Factorisation_Entiers

February 17, 2019

1 Factorisation des entiers

Marc Lorenzi
11 juillet 2016

```
In [1]: import random
        import matplotlib.pyplot as plt
```

```
In [2]: plt.rcParams['figure.figsize'] = (10, 6)
```

1.1 1 L'algorithme naïf

1.1.1 1.1 Trouver un diviseur d'un entier

On recherche le plus petit diviseur de n (il sera forcément premier) en essayant tous les entiers à partir de 2. On peut tout de même remarquer qu'un entier composé n a un diviseur inférieur à $\sqrt{n}$. En effet, si a divise n alors $\frac{n}{a}$ aussi et l'un des deux entiers a et $\frac{n}{a}$ est inférieur à $\sqrt{n}$ puisque leur produit vaut n.

```
In [3]: def plus_petit_diviseur(n):
            k = 2
            while k * k <= n and n % k != 0:
                k = k + 1
            if k * k > n:
                return n
            else:
                return k
```

Cette fonction a clairement une complexité en pire cas en $\mathcal{O}(\sqrt{n})$, atteinte lorsque n est premier. Si l'on accepte de faire, disons, un million d'opérations, on pourra trouver des diviseurs de nombres ayant une douzaine de chiffres.

1.1.2 1.2 Factoriser un entier

Pour factoriser l'entier n, on recherche son plus petit diviseur premier p. Ensuite, on factorise (récursivement) $\frac{n}{p}$ et on ajuste ...

La fonction **factoriser** prend un entier non nul n en paramètre. Elle renvoie la liste $[(p_1, k_1), \ldots, (p_s, k_s)]$ où les p_i sont des nombres premiers distincts et $n = \prod_{i=1}^{s} p_i^{k_i}$.

```
In [4]: def factoriser(n):
            if n == 1:
                return []
            else:
                p = plus_petit_diviseur(n)
                s = factoriser(n // p)
                return ajuster_valuation(p, s)

In [5]: def ajuster_valuation(p, s):
            if s == []:
                return [(p, 1)]
            else:
                (q, k) = s[0]
                if p == q:
                    return [(p, k + 1)] + s[1:]
                else:
                    return [(p, 1)] + s

In [6]: factoriser(123456789)

Out[6]: [(3, 2), (3607, 1), (3803, 1)]
```

Cet algorithme de factorisation permet de factoriser des entiers ayant une quinzaine de chiffres, mais guère plus. Il faut savoir qu'il existe des algorithmes très sophistiqués permettant de factoriser des nombres d'une cinquantaine de chiffres. Dans ce qui suit nous allons nous intéresser à un algorithme relativement simple, l'algorithme ρ de Pollard, qui permet la factorisation d'entiers d'une trentaine de chiffres. Mais avant cela nous allons dire quelques mots sur les suites récurrentes à valeurs dans un ensemble fini.

1.2 2 Détection d'un cycle dans une suite récurrente

1.2.1 2.1 L'algorithme de Floyd

Soit $f : E \to E$ une fonction où E est un ensemble fini. Soit $a \in E$. La suite définie par $x_0 = a$ et $x_{n+1} = f(x_n)$ est ultimement périodique : il existe un entier n_0 et un entier $T > 0$ tels que pour tout $n \geq n_0$ on ait $x_{n+T} = x_n$. Plus précisément, les valeurs $a, f(a), f(f(a))$, etc. sont distinctes, puis on tombe à l'indice n_0 sur un cycle. Si l'on relie graphiquement les valeurs successives de la suite, on ne forme pas un rond (la suite ne boucle pas sur son premier terme) mais un ρ.

L'algorithme de Floyd permet de trouver une période ultime de f (la longueur de la boucle d'un ρ) à partir de la valeur a en espace constant et dans un temps de l'ordre de cette période. On initialise une variable x à la valeur $f(a)$ et une variable y à la valeur $f(f(a))$. Puis, tant que $x \neq y$, on remplace x par $f(x)$ et y par $f(f(y))$. Le point y se déplace deux fois plus vite que x et finira par "rattraper" x le long d'un cycle.

On montre facilement que le nombre d'itérations effectuées par l'algorithme est inférieur à la longueur du ρ.

La fonction **floyd** ci-dessous prend en paramètres la fonction f et un élément a de l'ensemble de départ (et d'arrivée !) de f. Elle renvoie un triplet (x, p, T) où x est un point cyclique pour f (le point où la boucle du ρ se rattache à sa jambe), p est l'indice de x dans la suite, et T est une période ultime de f. Néanmoins, l'entier T n'est pas nécessairement la plus petite période possible.

```
In [7]: def floyd(f, a):
            x = f(a)
            p = 1
            y = f(f(a))
            q = 2
            while x != y:
                x = f(x)
                p = p + 1
                y = f(f(y))
                q = q + 2
            return (x, p, q - p)
```

Juste pour savoir, peut-on obtenir la plus petite période ? Oui, c'est facile maintenant que l'on dispose d'un point cyclique ! La fonction **floyd_opt** renvoie LA période du point cyclique x en un temps au pire double du temps d'exécution de **floyd**.

```
In [8]: def floyd_opt(f, a):
            x, p, T = floyd(f, a)
            y = f(x)
            t = 1
            while y != x:
                y = f(y)
                t = t + 1
            return (x, p, t)
```

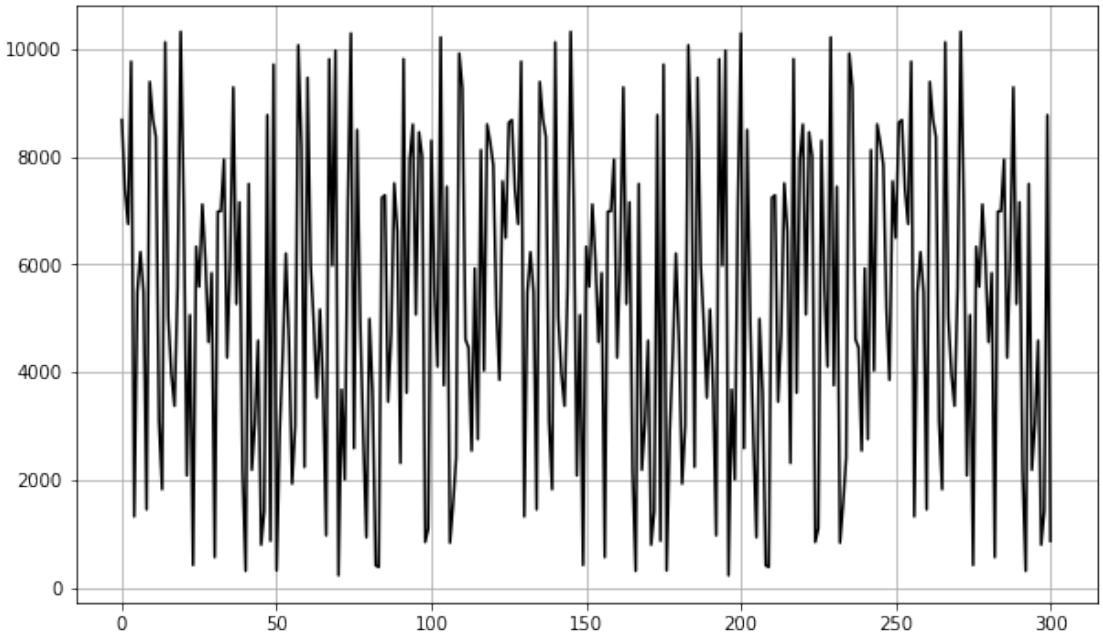
1.2.2 2.2 Un exemple

Prenons $E = \mathbb{Z}/10403\mathbb{Z}$. Prenons $f : E \to E$ définie par $f(x) = x^2 + 1$.

```
In [9]: def f(x):
            return (x * x + 1) % (10403)
```

La fonction ci-dessous renvoie la liste des n premières valeurs après l'indice n_0 de la suite définie par f et son premier terme x_0.

```
In [10]: def iterer (f, n0, n, x0):
             for k in range(n0):
                 x0 = f(x0)
             s = [x0]
             for k in range(n):
                 s.append(f(s[-1]))
             return s

In [11]: s = iterer(f, 9288, 300, 6)
         plt.grid()
         plt.plot(s, 'k')

Out[11]: [<matplotlib.lines.Line2D at 0x1125397f0>]
```



Pas facile de voir quoi que ce soit. Lançons l'algorithme de Floyd.

```
In [12]: floyd(f, 6)
```

```
Out[12]: (9288, 126, 126)
```

1.3 3 L'algorithme ρ de Pollard

1.3.1 3.1 L'implémentation

Soit c un entier quelconque. Pour tout entier non nul n, soit $f_n : x \mapsto x^2 + c$, définie sur $\mathbb{Z}/n\mathbb{Z}$. Appelons t_n le période ultime de f_n obtenue à partir d'une valeur initiale a.

Soit N un entier. Supposons que N est composé. Soit p un facteur premier de N. Si l'on applique l'algorithme de détection de cycle à f_p à partir d'un point a, on obtient un point cyclique pour f_p de période t_p. Il y a de grandes chances pour que $t_p \neq t_q$, ceci pour tous les autres facteurs premiers q de N. Par exemple, $t_p < t_q$. Soit a le point cyclique trouvé pour f_p. Soit $b = f_p^{t_p}(a)$. On a alors $a \equiv b[p]$ mais $a \not\equiv b[q]$. Et donc le pgcd de $b - a$ et de N est égal à p.

En fait, on modifie légérement l'algorithme de Floyd pour trouver ce qui nous intéresse, à savoir non pas une période, mais un pgcd différent de 1.

```
In [13]: def gcd(a, b):
             while b != 0:
                 a, b = b, a % b
             return a
```

```
In [14]: def f(x, N, c):
             return (x * x + c) % N
```

```
In [15]: def pollard(N):
             x = random.randint(0, N - 1)
             c = random.randint(0, N - 1)
             y = f (x, N, c)
             count = 0
             while gcd(y - x, N) == 1:
                 x = f(x, N, c)
                 y = f(f(y, N, c), N, c)
                 count += 1
             print("facteur trouvé en %d itérations" % count)
             return gcd(y - x, N)
```

1.3.2 3.2 Tests

```
In [16]: N = 1234567907 * 10987654367
         N

Out[16]: 13565005454706599869

In [17]: pollard(N)

facteur trouvé en 13338 itérations


Out[17]: 10987654367
```

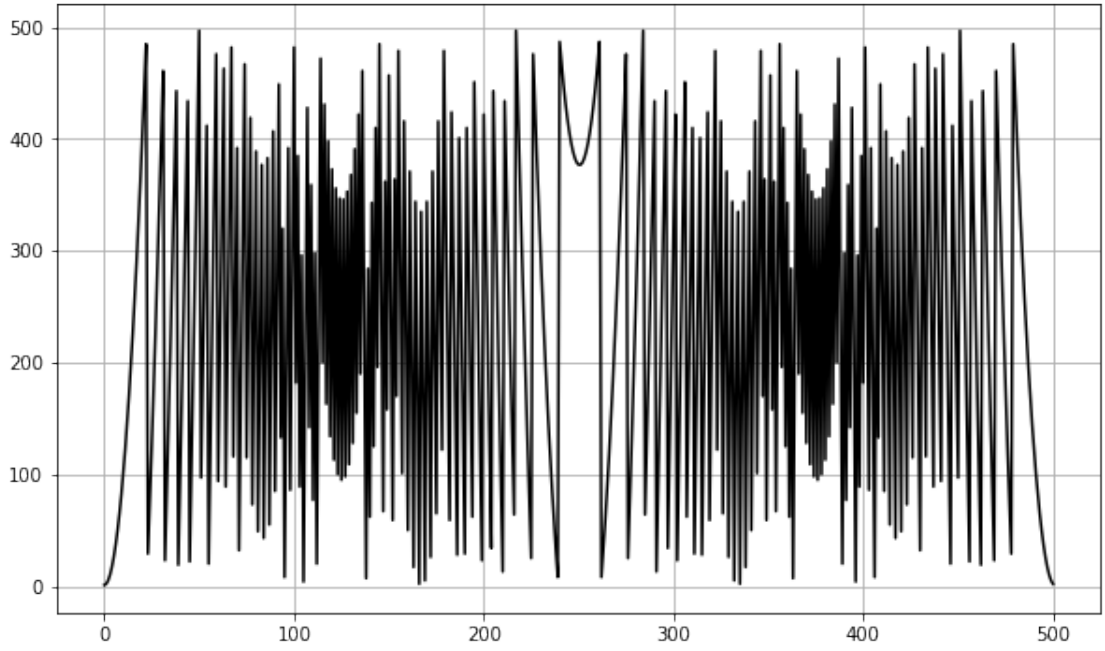
Encore un essai, avec un nombre d'une trentaine de chiffres sans petits diviseurs.

```
In [18]: N = 2 ** 101 - 1
         N

Out[18]: 2535301200456458802993406410751

In [19]: pollard(N)

facteur trouvé en 8131758 itérations


Out[19]: 7432339208719
```

1.3.3 3.3 Complexité de l'algorithme

Voici le graphe de la fonction $x \mapsto (x^2+1) \bmod 501$. Mise à part la symétrie évidente $f(501-x) = f(x)$ due au carré, le graphe apparaît très irrégulier.

```
In [20]: s = [(x ** 2 + 1) % 501 for x in range(501)]
         plt.plot(s, 'k')
         plt.grid()
         plt.show()
```



Nous n'allons pas rigoureusement prouver la complexité de l'algorithme de Pollard, mais juste donner quelques idées sur la question.

Soit N un entier composé. Soit p le plus petit facteur premier de N. En admettant que la fonction $x \mapsto x^2 + c$ se comporte comme une fonction "aléatoire", on peut voir les termes de la suite (x_n) de l'algorithme de Pollard comme des boules tirées au hasard avec remise dans une urne contenant p boules. La question est de savoir au bout de combien de temps on aura tiré deux fois la même boule.

Proposition : On considère une urne contenant p boules. On effectue des tirages avec remise dans l'urne. L'espérance du nombre de tirages avant que deux boules identiques aient été tirées est $\mathcal{O}(\sqrt{p})$.

Preuve : Soit s le nombre de tirages avant qu'une "collision" ne survienne. Pour $p \geq j \geq 2$, on a

$$P(s \geq j) = \frac{1}{p^{j-1}} \prod_{1 \leq i < j} (p - (i-1)) = \prod_{1 \leq i < j} (1 - \frac{i-1}{p})$$

C'est en effet la probabilité que les $j-1$ premières boules tirées soirent dictinctes : p possibilités pour le tirage de la première boule, puis $p-1$ pour la deuxième boule, ainsi de suite. Puis on divise par p^{j-1}, le nombre de possibilités. Bien évidemment, si $j > p$, $P(s \geq j) = 0$.

Utilisons l'inégalité $1 - x \leq e^{-x}$, vraie pour tout réel x. Il vient

$$P(s \geq j) \leq \prod_{1 \leq i < j} e^{-\frac{i-1}{p}} = e^{-(j-1)(j-2)/(2p)} \leq e^{-(j-2)^2/(2p)}$$

L'espérance de s vérifie donc

$$E(s) = \sum_{j \geq 1} P(s \geq j) \leq 1 + \sum_{j \geq 0} e^{-j^2/2p} \leq 2 + \int_0^\infty e^{-x^2/2p} dx$$

la dernière inégalité provenant d'une comparaison série-intégrale. Le changement de variable $x = \sqrt{2p}t$ dans l'intégrale nous donne

$$E(s) \leq 2 + \sqrt{2p}\int_0^\infty e^{-t^2}dt = 2 + \sqrt{\frac{p\pi}{2}} = \mathcal{O}(\sqrt{p})$$

Ainsi, la complexité en moyenne de l'algorithme de Pollard est $\mathcal{O}(\sqrt{p})$ où p est le plus petit facteur premier de N. C'est à dire, au pire, en $\mathcal{O}(N^{1/4})$, mais beaucoup mieux si N a par chance un facteur premier pas trop grand.

Cette complexité est à comparer avec le $\mathcal{O}(N^{1/2})$ de l'algorithme naïf. Concrètement, cela signifie que l'on pourra avec l'agorithme ρ de Pollard factoriser des entiers deux fois plus longs qu'avec l'algorithme naïf. Si l'on accepte, mettons, un million d'opérations, l'algorithme naïf factorise des entiers d'une douzaine de chiffres et l'algorithme de Pollard factorise des entiers d'environ 25 chiffres. Évidemment, Avec un peu de chance on peut faire mieux. Tentons par exemple de trouver un facteur du 10ème nombre de Fermat, $F_{10} = 2^{2^{10}} + 1$.

```
In [21]: def fermat(n):
             return 2 ** (2 ** n) + 1

In [22]: fermat(10)

Out[22]: 179769313486231590772930519078902473361797697894230657273430081157732675805500963132708

In [23]: pollard(fermat(10))

facteur trouvé en 5350 itérations


Out[23]: 6487031809
```

Voici le mieux que l'on puisse faire en quelques secondes. L'entier N ci-dessous est un nombre de 80 bits, produit de deux nombres premiers de 40 bits chacun.

```
In [24]: N = 740514396871 * 1069728598117
         N

Out[24]: 792149427650270601291907

In [25]: pollard(N)

facteur trouvé en 489797 itérations


Out[25]: 1069728598117

In [ ]:
```