Corrigé - Algo 3

Algorithmique Et Programmation (Université Paris Dauphine)

Travaux dirigés : Complexité 1

Rappels de cours *Les* notations asymptotiques *correspondent aux ensembles de fonctions :*
$\Omega(g(n)) = \{f(n) : 0 \leq cg(n) \leq f(n),\ \exists c > 0,\ \exists n_0 \in \mathbb{N},\ \forall n \geq n_0\}$,
$\Theta(g(n)) = \{f(n) : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n),\ \exists c_1, c_2 > 0,\ \exists n_0 \in \mathbb{N},\ \forall n \geq n_0\}$,
$O(g(n)) = \{f(n) : 0 \leq f(n) \leq cg(n),\ \exists c > 0,\ \exists n_0 \in \mathbb{N},\ \forall n \geq n_0\}$,
mais l'on note $f(n) = O(g(n))$ *lorsque* $f(n) \in O(g(n))$ *(idem pour* Ω *et* Θ*).*
Par définition la complexité d'un algorithme *est* $O(1)$ *s'il est* constant *(déterministe et sans paramètre en entrée). Sinon, il existe une variable* n *représentant la taille de son entrée (idéalement,* n *est le nombre de bits codant l'entrée). Si l'exécution au pire cas se décompose en au-plus* $f(n) = O(g(n))$ *exécutions de sous-algorithmes constants, on dit que l'algorithme principal est en* $O(g(n))$*, ou que sa complexité est* $O(g(n))$*. De même, il est en* $\Omega(g(n))$ *si elle se décompose en au-moins* $f(n) = \Omega(g(n))$ *exécutions de sous-algorithmes constants. Il est en* $\Theta(g(n))$ *s'il se décompose en au-plus* $O(g(n))$ *et au-moins* $\Omega(g(n))$ *exécutions d'algorithmes constants.*

Correction d'exercices :
Ex 1
Soit $f : \mathbb{Q} \to \mathbb{Q}_+$ une fonction puissance.

1) Montrons que $f(0) = 1$

On a pour $x \in \mathbb{Q}$, $f(x) = f(x+0) = f(x)f(0)$. Donc $f(0) = 1$.

2) Montrons que $f(x-y) = \frac{f(x)}{f(y)}$

On a pour $x, y \in \mathbb{Q}$, $f(x) = f(x-y+y) = f(x-y)f(y)$. Donc $f(x-y) = \frac{f(x)}{f(y)}$.

3) Montrons que $f(ax) = f(x)^a$

On a $f(ax) = f((a-1)x + x) = f((a-1)x)f(x) = ... = f(x)^a$.

4) Montrons que $f(1) = f(\frac{1}{a})^a$

On a $f(1) = f(a * (\frac{1}{a})) = f(\frac{1}{a})^a$ d'après 4).

5) Montrons que $f(x)$ est injective (à moins que $f(x) = 1$ pour tout x)

Par 1) et 3), on a que $f(x)$ est injective, car si $f(x) = f(y)$, alors $1 = f(x)/f(y) = f(x-y)$ donc $x - y = 0$.

Ex 2
Soit $g : \mathbb{R}_+ \to \mathbb{R}$ une fonction puissance.

1) Montrons que $g(1) = 0$

On a pour $x \in \mathbb{R}_+$, $g(x) = g(x * 1) = g(x) + g(1)$. Donc $g(1) = 0$.

2) Montrons que $g(\frac{x}{y}) = g(x) - g(y)$

On a pour $x, y \in \mathbb{R}_+$, $g(x) = g(\frac{x*y}{y}) = g(\frac{x}{y}) + g(y)$. Donc $g(\frac{x}{y}) = g(x) - g(y)$.

3) Montrons que $g(x^q) = qg(x)$ pour tout $q \in \mathbb{Q}$

On a $g(x^q) = g(x^{q-1} * x) = g(x^{q-1}) + g(x) = ... = qg(x)$.

4) Montrons que $g(x)$ est injective (à moins que $g(x) = 0$ pour tout x)

Par 1) et 3) on a que $g(x)$ est injective, car si $g(x) = g(y)$, alors $0 = g(x) - g(y) = g(\frac{x}{y})$ d'où $\frac{x}{y} = 1$.

Ex 4
Montrons que $log_b(n) = log_b(a) \cdot log_a(n)$.

Soit les réels x, y, z tels que $b^x = n, b^y = a$ et $a^z = n$, ainsi $b^x = a^z = b^{y^z}$ et $x = yz$ découle de la bijectivité de la fonction puissance, ce qui implique l'inégalité demandée (par la définition du log).

Ex 5
Montrons que $a^{log_b(n)} = n^{log_b(a)}$.
$\forall b, a \in \mathbb{N}, a, b > 1, n > 0, a^{log_b(n)} = e^{log_b(n) \cdot ln(a)} = e^{\frac{ln(n)}{ln(b)} \cdot ln(a)} = e^{\frac{ln(a)}{ln(b)} \cdot ln(n)} = n^{log_b(a)}$

Ex 6
Montrons que $(b^q)^{log_b(a)} = b^{qlog_b(a)} = a^q$
D'après l'exercice précédent, on a $(b^q)^{log_b(a)} = a^{log_b(b^q)}$, or $log_b(b^q) = q$. Donc $(b^q)^{log_b(a)} = a^q$.

Ex 7
Montrons que pour tous entiers n et $b > 1$, il existe $p \in \mathbb{N}$ tel que $n \leq b^p \leq bn$.
Il suffit de choisir $p = \lceil log_b(n) \rceil$. On a directement $n \leq b^p \leq b^{\lfloor log_b(n) \rfloor + 1} \leq bn$,
(on a $b^{\lfloor log_b(n) \rfloor} \leq n$ donc $b^{\lfloor log_b(n) \rfloor + 1} \leq bn$).

Ex 8

Soit $n \in \mathbb{N}$, on a $2^{\lfloor log_2(n) \rfloor} \leq n < 2^{\lfloor log_2(n) \rfloor + 1}$

Ex 9
Montrons que $log_b(n) = O(log_2(n))$ pour toute base $b > 1$.

En utilisant l'exercice 4, on a $log_b(n) = log_b(2) \cdot log_2(n)$, or $log_b(2)$ est une constante.
Avec un seuil $n_0 = 1$ et une constante $c = log_b(2) > 0$, on a $\forall n \geq n_0, 0 \leq log_b(n) \leq c \cdot log_2(n)$.

Ex 10

Montrons que toute constante $c > 0$, et pour n'importe quelle base du logarithme $log(n)$, il existe des constantes $a, b > 0$ telles que : $a * log(n) \leq log(cn) \leq b * log(n)$ pour n suffisamment grand.

On a $log(n^a) \leq log(cn) \leq log(n^b)$ donc par croissance du log : $n^a \leq cn \leq n^b$, on peut choisir $a = \lfloor log_n(cn) \rfloor$ et $b = \lfloor log_n(cn) \rfloor + 1$.

Ex 11

Constant $(O(1))$:
logarithmique $(O(log(n)))$:
linéaire $(O(n))$: $log(log(n)) + 2n$
sous-linéaire $(O(n \cdot log(n))$: $3log(n) + n \cdot log(n), log(log(n)) + 2n$
quadratique $(O(n^2))$: $2n^2, 3log(n) + n \cdot log(n), n^{1.34}, log(n) + n^2, log(log(n)) + 2n$
polynomial $(O(n^k)), k > 2$: $n^3, 2n^2, n + 3n^4, 3log(n) + n \cdot log(n), n^2 \cdot log(n), n^{1.34}, n^{2.6}, log(n) + n^2, log(log(n)) + 2n$

On remarque que $O(1) \subset O(log(n)) \subset O(n) \subset O(n \cdot log(n)) \subset O(n^2) \subset O(n^k)$ d'après la définition de $O(g(n))$.

Ex 12

Algo $A()$:

- $a = 10$ donc une première affectation
- $whilea > 0$ donc 11 tests (10 vrais et un faux)
- $a = a - 1$ donc 10 soustractions et 10 affectations (on calcul d'abord $a - 1$ puis on affecte la valeur à a)

Ce qui fait un total de 32 opérations élémentaires.

Algo $B()$:

- $i = 0, 1, ..., 10$ donc 11 affectations
- 11 tests pour le for et 10 additions (on calcul d'abord $i + 1$ puis on affecte la valeur à i)
- 10 appels à $A()$ donc 320 opérations

Ce qui fait un total de 352 opérations élémentaires.

Algo $B()$:

- $res = 1$ donc une première affectation
- $i = 1, ..., n + 1$ donc $n + 1$ affectations
- $n + 1$ tests pour le for et n additions (on calcul d'abord $i + 1$ puis on affecte la valeur à i)
- $a = i * a$ donc n multiplications et n affectations

Ce qui fait un total de $5n + 3$ opérations élémentaires.

Ex 13

$x = (0, 1, 1, 0, 1)$ soit $x = 0 * 2^0 + 1 * 2^1 + 1 * 2^2 + 0 * 2^3 + 1 * 2^4 = 2 + 4 + 16 = 22$.

Codons 1023 en base 2 : $1023 = 1024 - 1 = 2^{10} - 1 = \sum_{i=0}^{9} 2^i$ soit 1111111111 en base 2 (avec 10 chiffres !).

Ex 14

Si $x \in \{0,1\}^t$, alors $t = \lfloor log_2(n) \rfloor + 1$, ce qui correspond au nombre de chiffres nécessaire pour écrire n en base 2.

Ex 15
Pour tout entier n, il existe un unique entier k tel que $k \leq \lfloor log_d n \rfloor < k+1$ c'est-à-dire $d^k \leq n < d^{k+1}$. Le nombre de bits codant l'entier n en base d est $\lfloor log_d a \rfloor + 1$ en base $d \geq 2$.

L'espace mémoire (en bits) requit pour faire fonctionner l'algorithme *fact(n)* est la complexité soit en $O(n)$. Le nombre de bits codant l'entier n est $O(log_2 n) := t$, donc la taille requise en mémoire pour stocker l'entrée de l'algorithme est $O(2^t)$. L'espace mémoire pour faire fonctionner l'algorithme et sa complexité sont $\Omega(2^t)$, exponentiel par rapport à t, la taille de l'entrée (si on code l'entrée en binaire).

Ex 16

Les algos $A()$ et $B()$ sont en $\Theta(1)$, $fact(n)$ et $fibo_i t(n)$ sont en $\Theta(n)$.

Ex 17
Montrons que $fibo\_rec = \Omega(\phi^n)$.

On note F_i le i-ième terme de la suite de Fibonacci, soit $A(i)$ le nombre total d'additions effectués lors de l'algorithme *fibo_rec(i)*.

Montrons que $A(n) = F_{n+1} - 1$ par induction :
Pour $n = 0, 1$, on a $A(0) = A(1) = 0$.
Supposons que l'hypothèse soit vraie pour $A(n)$ et $A(n-1)$.
On a $A(n+1) = A(n) + A(n-1) + 1 = F_{n+1} - 1 + F_n - 1 + 1 = F_{n+2} - 1$.

On a $\forall n \in \mathbb{N}, F_n = \frac{\phi^n - (-\frac{1}{\phi})^n}{\sqrt{5}}$, soit $F_n + 1 \geq \frac{\phi^n}{\sqrt{5}}$. De plus, $A(n) \geq F_n + 1$ pour $n \geq 4$, donc $\forall n \in \mathbb{N}, n \geq 4, 0 \leq \frac{\phi^n}{\sqrt{5}} \leq A(n)$.

Ex 18

L'algo $A(n)$ calcul une approximation de $\sqrt{n}$, sa complexité est de $\Theta(1)$.
L'algo $F(n)$ calcul une approximation du n-ième terme de la suite de Fibonacci, sa complexité dépend de l'algo pour le calcul de la puissance de $phi**n$ et $phic**n$, on peut les calculer en $\Theta(log(n))$ (cf TP1).

Ex 19
L'algorithme *D(n,b)* écrit l'entier n en base $b \in \mathbb{N}$. Montrons que sa complexité est en $log_b(n)$.

Itération	Valeur de la variable n à la fin de l'itération
0	n
1	$\lfloor \frac{n}{b} \rfloor$
2	$\lfloor \frac{n}{b^2} \rfloor$
...	...
i	$\lfloor \frac{n}{b^i} \rfloor$
...	...
j	0

A la fin de l'itération $j-1$, la valeur de la variable n est $\lfloor \frac{n}{b^{j-1}} \rfloor \in [1; b-1]$ (> 0 car il y a une dernière itération après). On a

$$1 \leq \frac{n}{b^{j-1}}$$
$$b^{j-1} \leq n$$
$$log_b(b^{j-1}) \leq log_b(n)$$
$$j-1 \leq log_b(n)$$
$$j \leq log_b(n)+1$$

Ex 20

L'algo calculant le produit de deux matrices $n \times n$ est en $\Theta(n^3)$, chaque coefficient $c_{i,j}, i, j \in \{1, 2, ..., n\}$ est calculé par la somme $\sum_{k=1}^{n} a_{i,k} * b_{k,j}$ en temps linéaire, ce qu'il faut faire pour tous les n^2 coefficients.

Ex 21

Soit $a, b \in \mathbb{N}^*, b > 1$, si $\frac{a}{b} \leq \frac{a-1}{b-1}$ alors :

$$\frac{a}{b} \leq \frac{a-1}{b-1}$$
$$a(b-1) \leq (a-1)b$$
$$ab-a \leq ab-b$$
$$a \geq b$$

Ex 22

Soit $c > 0$, et $f(n) = 1 + c + c^2 + ... + c^n$,
1) $f(n) = \Theta(1)$ si $c < 1$

On remarque que $f(n)$ est une série géométrique de raison c, donc $sum_{i=0}^{n} c^i = \frac{1-c^{n+1}}{1-c}$.

On a $sum_{i=0}^{n} c^i = \frac{1-c^{n+1}}{1-c} < \frac{1}{1-c}$ donc on peut trouver une constante $b > 0$ tel que $1 \leq f(n) < \frac{1}{1-c} \leq b$.
i.e. pour le borne inf, on choisi la constante 1 et pour la borne sup, la constante b.

2) $f(n) = \Theta(n)$ si $c = 1$

Dans ca cas, $f(n) = n + 1$, donc on peut prendre la constante 1 pour la borne inf et 2 pour la borne sup :

$\exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq 1 * n \leq f(n) \leq 2 * n$.

3) $f(n) = \Theta(c^n)$ si $c > 1$

On a $f(n) = \frac{1-c^{n+1}}{1-c} = \frac{c^{n+1}-1}{c-1}$, or $\frac{c^{n+1}-1}{c-1} \geq \frac{c^{n+1}}{c} = c^n$ d'après l'exercice précédent, donc on peut prendre 1 comme constante pour la borne inf.
D'autre part, on a $\frac{c^{n+1}-1}{c-1} = c^n * \frac{c-\frac{1}{c^n}}{c-1} \leq c^n * \frac{c}{c-1}$ donc on peut prendre $\frac{c}{c-1}$ comme constante pour la borne inf.

$\exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq 1 * c^n \leq f(n) \leq \frac{c}{c-1} * c^n$.

Ex 23

Montrons que $g(n) = \Theta(f(n))$ ssi $f(n) = \Theta(g(n))$.

($=>$) On a $g(n) = \Theta(f(n))$, donc $\exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$. A partir de ce même seuil n_0, on a $0 \leq c_1 \cdot f(n) \leq g(n)$ soit $0 \leq f(n) \leq \frac{1}{c_1} \cdot g(n)$ et $0 \leq g(n) \leq c_2 \cdot f(n)$ soit $0 \leq \frac{1}{c_2} \cdot g(n) \leq f(n)$. Donc $f(n) = \Theta(g(n))$.
($<=$) On a $f(n) = \Theta(g(n))$, donc $\exists c_1, c_2 > 0, \exists n_1 \in \mathbb{N}, \forall n \geq n_1, 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$. A partir de ce même seuil n_1, on a $0 \leq c_1 \cdot g(n) \leq f(n)$ soit $0 \leq g(n) \leq \frac{1}{c_1} \cdot f(n)$ et $0 \leq f(n) \leq c_2 \cdot g(n)$ soit $0 \leq \frac{1}{c_2} \cdot f(n) \leq g(n)$. Donc $g(n) = \Theta(f(n))$.

2. Montrons que $\Theta(f(n)g(n)) = \Theta(f(n))\Theta(g(n)) = f(n)\Theta(g(n)) = \Theta(f(n))g(n)$.

(1) Montrons que $\Theta(f(n)g(n)) \subseteq f(n)\Theta(g(n))$. Soit $t(n) = \Theta(f(n)g(n))$, alors $\exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 \cdot f(n)g(n) \leq t(n) \leq c_2 \cdot f(n)g(n)$. Donc on a bien $t(n) = f(n)\Theta(g(n))$ en conservant le même n_0, c_1 et c_2. (Idem pour $\Theta(f(n))g(n)$.
(2) On a $f(n)\Theta(g(n)) \subseteq \Theta(f(n))\Theta(g(n))$ car $f(n) = \Theta(f(n))$ (il suffit de prendre des constantes égales à 1).
(3) Montrons que $\Theta(f(n))\Theta(g(n)) \subseteq \Theta(g(n)f(n))$. Soit $t(n) = \Theta(f(n))\Theta(g(n))$, alors $\exists c_1, c_2, c_3, c_4 > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 \cdot f(n) \cdot c_3 \cdot g(n) \leq t(n) \leq c_2 \cdot f(n) \cdot c_4 \cdot g(n)$. A partir du même seuil n_0, et avec des constantes $c_5 = c_1 \cdot c_3$ t $c_6 = c_2 \cdot c_4$ on a bien $\forall n \geq n_0, 0 \leq c_5 \cdot f(n) \cdot g(n) \leq t(n) \leq c_6 \cdot f(n) \cdot g(n)$.

3. Montrons que $\Theta(1) + \Theta(1) = \Theta(1)$

Soit $f_1(n), f_2(n) = \Theta(1)$, alors $\exists c_1, c_2 > 0, \exists n_1 \in \mathbb{N}, \forall n \geq n_1, 0 \leq c_1 \leq f_1(n) \leq c_2$ et $\exists c_3, c_4 > 0, \exists n_2 \in \mathbb{N}, \forall n \geq n_2, 0 \leq c_3 \leq f_2(n) \leq c_4$. On a $0 \leq c_1 + c_3 \leq f_1(n) + f_2(n) \leq c_2 + c_4, \forall n \geq max(n_1, n_2)$. En considérant le seuil $n_0 = max(n_1, n_2)$ et les constantes $c_5 = c_1 + c_3$ et $c_6 = c_2 + c_4$, on peut conclure que $\Theta(1) + \Theta(1) \subseteq \Theta(1)$.
Soit $f(n) = \Theta(1)$, alors $\exists c_1, c_2, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 \leq f(n) \leq c_2$. Alors on peut trouver deux fonctions f_1, f_2 et des constantes $c_3, c_4, c_5, c_6 > 0$ tel que $f(n) = f_1(n) + f_2(n)$, $c_1 = c_3 + c_5$ et $c_2 = c_4 + c_6$. Ainsi $\Theta(1) \subseteq \Theta(1) + \Theta(1)$

Montrons que $\sum_{i=1}^{n}\Theta(1)=\Theta(n)$.

Soit $f_1(n),...,f_n(n)=\Theta(1)$, alors $\exists c_1^i,c_2^i>0,\exists n_0^i\in\mathbb{N},\forall n\geq n_0^i,0\leq c_1^i\leq f_i(n)\leq c_2^i\forall i\in\{1,...,n\}$. Alors $\forall n\geq max\{n_0^i|i\in\{1,...,n\}\}$ (on prend le plus grand seuil, de sorte que toutes les inégalités soient vérifiés), on a $0\leq n\cdot c_1\leq\sum_{i=1}^{n}f_i(n)\leq n\cdot c_2$ avec $c_1=min\{c_1^i|i\in\{1,...,n\}\}$ et $c_2=max\{c_2^i|i\in\{1,...,n\}\}$. Donc $\sum_{i=1}^{n}\Theta(1)=\Theta(n)$.

Ex 24

On a $f(cn)=\sum_{i=0}^{p}c^i\cdot n^i\cdot(a_i+b_ilog(n)+b_ilog(c))$.
cas 1 : 0 < c < 1
Dans ce cas, $c^p<c^k,k\in\{0,...,p-1\}$ donc $c^p\sum_{i=0}^{p}n^i<\sum_{i=0}^{p}c^i\cdot n^i$.
De plus, comme $log(n)$ diverge vers $+\infty$ lorsque n tend vers $+\infty$, les termes constants (a_i et $b_ilog(c)$) deviennent négligeables par rapport à $b_ilog(n)$) lorsque n est suffisament grand, donc comparer $(a_i+b_ilog(n))$ et $(a_i+b_ilog(n)+b_ilog(c))$ à partir d'un certain seuil n_0 revient à comparer $b_ilog(n)$ avec $b_ilog(n)-\epsilon$, donc $\exists n_0\in\mathbb{N}$ tel que $\forall n\geq n_0,0\leq\frac{1}{2}(a_i+b_ilog(n))\leq a_i+b_ilog(n)+b_ilog(c)$.
A partir de ce même n_0, on peut conclure que $\forall n\geq n_0,0\leq\frac{c^p}{2}f(n)\leq f(cn)\leq f(n)$ donc $f(cn)=\Theta(f(n))$.
cas 2 : c > 1
Dans ce cas, $c^k<c^p,k\in\{0,...,p-1\}$ donc $\sum_{i=0}^{p}c^i\cdot n^i<c^p\sum_{i=0}^{p}n^i$.
De plus, comme $log(n)$ diverge vers $+\infty$ lorsque n tend vers $+\infty$, $\exists n_0\in\mathbb{N}$ tel que $\forall n\geq n_0,0\leq(a_i+b_ilog(n)+b_ilog(c)\leq 2(a_i+b_ilog(n))$.
A partir de ce même n_0, on peut en conclure que $\forall n\geq n_0,0\leq f(n)\leq f(cn)\leq 2c^pf(n)$ donc $f(cn)=\Theta(f(n))$.

Ex 25

($=>$) Si $g(n)=\Theta(f(n)),\forall n>0$ alors on a bien $g(c^t)=\Theta(f(c^t))$ pour $c>1$ et pour tout entier t.
($<=$) Si $g(c^t)=\Theta(f(c^t))$ pour $c>1$ et $t\in\mathbb{N}$, considérons $c^t\leq n<c^{t+1}$, alors on a $\exists c_1>0,\exists n_0\in\mathbb{N},\forall n>n_0,0\leq c_1\cdot f(c^t)\leq g(c^t)\leq g(n)$ car g est croissante. De plus d'après l'exercice 24, on sait que $f(cn)=\Theta(f(n))$, donc $\exists c_2,c_3>0,\exists n_0\in\mathbb{N},\forall n>n_0,0\leq g(n)<g(c^{t+1})\leq c_2\cdot f(c^{t+1})\leq c_2\cdot c_3\cdot f(c^t)$.
($g(n)<g(c^{t+1})$ par croissance de g, $g(c^{t+1})\leq c_2\cdot f(c^{t+1})$ car $g(c^{t+1})=\Theta(f(c^{t+1}))$ par hypothèse et $f(c^{t+1})\leq c_3\cdot f(c^t)$ par l'exercice 24). Donc on a bien $g(n)=\Theta(f(c^t))$.
De plus, on a également $n<c^{t+1}$ donc $\frac{n}{c}<c^t$. Comme la fonction f est croissante, on peut écrire $\forall n>0,0\leq f(\frac{1}{c}\cdot n)\leq f(c^t)\leq f(n)$ et d'après l'exercice 24, $\exists c_4>0,\exists n_0\in\mathbb{N},\forall n>n_0,0\leq c_4\cdot f(n)\leq f(\frac{1}{c}\cdot n)\leq f(c^t)\leq f(n)$ Donc $f(c^t)=\Theta(f(n))$, on peut donc conclure que $g(n)=\Theta(f(n))$.

Ex 26
Montrons que $log(n!)=\Theta(n\cdot log(n)$

Pour la borne supérieur, on remarque que $\forall n\in\mathbb{N},n!\leq n^n$. Donc $\forall n\in\mathbb{N}^*,0\leq log(n!)\leq log(n^n)=n\cdot log(n)$. On peut conclure que $log(n!)=O(n\cdot log(n))$.

Pour la borne inférieur, il faut voir que $\forall n\in\mathbb{N},n\geq 2,\frac{n}{2}^{\frac{n}{2}}\leq\prod_{i=n/2}^{n}i$ (il y a $\frac{n}{2}$ termes dans les

deux cas).
En rajoutant les termes manquants à droite, on a $\forall n \in \mathbb{N}, n \geq 2, \frac{n}{2}^{\frac{n}{2}} \leq \prod_{i=1}^{n} i = n!$ et donc $log(\frac{n}{2}^{\frac{n}{2}}) \leq log(n!)$ soit $\frac{n}{2}log(\frac{n}{2}) \leq log(n!)$. On peut montrer que $nlog(\frac{n}{2}) = \Omega(n \cdot log(n)$ à partir d'un certain seuil n_0, à partir de ce seuil n_0 et avec un coefficient $c = \frac{1}{2}$, on conclue que $log(n!) = \Omega(n \cdot log(n))$.

Ex 27

Montrons que $\sum_{i=1}^{n} \frac{1}{i} = \theta(log(n))$.

Dans un premier temps, on s'occupe de la borne supérieur, on applique l'indice :

$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n} \leq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^{\lfloor log_2(n) \rfloor}}$ (Note : $n = 2^{log_2(n)}$)

Ensuite on remarque que l'on peut regrouper tous les termes $\frac{1}{2^i}, i \in \{1, ..., \lfloor log_2(n) \rfloor\}$ et les sommer entre eux pour obtenir des "paquets" de 1 à chaque fois.

Exemple :

$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} \leq 1 + (\frac{1}{2} + \frac{1}{2}) + (\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}) + (\frac{1}{8}) = 1 + 1 + 1 + \frac{1}{8}$

Ce qui donne :

$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n} \leq 1 + \lceil log_2(n) \rceil$

On peut donc trouver un seuil n_0 tel que $\exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq \sum_{i=1}^{n} \frac{1}{i} \leq 2 \cdot log_2(n)$ et donc $\sum_{i=1}^{n} \frac{1}{i} = O(log(n))$.

Dans un second temps, on traite la borne inférieur :

$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + ... + \frac{1}{2^{\lceil log_2(n) \rceil}} \leq 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n}$

De la même manière que dans le premier cas, on va pouvoir regrouper les termes en $\frac{1}{2^i}, i \in \{1, ..., \lceil log_2(n) \rceil\}$ entre eux pour obtenir des "paquets" de $\frac{1}{2}$.

Exemple :

$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} \geq 1 + (\frac{1}{2}) + (\frac{1}{4} + (\frac{1}{4}) + (\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}) = 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2}$

Et donc : $1 + \frac{1}{2} \cdot \lfloor log_2(n) \rfloor \leq 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n}$
Donc on va pouvoir trouver un seuil n_1 tel que $\exists n_1 \in \mathbb{N}, \forall n \geq n_1, 0 \leq \frac{1}{2} \cdot log_2(n) \leq \sum_{i=1}^{n} \frac{1}{i}$ et donc $\sum_{i=1}^{n} \frac{1}{i} = \Omega(log(n))$.

Ex 28
Pour $n, p \in \mathbb{N}, p \leq n$ l'algo *C(n,p)* retourne le coefficient binomial $C_n^p = \frac{n!}{p!(n-p)!}$. On peut le montrer par induction : Initialisation : $C(0, 0)$ retourne 1 et $C_0^0 = 1$. Supposons que $C(n-1, p)$ retourne la valeur $C_{n-1}^p, \forall p \leq n-1$.

On a $C(n,p) = C(n-1,p-1) + C(n-1,p) = \frac{(n-1)!}{(p-1)!(n-p)!} + \frac{(n-1!)}{p!(n-1-p)!} = \frac{p(n-1)!}{p!(n-p)!} + \frac{(n-1)!(n-p)}{p!(n-p)!} = \frac{(n-1)![p+(n-p)]}{p!(n-p)!} = \frac{n!}{p!(n-p)!}$.

Pour la complexité de $C(n,p)$, on remarque que chaque appel récursif se conlu par un renvoie de 1, donc la valeur C_n^p est calculé à partir d'une somme de 1. L'algorithme effectue alors $C_n^p - 1$ additions (peut être montré par induction comme ci-dessus). Or $C_n^p \geq (\frac{n}{p})^p$ donc l'algorithme $C(n,p)$ est exponentiel.

Pour montrer que $C_n^p \geq (\frac{n}{p})^p$, on sait que $\frac{a}{b} \leq \frac{a-1}{b-1}$ ssi $b \leq a$ (ex 17), donc $\frac{n}{p} \leq \frac{n-1}{p-1} \leq \frac{n-2}{p-2} \leq ... \leq \frac{n-p+1}{1}$ or $C_n^p = \frac{n}{p} \cdot \frac{n-1}{p-1} \cdot \frac{n-2}{p-2} \cdot ... \cdot \frac{n-p+1}{1} \geq (\frac{n}{p})^p$.

Ex 29

En remarquant que $2u_i \leq u_{i+2}$ où u_i correspond au i-ième terme de la suite de Fibonacci. On se retrouve avec une suite géométrique de raison $\sqrt{2}$ (étant donné que l'on a un pas de 2). Donc $u_n \geq \sqrt{2}u_{n-1} \geq 2u_{n-2} \geq ... \geq \sqrt{2}^{n-1}u_1$ or $\sqrt{2} \geq 1.41$ donc l'algo de l'exercice 17 est en $\Omega(1.41^n)$.

Ex 30

L'algorithme *Euclide(a,b)* renvoie le pgcd de a et $b \in \mathbb{N}$.

On peut supposer que $pgcd(a,b) = 1$, le nombre de divisions à effectuer étant le même pour $Euclide(a,b)$ et $Euclide(\frac{a}{d}, \frac{b}{d})$ où $d = pgcd(a,b)$ (il suffit de multiplier par d les valeurs successivent de a et b dans $Euclide(\frac{a}{d}, \frac{b}{d})$). Lors du déroulement de l'algorithme, b prend successivement les valeurs $b = b_n, b_{n-1}, ..., b_3, b_2, 1$.
On a $b_{i+2} \geq b_{i+1} + b_i$, pour le voir on a $b_{i+1} = a_{i+2}mod(b_{i+2})$ donc $b_{i+1} \in [1; b_{i+2} - 1]$ et $b_i = b_{i+2}mod(b_{i+1})$. Soit $b_{i+2} - k, k \in [1; b_{i+2} - 1]$ la valeur prise par b_{i+1} alors $b_i \leq k$ et $b_{i+2} \geq b_{i+1} + b_i$.
On a $b = b_n \geq F_n$ où F_i est le i-ème élément de la suite de Fibonacci, or $F_n = \Omega(\phi^n)$, donc $b \geq \phi^n$ et $log(b) \geq n$.
En prenant le même seuil n_0 que $F_n = \Omega(\phi^n)$ et c=1, on a $\forall n \geq n_0, 0 \leq n \leq log(b)$.

Ex 31

```
def max(T):
    if len(T)>0:
        max = T[0]
        for i in range(1,len(T)):
            if max < T[i] :
                max = T[i]
        return max
```

L'algo *tri den* est en $\Theta(n+m)$ où n est la taille du tableau A donné en paramètre et m est le plus grand élément du tableau A. La complexité de l'algo dépend fortement du max de A, notons que m peut avoir une valeur exponentielle par rapport à n, (par exemple si $m = 2^n$), la taille des données en paramètre étant de n (la taille du tableau, indépendamment des éléments de celui-ci) cet algo peut être exponentielle (par exemple si $m = 2^n$).

L'algo *tri den* est en $\Theta(n)$ si le plus grand élément du tableau A (m) est en $\Theta(n)$.

Ex 32

L'algorithme $bincount(n)$ fait au moins $n-1$ affectations de $j=0$ donc il est clairement en $\Omega(n)$. En supposant que la boucle *while* parcours tout le tableau *tab* à chaque passage dans la boucle for, le nombre d'opérations est de l'ordre de $n \cdot log(n)$ donc l'algo est en $O(n^2)$.

Itération boucle for	Valeur dans tab à la fin de l'itération
0	000
1	100
2	010
3	110
4	001
5	101
6	011
7	111

En affinant la recherche, on constate que la première case du tableau *tab* varie à chaque passage dans la boucle for, plus généralement la case d'indice $i, i \in [0; t-1]$ est modifiée tous les 2^i tours. Donc on a $n + \frac{n}{2} + \frac{n}{4} + ... + 1 = 2^t + 2^{t-1} + 2^{t-2} + ... + 2^0 = 2^{t+1} - 1$ donc l'algo est en $O(2^{t+1} = 2n)$.

Ex 33

Pour l'algorithme de tri par insertion, le meilleur des cas est quand le tableau donné en entré est déjà trié dans l'ordre croissant. Dans ce cas, la condition de la boucle *while* n'est jamais vérifié et la complexité est en $\Theta(n)$ où n est le nombre d'éléments dans le tableau.

Dans le pire des cas, le tableau en entré est trié par ordre décroissant, et donc à chaque itération j de la boucle *for*, dans la boucle *while* ; i prend successivement les valeurs $j-1, j-2, ..., -1$. La complexité est donc en $\Theta(n^2)$.

Dans le cas moyen, on considère que à chaque itération $j \in \{1, ..., n-1\}$ de la boucle *for*, dans la boucle *while* ; i prend successivement les valeurs $j-1, j-2, ..., j/2$ (donc $\frac{j}{2}$ itérations de la boucle *while*).
Le nombre d'itérations de la boucle *while* lors de l'algorithme en entier est donc :

$$\begin{aligned}\sum_{j=1}^{n-1} \frac{j}{2} &= \frac{1}{2} \cdot \sum_{j=1}^{n-1} j \\ &= \frac{1}{2} \cdot \frac{n \cdot (n-1)}{2} \\ &= \frac{n \cdot (n-1)}{4}\end{aligned}$$

Le nombres d'opérations élémentaires étant constant lors d'une itération de la boucle *while*, on peut conclure que dans le cas moyen, l'algorithme du tri par insertion est de complexité $\Theta(n^2)$.

Ex 34

```
# fonction pour calculer un factoriel
def facto_it(n):
    r = 1
    for i in range(2,n+1):
        r *= i
    return r

# binom via la formule classique
def binom1(n,p):
    return facto_it(n)/(facto_it(p)*facto_it(n-p))

# binom ameliore en remarquant que C^p_n = (n*(n-1)*...*(n-p+1))/p!
def binom2(n,p):
    if (p > n - p):
        p = n - p
    num = 1
    for i in range(n-p+1,n+1):
        num *= i
    return num/facto_it(p)

# solution avec une seule boucle
def binom3(n,p):
    if (p > n-p):
        p = n - p
    binom = 1
    for i in range(1,p+1)
        binom = (binom * (n-p+i)) / i
    return binom

# en iteratif avec le triangle de Pascal

def BinomialIt(n, p):
    if ( (n-p) < p ) :
        p=n-p
    # Allocation tri1 and tri2
    tri1 = [[0] * (n+1) for _ in range(p+1)]
    tri2 = [[0] * (n+1) for _ in range(p+1)]

    # Construction triangle de Pascal1
    for i in range(p+1):
        tri1[i][i] = 1
        tri1[i][0] = 1
    for i in range(2,p+1):
        for j in range(1,i):
```

```
                tri1[i][j] = tri1[i-1][j-1] + tri1[i-1][j]
        # Intitialisation tri2
        print(tri1)
        if (n-2*p-1 == -1) :# n is even and p=n/2
            for j in range(p,-1,-1):
                tri2[p][j]=tri1[p][p-j]
        if (n-2*p-1 == 0):
            tri2[p][p]=1
            for j in range(p-1,-1,-1):
                tri2[p][j] = tri1[p][p-j-1] + tri1[p][p-j]
        if (n-2*p-1 >= 1):
            rect = [[0] * (p+1) for _ in range(n-2*p-1)]
            for i in range(n-2*p-1):
                rect[i][0]=1
            for j in range(1,p+1):
                rect[0][j] = tri1[p][j-1] + tri1[p][j]
            for i in range(1,n-2*p-1):
                for j in range(1,p+1):
                    rect[i][j] = rect[i-1][j-1] + rect[i-1][j]
            tri2[p][p]=1;
            for j in range(p-1,-1,-1):
                tri2[p][j] = rect[n-2*p-2][p-j-1] + rect[n-2*p-2][p-j]

        # Construction tri2
        for i in range(p-1,-1,-1):
            for j in range(i,-1,-1):
                tri2[i][j] = tri2[i+1][j+1] + tri2[i+1][j]
        return tri2[0][0]

    print(BinomialIt(8,2))
```

Chapitre 2 - Divide and conquer

Rappels de cours $T(n) = aT(\frac{n}{b} + \Theta(n^c)$ *est l'équation satisfaite par le temps d'exécution* $T(n)$ *d'un algorithme* ***récursif*** *de type* ***diviser pour régner****, avec* $T(1) = O(1)$.
Où le temps d'exécution $T(n)$ *est le nombre opérations élémentaires en fonction d'un paramètre* $n = \Theta(t)$ *avec t repréesentant la taille des donnéees. Un tel algorithme divise le problème d'origine de taille n en a sous-problèmes de taille* $\frac{n}{b}$*, et* $\Theta(n^c)$ *est le temps néecessaire à la division du problème et 'à la reconstitution de sa solution à partir des solutions des sous-problèmes.*

En utilisant le master theorem, on peut déterminer des bornes asymptotiques exactes dans trois cas (cf ex 47) :

1. *si* $c > log_b(a)$*, alors* $T(n) = \Theta(n^c)$
2. *si* $c = log_b(a)$*, alors* $T(n) = \Theta(n^c log(n))$
3. *si* $c < log_b(a)$*, alors* $T(n) = \Theta(n^{log_b(a)})$

Ex 35

$T(n) = 2T(\frac{n}{2}) + \Theta(n) \Longleftrightarrow T(2^t) = 2T(2^{t-1}) + \Theta(2^t)$

$$\begin{aligned}
T(2^t) &= 2T(2^{t-1}) + \Theta(2^t) \\
&= 2 \times (2T(2^{t-2}) + \Theta(2^{t-1})) + \Theta(2^t) \\
&= \cdots \\
&= \sum_{i=0}^{t} 2^i \cdot \Theta(2^{t-i}) \\
&= \sum_{i=0}^{t} 2^i \cdot \Theta(\frac{2^t}{2^i}) \\
&= \sum_{i=0}^{t} \Theta(2^t) \\
&= \Theta(2^t) \sum_{i=0}^{t} 1 \\
&= \Theta(2^t)(t+1) \\
&= \Theta(2^t \cdot (t+1))
\end{aligned}$$

Or $2^t \cdot (t+1) = n \cdot (log_2(n) + 1)$, donc $T(n) = \Theta(n \cdot \log(n))$.

Ex 36

$T(n) = T(\frac{n}{2}) + \Theta(1) \Longleftrightarrow T(2^t) = T(2^{t-1}) + \Theta(1)$

$$\begin{aligned} T(2^t) &= T(2^{t-1}) + \Theta(1) \\ &= T(2^{t-2}) + \Theta(1) + \Theta(1) \\ &= \cdots \\ &= \sum_{i=0}^{t} \Theta(1) \\ &= \Theta(log_2(n) + 1) \quad (cf\ ex\ 23) \end{aligned}$$

Donc $T(n) = \Theta(log_2(n))$.

Ex 37

Montrons que $T(b^t) = \Theta(\sum_{i=0}^{t} a^i \cdot (b^{t-i})^c)$:

$$\begin{aligned} T(b^t) &= aT(b^{t-1}) + \Theta((b^t)^c) \\ &= a \times (aT(b^{t-2}) + \Theta((b^{t-1}))^c) + \Theta((b^t)^c) \\ &= \cdots \\ &= a^t \cdot \Theta(1^c) + a^{t-1} \cdot \Theta(b^c) + ... + a^2 \cdot \Theta((b^{t-2})^c) + a \cdot \Theta(((b^{t-1}))^c) + \Theta((b^t)^c) \\ &= \sum_{i=0}^{t} a^i \cdot \Theta((b^{t-i})^c) \\ &= \Theta(\sum_{i=0}^{t} a^i \cdot (b^{t-i})^c) \end{aligned}$$

Ex 38

D'après l'exercice précédent on a $T(n) = \Theta(\sum_{i=0}^{log_b(n)} a^i \cdot (b^{log_b(n)-i})^c) = \Theta(\sum_{i=0}^{log_b(n)} a^i \cdot (\frac{n}{b^i})^c)$.

Si $c = log_b(a)$:

$$\begin{aligned} T(n) &= \Theta(\sum_{i=0}^{log_b(n)} a^i \cdot (\frac{n}{b^i})^{log_b(a)})) \\ &= \Theta(n^{log_b(a)} \cdot \sum_{i=0}^{log_b(n)} (\frac{a}{b^{log_b(a)}})^i) \\ &= \Theta(n^{log_b(a)} \cdot \sum_{i=0}^{log_b(n)} (1)^i) \\ &= \Theta(n^{log_b(a)} \cdot (log_b(n) + 1)) \end{aligned}$$

Donc $T(n) = \Theta(n^{log_b(a)} \cdot log_b(n))$

Ex 40

$x \cdot y = (10^{\frac{n}{2}} a + b) \cdot (10^{\frac{n}{2}} c + d) = 10^n a \cdot c + 10^{\frac{n}{2}} (a \cdot d + b \cdot c) + b \cdot d.$

Les quatres nouveaux produits peuvent être calculés de manière récursive en itérant plusieurs fois la formule ci-dessus pour se retrouver avec des multiplications de nombre de plus en plus petit, jusqu'Ã avoir des chiffres. L'équation de récurrence est donc $T(n) = 4T(\frac{n}{2}) + \Theta(n)$.

$$\begin{aligned}
T(n) &= 4T(\frac{n}{2}) + \Theta(n) \\
&= 4 \times (4T(\frac{n}{4}) + \Theta(\frac{n}{2})) + \Theta(n) \\
&= \cdots \\
&= \sum_{i=0}^{log_2(n)} 4^i \cdot \Theta(\frac{n}{2^i}) \\
&= \Theta(\sum_{i=0}^{log_2(n)} 4^i \cdot \frac{n}{2^i}) \quad (cf\ ex\ 37) \\
&= \Theta(\sum_{i=0}^{log_2(n)} 2^i \cdot n) \\
&= \Theta(n \cdot \sum_{i=0}^{log_2(n)} 2^i) \\
&= \Theta(n \cdot (2^{log_2(n)+1} - 1)) \\
&= \Theta(n \cdot (2n - 1))
\end{aligned}$$

Donc $T(n) = \Theta(n^2)$.

Ex 41

Montrons que $\sum_{i=0}^{t} 2^i = \Theta(2^t), \forall t \in \mathbb{N}$.

Montrons par récurrence que $\sum_{i=0}^{t} 2^i = 2^{t+1} - 1, \forall t \in \mathbb{N}$
Initialisation : Pour $t = 0$, $2^0 = 1 = 2^1 - 1$.
Supposons que $\sum_{i=0}^{t} 2^i = 2^{t+1} - 1, t \in \mathbb{N}$, on a $\sum_{i=0}^{t+1} 2^i = \sum_{i=0}^{t} 2^i + 2^{t+1} = 2^{t+1} - 1 + 2^{t+1} = 2^{t+2} - 1$.

Donc $2^t \leq \sum_{i=0}^{t} 2^i \leq 2^{t+1}$, donc avec $c_1 = 1$, $c_2 = 2$ et $n_0 = 0$ on peut conclure que $\sum_{i=0}^{t} 2^i = \Theta(2^t), \forall t \in \mathbb{N}$.

Ex 43

On remarque que $a \cdot d + b \cdot c = (a + b)(c + d) - a \cdot c - b \cdot d$, et que l'on a plus que 3 produits intermédiaires à calculer au lieu de 4 : $(a + b)(c + d), a \cdot c$ et $b \cdot d$. La nouvelle équation est donc $T(n) = 3T(\frac{n}{2}) + \Theta(n)$.

```python
def karat(x,y):
    if len(str(x)) == 1 or len(str(y)) == 1:
        return x*y
    else:
        m = max(len(str(x)),len(str(y)))
        m2 = m // 2

        a = x // 10**(m2)
        b = x % 10**(m2)
        c = y // 10**(m2)
        d = y % 10**(m2)

        z0 = karat(b,d)
        z1 = karat((a+b),(c+d))
        z2 = karat(a,c)

        return (z2 * 10**(2*m2)) + ((z1 - z2 - z0) * 10**(m2)) + (z0)
```

Ex 44
Montrons que $2^t + 3 \cdot 2^{t-1} + ... + 3^i \cdot 2^{t-i} + ... + 3^t = \Theta(3^t)$.
On remarque que

$$\begin{aligned} 2^t + 3 \cdot 2^{t-1} + ... + 3^i \cdot 2^{t-i} + ... + 3^t &= 2^t \sum_{i=0}^{t} (\frac{3}{2})^i \\ &= 2^t \Theta((\frac{3}{2})^t) \\ &= \Theta(3^t) \end{aligned}$$

En utilisant le résultat de l'exercice 22.

Ex 45

On a l'équation de récurrence $T(n) = 3T(\frac{n}{2}) + \Theta(n)$. On note $n = 2^t$.

$$\begin{aligned} T(2^t) &= 3T(\frac{2^t}{2}) + \Theta(2^t) \\ &= 3 \cdot (3T(\frac{2^t}{2^2}) + \Theta(\frac{2^t}{2})) + \Theta(2^t) \\ &= ... \\ &= \sum_{i=0}^{t} 3^i \Theta(2^{t-i}) \end{aligned}$$

D'après l'exercice précédent, $\sum_{i=0}^{t} 3^i \Theta(2^{t-i}) = \Theta(3^t) = \Theta(3^{log_2(n)})$, car $t = log_2(n)$. De plus d'après l'exercice 5, on a $3^{log_2(n)} = n^{log_2(3)}$.
Remarque : l'algorithme de Karatsuba est en $\Theta(n^{log_2(3)})$, or $n^{log_2(3)} \simeq 1.59$ donc il est plus efficace

que l'algorithme de multiplication classique.

Ex 46

D'après l'exercice 37 on a $T(n) = \Theta(\sum_{i=0}^{log_b(n)} a^i \cdot (b^{log_b(n)-i})^c) = \Theta(\sum_{i=0}^{log_b(n)} a^i \cdot (\frac{n}{b^i})^c)$.

$$\begin{aligned} T(n) &= \Theta(\sum_{i=0}^{log_b(n)} a^i \cdot (\frac{n}{b^i})^c)) \\ &= \Theta(n^c \cdot \sum_{i=0}^{log_b(n)} (\frac{a}{b^c})^i) \\ &= \Theta(n^c) \cdot \sum_{i=0}^{log_b(n)} (\frac{a}{b^c})^i \end{aligned}$$

Ex 47

1. Cas 1 : $c > lob_b(a)$

Dans ce cas, la raison de la série $\frac{a}{b^c}$ est inférieur à 1, donc d'après l'exercice 22 : $\sum_{i=0}^{log_b(n)} (\frac{a}{b^c})^i = \Theta(1)$.

D'après l'exercice précédent, on a $T(n) = \Theta(n^c) \cdot \sum_{i=0}^{log_b(n)} (\frac{a}{b^c})^i = \Theta(n^c) \cdot \Theta(1) = \Theta(n^c)$

2. Cas 2 : $c = lob_b(a)$ (cf exercice 38)

La raison de la série $\frac{a}{b^c}$ est exactement 1, donc d'après l'exercice 22 : $\sum_{i=0}^{log_b(n)} (\frac{a}{b^c})^i = \Theta(log_b(n))$.

Donc d'après l'exercice précédent, on a $T(n) = \Theta(n^c) \cdot \sum_{i=0}^{log_b(n)} (\frac{a}{b^c})^i = \Theta(n^c) \cdot \Theta(log_b(n)) = \Theta(n^c \cdot log_b(n))$

3. Cas 3 : $c < lob_b(a)$

La raison de la série $\frac{a}{b^c}$ est supérieur à 1, donc d'après l'exercice 22 : $\sum_{i=0}^{log_b(n)} (\frac{a}{b^c})^i = \Theta((\frac{a}{b^c})^{log_b(n)})$.

On a :

$$\begin{aligned} T(n) = \Theta(n^c) \cdot \Theta((\frac{a}{b^c})^{log_b(n)}) &= \Theta(n^c) \cdot \Theta(\frac{a^{log_b(n)}}{(b^{log_b(n)})^c}) \\ &= \Theta(n^c) \cdot \Theta(\frac{a^{log_b(n)}}{n^c}) \\ &= \Theta(a^{log_b(n)}) \\ &= \Theta(n^{log_b(a)}) \end{aligned}$$

Exercice 48

Un algo permettant d'additionner deux matrices $A, B \in \mathbb{Q}^{n\times m}$ de complexité $\Theta(n \times m)$ est le suivant :

```
# Compexite : O(n*m)
def matrix_plus(A,B):
    n, m = A.shape
    C = np.zeros((n, m))
    for i in range(n):
        for j in range(m):
            C[i,j]=A[i,j]+B[i,j]
    return C
```

Exercice 49

```
def mult(A,B):
    n, n = A.shape
    if(n==1):
        return np.array([[A[0,0]*B[0,0]]])

    A11=A[0:n//2, 0:n//2]
    A12=A[0:n//2, n//2:n]
    A21=A[n//2:n, 0:n//2]
    A22=A[n//2:n, n//2:n]
    B11=B[0:n//2, 0:n//2]
    B12=B[0:n//2, n//2:n]
    B21=B[n//2:n, 0:n//2]
    B22=B[n//2:n, n//2:n]

    P1 = strassen_matrix_mult(A11, B11)
    P2 = strassen_matrix_mult(A12, B21)
    P3 = strassen_matrix_mult(A11, B12)
    P4 = strassen_matrix_mult(A12, B22)
    P5 = strassen_matrix_mult(A21, B11)
    P6 = strassen_matrix_mult(A22, B21)
    P7 = strassen_matrix_mult(A21, B12)
    P8 = strassen_matrix_mult(A22, B22)

    C = np.zeros((n, n))
    C[0:n//2, 0:n//2] = matrix_plus(P1,P2)
    C[0:n//2, n//2:n] = matrix_plus(P3,P4)
    C[n//2:n, 0:n//2] = matrix_plus(P5,P6)
    C[n//2:n, n//2:n] = matrix_plus(P7,P8)

    return C
```

L'équation de récurrence associée à cet algo est $T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$.

— On doit faire 8 différents calculs de produits de sous-matrices

— Chaque sous-matrice considérée est de taille $2^{p-1} \times 2^{p-1}$ avec $A, B \in \mathbb{Q}^{2^p \times 2^p}$
— La partie itérative de l'algo constitue essentiellement à sommer les résultats des produits de sous-matrices pour retrouver le produit souhaité, ces additions peuvent être effectuées en $\Theta(n^2)$ avec l'algo de l'exercice précédent.

En utilisant le théorème maître, on peut conclure qu'un tel algo a une complexité de $\Theta(n^3)$, ce qui est équivalent à l'algo classique de multiplication de matrices.

Exercice 52

```python
def strassen_matrix_mult(A,B):
    n, n = A.shape
    if(n==1):
        return np.array([[A[0,0]*B[0,0]]])

    A11=A[0:n//2, 0:n//2]
    A12=A[0:n//2, n//2:n]
    A21=A[n//2:n, 0:n//2]
    A22=A[n//2:n, n//2:n]
    B11=B[0:n//2, 0:n//2]
    B12=B[0:n//2, n//2:n]
    B21=B[n//2:n, 0:n//2]
    B22=B[n//2:n, n//2:n]

    P1 = strassen_matrix_mult(matrix_plus(A11,A22), matrix_plus(B11,B22))
    P2 = strassen_matrix_mult(matrix_plus(A21,A22), B11)
    P3 = strassen_matrix_mult(A11, matrix_moins(B12,B22))
    P4 = strassen_matrix_mult(A22, matrix_moins(B21,B11))
    P5 = strassen_matrix_mult(matrix_plus(A11,A12), B22)
    P6 = strassen_matrix_mult(matrix_moins(A21,A11), matrix_plus(B11,B12))
    P7 = strassen_matrix_mult(matrix_moins(A12,A22), matrix_plus(B21,B22))

    C = np.zeros((n, n))
    C[0:n//2, 0:n//2] = matrix_plus(matrix_moins(matrix_plus(P1,P4),P5),P7)
    C[0:n//2, n//2:n] = matrix_plus(P3,P5)
    C[n//2:n, 0:n//2] = matrix_plus(P2,P4)
    C[n//2:n, n//2:n] = matrix_plus(matrix_moins(matrix_plus(P1,P3),P2),P6)

    return C
```

L'équation de récurrence associée à ce nouvel algo est $T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$.

— On n'a plus que 7 calculs de produits de sous-matrices à effectuer
— Chaque sous-matrice considérée est de taille $2^{p-1} \times 2^{p-1}$ avec $A, B \in \mathbb{Q}^{2^p \times 2^p}$
— La partie itérative de l'algo constitue essentiellement à sommer/soustraire les résultats des produits de sous-matrices pour retrouver le produit souhaité, ces opérations peuvent être effectuées en $\Theta(n^2)$.

En utilisant le théorème maître, on peut conclure qu'un tel algo a une complexité de $\Theta(n^{log_2(7)})$, ce qui est meilleur que l'algo classique de multiplication de matrices.

Exercice 54

La suite de Fibonacci est défini par $F_n = F_{n-1} + F_{n-2}, \forall n \geq 2, F_0 = F_1 = 1$.

On remarque que :

$$\begin{aligned}(F_n, F_{n-1}) &= (F_{n-1}, F_{n-2}) * X \\ &= (F_{n-2}, F_{n-3}) * X^2 \\ &= ... \\ &= (F_1, F_0) * X^{n-1}\end{aligned}$$

avec $X = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
On peut déterminer la n-ième valeur de la suite de Fibonacci avec des matrices, il suffit de calculer X^{n-1}. Pour calculer cette dernière, on va diagonaliser X :

La matrice X est diagonalisable.
Le polynome caractéristique associé à X est :

$$det \begin{pmatrix} 1-x & 1 \\ 1 & -x \end{pmatrix} = (1-x)(-x) - 1 = x^2 - x - 1$$

Dont les valeurs propres sont $\phi = \frac{1+\sqrt{5}}{2}$ (le nombre d'or) et $\phi' = \frac{1-\sqrt{5}}{2}$.

Les vecteurs propres $(\phi, 1)$ et $(\phi', 1)$ de X sont associés à chaque valeur propre ϕ, ϕ'.

On a :

$$\begin{aligned}\begin{pmatrix} \phi & 1 \\ \phi' & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} &= \begin{pmatrix} \phi + 1 & \phi \\ \phi' + 1 & \phi' \end{pmatrix} \\ &= \begin{pmatrix} \phi^2 & \phi \\ \phi'^2 & \phi' \end{pmatrix} \\ &= \begin{pmatrix} \phi & 0 \\ 0 & \phi' \end{pmatrix} \begin{pmatrix} \phi & 1 \\ \phi' & 1 \end{pmatrix}\end{aligned}$$

Soit $P = \begin{pmatrix} \phi & 1 \\ \phi' & 1 \end{pmatrix}$, on a $P^{-1} = \frac{1}{\phi - \phi'} \begin{pmatrix} 1 & -1 \\ -\phi' & \phi \end{pmatrix}$ (or $\phi - \phi' = \sqrt{5}$).
On a vu que $PX = DP$ donc $X = P^{-1}DP$ avec $D = \begin{pmatrix} \phi & 0 \\ 0 & \phi' \end{pmatrix}$.

Ce qui donne $X^n = (P^{-1}DP)^n = P^{-1}D^nP$, D étant une matrice diagonale, $D^n = \begin{pmatrix} \phi^n & 0 \\ 0 & \phi'^n \end{pmatrix}$, le calcul de ces deux termes est donc effectués en $O(n)$. Il faut ensuite calculer le produit $P^{-1}D^nP$.

Le calcul de ϕ^n et de ϕ'^n reste compliqué étant donné que $\sqrt{5}$ est un irrationelle.

Exercice 55

L'algo suivant permet de calculer x^n, avec $x \in \mathbb{Q}$ et $n \in \mathbb{N}$:

```python
def power_smart_rec(x,n):
    if n == 0:
        return 1
    temp = power_smart_rec(x,n//2)
    if n % 2 == 0:
        return temp * temp
    else :
        return temp * temp * x
```

Cf TP1, cet algo possède comme équation de récurrence $T(n) = T(\frac{n}{2}) + \Theta(1)$ donc il est en $\Theta(log(n))$ si on considère la taille de x comme constant.

- On fait bien un appel récursif
- Lors de l'appel récursif, on divise par 2 la valeur de n
- La partie itérative constitue les multiplications de *temp* et x, supposé de taille constante donc la multiplication s'effectue en $\Theta(1)$

Remarque, en utilisant l'algo précédent pour calculer ϕ^n et ϕ'^n on conserve le souci de l'arrondi venant de l'irrationalité de $\sqrt{5}$. En modifiant l'algo *power smart rec* pour qu'il fonctionne sur des matrices (et non plus des entiers), on peut calculer X^n directement sans P, D ou P^{-1} avec $log(n)$ multiplications de matrice 2×2.

```python
# Algo pour calculer directement X^n

# multplication de deux matrices 2*2
def mult(A,B):
    M = [[0, 0], [0, 0]]
    M[0][0] = A[0][0]*B[0][0] + A[0][1]*B[1][0]
    M[0][1] = A[0][0]*B[0][1] + A[0][1]*B[1][1]
    M[1][0] = A[1][0]*B[0][0] + A[1][1]*B[1][0]
    M[1][1] = A[1][0]*B[0][1] + A[1][1]*B[1][1]
    return M

def power_matrix(n):
    if n == 1:
        return [[1, 1], [1, 0]]
    temp = power_matrix(n//2)
    if n % 2 == 0:
        return mult(temp,temp)
    else :
        return mult(mult(temp,temp),[[1, 1], [1, 0]])
```

Exercice 56

Si la taille de x est une variable, lors de la partie itérative de l'algo précédent, on peut utiliser l'algorithme de Karatsuba (ex 43), qui a une complexité de $O(n^{1.59})$, la partie itérative devient donc de complexité $O(n^{1.59})$.
En appliquant à nouveau le théorème maître, on obtient une complexité de $O(n^{1.59})$ si l'on considère la taille de x comme des variables.

Remarque, en utilisant l'algo précédent pour calculer ϕ^n et ϕ'^n on conserve le souci de l'arrondi venant de l'irrationalité de $\sqrt{5}$. En modifiant l'algo de l'exercice précédent pour qu'il fonctionne sur des matrices (et non plus des entiers), on peut calculer X^n directement sans P, D ou P^{-1} avec $log(n)$ multiplications de matrice 2×2. Les termes de X^n sont assez grands étant donné que $F_n = \Theta(\phi^n)$, donc F_n possède $O(n)$ chiffres. Or à chaque itération i de l'algo, on effectue un nombre constant de multiplications de nombre à $\frac{n}{2^i}$ chiffres, donc :

$$\sum_{i=0}^{log_2(n)} (\frac{n}{2^i})^{1.58} = n^{1.58} \cdot \sum_{i=0}^{log_2(n)} (\frac{1}{2^{1.58}})^i = O(n^{1.58})$$

```python
# Algo pour calculer directement X^n

# on utilise l'algo de karatsuba (ex 43) pour calculer le produits de
# deux entiers dont les tailles sont des variables
def mult(A,B):
    M = [[0, 0], [0, 0]]
    M[0][0] = karat(A[0][0],B[0][0]) + karat(A[0][1],B[1][0])
    M[0][1] = karat(A[0][0],B[0][1]) + karat(A[0][1],B[1][1])
    M[1][0] = karat(A[1][0],B[0][0]) + karat(A[1][1],B[1][0])
    M[1][1] = karat(A[1][0],B[0][1]) + karat(A[1][1],B[1][1])
    return M

def power_matrix(n):
    if n == 1:
        return [[1, 1], [1, 0]]
    temp = power_matrix(n//2)
    if n % 2 == 0:
        return mult(temp,temp)
    else :
        return mult(mult(temp,temp),[[1, 1], [1, 0]])
```

Chapitre 3 - Structures de données arborescentes

Rappels de cours *Soit T un ensemble fini contenant un élément r. Une fonction $p : T\backslash\{r\} \to T$ est père si pour tout $x \in T$, il existe un entier k tel que $p^k(x) = r$; où l'on note $p^0(x) = x$ et $p^{k+1}(x) = p(p^(x))$.*

Un arbre est un ensemble fini T muni d'une racine $r \in T$ et d'une fonction père p. Les feuilles de l'arbre (T, r, p) sont les $x \in T$ tels que si $p^k(y) = x$, alors $x = y$ (et $k = 0$). Les éléments de T sont également appelés sommets de T.

La hauteur de $x \in T$ est l'entier maximum $h(x)$ tel que $p^{h(x)}(y) = x$ pour un $y \in T$, en particulier, $h(r)$ est aussi la hauteur de T.

Un arbre est binaire si chaque sommet a au-plus deux fils.

Une permutation est une bijection $\sigma_n : [n] \Longleftrightarrow [n]$.

Un algorithme de tri est comparatif s'il détermine la permutation rangeant les entiers dans l'ordre en effectuant un certain nombre $t \leq (n^2 + n)/2$ de comparaisons $(x < y)$? entre deux entiers x, y du tableau.

Soit un ensemble A de couples (i, j), avec $i, j \in [n]$. Un élément $x \in [n]$ est connecté à un élément $r \in [n]$ si il existe une suite de couples $(x = i_0, i_1), (i_1, i_2), ..., (i_{k1}, i_k = r)$, appelée chemin de x à r.

Un réseau G est une liste de listes de couples : la liste principale contient une liste pour $x \in [n]$ qui contient les couples $(y, long)$ où y est un sommet que l'on peut atteindre depuis x par une liaison de longueur $long$.

Ex 57

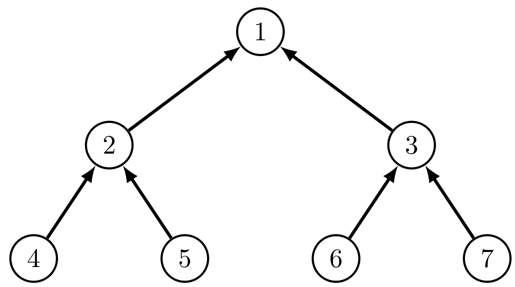
Soit $T = [n] = \{1, 2, 3, ..., n\}$ et $r = 1$, montrons que $p(x) = \lfloor \frac{x}{2} \rfloor$ est une fonction père.

Il suffit de trouver k tel que $\forall x \in T, \exists k \in \mathbb{N}, p^k(x) = 1$ pour $k = \lfloor log_2(x) \rfloor$.

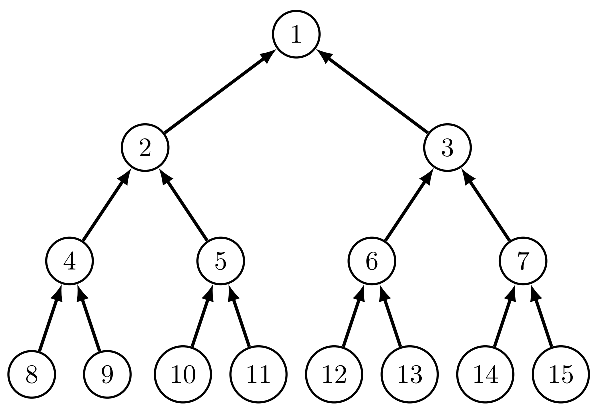
Ex 58

$(=>)$ Montrons qu'il est contradictoire que p soit une fonction père et qu'il existe un entier $k > 0$ tel que $p^k(x) = x$ pour $x \in T$. On a $x, p(x), p^2(x), ...p^{k-1}(x) \in T\backslash\{r\}$. Puisque $p^k(x) = x$ alors $p^i(x) \neq r \forall i \in \mathbb{N}$ (présence d'un cycle). Donc p n'est pas une fonction père.
$(<=)$ Montrons que si p n'est pas une fonction père alors il existe un entier $k > 0$ tel que $p^k(y) = y$ pour $y \in T$. $\exists x \in T\backslash\{r\}$ tel que $p^k(x) \in T\backslash\{r\}$ pour tout $k \in \{0, 1, 2, ..., |T| - 1\} = I$. Etant donné que $|I| > |T\backslash\{r\}|$, par le principe des tiroirs, $\exists i, j \in I, i < j$ tel que $p^i(x) = p^j(x)$, donc $p^k(y) = y$ avec $y = p^i(x)$ et $k = j - i$.

Ex 59



Les feuilles correspondent donc aux éléments 4,5,6 et 7.



Les feuilles correspondent donc aux éléments 8,9,10,11,12,13,14 et 15.

Ex 60

La hauteur de l'arbre $([n], 1, \lfloor \frac{x}{2} \rfloor)$ est $\lfloor log_2(n) \rfloor$. Le sommet étiquetté n est un des sommets les plus "éloignés" de la racine.

Ex 61

Les permutations de σ_3 sont :

$$(123)$$
$$(132)$$
$$(213)$$
$$(231)$$
$$(312)$$
$$(321)$$

Montrons que pour σ_n, il existe $n!$ permutations :

Démonstration en utilisant le principe de multiplications à une exprérience avec n étapes (car on a pas de répétitions) :

— étape 1 : on a n choix possibles

— étape 2 : on a $n-1$ choix possibles
— ...
— étape $n-1$: on a 2 choix possibles
— étape n : on a 1 choix possible

Ce qui nous donne $n \times (n-1) \times ... \times 2 \times 1 = n!$.

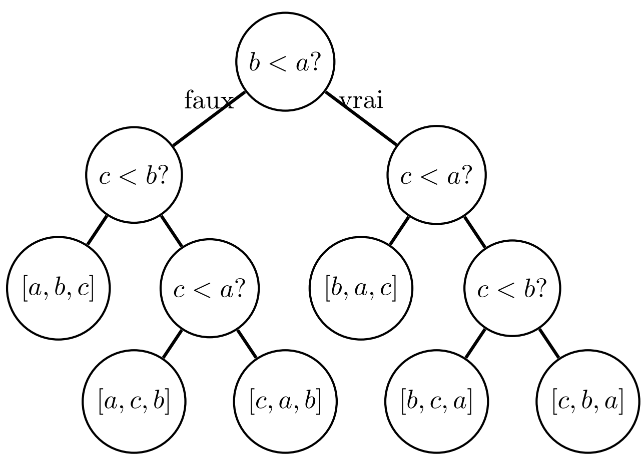
Ex 62

Faire un tri sur un ensemble de n entiers distincts consiste à trouver une permutation $(x_1, x_2, ..., x_n)$ tel que $x_i < x_{i+1}, \forall i \in [n-1]$.

Ex 63

L'exercice 31 n'effectue aucune comparaison entre les éléments d'entrée, ce n'est donc pas un algo comparatif.
L'algorithme du tri par insertion effectue dans le pire des cas $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ comparaison entre les éléments du tableau (de taille n), c'est donc un algo comparatif.
L'algorithme du tri fusion effectue de l'ordre de $n \cdot log(n)$ comparaisons donc c'est également un algo comparatif.

Ex 64



Ex 65

Considérons le tableau $[a, b, c, d]$ composés de 4 entiers distincts. Lors de l'algorithme du tri par insertion, dans le pire des cas (lorsque le tableau est trié dans l'ordre décroissant) on effectue 6 comparaisons (le résultat de la comparaison est toujours vrai) :

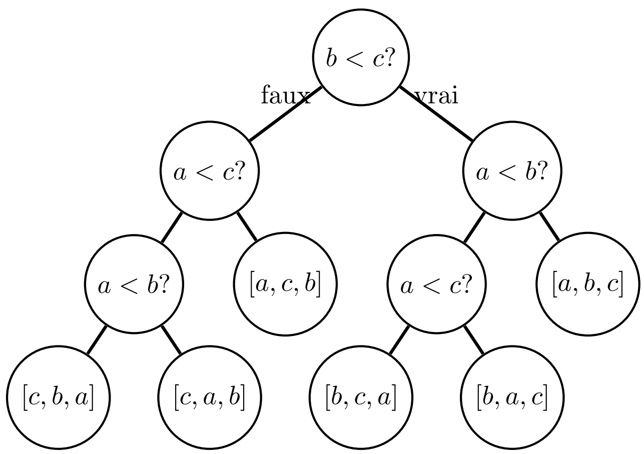
1. $a > b$?, après cette étape le tableau est $[b, a, c, d]$
2. $a > c$?
3. $b > c$?, après cette étape le tableau est $[c, b, a, d]$
4. $a > d$?
5. $b > d$?

6. $c > d?$, après cette étape le tableau est $[d, b, a, c]$

Ex 66

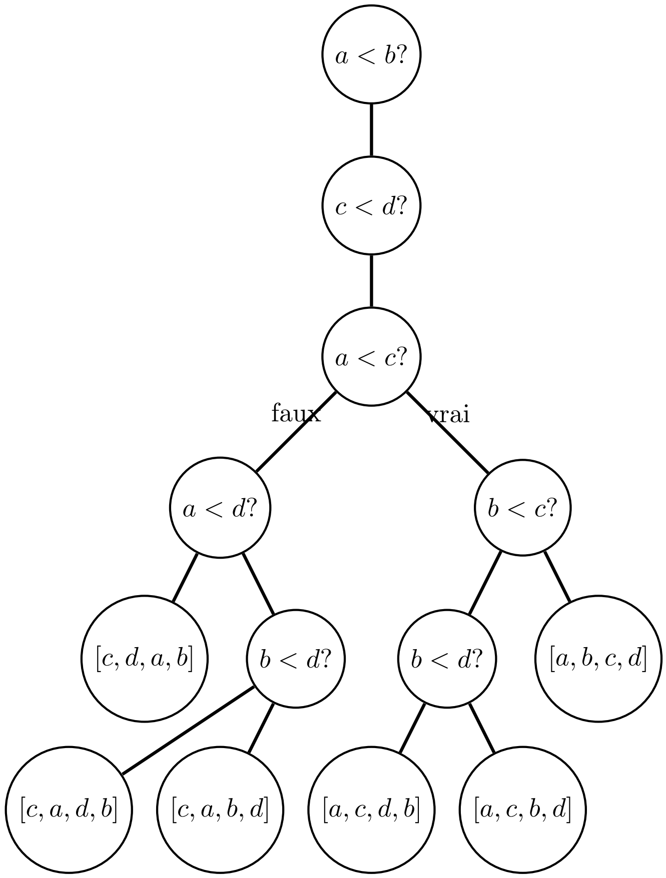
Soit T_n l'arbre des comparaisons de l'algo du tri par insertion d'un tableau de n entiers. La hauteur de T_n correspond au nombre maximum de comparaisons possibles lors de l'algo du tri par insertion. Or dans le pire des cas, on fait $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ comparaisons, donc la hauteur de T_n est égale à $\frac{n(n-1)}{2}$.

Ex 67



Ex 68

On considérera, sans perte de généralité, que a $<$ b et c $<$ d



Ex 69

Un arbre binaire T avec un hauteur de h peut avoir jusqu'à 2^h feuilles, dans ce cas toutes les feuilles (noeuds n'ayant aucun fils) sont à la même distance de la racine (c'est-à-dire h). Il s'agit d'un arbre dont tous les niveaux sont remplis, où tous les noeuds internes ont deux fils.

Ex 70

En représentant le déroulement de l'algo sous la forme d'un arbre, on détermine la complexité de l'algo en calculant la hauteur de l'arbre (correspondant au nombre d'itérations à effectuer dans le pire des cas).
Puisqu'il y a $n!$ permutations possibles pour un tableau de taille n, on aura exactement une feuille pour chacune de ces permutations dans notre arbre (chaque feuille correspond à une permutation différentes).
De plus, un arbre binaire de hauteur h a au-plus 2^h feuilles, et donc :

$$n! \leq 2^h$$

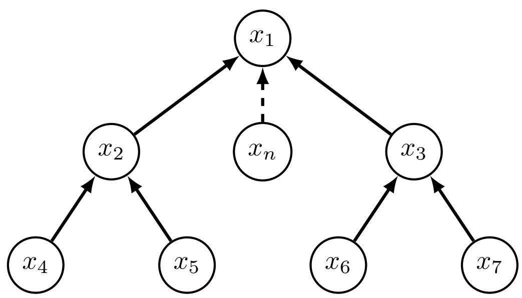
$$h \geq log(n!) = \Theta(n \cdot log(n)) \text{ d'après l'exercice 26}$$

Les algos de tri comparatif sont en $\Omega(n \cdot log(n))$.

Ex 71

$\Rightarrow$ Par induction sur n. Pour $n = 1$, $r = x_1$ n'a pas de fils, donc $\sum_{i=1}^{n} d_i = d_1 = 0 = n - 1$.
Pour $n \geq 2$, on peut supposer que x_n est une feuille de père x_1. Soit T' l'arbre obtenu à partir de T en supprimant x_n, où x_i a d'_i fils dans T' pour $i = 1, ..., n-1$. Alors, par induction, $\sum_{i=1}^{n-1} d'_i = n-2$ Dans T, x_1 a $d1 = d'_1 + 1$ fils, x_i a $d_i = d'_i$ fils pour $i = 2, ..., n-1$, et x_n a $d_n = 0$ fils. On a bien $\sum_{i=1}^{n} d_i = n - 1$.

$\Leftarrow$ Par induction sur n. Pour $n = 1$, $d_1 = 0$ est bien le nombre de fils de l'arbre T composé de l'unique sommet x_1. Pour $n \geq 2$, on peut supposer que $d1 \geq 1$ et $d_n = 0$. Soit $d'_1 = d_1 - 1$ et $d'_i = d_i$ pour $i = 1, ..., n-1$. Alors, $\sum_{i=1}^{n-1} d'_i = n - 2$, et donc par induction, il existe un arbre $T' = \{x_1, x_2, ..., x_{n-1}\}$ tel que x_i a d'_i fils. En ajoutant le sommet x_n de père x_1 à T' on obtient l'arbre T recherché.

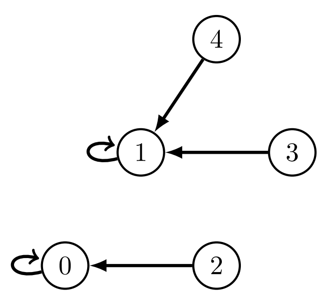


Ex 72

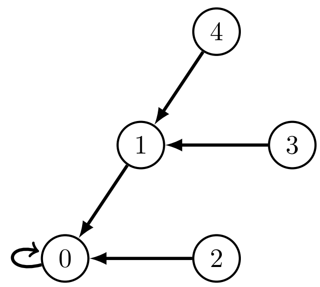
Étant donné un tableau $T = [x_1, x_2, ...x_n]$ où $0 \leq x_i \leq n-1, \forall i \in \{1, 2, ..., n\}$. Le graphe associé à T est construit de la manière suivante :

- le graphe possède n sommets, numéroté de 0 à $n - 1$.
- pour tous $i \in \{0, 1, ..., n-1\}$ on ajoute un arc (flèche) allant de i à $T[i]$.

Graphe associé au tableau $[0, 1, 0, 1, 1]$:



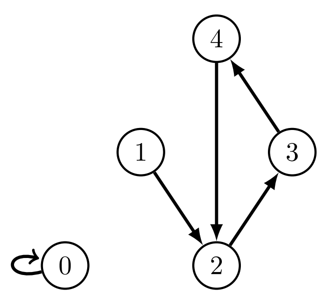
Graphe associé au tableau $[0, 0, 0, 1, 1]$:



Excécution de l'algo :

tab	check(tab)
$[0, 1, 0, 1, 1]$	0
$[0, 0, 0, 1, 1]$	1
$[0, 2, 0, 4, 1]$	1
$[2, 3, 0, 1, 1]$	0
$[0, 4, 0, 1, 3]$	0

Cet algo retourne 1 si tableau *arb* est une fonction père, 0 sinon. Si le tableau correspond bien à une fonction père, on peut le voir graphiquement ; dans ce cas le graphe associé est une arborescence, avec r l'unique racine (cas où $i = T[i]$), pour tout autre sommet v du graphe, il existe un chemin allant de v à r dans le graphe (remarque : une arborescence ne contient aucun cycle).
Cet algo peut mener à une boucle infinie dans certain cas, par exemple lorsque le graphe possède un cycle, par exemple : $[0, 2, 3, 4, 2]$



Ici, dans l'algo, lorsque l'on part du sommet 1 et que l'on applique plusieurs fois la fonction père, on tombe dans le cycle formé par les sommets 2,3 et 4 et les conditions de la deuxième boucle *while* restent toujours vérifiées, à l'infini...

Ex 73

On corrige l'algo précédant avec l'ajout d'un compteur ; on sait que dans le pire des cas on devrait faire $n - 1$ appels à la fonction père pour arriver à r, sinon cela veut dire qu'il y a un cycle et donc que ce n'est pas une fonction père.

```
def check2(tab):
    r=0
    while r<len(tab) and r != tab[r]:
        r=r+1
```

```
    if r == len(tab):
        return(0)
    for i in range(len(tab)):
        p=tab[i]
        j=0
        while p != r and p != i and j<len(tab):
            p=tab[p]
            j=j+1
        if p != r:
            return(0)
    return(1)
```

Ex 74

Les (A, r) tels que tout $x \in [n]$ est connecté à r correpondent aux graphes (avec des cycles potentiels) tels qu'il existe au moins un sous-graphe correspondant à une arborescence de racine r.

Ex 75

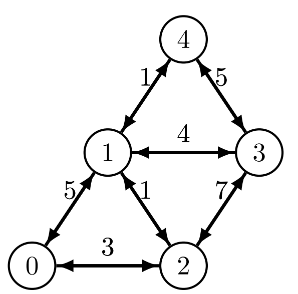
Excécution de l'algo :

tab	check(tab)
$[0, 2, 0, 1, 1]$	1
$[0, 0, 0, 1, 1]$	1
$[0, 2, 0, 4, 1]$	1
$[0, 2, 0, 0, 1]$	0

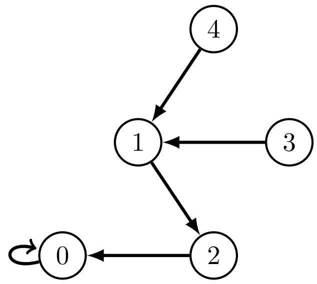
Graphe $G = [[(1, 5), (2, 3)], [(0, 5), (2, 1), (3, 4), (4, 1)], [(0, 3), (1, 1), (3, 7)], [(1, 4), (2, 7), (4, 5)], [(1, 1), (3, 5)]]$

- le nombre de sommet de G correspond à la taille du tableau associé, ici 5 numéroté de 0 à 4.
- dans $G[i], i \in \{0, 1, 2, 3, 4\}$ on a un tableau contenant des couples (k, l), pour chacun de ces couples on ajoute dans le graphe un arc (flèche) allant de i à k, en ajoutant un poids de l sur cette arête.

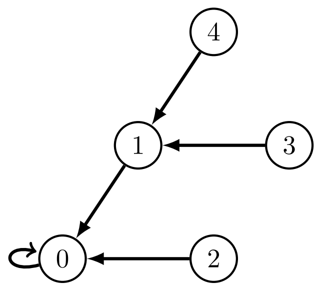
Ce qui donne le graphe suivant :



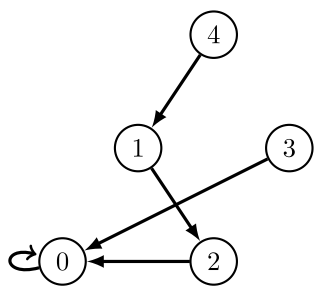
Graphe associé au tableau $[0, 2, 0, 1, 1]$:



Graphe associé au tableau $[0, 0, 0, 1, 1]$:



Graphe associé au tableau $[0, 2, 0, 0, 1]$:



Graph associé au tableau $[0, 2, 0, 1, 1]$:
L'algo retourne 1 si *arb* est un sous-graphe de G, 0 sinon. *arb* est un sous-graphe de G si on peut obtenir *arb* à partir de G en supprimant des arcs et/ou des sommets de celui-ci. L'algo ne considère pas les boucles (arc allant d'un sommet à lui-même).

L'idée de l'algo est de tester si tous les arcs de *arb* existent aussi dans G.

```
G = [[(1,5),(2,3)],[(0,5),(2,1),(3,4),(4,1)],[(0,3),(1,1),(3,7)],
[(1,4),(2,7),(4,5)],[(1,1),(3,5)]]

def checkG(G,arb):
    # boucle sur tous les arcs de G
    for i in range(len(arb)):
        # Si i n'a pas de boucle (le cas lorsque i est la racine)
        if i != arb[i]:
            test=0
            # parcours des arcs partant de i dans G
```

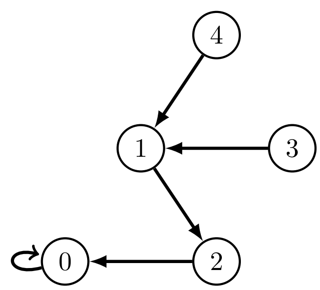
```
            for (x,long) in G[i]:
                # test si la destination des arcs sont les meme
                if x==arb[i]:
                    test=1
            if test==0:
                return(0)
    return(1)
```

Ex 76

Excécution de l'algo :

tab	check(tab)
$[0, 2, 0, 1, 1]$	9
$[0, 0, 0, 1, 1]$	13
$[0, 2, 0, 4, 1]$	10

Graphe associé au tableau $[0, 2, 0, 1, 1]$, de renvoi 9 :



L'algo retourne la somme des poids sur les arcs de arb, en récupérant le poids de chaque arc dans G.

L'idée de l'algo est d'aller chercher les poids dans G associé à chaque arc de arb et d'en faire la somme :

```
G = [[(1,5),(2,3)],[(0,5),(2,1),(3,4),(4,1)],[(0,3),(1,1),(3,7)],
[(1,4),(2,7),(4,5)],[(1,1),(3,5)]]

def cost(G,arb):
    # variable initialise a zero pour faire la future somme des poids
    c=0
    # boucle sur les arcs de arb
    for i in range(len(G)):
        # parcours des arcs partant de i
        for (x,long) in G[i]:
            # test si la destination des arcs est la meme
            if x==arb[i]:
                # ajout du poids de l'arc a la somme
                c= c + long
    return(c)
```

Ex 77

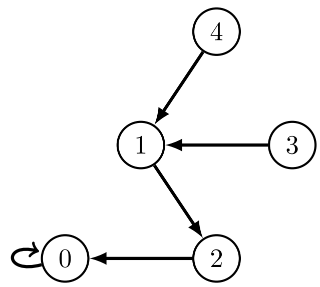
b.mincost vaut 9. L'algo énumère tous les tableaux $[(n-1) \times (n-1) \times ... \times (n-1)]$ (soit n^n tableaux), exemple pour $n = 3$:

$$[0,0,0]$$
$$[0,0,1]$$
$$[0,0,2]$$
$$[0,1,0]$$
$$[0,1,1]$$
$$[0,1,2]$$
$$[0,2,0]$$
$$[0,2,1]$$
$$[0,2,2]$$
$$[1,0,0]$$
$$[1,0,1]$$
$$[1,0,2]$$
$$[1,1,0]$$
$$[1,1,1]$$
$$[1,1,2]$$
$$[1,2,0]$$
$$[1,2,1]$$
$$[1,2,2]$$
$$[2,0,0]$$
$$[2,0,1]$$
$$[2,0,2]$$
$$[2,1,0]$$
$$[2,1,1]$$
$$[2,1,2]$$
$$[2,2,0]$$
$$[2,2,1]$$
$$[2,2,2]$$

Pour chacun des ces tableaux, teste s'il correspond à une arborescence ainsi qu'un sous-graphe de G, si c'est le cas il calcul le poids associé au tableau et conserve en mémoire le poids minimum.

Cet algo cherche une arborescence de poids min dans G.

Graphe associé au tableau $[0, 2, 0, 1, 1]$, correspondant à l'arborescence et sous-graphe de G de poids 9 renvoyé par l'algo :




```python
infini = 100
G = [[(1,5),(2,3)],[(0,5),(2,1),(3,4),(4,1)],[(0,3),(1,1),(3,7)],
[(1,4),(2,7),(4,5)],[(1,1),(3,5)]]

class BackTrack:
    def __init__(self , G):
        self.n=len(G)
        self.tab= self.n * [0]
        self.mincost=infini

    def enum(self,i):
        # boucles et appels recursifs pour construire les n*n tableaux possibles
        for j in range(self.n):
            self.tab[i]=j
            if i<self.n-1:
                self.enum(i+1)
            else:
                # test si le tableau construit est une arborescence et un
                # sous-graphe de G
                if check2(self.tab)==1 and checkG(G,self.tab)==1:
                    # calcul de la somme des poids des arcs
                    c=cost(G,self.tab)
                    # mise a jour de la somme minimum
                    if c<self.mincost:
                        self.mincost=c

b = BackTrack(G)
b.enum(0)
print(b.mincost)
```

Ex 78

L'algorithme de Dijkstra permet de déterminer le plus court chemin entre deux sommets u et v de G (suite d'arc $(i_0, i_1), (i_1, i_2), ..., (i_{n-2}, i_{n-1}), (i_{n-1}, i_n)$ avec $u = i_0$, $v = i_n$ et (i_j, i_{j-1}) un arc de G pour tout $j \in \{0, 1, ...n-1\}$).

Algo de Dijkstra en pseudo-code :

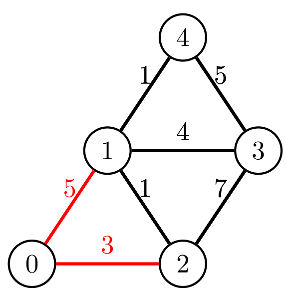
```
Entree : Un graphe G, un sommet r de G (source) et un sommet v de G (destination)
```

```
par = [r]
dist = infini pour tous sommet differents de r
dist[r] = 0
Tant que par ne contient pas v:
    Soit a le sommet dans par[0]
    Pour chaque sommet b hors de par tel qu'il existe une arete (a,b) dans G
        Si dist[b] > dist[a] + poids(a,b)
            dist[b] = dist[a] + poids(a,b)
    Choisir un sommet u hors de par de plus petite distance dist[u]
    Mettre u dans P
```

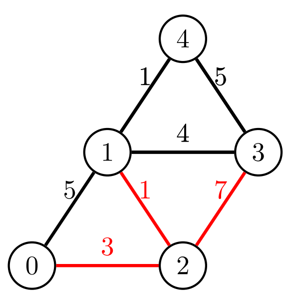
Déroulement de l'algo :

```
0 100 100 100 100
0
0 5 3 100 100
2 0
0 4 3 10 100
1 2 0
0 4 3 8 5
4 1 2 0
0 4 3 8 5
3 4 1 2 0
8
```

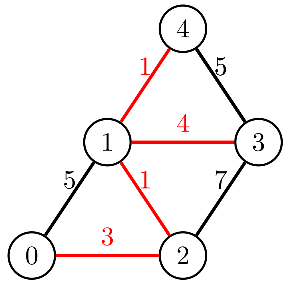
Déroulement étape par étape de l'algo :




```
dist = [0, 100, 100, 100, 100]
par = [0]
dist = [0, 5, 3, 100, 100]
par = [2, 0]
```



```
dist = [0, 4, 3, 10, 100]
par = [1, 2, 0]
```




```
dist = [0, 4, 3, 8, 5]
par = [4, 1, 2, 0]
dist = [0, 4, 3, 8, 5]
par = [3, 4, 1, 2, 0]
```

```python
infini = 100
G = [[(1,5),(2,3)],[(0,5),(2,1),(3,4),(4,1)],[(0,3),(1,1),(3,7)],
[(1,4),(2,7),(4,5)],[(1,1),(3,5)]]

def dijkstra(G,r,v):
    nb_som = len(G)
    # tableau avec les distances min allant de r a la position i (i=0,..,n-1)
    dist = nb_som * [infini]
    dist[r] = 0
    # tableau des sommets que l'on a explore
    par = [r]
    print(*dist)
    print(*par)
    # tant que l'on a pas explore le sommet v
    while par[0] != v:
        d_min = infini
        # on explore le sommet G[par[0]]
        for (x,long) in G[par[0]]:
            # si on a pas explore x
            if not (x in par):
                # calcul du minimum entre le plus court chemin en
                # memoire (dist[x]) et le chemin en explorant le sommet G[par[0]]
                d = min(dist[x],dist[par[0]]+long)
                dist[x] = d
                # recherche du prochain sommet a explorer
                # correspond au sommet i non explore tel que dist[i] soit min
                if d_min > d:
                    d_min = d
                    s = x
        # s futur sommet a explorer
```

```
            par = [s]+par
            print(*dist)
            print(*par)
        return dist[v]

dijkstra(G,0,3)
```

Ex bonus : Recherche dichotomique

Considérons l'algorithme suivant, soit v un entier et t un tableau d'entiers trié dans l'ordre croissant :

```
def dichotomie(t, v):
    a = 0
    b = len(t) - 1
    while a <= b:
        m = (a + b) // 2
        if t[m] == v:
            # on a trouv\'e v
            return True
        elif t[m] < v:
            a = m + 1
        else:
            b = m - 1
    # on a a > b
    return False
```

Calculons le nombre d'itérations i de l'algo *dichotomie* dans le pire des cas. Dans le pire des cas, le sous-tableau $t[a, b]$ ne possède qu'un seul élément. Soit n le nombre d'éléments dans t, on a :

Itération	nombre d'éléments dans $t[a, b]$ à la fin de l'itération
0	n
1	$\frac{n}{2}$
2	$\frac{n}{2^2}$
...	...
i	$\frac{n}{2^i}$
...	...
j	1

Cherchons le nombre d'itérations j :

$$1 \leq \frac{n}{2^j}$$
$$2^j \leq n$$
$$j \leq log(n)$$

Correction TP :

```python
# TP 1
import time
import math
import random

def facto_it(n):
    r = 1
    for i in range(2,n+1):
        r *= i
    return r

def facto_rec(n):
    if n==1:
        return 1
    return n*facto_rec(n-1)

def fibo_rec(n):
    if (n == 0):
        return 0
    if (n == 1):
        return 1
    return fibo_rec(n-1) + fibo_rec(n-2)

def fibo_it(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    x = 0
    y = 1
    for i in range(2,n+1):
        temp = x + y
        x = y
        y = temp
    return y

def fibo_rec_smart_aux(n):
    if n == 1:
        return 1, 0
    f_n_minus_1, f_n_minus_2 = fibo_rec_smart_aux(n-1)
    return f_n_minus_1 + f_n_minus_2, f_n_minus_1

def fibo_smart_rec(n):
    if n == 0:
        return 0
    f_n, _ = fibo_rec_smart_aux(n)
    return f_n
```

```

def power_it(a,b):
    res = 1
    for i in range(b):
        res *= a
    return res

def power_rec(a,b):
    if b == 0:
        return 1
    return a*power_rec(a,b-1)

def power_smart_rec(a,b):
    if b == 0:
        return 1
    temp = power_smart_rec(a,b//2)
    if n % 2 == 0:
        return temp * temp
    else :
        return temp * temp * a

# TP 2
def convert(t):
    (t,s) = divmod(t, 60)
    (t,mi) = divmod(t, 60)
    (t,h) =divmod(t,24)
    (years,days) = divmod(t,365)
    (mo,d) = divmod(days,31)
    return [years, mo, d, h,mi,s]


def printtab(tab):
    print str(tab[0]) + " annee, "+ str(tab[1])+ " mois "+str(tab[2])+
    " jours "+str(tab[3])+" heures "+str(tab[4])+" minutes "+str(tab[5])
    +" secondes"

n = 100

printtab(convert(math.log(n)))
printtab(convert(n))
printtab(convert(math.log(n)*n))
printtab(convert(n*n))
printtab(convert(n*n*n))
printtab(convert(2**n))
print("")


```

```python
def randomtab(n):
    t = []
    for i in range(n):
        t.append(random.randint(1,1000))
    return t

def insertion_sort(t):
    l = len(t)
    for i in range(l):
        j = i
        while j > 0 and t[j-1]>t[j]:
            t[j], t[j-1] = t[j-1], t[j]
            j -= 1
    return t

tab = randomtab(n)
print(tab)
print(insertion_sort(tab))

def buble_sort(tab):
    for i in range(len(tab)-1,-1,-1):
        for j in range(0,i):
            if tab[j+1]<tab[j] :
                tmp = tab[j+1]
                tab[j+1]=tab[j]
                tab[j]=tmp
    return tab

def fusion(tab1,tab2):
    if len(tab1)==0:
        return tab2
    if len(tab2)==0:
        return tab1
    if tab1[0]<=tab2[0]:
        return [tab1[0]]+fusion(tab1[1:],tab2)
    else:
        return [tab2[0]]+fusion(tab1,tab2[1:])

def merge_sort(tab):
    if len(tab)<=1:
        return tab
    else :
        return fusion(merge_sort(tab[len(tab)//2:]),merge_sort(tab[:len(tab)//2]))

def partitionner(tab,premier,dernier, pivot):
    tmp = tab[pivot]
    tab[pivot]=tab[dernier]
```

```
    tab[dernier]=tmp
    j = premier
    for i in range (premier,dernier):
        if tab[i]<= tab[dernier]:
            tmp = tab[i]
            tab[i]=tab[j]
            tab[j]=tmp
            j=j+1
    tmp = tab[j]
    tab[j]=tab[dernier]
    tab[dernier]=tmp
    return j


def quick_sort(tab,premier,dernier):
    if premier < dernier :
        pivot = random.randint(premier, dernier)
        pivot = partitionner(tab, premier, dernier, pivot)
        quick_sort(tab, premier, pivot-1)
        quick_sort(tab, pivot+1, dernier)



t = tabrand(12)
print(t)
quick_sort(t,0,len(t)-1)
print(t)

#TP 3


def multi(x,y):
    if len(str(x)) == 1 or len(str(y)) == 1:
        return x*y
    else:
        m = max(len(str(x)),len(str(y)))
        m2 = m // 2

        a = x // 10**(m2)
        b = x % 10**(m2)
        c = y // 10**(m2)
        d = y % 10**(m2)

        z0 = multi(b,d)
        z1 = multi(a,d)
        z3 = multi(b,c)
        z2 = multi(a,c)
```

```
        return (z2 * 10**(2*m2)) + ((z1 + z3) * 10**(m2)) + (z0)


# TP3 bis

import random
import math

def square(n):
    x = random.randint(1,100)
    while (math.fabs(math.sqrt(n)-x)>0.01):
        x = (x+n/x)/2
    return x


def maxisq():
    m=0
    for i in range(1,100000):
        x = random.randint(1,100000)
        j=0
        while (math.fabs(math.sqrt(i)-x)>0.01):
            x = (x+i/x)/2
            j+=1
        if j > m:
            m = j
    return m

print(square(10))
print(maxisq())

import matplotlib.pyplot as plt

def suit(n, mu):
    x = random.random()
    for i in range(n):
        x = 1 - mu*x*x
        t.append(i)
        t1.append(x)

t=[]
t1=[]
suit(25,2)
plt.plot(t,t1)
plt.ylabel('some numbers')
plt.show()
```

```python
# TP 4

def mystere(a,b):
    c=0
    while(b>0):
        if(b%2==1):
            c += a
        a = 2*a
        b = b//2
    return c

print(mystere(6,3))

# Retourne la valeur a*b
# C'est l'algorithme classique sauf que b est ecrit en base 2
# Soit b = 2^t+2^k+2^i+...+2^j
# Alors b*a = (2^t+2^k+2^i+...+2^j)*a


def corona(tab):
    for i in range(len(tab)):
        for j in range(len(tab[i])):
            if tab[i][j]!='X':
                s = 0
                if i>0 and tab[i-1][j]== 'X':
                    s+=1
                if j>0 and tab[i][j-1]=='X':
                    s+=1
                if i<len(tab)-1 and tab[i+1][j]=='X':
                    s+=1
                if j<len(tab[i])-1 and tab[i][j+1]=='X':
                    s+=1
                if s>1:
                    tab[i][j]='X'


t = [['O','O','O','X'],
     ['O','O','X','X'],
     ['O','O','O','X'],
     ['X','O','X','O']]

corona(t)
print(t)

import numpy as np
import time
import math
```

```python

# Compexite : O(l*m*k)
def classic_matrix_mult(A,B):
    l, k = A.shape
    k, m = B.shape
    C = np.zeros((l,m))
    for i in range(l):
        for j in range(m):
            for o in range(k):
                C[i,j] += A[i,o] * B[o,j]
    return C

# Compexite : O(n*m)
def matrix_plus(A,B):
    n, m = A.shape
    C = np.zeros((n, m))
    for i in range(n):
        for j in range(m):
            C[i,j]=A[i,j]+B[i,j]
    return C

def matrix_moins(A,B):
    n, m = A.shape
    C = np.zeros((n, m))
    for i in range(n):
        for j in range(m):
            C[i,j]=A[i,j]-B[i,j]
    return C
# Compexite : T(n) = 7T(n/2)+O(n*n) = O(n^(log_2(7)))


def strassen_matrix_mult(A,B):
    n, n = A.shape
    if(n==1):
        return np.array([[A[0,0]*B[0,0]]])

    A11=A[0:n//2, 0:n//2]
    A12=A[0:n//2, n//2:n]
    A21=A[n//2:n, 0:n//2]
    A22=A[n//2:n, n//2:n]
    B11=B[0:n//2, 0:n//2]
    B12=B[0:n//2, n//2:n]
    B21=B[n//2:n, 0:n//2]
    B22=B[n//2:n, n//2:n]

    P1 = strassen_matrix_mult(matrix_plus(A11,A22), matrix_plus(B11,B22))
    P2 = strassen_matrix_mult(matrix_plus(A21,A22), B11)
```

```
    P3 = strassen_matrix_mult(A11, matrix_moins(B12,B22))
    P4 = strassen_matrix_mult(A22, matrix_moins(B21,B11))
    P5 = strassen_matrix_mult(matrix_plus(A11,A12), B22)
    P6 = strassen_matrix_mult(matrix_moins(A21,A11), matrix_plus(B11,B12))
    P7 = strassen_matrix_mult(matrix_moins(A12,A22), matrix_plus(B21,B22))

    C = np.zeros((n, n))
    C[0:n//2, 0:n//2] = matrix_plus(matrix_moins(matrix_plus(P1,P4),P5),P7)
    C[0:n//2, n//2:n] = matrix_plus(P3,P5)
    C[n//2:n, 0:n//2] = matrix_plus(P2,P4)
    C[n//2:n, n//2:n] = matrix_plus(matrix_moins(matrix_plus(P1,P3),P2),P6)

    return C


H = np.random.rand(32,32)
I = np.random.rand(32,32)
print(classic_matrix_mult(H,I))
print(np.dot(H,I))
# remarque : Lors des tests de l'algo de Strassen, il faut que A et B soient
# des matrices carrees de taille 2^p (p entier)
# En comparant le temps de calcul de l'algo de Strassen avec l'algo classique,
# on remarque que ce dernier est plus rapide (bien que sa complexite theorique
# soit strictement superieur), cela est du a la constante associee a l'algo de
# Strassen qui est importante relativement a celle de l'algo classique
# Lors du calcul du produit de deux "petites" matrices, l'algo classique est
# donc plus rapide, cependant a partir d'un certain seuil, pour de "grandes"
# matrices l'algo de Strassen devient plus rapide que l'algo classique.


# Generalisation au matrice carrees de taille n*n (en considerant la matrice de
# taille 2^k * 2^k tel que 2^(k-1) < n < 2^k)

def strassen_matrix_mult_3(a, b):
    """Matrices multiplication (Strassen algorithm, any size of square matrix)

    :param a: A square matrix (numpy 2D array).
    :param b: A square matrix with same size as a.

    :return: The matrix product of a and b.
    """
    initial_size, _ = a.shape
    n = 2**math.ceil(math.log2(initial_size))
    a_extended = np.zeros((n, n))
    a_extended[0:initial_size, 0:initial_size] = a
    b_extended = np.zeros((n, n))
    b_extended[0:initial_size, 0:initial_size] = b
```

```
    c_extended = strassen_matrix_mult(a_extended, b_extended)
    return c_extended[0:initial_size, 0:initial_size]

# TP 5

from datetime import datetime

class Event:
    def __init__(self, title, when):
        self.title = title
        self.when = when

    def change_time(self,new_time):
        self.when = new_time

lesson = Event("Swimming lesson", datetime(2016, 12, 15, 17, 00))
lesson.change_time(datetime(2016,12,16,17,30))

class Appointment(Event):
    def __init__(self, title, when, with_whom):
        super().__init__(title,when)
        self.with_whom = with_whom

lunch = Appointment(title="Restaurant", with_whom="Donald", when=datetime(2015,12,25,12,0))
lunch.change_time(datetime(2015,12,25,12,30))

class Calendar:
    def __init__(self,event_list=None, owner=""):
        self.owner=owner
        if not event_list :
            self.event_list = []
        else:
            self.event_list = event_list

class Tree:
    def __init__(self,name="",children_list=None):
        self.name=name
        if not children_list :
            self.children_list = []
        else:
            self.children_list = children_list

    def height(self):
        if not self.children_list:
            return 0
        else:
            max=self.children_list[0].height()
```

```
            for i in range(1,len(self.children_list)):
                tmp=self.children_list[i].height()
                if tmp>max:
                    max=tmp
            return 1+max
    def display(self, indent=""):
    print(indent+self.name)
        for i in range(len(self.children_list)):
            self.children_list[i].display(indent+"    ")

    def size(self):
        if not self.children_list:
            return 1
        else:
            count=0
            for i in range(0,len(self.children_list)):
                count+=self.children_list[i].size()
            return count+1

    def nb_leaves(self):
        if not self.children_list:
            return 1
        else:
            count=0
            for i in range(0,len(self.children_list)):
                count+=self.children_list[i].nb_leaves()
            return count

    def search_dfs(self, name):
        if self.name == name:
            return self
        ret = None
        for i in range(0,len(self.children_list)):
            if ret == None :
                ret = self.children_list[i].search_dfs(name)
        return ret

    def display_bfs(self):
        tab = [self]
        while len(tab)>0:
            s = tab.pop(0)
            print(s.name)
            for i in range(0,len(s.children_list)):
                tab.append(s.children_list[i])


class MythologyTree(Tree):
```

```python
    def __init__(self,name="",children_list=None, status=None, gender=None):
        super().__init__(name,children_list)
        self.status=status
        self.gender=gender

P = MythologyTree(name="Persephone", status="God", gender="F")
D = MythologyTree(name="Demeter", status="God", gender="F", children_list=[P])
Z = MythologyTree(name="Zeus", status="God", gender="M")
C = MythologyTree(name="Cronos", status="Titan", gender="M", children_list=[Z,D])
H = MythologyTree(name="Hyperion", status="Titan", gender="M")
G = MythologyTree(name="Gaia", status="Deity", gender="F", children_list=[C,H])

print(G.name)
print(G.children_list[0].children_list[1].children_list[0].name)
print(G.height())

G.display("")

print(D.size())
print(G.nb_leaves())


# -*- coding: utf-8 -*-
def display_as_tree(lst, i=0, indent_root="", indent_others="    "):
    """Display a list as a binary tree.

    :param lst: A list.
    :param i: An integer. The index of the root node to consider.
    :param indent_root: A string to be printed before the root.
    :param indent_others: A string to be printed before its children.

    The list lst is considered as a binary tree, where the children of
    element lst[k] are lst[2 * k + 1] and lst[2 * k + 2] (if they exist).
    """
    if i >= len(lst):
        return
    print(indent_root + "|-> " + str(lst[i]))
    display_as_tree(lst, 2 * i + 1, indent_root=indent_others,
                    indent_others=indent_others + "|   ")
    display_as_tree(lst, 2 * i + 2, indent_root=indent_others,
                    indent_others=indent_others + "    ")


def siftdown(A,i):
    l = 2*i+1
    r = 2*i+2
    maxi = i
```

```
        if l<len(A) and A[l]>A[maxi]:
            maxi = l
        if r<len(A) and A[r]>A[maxi]:
            maxi = r
        if maxi != i:
            A[i], A[maxi] = A[maxi], A[i]
            siftdown(A,maxi)

def heapify(A):
    for i in range(len(A)//2-1,-1,-1):
        siftdown(A,i)

def heap_sort(A):
    heapify(A)
    result = []
    while len(A)>0:
        A[0], A[len(A)-1] = A[len(A)-1], A[0]
        result = [A[len(A)-1]]+result
        A = A[:len(A)-1]
        siftdown(A,0)
    return result

A = [4,1,3,8,2,0,5,6,9,7]
display_as_tree(A)
A = heap_sort(A)
print(A)
```

Exercices supplémentaires

Exercice 1 Montrer que $\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$. Donnez des exemples de fonctions f et g telles que $f(n) = O(g(n))$. Donnez des exemples de fonctions qui sont en $\Omega(n^3)$ mais pas en $\Theta(n^3)$.

Exercice 2 Donnez les complexités des algorithmes suivants en fonction des entiers n et m en paramÃ¨tre :

ALGORITHME $A(n, m)$
    $i \leftarrow 1$; $j \leftarrow 1$
    Tant que $(i \leq m)$ et $(j \leq n)$ faire
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$

ALGORITHME $B(n, m)$
    $i \leftarrow 1$; $j \leftarrow 1$
    Tant que $(i \leq m)$ ou $(j \leq n)$ faire
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$

ALGORITHME $C(n, m)$
    $i \leftarrow 1$; $j \leftarrow 1$
    Tant que $(j \leq n)$ faire
        Si $(i \leq m)$
            alors $i \leftarrow i + 1$
            sinon $j \leftarrow j + 1$

ALGORITHME $D(n, m)$
    $i \leftarrow 1$; $j \leftarrow 1$
    Tant que $(j \leq n)$ faire
        Si $(i \leq m)$
            alors $i \leftarrow i + 1$
            sinon $\{j \leftarrow j + 1; i \leftarrow 1\}$

Exercice 3 Donnez la complexité $\Theta(\cdot)$ de ces deux fonctions en considérant les cas $n \leq m$ et $n > m$.

```
def A(n,m):
        a = 0
        b = 1
        for i in range(0,n):
                for j in range(0,m):
                        a += b

def B(n,m):
        a = 0
        b = 1
        for i in range(0,n):
                for j in range(i,m):
                        a += b
```

Exercice 4 1) Démontrer que : $n^2 = O(10^{-5}n^3)$, $25n^4 - 19n^3 + 13n^2 = O(n^4)$, et $2^{n+100} = O(2^n)$.
2) Comparer (donner les relations d'inclusion) entre les ensembles suivants : $O(n \log n)$, $O(2^n)$, $O(\log n)$, $O(1)$, $O(n^2)$, $O(n^3)$, $O(n)$.
Soient quatre fonctions f, g, S et $T : \mathbb{N} \to \mathbb{N}$. Montrer que :
3) Si $f(n) = O(g(n))$, alors $g(n) = \Omega(f(n))$.
4) Si $f(n) = O(g(n))$, alors $f(n) + g(n) = O(g(n))$.
5) $f(n) + g(n) = \Theta(\max\{f(n), g(n)\})$.
6) $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$.
Soient $S(n) = O(f(n))$ et $T(n) = O(g(n))$.
7) Si $f(n) = O(g(n))$, alors $S(n) + T(n) = O(g(n))$.
8) $S(n)T(n) = O(f(n)g(n))$.

Algorithm 1 TRI-INSERTION(A)

for $j \leftarrow 2$ Ã $longueur[A]$ **do**
  $clé \leftarrow A[j]$
  $\triangleright$ Insertion de $A[j]$ dans la séquence triée $A[1, \dots, j-1]$.
  $i \leftarrow j - 1$
  while $i > 0$ et $A[i] > clé$ **do**
    $A[i+1] \leftarrow A[i]$
    $i \leftarrow i - 1$
  end while
  $A[i+1] \leftarrow clé$
end for

Exercice 5 $\star$ Donner la complexité du tri par insertion (**algorithme 1**). Donner une preuve de sa validité par induction sur j avec un invariant pour le sous-tableau $A[1..j]$.

Exercice 6 Pour tout entier k, on note le logarithme en base k par $\log_k x = (\ln k)^{-1} \ln x$. Au niveau de leur complexité, peut-on comparer deux algorithmes en $O(\log_k n)$ et en $O(\log_{k+1} n)$?

Exercice 7 On code un entier n avec $t(n) = \lceil \log_2 n \rceil$ bits (les caractères 0 ou 1 de sa décomposition en binaire). Donc $a := t(n)$ et $b := t(m)$ sont des entiers caractÃ©risant la taille de l'entrée des algorithmes de l'exercice précédent. Refaire l'exercice 2 en exprimant la complexité en fonction de a et b au lieu de n et m.

Correction exercices bonus :

Exercice 1

On va dans un premier temps montrer que $\Theta(g(n)) \subseteq \Omega(g(n)) \cap O(g(n))$.
Soit $f(n) = \Theta(g(n))$, montrons que $f(n) = \Omega(g(n))$ et $f(n) = O(g(n))$.
Puisque $f(n) = \theta(g(n))$, alors il existe une constate $c_1 > 0$ et un rang n_0 à partir duquel $\forall n \geq n_0, 0 \leq c_1 \cdot g(n) \leq f(n)$.
De même, il existe une constante $c_2 > 0$ et un rang n_0 à partir duquel $\forall n \geq n_0, 0 \leq f(n) \leq c_2 \cdot g(n)$.

Dans un second temps, on va montrer que $\Omega(g(n)) \cap O(g(n)) \subseteq \Theta(g(n))$.
Soit $f(n)$ tel que $f(n) = \Omega(g(n))$ et $f(n) = O(g(n))$, montrons que $f(n) = \Theta(g(n))$.
On sait qu'il existe une constante $c_1 > 0$ et un seuil $n_1 \in \mathbb{N}$ tel que $\forall n \geq n_1, 0 \leq c_1 \cdot g(n) \leq f(n)$. De même, il existe une constante $c_2 > 0$ et un seuil $n_2 \in \mathbb{N}$ tel que $\forall n \geq n_2, 0 \leq f(n) \leq c_2 \cdot g(n)$. En prenant $n_0 = max\{n_1, n_2\}$, on a $\forall n \geq n_0, 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

Exercice 4

1. $n^2 = O(10^{-5} \cdot n^3)$, avec $c = 10^5$ et $n_0 = 1$ on a $\forall n \geq n_0, n^2 \leq 10^5 \cdot 10^{-5} \cdot n^3$.
   $25n^4 - 19n^3 + 13n^2 = O(n^4)$, avec $c = 25$ et $n_0 = 1$, étnt donné que $-19n^3 + 13n^2 = n^2(13 - 19n) \leq 0$ pour n*$n \geq 1$ on a bien $25n^4 - 19n^3 + 13n^2 \leq 25n^4$.
   $2^{n+100} = O(2^n)$, avec $c = 2^{100}$ on a $2^{n+100} = 2^{100} \cdot 2^n$.
2. $O(1) \subseteq O(log_2(n)) \subseteq O(n) \subseteq O(n \cdot log_2(n)) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n)$
3. Si $f(n) = O(g(n))$ alors il existante une constante c telle qu'à partir d'un certain rang on a $f(n) \leq c \cdot g(n)$. Et donc on a $\frac{1}{c} \cdot f(n) \leq g(n)$ à partir du même rang, donc $g(n) = \Omega(f(n))$.
4. Si $f(n) = O(g(n))$ alors il existante une constante c telle qu'à partir d'un certain rang on a $f(n) \leq c \cdot g(n)$. Donc on a $f(n) + g(n) \leq (c+1) \cdot g(n)$ à partir du même rang, et donc $f(n) + g(n) = O(g(n))$.
5. $f(n) + g(n) = \Theta(max\{f(n), g(n)\})$ car $max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2 \cdot max\{f(n), g(n)\}$.
6. $O(f(n)+g(n)) = O(max\{f(n), g(n)\})$: En premier lieu $f(n)+g(n) \leq 2 \cdot max\{f(n), g(n)\}$ donc $O(f(n) + g(n)) \subseteq O(\max\{f(n), g(n)\}$. En second lieu, $max\{f(n), g(n)\} \leq f(n) + g(n)$ donc $O(max\{f(n), g(n)\}) \subseteq O(f(n) + g(n))$.
7. $S(n) = O(f(n))$ et $T(n) = O(g(n))$ et $f(n) = O(g(n))$ donc il existe des constantes $c_1, c_2, c_3 > 0$ telles que à partir de rangs $n_1, n_2, n_3 \in \mathbb{N}$, on ait pour tout $n \geq max\{n_1, n_2, n_3\}, S(n) \leq c_1 \cdot f(n), T(n) \leq c_2 \cdot g(n), f(n) \leq c_3 \cdot g(n)$. A partie du même rang, on a : $S(n) + T(n) \leq c_1 \cdot f(n) + c_2 \cdot g(n) \leq (c_3 + c_2) \cdot g(n)$ et donc $S(n) + T(n) = O(g(n))$.
8. $S(n) \cdot T(n) \leq c_1 \cdot f(n) \cdot T(n) \leq c_1 \cdot c_2 \cdot f(n) \cdot g(n)$ et donc $S(n) \cdot T(n) = O(f(n) \cdot g(n))$.

Chapitre 2 - Exercices supplémentaires

Rappels de cours *Un algorithme récursif (paradigme "diviser pour régner") a un temps d'exécution*

$$T(n) = aT(n/b) + f(n)$$

où n est la taille des entrées, a est le nombre de sous-problèmes, n/b (=$\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$) est la taille des sous-problèmes, et $f(n)$ est le temps requis pour diviser et combiner.

Exercice 8 Donner $g(n)$ telle que $T(n) = \Theta(g(n))$ oÃ¹T(n)$estdfiniercursivementpar$:

1) $T(n) = 8T(n/2) + n^3$.
2) $T(n) = 16T(n/4) + n$.
3) $T(n) = 8T(n/2) + n^2$.
4) $T(n) = 17T(n/16) + n \log n$.
5) $T(n) = 8T(n/3) + n^2$.
6) $T(n) = 9T(n/3) + n^2$.
7) $T(n) = 8T(n/2) + n^4 \log n$.
8) $T(n) = 2T(n/2) + n$.

Exercice 9 Soit l'algorithme de tri suivant :

Algorithm 2 TRI-FUSION(A, i, k)

```
if i < k then
    j ← ⌊(i+k)/2⌋
    TRI-FUSION(A, i, j)
    TRI-FUSION(A, j + 1, k)
    FUSIONNER(A, i, j, k)
end if
```

1) Appliquer l'algorithme tri-fusion (**Algorithme 2**) au tableau A suivant :

9	8	12	3	5	14	6

2) Sachant que la fusion de deux tableaux triés dont la somme des longueurs est n s'effectue en $\Theta(n)$, donner la complexité du tri-fusion.
3) Démontrer sa validité.

Exercice 10 *Somme des éléments d'un tableau*

Soit A un tableau de $n \geq 1$ entiers.
1) Ecrire en Python un algorithme itératif calculant la somme des éléments de A et démontrer sa validité.
2) Déterminer sa complexité.
3) Réécrire l'algorithme pour qu'il soit récursif est satisfasse $T(n) = 2T(n/2) + O(1)$.
4) A-t-on ainsi amélioré la complexité de l'algorithme ?

Exercice 11 *Fonctions mystères*

Soit la fonction F (dépendant d'un entier n) suivante :
1) Que calcule F ? Le démontrer.
2) Donner le nombre $m(n)$ de multiplications effectuées par $F(n)$.

Algorithm 3 F(n)

```
if n=0 then
  retourner 2
else
  retourner F(n − 1)*F(n − 1)
end if
```

Algorithm 4 G(n)

```
R ← 2
for  i = 1 a n do
  R ← R ∗ R
end for
retourner R
```

3) Déterminer la complexité de F et montrer comment l'améliorer.
Soit la fonction G (dépendant d'un entier n) suivante :
4) Que calcule G ? Le démontrer.
5) Déterminer la complexité de G.

Corrections

Ex 10

1.

```python
def sumtab(A):
    sum = 0
    for i in range(len(A)):
        sum += A[i]
    return sum
```

Invariant : A l'itération i de la boucle *for*, la variable *sum* est égale à la somme des éléments du sous-tableau $A[0 : i - 1]$.
Initialisation : Après la 1ère itération de la boucle *for*, $sum = A[0]$.
Supposons que l'invariant soit vrai pour $i \in \{1, ...n-1\}$, donc à l'itération $i+1$ de la boucle *for*, la variable $sum = S + A[i]$ où S correspond à la somme des i premiers éléments (correspondant au sous tableau $A[0, i-1]$ par hypothèse). Donc à la fin de l'itération $i+1$, *sum* correspond à la somme des $i + 1$ premiers éléments de A (correspondant au sous-tableau $A[0 : i]$).
Conclusion : La boucle *for* s'arrête après n itérations (n étant le nombre d'éléments dans A), et donc *sum* correspond à la somme de tous les éléments du tableau (soit le tableau $A[0 : n - 1]$).

2. L'algorithme ci-dessus est en $\Theta(n)$, où n est le nombre d'éléments du tableau A.

3.

```
def sumtab_rec(A):
    if len(A)==1:
        return A[0]
    return sumtab_rec(A[:len(A)//2])+sumtab_rec(A[len(A)//2:])
```

On a bien 2 appels récursif, chacun effectué sur une moitié du tableau A et une partie itérative en $O(1)$.
4.

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + O(1) \\
&= 2 \cdot (2T(\frac{n}{2^2}) + O(1)) + O(1) \\
&= ... \\
&= \sum_{i=0}^{log_2(n)} 2^i O(1) \\
&= (2^{log_2(n)+1} - 1)O(1) \\
&= (2n-1)O(1)
\end{aligned}$$

Les deux algorithmes sont donc en $\Theta(n)$.

Ex 11

1. $F(n)$ renvoie la valeur 2^{2^n}. On peut le montrer par récurrence :
Initialisation : $F(0)$ renvoie bien 2, et $2^{2^0} = 2$.
Supposons que $F(n)$ retourne la valeur 2^{2^n}, alors $F(n+1)$ retourne la valeur $2^{2^n} * 2^{2^n} = 2^{2^n+2^n} = 2^{2\cdot(2^n)} = 2^{2^{n+1}}$.

2. Il y a $2^n - 1$ multiplications lors du calcul de $F(n)$, on peut représenter les appels récursifs avec un arbre, ainsi on a une multiplication pour tous les noeuds correspondant aux calculs de $F(n), F(n-1), ..., F(2), F(1)$ (et non $F(0)$), ce qui donne $\sum_{i=0}^{n-1} 2^i = 2^n - 1$.
3. La complexité de $F(n)$ peut être calculée de la manière suivante :

$$\begin{aligned}
T(n) &= 2T(n-1) + O(1) \\
&= 2 \cdot (2T(n-2) + O(1)) + O(1) \\
&= ... \\
&= \sum_{i=0}^{n} 2^i O(1) \\
&= (2^{n+1} - 1)O(1)
\end{aligned}$$

On a $F(n) = \Theta(2^n)$.
4. L'algorithme $G(n)$ renvoie la valeur 2^{2^n} également, la preuve est identique à la question 1.
5. L'algorithme $G(n)$ est en $\Theta(n)$, donc bien meilleur que $F(n)$ pour le même résultat.