Mathématiques pour l'Informatique I

(Notes de cours)

L1 UMLV

C. Nicaud

February 10, 2014

Contents

Chapter 1

Rappels

1.1 Fonctions et applications

Soit f une fonction d'un ensemble E dans un ensemble F. Le *domaine de définition* de f est l'ensemble des éléments de E qui ont une image par f. On le note $\mathrm{Dom}(f)$. L'*image* de f est l'ensemble des images des éléments de $\mathrm{Dom}(E)$.

Si f est définie partout, on dit que f est une *application*. On note F^E l'ensemble des applications de E dans F. Attention à l'ordre dans la notation.

Il faut connaître les définitions suivantes, pour une application f de E dans F :

- f est injective quand deux éléments distincts ont des images distinctes: $\forall x, x' \in E$, $x \neq x' \Rightarrow f(x) \neq f(x')$. Par contraposée, c'est équivalent à $\forall x, x' \in E$, $f(x) = f(x') \Rightarrow x = x'$.
- f est surjective quand tout élément de F a un antécédent dans E: $\forall y \in F$, $\exists x_y \in E$ tel que $f(x_y) = y$.
- f est bijective quand elle est à la fois injective et surjective.

Soit E un ensemble. L'*identité* sur E est l'application Id_E de E dans E définie pour tout $x \in E$ par $f(x) = x$.

Théorème 1 (caractérisation des bijections) *Soit f une application de E dans F. f est bijection si et seulement si il existe une application g de F dans E telle que $f \circ g = Id_F$ et $g \circ f = Id_E$. On appelle alors g la* réciproque *de f et on la note f^{-1}.*

1.2 Dénombrements élémentaires

Soit E un ensemble fini. Son *cardinal*, noté $|E|$, est son nombre d'éléments.

Soient A et B deux sous-ensembles d'un ensemble fini E, on a

$$|A \cap B| + |A \cup B| = |A| + |B|.$$

En particulier, si A et B sont disjoints, c'est-à-dire que $A \cap B = \emptyset$, alors $|A \cup B| = |A| + |B|$. Et on a toujours, ce dont on se sert souvent,

$$|A \cup B| \leq |A| + |B|.$$

Soient E et F deux ensembles finis, on a

- $|\mathcal{P}(E)| = 2^{|E|}$,
- $|E \times F| = |E| \times |F|$,
- $|F^E| = |F|^{|E|}$,
- si f est une injection de E dans F alors $|E| \leq |F|$,
- si f est une surjection de E dans F alors $|E| \geq |F|$,
- si f est une bijection de E dans F alors $|E| = |F|$,

1.3 Changements de base

On est habitués à écrire les nombres en base 10, quand on écrit 7302 cela signifie

$$7302 = 7 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 2 \times 10^0.$$

En fait, on peut utiliser n'importe quelle base entière $b \geq 2$ pour écrire de façon unique les nombres.

Théorème 2 (Base) *Soit $b \geq 2$ un entier. Pour tout entier $n \geq 0$, il existe un $\ell \geq 0$ et un ℓ-uplet $(x_0, x_2, \ldots, x_{\ell-1}) \in \{0, \ldots, b-1\}^\ell$ tel que*

$$n = \sum_{i=0}^{\ell-1} x_i b^i = x_0 b^0 + x_1 b^1 + \ldots + x_{\ell-1} b^{\ell-1}.$$

Si de plus on impose que $n \geq 1$ et $x_{\ell-1} \neq 0$, on a unicité de l'écriture.

En informatique on s'intéresse surtout à la base $b = 2$, où les chiffres sont $\{0, 1\}$. Quand un nombre est écrit en base 2 on dit qu'il est écrit en *binaire*. Les chiffres $\{0, 1\}$ s'appellent des *bits*, une séquence de 8 bits s'appelle un *octet*.

Le plus grand nombre qu'on peut écrire en base 10 avec 3 chiffres est 999. Comme on commence à 0, il y a 1000 nombres que l'on peut écrire en utilisant 3 chiffres en base 10. Si on reprend la définition, avec ℓ chiffres on peut écrire tous les nombres de 0 jusqu'à $b^\ell - 1$; ce qui fait exactement b^ℓ nombres différents. Celà est dû au fait (exercice) que pour ℓ fixé, l'application f de $\{0, \ldots, b-1\}^\ell$ dans $\{0, \ldots, b^\ell - 1\}$ définie par

$$f\left((x_0, \cdots, x_{\ell-1})\right) = \sum_{i=0}^{\ell-1} x_i b^i$$

est une bijection. Par exemple, avec 10 chiffres binaires, on peut écrire $2^{10} = 1024$ nombres différents.

Si on raisonne dans l'autre sens, en partant d'un ensemble E avec $n \geq 2$ éléments, on peut associer de façon unique un élément de $\{0, \ldots, n-1\}$ à chaque élément de E, en prenant n'importe quelle bijection de E dans $\{0, \ldots, n-1\}$. Notons ϕ cette numérotation.Si on veut représenter les éléments de E dans l'ordinateur, on peut représenter leur numéro associé, et pour ça il faut suffisamment de place pour les coder tous. D'après ce qui précède, il faut trouver ℓ tel que

$$b^{\ell-1} < n \leq b^\ell,$$

pour avoir le nombre de chiffres minimum à utiliser pour bien tous les distinguer. C'est équivalent à

$$\ell - 1 < \log_b n \leq \ell \Leftrightarrow \lceil \log_b n \rceil = \ell.$$

Pour distinguer n objets, il faut donc utiliser $\lceil \log_b n \rceil$ chiffres. Pour le cas binaire, il faut $\lceil \log_2 n \rceil$ bits.

1.4 Rappels sur le logarithme en base 2

On rappelle que pour tout réel $x > 0$, $\log_2(x) = \frac{\ln x}{\ln 2}$ et vérifie les propriétés suivantes :

- $\log_2$ est croissante sur $\mathbb{R}_+^*$, avec $\log_2 1 = 0$ et $\log_2 2 = 1$;
- elle est continue, dérivable, et de dérivée $x \mapsto \frac{1}{\ln 2}\frac{1}{x}$;
- pour tous $x, y > 0$, $\log_2(xy) = \log_2 x + \log_2 y$;
- pour tout $x > 0$ et tout $y \in \mathbb{R}$, $\log_2(x^y) = y \log_2 x$;
- pour tout $x > 0$, $2^{\log_2 x} = x$ et pour tout $y \in \mathbb{R}$, $\log_2 2^y = y$, c'est donc une bijection de $\mathbb{R}_+^*$ dans $\mathbb{R}$ de réciproque $x \mapsto 2^x$.

Avec ce qui précède sur les bases de numération on a le théorème suivant :

Théorème 3 *Le nombre de chiffres nécessaires pour représenter un nombre $n \geq 1$ en binaire est $\lceil \log_2(n+1) \rceil$.*

En remarquant que diviser par deux revient à "décaler la virgule" en binaire où plutot que si on définit l'application d de $\mathbb{N}$ dans $\mathbb{N}$ qui a n associe $\lfloor \frac{n}{2} \rfloor$, il faut appliquer $\lfloor \log_2 n \rfloor$ fois d à un nombre n pour arriver à 0 : savoir que "on peut diviser n par deux environ $\log_2 n$ fois" est très utile en informatique.

Chapter 2

Mots et langages

2.1 Introduction

Dans une machine actuelle, on n'a notre disposition que des 0 et des 1 pour stocker l'information. Il s'agit donc d'arriver à représenter nos données uniquement avec des suites de 0 et de 1.

Par exemple, une représentation très utilisée pour les caractères est le code ASCII : à chaque caractère on associe une séquence de 8 bits, 1 octet, qui le représente de façon unique. Une chaine de caractère, en C, est donc une suite de caractères, qui eux-mêmes sont des groupes de 8 bits. Un caractères spécial est utilisé pour désigner la fin de la chaîne.

Si on prend comme unité le bit, un caractère est une suite de bits; si on prend comme unité le caractère, une chaîne est une suite de caractères. Il semble donc important de développer des connaissances sur les "suites de lettres", puisqu'on aura toujours à s'en servir.

La notion de code permet des représentations un peu plus compliquées que le code ASCII, mais peut-être plus efficaces (plus compactes par exemple). Il s'agit de représenter nos objets, toujours par des suites de bits, mais pas nécessairement de longueur fixe. On veut cependant que le découpage soit sans ambiguité afin qu'une même suite de bits ne représente pas 2 objets différents.

2.2 Mots

2.2.1 Définitions

Définition 4 *Un alphabet est un ensemble fini dont les éléments sont appelés des lettres.*

Exemple : $B = \{0, 1\}$ l'alphabet binaire, $A = \{a, b, c\}$, $C = \{a \cdots, z\}$. On peut considérer n'importe quel ensemble fini, par exemple $A = \{hello, word\}$ est un alphabet à deux lettres.

Définition 5 *Un mot sur l'alphabet A est une suite finie d'élément de A. On note A^* l'ensemble des mots sur A. La suite peut ne contenir aucun élément, il s'agit dans ce cas du mot vide, noté ϵ, qui ne contient aucune lettre.*

Exemple : 0011010, 0110 sont des mots sur l'alphabet $\{0, 1\}$. *abbab* est un mot sur $\{a, b\}$. *hello word word hello* est un mot sur $A = \{hello, word\}$.

Mathématiquement, on a

$$A^* = \bigcup_{n \geq 0} A^n$$

avec la convention $A^0 = \{\varepsilon\}$. On représente les n-uplets sans les virgules et les parenthèses.

Deux mots sont égaux s'ils ont les mêmes lettres dans le même ordre.

2.2.2 Concaténation

Définition 6 *Soit $u = u_1 u_2 \cdots u_n$ un mot de taille n sur l'alphabet A, les u_i étant les lettres le composant, et soit $v = v_1 v_2 \cdots v_m$ un autre mot, de taille m sur l'alphabet A. On note $u \cdot v$ la concaténation de u et de v, qui est le mot :*

$$u \cdot v = u_1 \cdots u_n v_1 \cdots v_m$$

Proposition 7 *La concaténation sur A^* est associative et admet pour élément neutre le mot vide ϵ. Elle n'est pas commutative.*

Définition 8 *Un monoïde est ensemble muni d'une loi interne associative et possédant un élément neutre. A^* muni de la concaténation est un monoïde.*

Définition 9 *Soient $(X, +)$ et $(Y, *)$ deux monoïdes de neutres respectifs ε_X et ε_Y. Une application f de X dans Y est un* morphisme de monoïde *(on dit juste* morphisme *quand il n'y a pas d'ambiguïté), quand*

- $f(\varepsilon_X) = \varepsilon_Y$
- *pour tout $x_1, x_2 \in X$, $f(x_1 + x_2) = f(x_1) * f(x_2)$*

2.2.3 Longueur

Définition 10 *La taille, ou encore la longueur, d'un mot $u \in A^*$ est son nombre de lettres. On la note $|u|$. dans le mot u. Le mot vide est de taille* 0.

Notation 11 *Soit $a \in A$, le nombre de a présents dans un mot $u \in A^*$ est noté $|u|_a$.*

Proposition 12 *Pour tout $u, v \in A^*$ on a :*

- $|uv| = |u| + |v|$
- *pour tout $a \in A$, $|uv|_a = |u|_a + |v|_a$*

2.2.4 Découpage des mots

Définition 13 *On dit qu'un mot v est un* préfixe *d'un mot u sur A, s'il existe un mot $w \in A^*$ tel que $u = vw$. On dit qu'un mot v est un* suffixe *d'un mot u sur A, s'il existe un mot $w \in A^*$ tel que $u = wv$. On dit qu'un mot v est un* facteur *d'un mot u sur A, s'il existe deux mots $w, w' \in A^*$ tel que $u = wvw'$.*

Définition 14 *Un préfixe (resp. suffixe, facteur) d'un mot u est un* prefixe strict *(resp.* suffixe strict, facteur strict*) s'il est différent de u.*

Définition 15 *Soit $u = u_1 \cdots u_n$ un mot de A^*. Un sous-mot v de u est un mot v de longueur m tel qu'il existe une injection strictement croissante ϕ de $\{1, \cdots, m\}$ dans $\{1, \cdots, n\}$ tel que :*

$$v = u_{\phi(1)} u_{\phi(2)} \cdots u_{\phi(m)}$$

2.2.5 Opérations sur les mots

Définition 16 *Le miroir d'un mot $u = u_1 \cdots u_n$ sur A est le mot $\overline{u} = u_n \cdots u_1$ obtenu lisant les lettres dans l'autre sens.*

Proposition 17 *Pour tout $u, v \in A^*$, $\overline{u \cdot v} = \overline{v} \cdot \overline{u}$*

2.2.6 Ordre sur les mots

On suppose que l'alphabet A est totalement ordonné.

Définition 18 *Soient u et v deux mots de A^* on dit que u est plus petit que v pour l'*ordre lexicographique*, noté $u \leq v$ ou encore $u \leq_{lex} v$ quand :*

- *ou bien u est préfixe de v,*
- *ou bien il existe $a < b$ dans A, il existe des mots w, u', v' dans A^*, tels que $u = wau'$ et $v = wbv'$.*

Donner des exemples.

Définition 19 *Soient u et v deux mots de A^* on dit que u est plus petit que v pour l'*ordre militaire*, noté $u \leq_{mil} v$, quand*

- *ou bien $|u| < |v|$,*
- *ou bien $|u| = |v|$ et $u \leq_{lex} v$.*

2.2.7 Lemme de Levy

Lemme 20 *Soient x, y, z, t des mots tels qu $xy = zt$. Alors*

- *ou bien $xw = z$ et $y = wt$*
- *ou bien $x = zw$ et $wy = t$*

Conséquence importante : si on applique le Lemme de Levy avec $z = x$, on a $w = \varepsilon$ et donc $y = t$. En en déduit que si $xy = xt$ alors $y = t$: on peut simplifier à gauche. De même, on peut simplifier à droite une équation sur les mots.

2.3 Langages

2.3.1 Définitions

Définition 21 *Un langage est un ensemble de mots. C'est donc un élément de $\mathcal{P}(A^*)$.*

Exemples : $\{a^n b \mid n \in \mathbb{N}\}$, $\{0^p \mid p \text{ premier}\}$, {mots de langue francaise} sont des langages.

Comme les langages sont des ensembles, on dispose bien entendu des opérations ensemblistes classiques : union, complémentaire, intersection, etc.

2.3.2 Extension des opérations sur les mots

Définition 22 *Soient X et Y deux langages, la* concaténation *de X et de Y, notée XY est l'ensemble des concaténations d'un mot de X par un mot de Y :*

$$XY = \{xy \mid x \in X,\ y \in Y\}$$

Lemme 23 *Si X, Y et Z sont des langages, on a*

- $X(Y \cup Z) = XY \cup XZ$
- $X(Y \cap Z) \subset XY \cap XZ$

Définition 24 *Soit X un langages, le* miroir *noté $\overline{X}$ est l'ensemble des miroirs des mots de X.*

$$\overline{X} = \{\overline{x} \mid x \in X\}$$

2.3.3 Autres opérations

Par convention $X^0 = \{\varepsilon\}$. On définit les puissances successives d'un langage, pour tout $n \geq 1$ par

$$X^n = X^{n-1}X$$

Définition 25 *Soit X un langage, l'*étoile *de X est le langage*

$$X^* = \bigcup_{n \geq 0} X^n$$

En particulier X^* contient toujours le mot vide.

Définition 26 *Soient X et Y deux ensembles de mots. Le* résiduel à gauche *de Y par X, noté $X^{-1}Y$ est l'ensemble*

$$X^{-1}Y = \{u \in A^* \mid \exists x \in X, \exists y \in Y, xu = y\}$$

Définition 27 *Soient X et Y deux ensembles de mots. Le* résiduel à droite *de X par Y, noté XY^{-1} est l'ensemble*

$$XY^{-1} = \{u \in A^* \mid \exists x \in X, \exists y \in Y, x = uy\}$$

2.3.4 Le lemme d'Arden

Le lemme ci-dessous est très utile pour résoudre des équations ou des systèmes d'équations sur les langages.

Lemme 28 Lemme d'Arden *Soient E et F deux langages sur A tels que $\varepsilon \notin E$. L'équation sur les langages $X = E \cdot X \ \cup \ F$, où X est le langage inconnu, admet pour unique solution E^*F.*

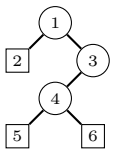
Chapter 3

Codes, codages et compression

3.1 Arbres binaires

On reste informels dans cette partie. La formalisation mathématique de la notion d'arbre est bien entendu possible, mais n'apporte rien dans le cadre de ce cours, où on se sert des arbres pour décrire l'algorithme de compression de Huffman.
▶ Un *arbre binaire* $\mathcal{B}$ est composé de *nœuds*. Il y a un *nœud* particulier appelé la racine de $\mathcal{B}$. Tout nœud de $\mathcal{B}$ qui n'est pas la racine est soit le *fils gauche* soit le *fils droit* d'un autre nœud $\mathcal{B}$. La racine n'est fils d'aucun nœud. On demande de plus que tout nœud soit atteignable depuis la racine : si on prend la clotûre transitive de la relation "est fils de" sur les nœuds de $\mathcal{B}$, tout nœud est en relation avec la racine.

▶ Si un nœud a au moins un fils on dit que c'est un *nœud interne*, sinon on dit que c'est une *feuille*.



On représente la racine en haut et les feuilles en bas.

- 1 est la racine;
- 2, 5 et 6 sont les feuilles;
- 1, 3 et 4 sont des nœuds internes.

Définition 29 (forêt) *Une* forêt *est un ensemble d'arbres.*

3.2 Codes

Définition 30 (code) *Un sous-ensemble non-vide X de A^* est un* code *quand tout mot de A^* s'écrit d'au plus une façon comme concaténation de mots de X : X est un code si et seulement si quand on a*

$$x_1 \cdots x_n = y_1 \cdots y_m,$$

où les x_i et les y_i sont des mots de X, alors $n = m$ et pour tout $i \in \{1, \dots n\}$, $x_i = y_i$.

▶ On n'impose pas que tout mot soit représentable comme concaténation de mots de X, mais que s'il l'est alors il l'est d'une unique façon.

Définition 31 (code de longueur fixe) *Soit X un langage qui n'est ni vide ni réduit au mot vide. Si tous les mots de X sont de même longueur $k \geq 1$, on dit que X est un* code de longueur fixe.

Théorème 32 *Un code de longueur fixe est un code.*

Exemple : ASCII, UNICODE, code bitmap pour les images, *etc.* sont des codes de longueur fixe.

Définition 33 (code préfixe) *Un langage X non vide et non réduit au mot vide est un* code préfixe *quand aucun mot de X n'est préfixe d'un autre mot de X.*

Théorème 34 *Un code préfixe est un code.*

▶ Attention à la terminologie, on devrait dire "code sans préfixes".

▶ Les codes préfixes sont très utilisés pour faire de la compression de données.

Définition 35 (code suffixe) *Un langage X non vide et non réduit au mot vide est un* code suffixe *quand aucun mot de X n'est suffixe d'un autre mot de X.*

Théorème 36 *Un code suffixe est un code.*

▶ Il existe des langages qui ne sont dans aucune des catégories précédentes mais qui sont pourtant des codes.

3.3 Codage

On note $B = \{0, 1\}$ l'alphabet binaire.

Définition 37 (codage) *Un codage est une application injective de A^* dans B^*.*

▶ Cette définition est trop générale en pratique, on ne peut pas stocker toutes les images de tous les mots.

Théorème 38 *Soit $C \subset B^*$ un code et soit φ une application injective de A dans C. On étend φ (par morphisme) à tout $u \in A^*$ par $\varphi(\varepsilon) = \varepsilon$ et pour tout $u = u_1 \cdots u_n$ mot de A^* de longueur $n \geq 1$,*

$$\varphi(u) = \varphi(u_1) \cdots \varphi(u_n).$$

Alors φ est un codage.

3.4 Algorithme d'Huffman

► L'idée de la compression de donnée est de représenter en machine un mot u de A^* par un codage φ et un mot v sur B^*, tels que $\varphi(u) = v$.

► Pour que cela soit efficace, il faut que

- l'encodage se calcule rapidement;
- la représentation de (φ, v) soit compacte;
- le décodage, pour retrouver u, se fasse rapidement.

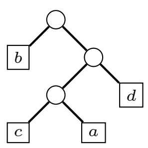
► L'algorithme de Huffman permet de réaliser un tel encodage en utilisant le théorème 38 : on va construire un code préfixe C et le codage φ à partir du mot à compresser u, puis on en déduira v.

► **1. Initialisation :** on crée une forêt avec un arbre pour tout $a \in A$. Chacun de ces arbres est réduit à juste une racine. Le *poids* de la racine de l'arbre a est $|u|_a$.

► **2. Construction de l'arbre :** tant qu'il y a plus d'un arbre dans la forêt on répète l'opération suivante : sélectionner les deux arbres $\mathcal{A}$ et $\mathcal{B}$ qui ont les plus petits poids à la racine $p_{\mathcal{A}}$ et $p_{\mathcal{B}}$. Les changer en un seul arbre $\overset{\circ}{\wedge}_{\mathcal{A}\ \mathcal{B}}$, de poids $p_{\mathcal{A}} + p_{\mathcal{B}}$ à la racine.

► **3. Codage pour les lettres :** une fois l'arbre terminé, les lettres sont sur les feuilles. A chaque lettre $a \in A$, on associe le mot binaire $\varphi(a)$ obtenu en suivant le chemin allant de la racine à la feuille a, en mettant, de gauche à droite, un 0 quand on va à gauche et un 1 quand on va à droite.

Exemple :



$\varphi(a) = 101$
$\varphi(b) = 0$
$\varphi(c) = 100$
$\varphi(d) = 11$

► La taille finale obtenue, $|\varphi(u)|$ vérifie :

$$|\varphi(u)| = \sum_{a \in A} |\varphi(a)| \cdot |u|_a.$$

► Si on utilise un code de longueur fixe, le mieux qu'on puisse faire c'est utiliser k bits par symbole, où k est l'unique entier tel que $2^{k-1} < |A| \leq 2^k$.

Chapter 4

Estimation asymptotique

Ce chapitre traite de l'estimation asymptotique de suites à valeurs positives. Il s'agit de travailler sur les objets mathématiques qui servent à mesurer les performances d'un algorithme.

4.1 Propriétés vraies "à partir d'un certain rang"

Définition 39 (Propriété) *Une* propriété *sur $\mathbb{R}$ est une application de $\mathbb{R}$ dans* $\{\texttt{Vrai}, \texttt{Faux}\}$.

Définition 40 (Opération) *Une* opération *sur $\mathbb{R}$ est une application de $\mathbb{R} \times \mathbb{R}$ dans $\mathbb{R}$. On note $x \star y$ l'image de (x, y) par l'opération $\star$, au lieu de $\star((x, y))$.*

Définition 41 (Propriété stable) *Si $\star$ est une opération sur $\mathbb{R}$ et si P une propriété sur $\mathbb{R}$, on dit que P est stable pour $\star$ quand pour tout $x, y \in \mathbb{R}$, si $P(x) = \texttt{Vrai}$ et $P(y) = \texttt{Vrai}$, alors $P(x \star y) = \texttt{Vrai}$.*

Exemple : La propriété "est un élément de $\mathbb{N}$" est stable pour l'addition et la multiplication.

Définition 42 (APCR) *Soit $(u_n)_{n\in\mathbb{N}}$ une suite à valeurs dans $\mathbb{R}$ et P une propriété sur $\mathbb{R}$. On dit que $(u_n)_{n\in\mathbb{N}}$ satisfait P à partir d'un certain rang, noté APCR, si il existe $n_0 \in \mathbb{N}$ tel que pour tout $n \geq n_0$, $P(n)$ est vraie.*

Exemple : La suite $(u_n)_{n\in\mathbb{N}}$ définie pour tout $n \in \mathbb{N}$ par $u_n = 7n - 31$ est positive APCR.

Théorème 43 (APCR et stabilité) *Soit P une propriété stable par $\star$, et soient $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ deux suites qui satisfont P APCR. Alors la suite $(u_n \star v_n)_{n\in\mathbb{N}}$ satisfait P APCR.*

▶ On peut étendre à la composition d'un nombre fixé de suites.

Définition 44 (Bornée) *Soit $(u_n)_{n\in\mathbb{N}}$ une suite à valeurs dans $\mathbb{R}$. On dit que $(u_n)_{n\in\mathbb{N}}$ est* bornée *s'il existe un réel $C \geq 0$ tel que pour tout $n \in \mathbb{N}$, $|u_n| \leq C$.*

Théorème 45 (Bornée APCR = bornée) *Soit $(u_n)_{n\in\mathbb{N}}$ une suite bornée APCR, alors $(u_n)_{n\in\mathbb{N}}$ est bornée (tout le temps).*

4.2 Hierarchie de fonctions

Définition 46 (Ordre de croissance) *Soit $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ deux suites réelles qui sont strictement positives APCR. On dit que $(f_n)_{n\in\mathbb{N}}$ croît plus lentement que $(g_n)_{n\in\mathbb{N}}$, noté $f_n \prec g_n$ quand*

$$\lim_{n\to\infty} \frac{f_n}{g_n} = 0.$$

▶ Remarque : le rapport est bien défini APCR.

Théorème 47 (Transistivité) *La relation $\prec$ sur l'ensemble des suites réelles strictement positives APCR est transitive.*

▶ On note $\mathbf{1}$ la suite constante égale à 1.

Théorème 48 (Echelle) *Pour tous réels ϵ et c avec $0 < \epsilon < 1 < c$, on a*

$$\mathbf{1} \prec \log n \prec (\log n)^c \prec n^\epsilon \prec n^c \prec c^n \prec n! \prec n^n \prec c^{c^n}.$$

Théorème 49 (Inverse) *Soient $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ strictement positives APCR, les suites $(\frac{1}{f_n})_{n\in\mathbb{N}}$ et $(\frac{1}{g_n})_{n\in\mathbb{N}}$ sont définies APCR et on a*

$$f_n \prec g_n \Leftrightarrow \frac{1}{g_n} \prec \frac{1}{f_n}.$$

Théorème 50 (Constante multiplicative) *Soient $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ telles que $f_n \prec g_n$. Pour toute constante strictement positive c on a $f_n \prec c \cdot g_n$.*

Théorème 51 (Cas fréquent) *Soient $f_n = n^a(\log n)^b$ et $g_n = n^\alpha(\log n)^\beta$, définies pour $n \geq 1$. On a*

$$f_n \prec g_n \Leftrightarrow \begin{cases} a < \alpha \\ ou \\ a = \alpha \text{ et } b < \beta \end{cases}$$

Définition 52 (Suites équivalentes) *On dit que deux suites non nulles APCR $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ sont équivalentes, noté $f_n \sim g_n$ quand*

$$\lim_{n\to\infty} \frac{f_n}{g_n} = 1.$$

Théorème 53 (Majoration) *Soient $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ strictement positives telles que $f_n \prec g_n$. Il existe une constante $C \geq 0$ telle que pour tout $n \in \mathbb{N}$, $f_n \leq C \cdot g_n$.*

4.3 La notation $\mathcal{O}$

4.3.1 Définition

Définition 54 ($\mathcal{O}$) *Soient $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ deux suites réelles. On dit que $f_n \in \mathcal{O}(g_n)$ quand il existe une constante $C \in \mathbb{R}_+^*$ telle que APCR on ait*

$$|f_n| \leq C \cdot |g_n|.$$

Autrement dit, il existe entier $n_0 \in \mathbb{N}$ tels que pour tout $n \geq n_0$,

$$|f_n| \leq C \cdot |g_n|.$$

▶ $\mathcal{O}(g_n)$ est donc un ensemble de suites.

▶ Si $(g_n)_{n\in\mathbb{N}}$ est non nulle APCR, $f_n \in O(g_n)$ signifie que la suite $\frac{f_n}{g_n}$ est bornée APCR.

▶ $\mathcal{O}(\mathbf{1})$ est l'ensemble des suites bornées APCR, qui est aussi l'ensemble des suites bornées.

Théorème 55 (Convergentes $\subset \mathcal{O}(1)$) *Toute suite convergente est dans $\mathcal{O}(1)$. La réciproque est fausse.*

4.3.2 Autres notations similaires

Définition 56 (Ω) *On note $f_n \in \Omega(g_n)$ quand il existe un réel $C > 0$ tel que APCR on ait $|f_n| \geq C \cdot |g_n|$.*

▶ On a donc $f_n \in \mathcal{O}(g_n) \Leftrightarrow g_n \in \Omega(f_n)$.

Définition 57 (Θ) *On note $f_n \in \Theta(g_n)$ quand on a à la fois $f_n \in \mathcal{O}(g_n)$ et $f_n \in \Omega(g_n)$.*

4.3.3 Définition équivalente

Théorème 58 (Définition équivalente) *On a $f_n \in O(g_n)$ si et seulement si il existe une suite $(h_n)_{n\in\mathbb{N}}$, bornée, telle que $f_n = h_n g_n$ APCR.*

▶ Cette définition alternative est parfois plus simple à utiliser que la définition initiale.

4.3.4 Manipulation des $\mathcal{O}$

Définition 59 (Opérations sur des ensembles de suites) *Soit $(u_n)_{n\in\mathbb{N}}$ une suite réelle et F et G deux ensembles de suites. On utilise les définitions suivantes :*

$$\begin{aligned}
u_n + F &= \{(u_n + v_n)_{n\in\mathbb{N}} \mid (v_n)_{n\in\mathbb{N}} \in F\} \\
u_n \times F &= \{(u_n \times v_n)_{n\in\mathbb{N}} \mid (v_n)_{n\in\mathbb{N}} \in F\} \\
F + G &= \{(u_n + v_n)_{n\in\mathbb{N}} \mid (u_n)_{n\in\mathbb{N}} \in F,\ (v_n)_{n\in\mathbb{N}} \in G\} \\
F \times G &= \{(u_n \times v_n)_{n\in\mathbb{N}} \mid (u_n)_{n\in\mathbb{N}} \in F,\ (v_n)_{n\in\mathbb{N}} \in G\}
\end{aligned}$$

Théorème 60 (Règles de calcul) *On a les règles de calculs suivantes, pour* $(f_n)_{n\in\mathbb{N}}$ *et* $(g_n)_{n\in\mathbb{N}}$ *positive APCR :*

- $f_n \in \mathcal{O}(f_n)$.
- $c \times O(f_n) \subset \mathcal{O}(f_n)$, *pour* $c \in I\!R$.
- $\mathcal{O}(f_n) \times \mathcal{O}(g_n) \subset \mathcal{O}(f_n \times g_n)$.
- $\mathcal{O}(f_n) + \mathcal{O}(g_n) \subset \mathcal{O}(\max(f_n, g_n))$.
- *Si* $f_n \in \mathcal{O}(g_n)$ *alors* $\mathcal{O}(f_n) + \mathcal{O}(g_n) \subset \mathcal{O}(g_n)$.
- *Si* $f_n \prec g_n$ *alors* $f_n \in \mathcal{O}(g_n)$.
- *Si* $f_n \sim g_n$ *alors* $f_n \in \mathcal{O}(g_n)$.

► Attention, si $((f_n)_{n\in\mathbb{N}})$ est strictement positive APCR, $O(f_n)^{-1} = \Omega(1/f_n)$.

4.4 Approximation par intégrale

Soit f une fonction de $I\!R_*^+$ dans $I\!R$, monotone (on prendra croissante pour illustrer la méthode) et intégrable. On considère la série de terme général f_n : $S_n = \sum_{i=1}^{n} f(i)$, dont on cherche un développement asymptotique.

En utilisant la croissance de f, on a pour tout $x \in [i, i+1]$, avec $i \in \mathbb{N}^*$: $f(i) \leq f(x) \leq f(i+1)$. On intègre sur l'intervalle: $f(i) \leq \int_1^n f(x)dx \leq f(i+1)$. On somme pour i de 1 à $n-1$:

$$S_n - f_n \leq \int_1^n f(x)dx \leq S_n - f(1)$$

Et donc

$$\int_1^n f(x)dx + f(1) \leq S_n \leq \int_1^n f(x)dx + f_n$$

Ce qui suffit souvent à trouver un équivalent asymptotique à S_n.

Chapter 5

Complexité d'un algorithme

▶ **Important :** Ce chapitre est beaucoup plus de l'informatique que des mathématiques et se prête mal à des notes succinctes comme le reste du cours. En conséquence une partie des considérations du cours **ne sont pas présentes ci-dessous**.

5.1 Introduction

Définition 61 (Algorithme) *Un* algorithme *est un procédé automatique pour résoudre un problème en un nombre fini d'étapes.*

▶ Le but de ce chapitre est de donner des outils pour comparer différentes solutions algorithmiques à un problème donné.

▶ Pour quantifier les performances d'un algorithme on doit se munir d'une notion de *taille* sur les entrées.

▶ La *complexité* d'un algorithme est la quantité de ressources nécessaires pour traiter des entrées. On la voit comme une fonction de n, la taille de l'entrée.

▶ Les principales ressources mesurées sont le *temps* (nombre d'instructions utilisées) et l'*espace* (quantité d'espace mémoire nécessaire).

▶ On distingue plusieurs types d'analyses de complexité : l'analyse dans le *meilleur des cas*, le *pire des cas* et *en moyenne*. Pour ce cours on étudie exclusivement le pire des cas. Donc si $T(\mathcal{A})$ est le nombre d'instructions nécessaires pour que l'algorithme fasse ses calculs sur l'entrée $\mathcal{A}$, on s'intéresse à la suite $(t_n)_{n\in\mathbb{N}}$ définie par

$$t_n = \max_{|\mathcal{A}|=n} T(\mathcal{A}),$$

où $|\mathcal{A}|$ est la taille de l'entrée $\mathcal{A}$.

▶ Il n'est souvent pas pertinent d'essayer de quantifier trop précisément t_n, vu qu'on raisonne au niveau de l'algorithme et non d'une implatation. On se contente donc d'estimer t_n avec un ordre de grandeur en Θ ou $\mathcal{O}$. Un résultat typique : la complexité de l'algorithme de tri par insertion est en $\mathcal{O}(n^2)$.

5.2 Principes généraux

▶ **(ligne la plus effectuée)** La façon la plus simple dévaluer la complexité d'un algorithme est la suivante : un programme est constitué d'un nombre fini de lignes. Appelons-les ℓ_1 à ℓ_k. Soit n_1 ... n_k le nombre de fois qu'elles sont effectuées. La complexité de l'algo est $\sum \ell_i n_i$. Soit ℓ_j l'une des lignes qui est effectuée le plus souvent. Si toutes les lignes s'effectuent en temps $O(1)$, la complexité de l'algorithme est majorée par $kn_jO(1)$ soit $O(n_j)$.

▶ Attention aux instructions qui appellent d'autres fonctions et qui ne s'éxécutent pas en temps constant.

▶ Quand le programme contient une ou plusieurs boucles imbriquées, on se retrouve à estimer des sommes. On peut souvent utiliser le théorème suivant.

Théorème 62 (Sommation) *Soit $(f_n)_{n\in\mathbb{N}}$ et $(g_n)_{n\in\mathbb{N}}$ deux suites positives, avec $f_n \in \mathcal{O}(g_n)$. On suppose de plus que $\sum_{i=0}^{n} g_i$ n'est pas toujours nul. Alors*

$$\sum_{i=0}^{n} f_i = O\left(\sum_{i=0}^{n} g_i\right).$$

Plus généralement, si $\alpha : \mathbb{N} \to \mathbb{N}$ est une application croissante qui tend vers l'infini et $a \in \mathbb{N}$:

$$\sum_{i=a}^{\alpha(n)} f_i = O\left(\sum_{i=a}^{\alpha(n)} g_i\right).$$

▶ On en déduit le résultat suivant qui est très utilisé en pratique, vu que les complexités ont souvent cette forme :

Théorème 63 $(n^a \ln^b n)$ *Soit $(f_n)_{n\in\mathbb{N}}$ une suite telle que $f_n \in \mathcal{O}(n^a \ln^b n)$ avec $a, b \in \mathbb{R}^+$. Alors $\sum_{i=0}^{n} f_i \in \mathcal{O}(n^{a+1} \ln^b n)$. Le résultat est encore vrai si on remplace les $\mathcal{O}$ par des Θ.*

5.3 Classes de complexités classiques

On voit souvent apparaître les complexités suivantes :

• $\mathcal{O}(\log n)$ Ce sont des algorithmes très rapides. Exemples typiques : recherche dichotomique, exponentiation rapide, *etc.*

• $\mathcal{O}(n)$ (on dit *linéaire*). Typiquement quand on parcourt un tableau ou une liste un nombre borné de fois : recherche dans un tableau, minimum d'une liste, *etc.*

• $\mathcal{O}(n \log n)$. Vous l'avez principalement vu pour les algorithmes efficaces de tri : tri rapide, tri fusion, tri par tas, *etc.* Cette complexité apparaît régulièrement lorsque l'on fait du "diviser pour régner".

• $\mathcal{O}(n^2)$ (on dit *quadratique*). Quand on manipule des tableaux à deux dimensions, ou qu'on effectue un assez grand nombre de calculs sur un tableau à une dimension : somme de deux matrices, transposée d'une matrice, tri insertion, tri bulle, tri selection, *etc.*

5.4 Trois exemples importants

5.4.1 Dichotomie

▶ On veut chercher si un entier x est dans un tableau trié T de taille n.

```
int dicho(int *t, int n, int x) {
  int a,b,mid;
  a = 0;  b = n;
  while(a <= b) {
    mid = (b+a)/2;
    if (t[mid] == x)
      return 1;
    if (t[mid] < x)
      a = mid + 1;
    else
      b = mid - 1;
  }
  return 0;
}
```

Théorème 64 *La complexité de l'algorithme* `dicho` *est en* $\mathcal{O}(\log n)$.

5.4.2 Exponentiation rapide

▶ On veut calculer x^n, où $n \in \mathbb{N}$ mesure la taille de l'entrée.

```
float puissance(float x,int n) {
  if (n==0)
    return 1;
  if (n&1)
    return x*puissance(x,n-1);
  return puissance(x*x,n/2);
}
```

Théorème 65 *La complexité de l'algorithme* `puissance` *est en* $\mathcal{O}(\log n)$.

5.4.3 Schéma de Hörner

▶ On veut calculer $P(x)$, où x est un réel (`float`) et P est un polynôme de degré n, réprésenté par un tableau : si $P(X) = a_0 + a_1X + \ldots + a_nX^n$, alors pour tout $i \in \{0, \ldots, n\}$ on a $P[i] = a_i$.

```
float Horner(float P[], int n, float x) {
  int i; float r = P[n];
  for(i=n-1;i>=0;i--)
    r = (r*x)+P[i];
  return r;
}
```

Théorème 66 *La complexité de l'algorithme* `Horner` *est en* $\mathcal{O}(n)$.