Chapitre 2

Exponentiation rapide

Les entiers sont de taille arbitraire en Python, ce qui permet de calculer x^n avec un grand entier n et un élément x d'un groupe multiplicatif, sans risque de débordement de capacité (la multi-précision est gérée automatiquement). Les applications concernent tout particulièrement le calcul de $x^n \mod m$ (exponentiation modulaire très utilisée en cryptographie), et le calcul de A^n où A est une matrice carrée. On peut ainsi calculer avec un nombre d'additions et de multiplications logarithmique [1] en n le terme général u_n des suites récurrentes linéaires d'ordre 2, comme la suite de Fibonacci :

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \tag{2.1}$$

On parle d'algorithme d'**exponentiation**, plutôt que de calcul de puissance, car l'on s'intéresse à la complexité du calcul en fonction de la taille de l'exposant et $n \mapsto x^n$ est une fonction exponentielle de n. La représentation binaire de n permet de réaliser le calcul efficacement, selon un « **algorithme d'exponentiation rapide** ».

La méthode d'exponentiation rapide (en anglais *binary ladder exponentiation*, *exponentiation by squaring*, *fast exponentiation*) est très utilisée dans les études sur la factorisation, la cryptographie ou l'étude des courbes elliptiques. En cryptographie la génération de clés publiques à partir de clés privées passe par l'exponentation, de même que la génération de signatures digitales. Cet algorithme joue souvent un rôle critique dans les performances globales de ces applications.
Les algorithmes sont programmés pour $x \in \mathbb{R}^*$ et $n \in \mathbb{Z}$; les puissances de zéro sont connues.

2.1 Algorithme naïf de calcul de puissance

La méthode naïve pour calculer x^n où x est un élément d'un groupe $(G, \times)$ et n un entier positif consiste à réaliser $n-1$ multiplications.

```
def puissance_naive(x, n):
    r = 1
    for i in range(n):
        r *= x
    return r
```

Cet algorithme est trivialement de **complexité exponentielle en la taille de n.** La version récursive est immédiate puisque $x^n = xx^{n-1}$:

```
def puiss(x, n = 0):
    return 1 if n == 0 else x * puiss(x, n - 1)
```

1. On peut difficilement ne pas tenir compte de l'augmentation de la taille des F_n avec n. Si l'on désigne par $M(n)$ la complexité de la multiplication de deux nombres sur n bits, la complexité temporelle globale est en $\Theta(M(n) \ln n)$.

ou avec une **fonction locale récursive auxiliaire** pour éviter de retransmettre le même x :

```
def puiss(x, n):
    aux = lambda k: 1 if k == 0 else x * aux(k - 1)
    return aux(n)
```

La complexité temporelle linéaire en n est confirmée par la récurrence vérifiée par le nombre d'opérations élémentaires : $T(n) = T(n-1) + \Theta(1)$ qui fournit $T(n) = \Theta(n)$.

2.2 Exponentiation binaire de droite à gauche

→ *binary ladder exponentiation, right-left form*

La représentation binaire de n :

$$n = \sum_{i=0}^{m} b_i \, 2^i \qquad \text{avec} \quad b_i \in \{0, 1\} \tag{2.2}$$

permet de calculer efficacement x^n :

$$x^n = \prod_{i=0}^{m} x^{b_i \, 2^i} = \prod_{\substack{i=0 \\ b_i=1}}^{m} x^{2^i} \tag{2.3}$$

Le principe est d'utiliser la représentation binaire de n, des bits de poids faibles vers les bits de poids forts (lecture de droite à gauche). Pour chaque bit b_i égal à 1, on multiplie par $z_i = x^{2^i}$, sinon l'on ne fait rien.
Il faut calculer x^n en utilisant la variable $z_i = x^{2^i}$ afin de calculer

$$z_{i+1} = x^{2^{i+1}} = \left(x^{2^i}\right)^2 = {z_i}^2 \tag{2.4}$$

```
1 def puissance_droite_gauche(x, n):
2     y = 1
3     while n > 0:
4         if n % 2 == 1:
5             y *= x
6         n //= 2
7         x *= x
8     return y
```

La ligne 4 peut être remplacée par `if n & 1` : tester la parité revient à tester le bit de poids faible.
La ligne 6 peut être remplacée par `n >>= 1` : diviser par 2 revient à décaler à droite d'une position la représentation binaire.

```
>>> puissance_droite_gauche(13, 125)
174952051072312187935769636348278264834502031341317604340490690745763820374569936848223268930305050663755959245336646760704304564689113816 93
```

Terminaison
n est un variant de boucle : il décroit strictement, reste positif et finit par s'annuler comme quotient de division dans $\mathbb{N}$

Correction

La correction a été démontrée précédemment par les expressions (3.3) et (3.4) mais nous indiquons un invariant de boucle pour illustrer la méthode.
Un invariant de boucle est $z = yx^n$. En effet, si n est pair :

$$n' = \frac{n}{2} \quad ; \quad x' = x^2 \quad \text{donc} \quad z' = y'(x')^{n'} = y\left(x^2\right)^{n/2} = yx^n = z$$

Si n est impair

$$n' = \frac{n-1}{2} \quad ; \quad x' = x^2 \quad ; \quad y' = yx \quad \text{donc} \quad z' = y'(x')^{n'} = yx\left(x^2\right)^{(n-1)/2} = yx^n = z$$

Or à l'entrée de la boucle, $y = 1$, donc z reste égal à la valeur x^n cherchée. En sortie $n_s = 0$, donc $z = y_s {x_s}^0 = y_s$. Donc la valeur renvoyée $y_s = z = x^n$ est correcte.

Complexité

Les calculs sont parfaitement décrits par l'expression anglo-saxonne « *square-and-multiply algorithm* ». Soit $\beta = \lfloor \log_2 n \rfloor + 1$ le nombre de bits de n. L'algorithme réalise β fois l'élévation au carré de x et la multiplication de y par x est réalisée autant de fois qu'il y a de 1 dans la représentation binaire de n. Le nombre de multiplications est donc au plus

$$M(n) = 2\beta = \Theta(\ln n) \tag{2.5}$$

qu'il faut comparer au nombre de multiplications $M(n) = \Theta(2^\beta) = \Theta(n)$ de l'algorithme naïf. Nous passons d'un algorithme de complexité temporelle exponentielle à un **algorithme de complexité linéaire** en la taille de n. La complexité dans le cas moyen est $M(n) = 1,5\,\beta$.

Il s'agit de la complexité relative à la taille de l'exposant n, mais si la taille de x augmente, il faut tenir compte de la complexité asymptotique de la multiplication. Pour l'exponentiation modulaire, dans le groupe $((\mathbb{Z}/m\mathbb{Z})^*, \times)$ des classes inversibles modulo m, la complexité de cet algorithme est en $\Theta(\ln n \, \ln^2 m)$ avec une multiplication modulaire naïve. La taille de x et l'algorithme de multiplication peuvent devenir les éléments déterminants de la complexité du calcul de x^n. Si les tailles de x et n sont du même ordre, l'algorithme a une complexité temporelle en $O(\ln^3 n)$, cubique en la taille des éléments manipulés.

Remarque 1. *Le programme réalise une multiplication de trop (car la première consiste à multiplier par 1) et une élévation au carré de trop (la dernière élévation au carré de x n'est jamais utilisée). Un bon exercice consiste à modifier le code pour ne réaliser que les opérations strictement nécessaires.*

Comme x est élément d'un groupe G, il est possible de travailler avec un exposant dans $\mathbb{Z}$, grâce à l'inverse de x dans G. On obtient la version modifiée suivante.

```
def puissance_droite_gauche(x, n):
    if n < 0:
        n = -n
        x = 1 / x
    y = 1
    while n > 0:
        if n % 2 == 1:
            y *= x
        n //= 2
        x *= x
    return y
```

2.3 Exponentiation binaire de gauche à droite

→ *binary ladder exponentiation, left-right form*

La décomposition binaire (3.2) de n peut être utilisée pour remarquer que $n = P(2)$ où P est le polynôme

$$P = \sum_{k=0}^{p} b_k \, X^k \tag{2.6}$$

Ainsi $x^n = x^{P(2)}$ peut être évalué par un schéma de Hörner :

$$x^n = x^{(\cdots(b_p \times 2 + b_{p-1})2 + b_{p-2})2 + \cdots + b_1)2 + b_0} \tag{2.7}$$

qui devient, avec $b_p = 1$:

$$x^n = (\cdots(x^2 \times x^{b_{p-1}})^2 \times x^{b_{p-2}})^2 \times \cdots \times x^{b_1})^2 \times x^{b_0} \tag{2.8}$$

Le parcours des bits de n du poids fort vers le poids faible conduit à un algorithme légèrement différent, qui nécessite en particulier de connaître au départ l'indice p du bit de poids fort. Une succession de divisions euclidiennes par 2 permet de réaliser cette opération en $\Theta(\ln n)$ opérations élémentaires, ce qui ne pénalise pas la complexité globale. On trouve donc $e = \lfloor \log_2 n \rfloor$ qui permet d'écrire $2^e \leqslant n < 2^{e+1}$, avant d'entrer dans la boucle principale.

```
def puissance_gauche_droite(x, n):
    if n == 0: return 1
    if n < 0:
        n, x = -n, 1 / x
    y, e, t = 1, 0, 0
    while n > 0:
        n, r = divmod(n, 2)
        e = e * 2 + r
        t += 1
    for _ in range(t):
        y *= y
        if e % 2 == 1: y *= x
        e //= 2
    return y
```

En sortie de boucle `while`, la variable e contient un entier dont la représentation binaire est le miroir de la représentation binaire de n. La boucle `for` récupère ainsi les bits de n du poids fort vers le poids faible. Un invariant de boucle `while` est : « la représentation binaire de n accolée à la représentation binaire miroir sur t bits de e est constante, égale à la valeur initiale de n ». Un variant de la boucle `while` est n.
La correction de la boucle `for` repose précisément sur l'expression (3.8).

Pourquoi parcourir les bits de gauche à droite alors que la version de droite à gauche semble plus simple ?

- La multiplication réalisée lorsque le bit de n est égal à 1 utilise toujours la même valeur de x initiale. À l'inverse cette multiplication est réalisée avec un x dont la taille augmente très vite dans l'algorithme de droite à gauche. Si x tient sur un seul mot, la multiplication sera réalisée en $O(\ln m)$ au lieu de $O(\ln^2 m)$. Cet algorithme est plus rapide lorsque n est grand que la version de droite à gauche.
- Cet algorithme de gauche à droite peut être modifié en décomposant n dans une base égale à une puissance de 2 (*windowing ladders*). On travaille alors avec des groupes de k bits avec une décomposition de n en base 2^k. Une valeur de k adaptée aux mots manipulés permet de diminuer le nombre d'opérations à réaliser de gauche à droite.
- Cet algorithme correspond à l'algorithme récursif le plus naturel (paragraphe suivant).

2.4 Algorithme récursif

2.4.1 Première version (top-down)

Propriété 1. *Le calcul d'une puissance par la méthode récursive d'exponentiation rapide repose sur l'expression :*

$$x^n = \begin{cases} x \cdot \left(x^{(n-1)/2}\right)^2 & \text{si } n \text{ impair} \\ \left(x^{n/2}\right)^2 & \text{si } n \text{ pair} \end{cases} \tag{2.9}$$

avec le cas de base $x^0 = 1$.

Algorithme 2.4.1 Puissance rapide récursive (*recursive powering ladder*)

Entrées : un élément x d'un groupe $(G, \times)$ et un entier $n \in \mathbb{N}$
Sorties : x^n, la n-ième puissance de x

1. **si** $n = 0$ **alors**
2. **retourne** 1
3. **fin si**
4. **si** n impair **alors**
5. **retourne** $x \cdot \left(x^{(n-1)/2}\right)^2$
6. **fin si**
7. **retourne** $\left(x^{n/2}\right)^2$

Les puissances sont calculées par appels récursifs. Il suffit de remarquer que l'on multiplie toujours par le même x pour réaliser que cet **algorithme récursif est équivalent à la méthode itérative de gauche à droite.**

```
def puis_rec(x, n):
    if n == 0:
        return 1
    p = puis_rec(x, n // 2)
    if n % 2 == 1:
        return x * p * p
    return p * p
```

Cet algorithme est programmé en observant que pour n impair :

$$\frac{n-1}{2} = \left\lfloor \frac{n-1}{2} \right\rfloor = \left\lfloor \frac{n}{2} \right\rfloor \tag{2.10}$$

Noter l'appel récursif non terminal : tous les calculs sont réalisés lors de la remontée. Comment peut-on remplacer les trois dernières lignes par une instruction conditionnelle à l'intérieur d'un seul `return` ?

Terminaison
Un seul appel récursif avec un argument entier, positif, qui décroit strictement lors de chaque appel. Le cas de base finit toujours par être appelé.

Correction
La récurrence implémentée est exactement la propriété 1, qui se vérifie facilement pour tout $n \in \mathbb{N}$.

Complexité
Le nombre d'opérations arithmétiques vérifie la relation de récurrence

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \tag{2.11}$$

• On considère d'abord le cas particulier où $n = 2^p$ et l'on pose $u_p = T(2^p) = T(2^{p-1}) + \Theta(1) = u_{p-1} + \Theta(1)$. La récurrence se résout en $u_p = u_0 + p\Theta(1) = T(1) + p\Theta(1) = \Theta(p)$. Donc $T(n) = \Theta(\log_2 n)$.
• Pour n quelconque, on pose $p = \lfloor \log_2 n \rfloor$, d'où $p \leqslant \log_2 n < p+1$, donc comme la fonction $x \mapsto 2^x$ est strictement croissante sur $\mathbb{R}$, $2^p \leqslant n < 2^{p+1}$. Comme la fonction $T(n)$ est croissante, il vient $T(2^p) \leqslant T(n) < T(2^{p+1})$, donc $u_p \leqslant T(n) < u_{p+1} = u_p + \Theta(1)$. On en déduit donc que $T(n) = \Theta(p)$ et $T(n) = \Theta(\ln n)$.
La complexité temporelle est logarithmique en la taille de l'exposant n.
La complexité spatiale est liée à la taille de la pile d'exécution de la fonction récursive. Le nombre d'appels est en $\log_2 n$ pour un contexte de taille bornée, donc la complexité spatiale est en $\Theta(\ln n)$.

2.4.2 Seconde version (top-down)

Propriété 2. *Le calcul d'une puissance par la méthode récursive d'exponentiation rapide repose sur l'expression :*

$$x^n = \begin{cases} x \cdot (x^2)^{\frac{n-1}{2}} & \text{si } n \text{ impair} \\ (x^2)^{\frac{n}{2}} & \text{si } n \text{ pair} \end{cases} \tag{2.12}$$

```
def puis_rec(x, n):
    if n == 0:
        return 1
    p = puis_rec(x * x, n >> 1)
    if n & 1:
        return x * p
    return p
```

La **récursivité descendante** se caractérise en particulier par un cas de base qui retourne la valeur initiale $x^0 = 1$ de la récurrence.

2.4.3 Algorithme récursif terminal (bottom-up)

L'algorithme précédent nous suggère une programmation récursive terminale : il ne reste plus qu'une multiplication par x, réalisée dans le cas impair à la remontée. Une fonction locale permet de rendre cette récursivité terminale.

```
def puis_rec_term(x, n):
    def puis(y, x, n):
        if n == 0:
            return y
        if n & 1:
            return puis(y * x, x * x, n >> 1)
        return puis(y, x * x, n >> 1)
    return puis(1, x, n)
```

La **récursivité montante** se caractérise par un cas de base qui retourne la valeur finale x^n.
Il est facile de dérécursifier cet algorithme, et l'on obtient l'algorithme d'exponentiation binaire de droite à gauche déjà expliqué.

2.5 Optimisations

La question de l'exponentiation n'est pas définitivement réglée par les méthodes précédentes.

Propriété 3. *La méthode de décomposition binaire de l'exposant n'est pas optimale du point de vue du nombre de multiplications et l'on peut généralement faire mieux.*

Soit à calculer x^{15}. La méthode binaire utilise 3 multiplications et 3 élévations au carré car $15 = (1111)_2$ et $x^{15} = x \times (x \times (x \times x^2)^2)^2$. Mais on peut calculer $x^3 = x \times x^2$, puis $x^{15} = x^3 \times ((x^3)^2)^2$. On réalise alors 2 multiplications et 3 élévations au carré, ce qui est mieux que la méthode binaire.

Il existe de nombreuses méthodes d'optimisation permettant de réduire le nombre d'opérations élémentaires, selon que l'on utilise souvent le même n (*fixed-exponent methods*) ou souvent le même x (*fixed-x methods*). Le paragraphe suivant présente une amélioration célèbre.