Algorithmique des arbres, Ocaml
L2, 2ème semestre

Version du 12 février 2021

http://dept-info.labri.fr/ENSEIGNEMENT/asda/
uf-info.ue.algo-arbres@diff.u-bordeaux.fr

Table des matières

1 Introduction **2**
1 Motivation : les arbres comme structures de données . . . 2
1.1 Les arbres : utilisation et définitions informelles . . . 2
2 Définitions . . . 5
2.1 Arbres binaires . . . 5
2.2 Vocabulaire standard sur les arbres binaires . . . 7
2.3 Arbres binaires particuliers . . . 9
3 Propriétés importantes des arbres binaires . . . 10
4 Représentation d'ensembles et de multi-ensembles . . . 11
5 Algorithmes génériques fondamentaux . . . 11
6 Références . . . 12

2 Prérequis mathématiques **13**
1 Comment rédiger une preuve par récurrence ? . . . 13
2 Complexité . . . 14
2.1 La fonction logarithme . . . 14
2.2 Quelques graphiques . . . 15
2.3 La notation $O()$ . . . 18

Chapitre 1

Introduction

Ce cours se situe dans l'apprentissage des structures de données. Nous allons aborder et approfondir la notion d'arbre, en commençant par des rappels sur les tableaux et les listes. L'objectif est de comprendre pourquoi et dans quels cas la structure d'arbre permet d'améliorer les algorithmes, que ce soit en conceptualisation, en simplicité ou en efficacité.

1 Motivation : les arbres comme structures de données

Les arbres sont une structure de données très utilisée en informatique, c'est-à-dire une façon d'organiser les données. Cette organisation doit permettre d'effectuer les tâches suivantes : tester si la structure de données est vide, tester si une donnée est présente (et y accéder), ajouter ou supprimer une donnée.

D'autres structures étudiées au semestre précédent permettent déjà d'effectuer ces tâches. C'est le cas par exemple des listes, des piles, des files et des tableaux. Les arbres sont une structure de données plus compliquée et plus élaborée. Cette complication apporte cependant un gain : les opérations sont en général bien plus rapides lorsqu'on utilise des arbres.

Le mot « arbre » est en fait un terme générique, qui désigne plusieurs sortes de structures de données différentes. Nous verrons quelques unes de ces variantes au cours de ce document. Les arbres permettent d'organiser l'information de façon plus élaborée que dans le cas de structures de données linéaires (telles les tableaux ou les listes). Cette plus grande complexité a un *coût* et un *gain* :

- Le coût à payer est une plus grande difficulté de manipulation des arbres par rapport à celle des structures linéaires.
- Le gain apporté est une facilité de modélisation, lorsqu'on veut représenter des structures *hiérarchiques* pour lesquelles les arbres sont très bien adaptés, ainsi qu'une bien meilleure efficacité algorithmique.

Ce court chapitre introductif présente quelques utilisations des arbres en informatique et quelques variations sur la structure de donnée d'arbre. Cette section donne des exemples informatiques où les arbres apparaissent naturellement. La Section 2 est importante : elle décrit précisément la structure de donnée « arbre binaire », et introduit plusieurs définitions qui seront utilisées dans ce cours. La Section 3 présente quelques propriétés simples des arbres binaires, qui sont utiles à la fois pour montrer que des algorithmes sont corrects et pour évaluer leur efficacité. La Section 4 explique la notion de multi-ensemble, que nous voulons représenter grâce à cette structure de données. La Section 5 liste quelques tâches importantes et fréquemment effectuées sur les arbres binaires.

1.1 Les arbres : utilisation et définitions informelles

La notion d'arbre est introduite dans cette section avec des applications concrètes. Nous en profitons pour introduire des termes importants, à retenir. Ils sont écrits avec cette ***police de caractères***.

Vocabulaire et premier exemple : système de fichiers

Les arbres sont particulièrement adaptés à la représentation et à la manipulation d'information déjà *hiérarchique*. Par exemple, un système de fichiers a une structure hiérarchique : un répertoire peut contenir des sous-répertoires ou fichiers, qui eux-mêmes peuvent contenir des sous-répertoires ou fichiers, etc. On peut représenter des répertoires et fichiers par un arbre, comme indiqué en Fig. 1.1.

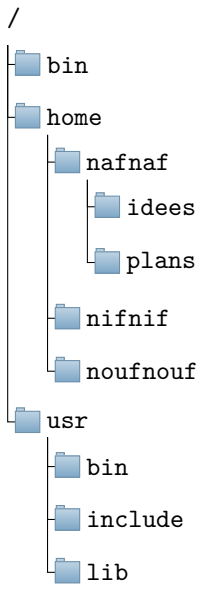


Fig. 1.1 : Vue hiérarchique de répertoires

L'usage en informatique est de dessiner cette structure avec la racine `/` du système de fichiers *en haut*, et en indiquant les répertoires au-dessous de celui qui les contient. On obtient la représentation de la Fig. 1.2. Chaque répertoire correspond à un *nœud* de l'arbre. Distinguer le nœud ***racine*** (ici `/`) permet de parler de relations de parenté entre nœuds :

- Chaque nœud, sauf la racine, a un *unique* ***père***. Dans l'exemple concret du système de fichiers, le père d'un répertoire (autre que la racine) est le répertoire qui le contient. Ainsi, le père du seul nœud dont le nom est `include` est le nœud `usr`.
- Inversement, chaque nœud peut avoir (ou non) des ***fils***. Par exemple, le nœud `usr` a 3 fils (nommés `bin`, `include` et `lib`). En revanche, le nœud `include` n'a aucun fils. Un nœud qui n'a pas de fils est appelé une ***feuille***.
- Un nœud qui n'est pas une feuille est appelé ***nœud interne***.
- L'***arité*** d'un nœud est son nombre de fils. Les feuilles sont donc les nœuds d'arité 0.
- Enfin, deux nœuds qui ont même père sont appelés ***frères***.

Dans la Fig. 1.2, on a indiqué chaque lien de parenté par une flèche du père vers le fils. En général, on omet les flèches, qui sont redondantes avec le choix de placer la racine en haut, et de « faire croître » les arbres vers le bas.

Remarquez également que les noms des répertoires sont de l'information attachée aux nœuds, ils ne désignent pas les nœuds eux-mêmes. En particulier, le même nom peut être utilisé pour 2 nœuds différents. Sur cet exemple, c'est le cas : deux des répertoires s'appellent `bin` (pour un arbre qui représente un système de fichiers, deux frères ne peuvent bien sûr pas avoir le même nom).

Étant donné un nœud d'un arbre, il y a un unique chemin reliant ce sommet à la racine et ne passant pas 2 fois par le même sommet. Une ***branche*** est un chemin qui va de la racine à une feuille. Par exemple, `/` → `home` → `nafnaf` → `plans` est une branche. Il y a donc autant de branches que de feuilles. Enfin,

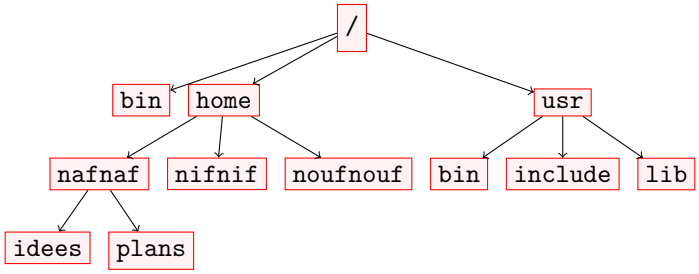


FIG. 1.2 : Représentation arborescente du même système de fichiers

un ***sous-arbre*** enraciné à un nœud particulier est l'arbre obtenu en ne conservant que le nœud et ses descendants. Par exemple, le sous-arbre enraciné au seul nœud marqué `usr` est l'arbre de la FIG. 1.3 suivante.

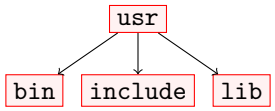


FIG. 1.3 : Sous-arbre enraciné au nœud marqué `usr`

Notez enfin que certains détails sont pour l'instant ignorés. Par exemple, doit-on considérer que l'ordre des fils a une importance, ou au contraire, que permuter deux fils sur la figure représente le même arbre ?

Dans ce cours, on utilisera des arbres binaires dans lesquels l'ordre des fils est important. Un ***arbre*** est dit ***binaire*** lorsque tout nœud a 0, 1 ou 2 fils. Comme on utilisera des arbres dans lesquels l'ordre des fils importe, on distingue le ***fils gauche*** et le ***fils droit*** de tout nœud interne. De la même façon, on parle de ***sous-arbre gauche*** d'un nœud interne le sous-arbre enraciné en son fils gauche (idem pour le ***sous-arbre droit***).

Des utilisations très diverses

Les arbres ont de multiples utilisations en informatique. Nous en mentionnons quelques-unes dans cette partie. Une utilisation très courante des arbres est fournie par le format XML (eXtended Markup Language). Il s'agit d'un format standard de documents, utilisé dans de multiples contextes pour stocker ou transmettre de l'information. Le code suivant est un exemple de fichier XML. Le format est utilisé dans de nombreuses applications, comme les bases de données, les données GPS, le dessin vectoriel, etc. Vous pouvez en visualiser d'autres, par exemple en regardant des traces fournies par un GPS, ou les fichiers générés par un logiciel de dessin vectoriel (comme Inkscape).

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book isbn="978-0201896831">
    <title>The Art of Computer Programming, Fundamental Algorithms</title>
    <author>Donald E. Knuth</author>
    <year>1997</year>
    <volume>1</volume>
    <price>65</price>
  </book>
  <book isbn="978-2258034310">
    <title>Calvin et Hobbes, tome 1 : Adieu, monde cruel !</title>
    <author>Bill Watterson</author>
    <year>1999</year>
    <volume>1</volume>
    <price>10</price>
```

```
  </book>
</bookstore>
```

Ce code se représente naturellement par un arbre, donné en FIG. 1.4, où les valeurs concrètes ont été omises.

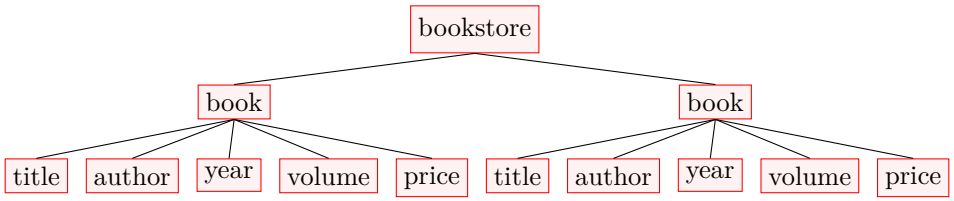


FIG. 1.4 : Une représentation du document XML

De façon générale, les arbres sont utilisés pour représenter et manipuler de l'information naturellement hiérarchique, mais aussi pour rendre plus rapide l'accès à l'information, tout en conservant un temps d'accès raisonnable aux données. Les domaines d'applications des arbres en informatique ne se limitent pas aux deux exemples déjà rencontrés. Ils sont utilisés dans des domaines aussi variés que les algorithmes de routage, la compression de données, des algorithmes de fouille de données et d'apprentissage, des algorithmes de résolution de jeux (comme les échecs), ou des algorithmes de rendu de jeux 3D, voir ici et là. Ils interviennent aussi dans la représentation de grandes bases de données, la représentation de certains formats de document (nous avons vu XML, mais le format PDF utilise aussi les arbres, par exemple). Enfin, ils interviennent pour approcher des problèmes difficiles sur les graphes (et cette liste d'applications n'est pas exhaustive). La structure d'arbre est donc réellement fondamentale en informatique.

2 Définitions

Cette section introduit formellement le vocabulaire utilisé dans ce document. On distingue plusieurs sortes d'arbres. Les arbres les plus simples sont les arbres binaires, dans lesquels chaque nœud interne a au plus 2 fils.

2.1 Arbres binaires

La définition d'un arbre binaire étiqueté est *récursive*. Un ***arbre binaire étiqueté*** est :

- soit l'arbre vide,
- 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.

Le mot ***sommet*** est un synonyme du mot ***nœud***. Un arbre non vide t est donc décrit par un triplet (v, ℓ, r) formé :

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

Dans ce cours, les arbres sont utilisés comme structure de données, pour stocker et organiser de l'information. C'est pourquoi les arbres que nous considérons sont souvent *étiquetés*, c'est-à-dire que chaque nœud porte une valeur d'un type donné (par exemple, entier), que l'on appelle son ***étiquette***. On appellera ***squelette*** d'un ***arbre binaire*** la structure obtenue en supprimant les étiquettes des nœuds. Un ***squelette d'arbre binaire*** est donc :

- soit l'arbre vide,
- soit un nœud appelé la ***racine*** de l'arbre, formé
  - d'un arbre appelé ***sous-arbre gauche***,
  - d'un arbre appelé ***sous-arbre droit***.

La structure de donnée « arbre binaire » se représente très naturellement par des figures. La définition du type *arbre binaire* étant récursive, il est naturel de définir récursivement sa représentation graphique. On doit d'abord choisir une convention pour représenter l'arbre vide. Dans ce document, on le représentera par un carré (c'est juste une convention adoptée dans ce document, qu'on est bien sûr libre de changer).

FIG. 1.5 : L'arbre vide

Pour dessiner un arbre non vide, on procède récursivement :

- on dessine la racine en haut de la figure,
- on dessine en dessous à gauche le sous-arbre gauche, et on relie la racine de l'arbre à celle de son sous-arbre gauche par une arête.
- on dessine en dessous à droite le sous-arbre droit, et on relie la racine de l'arbre à celle de son sous-arbre droit par une arête.

Si l'arbre est étiqueté, on indique les valeurs des étiquettes dans les nœuds. On peut par exemple obtenir le dessin de la FIG. 1.6. La convention utilisée dans ce document est de dessiner chaque nœud dans un cercle. Le squelette d'un arbre étiqueté, obtenu en supprimant les étiquettes, sera représenté sans information à l'intérieur des nœuds.

Attention : même si on représente parfois l'arbre vide sur les figures, l'arbre vide ***n'est pas*** un nœud. L'arbre de la FIG. 1.6 a donc seulement ***4 nœuds***.

(a) Un arbre binaire (b) Son squelette

FIG. 1.6 : Un arbre binaire et son squelette

Cette figure représente l'arbre dont la racine est étiquetée par 1, et dont les sous-arbres gauche et droit sont ceux indiqués sur la FIG. 1.7.

En pratique, il est souvent inutile de représenter les sous-arbres vides sur les figures, parce qu'on peut déduire où ils devraient se trouver. L'arbre de la FIG. 1.6 (a) est donc aussi représenté comme sur la FIG. 1.8.

Dans ce cours, l'ordre des fils a une importance : on distingue le sous-arbre gauche du sous-arbre droit. Ainsi, l'arbre de la FIG. 1.9 n'est pas le même que celui de la FIG. 1.8.

(a) Sous-arbre gauche (b) Sous-arbre droit

FIG. 1.7 : Sous-arbres gauche et droit de l'arbre de la FIG. 1.6 (a)

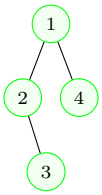


FIG. 1.8 : Un arbre binaire, sans représentation explicite des sous-arbres vides

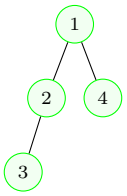


FIG. 1.9 : Un arbre binaire, sans représentation explicite des sous-arbres vides

De même, les deux arbres (non étiquetés) de la FIG. 1.10, sont symétriques, mais différents.

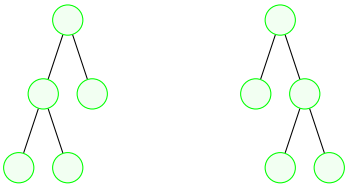


FIG. 1.10 : Deux arbres symétriques mais différents

Sur les figures, il faut donc porter attention à l'orientation gauche-droite des arêtes (une ***arête*** relie deux nœuds), en particulier lorsqu'un nœud n'a qu'un seul sous-arbre non vide : le fait qu'il à gauche ou à droite est important. Par exemple, les arbres des FIG. 1.8 et 1.9 ne diffèrent que parce que le nœud d'étiquette 3 est fils droit dans l'un et fils gauche dans l'autre (voir la Section 2.2 ci-dessous pour la définition de fils gauche et droit).

2.2 Vocabulaire standard sur les arbres binaires

Important

Lisez attentivement les termes définis dans cette section : ils serviront tout au long du cours.

Dans un arbre, chaque sommet s définit un sous-arbre dont il est la racine. On appelle cet arbre le ***sous-arbre enraciné en*** s. Par exemple, sur l'arbre de la FIG. 1.6 (a), le sous-arbre enraciné au nœud étiqueté 2 est le sous-arbre gauche, représenté en FIG. 1.7 (a). Le sous-arbre enraciné au nœud étiqueté 3 est simplement constitué de ce nœud, avec deux sous-arbres gauche et droit vides.

Le ***fils gauche*** d'un sommet s est la racine du sous-arbre gauche de l'arbre enraciné en s. Le fils gauche d'un sommet s n'existe donc que si le sous-arbre gauche de l'arbre enraciné en s est non vide. De même, le ***fils droit*** d'un sommet s est la racine du sous-arbre droit de l'arbre enraciné en s, et il n'existe que si ce sous-arbre est non vide. Par exemple, sur l'arbre de la FIG. 1.6 (a), le nœud étiqueté 2 n'a pas de fils gauche, et son fils droit est le nœud étiqueté 3.

Si un sommet s a pour fils (gauche ou droit) un sommet t, on dit que s est le ***père*** de t. Par définition, chaque sommet a exactement un père, sauf la racine qui n'en a pas. Par exemple, sur l'arbre de la FIG. 1.6 (a), le père du nœud étiqueté 3 est celui étiqueté 2, qui a lui-même comme père la racine.

L'***arité*** d'un nœud est son nombre de fils. Dans un arbre binaire, chaque nœud a donc une arité valant 0, 1 ou 2. Par exemple, sur la FIG. 1.6 (a), l'arité de la racine est 2, l'arité du nœud étiqueté 2 est 1, et l'arité des nœuds étiquetés 3 et 4 est 0.

Une ***feuille*** est un nœud d'arité 0. Un ***nœud interne*** (ou ***sommet interne***) est un nœud qui n'est pas une feuille, c'est-à-dire qui a au moins un fils.

Sur les figures, chaque nœud est relié à son père par une ***arête***. Une ***branche*** d'un arbre est une suite de sommets reliés par des arêtes allant de la racine à une feuille, sans repasser deux fois par le même nœud. Autrement dit, une branche est une suite de sommets $(s_0, s_1, \ldots, s_k)$ telle que

- s_0 est la racine de l'arbre,
- pour chaque i entre 1 et k, s_{i-1} est le père de s_i.

Attention, il ne faut pas confondre *branche* et *arête*. Par exemple, l'arbre de la FIG. 1.11 a trois arêtes : (1,2), (2,3) et (1,4). Mais il a seulement deux branches : la branche $(1, 2, 3)$ allant de la racine à la feuille étiquetée 3 (indiquée en pointillés sur la figure), et la branche (1.4) allant de la racine à la feuille étiquetée 4.

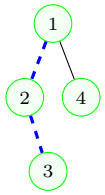


FIG. 1.11 : La branche associée à la feuille 3

Comme il y a exactement une branche menant à chaque feuille, il y a autant de branches que de feuilles. L'arbre vide, qui n'a aucun nœud, donc aucune feuille, n'a donc aucune branche. Si un arbre a un seul sommet s (qui est donc à la fois sa racine et une feuille), il a une unique branche : (s).

La ***longueur*** d'une branche est le nombre d'arêtes qu'elle contient. Par exemple, la longueur de la branche allant de la racine au nœud étiqueté 3 sur la FIG. 1.11 est égale à 2.

La ***hauteur*** d'un arbre est la longueur maximale d'une de ses branches. Par convention, la hauteur de l'arbre vide est -1. La hauteur d'un arbre qui ne consiste qu'en un seul sommet s est 0, puisque la seule branche de l'arbre, (s), ne contient aucune arête. La hauteur de l'arbre de la FIG. 1.11 est 2.

La ***taille*** d'un arbre est son nombre de nœuds. La taille de l'arbre vide est donc 0. La taille de l'arbre de la FIG. 1.11 est 4.

Enfin, la ***profondeur*** (ou le ***niveau***) d'un nœud dans un arbre est le nombre d'arêtes qui sépare le nœud de la racine de l'arbre. C'est aussi le nombre de fois qu'il faut remonter au nœud père pour arriver à la racine. En particulier :

- la profondeur de la racine d'un arbre non vide est toujours 0,
- la profondeur d'une feuille est la longueur de la branche allant de la racine jusqu'à cette feuille.

Sur l'arbre de la FIG. 1.11, la profondeur du nœud étiqueté 1 est 0, celle des nœuds étiquetés 2 et 4 est 1, et celle du nœud étiqueté 3 est 2.

Remarque. 1.1 Nous verrons plus tard dans ce cours des arbres dont l'arité des nœuds n'est pas limitée à 0, 1 ou 2. Les définitions précédentes s'adaptent naturellement (sauf les définitions de fils gauche et fils droit, qui sont spécifiques aux arbres binaires).

2.3 Arbres binaires particuliers

Important

Les termes définis en Section 2.2 sont relativement standardisés. Ce n'est pas le cas des termes de cette section : dépendant des sources que vous pourrez consulter, la terminologie peut varier, ce qui peut conduire à des erreurs : certains adjectifs changent de signification selon les auteurs. La terminologie ci-dessous sera fixée tout au long du cours.

On dit qu'un arbre binaire est ***complet*** (***full*** en anglais) si l'arité de chaque nœud interne est 2. Autrement dit, dans un arbre binaire complet, il n'y a pas de nœud d'arité 1 : soit un nœud est une feuille, soit il a deux fils.

Remarque. 1.2 L'arbre vide est complet par définition, puisqu'il n'a pas de nœud d'arité 1.

Les deux arbres de la Fig. 1.10 sont complets, contrairement à l'arbre de la Fig. 1.11.

On dit qu'un arbre est ***parfait*** s'il est complet et que toutes ses feuilles ont même profondeur. Pour chaque hauteur h, il n'y a qu'un squelette d'arbre parfait de hauteur h. La Fig. 1.12 présente les squelettes d'arbres parfaits de hauteur inférieure ou égale à 3.

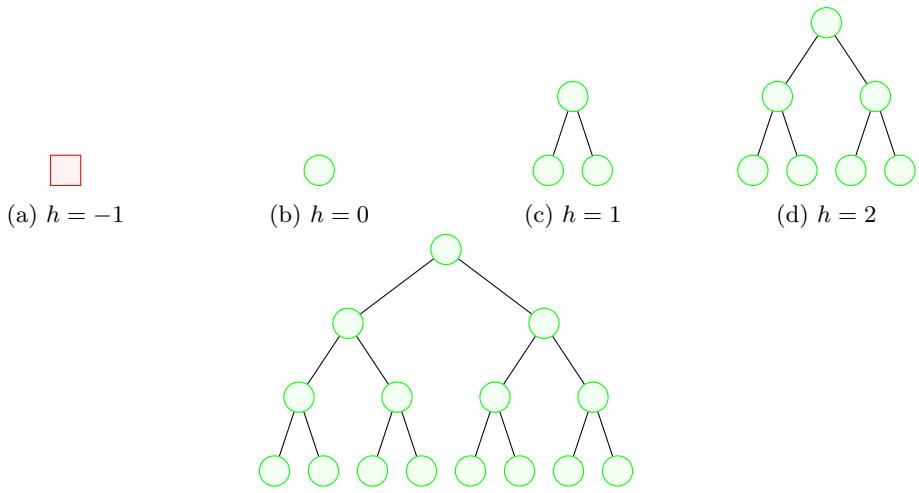
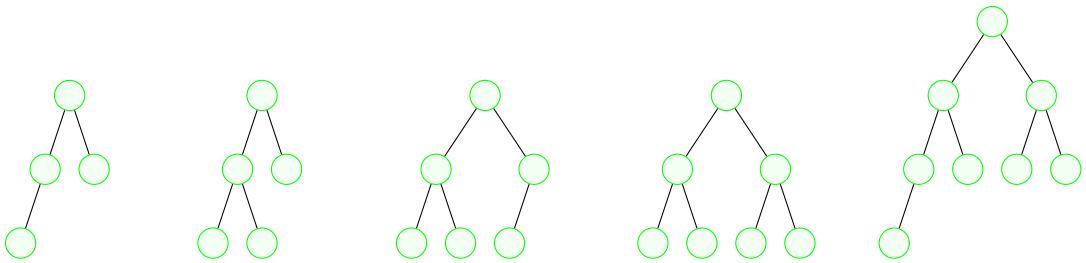


Fig. 1.12 : Les squelettes d'arbres parfaits jusqu'à hauteur 3

Enfin, on dit qu'un arbre est ***quasi-parfait***, si les trois conditions suivantes sont remplies :

- toutes les feuilles sont à profondeur h ou $h-1$, où h est la hauteur de l'arbre,
- chaque niveau jusqu'à la profondeur $h-1$ est complet, c'est-à-dire a le nombre maximal possible de nœuds à ce niveau (1 au niveau 0, 2 au niveau 1, 4 au niveau 2, 8 au niveau 3, ainsi de suite).
- les feuilles de profondeur h sont regroupées le plus à gauche possible.

En particulier, si un arbre est parfait, il est quasi-parfait. Pour un nombre n de nœuds, il y a un seul squelette d'arbre quasi-parfait. Les squelettes d'arbres quasi-parfaits de taille 4, 5, 6, 7 et 8 sont indiqués en Fig. 1.13.



(a) $n = 4$ (b) $n = 5$ (c) $n = 6$ (d) $n = 7$ (e) $n = 8$

FIG. 1.13 : Les squelettes d'arbres quasi-parfaits à $n = 4, 5, 6, 7$ et 8 nœuds

3 Propriétés importantes des arbres binaires

Les arbres binaires ont de nombreuses propriétés simples. Ces propriétés sont *utiles* pour programmer certaines fonctions (calcul de la hauteur ou de la taille d'un arbre, par exemple), ainsi que pour évaluer la complexité des algorithmes (voir le chapitre 2). Il faut connaître ces propriétés, mais surtout, il est important de ***comprendre pour pouvoir les retrouver***.

Soit t un arbre binaire. On note :

- $h(t)$ la hauteur de t,
- $n(t)$ la taille de t, c'est-à-dire son nombre total de nœuds,
- $i(t)$ le nombre de nœuds internes de t,
- $l(t)$ le nombre de feuilles de t.

Les égalités suivantes conduisent à des algorithmes récursifs permettant de calculer ces paramètres. On note $\square$ l'arbre vide. Si t est non vide, on note $\mathsf{left}(t)$ le sous-arbre gauche de t et $\mathsf{right}(t)$ son sous-arbre droit. On a alors les propriétés suivantes.

a) On a $h(\square) = -1$ et si t est non vide, $h(t) = 1 + \max[h(\mathsf{left}(t)), h(\mathsf{right}(t))]$.

b) On a $n(\square) = 0$ et si t est non vide, $n(t) = 1 + n(\mathsf{left}(t)) + n(\mathsf{right}(t))$.

c) On a $i(\square) = 0$ et si t est non vide, $i(t) = \begin{cases} 0 & \text{si } \mathsf{left}(t) = \mathsf{right}(t) = \square, \\ 1 + i(\mathsf{left}(t)) + i(\mathsf{right}(t)) & \text{sinon.} \end{cases}$

d) On a $l(\square) = 0$ et si t est non vide, $l(t) = \begin{cases} 1 & \text{si } \mathsf{left}(t) = \mathsf{right}(t) = \square, \\ l(\mathsf{left}(t)) + l(\mathsf{right}(t)) & \text{sinon.} \end{cases}$

e) On a $n(t) = i(t) + l(t)$.

Il y a une relation simple entre le nombre de feuilles et le nombre de nœuds internes dans les arbres ***complets***. Si un arbre non vide est complet, alors il a une feuille de plus que de nœuds internes, c'est-à-dire :

f) Si t est un arbre complet, on a $i(t) = l(t) - 1$.

On a par ailleurs les relations suivantes entre la taille de t et sa hauteur.

g) On a $h(t) + 1 \leqslant n(t) \leqslant 2^{h(t)+1} - 1$.

h) L'arité de tous les nœuds internes de t est 1 si et seulement si $h(t) + 1 = n(t)$.

i) L'arbre t est parfait si et seulement si $n(t) = 2^{h(t)+1} - 1$.

La relation $n(t) = 2^{h(t)+1} - 1$ valable lorsque t est parfait est particulièrement importante. Elle montre que pour ces arbres, on a $1 + n(t) = 2^{h(t)+1}$, soit

$$h(t) = \log_2(1 + n(t)) - 1.$$

Intuitivement, la hauteur $h(t)$ d'un tel arbre t reste petite, même quand sa taille $n(t)$ est grande. Par exemple, l'arbre parfait de hauteur 20 a $2^{21} - 1 = 2097151$ nœuds (plus de 2 millions de nœuds). C'est important d'un point de vue algorithmique, car on verra qu'on peut effectuer des opérations sur les arbres (rechercher, ajouter ou supprimer une valeur) d'autant plus efficacement que la hauteur est petite.

4 Représentation d'ensembles et de multi-ensembles

Tout au long de ce cours, nous allons manipuler des données mémorisées dans un arbre. La première idée est de partir d'un ensemble d'éléments. Par exemple, si les données à manipuler sont les entiers entre 0 et 10, l'ensemble correspondant serait $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Malheureusement, les ensembles ne suffisent pas, car un ensemble contient un élément ou pas, mais ne peut pas contenir deux fois le même élément. On veut donc manipuler des objets mathématiques dans lesquels un élément peut apparaître plusieurs fois, mais où l'ordre n'importe pas. L'objet mathématique adapté est en fait une fonction d'un ensemble E dans $\mathbb{N}$: pour un élément $x \in E$, l'entier $f(x)$ compte le nombre d'occurrences de x. Cet objet s'appelle un *multi-ensemble*. En pratique, on écrira un multi-ensemble fini simplement en donnant ses éléments, chaque élément pouvant être présent plusieurs fois. Au lieu d'utiliser la notation $\{\ldots\}$, on utilisera $\{\!\{\ldots\}\!\}$.

On utilisera souvent des valeurs entières, parce que c'est un type simple. Par exemple, on pourra considérer le multi-ensemble $S = \{\!\{12, 2, 2, 3, 5, 6, 7, 6\}\!\}$. Comme l'ordre des éléments dans un multi-ensemble n'est pas important, on peut aussi noter $S = \{\!\{7, 2, 12, 3, 5, 2, 6, 6\}\!\}$, ou bien encore $S = \{\!\{2, 2, 3, 5, 6, 6, 7, 12\}\!\}$. Ce multi-ensemble peut être représenté par un tableau, par une liste, par des arbres, etc. Le chapitre suivant présentera les avantages qu'on a à utiliser des arbres plutôt que des tableaux ou des listes pour représenter des ensembles ou des multi-ensembles.

5 Algorithmes génériques fondamentaux

Après un rappel sur les structures de données linéaires (tableau et listes), les chapitres suivants présentent des problèmes et algorithmes sur les arbres. Ces algorithmes sont guidés par l'utilisation qu'on souhaite faire des arbres. Cependant, certaines opérations sont génériques et importantes. Elles reviennent donc fréquemment. Parmi celles-ci, il est conseillé de réfléchir à des algorithmes pour exécuter les tâches suivantes. Vous pouvez retrouver les algorithmes correspondants, écrits en OCaml, sur la page du cours de Programmation fonctionnelle, https://www.labri.fr/~zeitoun/enseignement/17-18/PF/.

Pour simplifier, nous nous limitons volontairement ici aux arbres binaires. À partir d'un arbre donné en entrée, il s'agit de calculer :

- son nombre de feuilles,
- son nombre de nœuds internes,
- sa taille, c'est-à-dire son nombre de nœuds, en comptant les feuilles comme les nœuds internes,
- sa hauteur, c'est-à-dire la longueur maximale d'une de ses branches,
- la liste de ses nœuds en parcours infixe,
- la liste de ses nœuds en parcours préfixe,
- la liste de ses nœuds en parcours postfixe.

Par ailleurs, comme les arbres seront utilisés pour mémoriser de l'information, une tâche élémentaire est, étant donné un arbre t dont les nœuds sont étiquetés par des éléments d'un ensemble E, ainsi qu'un élément x de E, de calculer

- un booléen, vrai si l'élément x apparaît dans l'arbre, et faux sinon.
- de façon plus précise, on peut demander que la fonction renvoie, en plus de ce booléen, un chemin de la racine à un sommet qui contient l'élément x. Ce chemin peut être représenté par une liste de directions (gauche ou droite).
- enfin, les arbres sont fréquemment utilisés comme structure de données pour représenter des dictionnaires. Un ***dictionnaire*** est une liste de couples, chaque couple étant de la forme (k, v), où k est appelée ***clé*** et v est appelée ***valeur***. Chaque clé apparaît au maximum une fois dans l'arbre. Par contre, une valeur peut apparaître plusieurs fois (associée à des clés différentes). Par exemple, dans un dictionnaire au sens habituel, chaque clé est un mot et la valeur associée à un mot contient ses différentes définitions.

  Dans ce cadre, plutôt que de retourner simplement un booléen disant si x apparaît dans l'arbre, il est intéressant d'avoir un algorithme recherchant x comme *clé* dans l'arbre, et si x est effectivement présent comme clé, retournant l'unique valeur associée à x.

6 Références

Le livre [1] est librement disponible en ligne. Il contient plusieurs algorithmes classiques, en particulier sur les arbres. Attention, la terminologie sur les arbres (parfaits, complets, etc.) diffère de celle de ce document.

Bibliographie

[1] D. Beauquier, J. Berstel et Ph. Chrétienne. *Éléments d'Algorithmique*. Masson, 1992. URL : http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.html.

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 chacune des propriétés $(\mathcal{P}_k)$ est vraie, pour $1 \leqslant k \leqslant n$, et montrons que la propriété $(\mathcal{P}_{n+1})$ est également vraie.
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.

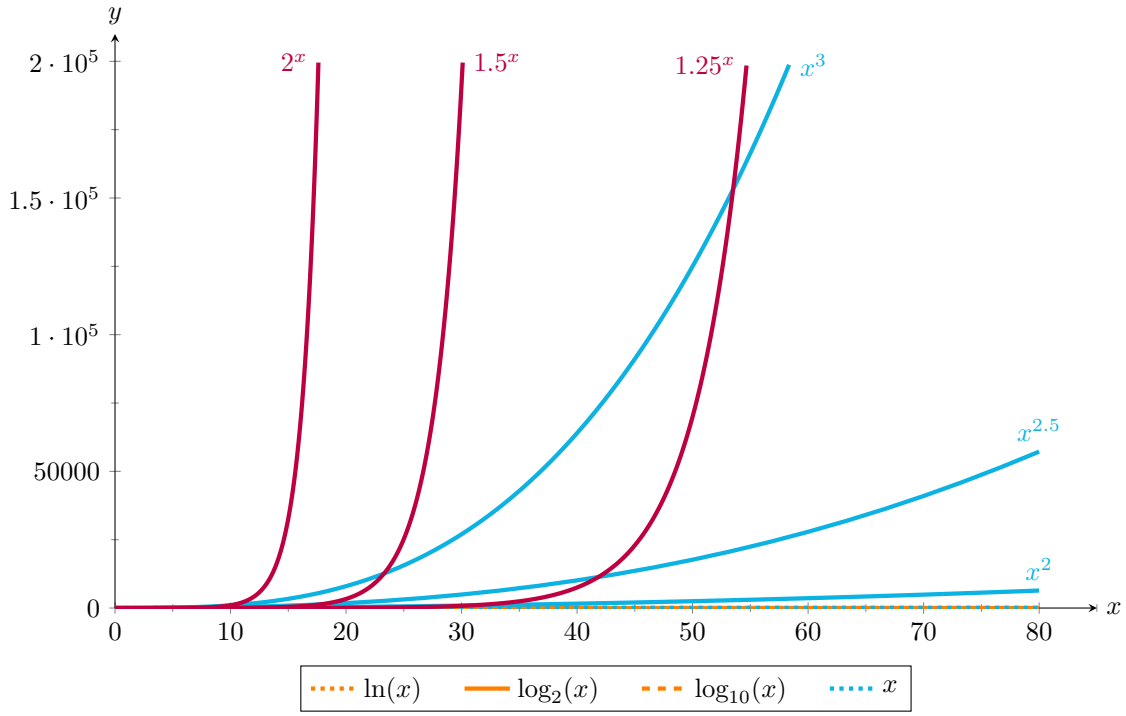


Fig. 2.1 : Croissance de fonctions importantes

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 $\log_{10}(x)$.

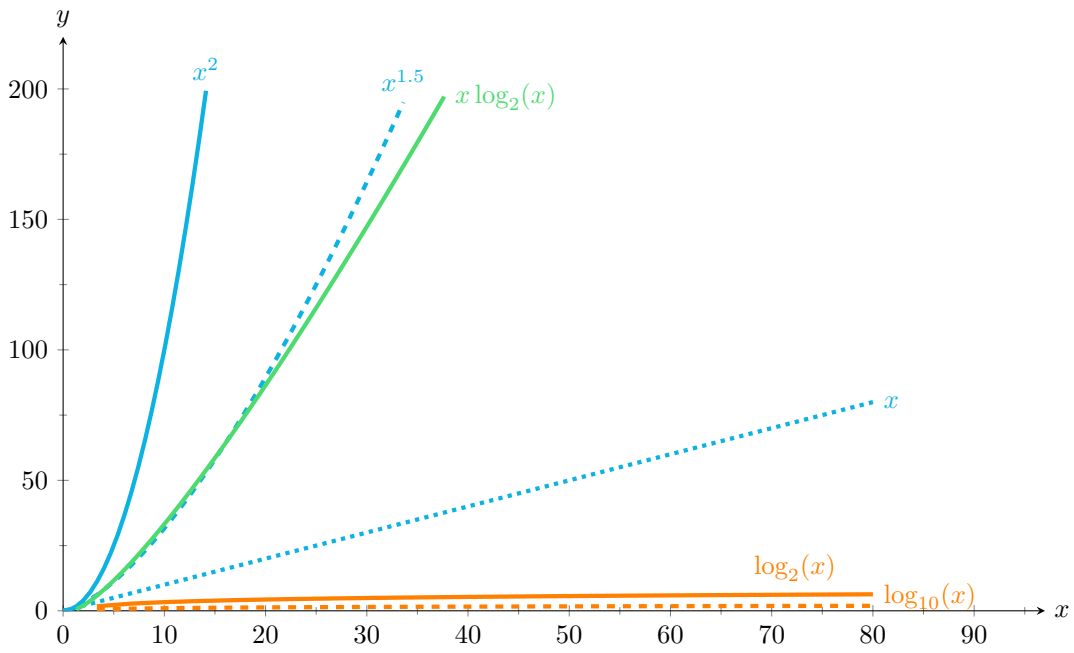


FIG. 2.2 : Croissance de fonctions importantes (2)

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 $n = 10^9$.

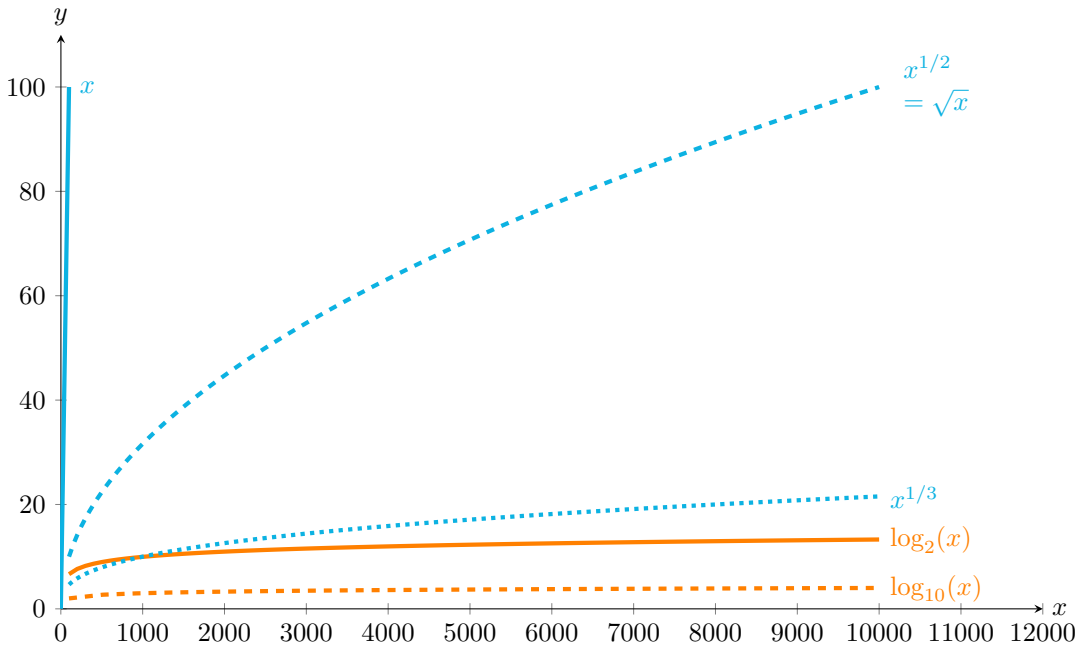


FIG. 2.3 : Croissance de fonctions importantes (3)

	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

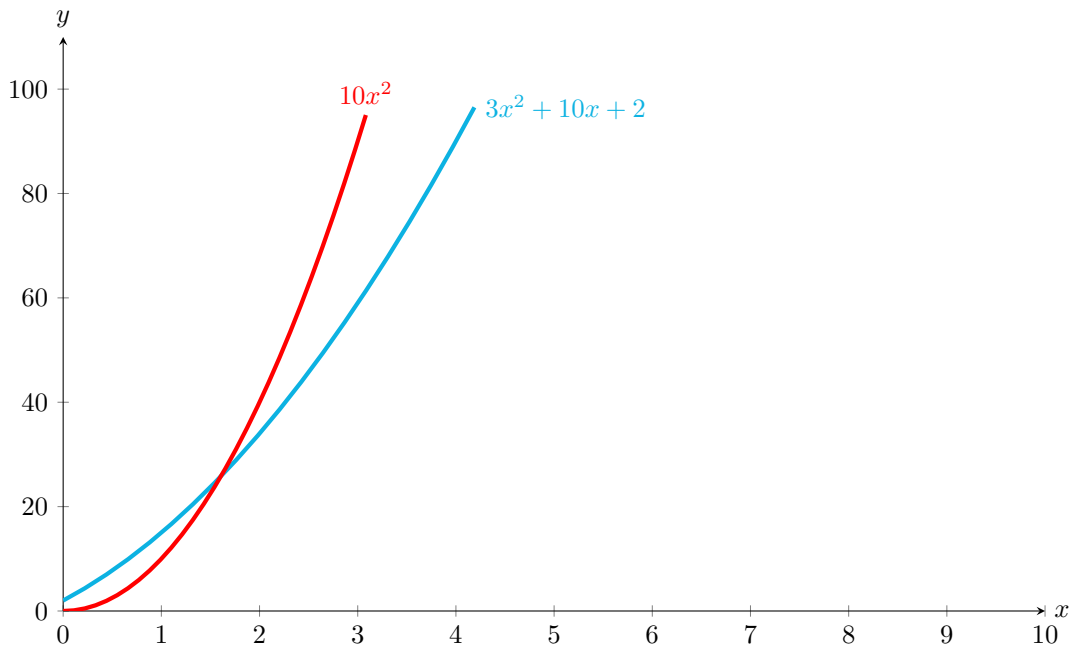


FIG. 2.4 : $3x^2 + 10x + 2 = O(x^2)$

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, $f(n) \leqslant 1001 \cdot n^2$.