Chapitre 2

Prérequis mathématiques

Ce cours nécessite des prérequis mathématiques suivants :

- Pour montrer des propriétés de programmes, il faut savoir faire une preuve par récurrence (les preuves demandées seront assez simples).
- Pour évaluer la complexité des algorithmes, vous devez :
  - comprendre les fonctions $\log_b$, en particulier pour les bases $b = 2$ et $b = 10$.
  - comprendre la notation $O()$.

Les deux derniers points sont utiles pour évaluer l'efficacité des algorithmes, et simplifier cette évaluation.

1 Comment rédiger une preuve par récurrence ?

L'objectif de cette section est de préciser ce qui est attendu pour certaines preuves dans ce cours, sur un exemple volontairement très simple. La raison pour laquelle on prouvera des propriétés est que cela est nécessaire pour assurer la correction de certains algorithmes ou évaluer leur complexité (cf. Section 2).

Les démonstrations par récurrence sont souvent utilisées pour montrer des propriétés sur les entiers. Supposons qu'on veut montrer qu'une propriété est vraie pour tout entier $n \geqslant n_0$ (où $n_0 \in \mathbb{N}$ est un entier fixé, par exemple $n_0 = 0$ ou 1). Le principe est de montrer que la propriété est :

1. vraie pour $n = n_0$, et
2. qu'elle se « transmet » d'un entier à son successeur : pour tout entier $n \geqslant n_0$, si la propriété est vraie pour tout entier entre n_0 et n, alors elle est aussi vraie pour l'entier $n + 1$.

L'intuition est que grâce au premier point, la propriété est vraie pour $n = n_0$, et grâce au second, elle est vraie pour tout entier de $[n_0\,;n_0+1]$, et donc pour tout entier de $[n_0\,;n_0+2]$, etc.

On peut utiliser la démonstration par récurrence pour montrer des propriétés sur les arbres (binaires ou non) et pas seulement des propriétés sur les entiers. Pour cela, on peut raisonner par récurrence sur un paramètre de l'arbre à valeur dans $\mathbb{N}$, comme sa taille ou sa hauteur.

À titre d'exemple, montrons la propriété suivante par récurrence.

Tout arbre binaire non vide a au moins une feuille.

C'est une propriété qui semble évidente, mais l'objectif est que la preuve suivante serve de modèle pour la rédaction de propriétés plus compliquées.

Soit donc t un arbre binaire. On raisonne par *récurrence* sur la taille $n(t) = n$ de l'arbre t. On veut donc montrer la propriété suivante pour tout entier $n \geqslant 1$:

Tout arbre binaire non vide qui a n nœuds a au moins une feuille. $(\mathcal{P}_n)$

Les étapes pour montrer cette propriété sont les suivantes.

1. On remarque que comme t est non vide, on a effectivement $n \geqslant 1$.
2. Si $n = 1$, alors la racine de t est une feuille, et le résultat voulu est obtenu. ✓
3. Supposons la propriété ($\mathcal{P}_n$) vraie si $1 \leqslant k \leqslant n$, et montrons qu'elle est vraie également pour $k = n+1$.
4. Soit donc t de taille $n+1$. On a $n+1 > 1$ (sinon, on est dans le cas d'un arbre avec un unique nœud, déjà traité).
   - Comme t est formé d'un nœud racine et de sous-arbres ℓ et r, on a :
   $$n + 1 = n(t) = 1 + n(\ell) + n(r). \tag{2.1}$$
   - Comme $n(t) > 1$, soit ℓ, soit r n'est pas vide. Supposons que c'est ℓ (si c'est r, le raisonnement est identique en replaçant ℓ par r).
   - D'après l'équation (2.1), on a $n(\ell) = n - n(r) \leqslant n$.
   - Puisque ℓ a au plus n nœuds et qu'on a supposé que l'hypothèse de récurrence ($\mathcal{P}_n$) est vraie pour tout arbre ayant entre 1 et n nœuds, on peut l'appliquer à ℓ : ceci implique que ℓ a au moins une feuille.
   - Mais cette feuille est aussi une feuille de t : on a donc montré le résultat. ✓

2 Complexité

Un algorithme est destiné à traiter des entrées arbitrairement grandes, et fréquemment, plus l'entrée d'un algorithme est grande, plus l'algorithme mettra du temps sur cette entrée. Une façon d'évaluer le temps de calcul d'un algorithme est de calculer sa complexité dans le cas le pire. Si f est une fonction de $\mathbb{N}$ dans $\mathbb{N}$, on dit qu'un algorithme a une complexité $f(n)$ s'il effectue au maximum $f(n)$ opérations élémentaires sur chacune des entrées de taille n. La complexité d'un algorithme est donc une fonction $f : \mathbb{N} \to \mathbb{N}$, qui sert à mesurer l'efficacité de l'algorithme.

2.1 La fonction logarithme

La fonction logarithme $x \mapsto \ln(x)$ est souvent définie comme la primitive de la fonction $x \mapsto \frac{1}{x}$. Dans ce cours, cette propriété n'est pas utile : il est plus important de comprendre que les fonctions « logarithme »

- sont croissantes,
- mais croissent très lentement, plus lentement que toute fonction $x \mapsto x^a$ pour $a > 0$.

Deux fonctions « logarithme » pourront être utilisées dans ce cours :

- la fonction $x \mapsto \log_2(x)$, le logarithme à base 2,
- la fonction $x \mapsto \log_{10}(x)$, le logarithme à base 10.

Ces deux fonctions sont proportionnelles : si on note $\alpha = \log_2(10) \simeq 3,32$, on a $\log_2(n) = \alpha \cdot \log_{10}(n)$. Cela montre qu'il suffit de comprendre la fonction $\log_2$ pour comprendre la fonction $\log_{10}$ (qu'on obtient par division par α). La fonction $\log_2$ est la fonction réciproque de la fonction 2^n, ce qui signifie que

$$\log_2(2^n) = n, \text{ et}$$
$$2^{\log_2(n)} = n.$$

La fonction $\log_2$ est croissante, et donc

$$2^n \leqslant k < 2^{n+1} \iff n \leqslant \log_2(k) < n+1 \iff \lfloor \log_2(k) \rfloor = n.$$

Comme les entiers dans l'intervalle $[2^n, 2^{n+1}[$ sont exactement les entiers qui s'écrivent avec $n+1$ chiffres en base 2, on en déduit que $1 + \lfloor \log_2(k) \rfloor$ est le nombre de chiffres de k dans son écriture en base 2. De

la même façon, $1 + \lfloor \log_{10}(k) \rfloor$ est le nombre de chiffres de k dans son écriture en base 10. Par exemple, le logarithme à base 10 de 123456789 est compris entre 8 et 9. Cela doit donner une idée de l'ordre de grandeur de $\log_2(n)$ ou de $\log_{10}(n)$: ces valeurs sont bien plus petites que n. De façon générale, on montre que si $a > 0$ et $b > 1$:

$$\lim_{n\to\infty} \frac{\log_b(n)}{n^a} = 0.$$

À l'inverse, une fonction « exponentielle » à base $b > 1$ croît plus vite que toute fonction « puissance » :

$$\lim_{n\to\infty} \frac{b^n}{n^a} = +\infty.$$

2.2 Quelques graphiques

Pour comprendre l'ordre de grandeur de ces fonctions, il est utile de tracer leurs courbes. Sur la Fig. 2.1, on a tracé

- 3 fonctions exponentielle : à bases 2, 1.5 et 1.25,
- 3 fonctions puissance : de degré 1, 2 et 3,
- 3 fonctions logarithme : à base e, 2 et 10.

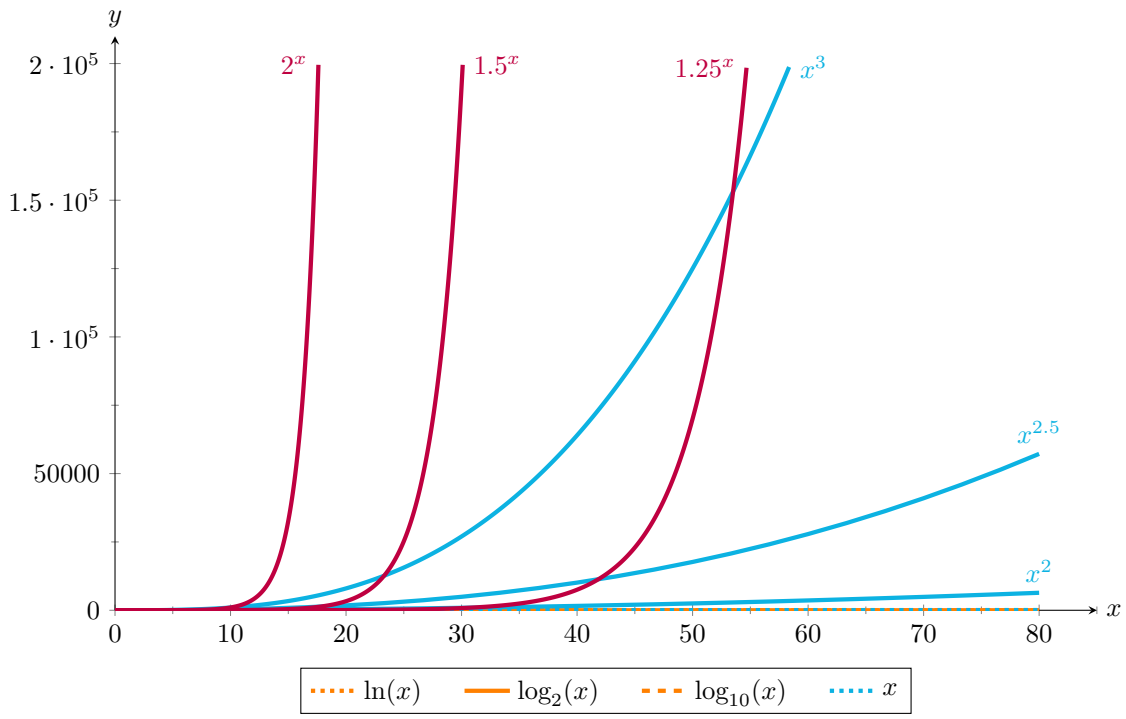

Fig. 2.1 : Croissance de fonctions importantes

L'échelle n'est pas la même sur les axes x et y, car les exponentielles croissent très vite. On constate que la plus petite des 3 fonctions « exponentielle », $x \mapsto 1.25^x$, rattrape la fonction $x \mapsto x^3$ pour $x \simeq 50$ (en réalité à partir de $x = 54$). Par comparaison, les 3 fonctions « logarithme » ainsi que la fonction $x \mapsto x$ semblent ne pas décoller de l'axe des abscisses (cela est dû à l'échelle choisie en x et y). Cet exemple montre qu'un algorithme effectuant 2^n opérations élémentaires sur une entrée de taille n se terminera au bout d'un temps prohibitif.

Sur la Fig. 2.2, on a changé l'échelle en y et on a représenté

- 3 fonctions puissance : $x \mapsto x^2$, $x \mapsto x^{1.5}$ et $x \mapsto x$,
- la fonction $x \mapsto x \log_2(x)$,
- 2 fonctions logarithme : $\log_2(x)$ et $\log_{10}(x)$.

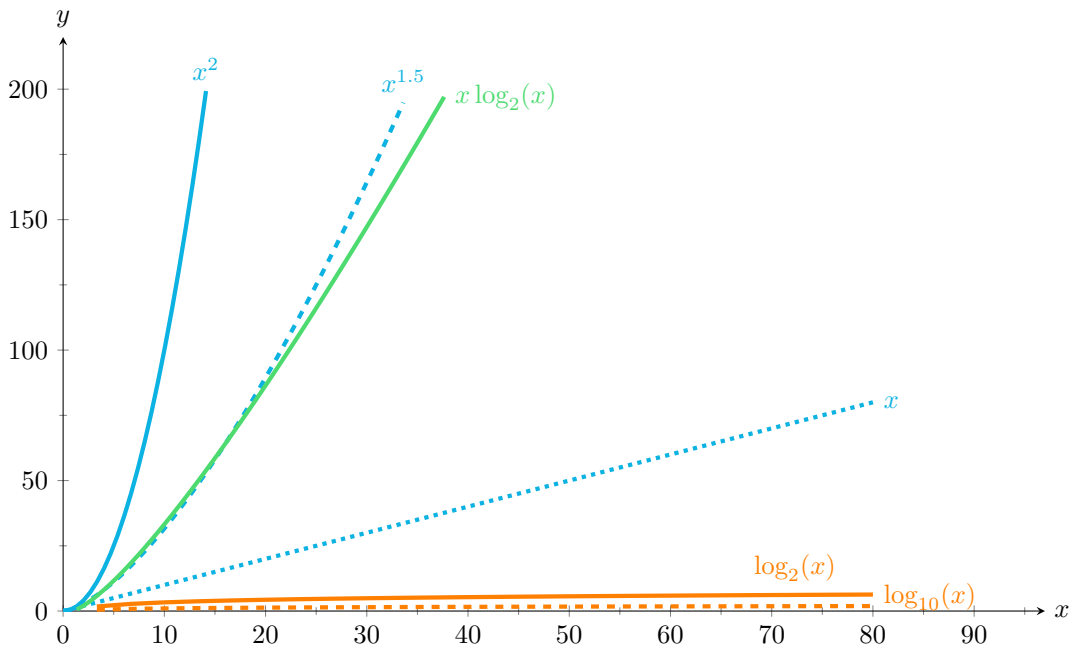

FIG. 2.2 : Croissance de fonctions importantes (2)

La fonction $x \log_2(x)$ semble croître moins rapidement que les deux fonctions $x \mapsto x^2$ et $x \mapsto x^{1.5}$.Cela peut se formaliser, en utilisant le fait que $\lim_{n\to\infty} \frac{\log_b(n)}{n^a} = 0$ pour $a > 0$ et $b > 1$. On constate aussi que les logarithmes croissent bien moins rapidement que la fonction identité $x \mapsto x$.

Enfin, la FIG. 2.3 montre, avec une échelle à nouveau différente, deux logarithmes et trois puissances. Les fonctions logarithmes semblent arriver à un palier, ce qui est trompeur, car elles tendent vers $+\infty$. Mais cela illustre la lenteur de leur croissance. Formellement, en utilisant $\lim_{n\to\infty} \frac{\log_b(n)}{n^a} = 0$ pour $a > 0$ et $b > 1$ pour $a = 1/3$ et $b = 2$, on obtient par exemple :

$$\lim_{n\to\infty} \frac{\log_2(n)}{n^{1/3}} = 0.$$

Une autre façon d'appréhender la croissance de ces fonctions est de calculer quelques valeurs. Pour donner un ordre d'idées, l'âge de l'univers est estimé à moins de $(4\,,5) \cdot 10^{17}$ secondes et le nombre de particules dans l'univers observable à moins de 10^{85}. Le tableau suivant donne le nombre de secondes qu'il faudrait attendre sur un ordinateur effectuant 10^9 opérations par seconde pour des algorithmes dont la complexité est donnée dans la première colonne, sur des entrées dont la taille est donnée en première ligne. En particulier, si un algorithme effectue $1\,.5^n$ opérations sur les données de taille n, on passe d'une réponse instantanée pour $n = 10$ à plus de $4 \cdot 10^8$ secondes pour $n = 100$, soit plus de 12 ans. Un tel algorithme ne serait donc pas praticable (c'est bien pire pour une complexité de 2^n, qui nécessiterait 10^{21} secondes pour $n = 100$, soit plus de 1000 fois l'âge de l'univers). La durée correspondant à la fonction n^2 devient longue pour $n = 10^7$ (10^5 secondes, soit un peu plus d'un jour) et impraticable pour $n = 10^8$ (près de 4 mois) et $n = 10^9$ (plus de 31 ans). Les trois autres complexités donnent des réponses assez rapides jusqu'à $n = 10^9$.

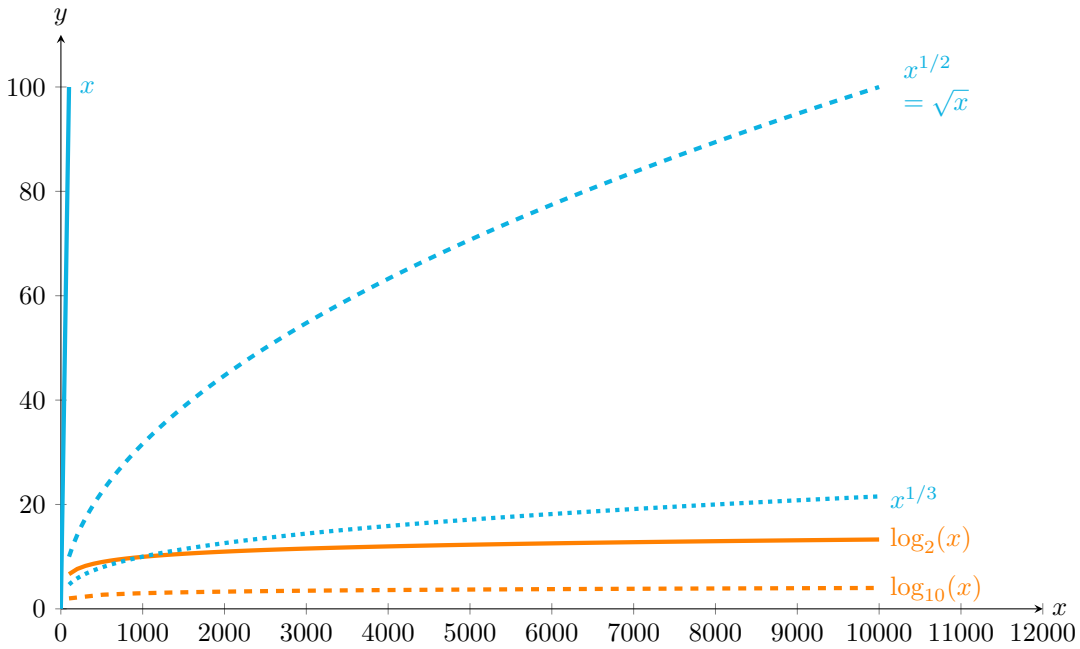

FIG. 2.3 : Croissance de fonctions importantes (3)

	10	100	1000	10^4	10^5	10^6	10^7	10^8	10^9
$\log_{10}(n)$	10^{-9}	$2 \cdot 10^{-9}$	$3 \cdot 10^{-9}$	$4 \cdot 10^{-9}$	$5 \cdot 10^{-9}$	$6 \cdot 10^{-9}$	$7 \cdot 10^{-9}$	$8 \cdot 10^{-9}$	$9 \cdot 10^{-9}$
n	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	1
$n \log_{10}(n)$	10^{-8}	$2 \cdot 10^{-7}$	$3 \cdot 10^{-6}$	$4 \cdot 10^{-5}$	$5 \cdot 10^{-4}$	$6 \cdot 10^{-3}$	$7 \cdot 10^{-2}$	$8 \cdot 10^{-1}$	9
n^2	10^{-7}	10^{-5}	10^{-3}	10^{-1}	10	1000	10^5	10^7	10^9
1.5^n	$6 \cdot 10^{-9}$	$4 \cdot 10^8$	-	-	-	-	-	-	-
2^n	10^{-6}	10^{21}	-	-	-	-	-	-	-

TAB. 2.1 : Estimation du temps nécessaire en secondes, à un milliard d'opérations par seconde

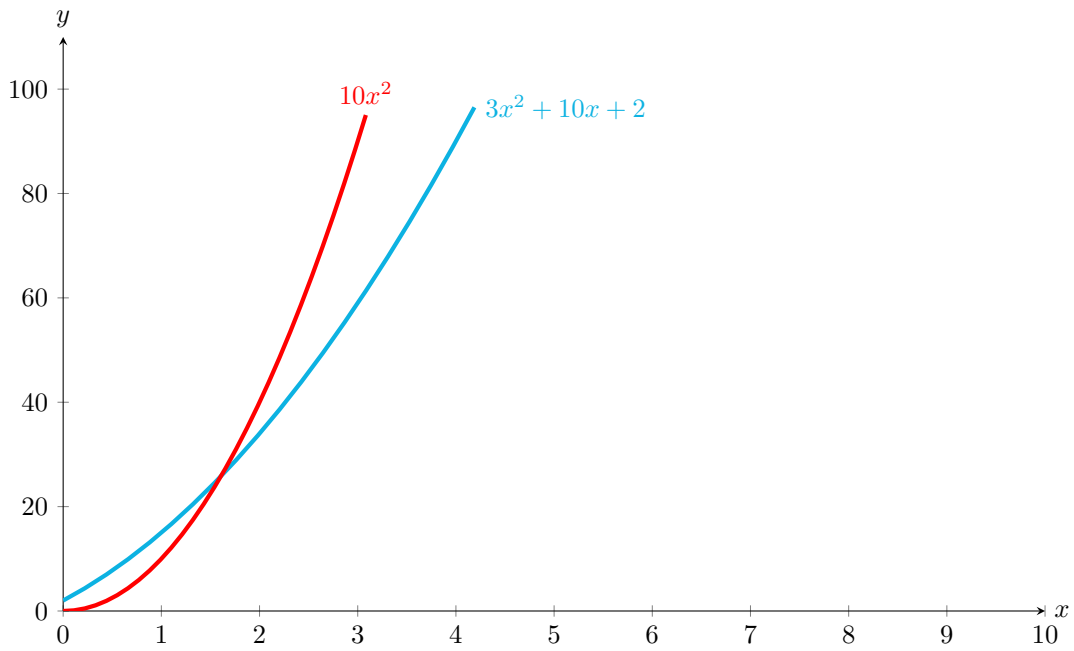

FIG. 2.4 : $3x^2 + 10x + 2 = O(x^2)$

2.3 La notation $O()$

Lorsqu'on évalue la complexité d'un algorithme, on a surtout besoin de l'ordre de grandeur de la fonction de complexité. Cet ordre de grandeur est souvent obtenu par additions et multiplications de fonctions simples, comme les fonctions puissance $n \mapsto n^k$, la fonction logarithme $n \mapsto \log(n)$ et des fonctions exponentielles, comme $n \mapsto 2^n$.

Pour simplifier les calculs, on retient d'une expression dénotant une telle fonction seulement le terme dominant. Le premier point est qu'on raisonne à constante multiplicative près. Ainsi, on n'a pas envie de différencier n^7 et $42n^7$, qui ne diffèrent que de la constante multiplicative 42.

Le second point est qu'on cherchera dans ce cours à ***majorer*** la complexité, par une fonction simple lorsque la taille n de l'entrée est grande. On se placera toujours dans le pire des cas sur l'ensemble des entrées possibles d'une taille donnée.

Cela conduit à la notation $O()$, utilisée pour obtenir des ordres de grandeur (pas nécessairement précis mais représentatifs de la croissance d'une fonction).

Plus précisément, si f et g sont des fonctions de $\mathbb{N}$ dans $\mathbb{R}_+$ on note $f = O(g)$, ou encore $f(n) = O(g(n))$, et on dit que « f est un grand O de g », si

$$\text{Il existe } C > 0 \text{ tel que, pour } n \text{ suffisamment grand, on a } f(n) \leqslant C \cdot g(n).$$

Il faut noter que l'on ne demande pas que l'inégalité $f(n) \leqslant C \cdot g(n)$ soit vraie pour tout n, mais seulement lorsque n est assez grand. L'inégalité peut donc être fausse pour un nombre fini de valeurs de n. La FIG. 2.4 illustre que $3x^2 + 10x + 2 \leqslant 10x^2$ si $x > 2$, on a donc $3x^2 + 10x + 2 = O(x^2)$.

Par ailleurs, on a le choix sur la constante C. On a ainsi choisi $C = 10$ dans l'exemple précédent, Autre exemple, si $f(n) = 1000n^2 + 42^{42}n + 1$, on a $f(n) = O(n^2)$, car pour n assez grand, $f(n) \leqslant 1001 \cdot n^2$.

Exercice 2.1 Exprimer toutes les relations de la forme $f = O(g)$ pour f et g parmi les fonctions suivantes : $f_1(n) = 42n^7 + 12n^5 - 12345$, $f_2(n) = (\log(n))^{42}$, $f_3(n) = n\log_2(n)$, $f_4(n) = n\log_{10}(n)$, $f_5(n) = n^7$, $f_6(n) = n^{0.001}$, $f_7(n) = 1.001^n$ et $f_8(n) = 2^n$.