Expo_Rapide

April 2, 2023

1 L'exponentiation rapide

Marc Lorenzi - 28 juin 2018

[1]:
```
import matplotlib.pyplot as plt
from math import log, sqrt
```

[2]:
```
plt.rcParams['figure.figsize'] = (8, 3)
```

Comment calculer le millionième nombre de Fibonacci en moins de 3 secondes ? Lisez et vous saurez.

1.1 1. L'algorithme

1.1.1 1.1 La vision naïve du calcul des puissances

Soit $\mathcal{M}$ un monoïde de neutre e dont la loi est notée multiplicativement. Pour tout $x \in \mathcal{M}$, on définit par récurrence sur n la $n$ième puissance de x, x^n par :

- $x^0 = e$
- Pour tout $n \in \mathbb{N}^*$, $x^n = x^{n-1}x$.

Pour éviter les complications inutiles, supposons pour le moment que x est un nombre (entier, réel, complexe, peu importe). La définition mathématique des puissances nous donne sans aucun doute un algorithme pour les calculer. La fonction `puissance_naive` prend en paramètre un nombre x et un entier naturel n. Elle renvoie x^n.

[3]:
```
def puissance_naive(x, n):
    if n == 0: return 1
    else: return puissance_naive(x, n - 1) * x
```

[4]:
```
puissance_naive(2, 11)
```

[4]: 2048

Python n'aime pas un trop grand nombre d'appels récursifs imbriqués : essayez donc ci-dessus de calculer 2^{3000}. Mais il est très facile de réécrire notre fonction avec une simple boucle. On en profite pour ajouter un compteur `c` qui enregistre le nombre de multiplications effectuées par la fonction.

[5]:
```
def puissance_naive(x, n):
    p = 1
    c = 0
    for k in range(n):
        p = p * x
        c += 1
    return (p, c)
```

[6]:
```
puissance_naive(2, 3000)
```

[6]:
```
(1230231922161117176931558813276752514640713895736833715766118029160058800614672
9487753600678385934595824296492540518049085128841808982368235850824820653483312 3
4959350355845017413023320111360666922624728239756880416434478315693675013413090 7
5720869037679329665881066294182449348845172650530371291600534674790862370267348 0
9193539368131057366204023527447769038404778836511003224093019834883638029305404 8
2487909763484098253940728685132044408863734754271212592471778643949486688511721 0
5156197043278074745482377680846418069710308386181218434856552274019579668262220 5
5118455120805520103100502558015893496459280011337454742207150136834139075427790 6
3759833876101354235184245096670042160720629411581502371248008430447184842098610 3
2058041799220666224732872212208851364368390767036020916265367064113093699700217 0
5006755013747239987660058275793007232534748906122501351718891748990799112915123 9
977387217851901822998937 6,
 3000)
```

Rien de surprenant, il faut faire 3000 multiplications pour calculer 2^{3000}. Imaginons maintenant que l'on veuille calculer $2^{10^{10}}$. Inutile d'essayer, il faudrait faire dix milliards de multiplications, qui plus est avec des entiers énormes. Alors comment calculer des puissances avec d'énormes exposants ? Il nous faut une idée géniale.

1.1.2 1.2 L'idée géniale

Mettons que je veuille calculer 2^{16}. Je peux remarquer que $2^{16} = ((2^2)^2)^2)^2$. Chaque élévation au carré comptant pour une multiplication, cela fera au total 4 multiplications, et pas 16. On tient quelque chose.

Maintenant, me direz vous, c'était facile parce que 16 est une puissance de 2. Et si je veux calculer 2^{13} ? Eh bien j'écris $2^{13} = 2^8 2^4 2^1 = (2^4)^2 2^4 2^1$. Il suffira de 5 multiplications. Sur l'exemple, on voit. Mais comment faire dans le cas général ?

Prenons d'abord les choses "à l'envers", c'est à dire récursivement. Je veux écrire une fonction puissance qui sache calculer x^n. Si $n = 0$ c'est évident. Sinon, $n = 2p$ ou $n = 2p + 1$, où p est un entier. Je calcule $y = (x^2)^p$. Comment ? Eh bien en calculant $x \times x$ puis en appelant récursivement la fonction puissance avec le paramètre $p < n$. Et ensuite ?

- Si $n = 2p$ est pair, alors $y = x^{2p} = x^n$.
- Si $n = 2p + 1$ est impair, alors $xy = xx^{2p} = x^{2p+1} = x^n$.

D'où notre fonction. Comme dans le cas naïf, un compteur c enregistre le nombre de multiplications effectuées.

[7]:
```
def puissance1(x, n):
    if n == 0: return (1, 0)
    else:
        (y, c) = puissance1(x * x, n // 2)
        if n % 2 == 0: return (y, c + 1)
        else: return (x * y, c + 2)
```

[8]:
```
puissance1(2, 3000)
```

[8]:
```
(1230231922161117176931558813276752514640713895736833715766118029160058800614672
9487753600678385934595824296492540518049085128841808982368235850824820653483312 3
4959350355845017413023320111360666922624728239756880416434478315693675013413090 7
5720869037679329665881066294182449348845172650530371291600534674790862370267348 0
9193539368131057366204023527447769038404778836511003224093019834883638029305404 8
2487909763484098253940728685132044408863734754271212592471778643949486688511721 0
5156197043278074745482377680846418069710308386181218434856552274019579668262220 5
5118455120805520103100502558015893496459280011337454742207150136834139075427790 6
3759833876101354235184245096670042160720629411581502371248008430447184842098610 3
2058041799220666224732872212208851364368390767036020916265367064113093699700217 0
5006755013747239987660058275793007232534748906122501351718891748990799112915123 9
9773872178519018229989376,
 19)
```

Eh oui, 19 multiplications seulement pour élever à la puissance 3000. Essayons une élévation à la puissance dix milliards. Mais pas $2^{10^{10}}$ puisque ce nombre possède environ trois milliards de chiffres.

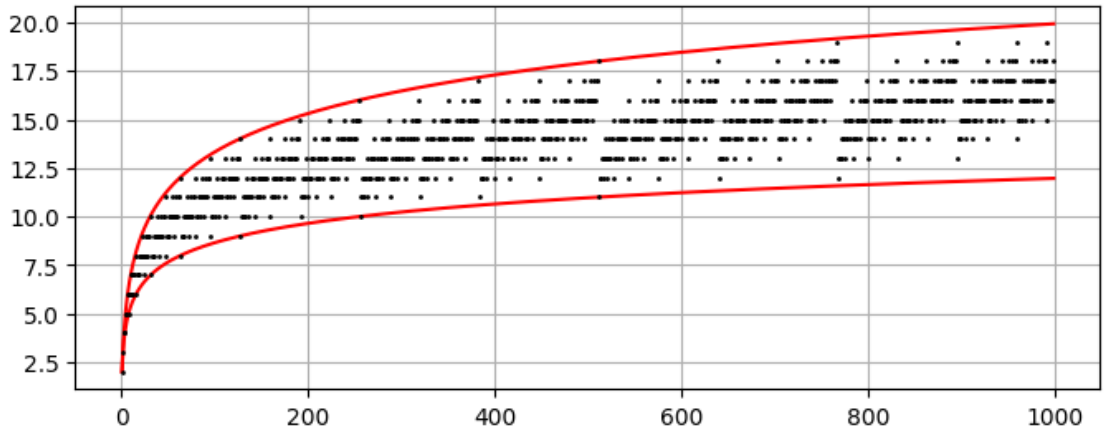
[9]:
```
puissance1(1.000000001, 1e10)
```

[9]:
```
(22026.482191845393, 45)
```

Mais oui, 45 multiplications, et pas 10 milliards comme avec l'algorithme naïf.

Traçons maintenant le graphe du nombre de multiplications en fonction de n. On trace aussi 2 courbes en rouge. Elles ne sont pas tout à fait choisies au hasard : ce sont les courbes des fonctions $x \mapsto \lg x + 2$ et $x \mapsto \lg(x+1)$, où lg désigne le logarithme en base 2. Patience, on en reparle plus loin …

[10]:
```
rg = range(1, 1000)
s1 = [puissance1(2, n)[1] for n in rg]
s2 = [log(n) / log(2) + 2 for n in rg]
s3 = [2 * log(n + 1) / log(2) for n in rg]
plt.plot(rg, s2, 'r')
plt.plot(rg, s3, 'r')
plt.plot(rg, s1, 'ko', ms=1)
plt.grid()
plt.show()
```



Que voit-on ?

- Les points noirs se répartissent de façon compliquée. Eh oui, le nombre de multiplications n'est pas une fonction croissante de n. Notre algorithme a une complexité compliquée :-).
- Les points noirs sont entre les courbes rouges et on a même certains points noirs SUR les courbes. Elles ont l'air vraiment bien choisies.

Nous allons montrer le second point.

1.1.3 1.3 La complexité - Relations de récurrence

Je recopie ici la fonction puissance pour l'avoir sous les yeux. J'ai également éliminé les références au compteur de multiplications.

[11]:
```
def puissance2(x, n):
    if n == 0: return 1
    else:
        y = puissance2(x * x, n // 2)
        if n % 2 == 0: return y
        else: return x * y
```

Pour tout $n \in \mathbb{N}$, notons C_n le nombre de multiplications nécessaires au calcul de x^n. Cette quantité ne dépend pas de x : cela peut être montré par récurrence forte sur n, je ne le ferai pas.

Proposition : On a :

- $C_0 = 0$
- Pour tout $p > 0$, $C_{2p} = C_p + 1$
- Pour tout $p \geq 0$, $C_{2p+1} = C_p + 2$

Démonstration : Les deux premiers points sont laissés au lecteur. Traitons le cas de C_n, où $n = 2p + 1$, $p \geq 0$. Comme $n > 0$, le premier test échoue. La fonction calcule donc x^2, ce qui fait une multiplication. Puis elle s'appelle récursivement avec p comme second paramètre. L'appel

récursif effectue C_p multiplications. Enfin, comme n est impair, le second test échoue et le calcul de xy effectue encore une multiplication. Au total : $C_p + 2$ multiplications, comme prévu.

Écrivons une fonction `complexite` qui calcule la complexité de notre exponentiation rapide. Oui, je sais, ça donne un peu le vertige, surtout si je commence à me demander quelle est la complexité de la fonction `complexite` :-).

[12]:
```
def complexite(n):
    if n == 0: return 0
    else:
        c = complexite(n // 2)
        if n % 2 == 0: return c + 1
        else: return c + 2
```

[13]:
```
print([(n, complexite(n)) for n in range(1, 20)])
```

```
[(1, 2), (2, 3), (3, 4), (4, 4), (5, 5), (6, 5), (7, 6), (8, 5), (9, 6), (10,
6), (11, 7), (12, 6), (13, 7), (14, 7), (15, 8), (16, 6), (17, 7), (18, 7), (19,
8)]
```

Exercice obligé : Soit A_n le nombre d'additions effectuées lors de l'appel à `complexite(n)`. Montrer que $A_n = C_n$. En d'autres termes, la complexité de `complexite` est la même que celle de la fonction qui a pour complexité `complexite` :-).

1.1.4 1.4 Encadrements de la complexité

Montrons tout d'abord une majoration de C_n, en accord avec ce que nous avons "vu" sur le graphe un peu plus haut.

Proposition. Pour tout $n \in \mathbb{N}$, $C_n \leq 2\lg(n+1)$, où lg désigne le logarithme en base 2.

Démonstration. On fait une récurrence forte sur n. Pour $n = 0$ c'est évident puisque

$$C_0 = 0 \leq 2\lg(0+1) = 0$$

Soit $n > 0$. Supposons l'inégalité vraie pour tous les entiers strictement inférieurs à n.

- Cas 1 : $n = 2p$, où $p > 0$. On a

$$C_n = C_p + 1 \leq 2\lg(p+1) + 1$$

par l'hypothèse de récurrence. On a gagné si on montre que

$$2\lg(p+1) + 1 \leq \lg(2p+1)$$

ou encore

$$\lg \frac{2p+1}{p+1} \geq \frac{1}{2}$$

Soit $f : x \mapsto \frac{2x+1}{x+1}$, définie sur $\mathbb{R}_+$. f est dérivable, et pour tout $x \geq 0$ on a

$$f'(x) = \frac{1}{(x+1)^2} > 0$$

Notre fonction est donc strictement croissante sur $\mathbb{R}_+$. Ainsi, pour tout entier $p \geq 1$, $f(p) \geq f(1) = \frac{3}{2}$. Donc,

$$2 \lg \frac{2p+1}{p+1} \geq 2 \lg \frac{3}{2} \geq 2 \lg \sqrt{2} = 1$$

- Cas 2: $n = 2p + 1$ où $p \geq 0$. On a

$$C_n = C_p + 2 \leq 2 \lg(p+1) + 2 = 2 \lg(2(p+1))$$

(eh oui, $\lg 2 = 1$!). Bref,

$$C_n \leq 2 \lg(2p+2) = 2 \lg(n+1)$$

Ce cas était plus facile que l'autre.

Passons à la minoration.

Proposition. Pour tout entier $n \geq 1$, $C_n \geq \lg n + 2$.

Démonstration. Encore une fois ce sera une récurrence forte. Mais à partir de $n = 1$, parce que $\lg 0$ ça fait mauvais effet. Pour $n = 1$, on a $C_1 = 2 = \lg 1 + 2$, donc tout va bien.

Soit $n > 1$. Supposons l'inégalité vraie pour tous les entiers strictement inférieurs à n et strictement positifs.

- Cas 1 : $n = 2p$ où $p > 0$. On a

$$C_n = C_p + 1 \geq \lg p + 2 + 1 = \lg(2p) + 2 = \lg n + 2$$

- Cas 2 : $n = 2p + 1$ où $p > 0$. On a

$$C_n = C_p + 2 \geq \lg p + 2 + 2$$

On a gagné si on montre que

$$\lg p + 2 \geq \lg(2p+1)$$

ou encore

$$\lg \frac{2p+1}{p} \leq 2$$

On pose, pour $x > 0$,

$$f(x) = \frac{2x+1}{x}$$

f est dérivable, de dérivée

$$f'(x) = -\frac{1}{x^2} < 0$$

Notre fonction est donc strictement décroissante sur $\mathbb{R}_+^*$. Ainsi, pour tout entier $p \geq 1$, $f(p) \leq f(1) = 3$. Donc,

$$\lg \frac{2p+1}{p} \leq \lg 3 \leq \lg 4 = 2$$

Bilan. Pour tout $n \geq 1$,

$$2 + \lg n \leq C_n \leq \lg(n+1)$$

Le nombre de multiplications effectuées par la fonction d'exponentiation rapide est **logarithmique** en n. D'où l'appellation « rapide ».

1.1.5 1.5 La valeur exacte de la complexité

Il se trouve que l'on peut calculer C_n de façon exacte (si l'on peut dire).

Proposition. Pour tout $n \in \mathbb{N}$, $C_n = \ell(n) + \gamma(n)$, où $\ell(n)$ est le nombre de chiffres de l'écriture binaire de n, et $\gamma(n)$ est le nombre de 1 dans cette même écriture (On convient que $\ell(0) = 0$).

Démonstration. En base 2, multiplier par 2, c'est juste ajouter un 0 ! On remarque donc que

- $\ell(0) = \gamma(0) = 0$.
- Pour tout $p \geq 1$, $\ell(2p) = \ell(p) + 1$ et $\gamma(2p) = \gamma(p)$.
- Pour tout $p \geq 0$, $\ell(2p+1) = \ell(p) + 1$ et $\gamma(2p+1) = \gamma(p) + 1$.

Posons, pour tout entier n, $C'_n = \ell(n) + \gamma(n)$. On a $C'_0 = C_0 = 0$ et les suites (C_n) et (C'_n) vérifient les mêmes relations de récurrence. On en déduit (par récurrence) que $C_n = C'_n$.

Exercice. Montrer que pour tout $n \geq 1$, $\ell(n) = \lfloor \lg n \rfloor + 1$.

Remarque :

- Supposons que $n = 2^k, k \geq 0$ est une puissance de 2. Alors $\ell(n) = k + 1$ et $\gamma(n) = 1$, donc $C_n = k + 2 = \lg n + 2$. Ainsi, notre minorant de C_n est optimal, il est égal à C_n pour des valeurs de n aussi grandes que l'on veut.
- Supposons maintenant que $n = 2^k - 1, k \geq 0$. Alors $\ell(n) = k$ et $\gamma(n) = k$, donc $C_n = 2k = \lg(n+1)$. Ainsi, notre majorant de C_n est optimal, il est égal à C_n pour des valeurs de n aussi grandes que l'on veut.

Nos fonctions en rouge étaient vraiment très bien choisies.

1.1.6 1.6 Traçons ℓ et γ

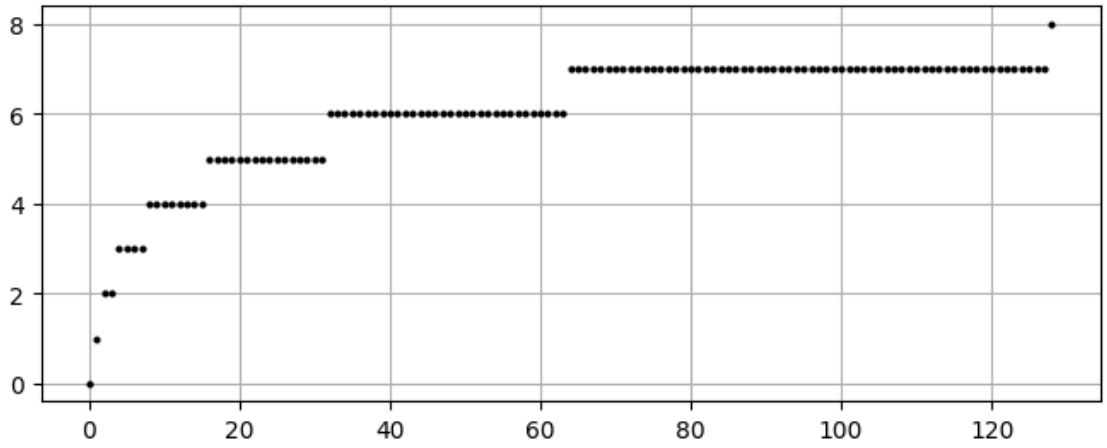
On commence par ℓ.

[14]:
```
def ell(n):
    if n == 0: return 0
    else: return ell(n // 2) + 1
```

[15]:
```
[ell(n) for n in range(17)]
```

[15]:
```
[0, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5]
```

[16]:
```
s = [ell(n) for n in range(129)]
plt.plot(s, 'ok', ms=2)
plt.grid()
plt.show()
```



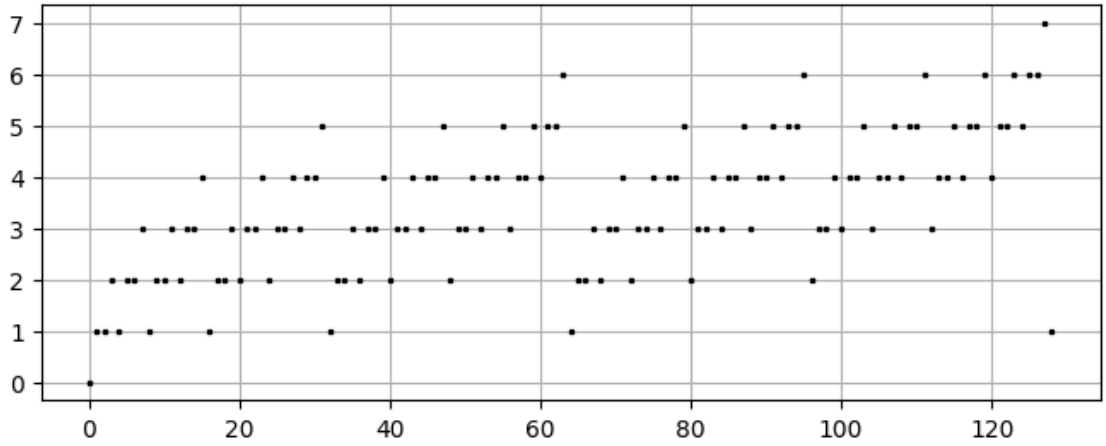
Passons à $\gamma(n)$.

[17]:
```
def gamma(n):
    if n == 0: return 0
    else:
        g = gamma(n // 2)
        if n % 2 == 0: return g
        else: return g + 1
```

[18]:
```
[gamma(n) for n in range(17)]
```

[18]:
```
[0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1]
```

[19]:
```
s = [gamma(n) for n in range(129)]
plt.plot(s, 'sk', ms=2)
plt.grid()
plt.show()
```



1.1.7 1.7 Une version itérative

Je n'ai rien contre les fonctions récursives, mais comme je l'ai déjà dit plus haut, Python est un tantinet réticent à effectuer un très grand nombre d'appels récursifs imbriqués. Par défaut, le nombre maximal d'appels récursifs est 256 (ce nombre est cependant modifiable).

Profitons du sectarisme de Python pour écrire une fonction non récursive qui effectue une exponentiation rapide. La voici.

[20]:
```
def puissance_iter(x, n):
    z, y, c, m = 1, x, 0, n
    while m > 0:
        if m % 2 == 1:
            z = z * y
            c = c + 1
        y = y * y
        c = c + 1
        m = m // 2
    return (z, c)
```

[21]:
```
puissance_iter(2, 3000)
```

[21]:
```
(12302319221611171769315588132767525146407138957368337157661180291600588006146729487753600678385934595824296492540518049085128841808982368235850824820653483312349593503558450174130233201113606669226247282397568804164344783156936750134130907572086903767932966588106629418244934884517265053037129160053467479086237026734800
```

```
91935393681310573662040235274477690384047788365110032240930198348836380293054048
24879097634840982539407286851320444088637347542712125924717786439494866885117210
51561970432780747454823776808464180697103083861812184348565522740195796682622205
51184551208055201031005025580158934964592800113374547422071501368341390754277906
37598338761013542351842450966700421607206294115815023712480084304471848420986103
20580417992206662247328722122088513643683907670360209162653670641130936997002170
50067550137472399876600582757930072325347489061225013517188917489907991129151239
97738721785190182299893 76,
 19)
```

Bon, ça marche pour 2^{3000} et le nombre de multiplications est identique à celui de la fonction récursive. Mais voilà pourquoi les théoriciens adorent les fonctions récursives ! Parce que je ne sais pas ce qu'il en est pour vous mais j'ai beau user mes yeux sur cette fonction, je ne comprends pas ce qu'elle fait !!!

Nous allons donc MONTRER que `puissance_iter(x, n)` renvoie effectivement x^n. Pour cela, nous allons mettre en évidence un *invariant de boucle*. C'est quoi ça ? Eh bien c'est "quelque chose qui ne varie pas au cours de la boucle". Plus précisément, un invariant de boucle est un objet mathématique dont la valeur avant une itération est la même que sa valeur après cette itération.

Cet objet aura donc la même valeur après la dernière itération que celle qu'il avait avant la première … à condition, évidemment que la boucle termine :-).

Quel genre d'objet peut-on considérer ? Par exemple une propriété, dont la valeur est un booléen. Ou un nombre, dont la valeur est … un nombre. Ou tout ce que l'on veut.

Ici, nous allons montrer que l'entier zy^m est un invariant de boucle. Soient m', z', y' les valeurs de m, z, y avant une certaine itération, et m'', z'', y'' les valeurs de ces mêmes variables après cette itération. Deux cas sont à considérer :

- Cas 1 : $m' = 2p$ où $p > 0$. Le test échoue, on a $z'' = z'$, $y'' = y'^2$ et $m'' = p$. Donc,

$$z''y''^{m''} = z'(y'^2)^p = z'y'^{2p} = z'y'^{m'}$$

- Cas 2 : $m' = 2p + 1$ où $p \geq 0$. Le test réussit, on a $z'' = z'y'$, $y'' = y'^2$ et $m'' = p$. Donc,

$$z''y''^{m''} = z'y'(y'^2)^p = z'y'^{2p+1} = z'y'^{m'}$$

Conséquence : la valeur de zy^m AVANT la toute première itération est la même que celle APRÈS la toute dernière itération.

- Valeur avant première itération : $1x^n = x^n$.
- Valeur après dernière itération : $zy^0 = z$.

Ainsi, la valeur de z après la dernière itération est x^n. Et cela tombe bien puisque notre fonction renvoie justement z.

Remarque. Quand je parle de la dernière itération, je suis très optimiste : la fonction termine-t-elle ? Mais oui, rassurons-nous. La boucle `while` est exécutée tant que $m > 0$, c'est à dire $\ell(m) > 0$. Avant la première itération, $\ell(m) = \ell(n)$. À chaque itération, m est divisé par 2, donc $\ell(m)$ est diminué de 1. Ainsi, la valeur de $\ell(m)$ après k itérations est $\ell(n) - k$. Conclusion :

Proposition. Lors de l'appel à `puissance_iter(x, n)`, la boucle `while` effectue $\ell(n)$ itérations.

Exercice. Montrer que lors de la $k$ième itération, la fonction `puissance_iter` effectue 2 multiplications si le $k$ième chiffre de l'écriture binaire de n est 1, et 1 multiplication si ce chiffre est 0.

On en déduit facilement :

Proposition. Le nombre de multiplications effectuées par la fonction itérative est le même que celui de la fonction récursive.

1.1.8 1.8 Généralisons

Souvenez vous de l'introduction de ce notebook où je parlais de puissances dans un monoïde. Était-ce juste pour faire savant ? Pas du tout. Il est très facile d'écrire une fonction générale d'exponentiation dans un monoïde quelconque. Il suffit de rajouter quelques paramètres.

Un monoïde est caractérisé par

- Son opération
- Son élément neutre e

Passons donc ces deux objets en paramètres ! Voici la version itérative.

[22]:
```python
def puissance(x, n, mul, e):
    z, y, m = e, x, n
    while m > 0:
        if m % 2 == 1: z = mul(z, y)
        y = mul(y, y)
        m = m // 2
    return z
```

[23]:
```python
puissance(7, 10 ** 1000, lambda x, y: x + y, 0)
```

[23]:
```
70000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000
```

Eh oui, car un multiple n'est jamais qu'une puissance additive.

Un autre exemple ? L'élévation à une puissance dans l'anneau des entiers modulo p, $\mathbb{Z}/p\mathbb{Z}$.

[24]:
```
def puissance_mod(x, n, p):
    return puissance(x, n, lambda x, y: (x * y) % p, 1)
```

[25]:
```
p = 2 ** 1279 - 1
print(p)
puissance_mod(3, (p - 1) // 2, p)
```

```
10407932194664399081925240327364085538615262247266704805319112350403608059673360
29801223944173232418484242161395428100779138356624832346490813990660567732076292
41295093892203457731833496615835504729594205476898112116936771475484788669625013
84438260291732348885311160828538416585028255604666224831890918801847068222203140
52102669843548873295802887805086973618690071472071055570316872908 7
```

[25]:
```
10407932194664399081925240327364085538615262247266704805319112350403608059673360
29801223944173232418484242161395428100779138356624832346490813990660567732076292
41295093892203457731833496615835504729594205476898112116936771475484788669625013
84438260291732348885311160828538416585028255604666224831890918801847068222203140
52102669843548873295802887805086973618690071472071055570316872908 6
```

Ce calcul nous montre que le nombre 3 n'est pas un résidu quadratique modulo le nombre de Mersenne $M_{1279} = 2^{1279} - 1$. Voir le notebook à ce sujet …

Un dernier exemple, pour montrer l'extrême généralité de tout cela. L'élévation à la puissance n dans le monoïde $\mathbb{R}^{\mathbb{R}}$ des fonctions de $\mathbb{R}$ vers $\mathbb{R}$ muni de la composition des applications. Et le neutre ? L'identité, bien sûr.

[26]:
```
def puissance_fonc(f, n):
    g = puissance(f, n, lambda f, g:lambda x:f(g(x)), lambda x:x)
    return g
```

Élevons la fonction $f : x \mapsto \sqrt{1+x}$ à la puissance 1000000. C'est à dire $f \circ f \ldots \circ f$, un million de fois.

[27]:
```
fmuch = puissance_fonc(lambda x:sqrt(1 + x), 1000000)
```

[28]:
```
fmuch
```

[28]:
```
<function __main__.puissance_fonc.<locals>.<lambda>.<locals>.<lambda>(x)>
```

Question à 1.000.000 euros : Remplacez 1000000 ci-dessus par 10^{1000}, et appuyez sur Entrée. Le calcul termine en zéro seconde. Exponentiation rapide, certes, mais là vous allez devoir expliquer. Parce que pour faire un parallèle avec les nombres, le nombre de chiffres de $2^{10^{1000}}$ est très très supérieur au nombre d'atomes de l'univers. Que calcule *exactement* Python lorsqu'on appuie sur Entrée ?

Calculons $f^{1000000}(0.3)$. Surtout n'oubliez pas de remettre 1000000 dans la cellule du dessus.

[29]:
```
fmuch(0.3)
```

[29]:
```
1.618033988749895
```

Tiens, le nombre d'or ? Ne le réveillons pas.

1.2 2. Une illustration : les nombres de Fibonacci

À propos du nombre d'or … Il existe de nombreux algorithmes pour calculer les nombres de Fibonacci. Le sujet de ce notebook étant l'exponentiation rapide, nous allons voir comment des calculs de puissances dans un anneau $\mathbb{A}$ judicieusement choisi permettent d'obtenir très efficacement la valeur de ces nombres.

1.2.1 2.1 Un anneau en or

Notons Φ la racine positive de l'équation $x^2 = x + 1$. On a $\Phi = \frac{1+\sqrt{5}}{2}$. Cela implique que, tout comme $\sqrt{5}$, Φ est irrationnel. Et surtout, que $\Phi^2 = \Phi + 1$.

Soit $\mathbb{A} = \{a+b\Phi, a, b \in \mathbb{Z}\}$. On montre facilement que $\mathbb{A}$ est un sous-anneau de $\mathbb{R}$. Par l'irrationalité de Φ, tout élement z de $\mathbb{A}$ s'écrit **de façon unique** sous la forme $z = a + b\Phi$, avec $a, b \in \mathbb{Z}$.

Soient $z = a + b\Phi$ et $z' = c + d\Phi$ deux éléments de notre anneau. On a

$zz' = (a + b\Phi)(c + d\Phi) = ac + (ad + bc)\Phi + bd\Phi^2$.

Mais $\Phi^2 = \Phi + 1$. En remplaçant, on obtient

$$(a + b\Phi)(c + d\Phi) = (ac + bd) + (ad + bc + bd)\Phi$$

Que faire de tout cela en Python ? Modélisons les éléments de l'anneau $\mathbb{A}$ par des couples d'entiers. Plus précisément, le couple (a, b) représente l'élément $a + b\Phi$. Par exemple, le neutre de l'anneau pour la multiplication, qui est bien entendu le nombre 1, est modélisé par le couple $(1, 0)$. Et le nombre d'or $\Phi = 0 + 1\Phi$ est modélisé par le couple $(0, 1)$.

La multiplication de deux éléments de notre anneau est immédiate à coder :

[30]:
```
def mul(x, y):
    a, b = x
    c, d = y
    return (a * c + b * d, a * d + b * c + b * d)
```

De là, l'élévation de $z \in \mathbb{A}$ à la puissance n est également immédiate, grâce à notre fonction générale d'exponentiation rapide.

[31]:
```
def puissphi(z, n):
    return puissance(z, n, mul, (1, 0))
```

[32]:
```
puissphi((0,1), 10)
```

[32]:
```
(34, 55)
```

1.2.2 2.2 Quel rapport avec les nombres de Fibonacci ?

Le calcul précédent nous montre que $\Phi^{10} = 34 + 55\Phi$. Évidemment ce 34 et ce 55 nous rappellent quelque chose.

Définition : la suite de Fibonacci $(F_n)_{n\geq 0}$ est définie par

- $F_0 = 0$
- $F_1 = 1$
- $\forall n \in \mathbb{N}^*, F_{n+1} = F_{n-1} + F_n$

Les premières valeurs de la suite sont 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 …

Proposition. pour tout $n \in \mathbb{N}^*$, on a $\Phi^n = F_{n-1} + F_n\Phi$.

Démonstration. Faisons une récurrence simple sur n.

Pour $n = 1$ c'est immédiat.

Soit donc $n \geq 1$, supposons $\Phi^n = F_{n-1} + F_n\Phi$.

On a alors $\Phi^{n+1} = \Phi^n\Phi = (F_{n-1} + F_n\Phi)\Phi = F_{n-1}\Phi + F_n\Phi^2$.

Mais $\Phi^2 = 1 + \Phi$. On remplace et on obtient $\Phi^{n+1} = F_n + (F_{n-1} + F_n)\Phi = F_n + F_{n+1}\Phi$.

Nous voici donc en possession d'un algorithme pour calculer le $n$ième nombre de Fibonacci avec un nombre *logarithmique* d'opérations : élever Φ à la puissance n et récupérer la partie Φ-maginaire, si j'ose dire. La fonction `fibonacci` fait le travail.

[33]:
```
def fibonacci(n): return puissphi((0, 1), n)[1]
```

[34]:
```
for k in range(20):
    print('%5d' % k, end='')
print()
for k in range(20):
    print('%5d' % fibonacci(k), end='')
```

```
    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
16   17   18   19
    0    1    1    2    3    5    8   13   21   34   55   89  144  233  377  610
987 1597 2584 4181
```

1.2.3 2.3 Le moment de vérité

Combien de temps pour calculer le millionième nombre de Fibonacci ? Évaluez la cellule ci-dessous.

[35]:
```
f = fibonacci(10 ** 6)
```

Pari gagné. Pour information, ce nombre possède 208988 chiffres.

[36]:
```
import sys
sys.set_int_max_str_digits(300000)
```

[37]:
```
print(len(str(f)))
```

```
208988
```