# Algorithmique des structures arborescentes

L2 Info et Math-info, 2018–19

Marc Zeitoun

28 janvier 2019

# Plan

Objectifs et organisation de l'UE

Arbres binaires : vocabulaire & propriétés

Comparaison C vs. OCaml (démo)

Complexité : rappels

# Objectifs et organisation de l'UE

# Objectifs de l'UE

- **Comprendre** les arbres comme structures de données.
- Concept centraux
  - **Récursivité**.
  - **Correction et efficacité des algorithmes**.

# Objectifs de l'UE

- **Comprendre** les arbres comme structures de données.
- Concept centraux
  - **Récursivité**.
  - **Correction et efficacité des algorithmes**.

Points importants :

- Comprendre les algorithmes du cours.
- Concevoir des algorithmes simples.
- Évaluer la complexité d'algorithmes simples.
- Montrer des propriétés simples des arbres et algorithmes.

# Organisation

- L'enseignement dure **12 semaines**.
- **CM** : $\mathbf{6} \times$ 1h20.
  Créneau habituel : jeudi 9h30–10h50, une semaine sur 2.
- **CI** et **TM** : 4 blocs de 3 semaines :
  - $\mathbf{2} \times$ 1h20 CI,
  - 1h20 CI + 1h20 TM.
  - 1h20 CI + 1h20 TM.

# Organisation

- L'enseignement dure **12 semaines**.
- **CM** : **6** $\times$ 1h20.
  Créneau habituel : jeudi 9h30–10h50, une semaine sur 2.
- **CI** et **TM** : 4 blocs de 3 semaines :
  - **2** $\times$ 1h20 CI,
  - 1h20 CI + 1h20 TM.
  - 1h20 CI + 1h20 TM.

- Nous joindre : uf-info.ue.algo-arbres@diff.u-bordeaux.fr.

# Supports d'enseignement

Le cours est fait en CM **et en CI**.

- Site **Moodle** de l'UE, ouverture cette semaine.
  - **Polycopié** de cours à lire **pendant** le semestre.
  - **Diaporamas** de cours.
  - Feuilles d'**exercices**.
  - 2 **livres** en français, pdf accessibles.

# Évaluation

- **Contrôle continu (CC).**
  - **Test 15%** 30mn semaine du 11–15 février.
  - **Test 15%** 30mn, semaine du 11–15 mars.
  - **TP noté 30%** 1h20, semaine du 15 avril.
  - **DS 40%** 1h20, jeudi 28 mars.
  - Évaluation **Moodle** (1 point bonus).
- **Examen session 1 (EX1) et 2 (EX2)**
- **Note session 1** : (EX1+CC)/2
- **Note session 2** : max(EX2, (EX2+CC)/2)

# Comment réussir

- Lire et **comprendre** le cours
- Être **actif/active** en CI et TM
- **Chercher** les exercices soi-même
- Ne pas hésiter à nous contacter par mail

# Enseignants de l'UE

- MI A1 Théo Pierron
- MI A2 Philippe Duchon
- INF A1 Simon Archipoff
- INF A2 Irène Durand
- INF A3 Stefka Gueorguieva
- INF A4 Frédérique Carrère
- INF A5 Marc Zeitoun
- TM MI A2, INF A2, INF A5 Frédéric Mazoit et Irène Durand

# Contenu de l'UE

- Rappels sur la **complexité**.
- **Arbres** binaires : vocabulaire, propriétés, parcours.
- Arbres binaires de **recherche**,
- Arbres **équilibrés**.
- Autres sortes d'arbres,
- **Tas**.
- Algorithme de **compression** basé sur les arbres.

Algorithmes programmés en OCaml, parfois en C.

# Plan de ce premier cours

 Voir le polycopié pour plus de détails.

- Arbres binaires : vocabulaire et propriétés
- Parcours récursif en profondeur

   Démo de comparaison en **C** et **Ocaml**.
- Complexité : définition, notation $O()$, rédaction de preuve.

# L'arbre comme structure de données

- Apparaît naturellement : XML, arborescences de répertoires, ...
- Structure qui permet de ranger efficacement des données.
- Gain de temps spectaculaire par rapport à listes/piles/files.
  - **10000 ans** $\rightarrow$ quelques secondes.
- Cette UE est un prérequis pour algo des graphes.

# Arbres binaires : définition

**Définition récursive**. Un ***arbre binaire étiqueté*** est :

- soit l'arbre vide, noté $()$ ou Empty dans ce cours.
- soit est formé
  - d'un nœud, appelé sa racine, portant une information appelée étiquette du nœud,
  - et de deux arbres binaires, appelés sous-arbres gauche et droit.

# Arbres binaires : définition

**Définition récursive**. Un ***arbre binaire étiqueté*** est :

- soit l'arbre vide, noté $()$ ou Empty dans ce cours.
- soit est formé
  - d'un nœud, appelé sa racine, portant une information appelée étiquette du nœud,
  - et de deux arbres binaires, appelés sous-arbres gauche et droit.

Un arbre non vide $t$ est donc décrit par un triplet $(v, \ell, r)$ formé :

- d'une valeur $v$ de type fixé, qui est l'étiquette de la racine de $t$,
- d'un arbre $\ell$, qui est le sous-arbre gauche de $t$,
- d'un arbre $r$, qui est le sous-arbre droit de $t$.

# Arbres binaires : représentation graphique

**Définition** récursive $\Longrightarrow$ **dessin** obtenu récursivement.

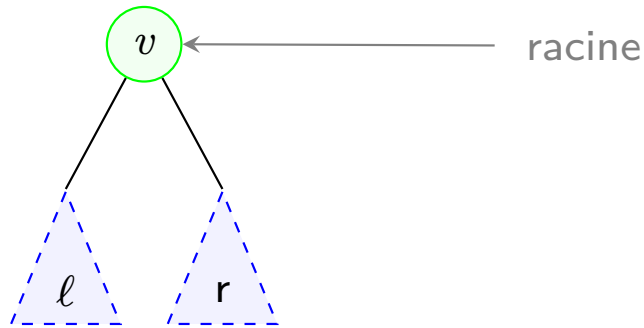
- arbre vide :

# Arbres binaires : représentation graphique

**Définition** récursive $\Longrightarrow$ **dessin** obtenu récursivement.

- arbre vide :

- arbre non vide $(v, \ell, r)$ :



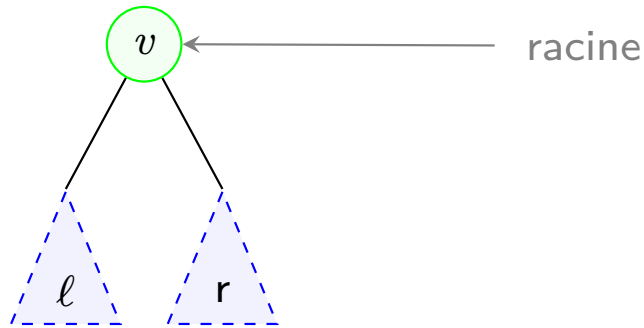

et le dessin continue, récursivement, pour $\ell$ et $r$.

# Arbres binaires : représentation graphique

**Définition** récursive $\Longrightarrow$ **dessin** obtenu récursivement.

- arbre vide :

- arbre non vide $(v, \ell, r)$ :

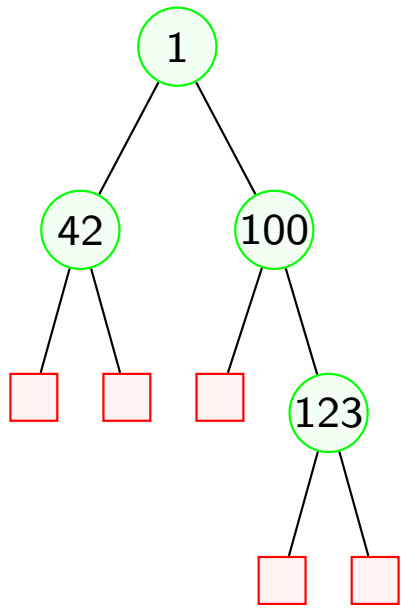



et le dessin continue, récursivement, pour $\ell$ et $r$.

Attention ! L'arbre vide n'est pas un nœud !

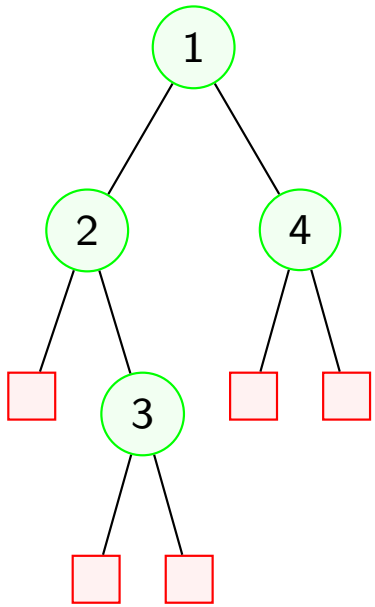
# Exemple

$$t = (\underbrace{1}_{v}, \quad \underbrace{(42, (), ())}_{\ell}, \quad \underbrace{(100, (), (123, (), ()))}_{r})$$

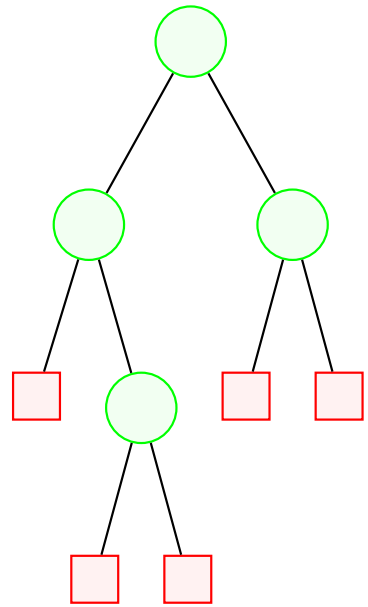


# Squelette d'un arbre

Squelette d'un arbre binaire : obtenu en supprimant les étiquettes.

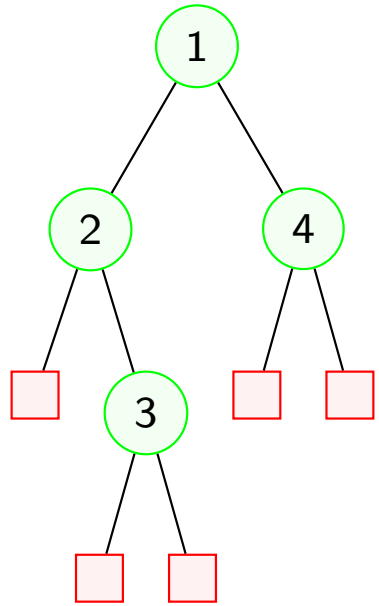



(a) Un arbre binaire

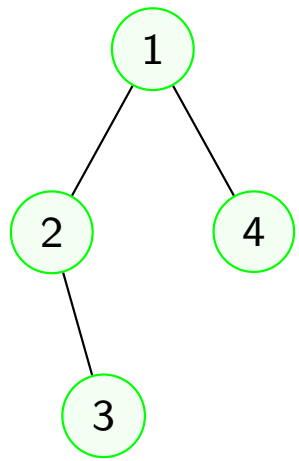


(b) Son squelette

# Sous-arbres vides : inutiles !




(a) Un arbre binaire



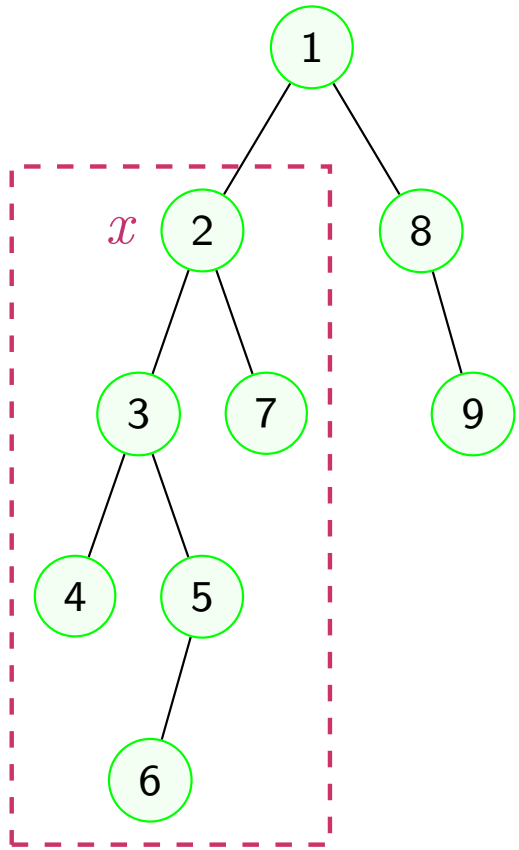

(b) Le même, sans dessiner les sous-arbres vides

# Vocabulaire à connaître

- Sous-arbre enraciné en un nœud,
- Fils gauche, fils droit d'un nœud,
- Père d'un nœud,
- Arité d'un nœud,
- Feuille,
- Arête,
- Branche,
- Hauteur d'un arbre,
- Taille d'un arbre,
- Profondeur (ou niveau) d'un nœud.

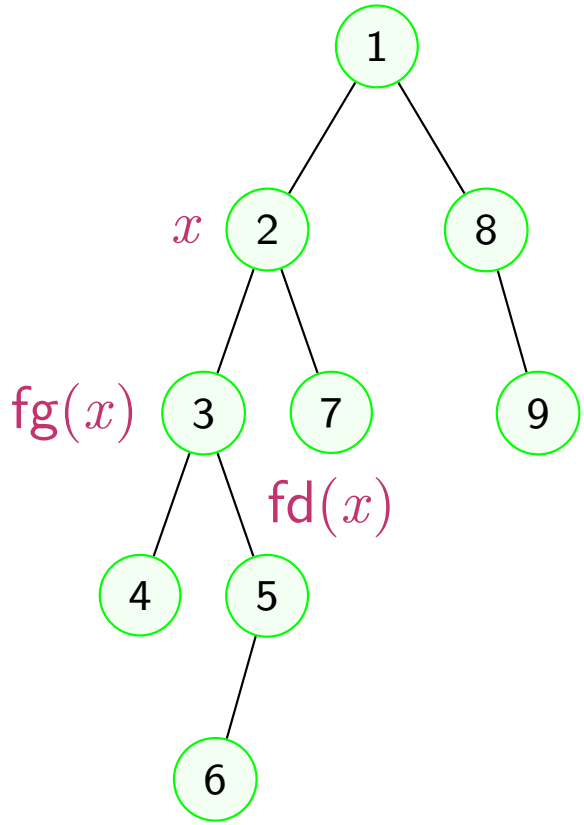
# Sous-arbre enraciné en un nœud $x$

Arbre situé « en dessous du nœud $x$ » ($x$ est inclus).



# Fils gauche, fils droit d'un nœud $x$

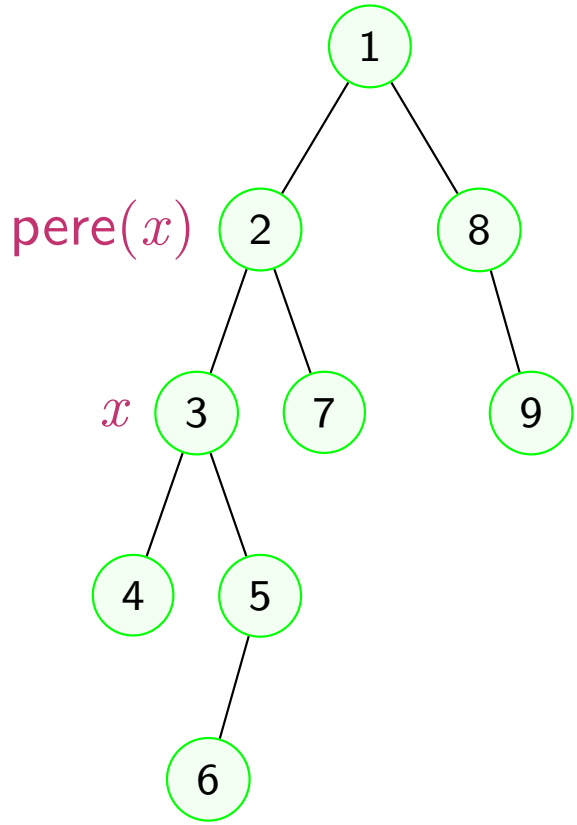
Fils gauche du nœud $x$, noté $\mathrm{fg}(x)$ = racine du sous-arbre gauche.




- Le nœud d'étiquette 5 a un fils gauche mais pas de fils droit,
- Celui d'étiquette 8 a un fils droit mais pas de fils gauche,
- Ceux d'étiquettes 4, 6, 7, 9 n'ont pas de fils.

# Père d'un nœud

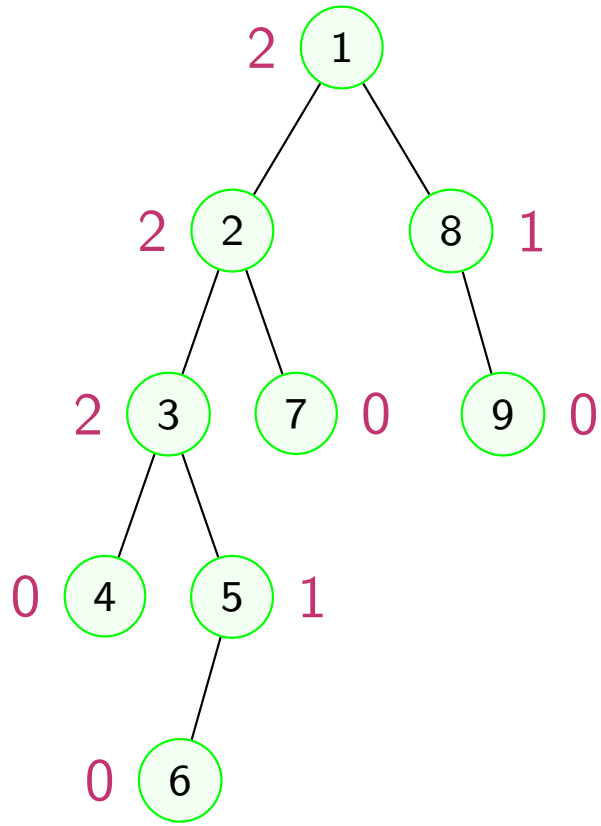
Le père d'un fils de $x$ est $x$.




Tout nœud sauf la racine a un (seul) père.

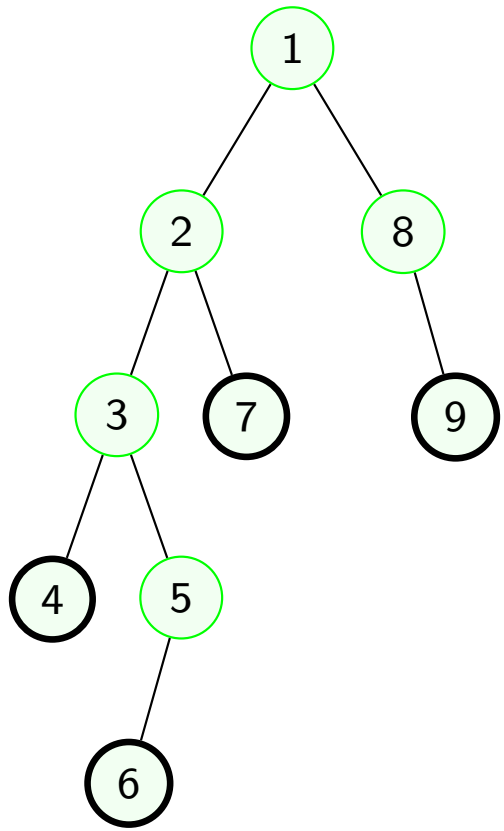
# Arité d'un nœud

Arité d'un nœud $x$ = nombre de fils de $x$.



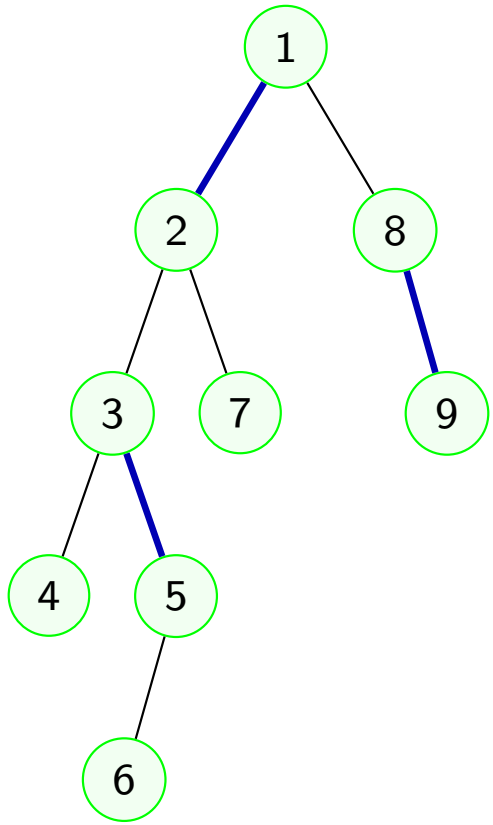
# Feuille

Feuille = nœud dont l'arité est **0**.



# Arête

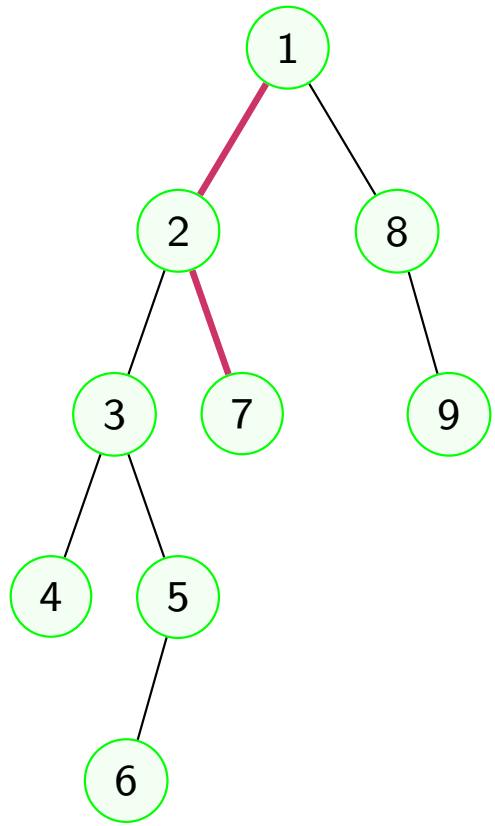
Relie un fils et son père.




Trois arêtes de cet arbre, en bleu.

# Branche

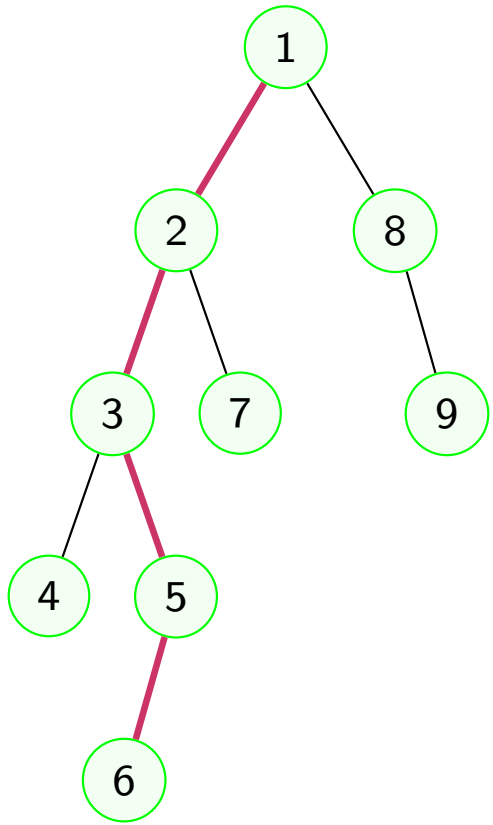
Chemin de la racine à une feuille.




- Exemple : $1 \to 2 \to 7$.
- Une branche **descend** de la **racine** jusqu'à une **feuille**.
- Longueur d'une branche = son nombre d'arêtes.

# Hauteur d'un arbre

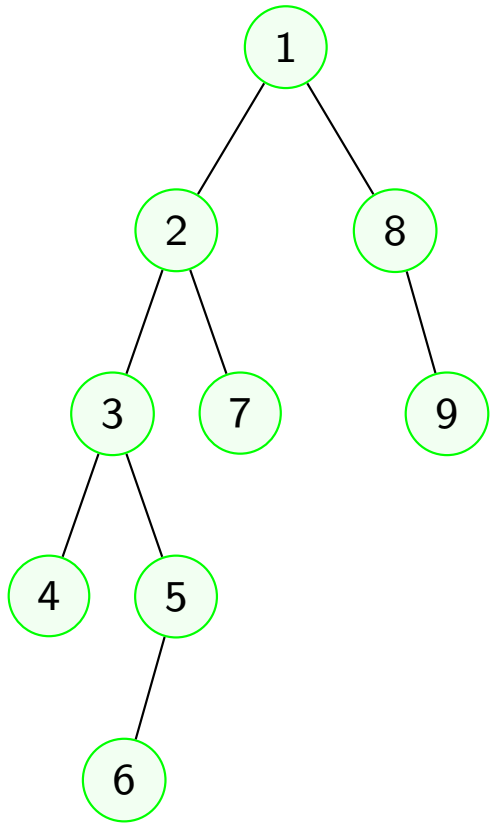
Longueur maximale d'une branche.




La hauteur de cet arbre est **4**.

# Taille d'un arbre

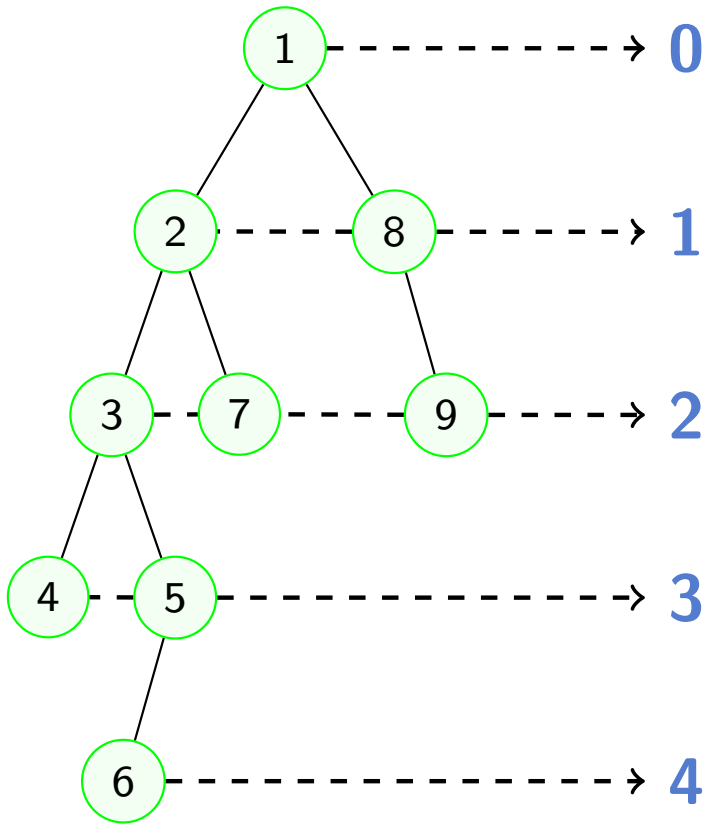
Nombre de nœuds.




Cet arbre a comme taille **9**.

# Profondeur (ou niveau) d'un nœud

Nombre d'arêtes qui séparent le nœud de la racine.




Le terme **niveau** est synonyme de **profondeur**.

# Arbres binaires particuliers

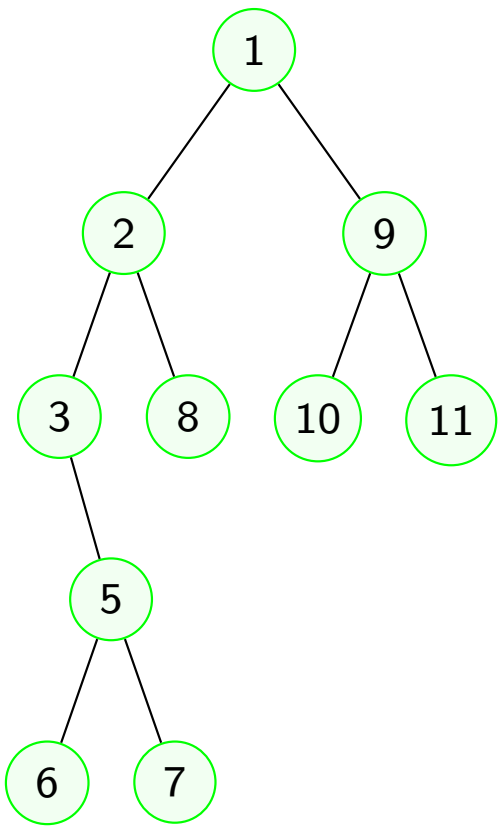
Un arbre binaire est

- **complet** s'il n'a pas de nœud d'arité 1.
- **parfait** s'il est complet et toutes les feuilles ont même niveau.
- **quasi-parfait** si
  - il est complet jusqu'au niveau $h-1$, et
  - ses feuilles sont à profondeur $h$ ou $h-1$, et
  - les feuilles de profondeur $h$ sont « le plus à gauche possible ».

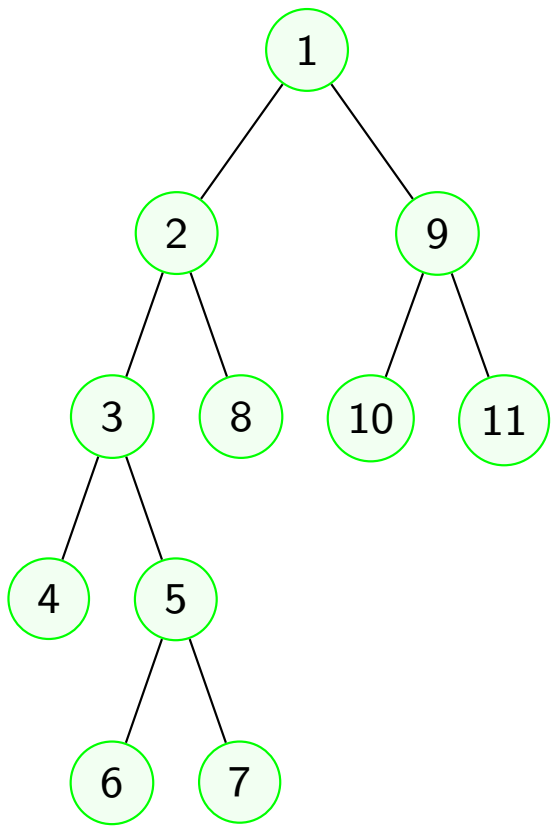
**Remarque** : tout arbre parfait est quasi-parfait.

# Arbre complet

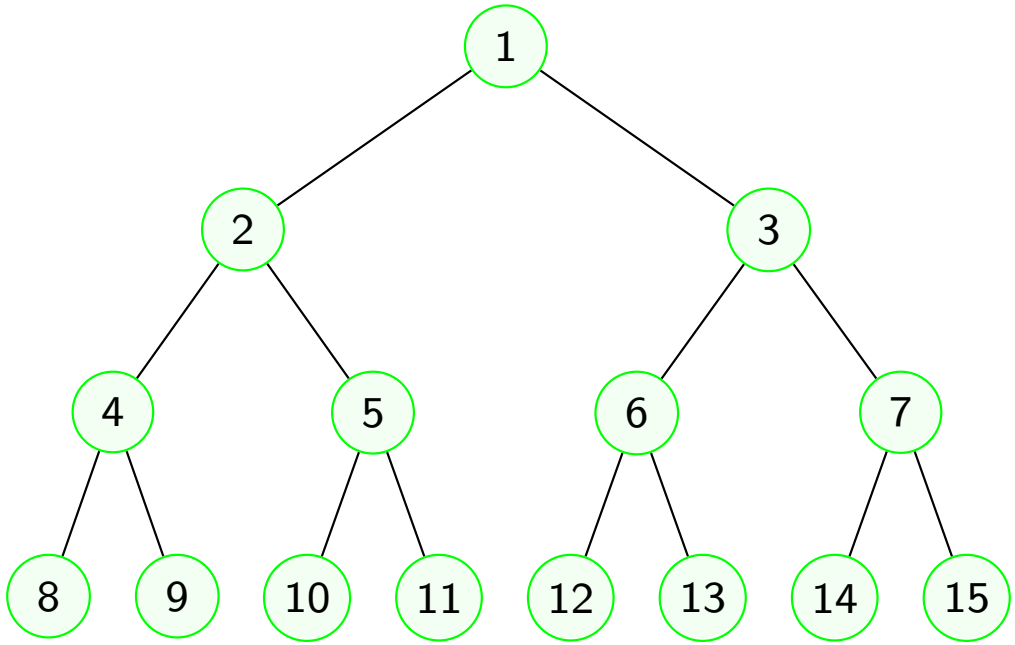
**Non complet**



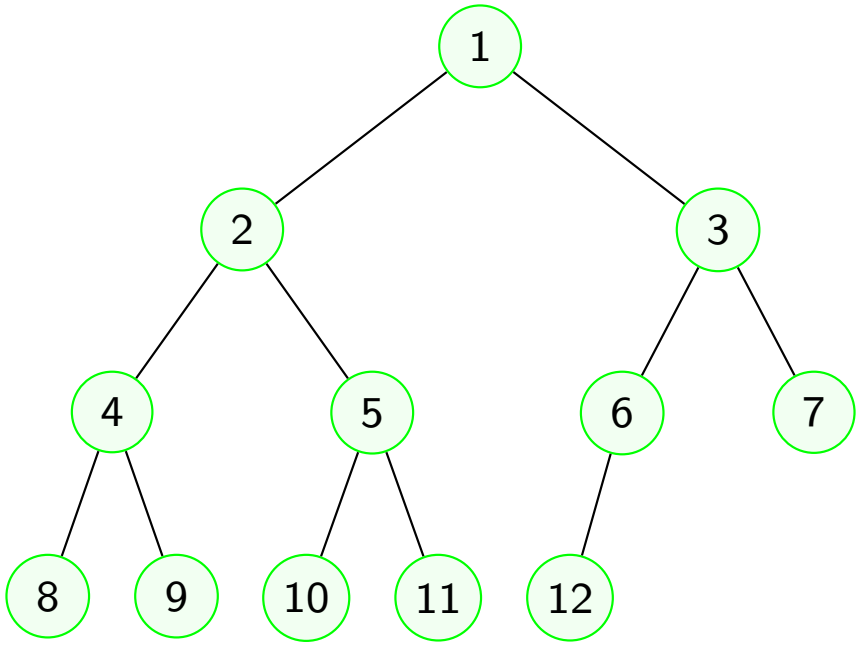

**Complet**



# Arbre parfait



# Arbre quasi-parfait



# Propriétés sur les arbres

- La hauteur, nombre de nœuds, nombre de feuilles, etc., vérifient des relations simples.
- Un **objectif** de l'UE est que vous devez **comprendre** comment montrer ces relations par récurrence.
- Récurrences : sur la taille ou la hauteur des arbres.

**Pourquoi une preuve formelle et comment rédiger ?**

# Tout arbre binaire non vide a au moins une feuille

On raisonne par **récurrence** sur la taille $n(t) = n$ de l'arbre $t$.

- Comme $t$ est non vide, on a $n \geqslant 1$.

# Tout arbre binaire non vide a au moins une feuille

On raisonne par **récurrence** sur la taille $n(t) = n$ de l'arbre $t$.

- Comme $t$ est non vide, on a $n \geqslant 1$.
- Si $n = 1$, alors la racine de $t$ est une feuille. ✓

# Tout arbre binaire non vide a au moins une feuille

On raisonne par **récurrence** sur la taille $n(t) = n$ de l'arbre $t$.

- Comme $t$ est non vide, on a $n \geqslant 1$.
- Si $n = 1$, alors la racine de $t$ est une feuille. ✓
- Supposons le résultat vrai pour tout arbre de taille $\leqslant n - 1$.

# Tout arbre binaire non vide a au moins une feuille

On raisonne par **récurrence** sur la taille $n(t) = n$ de l'arbre $t$.

- Comme $t$ est non vide, on a $n \geqslant 1$.
- Si $n = 1$, alors la racine de $t$ est une feuille. ✓
- Supposons le résultat vrai pour tout arbre de taille $\leqslant n - 1$.
- Soit $t$ de taille $n > 1$.
  - Comme $t$ est formé d'un nœud racine et de sous-arbres $\ell$ et $r$ :

$$n = 1 + n(\ell) + n(r). \tag{1}$$

où $n(\ell) =$ taille de $\ell$ et $n(r) =$ taille de $r$.

# Tout arbre binaire non vide a au moins une feuille

On raisonne par **récurrence** sur la taille $n(t) = n$ de l'arbre $t$.

- Comme $t$ est non vide, on a $n \geqslant 1$.
- Si $n = 1$, alors la racine de $t$ est une feuille. ✓
- Supposons le résultat vrai pour tout arbre de taille $\leqslant n - 1$.
- Soit $t$ de taille $n > 1$.
  - Comme $t$ est formé d'un nœud racine et de sous-arbres $\ell$ et $r$ :

$$n = 1 + n(\ell) + n(r). \tag{1}$$

    où $n(\ell) =$ taille de $\ell$ et $n(r) =$ taille de $r$.
  - Comme $n > 1$, soit $\ell$ soit $r$ n'est pas vide, par exemple $\ell$.

# Tout arbre binaire non vide a au moins une feuille

On raisonne par **récurrence** sur la taille $n(t) = n$ de l'arbre $t$.

- Comme $t$ est non vide, on a $n \geqslant 1$.
- Si $n = 1$, alors la racine de $t$ est une feuille. ✓
- Supposons le résultat vrai pour tout arbre de taille $\leqslant n - 1$.
- Soit $t$ de taille $n > 1$.
  - Comme $t$ est formé d'un nœud racine et de sous-arbres $\ell$ et $r$ :

$$n = 1 + n(\ell) + n(r). \tag{1}$$

    où $n(\ell) =$ taille de $\ell$ et $n(r) =$ taille de $r$.
  - Comme $n > 1$, soit $\ell$ soit $r$ n'est pas vide, par exemple $\ell$.
  - D'après l'équation (1), on a $n(\ell) \leqslant n - 1$.

# Tout arbre binaire non vide a au moins une feuille

On raisonne par **récurrence** sur la taille $n(t) = n$ de l'arbre $t$.

- Comme $t$ est non vide, on a $n \geqslant 1$.
- Si $n = 1$, alors la racine de $t$ est une feuille. ✓
- Supposons le résultat vrai pour tout arbre de taille $\leqslant n - 1$.
- Soit $t$ de taille $n > 1$.
  - Comme $t$ est formé d'un nœud racine et de sous-arbres $\ell$ et $r$ :

$$n = 1 + n(\ell) + n(r). \tag{1}$$

    où $n(\ell) =$ taille de $\ell$ et $n(r) =$ taille de $r$.
  - Comme $n > 1$, soit $\ell$ soit $r$ n'est pas vide, par exemple $\ell$.
  - D'après l'équation (1), on a $n(\ell) \leqslant n - 1$.
  - Par hypothèse de récurrence, $\ell$ a au moins une feuille.

# Tout arbre binaire non vide a au moins une feuille

On raisonne par **récurrence** sur la taille $n(t) = n$ de l'arbre $t$.

- Comme $t$ est non vide, on a $n \geqslant 1$.
- Si $n = 1$, alors la racine de $t$ est une feuille. ✓
- Supposons le résultat vrai pour tout arbre de taille $\leqslant n - 1$.
- Soit $t$ de taille $n > 1$.
  - Comme $t$ est formé d'un nœud racine et de sous-arbres $\ell$ et $r$ :

$$n = 1 + n(\ell) + n(r). \tag{1}$$

    où $n(\ell) =$ taille de $\ell$ et $n(r) =$ taille de $r$.
  - Comme $n > 1$, soit $\ell$ soit $r$ n'est pas vide, par exemple $\ell$.
  - D'après l'équation (1), on a $n(\ell) \leqslant n - 1$.
  - Par hypothèse de récurrence, $\ell$ a au moins une feuille.
  - Cette feuille est aussi une feuille de $t$. ✓

# Propriétés des arbres binaires

Si $t$ est un arbre, on note

- $h(t)$ sa hauteur,
- $n(t)$ son nombre de nœuds.
- $i(t)$ son nombre de nœuds internes.
- $\ell(t)$ son nombre de feuilles.

**Comment calculer** récursivement ces paramètres ?

# Propriétés

1. On a $h(t) + 1 \leqslant n(t) \leqslant 2^{h(t)+1} - 1$.
2. L'arité de tous les nœuds internes de $t$ est 1 si et seulement si

$$h(t) + 1 = n(t).$$

3. L'arbre $t$ est parfait si et seulement si

$$n(t) = 2^{h(t)+1} - 1.$$

4. Si $t$ est un arbre **complet**, on a $i(t) = \ell(t) - 1$.

# Parcours en profondeur

- But : effectuer un traitement sur tous les nœuds d'un arbre.
- Se programme très simplement de façon récursive.
- **Postfixe** :
  - Parcourir le sous-arbre gauche,
  - Parcourir le sous-arbre droit,
  - Effectuer le traitement sur la racine.
- **Infixe**.
- **Préfixe**.

# Parcours prefixe, infixe, postfixe : exemple

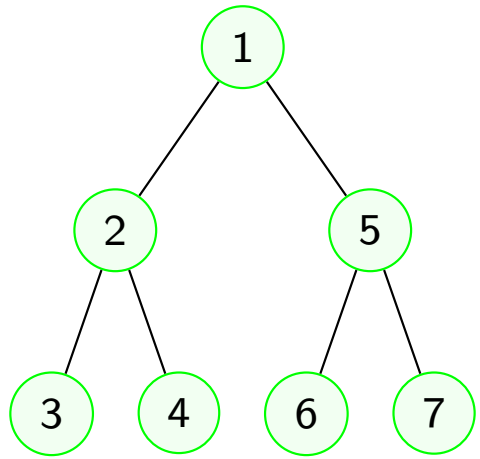



FIG. : P2 : Arbre parfait de hauteur 2

Dans quel ordre les nœuds sont-ils traités ?

- Préfixe : 1 2 3 4 5 6 7
- Infixe : 3 2 4 1 6 5 7
- Postfixe : 3 4 2 6 7 5 1

# Pourquoi OCaml ?

Bien adapté à la récursion, et les arbres sont une structure récursive.

 Pas besoin d'allouer ni de désallouer.

 Démo !

# Complexité d'un algorithme

Il est souhaitable qu'un algorithme soit correct **et efficace**.

- La **complexité** d'un algorithme mesure son **efficacité**.
- Un algorithme doit fonctionner sur de multiples entrées.
- Intuitivement : plus une entrée est grande, plus l'algorithme risque de mettre du temps pour la traiter.
- La **complexité (dans le cas le pire)** d'un algorithme est une fonction qui à $n \in \mathbb{N}$ associe le nombre maximal d'opérations élémentaires faites par l'algorithme sur les entrées **de taille** $n$.
- Opérations considérées comme **élémentaires** : affectation, arithmétique $+, -, *, /$ et comparaisons d'entiers, etc.

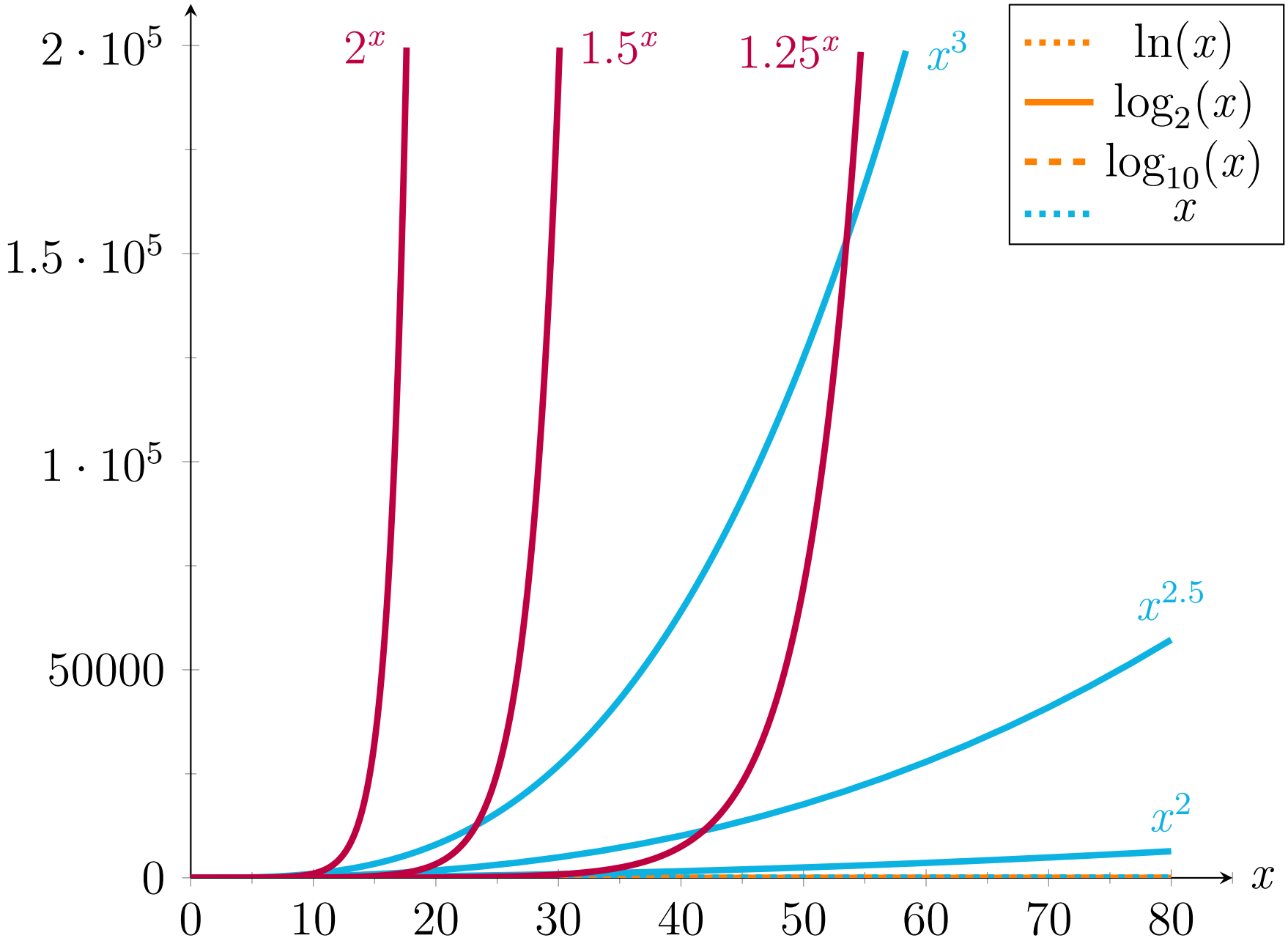
# Fonctions importantes

En pratique, les fonctions de complexité sont construites à l'aide :

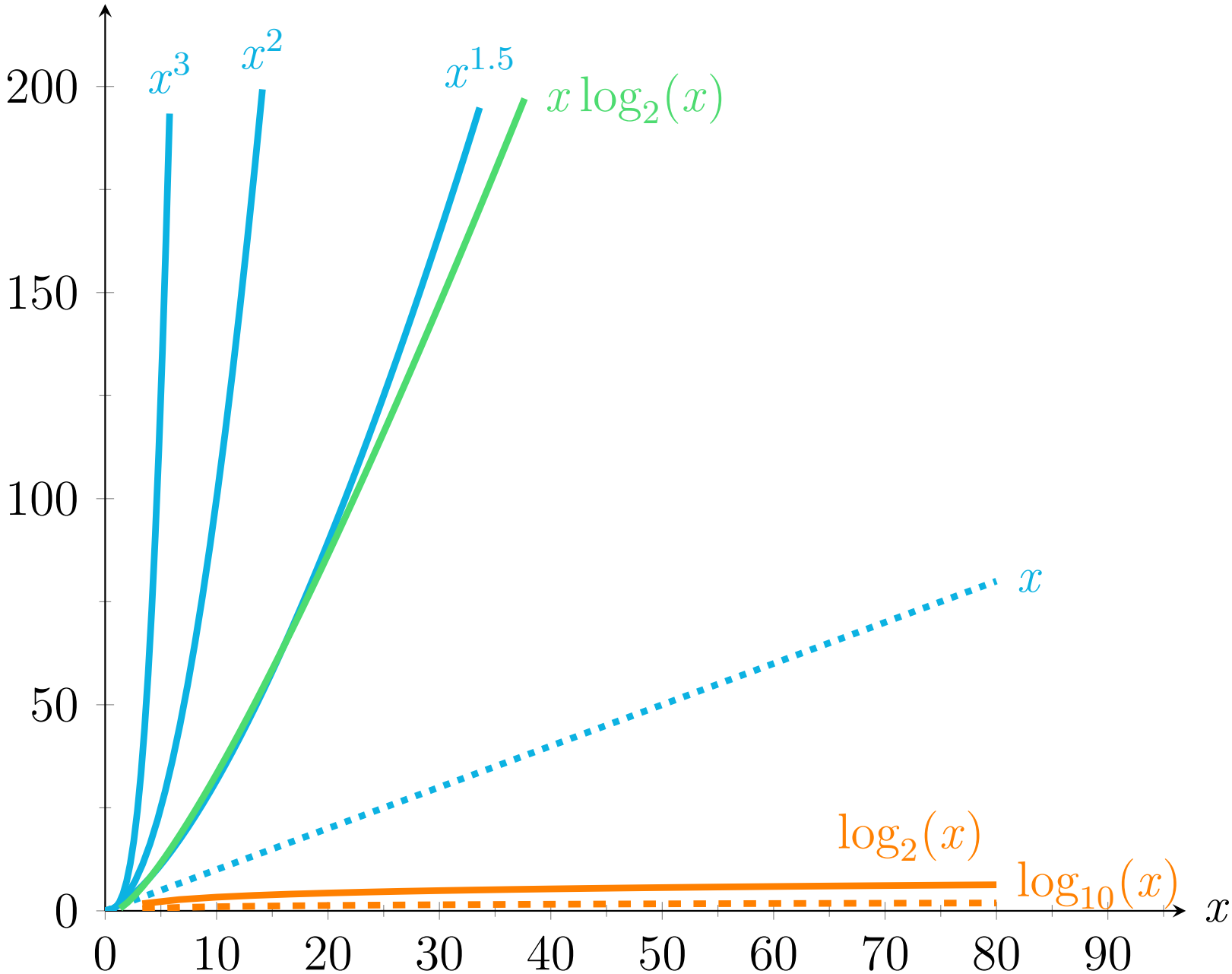
- des fonctions puissance $n \longmapsto n^k$, souvent pour $k \in \mathbb{N}$,
- des fonctions logarithme $n \longmapsto \log_2(n)$ ou $n \longmapsto \log_{10}(n)$.
- des fonctions exponentielle $n \longmapsto a^n$ pour $a > 1$, souvent entier.

Il faut avoir une intuition de leur rapidité de croissance.

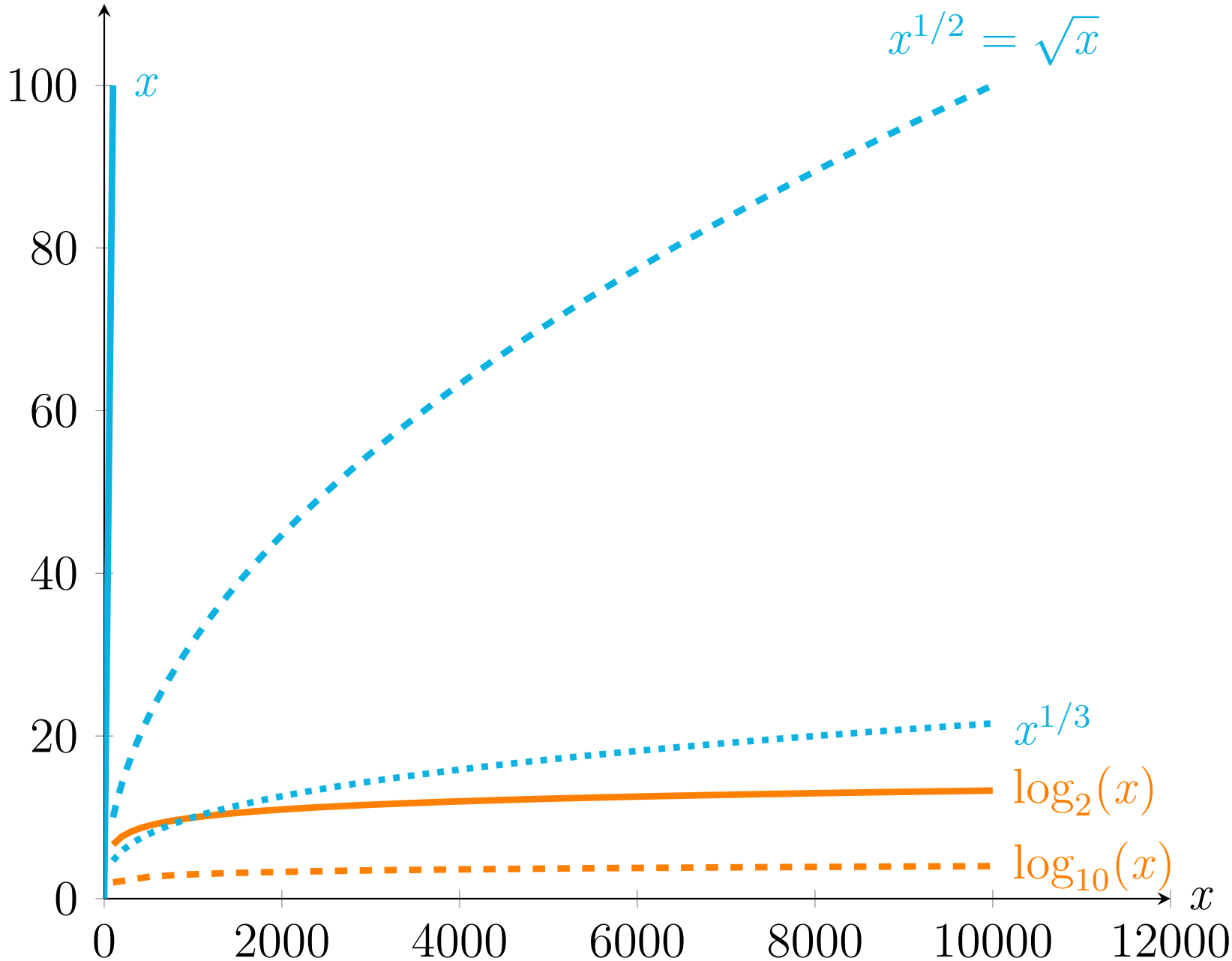
# Croissance des fonctions classiques



# Croissance des fonctions classiques (2)



# Croissance des fonctions classiques (3)



# Fonction logarithme

## A retenir :

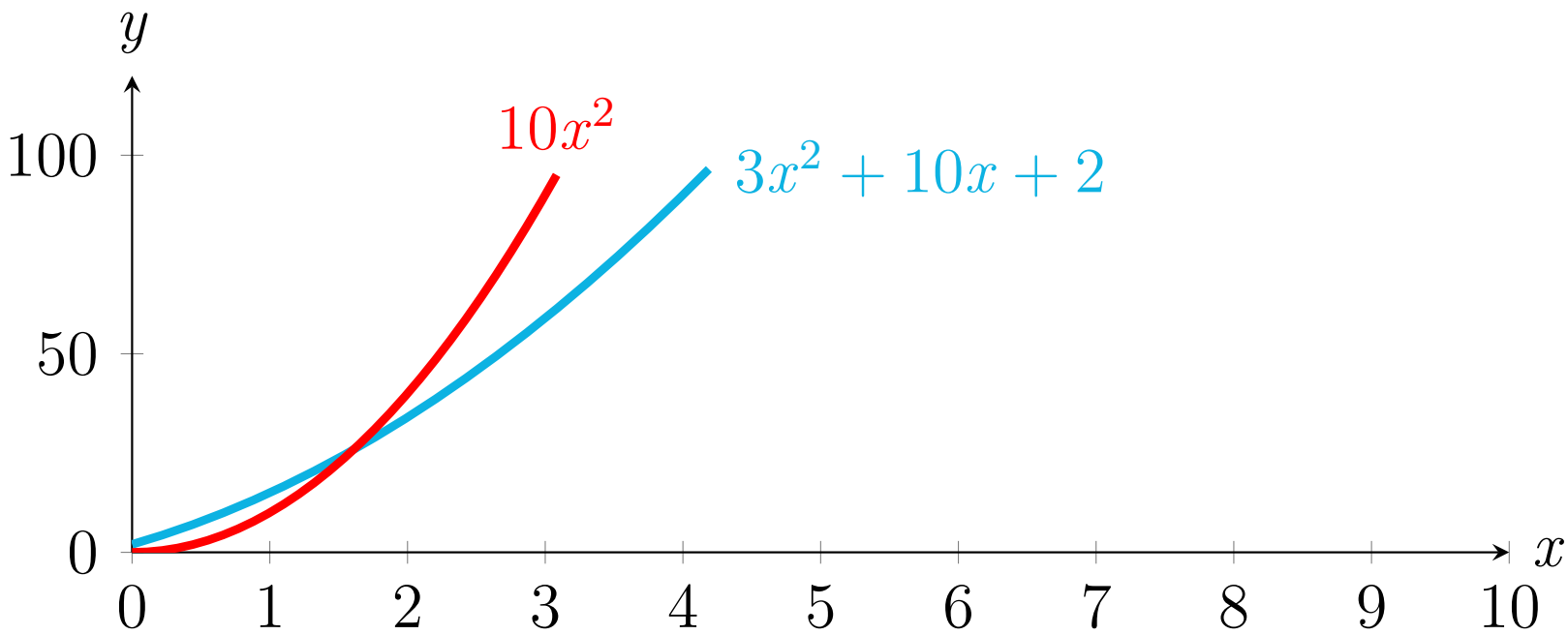
- $\log_{10}(10^x) = x$ et $10^{\log_{10}(x)} = x$.
- $\log_2(2^x) = x$ et $2^{\log_2(x)} = x$.
- $\lfloor \log_{10}(x) \rfloor$ = (nombre de chiffres de $x$, moins 1).
  où $\lfloor x \rfloor$ est la partie entière de $x$ ($\lfloor 3\,,14 \rfloor = \lfloor 3\,,9 \rfloor = \lfloor 3 \rfloor = 3$).
  - Par exemple : $6 \leqslant \log_{10}(1234567) < 7$.
- $\log_2(x) = \log_2(10) \cdot \log_{10}(x) \simeq 3\,,32 \log_{10}(x)$.
  - Les fonctions $\log_2$ et $\log_{10}$ sont proportionnelles.
- $\log_2(x)$ croît moins vite que toute fonction $x^a$ avec $a > 0$.
  - Formellement, $\lim\limits_{x \to \infty} \dfrac{\log_2^b(x)}{x^a} = 0$ si $a > 0$.

# Notation O()

Si $f, g$ sont des fonctions de $\mathbb{N}$ dans $\mathbb{N}$, on écrit $f = O(g)$ si

$$\exists C > 0,\ \exists N > 0,\ \forall n, \quad n \geqslant N \implies f(n) \leqslant Cg(n)$$

- **Exemple** : $3n^2 + 10n + 2 = O(n^2)$ car à partir de $N = 2$, $3n^2 + 10n + 2 \leqslant 10n^2$.



# Utilisation de la notation $O()$

Sert à **simplifier** des expressions de façon **raisonnable**.

- Si $n \geqslant 1$, on a $7n^2 + 2n + 1 \leqslant 7n^2 + 2n^2 + n^2 = 10n^2$ donc

$$7n^2 + 2n + 1 = O(n^2)$$

- De même, si $n \geqslant 1$ :
  $25n^7 + 12n^3 + 4n\log_2(n) + 1 \leqslant 25n^7 + 12n^7 + 4n^7 + n^7 = 42n^7$,
  donc

$$25n^7 + 12n^3 + 4n\log_2(n) + 1 = O(n^7)$$

# Utilisation de la notation $O()$ – (2)

- Si on a calculé une complexité en $n \log_2(n)$, on n'utiliserait pas $n \log_2(n) = O(n^2)$, même si c'est vrai, parce que
  - $n \log_2(n)$ est une expression simple,
  - on perd trop de précision en majorant par $n^2$.

- Par contre, une complexité en $\frac{1}{2}n^2 + 7n \log_2(n) + 12n + 8$ se majore par $Cn^2$ à partir de $n = 1$ (pour $C = 28$), donc

$$\frac{1}{2}n^2 + 7n \log_2(n) + 12n + 8 = O(n^2)$$

# Notation O() : formules utiles

- Complexité $O(1) =$ temps constant.
- Si $a < b$, on a $n^a = O(n^b)$.
- Si $a > 0$ et $c > 1$, on a $\log_c(n) = O(n^a)$.
- si $a > 0$ et $c > 1$, on a $n^a = O(c^n)$.

# Évaluer la complexité d'un algorithme

**Comment évaluer une complexité et rédiger ?**

- Exemple : tri par insertion