COURS : Algorithme et Programmation Université Paris-Dauphine MIDO L2
denis.cornaz@lamsade.dauphine.fr

Introduction (validité, lisibilité)

Sans que cela saute aux yeux, le code suivant semble renvoyer le quotient de la division entière de $a \in \mathbb{N}$ par $b \in \mathbb{N} \setminus \{0\}$.

```
def q(a,b):
        q=0
        while a>=q*b:
                q=q+1
        return q-1
```

Il existe deux uniques entiers q, r tels que $a = qb + r$ avec $0 \leq r < b$. Puisque $a \geq 0$ on rentre nécessairement dans la boucle `while`, et on en sort avec

(i) $a < qb$

(ii) $a \geq (q-1)b$

En posant $r = a - (q-1)b$, la condition (i) est équivalente à $r < b$, la condition (ii) est équivalente à $r \geq 0$; et il en découle que le code est *valide*.

Pour plus de *lisibilité* on préfèrera le code suivant:

```
def q(a,b):
        q=0
        while a - q*b >= b:
                q=q+1
        return q
```

Si $a < b$ le code renvoie 0, ce qui est correct. Sinon, on sort de la boucle `while` avec

(i) $a - qb < b$

(ii) $a - (q-1)b \geq b$

Clairement la condition (i) est équivalente à $r = a - bq < b$, et la condition (ii) est équivalente à $r \geq 0$. La validité du code est presque immédiate.

1 Algorithmes et fonctions logarithmes

1.1 Exemples

La fonction récursive python suivante

```
y=0
def f(x):
        """ compte les feuilles """
        global y
        if x>0:
                f(x//2)
                f(x//2)
        else:
                y=y+1
```

attribue à y une nouvelle valeur $f(x) = 2^{\lfloor \log_2 x \rfloor + 1}$. On peut représenter le déroulement de cette fonction par un arbre binaire. Avec `x=8` en entrée, les premiers appels résursifs se déroulent ainsi

```
                                    f(8)
                  f(4)
        f(2)
    f(1)
f(0)
```

la fonction `f(0)` incrémente la variable globale `y` et se termine, c'est `f(1)` qui continue son déroulement avec un second appel à `f(0)` qui termine le premier appel à `f(1)` dans `f(2)` qui enchaine sur son second appel de `f(1)`

```
                                    f(8)
                  f(4)
        f(2)
    f(1)       f(1)
f(0)  f(0)
```

Le déroulement est semblable quand le premier appel à `f(4)` de `f(8)` se termine

```
                                    f(8)
                  f(4)
        f(2)                  f(2)
    f(1)       f(1)       f(1)       f(1)
f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)
```

Et à la fin on a le schéma suivant:

```
                                    f(8)
                  f(4)                                  f(4)
        f(2)                  f(2)                  f(2)                  f(2)
    f(1)       f(1)       f(1)       f(1)       f(1)       f(1)       f(1)       f(1)
f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)
```

Remarquons qu'on garde ce schéma tant que $x < 16$. Indiçons les lignes $0, 1, \ldots$ et comptons les appels sur cette ligne, par exemple à la ligne 4 il y a 16 appels, et notons

- $p(x)$ le nombre d'appels à la ligne x, et
- $l(x)$ la ligne correspondant à x appels.

```
                                    f(15)                                                   0
                  f(7)                                  f(7)                              1
        f(3)                  f(3)                  f(3)                  f(3)            2
    f(1)       f(1)       f(1)       f(1)       f(1)       f(1)       f(1)       f(1)     3
f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  f(0)  4
 1     2     3     4     5     6     7     8     9     10    11    12    13    14    15    16
```

On a par exemple $l(4) = 2$ et $p(3) = 8$. Ces fonctions ont des propriétés étranges

$$p(x+1) = p(x)p(1)$$

puisque à la ligne suivante j'ai deux fois plus d'appels et $2 = p(1)$, et

$$l(2x) = l(x) + l(2)$$

si je veux avoir deux fois plus d'appels, il faut aller une ligne plus bas et $1 = l(2)$.

Elles semblent transformer les $+$ en $\times$ et inversement. On montrera que

- toute fonction continue $p : \mathbb{Q} \to \mathbb{R}_*^+$ telle que $p(x+y) = p(x)p(y)$ est la fonction $p(x) = b^x$ avec $b = p(1)$;

- toute fonction continue $l : \mathbb{Q}_*^+ \to \mathbb{R}$, avec $l(b) = 1$, telle que $l(xy) = l(x) + l(y)$ est la fonction $l(x) = y$ telle que $b^y = x$.

plus d'autres propriétés très simples qui découlent uniquement de la définition des fonctions $l(x)$ qu'on appelle logarithme, et on note $\log_b x$ le réel y tel que $b^y = x$, et dans ce cours la base b du logarithme sera $b > 1$. Ainsi par exemple, $\log_b a \times \log_a x = \log_b x$, ou $a^{\log_b x} = x^{\log_b a}$.

Un autre exemple de fonction récursive:

```
y=0
def f(x):
        """ compte les sommets internes """
        global y
        if x>0:
                f(x//2)
                f(x//2)
                y=y+1
```

attribue à y une nouvelle valeur $f(x) = 2^{\lfloor \log_2 x \rfloor + 1} - 1$. Cette fonction, à l'inverse de l'autre qui incrémentait `y` uniquement pour une entrée `x=0`, incrémente `y` uniquement si `x` est non-nul. Pour dénombrer les sommets d'un arbre binaire, ou plus généralement de degré b pour un entier ≥ 2, on montrera que

$$b^{n+1} = 1 + \sum_{i=0}^{n} (b-1) b^i$$

Dans l'exemple

```
y=0
def f(x):
        global y
        if x>0:
                f(x//2)
                f(x//2)
                y=y+x
```

la nouvelle valeur de y est la somme des valeurs x à chaque appel récursif (donc dans tout l'arbre). Par exemple pour $x = 15$, en se reportant à l'arbre, on a $f(x) = 15 + (2 \times 7) + (4 \times 3) + (8 \times 1)$. Pour un x quelconque, il nous faut évaluer la valeur

$$\left\lfloor \frac{\left\lfloor \frac{x}{2} \right\rfloor}{2} \right\rfloor$$
$$\left\lfloor \frac{\vdots}{2} \right\rfloor$$

que l'on obtient par des divisions de x par 2 et des arrondis successifs. Pour voir que cette valeur est en fait égale à

$$\left\lfloor \frac{x}{2^i} \right\rfloor$$

où i est le nombre de divisions/arrondis successifs, il faut utiliser la décomposition en base 2, ou en base $b \geq 2$, si l'on veut généraliser à tout entier supérieur à 2. Soit $b \geq 2$ un entier, on montrera que pour tout entier x il existe une unique suite d'entiers $x_0, x_1, \ldots, x_{\lfloor \log_b x \rfloor}$ telle que

$$x = \sum_{i=0}^{\lfloor \log_b x \rfloor} x_i b^i$$

où $0 \leq x_i < b$ est un entier, pour tout $i = 0, \ldots, \lfloor \log_b x \rfloor$. Finalement, pour la fonction python `f(x)` on a

$$f(x) = \sum_{i=0}^{\lfloor \log_2 x \rfloor} 2^i \left\lfloor \frac{x}{2^i} \right\rfloor$$

Cette expression n'est pas du type $x, x\log x, x^2, \ldots, 2^x$, essayons donc de l'encadrer par des fonctions de ce type. On a

$$\begin{array}{rcccccccc}
x & = & x_0 & +2x_1 & +2^2x_2 & +2^3x_3 & +\ldots & +2^nx_n \\
2\left\lfloor \frac{x}{2} \right\rfloor & = & & +2x_1 & +2^2x_2 & +2^3x_3 & +\ldots & +2^nx_n \\
2^2\left\lfloor \frac{x}{2^2} \right\rfloor & = & & & +2^2x_2 & +2^3x_3 & +\ldots & +2^nx_n \\
2^3\left\lfloor \frac{x}{2^3} \right\rfloor & = & & & & +2^3x_3 & +\ldots & +2^nx_n \\
\vdots & & & & \vdots & & & \\
2^n\left\lfloor \frac{x}{2^n} \right\rfloor & = & & & & & & 2^nx_n
\end{array}$$

où $n = \lfloor \log_2 x \rfloor$ est le plus grand entier tel que x_n soit non nul dans la décomposition en base 2. Il en découle

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

puisqu'à chaque ligne $2^i\left\lfloor \frac{x}{2^i} \right\rfloor \geq 2^n \geq \frac{x}{2}$. On note alors $f(x) = \Theta(x\log x)$. D'une manière générale,

Definition 1 *On note*

$$f(n) = \Theta(g(n))$$

s'il existe deux constantes réelles $\alpha, \beta > 0$ telles que

$$\alpha\, g(n) \leq f(n) \leq \beta\, g(n)$$

pour tout n supérieur à une autre constante n_0.

et ici $\frac{1}{2}nx \leq f(x) \leq 2nx$ et $\frac{1}{2}\log_2 x \leq n \leq \log_2 x$, d'où $\frac{1}{4}x\log_2 x \leq f(x) \leq 2x\log_2 x$, et on a vu que $\log_b x = \Theta(\log_a x)$ pour tous entiers $a, b > 1$ (ainsi il est impropre de préciser la base du log dans un $\Theta(\cdot)$).

1.2 Outils d'analyse algorithmique

On classe les fonctions que l'on étudie en algorithmique en trois types principaux

- $f(x) = \Theta(x^p)$ ($p = 0$ est la classe constante, $p = 1$ est la classe linéaire)
- $f(x) = \Theta(x^p \log x)$ ($p = 0$ est la classe logarithmique, $p = 1$ est la classe pseudo-linéaire)
- $f(x) = \Theta(b^x)$ où $b > 1$ est un réel (c'est la classe exponentielle)

Par exemple on montrera que $\log n! = \Theta(n\log n)$, et on donnera p tel que $a^{\log_b x} = \Theta(x^p)$.

L'analyse algorithmique se fait donc modulo la proportionnalité, reprenons une fonction python simple comme

```
y=0
def f(x):
        global y
        if x>0:
                f(x//2)
                f(x//2)
                y=y+x**2
```

la nouvelle valeur de y est

$$f(x) = \sum_{i=0}^{\lfloor \log_2 x \rfloor} 2^i \left\lfloor \frac{x}{2^i} \right\rfloor^2$$

Supposons que $x = 2^p$ est une puissance de 2, à la fois, cela simplifie l'expression, et c'est suffisant pour montrer que $f(x) = \Theta(x^2)$.

$$f(2^p) = \sum_{i=0}^{p} 2^i \left\lfloor \frac{2^p}{2^i} \right\rfloor^2 = \sum_{i=0}^{p} 2^i 2^{2(p-i)} = 2^p \sum_{i=0}^{p} 2^{p-i} = 2^p(2^{p+1}-1) = \Theta(2^{2p})$$

On a $2^p \leq x < 2^{p+1}$ pour tout x, ce qui permet de passer du cas $x = 2^p$ au cas x quelconque:

$$\frac{c}{4}x^2 < \frac{c}{4}2^{2p+2} = c.2^{2p} \leq f(2^p) \leq f(x) < f(2^{p+1}) \leq d.2^{2p+2} = 4d.2^{2p} \leq 4d.x^2$$

Cette technique peut se généraliser au cas $x = b^p$ pour un entier b quelconque.

Théorème 1 *Soit $f : \mathbb{R}^+ \to \mathbb{R}^+$ une fonction croissante telle qu'il existe des constantes $a, c \geq 0$ et $b > 1$ satisfaisant, pour tout entier p,*

$$f(b^p) = \Theta(b^{pc}p^a),$$

alors

$$f(x) = \Theta(x^c \log^a x)$$

Preuve. Pour tout $x \geq 1$, il existe $p \geq 0$ tel que:

$$b^p \leq x < b^{p+1} = b\, b^p$$

d'où

$$b^{-c}\, x^c \leq b^{pc} \leq x^c$$

En supposant dans la suite que $p \geq 1$ (puisque x est suffisamment grand),

$$p \leq \log_b x < p + 1 \leq 2p$$

d'où

$$2^{-a}\, \log_b^a x \leq p^a \leq \log_b^a x$$

Finalement, on obtient:

$$2^{-a} b^{-c}\, x^c \log_b^a x \leq b^{pc}p^a \leq x^c \log_b^a x$$

Par ailleurs, il existe deux constantes $\alpha, \beta > 0$ et $p_0 \geq 1$ telles que pour tout $p \geq p_0$

$$\alpha\, b^{pc}p^a \leq f(b^p) \leq f(x) \leq f(b^{p+1}) \leq \beta\, b^{(p+1)c}(p+1)^a$$

donc ($p \geq 1$)

$$\alpha\, b^{pc}p^a \leq f(b^p) \leq f(x) \leq f(b^{p+1}) \leq \beta\, b^c\, 2^a\ b^{pc}p^a$$

Il s'ensuit

$$\alpha'\ x^c \log_b^a x \leq f(x) \leq \beta'\ x^c \log_b^a x$$

avec $\alpha' = \alpha\, 2^{-a} b^{-c}$ et $\beta' = \beta\, b^c\, 2^a$. □

Corollaire 1 *Soit $f : \mathbb{R}^+ \to \mathbb{R}^+$ une fonction croissante telle qu'il existe des constantes $a, c, d \geq 0$ et $b > 1$ satisfaisant, pour tout entier p,*

$$f(b^p) = \Theta(b^{pc}(p+d)^a),$$

alors

$$f(x) = \Theta(x^c \log^a x)$$

Preuve. Il suffit de vérifier que $f(b^p) = \Theta(b^{pc}(p+d)^a)$ implique $f(b^p) = \Theta(b^{pc}p^a)$. En effet, ce la découle de

$$\alpha\, b^{pc}p^a \leq \alpha\, b^{pc}(p+d)^a \leq f(b^p) \leq \beta\, b^{pc}(p+d)^a \leq 2^a \beta\, b^{pc}p^a$$

où la dernière inégalité est valide pour $p \geq d$. □

On a vu que les propriétés des fonctions puissances b^x et logarithmes $\log x$ s'établissent naturellement pour les $x \in \mathbb{Q}$ rationnels. Mais ces fonctions sont souvent à valeur non-rationnelles. On montrera que $\log_2 3$, $\log_{10} 2$, $2^{1/2}$ et $5^{1/2}$ ne sont pas rationnel.

L'analyse algorithmique approxime $f(x) = \Theta(g(x))$ les fonctions $f(x)$ étudiées. Ainsi puisque (pour $a, b > 1$)

$$c.\log_b x \leq \log_a x \leq d.\log_b x \quad \text{avec} \quad c = d = \log_a b, \quad \text{donc} \quad \log_a x = \Theta(\log_b x)$$

on notera $\Theta(\log x)$ sans jamais préciser la base du logarithme. Par exemple, on montrera que l'égalité $\int_1^x \frac{1}{t}.dt = \ln x$ trouve un équivalent approximé en $\sum_{i=1}^n \frac{1}{i} = \Theta(\log n)$, qui se démontre avec une observation simple: on a par exemple pour n compris entre 4 et 7

$$\frac{1}{4}+\frac{1}{4}+\frac{1}{2} \leq 1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\left(\frac{1}{5}+\frac{1}{6}+\frac{1}{7}\right) \leq \frac{1}{1}+\frac{1}{2}+\frac{1}{2}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}$$

et d'une manière générale

$$\frac{p}{2} = \sum_{i=1}^{p} 2^{i-1}2^{-i} = \frac{1}{2^p}+\ldots+\frac{1}{8}+\frac{1}{4}+\frac{1}{4}+\frac{1}{2} \leq \sum_{i=1}^{n}\frac{1}{i} \leq \frac{1}{1}+\frac{1}{2}+\frac{1}{2}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{2^p} = \sum_{i=0}^{p} 2^i 2^{-i} = p+1$$

où $2^p \leq n < 2^{p+1}$. On montrera aussi, étant donnée une base $b > 1$, qu'il existe une constante $c = \lim_{h\to 0}\frac{b^h-1}{h}$ telle que les dérivées des fonctions $p(x) = b^x$ et $l(x) = \log_b x$ sont $p'(x) = c.p(x)$ et $l'(x) = \frac{1}{c.x}$. (Ici la constante de Neper, ou nombre d'Euler qui est le e $= b$ tel que $c = 1$, n'est pas pertinent du point de vue approximé $\Theta(\cdot)$ de l'algorithmique.)

L'égalité $f(x) = \Theta(g(x))$ est équivalente aux deux égalités $f(x) = \Omega(g(x))$ et $f(x) = O(g(x))$, avec:

$f(n) = \Omega(g(n))$	$f(n) = O(g(n))$
$\Updownarrow$	$\Updownarrow$
il existe une constante réelle $\alpha > 0$	il existe une constante réelle $\beta > 0$
$\alpha\, g(n) \leq f(n)$	$f(n) \leq \beta\, g(n)$
pour tout n supérieur à une autre constante n_0	pour tout n supérieur à une autre constante n_0

2 Complexité

Par définition, la complexité d'un algorithme est $\Theta(1)$ si son temps d'exécution est indépendant de la taille de ses données en entrée. Aucune distinction à ce niveau donc entre ces trois programmes `a(),b(),c()`

```
def a():
     for i in range(1):

def b():
     for i in range(1000):

def c():
     a()
     b()
     b()
```

le premier, `a()`, déclare une variable `i`, locale à la boucle, initialisée à `i=0`, teste si `i==1`, incrémente `i` à la fin de chaque passage dans la boucle, soit 2 affectations, 1 addition et 2 tests. Pour `b()` on a 1001 affectations, 1000 additions et 1001 tests. Quant à `c()` il effectue $4 + 3 \times 3002$ opérations élémentaires.

La complexité d'un algorithme s'étudie indépendamment de toute considération technologique, en considérant que la taille des données n'est pas bornée. Un algorithme dont le temps d'exécution dépend de la taille des données en entrée, fonction d'un paramètre n non-borné, est

- $\Theta(f(n))$ si son exécution se décompose en $\Theta(f(n))$ algorithmes $\Theta(1)$
- $O(f(n))$ si son exécution se décompose en au-plus $O(f(n))$ algorithmes $\Theta(1)$
- $\Omega(f(n))$ si son exécution se décompose en au-moins $\Omega(f(n))$ algorithmes $\Theta(1)$

On distinguera dans ce cours principalement deux types de complexité algorithmique:

1. Polynomiale: en $\Theta(n^a \log^c n)$, pour $a, c \geq 0$ des entiers
2. Exponentielle: en $\Omega(b^n)$, pour un réel $b > 1$

2.1 Exemples

2.1.1 Euclide et Fibonacci

Étudions la complexité des algorithmes classiques d'Euclide

```
def e(a,b):
        if a*b==0:
                return a+b
        else:
                return e(b,a%b)
```

et de Fibonacci.

```
def f(n):
        if n<2:
                return n
        else:
                return f(n-1)+f(n-2)
```

Le premier algorithme retourne le pgcd, puisque, en notant $r_0 = \max\{a, b\}$ et $r_1 = \min\{a, b\}$ on a

$$e(a, b) = e(b, a) = \begin{cases} r_0 & \text{si } r_1 = 0 \\ e(r_1, r_2) & \text{sinon} \end{cases}$$

où $r_0 = q_1 r_1 + r_2$, pour un entier q_1 et $0 \leq r_2 < r_1$. En effet, d'une part si $r_1 = 0$ le plus grand diviseur commun de r_0, r_1 est r_0, et d'autre part

$$(\exists q, q' \text{ entiers tels que } r_0 = qd \text{ et } r_1 = q'd) \iff (\exists q', q'' \text{ entiers tels que } r_1 = q'd \text{ et } r_2 = q''d)$$

Le second, qui retourne le $n$ème terme de la suite de Fibonacci, a un lien avec la complexité du premier. On montre que la complexité de `f(n)` est exponentielle car

$$T(n) \geq f(n) = \Theta(\phi^n) = \Omega(1.6^n)$$

où $T(n)$ est le nombre d'opérations élémentaires de `f(n)`. En effet, $T(n)$ est au-moins égal au nombre d'appels récursifs à `f(1)`, qui est égal à la valeur retournée $f(n)$. Montrons que $f(n) = \frac{1}{\sqrt{5}}(\phi^n - \bar{\phi}^n)$, avec $\phi = \frac{1+\sqrt{5}}{2}$ et $\bar{\phi} = \frac{1-\sqrt{5}}{2}$. Pour cela, on remarque que

1. $g(n) = g(n-1) + g(n-2)$ pour $g(n) = x^n$ avec x satisfaisant $x^2 = x + 1$,
2. $h(n) = h(n-1) + h(n-2)$ avec $h(n) = ag_1(n) + bg_2(n)$ où $g_i(n) = g_i(n-1) + g_i(n-2)$ pour $i = 1, 2$, et
3. si $h(0) = 0$ et $h(1) = 1$, alors $h(n) = f(n)$.

On montre ensuite que la complexité de `e(a,b)` est $O(n)$ où $n = \Theta(\log r_1)$ est la taille des entiers a, b. Pour cela, on détaille les appels récursives de $e(r_0, r_1), e(r_2, r_1) \ldots, e(r_{n-1}, r_n), e(r_n, 0)$.

$$\begin{aligned}
r_0 &= q_1 r_1 + r_2 && \text{avec} && q_1 \geq 1, && 0 \leq r_2 < r_1 < r_0 \\
r_1 &= q_2 r_2 + r_3 && \text{avec} && q_2 \geq 1, && 0 \leq r_3 < r_2 < r_1 < r_0 \\
&\vdots && \vdots && \vdots \\
r_{n-2} &= q_{n-1} r_{n-1} + r_n && \text{avec} && q_{n-1} \geq 1, && 0 \leq r_n < \ldots < r_1 < r_0 \\
r_{n-1} &= q_n r_n + 0 && \text{avec} && q_n \geq 1, && 0 = r_{n+1} < r_n < \ldots < r_1 < r_0
\end{aligned}$$

Avec $u_i := r_{n-i+1}$, on a $u_0 = 0$, $u_1 \geq 1$, et $u_i \geq u_{i-1} + u_{i-2}$, donc $r_1 = u_n \geq f(n) = \Theta(\phi^n)$.

L'algorithme d'Euclide a une généralisation qui n'augmente pas sa complexité:

```
def B(a,b):
        """on suppose a>=b"""
        if b==0:
                return [1,0,a]
        else:
                [x,y,d]=B(b,a%b)
                return [y,x-(a//b)*y,d]
```

On montre que l'algorithme est valide au sens que pour tout $a, b \in \mathbb{N}$, il retourne $x, y \in \mathbb{Z}$ et $d \in \mathbb{N}$ tels que $ax + by = d$, où d est le plus grand diviseur commun de a, b. Si $b = 0$, c'est vrai avec $x = 1$, $y = 0$ et $d = a$ qui est bien le pgcd de a, b. Si $b > 0$, par induction, l'étape `[x,y,d]=B(b,a%b)` fonctionne et on a donc: $bx + ry = d$ (avec $a = qb + r$). Il suffit de vérifier que les valeurs retournées sont valides, c'est-à-dire que $ay + b(x - qy) = d$. (Cet algorithme est une preuve du théorème de Bachet-Bézout.)

Mais on peut montrer plus rapidement l'exponentialité de `f(n)` en observant que $f(n) \geq 2^i f(n - 2i)$, ce qui implique $f(n) \geq 2^{n/2}.2^{-1/2} f(1)$, d'où $\phi^n = \Omega(b^n)$ avec $b = \sqrt{2} > 1$. On peut par ailleurs modifier `f(n)` en un algorithme $\Theta(n)$ en stockant les valeurs dans un tableau.

```
A=[0 for i in range(n+1)]
A[1]=1
def ft(i):
        global A,n
        if i==0 or A[i]>0:
                return A[i]
        else:
                A[i]=ft(i-1)+ft(i-2)
                return A[i]
```

2.1.2 Eléments de calculs

Après exécution de `h(n)`,

```
y=0
def h(n):
        global y
        for i in range(1,n+1):
                for j in range(n//i):
                        y=y+1
```

on a $y = \sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor = \Theta(\sum_{i=1}^{n} \frac{n}{i}) = \Theta(n \log n)$; donc la complexité de `h(n)` est $\Theta(n \log n)$.

La fonction récursive `p(n,c)` ci-dessous est $\Theta(n^c)$:

```
y=0
def p(n,i):
        global y
        y = y+1
        if i>=1:
                for j in range(n):
                        p(n,i-1)
```

Par exemple, après l'appel à `p(3,2)`, la variable `y` est incrémentée à chaque appel récursif

```
                                    p(3,2)
          p(3,1)                    p(3,1)                    p(3,1)
  p(3,0) p(3,0) p(3,0)      p(3,0) p(3,0) p(3,0)      p(3,0) p(3,0) p(3,0)
```

donc on a $y = 13$. En général, on a $y = \sum_{i=0}^{c} n^i = \frac{n^{c+1}-1}{n-1} = \Theta(n^c)$; donc la complexité de `p(n,c)` est $\Theta(n^c)$. D'où le fait que la complexité de `pp(n)` ci-dessous soit $\Theta(\sum_{i=1}^{n} i^c)$.

```
def pp(n):
        global c
        for i in range(1,n+1):
                p(i,c)
```

La complexité de `pp(n)` est $\Theta(n^{c+1})$ puisque

Lemme 1 $\sum_{i=1}^{n} i^c = \Theta(n^{c+1})$.

Preuve. En effet, d'une part, $\sum_{i=1}^{n} i^c \leq \sum_{i=1}^{n} n^c = n^{c+1}$, d'autre part $\sum_{i=1}^{n} i^c \geq \frac{n}{2} \times \left(\frac{n}{2}\right)^c = \Omega(n^{c+1})$. □

2.2 Complexité exponentielle

Un algorithme en $\Omega(b^n)$ pour une constante $b > 1$ est dit exponentiel. C'est le cas donc de `f(n)` mais pas de `ft(n)`.

Théorème 2 *Considérons un algorithme dont la taille des entrées peut-être représentée par un entier n, et dont le temps d'exécution $T(n)$ satisfait:*

$$T(n) = \begin{cases} \Theta(1) & si\ n < b \\ aT(n-b) + \Theta(n^c) & si\ n \geq b \end{cases}$$

pour des entiers $a, b, c \geq 1$. Alors $T(n) = \Theta(n^{c+1})$ si $a = 1$, et sinon l'algorithme est exponentiel.

Preuve. On a $n = qb + r$ avec $0 \leq r < b$, et on peut montrer, par récurrence sur q, que $T(n) = \Theta\Big(a^q + \sum_{i=0}^{q-1} a^i (n-ib)^c\Big)$. En effet, si $q = 0$, on a $T(n) = T(r) = \Theta(1)$, de plus pour $n = (q+1)b + r$, on a

$$\begin{aligned} T(n) &= aT(qb+r) + \Theta(n^c) = a \times \Theta\Big(a^q + \textstyle\sum_{i=0}^{q-1} a^i (n-b-ib)^c\Big) + \Theta(n^c) \\ &= \Theta\Big(a^{q+1} + \textstyle\sum_{i=0}^{q-1} a^{i+1} (n-b-ib)^c\Big) + \Theta(n^c) = \Theta\Big(a^{q+1} + \textstyle\sum_{i=1}^{q} a^i (n-ib)^c\Big) + \Theta(n^c) \\ &= \Theta\Big(a^{q+1} + \textstyle\sum_{i=0}^{q} a^i (n-ib)^c\Big) \end{aligned}$$

où l'on peut choisir des constantes α, β valables pour chaque notation $\Theta(\cdot)$. Si $a \geq 2$, puisque $a^q \geq a^{\frac{n}{b}-1} = \frac{1}{a} a^{n/b}$, alors $T(n) = \Omega(d^n)$ avec $d = a^{1/b} > 1$; et alors l'algorithme est exponentiel. Supposons donc $a = 1$. On a $\sum_{i=0}^{q-1} (n-ib)^c = \sum_{i=0}^{q-1} (r + (q-i)b)^c = \sum_{i=1}^{q} (r+ib)^c$, de plus, puisque par ailleurs $\sum_{i=1}^{q} i^c = \Theta(q^{c+1})$, on a:

$$\Omega(n^{c+1}) \leq b^c \Theta(q^{c+1}) = \sum_{i=1}^{q} (ib)^c \leq \sum_{i=1}^{q} (r+ib)^c \leq 2 \sum_{i=1}^{q} (ib)^c = 2b^c \Theta(q^{c+1}) \leq O(n^{c+1})$$

d'où, pour tout b et pour $a = 1$, $T(n) = \Theta(n^{c+1})$. □

Corollaire 2 *Considérons un algorithme dont la taille des entrées peut-être représentée par un entier n, et dont le temps d'exécution $T(n)$ satisfait:*

$$T(n) = \begin{cases} \Theta(1) & si\ n < b \\ \sum_{i=1}^{i=k} T(n-b_i) + \Theta(n^c) & si\ n \geq b \end{cases}$$

pour des entiers $b_1, \ldots, b_k, c \geq 1$. Alors $T(n) = \Theta(n^{c+1})$ si $k = 1$, et sinon l'algorithme est exponentiel. □

Ceci implique que le version suivante de `f(n)` n'est pas exponentielle :

```
def fc(n):
        if n==0:
                return [0,1]
        else:
                [u,v]=fc(n-1)
                return [v,u+v]
```

En effet, `fc(n)` est en $\Theta(n)$, c'est le cas $a = 1$, $b = 1$, et $c = 0$.

3 Récursivité de la forme $T(n) = aT(n/b) + \Theta(n^c)$

On rappel le tri fusion qui prend un tableau `t` d'entiers et retourne le tableau trié `tt=tf(t)` dans l'ordre croissant:

```
def tf(t):
    if  len(t) <= 1:        #len(t) = longueur de t
        return t         #si len(t)=1, alors t est déjà trié
    m = len(t)//2
    t1 = t[:m]        #t1=première moitié de t
    t2 = t[m:]        #t2= seconde moitié de t
    tt1 = tf(t1)        #tt1=t1 trié
    tt2 = tf(t2)        #tt2=t2 trié
    tt = fus(tt1,tt2)        #fusion de tt1,tt2 en tt tableau trié
    return tt
```

Si $T(n)$ est le temps d'exécution de `tf(t)` en fonction du nombre n d'entiers dans le tableau, alors `tt1 = tf(t1)` prend un temps $T(n/2)$, `tt1 = tf(t2)` prend un temps $T(n/2)$, et `tt = fus(t1,t2)` prend un temps $\Theta(n)$ puisque qu'on peut l'implémenter ainsi:

```
def fus(t1,t2):        #fusion: si t1 et t2 sont triés, alors tf est l'union triée de t1,t2
    i = 0
    j = 0
    l1 = len(t1)
    l2 = len(t2)
    tf = []
    while i<l1 and j<l2:
        if t1[i] < t2[j]:
            tf.append(t1[i])
            i+= 1
        else:
            tf.append(t2[j])
            j+= 1
    while i<l1:
        tf.append(t1[i])
        i+=1
    while j<l2:
        tf.append(t2[j])
        j+=1
    return tf
```

Toutes les autres opérations ont un temps $O(1)$, donc

$$T(n) = 2T(n/2) + \Theta(n), \quad \text{donc pour le tri fusion: } a = b = 2 \text{ et } c = 1$$

D'une manière générale,

Lemme 2 *Si le temps d'exécution $T(n)$ d'un algorithme récursif, en fonction de la taille n des données, satisfait:*

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq n_0 \\ aT(n/b) + \Theta(n^c) & \text{sinon} \end{cases}$$

pour des constantes entières $a, b, n_0 \geq 1$ et une réelle $c \geq 0$, alors pour tout entier $p \geq 0$

$$T(b^p) = \Theta\Big(\sum_{i=0}^{p} a^i b^{c(p-i)} \Big)$$

Preuve. Soit $f(b^p) = \sum_{i=0}^{p} a^i b^{c(p-i)}$. Soient α', β' les constantes de la fonction $\Theta(n^c)$, et α'', β'' les constantes de la fonction $\Theta(1)$. On définit $\alpha = \min\{1, \alpha', \alpha''\}$ et $\beta = \max\{1, \beta', \beta''\}$, ainsi on a

$$\alpha \leq f(b^0) = a^0 b^{c(0-0)} = 1 \leq \beta$$

d'où

$$\alpha f(b^0) \leq T(b^0) \leq \beta f(b^0)$$

Par ailleurs, si

$$\alpha f(b^p) \leq T(b^p) \leq \beta f(b^p)$$

alors

$$\begin{array}{ccccc}
aT(b^p) + \alpha\, b^{pc+p} & \leq & T(b^{p+1}) & \leq & aT(b^p) + \beta\, b^{pc+p} \\
a\alpha f(b^p) + \alpha\, b^{pc+p} & \leq & T(b^{p+1}) & \leq & a\beta f(b^p) + \beta\, b^{pc+p} \\
\alpha\Big(\sum_{i=0}^{p} a^{i+1} b^{c(p-i)} + b^{pc+p}\Big) & \leq & T(b^{p+1}) & \leq & \beta\Big(\sum_{i=0}^{p} a^{i+1} b^{c(p-i)} + b^{pc+p}\Big) \\
\alpha\Big(\sum_{i=1}^{p+1} a^{i} b^{c(p+1-i)} + b^{pc+p}\Big) & \leq & T(b^{p+1}) & \leq & \beta\Big(\sum_{i=1}^{p+1} a^{i} b^{c(p+1-i)} + b^{pc+p}\Big) \\
\alpha\Big(\sum_{i=0}^{p+1} a^{i} b^{c(p+1-i)}\Big) & \leq & T(b^{p+1}) & \leq & \beta\Big(\sum_{i=0}^{p+1} a^{i} b^{c(p+1-i)}\Big)
\end{array}$$

d'où $\alpha f(b^{p+1}) \leq T(b^{p+1}) \leq \beta f(b^{p+1})$. □

Cette égalité obtenue simplement pour le cas particulier $n = b^p$ permettra en fait d'obtenir l'expression de $T(n)$ pour tout entier n. Pour le tri-fusion par exemple, $a = b = 2$ et $c = 1$, on obtient

$$T(2^p) = \Theta\big(\sum_{i=0}^{p} 2^i 2^{(p-i)} = (p+1)2^p\big)$$

D'après Corollaire 1, on retrouve $T(n) = \Theta(n \log n)$. Avant de traiter le cas général, on montre

Lemme 3 *Pour tout réel $r > 0$*

$$\sum_{i=0}^{n} r^i = \begin{cases} \Theta(1) & si\ r < 1 \\ \Theta(n) & si\ r = 1 \\ \Theta(r^n) & si\ r > 1 \end{cases}$$

Preuve. Si $r < 1$ alors $0 < r^{n+1} < 1$ d'où

$$0 \leq \frac{1 - r^{n+1}}{1 - r} \leq \frac{1}{1 - r}$$

Si $r = 1$ alors $\sum_{i=0}^{n} r^i = n + 1 = \Theta(n)$. Si $r > 1$, alors $r^{n+1} > r > 1$ et on a

$$r^n = \frac{r^{n+1}}{r} \leq \frac{r^{n+1} - 1}{r - 1} \leq \frac{r}{r - 1}\, r^n$$

où la première inégalité découle de $a \geq b \Leftrightarrow \frac{a-1}{b-1} \geq \frac{a}{b}$. □

Théorème 3 (*"Master Theorem") Considérons un algorithme dont la taille des entrées peut-être représentée par un entier n, et dont le temps d'exécution $T(n)$ satisfait:*

$$T(n) = \begin{cases} \Theta(1) & si\ n \leq n_0 \\ aT(n/b) + \Theta(n^c) & si\ n > n_0 \end{cases}$$

pour des entiers a, b, c et n_0. Alors

$$T(n) = \begin{cases} \Theta(n^c) & si\ c > \log_b a \\ \Theta(n^c \log n) & si\ c = \log_b a \\ \Theta(n^{\log_b a}) & si\ c < \log_b a \end{cases}$$

Preuve. D'après Corollaire 1, il suffit de monter que

$$T(b^p) = \begin{cases} \Theta(b^{cp}) & \text{si } c > \log_b a \\ \Theta(b^{cp}p) & \text{si } c = \log_b a \\ \Theta(a^p) & \text{si } c < \log_b a \end{cases}$$

puisque $a^p = n^{\log_b a}$ avec $n = b^p$. Puisque

$$\begin{aligned} \sum_{i=0}^{p} a^i b^{c(p-i)} &= b^{cp} \sum_{i=0}^{p} a^i b^{-ic} \\ &= b^{cp} \sum_{i=0}^{p} \left(\frac{a}{b^c}\right)^i \end{aligned}$$

d'après Lemme 2, on a

$$T(b^p) = \Theta\left(b^{cp} \sum_{i=0}^{p} r^i\right) \quad \text{avec} \quad r = \frac{a}{b^c}$$

et donc, d'après Lemme 3

$$T(b^p) = \begin{cases} \Theta(b^{cp}\Theta(1)) & \text{si } a < b^c \\ \Theta(b^{cp}\Theta(p)) & \text{si } a = b^c \\ \Theta(b^{cp}\Theta(\frac{a^p}{b^{cp}})) & \text{si } a > b^c \end{cases}$$

□

Remarquons que le n_0 dans l'hypothèse du théorème ne joue aucun rôle dans sa conclusion (qui dépend uniquement de a, b, c).

4 Performance des algorithmes

4.1 Puissance x^n

On utilisera le Master theorem pour élaborer un algorithmme d'élévation à la puissance n de complexité $\Theta(\log n)$.

4.2 Multiplication d'entiers (Karatsuba)

On peut caractériser la taille de deux entiers a, b par $n = \max\{\log a, \log b\}$ (en particulier, la taille d'un entier a en base 10 est son nombre de chiffres, soit $1 + \lfloor \log_{10} a \rfloor$).

1. L'addition ou la multiplication d'entiers de taille bornée est $\Theta(1)$, en particulier celle de deux chiffres $0, 1, \ldots, 9$.
2. L'algorithme d'addition de deux entiers a, b non-bornés, appris à la petite école, a une complexité $\Theta(n)$
3. L'algorithme de multiplication de deux entiers a, b, appris à la petite école, a une complexité $\Theta(n^2)$

Examinons un algorithme récursif de multiplication de deux entiers x, y de taille n (c-à-d $\log x$, $\log y = \Theta(n)$). Avec $x = a\,10^{n/2} + b$ et $y = c\,10^{n/2} + d$, où a, b, c, d sont des entiers de taille $n/2$, alors $xy = ac\,10^n + (ad + bc)\,10^{n/2} + bd$. On peut recomposer xy à partir de ac, ad, bc, bd en $\Theta(n)$ puisqu'il ne s'agit alors que d'additions à effectuer. On peut calculer ac, ad, bc, bd avec des appels récursifs, donc le temps d'exécution satisfait

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq n_0 \\ 4T(n/2) + \Theta(n) & \text{si } n > n_0 \end{cases}$$

C'est le cas $a = 4$, $b = 2$ et $c = 1$ du Master Theorem, on obtient $T(2^p) = \Theta\big(\sum_{i=0}^{p} 4^i 2^{(p-i)} = 2^p \sum_{i=0}^{p} 2^i\big) = \Theta(2^{2p})$, et on trouve $T(n) = \Theta(n^2)$ d'après le Corolaire 1[1] ce qui est pareil que l'algorithme élémentaire. La remarque suivante est cruciale: $ad + bc = (a + b)(c + d) - ac - bd$. En effet, elle permet de calculer xy en $\Theta(n)$ avec, en plus, seulement 3 appels récursifs au lieu de 4. On passe ainsi à une équation

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq n_0 \\ 3T(n/2) + \Theta(n) & \text{si } n > n_0 \end{cases}$$

C'est le cas $a = 3$, $b = 2$ et $c = 1$ du Master Theorem, on obtient $T(2^p) = \Theta\big(\sum_{i=0}^{p} 3^i 2^{(p-i)} = 3^p \sum_{i=0}^{p} (\frac{2}{3})^i\big) = \Theta(3^p)$, et on trouve $T(n) = \Theta(n^{\log_2 3})$ d'après le Corolaire 1[2].

Repensons à la remarque cruciale: $ad + bc = (a + b)(c + d) - ac - bd$; qui permet de multiplier des entiers x, y de taille $n = O(\log(\max\{x, y\}))$ en $\Theta(n^{\log_2 3})$ au lieu de $\Theta(n^2)$. Visuellement

$$ad + bc = \begin{pmatrix} \cdot & + \\ + & \cdot \end{pmatrix} = \begin{pmatrix} + & + \\ + & + \end{pmatrix} - \begin{pmatrix} + & \cdot \\ \cdot & \cdot \end{pmatrix} - \begin{pmatrix} \cdot & \cdot \\ \cdot & + \end{pmatrix}$$

où une matrice représente une combinaison à coefficients $\{0, 1\}$ des sous-produits ac, ad, bc, bd

$$\begin{pmatrix} ac & ad \\ bc & bd \end{pmatrix}$$

La structure de la matrice permet de savoir si elle peut s'obtenir en un seul produit ou non,

$$\begin{pmatrix} + & \cdot \\ \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & + \\ \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot \\ + & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot \\ \cdot & + \end{pmatrix} \begin{pmatrix} + & + \\ \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot \\ + & + \end{pmatrix} \begin{pmatrix} + & \cdot \\ + & \cdot \end{pmatrix} \begin{pmatrix} \cdot & + \\ \cdot & + \end{pmatrix} \begin{pmatrix} + & + \\ + & + \end{pmatrix}$$

s'obtiennent en un seul produit, mais pas

$$\begin{pmatrix} \cdot & + \\ + & + \end{pmatrix} \begin{pmatrix} + & + \\ + & \cdot \end{pmatrix} \begin{pmatrix} + & + \\ \cdot & + \end{pmatrix} \begin{pmatrix} + & \cdot \\ + & + \end{pmatrix} \begin{pmatrix} + & \cdot \\ \cdot & + \end{pmatrix} \begin{pmatrix} \cdot & + \\ + & \cdot \end{pmatrix}$$

4.3 Multiplication de matrices (Strassen)

On suppose ici que la taille des entrées des matrices est bornée, c'est-à-dire que les multiplications et additions d'entiers sont des opérations élémentaires en $\Theta(1)$.

Un algorithme naturel en $\Theta(n^3)$ pour déterminer le produit $C = AB$ de deux n-matrices carrées est:

```
def mult(A,B):
        n = len(A)
        C = [[0 for i in range(n)] for j in range(n)]
        for i in range(n):
                for j in range(n):
                        for k in range(n):
                                C[i][j] += A[i][k] * B[k][j]
        return C
```

puisque par définition

$$C = \left(\sum_{k=1}^{n} a_{ik} b_{kj} \right)_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} \quad \text{avec} \quad A = \big(a_{ij}\big)_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} \text{ et } B = \big(b_{ij}\big)_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}}$$

[1] ou, puisque $2^{2p} \leq n^2 \leq 2^{2p+2}$ et

$$\frac{c}{4} n^2 \leq \frac{c}{4} 2^{2p+2} \leq c\, 2^{2p} \leq T(2^p) \leq T(n) \leq T(2^{p+1}) \leq d\, 2^{2p+2} \leq 4d\, 2^{2p} \leq 4d\, n^2$$

[2] ou, puisque $3^p \leq n^{\log_2 3} = 3^{\log_2 n} \leq 3^{p+1}$ et

$$\frac{c}{3} n^{\log_2 3} \leq \frac{c}{3} 3^{p+1} \leq c\, 3^p \leq T(2^p) \leq T(n) \leq T(2^{p+1}) \leq d\, 3^{p+1} \leq 3d\, 3^p \leq 3d\, n^{\log_2 3}$$

On peut définir, pour $X \in \{A, B, C\}$

$$X_{11} = \big(x_{ij}\big)_{\substack{1 \le i \le n/2 \\ 1 \le j \le n/2}} \quad X_{12} = \big(x_{ij}\big)_{\substack{1 \le i \le n/2 \\ n/2 < j \le n}} \quad X_{21} = \big(x_{ij}\big)_{\substack{n/2 < i \le n \\ 1 \le j \le n/2}} \quad X_{22} = \big(x_{ij}\big)_{\substack{n/2 < i \le n \\ n/2 < j \le n}}$$

ainsi

$$\begin{aligned}
A_{11}B_{11} + A_{12}B_{21} &= \left(\textstyle\sum_{k=1}^{n/2} a_{ik}b_{kj}\right)_{\substack{1 \le i \le n/2 \\ 1 \le j \le n/2}} + \left(\textstyle\sum_{k>n/2}^{n} a_{ik}b_{kj}\right)_{\substack{1 \le i \le n/2 \\ 1 \le j \le n/2}} = \left(\textstyle\sum_{k=1}^{n} a_{ik}b_{kj}\right)_{\substack{1 \le i \le n/2 \\ 1 \le j \le n/2}} = C_{11} \\
A_{11}B_{12} + A_{12}B_{22} &= \left(\textstyle\sum_{k=1}^{n/2} a_{ik}b_{kj}\right)_{\substack{1 \le i \le n/2 \\ n/2 < j \le n}} + \left(\textstyle\sum_{k>n/2}^{n} a_{ik}b_{kj}\right)_{\substack{1 \le i \le n/2 \\ n/2 < j \le n}} = \left(\textstyle\sum_{k=1}^{n} a_{ik}b_{kj}\right)_{\substack{1 \le i \le n/2 \\ n/2 < j \le n}} = C_{12} \\
A_{21}B_{11} + A_{22}B_{21} &= \left(\textstyle\sum_{k=1}^{n/2} a_{ik}b_{kj}\right)_{\substack{n/2 < i \le n \\ 1 \le j \le n/2}} + \left(\textstyle\sum_{k>n/2}^{n} a_{ik}b_{kj}\right)_{\substack{n/2 < i \le n \\ 1 \le j \le n/2}} = \left(\textstyle\sum_{k=1}^{n} a_{ik}b_{kj}\right)_{\substack{n/2 < i \le n \\ 1 \le j \le n/2}} = C_{21} \\
A_{21}B_{12} + A_{22}B_{22} &= \left(\textstyle\sum_{k=1}^{n/2} a_{ik}b_{kj}\right)_{\substack{n/2 < i \le n \\ n/2 < j \le n}} + \left(\textstyle\sum_{k>n/2}^{n} a_{ik}b_{kj}\right)_{\substack{n/2 < i \le n \\ n/2 < j \le n}} = \left(\textstyle\sum_{k=1}^{n} a_{ik}b_{kj}\right)_{\substack{n/2 < i \le n \\ n/2 < j \le n}} = C_{22}
\end{aligned}$$

d'où

$$AB = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = C$$

À partir des 8 sous-produits $A_{ij}B_{k\ell}$ ci-dessus, on peut reconstituer le produit C en $\Theta(n^2)$, le temps des additions, donc, si n est une puissance de 2, ou un entier quelconque par le Corollaire 1, on a un algorithme récursif dont la complexité est, d'après le Master Theorem:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \le n_0 \\ 8T(n/2) + \Theta(n^2) & \text{si } n > n_0 \end{cases} = \Theta(n^{\log_2 8}) = \Theta(n^3)$$

Inspirons nous de cette approche pour élaborer un algorithme récursif pour multiplier les matrices n-carrées en

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \le n_0 \\ 7T(n/2) + \Theta(n^2) & \text{si } n > n_0 \end{cases} = \Theta(n^{\log_2 7}) < \Theta(n^3)$$

Visuellement, on cherche

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \quad C_{12} = A_{11}B_{12} + A_{12}B_{22} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} = \begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \quad C_{22} = A_{21}B_{12} + A_{22}B_{22} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}$$

où la matrice membre de droite des égalités représentent une combinaisons $\{-1, 0, +1\}$ des sous-produits $A_{ij}B_{k\ell}$ par des matrices

$$\begin{pmatrix} A_{11}B_{11} & A_{12}B_{11} & A_{21}B_{11} & A_{12}B_{11} \\ A_{11}B_{12} & A_{12}B_{12} & A_{21}B_{12} & A_{12}B_{12} \\ A_{11}B_{21} & A_{12}B_{21} & A_{21}B_{21} & A_{12}B_{21} \\ A_{11}B_{22} & A_{12}B_{22} & A_{21}B_{22} & A_{12}B_{22} \end{pmatrix}$$

La structure de ces matrices permet de voir si les combinaisons correspondantes peuvent d'obtenir en un seul produit, par exemple

$$-A_{11}B_{11} + A_{21}B_{11} - A_{11}B_{12} + A_{21}B_{12} = \begin{pmatrix} - & \cdot & + & \cdot \\ - & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = (-A_{11} + A_{21})(B_{11} + B_{12})$$

s'obtient par un unique produit mais pas

$$-A_{11}B_{11} + A_{21}B_{11} - A_{11}B_{12} = \begin{pmatrix} - & \cdot & + & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = -A_{11}(B_{11} + B_{12}) + A_{21}B_{11}$$

On obtient ainsi les quatre sous-matrices composant $C = AB$ à partir de seulement 7 produits de sous-matrices: d'abord on peut utiliser 4 produits de sous-matrices pour obtenir deux sous-matrice de C

$$\begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & + & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \end{pmatrix}$$

$$\begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

(En l'occurrence, avec $P_1 = (A_{11} + A_{12})B_{22}$, $P_2 = A_{11}(B_{12} - B_{22})$ et $P_3 = (A_{21} + A_{22})B_{11}$, $P_4 = A_{22}(-B_{11} + B_{21})$ on obtient $C_{12} = P_1 + P_2$ et $C_{21} = P_3 + P_4$.) Puis on remarque qu'en combinant ces matrices avec une cinquième (on calcule $P_5 = (A_{11} + A_{22})(B_{11} + B_{22})$), on peut obtenir deux nouvelles matrices

$$\begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & - & \cdot & + \end{pmatrix} = \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} - \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & + & \cdot & \cdot \end{pmatrix}$$

$$\begin{pmatrix} + & \cdot & - & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix} = \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \end{pmatrix} - \begin{pmatrix} \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Au final, on a donc les deux sous-matrices de C manquantes avec seulement deux autres matrices

$$\begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & - & \cdot & + \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & - \\ \cdot & + & \cdot & - \end{pmatrix}$$

$$\begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix} = \begin{pmatrix} + & \cdot & - & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix} + \begin{pmatrix} - & \cdot & + & \cdot \\ - & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

soit bien seulement 7 matrices en tout. (On calcule $P_6 = (A_{12} - A_{22})(B_{21} + B_{22})$ et $P_7 = (-A_{11} + A_{21})(B_{11} + B_{21})$, et on obtient $C_{11} = P_6 + P_5 + P_4 - P_1$ et $C_{22} = P_7 + P_5 + P_2 - P_3$.)

5 Force brute

Après l'initialisation d'un tableau `A` de taille `n=len(A)` et d'un entier `b`, par exemple

```
b=2
n=3
X=[0 for i in range(n)]
```

l'appel à `count(0)` affichera l'expression en base b de tous les entiers de 0 à $b^n - 1$.

```
def count(i):
        for j in range(b):
                X[i]=j
                if i==n-1:
                        print X
                else:
                        count(i+1)
```

Pour les valeurs en exemple, les premiers appels récursifs à `count(0)` avec `X[0]=0`, puis `count(1)` avec `X[1]=0`, puis `count(2)` avec `X[2]=0` et `print` puis `X[2]=1` et `print`, donneront le schéma

```
                                        ...
                    0..
          00.
    000       001
```

Après l'exécution de `X[0]=1` lors du deuxième passage dans `for` de l'appel à `count(0)`, on aura le schéma

```
                                        ...
                    0..                                         1..
          00.                   01.
    000       001           010       011
```

On affiche bien les expressions en base 2 des entiers de 0 à 7:

```
                                        ...
                    0..                                         1..
          00.                   01.                   10.                   11.
    000       001           010       011           100       101       110       111
```

Proposition 1 *Pour tout entier n et tout entier $b < n$, l'appel à* `count(0)` *affiche l'expression en base b de tous les entiers de 0 à $b^n - 1$.*

Preuve. Il suffit de montrer la proposition générale suivante : pour tout $i = 1, \ldots, n$, `count(n-i)` affiche tous les différents tableaux `X` avec

1. `X[:n-i]` inchangé, et
2. `X[n-i:]` qui sont les expressions en base b des entiers de $0, \ldots, b^i - 1$.

puisque l'énoncé est le cas particulier $i = n$.

C'est vrai pour $i = 1$, car l'appel `count(n-1)` affiche tous les tableaux `X` avec `X[:n-1]` inchangé et `X[n-1]=j` pour $j = 0, \ldots, b-1$. Supposons que ce soit vrai jusqu'à un certain $i \geq 1$. Alors l'appel à `count(n-i-1)` affecte `X[n-i-1]=j` puis appelle `count(n-i)` pour $j = 0, \ldots, b-1$. Donc, par hypothèse de récurrence, `count(n-i-1)` affiche tous les différents tableaux `X` avec

1. `X[:n-i-2]` inchangé,
2. pour $j = 0, \ldots, b-1$:
   (a) `X[n-i-1]=j`
   (b) `X[n-i:]` qui sont les expressions en base b des entiers de $0, \ldots, b^i - 1$.

Ainsi, si `count(n-i)` satisfait la proposition, alors `count(n-(i+1))` aussi. □

Pour $b = 2$, `count(0)` est équivalent à l'énumération d'objets mathématiques classiques puisque les ensembles suivants sont en bijection

- tableaux `A` de longueur `n=len(A)` avec `A[i]=0` ou `A[i]=1` pour tout $i = 0, \ldots, n-1$
- vecteurs $x \in \{0,1\}^n$
- sous-ensembles de $\{1, \ldots, n\}$

On peut ainsi modifier la procédure `count` pour énumérer tous les vecteurs $x \in \{0,1\}^n$ satisfaisant des contraintes $Ax \leq \mathbf{1}$ où A est une matrice 0-1. Par exemple avec

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

On initialise

```
n=5
X=[0 for i in range(n)]
G=[[0,1],[0,2],[1,2],[0,4],[2,3],[3,4]]
```

Puis on énumère tous les vecteurs $x \in \{0,1\}^5$ en testant, avant affichage, que les contraintes $Ax \leq \mathbf{1}$, stockées dans `G`, sont satisfaites

```
def stable(i):
        for j in range(2):
                X[i]=j
                if i==n-1:
                        t=1
                        for e in range(m):
                                if X[G[e][0]]==1 and X[G[e][1]]==1:
                                        t=0
                        if t==1:
                                print X
                else:
                        stable(i+1)
```

****

Pour énumérer toutes les $n!$ permutations, il faut initialiser, par exemple:

```
n=5
X=[0 for i in range(n)]
U=[0 for i in range(n)]
```

où `U[i]==1` si $i \in \{0, 1, \ldots, n-1\}$ est présent dans `X`, et `U[i]==0` sinon. L'appel à `permut(0)` énumère ainsi toutes les permutations:

```
def permut(i):
        for j in range(n):
                if U[j]==0 :
                        X[i]=j
                        U[j]=1
                        if i==n-1:
                                print X
                        else:
                                permut(i+1)
                        U[j]=0
```

Un autre exemple est l'énumération des placements de 8 reines dans un échiquier sans aucune menace entre elles. L'échiquier sans pièce est représenté par

```
n=8
Q=[-1 for i in range(n)]
r=[0 for i in range(n)]
c=[0 for i in range(2*n-1)]
d=[0 for i in range(2*n-1)]
```

Les tableaux `Q` ayant des valeurs `Q[i]` dans $\{0,1,\ldots,7\}$ sont en bijection avec les échiquiers contenant exactement une reine par colonne. L'instruction `Q[i]=j` place une reine en colonne i ligne j, et

1. `r[j]=1` signifie que la $j$ème ligne
2. `c[i+j]=1` signifie que la $(i+j)$ème diagonale Nord-Ouest
3. `d[i-j+n-1]=1` signifie que la $(i-j+n-1)$ème diagonale Nord-Est

est alors menacée par cette reine. Ainsi `queen(0)` affiche tous les placements possibles:

```
def queen(i):
        for j in range(n):
                if r[j]==0 and c[i+j]==0 and d[i-j+n-1]==0:
                        Q[i]=j
                        r[j]=c[i+j]=d[i-j+n-1]=1
                        if i==n-1:
                                print Q
                        else:
                                queen(i+1)
                        r[j]=c[i+j]=d[i-j+n-1]=0
```

Illustrons le cas $n=4$. Les numéros des lignes et diagonales Nord-Ouest et Nord-Est sont respectivement:

j

3	3	3	3
2	2	2	2
1	1	1	1
0	0	0	0

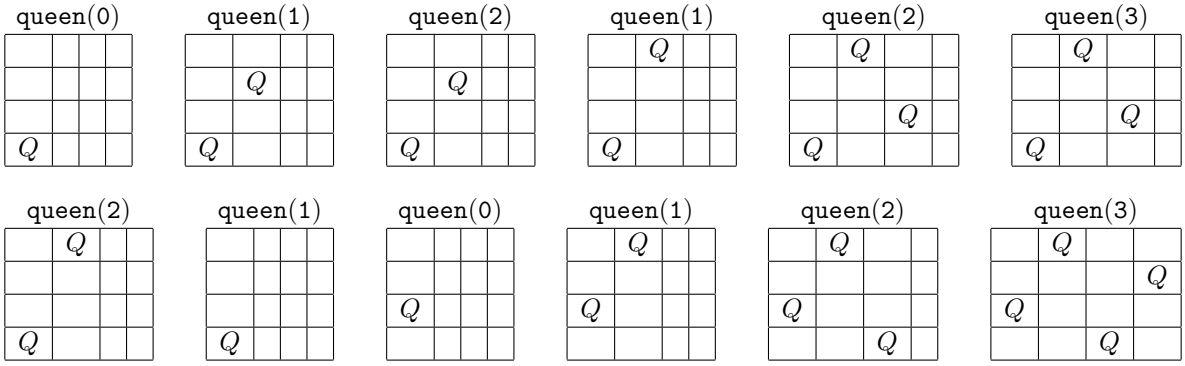
$i+j$

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

$i-j+3$

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

Les premiers appels récursifs placent:



et affiche un premier placement sans menace.

6 Tri

L'algorithme implémenté en python par `tf(t)`, de la forme $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$, trie dans l'ordre croissant le tableau d'entier `t`. L'algorithme suivant, implémenté par `heapsort(A)`, retourne le tableau trié `B` qui contient les éléments de A sauf `A[0]==0`.

```
def tas(A,i):        #A[0]==0 and A[i]>0
        n=len(A)
        if 2*i<n:
                x=2*i
                if 2*i+1<n and A[2*i+1]<A[x]:
                        x=2*i+1
                if A[x]<A[i]:
                        z=A[x]
                        A[x]=A[i]
                        A[i]=z
                        tas(A,x)
def heap(A):  #A[0]==0 and A[i]>0
        n=len(A)-1
        for i in range(n/2+1):
                tas(A,n/2-i)
def heapsort(A):   #A[0]==0 and A[i]>0
        n=len(A)
        heap(A)
        B=[0 for i in range(n-1)]
        for i in range(n-1):
                B[i]=A[1]
                A[1]=A[n-1-i]
                A=A[:n-1-i]
                tas(A,1)
        return B
```

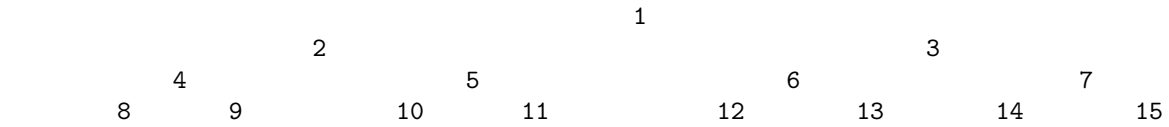
On note $A[1], \ldots, A[n]$ les d'éléments non-nuls du tableau. Si on a:

$$A[i] \leq \min\{A[2i], A[2i+1]\} \quad \forall i = 2, 3, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$$

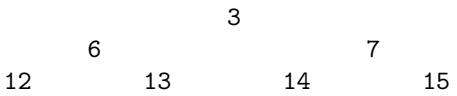
alors, on montre par une récurrence sur $h = \lfloor \log_2 n \rfloor - 1$, c'est-à-dire la hauteur de l'arbre binaire de racine 1, où $2i$ et $2i+1$ sont les fils de i, qu'après l'appel de la procédure `tas(A,1)`

$$A[i] \leq \min\{A[2i], A[2i+1]\} \quad \forall i = 1, 2, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$$

Cette propriété est vraie pour le tableau grâce à la structure d'arbre binaire qu'il possède implicitement,



et elle est aussi vraie pour les sous-tableaux possédant cette structure, comme par exemple le sous-tableau engendré par 3



Si on a:

$$A[i] \leq \min\{A[2i], A[2i+1]\} \quad \forall i = 6, 7$$

alors, après l'appel de la procédure `tas(A,3)`, on a

$$A[i] \leq \min\{A[2i], A[2i+1]\} \quad \forall i = 3, 6, 7$$

Donc `heap(A)` permute les valeurs des éléments pour avoir $A[i] \leq \min\{A[2i], A[2i+1]\}$ partout. Ainsi $A[1]$ est la valeur non-nulle minimum du tableau et `heapsort(A)` renvoie les valeurs non-nulles triées dans l'ordre croissant. La complexité au pire cas de `heapsort(A)` est $\Theta(n \log n)$ puisqu'au pire cas, `tas(A,i)` est $\Theta(\log n)$.

Un autre algorithme de tri est:

```
def trins(A):
        for j in range(1,len(A)):
                x=A[j]
                i=j-1
                while i>=0 and A[i]>x:
                        A[i+1]=A[i]
                        i -= 1
                A[i+1]=x
```

On montre, par induction sur j, qu'après `trins(A)` la tableau est trié car:

- `A[:j]` est trié si $j = 1$
- si `A[:j]` est trié, alors `A[:j+1]` est trié après l'exécution des tâches dans le corps de la boucle `for`

Le nombre de passages dans la boucle `while` dépend de `A[:j]`

(1) Si `A[0]>x`, alors on passe j fois

(2) Si `A[j-1]<x`, alors on ne passe pas dans la boucle

Si la tableau est déjà trié, c'est le cas (2) à chaque passage et la complexité est $\Theta(n)$. Au contraire, s'il est trié à l'envers, c'est le cas (1) à chaque passage et et la complexité est $\Theta(n^2) = \sum_{j=1}^{n-1} j = n(n-1)/2$. En moyenne, ça reste $\Theta(n^2) = \sum_{j=1}^{n-1} \frac{j}{2} = n(n-1)/4$.

Tout algorithme (déterministe) de tri basé sur des comparaisons peut être représenté par un arbre binaire dont la racine est la première comparaison, la comparaison suivante est le fils gauche ou droite selon la résultat de la comparaison effectuée au sommet père. Les feuilles de l'arbre sont les différentes permutations effectuées par l'algorithme. Si l'algorithme est valide, il trie n'importe quelle permutation et le nombre de feuilles est donc $\geq n!$. Par-ailleurs, l'arbre étant binaire, il a au plus 2^h feuilles, où h est la hauteur de l'arbre. D'où $n! \leq 2^h$ et $h \geq \log_2 n! = \Theta(n \log n)$. Ainsi la complexité au pire cas d'un tel algorithme est $\Theta(n \log n)$, et cette borne est atteinte par les tri fusion et par tas.

Un autre algorithme de tri est

```
def max(A):
        m=A[0]
        for i in range(1,len(A)):
                if A[i]>m:
                        m=A[i]
        return m
def triden(A):
        n=len(A)
        m=max(A)
        B=[0 for i in range(n)]
        C=[0 for j in range(m+1)]
        for i in range(n):
                C[A[i]] += 1
        for j in range(1,m+1):
                C[j]=C[j-1]+C[j]
        for i in range(n):
                B[C[A[i]]-1]=A[i]
        return B
```

Sa validité vient du fait qu'il y a exactement `C[A[i]]-1` éléments de `A` plus petit que `A[i]`. Sa complexité est $\Theta(n + m)$ où m est la valeur renvoyée par `max(A)`.