MAO-MI

Version « Automne 2016 »

Jean-François Burnol

8 juin 2017

Table des matières

1 Présentation

1.1 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 3)) quelques sujets classiques en algorithmique et arithmétique choisis pris parmi ceux-ci :

— coût en temps et espace d'un programme,
— récursivité,
— algorithmes de tris,
— Euclide et Bezout pour les entiers et les polynômes,
— exponentiation rapide,
— multiplication rapide (pour les entiers et les polynômes),
— nombres premiers, polynômes irréductibles,
— arithmétique modulaire, cryptographie RSA,
— corps finis,
— logarithme discret (méthode de Shanks),
— tests de primalité (Miller-Rabin, Agrawal-Kayal-Saxena).

Il sera impossible d'être exhaustif, évidemment, vu le temps limité. Idéalement nous irons jusqu'à une implémentation de l'algorithme de primalité de Agrawal, Kayal, et Saxena.

Documentation sur l'algorithme de primalité AKS (page 7)

2. https://docs.python.org/3/reference/index.html

1.2 Python3 avec Spyder

Nous utiliserons le langage de programmation Python3 [3], avec comme environnement de travail le logiciel Spyder qui donne accès simultanément à une console interactive IPython, à un éditeur de scripts Python (fichiers avec extension `.py`), et à la documentation provenant des sources des modules standard.

Note : Sur les machines des salles de TPs, je conseille de choisir l'environnement de bureau *Xfce*. Il est léger et rapide. On y lance le logiciel Spyder via le menu *Développement* dans le menu *Applications*.

1.2.1 Installation sur son ordinateur personnel

Les feuilles de TPs seront accessibles via internet. Pour ceux qui souhaiteront prolonger le travail chez eux, il leur est possible d'installer la distribution *Anaconda* qui est celle déployée en salles machines.

Aller sur http://continuum.io/downloads, cliquer sur `I want python 3.4` et suivre les instructions suivant votre type ordinateur et de système d'exploitation. C'est une assez grosse installation car elle contient d'emblée de nombreuses librairies scientifiques telles que `numpy`, `scipy`, `matplotlib`, `SciKit-Learn`, etc... et aussi par ailleurs `IPython`, `Cython`, `Pygments`, `Sphinx`, ...

Une fois le logiciel installé sur votre ordinateur personnel vous pourrez mettre à jour les différentes librairies ou en installer de nouvelles via l'utilitaire en ligne de commande `conda` qui vient avec.

Après installation on peut lancer *Spyder* 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`.

1.2.2 Prise en main de Spyder en salles de TPs

Lancez le logiciel `Spyder` : suivant l'environnement de bureau choisi, on le trouvera dans *Programmation* ou dans *Développement*, ou via une icône ; en cas de problème, reloguer vous en choisissant *Xfce* comme environnement de bureau ; si les problèmes persistent vous devriez pouvoir lancer dans une Console l'exécutable `spyder` via la commande `/usr/local/anaconda3/bin/spyder &`.

La fenêtre principale s'affiche. Elle comporte normalement trois panneaux. Vérifiez que celui 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.

Astuce : Pour des essais rapides, il n'est pas nécessaire de lancer `Spyder`, la commande `ipython` directement dans une Console entamera une session interactive. Pour la quitter : `CTRL-D`. De même vous pouvez taper `python` directement dans une Console.

3. https://docs.python.org/3/reference/index.html

Pour les machines des salles de TPs, il est possible que `PATH` ne soit pas ajusté, je signale donc le chemin d'accès complet : `/usr/local/anaconda3/bin/ipython`.

Le panneau de gauche de `Spyder` permet de rédiger des scripts entiers, de les sauvegarder sous forme de fichier `.py` et de lancer leur exécution [1] dans la console interactive IPython du panneau du bas à droite (vous pouvez fermer les autres onglets de ce panneau, s'il y en a).

Créez un répertoire `MAO` dans votre dossier de départ et configurez `Spyder` (via `Outils::Préférences::Répertoire de travail global`) pour qu'il utilise ce répertoire comme l'endroit où seront stockés les scripts Python (fichiers avec extension `.py`). Puis, ouvrez un nouveau fichier via `Fichier::Nouveau fichier...`, de sorte qu'il soit créé d'office dans votre répertoire `MAO`. Vous pouvez fermer l'onglet avec le fichier `temp.py` qui était peut-être affiché au premier démarrage.

Plus tard, vous prendrez le temps d'explorer aussi les autres Préférences de `Spyder`, par exemple celles en relation avec l'affichage automatique de la documentation extraite de la source des modules Python.

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

```
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.

Plus tard vous expérimenterez avec les flèches haut et bas pendant ou après la rédaction d'un bloc multi-lignes, ainsi qu'avec la touche tabulation et le `?` (et retour chariot) suivant un mot clé.

1.2.3 Documentation sur Python

Les bases du langage Python3 sont simples. On trouve sur internet encore majoritairement de la documentation pour la version antérieure du langage, Python2. C'est le cas des deux premières références ci-dessous, que je conseille néanmoins car chacune donne beaucoup d'informations en une seule page.

1. exécution totale ou partielle : on peut séparer des « cellules » par des `#%%`.

Note : Pour les différences entre Python3 et Python2 que même les débutants doivent connaître, voir plus bas *Python3 vs Python2* (page 4).

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*).

Les références suivantes sont pour 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.

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

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

1.2.4 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 : 31-08-2015 à 15:27:11 CEST.

2

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 0.5×10^{-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 [6]).

Date de dernière modification : `17-12-2014 à 09:23:15 CET.`

6. http://fr.wikipedia.org/wiki/Test_de_primalit%C3%A9_de_Miller-Rabin

3

Feuille de travaux pratiques 1

3.1 Exponentiation rapide

3.1.1 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
```

Je signale au passage que la notation ^ est utilisée pour tout-à-fait autre chose que l'exponentiation (pour ceux qui veulent savoir, le caret fait en Python un « bitwise xor[7] », c'est-à-dire l'ad-

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

dition modulo 2 bit par bit sur les représentations binaires ; à propos pourquoi cela est-il bien défini même pour de longs entiers dont la représentation peut prendre un nombre arbitraire de mots dans la mémoire ?).

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

La fonction a été composée dans l'éditeur de script de Spyder. Une bonne habitude est de précéder systématiquement chaque nouvelle fonction dans son script par un `#%%` car alors on peut provoquer son interprétation via `Exécuter la cellule`.

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 [11]: %timeit -n 10000 puissance(2,10)
10000 loops, best of 3: 1.82 µs per loop

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

In [13]: %timeit -n 10000 2**10
```

```
10000 loops, best of 3: 22.7 ns per loop

In [14]: %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 $n \neq 0, 1$.

3.1.2 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):
    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 [20]: %timeit -n 10000 puissanceiter(2,100)
10000 loops, best of 3: 14.1 µs per loop

In [21]: %timeit -n 10000 puissanceiter(3,100)
10000 loops, best of 3: 13.2 µs per loop

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

In [23]: %timeit -n 10000 puissance(2,100)
```

```
10000 loops, best of 3: 20.2 µs per loop

In [24]: %timeit -n 10000 puissance(3,100)
10000 loops, best of 3: 20.5 µs per loop

In [25]: %timeit -n 10000 puissance(17,100)
10000 loops, best of 3: 21.6 µs per loop
```

On a gagné un peu.

3.1.3 puissanceiteravecfor(n,m)

Exercice

La solution proposée pour *puissanceiter(n,m)* (page 11) 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.

```python
#%%
def puissanceiteravecfor(n,m):
    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 19).

3.1.4 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 »).

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. Il serait plus rapide d'utiliser le `&` qui est un `ET` binaire, donc ici `if m&1:` (car la représentation binaire de `1` a un unique bit non nul, donc le `m&1` est `1` si `m` est impair, `0` sinon). Comme on doit à la fois tester la parité et diviser par `2`, on pourrait aussi envisager l'emploi de `divmod(m,2)`. Bon, à vous de choisir selon votre goût. Personnellement j'ai opté pour le `m>>1` pour diviser par `2` (cf. Shifting operations [8] pour les opérateurs >> et <<).

```
#%%
def puissancerap(n,m):
    """Exponentiation entière rapide récursive."""
    if m==0:
        return 1
    if m&1:
        return n*puissancerap(n*n,m>>1)
    else:
        return puissancerap(n*n,m>>1)
```

```
In [27]: puissancerap(2,10)
Out[27]: 1024

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 [31]: %timeit -n 10000 puissancerap(3,100)
10000 loops, best of 3: 2.35 µs per loop

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.

3.1.5 puissancerapiter(n,m)

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

Exercice

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

```
#%%
def puissancerapiter(n,m):
    if m==0:
        return 1
    p=1
    while m>1:
        if m&1:   # teste si m est impair
            p=n*p # ou encore p*=n
        m=m>>1    # ou encore m>>=1, ou m=m//2 ou m//=2
        n=n*n     # ou  aussi n*=n
    return p*n
```

```
In [34]: puissancerapiter(2,10)
Out[34]: 1024

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 [39]: %timeit -n 10000 puissancerapiter(17,100)
10000 loops, best of 3: 1.82 µs per loop

In [40]: %timeit -n 10000 puissancerap(17,100)
10000 loops, best of 3: 2.6 µs per loop

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.

Note : 1. Notre algorithme itératif implicitement calcule les chiffres de l'écriture en binaire de l'exposant, en partant du moins significatif (qui détermine la parité initiale) vers le plus significatif. Si on organise les choses en calculant d'abord ces bits, puis en allant dans l'autre sens à chaque fois on doit (ou non) multiplier par n qui ne varie pas contrairement à ce qui se passe dans notre `puissancerapiter(n,m)`. Dans certains contextes, par exemple l'exponentiation modulo quelque chose, cela peut-être plus efficace.

2. Il est clair que si l'on devait faire l'implémentation en langage *C*, on disposerait d'un accès quasi-direct aux bits de l'exposant, au moins par mots de 32 ou 64 bits, et en tout cas on n'aurait pas besoin à chaque étape de le diviser par 2, comme j'ai dû le faire ici (même si cette division a été efficacement faite par un « bitwise shift [9] »). Il est probable que cela compte parmi les causes principales de la lenteur de notre fonction en comparaison avec le `n**m` natif.

---

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

3.2 Fibonacci

3.2.1 `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):
    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} !

3.2.2 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).

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

def fiborecursif(n):
    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.

3.2.3 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):
    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.

3.2.4 `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):
    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
```

3.2.5 `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. Bien sûr, il serait à ce stade plus satisfaisant d'utiliser la *programmation objet* de Python pour définir une classe matrice et des méthodes de produit et de puissance, mais on va le faire plus « à la main » en manipulant uniquement des variables a, b, c représentant des matrices symétriques $\begin{pmatrix} a & b \\ b & c \end{pmatrix}$ qui de plus vérifieront toujours, dans l'algorithme ci-dessous, $a = b + c$ car les matrices seront toutes parmi les A^n ci-dessus.

Exercice

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

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

Exercice

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

```python
def fiborapide(n):
    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 [57]: %timeit -n 100 fiboiteratif(10)
100 loops, best of 3: 1.42 µs per loop

In [58]: %timeit -n 100 fiborapide(10)
100 loops, best of 3: 2.13 µs per loop

In [59]: %timeit -n 100 fiboiteratif(20)
100 loops, best of 3: 2.81 µs per loop

In [60]: %timeit -n 100 fiborapide(20)
100 loops, best of 3: 2.59 µs per loop

In [61]: %timeit -n 100 fiboiteratif(30)
100 loops, best of 3: 4.21 µs per loop

In [62]: %timeit -n 100 fiborapide(30)
100 loops, best of 3: 3.26 µs per loop

In [63]: %timeit -n 100 fiboiteratif(40)
100 loops, best of 3: 5.59 µs per loop

In [64]: %timeit -n 100 fiborapide(40)
100 loops, best of 3: 3.22 µs per loop

In [65]: %timeit -n 100 fiboiteratif(50)
100 loops, best of 3: 7.2 µs per loop

In [66]: %timeit -n 100 fiborapide(50)
100 loops, best of 3: 5.36 µs per loop

In [67]: %timeit -n 100 fiboiteratif(100)
100 loops, best of 3: 14.1 µs per loop

In [68]: %timeit -n 100 fiborapide(100)
100 loops, best of 3: 6.7 µs per loop

In [69]: %timeit -n 100 fiboiteratif(500)
```

```
100 loops, best of 3: 85.9 µs per loop

In [70]: %timeit -n 100 fiborapide(500)
100 loops, best of 3: 11.3 µs per loop

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 [73]: %timeit -n 100 fiboiteratif(2000)
100 loops, best of 3: 376 µs per loop

In [74]: %timeit -n 100 fiborapide(2000)
100 loops, best of 3: 11.9 µs per loop

In [75]: %timeit -n 100 fiboiteratif(3000)
100 loops, best of 3: 609 µs per loop

In [76]: %timeit -n 100 fiborapide(3000)
100 loops, best of 3: 16.3 µs per loop

In [77]: %timeit -n 100 fiboiteratif(4000)
100 loops, best of 3: 865 µs per loop

In [78]: %timeit -n 100 fiborapide(4000)
100 loops, best of 3: 23.6 µs per loop

In [79]: %timeit -n 100 fiboiteratif(5000)
100 loops, best of 3: 1.14 ms per loop

In [80]: %timeit -n 100 fiborapide(5000)
100 loops, best of 3: 26.8 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 $n = 100$, `rapide` est 2 fois plus rapide que `iteratif`, pour $n = 500$ il est 7.5 fois plus rapide, pour $n = 1000$, 19 fois plus rapide, pour $n = 5000$, 42 fois plus rapide, pour $n = 10000$, 43 fois plus rapide… ah ça a l'air de se tasser ??

```
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.

3.3 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).

3.3.1 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
```

3.3.2 `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.

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 :

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. http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.loglog
23. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.linregress.html#scipy.stats.linregress

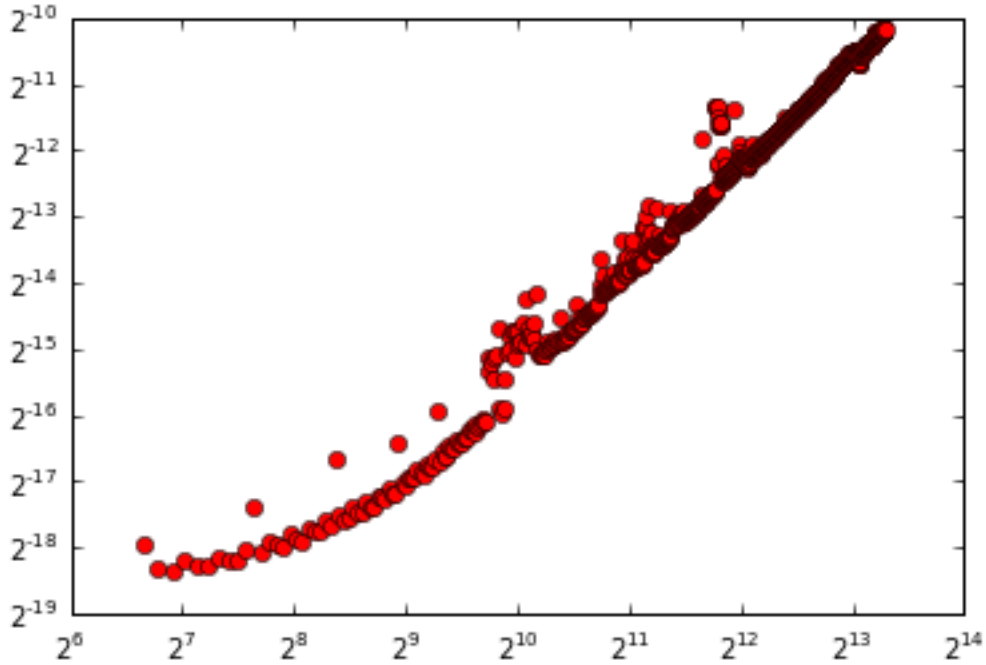
1. d'une part la formule $1+\lfloor \log_{10} N \rfloor$ cesse rapidement de fonctionner car nos nombres avec des centaines de chiffres sont trop grands pour la conversion en type float nécessaire au log,
2. la méthode via `len(str(N))` nécessite une conversion en décimal qui, comme je l'ai dit, est 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^α est absorbée dans le c, cela ne change rien au α 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))) # 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 :

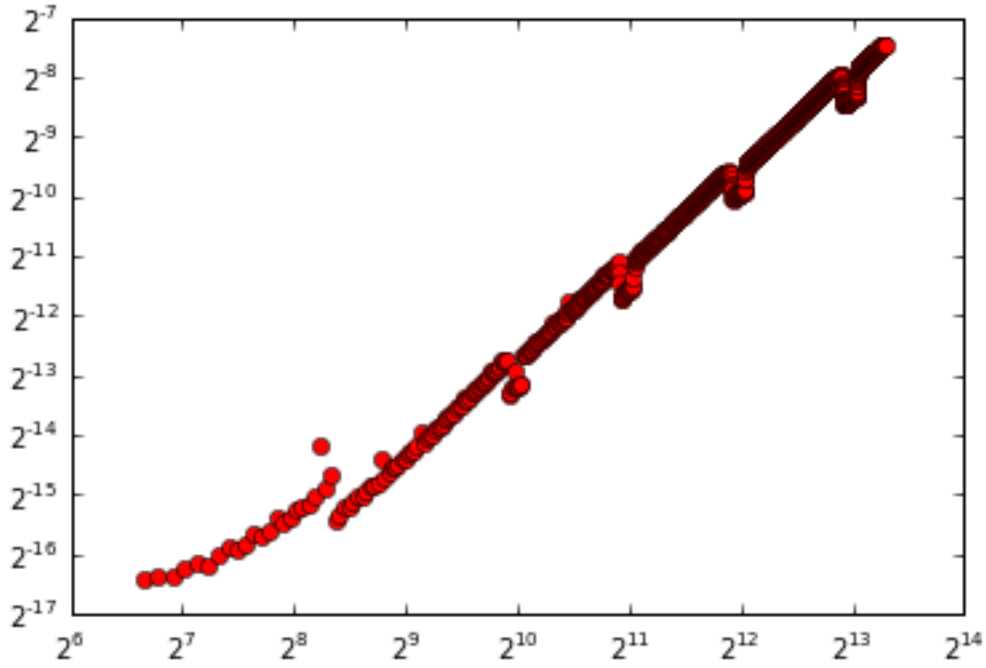
```
In [4]: tempspourmultiplication(10,1000,10)
1.49359699075 0.992311461885 0.00592362802229
```



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 :

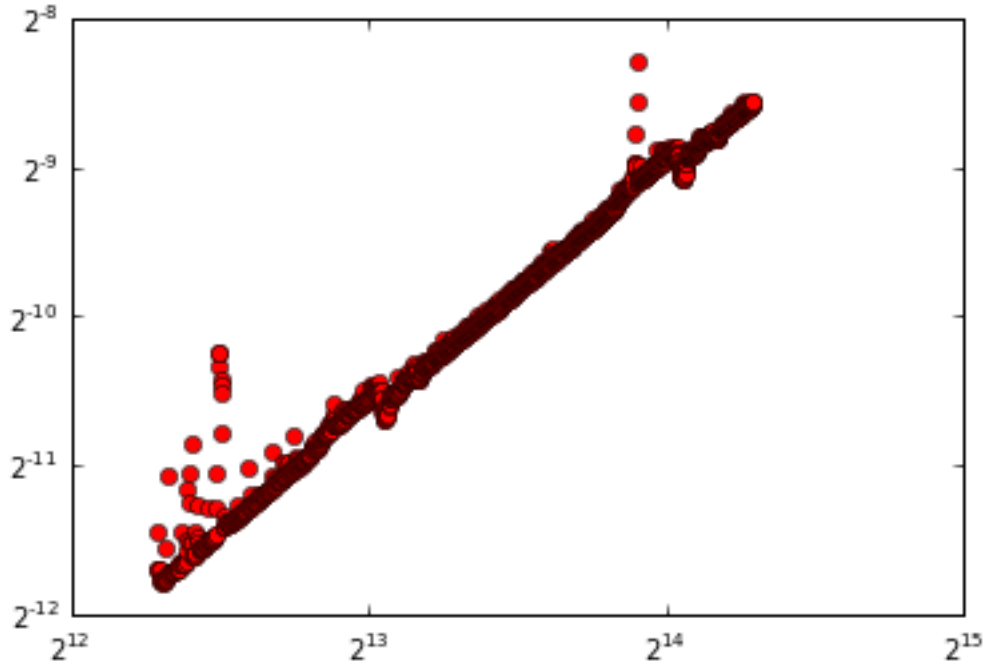
```
In [5]: tempspourmultiplication(10,1000,10)
1.5381069965 0.99478525524 0.00501445415698
```



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.

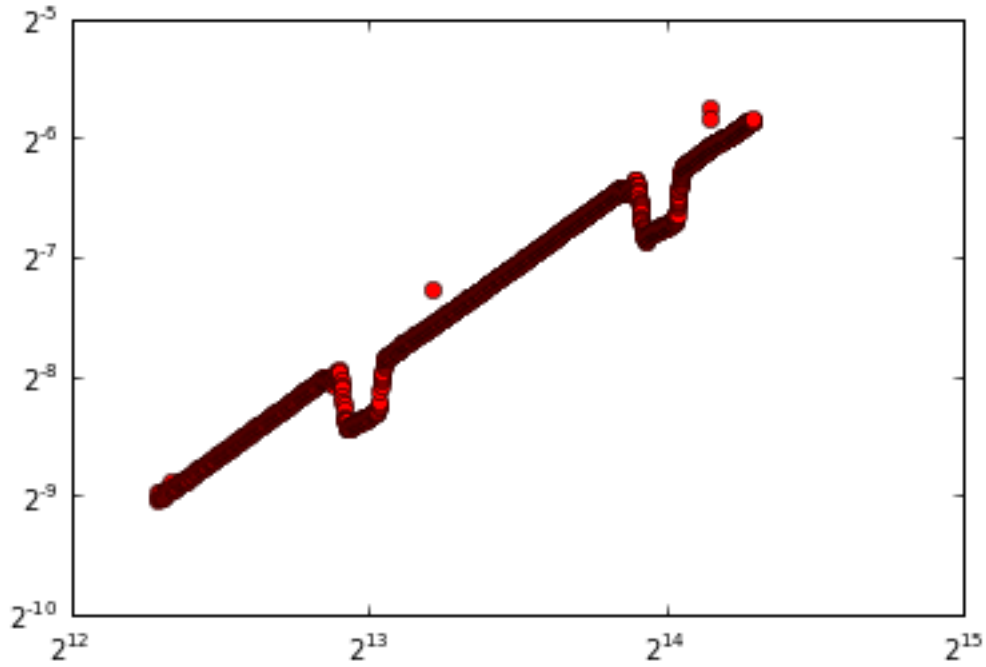
Regardons entre 5000 et 20000 bits binaires (donc grosso modo entre 1500 et 6000 chiffres décimaux). Il faut s'armer d'un peu de patience :

```
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.

3.3.3 tempspourmultiplicationII(M,N,reps)

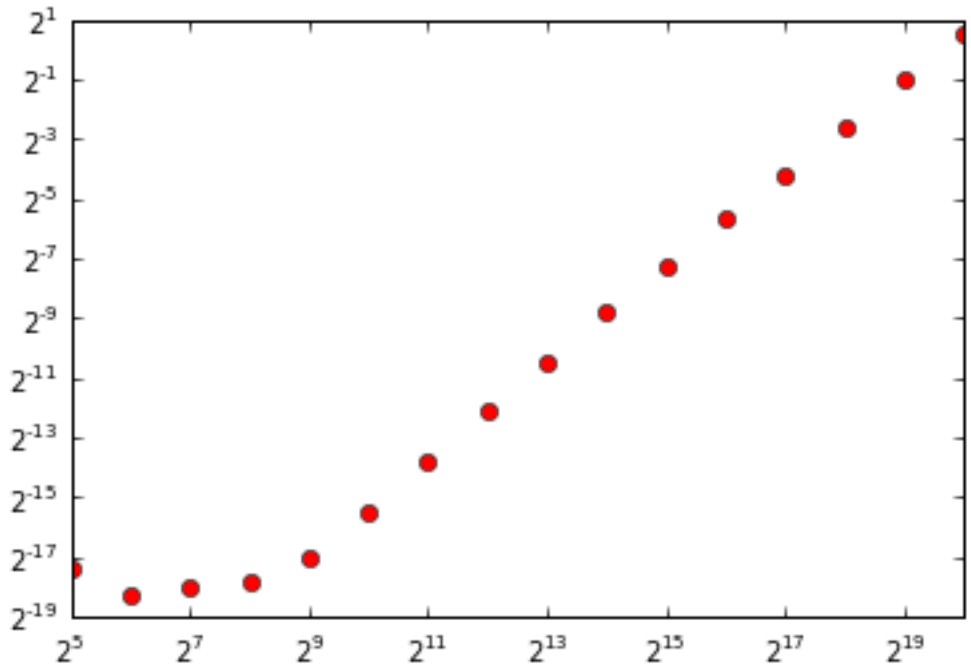
En réalité, Python peut manipuler des nombres beaucoup plus grands. Pour explorer cela j'ai donc fait une deuxième procédure qui carrément double le nombre de bits à chaque itération, afin de regarder ce qui se passe pour les nombres comportant des dizaines voire des centaines de milliers de chiffres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

```
#%%
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)
```

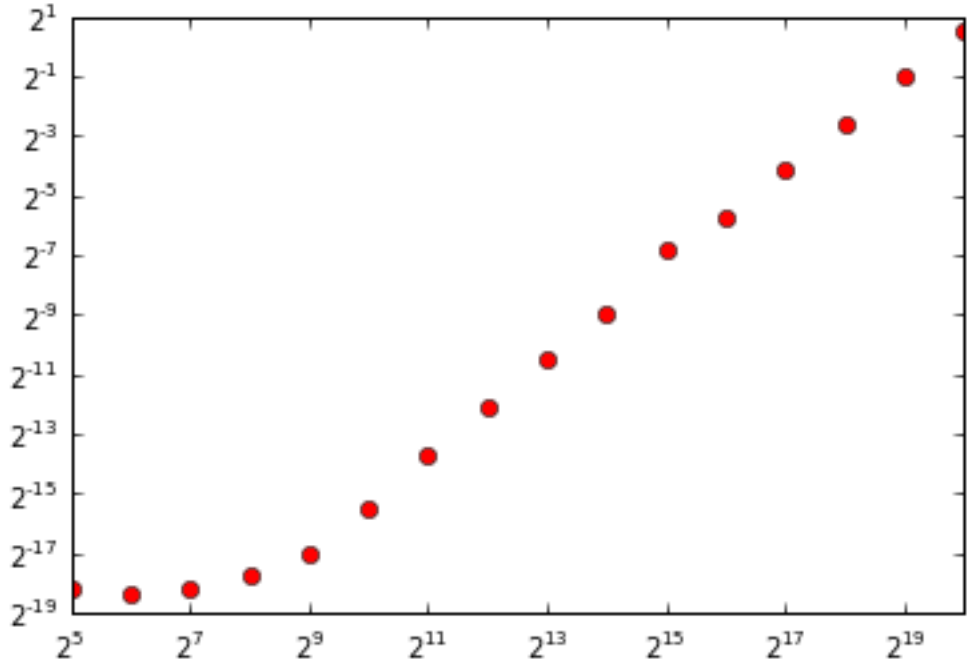
Voici ce que cela donne (il faut attendre un peu que la procédure aboutisse ; si un de vos camarades vous dit attendre depuis plusieurs minutes, essayez d'abord un 15 par exemple à la place de 20) :

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



Une deuxième fois :

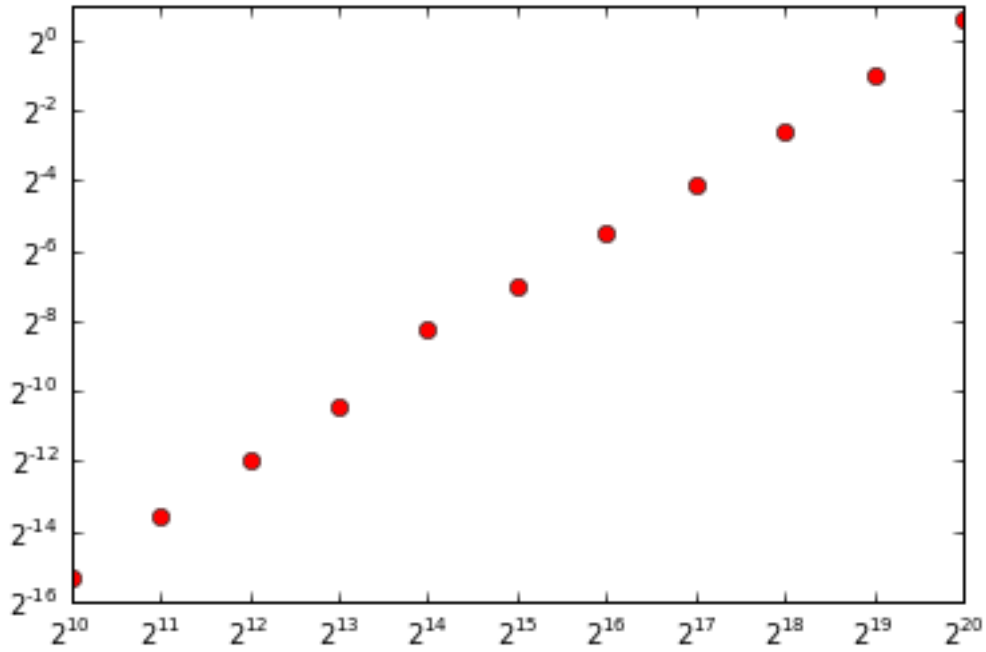
```
In [9]: tempspourmultiplicationII(5,20)
1.38055880444 0.985905166856 0.0626130661904
```



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.

3.3.4 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 20) 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 : `13-10-2016 14:58:45 CEST`.

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/

4

Feuille de travaux pratiques 2

4.1 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 3) 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/3/tutorial/introduction.html

4.2 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 = ['Noura', 'Imane', 'Arij', 'Samuel', 'Obayda', 'Atmina',
   ...: 'Ivan', 'Léo', 'Florentin', 'Geoffrey', 'Stéphane',
   ...: 'Tarlan', 'Léa', 'Théodore', 'Ramy', 'Jean-Baptiste',
   ...: 'Thibault', 'Matthieu', 'Amine', 'Sophie', 'Loïc',
   ...: 'Mohamed', 'Salim', 'Guillaume', 'Corentin', 'Inès',
   ...: 'Aloïs', 'Thomas', 'Abdillahi', 'Aron', 'Necib']

In [2]: type(L), len(L)
Out[2]: (list, 31)

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

In [4]: M[1], M[5]
Out[4]: ('Imane', 'Atmina')

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

In [6]: M
Out[6]:
['Abdillahi',
 'Aloïs',
 'Amine',
 'Arij',
 'Aron',
 'Atmina',
 'Corentin',
 'Florentin',
 'Geoffrey',
 'Guillaume',
 'Imane',
 'Inès',
 'Ivan',
 'Jean-Baptiste',
 'Loïc',
 'Léa',
 'Léo',
 'Matthieu',
 'Mohamed',
 'Necib',
 'Noura',
 'Obayda',
 'Ramy',
 'Salim',
 'Samuel',
 'Sophie',
 'Stéphane',
 'Tarlan',
```

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

```
 'Thibault',
 'Thomas',
 'Théodore']

In [7]: L.index('Théodore'), M.index('Théodore')
Out[7]: (13, 30)

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,'Inès'), indicedansliste(M,'Inès')
Out[11]: (25, 11)
```

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

4.2.1 `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):
    for i in range(len(liste)-1):
       if liste[i+1]<liste[i]:
           return False
    return True
```

4.2.2 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."""
    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
```

4.2.3 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).

```
#%%
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."""
    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 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

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)`.
35. https://docs.python.org/3/library/functions.html#sorted

à ce que ferait `N.sort()`.

Exercice

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

4.2.4 `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].

```python
#%%
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."""
    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
```

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

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 36), 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

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`.

4.2.5 `trirapide(L)`

Exercice

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

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.

```
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."""
    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
```

4.2.6 `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`.

```
#%%
def trirapidesurplace(debut,fin,liste):
    """Trie par algorithme ``QuickSort``.

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

    Le pivot est pris au milieu."""
    # 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

4.3 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

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

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 σ 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 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 σ 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`.

4.3.1 `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.

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

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 permute(L,P):
    M = []
    for i in range(len(P)):
        M.append(L[P[i]])
    return M
#%%
def permute(L,P):
    M = [None]*len(P) # initialisation
    for i in range(len(P)):
        M[i] = L[P[i]]
    return M
#%%
def permute(L,P):
    """\
    Action à droite de la permutation P sur la liste L.

    Pas de contrôle des arguments.
    """
    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.

4.3.2 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].

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

    Modifie le L d'origine.
    """
    # L[:] = permute(L,P)
    # ou même directement :
    L[:] = [L[P[i]] for i in range(len(P))]
```

4.3.3 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 ?

```
#%%
def inverseperm(P):
    """\
    Renvoie la permutation inverse.

    Ne modifie pas la permutation originelle.
    """
    Q = [None]*len(P)
    for i in range(len(P)):
        Q[P[i]] = i
    return Q
```

4.3.4 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…

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

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):
    for i in range(len(liste)-1):
       if liste[perm[i+1]]<liste[perm[i]]:
           return False
    return True
```

4.3.5 `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 `x=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.
    """
    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.

4.3.6 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)
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 :

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

```
>>> 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.

4.3.7 `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.
    """
    M = [(x,i) for i, x in enumerate(L)]
    return [i for x, i in triparinsertion(M)]
```

```
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 48), c'est normal.

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

4.3.8 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 48) ? 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.
    """
    # 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] # est l'emplacement de cet élément
                          # 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):
    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.

4.3.9 `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.
    """
    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 51).

```
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]
```

4.3.10 `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.

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

```
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
```

4.3.11 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:

```
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,)]
```

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

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 !

4.3.12 `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.

Date de dernière modification : `29-09-2016 à 20:48:51 CEST`.

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

5

Feuille de travaux pratiques 3

Note : À propos de « routine »

Dans le texte ci-dessous, j'utilise par moments le mot « routine ». Il semble que ce mot (ne pas confondre avec « rustine ») soit peu connu par les nouvelles générations. Il est utilisé ici comme synonyme de « procédure » ou « fonction ». Voici quelques citations fort anciennes qui témoignent de cet usage :

> INFORMAT. *Tout ou partie d'un programme ayant un emploi général ou répété* (MORVAN, Informat. 1981). *Le calculateur effectue la division à l'aide de soustractions répétées selon la routine* (BERKELEY, Cerveaux géants, 1957, p. 159). (source: http://atilf.atilf.fr/tlf.htm).

48. https://docs.python.org/3/library/stdtypes.html#int.bit_length

5.1 Racines énièmes de grands entiers

On a vu que Python autorisait 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.

5.1.1 Aparté sur les nombres en virgule flottante

Mais :

1. ça ne peut pas marcher car 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 chiffres décimaux de précision (plus précisément : 53 chiffres binaires) donc `t*t==x` répondra très probablement `False`, et en modifiant son dernier chiffre vers le haut ou le bas on ne sera pas plus avancé.
2. et d'ailleurs pour ce `math.sqrt(x)` il y a donc déjà une conversion implicite de `x` en un `float` [49] ce qui limite sa taille.

Par exemple :

```
>>> 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
```

Sur mon ordinateur le plus grand nombre représentable en virgule flottante par Python est (mais lire plus loin l'explication plus précise) :

$$1.7976931348623157 \cdot 10^{308}$$

En effet :

```
>>> import sys
>>> sys.float_info.max
1.7976931348623157e+308
```

On constate que $2^{1024} \approx 1.7976931348623159077293 \cdot 10^{308}$ est juste un peu au-dessus de ce maximum.

```
>>> float(2**1023)
8.98846567431158e+307
>>> float(2**1024)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
```

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

Le plus grand nombre réel exactement représentable comme un `float`[50]. n'est pas $1.7976931348623157 \cdot 10^{308}$ mais $2^{1024} - 2^{971} = 2^{1024}(1 - 2^{-53})$:

```
>>> import sys
>>> x = sys.float_info.max
>>> x
1.7976931348623157e+308
>>> x.hex()
'0x1.fffffffffffffp+1023'
>>> import textwrap
>>> print(textwrap.fill(repr(x.as_integer_ratio()),60))
(17976931348623157081452742373170435679807056752584499659891
74768031572607800285387605895586327668781715404589535143824 6
42343213268894641827684675467035375169860499105765512820762 4
54900903893289440758685084551339423045832369032229481658085 5
93321233482747978262041447231687381771809192998812504040261 8
4124858368, 1)
>>> x.as_integer_ratio()[0]-(2**1024-2**971)
0
```

Attention aux petites surprises avec les nombres à virgule flottante :

```
>>> x=sys.float_info.max
>>> x
1.7976931348623157e+308
>>> y=x+1
>>> y
1.7976931348623157e+308
>>> y==x
True
>>> 0.1+0.1+0.1 == 0.3
False
>>> 1/3+1/3+1/3+1/3+1/3 == 5/3
False
```

Pour en apprendre plus sur les types numériques en Python :

— Numeric Types — int, float, complex [51].

— Floating Point Arithmetic: Issues and Limitations [52].

Bref, il nous faut donc une autre approche si l'on veut pouvoir gérer les nombres de plusieurs centaines de chiffres dans notre problème d'extraire une racine carrée : il nous faut un algorithme utilisant des opérations uniquement sur des nombres entiers (par chance Python n'a pas de limite de taille sur les nombres entiers).

5.1.2 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 :

50. https://docs.python.org/3/library/functions.html#float
51. https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
52. https://docs.python.org/3/tutorial/floatingpoint.html

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

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 (pour une fois il s'agit donc qu'une question de pure mathématiques) $t_{n+1} = \lfloor \frac{1}{2}(t_n + q_n) \rfloor$. 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 $t_n = \lfloor \sqrt{x} \rfloor$.

5.1.3 `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 [53] de la variable `n` qui calculera un couple `t, q` initial approprié. Pour le moment, cette fonction `tqdefaut(n)` sera simplement définie comme renvoyant `n, 1`.

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.

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

```
#%%
def tqdefaut(n):
    """\
```

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

```
    Choix du point de départ pour algorithme de racine carrée.

    Doit être t, q avec q = n//t  et t*t > n. Peut supposer n>1.
    Destiné à être remplacé par un choix meilleur.
    """
    return n, 1
#%%
def racinecarreeint(n, tqinitial=tqdefaut):
    """\
    Calcule la partie entière de la racine carrée.

    Fonctionne avec des entiers arbitrairement grands.
    L'entier n est supposé positif.
    """
    if n<2:
        return n
    t, q = tqinitial(n)
    while q<t:
        t = (t+q)//2
        q = n//t
    return t
```

```
>>> racinecarreeint(12345678901234567890123456789012345678901234567890123456789012345)
11111111061111111099361111058
>>> 11111111061111111099361111058**2
12345678901234567890123456784876549901612666802098793 64
>>> 11111111061111111099361111059**2
12345678901234567890123456807098772023834888789321014 81
>>> racinecarreeint(12345678901234567890123456784876549901612666802098793 63)
11111111061111111099361111057
>>> import random
>>> for i in range(100):
...     a = 10**100
...     b = 2*a
...     n = random.randrange(a,b)
...     m = racinecarreeint(n)
...     x = m*m
...     y = x + 2*m + 1
...     if x > n:
...         print('ERREUR', n)
...         break
...     if y <= n:
...         print('ERREUR !', n)
...         break
... print('OK')
...
OK
```

Exercice

Majorer en fonction de la taille N de n en binaire (i.e. N est le nombre de chiffres de n en écriture binaire) le nombre d'étapes nécessaires à la procédure *racinecarreeint(n)* (page 60).

Indication : dans l'algorithme décrit précédemment, pour tous les j jusqu'à l'avant dernier $t_{j+1}^2 - x < (t_j^2 - x)/4$. C'est même largement mieux sauf pour les tout premiers, mais ça nous suffira.

En déduire que l'algorithme termine en `O(N^(a+1))`, si la division euclidienne fonctionne en `O(N^a)`. On a `a<2` en Python, comme on l'a étudié dans *Temps d'exécution de la multiplication en Python* (page 23).

On devrait choisir un meilleur point de départ pour la première valeur de `t`. *Et améliorer alors notre estimation du coût en temps de l'algorithme.*

5.1.4 Choix du point de départ par `int.bit_length()`[54]

Pour le choix de la première valeur de `t` : si `x` s'écrit avec `A` chiffres en décimal il est `<10**A`, donc `t=10**B` avec `B = (A+1)//2` serait un choix raisonnable. Mais pour obtenir ce `A` je ne vois que `len(str(x))`[55] qui est coûteux car il impose la conversion de `x` de sa représentation interne vers une chaîne de chiffres décimaux, ou encore 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 72)).

Il est néanmoins *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 <= 4x`. Et pourquoi diviser seulement par 2 ? Autant diviser par un plus gros truc genre `2**32` (en faisant attention à la fin).

Important : En fait le mieux ça serait de connaître l'exposant k avec $2^{k-1} \leqslant x < 2^k$, c'est-à-dire le nombre de chiffres de x en binaire. Heureusement il existe une fonction « *built-in* » en Python qui fournit cette information, plus précisément il s'agit d'une méthode de la classe `int`[57] :

```
>>> x=2**1000
>>> x.bit_length()
1001
>>> 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()`[58] a un temps d'exécution[59] qui dépend peu de l'entier en question, et en tout cas pas vraiment de la taille qu'il occupe en mémoire :

Note : La fonction `%timeit` est une spécificité de IPython, cela ne fonctionne pas dans une console Python pure.

---

54. https://docs.python.org/3/library/stdtypes.html#int.bit_length
55. Il y a `len(repr(x))` qui est légèrement plus rapide que `len(str(x))`. Voir `repr()`.
57. https://docs.python.org/3/library/functions.html#int
58. https://docs.python.org/3/library/stdtypes.html#int.bit_length
59. 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.

```
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 `tqparbitlength(n)` qui choisira pour `t` la plus petite puissance de 2 vérifiant `t*t>n`.

Note mathématique: 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 [60] 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()` [61].

```
In [1]: math.ceil(3.5)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-6ba7537884f0> in <module>()
----> 1 math.ceil(3.5)

NameError: name 'math' is not defined

In [2]: from math import ceil
```

60. https://fr.wikipedia.org/wiki/Partie_enti%C3%A8re_et_partie_fractionnaire
61. https://docs.python.org/3/library/math.html#math.ceil

```
In [3]: ceil(3.5)
Out[3]: 4
```

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 [62] >>.

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)
```

Je suppose que Python utilise en interne une représentation en complément à deux [63] 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 [64] avec un nombre de bits suffisant, mais ce qui est sûr c'est que les opérations binaires sont implantées de manière à donner le même résultat que si les nombres négatifs étaient écrits en complément à deux [65] avec une infinité de 1 sur la gauche.

```
#%%
def tqparbitlength(n):
    """\
    Choix du point de départ pour l'algorithme de racine carrée.

    Trouve la plus petite puissance de 2 avec t*t > n.
    Accepte même n=0 ou n=1.
    """
    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, n//t
```

Voici ce que donne son utilisation :

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

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

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

62. https://docs.python.org/3/library/stdtypes.html#bitwise-operations-on-integer-types
63. https://fr.wikipedia.org/wiki/Compl%C3%A9ment_%C3%A0_deux
64. https://fr.wikipedia.org/wiki/Compl%C3%A9ment_%C3%A0_deux
65. https://fr.wikipedia.org/wiki/Compl%C3%A9ment_%C3%A0_deux

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

In [51]: x=12345678901234567890123456789012345678901234567890123456789012345

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

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

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

Pour 2**2999 on a divisé par plus de 100 le temps de calcul !

Pour les entiers de moins de 300 chiffres en décimal on pourrait essayer de procéder via `math.sqrt()` [66] :

```
#%%
import math
#%%
def tqparsqrt(n):
    """\
    Point de départ pour algorithme de racine carrée.

    Utilise math.sqrt() si n suffisamment petit (< 10^308 environ)
    """
    try:
        # on multiplie par 2 pour assurer d'être > la véritable
        # racine carrée. Question : pourquoi n'ai-je pas à la place
        # simplement additionné 1 ?
        t = 2*round(math.sqrt(n))
    except OverflowError as err:
        return tqparbitlength(n)
    else:
        return t, n//t
```

Mais c'est moins efficace que le choix du point de départ via `tqparbitlength()` :

```
In [49]: x = 12345678901234567890123456789012345678901234567890123456789012345 # 45 chiffres

In [50]: y = y=10**45*x+x # 90 chiffres en décimal

In [51]: z= y*y+1 # 179 chiffres en décimal

In [52]: %timeit racinecarreeint(x, tqparbitlength)
100000 loops, best of 3: 2.92 µs per loop

In [53]: %timeit racinecarreeint(x, tqparsqrt)
100000 loops, best of 3: 3.24 µs per loop

In [54]: %timeit racinecarreeint(y, tqparbitlength)
100000 loops, best of 3: 3.07 µs per loop
```

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

```
In [55]: %timeit racinecarreeint(y, tqparsqrt)
100000 loops, best of 3: 4.3 µs per loop

In [56]: %timeit racinecarreeint(z, tqparbitlength)
100000 loops, best of 3: 5.39 µs per loop

In [57]: %timeit racinecarreeint(z, tqparsqrt)
100000 loops, best of 3: 7.18 µs per loop

In [58]: w=10**308-1

In [59]: %timeit racinecarreeint(w, tqparbitlength)
100000 loops, best of 3: 13.3 µs per loop

In [60]: %timeit racinecarreeint(w, tqparsqrt)
100000 loops, best of 3: 15 µs per loop
```

5.1.5 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, Fini : 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.
    """
    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 `t0 = n` 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.

5.1.6 t0parbitlength(n,k)

Exercice

Faire une fonction `t0parbitlength(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 t0parbitlength(n,k):
    """\
    Renvoie la plus petite puissance t de 2 telle que t^k>n

    Suppose n au moins 1.
    """
    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.

5.1.7 racineeniemeint(n,k)

Exercice

Faire une fonction `racineeniemeint(n,k)` comme *racineeniemeint_temp(n,k)* (page 66) mais avec un meilleur point de départ.

```
#%%
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.
```

```
    """
    if n<2:
        return n
    t = t0parbitlength(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
```

5.1.8 estunepuissance(n)

Exercice

Faire une fonction `estunepuissance(n)` qui étant donné un entier non négatif n arbitrairement grand détermine si on peut l'écrire sous la forme a^k avec $a > 1$ et $k > 1$. On montrera que l'algorithme suivant fonctionne :

```
Traiter séparément 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 répondre Non.

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

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

Itérer.
```

Faire une implémentation.

```
#%%
def estunepuissance(n):
    """\
    Détermine si n=a**k avec a, k >1.

    Assume n positif.
    """
    if n==0:
        print(n, 'est nul')
        return False
    if n==1:
        print(n, 'est 1')
        return False
    a=racinecarreeint(n, tqparbitlength)
    if a*a==n:
        print('n est le carré de', a)
        return True
    k=3
    while True:
        a=racineeniemeint(n,k)
        if a==1:
            print(n, "n'est pas une puissance")
            return False
        if a**k==n:
            print('n est la puissance {}ième de {}'.format(k,a))
            return True
        k += 2
```

Pourquoi la bidouille à la fin avec `k += 2` ?

```
In [42]: estunepuissance(6801627477633567653059023813514866104681l7\
    ...: 9785399615827535771334429109739038826587477261293880997047\
    ...: 9897221871700185302136628522737202181929117206596730265816\
    ...: 2926791072174084064517915341702614660205041728348541163091831231)
n est la puissance 17ième de 11193791827391
Out[42]: True
In [43]: estunepuissance(8924213474370287383311682064643843\
    ...: 2087886269453603168516372613672329485975176474766l3\
    ...: 3996747713843061763647197739131594553421505478l296\
    ...: 2070738550329557701536767872707247851191279951702l\
    ...: 8223806270178158481691206104822375007217164227598l\
    ...: 7107363991480827764350056032928691479966674929179l\
    ...: 6752692456906041675483863654519960926585967798514l\
    ...: 3441469163027420396290499467136284960031955021762l\
    ...: 0358209562548422446605198402126251011131130638444l\
    ...: 5624631125969827735759271716698431871516014082226l\
    ...: 3337187315230268752574193315332928207662740691823l\
    ...: 3351567291304717885026799023089220229612846862221l\
    ...: 7203062828251458035361637423513088403632620861342l\
    ...: 1840246218177529646872011766370825197710366033519l\
    ...: 7828571294738042161288715424119567590563117307114l\
    ...: 5423066058947479306165931939785423433995100625787l\
    ...: 9525344091738211508470033720726562513503924329074l\
    ...: 1069562587196178511927781894596834187805216349980l\
    ...: 8933750794086811425923945087343476734587288047273l\
    ...: 9611937529526433554476303142437825510673921447914l\
    ...: 5988426396573680527551031594274197973285500656238l\
```

```
    ...: 8870042256996519995610277163360185292235358061126 7\
    ...: 8291428055541633423286442912896112690686289655405 9\
    ...: 0855444211978065701467904927041982195004196752795 4\
    ...: 9141270665431432582080308351920030422488033882960 0\
    ...: 5166784637427266397478317248185817779771789531933 6\
    ...: 0719769499096009754706842276613387251162803716722 1\
    ...: 0764529624089114703099991167782893721749117953623 5\
    ...: 639)
n est la puissance 97ième de 198819731987319
Out[43]: True
```

Exercice

Lorsque vous aurez un peu de temps libre, essayez de donner une majoration du temps d'exécution de *estunepuissance(n)* (page 68). Au moins, montrer que l'algorithme s'exécute en un temps polynomial en la taille de n.

5.2 Binaire vs décimal

5.2.1 longueurbinaire(n)

Exercice

Voici un super programme :

```
#%%
def longueurbinaire(n):
    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 ? (attention, l'auteur – d'ailleurs de tout ici – est votre Professeur mais je ne veux pas vous influencer). Pouvez-vous faire plus rapide ? (de légers gains sont possibles en remplaçant les `K = K<<1` par `K <<= 1` etc…).

Vous trouvez que c'est nul par rapport à `int.bit_length()` [67] ? attendez un peu..

5.2.2 `longueurdecimale(n)`

Exercice

En s'inspirant de *longueurbinaire(n)* (page 70), 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):
    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 # K = K//2
            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
```

67. https://docs.python.org/3/library/stdtypes.html#int.bit_length

```
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 71) est plus rapide que `len(str(n))` (mais ce n'est pas le cas pour des entiers de seulement cent ou même mille chiffres).

5.2.3 longueurdecimale_lent(n)

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

```
#%%
def longueurdecimale_lent(n):
    # pour n nul, produit zéro et pas 1
    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 71) 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
```

5.2.4 `longueurdecimale2(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 ?

```
#%%
# import math
#%%
def longueurdecimale2(n):
    #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]: longueurdecimale2(x)
Out[35]: 47713

In [36]: %timeit len(str(x))
10 loops, best of 3: 42.6 ms per loop

In [37]: %timeit longueurdecimale2(x)
100 loops, best of 3: 2.01 ms per loop
```

5.3 Un peu d'arithmétique

On n'aura probablement pas le temps de faire grand chose dans cette section pendant la séance mais je veux avancer un peu néanmoins.

5.3.1 `pgcd(a,b)`

Le grand classique. Le pgcd est habituellement caractérisé comme le générateur positif de l'idéal $a\mathbb{Z}+b\mathbb{Z}$, ce qui a l'avantage de donner une définition aussi si $a=0$ et $b=0$, à savoir $\mathrm{pgcd}(0,0)=0$.

Exercice

Implémenter l'algorithme d'Euclide :

```
dernier reste non nul dans l'itération (a,b)<-(b,r) avec r
le reste dans la division euclidienne de a par b (a = qb + r)
```

L'algorithme d'Euclide présuppose habituellement `a, b > 0`. Votre implémentation Python devra autoriser des arguments négatifs ou nuls et le pgcd devra être positif (ou nul dans le cas exceptionnel `a = b = 0`).

```
(attention à ne pas faire de division si b = 0 ; et, si b divise a,
alors c'est lui (en valeur absolue) qui est considéré comme étant le
dernier reste)
```

```
#%%
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)
```

Le `abs(a)` à la fin est un peu énervant car la plupart du temps on utilise la routine avec `a` et `b` positifs.

Lorsque le reste `r` dans `a=bq+r` est plus grand que `b/2` on pourrait se dire que le remplacer par `b-r` est une bonne idée, car comme cela à chaque étape on est sûr de diviser par deux le deuxième terme du couple `(a,b)`. Mais calculer `b/2` (en Python par `b//2` ou plus rapide par `b>>1`) ou, peut-être mieux, calculer `b-r`, pour les comparer à `r`, rajoute un surcoût ; bref, je n'ai pas constaté d'avantage évident, au contraire. De toute façon si `r>b/2`, alors à l'étape d'après le quotient sera 1 et `b,r` deviendra `r, b-r`. Tester `r>b/2` rajoute à toutes les étapes un surcoût et le gain ne vient que lorsque vraiment `r>b/2` ; on aura donc évité une division euclidienne dans certains cas au prix d'un surcoût à chaque étape.

On notera qu'avec Python `a%b` est toujours positif lorsque `b>0`, indépendamment du signe de `a`, autrement dit il s'agit bien du reste de la division euclidienne.

```
>>> -5%2
1
>>> divmod(-5,2) # quotient et reste
(-3, 1)
```

Mais dans de nombreux autres langages qui ont une interface pour l'arithmétique, par exemple en Maple, il faut faire attention que le reste de la division entière est du signe de ce que l'on divise (lorsque le diviseur est positif) :

```
> irem(-5, 2);
                              -1
> iquo(-5, 2);
                              -2
```

Par conséquent la fonction `reste modulo b` n'est pas `b`-périodique dans ces langages (par exemple `irem(-7,10)` n'est pas égal à `irem(3,10)`), ce qui est très très très pénible. Heureusement Python a la bonne définition. Par contre attention au cas `b<0` qui donne toujours un reste négatif (ou nul ; il est en tout cas strictement inférieur en valeur absolue à `b`) :

```
>>> 5%-2
-1
>>> divmod(5,-2) # quotient et reste
(-3, -1)
```

C'est un peu gênant d'avoir un reste négatif, mais en tout cas il est périodique modulo `b`, c'est l'essentiel. La vraie division euclidienne de 5 par −2 est `5 = (-2)(-2)+1`, pas `5 = (-3)(-2)+(-1)`.

5.3.2 bezout(a,b)

Décrivons mathématiquement un peu mieux l'algorithme d'Euclide (en supposant $a, b > 0$). On pose $r_{-1} = a, r_0 = b$. La n ième étape fait la division euclidienne de r_{n-2} par r_{n-1} pour produire un quotient q_n et un reste r_n. Il y a en tout N étapes, et N est le premier n avec $r_n = 0$, donc le pgcd d est r_{N-1} et $r_N = 0$.

$$\begin{aligned} r_{n-2} &= q_n r_{n-1} + r_n \\ \begin{pmatrix} r_{n-2} \\ r_{n-1} \end{pmatrix} &= \begin{pmatrix} q_n & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} r_{n-1} \\ r_n \end{pmatrix} \\ \begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} q_1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} q_2 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_n & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} r_{n-1} \\ r_n \end{pmatrix} = M_n \begin{pmatrix} r_{n-1} \\ r_n \end{pmatrix} \end{aligned}$$

On définit M_0 comme étant la matrice identité. Le déterminant de M_n est $(-1)^n$. Notons C_n la première colonne et D_n la deuxième colonne de M_n.

$$(C_n \mid D_n) \begin{pmatrix} q_{n+1} & 1 \\ 1 & 0 \end{pmatrix} = (q_{n+1} C_n + D_n \mid C_n)$$

Donc, par récurrence, les matrices M_n sont de la forme $\begin{pmatrix} v_n & v_{n-1} \\ u_n & u_{n-1} \end{pmatrix}$ avec les relations :

$$\begin{aligned} v_{n+1} &= q_{n+1} v_n + v_{n-1}, & v_0 &= 1, v_{-1} = 0 \\ u_{n+1} &= q_{n+1} u_n + u_{n-1}, & u_0 &= 0, u_{-1} = 1 \\ M_n^{-1} &= (-1)^n \begin{pmatrix} u_{n-1} & -v_{n-1} \\ -u_n & v_n \end{pmatrix} \\ M_n^{-1} \begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} r_{n-1} \\ r_n \end{pmatrix} \\ r_{n-1} &= (-1)^n (u_{n-1} a - v_{n-1} b) \end{aligned}$$

Une identité de Bézout $d = Ua + Vb$ en résulte au final avec $U = (-1)^N u_{N-1}$ et $V = (-1)^{N-1} v_{N-1}$. On peut soit incorporer le signe dans les relations de récurrence, soit, ce pour quoi j'ai une préférence, ne travailler qu'avec des nombres positifs et maintenir une variable s pour $(-1)^n$.

Exercice

Faire `bezout(a,b)` qui produira un tuple `(d, U, V, a/d, b/d)` avec `d` le pgcd et `Ua+Vb = d`. On pourra supposer que `a` et `b` ne sont pas simultanément nuls.

```
#%%
def bezout(a,b):
    """\
    Calcule d, U, V, a/d, b/d avec d= Ua + Vb = pgcd(a,b).

    Fonctionne si (a,b) est (0,0) en renvoyant (0, 1, 0, 1, 0)
    mais c'est un cas pathologique. Dans tous les autres cas
    d > 0.
    """
    v, v_, u, u_ = 1, 0, 0, 1
    s = 1
    while b:
        q, r = divmod(a,b)
```

```
        v, v_ = q*v + v_, v
        u, u_ = q*u + u_, u
        a, b = b, r
        s = -s
    U = s*u_
    V = -s*v_
    if a < 0:
        return -a, -U, -V, -v, -u
    else:
        return a, U, V, v, u
```

```
>>> bezout(40719096145511827,61981561294553353)
(191352241, 122342142, -80373281, 212796547, 323913433)
>>> bezout(100,40)
(20, 1, -2, 5, 2)
>>> bezout(-100,-40)
(20, -1, 2, -5, -2)
>>> bezout(-360,-100)
(20, -2, 7, -18, -5)
```

5.3.3 puissancemod(a,n,b)

Python implémente nativement l'exponentiation modulaire $a^n \bmod b$, c'est-à-dire sans calculer d'abord l'entier a^n pour ensuite le réduire modulo b. Il procède sans doute par un algorithme comme celui d'exponentiation de notre première séance *Feuille de travaux pratiques 1* (page 9), mais avec bien sûr chaque multiplication faite modulo b :

```
Help on built-in function pow in module builtins:

pow(...)
    pow(x, y[, z]) -> number

    With two arguments, equivalent to x**y.  With three arguments,
    equivalent to (x**y) % z, but may be more efficient (e.g. for ints).
```

```
>>> 17**53
16360359141703731870462854470993114904566829992127299925766822433 7
>>> pow(17,53,1000)
337
```

Mais l'exposant doit être positif ou nul.

```
>>> pow(17,-53,1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pow() 2nd argument cannot be negative when 3rd argument specified
```

Il est un peu contre-intuitif que la classe de l'erreur levée soit alors `TypeError`[68] et non pas `ValueError`[69].

Exercice

68. https://docs.python.org/3/library/exceptions.html#TypeError
69. https://docs.python.org/3/library/exceptions.html#ValueError

Faire `puissancemod(a,n,b)` qui si `n` est positif ou nul donnera la main à `pow(a,n,b)` et pour `n<0` utilisera d'abord `bezout(a,b)` pour tester si `a` est inversible modulo `b` et produire son inverse. La procédure exigera `b>0`. Utiliser `raise`[70] pour lever éventuellement des exceptions `ValueError`[71] si `b<=0` ou `ZeroDivisionError`[72] si `a` n'est pas inversible modulo `b`.

Voir Raising Exceptions[73].

```
#%%
def puissancemod(a,n,b):
    """\
    Exponentiation modulaire a**n mod b.

    Lève une exception si b<=0 ou si b>0 mais a n'est pas
    inversible modulo b.
    """
    if b<=0:
        raise ValueError('Le module {} doit être '
                         'strictement positif'.format(b))
        return None
    if n>=0:
        return pow(a,n,b)
    d, U, _, _, _ = bezout(a,b)
    if d==1:
        return pow(U, abs(n), b)
    else:
        raise ZeroDivisionError("Le premier argument {} n'est pas inversible "
                                "modulo le troisième {}".format(a,b))
        return None
```

```
>>> puissancemod(17, 53, 1000)
337
>>> puissancemod(17, -53, 1000)
273
>>> 337*273
92001
```

5.3.4 estpremier(n)

Exercice

Faire `estpremier(n)` en divisant successivement à partir de `p=2`, on s'arrête si le dernier diviseur testé est plus grand que la racine carrée de `n`. On pourra supposer `n` au moins 1.

```
#%%
def estpremier(n):
    if n==1:
        return False
    if not n&1: # n est pair
```

70. https://docs.python.org/3/reference/simple_stmts.html#raise
71. https://docs.python.org/3/library/exceptions.html#ValueError
72. https://docs.python.org/3/library/exceptions.html#ZeroDivisionError
73. https://docs.python.org/3.4/tutorial/errors.html#raising-exceptions

```
        return n==2
    p = 3
    while True:
        q, r = divmod(n, p)
        if r==0:
            # alors ce p est forcément premier (car n n'est divisible
            # par aucun entier < p), donc n est premier ssi n==p
            return q==1
        # ici n n'est divisible par aucun entier <=p
        if q<=p:
        # alors 1 < n < (p+1)p ne peut pas être divisible deux fois
        # par un nombre premier > p, ni par deux nombres premiers > p
        # donc n est premier. En fait, même conclusion si q<=p+3 car
        # alors 1 < n < (p+4)p < (p+2)^2
            return True
        p +=2
```

Un peu longuet :

```
>>> estpremier(28937196319327)
True
```

Revenez la semaine prochaine on apprendra à faire mieux !

5.3.5 premierpremierapres(n)

Exercice

Faire `premierpremierapres(n)` qui trouve le premier nombre premier au moins égal à `n`.

```
#%%
def premierpremierapres(n):
    if n<=2:
        return 2
    if not n&1:
        n +=1
    while not estpremier(n):
        n +=2
    return n
```

```
>>> premierpremierapres(1000000000)
1000000007
>>> premierpremierapres(1000000000000)
1000000000039
>>> premierpremierapres(28937196319328)
28937196319349
```

5.3.6 diviseurspremiers(n)

Exercice

Faire `diviseurspremiers(n)` qui renvoie la liste par ordre croissant des diviseurs premiers de `n` (sans leurs multiplicités). On pourra supposer `n>=1`. Implémenter l'algorithme suivant :

```
Diviser n par 2 tant que c'est possible.

Tester si n==1, sinon poser p=3 puis itérer :

   Diviser n par p tant que c'est possible,

   Test d'arrêt (voir plus bas)

   Itérer en incrémentant p par 2 à chaque fois.

Test d'arrêt : on ne peut plus diviser n par p ; si le quotient q
est inférieur ou égal à p, alors soit n>1 et est premier soit
n=1. Dans les deux cas c'est fini.
```

Le type produit par la fonction sera `list`[74].

```python
#%%
def diviseurspremiers(n):
    """\
    Renvoie la liste, sans les multiplicités, des diviseurs premiers de n.

    On suppose n au moins 1.
    """
    L = []
    if not n&1:
        L.append(2)
        n >>= 1
        while not n&1:
            n >>= 1
    if n==1:
        return L
    p = 3
    while True:
        q, r = divmod(n, p)
        if r==0:
            L.append(p)
            while True:
               n = q
               q, r = divmod(n, p)
               if r:
                   break
        # ici n n'est plus divisible par aucun premier <=p
        if q <= p:
        # alors n < (p+1)p est soit 1 soit un nombre premier, car s'il
        # était divisible soit par deux nombres premiers distincts, soit
        # par le carré d'un premier, il serait >= (p+2)^2
            if n>1:
                L.append(n)
            return L
        p +=2
```

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

```
>>> diviseurspremiers(232307310937188460801)
[123457]
>>> diviseurspremiers(40719096145511827)
[541, 563, 577, 599, 613, 631]
>>> diviseurspremiers(61981561294553353)
[541, 577, 613, 647, 683, 733]
>>> diviseurspremiers(40719096145511827*61981561294553353)
[541, 563, 577, 599, 613, 631, 647, 683, 733]
>>> diviseurspremiers(pgcd(40719096145511827,61981561294553353))
[541, 577, 613]
>>> x = 4700488131248
>>> diviseurspremiers(x)
[2, 7, 13, 101, 31963933]
>>> y = 7079568471408
>>> diviseurspremiers(y)
[2, 3, 487, 302856283]
```

Ceci est assez longuet :

```
>>> diviseurspremiers(9680477938441039)
[31963933, 302856283]
```

Notre algorithme est à peu près le plus naïf possible. Il en existe de moins naïfs, mais c'est tout de suite assez rusé, voire complexe, voire même pour certains extrêmement compliqué. On verra ça un peu la prochaine fois.

Date de dernière modification : 18-11-2016 à 10:16:47 CET.

6

Feuille de travaux pratiques 4

6.1 Témoins de non-primalité

Soit n un nombre entier. Par exemple $n = 2821$. Je peux dire que 31 est un témoin de la non-primalité de n car je peux faire la division et trouver que « ça tombe juste » : $2821 = 31 \times 91$. Mais comment trouver ce 31 ? La semaine dernière on a fait une procédure *estpremier(n)* (page 78) qui divise jusqu'à soit trouver le plus petit diviseur de n (nécessairement un nombre premier) soit finir par dépasser la racine carrée de n, auquel cas on est sûr que n lui-même est premier. Dans le cas de 2821, on aurait immédiatement trouvé le diviseur 7.

Dans le pire des cas on teste environ $\sqrt{n}$ candidats, ou plutôt $.5\sqrt{n}$ car on ne va bien sûr pas tenter des diviseurs pairs une fois qu'on sait que n lui-même est impair (et de même une fois qu'on sait que n est premier avec 6, on n'a qu'à tester comme diviseurs les entiers de la forme $6k+1$, ou $6k+5$, donc on n'a à tester qu'environ $.33\sqrt{n}$ candidats et on peut continuer comme cela … sauf que ça devient rapidement compliqué pour un coefficient $\prod_{p \leqslant x}(1 - p^{-1})$ qui diminue très lentement).

Une méthode qu'on pourrait tenter est de prendre un nombre $1 \leqslant x < n$ au hasard et calculer le pgcd. Quand obtiendrons nous un pgcd > 1 donc une preuve que n est composé? Comme $n = 7 \times 13 \times 31$, on sait par le théorème des restes chinois qu'il y a $6 \times 12 \times 30 = 2160$ classes de congruences inversibles, ainsi parmi les 2820 entiers de 1 à $n - 1 = 2820$ il y en a tout de même 660 qui ne sont pas inversibles. Soit 1 chance sur $2820/660 = 4,27..$ de tomber sur un nombre ayant un facteur commun avec 2821. Disons plutôt qu'on a une probabilité de $2160/2820 = 0,7659574.. < 0,766$ de tomber sur une classe inversible et donc, en calculant le pgcd (ce qui est rapide) de ne pas être en mesure de conclure.

Cette probabilité de ne pas savoir conclure sera moins de $0,766^2$ si on fait deux tentatives indépendantes, puis moins de $0,766^3$ avec trois tentatives

Exercice

Utiliser la calculatrice Python pour examiner les premières puissances $0,766^k$, en particulier déterminer le premier k pour lequel c'est $< 0,5$ puis le premier k pour lequel c'est $< 0,01$.

Toute l'idée des tests probabilistes est que dès que l'on a ne serait-ce qu'un peu de chance de réussir, alors en étant obstiné on finit toujours par réussir. Bien sûr, plus faible est la probabilité d'échec pour un unique essai, mieux ce sera pour de multiples essais.

6.2 Bref aparté sur l'indicatrice d'Euler

Comme on sait le nombre de classes de congruence inversibles modulo n est $\phi(n) = n\prod_{p|n}(1-p^{-1})$ (le produit porte sur tous les diviseurs premiers p de n). Si on prend au hasard une classe de congruence non nulle la probabilité qu'elle soit inversible est $\phi(n)/(n-1)$: par exemple pour $n = 2821 = 7\times 13\times 31$ on retrouve notre 2160/2820. Mais en général le $1/(n-1)$ est un peu ennuyeux : on a une jolie formule pour $\phi(n)/n$, il est donc plus commode de dire que la probabilité de piocher une classe inversible (y-compris en s'autorisant bêtement à prendre la classe nulle) est $\phi(n)/n$.

Si vous avez besoin de vous rafraîchir la mémoire, vous pouvez consulter:

<https://fr.wikipedia.org/wiki/Indicatrice_d%27Euler>

Notez aussi que l'on pose $\phi(1) = 1$. Une propriété essentielle, liée au théorème chinois, est $\phi(ab) = \phi(a)\phi(b)$ lorsque a et b sont premiers entre eux.

Exercice

Faire un programme Python `fonctionphi(n)` qui calcule (brutalement) $\phi(n)$ en comptant le nombre de classes inversibles. Vous aurez donc besoin de *pgcd(a,b)* (page 74) pour déterminer si une classe x est inversible modulo n.

Puis faire un programme `frequenceInversibles(M,N)` pour calculer la valeur moyenne de $\phi(n)/n$ pour $1 \leqslant n < 500$, $500 \leqslant n < 1000$ et $1000 \leqslant n < 1500$.

Faire aussi `frequenceInversiblesPourImpairs(M,N)` qui ne regarde que les n impairs entre M et N.

Multiplier par π^2 les résultats et commenter.

```
#%%
def fonctionphi(n):
    """\
    Indicatrice d'Euler.

    On suppose n>0.
    """
    if n==1:
        return 1
    return sum(1 for x in range(1,n) if pgcd(n,x)==1)
```

```
#%%
def frequenceInversibles(M,N):
    """\
    Fréquence moyenne des classes inversibles modulo n pour M<= n < N.
    """
    return sum(fonctionphi(n)/n for n in range(M,N))/(N-M)
#%%
def frequenceInversiblesPourImpairs(M,N):
    """\
    Fréquence moyenne des classes inversibles modulo n impair.
    """
    if not M&1: # M est pair
        M +=1   # maintenant il est impair
    return sum(fonctionphi(n)/n for n in range(M,N,2))/((N-M+1)>>1)
```

Mise en œuvre :

```
In [7]: frequenceInversibles(1,500)
Out[7]: 0.6088452464020063

In [8]: frequenceInversibles(500,1000)
Out[8]: 0.607883069907744

In [9]: frequenceInversibles(1000,1500)
Out[9]: 0.6080878882886404

In [10]: from math import pi as Pi

In [11]: Out[9]*Pi**2
Out[11]: 6.0015868985026986

In [12]: frequenceInversiblesPourImpairs(1,500)
Out[12]: 0.8109076289534366

In [13]: frequenceInversiblesPourImpairs(500,1000)
Out[13]: 0.8103123253387697

In [14]: frequenceInversiblesPourImpairs(1000,1500)
Out[14]: 0.8108729831143751

In [15]: _*Pi**2
Out[15]: 8.002995562870092
```

Ainsi, en moyenne, et dans ces eaux là, pour un nombre impair aléatoire si on prend une classe de congruence au hasard, il y a semble-t-il plus de 80% de chances qu'elle soit inversible, donc seulement 20% de chances de prouver que notre nombre impair est composé.

C'est pire que pour notre 2821 qui déjà n'était pas super favorable. Et parfois cela peut être bien pire. Par exemple si $n = p^2$ alors $\phi(n)/n = 1 - p^{-1}$ ce qui veut dire qu'il n'y a qu'une chance sur $\sqrt{n}$ de piocher au hasard une classe non inversible.

Exercice

Soit $n = PQ$ un produit de deux nombres premiers distincts ayant tous les deux le même nombre de chiffres (en décimal). Montrer que la probabilité de piocher au hasard une classe non-inversible est inférieure à $3,5/\sqrt{n}$.

En fait le majorant plus précis est $(\sqrt{10}+1/\sqrt{10})/\sqrt{n}$.

Cela veut dire que tenter d'établir qu'un tel entier est composé en cherchant par pgcd un témoin de sa non-primalité a une (in-)efficacité comparable à la méthode brutale de division par tous les entiers jusqu'à atteindre le premier diviseur (qui dans notre exemple est $P > \sqrt{n}/\sqrt{10}$.)

Un exemple est $n = 63605523217059931493$.

Déjà vous pouvez essayer notre *estpremier(n)* (page 78) de la précédente séance, j'ai peur que ça dure assez longtemps : j'ai lancé le *diviseurspremiers(n)* (page 79) de la semaine dernière qui est comme *estpremier(n)* (page 78) sauf qu'il continue jusqu'à trouver tous les facteurs premiers (et une fois ce premier P trouvé, ça termine de suite car $n/P = Q$ étant plus petit que P^2 le programme sait que Q est forcément premier). Au bout de vingt minutes mon ordinateur était devenu très chaud, les ventilateurs tournaient à plein régime, j'ai abrégé ses souffrances par CTRL–C.

Et donc en prenant au hasard un entier $1 \leqslant x < 63605523217059931493$ et en calculant le pgcd ça ne sera pas mieux. Comme je connais la factorisation je peux calculer qu'il y a dans ce cas 99.99999997476066% de chances de tomber sur une classe inversible! (c'est-à-dire sur un $1 \leqslant x < n$ tel que $\text{pgcd}(x, n) = 1$) et donc seulement 0.00000002524% de chances de prouver par ce biais que n n'est pas un nombre premier.

Heureusement, Fermat est passé par là.

6.3 Témoins de Fermat

```
>>> pow(2,63605523217059931492,63605523217059931493)
48551466010673702078
```

Ce calcul de $2^{n-1} \bmod n$ se fait très vite en Python: nous avions étudié l'exponentiation rapide, et avec le même algorithme où on reduit modulo n chaque multiplication, on peut faire ce genre de calculs rapidement (même à la main on pourrait y arriver, il y au plus $66 \cdot 2 = 132$ multiplications modulaires à faire car $n < 2^{66}$, on me donnait à l'École Primaire des lignes à faire, ça ne doit pas être bien plus long). Le résultat ci-dessus **prouve** que 63605523217059931493 n'est PAS un nombre premier. Car Fermat a démontré $2^{p-1} \equiv 1 \bmod p$ pour tout nombre premier impair, et même plus généralement $a^{p-1} \equiv 1 \bmod p$ pour tout a non divisible par p.

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. Tout $1 \leqslant x < n$ qui n'est pas premier avec n est automatiquement un témoin de Fermat (pourquoi ?), donc cette notion généralise nos témoins de la section précédente. Il y en a (en général) beaucoup plus. Mais certains entiers exceptionnels n, les nombres de Carmichael, n'ont pas de témoins de Fermat autres que les nombres non-inversibles modulo n ; les premiers d'entre eux sont 561, 1105, 1729, 2465, 2821, 6601, 8911. On y retrouve notre 2821 de tout à l'heure.

Par contre 63605523217059931493 n'est pas du tout un nombre de Carmichael. En fait parmi $1 \leqslant x < 63605523217059931493$ il n'y a que **quatre** entiers qui ne sont PAS des témoins de Fermat. Cela fait une probabilité de $99.99999999999999993711238\%$ en prenant un tel x au hasard de prouver que notre entier n'est pas premier.

Ce sont:

```
1
11553595800450117393
```

```
52051927416609814100
63605523217059931492 (congru à -1)
```

```
>>> pow(11553595800450117393,63605523217059931492,63605523217059931493)
1
```

```
>>> pow(52051927416609814100,63605523217059931492,63605523217059931493)
1
```

Exercice

Bon, je révèle que:

$$63605523217059931493 = 7119001843 \times 8934612551$$

avec les facteurs premiers P et Q.

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 du **Lemme** suivant: dans un groupe cyclique (noté multiplicativement) de cardinalité m, le nombre de solutions de l'équation $x^a = e$ est $\text{pgcd}(a, m)$.
>
> Et, bien sûr vous aurez besoin du fait fondamental que pour tout nombre premier P, le groupe multiplicatif $(\mathbb{Z}/P\mathbb{Z})^*$ est **cyclique**.

Lorsque le $\text{pgcd}(P-1, Q-1)$ vaut deux, il y a donc quatre solutions exactement ; les donner explicitement en partant d'une identité de Bezout reliant P et Q. Retrouver les solutions données plus haut.

Montrer que pour tout $n = PQ$ produit de deux nombres premiers impairs distincts, au moins la moitié des $1 \leqslant x < N$ sont des Témoins de Fermat de la non-primalité de N.

Exercice

Faire des routines `estTemoinDeFermat(x,n)` et `nombreDeTemoinsDeFermat(n)`. Pour la première on pourra supposer $1 \leqslant x < n$.

```python
#%%
def estTemoinDeFermat(x,n):
    """\
    Décide si x (on suppose 1<= x < n) est un Témoin de Fermat pour n
    """
    # on pourrait accélérer peut-être, lorsque n est impair,
    # en remplaçant x par n-x si n-x < x
    return pow(x,n-1,n)!=1

#%%
def nombreDeTemoinsDeFermat(n):
    """\
    Compte ceux des 1<= x < n qui sont des témoins de Fermat pour n.
    Pour n impair on n'aurait pas besoin de tester n-1 qui n'est jamais
    Témoin de Fermat car (n-1)^(n-1) est (-1)^(n-1) donc 1 modulo n.
```

```
    """
    if n<=2:
        return 0
    return sum(1 for x in range(2,n) if estTemoinDeFermat(x,n))
```

Exercice

Faire une routine `frequenceTemoinsDeFermat(M,N)` qui fait la moyenne sur les nombres impairs entre `M` et `N` des quotients `nombreDeTemoinsDeFermat(n)/n`. Utiliser la routine pour des tranches successives de cent entiers pour se faire une idée si possible du comportement de la fréquence moyenne des Témoins de Fermat.

(pour le moment on n'étudie que de tout petits entiers, plus bas dans cette feuille on fera des choses plus ambitieuses)

```
#%%
def frequenceTemoinsDeFermat(M,N):
    """\
    Fréquence moyenne des Témoins de Fermat pour M<= n < N et n impair.
    """
    if not M&1: # M est pair
        M +=1   # maintenant il est impair
    return sum(nombreDeTemoinsDeFermat(n)/n for n in range(M,N,2))/((N-M+1)>>1)
```

Voici ce que j'obtiens

```
In [71]: for i in range(1,30):
    ...:     print(i*100, frequenceTemoinsDeFermat(i*100,i*100+100))
    ...:
100 0.5429447824201559
200 0.6481289093093094
300 0.651771594759785
400 0.630596040353454
500 0.6926626824861098
600 0.6598365155934951
700 0.6946890978767608
800 0.6868369445832093
900 0.707714707769848
1000 0.6645545179604452
1100 0.7370626952074392
1200 0.6841760977730031
1300 0.7662970889850398
1400 0.6507808702278696
1500 0.7451549630641597
1600 0.6881131228092259
1700 0.7383865821878337
1800 0.741671133643368
1900 0.7306460799086113
2000 0.7039908846260049
2100 0.7923286581082519
2200 0.6930714802624418
2300 0.6936589086866491
2400 0.7767627124763224
2500 0.7706140249808056
2600 0.6929506654613695
```

```
2700 0.7046685846945487
2800 0.7386203296489673
2900 0.7732942173323234

In [72]: for i in range(100,110):
    ...:     print(i*100, frequenceTemoinsDeFermat(i*100,i*100+100))
    ...:
10000 0.776448391691742
10100 0.7582303507153564
10200 0.7951562472662245
10300 0.7554803638311356
10400 0.797720325363074
10500 0.8226574362067223
10600 0.7549311016673437
10700 0.7773154247888057
10800 0.7970019424457468
10900 0.797027878987372
```

Bien sûr les nombres premiers n'ont aucun témoin de Fermat (et ce sont les seuls) et pour eux la fréquence est nulle. Donc la fréquence sur les nombres composés est un peu supérieure à ce qui est affiché dans les listes ci-dessus. Dans l'ensemble on voit qu'on a, en moyenne, pas mal de témoins de Fermat, donc la stratégie qui consiste à prendre un $1 < x < n$ au hasard aboutira toujours après un assez faible nombre de répétitions si n est composé. Mais s'il est premier, pour éviter de faire un programme dont on ne peut pas majorer a priori le temps d'exécution, il faudrait garder la mémoire des x testés et ne jamais en re-tester un nouveau: au bout d'un (long) moment on aura testé toutes les classes de congruence et donc prouvé que l'entier est premier. (Je rappelle que les nombres de Carmichael ne peuvent pas mettre en échec ceci, car ils ont tout de même des classes de congruence non inversibles, donc des témoins de Fermat).

Je ne sais pas, mais je dois avoir des collègues qui savent, quel est le comportement mathématique précis de la fréquence moyenne des témoins de Fermat, lorsque les entiers deviennent grands. (Il a été établi il y a quelques années qu'il y a une infinité de nombres de Carmichael, je n'ai pas eu le temps avant la séance de TPs de regarder s'il faisait partie des résultats qu'on peut en construire une suite telle que les fréquences associées de témoins de Fermat tendent vers zéro).

6.4 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 $\mathrm{pgcd}(X + 1, n)$ et $\mathrm{pgcd}(X - 1, n)$ 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 $n-1=2^k m$ avec m impair et $k \geqslant 1$. On dit que $1<x<n$ 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 $z=x^{(n-1)/2}$ inclus n'est égal à -1 modulo n.

Exercice

Montrer que tous les témoins de Fermat sont des témoins de Miller et montrer que si l'entier impair n possède un témoin de Miller alors il est un nombre composé.

Exercice

Faire une procédure `estTemoinDeMiller(x,n,m,k)` qui dit si x est un témoin de Miller pour $n=2^k m+1$ avec m impair, $k \geqslant 1$. Les entiers m et k sont pré-calculés car on a l'intention d'appeler cette procédure ensuite de nombreuses fois avec des x différents et le même n.

```
#%%
def estTemoinDeMiller(x,n,m,k):
    """\
    Renvoie True si x est un témoin de Miller pour n.

    ATTENTION:

    - la procédure doit être appelée avec m et k précalculés
      n-1 = 2^k m avec m impair, et k au moins 1,

    - de plus la routine présuppose 0<x<n.
    """
    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 vaudront tous 1, jamais -1 modulo n
        # donc on a un témoin de Miller, mais finalement je préfère
        # commenter le test ci-dessous
        # if y==1:
        #    return True
        if y==n-1:
            return False
    #
    # dans une version antérieure du code, je faisais plus de choses,
    # (je construisais une factorisation à chaque fois qu'on avait
    # un témoin de Miller qui n'était pas un témoin de Fermat)
    # et je sortais parfois du for par un break. Dans ce cas j'avais
    # besoin du else ci-dessous pour avoir une clause qui n'était pas
    # exécutée après un tel break. Mais bref on n'en a pas besoin ici.
    #
    # else:
        # return True
    #
```

```
    # Comme j'ai maintenant supprimé le test y==1 dans la boucle
    # je sais juste que le dernier y n'est pas -1 modulo n
    #
    # S'il vaut 1, alors parmi les carrés itérés à partir de x**m
    # il y a eu un premier moment où on a trouvé 1. Celui d'avant
    # ne pouvait pas être -1, ni 1, donc n est composé.
    #
    # Si y ne vaut pas 1 mod n, alors son carré est soit différent
    # de 1 mod n et on a un témoin de Fermat, soit égal à 1, et
    # comme y n'est ni 1 ni -1, n est composé, on a un témoin de Miller.
    #
    #    Je note en passant que c'était pas si idiot mon test pour
    #    y == 1 mod n dans la boucle, ça nous simplifie la compréhension
    #    du mécanisme, car ici on aurait été sûr que y est ni 1 ni -1.
    return True
```

Important : Le grand avantage des témoins de Miller c'est qu'on a le théorème de Rabin : si n est un entier impair composé alors au moins 75% de ses classes inversibles sont des témoins de Miller.

Comme les classes non inversibles (non nulles) sont aussi des témoins de Miller, on a donc encore plus que cette proportion de 75%. Et comme on va l'explorer plus tard, en moyenne c'est sûrement bien mieux.

Exercice

Faire `nombreDeTemoinsDeMiller(n)` (pour $n > 1$ impair).

Puis faire une routine `frequenceTemoinsDeMiller(M,N)` qui fait la moyenne sur les nombres impairs entre `M` et `N` des quotients `nombreDeTemoinsDeMiller(n)/n`. Utiliser la routine pour des tranches successives de cent entiers pour se faire une idée si possible du comportement de la fréquence moyenne des Témoins de Miller.

```
#%%
def nombreDeTemoinsDeMiller(n):
    """\
    Attention, n doit être impair >1.
    """
    if n==3:
        return 0
    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
    return sum(1 for x in range(2,n-1) if estTemoinDeMiller(x,n,m,k))

#%%
def frequenceTemoinsDeMiller(M,N):
    """\
    Fréquence moyenne des Témoins de Miller pour M<= n < N et n impair.
    """
```

```
    if not M&1: # M est pair
        M +=1   # maintenant il est impair
    return sum(nombreDeTemoinsDeMiller(n)/n for n in range(M,N,2))/((N-M+1)>>1)
```

Voici ce que j'obtiens. On ne constate pas de progrès foudroyant par rapport aux témoins de Fermat. Mais la grande différence est le théorème de Rabin, totalement général.

Important : On pourrait être paniqué par des résultats `<0.75`. Tout ça c'est de la faute des nombres premiers! Ils ont, eux, zéro témoin de Miller, et ça tire la fréquence moyenne vers le bas. Comme la fréquence relative des nombres premiers diminue (en gros), l'effet diminue (en gros) au fur et à mesure des tranches.

Nos résultats numériques ici sont sur des entiers ridiculement petits. Plus bas, vous verrez qu'on va faire des recherches plus audacieuses sur les témoins de Miller. Mais je n'ai pas poussé plus loin l'étude quantitative du plus apporté par rapport aux seuls témoins de Fermat.

```
In [79]: for i in range(1,30):
    ...:     M = i*100
    ...:     N = M+100
    ...:     print(M, frequenceTemoinsDeFermat(M,N), frequenceTemoinsDeMiller(M,N))
    ...:

100 0.5429447824201559 0.5613059115158285
200 0.6481289093093094 0.6649440535723399
300 0.651771594759785 0.667240711924162
400 0.630596040353454 0.6476497830478551
500 0.6926626824861098 0.7119570462755802
600 0.6598365155934951 0.6728469393950488
700 0.6946890978767608 0.7086286541813763
800 0.6868369445832093 0.6950451934864286
900 0.707714707769848 0.7146837350194727
1000 0.6645545179604452 0.6756878761904496
1100 0.7370626952074392 0.7552652050849492
1200 0.6841760977730031 0.6939223055370201
1300 0.7662970889850398 0.7740996190933843
1400 0.6507808702278696 0.6562402403789513
1500 0.7451549630641597 0.7536461823584528
1600 0.6881131228092259 0.6955670796609662
1700 0.7383865821878337 0.7562545264348793
1800 0.741671133643368 0.7522132464136818
1900 0.7306460799086113 0.7365204977223222
2000 0.7039908846260049 0.7130682042073211
2100 0.7923286581082519 0.7972147246329956
2200 0.6930714802624418 0.6972036040062504
2300 0.6936589086866491 0.6977351801810492
2400 0.7767627124763224 0.7961598523814364
2500 0.7706140249808056 0.7766378749608556
2600 0.6929506654613695 0.6980268037062111
2700 0.7046685846945487 0.7146720276703256
2800 0.7386203296489673 0.7558852035688249
2900 0.7732942173323234 0.7774495382005617

In [80]: for i in range(100,110):
    ...:     M = i*100
    ...:     N = M+100
```

```
    ...:     print(M, frequenceTemoinsDeFermat(M,N), frequenceTemoinsDeMiller(M,N))
    ...:

10000 0.776448391691742 0.7788027459843475
10100 0.7582303507153564 0.759530871494882
10200 0.7951562472662245 0.7977809201173511
10300 0.7554803638311356 0.7586199948295501
10400 0.797720325363074 0.7995483603795962
10500 0.8226574362067223 0.8380965388641374
10600 0.7549311016673437 0.7585156306414972
10700 0.7773154247888057 0.7793583149596324
10800 0.7970019424457468 0.7992485213747568
10900 0.797027878987372 0.7988609596599682
```

6.5 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 (il l'est avec une probabilité d'erreur $< 1/4^{\text{reps}}$).

Vous aurez besoin de:

```
>>> from random import randrange
```

```
#%%
def testMillerRabin(n, reps=10):
    """\
    Test de Miller-Rabin "probabiliste" sur n IMPAIR>3.

    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 SEULEMENT PROBABLE, que n est premier.
      La « probabilité d'erreur » est moins de 1/4^{reps}.

    Pour reps=30, 4^{-30} < 10^{-18}
    Pour reps=10, 4^(-10) < 10^{-6}

    Le 1/4 semble très largement surestimé, dès que n a plus de
    quelques chiffres. Si n a des dizaines de chiffres un seul test
    positif semble (en moyenne) déjà un très fort argument que le
    nombre est vraiment premier.
    """
    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
    # deux fois le même x. Mais plus n est grand moins les répétitions
    # de x sont probables (si l'on fait confiance au générateur
    # pseudo-aléatoire de Python)
    for test in range(reps):
        if estTemoinDeMiller(randrange(2,n-1),n,m,k):
            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
```

6.6 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.

```
#%%
premierspluspetitsquetrente = {2,3,5,7,11,13,17,19,23,29}
#%%
def estpetitpremier(n):
    """\
    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 vérifie que aucun de 2, 3, 5, 7, 11, 13, et 17 n'est
    un temoin de Miller pour n.

    Références:

    <http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html>

    <http://mathworld.wolfram.com/StrongPseudoprime.html>
    """
    if n < 30:
        return n in premierspluspetitsquetrente
```

```
    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
    # en Python, "and" arrête intelligemment dès qu'il trouve
    # un truc faux. Par exemple si 2 est un témoin de Miller,
    # la première valeur logique est False, donc ça s'arrête de suite
    return not estTemoinDeMiller(2,n,m,k)\
           and not estTemoinDeMiller(3,n,m,k)\
           and not estTemoinDeMiller(5,n,m,k)\
           and not estTemoinDeMiller(7,n,m,k)\
           and not estTemoinDeMiller(11,n,m,k)\
           and not estTemoinDeMiller(13,n,m,k)\
           and not estTemoinDeMiller(17,n,m,k)
```

À propos, en Python, les évaluations logiques sont « court-circuitées » au mieux. Essayez ceci avec `%timeit` pour vous en convaincre.

```
#%%
import time
def attends():
    time.sleep(.1)# 1 dixième de seconde
    return False
def attendscinqfois():
    return attends() and attends() and attends() and attends() and attends()
```

6.7 `estpremier(n)` (Miller-Rabin-Bach)

À part les supercalculateurs on a aussi les grandes hypothèses non démontrées qui structurent la théorie des nombres. Bach a démontré des théorèmes, en admettant la validité de « l'hypothèse de Riemann généralisée », dont on peut déduire la belle chose suivante:

Important : Tout entier impair composé n possède un témoin de Miller x qui est $\leqslant 2\log(n)^2$ (il s'agit bien sûr du logarithme népérien).

Exercice

Faire `estpremier(n)` qui détermine avec certitude (modulo Riemann généralisé !) si un entier est premier.

```
#%%
premierspluspetitsquetrente = {2,3,5,7,11,13,17,19,23,29}
#%%
def estpremier(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 correcte
    utilise un théorème de Bach qui est conditionnel à l'hypothèse
    de Riemann généralisée, NON ENCORE DÉMONTRÉE.
    """
    if n<=30:
        return n in premierspluspetitsquetrente
    if not n&1: # n est pair
        return False # car n > 2
    # ici n est impair et n-1 = 2^k m avec m impair
    k = 1
    m = n>>1 # égal à (n-1)/2
    while not m&1:
        # tant que m est pair le diviser par deux et incrémenter k
        k += 1
        m >>= 1

    # ALGORITHME DE MILLER RABIN "DÉTERMINISTE"
    # =========================================

    # 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 2(log(2)^2)=0.9609...  < 1
    # donc on pourrait prendre simplement sans se compliquer la vie:
    # X = n.bit_length()**2
    #  pour gagner un peu:
    X = (961*n.bit_length()**2)//1000 + 1
    #    16 pour n<16
    #    25 pour 16<=n<32 (donc pour nous seulement 31)
    #    35 pour 32<=n<64
    #    48 pour 64<=n<128
    # donc attention à n=31 et 33  !!
    # on ne doit pas tester jusqu'à x=X-1 car ce dernier x sera
    # plus grand que n. (Pour n=35, X = 35 et X-1 est < n, ok)

    # Bref, autant faire ce qui suit, au cas où on changerait encore
    # la formule pour X à l'avenir :
    if n-1 <= X:
        X = n-1 # on ira donc au pire jusqu'à x=n-2, tester x=n-1 est inutile
        # car n-1 n'est jamais un Témoin de Miller.

    x = 2 # en effet inutile de tester avec x=1, ce n'est jamais un Témoin
    while x < X:
        if estTemoinDeMiller(x,n,m,k):
            return False
        x += 1
    # on est sorti de la boucle while sans avoir trouvé de témoin de Miller
    # x <= 2(log n)^2
    # On peut donc affirmer que n est un nombre premier.
    # ... si l'Hypothèse de Riemann est vraie ...
    return True
```

6.8 La surprenante efficacité du test probabiliste de Miller-Rabin

Exercice

Faire une routine `trouveVraiFauxPremier(m,nombre=1,reps=1)` qui fera les choses suivantes:

> Elle étudie les nombres impairs avec `m` chiffres (`m<15`). Elle en prend un aléatoire et lui applique le test probabiliste de Miller-Rabin, avec `reps` répétitions (on va surtout utiliser avec une seule). Si le test dit que le nombre est premier le programme le vérifie par l'emploi de *estpetitpremier(n)* (page 94).
>
> À chaque fois qu'un entier a faussement été déclaré premier on l'imprime, ainsi que le nombre total d'entiers testés jusqu'alors, et le nombre de ceux qui ont été déclarés premiers.
>
> On s'arrête après avoir trouvé `nombre` entiers composés faussement déclarés premiers.

Voici (un peu plus bas) le genre de données que j'ai collectées.

- première colonne: nombre total d'entiers impairs testés,
- deuxième colonne: nombre total de ceux déclarés premiers,
- troisième colonne: nombre d'entiers testés depuis le dernier faux premier,
- quatrième colonne: nombre de ceux déclarés premiers depuis le dernier faux, (tous le sont vraiment sauf le tout dernier),
- cinquième colonne: la fréquence dans cette tranche des vrais nombres premiers,
- dernière colonne: le nombre composé qui a été déclaré (faussement) premier.

Par exemple pour les nombres de 8 chiffres il a fallu 294229 tests positifs de Miller Rabin pour en avoir 20 d'inexacts! et tout cela avec un unique test de Miller Rabin probabiliste, sans aucune répétition! Autant dire que la probabilité moyenne qu'un test de Miller Rabin unique dise faux est largement inférieure au fameux 1/4 !

Et aussi, ce que l'on constate expérimentalement, c'est que plus nos nombres ont de chiffres plus il est probable qu'en entier aléatoire déclaré premier le soit vraiment.

```
In [44]: trouveVraiFauxPremier(6,20)
    4201     620    4201     620   0.1473  (1: 420061)
   12794    1902    8593    1282   0.1491  (2: 950779)
   20831    3086    8037    1184   0.1472  (3: 188501)
   20958    3105     127      19   0.1417  (4: 354445)
   22730    3377    1772     272   0.1529  (5: 497377)
   27934    4244    5204     867   0.1664  (6: 959641)
   28553    4343     619      99   0.1583  (7: 587861)
   30802    4723    2249     380   0.1685  (8: 344531)
   32444    4978    1642     255   0.1547  (9: 182527)
   35309    5422    2865     444   0.1546  (10: 996073)
   47123    7180   11814    1758   0.1487  (11: 194221)
   56048    8623    8925    1443   0.1616  (12: 293591)
   56200    8649     152      26   0.1645  (13: 479689)
   59234    9120    3034     471   0.1549  (14: 954271)
   59811    9209     577      89   0.1525  (15: 229411)
   62344    9589    2533     380   0.1496  (16: 410041)
   63321    9728     977     139   0.1412  (17: 819541)
   68703   10583    5382     855   0.1587  (18: 585229)
   73346   11287    4643     704   0.1514  (19: 385003)
   76569   11766    3223     479   0.1483  (20: 512177)
```

```
In [45]: trouveVraiFauxPremier(7,20)
    1271     182    1271     182   0.1424  (1: 3167539)
   30938    4088   29667    3906   0.1316  (2: 6774499)
   37751    4917    6813     829   0.1215  (3: 1207361)
   38536    5013     785      96   0.1210  (4: 7636831)
  103239   13448   64703    8435   0.1303  (5: 5779201)
  157807   20429   54568    6981   0.1279  (6: 2603381)
  191665   24880   33858    4451   0.1314  (7: 7636877)
  195076   25326    3411     446   0.1305  (8: 5195629)
  202194   26240    7118     914   0.1283  (9: 2746477)
  297299   38604   95105   12364   0.1300  (10: 2011969)
  364923   47429   67624    8825   0.1305  (11: 2604331)
  400400   52083   35477    4654   0.1312  (12: 8769949)
  422414   54982   22014    2899   0.1316  (13: 3285829)
  446799   58173   24385    3191   0.1308  (14: 2744989)
  542742   70766   95943   12593   0.1312  (15: 1169101)
  549371   71602    6629     836   0.1260  (16: 1987021)
  567116   73905   17745    2303   0.1297  (17: 3270403)
  567795   73998     679      93   0.1355  (18: 7267051)
  718554   93801  150759   19803   0.1313  (19: 1561921)
  820199  106868  101645   13067   0.1285  (20: 3059101)

In [46]: trouveVraiFauxPremier(8,20)
   32431    3688   32431    3688   0.1137  (1: 14652991)
   74396    8502   41965    4814   0.1147  (2: 71936899)
  296392   33585  221996   25083   0.1130  (3: 10438855)
  392528   44588   96136   11003   0.1144  (4: 49937191)
  460495   52230   67967    7642   0.1124  (5: 19328653)
  572091   64820  111596   12590   0.1128  (6: 69273137)
  696590   78940  124499   14120   0.1134  (7: 43661257)
  891286  100739  194696   21799   0.1120  (8: 97468309)
 1031011  116668  139725   15929   0.1140  (9: 35732497)
 1035438  117158    4427     490   0.1105  (10: 56349751)
 1106076  125273   70638    8115   0.1149  (11: 80918281)
 1161821  131468   55745    6195   0.1111  (12: 25327009)
 1229556  139210   67735    7742   0.1143  (13: 92696911)
 1343956  152109  114400   12899   0.1127  (14: 48809143)
 1630025  184481  286069   32372   0.1132  (15: 96878101)
 1797483  203313  167458   18832   0.1125  (16: 33630409)
 1903119  215266  105636   11953   0.1131  (17: 36092431)
 1990552  225095   87433    9829   0.1124  (18: 49991663)
 2363273  267441  372721   42346   0.1136  (19: 32239111)
 2598606  294429  235333   26988   0.1147  (20: 33065281)

In [47]: trouveVraiFauxPremier(9,10)
  222978   22356  222978   22356   0.1003  (1: 273480637)
  875363   87638  652385   65282   0.1001  (2: 437247841)
 1210098  121030  334735   33392   0.0998  (3: 159764257)
 1659799  166127  449701   45097   0.1003  (4: 184603417)
 2272538  227792  612739   61665   0.1006  (5: 423302657)
 2563108  257061  290570   29269   0.1007  (6: 754550539)
 3696508  370528 1133400  113467   0.1001  (7: 268626181)
 4664904  467579  968396   97051   0.1002  (8: 258877721)
 5185289  519317  520385   51738   0.0994  (9: 455973841)
 5360725  536529  175436   17212   0.0981  (10: 160587841)
```

```
In [48]: trouveVraiFauxPremier(10,5)
 2772680  248359 2772680  248359   0.0896  (1: 2924783051)
 3748219  336648  975539   88289   0.0905  (2: 1229751667)
10987808  987001 7239589  650353   0.0898  (3: 6916728821)
23774751 213693012786943 1149929   0.0899  (4: 3326803201)
28902606 2597290 5127855  460360   0.0898  (5: 2515507021)
```

Important : Vous pouvez constater l'incroyable stabilité de la probabilité qu'un nombre impair soit premier. Aussi au passage on voit le nombre énorme de nombres premiers ! Par contre il y a une grande volatilité du temps d'attente entre la découverte de deux faux premiers (bien sûr ces faux premiers n'ont rien d'intrinsèquement faux, si on refait sur eux un test de Rabin Miller, il va voir qu'ils sont composés … enfin … peut-être on ne sait jamais avec ces satanés tests probabilistes ! et d'ailleurs si un nombre a été déclaré à tort premier, c'est peut-être qu'il a des choses à se reprocher et qu'il a plus de chances que les autres qu'un autre test le déclare à nouveau faussement premier …).

Mais surtout on voit qu'en moyenne un test probabiliste de Miller-Rabin a sûrement beaucoup moins que une chance sur quatre de se tromper, et que plus les nombres sont grands plus c'est valable.

MÀJ 15 février: il existe en effet des résultats théoriques qui vont dans ce sens. 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. D'après

> 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

$$p(k) < k^2 4^{2-\sqrt{k}}$$

On notera que cette majoration ne commence à devenir inférieure à 1 que pour $k \geqslant 64$ (ce qui correspond a des nombres de 20 chiffres décimaux au moins). 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 ($524288 \leqslant n < 1048576$), des tests de Miller Rabin aléatoires uniques en ont déclaré 1477188 premiers, et parmi ceux-ci 838 étaient en fait composés. Cela donne une évaluation $p(20) \approx 0.00057$.

Soit $q(m)$ l'analogue de $p(k)$ mais pour les nombres impairs aléatoires de m chiffres décimaux. En faisant chauffer mon ordinateur avec des routines comme `trouveVraiFauxPremier(m,nombre)` 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 simple test de Fermat aléatoire. Les probabilités d'erreurs sont un peu plus élevées, mais elles décroissent aussi :

```
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
```

Dans les nombres ci-dessus il ne faut surtout pas faire confiance aux décimales. Par ailleurs dans mes tests de Fermat ou de Miller-Rabin, la classe de congruence est toujours prise distincte de $\pm 1 \bmod N$. Des tests autorisant ces classes auraient des probabilités plus grandes d'échec.

Pour illustrer la volatilité, voici un exemple de recherche de faux premiers de 6 chiffres ayant survécu à deux tests de Rabin Miller :

```
In [49]: trouveVraiFauxPremier(6,5,2)
    7331    1137    7331    1137   0.1550  (1: 736291)
  421115   64412  413784   63275   0.1529  (2: 556421)
  455250   69608   34135    5196   0.1522  (3: 219781)
 1438842  220294  983592  150686   0.1532  (4: 665281)
 1443288  220988    4446     694   0.1559  (5: 479119)
```

D'après les deux dernières lignes après avoir vu défiler 150685 vrais premiers pour voir le nombre composé 665281 il a suffi d'attendre ensuite le 694e pour trouver un nouveau faux premier 479119 !

Pour les nombres premiers de six chiffres nous avons obtenu 20 erreurs pour 11766 cas. Mais j'ai recommencé en demandant 100 faux premiers. Voici en ne conservant que la dernière ligne :

```
In [50]: trouveVraiFauxPremier(6,100)
  825235  125739    7332    1171   0.1596  (100: 473639)
In [51]: trouveVraiFauxPremier(6,100)
  740856  113978   11313    1753   0.1549  (100: 871009)
In [52]: trouveVraiFauxPremier(6,100)
  765866  117666    5701     866   0.1517  (100: 928801)
```

On a 100 erreurs pour plus de 110000 cas … peut-être mon ordinateur lors du test avec 20 erreurs était particulièrement mal luné. Je refais :

```
In [53]: trouveVraiFauxPremier(6,20)
  172826   26242    9809    1504   0.1532  (20: 438751)
In [54]: trouveVraiFauxPremier(6,20)
  146376   22458    1097     175   0.1586  (20: 169541)
In [55]: trouveVraiFauxPremier(6,20)
  147232   22704     585     104   0.1761  (20: 206981)
In [56]: trouveVraiFauxPremier(6,20)
  110519   16882    5840     880   0.1505  (20: 577011)
```

En effet. La morale c'est que pour estimer à peu près correctement les probabilités il faut faire tourner suffisamment pour avoir cumulé au moins plusieurs centaines d'erreurs et pas seulement vingt comme ici, ou même cent.

Pour finir, essayons de trouver des nombres premiers de sept chiffres qui auront survécu à trois tests de Rabin-Miller :

```
In [57]: trouveVraiFauxPremier(7,5,3)
10016000 1304225
KeyboardInterrupt
```

J'ai arrêté par CTRL–C. À ce stade le programme avait examiné plus de dix millions de nombres impairs (avec des répétitions car il n'y a que quatre millions cinq cent mille nombres impairs avec sept chiffres), avait trouvé un million trois cent quatre mille deux cent vingt cinq candidats (pas tous distincts) à être premiers et tous l'étaient vraiment.

Vu le temps que j'ai dû attendre pour obtenir 5 faux premiers de dix chiffres ayant été les seuls sur 2597290 à tromper un unique test de Miller Rabin, je n'ose sûrement pas lancer la recherche d'un nombre impair composé à dix chiffres qui aurait survécu à 2 tests de Miller Rabin !

Voici le code que j'ai utilisé. Il emploie les routines *testMillerRabin(n, reps)* (page 93) et *estpetitpremier(n)* (page 94) :

Note : La technique pour imprimer des résultats les uns sur les autres sur la même ligne ne fonctionne dans Spyder que dans une console IPython, pas dans la console Python avec son invite >>>.

```
#%%
from sys import stdout
from random import randrange
def trouveVraiFauxPremier(m,nombre=1,reps=1):
    """\
    Trouve un nombre composé avec m chiffres qui a survécu
    à un (ou plusieurs) tests de Miller Rabin.

    ATTENTION: m<=14 pour que le test de validation qu'un nombre
    est vraiment premier soit garanti sans exception.
    """
    k, K, j, J, l = 0, 0, 0, 0, 0
    N, M = 10**(m-1)+1, 10**m
    while True:
        k += 1
        x=randrange(N,M,2)
        if testMillerRabin(x,reps):
            j +=1
            if not estpetitpremier(x):
                l += 1
                K += k # nombre total d'essais jusqu'à présents
                J += j # nombre total de PSEUDOS nombres premiers trouvés
                # parmi eux J-l étaient de VRAIS nombres premiers
                # (j-1)/k est la proportion de vrais nombres premiers
                # trouvés dans la dernière tranche de nombres impairs
                # aléatoire
                print('\r{:8}{:8}{:8}{:8} {:8.4f}  ({}: {})'\
                      .format(K, J, k, j, (j-1)/k, l, x))
                stdout.flush()
                j, k = 0, 0
                if l == nombre:
                    return None
        if k%1000==0:
            print ('\r{:8}{:8}'.format(k, j), end='')
            stdout.flush()
```

6.9 Construire de grands nombres premiers

Voici pour vous amuser à produire des nombres premiers (ou probablement premiers) aléatoires, ayant quelques dizaines de chiffres.

```
#%%
def pseudopremieraleatoire(m, reps=1):
    """\
    Trouve un nombre pseudo-premier p avec m chiffres.

    reps est le nombre de tests de Miller. Par défaut un seul test!
    Pour des nombres de dizaines de chiffres, semble suffisant...
    """
    # On calcule une fois pour toutes les bornes :
    A = 10**(m-1)
    A, B = A+1, 10*A
    # A = 10**(m-1)+1, B = 10**m
    while True:
        p = randrange(A, B, 2)
        if testMillerRabin(p, reps):
            return p
#%%
def pseudopremieraleatoirebis(A, B):
    """\
    Sous-routine de premieraleatoire(m).

    Trouve un nombre pseudo-premier p dans [A, B[ avec A IMPAIR.

    Fait un unique test de Miller-Rabin.
    """
    while True:
        p = randrange(A, B, 2)
        if testMillerRabin(p, 1):
            return p
#%%
def premieraleatoire(m):
    """\
    Trouve un nombre (vraiment) premier n avec 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 pour
    sélectionner un candidat puis valide sa primalité en
    vérifiant qu'aucun x <= 2(log n)^2 n'est un témoin de Miller.
    En cas d'échec on recommence.
    """
    # On calcule une fois pour toutes les bornes:
    A = 10**(m-1)
    A, B = A+1, 10*A
    # donc A = 10**(m-1)+1, B = 10**m
    if m<15:
        while True:
            n = randrange(A, B, 2)
            if estpetitpremier(n):
                return n
```

```
    else:
        while True:
            n = pseudopremieraleatoirebis(A, B)
            if estpremier(n):
                return n
```

MÀJ le 14 février :

— j'ai un peu amélioré les routines de manières à ne pas refaire plusieurs fois les mêmes choses, c'est pour cela que `pseudopremieraleatoirebis(A,B)` est utilisée de préférence à `pseudopremieraleatoire(m)` par `premieraleatoire(m)`.

— Aussi j'ai testé le fait qu'il est plus rapide d'utiliser des choses comme `randrange(A,B,2)` (avec `A` impair) pour obtenir un nombre impair aléatoire que de faire `randrange(A,B)` et de rejeter les nombres pairs :

```
#%%
def testrandrangeA(m, nombre):
    """\
    Pour tester l'efficacité relative de randrange avec impair
    ou tous
    """
    A = 10**(m-1)
    A, B = A+1, 10*A
    while nombre:
        p = randrange(A, B, 2)
        nombre -=1
    return None
#%%
def testrandrangeB(m, nombre):
    """\
    Pour tester l'efficacité relative de randrange avec impair
    ou tous
    """
    A, B = 10**(m-1), 10**m
    while nombre:
        p = randrange(A, B)
        if p&1:
            nombre -=1
    return None
#%%
```

```
In [31]: %timeit testrandrangeA(100,10000)
10 loops, best of 3: 32.3 ms per loop

In [32]: %timeit testrandrangeB(100,10000)
10 loops, best of 3: 48.9 ms per loop

In [33]: %timeit testrandrangeA(100,10000)
10 loops, best of 3: 32.1 ms per loop

In [34]: %timeit testrandrangeB(100,10000)
10 loops, best of 3: 48.8 ms per loop

In [35]: %timeit testrandrangeA(100,10000)
10 loops, best of 3: 32.2 ms per loop

In [36]: %timeit testrandrangeB(100,10000)
```

```
10 loops, best of 3: 49 ms per loop
```

Pouvez-vous donner une explication plausible au rapport `48/32=3/2` ?

Vous pouvez maintenant construire de grands nombres composés avec de grands facteurs, mais gare à ne pas les perdre ces facteurs !

Date de dernière modification : `15-02-2015 à 18:58:21 CET`.

7 Feuille de travaux pratiques 5

7.1 Avertissement

Je rappelle l'importance de s'être familiarisé(e) avec la documentation officielle [75] de `Python3`.

Nous allons travailler avec des objets de type `list` [76]. Dans la liste des méthodes associées [77] du tutoriel on trouve `list.sort()` [78].

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 est plus ou moins facile de produire des variantes travaillant « sur place », à savoir sans nécessiter une copie initiale, même de type « superficiel », de la liste `L` ; pour le tri par tas, tel qu'implémenté ci-dessous, c'est immédiat, suffit de travailler directement sur la liste donnée en argument).

7.2 Tri par fusion

7.2.1 `fusionne(L, n, i, j)`

Exercice

75. https://docs.python.org/3/tutorial/introduction.html
76. https://docs.python.org/3/library/stdtypes.html#list
77. https://docs.python.org/3/tutorial/datastructures.html#more-on-lists
78. https://docs.python.org/3/library/stdtypes.html#list.sort

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 :

```
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).

La routine fusionne(L, n, i, j) doit réordonner les entrées d'indices
i à min(n, j+(j−i)−1). C'est ce qu'on appelle la 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.

```python
#%%
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. De plus il devrait
       être mieux documenté, mais je reviens de vacances, et je suis
       plus vieux qu'avant.
    """
    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'assignations : 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` assignations. 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` assignations. 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` assignations. Comme `q<=p` c'est mieux.

Exercice

Faites `fusionneB(L, n, i, j)` qui implémente la méthode décrite à la fin du paragraphe précédent.

```
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.
    """
    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).

7.2.2 `triparfusion(L)`

Exercice

Faire une fonction `triparfusion(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, 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 triparfusion(liste):
    """Trie une liste par ordre ascendant, par fusion.

    Ne modifie pas la liste initiale.

    Opère par fusions itérées."""

    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 triparfusion(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 triparfusionB(N)
10 loops, best of 3: 3.07 ms per loop
```

On est environ 126 fois plus lent…

On a utilisé `sorted()` [79] 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 triparfusion(U)
10 loops, best of 3: 4.01 ms per loop

In [28]: %timeit -n 10 triparfusionB(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 triparfusion(U)
10 loops, best of 3: 3.9 ms per loop

In [33]: %timeit -n 10 triparfusionB(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 triparfusion(U)
10 loops, best of 3: 3.89 ms per loop

In [36]: %timeit -n 10 triparfusionB(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

79. https://docs.python.org/3/library/functions.html#sorted

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

7.3 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` [80], 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.

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 un même tableau (plus précisément une `list` [81]) `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 (donc pas de condition si `i` est une feuille).

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

80. https://docs.python.org/3/library/stdtypes.html#list
81. https://docs.python.org/3/library/stdtypes.html#list

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.

7.3.1 etendletas(L, n)

Exercice

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 :

```
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 etendletas(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.
    """
    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
    # faut avouer que Python est rudement compact.
```

7.3.2 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 :

```
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.
```

```
#%%
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).
    """
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
```

7.3.3 tripartas(L)

Exercice

Faire une fonction `tripartas(L)` qui produira une nouvelle liste triée par ordre ascendant.

On implémentera l'algorithme suivant :

```
En entrée une liste L. Soit n sa longueur.

En faire une copie superficielle M.

Transformer M en tas en appelant etendletas(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...
```

```
#%%
def tripartas(L):
    """Tri par tas.

       Je devrais me décider à mettre des majuscules dans mes noms
       de routines.
    """
    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
        etendletas(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 tripartas(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 tripartas(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 = tripartas(U)

In [62]: estordonnee(V)
Out[62]: True
```

Sur un autre ordinateur, j'ai commencé par obtenir des temps assez différents :

```
In [4]: import random

In [7]: N=list(range(1000,0,-1))

In [8]: P=tripartas(N)

In [9]: estordonnee(P)
Out[9]: True

In [10]: %timeit -n 10 tripartas(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=tripartas(U)

In [15]: estordonnee(V)
Out[15]: True

In [16]: %timeit -n 10 tripartas(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 tripartas(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 [82].

La routine `estordonnee(liste)` est définie dans la *Feuille de travaux pratiques 2* (page 33).

Date de dernière modification : `08-12-2015 à 13:41:35 CET.`

82. http://ipython.readthedocs.org/en/stable/interactive/magics.html?highlight=magic%20alias#magic-timeit

8

Examen du 20 février 2015

8.1 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 :

   ```
   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`[83] `(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)` ?

8.2 Exercice 2

On va travailler avec des matrices 2×2. 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.

8.3 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`.

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

1. Faire une procédure `ListeM(N)` qui produit pour `N>=1` la `list`[84] `[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`[85] `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`.

8.4 Exercice 4

On a une `list`[86] `liste` **déjà triée** par ordre croissant, sans éléments identiques. On veut une procédure `PlusGrandPlusPetit(x,liste)` qui renvoie un `tuple`[87] `(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]
```

84. https://docs.python.org/3/library/stdtypes.html#list
85. https://docs.python.org/3/library/stdtypes.html#list
86. https://docs.python.org/3/library/stdtypes.html#list
87. https://docs.python.org/3/library/stdtypes.html#tuple

8.5 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.

9

Examen du 10 décembre 2015

9.1 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()` [88] 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` [89] peut avoir des entrées qui sont d'autres objets, par exemple des `tuple` [90] :

```
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)
```

88. https://docs.python.org/3/library/functions.html#pow
89. https://docs.python.org/3/library/stdtypes.html#list
90. https://docs.python.org/3/library/stdtypes.html#tuple

4. Lorsque l'on applique la fonction `sorted()` [91] 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` [92] se retrouvent classés par ordre croissant de leurs premières coordonnées.

9.2 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]
```

91. https://docs.python.org/3/library/functions.html#sorted
92. 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,
```

9.3 Exercice 2

On suppose donné deux `list`[93] `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.
```

93. 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`[94] 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
```

9.4 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
```

94. 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
```

9.5 Exercice 4

Ce dernier exercice suppose que *Exercice 2* (page 125) et *Exercice 3* (page 126) 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`[95] 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)
```

95. 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`[96] 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)
```

96. 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`[97] 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`[98] `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]`.

97. https://docs.python.org/3/library/stdtypes.html#tuple
98. 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
```

9.6 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.

10

Examen du 1er juin 2016

10.1 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()` [99] 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`.

99. 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.

10.2 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.

10.3 Exercice 2

1. On rappelle la notion de *Témoins de Miller* (page 89) 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 89) 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.

10.4 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.

11

Examen du 8 décembre 2016

11.1 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.

11.2 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 ?

11.3 Exercice 2

Cet exercice est dans la continuation de l'*Exercice 1* (page 144).

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 144)) 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.
```

11.4 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.

11.5 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.

11.6 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()` [100] 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
```

100. 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.

12

Examen du 8 juin 2017

12.1 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 $10^{N-1} \leqslant n < 10^N$.

12.2 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.

12.3 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` ?

12.4 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^{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.

13

Téléchargement

http://jf.burnol.free.fr/MAO-M1.pdf (version du 8 juin 2017, avec examens).

Date de dernière modification : 08-06-2017 à 16:47:24.