerapide(L):
    """Produit une permutation P réalisant le tri de L.

    >>> L = [6, 3, 10, 1, 8, 5]
    >>> triagerapide(L)
    [3, 1, 5, 0, 4, 2]
    """
```

```
    longueur = len(L)
    M = L[:]
    P = list(range(longueur))
    if longueur >= 2:
        triagerapidesurplace(0, longueur, M, P)
    return P
```

Voici maintenant en console IPython les vérifications demandées :

```
In [105]: V = []

In [106]: for i in range(100):
     ...:     V.append(random.randrange(1, 100000))
     ...:

In [107]: triageparinsertion(V) == triagerapide(V)
Out[107]: True

In [108]: %timeit -n 10 triagerapide(V)
10 loops, best of 3: 315 µs per loop

In [109]: %timeit -n 10 triageparinsertion(V)
10 loops, best of 3: 285 µs per loop

In [110]: for i in range(400):
     ...:     V.append(random.randrange(1, 100000))
     ...:

In [111]: %timeit -n 10 triageparinsertion(V)
10 loops, best of 3: 5.17 ms per loop

In [112]: %timeit -n 10 triagerapide(V)
10 loops, best of 3: 2.04 ms per loop

In [113]: for i in range(500):
     ...:     V.append(random.randrange(1, 100000))
     ...:

In [114]: %timeit -n 10 triageparinsertion(V)
10 loops, best of 3: 19.5 ms per loop

In [115]: %timeit -n 10 triagerapide(V)
10 loops, best of 3: 4.88 ms per loop

In [116]: for i in range(4000):
     ...:     V.append(random.randrange(1, 100000))
     ...:

In [117]: len(V)
Out[117]: 5000

In [118]: %timeit -n 10 triageparinsertion(V)
10 loops, best of 3: 506 ms per loop

In [119]: %timeit -n 10 triagerapide(V)
10 loops, best of 3: 29.2 ms per loop
```

Le test conditionnel `triageparinsertion(V) == triagerapide(V)` plus haut était risqué ; car si `V` a des éléments répétés, alors `triagerapide(V)` n'est pas garanti de ne pas les permuter :

```
In [120]: L = [9, 9, 9, 6, 6, 6, 3, 3, 3, 1, 1, 1]

In [121]: triageparinsertion(L)
Out[121]: [9, 10, 11, 6, 7, 8, 3, 4, 5, 0, 1, 2]

In [122]: triagerapide(L)
Out[122]: [9, 10, 11, 6, 8, 7, 3, 5, 4, 1, 2, 0]

In [124]: triagerapide(L)==triageparinsertion(L)
Out[124]: False
```

Cet algorithme de triage rapide n'est pas stable.

### 2.3.9 `triagerapideII(L)`

Mais nous avons la méthode générale de décoration !

---

**Exercice**

Faire `triagerapideII(L)` par décoration de `trirapidebis(L)`.

---

```
def triagerapideII(L):
    """Produit une permutation réalisant le tri croissant.

    Obtenu par décoration de trirapidebis.

    >>> L = [6, 3, 10, 1, 8, 5]
    >>> triagerapideII(L)
    [3, 1, 5, 0, 4, 2]
    """
    M = [(x, i) for i, x in enumerate(L)]
    return [i for x, i in trirapidebis(M)]
```

La bonne surprise c'est que c'est même plus efficace que notre *triagerapide(L)* (page 52).

```
In [23]: V = [random.randrange(100000) for i in range(5000)]

In [24]: %timeit triagerapide(V)
10 loops, best of 3: 30.8 ms per loop

In [25]: %timeit triagerapideII(V)
10 loops, best of 3: 26.6 ms per loop
```

De plus `triagerapideII(L)` est « stable » :

```
In [30]: liste = [1]*10

In [31]: triagerapide(liste)
Out[31]: [5, 9, 6, 1, 7, 3, 8, 0, 4, 2]
```

```
In [32]: triagerapideII(liste)
Out[32]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### 2.3.10 triage(liste, tri=sorted)

**Exercice**

Faire une fonction `triage()` avec un argument optionnel qui sera une fonction de tri (égale par défaut à `sorted()` [45]), et qui fonctionnera par décoration de cette fonction de tri.

```
def triage(liste, tri=sorted):
    """Produit une permutation de triage de liste.

    Obtenu par décoration de l'argument optionnel tri.
    """
    M = [(x, i) for i, x in enumerate(liste)]
    return [i for x, i in tri(M)]
```

Cet exercice montre qu'en Python les fonctions sont des objets comme les autres et en particulier peuvent être passées en arguments à d'autres fonctions.

```
In [34]: estuntriage(triage(V), V) # V = 5000 nombres au hasard
Out[34]: True

In [35]: %timeit triage(V)
100 loops, best of 3: 4.33 ms per loop

In [36]: %timeit triage(V, trirapidebis)
10 loops, best of 3: 26.6 ms per loop

In [39]: estuntriage(triage(V, trirapidebis), V)
Out[39]: True
```

### 2.3.11 decompositionencycles(perm)

On ne peut pas quitter ce domaine des permutations sans une routine qui détermine les décompositions en cycles.

**Exercice**

Faire une fonction `decompositionencycles(perm)` qui renvoie une liste composée de `tuple` [46] représentant les cycles de la permutation.

Voici le comportement espéré:

45. https://docs.python.org/3/library/functions.html#sorted
46. https://docs.python.org/3/library/stdtypes.html#tuple

```
In [202]: P = list(range(15))

In [203]: P
Out[203]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

In [204]: random.shuffle(P)

In [205]: P
Out[205]: [8, 3, 5, 13, 11, 1, 7, 6, 2, 12, 14, 9, 10, 0, 4]

In [206]: decompositionencycles(P)
Out[206]: [(0, 8, 2, 5, 1, 3, 13), (4, 11, 9, 12, 10, 14), (6, 7)]

In [207]: random.shuffle(P)

In [208]: P
Out[208]: [13, 1, 5, 10, 4, 6, 8, 3, 2, 12, 7, 0, 14, 9, 11]

In [209]: decompositionencycles(P)
Out[209]: [(0, 13, 9, 12, 14, 11), (1,), (2, 5, 6, 8), (3, 10, 7), (4,)]

In [210]: random.shuffle(P)

In [211]: P
Out[211]: [0, 2, 14, 1, 5, 4, 3, 11, 8, 13, 9, 12, 7, 6, 10]

In [212]: decompositionencycles(P)
Out[212]: [(0,), (1, 2, 14, 10, 9, 13, 6, 3), (4, 5), (7, 11, 12), (8,)]
```

Par la suite vous pourrez modifier la routine pour que les cycles soient ordonnés par leurs longueurs, par exemple.

On note au passage qu'un `tuple`[47] qui ne comporte qu'un seul élément est imprimé par IPython avec une virgule après cet unique élément, ce n'est pas une erreur de ma routine (secrète) `decompositionencycles(perm)`.

J'ai dit que c'était secret, alors à vous de travailler un peu. C'est fini le prof qui fait tout à votre place. Vous avez mangé votre pain blanc !

### 2.3.12 `signature(perm)`

**Exercice**

Faire une fonction `signature(perm)` pour calculer la signature d'une permutation. Il y a plusieurs approches possibles, par exemple une fois la décomposition en cycles connue on en déduit la signature comme valant +1 s'il y a un nombre pair de cycles de longueurs paires, −1 sinon (mais il sera plus efficace de modifier directement l'algorithme qui détermine les cycles pour qu'à la place il calcule la signature, sans conserver en mémoire les cycles).

Je devrais faire de ces + des liens vers des publicités Google, ça me ferait un joli complément de salaire.

47. https://docs.python.org/3/library/stdtypes.html#tuple

*Date de dernière modification* : 03–10–2017 à 15:56:38.

# 3

# Feuille de travaux pratiques 3

*Date de dernière modification* : 11-10-2017 à 08:42:40.



## 3.1 Pièges avec les nombres en virgule flottante

**Exercice**

Quelle est l'écriture en base 2 de $\frac{1}{10}$ ?

Je vous conseille d'y réfléchir un bon moment avant de capituler. Si vous avez déjà une fois dans votre vie réfléchi à comment trouver l'écriture décimale d'une fraction $A/B$, vous devriez pouvoir recycler vos connaissances à la base 2… sinon, ça va pas être évident !

Tout d'abord on regarde $\frac{1}{5}$, pour avoir un dénominateur premier avec la base (ici, 2). Ensuite si le dénominateur $d = 5$ était de la forme $2^n - 1$, analogue de $999\cdots99$ en base 10, on saurait que $\frac{1}{d} = 0.\overline{0000...01}$, la barre indiquant que le motif (dont la longueur serait $n$) se répète indéfiniment. Et pour $\frac{a}{d}$ avec un $a < 2^n - 1$ l'expansion binaire sera périodique immédiatement avec comme

motif l'écriture binaire de $a$, avec éventuellement des zéros pour qu'il y ait $n$ chiffres. Tout cela, vous devriez en être très familier en base 10, et ça marche pareil en base 2. L'astuce lorsque l'on part, comme ici, de $\frac{1}{5}$ est donc de trouver un facteur $K$ tel que $5K$ soit de la forme $2^n - 1$. Il faut déterminer la plus petite puissance de 2 congrue à 1 modulo 5. Le groupe $(\mathbb{Z}/5\mathbb{Z})^*$ est fini (cyclique même car 5 est premier, mais ce n'est pas important ici), et le $n$ cherché est l'ordre de 2 dans ce groupe multiplicatif. $1, 2, 4, 8 = 3, 6 = 1$, ainsi $n = 4$, $2^4 - 1 = 16 - 1 = 15 = 5 \times 3$. On écrit donc :

$$\frac{1}{5} = \frac{3}{16-1} = 3 \times 0.\overline{0001} = 0.\overline{0011}$$

L'écriture de $\frac{1}{10}$ en binaire est ainsi :

$$\frac{1}{10} = 0.00\overline{011}$$

Le motif surligné se répètant indéfiniment.

Bon, on voit donc qu'il faut un *nombre infini* de chiffres binaires pour représenter 1/10 exactement. Du coup, quel que soit l'ordinateur sur lequel vous utilisez Python, la représentation interne de 0.1 est *nécessairement inexacte*. Du coup ce qui suit ne devrait pas trop surprendre :

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>> 0.1 + 0.1 + 0.1 == 0.3
False
>>> 7 * 0.1 == 0.7
False
```

Il faut toujours conserver ceci à l'esprit lorsque l'on utilise les nombres en virgule flottante.

Parfois « ça marche » :

```
>>> 1/10 == 0.1
True
>>> 10 * 0.1 == 1
True
>>> 5 * 0.1 == 0.5
True
>>> 8 * 0.125 == 1
True
>>> 30 * 0.1 == 3
True
```

Mais souvent non, comme déjà illustré. Sur mon ordinateur, Python utilise une précision de 53 chiffres binaires pour les nombres en virgule flottante. Il y a l'inexactitude liée à cette approximation, accentuée par les problèmes d'arrondis dès que l'on fait une opération. Par exemple une multiplication pourrait avoir besoin de $53+53 = 106$ chiffres binaires, mais elle doit être arrondie à 53 seulement.

```
>>> x = 0.1
>>> x.hex()  # ou directement (0.1).hex()
'0x1.999999999999ap-4'
```

Ceci nous informe que la « mantisse » hexadécimale est `199999999999A`, qui occupe $1+13\times4 = 53$ chiffres binaires. Le `p-4` signifie qu'il faut encore multiplier par `2**(-4) = 1/16`. La mémoire de l'ordinateur stocke seulement 52 chiffres binaires de la mantisse, car le 1 initial est tacite. Mais bref, nous voyons que pour Python, `0.1` est en fait stocké sous une forme ayant une valeur binaire :

```
0.0001100110011001100110011001100110011001100110011001100110011010
```

alors que depuis l'exercice précédent on sait que l'écriture exacte est infiniment longue :

```
0.0001100110011001100110011001100110011001100110011001100110011001100110011....
```

Remarquez que la valeur stockée est bien l'arrondi de la valeur exacte, pour une mantisse avec 53 chiffres binaires exactement (dont un 1 en premier). En particulier on voit que la valeur stockée est un peu plus grande que la valeur exacte. Lorsque l'on fait `0.1 + 0.1 + 0.1` le petit écart est multiplié par trois, et il n'est pas surprenant que le résultat de l'opération diffère de la représentation interne de `0.3` (mais attention à la surprise à la fin de cette section).

---

**Exercice**

Quelle est l'écriture binaire exacte de 0.3 ? Quelle est son approximation stockée par Python ?

---

```
>>> (0.3).hex()
'0x1.3333333333333p-2'
>>> (0.3).as_integer_ratio()
(5404319552844595, 18014398509481984)
```

L'approximation utilisée par Python est :

```
0.0100110011001100110011001100110011001100110011001100110011
```

qui est l'écriture binaire de

$$\frac{5404319552844595}{18014398509481984} = \frac{5404319552844595}{2^{54}}$$

ce qui n'est pas 3/10, mais seulement une (bonne) approximation :

```
>>> 3*2**54
54043195528445952
>>> 10*5404319552844595
54043195528445950
```

La différence vaut donc $2/(10 \cdot 2^{54}) \approx 10^{-17}$, grosso modo cela veut dire que la représentation interne ne restitue qu'environ seize à dix-sept chiffres décimaux de la valeur abstraite exacte.

Valeur exacte qui est en binaire :

```
0.0100110011001100110011001100110011001100110011001100110011....
```

avec un motif `1001` se répétant une infinité de fois. La valeur utilisée par Python est son arrondi à 53 chiffres binaires significatifs (on compte à partir du premier 1).

Suppons qu'on parte de la valeur approchée de `0.1` (décimal) donnée en binaire :

```
y =  0.0001100110011001100110011001100110011001100110011001100110011010
```

Si on calcule maintenant `y + y + y` exactement, on obtient :

```
2y = 0.0011001100110011001100110011001100110011001100110011001100110010
on additionne avec les retenues, en partant de la droite :
3y = 0.0100110011001100110011001100110011001100110011001100110011001110
```

En partant du premier 1 inclus, on sait que de toute façon Python ne stocke que 53 chiffres binaires. Donc on arrondit cette valeur théorique. En faisant attention, on constate que cet arrondi donne :

```
3y arrondi = 0.0100110011001100110011001100110011001100110011001100110100
convertissons en hex:
  2**(-2) fois 1.   3   3   3   3   3   3   3   3   3   3   3   3   4
```

Ce n'est pas comme cela que Python évalue `0.1 + 0.1 + 0.1` mais vérifions à tout hasard :

```
>>> (0.1 + 0.1 + 0.1).hex()
'0x1.3333333333334p-2'
```

C'est bien notre résultat précédent. Ainsi Python obtient comme résultat de cette opération l'arrondi correct du résultat théorique *évalué à partir de sa représentation interne approchée de* `0.1`. Et pour comparaison :

```
>>> (0.3).hex()
'0x1.3333333333333p-2'
```

On voit donc dans ce cas que tout le problème vient de l'erreur faite initialement en approchant `0.1` par un développement binaire fini. Le calcul `0.1 + 0.1 + 0.1` est fait de manière précise, bien que nécessitant un arrondi final pour le caser en 53 chiffres binaires, mais le point de départ avait un écart suffisamment grand d'avec la valeur exacte pour que le dernier chiffre binaire de ce résultat diffère du dernier chiffre binaire de *l'arrondi à 53 chiffres binaires* de la valeur théorique exacte.

Et comme si tout cela n'était pas assez compliqué, je n'ai même pas encore évoqué le problème supplémentaire qui est que Python doit afficher une valeur *décimale* pour le résultat qu'il aura obtenu en binaire lors de son évaluation de `0.1 + 0.1 + 0.1`…

Bon, on est déjà allé loin, mais une dernière chose. On va vu que la représentation interne de `0.1` était plus grande que sa valeur exacte. Et que la valeur calculée de `0.1 + 0.1 + 0.1` était affichée par Python comme `0.30000000000000004`. À votre avis que va donner :

```
0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
```

Eh bien un truc surprenant à ce stade :

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
0.9999999999999999
```

C'est strictement inférieur à 1. Amusant, non ? Les arrondis peuvent parfois aller vers le bas, et même si la représentation interne initiale était supérieure à la valeur théorique, cela peut donner après plusieurs additions un résultat *inférieur* à la valeur théorique !

La fonction `math.fsum()` [48] du module `math` [49] est censée minimiser le cumul d'erreurs d'arrondis. Voici un test :

48. https://docs.python.org/3/library/math.html#math.fsum
49. https://docs.python.org/3/library/math.html#module-math

```
>>> L = [0.1 for i in range(30)]
>>> for i in range(30):
...     print('{:<20} {:<20}'.format(sum(L[:i]), math.fsum(L[:i])))
...
0                    0.0
0.1                  0.1
0.2                  0.2
0.30000000000000004  0.30000000000000004
0.4                  0.4
0.5                  0.5
0.6                  0.6000000000000001
0.7                  0.7000000000000001
0.7999999999999999   0.8
0.8999999999999999   0.9
0.9999999999999999   1.0
1.0999999999999999   1.1
1.2                  1.2000000000000002
1.3                  1.3
1.4000000000000001   1.4000000000000001
1.5000000000000002   1.5
1.6000000000000003   1.6
1.7000000000000004   1.7000000000000002
1.8000000000000005   1.8
1.9000000000000006   1.9000000000000001
2.0000000000000004   2.0
2.1000000000000005   2.1
2.2000000000000006   2.2
2.3000000000000007   2.3000000000000003
2.400000000000001    2.4000000000000004
2.500000000000001    2.5
2.600000000000001    2.6
2.700000000000001    2.7
2.800000000000001    2.8000000000000003
2.9000000000000012   2.9000000000000004
```

Certes en effet la seconde colonne est en générale plus proche du résultat naïvement espéré, mais ça conserve un air de mystère. Conclusion: **méfiance avec les flottants** ! même en l'absence de toute fonction compliquée genre sinus ou cosinus, déjà avec les opérations de l'École primaire on confie à Python des nombres en base 10 (avec des chiffres après la virgule), il les convertit en des *approximations* binaires, puis il fait les opérations *qui induisent des arrondis*, et finalement il affiche le résultat après une dernière conversion de base, cette fois-ci du binaire vers le décimal, et elle aussi ne peut être en générale qu'*approximative* !

Dans certains domaines comme la comptabilité il est indispensable que les calculs soient vraiment faits en base 10, et pas en base 2 à cause de tous les problèmes évoqués ci-dessus. Python met à disposition un module `decimal` [50] qui sait faire des calculs en précision arbitraire en base 10.

Références:

— Numeric Types — int, float, complex [51].
— Floating Point Arithmetic: Issues and Limitations [52].
— Documentation du module decimal [53].

50. https://docs.python.org/3/library/decimal.html#module-decimal
51. https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
52. https://docs.python.org/3/tutorial/floatingpoint.html
53. https://docs.python.org/fr/3/library/decimal.html#module-decimal

Une autre limitation des nombres en virgule flottante est que non seulement la mantisse a un certain nombre maximal de chiffres, de plus l'exposant est limité.

**Exercice**

Quel est le plus grand nombre réel exactement représentable comme un `float` [54] sur votre ordinateur. Indication: utiliser `sys.float_info.max`.

```
>>> import sys
>>> x = sys.float_info.max
>>> x
1.7976931348623157e+308
>>> x.hex()
'0x1.fffffffffffffp+1023'
```

Ceci est donc exactement $(2 - 16^{-13}) \cdot 2^{1023} = (1 - 2^{-53}) \cdot 2^{1024}$.

Surprise:

```
>>> y = x + 1
>>> y
1.7976931348623157e+308
>>> y == x
True
```

**Exercice**

Quel est le plus petit nombre réel strictement positif exactement représentable comme un `float` [55] sur votre ordinateur ?

```
>>> z = sys.float_info.min
>>> z
2.2250738585072014e-308
>>> z.hex()
'0x1.0000000000000p-1022'
```

Il s'agit donc apparemment de $2^{-1022}$. Mais en fait non, car on peut obtenir des nombres dits « sous-normaux »:

```
>>> float.fromhex('0x0.0000000000001p-1022')
5e-324
>>> float.fromhex('0x0.0000000000009p-1022')
5e-324
>>> float.fromhex('0x0.0000000000008p-1022')
0.0
>>> float.fromhex('0x0.000000000000081p-1022')
5e-324
>>> float.fromhex('0x0.00000000000000800000001p-1022')
5e-324
>>> float.fromhex('0x0.00000000000008000000000001p-1022')
5e-324
```

54. https://docs.python.org/3/library/functions.html#float
55. https://docs.python.org/3/library/functions.html#float

```
>>> float.fromhex('0x0.0000000000000800000000000p-1022')
0.0
```

On voit donc qu'il y a un arrondi-là sur l'input, et que le plus grand nombre réel que Python va arrondir à zéro est semble-t-il $8 \cdot 16^{-14} \cdot 2^{-1022} = 2^{-1075}$. Ensuite ceux un peu plus grands sont arrondis à $2^{-1074}$ qui est le plus petit réel non nul exactement représenté, et le prochain exactement représentable est son double $2^{-1073}$.

```
>>> import sys
>>> w = sys.float_info.min / 2**52
>>> w
5e-324
>>> w.hex()
'0x0.0000000000001p-1022'
>>> w / 2
0.0
>>> w == 2 * w
False
>>> (2 * w).hex()
'0x0.0000000000002p-1022'
```

## 3.2 Racines énièmes de grands entiers

Python autorise la manipulation d'entiers arbitrairement grands. La question suivante paraît simple : *étant donné un entier positif* `x` *déterminer s'il est le carré d'un autre entier*. Évidemment on ne va pas calculer `1**2, 2**2, 3**2, ...` jusqu'à dépasser strictement `x` ou tomber exactement sur lui.

Donc on se dit : je vais utiliser la fonction « racine carrée » de Python: `math.sqrt(x)`. Ensuite je prends le plus proche entier `t = round(math.sqrt(x))` et j'examine si `t * t` est plus petit, plus grand ou égal à `x`, et j'ajuste si besoin.

Ça ne marche pas du tout :

1. ce `t = round(math.sqrt(x))`, si `x` est vraiment un carré disons de 100 chiffres, ne sera de toute façon qu'une approximation avec environ 16 ou 17 chiffres décimaux de précision (plus précisément : 53 chiffres binaires) donc même si `x` est véritablement un carré parfait `t * t == x` aura très certainement `False` comme valeur de vérité, et il reste environ 34 chiffres décimaux après les 16 premiers à déterminer.
2. de plus pour ce `math.sqrt(x)` il y a donc déjà une conversion implicite de `x` en un `float` [56] ce qui limite sa taille, comme on a vu précédemment.

```
>>> import math
>>> math.sqrt(2**1000)
3.273390607896142e+150
>>> math.sqrt(2**2000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
```

56. https://docs.python.org/3/library/functions.html#float

### 3.2.1 Méthode babylonienne d'approximation d'une racine carrée

Je rappelle l'algorithme célèbre pour calculer de manière approchée la racine carrée d'un nombre réel : on itère la fonction $t \mapsto f(t) = \frac{1}{2}(t + \frac{x}{t})$. Si $u_0 > \sqrt{x}$, la suite $u_{n+1} = f(u_n)$ est strictement décroissante et converge rapidement vers $\sqrt{x}$. Si $x$ est un nombre entier, les $u_n$ seront des nombres rationnels. Il nous faut un algorithme uniquement avec des nombres entiers.

On modifie la formule de récurrence babylonienne en appliquant la fonction partie entière :

$$t_{n+1} = \lfloor f(t_n) \rfloor$$

On prendra pour $t_0$ un *entier* dont on est sûr qu'il est $> \sqrt{x}$, par exemple $t_0 = x$ (en supposant donc $x > 1$), mais ensuite il faudra réfléchir à comment améliorer le choix de ce $t_0$. Tant que $t_n > \sqrt{x}$, on sait que $t_n > f(t_n) > \sqrt{x}$, donc certainement $t_{n+1} < t_n$.

Mais comment repérer le premier $n$ avec $t_n \leqslant \sqrt{x}$ ? (à ce stade on arrêtera l'algorithme). Déjà comment calculer $t_{n+1}$ avec uniquement des opérations entières ? Si on fait la division euclidienne $x = q_n t_n + r_n$, on a $f(t_n) = \frac{1}{2}(t_n + q_n + \frac{r_n}{t_n})$.

---

**Exercice**

Montrer $t_{n+1} = \lfloor \frac{1}{2}(t_n + q_n) \rfloor$. Il faut rédiger une démonstration sur un papier, avec un crayon.

---

En Python cela donne : `q = x // t` puis `t = (t + q) // 2`.

---

**Exercice**

Montrer que le premier $n$ avec $t_n \leqslant \sqrt{x}$ est aussi le premier $n$ avec $t_n \leqslant q_n$. Autrement dit, montrer les implications :

1. $t_n > \sqrt{x} \Rightarrow t_n > q_n$, et
2. $t_n \leqslant \sqrt{x} \Rightarrow t_n \leqslant q_n$.

---

Notre premier $t_n \leqslant \sqrt{x}$ est la partie entière de $f(t_{n-1})$, qui lui-même est obligatoirement $> \sqrt{x}$ car $t_{n-1} > \sqrt{x}$. Donc ce $t_n$ est le plus grand entier inférieur à $\sqrt{x}$ (puisque déjà il est le plus grand entier inférieur à $f(t_{n-1}) > \sqrt{x}$). Donc $t_n = \lfloor \sqrt{x} \rfloor$.

### 3.2.2 `racinecarreeint(n)`

---

**Exercice**

Implémenter en Python `racinecarreeint(n)` suivant l'algorithme décrit précédemment.

La routine devra admettre un second argument optionnel, qui sera une fonction [57] de la variable `n` qui calculera un `t` initial approprié. Pour le moment, cette fonction `tdefaut(n)` sera simplement définie comme renvoyant `n`.

Pour définir un argument optionnel avec une valeur par défaut, la syntaxe est `def foo(..., x=X):`. Cela signifie que `foo(..)` peut être utilisée sans le dernier argument et alors c'est `X` qui en sera la valeur par défaut.

57. en Python les fonctions sont des objets comme les autres et peuvent donc être passées en arguments à d'autres fonctions.

Donc on demande quelque chose comme `def racinecarreeint(n, f=tdefaut):`.

```
def tdefaut(n):
    """Choix du point de départ pour algorithme de racine carrée.

    Doit être t avec t*t > n. On peut supposer n > 1.
    Destiné à être remplacé par un choix meilleur.
    """
    return n
```

```
def racinecarreeint(n, tinitial=tdefaut):
    """Calcule la partie entière de la racine carrée.

    Fonctionne avec des entiers arbitrairement grands.
    L'entier n est supposé positif.

    >>> [racinecarreeint(x) for x in range(10)]
    [0, 1, 1, 1, 2, 2, 2, 2, 2, 3]
    >>> racinecarreeint(10000000)
    3162
    >>> racinecarreeint(10000000000000)
    3162277
    """
    if n < 2:
        return n
    t = tinitial(n)
    q = n // t
    while q < t:
        t = (t + q) // 2
        q = n // t
    return t
```

Exemple :

```
>>> racinecarreeint(123456789012345678901234567890123456789012345678901234567890123456789012345)
11111111061111110993611110 58
```

Le résultat a 28 chiffres, et on peut le comparer à l'information partielle fournie par l'emploi de `math.sqrt()` [58] :

```
>>> math.sqrt(123456789012345678901234567890123456789012345678901234567890123456789012345)
1.111111106111111e+27
```

### 3.2.3 testracinecarreeint(N, reps)

**Exercice**

Ecrire une procédure `testracinecarreeint(N, reps)` qui choisit au hasard `reps` nombres `n` de exactement `N` chiffres, évalue `m = racinecarreeint(n)` et vérifie que `m` est bien le plus grand entier tel que `m * m <= n`. À la première erreur la procédure s'arrête et imprime le `n` et `m` problématiques (ce qui n'arrivera jamais…). Sinon elle imprime `OK`. La valeur par défaut de l'argument

58. https://docs.python.org/3/library/math.html#math.sqrt

optionnel reps sera 100.

```
def testracinecarreeint(N, reps=100):
    """Test de validité de la procédure ``racinecarreeint(n, tdefaut)``.

    >>> testracinecarreeint(10)
    OK
    """
    b = 10**N
    a = b // 10
    for i in range(reps):
        n = randrange(a, b)
        m = racinecarreeint(n)
        x = m * m
        y = x + 2 * m + 1
        if x > n:
            print('ERREUR par excès', n, m)
            break
        if y <= n:
            print('ERREUR par défaut', n, m)
            break
    else:
        print('OK')
```

### 3.2.4 t0parbitlength(n)

Pour le choix de la première valeur de t : si x s'écrit avec A chiffres en décimal alors x < 10**A, et donc une possibilité est t = 10**B avec B = (A + 1) // 2 (le plus petit entier dont le double est au moins A).

Ce A égal au nombre de chiffres de l'écriture décimale de x s'obtient par len(str(x)). Il y a aussi len(repr(x)) qui est légèrement plus rapide que len(str(x)). Voir repr() [59]. Mais bref, len(str(x)) comme len(repr(x)) sont coûteuses car nécessitant conversion de x de sa représentation interne binaire vers une chaîne de chiffres décimaux.

On peut aussi itérer x=x//10 jusqu'à le réduire à zéro et compter le nombre d'étapes, ce qui me semble tout autant coûteux. En fait c'est bien pire : voir plus bas *longueurdecimale_lent(n)* (page 77).

Il est *indispensable* de bien choisir le t initial. Tant que t est, disons, au moins 10 fois $\sqrt{x}$, alors q = x // t est moins d'un centième de t, donc notre formule (t + q) // 2 c'est à peu près t // 2 et en fait ce dernier est un meilleur choix : c'était idiot de calculer le q. Ce serait mieux, à tout faire, d'itérer t = t // 2 jusqu'au premier qui vérifie t * t <= 4 * x. Et pourquoi diviser seulement par 2 ? Autant diviser par un plus gros truc genre 2**32 (en faisant attention à la fin).

**Important :** Le mieux ça serait de connaître l'exposant $k$ tel que $2^{k-1} \leqslant x < 2^k$, c'est-à-dire il nous faut le nombre de chiffres de $x$ en binaire. Par chance il existe une fonction « *built-in* » en Python qui fournit cette information, plus précisément il s'agit d'une méthode de la classe int [60] :

```
>>> x = 2**1000
>>> x.bit_length()
1001
```

59. https://docs.python.org/3/library/functions.html#repr
60. https://docs.python.org/3/library/functions.html#int

```
>>> x = 2**1000 - 1
>>> x.bit_length()
1000
```

On ne peut faire ni `bit_length(127)` ni même `127.bit_length()` mais `int(127).bit_length()` et `(127).bit_length()` fonctionnent.

---

La méthode `int.bit_length()` [61] a un temps d'exécution [62] qui dépend peu de l'entier en question, et en tout cas pas vraiment de la taille qu'il occupe en mémoire.

Voici quelques exemples :

```
In [1]: x = 2**1000 - 1

In [2]: %timeit x.bit_length()
10000000 loops, best of 3: 97.7 ns per loop

In [3]: y = 3**1000

In [4]: %timeit y.bit_length()
10000000 loops, best of 3: 113 ns per loop

In [5]: z = 3**2000

In [6]: %timeit z.bit_length()
10000000 loops, best of 3: 94.8 ns per loop

In [7]: x  =  10

In [8]: %timeit x.bit_length()
10000000 loops, best of 3: 79.8 ns per loop

In [9]: x = 3**100000

In [10]: %timeit x.bit_length()
10000000 loops, best of 3: 93.9 ns per loop
```

---

**Exercice**

Faire une procédure `t0parbitlength(n)` qui renvoie le plus petit `t` égal à une puissance de 2 et vérifiant `t * t > n`.

---

```
def t0parbitlength(n):
    """Choix du point de départ pour l'algorithme de racine carrée.

    Trouve la plus petite puissance t de 2 avec t * t > n.
    Accepte même n = 0 (alors t = 1) ou n = 1 (alors t = 2).

    >>> [t0parbitlength(x) for x in range(10)]
    [1, 2, 2, 2, 4, 4, 4, 4, 4, 4]
    >>> t0parbitlength(63)
```

61. https://docs.python.org/3/library/stdtypes.html#int.bit_length
62. Les temps d'exécutions sur cette feuille sont obtenus sur plusieurs ordinateurs aux performances variables, on ne peut pas comparer les temps d'une section à ceux d'une autre.

```
    8
    >>> t0parbitlength(64)
    16
    """
    t = 1 << ((n.bit_length() + 1) >> 1)
    # ou t = 2 << ((n.bit_length() - 1) >> 1) mais alors n=0 pas possible
    # car (0).bitlength() renvoie zéro et (-1) >> 1 donne -1.
    return t
```

Explications mathématiques :

pour $x$ un nombre réel et $N$ un nombre entier l'inégalité $N \geqslant x$ équivaut à $N \geqslant \lceil x \rceil$, la notation $\lceil x \rceil$ désignant l'arrondi de $x$ vers l'entier suivant « en allant vers $+\infty$ ». Si $x$ n'est pas entier on a en fonction de la partie entière $\lceil x \rceil = [x] + 1$, mais si $x$ est entier bien sûr $\lceil x \rceil = x = [x]$. Je rappelle que la « partie entière » $[x]$ aussi notée $\lfloor x \rfloor$ est l'entier précédant $x$ « en venant de $-\infty$ ». On devine donc la formule $\lceil x \rceil = -\lfloor -x \rfloor$. Voir la page partie entière et partie fractionnaire [63] de Wikipédia.

En Python, pour deux entiers `a // b` calcule le quotient euclidien de `a` par `b`, c'est à dire $\lfloor \frac{a}{b} \rfloor$, à condition, attention, que `b` est strictement positif.

Parfois on a besoin de $\lceil \frac{a}{b} \rceil$. On peut donc l'obtenir par `-((-a) // b)`, mais il y a aussi cette formule que je laisse en exercice: `1 + (a - 1) // b`. Il y a aussi une fonction `math.ceil()` [64].

Finalement lorsque `b` est 2 ou plutôt une puissance `2**k` une façon plus efficace pour Python de calculer `a // 2**k` est `a >> k`, qui utilise l'opérateur de décalage binaire [65] `>>`.

Donc par exemple `1 + (a - 1) // 2 = (a + 1) // 2 = (a + 1) >> 1` est le « plafond » de $\frac{a}{2}$, c'est-à-dire le plus petit entier $N$ plus grand que le réel $\frac{a}{2}$. Vous aurez besoin de toutes ces explications pour comprendre la solution de l'exercice.

Note: cela m'a un peu surpris au départ (j'aurais dû réfléchir, voir la suite) mais `a >> k` est le quotient euclidien `a // 2**k` également lorsque `a` est négatif.

```
In [5]: 15 >> 2, (-15) >> 2
Out[5]: (3, -4)

In [6]: 15 // 4, (-15) // 4
Out[6]: (3, -4)
```

Python utilise (je suppose) en interne une représentation en complément à deux [66] des nombres négatifs, donc en effet −15 sera représenté par `..11110001`, et après un double décalage vers la droite `..11111100` correspond bien à −4. On peut comprendre l'effet des opérateurs binaires comme si les nombres négatifs avaient une infinité de 1 sur la gauche, mais bien sûr la représentation interne réelle est finie ! Je présume qu'elle utilise effectivement un complément à deux [67] avec un nombre de bits suffisant, mais ce qui est sûr c'est que les opérations binaires sont implémentées de manière à donner le même résultat que si les nombres négatifs étaient écrits en complément à deux [68] avec une infinité de 1 sur la gauche.

Voici ce que donne son utilisation :

63. https://fr.wikipedia.org/wiki/Partie_enti%C3%A8re_et_partie_fractionnaire
64. https://docs.python.org/3/library/math.html#math.ceil
65. https://docs.python.org/3/library/stdtypes.html#bitwise-operations-on-integer-types
66. https://fr.wikipedia.org/wiki/Compl%C3%A9ment_%C3%A0_deux
67. https://fr.wikipedia.org/wiki/Compl%C3%A9ment_%C3%A0_deux
68. https://fr.wikipedia.org/wiki/Compl%C3%A9ment_%C3%A0_deux

```
In [47]: x = 2**2999

In [48]: racinecarreeint(x) == racinecarreeint(x, t0parbitlength)
Out[48]: True

In [49]: %timeit racinecarreeint(x)
100 loops, best of 3: 6.35 ms per loop

In [50]: %timeit racinecarreeint(x, t0parbitlength)
10000 loops, best of 3: 60.3 µs per loop

In [51]: x=12345678901234567890123456789012345678901234567890123456789012345

In [52]: racinecarreeint(x) == racinecarreeint(x, t0parbitlength)
Out[52]: True

In [53]: %timeit racinecarreeint(x)
10000 loops, best of 3: 27.4 µs per loop

In [54]: %timeit racinecarreeint(x, t0parbitlength)
100000 loops, best of 3: 2.88 µs per loop
```

Pour x égal à 2**2999 on a donc divisé par plus de 100 le temps de calcul !

### 3.2.5 racineeniemeint_temp(n, k)

Pour calculer la racine k ième, la méthode de Newton pour trouver un zéro de $t^k - x$ suggère d'itérer $t \mapsto t - (t^k - x)/(kt^{k-1})$.

---

**Exercice**

Montrer que l'algorithme :

```
débuter avec t tel que t**k > n, par exemple t = n si n>1
Faire q <-- n // t**(k - 1)
  si q >= t, fin : l'entier cherché est t.
  si q <  t, faire t <-- ((k - 1) * t + q) // k
Itérer
```

calcule $\lfloor \sqrt[k]{n} \rfloor$.

Faire une fonction `racineeniemeint_temp(n, k)` implémentant cet algorithme, avec `t=n` comme point de départ.

---

Je laisse à votre sagacité la justification de l'algorithme. Voici en tout cas une implémentation :

```
def racineeniemeint_temp(n, k):
    """Calcule la partie entière de la racine k ième de n.

    Fonctionne avec des entiers arbitrairement grands.
    L'entier n est supposé positif. Et k est au moins 2.
    En fait ça marche aussi avec k=1.

    >>> racineeniemeint_temp(10000000, 5)
```

```
    25
    """
    if n < 2:
        return n
    t = n
    j = k - 1
    q = n // (t**j)
    while q < t:
        t = (j * t + q) // k
        q = n // (t**j)
    return t
```

```
>>> racineeniemeint_temp(12345678901234567890123456789012345678901234567890123456789012345, 5)
65811682742
>>> 65811682742**5
1234567890057834473855648396307975427707570808680187232
>>> 65811682743**5
1234567890151629970236982413072479625533045826549521943
```

Prendre `n` comme valeur initiale de `t` est assez catastrophique : plus `k` est grand, plus on aura longtemps `q = 0`, et plus `t` decroîtra très lentement au départ. Plus encore que pour la racine carrée, il faut se positionner au plus près de la vraie racine kième (que l'on ne connaît pas !) avant d'enclencher l'algorithme.

### 3.2.6 tparbitlength(n, k)

**Exercice**

Faire une fonction `tparbitlength(n, k)` qui renvoie la plus petite puissance de deux dont la puissance `k` ième est strictement plus grande que `n` (on pourra supposer `n > 0`).

```
def tparbitlength(n, k):
    """Renvoie la plus petite puissance t de 2 telle que t^k>n

    Suppose n au moins 1.

    >>> tparbitlength(63, 2)
    8
    >>> tparbitlength(64, 2)
    16
    >>> tparbitlength(60, 3)
    4
    >>> tparbitlength(100, 3)
    8
    >>> tparbitlength(100000, 5)
    16
    """
    return 2 << ((n.bit_length() - 1) // k)
```

Explication : avec $l$ la longueur binaire de `n`, l'exposant recherché est $\lceil \frac{l}{k} \rceil$ qui est aussi $1 + \lfloor \frac{l-1}{k} \rfloor$. On aurait pu utiliser `1 << ((n.bit_length() + k - 1) // k)` qui aurait l'avantage de fonctionner aussi avec `n` nul.

### 3.2.7 racineeniemeint(n, k)

**Exercice**

Faire une fonction `racineeniemeint(n, k)` comme *racineeniemeint_temp(n, k)* (page 71) mais qui utilisera `tparbitlength(n, k)` pour le `t` initial.

```
def racineeniemeint(n, k):
    """Calcule la partie entière de la racine k ième de n.

    Fonctionne avec des entiers arbitrairement grands.
    L'entier n est supposé positif.

    >>> racineeniemeint(63, 2)
    7
    >>> racineeniemeint(64, 2)
    8
    >>> racineeniemeint(60, 3)
    3
    >>> racineeniemeint(100, 3)
    4
    >>> racineeniemeint(100000, 5)
    10
    >>> racineeniemeint(99999, 5)
    9
    """
    if n < 2:
        return n
    t = tparbitlength(n, k)
    # on pourrait tester si t == 1 ici
    j = k - 1
    q = n // (t**j)
    while q < t:
        t = (j * t + q) // k
        q = n // (t**j)
    return t
```

```
In [95]: x=12345678901234567890123456789012345678901234567890123456789012345

In [96]: racineeniemeint(x,5)
Out[96]: 65811682742

In [97]: racineeniemeint_temp(x,5)
Out[97]: 65811682742

In [98]: %timeit racineeniemeint_temp(x,5)
1000 loops, best of 3: 347 µs per loop

In [99]: %timeit racineeniemeint(x,5)
100000 loops, best of 3: 4.43 µs per loop
```

### 3.2.8 estunepuissance(n)

**Exercice**

Montrer que l'algorithme suivant détermine si un entier non négatif $n$ peut s'écrire sous la forme $a^k$ avec $a > 1$ et $k > 1$ :

```
Traiter à part le cas n == 1. Dans la suite n > 1.

Poser k = 2.

Calculer a = partie entière de la racine k ième de n,

 Si a = 1 s'arrêter et renvoyer False.

 Sinon regarder si n == a**k, si c'est le cas s'arrêter en imprimant
 la solution trouvée et en renvoyant True.

 Si ce n'est pas le cas, incrémenter k.

Itérer.
```

En donner une implémentation `estunepuissance(n)`.

```python
def estunepuissance(n):
    """Renvoie True si on peut trouver a et k (k > 1) avec n == a**k

    Assume n positif.

    >>> estunepuissance(32)
    n est la puissance 5ième de 2
    True
    >>> estunepuissance(243)
    n est la puissance 5ième de 3
    True
    >>> estunepuissance(2000)
    False
    """
    if n == 0:
        print(n, 'est nul')
        return False
    if n == 1:
        print(n, 'est 1')
        return False
    a = racinecarreeint(n, t0parbitlength)
    if a * a == n:
        print('n est le carré de', a)
        return True
    k = 3
    while True:
        a = racineeniemeint(n, k)
        if a == 1:
            return False
        if a**k == n:
            print('n est la puissance {}ième de {}'.format(k, a))
            return True
        k += 2
```

Pourquoi k += 2 à la fin ?

```
In [42]:␣
 ↪estunepuissance(6801627477633567653059023813514866104681179785399615827535771334429109739038826587477206
n est la puissance 17ième de 11193791827391
Out[42]: True
In [43]:␣
 ↪estunepuissance(8924213474370287383311682064643843208788626945360316851637261367232948597517647476633990
n est la puissance 97ième de 198819731987319
Out[43]: True
```

## 3.3 Binaire vs décimal

### 3.3.1 longueurbinaire(n)

**Exercice**

Voici un super programme :

```
def longueurbinaire(n):
    """Calcule la longueur de l'écriture binaire de l'entier n positif.

    >>> longueurbinaire(1000)
    10
    >>> longueurbinaire(10000)
    14
    """
    longueur = 1
    K = 1
    while True:
        m = n >> K
        if m == 0:
            break
        longueur = longueur + K
        K = K << 1
        n = m
    while K > 1:
        while m == 0:
            K = K >> 1
            m = n >> K
        longueur = longueur + K
        n = m
        m = 0
    return longueur
```

Commencez, si possible, à deviner ce qu'il fait. Indication :

```
>>> longueurbinaire(2**3000)
3001
>>> longueurbinaire(2**3000 - 1)
3000
```

Expliquer le fonctionnement, ligne par ligne. À votre avis s'agit-il d'un programme efficace ?

### 3.3.2 longueurdecimale(n)

**Exercice**

En s'inspirant de *longueurbinaire(n)* (page 75), faire un programme qui donne la longueur de l'écriture décimale d'un (grand entier) n.

Puis, comparer l'efficacité sur de très grands entiers avec l'approche via `len(str(n))`.

```
def longueurdecimale(n):
    """Calcule la longueur de l'écriture décimale de l'entier n positif.

    >>> longueurdecimale(0)
    1
    >>> longueurdecimale(764529624089114703099991167782893721749117953623 5)
    49
    >>> longueurdecimale(999999999999999999999999999999999999999999999999)
    49
    """
    longueur = 1
    K = 1
    while True:
        m = n // (10**K)  # retire à n les K chiffres les moins significatifs
        if m == 0:  # signifie que ne reste plus à n qu'au plus K chiffres
            break
        longueur += K
        K <<= 1     # remplace K par 2*K
        n = m
    while K > 1:
        while m == 0:
            K >>= 1
            m = n // (10**K)
        longueur += K
        n = m
        m = 0
    return longueur
```

```
In [116]: x = 3**2000

In [117]: longueurdecimale(x)
Out[117]: 955

In [118]: len(str(x))
Out[118]: 955

In [119]: %timeit longueurdecimale(x)
10000 loops, best of 3: 34.1 µs per loop

In [120]: %timeit len(str(x))
100000 loops, best of 3: 19.5 µs per loop

In [121]: x = 3**10000

In [122]: longueurdecimale(x)
Out[122]: 4772
```

```
In [123]: len(str(x))
Out[123]: 4772

In [124]: %timeit longueurdecimale(x)
1000 loops, best of 3: 326 µs per loop

In [125]: %timeit len(str(x))
1000 loops, best of 3: 436 µs per loop

In [126]: x = 3**100000

In [127]: longueurdecimale(x)
Out[127]: 47713

In [128]: len(str(x))
Out[128]: 47713

In [129]: %timeit longueurdecimale(x)
100 loops, best of 3: 17.9 ms per loop

In [130]: %timeit len(str(x))
10 loops, best of 3: 42.7 ms per loop
```

Pour les entiers avec des milliers de chiffres, *longueurdecimale(n)* (page 76) est plus rapide que `len(str(n))` (mais ce n'est pas le cas pour des entiers de seulement cent ou même mille chiffres).

### 3.3.3 longueurdecimale_lent(n)

Juste pour notre édification, voici aussi avec une méthode plus naïve :

```
def longueurdecimale_lent(n):
    """Calcule (lentement) la longueur de l'écriture décimale de l'entier n

    n est positif, pour n nul, renvoie zéro et pas 1.

    >>> longueurdecimale_lent(0)
    0
    >>> longueurdecimale_lent(764529624089114703099991167782893721749117953623 5)
    49
    >>> longueurdecimale_lent(999999999999999999999999999999999999999999999999)
    49
    """
    longueur = 0
    while n:
        n //= 10
        longueur += 1
    return longueur
```

```
In [142]: longueurdecimale_lent(x)
Out[142]: 47713

In [143]: %timeit longueurdecimale_lent(x)
1 loops, best of 3: 1.51 s per loop
```

C'est donc catastrophique sur ce très grand entier. Néanmoins la méthode naïve est plus rapide que *longueurdecimale(n)* (page 76) pour les petits entiers (jusqu'à cinquante chiffres environ au vu de mes brefs essais) :

```
In [144]: %timeit longueurdecimale_lent(12345678901234567890)
100000 loops, best of 3: 2.68 µs per loop

In [145]: %timeit longueurdecimale(12345678901234567890)
100000 loops, best of 3: 5.11 µs per loop
```

De toute façon il vaut largement mieux utiliser `len(str(n))` tant que `n` n'a au plus que quelques centaines de chiffres.

```
In [146]: %timeit len(str(12345678901234567890))
1000000 loops, best of 3: 296 ns per loop
```

Il y a aussi `len(repr(n))` qui est plus rapide que `len(str(n))`, pour les entiers jusqu'à quelques centaines de chiffres, au delà c'est pareil.

```
In [147]: %timeit len(repr(12345678901234567890))
1000000 loops, best of 3: 236 ns per loop

In [148]: x = 10**100

In [149]: %timeit len(str(x))
1000000 loops, best of 3: 642 ns per loop

In [150]: %timeit len(repr(x))
1000000 loops, best of 3: 573 ns per loop

In [151]: x = 10**500

In [152]: %timeit len(str(x))
100000 loops, best of 3: 6.16 µs per loop

In [153]: %timeit len(repr(x))
100000 loops, best of 3: 6.06 µs per loop

In [154]: x = 10**1000

In [155]: %timeit len(str(x))
10000 loops, best of 3: 21.3 µs per loop

In [156]: %timeit len(repr(x))
10000 loops, best of 3: 21.2 µs per loop
```

### 3.3.4 `longueurdecimalebis(n)`

**Exercice**

Si l'on connaît la longueur en binaire de `x` peut-on en déduire exactement la longueur en décimal ? Quel est le mieux que l'on puisse faire, et comment ?

Comment s'explique le programme ci-dessous ?

```python
def longueurdecimalebis(n):
    """Calcule (mystérieusement) la longueur de l'écriture décimale de n

    n est supposé strictement positif (pour n nul, la procédure renvoie
    bizarrement -1 ...)

    >>> longueurdecimalebis(0)
    -1
    >>> longueurdecimalebis(1)
    1
    >>> longueurdecimalebis(2)
    1
    >>> longueurdecimalebis(76452962408911470309999116778289372174911795362 35)
    49
    >>> longueurdecimalebis(99999999999999999999999999999999999999999999999)
    49

    """
    # longueur = math.floor(0.30102999566398*(n.bit_length()-1))
    # ou, pour éviter de devoir importer le module math:
    longueur = (30102999566398 * (n.bit_length() - 1)) // (10**14)
    # Quasi tout le temps d'exécution est dédié au seul calcul
    # qui suit :
    m = n // (10**longueur)
    # on pourrait s'arrêter ici par
    # return longueur + len(repr(m))
    while m:
        m //= 10
        longueur += 1
    return longueur
```

```
In [33]: x = 3**100000

In [34]: len(str(x))
Out[34]: 47713

In [35]: longueurdecimalebis(x)
Out[35]: 47713

In [36]: %timeit len(str(x))
10 loops, best of 3: 42.6 ms per loop

In [37]: %timeit longueurdecimalebis(x)
100 loops, best of 3: 2.01 ms per loop
```

---

---

*Date de dernière modification* : 11-10-2017 à 08:42:40.

# 4

# Feuille de travaux pratiques 4

*Date de dernière modification* : 22-11-2017 à 09:41:38.



## 4.1 Nombres premiers

Un nombre entier strictement positif n est dit premier si n>1 et si aucun entier 1<m<n ne divise n.

### 4.1.1 `estpremier(n)`

**Exercice**

Faire une procédure `estpremier(n)` qui doit être utilisée avec `n` entier strictement positif et qui renverra `True` si `n` est premier.

On implémentera l'algorithme suivant :

```
si n vaut 1, renvoyer False

si n est pair, renvoyer True seulement si n == 2, sinon False

m = 3

boucle :

   soient q et r le quotient et le reste dans la division euclidienne
   de n par m

   si r == 0 renvoyer True si q == 1, renvoyer False sinon

   si q <= m renvoyer True

   incrémenter m de deux et boucler
```

```python
def estpremier(n):
    """True si n (supposé entier > 0) est premier.

    >>> estpremier(1)
    False
    >>> estpremier(2)
    True
    >>> estpremier(4)
    False
    >>> estpremier(3)
    True
    >>> estpremier(2017)
    True
    >>> estpremier(2019)
    False
    >>> estpremier(2997)
    False
    >>> estpremier(2999)
    True
    """
    if n == 1:
        return False
    if not n & 1:  # n est pair
        return n == 2
    m = 3
    while True:
        q, r = divmod(n, m)
        if r == 0:
            # alors ce m est forcément premier (car n n'est divisible
            # par aucun entier < m), donc n est premier ssi n == m
            return q == 1
        # Ici n n'est divisible par aucun entier <= m
```

```
        if q <= m:
            # alors 1 < n < (m + 1) * m
            # Il ne peut donc pas être divisible par deux nombres premiers
            # chacun > m (sinon n < (m + 1) * m serait faux).
            # Mais on sait que n n'est divisible par aucun entier <= m
            # donc, comme n admet forcément un diviseur premier, c'est
            # que n est premier.
            return True
        m += 2
```

Assurez-vous de comprendre les explications mathématiques !

— pourquoi cela s'arrête-t-il en un nombre fini d'étapes ?

— montrer que le test `q <= m` pourrait être remplacé par `q <= m + 3`.

Mais l'exécution de :

```
>>> estpremier(28937196319327)
True
```

prend pas loin de 1 seconde sur l'ordinateur qui a servi au test. Ce qui est beaucoup !

---

**Exercice**

À votre avis, si la procédure prend 1 seconde pour vérifier qu'un premier de 14 chiffres est premier, combien de temps au minimum pour vérifier qu'un premier de 28 chiffres est premier ?

---

Avec le premier de 14 chiffres, il y a eu moins de $\frac{1}{2}10^7$ itérations de la boucle, tandis qu'avec un premier de 28 chiffres il y en aura au minimum $\frac{1}{2}\sqrt{10^{27}} = \frac{1}{2}\sqrt{10}10^{13} > \frac{1}{2}3.16 \cdot 10^{13}$. Cela prendra donc au minimum $3.16 \cdot 10^6$ plus de temps, et c'est une sous-estimation car les divisions euclidiennes sont plus coûteuses avec le `n` plus grand. Donc, si `n` se trouve être un nombre premier de 28 chiffres, `estpremier(n)` mettra pour le vérifier au minimum `3160000s`, c'est-à-dire *plus de trente-six jours* !

La conclusion est que nous n'avons pas les moyens à ce stade de produire un nombre premier de 28 chiffres : l'idée serait de choisir au hasard un entier avec 28 chiffres (disons impair pour ne pas perdre de temps) et à tester sa primalité jusqu'à en obtenir un qui passe le test. Mais comme la vérification prend un temps prohibitif lorsque l'entier testé est vraiment premier, ça ne fonctionne pas.

---

**Note :** La « probabilité » qu'un nombre impair de 28 chiffres soit premier est aux alentours de $1/16^e$. Donc on est sûr que cette méthode en trouve un et le valide après au pire quelques dizaines de tentatives. La plupart des entiers tirés au hasard auront de petits diviseurs premiers et donc seront éliminés rapidement ; mais ceux ayant des diviseurs premiers de dix chiffres par exemple prendront du temps à être éliminés, et si le nombre est premier la preuve de sa primalité sera longue à obtenir.

---

### 4.1.2 `pluspetitdiviseurpremier(n)`

---

**Exercice**

Faire une procédure `pluspetitdiviseurpremier(n)` qui renvoie le plus petit diviseur premier de `n`.

On adaptera l'algorithme utilisé pour *estpremier(n)* (page 81).

La procédure utilisera `assert` [69] pour engendrer un message d'erreur si `n` n'est pas un entier strictement plus grand que 1.

```
def pluspetitdiviseurpremier(n):
    """Renvoie le plus petit diviseur premier de n.

    >>> pluspetitdiviseurpremier(2017)
    2017
    >>> pluspetitdiviseurpremier(2019)
    3
    >>> pluspetitdiviseurpremier(2021)
    43
    >>> pluspetitdiviseurpremier(2023)
    7
    >>> pluspetitdiviseurpremier(2997)
    3
    >>> pluspetitdiviseurpremier(2999)
    2999
    >>> pluspetitdiviseurpremier(3001)
    3001
    >>> pluspetitdiviseurpremier(3007)
    31
    """
    assert type(n) == int and n > 1,\
        "Erreur: l'argument doit être un entier > 1"
    if not n & 1:  # n est pair
        return 2
    m = 3
    while True:
        q, r = divmod(n, m)
        if r == 0:
            return m
        if q <= m:
            return n
        m += 2
```

### 4.1.3 `listedesdiviseurspremiers_betement(n)`

**Exercice**

Faire une procédure `listedesdiviseurspremiers_betement(n)` qui renvoie la liste des diviseurs premiers (chacun apparaissant autant de fois que sa multiplicité comme diviseur de `n`) de `n`.

On utilisera de manière répétée *pluspetitdiviseurpremier(n)* (page 83).

La procédure utilisera `assert` [70] pour engendrer un message d'erreur si `n` n'est pas un entier stric-

69. https://docs.python.org/3/reference/simple_stmts.html#assert
70. https://docs.python.org/3/reference/simple_stmts.html#assert

tement positif. Pour `n == 1` la liste renvoyée sera vide.

```
def listedesdiviseurspremiers_betement(n):
    """Renvoie la liste des diviseurs premiers de n.

    >>> listedesdiviseurspremiers_betement(1)
    []
    >>> listedesdiviseurspremiers_betement(1000)
    [2, 2, 2, 5, 5, 5]
    >>> listedesdiviseurspremiers_betement(2017)
    [2017]
    >>> listedesdiviseurspremiers_betement(2019)
    [3, 673]
    >>> listedesdiviseurspremiers_betement(2021)
    [43, 47]
    >>> listedesdiviseurspremiers_betement(2023)
    [7, 17, 17]
    >>> listedesdiviseurspremiers_betement(2997)
    [3, 3, 3, 3, 37]
    >>> listedesdiviseurspremiers_betement(2999)
    [2999]
    >>> listedesdiviseurspremiers_betement(3001)
    [3001]
    >>> listedesdiviseurspremiers_betement(3007)
    [31, 97]
    """
    assert type(n) == int and n > 0,\
        "Erreur: l'argument doit être un entier > 0"
    L = []
    while n > 1:
        p = pluspetitdiviseurpremier(n)
        L.append(p)
        n //= p
    return L
```

### 4.1.4 `listedesdiviseurspremiers(n)`

Critique de l'algorithme précédent (et donc de son implémentation):

- il est un peu bête d'appeler de manière répétée *pluspetitdiviseurpremier(n)* (page 83) ce dernier faisant à chaque fois un `assert`[71] inutile.
- mais de très loin le plus grave, c'est que *pluspetitdiviseurpremier(n)* (page 83) recommence à chaque fois « du début », alors qu'en réalité au bout d'un certain moment on sait que ce qui reste de notre entier n'est pas divisible par les « petits nombres premiers ». C'est vraiment idiot ça.

**Exercice**

Faire à nouveau une procédure `listedesdiviseurspremiers(n)` qui renvoie la liste des diviseurs premiers (énumérés autant de fois que leurs multiplicités comme diviseurs) de `n`.

La procédure utilisera (une seule fois) `assert`[72] pour engendrer un message d'erreur si `n` n'est pas un entier strictement positif. Pour `n == 1` la liste renvoyée sera vide.

71. https://docs.python.org/3/reference/simple_stmts.html#assert
72. https://docs.python.org/3/reference/simple_stmts.html#assert

Pour l'implémentation on essaiera de faire moins bêtement que la fois précédente.

```
def listedesdiviseurspremiers(n):
    """Renvoie la liste des diviseurs premiers de n.

    >>> listedesdiviseurspremiers(1)
    []
    >>> listedesdiviseurspremiers(1000)
    [2, 2, 2, 5, 5, 5]
    >>> listedesdiviseurspremiers(2017)
    [2017]
    >>> listedesdiviseurspremiers(2019)
    [3, 673]
    >>> listedesdiviseurspremiers(2021)
    [43, 47]
    >>> listedesdiviseurspremiers(2023)
    [7, 17, 17]
    >>> listedesdiviseurspremiers(2997)
    [3, 3, 3, 3, 37]
    >>> listedesdiviseurspremiers(2999)
    [2999]
    >>> listedesdiviseurspremiers(3001)
    [3001]
    >>> listedesdiviseurspremiers(3007)
    [31, 97]
    """
    assert type(n) == int and n > 0,\
        "Erreur: l'argument doit être un entier > 0"
    L = []
    # d'abord, se débarrasser des puissances de 2
    while not n & 1:
        L.append(2)
        n >>= 1
    p = 3
    q = 4  # voir les explications plus bas pour ce choix bizarre
    while q > p:
        q, r = divmod(n, p)
        while not r:  # tant que p divise n
            L.append(p)
            n = q
            q, r = divmod(n, p)
        # si p divisait n on l'a retiré de n autant de fois que possible
        # donc on boucle, mais quand faut-il s'arrêter ?
        # le n (impair) qui reste n'est divisible par aucun nombre premier <= p
        # si n n'est pas premier il est au moins (p+2)**2 = p**2 + 4*p + 4
        # donc le dernier q calculé est au moins p+4
        # si le dernier q calculé est < p+4, c'est que n est premier
        # d'où le critère d'arrêt q < p + 4, mais comme on remplace
        # p par p+2, ça devient q < p + 2, dont le contraire (critère de
        # prolongement de la boucle) est q > p + 1 (ce sont des entiers)
        # pour éviter un +1 on va juste utiliser while q > p
        # il faut initialiser q avant la boucle, on le fait arbitrairement
        # car de toute façon il est recalculé immédiatement
        p += 2
    # si on sort de la boucle avec un n encore >1, on sait qu'il est premier
    if n > 1:
        L.append(n)
```

```
    return L
```

```
In [2]: x = (2*3*5*11*37)**5*(7919*17389)**3*15485863

In [3]: x
Out[3]: 109735863116741490170703390022890635848523084673000000

In [4]: %timeit listedesdiviseurspremiers_betement(x)
15.1 ms ± 6.74 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [5]: %timeit listedesdiviseurspremiers(x)
3.23 ms ± 973 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [6]: L = listedesdiviseurspremiers(x)

In [7]: L
Out[7]:
[2,
 2,
 2,
 2,
 2,
 3,
 3,
 3,
 3,
 3,
 5,
 5,
 5,
 5,
 5,
 11,
 11,
 11,
 11,
 11,
 37,
 37,
 37,
 37,
 37,
 7919,
 7919,
 7919,
 17389,
 17389,
 17389,
 15485863]

In [10]: A = 1

In [11]: for i in L:
    ...:     A *= i
    ...:

In [12]: A
```

```
Out[12]: 109735863116741490170703390022890635848523084673000000

In [13]: x
Out[13]: 109735863116741490170703390022890635848523084673000000
```

## 4.2 Classes inversibles

Soit $n > 1$ un entier. Le nombre des classes de congruence inversibles modulo $n$ est, comme l'on sait :

$$\phi(n) = n \prod_{p|n} (1 - p^{-1}) \,,$$

où le produit porte sur tous les diviseurs premiers $p$ de $n$. Un produit vide vaut par convention 1 donc $\phi(1) = 1$, ce qui est d'ailleurs compatible avec la définition suivante :

$$\phi(n) = \#\{m \mid 1 \leqslant m \leqslant n \text{ et est premier avec } n\}$$

Si vous avez besoin de vous rafraîchir la mémoire, vous pouvez consulter:

<https://fr.wikipedia.org/wiki/Indicatrice_d%27Euler>

Une propriété essentielle, liée au théorème chinois, est :

$$\text{pgcd}(a, b) = 1 \implies \phi(ab) = \phi(a)\phi(b)$$

Si l'on fixe $n > 1$ et qu'on prend au hasard une classe de congruence non nulle la probabilité qu'elle soit inversible est $\phi(n)/(n-1)$. Comme on a une jolie formule ci-dessus pour $\phi(n)/n$, on va plutôt s'intéresser, pour $n$ fixé, à la probabilité qu'une classe de congruence (éventuellement nulle) soit inversible. C'est donc $\phi(n)/n$.

### 4.2.1 `fonctionphi(n)`

**Exercice**

Faire une fonction Python `fonctionphi(n)` qui renvoie $\phi(n)$ en comptabilisant le nombre des classes modulo `n` qui sont inversibles, ou plus concrètement les entiers `1<=x<=n` qui sont premiers avec `n`.

C'est-à-dire on examine si `pgcd(n, x)` vaut 1. On utilisera donc la procédure *pgcd(a, b)* (page 2). Si par exemple, vous l'avez dans un fichier `tp1.py`, il vous suffit d'ajouter :

```
import tp1
```

vers le haut de votre fichier pour cette séance, et alors d'invoquer la fonction par :

```
tp1.pgcd(a, b)
```

Vous pouvez aussi simplement copier-coller votre définition dans le fichier pour cette séance. Ou utiliser le même fichier pour tout ! (pas recommandé).

```
def fonctionphi(n):
    """Indicatrice d'Euler.

    On pré-suppose que l'argument est un entier strictement positif.

    Par exemple phi(1000) vaut 1000*(1-1/2)*(1-1/5) = 400, mais on fait ce
    calcul en connaissant les diviseurs premiers (2 et 5) de 1000.

    La procédure quant à elle calcule par force brute tous les pgcd avec les
    entiers inférieurs pour identifier et compter les classes de congruence
    inversibles.

    >>> fonctionphi(100)
    40
    >>> fonctionphi(360)
    96
    """
    if n == 1:
        return 1
    return sum(1 for x in range(1, n) if tp1.pgcd(n, x) == 1)
```

On pourrait gagner un facteur 2 en efficacité, puisque la classe x est inversible si et seulement si n−x l'est. Il suffit donc d'examiner :

— si n=2m est pair, les classes de 1 à m−1 (m n'est pas inversible puisque 2m vaut zero modulo n, mais 2 ne vaut pas zero modulo n… sauf si n est 2, et dans ce cas m=1 est en effet inversible…)

— si n=2m+1 est impair (>1), les classes de 1 à m,

et de multiplier par deux le nombre de classes inversibles trouvées (SAUF si n vaut 2 ! ou 1 ! toujours se méfier des petits entiers…).

### 4.2.2 frequenceinversibles(m, n)

**Exercice**

Faire une fonction Python `frequenceinversibles(m, n)` qui renvoie la moyenne des $\phi(j)/j$ sur les entiers *impairs* $j$ vérifiant $m \leqslant j < n$.

```
def frequenceinversibles(m, n):
    """Fréquence moyenne des classes inversibles modulo un nombre impair.
    """
    if not m & 1:  # point de départ m est pair
        m += 1     # maintenant il est impair
    return sum(fonctionphi(j)/j for j in range(m, n, 2))/((n - m + 1) >> 1)
```

Mise en œuvre :

```
>>> from tp4 import *
>>> frequenceinversibles(1, 500)
0.8109076289534366
>>> frequenceinversibles(500, 1000)
```

```
0.8103123253387697
>>> frequenceinversibles(1000, 1500)
0.8108729831143751
>>> frequenceinversibles(1500, 2000)
0.8107785585334791
>>> from math import pi
>>> _ * pi**2
8.00206362961091
```

Ainsi, au moins pour ces petits entiers, on conclut la chose suivante :

> Pour un nombre impair n pris au hasard il y a environ 81% de chances (ou plus précisément $8/\pi^2$) qu'un deuxième entier x aléatoire soit premier avec n.

### 4.2.3 probapremiersentreeux(N)

L'affirmation précédente est assez osée car elle repose sur des calculs numériques certes exacts mais n'utilisant que de petits entiers. Et il nous est totalement impossible d'exécuter la fonction *frequenceinversibles(m, n)* (page 89) avec m et n des entiers de disons 30 chiffres.

On se tourne donc maintenant vers une approche utilisant des tirages aléatoires. Vous aurez besoin de :

```
import random
```

pour pouvoir utiliser `random.randrange()` [73].

---

**Exercice**

Faire une procédure `probapremiersentreeux(N, reps=1000)` qui choisit aléatoirement `reps` fois de suite d'abord un entier impair n de N chiffres puis un entier x avec `1 <= x < n` et qui compte le nombre de fois que `pgcd(n, x) == 1`. La procédure renvoie ce décompte divisé par `reps`.

---

```
def probapremiersentreeux(N, reps=1000):
    """Probabilité que pgcd(n, x) = 1

    Avec (n, x) aléatoire, n impair de N chiffres, 1 <= x < n
    """
    A = 10 ** (N-1)
    B = 10 * A
    S = 0
    for i in range(reps):
        # on prend un entier aléatoire impair de N chiffres :
        n = random.randrange(A + 1, B, 2)
        # puis un entier x aléatoire avec 1 <= x < n
        x = random.randrange(1, n)
        if tp1.pgcd(n, x) == 1:
            S += 1
    return S/reps
```

Voici un résultat typique. Vos résultats seront analogues mais probablement pas identiques, car ça reste aléatoire :

73. https://docs.python.org/3/library/random.html#random.randrange

```
>>> probapremiersentreeux(10)
0.8
>>> probapremiersentreeux(10)
0.821
>>> probapremiersentreeux(20)
0.817
>>> probapremiersentreeux(30)
0.799
>>> probapremiersentreeux(40)
0.815
>>> probapremiersentreeux(50)
0.808
>>> probapremiersentreeux(60)
0.8
>>> probapremiersentreeux(70)
0.793
>>> probapremiersentreeux(100, 10000)
0.8134
```

On voit donc que l'heuristique observée avec des entiers de trois ou quatre chiffres vaut aussi avec des entiers de 100 chiffres !

### 4.2.4 `quasipremierprobable(N, reps)`

Considérons maintenant cette stratégie pour trouver un diviseur d'un entier `n` (et donc, récursivement, pour le factoriser entièrement): il s'agit simplement de choisir `1 <= x < n` au hasard, et d'évaluer `pgcd(n, x)`. Si ce dernier est `> 1` on a trouvé un diviseur, sinon on a trouvé une classe inversible `x` modulo `n`.

---

**Note :** L'astuce ici est de ne pas seulement vérifier si `x` divise `n` mais de passer par le calcul de `pgcd(n, x)` qui n'est pas en fait très coûteux (pas beaucoup plus que la seule division euclidienne de `n` par `x`). Pour une faible multiplication de coût, on a augmenté nos chances de trouver un diviseur de `n`.

---

Supposons que 100 fois de suite on ait obtenu `pgcd(n, x) == 1` (et donc échoué à trouver un diviseur de notre entier impair `n`). Si notre `n` (impair) se comporte de manière « normale », cela ne devrait avoir eu qu'environ $0.81^{100} \approx 7 \cdot 10^{-10}$ de chances de se réaliser, d'après ce qui précède.

Soit encore 7 chances sur dix milliards. Si l'on observe vraiment ce phénomène, on est légitimement amené à penser que notre `n` a beaucoup moins de diviseurs que la normale. On peut peut-être espérer qu'il est le produit d'un faible nombre de nombres premiers, voire est carrément lui-même un nombre premier.

---

**Exercice**

Faire une fonction Python `quasipremierprobable(N, reps=100)` qui prend au hasard un nombre impair aléatoire `n` de `N` chiffres, puis `reps` fois un entier `1 <= x < n` et examine si `pgcd(n, x) == 1`. Lorsque cela se produit `reps` fois de suite, la procédure s'arrête et renvoie le `n` trouvé.

---

```
def quasipremierprobable(N, reps=100):
    """Renvoie un nombre de N chiffres impair qui n'a pas beaucoup de diviseurs.
```

```
    """
    A = 10 ** (N-1)
    B = 10 * A
    while True:
        # on prend un entier aléatoire impair de N chiffres :
        n = random.randrange(A + 1, B, 2)
        for i in range(reps):
            x = random.randrange(1, n)
            if tp1.pgcd(n, x) > 1:
                break
        else:
            return n
```

Voici un exemple d'utilisation :

```
>>> quasipremierprobable(10)
6625052437
>>> estpremier(6625052437)
False
>>> listedesdiviseurspremiers(6625052437)
[61, 108607417]
>>> quasipremierprobable(10)
6546790409
>>> listedesdiviseurspremiers(_)
[6607, 990887]
>>> quasipremierprobable(10)
7692027353
>>> listedesdiviseurspremiers(_)
[2081, 3696313]
>>> quasipremierprobable(10)
1136482223
>>> listedesdiviseurspremiers(_)
[4327, 262649]
>>> quasipremierprobable(10)
7974336673
>>> listedesdiviseurspremiers(_)
[167, 1249, 38231]
>>> quasipremierprobable(10)
7804842583
>>> listedesdiviseurspremiers(_)
[24247, 321889]
>>> quasipremierprobable(10)
9086322793
>>> listedesdiviseurspremiers(_)
[239, 38018087]
```

C'est amusant car on semble surtout trouver des entiers qui ont deux ou trois diviseurs premiers, plutôt que d'être eux-mêmes des premiers. Avec un nombre de tests plus élevé on obtient plus souvent de vrais nombres premiers :

```
>>> quasipremierprobable(10, 1000)
7897222121
>>> listedesdiviseurspremiers(_)
[7897222121]
>>> quasipremierprobable(10, 1000)
7171397509
>>> listedesdiviseurspremiers(_)
```

```
[7171397509]
>>> quasipremierprobable(10, 1000)
8927699647
>>> listedesdiviseurspremiers(_)
[17957, 497171]
>>> quasipremierprobable(10, 1000)
8956280033
>>> listedesdiviseurspremiers(_)
[8956280033]
>>> quasipremierprobable(10, 1000)
1897444079
>>> listedesdiviseurspremiers(_)
[1897444079]
```

**Note :** Voici un *thème de recherches* : pour N (le nombre de chiffres), comment choisir reps pour que les entiers fournis par la procédure *quasipremierprobable(N, reps)* (page 91) soient de vrais premiers, avec une probabilité d'erreur de moins de, par exemple, 1 pour mille ?

Pour 14 chiffres, faire 100000 répétitions ne suffit pas encore vraiment semble-t-il :

```
>>> quasipremierprobable(14, 100000)
36315056261947
>>> listedesdiviseurspremiers(_)
[1212331, 29954737]
>>> quasipremierprobable(14, 100000)
68529282327937
>>> listedesdiviseurspremiers(_)
[68529282327937]
>>> quasipremierprobable(14, 100000)
27871821652681
>>> listedesdiviseurspremiers(_)
[27871821652681]
>>> quasipremierprobable(14, 100000)
50933069111789
>>> listedesdiviseurspremiers(_)
[3079537, 16539197]
>>> quasipremierprobable(14, 100000)
94182492124807
>>> listedesdiviseurspremiers(_)
[1056361, 89157487]
>>> quasipremierprobable(14, 100000)
86843310950497
>>> listedesdiviseurspremiers(_)
[86843310950497]
```

Assez souvent on tombe sur des produits de deux facteurs premiers.

On ne peut pas tester avec 30 chiffres, car le temps d'exécution de notre *listedesdiviseurspremiers(n)* (page 85) (ou même de *estpremier(n)* (page 81)) est prohibitif !

### 4.2.5 factorisationnormale(N)

**Exercice**

Pour mettre en contexte les résultats précédents, faire une procédure Python `factorisationnormale(N)` qui choisit au hasard un nombre impair de `N` chiffres et renvoie la liste de ses diviseurs premiers et lui-même.

Tester avec `N` égal à 14.

```
def factorisationnormale(N):
    """Renvoie un nombre aléatoire impair de N chiffres et sa factorisation.
    """
    A = 10**(N-1)
    B = 10*A
    n = random.randrange(A + 1, B, 2)
    return listedesdiviseurspremiers(n), n
```

Voici typiquement ce que l'on obtient :

```
>>> factorisationnormale(14)
([5, 5, 19, 14437, 5774647], 39600074901025)
>>> factorisationnormale(14)
([479027, 55624339], 26645560238153)
>>> factorisationnormale(14)
([11, 37, 81464435567], 33156025275769)
>>> factorisationnormale(14)
([3, 3307859, 8647231], 85811462665287)
>>> factorisationnormale(14)
([71, 2153, 167729117], 25639576011971)
>>> factorisationnormale(14)
([1607, 43726854491], 70269055167037)
>>> factorisationnormale(14)
([19139, 3892412753], 74496887679667)
>>> factorisationnormale(14)
([5, 17, 2647, 9677, 14389], 31328833213235)
>>> factorisationnormale(14)
([3, 151, 145357449437], 65846924594961)
>>> factorisationnormale(14)
([59198296129103], 59198296129103)
>>> factorisationnormale(14)
([11, 59, 457, 509, 66851], 10092217169287)
>>> factorisationnormale(14)
([11, 13, 97, 6218578391], 86257900861561)
>>> factorisationnormale(14)
([6917, 12361071083], 85501528681111)
>>> factorisationnormale(14)
([3, 3, 3, 353, 6270658783], 59765648860773)
>>> factorisationnormale(14)
([61, 40037, 36542071], 89245128694247)
>>> factorisationnormale(14)
([10321, 6890738381], 71119310830301)
>>> factorisationnormale(14)
([5, 7, 185069, 11824493], 76592148325595)
>>> factorisationnormale(14)
([3251, 15823, 1778221], 91472707160633)
>>> factorisationnormale(14)
([3, 3, 5, 829874774147], 37344364836615)
```

On s'aperçoit qu'en fait, typiquement notre entier de 14 chiffres n'a pas beaucoup de diviseurs premiers. Ce qui n'est pas surprenant puisque grosso modo lors d'une multiplication les nombres

de chiffres s'additionne. Donc ici 14 chiffres ce n'est pas grand chose.

On est dans l'impossibilité complète d'explorer numériquement le problème pour des nombres disons de 100 chiffres, car notre algorithme de factorisation est totalement inutilisable pour ce genre de nombres.

Les mathématiciens connaissent cependant par des approches théoriques le comportement moyen du nombre de diviseurs premiers, mais nous n'aborderons pas ces sujets ici.

## 4.3 Témoins et tests (aléatoires) de non-primalité

Considérons le cas d'un entier impair $n$ qui se trouve être le produit $PQ$ de deux grands nombres premiers. Un example est donné par:

$$n = 63605523217059931493 = 8934612551 \times 7119001843$$

```
>>> estpremier(8934612551)
True
>>> estpremier(7119001843)
True
>>> 8934612551 * 7119001843
63605523217059931493
```

Si on ne connaît pas la factorisation, quelle chance a-t-on par la stratégie de la section précédente de la trouver ?

$$\begin{aligned}\frac{\phi(n)}{n} &= (1 - \frac{1}{8934612551})(1 - \frac{1}{7119001843}) \\ &\approx 0.99999999974760...\end{aligned}$$

La probabilité qu'un million de tirages aléatoires d'un `1 <= x < n` donnent à chaque fois une classe inversible est plus grande que

$$(0.99999999974760)^{1000000} \approx 99,975\%$$

Donc, après un million de tentatives on ne sait toujours pas distinguer $n$ d'une véritable nombre premier.

Pour que $(1-\epsilon)^K$ commence à donner quelque chose de significativement inférieur à 1, il faut prendre $K$ de l'ordre de $\epsilon^{-1}$ (justifiez-le).

Donc ici cela veut dire qu'il faut prendre $K$ de l'ordre de

$$\begin{aligned}\left(1 - (1 - \frac{1}{8934612551})(1 - \frac{1}{7119001843})\right)^{-1} &\approx \frac{1}{\frac{1}{8934612551} + \frac{1}{7119001843}} \\ &= \frac{n}{P+Q} \approx \frac{n}{2\sqrt{n}} = \frac{1}{2}\sqrt{n}\end{aligned}$$

Mais ceci signifie qu'il faut faire autant de tentatives que simplement l'approche brutale qui tente de diviser $n$ par tous les impairs inférieurs à $\sqrt{n}$.

Autrement dit, *notre méthode de factorisation est complétement stupide et inefficace sur ce type d'entiers*.

### 4.3.1 Témoins de Fermat

Heureusement, Fermat est passé par là. Il nous dit de calculer $2^{n-1} \bmod n$: si le résultat n'est pas $1 \bmod n$, c'est que $n$ n'est **pas premier**.

```
>>> n = 63605523217059931493
>>> pow(2, n-1, n)
48551466010673702078
```

Tada !

Le résultat ci-dessus **prouve** que 63605523217059931493 n'est PAS un nombre premier. Car Fermat a démontré $a^{p-1} \equiv 1 \bmod p$ pour tout $a$ non divisible par $p$ lorsque $p$ est premier.

Appelons *Témoin de Fermat* (de la non-primalité) de $n$, tout $1 \leqslant x < n$ tel que $x^{n-1} \bmod n$ n'est pas 1.

---

**Exercice**

Montrer que tout `1 <= x < n` qui n'est pas premier avec `n` est automatiquement un témoin de Fermat.

---

Par conséquent si après avoir choisi aléatoirement `1 <= x < n` au lieu de tester si `pgcd(n, x) == 1` on teste si `pow(x, n-1, n) == 1` on trouvera au moins aussi rapidement un résultat négatif et par conséquent une *preuve de non-primalité* de `n` !

---

**Exercice**

En utilisant le Théorème des restes chinois, montrer en toute généralité que si $P$ et $Q$ sont deux nombres premiers distincts, alors le nombre de classes inversibles $x$ dans $\mathbb{Z}/PQ\mathbb{Z}$ avec $x^{PQ-1} = 1$ est $\text{pgcd}(P-1, Q-1)$[2].

Vous aurez besoin de :

1. dans un groupe cyclique (noté multiplicativement) de cardinalité $m$, le nombre de solutions de l'équation $x^a = e$ est $\text{pgcd}(a, m)$.
2. pour tout nombre premier $P$, le groupe multiplicatif $(\mathbb{Z}/P\mathbb{Z})^*$ est cyclique.

Lorsque $\text{pgcd}(P-1, Q-1)$ vaut deux, il y a donc quatre solutions exactement ; les donner explicitement en partant d'une identité de Bezout reliant $P$ et $Q$.

---

**Exercice**

En utilisant les résultats de l'exercice précédent, montrer qu'il n'y a que quatre `1 <= x < n` qui ne sont pas des témoins de Fermat de la non-primalité de `n = 63605523217059931493`.

Cela fait une probabilité de 99.99999999999999993711238% en prenant un `x` au hasard de constater que `pow(x, n-1, n) == 1` est faux et donc de prouver ainsi que notre entier n'est pas premier.

---

**Exercice**

Montrer que pour tout `n=PQ` produit de deux nombres premiers impairs distincts, au moins la moitié des `1 <= x < n` sont des Témoins de Fermat de la non-primalité de `n`.

### 4.3.2 `estTemoinDeFermat(x, n)`

**Exercice**

Faire une fonction `estTemoinDeFermat(x, n)` qui renvoie `True` si `x` est un Témoin de Fermat de la non-primalité de `n`.

La procédure pré-supposera `1 <= x < n`, ou en tout cas que `x` n'est pas `0` ou un multiple de `n` (car de tels `x` donnent toujours `True` mais ne prouvent rien concernant la non-primalité de `n`).

```
def estTemoinDeFermat(x, n):
    """Décide si x (on suppose 1<= x < n) est un Témoin de Fermat pour n.
    """
    return pow(x, n - 1, n) != 1
```

### 4.3.3 Témoins de Miller

Une autre idée de Fermat va permettre une amélioration à la fois théorique et pratique.

Soit $n > 1$. Supposons qu'on puisse trouver deux classes de congruences $X$ et $Y$ distinctes, tels que $n$ divise $X^2 - Y^2 = (X + Y)(X - Y)$. Alors $X + Y$ ne peut pas être inversible modulo $n$ car sinon $X - Y \equiv 0 \bmod n$ ce qu'on a exclu. Donc, si de plus $X \not\equiv -Y \bmod n$ on a un témoin de non-primalité de $n$. Concrètement on calcule le pgcd de $X + Y$ et de $n$.

Prenons le cas particulier $Y \equiv 1 \bmod n$. La situation est donc qu'on a une classe $X$ avec $X^2 = 1 \bmod n$. Si $X$ n'est ni 1 ni $-1$ modulo $n$, alors ni $X + 1$ ni $X - 1$ ne peut être inversible, et donc $\text{pgcd}(X + 1, n)$ et $\text{pgcd}(X - 1, n)$ donne chacun un diviseur non trivial de $n$. Si de plus $n$ est impair ces diviseurs sont même premiers entre eux, car tout facteur commun devra diviser $2 = (X + 1) - (X - 1)$, or il divise déjà $n$ qui est impair.

Considérons donc un entier *impair* $n > 1$ et soit $1 \leqslant x < n$ un entier qui n'est pas un témoin de Fermat, donc tel que $x^{n-1} \equiv 1 \bmod n$. Il y a encore une chance de sauver la situation en observant que $n - 1$ est pair, disons $n - 1 = 2m$, et de calculer $y = x^m \bmod n$. Ainsi $y^2 \equiv 1 \bmod n$. Donc si $y$ n'est ni 1 ni $-1$ modulo $n$, on a prouvé que $n$ est composé. On dit que $x$ est un *témoin de Miller*.

Voici la définition plus complète:

> Tout d'abord on factorise $n - 1$ sous la forme $2^k m$ avec $m$ impair et $k \geqslant 1$.
>
> On dit que $1 \leqslant x < n$ (ou plus généralement tout entier $x$ non multiple de $n$) est un *témoin de Miller* si $x^m$ n'est ni 1 ni $-1$ modulo $n$, et si aucun de ses carrés itérés jusqu'à $z = x^{(n-1)/2} \bmod n$ inclus n'est égal à $-1$ modulo $n$.

**Exercice**

1. Tous les témoins de Fermat sont des témoins de Miller.
2. Si l'on connaît un témoin de Miller qui n'est *pas* un témoin de Fermat, alors on peut trouver un facteur non trivial de l'entier `n`.

Explication: en effet on aura trouvé au passage un `z` modulo `n` qui n'est ni 1 ni –1 et dont le carré est 1. Chacun de `pgcd(n, z+1)` et de `pgcd(n, z-1)` donnera un facteur non trivial de `n`.

3. Si l'entier impair `n` possède un témoin de Miller alors il est un nombre composé.

---

**Important :** Le grand intérêt théorique des témoins de Miller c'est le théorème de Rabin : si $n$ est un entier impair composé alors au moins 75% de ses classes inversibles sont des témoins de Miller.

---

Et comme les classes non-inversibles sont aussi des témoins de Fermat donc de Miller, cela signifie que pour $n$ impair composé donné, la probabilité d'échouer $K$ fois de suite à ce qu'un $1 \leqslant x < n$ aléatoire soit un témoin de Miller est inférieure à $0.25^K$.

Par exemple pour $K = 100$, cela vaut moins de $7 \cdot 10^{-61}$. Or $10^{61}$ est un nombre absolument gigantesque. Une telle probabilité est absurdement faible. Il est totalement improbable qu'un tel évènement arrive jamais…

D'où ce raisonnement fallacieux (euphémisme pour ne pas dire « faux »): si pour un nombre impair $n$, 100 tirages aléatoires d'un $1 \leqslant x < n$ échouent à trouver un témoin de Miller de non-primalité de $n$, alors $n$ est un nombre premier avec une probabilité d'erreur inférieure à $0.25^{100} < 7 \cdot 10^{-61}$.

Ce raisonnement est **faux**, car il manipule les probabilités de manière trop cavalière, en particulier en tenant pas compte de la fréquence des nombres premiers. Mais la conclusion est néanmoins valable. Et la majoration par $0.25^{100}$ est largement sur-évaluée comme nous allons le voir dans la section suivante.

### 4.3.4 Nombres premiers « industriels »

En pratique, un nombre impair qui survit disons 100 fois (et même beaucoup moins que 100 fois) au test de Rabin-Miller est un nombre premier de « qualité industrielle » (expression humoristique de Henri Cohen).

Voici un important résultat théorique :

Soit $p(k)$ la probabilité qu'un nombre impair aléatoire avec $k$ bits soit faussement déclaré premier par un unique test probabiliste de Miller Rabin.

$$p(k) < k^2 4^{2-\sqrt{k}}$$

Référence

> Damgård, I.; Landrock, P. & Pomerance, C. (1993), "Average case error estimates for the strong probable prime test", Mathematics of Computation 61 (203): 177–194, doi:10.2307/2152945

Cette majoration théorique ne commence à devenir inférieure à 1 que pour $k \geqslant 64$, qui correspond a des nombres de 20 chiffres décimaux : en réalité les $p(k)$ sont petites bien avant cela.

Par exemple j'ai pris dix millions de fois au hasard un nombre impair de 20 chiffres binaires (grosso modo $500000 < n < 1000000$ ou plus précisément $524288 \leqslant n < 1048576$), et des tests de Miller Rabin aléatoires uniques en ont déclaré 1477188 premiers, et parmi ceux-ci seulement 838 étaient en fait composés. Cela donne une évaluation $p(20) \approx 0.00057$.

Pour $k = 100$ (ce qui correspond à environ 30 chiffres décimaux) la majoration donnée par la formule ci-dessus est $p(100) < 0.152...$, déjà inférieure à 0.25, mais la vraie valeur est sûrement encore bien inférieure.

Pour $k = 200$ (60 chiffres décimaux), $p(200) < 0.002$, et il n'y a donc d'après la majoration théorique (qui est sûrement largement au-dessus de la réalité) que deux chances sur mille qu'un entier avec 60 chiffres décimaux soit faussement déclaré premier par un *unique* test de Miller-Rabin.

Soit $q(m)$ l'analogue de $p(k)$ pour les nombres impairs aléatoires de $m$ chiffres décimaux. En faisant chauffer mon ordinateur j'ai obtenu expérimentalement la table suivante de valeurs (très très) approchées :

```
m=3  q=0.017444... (valeur exacte)
m=4  q=0.008059... (valeur exacte)
m=5  q=0.0026
m=6  q=0.00081
m=7  q=0.00025
m=8  q=0.000058
m=9  q=0.000017
```

J'ai aussi fait des essais en utilisant un test de Fermat aléatoire plutôt qu'un test de Miller. Les probabilités d'erreurs sont un peu plus élevées, mais elles décroissent aussi avec le nombre de chiffres :

```
m=3  q=0.0597
m=4  q=0.0310
m=5  q=0.0117
m=6  q=0.0039
m=7  q=0.0012
m=8  q=0.00035
m=9  q=0.00010
m=10 q=0.000038
```

C'est largement plus petit que le fameux $1/4 = 0.25$. La conclusion est que d'un point de vue probabiliste le test de Fermat est déjà extrêmement efficace pour repérer les nombres composés.

---

**Note :** Culture : « efficace » ne veut pas dire « parfait ».

Certains entiers composés exceptionnels $561, 1105, 1729, 2465, 2821, 6601, 8911...$, appelés nombres de Carmichael n'ont pas de témoins de Fermat autres que les nombres non-inversibles modulo $n$. La non-primalité de ces entiers ne peut pas être établie par des tests de Fermat.

Et on sait depuis quelques années qu'il y a une infinité de nombres de Carmichael.

C'est aussi pour cette raison qu'on pratique plutôt les tests de Miller, car d'après le théorème de Rabin, tout nombre impair composé admet des témoins de non-primalité par des tests de Miller. Il n'y a donc pas d'analogue des entiers de Carmichael en ce qui concerne les test de Miller.

---

### 4.3.5 `estTemoinDeMiller(x, n)`

---

**Exercice**

Faire une procédure `estTemoinDeMiller(x, n)` qui renvoie `True` si `x` est un témoin de Miller pour l'entier impair `n` et sinon `False`.

En particulier, la procédure renvoie `False` si `x` est nul ou un multiple de `n`.

```
def estTemoinDeMiller(x, n):
    """Renvoie True si x est un témoin de Miller de l'entier impair n.
    """
    # on réduit x modulo n et on vérifie que ce n'est pas la
    # classe nulle
    x = x % n
    if x == 0:
        return False
    # d'abord on calcule k et m, de sorte que n - 1 = (2**k)*m avec m impair
    k = 1
    m = (n - 1) >> 1
    while not m & 1:
        # tant que m est pair le diviser par deux et incrémenter k
        k += 1
        m >>= 1
    # ensuite on calcule y initialement égal à x modulo n à la puissance m
    y = pow(x, m, n)
    if y == 1 or y == n - 1:
        # x n'est pas un témoin de Miller
        return False
    for j in range(k - 1):
        y = (y * y) % n
        # si y vaut 1, ses carrés itérés vaudront tous 1, jamais -1 modulo n
        # donc on a un témoin de Miller.
        if y == 1:
            return True
        # si y vaut -1 modulo n, on n'a pas un témoin de Miller
        if y == n - 1:
            return False
    # si on arrive ici c'est avec y valant x à la puissance 2**(k-1) * m
    # donc y au carré modulo n vaut x à la puissance n - 1 (modulo n)
    #
    # - si y au carré n'est pas 1 modulo n on a un Témoin de Fermat donc Miller
    # - si y au carré est 1 modulo n, comme y n'est ni 1 ni -1 on a un Témoin
    #   de Miller
    #
    # dans les deux cas si on arrive ici, c'est que x est un Témoin de Miller.
    return True
```

### 4.3.6 `testMillerRabin(n, reps)`

Il est temps de passer aux choses sérieuses.

**Exercice**

Faire `testMillerRabin(n, reps=10)` qui essaie `reps` fois (défaut `10`) un nombre aléatoire compris entre 2 et $n-2$ pour voir si un témoin de Miller a été trouvé. Si oui, $n$ est composé. Si non, $n$ est peut-être premier.

La procédure utilisera `estTemoinDeMillerBis(x, n, m, k)` qui est comme *estTemoinDeMiller(x,*

*n)* (page 99) à cette différence que `m` et `k` sont déjà précalculés (au lieu de les re-calculer à chaque nouvel `x`).

```
def estTemoinDeMillerBis(x, n, m, k):
    """Renvoie True si x est un témoin de Miller de l'entier impair n.

    Cette routine est appelée avec m et k déjà calculés de sorte que
    n = 1 + (2**k)*m. De plus 1 <= x < n.
    """
    y = pow(x, m, n)
    if y == 1 or y == n - 1:
        return False
    for j in range(k - 1):
        y = (y * y) % n
        if y == 1:
            return True
        if y == n - 1:
            return False
    return True
```

```
def testMillerRabin(n, reps=10):
    """Test de Miller-Rabin "probabiliste".

    Renvoie True ou False selon que n est peut-être premier
    ou sûrement composé:

    - Si la procédure renvoie False, il est CERTAIN que n est composé.

    - Si elle renvoie True, il est PROBABLE que n est premier.
      La « probabilité d'erreur » est majorée par :math:`1/4^{reps}`.

    Avec reps==10, :math:`4^(-10) < 10^{-6}`
    Avec reps==30, :math:`4^{-30} < 10^{-18}`

    L'estimation avec le 1/4 est très largement surestimée, dès que n a plus
    que quelques chiffres. Si n a des dizaines de chiffres un seul test
    positif est déjà un très fort argument en faveur de la primalité de n.
    """
    if n < 10:
        return n == 2 or n == 3 or n == 5 or n == 7
    if not n & 1:
        return False
    # on calcule k et m, de sorte que n - 1 = (2**k)*m avec m impair
    k = 1
    m = (n - 1) >> 1
    while not m & 1:
        # tant que m est pair le diviser par deux et incrémenter k
        k += 1
        m >>= 1
    # on va faire reps fois la recherche d'un Témoin de Miller peut-être ça
    # serait mieux de s'assurer qu'on ne teste pas deux fois le même x. Mais
    # plus n est grand moins les répétitions de x sont probables.
    for i in range(reps):
        if estTemoinDeMillerBis(random.randrange(2, n - 1), n, m, k):
            # on est SÛR(E) que n n'est PAS premier
            return False
```

```
    # si on arrive ici on a cherché reps fois à piocher un témoin
    # de Miller et on n'en a trouvé aucun. On prend donc le risque
    # de dire que n est premier.
    return True
```

## 4.4 Tests de primalité déterministes

### 4.4.1 estpetitpremier(n)

Avec les super-calculateurs on peut faire des recherches exhaustives sur les nombres premiers disons jusqu'à vingt chiffres. Et on a obtenu numériquement le résultat suivant: tout entier impair $> 1$ qui a au plus quatorze chiffres en décimal est premier si et seulement si ni 2, ni 3, ni 5, ni 7, ni 11, ni 13, ni 17 ne sont des témoins de Miller de $n$.

Il y a d'autres résultats de ce genre. Voir :

<http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html>
<http://mathworld.wolfram.com/StrongPseudoprime.html>

---

**Exercice**

Faire `estpetitpremier(n)` qui détermine avec certitude si un entier d'au plus quatorze chiffres en décimal est un nombre premier.

---

```
def estpetitpremier(n):
    """Renvoie True si l'entier n < 10^14 est premier.

    ATTENTION: n doit donc avoir au plus 14 chiffres, plus
    précisément n < 341 550 071 728 321

    L'algorithme consiste à vérifier qu'aucun de 2, 3, 5, 7, 11, 13, et 17
    n'est un temoin de Miller pour n. Le fait qu'il fonctionne a été établi
    par une combinaison de considérations théoriques et de calculs exhaustifs.

    Références:

    <http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html>

    <http://mathworld.wolfram.com/StrongPseudoprime.html>
    """
    if n < 30:
        return n in {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
    if not n & 1:
        return False
    k = 1
    m = n >> 1
    while not m & 1:  # tant que m est pair, le diviser par 2
        k += 1
        m >>= 1
    # le "and" de Python arrête l'évaluation Booléenne dès qu'il trouve
    # un truc faux. Par exemple si 2 est un témoin de Miller,
    # la première valeur logique est False, donc l'évaluation s'arrête
    # de suite (évaluation « short-circuit »).
```

```
    return not estTemoinDeMillerBis(2, n, m, k)\
        and not estTemoinDeMillerBis(3, n, m, k)\
        and not estTemoinDeMillerBis(5, n, m, k)\
        and not estTemoinDeMillerBis(7, n, m, k)\
        and not estTemoinDeMillerBis(11, n, m, k)\
        and not estTemoinDeMillerBis(13, n, m, k)\
        and not estTemoinDeMillerBis(17, n, m, k)
```

### 4.4.2 Test de Miller-Rabin-Bach

Bach a démontré la belle chose suivante:

**Important : Théorème de Bach**

Si l'hypothèse de Riemann généralisée [74] est vraie alors tout entier impair composé $n$ possède un témoin de Miller qui est $\leqslant 2\log(n)^2$.

**Note :** Le log est bien sûr le log népérien. Mais le majorant $2\log(n)^2$ peut se remplacer par simplement $k^2$ avec $k$ le nombre de chiffres de l'écriture binaire de $n$. Justifiez cette affirmation.

### 4.4.3 `estpremierparMRB(n)`

**Exercice**

Faire `estpremierparMRB(n)` qui détermine avec certitude (modulo Riemann généralisé !) si un entier est premier.

```
def estpremierparMRB(n):
    """Test de Miller-Rabin de primalité en version "déterministe".

    Détermine en temps polynomial si n est premier ou non.

    ATTENTION: la preuve que cet algorithme est correct dépend d'un théorème
    de Bach qui est conditionnel à l'hypothèse de Riemann généralisée, qui
    n'est pas encore un théorème mathématiquement établi.
    """
    if n <= 30:
        return n in {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
    if not n & 1:     # n est pair
        return False  # car n > 2

    # ici n est impair et on commence par calculer k et m tels que
    # n - 1 = (2**k) * m avec m impair
    k = 1
    m = n >> 1        # égal à (n-1)/2
    while not m & 1:
```

74. https://fr.wikipedia.org/wiki/Hypoth%C3%A8se_de_Riemann_g%C3%A9n%C3%A9ralis%C3%A9e

```
        # tant que m est pair le diviser par deux et incrémenter k
        k += 1
        m >>= 1

    # On cherche un témoin de Miller-Rabin <= 2(log(n)**2)
    #    (avec bien sûr log = logarithme népérien)
    # Si aucun n'est trouvé, c'est que n est un nombre premier

    # numériquement :math:`2(log(2)^2) = 0.9609... < 1`
    # donc on peut prendre simplement sans se compliquer la vie :

    X = n.bit_length()**2

    # en effet avec k = n.bit_length(), n < 2*k, donc log(n) < k log(2)
    # et par conséquent 2 * log(n)**2 < 2*log(2)**2 * k**2 < k**2

    if X > n-1:
        X = n-1

    for x in range(2, X):
        if estTemoinDeMillerBis(x, n, m, k):
            return False

    # Aucun Témoin de Miller de non-primalité trouvé,
    # on peut donc affirmer que n est un nombre premier.
    # ... si l'Hypothèse de Riemann généralisée est vraie ...
    return True
```

### 4.4.4 pseudopremieraleatoire(m, reps)

**Exercice**

Faire `pseudopremieraleatoire(m, reps=1)` qui choisit aléatoirement un nombre impair de `m` chiffres jusqu'à en trouver un qui passe `reps` fois un test de Miller aléatoire.

La valeur par défaut de `reps` est 1 à cause des remarques qui sont faites plus bas.

```
def pseudopremieraleatoire(m, reps=1):
    """Trouve un nombre Miller-pseudo-premier p de m chiffres.

    reps est le nombre de tests de Miller. Par défaut un seul test!
    Pour des nombres de dizaines de chiffres, cela semble suffisant...
    """
    # On calcule une fois pour toutes les bornes :
    A = 10 ** (m - 1)
    B = 10 * A
    A = A + 1
    while True:
        p = random.randrange(A, B, 2)
        if testMillerRabin(p, reps):
            return p
```

### 4.4.5 premieraleatoire(m)

**Exercice**

Faire `premieraleatoire(m)` qui renvoie un *vrai* nombre premier de `m` chiffres choisi de la manière suivante :

- pour `m < 15` on tire au hasard un nombre impair de `m` chiffres jusqu'à en trouver un qui soit validé par *estpetitpremier(n)* (page 102).
- pour `m >= 15` on tire au hasard un nombre impair de `m` chiffres jusqu'à en trouver un qui survive successivement à un unique test aléatoire de Miller `testMillerRabin(n)` (*testMillerRabin(n, reps)* (page 100)), puis au test déterministe de Miller-Rabin-Bach *estpremierparMRB(n)* (page 103).

```
def premieraleatoire(m):
    """Trouve un nombre aléatoire (vraiment) premier de m chiffres décimaux.

    Si m est au plus 14 utilise le test de primalité rapide qui dit que n est
    premier si aucun de 2, 3, 5, 7, 11, 13, 17 n'est un témoin de Miller.

    Si m >= 15 utilise d'abord un test de Miller Rabin unique puis un test de
    Miller Rabin Bach pour sélectionner un candidat.

    Dans la pratique la probabilité que le test de Miller-Rabin-Bach ne
    confirme pas la primalité après le test (unique) de Miller-Rabin semble
    très faible.
    """
    A = 10 ** (m - 1)
    B = 10 * A
    A = A + 1
    if m < 15:
        while True:
            n = random.randrange(A, B, 2)
            if estpetitpremier(n):
                return n
    else:
        while True:
            n = random.randrange(A, B, 2)
            x = random.randrange(2, n-1)
            if not estTemoinDeMiller(x, n):
                print("Premier probable :", n)
                if estpremierparMRB(n):
                    print("OK, vraiment premier")
                    return n
                else:
                    print("(non, en fait n'était pas premier)")
```

**Note :** La probabilité qu'un nombre qui n'a pas été prouvé non-premier par un test de Miller-Rabin aléatoire soit ensute prouvé non-premier par le test déterministe de Miller-Rabin-Bach semble extrêmement faible et diminue avec le nombre de chiffres.

Dans mes tests :

- pour des nombres de 6 chiffres, il faut aux alentours de 20000 nombres impairs passant un unique test de Miller-Rabin pour trouver parmi eux 20 nombres qui ne sont pas de

vrais nombres premiers,
- pour des nombres de 8 chiffres, j'ai typiquement dû accumuler plus de 250000 nombres premiers probables (au sens d'un unique test de Miller-Rabin) pour en avoir 20 de faux parmi eux,
- pour des nombres de 10 chiffres, il a fallu dans mon dernier essai accumuler environ 3500000 nombres premiers probables (au sens d'un unique test de Miller-Rabin) pour en avoir 20 de faux parmi eux. Mais cela fait chauffer mon ordinateur qui déclenche son ventilateur et prend une quinzaine de minutes au moins pour examiner environ 40 millions de nombres impairs aléatoires de 10 chiffres, dont environ 3,6 millions survivent à un unique test de Miller-Rabin et parmi eux seulement 20 (par exemple, `5938710121`) n'étaient pas de vrais nombres premiers.

Il faut prendre ces résultats avec des pincettes, car il y a une énorme volatilité.

Je ne peux pas pour le moment exécuter ces tests avec des nombres de 15 chiffres [75] ou plus car je n'ai toujours pas implémenté dans ma feuille Python pour ces TPs de test de primalité suffisamment rapide, mais il est clair que plus les nombres testés ont de chiffres, plus le test aléatoire de Miller-Rabin est digne de confiance pour repérer les nombres premiers. Ceci est bien conforme à l'évaluation théorique de Damgård, I.; Landrock, P. et Pomerance, C. citée plus haut, $p(k) < k^2 4^{2-\sqrt{k}}$ mais je ne connais pas la vraie loi de décroissance du $p(k)$.

---

Vous pouvez maintenant aisément construire de grands nombres composés de plusieurs dizaines de chiffres avec de grands facteurs premiers, mais gare à ne pas égarer ces facteurs premiers !

---

*Date de dernière modification* : `22-11-2017 à 09:41:38.`

75. C'est pure fainéantise car il est connu que si le nombre entier impair `n < 3_825_123_056_546_413_051` (en particulier si `n` a au plus 18 chiffres décimaux) est tel que ni 2, ni 3, ni 5, ni 7, ni 11, ni 13, ni 17, ni 19, et ni 23 n'est un témoin de Miller de la non-primalité de `n` alors `n` est vraiment un nombre premier.

Donc je pourrais facilement étendre *estpetitpremier(n)* (page 102) pour obtenir un test de primalité rapide pour les entiers d'au plus 18 chiffres décimaux.

# 5

# Feuille de travaux pratiques 5

*Date de dernière modification* : 29-11-2017 à 19:12:30.



## 5.1 Avertissement

Je rappelle l'importance de s'être familiarisé(e) avec la documentation officielle [76] de `Python3`.

Nous allons travailler avec des objets de type `list` [77]. Dans la liste des méthodes associées [78] du tutoriel on trouve `list.sort()` [79].

Pour nous faire mal, nous allons continuer à implémenter par nous-mêmes différents algorithmes de tris, à savoir le tri par fusion et le tri par tas (pour le tri par insertion et le tri « rapide » (Quick Sort) cf. *Feuille de travaux pratiques 2* (page 33)). Nos routines retourneront une copie triée.

Il peut, suivant l'algorithme, être plus ou moins simple de faire en sorte que l'algorithme travaille « sur place », c'est-à-dire sans même avoir besoin de faire une copie (superficielle, voir l'aparté qui suit) de la liste `L`. Pour le tri par tas, tel qu'implémenté ci-dessous, ce serait immédiat, car il suffit de travailler directement sur la liste donnée en argument.

76. https://docs.python.org/fr/3/tutorial/introduction.html
77. https://docs.python.org/3/library/stdtypes.html#list
78. https://docs.python.org/3/tutorial/datastructures.html#more-on-lists
79. https://docs.python.org/3/library/stdtypes.html#list.sort

## 5.2 Aparté sur les différentes façons de copier une liste en Python

Considérons ceci :

```
>>> L = [1000, -5, 'babar']
>>> M = L
>>> M[0] = 'x'
>>> L
['x', -5, 'babar']
```

Vous voyez que la modification de `M` a induit une modification de `L`. C'est parce que l'affectation `M = L` n'a pas fait une copie de la liste, mais une copie de la *référence* à la liste. En mémoire il y a un seul exemplaire de la donnée qui stocke `[1000, -5, 'babar']`.

Une autre illustration de ce phénomène :

```
>>> def foo(L):
...     L[0] = 'y'
...
>>> foo(L)
>>> L
['y', -5, 'babar']
```

L'appel de procédure a modifié notre `L` d'origine, ce qui peut surprendre si l'on compare avec :

```
>>> def bar(x):
...     x = x + 1
...
>>> x = 5
>>> bar(x)
>>> x
5
```

Ici le `x` d'origine n'a pas bougé. Incohérent ? non, logique. L'objet « 5 » n'est pas modifiable. Le `x = x + 1` crée un *nouvel* objet « 6 », et fait de `x` une référence à cet objet. Mais à la sortie de la procédure, le `x` redevient l'ancienne référence. Dans le cas de `L` l'affectation `L[0] = 'y'` modifie l'objet lui-même vers lequel le `L` local pointe. On dit que le type `list` [80] est (en anglais) *mutable*.

Bref, comment faire un `foo(L)` qui se comporte comme le `bar(x)`. Bien sûr, ceci échoue :

```
>>> def foo(L):
...     M = L
...     M[0] = 'z'
...
>>> foo(L)
>>> L
['z', -5, 'babar']
```

Il faut faire ceci :

```
>>> def foo(L):
...     M = L.copy()
...     M[0] = 'nouveau'
...     return M
...
```

80. https://docs.python.org/3/library/stdtypes.html#list

```
>>> L
['z', -5, 'babar']
>>> foo(L)
['nouveau', -5, 'babar']
>>> L
['z', -5, 'babar']
```

On voit que la liste `L` n'a pas été affectée. En Python3, `L.copy()` est la syntaxe recommandée. On peut aussi utiliser `M = L[:]` et c'est ce que je fais dans mes codes, par habitude. Pour Python2, il fallait `M = L[:]`.

Attention cependant à certaines subtilités : cette copie reste une copie dite « superficielle ». On va le voir en considérant un `L` avec des items eux-mêmes « mutables ».

```
>>> def foo(L):
...     M = L.copy()
...     if type(M[0]) == list:
...         M[0][0] = 'tada'
...     else:
...         M[0] = 'tutu'
...
>>> L = [[0, 1, 2], 7, 'fleur']
>>> foo(L)
>>> L
[['tada', 1, 2], 7, 'fleur']
>>> L[0] = 1
>>> L
[1, 7, 'fleur']
>>> foo(L)
>>> L
[1, 7, 'fleur']
```

Vous voyez que dans le premier cas, le contenu de la liste `L` a été modifié, pas dans le second. Comment le comprendre ? Dans les deux cas, il y a eu une copie de toutes les références vers les items de `L`. Mais ensuite si le premier item est lui aussi de type liste le code a fait `M[0][0] = 'tada'`, ce qui signifie « remplace en première position de `M[0]` la référence par une référence vers l'objet `'tada'` ». Mais même si `M[0]` est bien une copie de `L[0]` c'est une copie qui pointe vers le même objet de class `list` [81]. Donc la modification de `M[0][0]` se voit sur l'original. Alors que si l'on avait fait :

```
>>> def foo(L):
...     M = L.copy()
...     M[0] = 'tutu'
...
>>> L = [[0, 1, 2], 7, 'fleur']
>>> foo(L)
>>> L
[[0, 1, 2], 7, 'fleur']
```

on n'aurait pas modifié le `L`. Car le `M[0] = 'tutu'` remplace dans `M` le premier item par une référence à `'tutu'`, mais le premier item de `M` n'est pas le premier item de `L`, mais une copie de ce premier item, donc la modification de `M[0]` n'impacte pas le `L` d'origine.

En effet `M[0]` est initialement une copie de `L[0]` donc modifier la copie ce n'est pas modifier l'ori-

81. https://docs.python.org/3/library/stdtypes.html#list

ginal. Par contre si l'on manipule `M[0][0]` on ne fait pas une *modification* de `M[0]`, mais une *utilisation*, et c'est toute la différence car les deux copies `M[0]` et `L[0]` font référence au même objet et par leur intermédiaire on peut modifier cet objet s'il est « mutable ».

On voit donc qu'on va devoir faire des copies plus profondes, par exemple comme ceci :

```
>>> def foo(L):
...     M = L.copy()
...     for i in range(len(M)):
...         if type(M[i]) == list:
...             M[i] = M[i].copy()
...             M[i][0] = 'TADA'
...     print('M est ', M)
...
>>> L
[[0, 1, 2], 7, 'fleur']
>>> foo(L)
M est  [['TADA', 1, 2], 7, 'fleur']
>>> L
[[0, 1, 2], 7, 'fleur']
```

En conclusion, si l'on veut vraiment préserver un `L` de classe `list` [82] dont les items peuvent aussi être « mutables », un `M = L.copy()` ou `M = L[:]` ne suffit pas. Python fournit dans sa librairie standard la fonction `copy.deepcopy()` [83].

Dans nos algorithmes de tris nous n'avons aucune raison de faire des choses comme `M[0][0] = ...`. Autrement dit, tout cet aparté n'a fait que vous faire perdre du temps, mais rappelez vous de ne pas faire `M = L` mais bien `M = L[:]` ou `M = L.copy()`, si vous avez besoin par exemple de permuter des choses dans `M`, ou de les remplacer par d'autres objets, sans que cela n'affecte le `L` originel.

## 5.3 Tri par fusion

### 5.3.1 `fusionne(L, n, i, j)`

**Exercice**

On suppose donnée une liste `L` avec `n` entrées (donc d'indices allant de `0` à `n-1`), qui sont des objets deux à deux comparables par `<`.

On implémentera l'algorithme suivant :

```
fusionne(L, n, i, j)

En entrée une liste L, sa longueur n, et deux indices i<j.

Si i ou si j est au moins n, ne rien faire.

Sinon la routine suppose qu'il est vrai que les objets d'indices
i à j-1 (il y en a donc j-i) sont ordonnés ainsi que ceux de j à
min(n-1, j+(j-i)-1) (il y en a au plus j-i).
```

82. https://docs.python.org/3/library/stdtypes.html#list
83. https://docs.python.org/3/library/copy.html#copy.deepcopy

```
La routine doit réordonner les entrées d'indices i à
min(n, j+(j-i)-1). C'est ce qu'on appelle une fusion.
```

La routine demandée n'est donc pas une fusion générale, mais restreinte à ce qui sera suffisant pour rédiger par la suite une procédure de tri par fusion : la deuxième plage de valeurs d'indices est de même longueur que la première plage de valeurs d'indices (dans la mesure où cela ne la ferait pas déborder au-delà de la fin de `L`) et est positionnée dans sa prolongation directe. Pour l'algorithme de tri par fusion, la première plage d'indices aura une longueur égale à une puissance de deux mais nous n'utiliserons pas cette information supplémentaire.

```
def fusionne(L, n, i, j):
    """Fusionne deux sous-listes consécutives déjà triées.

    Hypothèses :

    1. La première sous-liste commence à l'indice i et termine à j-1

    2. La deuxième sous-liste commence à l'indice j et a la même
       longueur que la première, sauf si cela la ferait "déborder"
       au-delà de n.

    3. Le paramètre n est la longueur de L.

    4. La routine suppose que i et j sont positifs mais doit traiter
       silencieusement les cas avec i ou j au-delà de n.

    Ce code est sûrement grandement améliorable.

    >>> L = [7, 9, 3, 8, 14, 5, 10, 12, 1]
    >>> fusionne(L, 9, 2, 5)
    >>> L
    [7, 9, 3, 5, 8, 10, 12, 14, 1]
    >>> fusionne(L, 9, 5, 8)
    >>> L
    [7, 9, 3, 5, 8, 1, 10, 12, 14]
    >>> fusionne(L, 9, 2, 5)
    >>> L
    [7, 9, 1, 3, 5, 8, 10, 12, 14]
    >>> fusionne(L, 9, 2, 10)
    >>> L
    [7, 9, 1, 3, 5, 8, 10, 12, 14]
    """
    if i >= n:
        return None
    if j >= n:
        return None
    k, l = i, j
    m = j + (j - i)
    if m > n:
        m = n
    x = L[k]
    y = L[l]
    M = []
    while True:
```

```
        if y < x:
            M.append(y)
            l = l + 1
            if l == m:
                break
            y = L[l]
        else:
            M.append(x)
            k = k + 1
            if k == j:
                break
            x = L[k]
    if l == m:
        # la seconde sous-liste entièrement déjà dans M
        # on doit déplacer ce qui reste de la première
        # je n'ai pas osé : L[n-(j-k):]=L[k:j]
        # car j'ai eu peur d'un chevauchement d'indices
        # donc je fais
        M.extend(L[k:j])  # ce qui reste de la première sous-liste
        # ancien code, moins Pythonesque :
        #    while True:
        #        M.append(x)
        #        k = k + 1
        #        if k == j:
        #            break
        #        x = L[k]
    # Maintenant il faut copier en bloc tout M vers L, à partir
    # de son indice i
    L[i:l] = M[:]  # il est possible que l < m
    # ancien code, moins Pythonesque
    # q  =  len(M)
    # while l > i:
    #     l, q  =  l - 1, q - 1
    #     L[l] =  M[q]
    return None
```

La solution proposée dans le corrigé me semble sous-optimale en ce qui concerne le nombre d'affectations : notions `p` le nombre d'éléments de la première plage d'indice, et `q` celui de la seconde (par hypothèse `q<=p`). on commence par une `M` vide et dans le pire des cas on aura fait `p+q` créations d'éléments additionnels à `M`, puis à la fin le transfert vers `L`, qui est fait en bloc, mais comptons donc `2p+2q` affectations. Si on commençait par d'abord transférer dans `M` tous les éléments de la première plage, puis ensuite par remplir `L` à partir de l'indice `i`, on ferait au pire `p+(p+q)=2p+q` affectations. Pour cela il faut trois indices : un indice pour dire où l'on en est dans `M`, un indice pour dire où l'on en est dans la deuxième plage d'indice, et un troisième indice pour dire où va la nouvelle valeur dans `L` (le troisième indice peut être calculé connaissant les deux autres, mais c'est plus simple de le gérer séparément).

En y pensant un peu plus si on procédait en positionnant d'abord le plus grand (et non pas le plus petit) élément, il faudrait d'abord copier dans `M` les éléments de la seconde plage d'indices pour faire de la place, au final on ferait au pire `q+(p+q)=p+2q` affectations. Comme `q<=p` c'est mieux.

---

**Exercice**

Faites `fusionneB(L, n, i, j)` qui implémente la méthode décrite à la fin du paragraphe précé-

dent.

```python
def fusionneB(L, n, i, j):
    """Fusionne deux sous-listes consécutives déjà triées.

    On suppose que la seconde sous-liste a la même longueur que la
    première, sauf si elle déborde au-delà de la fin de L.

    >>> L = [7, 9, 3, 8, 14, 5, 10, 12, 1]
    >>> fusionneB(L, 9, 2, 5)
    >>> L
    [7, 9, 3, 5, 8, 10, 12, 14, 1]
    >>> fusionneB(L, 9, 5, 8)
    >>> L
    [7, 9, 3, 5, 8, 1, 10, 12, 14]
    >>> fusionneB(L, 9, 2, 5)
    >>> L
    [7, 9, 1, 3, 5, 8, 10, 12, 14]
    >>> fusionneB(L, 9, 2, 10)
    >>> L
    [7, 9, 1, 3, 5, 8, 10, 12, 14]
    """
    if i >= n:
        return None
    if j >= n:
        return None
    k = j + (j - i)
    if k > n:
        k = n
    M = L[j:k]  # copier les valeurs de la seconde liste
    # cela fait de la place où l'on peut écrire dans L à partir du "haut"
    p = k - j - 1  # indice pour accéder à M par le haut
    q = j - 1      # indice pour accéder à la première plage par le haut
    r = k - 1      # indice pour mettre les valeurs dans L par le haut
    x = L[q]   # dernier élément de la première plage
    y = M[p]   # dernier élément de M (=dernier élément de la seconde plage)
    while True:
        if x > y:
            # le test est choisi pour ne permuter que ce qui doit
            # l'être (tri « stable »)
            L[r] = x
            q -= 1        # on parcourt première plage de droite à gauche
            if q >= i:    # on veut éviter un L[-1] si i=0, q=-1
                x = L[q]  # y ne change pas
            else:
                break     # il peut rester des choses dans M
        else:
            L[r] = y
            p -= 1  # on parcourt copie de seconde plage de droite à gauche
            if p >= 0:
                y = M[p]  # x ne change pas
            else:
                break     # M a été parcouru entièrement. Plus rien à faire.
        r -= 1            # diminuer r de 1 et boucler
    if p >= 0:
        L[i:r] = M[:p+1]
        # p+1 == r-i. Pourquoi ?
```

```
        # (ind.: p+q-r=-1 est invariant mais subtilité à la fin)
    return None
```

Dans le code ci-dessus, on pourrait éviter les `if` et les `break`, si on acceptait de faire `x=L[q], y=M[p]` au début de chaque boucle, or, seulement l'un des deux indices aura bougé, donc seulement l'un de `x` ou de `y` devra être mis à jour. Ça m'a paru dispendieux, et j'ai fait ces `if/break`. Cela induit le coût d'un `while True:` qui est probablement négligeable, peut-être même que Python traite spécialement `while True:` et ne teste rien ? (je ne sais pas).

### 5.3.2 tri_par_fusion(L)

---

**Exercice**

Faire une fonction `tri_par_fusion(L)` qui renvoie une nouvelle liste, égale à `L` triée par ordre ascendant. L'algorithme implémenté sera le suivant :

```
tri_par_fusion(L)

Si L est vide ou n'a qu'un seul élement, (presque) rien à faire

Si L a au moins deux éléments, trier les éléments deux par deux,
c'est-à-dire d'abord ceux d'indices 0 et 1, puis ceux d'indices
2 et 3, etc...

Fusionner ensuite les sous-listes de longueur deux, deux par deux.

Fusionner ensuite les sous-listes de longuuer quatre, deux par deux.

Etc.
```

Attention on demande que la liste d'origine ne soit pas modifiée.

---

```
def tri_par_fusion(liste):
    """Trie une liste par ordre ascendant, par fusion.

    Ne modifie pas la liste initiale.

    Opère par fusions itérées.

    >>> L = [7, 9, 3, 8, 14, 5, 10, 12, 1]
    >>> tri_par_fusion(L)
    [1, 3, 5, 7, 8, 9, 10, 12, 14]
    """
    L = liste[:]  # copie la liste (copie « superficielle »)
    n = len(L)
    if n < 2:     # liste vide ou singleton
        return L
    k = 1
    while k < n:
        i = 0
        while i < n:
            j = i + k
            fusionne(L, n, i, j)
```

```
            i = j + k
        k <<= 1
    return L
```

On va maintenant procéder à l'humiliation habituelle de vérifier le temps d'exécution. On teste sur la liste `N=[1000,999,998,...,1]` :

```
In [14]: N=list(range(1000,0,-1))

In [15]: %timeit -n 10 tri_par_fusion(N)
10 loops, best of 3: 3.01 ms per loop

In [16]: %timeit -n 10 sorted(N)
10 loops, best of 3: 23.8 µs per loop

In [17]: 3.01*1000/23.8
Out[17]: 126.47058823529412

In [18]: %timeit -n 10 tri_par_fusionB(N)
10 loops, best of 3: 3.07 ms per loop
```

On est environ 126 fois plus lent…

On a utilisé `sorted()` [84] pour ne pas modifier la liste `N`, contrairement à ce que ferait `N.sort()`.

Je recommence avec une liste aléatoire `U` de mille nombres.

```
In [25]: import random

In [26]: U = [random.randrange(1,10000) for i in range(1000)]

In [27]: %timeit -n 10 tri_par_fusion(U)
10 loops, best of 3: 4.01 ms per loop

In [28]: %timeit -n 10 tri_par_fusionB(U)
10 loops, best of 3: 3.83 ms per loop

In [29]: %timeit -n 10 sorted(U)
10 loops, best of 3: 248 µs per loop

In [30]: 3.83*1000/248
Out[30]: 15.443548387096774

In [31]: U = [random.randrange(1,10000) for i in range(1000)]

In [32]: %timeit -n 10 tri_par_fusion(U)
10 loops, best of 3: 3.9 ms per loop

In [33]: %timeit -n 10 tri_par_fusionB(U)
10 loops, best of 3: 3.84 ms per loop

In [34]: U = [random.randrange(1,10000) for i in range(1000)]

In [35]: %timeit -n 10 tri_par_fusion(U)
10 loops, best of 3: 3.89 ms per loop
```

84. https://docs.python.org/3/library/functions.html#sorted

```
In [36]: %timeit -n 10 tri_par_fusionB(U)
10 loops, best of 3: 3.82 ms per loop

In [37]: %timeit -n 10 sorted(U)
10 loops, best of 3: 257 µs per loop

In [38]: 3.82*1000/257
Out[38]: 14.863813229571985
```

Il semble qu'on soit seulement environ 15 fois plus lent: c'est pas mal ! De plus il apparaît qu'il y a un très très léger gain avec « variante B ».

Une dernière remarque est que notre temps de calcul est à peu près le même pour la liste `N=[1000,999,998,997,...]` que pour les listes aléatoires `U` avec 1000 éléments, tandis que Python lui traite la liste de type `N` dix fois plus vite que la liste de type `U`. Pour nos routines, je ne suis pas sûr de bien comprendre qu'elles soient environ 20% plus rapides dans le cas `N` que dans le cas `U`. Ah si avec la liste de type `N` on est toujours dans le cas où on saute la toute dernière étape (par exemple avec `fusionneB`, le `M` est entièrement traité à la sortie du `while True:`).

---

**Exercice**

Combien de comparaisons deux-à-deux sont-elles faites par l'algorithme précédent ?

---

## 5.4 Tri par tas

Toutes les formes de tris que nous avons vues jusqu'à présent, on peut imaginer être capable de les inventer par soi-même, si seulement, pour une fois, on se donnait la peine de réfléchir. Pour le tri par tas c'est moins sûr, car ça passe par une structure un peu bizarre dont a priori on ne voit pas bien en quoi elle peut résoudre le problème.

Tout est expliqué sur

https://fr.wikipedia.org/wiki/Tri_par_tas

alors je peux me contenter d'une description rapide.

Tout d'abord notre objet `L` de type `list` [85], indexé à partir de zéro, peut être vu comme un arbre binaire: la racine est en position zéro, chaque noeud d'indice `i` peut avoir zéro, un, ou deux descendants d'indices `2i+1` et `2i+2` respectivement. Si `2i+1>=len(L)`, le noeud d'indice `i` est une feuille, c'est-à-dire n'a pas de descendant.

Si on associe à l'indice `i` l'écriture en binaire de `i+1` (notons cette écriture par exemple `[i]`), la racine correspond à 1, ses deux descendants directs à 10 et 11, les descendants de 10 à 100 et 101, ceux de 11 à 110 et 111, etc... et d'une manière générale `j` est dans le sous-arbre engendré par `i` si et seulement si `[j]` débute par `[i]`.

Appelons profondeur de `i` la longueur en binaire de `i+1` moins un, donc la racine est de profondeur nulle, ses descendants directs de profondeur un, etc... deux feuilles distinctes `i` et `j` de mêmes profondeurs n'ont par ce qui précède aucun descendant même indirect commun, on a bien un arbre.

85. https://docs.python.org/3/library/stdtypes.html#list

Seul le dernier niveau de profondeur maximal peut être partiellement rempli, et si la feuille d'indice `i` est présente, toutes les feuilles d'indices inférieurs (que ce soit au sens des entiers ou de l'ordre lexicographique sur l'écriture binaire) le sont aussi. Si `P` est la profondeur maximale des feuilles, et `n` le nombre de noeuds, alors `2**P <= n < 2**(P+1)`. Par exemple avec `n==8` on a une racine, deux descendants, quatre descendants de descendants (dont trois feuilles), et enfin une pauvre petite feuille toute seule à la profondeur 3.

On va travailler avec une même `list`[86] `L` de longueur `N`, mais via des sections initiales ne retenant que les `n` premiers indices.

La notion cruciale est le « tas » : on dira que les `n` premiers indices forment un tas si la valeur stockée dans `L` à l'indice `i` est toujours au moins égale à celles stockées dans ses descendants directs. En particulier si `i` est une feuille, il n'y a pas de condition à vérifier.

La première opération cruciale est l'ajout d'une nouvelle valeur à un tas de longueur `n` pour en faire un tas de longueur `n+1`. On commence par la mettre à la fin. Si son parent a une valeur plus grande il n'y a rien à faire, sinon on échange les deux. Ce-faisant on n'a pas perturbé la propriété de tas pour ce noeud car l'autre descendant était inférieur au parent, et on a remplacé le parent par un truc plus grand. On compare maintenant notre valeur à son nouveau parent: si elle est inférieure ou égale, on s'arrête, sinon on fait l'échange. On remonte ainsi avec au plus un nombre d'étapes égal à la profondeur du tas initial (sauf si `n==2**(P+1)-1`, le nombre de comparaisons à faire peut alors être `P+1` à la place de `P`), et au pire cas la nouvelle valeur se retrouve en position racine.

---

**Exercice**

1. Pourquoi ne pas d'abord comparer la nouvelle valeur à la racine et procéder en descendant plutôt qu'en montant ?

2. Est-on vraiment sûr lors de l'échange que la propriété de tas est maintenue en-dessous de la valeur que l'on a fait descendre ?

---

Donc en parcourant le tableau initial de gauche à droite et en ajoutant un par un les nouveaux éléments on le transforme en un tas.

Pour la seconde phase on a maintenant en `L[0]` un élément maximal. On fait l'échange avec le dernier élément. On se retrouve avec un segment initial de longueur `N-1` qui est presque un tas, le problème est uniquement que sa racine ne vérifie peut-être pas la condition d'être plus grande que ses (au plus deux) descendants. Il faut donc une opération dite de « tamisage » qui va descendre la nouvelle valeur le long de l'arbre : on compare au plus grand des deux descendants, si c'est plus grand, rien à faire, sinon, on échange avec ce plus grand descendant. Du coup la nouvelle racine sera ok, mais le sous-arbre en dessous de l'endroit échangé doit être corrigé. On itère. Au bout d'un nombre fini d'étapes au plus égal à la profondeur, la valeur qui a remplacé l'ancienne racine trouve une place adéquate pour maintenir la propriété de tas.

On peut alors itérer en remplaçant `N` par `N-1` et à la fin la liste est triée par ordre ascendant.

### 5.4.1 `etend_le_tas(L, n)`

---

**Exercice**

86. https://docs.python.org/3/library/stdtypes.html#list

On suppose donnée une liste L avec au moins n+1 entrées, qui a la propriété de tas pour les indices de 0 à n−1. Transformer L pour avoir la propriété de tas pour les indices de 0 à n.

On implémentera l'algorithme suivant :

```
etend_le_tas(L, n)

En entrée une liste L, et un entier n.

Si n==0, arrêt. (return None)

Sinon, poser p = [(n-1)/2].
    Si L[p]>=L[n] arrêt.
    Si L[p]<L[n], échanger les valeurs, poser n=p, boucler.
```

```
def etend_le_tas(L, n):
    """Étend un tas de longueur n en un tas de longueur n+1.

    On suppose que L a la place pour au moins n+1 éléments, et que
    les n premiers forment déjà un tas.

    >>> L = [10, 5, 8, 4, 12, 2, 6, 11, 3, 9, 7, 1]
    >>> etend_le_tas(L, 4)
    >>> L
    [12, 10, 8, 4, 5, 2, 6, 11, 3, 9, 7, 1]
    >>> etend_le_tas(L, 5)
    >>> etend_le_tas(L, 6)
    >>> L
    [12, 10, 8, 4, 5, 2, 6, 11, 3, 9, 7, 1]
    >>> etend_le_tas(L, 7)
    >>> L
    [12, 11, 8, 10, 5, 2, 6, 4, 3, 9, 7, 1]
    """
    while n > 0:
        p = (n - 1) >> 1
        if L[p] >= L[n]:
            return None
        else:
            L[p], L[n] = L[n], L[p]
            n = p
```

### 5.4.2 tamise(L, n)

**Exercice**

On suppose donnée une liste L avec au moins n+1 entrées, dont le segment initial avec les indices de 0 à n est un tas sauf que peut-être la racine n'est pas supérieure à ses (au plus deux) descendants. Faites descendre la valeur originellement à la racine vers une place convenable pour rétablir la propriété de tas. Attention la longueur de ce segment est n+1, pas n.

On implémentera l'algorithme suivant :

```
tamise(L, n)
```

```
En entrée une liste L, et un entier n, dernier indice à considérer.

On pose i=0.

Si 2i+1>n, arrêt. (return None).

Si 2i+2>n, on compare L[i] à L[2i+1], et on échange si L[i] est
strictement plus petit. Ensuite arrêt.

Sinon on échange L[i] avec le plus grand de L[2i+1] et L[2i+2],
on remplace i par suivant le cas 2i+1 ou 2i+2 et on boucle.
```

```python
def tamise(L, n):
    """Tamise la racine vers sa bonne place pour que L soit un tas.

    On suppose que L a initialement la propriété de tas sauf en ce
    qui concerne sa racine, et que n est le dernier indice (donc la
    longueur est n+1).
    >>> L = [1, 11, 8, 10, 9, 2, 6, 4, 3, 5, 7]
    >>> tamise(L, 10)
    >>> L
    [11, 10, 8, 4, 9, 2, 6, 1, 3, 5, 7]
    """
    i = 0
    while True:
        # il n'y a pas de ifcase en Python.
        j = 2 * i + 1
        if j > n:
            return None  # on est déjà à une feuille
        if j == n:       # (feuille toute seule au bout, sans frère).
            if L[i] < L[n]:
                L[i], L[n] = L[n], L[i]
            return None
        else:
            if L[j+1] >= L[j]:
                j = j + 1
            if L[i] < L[j]:
                L[i], L[j] = L[j], L[i]
                i = j
            else:
                return None
```

### 5.4.3 tri_par_tas(L)

**Exercice**

Faire une fonction `tri_par_tas(L)` qui produira une nouvelle liste triée par ordre ascendant.

On implémentera l'algorithme suivant :

```
tri_par_tas(L)

En entrée une liste L. Soit n sa longueur.
```

```
En faire une copie superficielle M.

Transformer M en tas en appelant etend_le_tas(M, j) pour j allant de 1 à
n-1.

Échanger M[0] et M[n-1], puis tamiser les n-1 premières valeurs de M.

Échanger M[0] et M[n-2], puis tamiser les n-2 premières valeurs de M.

etc...
```

```python
def tri_par_tas(L):
    """Tri par tas.

    Cet exemple de liste à trier pris de
    <https://fr.wikipedia.org/wiki/Tri_par_tas>

    >>> tri_par_tas([10, 4, 8, 5, 12, 2, 6, 11, 3, 9, 7, 1])
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
    """
    n = len(L)
    M = L[:]
    if n < 2:
        return M
    # plus simple de supposer pour la suite
    # qu'il y au moins deux éléments.
    for j in range(1, n):          # j=1, 2, ..., n-1
        etend_le_tas(M, j)
    M[0], M[n-1] = M[n-1], M[0]
    for j in range(n - 2, 1, -1):  # j=n-2, n-3, ..., 2
        tamise(M, j)
        M[0], M[j] = M[j], M[0]
    # on a arrêté avant tamise(M,1) car son effet aurait été
    # de mettre le plus grand de M[0] et M[1] en premier
    # et ensuite on les aurait échangés. Un peu dispendieux.
    if M[0] > M[1]:
        M[0], M[1] = M[1], M[0]
    return M
```

**Exercice**

Combien de comparaisons deux-à-deux sont-elles faites par l'algorithme précédent ? (on demande un ordre de grandeur dans le pire cas envisageable).

On va maintenant procéder à l'humiliation habituelle de vérifier le temps d'exécution. On teste sur la liste `N=[1000,999,998,...,1]` :

```
In [52]: N=list(range(1000,0,-1))

In [53]: %timeit -n 10 tri_par_tas(N)
10 loops, best of 3: 6.47 ms per loop

In [54]: %timeit -n 10 sorted(N)
```

```
10 loops, best of 3: 23.8 µs per loop

In [55]: 6.47*1000/23.8
Out[55]: 271.8487394957983

In [56]: import random

In [57]: U = [random.randrange(1,10000) for i in range(1000)]

In [58]: %timeit -n 10 tri_par_tas(U)
10 loops, best of 3: 6.98 ms per loop

In [59]: %timeit -n 10 sorted(U)
10 loops, best of 3: 290 µs per loop

In [60]: 6.98*1000/290
Out[60]: 24.06896551724138

In [61]: V = tri_par_tas(U)

In [62]: import tp2

In [63]: tp2.estordonnee(V)
Out[63]: True
```

Sur un autre ordinateur, j'ai commencé par obtenir des temps assez différents :

```
In [5]: import random

In [6]: import tp2

In [7]: N=list(range(1000,0,-1))

In [8]: P=tri_par_tas(N)

In [9]: tp2.estordonnee(P)
Out[9]: True

In [10]: %timeit -n 10 tri_par_tas(N)
10 loops, best of 3: 5.69 ms per loop

In [11]: %timeit -n 10 sorted(N)
10 loops, best of 3: 40.3 µs per loop

In [12]: 5.69*1000/40.3
Out[12]: 141.19106699751862

In [13]: U = [random.randrange(1,10000) for i in range(1000)]

In [14]: V=tri_par_tas(U)

In [15]: tp2.estordonnee(V)
Out[15]: True

In [16]: %timeit -n 10 tri_par_tas(U)
10 loops, best of 3: 6.3 ms per loop
```

```
In [17]: %timeit -n 10 sorted(U)
10 loops, best of 3: 421 µs per loop

In [18]: 6.3*1000/421
Out[18]: 14.964370546318289
```

Cependant, plus tard sur cette même machine :

```
In [21]: %timeit -n 10 sorted(U)
10 loops, best of 3: 281 µs per loop

In [22]: %timeit -n 10 tri_par_tas(U)
10 loops, best of 3: 6.3 ms per loop

In [23]: 6.3*1000/281
Out[23]: 22.419928825622776
```

Difficile donc de tirer des conclusions très quantitatives avec notre emploi naïf de %timeit [87].

La routine `tp2.estordonnee(liste)` est définie dans la *Feuille de travaux pratiques 2* (page 33).

---

*Date de dernière modification* : 29-11-2017 à 19:12:30.

87. http://ipython.readthedocs.org/en/stable/interactive/magics.html?highlight=magic%20alias#magic-timeit

# A

## Documentation sur l'algorithme de primalité AKS

Cet algorithme (2002) a été le premier avec une preuve du fait qu'il détermine en temps polynomial si un entier N donné est premier ou non. Dans la pratique, il n'a pas supplanté (à ma connaissance) d'autres algorithmes antérieurs nettement plus rapides. Mais ceux-ci soit sont d'un type « probabilisé », c'est-à-dire avec une réponse du type « *ce nombre* N *est premier avec une probabilité de 99.9999999999999999%* » [1], soit n'ont pas de borne polynomiale (en la taille de l'entier N en base 2 ou en base 10) sur leur temps d'exécution, soit encore ont une telle borne polynomiale mais avec une justification qui dépend de la validité de certaines hypothèses en théorie des nombres qui sont restées à ce jour sans démonstrations.

Les mathématiques nécessaires pour comprendre l'algorithme AKS et la preuve de son fonctionnement sont d'un niveau relativement élémentaire : un peu d'algèbre (corps, anneaux de polynômes), un peu de théorie des groupes finis (commutatifs…), un peu d'arithmétique, un peu de polynômes cyclotomiques…

**http://annals.math.princeton.edu/wp-content/uploads/annals-v160-n2-p12.pdf**
L'article (*en anglais*) paru dans Annals of Mathematics (2004 ; version remaniée de la prépublication d'origine d'août 2002).

**http://smf4.emath.fr/Publications/Gazette/2003/98/smf_gazette_98_14-29.pdf**
Un article paru dans la Gazette des mathématiciens, octobre 2003, traduction d'un article par un mathématicien allemand dans les Notices de l'AMS. Riche en détails sur le contexte mathématique et sociologique dans lequel la preuve est apparue.

**http://www.dms.umontreal.ca/~andrew/PDF/Bulletin04.pdf** Un article très complet de Andrew Granville. (*en anglais*)

1. Si une seule itération du programme est capable de dire « *ce nombre* N *est premier avec une probabilité de 75%* » alors 30 itérations suffisent pour une probabilité de *99.9999999999999999%*, et si après 150 itérations le programme s'acharne à dire que N est premier, c'est avec une probabilité d'erreur inférieure à $0.5 \times 10^{-90}$. Or on estime qu'il y a environ $10^{80}$ atomes dans l'univers (observable) ! D'ailleurs une probabilité inférieure à $10^{-40}$ serait déjà quelque chose d'infimissime.

À noter cependant que ces estimations de probabilité sont théoriques, elles supposent que le logiciel dispose d'un générateur parfait de nombres aléatoires, ce qui n'est pas le cas ; il faudrait interfacer avec un dispositif physique quantique, le quantique étant l'unique source de phénomènes intrinsèquement aléatoires (autant que je puisse en juger du haut de mon ignorance).

Un exemple de tel algorithme est le test de Miller-Rabin. Lorsque l'algorithme dit que le nombre est composé, c'est avec certitude.

Il existe aussi des tests qui, en temps polynomial, disent soit « N *est premier* », soit « N *est probablement composé* ». Il est expliqué dans l'article de Granville, que Berrizbeitia-Cheng-Bernstein-Mihailescu-Avanzi ont obtenu en partant de AKS un algorithme plus efficace que celui précédemment connu de Adleman-Huang.

En alternant les deux sortes d'algorithmes probabilistes on est sûr d'obtenir au final une des deux réponses « N *est premier* » ou « N *est composé* », mais on ne sait pas combien de temps il faudra attendre.

**http://www.trigofacile.com/maths/curiosite/primarite/aks/pdf/algorithme-aks.pdf**
Un mémoire détaillé rédigé par un étudiant.

**Voir aussi:**

**http://www.math.univ-toulouse.fr/~hallouin/Documents/Primalite.pdf** Un survol d'autres tests de primalité (dont celui de Miller-Rabin [90] ).

---

*Date de dernière modification* : 17-12-2014 à 09:23:15 CET.

90. http://fr.wikipedia.org/wiki/Test_de_primalit%C3%A9_de_Miller-Rabin

# B

# Examen du 20 février 2015



## B.1 Exercice 1

On définit une fonction `F(x)` sur les entiers strictement positifs par l'algorithme suivant :

```
si x est impair alors F(x) = 3x - 1
si x est pair   alors F(x) = x/2
```

1. Faire une procédure Python `F(x)` qui implémente cette fonction. Elle ne vérifiera pas que son argument est bien un entier au moins égal à 1.
2. Faire une procédure `iterF(x,m)` qui imprime `x` puis, séparés par des virgules, les `m` premiers itérés `F(x), F(F(x)), F(F(F(x))), ...` (par exemple `F(F(F(x)))` est le troisième itéré).

   Par exemple pour `N=50`, la procédure imprimera `50, 25, 74, 37, 110, ...` jusqu'au `m`-ième itéré.

   Voici une partie de la procédure :

   ```python
   def iterF(x,m):
       print(x, end = ", ")   # (syntaxe Python3)
       for ... :
           ...
       return None
   ```

3. On constate expérimentalement que quelque soit le point de départ `x` il y a toujours un itéré qui vaut 1 ou 5 ou 17.

   Faire une procédure `longueurF(x)` qui renvoie le premier `N` tel que le $N^e$ itéré est 1 ou 5 ou 17. Par exemple si `x=34`, il faut que la procédure renvoie `1`, si `x=5` il faut `0`.

4. Faire `lepluslong(X)` qui examine tous les entiers x de 1 à `X` (**inclus**) et renvoie un `tuple`[91] `(x,N)` avec x celui pour lequel `longueurF(x)` est le plus grand, et `N = longueurF(x)` (si plusieurs x conviennent on demande le plus petit d'entre eux).
   Que donne `lepluslong(10000)` ?

## B.2 Exercice 2

On va travailler avec des matrices $2\times2$. On utilisera en Python la liste de listes `[[a, b], [c, d]]` pour représenter la matrice $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$.

1. Faire une procédure `MatMulMod(A,B,N)` qui calcule le produit matriciel `AB` modulo l'entier `N`. Les matrices sont donc supposées à coefficients entiers. Voici une partie de la procédure :

```
def MatMulMod(A,B,N):
    a = (A[0][0]*B[0][0]+A[0][1]*B[1][0])%N
    ...
    ...
    ...
    return [[a,b],[c,d]]
```

2. Faire une procédure `MatPowMod(A,k,N)` qui calcule $A^k \bmod N$ suivant la **récursion** suivante :

```
A^0 est la matrice identité
A^1 est A (modulo N)
si k = 2l      est pair   A^(2l)   = (A^2)^l (modulo N)
si k = 2l + 1  est impair A^(2l+1) =  A fois (A^2)^l  (modulo N)

  (le symbole ^ a été utilisé pour représenter les puissances)
```

3. Calculer avec votre procédure la matrice $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ à la puissance 123456789987654321123456789, modulo 999.

## B.3 Exercice 3

La fonction `M(n)` de l'entier n est définie par :

```
M(0) = 0
M(1) = 1

si n>1:
  on fait la somme S des M(k) pour k diviseur de n, 1 <= k < n.
  M(n) = - S.
```

Voici les premières valeurs `M(2)=-1`, `M(3)=-1`, `M(4)=0`, `M(5)=-1`, `M(6)=1`. Autres exemples `M(30) = -1`, `M(77) = 1` et `M(100)=0`.

91. https://docs.python.org/3/library/stdtypes.html#tuple

1. Faire une procédure `ListeM(N)` qui produit pour `N>=1` la `list`[92] `[M(0),M(1),..., M(N)]` (elle contient donc `N+1` valeurs). Voici une partie de la procédure :

```
def ListeM(N):
    L = [0, 1]
    if N == 1:
       return L
    for ...
        L.append (....
    return L
```

   Remarque: il se trouve que `M(n)` ne prend que les valeurs `-1, 0, +1`, ce que sa définition ne montre pas immédiatement.

2. Faire une procédure `Sommes(L)` qui prend en entrée une `list`[93] `L` et renvoie une nouvelle liste `SL` telle que `SL[n] = L[0]+L[1]+...+L[n]` pour tous les `n` possibles.

3. On pose `S(n) = M[0]+M[1]+...+M[n]` pour tous les entiers `n`.
   Faire une procédure `OK(N)` qui renvoie `True` si `M(n)=-1, 0, ou 1` et si `S(n)^2 < n` pour tous les entiers `n` entre `2` et `N` (inclus), et `False` sinon (de plus dans ce cas, la procédure imprime la valeur de `n`).
   Vérifier que `OK(3000)` est `True`.

4. Faire une procédure `moyenne(N)` qui calcule $\frac{1}{N}\sum_{k=1}^{N}\frac{S(k)^2}{k}$, et donner les valeurs de `moyenne(N)` pour `N = 500, 1000, 1500, 2000, 2500, 3000`.

## B.4 Exercice 4

On a une `list`[94] `liste` **déjà triée** par ordre croissant, sans éléments identiques. On veut une procédure `PlusGrandPlusPetit(x,liste)` qui renvoie un `tuple`[95] `(i,'=')` ou `(i,'<')` avec les significations suivantes :

```
(i, '=')  si x = liste[i]
(i, '<')  si liste[i]<x et i est le plus grand avec liste[i]<= x
(-1, '<') si x < liste[0]
```

Implémenter l'algorithme suivant :

```
initialisation
  L = longueur de liste,
  I = 0

tant que L>1
  J = I + [L/2]  (on a utilisé la notation [t] pour la partie entière de t)
  comparaison de x et de liste[J]

    si égalité fini.
    si x>liste[J] remplacer I par J et L par L - [L/2].
    si x<liste[J] remplacer L par [L/2] et laisser I invariant.

si L=1, comparer x avec liste[I]. Conclure. On notera que
à ce stade nécessairement x>=liste[I] sauf si I =0 et x<liste[0]
```

92. https://docs.python.org/3/library/stdtypes.html#list
93. https://docs.python.org/3/library/stdtypes.html#list
94. https://docs.python.org/3/library/stdtypes.html#list
95. https://docs.python.org/3/library/stdtypes.html#tuple

## B.5 Corrigé

Le code n'est pas vraiment commenté, ce qui n'est pas bien.

```
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 19 10:50:56 2015

@author: Herr Doktor Professor B.
"""
#%%
def F(x):
    if x&1:    # x est impair
        return 3*x-1
    else:      # x est pair
        return x>>1
#%%
def iterF(x,m):
    print(x, end = ", ")
    for i in range(m):
        x=F(x)
        print(x, end = ", ")
    # on ne s'est pas préoccupé d'inhiber la dernière virgule.
    return None
#%%
#
# si on veut inhiber la dernière virgule :
def iterF(x,m):
    # on suppose m au moins égal à 1.
    print(x, end = ", ")
    for i in range(1,m):
        x=F(x)
        print(x, end = ", ")
    print(F(x))
    return None
#%%
def longueurF(x):
    N = 0
    while x!=1 and x!=5 and x!=17:
        x=F(x)
        N += 1
    return N
#%%
def lepluslong(X):
    x = 1
    N = 0
    for i in range(2, X+1):
        N1 = longueurF(i)
        if N1>N:
            x=i
            N=N1
    return x, N
#%%
#
```

```
#In [2]: lepluslong(10000)
#Out[2]: (9735, 234)
#
#%%
def Det(A):
    return A[0][0]*A[1][1]-A[1][0]*A[0][1]
#%%
def MatMulMod(A,B,N):
    a = (A[0][0]*B[0][0]+A[0][1]*B[1][0])%N
    b = (A[0][0]*B[0][1]+A[0][1]*B[1][1])%N
    c = (A[1][0]*B[0][0]+A[1][1]*B[1][0])%N
    d = (A[1][0]*B[0][1]+A[1][1]*B[1][1])%N
    return [[a,b],[c,d]]
#%%
def MatPowMod(A,k,N):
    if k==0:
        return [[1,0],[0,1]]
    if k==1:
        return [[A[0][0]%N,A[0][1]%N],[A[1][0]%N,A[1][1]%N]]
    if k%2: # k est impair
        return MatMulMod(A,MatPowMod(MatMulMod(A,A,N),k//2,N),N)
    else:   # k est pair
        return MatPowMod(MatMulMod(A,A,N),k//2,N)
#%%
#
#In [3]: MatPowMod([[1,2],[3,4]],123456789987654321123456789,999)
#Out[3]: [[352, 81], [621, 973]]
#
#%%
def ListeM(N):
    L = [0,1]
    if N == 1:
        return L
    for n in range(2,N+1):
        L.append(-sum(L[k] for k in range(1,n) if n%k == 0))
    return L
#%%
def Sommes(L):
    M = []
    somme = 0
    for n in range(len(L)):
        somme = somme + L[n]
        M.append(somme)
    return M
#%%
#
# plus élégant :
def Sommes(L):
    M = []
    somme = 0
    for x in L:
        somme += x
        M.append(somme)
    return M
#%%
def OK(N):
    M = ListeM(N)
```

```
    S = Sommes(M)
    for n in range(2,N+1):
        if S[n]**2>= n:
            print (n)
            return False
        if not M[n] in {-1, 0, 1}:
            print (n)
            return False
    return True
#%%
#
#In [4]: OK(3000)
#Out[4]: True
#
#%%
def moyenne(N):
    M = ListeM(N)
    S = Sommes(M)
    return sum(S[n]**2/n for n in range(1,N+1))/N
#%%
#
#In [5]: for i in range(1,7):
#   ...:     print(moyenne(i*500))
#   ...:
#0.06838285847454163
#0.04741248698793715
#0.04257855194091775
#0.03979189755772839
#0.035744529425171
#0.04119676191356717
#
#%%
def PlusGrandPlusPetit(x,liste):
    I = 0
    L = len(liste)
    while L>1:
        L1=L>>1
        J = I + L1
        if x==liste[J]:
            return J, '='
        elif x>liste[J]:
            I=J
            L=L-L1
        else:
            L=L1
    if x==liste[I]:
            return I, '='
    elif x>liste[I]:
        return I, '<'
    else:
        return -1, '<'
    # il aurait mieux valu comparer x à liste[0] dès le début
```

*Date de dernière modification* : 21-02-2015 à 13:35:25 CET.

# C

# Examen du 10 décembre 2015



## C.1 Quelques rappels utiles pour cet examen

1. En Python `A % N` donne si `N>0` le représentant entre `0` et `N-1` de la classe de congruence de `A` modulo `N`, aussi si `A<0`.
2. La fonction `pow()` [96] admet un troisième argument optionnel `N`: `pow(A, n, N)` fait `(A**n) % N`, de manière plus efficace que de calculer l'entier `A**n` pour ensuite le réduire modulo `N`. Par contre, on ne peut pas l'utiliser avec `n<0`, même pour `A` inversible modulo `N`.
3. Un objet `L` de type `list` [97] peut avoir des entrées qui sont d'autres objets, par exemple des `tuple` [98] :

```
In [2]: L = [(3,"a"), (2, "b"), (1,"c")]

In [3]: L[2]
Out[3]: (1, 'c')

In [4]: L[2][0]
Out[4]: 1

In [5]: L[2][1]
Out[5]: 'c'

In [6]: type(L), type(L[2]), type(L[2][0]), type(L[2][1])
Out[6]: (list, tuple, int, str)
```

96. https://docs.python.org/3/library/functions.html#pow
97. https://docs.python.org/3/library/stdtypes.html#list
98. https://docs.python.org/3/library/stdtypes.html#tuple

4. Lorsque l'on applique la fonction `sorted()` [99] sur ce genre de liste, le tri se fait d'abord sur la première coordonnée de chaque entrée, puis secondairement sur la deuxième, etc…

   Ainsi, avec la liste `L` ci-dessus, on obtient :

```
In [7]: M=sorted(L)

In [8]: M
Out[8]: [(1, 'c'), (2, 'b'), (3, 'a')]
```

   En effet les premiers indices sont numériques et distincts, les `tuple` [100] se retrouvent classés par ordre croissant de leurs premières coordonnées.

## C.2 Exercice 1

Je rappelle que si $P$ est un nombre premier, alors le groupe des éléments inversibles $(\mathbb{Z}/P\mathbb{Z})^*$ est cyclique de cardinalité $P-1$, c'est-à-dire qu'il existe un entier $g$ premier à $P$ tel que $(\mathbb{Z}/P\mathbb{Z})^* = \{1, g, g^2, g^3, \ldots, g^{P-2}\}$. On dit que $g$ est un générateur. Les autres générateurs sont les $g^a$ avec $a$ premier à $P-1$.

Il n'y a pas de formule générale connue donnant en fonction de $P$ le plus petit générateur $g$. On soupçonne par exemple qu'il y a une infinité de $P$ pour lesquels $g = 2$ (ou $g = 3$, etc…) convient, mais ce n'est pas démontré (à ma connaissance).

Si l'on connaît les facteurs premiers distincts $q_1, \ldots, q_r$ de $P-1$ et qu'on appelle « cofacteurs » les entiers $m_i = (P-1)/q_i$ alors un critère nécessaire et suffisant pour vérifier que $g$ est un générateur est que les $g^{m_i}$ ne sont **pas** congrus à 1 modulo $P$ (bien sûr $g$ est supposé non-divisible par $P$).

Par exemple, $P = 1021$ est premier, et les facteurs premiers de 1020 sont 2, 3, 5, 17. Les cofacteurs sont donc ici 510, 340, 204, et 60 ; si aucun de $g^m \bmod 1021$ n'est 1, pour $m = 510, 340, 204, 60$, alors $g$ est un générateur.

Faites une procédure `estgenerateur(x, P, coFacteurs)`, qui étant donné un entier `x` et un nombre premier `P` ainsi que la liste de ses coFacteurs, réponde `True` si `x` est un générateur et `False` sinon. La raison pour passer les coFacteurs en argument est qu'on ne veut pas avoir à les calculer à nouveau pour chaque `x`. On devrait plutôt utiliser les possibilités de programmation objet de Python, définir une classe `CorpsFini` avec une méthode `estgenerateur`, mais on n'a pas parlé de ceci en cours, alors on procède plus naïvement.

```
def estgenerateur(x, P, coFacteurs):
    """\
    Détermine si l'entier positif x est un générateur de (Z/PZ)*

    On suppose que coFacteurs est une "list" des entiers m de la forme
    (P-1)/q, avec q parcourant les diviseurs premiers distincts de P-1.

    L'entier P est supposé premier. coFacteurs est déterminé par P, mais
    on le suppose pré-calculé. Pour P=2, coFacteurs sera [].
    """
    if x%P == 0:
        return False
    [compléter le code]
```

99. https://docs.python.org/3/library/functions.html#sorted
100. https://docs.python.org/3/library/stdtypes.html#tuple

Par exemple on doit pouvoir ensuite utiliser la fonction de la manière suivante :

```
In [10]: P = 1021

In [12]: estgenerateur (2, P, [510, 340, 204, 60])
Out[12]: False

In [14]: estgenerateur (10, P, [510, 340, 204, 60])
Out[14]: True

In [15]: for i in range(2,50):
    ...:     if estgenerateur(i,P,[510, 340, 204, 60]):
    ...:         print(i, end=", ")
    ...:
10, 22, 30, 31, 34, 35, 37, 40, 43, 46,

In [16]: P = 7604629321 # on admet que c'est un nombre premier

In [18]: (2**3)*3*5*(83**2)*9199 # le professeur a calculé la factorisation
Out[18]: 7604629320

In [19]: coF = [(P-1)//x for x in [2, 3, 5, 83, 9199]]

In [20]: coF
Out[20]: [3802314660, 2534876440, 1520925864, 91622040, 826680]

In [21]: for i in range(2,50):
    ...:     if estgenerateur(i,P,coF):
    ...:         print(i, end=", ")
    ...:
23, 29, 41, 43, 47,
```

## C.3 Exercice 2

On suppose donné deux `list`[101] `LA` et `LB`, *déjà triées par ordre ascendant*. On veut les plus petits indices `p` et `q` avec `LA[p]=LB[q]`, et s'ils n'existent pas renvoyer `None, None`. On implémentera l'algorithme suivant :

```
Initialiser: p=0, q=0
Boucler:
    si LA[p]==LB[q] retour p, q
    sinon, si LA[p]>LB[q], alors q<- q+1.
           si LA[p]<LB[q], alors p<- p+1.
Sortie de boucle par return, ou lorsque p ou q devient trop grand.
```

Le code ressemblera à :

```
#%%
def pluspetitcommunsimple (L1, L2):
    """\
    Renvoyer le premier couple (p,q) avec L1[p]=L2[q].

    On suppose que L1 et L2 sont déjà triées par ordre croissant.
```

101. https://docs.python.org/3/library/stdtypes.html#list

```
    Si pas de coïncidence, faire return None, None
    """
    l1 = len(L1)
    l2 = len(L2)
    p = 0
    q = 0
    while ...
    ...
         return p, q
    ...
    return None, None
```

On remarque que l'algorithme termine en moins d'étapes que le nombre d'entrées de `L1` plus celui de `L2`.

On considère dans un deuxième temps des listes composées de `tuple`[102] du genre `L[p] = (m, i)` avec des entiers `m` et `i` dépendant de `p`, et par hypothèse les couples `(m,i)` sont rangés par ordre croissant des `m`, pour chacune des deux listes `L1` et `L2`. Faire une fonction `pluspetitcommun (L1, L2)` qui renverra le premier couple `(i,j)` qui aura donné lieu à une coïncidence `L1[p] = (m, i)`, `L2[q] = (m, j)` avec le même `m`.

```
#%%
def pluspetitcommun (L1, L2):
    """\
    Étant données deux liste L1 et L2 triées de tuple's (a, i)
    trouve la première coïncidence des "a" et renvoie (i, j).

    Si pas de coïncidence, renvoie (None, None).
    """
    l1 = len(L1)
    l2 = len(L2)
    p = 0
    q = 0
    while ...
    ...
       if L1[p][0]==L2[q][0]:
           return L1[p][1], L2[q][1]
    ...
    return None, None
```

## C.4 Exercice 3

On suppose `y>0` et `x` positif ou nul. On veut déterminer l'inverse multiplicatif de `x` modulo `y`, ou renvoyer `0` si `x` et `y` ont un facteur commun. Implémenter l'algorithme classique suivant :

```
Initialiser: s=0, d=y, u=1, e=x. (y>0 et x positif ou nul).
Boucler tant que e est non nul :
    q, r = divmod(d,e)
    remplacer s par u, et u par s - qu, et d par e, et e par r
Une fois que e==0 :
   si d ne vaut pas 1 renvoyer 0,
   si d vaut 1 renvoyer s lorsque s>=0 et s+y si s<0
```

102. https://docs.python.org/3/library/stdtypes.html#tuple

```
      (on peut prouver que le s final vérifie |s|<= y,
       et même |s|<y, avec comme unique exception y=x=1
       qui donne s=-1, et dans ce cas s+y sera 0 qui est <y)
```

```
#%%
def inversemodulo(x,y):
    """\
    Calcule l'inverse de x modulo y. Doit renvoyer un entier positif.

    C'est l'algorithme classique d'Euclide du pgcd, avec décoration pour
    Bezout. Si x n'est pas inversible modulo y, retourne 0.

    On suppose y>0 et x positif ou nul et on veut s entre 0 et y-1 avec
    sx congru à 1 modulo y.
    """
    s,u,d,e = 0,1,y,x
    ...
       return s
    ...
       return s+y
    ...
    return 0
    ...
```

```
>>> inversemodulo(199,7604629321)
>>> 2369281497
>>>  # vérifions que ça marche.
>>> 199*2369281497 % 7604629321
>>> 1
```

## C.5 Exercice 4

Ce dernier exercice suppose que *Exercice 2* (page 133) et *Exercice 3* (page 134) ont été faits.

Étant donné un nombre premier `P` et un générateur `g` de son groupe multiplicatif, faire une routine `TPourLogDiscret(P,g)` qui va renvoyer le `tuple`[103] suivant :

```
P, Q, g, L, M avec

P : le nombre premier donné en input (on ne vérifie pas qu'il est premier)
Q : plus petit entier de carré au moins P
g : le nombre g donné en input (on ne vérifie pas qu'il est générateur)
L : la liste triée des tuple (g**(Q*i) modulo P, i), i de 0 à Q-1
M : la liste (pas triée) des (g**(-j) modulo P, j), j de 0 à Q-1
```

```
#%%
from math import sqrt, ceil
#%%
def TPourLogDiscret(P, g):
    """\
    Renvoie le tuple (P, Q, g, L, M)
```

103. https://docs.python.org/3/library/stdtypes.html#tuple

```
    On ne vérifie pas que P est un nombre premier et g un générateur.
    On calcule :

    - Q = plus petit entier avec Q**2 au moins P.
    - L = la liste triée des couples ((g**Q)**i mod P, i) pour i de 0 à Q-1
    - M = la liste des couples (g**(-i) mod P, i) pour i de 0 à Q-1

    M n'est pas triée car on devra le faire plus tard après avoir
    multiplié tous ses éléments.

    Exemple :

    >>> T=TPourLogDiscret(13,2)
    >>> T
    (13, 4, 2, [(1, 0), (1, 3), (3, 1), (9, 2)],
               [(1, 0), (7, 1), (10, 2), (5, 3)])

    Remarque : le dernier élément de L est g**(Q(Q-1)). On peut avoir
    Q(Q-1) égal ou supérieur à P-1. Par exemple P=11, Q=4, Q(Q-1)=12.
    Donc L est peut-être plus long que nécessaire. Mais Q(Q-2)=(Q-1)**2-1
    est < P-1. Ainsi seul le dernier tuple de L peut être superflu.
    """
    Q = ceil(sqrt(P))
    a = ... #  g à la puissance Q modulo P
    b = ... #  inverse multiplicatif de g modulo P
    # Construire L liste TRIÉE des tuple (a**i modulo P, i)
    # pour i de 0 à Q-1
    # Construire M, liste des tuple (b**j modulo P, j) pour j de 0 à Q-1
    return P, Q, g, L, M
```

Puis, implémenter une fonction `LogDiscret(x, T)` dont le premier argument sera un entier `x` et `T` un `tuple`[104] tel que fourni par la fonction `TPourLogDiscret(P, g)`; on demande que `LogDiscret(x, T)` calcule, si `x` est premier avec `P`, l'entier unique `n<P-1` tel que `x=g**n modulo P`. Si `x` est divisible par `P`, on renvoie `None`. On implémentera l'algorithme suivant :

```
À partir de M, former la nouvelle liste M' des tuple (x*g**(-j) mod P, j),
et la trier (par rapport à la première entrée) via sorted().

Trouver une coïncidence (cf Exercice 2) entre L et M', ce qui donnera un
couple (i,j) avec g**(Q*i) = x g**(-j) modulo P, donc x = g**n modulo P
avec n=Q*i+j.

Réduire n modulo P-1, au cas où.

Bien sûr on aura au début éliminé les cas avec x divisible par P.
```

Je rappelle qu'à partir de `T`, on a `P = T[0]`, `g=T[2]`, …

```
#%%
def LogDiscret(x, T):
    """\
    Avec T = (P, Q, g, L, M), calcule n < P-1 tel que g**n = x modulo P.

    On suppose x positif ou nul, et que T a été fait par TPourLogDiscret(P,g)
```

104. https://docs.python.org/3/library/stdtypes.html#tuple

```
    Si P divise x, renvoie None.
    """
    P = T[0]
    ...
    LB = sorted((..., u[1]) for u in T[4])
    i, j = pluspetitcommun (...)
    return (...) % (P-1)
```

On veut pouvoir obtenir :

```
In [30]: P = 7604629321 # c'est un premier, promis.

In [31]: coF = [(P-1)//x for x in [2, 3, 5, 83, 9199]]

In [32]: estgenerateur(23, P, coF)
Out[32]: True

In [33]: T=TPourLogDiscret(7604629321, 23)

In [34]: LogDiscret (2015, T)
Out[34]: 4891928574

In [35]: pow(23, 4891928574, 7604629321)
Out[35]: 2015

In [36]: estgenerateur(2014, P, coF)
Out[36]: True

In [37]: T=TPourLogDiscret(P,2014)

In [38]: LogDiscret(2015, T)
Out[38]: 687943014

In [39]: pow(2014, 687943014, 7604629321)
Out[39]: 2015
```

D'où ce superbe résultat :

$$2014^{687943014} \equiv 2015 \pmod{7604629321}$$

Comme `Q=87205` dans cet exemple on a fait beaucoup moins de calculs (*) que d'essayer tous les $2014^n$ jusqu'à en trouver un qui donne 2015. Car avec notre algorithme, après la préparation de `T` (qui est en `O(Q log Q)`) on doit faire `Q=87205` multiplications modulo `P`, puis un tri qui prend `O(Q log Q)`, puis une recherche finale en `O(Q)`.

(*) à propos du temps de calcul il est malhabile dans la construction des `L` et `M` dans `TPourLogDiscret(P,g)` d'utiliser plein de `pow(m,n,P)`, il est beaucoup plus efficace d'engendrer par multiplication itérée modulo `P` les nombres entrant dans les `tuple`[105] de `L` (avant le tri) et de `M`. Cependant une solution avec plein de `pow(m,n,P)` sera acceptée.

Bien sûr si on avait le temps et la mémoire on pourrait précalculer tous les `(g**n modulo P, n)` : on remplirait un `list`[106] `L`, préinitialisé par `L=[None]*P`, par des instructions `L[g**n mod P]=n`, et ensuite, pour tout `x` on aurait « immédiatement » le `n` avec `g**n = x modulo P`, simplement en regardant `L[x]`.

105. https://docs.python.org/3/library/stdtypes.html#tuple
106. https://docs.python.org/3/library/stdtypes.html#list

Avec notre `P = 7604629321`, cette approche nécessiterait des giga-octets de mémoire (et beaucoup de temps de calcul initialement). Pour `P=1021` par contre, bien sûr cette option « tout précalculer » est largement faisable :

```
In [53]: P=1021         # on va utiliser g=10

In [54]: L=[None]*P     # initialiser un tableau

In [55]: x=1; L[1] = 0; # car 10**0 = 1

In [56]: for i in range (1, P-1):
    ...:     # À COMPLÉTER – AJOUTER LE CODE À VOTRE FICHIER .py
    ...:     # éventuellement dans un bloc de commentaires à la fin.

In [57]: L[2015%1021]
Out[57]: 300

In [58]: pow(10,300,1021)
Out[58]: 994

In [59]: 994+1021
Out[59]: 2015
```

## C.6 Corrigé

```
# -*- coding: utf-8 -*-
"""
Created on Tue Dec  8 13:30:40 2015

Pour préparation examen du jeudi 10 décembre 2015.

@author: Herr Professor Doktor B.
"""
#%%
def estgenerateur(x, P, coFacteurs):
    """\
    Détermine si l'entier positif x est un générateur de (Z/PZ)*

    On suppose que coFacteurs est une "list" des entiers m de la
    forme (P-1)/q, avec q parcourant les diviseurs premiers de P-1.

    L'entier P est supposé premier. coFacteurs est déterminé par P,
    mais on le suppose pré-calculé. Pour P=2, coFacteurs sera [].
    """
    if x%P == 0:
        return False
    for m in coFacteurs:
        if pow(x, m, P)==1:
            return False
    return True
#%%
# (exemples)
# LePremier   = 1021
# LeX         = 32   # X**2 > 1020
# LePhi       = 1020 # P -1
```

```
# LesFacteurs = [2, 3, 5, 17] # diviseurs premiers de LePhi
# LesCoFacteurs = [LePremier//x for x in LesFacteurs]
#In [45]: for i in range(2,100):
#    ...:     if estgenerateur(i, 1021, LesCoFacteurs):
#    ...:         print(i, end=", ")
#    ...:
# 10, 22, 30, 31, 34, 35, 37, 40, 43, 46, 50, 53, 59, 65, 66, 76, 77,
# 82, 90, 93, 94, 95,
#%%
# LePremier   = 7604629321
# LesFacteurs = [2, 3, 5, 83, 9199]
# LesCoFacteurs = [LePremier//x for x in LesFacteurs]
#
#In [11]: for i in range(2,200):
#    ...:     if estgenerateur(i, LePremier, LesCoFacteurs):
#    ...:         print(i, end=", ")
#    ...:
#23, 29, 41, 43, 47, 53, 69, 73, 87, 89, 92, 94, 97, 101, 103, 107, 109,
#116, 118, 123, 131, 134, 138, 159, 164, 177, 179, 181, 184, 193, 199,
#%%
def pluspetitcommun (L1, L2):
    """\
    Étant données deux liste L1 et L2 triées de tuple's (a, i)
    trouve la première coïncidence des "a" et renvoie (i, j).

    Si pas de coïncidence, renvoie (None, None).
    """
    l1 = len(L1)
    l2 = len(L2)
    p = 0
    q = 0
    while True:
        if L1[p][0]==L2[q][0]:
            return L1[p][1], L2[q][1]
        if L1[p][0]>L2[q][0]:
            q = q+1
            if q==l2:
                return None, None
        else:
            p = p+1
            if p==l1:
                return None, None
#%%
def inversemodulo(x,y):
    """\
    Calcule l'inverse de x modulo y. Doit renvoyer un entier positif.

    C'est l'algorithme classique d'Euclide du pgcd, avec décoration pour
    Bezout. Si x n'est pas inversible modulo y, retourne 0.

    On suppose y>0 et x positif ou nul et on veut s entre 0 et y-1 avec
    s fois x congru à 1 modulo y.
    """
#    s,t,u,v,d,e = 0,1,1,0,y,x
    s,u,d,e = 0,1,y,x
    while e: # tant que e est non nul, continuer. (algorithme d'Euclide)
        q, r = divmod(d, e)
```

```
#        s, t, u, v, d, e = u, v, s-q*u, t -q*v, e, r
        s, u, d, e = u, s-q*u, e, r
    if d==1:
        if s<0:
            return s+y
        else:
            return s
    return 0
#%%
# (exemples)
#inversemodulo(199,7604629321)
#Out[40]: 2369281497
#
#199*2369281497 % 7604629321
#Out[41]: 1
#%%
#%%
def listedepuissancesmoduloN(A, K, N):
    """\
    Renvoie [(A**i mod N, i) pour i =0, ..., K-1]

    Ainsi, tous les entiers calculés sont < N.
    """
    # Ceci ne serait pas très efficace :
    # return [(pow(A, i, N) % N, i) for i in range(K)]
    # car en effet tous ces pow sont coûteux.
    #
    # On va plutôt itérer des multiplications.
    # Et en effet, j'ai constaté que c'est beaucoup plus rapide
    # lorsque par exemple K vaut à peu près 80000.
    #
    x = 1        # sera A**i modulo N
    L = [(1, 0)] # A**0 = 1
    for i in range (1, K):
        x = A*x % N            # le * a priorité devant le %
        L.append((x, i))
    return L
#%%
from math import sqrt, ceil
#%%
def TPourLogDiscret(P, g):
    """\
    Renvoie le tuple (P, Q, g, L, M)

    On ne vérifie pas que P est un nombre premier et g un générateur.
    On calcule :

    - Q = plus petit entier avec Q**2 au moins P.
    - L = la liste triée des couples ((g**Q)**i mod P, i) pour i de 0 à Q-1
    - M = la liste des couples (g**(-i) mod P, i) pour i de 0 à Q-1

    M n'est pas triée car on devra le faire plus tard après avoir
    multiplié tous ses éléments.

    Exemple :

    >>> T=TPourLogDiscret(13,2)
```

```
    >>> T
    (13, 4, 2, [(1, 0), (1, 3), (3, 1), (9, 2)],
               [(1, 0), (7, 1), (10, 2), (5, 3)])

    Remarque : le dernier élément de L est g**(Q(Q-1)). On peut avoir
    Q(Q-1) égal ou supérieur à P-1. Par exemple P=11, Q=4, Q(Q-1)=12.
    Donc L est peut-être plus long que nécessaire. Mais Q(Q-2)=(Q-1)**2-1
    est < P-1. Ainsi seul le dernier tuple de L peut être superflu.
    """
    Q = ceil(sqrt(P))  # plus petit entier avec Q**2 au moins P, donc > P-1.
    a = pow(g, Q, P)
    b = inversemodulo(g, P)
    L = sorted(listedepuissancesmoduloN(a, Q, P))
    M = listedepuissancesmoduloN(b, Q, P)
    return P, Q, g, L, M
#%%
def LogDiscret(x, T):
    """\
    Avec T = (P, Q, g, L, M), calcule n tel que g**n = x modulo P.

    On suppose x >= 0, et que T a été fait par TPourLogDiscret(P,g).

    Si P divise x, renvoie None.
    """
    P = T[0]
    if not x % P:
        return None
    LB = sorted((x*u[0] % P, u[1]) for u in T[4])
    i, j = pluspetitcommun(T[3], LB)
    return (T[1]*i + j) % (P-1)
    # le modulo P-1 est nécessaire car il n'est pas impossible que
    # le premier couple (i,j) trouvé donne Qi+j>= P-1.
    # Par exemple avec P=11, g=2, et x=5, on trouverait avec notre
    # algorithme n = 14 = 4*3+2 et non pas 4 = 4*1 + 0, la raison
    # étant ici que 2**12 mod 11 vaut 4 qui vient avant 2**4 mod 11
    # qui vaut 5. Donc l'algo trouve 5 = (2**12)*(2**2) mod 11
    # au lieu de 5 = (2**4)*(2**0) mod 11.
#%%
# à tester avec
# P = 7604629321
# g = 23
#
#
#%%
#In [53]: P=1021
#
#In [54]: L=[None]*P
#
#In [55]: x=1; L[1] = 0;
#
#In [56]: for i in range (1, P-1):
#    ...:     x=10*x % P; L[x]=i
#    ...:
#
#In [57]: L[2015%1021]
#Out[57]: 300
#
```

```
#In [58]: pow(10,300,1021)
#Out[58]: 994
#
#In [59]: 994+1021
#Out[59]: 2015
```

---

*Date de dernière modification* : 10-12-2015 à 18:22:41 CET.

# D

# Examen du 1er juin 2016



## D.1 Quelques rappels utiles pour cet examen

1. En Python `A % N` (que l'on lit `A modulo N`) donne si `N>0` le représentant entre `0` et `N-1` de la classe de congruence de `A` modulo `N`. Le résultat est toujours positif ou nul, même si `A<0`. On évitera de l'utiliser avec `N<0` car c'est toujours difficile de mémoriser pour chaque langage de programmation la convention suivie dans ce cas.
2. La fonction `pow()` [107] admet un troisième argument optionnel `N`, et alors `pow(A, n, N)` calcule une exponentiation modulaire `(A**n) % N` mais de manière beaucoup plus rapide que de calculer d'abord `A**n` (`A` à la puissance `n`) puis ensuite de réduire modulo `N`. On ne peut pas l'utiliser directement avec un exposant négatif, il faut d'abord pour cela calculer l'inverse multiplicatif de `A` modulo `N`, s'il existe.
3. Le code suivant:

```
def PGCD(a,b):
    """\
    Calcule le PGCD.

    Toujours positif. Nul dans le cas exceptionnel a==b==0.
    """
    a, b = abs(a), abs(b)
    while b:
        a, b = b, a%b
    return a
```

calcule le PGCD de deux nombres entiers relatifs. Le résultat est toujours positif ou nul. Lorsque `a` vaut zéro, le résultat est la valeur absolue de `b`, et vice versa. Le PGCD vaut zéro si et seulement si `a=b=0`.

107. https://docs.python.org/3/library/functions.html#pow

4. Avec `from random import randrange` dans son fichier Python, on peut ensuite dans les procédures faire `n=randrange(a,b)` pour obtenir un entier aléatoire supérieur ou égal à l'entier `a` et inférieur strictement à l'entier `b`.
5. On rappelle qu'avec Python3, il faut utiliser `//` pour la division entière. Sinon par exemple `4/2` calcule le flottant qui sera imprimé `2.0` et non pas l'entier 2.

## D.2 Exercice 1

1. Faites une procédure `syraF(x)`, qui étant donné un entier $x$ strictement positif calcule un nouvel entier strictement positif suivant la règle suivante :

```
notons y = x modulo 6.

si y vaut zéro, syraF(x) retourne x divisé par 6.
si y vaut un, syraF(x) retourne 11x+1.
si y vaut deux ou quatre, syraF(x) retourne x divisé par 2.
si y vaut trois, syraF(x) retourne x divisé par 3.
si y vaut cinq, syraF(x) retourne 11x-1.
```

2. Avec $x_0 = x$ on définit par récurrence $x_{n+1} = \text{syraF}(x_n)$. Faire une procédure `itersyraF(x, n)` qui imprime via `print()` toutes les valeurs depuis $x_0 = x$ jusqu'à $x_n$ (on pourra la conclure par `return None`).
3. On conjecture et c'est vérifié numériquement que quelque soit le point de départ $x$ il existe toujours un entier $n$ tel que $x_n = 1$. Faire une procédure `pluspetititéré(x)` qui retourne le $n$ en question.
4. En prenant des entiers de six chiffres au hasard via `x=randrange(100000,1000000)`, trouvez-en un qui nécessite au moins 200 itérations de `syraF(x)` avant d'obtenir pour la première fois la valeur 1.

## D.3 Exercice 2

1. On rappelle la notion de *Témoins de Miller* (page 97) étudiée dans la feuille de travaux pratiques numéro quatre http://math.univ-lille1.fr/~burnol/MAO/tp4.html. On considère le code suivant :

```
def PEPR(n,x):
    """\
    (description de ce que fait la procédure; n et x sont des entiers
    strictement positifs, avec n > 1 IMPAIR)
    """
    x = x%n
    k = 1
    m = (n-1)>>1
    while not m&1:
        k += 1
        m >>= 1
    y = pow(x,m,n)
    if y==1 or y==n-1:
         return True
    for j in range(k-1):
```

```
16          y=(y*y)%n
17          if y==n-1:
18              return True
19      return False
```

(a) Décrire ce que fait la procédure ligne par ligne.

(b) PEPR signifie « peut-être premier ». Expliquer en considérant acquises les explications sur les *Témoins de Miller* (page 97) de la Feuille de TP numéro quatre pourquoi si la routine renvoie `False` lorsque `x` n'est pas divisible par `n` alors on est sûr que l'entier `n` impair n'est pas un nombre premier.

2. Des calculs numériques exhaustifs ont montré que si l'entier $n > 1$ est inférieur strictement à 170584961, est non divisible par $2, 3, 5, 7, 29, 67, 679067$ (ce dernier est un nombre premier), et est « peut-être premier » pour $x = 350$ et $x = 3958281543$, alors il est véritablement un nombre premier. Coder la procédure `EstPetitPremier(n)` donnée en pseudo-code :

```
si n vaut 1 ou est supérieur ou égal à 170584961, renvoyer False.

s'il est divisible par 2, ou 3, ou 5, ou 7, ou 29, ou 67, ou 679067 :
  renvoyer True s'il est premier (c'est-à-dire l'un d'entre eux),
  renvoyer False sinon.

sinon :
  renvoyer True s'il est  « peut-être premier » à la fois pour
  x=350 et x=3958281543.
```

3. On considère le code suivant :

```
1   def UnFacteur(N):
2       j=0
3       while j<10:
4           a=randrange(1,N-2)
5           x=randrange(N)
6           y=x
7           while 1:
8               x=(x*x+a)%N
9               y=(y*y+a)%N
10              y=(y*y+a)%N
11              g=PGCD(x-y,N)
12              if g==N:
13                  print('.',end='')
14                  j += 1
15                  break
16              if g>1:
17                  return g
18      return N
```

(a) Expliquer ligne par ligne ce que fait ce code, et en particulier l'effet du `break` en ligne 15 (on ne demande pas d'expliquer à quoi sert l'algorithme, ni sur quel raisonnement ou heuristique de fonctionnement il repose).

(b) pour le `g` en ligne 11, quelles valeurs peut-il prendre ? peut-il être nul ? (on suppose `N>1`).

(c) Comment coder la boucle `while j<10:` avec un `for` ?

(d) Expliquer que la routine soit tourne éternellement, soit produit en valeur de retour `N`, soit produit comme valeur de retour un diviseur propre de `N`.

(e) (cette question est plus difficile que les précédentes et nécessite un petit raisonnement tenant compte en particulier du fait que toute itération d'une fonction sur un ensemble fini termine toujours par un cycle qui se répète) Justifier qu'en fait la routine ne peut pas tourner éternellement.

4. Faire deux routines `Factorise(N)` et `FactoriseImpair(N)` en implémentant le pseudo-code suivant :

```
si N vaut 1, Factorise renvoie None
si N est pair, Factorise imprime 2 et s'appelle récursivement
avec N divisé par 2
sinon, Factorise appelle FactoriseImpair

si EstPetitPremier(N) est True, FactoriseImpair imprime N et
fait return None

sinon, soit M=UnFacteur(N).

 - Si ce dernier vaut N, FactoriseImpair imprime la phrase
   "probablement pas de (petits) facteurs" et N
 - Sinon elle s'appelle récursivement sur M puis sur le quotient
   de N par M.

et elle renvoie None.
```

5. Exécuter `Factorise(273679535344943259768959949968)` et en déduire la factorisation en nombres premiers de cet entier.

## D.4 Corrigé

```python
# -*- coding: utf-8 -*-
"""
24 mai 2016 et 31 mai 2016

Corrigé (succinct) de l'examen de rattrapage.
"""
#%%
def syraF(x):
    y = x%6
    if y==0:
        return x//6
    elif y==1:
        return 11*x+1
    elif y==2:
        return x//2
    elif y==3:
        return x//3
    elif y==4:
        return x//2
    else:
        return 11*x-1
```

```
#%%
def itersyraF(x, n):
    print(x)
    # cette boucle aura n exécutions, la première imprime x_1
    # donc la dernière imprimera x_n
    for j in range(n):
        x=syraF(x)
        print(x)
    return None
#%%
def pluspetititéré(x):
    n=0
    while x!=1:
        n+=1
        x=syraF(x)
    return n
#%%
from random import randrange
def chercheX(N):
    """\
    Cherche au hasard un entier de 6 chiffres nécessitant au moins N itérations.
    """
    while 1: # boucle infinie
        x = randrange(100000,1000000)# entier avec six chiffres
        J = pluspetititéré(x)
        if J>=N:
            break
    return x, J
#%%
# chercheX(200)
# Out[9]: (725866, 211)
# chercheX(200)
# Out[10]: (542938, 247)
#%%
def PGCD(a,b):
    """\
    Calcule le PGCD.

    Toujours positif. Nul dans le cas exceptionnel a==b==0.
    """
    a, b = abs(a), abs(b)
    while b:
        a, b = b, a%b
    return a
#%%
def PEPR(n,x):
    """\
    Pour n impair. Si x n'est pas premier avec n, renverra nécessairement
    False. Si x est premier avec n, renvoie False si et seulement si
    x n'est PAS un témoin de Miller de non-primalité de n, autrement dit
    si n a une chance d'être un nombre premier au sens du test de Rabin
    Miller.
    """
    x = x%n # calcule x modulo n
    k = 1
    m = (n-1)>>1 # calcule (n-1)/2
```

```
    while not m&1: # tant que m est pair
        k += 1     # incrémente k, qui initialement valait 1
        m >>= 1    # remplace m par m divisé par 2
    # donc on a au final n - 1 = 2**k fois m avec m impair
    y = pow(x,m,n) # calcule y = x**m modulo n
    if y==1 or y==n-1:
         return True # si y vaut 1 ou -1 modulo n, renvoie True
    for j in range(k-1): # fait cette boucle k-1 fois
        y=(y*y)%n # remplace y par son carré modulo n
        # donc lors de la j ième itération on a x à la puissance
        # (2 à la puissance j) fois m modulo n.
        # À la dernière itération on a
        # donc le dernier y qui vaut x à la puissance (2**(k-1)m) modulo n
        # si l'un de ces y vaut -1 modulo n on arrête la boucle pour
        # renvoyer True
        if y==n-1:
            return True
    # on arrive ici si le y initial ne valait ni 1, ni -1 modulo n
    # et si aucun des carrés successifs ne valait -1 modulo n.
    return False

# Ainsi, en comparant au code de la feuille de TP 4 sur les témoins de Miller
# mais en faisant attention que l'on doit aussi supposer ici que la « base » x
# est un inversible modulo n, cela signifie que le code ci-dessus, lorsque x
# et n sont premiers entre eux, renvoie False si et seulement si x est un
# témoin de Miller de la non-primalité de n.

# Si la routine renvoie True, une certaine puissance de x était 1 ou -1 modulo
# n, donc x était forcément premier avec n (au début on a remplacé x par son
# résidu modulo n, mais cela ne change rien au fait d'être premier ou non avec
# n).

# La contraposée de (renvoie True) => x premier avec n est
#           (x pas premier avec n) => (renvoie False)

# Par exemple si x est n ou un multiple de n, la routine renvoie nécessairement
# False. On ne peut rien conclure alors sur n du fait que la routine a renvoyé
# False.

# Si par contre x est premier avec n, alors la routine renvoie False lorsque x
# est un témoin de Miller pour n, et alors on sait que cela veut dire que n
# n'est PAS un nombre premier.

# En conclusion si la routine renvoie False, on doit vérifier séparément en
# calculant D = PGCD(x,n) si par hasard x n'est pas premier avec n. Si 1 < D <
# n on a un diviseur de n, si D=1, on sait que x est un témoin de Miller de la
# NON-primalité de n (mais on ne connaît pas de diviseur), si D=n, c'est que
# le x était un multiple de n, et on ne peut rien conclure du tout sur n.

#%%
def EstPetitPremier(n):
    if n>=170584961 or n==1:
        return False
    for p in {2, 3, 5, 7, 29, 67, 679067}:
        if n%p==0:
            return n==p # ceci done True ssi n et p sont égaux
    return PEPR(n, 350) and PEPR(n, 3958281543)
```

```
#%%
def UnFacteur(N):
    j=0                    # initialise j
    while j<10:            # va faire une boucle au plus dix fois
        a=randrange(1,N-2) # a pris aléatoirement entre 1 et N-3 inclus
        x=randrange(N)     # x pris aléatoirement entre 0 et N-1 inclus
        y=x                # y initialisé comme valant x
        while 1:           # boucle infinie
            x=(x*x+a)%N    # remplace x par x^2+a modulo N
            y=(y*y+a)%N    # remplace deux fois y par y^2+a modulo N
            y=(y*y+a)%N
            g=PGCD(x-y,N)  # calcule le PGCD de x-y et N
            # x-y est entre -(N-1) et +(N-1)
            if g==N:       # si g=N, c'est que N divise x-y,
                           # donc que x=y.
                print('.',end='')# on imprime un point sans changer de ligne
                j += 1     # incrémente le compteur d'itérations
                break      # on revient au début du while j<10
            # ici g est entre 1 et N-1 au plus.
            if g>1:
                # ici g > 1. C'est un diviseur de N, pas égal à N
                return g
    # on arrive ici seulement si on a dix fois échoué à trouver un g<N.
    return N

# Si la routine se termine elle renvoie soit N, soit un g qui sera un diviseur
# propre de N.

# On va montrer que la routine se termine toujours. La seule façon de ne pas
# se terminer est que l'un des while 1: ne s'achève jamais. C'est donc que
# pour certains a et x (initial), on a ensuite x toujours distinct de y modulo
# N (car en cas d'égalité, alors g=N et on sort de la boucle par le break). Or
# si on note F(t) = (t*t + a) modulo N, et qu'on définit la suite x_n par
# x_{n+1} = F(x_n), avec x_0 = y_0 = x il est clair par récurrence que y_n =
# x_{2n}. Par l'indication de l'énoncé, l'itération de F se termine
# nécessairement par un cycle, car on est sur un ensemble fini. Si P est la
# longueur de ce cycle, pour tout n suffisamment grand on aura x_{n+P} = x_n et
# donc si n = kP est suffisamment grand on a x_{(k+1)P} = x_{kP} d'où en
# itérant aussi x_{2kP} = x_{kP}.
# Mais alors avec n=kP on a y_n = x_{2n} = x_n, contradiction.

#%%
def Factorise(N):
    if N==1:
        return None
    if N%2==0:
        print(2)
        Factorise(N//2)
    else:
        FactoriseImpair(N)
#
def FactoriseImpair(N):
    if EstPetitPremier(N):
        print(N)
        return None
    M=UnFacteur(N)
    if M==N:
```

```
        print("premier probable", N)
        return None
    FactoriseImpair(M)
    FactoriseImpair(N//M)
    return None
#%%

# Factorise(2736795353449432597689599499 68)
# 2
# 2
# 2
# 2
# 7
# 7
# 6277
# 6277
# 6277
# 10559113
# 133672613

# Donc 2^4 fois 7^2 fois 6277^3 fois 10559113 fois 133672613
```

*Date de dernière modification* : 01-06-2016 à 19:19:59 CEST.

# E

# Examen du 8 décembre 2016



## E.1 Avant de commencer

1. Recopier (par copier-coller) ceci dans votre fichier Python:

```
#%%
def pgcd(a,b):
    """\
    Calcule le PGCD.

    Toujours positif. Nul dans le cas exceptionnel a==b==0.
    """
    while b:
        a, b = b, a%b
    return abs(a)
```

ainsi que la liste des (430) nombres premiers inférieurs à 3000:

```
#%%
PREMIERS=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263,
269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347,
349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421,
431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499,
503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593,
599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661,
673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757,
761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853,
857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941,
```

```
947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013, 1019, 1021,
1031, 1033, 1039, 1049, 1051, 1061, 1063, 1069, 1087, 1091, 1093,
1097, 1103, 1109, 1117, 1123, 1129, 1151, 1153, 1163, 1171, 1181,
1187, 1193, 1201, 1213, 1217, 1223, 1229, 1231, 1237, 1249, 1259,
1277, 1279, 1283, 1289, 1291, 1297, 1301, 1303, 1307, 1319, 1321,
1327, 1361, 1367, 1373, 1381, 1399, 1409, 1423, 1427, 1429, 1433,
1439, 1447, 1451, 1453, 1459, 1471, 1481, 1483, 1487, 1489, 1493,
1499, 1511, 1523, 1531, 1543, 1549, 1553, 1559, 1567, 1571, 1579,
1583, 1597, 1601, 1607, 1609, 1613, 1619, 1621, 1627, 1637, 1657,
1663, 1667, 1669, 1693, 1697, 1699, 1709, 1721, 1723, 1733, 1741,
1747, 1753, 1759, 1777, 1783, 1787, 1789, 1801, 1811, 1823, 1831,
1847, 1861, 1867, 1871, 1873, 1877, 1879, 1889, 1901, 1907, 1913,
1931, 1933, 1949, 1951, 1973, 1979, 1987, 1993, 1997, 1999, 2003,
2011, 2017, 2027, 2029, 2039, 2053, 2063, 2069, 2081, 2083, 2087,
2089, 2099, 2111, 2113, 2129, 2131, 2137, 2141, 2143, 2153, 2161,
2179, 2203, 2207, 2213, 2221, 2237, 2239, 2243, 2251, 2267, 2269,
2273, 2281, 2287, 2293, 2297, 2309, 2311, 2333, 2339, 2341, 2347,
2351, 2357, 2371, 2377, 2381, 2383, 2389, 2393, 2399, 2411, 2417,
2423, 2437, 2441, 2447, 2459, 2467, 2473, 2477, 2503, 2521, 2531,
2539, 2543, 2549, 2551, 2557, 2579, 2591, 2593, 2609, 2617, 2621,
2633, 2647, 2657, 2659, 2663, 2671, 2677, 2683, 2687, 2689, 2693,
2699, 2707, 2711, 2713, 2719, 2729, 2731, 2741, 2749, 2753, 2767,
2777, 2789, 2791, 2797, 2801, 2803, 2819, 2833, 2837, 2843, 2851,
2857, 2861, 2879, 2887, 2897, 2903, 2909, 2917, 2927, 2939, 2953,
2957, 2963, 2969, 2971, 2999]
```

2. On rappelle qu'avec Python3, il faut utiliser `//` pour la division entière. Sinon par exemple `4/2` calcule le flottant qui sera imprimé `2.0` et non pas l'entier 2. De plus `divmod(a,b)` calcule à la fois le quotient et le reste dans la division euclidienne (`a` et `b` positifs).
3. Avec `from random import randrange` dans son fichier Python, on peut ensuite dans les procédures faire `n=randrange(a,b)` pour obtenir un entier aléatoire supérieur ou égal à l'entier `a` et inférieur strictement à l'entier `b`.
4. Les exercices se suivent.

## E.2 Exercice 1

1. Faire une fonction `OrdreDe10(b)` qui fera l'algorithme suivant:

```
on suppose (sans le vérifier) que l'argument b est un entier > 1.

si b n'est pas premier avec 10, renvoyer 0.

(ne pas être premier avec 10 signifie être divisible par 2 ou par 5)

sinon, poser u = 1, et k = 0.

   itérer u<- (10*u)%b, k<- k+1
   jusqu'à ce que u == 1 à nouveau.

   renvoyer k.
```

   Cette fonction calcule l'ordre de `10` dans le groupe multiplicatif des entiers modulo `b`.
2. Que vaut `OrdreDe10(67)` ?

3. Faire une fonction `LesOrdresDe10(L)` qui fera l'algorithme suivant:

```
on suppose que L est une liste d'entiers tous > 1, L = [b0, b1, ...]

la fonction renvoie la liste [OrdreDe10(b0), OrdreDe10(b1), ...]
```

4. Faire une procédure `PlusPetitAvecOrdreAuMoins(n)` qui trouvera le plus petit nombre premier `P<3000` tel que `OrdreDe10(P)` est au moins `n`. Si aucun n'est trouvé, renvoyer `None`. Attention, éviter d'évaluer à l'avance tous les `OrdreDe10(P)` pour `P` dans `PREMIERS` car cela pourrait prendre beaucoup de temps (notre fonction `OrdreDe10(P)` est très naïve).
5. Quel est le plus petit nombre premier renvoyé tel que l'ordre de 10 dans son groupe multiplicatif est au moins cent ? au moins deux cents ?

## E.3 Exercice 2

Cet exercice est dans la continuation de l'*Exercice 1* (page 152).

Étant donnés deux entiers premiers entre eux `a` et `b`, avec `0<a<b`, l'algorithme de l'école pour trouver les chiffres après la virgule (en base 10) de la fraction `a/b` prend la forme suivante:

```
D'abord on fait la division euclidienne de 10a par b: 10a = qb + r.

q est forcément 0, 1, ..., ou 9. C'est le premier chiffre après la
virgule.

Ensuite on fait la division euclidienne de 10r par b: 10r = q'b + r'

q' est le deuxième chiffre après la virgule. Etc...
```

1. Faire une procédure `ChiffresApresVirgule(a, b, N)`:

```
En input des entiers strictement positifs a, b, N.

Remplacer (a, b) par (a', b) avec a' le reste dans la division
euclidienne de a par b de sorte que a' < b.

ATTENTION: mes notations ici ne sont pas à reprendre en Python
puisque ' est réservé pour les chaînes de caractères.

Remplacer (a', b) par (a'', b') en divisant par le pgcd de a' et b.

Renvoyer une liste L = [q1, q2, ...., ] comportant les N premiers
chiffres après la virgule de l'écriture décimale de la fraction a/b.
```

Dans la description de la division ci-dessus, telle que pratiquée à l'école, on voit que après avoir calculé `n` chiffres, le numérateur initial a été remplacé par `(10**n)*a` modulo `b`. Lorsque le dénominateur `b` est premier avec 10, les puissances successives `10**n` sont dans le groupe multiplicatif modulo `b`, et (voir l'*Exercice 1* (page 152)) lorsque `n` atteint une certaine valeur `10**n` vaut 1 modulo `b` et par conséquent c'est comme si l'on était revenu au point départ. Ainsi `a/b` a un développement décimal périodique, la période commence juste après la virgule, ne dépend pas de `a` et a une longueur égale à `OrdreDe10(b)`.

**Attention :** (note ajoutée après la correction du sujet)

Comme ceci reprend la description initiale, il y est pré-supposé que `a` et `b` sont premiers entre eux, mais ce n'était PAS le cas dans la description de l'algorithme qui demandait **explicitement** de normaliser `a` et `b`, ce qui a été fait rarement. Si `b` n'est pas divisé par `pgcd(a,b)` d'abord, alors il est FAUX que la longueur de la période soit `OrdreDe10(b)`.

2. Faire une procédure `PeriodeApresVirgule(a,b)`:

```
En input des entiers strictement positifs a, b.

Si b n'est pas premier avec 10, renvoyer None.(*)

(*) lors de l'examen de rattrapage je vais sans doute ajouter ce
cas, la différence est que la période ne commence pas forcément
tout de suite après la virgule.

Remplacer (a, b) par (a', b) avec a' le reste dans la division
euclidienne de a par b de sorte que a' < b.

Remplacer (a', b) par (a'', b') en divisant par le pgcd de a' et b.

Renvoyer une liste L = [q1, q2, ...., ] qui comporte exactement
la période qui se répète dans le développement décimal de a/b,
après la virgule.
```

## E.4 Exercice 3

Cet exercice prend la suite du précédent.

Faire `VerifieTheoreme(a, P)`:

```
En input un nombre premier P (on ne le vérifie pas) et un nombre
entier a vérifiant 0 < a < P.

Si P est 2 ou 5 renvoyer True.

Si l'ordre de 10 dans le groupe multiplicatif de P (voir Exercice 1)
est impair, renvoyer True.

Si l'ordre de 10 est pair, et vaut 2N, calculer la liste L=[q1, q2,
..., q(2N)] des 2N chiffres après la virgule de la fraction a/P et
vérifier que la relation q(k)+q(k+N)=9 est toujours vraie pour k=1,
..., N. (attention qu'en Python les listes sont indicées à partir de
zéro, mais j'utilise 1 dans cet énoncé).

Si c'est le cas, renvoyer True. Sinon, renvoyer False.
```

Ensuite, faire une procédure `TEST(X)` qui va aléatoirement choisir `X` fois de suite un nombre premier `P` dans la liste `PREMIERS` (qui a longueur 430) et ensuite un nombre entier `a` tel que `0<a<P` et qui imprimera des lignes avec `a` et `P` puis `OK` si `VerifieTheoreme(a,P)` renvoie `True`, et sinon `a` et `P` suivi de `ERREUR: IMPOSSIBLE!!` et s'arrêtera là.

Par exemple

```
In [39]: TEST(10)
1220 / 1301 : OK
267 / 419 : OK
2024 / 2357 : OK
241 / 1061 : OK
217 / 229 : OK
1447 / 2339 : OK
160 / 229 : OK
1188 / 2503 : OK
270 / 277 : OK
734 / 1399 : OK
Out[39]: True
```

Des points bonus sont prévus pour ceux qui auront le temps de rédiger (sur une feuille à part) une démonstration mathématique de ce théorème.

## E.5 Corrigé

Les points ont été distribués comme suit entre les exercices:

- Exo 1: 8 points, 2+1+2+2+1
- Exo 2: 6 points, 3+3
- Exo 3: 6 points, 3+3

Pendant des semaines je n'ai pas osé corriger car j'ai cru que j'allais donner 20 à tout le monde, mais finalement non, ce n'est pas aussi catastrophique que cela, j'ai tout de même réussi à donner des trucs du genre 1/20 ou 3/20, ouf, bon, mais il y a quand même un bon paquet de notes entre 17 et 20,... faudra que je me rattrape la prochaine fois. Je fais de mon mieux moi, mais les étudiants travaillent et puis ils veulent de bonnes notes alors ils s'appliquent. Finalement j'étais assez content des copies dans l'ensemble, même que les étudiants ils font des commentaires eux, contrairement à moi ce qui n'est pas bien, mais j'étais trop fatigué au moment d'ajouter ce corrigé au repo, et maintenant c'est trop tard.

Bravo les étudiants, il y a même un code que certains ont trouvé qui est nettement plus Pythonesque que celui que j'avais préparé... mais faudra tout de même que je fasse un exercice plus difficile la prochaine fois. Par contre ils ont fait en masse une certaine erreur mathématique, mais lorsque j'ai corrigé les copies j'avais oublié pourquoi j'avais codé d'une certaine façon, et c'est seulement là maintenant le soir que je me rends compte du truc. Je vais devoir aller retirer des points... il n'y avait pas de piège proprement dit puisque toutes les directives étaient dans l'énoncé !

À propos le théorème sur les a/p, p premier s'appelle Théorème de Midy:

https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_Midy

Je ne connaissais pas cette référence en décembre 2016, mais c'est bien de pouvoir l'ajouter maintenant ici. Si vous cliquez sur le signe ci-dessous vous verrez une démonstration après le corrigé des exercices. Remarque: j'ai un théorème complet à ce sujet et il faudrait que je le rédige, il est possible que le résultat en ma possession ne soit pas dans la littérature (récréative...).

Il faudrait plus de commentaires (des « docstrings » en particulier).

```
# -*- coding: utf-8 -*-
"""8 décembre 2016

Corrigé de l'examen.
```

```
20 janvier 2017: meilleur code pour PlusPetitAvecOrdreAuMoins(n)

"""
#%%
def pgcd(a,b):
    """\
    Calcule le PGCD.

    Toujours positif. Nul dans le cas exceptionnel a==b==0.
    """
    while b:
        a, b = b, a%b
    return abs(a)

#%%
PREMIERS=[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263,
269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347,
349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421,
431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499,
503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593,
599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661,
673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757,
761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853,
857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941,
947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013, 1019, 1021,
1031, 1033, 1039, 1049, 1051, 1061, 1063, 1069, 1087, 1091, 1093,
1097, 1103, 1109, 1117, 1123, 1129, 1151, 1153, 1163, 1171, 1181,
1187, 1193, 1201, 1213, 1217, 1223, 1229, 1231, 1237, 1249, 1259,
1277, 1279, 1283, 1289, 1291, 1297, 1301, 1303, 1307, 1319, 1321,
1327, 1361, 1367, 1373, 1381, 1399, 1409, 1423, 1427, 1429, 1433,
1439, 1447, 1451, 1453, 1459, 1471, 1481, 1483, 1487, 1489, 1493,
1499, 1511, 1523, 1531, 1543, 1549, 1553, 1559, 1567, 1571, 1579,
1583, 1597, 1601, 1607, 1609, 1613, 1619, 1621, 1627, 1637, 1657,
1663, 1667, 1669, 1693, 1697, 1699, 1709, 1721, 1723, 1733, 1741,
1747, 1753, 1759, 1777, 1783, 1787, 1789, 1801, 1811, 1823, 1831,
1847, 1861, 1867, 1871, 1873, 1877, 1879, 1889, 1901, 1907, 1913,
1931, 1933, 1949, 1951, 1973, 1979, 1987, 1993, 1997, 1999, 2003,
2011, 2017, 2027, 2029, 2039, 2053, 2063, 2069, 2081, 2083, 2087,
2089, 2099, 2111, 2113, 2129, 2131, 2137, 2141, 2143, 2153, 2161,
2179, 2203, 2207, 2213, 2221, 2237, 2239, 2243, 2251, 2267, 2269,
2273, 2281, 2287, 2293, 2297, 2309, 2311, 2333, 2339, 2341, 2347,
2351, 2357, 2371, 2377, 2381, 2383, 2389, 2393, 2399, 2411, 2417,
2423, 2437, 2441, 2447, 2459, 2467, 2473, 2477, 2503, 2521, 2531,
2539, 2543, 2549, 2551, 2557, 2579, 2591, 2593, 2609, 2617, 2621,
2633, 2647, 2657, 2659, 2663, 2671, 2677, 2683, 2687, 2689, 2693,
2699, 2707, 2711, 2713, 2719, 2729, 2731, 2741, 2749, 2753, 2767,
2777, 2789, 2791, 2797, 2801, 2803, 2819, 2833, 2837, 2843, 2851,
2857, 2861, 2879, 2887, 2897, 2903, 2909, 2917, 2927, 2939, 2953,
2957, 2963, 2969, 2971, 2999]
#%%
def OrdreDe10(b):
    if pgcd(10,b)>1:
        return 0
```

```
    u=1
    k=0
    # je ne suis pas exactement les indications du Professeur.
    u=(10*u)%b
    k+=1
    while u>1:
        u, k = (10*u)%b, k+1
    return k
#In [2]: OrdreDe10(67)
#Out[2]: 33
#%%
def LesOrdresDe10(L):
    return [OrdreDe10(b) for b in L]
#%%
#
# Cette version a été trouvée par plusieurs étudiant(e)s. Le point
# est que PREMIERS, de type list, est déjà ordonné et est parcouru
# par "for" dans l'ordre naturel, donc ça fonctionne bien pour
# trouver le premier qui marche et arrêter la boucle à ce moment
# là.
def PlusPetitAvecOrdreAuMoins(n):
    for p in PREMIERS:
        if OrdreDe10(p)>=n:
            return p
    return None
#
# La version du Professeur que voici était nettement moins
# Pythonesque, et j'ai donné un point supplémentaire au delà
# du barème pour ceux qui l'ont fait « mieux que le prof ».
#
# def PlusPetitAvecOrdreAuMoins(n):
#     k=0
#     K=len(PREMIERS)
#     while 1:
#         P=PREMIERS[k]
#         if OrdreDe10(P)>=n:
#             return P
#         k+=1
#         if k==K:
#             return None
#In [3]: PlusPetitAvecOrdreAuMoins(100)
#Out[3]: 109
#
#In [4]: PlusPetitAvecOrdreAuMoins(200)
#Out[4]: 223
#%%
def ChiffresApresVirgule(a, b, N):
# attention beaucoup d'étudiants n'ont pas fait les réductions initiales
# qui pourtant étaient demandées explicitement.
    a=a%b
    d=pgcd(a,b)
    a=a//d
    b=b//d
    L = []
    k=0
    while k<N:
        q, a = divmod(10*a,b)
```

```
        L.append(q)
        k += 1
    return L
#%%
def PeriodeApresVirgule(a,b):
#
# Pourquoi n'ai-je pas fait ChiffresApresVirgule
# (a,b,OrdreDe10(b))? ah, j'avais oublié pendant ma correction des
# copies pourquoi j'avais codé comme ceci, mais ça me turlupinait
# un peu (comme un bruit de fond parasite dont je n'arrivais pas à
# me débarrasser), j'avoue. La réponse est simple la période de
# a/b est OrdreDe10(b) seulement si la fraction est
# irréductible... il faut absolument d'abord réduire la fraction
# (comme demandé dans l'énoncé), et seulement après on évalue
# OrdreDe10(b). Et une fois qu'on a fait tous ces efforts appeler
# ChiffresApresVirgule paraît sous-optimal car il va faire un pgcd
# (qui donnera 1) pour rien. Du coup j'avais rédigé comme suit:
#
    if pgcd(10,b)>1:
        return None
    a=a%b
    d=pgcd(a,b)
    a=a//d
    b=b//d
    L = []
    k=0
    N=OrdreDe10(b)
    while k<N:
        q, a = divmod(10*a,b)
        L.append(q)
        k += 1
    return L
#
# Mémo: aller retirer des points dans les copies des étudiants à qui
# j'ai donné le maximum alors qu'ils ont fourni la réponse fausse
# ChiffresApresVirgule (a,b,OrdreDe10(b)) commettant ainsi une erreur
# mathématique !! (ça m'était sorti de la tête pendant la correction,
# et c'est une bonne illustration qu'il faut commenter son code...)
# J'avais déjà retiré 0,5pt au moins car ils n'ont pas suivi l'énoncé
# qui demandait de normaliser a, b d'abord. Mais je vais sévir un peu plus.

#In [6]: PeriodeApresVirgule(1,7)
#Out[6]: [1, 4, 2, 8, 5, 7]
# période paire, 1+8=9, 4+5=9, 2+7=9

#In [7]: PeriodeApresVirgule(3,19)
#Out[7]: [1, 5, 7, 8, 9, 4, 7, 3, 6, 8, 4, 2, 1, 0, 5, 2, 6, 3]
# période paire aussi et ça marche

#%%
def VerifieTheoreme(a, P):
    if P==2 or P==5:
        return True
    M = OrdreDe10(P)
    if M&1:
        # M impair
        return True
```

```
    # on connaît déjà la période, évitons de la calculer une deuxième fois.
    L = ChiffresApresVirgule(a, P, M)
    N=M>>1
    # ça serait plus spectaculaire en faisant aussi print(L[i]+L[N+i]).
    # à propos le théorème marche aussi avec P**2, P**3, P**4...
    for i in range(N):
        if L[i]+L[N+i]!=9:
            return False
    return True
#In [8]: PeriodeApresVirgule(31,69)
#Out[8]: [4, 4, 9, 2, 7, 5, 3, 6, 2, 3, 1, 8, 8, 4, 0, 5, 7, 9, 7, 1, 0, 1]
# période paire (22 chiffres) et ça ne marche pas ??? ah oui 69 n'est pas un
# nombre premier !!!

#In [12]: L=PeriodeApresVirgule(17,31)
#
#In [13]: print(L, "longueur=", len(L))
#[5, 4, 8, 3, 8, 7, 0, 9, 6, 7, 7, 4, 1, 9, 3] longueur= 15
# celui-ci a une période 15 (la moitié de 30=31-1); en particulier
# 10 n'est pas un générateur du groupe cyclique (Z/31Z)*.
#%%
from  random import randrange
#%%
def TEST(X):
    for x in range(X):
        P = PREMIERS[randrange(0,430)]
        a = randrange(1,P)
        if VerifieTheoreme(a,P):
            print(a, "/", P, ": OK")
            # ça serait plus sympathique d'imprimer la période pour un
            # contrôle visuel, mais elle peut être longue...
        else:
            print(a, "/", P, ": ERREUR IMPOSSIBLE!!")
            # on s'arrête tout de suite, mais n'arrivera jamais...
            return False
    return True
#In [14]: TEST(10)
#183 / 601 : OK
#18 / 23 : OK
#274 / 347 : OK
#153 / 211 : OK
#126 / 167 : OK
#2347 / 2411 : OK
#400 / 1229 : OK
#390 / 2503 : OK
#141 / 179 : OK
#199 / 1993 : OK
#Out[14]: True
```

En ce qui concerne la question mathématique, supposons que la période est paire $2N$ notons $X$ l'entier correspondant aux $N$ premiers chiffres et $Y$ celui correspondant aux $N$ suivants, de sorte que $a/p = 0, XYXYXYXY\ldots$. Je rappelle que $0.1111.. = 1/9$, $0.01010101.. = 1/99$, $0.001001001.. = 1/999$ etc… (série géométrique ou raisonnement de l'école primaire), donc $a/p = (X \cdot 10^N + Y)/(10^{2N} - 1)$. Or la fraction $a/p$ est irréductible donc $p$ divise $10^{2N} - 1 = (10^N - 1)(10^N + 1)$, ce que l'on sait d'ailleurs déjà puisque $2N$ est l'ordre de 10 modulo $p$. Par contre $p$ ne divise pas $10^N - 1$ sinon l'ordre de 10 serait non pas $2N$ mais un diviseur de $N$. Donc $p$ divise $10^N + 1$, disons

$10^N + 1 = Qp$, et alors $X \cdot 10^N + Y = (10^{2N} - 1)a/p = (10^N - 1)aQ$.

Réduisant modulo $10^N - 1$, on obtient $X + Y \equiv 0 \pmod{10^N - 1}$, c'est-à-dire que $X + Y$ est un multiple de $10^N - 1$. On ne peut pas avoir $X + Y = 0$ (l'un des deux est non nul), et on ne peut pas non plus avoir $X = Y = 99..99$ donc au moins l'un des deux est $< 10^N - 1$, et la seule possibilité est $X + Y = 10^N - 1$. Il reste à expliquer que si deux nombres ont une somme qui ne comporte que des 9, alors les chiffres se complètent un par un pour faire 9: on commence par regarder les derniers chiffres $x$ et $y$, on sait que $x + y = 9$ ou $x + y = 19$, mais ce dernier cas est impossible donc $x + y = 9$, et ensuite on remonte vers la gauche.

Le théorème marche aussi pour des fractions du type $a/p^r$, le principe est le même à savoir de déduire du fait que $p^r$ divise $10^{2N} - 1 = (10^N - 1)(10^N + 1)$ mais pas $10^N - 1$ qu'en réalité $p^r$ divise $10^N + 1$. Il faut réfléchir un peu mais notez que la différence des deux termes est 2, ce qui peut servir.

## E.6 Exercice 4

Nota Bene: *cet exercice n'était pas inclus dans l'examen, car le Professeur a eu peur, au tout dernier moment, de faire trop long; a posteriori le Professeur a eu raison, c'était déjà bien comme cela les 3 exercices en moins de deux heures.*

Note: la fonction `pow()` [108] admet un troisième argument optionnel `N`, et alors `pow(A, n, N)` calcule très rapidement $A^n \pmod N$ (`(A**n)%N`), mais uniquement pour `n` positif.

N'abordez cet exercice que si vous avez vraiment fait tous les autres.

Si $P$ est un nombre premier on sait que le groupe multiplicatif de $\mathbb{Z}/P\mathbb{Z}$ est *cyclique*. Un élément $g$ est un générateur si le plus petit exposant tel que $g^n = 1 \pmod P$ est $n = P - 1$. Pour tester si cela est réalisé il suffit de vérifier $g^n \neq 1 \pmod P$ pour tous les $n = (P - 1)/Q$ et chaque nombre premier $Q$ qui divise $P - 1$.

En utilisant la liste `PREMIERS` écrire une procédure efficace `EstGenerateur(g,P)` qui fera l'algorithme suivant:

```
on suppose (sans le vérifier) que P est premier < 3000**2=9000000.

si g n'est pas premier avec P renvoyer False.

sinon, il faut trouver de manière efficace l'un après l'autre les
diviseurs premiers Q de Y=P-1 (à chaque fois qu'un Q est trouvé,
diviser Y par Q autant de fois que possible; lorsqu'un nouveau Q
est testé avec un Y encore >1 et que la division euclidienne
donne Y=qQ+r avec un r>0, alors si Y n'est pas premier il est au
moins (Q+2)^2 (je suppose ici Q au moins 3) ce qui est impossible
si q est au plus Q+3, ainsi le test q <= Q+3 est un critère d'arrêt
pour affirmer que le dernier Y est premier --- dans le cas où le
plus grand nombre premier Q qui divise le Y=P-1 initial le fait
avec multiplicité, l'algorithme n'emprunte jamais cette branche
et s'arrête avec Y ramené à 1.)

bref, trouver les Q un par un et à chaque fois tester si g**((P-1)/Q)
modulo P vaut 1.

   si c'est le cas, s'arrêter et renvoyer False
```

108. https://docs.python.org/3/library/functions.html#pow

```
   sinon continuer avec le prochain Q.

si aucun Q n'est trouvé avec g**((P-1)/Q) modulo P valant 1 renvoyer True.
```

Finalement, indiquer parmi les 200 premiers nombres premiers:

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263,
269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347,
349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421,
431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499,
503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593,
599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661,
673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757,
761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853,
857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941,
947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013, 1019, 1021,
1031, 1033, 1039, 1049, 1051, 1061, 1063, 1069, 1087, 1091, 1093,
1097, 1103, 1109, 1117, 1123, 1129, 1151, 1153, 1163, 1171, 1181,
1187, 1193, 1201, 1213, 1217, 1223
```

ceux pour lesquels 10 est un générateur (c'est-à-dire les P pour lesquels les fractions a/P ont des périodes de longueur P-1).

---

*Date de dernière modification* : 22-01-2017 à 23:14:19 CET.

# F

# Examen du 8 juin 2017



## F.1 Avant de commencer

1. On rappelle qu'avec Python3, il faut utiliser `//` pour la division entière. Sinon par exemple `4/2` calcule le flottant qui sera imprimé `2.0` et non pas l'entier 2.
2. Avec `from random import randrange` dans son fichier Python, on peut ensuite dans les procédures faire `n=randrange(a,b)` pour obtenir un entier aléatoire supérieur ou égal à l'entier `a` et inférieur strictement à l'entier `b`.
3. Un entier positif $n$ s'écrit avec $N$ chiffres en base 10 si et seulement si $10^{N-1} \leqslant n < 10^N$.

## F.2 Exercice 1

1. Voici un algorithme en pseudo-code :

```
En entrée deux entiers positifs N et M

Initialiser X=1

Boucle:

 si N=0 renvoyer X fois M
 sinon, si M=0 renvoyer X fois N
 sinon, si N et M sont pairs, N<--N/2, M<--M/2, X<--2X
 sinon, si N est pair, N<--N/2
 sinon, si M est pair, M<--M/2
 sinon, si N>=M, N<--N-M
 sinon, M<--M-N
```

Attention que les notations utilisées dans cette description ne sont pas celles de Python. Implémenter cet algorithme dans une fonction `mafonction(N,M)`.

2. Vérifier que `mafonction(137205626, 12539182)` renvoie `3778`.
3. Que calcule cet algorithme ?
4. Rédiger par écrit une démonstration mathématique justifiant votre réponse.

## F.3 Exercice 2

Voici une implémentation de l'algorithme d'Euclide:

```
def pgcd(a, b):
    """\
    Calcule le PGCD via l'algorithme classique d'Euclide.
    """
    a, b = abs(a), abs(b)
    while b:             # tant que b est non nul
        a, b = b, a % b  # a<-b, b<- reste dans division euclidienne de a par b
    return a
    # à la fin, on renvoie le « dernier reste non nul » ...
```

On dit que deux entiers sont premiers entre eux si leur PGCD vaut 1.

1. Faites une fonction `probapremiersentreeux(N, reps)` qui prend en entrée un argument entier `N` et un argument optionnel `reps` de valeur par défaut `1000` et qui implémentera le calcul suivant:

```
On prend, reps fois de suite, au hasard deux entiers positifs
a et b de chacun N chiffres.

On renvoie

  "le nombre total de fois que a et b sont premiers entre eux"/reps

La valeur de retour est donc un nombre de type float.
```

2. Que renvoie `probapremiersentreeux(N)` pour `N = 10, 50, 100` ?

## F.4 Exercice 3

On définit une fonction mathématique $f(u, v)$ qui à tout couple d'entiers $(u, v)$ associe un nouveau couple $(2u + 5v, u + 2v)$.

On note $F_k(u, v)$ les fonctions qui vérifient les récurrences suivantes:

1. $F_0(u, v) = (u, v)$
2. $F_1(u, v) = f(u, v) = (2u + 5v, u + 2v)$
3. $F_{k+1}(u, v) = f(F_k(u, v))$

En fait le point 2. est un cas particulier du point 3, et $F_k$ est simplement la $k^{\text{e}}$ itérée de la fonction $f$.

On écrira aussi $F(u, v, k) = F_k(u, v)$.

On s'intéressera en particulier aux couples $(u_n, v_n) = F(1, 0, n)$. Ainsi $(u_0, v_0) = (1, 0)$, $(u_1, v_1) = (2, 1)$, $(u_2, v_2) = (9, 4)$, …

1. Implémenter en Python une `fonctionF(u, v, k)` qui renvoie la valeur de $F_k(u, v)$.
2. Que renvoient `fonctionF(1, 0, 8)` et `fonctionF(1, 0, 15)` ?
3. On admet que pour les couples $(u_n, v_n)$ définis ci-dessus on a les formules de récurrence $u_{2m} = u_m^2 + 5v_m^2$ et $v_{2m} = 2u_m v_m$.
   Implémenter une fonction `fonctionG(k)` qui renvoie le couple $(u_k, v_k)$ selon l'algorithme suivant :

```
si k=0 renvoyer (1, 0)
si k est pair (k = 2m), calculer (appel récursif) le couple fonctionG(m)
   et lui appliquer les formules de récurrences indiquées
si k est impair (k = 2m+1), calculer G(k) par la formule f(G(2m))
```

4. Vérifier que `fonctionG(8)` et `fonctionG(15)` donnent les résultats escomptés.
5. Calculer le $(U, V)$ qui est renvoyé par `fonctionG(1000)`. Indiquer sur votre copie les six premiers et six derniers chiffres de $U$ et $V$.
6. Que calcule Python comme valeur pour $U^2 - 5V^2$ ?
7. Rédiger par écrit une démonstration mathématique du résultat précédent.

---

*Date de dernière modification* : 08-06-2017 à 11:56:46.

# G

# Examen du 12 décembre 2017



## G.1 Quelques rappels

1. La syntaxe `return x, y` équivaut à `return (x, y)` et l'objet renvoyé est le `tuple`[109] `(x, y)`.
2. La syntaxe `a, b, c = (x, y, z)` équivaut à `a, b, c = x, y, z` et fait les affectations (simultanées) `a = x`, `b = y`, `c = z`.
3. Après `L = [x] + [y, z]` la `list`[110] `L` vaut `[x, y, z]`. Après `L = [x]` puis `L.extend([y, z])` aussi. Par contre `L = [x].extend([y, z])` n'assigne pas à `L` un type de `list`[111] mais plutôt le rend égal à `None`[112] ce qui n'est sûrement pas le but recherché. Ce qui se passe c'est que `extend()`[113] est une méthode de l'objet `[x]` de type `list`[114], et modifie cet objet tout en renvoyant `None`[115]. On récupère donc `None`[116] et on a perdu toute référence à la liste `[x, y, z]` pourtant créée quelque part.
4. En Python3, on utilise `//` pour la division entière, par exemple `4//2` donne `2` tandis que `4/2` donne un `float`[117] imprimé `2.0` qui n'est pas le `2` de classe `int`[118] et peut invalider la

109. https://docs.python.org/3/library/stdtypes.html#tuple
110. https://docs.python.org/3/library/stdtypes.html#list
111. https://docs.python.org/3/library/stdtypes.html#list
112. https://docs.python.org/3/library/constants.html#None
113. https://docs.python.org/3/tutorial/datastructures.html?highlight=list%20methods#more-on-lists
114. https://docs.python.org/3/library/stdtypes.html#list
115. https://docs.python.org/3/library/constants.html#None
116. https://docs.python.org/3/library/constants.html#None
117. https://docs.python.org/3/library/functions.html#float
118. https://docs.python.org/3/library/functions.html#int

suite de l'exécution d'une procédure conçue pour des `int` [119].

5. On rappelle que `divmod()` [120] est une fonction de deux variables qui, dans le cas de deux `int` [121] a et b renvoie le `tuple` [122] `(a // b, a % b)` avec l'avantage de n'avoir fait la division en interne qu'une seule fois.
6. Avec `from random import randrange` on peut utiliser `randrange(a, b)` pour obtenir un entier aléatoire supérieur ou égal à l'entier a et inférieur strictement à l'entier b.

## G.2 Exercice 1

Voici un algorithme écrit en pseudo-code:

```
ajouter(L, M)
-------------

en entrée : deux listes dont les éléments sont des entiers positifs < 7

en sortie : une liste dont les éléments sont des entiers positifs < 7

Faire longL <-- len(L) et longM <-- len(M)

Si longM < longL alors échanger L et M ainsi que longL et longM

Faire K <-- M[:] et e <-- 0

Boucle pour 0 <= i < longL :
    Faire x <-- L[i] + M[i] + e
    Puis K[i] <-- x si x < 7 et x - 7 sinon.
    Et aussi e <-- 0 si x < 7 et 1 sinon

Boucle pour longL <= j < longM :
    Si e est nul faire break
    Sinon faire x <-- M[j] + e.
    Si alors x < 7, faire K[j] <-- x et e <-- 0,
    sinon faire K[j] <-- 0 et e <-- 1.

Si e vaut 1, faire K.append(1)

Renvoyer K
```

1. Implémenter en Python3 cette fonction `ajouter(L, M)`.
2. Vérifier que `ajouter([6, 6, 6, 6], [2, 3, 4])` et aussi `ajouter([2, 3, 4], [6, 6, 6, 6])` renvoient `[1, 3, 4, 0, 1]`. Sinon il y a une erreur dans votre implémentation.
3. Écrire une fonction `machin(L)` suivant l'algorithme suivant:

```
en entrée : une liste L d'entiers

en sortie : un entier

```

119. https://docs.python.org/3/library/functions.html#int
120. https://docs.python.org/3/library/functions.html#divmod
121. https://docs.python.org/3/library/functions.html#int
122. https://docs.python.org/3/library/stdtypes.html#tuple

```
Initialisation S <-- 0, T <-- 1

Boucle pour 0 <= i < len(L) :
    S <-- S + L[i] * T
    T <-- 7 * T

Renvoyer S
```

4. Vérifier que `machin([6, 6, 6, 6])` donne `2400` et `machin([2, 3, 4])` donne 219. D'une manière générale lorsque `machin(L)` donne un entier `N` quelle est l'expression mathématique de `N` en fonction des éléments $x_i =$ `L[i]` de `L` pour `i = 0, ..., len(L) - 1` ?
5. *Lorsque les éléments entiers* $x_i =$ `L[i]` de `L` *vérifient* $\forall i, 0 \leqslant x_i < 7$ comment appelle-t-on ces éléments en fonction de l'entier `N` calculé par `machin(L)` ?
6. On considère la fonction suivante :

```python
def bidule(L):
    """Reconstruction d'un entier

    >>> bidule([6, 6, 6, 6])
    2400
    >>> bidule([2, 3, 4])
    219
    >>> bidule([2, 3, 4, 0, 0, 0, 0, 0, 0, 0])
    219
    """
    S = 0
    while L:
        S = 7 * S + L.pop()
    return S
```

   À votre avis, que fait ce `bidule(L)` ? Justifier.
7. Écrire une procédure `truc(N)` qui à partir d'un entier positif `N` donne une liste `L` (dont les éléments sont des entiers compris entre `0` et `6` inclus) telle que `machin(L)` redonne `N`.

   Par exemple, on peut définir récursivement par `[N % 7] + truc(N // 7)` si `N >= 7` et `[N]` si `N < 7`. Mais utiliser plutôt `divmod(N, 7)` pour obtenir quotient et reste.
8. Que renvoie `machin(ajouter(truc(N), truc(M)))` ?

## G.3 Exercice 2

1. On étudier la suite des entiers $(v_n)_{n\geqslant 0}$ vérifiant la récurrence $v_0 = 0$, $v_1 = 1$ et $v_{n+1} = 3v_n + 5v_{n-1}$.

   Écrire une procédure `lasuite(N)` qui, pour `N` entier positif (y-compris zéro) renvoie la `list`[123] `L` de longueur `N` débutant par `[0, 1, 3, 14, 57, 241, ...]` et telle que `L[i]` est l'entier $v_i$.

   Pour `N` nul, `lasuite(0)` renvoie `[]`.

   Par la suite (sic) on travaillera avec `Kcent = lasuite(100)`.
2. Construire la `list`[124] `Kcentlongueurs` qui donne les longueurs (de l'écriture en base 10) des éléments de `Kcent` (on ne l'utilisera pas dans les questions suivantes).

123. https://docs.python.org/3/library/stdtypes.html#list
124. https://docs.python.org/3/library/stdtypes.html#list

Devinez-vous sur le résultat une formule approximative pour la longueur de $v_n$ en base 10, en fonction de $n$ ?

On ne demande pas quelque chose de très précis. On pourra comparer les longueurs pour $n = 99$ et $n = 50$ et $n = 25$ …

3. On rappelle la procédure :

```python
def pgcd(a, b):
    """Calcule le PGCD par l'algorithme d'Euclide.

    Le résultat est positif, et nul uniquement si a==b==0. Les arguments
    sont des entiers éventuellement négatifs.

    Exemples
    --------

    >>> pgcd (121, 143)
    11
    >>> pgcd (-1000, -225)
    25
    """
    while b:
        a, b = b, a % b
    return abs(a)
```

Vérifier que `pgcd(Kcent[99], Kcent[77])` vaut `1306469` qui est aussi `Kcent[11]`.

4. Écrire une procédure `valide(reps)` qui `reps` fois de suite choisit `a` et `b` des entiers au hasard entre 0 et 99 (inclus) et vérifie que `pgcd(Kcent[a], Kcent[b]) == Kcent[pgcd(a, b)]`.

   La procédure imprime `OK` à la fin. Si un contre-exemple est trouvé elle s'interrompt prématurément et imprime les valeurs de `a` et `b`. Mais cela n'arrivera jamais.

## G.4 Exercice 3

Voici un algorithme écrit en pseudo-code :

```
foobar(N)
---------

en entrée : un entier positif N (on ne le vérifie pas)

en sortie : deux entiers I, J

si N est inférieur ou égal à 3, FIN: renvoyer 1, N

initialisation : I, J égaux à 1, N respectivement

boucle :

    tant que J est divisible par 4
    faire I <-- 2 * I, J <-- J // 4

P <-- 3
Q <-- 9
K <-- J
```

```
si K est pair faire K <-- K // 2

tant que Q est inférieur ou égal à K :

    boucle :

        tant que K est divisible par Q
        faire I <-- P * I, J <-- J // Q, K <-- K // Q

    si K est divisible par P, faire K <-- K // P

    faire P <-- P + 2
    faire Q <-- P ** 2

FIN: renvoyer I, J
```

1. Implémenter en Python3 la fonction `foobar(N)`.
   On essaiera de ne pas calculer plusieurs fois les mêmes quantités, en s'aidant au besoin de variables supplémentaires.
2. Quel est le `tuple`[125] renvoyé par `foobar(1000)`, et par `foobar(18577877449223257)` ?
3. Si l'on supprime le test pour `N <= 3` l'algorithme peut tomber dans une boucle infinie; dans quel cas précisément ?
4. Que calcule en fait cet algorithme ?
5. Rédiger par écrit une démonstration mathématique justifiant votre réponse à la question précédente.

## G.5 Corrigé

C'est succinct car je n'ai pas vraiment fait les commentaires de code et surtout je ne détaille pas certaines démonstrations mathématiques autant qu'il fallait le faire à l'examen.

### G.5.1 Exercice 1

1.
```python
def ajouter(L, M):
    """Fait l'addition de deux listes, représentant des chiffres en base 7

    >>> ajouter([2, 3, 4], [6, 6, 6, 6])
    [1, 3, 4, 0, 1]
    >>> ajouter([6, 6, 6, 6], [2, 3, 4])
    [1, 3, 4, 0, 1]
    """
    longL = len(L)
    longM = len(M)
    if longM < longL:
        L, M = M, L
        longL, longM = longM, longL
    K = M[:]
```

125. https://docs.python.org/3/library/stdtypes.html#tuple

```
    e = 0
    for i in range(longL):
        x = L[i] + M[i] + e
        if x > 7:
            x = x - 7
            e = 1
        else:
            e = 0
        K[i] = x
    for j in range(longL, longM):
        if e == 0:
            break
        x = M[j] + e
        if x == 7:
            x = 0
            e = 1
        else:
            e = 0
        K[j] = x
    if e == 1:
        K.append(1)
    return K
```

2. Mon code marche.

3. 
```
def machin(L):
    """Reconstruction d'un entier à partir de ses chiffres en base 7

    >>> machin([6, 6, 6, 6])
    2400
    >>> machin([2, 3, 4])
    219
    """
    S, T = 0, 1
    for i in range(len(L)):
        S += L[i] * T
        T *= 7
    return S
```

4. L'expression mathématique est $N = \sum_{0 \leqslant i < \mathrm{len}(L)} x_i 7^i$. On peut en faire la démonstration par récurrence sur la longueur `len(L)`.
5. On les appelle les « chiffres de l'écriture de l'entier `N` en base 7 », les moins significatifs venant en premier dans l'objet de type `list` [126].
6. La procédure :

```
def bidule(L):
    """Reconstruction d'un entier

    >>> bidule([6, 6, 6, 6])
    2400
    >>> bidule([2, 3, 4])
    219
    >>> bidule([2, 3, 4, 0, 0, 0, 0, 0, 0, 0])
    219
    """
```

126. https://docs.python.org/3/library/stdtypes.html#list

```
    S = 0
    while L:
        S = 7 * S + L.pop()
    return S
```

renvoie la même chose que `machin(L)` mais a un effet secondaire spectaculaire qui est de remplacer la liste passée en argument par la liste vide. Dans la vraie vie on aurait commencé par faire un `M = L[:]` et on aurait travaillé avec ce `M` pour éviter de modifier le `L` passé en argument à la procédure.
La justification se fait aussi par récurrence, en vérifiant la validité de la formule mathématique précédente.

7.
```
def truc(N):
    """Chiffres en base 7, récursivement

    >>> truc(5)
    [5]
    >>> truc(47)
    [5, 6]
    >>> truc(2400)
    [6, 6, 6, 6]
    >>> truc(219)
    [2, 3, 4]
    """
    if N < 7:
        return [N]
    q, r = divmod(N, 7)
    return [r] + truc(q)
```

8. `machin(ajouter(truc(N), truc(M)))` renvoie l'entier `N+M`.

### G.5.2 Exercice 2

1.
```
def lasuite(N):
    """renvoie la liste aux entrées récurrentes [0, 1, 3, 14, 57, ... ]

    La récurrence est :math:`v_{n+1} = 3v_n + 5v_{n-1}`.
    >>> lasuite(0)
    []
    >>> lasuite(1)
    [0]
    >>> lasuite(5)
    [0, 1, 3, 14, 57]
    >>> lasuite(7)
    [0, 1, 3, 14, 57, 241, 1008]
    """
    if N == 0:
        return []
    elif N == 1:
        return [0]
    elif N == 2:
        return [0, 1]
    L = [0, 1]
    u, v = 0, 1
    for j in range(2, N):
```

```
        u, v = v, 3 * v + 5 * u
        L.append(v)
    return L
```

2. (retours à la ligne ajoutés manuellement à l'output)

```
>>> [len(str(x)) for x in Kcent]
[1, 1, 1, 2, 2, 3, 4, 4, 5, 5, 6, 7, 7, 8, 8, 9, 10,\
 10, 11, 12, 12, 13, 13, 14, 15, 15, 16, 17, 17, 18,\
 18, 19, 20, 20, 21, 22, 22, 23, 23, 24, 25, 25, 26,\
 27, 27, 28, 28, 29, 30, 30, 31, 32, 32, 33, 33, 34,\
 35, 35, 36, 36, 37, 38, 38, 39, 40, 40, 41, 41, 42,\
 43, 43, 44, 45, 45, 46, 46, 47, 48, 48, 49, 50, 50,\
 51, 51, 52, 53, 53, 54, 55, 55, 56, 56, 57, 58, 58,\
 59, 60, 60, 61, 61]
```

On constate que la longueur de $v_n$ (nombre de chiffres de l'écriture décimale) est grosso modo $0.6n$.

3.
```
>>> pgcd(Kcent[99], Kcent[77]) == Kcent[11]
True
>>> Kcent[11]
1306469
```

4.
```
def valide(reps):
    """Vérification d'un théorème mathématique par tests aléatoires

    >>> valide(10)
    OK
    >>> pgcd(Kcent[99], Kcent[77]) == 1306469 == Kcent[11]
    True
    """
    for k in range(reps):
        a = randrange(0, 100)
        b = randrange(0, 100)
        D = pgcd(Kcent[a], Kcent[b])
        d = Kcent[pgcd(a, b)]
        if D != d:
            print("ERREUR!")
            print(a, b)
            break
    else:
        print("OK")
```

### G.5.3 Exercice 3

1.
```
def extractradical(N):
    """renvoie I, J de sorte que N == (I**2) * J et J sans carrés

    >>> extractradical(0)
    (1, 0)
    >>> extractradical(1024)
    (32, 1)
    >>> extractradical(512)
```

```
    (16, 2)
    >>> extractradical(360)
    (6, 10)
    >>> extractradical(101871000)
    (210, 2310)
    >>> extractradical(18577877449223257)
    (11111, 150483817)
    """
    if N <= 3:
        return 1, N
    I, J = 1, N  # flake8 se plaint: E741 ambiguous variable name 'I'
    # avec flake8 récent, qui incorpore pycodestyle
    # https://www.python.org/dev/peps/pep-0008/#names-to-avoid
    # sous prétexte que le I dans certaines polices se confond
    # avec un 1 ou l, mais tout programmeur utilise une police
    # à chasse fixe décente qui justement évite ces problèmes !
    # ce E741 est tout simplement pénible.
    while J & 3 == 0:
        I <<= 1
        J >>= 2
    P, Q, K = 3, 9, J
    if K & 1 == 0:
        K >>= 1
    while Q <= K:
        q, r = divmod(K, Q)
        while r == 0:
            I *= P
            J //= Q
            K = q
            q, r = divmod(K, Q)
        q, r = divmod(K, P)
        if r == 0:
            K = q
        P += 2
        Q = P ** 2
    return I, J
```

Une erreur intéressante: certains testent si J est divisible par 4 ainsi:

```
if J/4 == J//4:
```

Mais le terme de gauche est un `float`[127] qui a (sur les implémentations usuelles de Python) entre 16 et 17 chiffres décimaux de précision. Dans le cas de 18577877449223257 il y a justement (je ne l'avais pas fait exprès) 17 chiffres, et…:

```
>>> 18577877449223257/4 == 18577877449223257//4
True
```

Donc cette entrée met en évidence l'erreur de codage. L'autre erreur relève du style: chaque terme représente une division, on ne doit jamais faire deux divisions lorsqu'une seule suffit. Et d'ailleurs mon corrigé ne fait pas du tout de division, mais utilise l'opérateur binaire &.

2.
```
>>> extractradical(1000)
(10, 10)
```

127. https://docs.python.org/3/library/functions.html#float

```
>>> extractradical(18577877449223257)
(11111, 150483817)
```

3. Dans le cas `N == 0`.
4. Au final $I$ est le plus grand entier tel que $I^2$ divise $N$.
5. (esquisse) La relation $N = I^2 J$ est un invariant de boucle. Les nombres premiers divisant le dernier $J$ (s'il est $> 1$) ont multiplicité 1.

---

*Date de dernière modification* : 23–12–2017 à 15:56:18.

# H

## Téléchargement

http://jf.burnol.free.fr/MAO-M1.pdf (version du 12 décembre 2017)

---

*Date de dernière modification* : 24-01-2018 à 15:22:21.