Booléens et codages

UE Informatique, M1 Mathématiques et applications,
EADS, Université d'Aix-Marseille *

2024–2025

Notations

On note $\mathbf{N}$, $\mathbf{Z}$, $\mathbf{Q}$, $\mathbf{R}$ les ensembles de nombres usuels. Si $a \in \mathbf{N}$, on note $[\![a]\!] = \{i \in \mathbf{N} \mid i < a\}$; et si $a, b \in \mathbf{Z}$, on note $[\![a{:}b]\!] = \{i \in \mathbf{Z} \mid a \leq i < b\}$ – donc $[\![a]\!] = [\![0{:}a]\!]$ quand $a \in \mathbf{N}$. Pour tout ensemble X, on note $X^\bullet = X \setminus \{0\}$ et $\mathcal{P}(X)$ l'ensemble des parties de X.

1 Booléens et mots binaires

On appelle *booléen*[1] l'un des éléments d'un ensemble $\mathbf{B}$ à deux éléments, interprétés comme des valeurs de vérité : typiquement $\{0, 1\}$, $\{\mathsf{Faux}, \mathsf{Vrai}\}$ ou $\{\bot, \top\}$, où les valeurs $1, \mathsf{Vrai}, \top$ représentent le « vrai », et $0, \mathsf{Faux}, \bot$ le « faux ». Dans la suite, on fixe $\mathbf{B} = \{0, 1\} = [\![2]\!]$ avec cette interprétation. On parle aussi de *bit* (« petit morceau » en anglais, mais aussi contraction de *binary digit*, « chiffre binaire ») : le bit 1 est « présent » ou « activé », et le bit 0 est « absent » ou « désactivé ».

1.1 Logique booléenne

L'interprétation des booléens comme valeurs de vérité définit ce qu'on appelle la *logique booléenne*, c'est-à-dire l'interprétation des connecteurs logiques par leur action sur les valeurs booléennes, définie par des *tables de vérité* (qui ne sont rien d'autre que les tables de calcul pour les opérations booléennes).

Les connecteurs (ou opérations) les plus classiques sont :

— la *négation* (le « non »), notée $\neg$,

— la *conjonction* (le « et »), notée $\wedge$,

— la *disjonction* (le « ou »), notée $\vee$,

— l'*implication* notée $\Rightarrow$,

— l'*équivalence* notée $\Leftrightarrow$,

— la *disjonction exclusive* (le « ou exclusif », ou *XOR*), notée $\oplus$,

1. Du nom du logicien anglais George Boole, `https://fr.wikipedia.org/wiki/George_Boole`[W].

qui sont définies par les tables :

x	$\neg x$
0	1
1	0

et

x	y	$x \wedge y$	$x \vee y$	$x \Rightarrow y$	$x \Leftrightarrow y$	$x \oplus y$
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	0	0	1
1	1	1	1	1	1	0

.

1.1.1. Exercice (Opérations booléennes)**.**

1. Combien y a-t-il d'opérations boolénnes binaires (c'est-à-dire de fonctions $\mathbf{B}^2 \to \mathbf{B}$) ?
2. Montrez que toutes ces opérations peuvent s'écrire en utilisant uniquement les connecteurs $\wedge$, $\vee$ et $\neg$.
3. Combien y a-t-il d'opérations boolénnes binaires commutatives ?
4. Montrez que les seules opérations boolénnes binaires commutatives et associatives sont les fonctions constantes, et les quatre opérations $\wedge$, $\vee$, $\Leftrightarrow$ et $\oplus$. De plus $\wedge$ et $\Leftrightarrow$ (resp. $\vee$ et $\oplus$) admettent 1 (resp. 0) comme élément neutre. ◇

Ces opérations correspondent à plusieurs structures dont l'ensemble $\mathbf{B}$ peut être muni. On en distingue ici deux (on en verra d'autres) :

(i) On considère l'ordre usuel $\leq$ sur les booléens : c'est la seule relation d'ordre (large) telle que $0 \leq 1$. Cet ordre est évidemment total, et on observe directement que $x \wedge y = \min(x, y)$ et $x \vee y = \max(x, y)$. Par ailleurs, cet ordre est celui de l'implication : on a $x \leq y$ ssi $(x \Rightarrow y) = 1$.

(ii) On peut également identifier $\mathbf{B}$ avec le corps $\mathbf{Z}/2\mathbf{Z}$, avec $\oplus$ comme addition et $\wedge$ comme multiplication (et alors $x \vee y = x + y - xy$ et $x \Leftrightarrow y = x + y + 1$).

1.1.2. *En Python*  . Les booléens sont représentés par les constantes `True` et `False` du type `bool`. Les tests (par exemple le test d'égalité `a == b` entre deux objets `a` et `b`) sont des expressions dont l'évaluation produit un booléen ; et on peut assembler ces tests en expressions booléennes composées au moyen des connecteurs `and` ($\wedge$), `or` ($\vee$) et `not` ($\neg$) : la question 2 de l'exercice 1.1.1 montre que ces connecteurs sont suffisants pour exprimer toutes les combinaisons possibles.

1.2 Mots binaires

Un *mot binaire* est une suite finie de bits $u = (b_1, \dots, b_n) \in \mathbf{B}^n$: n est la longueur de u, notée $|u|$, et on note simplement $u = b_1 \cdots b_n$. Lorsqu'une ambigüité est possible entre l'écriture d'un mot binaire et celle d'un nombre (par exemple 10), on pourra souligner le mot binaire ($\underline{10} = (1, 0)$).

1.2.1. *Exemple* (Les octets). Une unité d'information courante est l'*octet* : un mot binaire de longueur 8. Il y a $2^8 = 256$ octets possibles.

Les mots binaires jouent un rôle primordial dans la représentation de l'information. Une première observation est qu'un mot binaire de longueur n peut être identifié avec l'ensemble des positions des ses bits activés : chaque bit représente un élément d'information qui peut être présent (1) ou absent (0). Formellement :

1.2.2. Proposition. *Étant donné un ensemble fini X à n éléments, et une énumération $x_1, \dots, x_n$ des éléments de x, la fonction*

$$b_1 \cdots b_n \mapsto \{x_i \mid b_i = 1\}$$

définit une bijection de $\mathbf{B}^n$ dans $\mathcal{P}(X)$.

1.2.3. Définition (Ordre sur les mots binaires)**.** On étend l'ordre des booléens à chaque $\mathbf{B}^n$, composante par composante :

$$b_1 \cdots b_n \leq b'_1 \cdots b'_n \qquad \text{ssi} \qquad \forall i \in [\![n]\!],\ b_i \leq b'_i\,.$$

C'est une application de la construction standard du produit d'ensembles ordonnés : c'est bien un ordre large, mais il n'est plus total dès que $n > 1$ (on n'a ni $\underline{01} \leq \underline{10}$ ni $\underline{10} \leq \underline{01}$).

1.2.4. Proposition. *La bijection définie dans la proposition 1.2.2 définit un isomorphisme d'ordres (c'est-à-dire une bijection croissante) de* $(\mathbf{B}^n, \leq)$ *vers* $(\mathcal{P}(X), \subseteq)$.

De la même manière, on peut étendre les opérations booléennes aux mots binaires :

1.2.5. Définition (Ordre et opération sur les mots binaires)**.** On étend les opérations des booléens à chaque $\mathbf{B}^n$, composante par composante :

$$\neg(b_1 \cdots b_n) = \neg b_1 \cdots \neg b_n$$
$$b_1 \cdots b_n \oplus b'_1 \cdots b'_n = (b_1 \oplus b'_1, \dots, b_n \oplus b'_n)$$
$$\textit{etc.}$$

En particulier, les opérations $\wedge$, $\vee$, $\oplus$ et $\Leftrightarrow$ restent associatives et commutatives, les distributivités restent valides, et les unités sont définies composante par composante (par exemple, l'unité du $\wedge$ sur $\mathbf{B}^n$ est $\underline{1}^n = \underline{1 \cdots 1}$).

1.2.6. Exercice. À quelles opérations ensemblistes correspondent les six opérations booléennes définies en section 1.1 à travers l'isomorphisme de la proposition 1.2.2 ? ◇

1.3 Fonctions booléennes

On appelle *fonction booléenne* toute fonction sur les mots binaires $f : \mathbf{B}^n \to \mathbf{B}^k$ avec $n, k \in \mathbf{N}$: n est l'*arité* de f ou sa *dimension d'entrée* et k est sa *dimension de sortie*. Dans le cas où $k = 1$, on parle aussi d'*opération booléenne* d'arité n, ou *opération booléenne n-aire*, et les opérations booléennes déjà rencontrées sont unaire (la négation) ou binaires (les autres). Une fonction booléenne $\mathbf{B}^n \to \mathbf{B}^k$ n'est rien d'autre qu'un k-uplet d'opérations booléennes n-aires.

1.3.1. Exercice (Généralités sur les fonctions booléennes)**.**

1. Combien y a-t-il d'opérations booléennes n-aires ? Et combien y a-t-il de fonctions $\mathbf{B}^n \to \mathbf{B}^k$?
2. Montrez qu'on peut écrire toute opération booléenne en n'utilisant que les constantes 0 et 1, et les opérations unaire $\neg$ et binaires $\wedge$, $\vee$. ◇

Les opérations associatives et commutatives $\wedge$, $\vee$ et $\oplus$ s'étendent naturellement en des opérations d'arité quelconque avec : [2]

$$\wedge(b_1 \cdots b_n) = \begin{cases} 1 & \text{si } b_i = 1 \text{ pour tout } i \in [\![n]\!] \\ 0 & \text{sinon} \end{cases}$$

$$\vee(b_1 \cdots b_n) = \begin{cases} 1 & \text{s'il existe } i \in [\![n]\!] \text{ tel que } b_i = 1 \\ 0 & \text{sinon} \end{cases}$$

$$\oplus(b_1 \cdots b_n) = \begin{cases} 1 & \text{si le nombre de } i \in [\![n]\!] \text{ tels que } b_i = 1 \text{ est impair} \\ 0 & \text{sinon} \end{cases}$$

2. L'opération $\Leftrightarrow$ peut s'étendre de la même manière mais elle n'est pas associée à une structure intéressante donc on la néglige ici.

(et le cas où $n = 0$ donne la fonction constante égale à l'unité de l'opération). Du point de vue de l'ordre sur $\mathbf{B}$, on peut de manière équivalente poser :

$$\wedge(b_1 \cdots b_n) = \max(b_1, \dots, b_n) \qquad \text{et} \qquad \vee(b_1 \cdots b_n) = \min(b_1, \dots, b_n)\,.$$

Et, pour la structure de corps sur $\mathbf{B}$ donnée par son identification avec $\mathbf{Z}/2\mathbf{Z}$, on a aussi :

$$\wedge(b_1 \cdots b_n) = \prod_{i=1}^{n} b_i \qquad \text{et} \qquad \oplus(b_1 \cdots b_n) = \sum_{i=1}^{n} b_i\,.$$

1.3.2. Exercice (Opérations n-aires à partir des opérations binaires)**.** Comme ces versions n-aires sont des opérations booléennes, l'exercice précédent montre qu'on peut les écrire en n'utilisant que les constantes et les opérations unaires et binaires. De fait, pour $n > 0$, la version n-aire d'une opération binaire associative, commutative et munie d'un élément neutre se calcule en appliquant $n - 1$ fois l'opération binaire.

Montrez qu'on ne peut pas faire mieux : une expression booléenne[3] utilisant k connecteurs binaires (en plus de connecteurs unaires et de constantes) ne peut dépendre que d'au plus $k + 1$ arguments (quand $k < n - 1$, ceci exclut le $\wedge$ n-aire, qui dépend de tous ses arguments). ◇

2 Codages binaires de longueur fixée

L'interprétation ensembliste n'est qu'une interprétation possible des mots booléens, parmi quantité d'autres. L'idée est que toute information finie peut être représentée par un mot booléen, une fois qu'on a fixé un codage approprié.

2.0.1. Définition (Codage sur n bits)**.** Soit X un ensemble fini : une *convention de codage sur n bits* des éléments de X est donné par une fonction $\gamma : X \to \mathbf{B}^n$ (la fonction de *codage*) et une fonction partielle $\delta : \mathbf{B}^n \rightharpoonup X$ (la fonction de *décodage*) telles que $\delta(\gamma(x)) = x$ pour tout $x \in X$. Un *code* de x selon δ est n'importe quelle préimage u de x par δ : $\delta(u) = x$ (la fonction de codage choisit un de ces codes).

Comme on demande que $\delta \circ \gamma$ soit la fonction identité de X, γ est nécessairement injective (deux éléments différents ont des codages différents) et δ est nécessairement surjective (tout élément admet un code). En général, on ne demande par contre pas que γ soit surjective (un mot binaire peut ne jamais être le codage choisi pour un élément), ni même que δ soit totalement définie (un mot binaire peut ne pas être un code valide), ni encore que δ soit injective (un élément peut être l'image de plusieurs codes).

2.0.2. Exercice (Généralités sur les codages de longueur fixée)**.**

1. À quelle condition un ensemble X peut-il admettre un codage sur n bits ?
2. Vérifiez que, dès que X est non vide, toute convention de codage peut-être étendue pour que la fonction de décodage soit totale.
3. À quelle condition un ensemble X peut-il admettre un codage sur n bits, avec une fonction de codage surjective (ou, de manière équivalente, une fonction de décodage totale et injective) ? ◇

3. On formalisera la notion d'expression booléenne dans un chapitre futur, mais on se contente d'une notion intuitive ici : c'est une « expression », avec des constantes, des variables (les bits d'entrée) et des opérations unaires ou binaires.

2.1 Codage binaire usuel des entiers naturels

À tout mot binaire $u = b_{k-1}\cdots b_0$, de longueur k, on peut associer l'entier naturel $\delta_2(u) = \sum_{i=0}^{k-1} b_i 2^i < 2^k$, ce qui définit une bijection de $\mathbf{B}^k$ dans $[\![2^k]\!]$. [4]

Bien sûr, on pourrait choisir un codage suivant un principe différent :

2.1.1. *Exemple* (Codage gros-boutiste). Dans le codage précédent, on a choisi de commencer par le bit de poids fort, comme c'est l'usage quand on écrit les nombres dans une base fixée – mais ça nous force à indicer les bits à l'envers. On parle de codage petit-boutiste (*little-endian*). [5] On peut plutôt considérer : $b_0\cdots b_{k-1} \mapsto \sum_{i=0}^{k-1} b_i 2^i < 2^k$, qui est un codage tout aussi valide, dit gros-boutiste (*big-endian*).

2.1.2. *Exemple* (Codage unaire des entiers naturels de valeur bornée). À tout mot binaire $u = b_0\cdots b_{k-1}$, de longueur k, on peut associer l'entier naturel $\delta_1(u) = \sum_{i=0}^{k-1} b_i \leq k$, ce qui définit une surjection de $\mathbf{B}^k$ dans $[\![k+1]\!]$. On peut fixer un codage réciproque, par exemple en posant $\gamma_1(i) = \underline{1}^i\underline{0}^{k-i}$ (i bits 1, suivis de $k-i$ bits 0).

On a tout de même de nombreuses raisons de préférer le codage en base 2 (petit ou gros-boutiste, peut importe) : à longueur fixée, on code bien plus d'entiers naturels ; le décodage est injectif (on n'a pas à choisir un codage arbitraire) ; et on sait depuis l'école que, lorsqu'on manipule de grands nombres, il est plus facile de travailler avec leur écriture dans une base fixée (10 la plupart du temps, 2 un peu plus tard, et une base quelconque dès qu'on a fait un peu d'arithmétique).

En effet, de même qu'on peut coder des informations variées comme des mots binaires, on voudra représenter des opérations, et plus généralement des fonctions, agissant sur ces objets à travers le codage.

2.1.3. Exercice (Addition dans $\mathbf{Z}/4\mathbf{Z}$). En identifiant $\mathbf{B}^4$ avec $\mathbf{B}^2\times\mathbf{B}^2$, on peut voir toute fonction $\mathbf{B}^4 \to \mathbf{B}^2$ comme une opération binaire sur $\mathbf{Z}/2^2\mathbf{Z} = \mathbf{Z}/4\mathbf{Z}$, modulo codage (disons petit-boutiste). Vérifiez alors que la fonction

$$\begin{aligned}\mathbf{B}^4 &\to \mathbf{B}^2\\ b_1b_0c_1c_0 &\mapsto ((b_0\wedge c_0)\oplus b_1\oplus c_1)(b_0\oplus c_0)\end{aligned}$$

représente l'addition, c'est-à-dire que $\delta_2((b_0\wedge c_0)\oplus b_1\oplus c_1, b_0\oplus c_0) = \delta_2(b_1b_0)+\delta_2(c_1c_0) \mod 4$. ⋄

L'idée de la fonction de l'exercice précédent est que chaque $\oplus$ représente une somme de bits modulo 2, et $b_0\wedge c_0$ calcule la retenue éventuelle. Une bonne notion pour généraliser cette construction est celle de circuit booléen : on y reviendra dans la dernière période du cours. Une autre approche est de décrire algorithmiquement l'opération d'addition (par exemple) sur la représentation binaire des entiers : c'est ce qu'on fera dans la section suivante.

2.1.4. *En Python* 🐍. Les entiers de Python correspondent au type `int`. Il s'agit d'entiers signés, c'est-à-dire qu'ils représentent des entiers relatifs (dans $\mathbf{Z}$), et pas seulement des entiers naturels (dans $\mathbf{N}$). Par ailleurs, la taille de ces entiers n'est pas bornée ! On y reviendra donc en détail plus tard.

On peut toutefois déjà expliquer les opérateurs `&`, `|` et `^`. Chacun réalise une opération booléenne sur les mots binaires représentant les nombres : $\wedge$ pour `&`, $\vee$ pour `|`, $\oplus$ pour `^`. Ceci explique pourquoi, si on a l'habitude de LaTeX, on se fait parfois surprendre par `2^2 == 0`…

4. On reviendra sur la justification de cette propriété, et plus généralement sur le codage des entiers et d'autres nombres dans le chapitre suivant.

5. `https://fr.wikipedia.org/wiki/Boutisme`[W]

2.2 Codages de caractères

2.2.1. *Notation* (Octets). Dans la suite, on confondra souvent un octet $u \in \mathbf{B}^8$ avec l'entier $\delta_2(u) \in [\![2^{|u|}]\!]$ qu'il code. Par ailleurs, on abrègera la notation, en oubliant les 0 initiaux de u (ce qui ne change pas $\delta_2(u)$). On se permettra donc de prétendre que $\underline{10}$ est un octet, et d'écrire $\underline{10} = 2$ (on rappelle qu'on souligne les écritures binaires pour éviter la confusion avec les écritures décimales).

Comme pour les nombres, il est crucial en informatique de disposer d'un codage des caractères pour pouvoir traiter du texte (le lire, l'afficher, l'écrire dans un fichier, *etc.*). Mais contrairement aux nombres, il n'y a pas de codage vraiment canonique et tout est affaire de convention.

2.2.2. *Exemple* (Codages de caractères sur un octet). Les codages « historiques » fonctionnaient sur 8 bits (ou même 7 bits, le huitième étant réservé pour des usages spécifiques), c'est-à-dire un octet par caractère. On peut donc fixer un codage pour 256 caractères différents, ce qui semble bien suffisant… si on se limite à l'anglais courant : il faut déjà compter les 26 lettres de l'alphabet, en majuscule et en minuscule (52 lettres donc), les 10 chiffres arabes, les signes de ponctuation, quelques symboles mathématiques et informatiques, *etc.*

On arrive vite à une petite centaine de caractères imprimables, plus les caractères « blancs » (espace, saut de ligne, tabulation), auxquels il faut ajouter quelques caractères de contrôle (historiquement utilisés pour communiquer avec les dispositifs d'entrée et d'affichage) : on peut coder tout ça sur 7 bits, et c'est ce que fait le standard ASCII[6]. Le codage est choisi pour respecter quelques contraintes basiques : les caractères imprimables sont rassemblés en une unique plage (de $\underline{0100001} = 33$ à $\underline{1111110} = 126$) ; les chiffres, les lettres minuscules et les lettres majuscules occupent trois plages séparées, chacune respectant l'ordre numérique ou alphabétique ; on passe d'une lettre minuscule à la majuscule correspondante en passant un bit fixé (celui de valeur $2^6 = 32$) de 1 à 0. Par exemple, $\gamma_{\texttt{ASCII}}(\texttt{A}) = \underline{1000001} = 65$, $\gamma_{\texttt{ASCII}}(\texttt{B}) = \underline{1000010} = 66$, $\gamma_{\texttt{ASCII}}(\texttt{Z}) = \underline{1011010} = 90 = 65 + 25$, et $\gamma_{\texttt{ASCII}}(\texttt{a}) = \underline{1100001} = 97 = 65 + 2^6$.

En exploitant le 8e bit, on peut doubler le nombre de codes disponibles, ce qui permet d'étendre la table des caractères. C'est nécessaire dès qu'on veut utiliser des caractères accentués par exemple. Sauf qu'il n'y a de place que pour 128 nouveaux caractères, et chaque langue a besoin de caractères différents, et donc d'une extension différente : c'est l'enfer des pages de code.[7]

2.2.3. *Exemple* (Unicode). Le standard Unicode[8] a pour but de permettre le codage de tout texte écrit, en particulier en associant à chaque caractère de chaque système d'écriture un unique *point de code*[9] : ce dernier est simplement un entier, pris dans $[\![17 \times 2^{16}]\!]$ (ce qui laisse beaucoup de place : l'ambition du standard est bien de représenter tout ce qui s'écrit dans le monde, et d'organiser rationnellement cette table de caractères). Par souci de compatibilité, les caractères ASCII conservent le même point de code dans Unicode : par exemple la lettre latine `a` a le point de code 97.

Par ailleurs, le standard prévoit plusieurs codages binaires des points de code (qui sont des entiers). Le plus simple à décrire est le codage `UTF-32`, un codage à taille fixe au sens de la

6. `https://fr.wikipedia.org/wiki/ASCII`[W]
7. `https://fr.wikipedia.org/wiki/Page_de_code`[W]
8. `https://fr.wikipedia.org/wiki/Unicode`[W]
9. Attention, il y a une subtilité : un caractère est ici considéré comme un signe graphique muni d'une signification propre. Ce peut être une lettre simple (par exemple `a` vue comme une lettre de l'alphabet latin), une lettre accentuée (par exemple `â`), une lettre avec une signification particulière (par exemple la notation $\mathbf{N}$ pour l'ensemble des entiers naturels, à distinguer de la majuscule latine `N`), un autre caractère imprimable (par exemple `^` vu comme caractère à part entière), un signe diacritique (par exemple `^` vu comme un accent qui peut compléter une lettre `a` pour obtenir `â`), *etc.* Chacun est muni d'un point de code distinct.

definition 2.0.1 : chaque point de code est représenté sur 32 bits. C'est aussi un codage très inefficace : $17 \times 2^{16} = 2^{20} + 2^{16} < 2^{21}$. On reviendra sur les autres codages d'Unicode dans la section suivant : il s'agit de codages à taille variable, ce qui nous demandera d'introduire d'autres notions.

2.2.4. *En Python* . Les caractères en Python sont des objets du type `str` : ce sont des chaînes de caractères de longueur 1 – là encore, on reviendra sur les chaînes de caractères bientôt. Python prend nativement en charge le standard Unicode : ses caractères sont exactement des points de code. La fonction `ord()` permet de récupérer le point de code d'un caractère en tant qu'entier : par exemple `ord('a')== 97`. Et réciproquement, `chr()` produit le caractère correspondant à un point de code : `chr(97) == 'a'`.

2.2.5. Exercice (Chiffre de César sur un caractère)**.** Définissez une fonction Python suivant le prototype suivant :

```
def cesar1(c, k):
    """Retourne le caractère associé à `c` par le chiffre de César de clé `k`.
    La chaîne `c` doit être réduite à un caractère et `k` doit être un entier.
    Si `c` est une lettre de l'alphabet (de 'a' à 'z', ou de 'A' à 'Z')
    le résultat est obtenu en renvoyant la `k`-ième lettre après `c`,
    quitte à revenir au début de l'alphabet,
    en conservant la casse (majuscule ou minuscule).
    Dans les autres cas, on renvoie `c`. """
    ...
```

Par exemple `cesar1('a',3)` doit renvoyer `'d'` et `cesar1('Z',1)` doit renvoyer `'A'`. Vous pouvez au besoin consulter la page Wikipédia : `https://fr.wikipedia.org/wiki/Chiffrement_par_décalage`[W]. ◇

3 Données de taille non bornée

L'algorithmique et l'informatique ne se limitent pas au traitement de données choisies parmi un ensemble fini (aussi grand soit-il). On s'intéresse plutôt à des objets certes finis (puisqu'on veut pouvoir les consulter ou les produire en temps fini) mais de taille non bornée *a priori* : les nombres entiers, les approximations décimales de π, les chaînes de caractères, les documents PDF, les sessions de connexion à un serveur, les programmes Python, les équations diophantiennes, les tables dans une base de données, *etc.* – tous ces objets se décrivent de manière finie, mais la taille de cette description n'est pas fixée à l'avance.

La manière la plus basique et universelle de représenter des données de taille non bornée est tout simplement de considérer des suites finies de symboles : des mots. L'ensemble des mots est naturellement muni d'une structure de *monoïde*.

3.1 Le monoïde des mots

On a déjà croisé plusieurs opérations associatives, chacune munie d'un élément neutre : c'est la notion de monoïde.

3.1.1. Définition (Monoïde)**.** Un *monoïde* est un ensemble M muni d'une opération binaire $\cdot$ associative, admettant un élément neutre – c'est-à-dire qu'il existe $e \in M$ tel que $e \cdot x = x \cdot e = x$ pour tout $x \in M$. Ce monoïde est *commutatif* si $\cdot$ est commutative.

Un monoïde, c'est donc « comme un groupe », mais sans demander l'existence d'un inverse pour chaque élément. C'est une structure bien plus générale que celle de groupe – et sa théorie

est bien moins riche – mais elle intervient de manière assez courante en informatique et en mathématiques discrètes.

3.1.2. *Exemple* (Monoïdes). On a déjà vu plusieurs structures de monoïde sur $\mathbf{B}$: une pour chacune des opérations $\wedge$, $\vee$, $\Leftrightarrow$ et $\oplus$, toutes commutatives. Un exemple de monoïde non commutatif est l'ensemble des fonctions $X \to X$ muni de la composition (en supposant que X a au moins deux éléments). Tout groupe est *a fortiori* un monoïde. On peut également considérer l'addition sur $\mathbf{N}$, ou la multiplication dans un anneau quelconque : aucune n'est une loi de groupe, mais c'est bien une loi de monoïde.

On engendre un monoïde librement à partir de n'importe quel ensemble, en considérant les suites finies d'éléments : [10]

3.1.3. Définition (Mots)**.** On fixe un ensemble A, qu'on appelle *alphabet*, et on appelle *symboles* ou *lettres* les éléments de A. Un *mot* sur l'alphabet A est une suite finie de symboles $u = (a_0, \dots, a_{n-1}) \in A^n$: n est la *longueur* de u, notée $|u|$, et on note simplement $u = a_0 \cdots a_{n-1}$. On note $A^* = \bigcup_{n\in\mathbf{N}} A^n$ l'ensemble des mots sur A. On note $\varepsilon = ()$ le *mot vide*, qui est l'unique élément de A^0.

3.1.4. *Observation.* Un mot binaire n'est rien d'autre qu'un mot sur $\mathbf{B}$. Par ailleurs, $\emptyset^* = \{\varepsilon\}$, et A^* est infini dès que A est non vide.

3.1.5. Définition (Opérations sur les mots)**.** Étant donnés $u = a_0 \cdots a_{n-1} \in A^*$ et $i \in [\![n]\!]$, on note $u[i] = a_i$. Si de plus $v = b_0 \cdots b_{p-1} \in A^*$, on note $u \cdot v = a_0 \cdots a_{n-1} b_0 \cdots b_{p-1}$ la *concaténation* des mots u et v.

3.1.6. Proposition (Structure de monoïde sur A^*)**.** *La concaténation est associative et ε est son élément neutre.*

3.1.7. Définition (Morphisme de monoïdes)**.** Étant donnés deux monoïdes $(M, \cdot, e)$ et $(M', \cdot', e')$, une fonction $f : M \to M'$ est un *morphisme de monoïdes* si $f(e) = e'$ et $f(x \cdot y) = f(x) \cdot' f(y)$ pour tous $x, y \in M$. C'est un *isomorphisme* s'il est bijectif.

3.1.8. Exercice (Généralités sur le monoïde des mots)**.**

1. Vérifiez que le monoïde A^* est non-commutatif dès que A a au moins deux éléments.
2. Vérifiez que la longueur $u \mapsto |u|$ est un morphisme de monoïdes de A^* dans $\mathbf{N}$ muni de l'addition, et que c'est un isomorphisme ssi A est un singleton.
3. Vérifiez que A^* est dénombrable ssi A l'est. ◇

Dans la suite, on considèrera presque toujours le cas où l'alphabet A est fini (donc A^* est dénombrable), de cardinal $\#A \geq 2$ (et donc on a $\#A^n \geq 2^n$ mots de longueur n). On garde cette hypothèse implicite, et on mentionnera explicitement les cas où $\#A < 2$ ou bien $\#A$ est infini.

3.1.9. *Notation.* Étant donné un mot u, on note $u^n = u \cdot \cdots \cdot u$ la concaténation de n copies de u, et $\overline{u}$ le mot obtenu en renversant u : $\overline{a_0 \cdots a_{n-1}} = a_{n-1} \cdots a_0$. .

3.1.10. *En Python* 🐍. Il y a plusieurs types susceptibles de représenter des suites finies, les plus élémentaires étant `str`, `tuple`, `list`, `bytes` et `bytearray`. En particulier, ces types admettent chacun une représentation du mot vide (respectivement `""`, `()`, `[]`, `b""` et `bytearray()`) et sont munis d'une opération de concaténation avec l'opérateur `+` (le même que pour la somme des nombres). La longueur se calcule avec `len()`, et `u[i]` donne l'élément numéro `i` quand `0 <= i < len(u)`. L'itération d'un mot se note comme la multiplication par un entier : `3*"a" == "aaa"`.

10. Voir aussi `https://fr.wikipedia.org/wiki/Monoïde#Monoïde_libre`[W].

Les types `str` des *chaînes de caractères*, `bytes` des *chaînes d'octets*, et `bytearray` des *tableaux d'octets* correspondent au cas où l'alphabet est fini : les éléments d'une chaîne de caractères sont des points de code dans le standard Unicode ; les éléments d'une chaîne ou d'un tableau d'octets sont des… octets. On a déjà vu que les caractères en Python sont des chaînes de longueur 1 : en particulier, `"a"[0] == "a"`. D'une certaine manière, Python travaille modulo l'identification entre A^1 et A. Les types `bytes` et `bytearray` diffèrent l'un de l'autre par la manière dont on peut construire et modifier un mot : en particulier, on ne peut pas changer les octets d'une chaîne, mais on peut changer ceux d'un tableau (on y reviendra dans un prochain chapitre).

A contrario, les « symboles » des mots de type `tuple` ou `list` sont des objets Python quelconques, ou plutôt des références (des adresses en mémoire) vers des objets Python quelconques. Suivant le point de vue, on peut considérer que l'alphabet est infini (on ne pose pas de limite *a priori* sur la taille des objets qu'on considère, tant que ça rentre dans la mémoire de la machine) ou fini (une adresse est un entier de taille fixée par l'architecture, généralement dans $[\![2^k]\!]$ avec $k \leq 64$). Là encore `tuple` et `list` diffèrent l'un de l'autre par la manière dont on peut construire et modifier un mot : `list` est plus permissif.

3.1.11. Exercice (Programmes élémentaires sur les mots)**.**

1. Écrivez une fonction `gonfle(s:str,k:int)->str` qui prend en argument une chaîne de caractères `s`, et un entier `k`, et retourne la chaîne obtenue en itérant `k` fois chaque caractère de `s` : `gonfle("ab",3)=="aaabbb"`.
2. Écrivez une fonction `palindrome(u)` qui teste si le mot `u` (de n'importe quel type séquentiel) est un palindrome. ◇

3.2 Cas d'un codage de longueur fixe

La notion de codage binaire de taille fixe (definition 2.0.1) se généralise directement à n'importe quel alphabet : on peut représenter les éléments d'un ensemble fini X par des codes choisis dans A^n dès que $\#X \leq \#A^n$.

Étant donné un codage de longueur fixée $\gamma : X \to A^n$, lorsqu'on veut représenter une suite finie $u \in X^*$, il suffit de concaténer les codes des symboles de u. Formellement, si $u = x_0 \cdots x_{k-1}$, on pose

$$\gamma^*(u) = \gamma(x_0) \cdot \cdots \cdot \gamma(x_{k-1}) \in A^{kn}\,.$$

3.2.1. Théorème (Extension d'un codage des symboles aux mots)**.** *La fonction γ^* est l'unique morphisme de monoïdes $X^* \to A^*$ qui coïncide avec γ sur $X = X^1$. Ce morphisme est de plus injectif.*

Démonstration. Le premier résultat se déduit directement des définitions, par récurrence sur la longueur des mots dans X^*. Pour l'injectivité, on montre que $\gamma(u) = \gamma(v)$ implique $u = v$ par récurrence sur $|u| + |v|$: le cas de base et direct, et le pas de récurrence se déduit de l'injectivité de γ. □

Ceci permet de composer les codages :

3.2.2. *Observation.* Étant donnés un codage γ_1 des éléments de A par des codes dans B^n, et un codage γ_2 des éléments de B par des codes dans C^k, on obtient un codage $\gamma_2^* \circ \gamma_1$ des éléments de A par des codes dans C^{kn} et $(\gamma_2^* \circ \gamma_1)^* = \gamma_2^* \circ \gamma_1^*$ est un morphisme injectif de A^* dans C^*.

3.2.3. *Exemple* (Unicode vers UCS-32 vers octets vers binaire). On a vu qu'à chaque point de code Unicode x est associé un mot binaire $\gamma_{\text{UCS-32}}(x) \in \mathbf{B}^{32}$, qu'on peut considérer comme un

unique symbole dans $[\![2^{32}]\!]$. Un tel nombre peut être représenté sur quatre octets, pour donner un mot de longueur 4 sur $[\![2^8]\!]$. Et chaque octet est lui-même un mot de longueur 8 sur $[\![2]\!] = \mathbf{B}$: le mot binaire de départ, de longueur $32 = 4 \times 8$ se retrouve par concaténation des octets. Une chaîne Unicode de longueur n codée en UCS-32 peut donc être vue comme un mot de longueur n sur $[\![2^{32}]\!]$, un mot d'octets de longueur $4n$, ou un mot binaire de longueur $32n$.

3.3 Codes de longueur variable

On peut vouloir faire varier la longueur des codes, par exemple en donnant des codes plus courts aux éléments les plus courants, dans le but de réduire la taille du code total – cette problématique générale est celle de la *compression*, vaste domaine [11] qu'on ne couvrira pas ici. Le principe de codes de longueurs variable est cependant déjà à l'œuvre dans le codage UTF-8 d'Unicode (voir ci-dessous).

Si le codage du mot est obtenu en concaténant les codes des symboles, mais que ces derniers sont de longueur variable, il faut faire attention à conserver un décodage univoque (c'est-à-dire un codage injectif) :

3.3.1. *Exemple.* Si on pose $\gamma(a) = \underline{1}$, $\gamma(b) = \underline{0}$ et $\gamma(c) = \underline{10}$, on a bien trois codes différents, mais alors $\gamma^*(c) = \underline{10} = \gamma^*(ab)$: la fonction obtenue n'est plus injective !

Une solution est de s'assurer qu'aucun code de symbole n'est préfixe d'un autre.

3.3.2. Définition (Préfixe, suffixe)**.** Un *préfixe* (resp. *suffixe*) du mot u est un mot v_1 tel qu'on puisse écrire u sous la forme $u = v_1 \cdot v_2$ (resp. $u = v_2 \cdot v_1$).

3.3.3. Définition (Codage préfixe)**.** Soit X un ensemble fini et A un alphabet : un *codage préfixe* des éléments de X est donné par une fonction $\gamma : X \to A^* \setminus \{\varepsilon\}$ telle que pour tous $x, y \in X$, $\gamma(x)$ est un préfixe de $\gamma(y)$ ssi $x = y$.

3.3.4. *Observation.* En particulier γ est injective, et on définit une fonction partielle surjective δ inverse de γ en posant $\delta(u) = x$ si $u = \gamma(x)$ (au plus un $x \in X$ satisfait cette propriété) et $\delta(u)$ non défini s'il n'y a pas de tel x.

3.3.5. *Exemple* (UTF-8). Le codage UTF-8 d'Unicode (voir `https://fr.wikipedia.org/wiki/UTF-8`[W]) associe à chaque point de code une suite de 1 à 6 octets, c'est-à-dire un mot dans $(\mathbf{B}^8)^*$, de longueur 1 à 6. Les mots sur 1 octet sont les codes ASCII sur 7 bits, précédés d'un bit 0 : de $\underline{0}^8$ à $\underline{01}^7$. Les autres commencent tous par un octet dont le premier bit est 1 : si cet octet commence par $\underline{110}$, il y a 2 octets ; si cet octet commence par $\underline{1110}$, il y a 3 octets ; et ainsi de suite. De plus, chacun des octets autres que le premier commence par $\underline{10}$.

3.3.6. Exercice (Codages préfixes)**.**

1. Vérifiez que si γ est un codage préfixe alors γ^* défini comme dans la section précédente est à nouveau un morphisme injectif.
2. Vérifiez qu'un codage de taille fixe est toujours un codage préfixe.
3. Écrivez une fonction `prefixe(u,v)` qui renvoie `True` si le mot `u` est préfixe de `v`, et `False` sinon.
4. Vérifiez que le codage UTF-8 est un codage préfixe, qui vérifie de plus la propriété suivante : aucun suffixe strict non vide d'un code n'est préfixe d'un code. ◇

11. `https://fr.wikipedia.org/wiki/Compression_de_données`[W]

4 Représenter les nombres

On a déjà vu l'écriture binaire des entiers naturels dans les sections précédentes. On la formalise et la généralise en toute base, avant de s'intéresser à la représentation des nombres en général.

4.1 Entiers naturels en base quelconque

4.1.1. Théorème (Existence et unicité de l'écriture en base B)**.** *Si on fixe $B > 1$, alors tout entier $n \in [\![B^k]\!]$ admet une unique écriture sous la forme*

$$n = \sum_{i=0}^{k-1} c_i B^i$$

avec $0 \leq c_i < B$ pour $0 \leq i < k$.

Démonstration. Si $k = 0$ le résultat est direct. Lorsqu'on a une telle écriture avec $k > 0$, on peut écrire :

$$n = \left(\sum_{i=0}^{k-2} c_{i+1} B^i \right) \times B + c_0$$

et donc c_0 est le reste de la division euclidienne de n par B, et $\sum_{i=0}^{k-2} c_{i+1} B^i$ est une écriture du quotient. L'existence et l'unicité de l'écriture se déduisent de celles du quotient et du reste par une récurrence évidente. □

4.1.2. Définition (Écriture en base B)**.** On dit que le mot $\gamma_{B,k}(n) = c_0 \cdots c_{k-1}$ est une *écriture en base B de n.*[12] La *taille* de n en base B, notée $|n|_B$, est le plus petit k tel que $n < B^k$ (en particulier $|n|_B = 0$ ssi $n = 0$), c'est-à-dire le plus petit k tel que n admette une écriture en base B de longueur k : on note alors $\gamma_B(n) = \gamma_{B,|n|_B}(n)$.

4.1.3. *Observation.* Si $|n|_B \leq k \leq l$ alors $\gamma_{B,l}(n) = \gamma_{B,k}(n) \cdot \underline{0}^{l-k}$.

4.1.4. Exercice (Écriture en base B en Python)**.**

1. Définissez une fonction `representation(n,B)` qui renvoie la liste des c_i, pour $0 \leq i < |n|_B$), et sa fonction réciproque `valeur(c,B)` qui calcule la somme étant donnée la liste `c` des coefficients et la base `B`.
2. Déduisez-en une fonction `conversion(c,B1,B2)` qui transforme une écriture de la base `B1` à la base `B2`.
3. Modifiez la fonction `representation()` en `representation(n,B,k=None)` : si l'argument optionnel `k` n'est pas fourni (`k is None`), le comportement est le même que précédemment ; s'il est fourni et que c'est un entier `k` $\geq |$`n`$|_B$, on renvoie l'écriture de `n` en base `B` de longueur `k` ; sinon une exception `ValueError` est soulevée. ◇

4.1.5. Définition (Ordre lexicographique, ordre lexicographique inverse)**.** Si l'ensemble A est muni d'une relation d'ordre $\leq$, on en déduit la relation d'*ordre lexicographique* $\leq_{\text{lex}}$ sur A^* de la

12. Contrairement à l'habitude petit-boutiste de l'écriture décimale ou binaire, on choisit ici une écriture gros-boutiste, plus conforme avec l'idée que les chiffres sont obtenus par divisions euclidiennes successives. Ça introduit une confusion de notations dans le cas $B = 2$: dans toute la suite, γ_2 est la fonction définie ici.

manière suivante : [13]

$$\begin{aligned} \varepsilon &\leq_{\text{lex}} u \qquad \text{pour tout } u \in A^* \\ a \cdot v &\leq_{\text{lex}} b \cdot v \qquad \text{si } a < b \\ a \cdot v &\leq_{\text{lex}} a \cdot v \qquad \text{si } u \leq_{\text{lex}} v. \end{aligned}$$

L'*ordre lexicographique inverse* $\leq_{\overline{\text{lex}}}$ est obtenu en posant $u \leq_{\overline{\text{lex}}} v$ ssi $\overline{u} \leq_{\text{lex}} \overline{v}$.

4.1.6. *Remarque.* C'est l'ordre du dictionnaire : $u \leq_{\text{lex}} v$ si la première lettre différente (lorsqu'elle existe) est plus petite, ou bien si u est un préfixe de v.

4.1.7. Exercice (Ordre sur les entiers et ordre lexicographique)**.**

1. Vérifiez que si $k \geq \max(|n|_B, |m|_B)$ alors $n \leq m$ ssi $\gamma_{B,k}(n) \leq_{\overline{\text{lex}}} \gamma_{B,k}(m)$.
2. Déduisez-en que $n \leq m$ ssi $|n|_B < |m|_B$, ou $|n|_B = |m|_B$ et $\gamma_B(n) \leq_{\overline{\text{lex}}} \gamma_B(n)$. ◇

Si la taille des entiers qu'on considère est fixée, disons k, les opérations peuvent produire des débordements. On peut choisir de travailler dans $\mathbf{Z}/B^k\mathbf{Z}$, mais on peut aussi conserver l'information des débordements. Ainsi, l'algorithme usuel de l'addition se décrit très simplement par récurrence : le débordement n'est rien d'autre que la retenue.

4.1.8. Définition (Addition en base B)**.** Si $u, v \in [\![B]\!]^k$ alors on définit $\texttt{add}(u, v) = (w, r)$ avec $w \in [\![B]\!]^k$ et $r \in \mathbf{B}$, par récurrence sur k. Si $k = 0$, $w = \varepsilon$ et $r = 0$; sinon :

— on écrit $u = u' \cdot b$ et $v = v' \cdot c$;

— on obtient $\texttt{add}(u', v') = (w', r')$ par hypothèse de récurrence ;

— si $b + c + r' < B$, on pose $w = w' \cdot (b + c + r')$ et $r = 0$; sinon $B \leq b + c + r' < 2B - 1$, et on pose $w = w' \cdot (b + c + r' - B)$ et $r = 1$.

4.1.9. Proposition. *Si $m, n \in [\![B^k]\!]$ et $(w, r) = \texttt{add}(\gamma_{B,k}(m), \gamma_{B,k}(n))$, alors $r = 1$ ssi $m + n \geq B^k$, $w = \gamma_{B,k}(m + n \mod B^k)$ et $w \cdot r = \gamma_{B,k+1}(m + n)$.*

4.1.10. Exercice (Opérations en base B)**.**

1. Définissez une fonction `addition_taille_fixe(n,m,k,B)` qui attend des listes `n` et `m` de chiffres dans $[\![\texttt{B}]\!]$, de longueur k, et qui utilise l'algorithme usuel de l'addition pour renvoyer un couple `p,r` où `p` est la liste des chiffres de `valeur(n,B)+valeur(m,B)` modulo $2^{\texttt{k}}$, et `r` est la retenue éventuelle (`1` s'il y a dépassement, `0` sinon).
2. Définissez une fonction `addition(n,m,B)` similaire à la précédente, mais sans contrainte de taille : on renvoie `representation(valeur(n,B)+valeur(m,B))`, toujours calculé par l'algorithme usuel de l'addition, et sans supposer que les deux écritures sont de même longueur. ◇
3. Si on admet que les comparaisons et les additions de chiffres sont effectuées en temps constant, estimez le temps nécessaire pour calculer les chiffres de l'addition de deux nombres, en fonction de leur taille.
4. (Optionnel.) De la même manière réalisez les opérations de soustraction (en soulevant une exception si le résultat n'est pas un entier naturel), de multiplication, de division euclidienne (en soulevant une exception en cas de division par zéro), et estimez leur complexité en temps.

13. Formellement, on définit la valeur de vérité de $u \leq_{\text{lex}} v$ par récurrence sur $|u| + |v|$.

4.2 Entiers relatifs

Tout entier $k \in \mathbf{Z}$ peut s'écrire sous la forme $(-1)^b|k|$ avec $b \in \mathbf{B} = \{0,1\}$ et $|k| \in \mathbf{N}$. La *valeur absolue* $|k|$ est définie de manière unique par cette propriété. [14] L'application

$$(b,n) \mapsto (-1)^b n$$

est donc une surjection $\mathbf{B} \times \mathbf{N} \to \mathbf{Z}$; mais ce n'est pas une injection, car $0 = -1 \times 0$. Autrement dit, ce codage n'est pas univoque : le signe de 0 n'est pas bien défini.

Par contre, on a bien une bijection :

$$\begin{array}{ccc} \mathbf{B} \times \mathbf{N} & \to & \mathbf{Z} \\ (b,n) & \mapsto & (-1)^b(n+b) \end{array}$$

de réciproque :

$$\begin{array}{ccc} \mathbf{Z} & \to & \mathbf{B} \times \mathbf{N} \\ k & \mapsto & (\sigma(n), |k| - \sigma(n)) \end{array}$$

où $\sigma(n) = 0$ si $n \geq 0$ et $\sigma(n) = 1$ si $n < 0$. Pour coder des entiers comme des mots sur B de longueur fixée k, il semble donc naturel de travailler dans $[\![-B^k{:}B^k]\!]$.

4.2.1. Théorème (Existence et unicité de l'écriture signée en base B)**.** *. Tout entier $n \in [\![-B^k{:}B^k]\!]$ admet une unique écriture sous la forme*

$$n = -\sigma(n)B^k + \sum_{i=0}^{k-1} c_i B^i$$

avec $0 \leq c_i < B$ pour $0 \leq i < k-1$.

Démonstration. Si $n \geq 0$, $\sigma(n) = 0$ et on applique le cas des entiers naturels (théorème 4.1.1). Sinon, $\sigma(n) = 1$ et $0 \leq n + B^k < B^k$: on applique à nouveau le théorème 4.1.1. □

4.2.2. Définition (Écriture signée en base B)**.** Si $n \in [\![-B^k{:}B^k]\!]$, l'*écriture signée de taille k en base B* est le couple $(\gamma_{B,k}(n), \sigma(n))$ où $\gamma_{B,k}(n) = c_0 \cdots c_{k-1}$ est obtenu par le théorème 4.2.1. La *taille* de n en base B, notée $|n|_B$, est le plus petit k tel que $n \in [\![-B^k{:}B^k]\!]$, On note alors $\gamma_B(n) = \gamma_{B,|n|_B}(n)$.

L'avantage de cette représentation est que l'algorithme de l'addition est essentiellement le même. Il faut par contre travailler un peu pour le calcul de l'opposé (et donc la soustraction) : changer le signe $\sigma(n)$ ne suffit pas.

4.2.3. Définition. Pour tout chiffre $c \in [\![B]\!]$, on note $\hat{c} = B - 1 - c \in [\![B]\!]$. Pour tout mot $u = c_0 \cdots c_{k-1} \in [\![B]\!]^*$, on note $\hat{u} = \widehat{c_0} \cdots \widehat{c_{k-1}} \in [\![B]\!]^*$. On dit que $\hat{u}$ est le *complément à $B-1$* de u.

4.2.4. *Observation.* On a $\texttt{add}(u, \hat{u}) = (\underline{(B-1)}^k, 0)$. De plus, dès que $k > 0$, $\texttt{add}(\underline{(B-1)}^k, \gamma_{B,k}(1)) = (\gamma_{B,k}(0), 1)$.

C'est-à-dire que si u est l'écriture de $n \in \mathbf{N}$, et si on ajoute (*via* `add`) $\hat{u}$ puis l'écriture de 1 à u, on obtient l'écriture de 0 (et la retenue 1). Et donc $\texttt{add}(\hat{u}, \gamma_{B,k}(1))$ est naturellement la représentation de l'opposé de n, sauf pour le bit de signe, qu'il faut inverser : c'est le complément à B. [15]

14. Attention à la possible confusion avec la notation pour la taille : la taille d'un entier est toujours paramétrée par la base B.

15. Le complément à $B-1$, plus 1, ce qui est une bonne mnémotechnique, sans être une appellation très correcte. Voir aussi : `https://fr.wikipedia.org/wiki/Complément_à_2`[W].

4.2.5. Exercice (Entiers relatifs en base B)**.**

1. Expliquez à quelle condition sur l'écriture des nombres relatifs n et m on a $n \leq m$.
2. Définissez une fonction `representationZ(n,B,k=None)` qui renvoie le couple `(c,s)` où `s` $= \sigma(n)$ et `c` est l'écriture de `n` en base `B` de taille `k` si `k` est fourni, ou de taille $|\texttt{n}|_{\texttt{B}}$ sinon. Définissez la fonction réciproque `valeur(c,s,B)`.
3. Définissez une fonction `oppose(c,s,B)` qui renvoie l'écriture de l'opposé de `valeur(c,s,B)` en utilisant le complément à B. ◇
4. (Optionnel.) De la même manière réalisez les opérations d'addition, soustraction, multiplication et division euclidienne, sur les représentations d'entiers relatifs formées d'une liste `c` de chiffres, et d'un bit signe `s`.

4.2.6. *Remarque.* Dans la représentation usuelle des nombres en machine, la taille étant fixée par l'architecture matérielle, le bit de signe est directement intégré au mot binaire, et remplace le bit de poids fort. Typiquement, pour l'écriture signée sur 32 bits, on a 1 bit pour le signe, et on représente les entiers dans $[\![-2^{31}{:}2^{31}]\!]$.

4.2.7. *En Python* 🐍*.* Dans la version standard de l'interpréteur Python,[16] les entiers (du type `int`) sont codés comme des couples (s, u) où $s \in \mathbf{B}^2$ indique le signe ($\underline{00}$ si strictement positif, $\underline{10}$ si nul, $\underline{01}$ si strictement négatif), et u est un codage de la valeur absolue en base 2^{30} ou 2^{15} suivant les architectures, de taille variable, ajustée en fonction de la taille de l'entier.[17] Au contraire des entiers en taille fixe, les opérations sont plus naturelles à réaliser avec ce type de codage, plutôt qu'un codage en complément à B, lorsque la taille est variable.

4.3 Rationnels

4.3.1. *Observation.* L'ensemble $\mathbf{Q}$ des rationnels est le quotient de $\mathbf{Z} \times \mathbf{Z}^\bullet$ par la relation d'équivalence définie par $(d, n) \sim (d', n')$ ssi $dn' = d'n$: $\frac{d}{n}$ est la classe d'équivalence de (d, n).

Pour représenter les rationnels, il est donc suffisant de représenter les couples d'entiers. C'est trivial pour des entiers en taille fixée : il suffit de prendre la concaténation des codes. Mais si la taille des codes n'est pas fixée, il faut travailler un peu plus : plus généralement, il s'agit de représenter les couples de mots (voire les suites finies de mots) comme des mots. Une méthode standard est d'utiliser un séparateur :

4.3.2. *Observation.* Pour tout alphabet A et tout symbole $S \notin A$, la fonction :

$$(u_0, \dots, u_{k-1}) \mapsto u_0 \cdot S \cdot \cdots \cdot u_{k-1} \cdot S$$

(la concaténation des u_i chacun suivi de S) définit un morphisme injectif $(A^*)^* \to (A \cup \{S\})^*$.[18]

Si on ne veut pas introduire de nouveau caractère, il suffit de composer avec un codage $A \cup \{S\} \to A^*$.

4.3.3. Exercice (Ajout d'un séparateur à l'alphabet)**.** Montrez qu'il existe toujours un codage préfixe $A \cup \{S\} \to A^*$, n'utilisant que des mots de longueur au plus 2. ◇

Le codage $\gamma : \mathbf{Q} \to \mathbf{Z} \times \mathbf{Z}$ consiste à mettre la fraction $\frac{d}{n}$ en forme canonique $\frac{d'}{n'}$ avec d' et n' premiers entre eux, et $n' \in \mathbf{N}^\bullet$. Le décodage $\delta : (d, n) \mapsto \frac{d}{n}$ n'est évidemment pas injectif. Le codage des opérations peut se faire en utilisant les formules habituelles (celles apprises

16. CPython `https://github.com/python/cpython/`, codé en C.
17. Les détails sont expliqués dans le code source : `https://github.com/python/cpython/blob/3.13/Include/cpython/longintrepr.h#L64`.
18. Ici on considère des mots sur A^* qui est infini dénombrable.

au collège), mais doit systématiquement effectuer la mise sous forme canonique. Il est donc important d'utiliser un algorithme efficace pour le calcul du PGCD : le PGCD binaire[19] ou, mieux, l'algorithme de Lehmer, plus efficace pour les grands nombres[20] (c'est celui utilisé par CPython pour `math.gcd`[21]).

4.3.4. *En Python* . Il n'y a pas de représentation standard des rationnels en machine, mais Python fournit une classe `Fraction` dans le module `fractions`.

4.4 Représentation en virgule flottante

Comme on l'a établi dans l'exercice 3.1.8, l'ensemble des mots binaires est dénombrable : il est donc évident qu'on ne peut pas coder les nombres réels comme des mots finis, $\mathbf{R}$ étant indénombrable.

On en est donc réduits à travailler avec des valeurs approchées. Le principe est similaire à celui de la notation scientifique (un nombre décimal fois une puissance de 10), et généralisable en toute base : c'est la représentation en virgule flottante.

La base B étant fixée, on associe à tout couple (n, k) de relatifs le nombre $n \times B^k$: on appelle n la *mantisse* et k l'*exposant* dans cette représentation,. On ne représente donc que des rationnels particuliers, ceux dont le dénominateur est une puissance de B (les *décimaux* en base 10, les *rationnels dyadiques* en base 2). À nouveau, la représentation n'est pas canonique : (Bn, k) et $(B, k+1)$ sont deux représentations de $n \times B^{k+1}$.

4.4.1. *Observation.* S'il n'y a pas de contrainte sur les tailles de n et k, on obtient une représentation canonique de $n \times B^k$, quand $(n, k) = (0, 0)$ ou n n'est pas multiple de B.

4.4.2. Exercice. En binaire, si la mantisse est représentée sur m bits et l'exposant sur e bits, dont 1 bit de signe pour chacun, en utilisant le complément à 2 pour les négatifs, on a $n \in [\![-2^{m-1}{:}2^{m-1}]\!]$ et $k \in [\![-2^{e-1}{:}2^{e-1}]\!]$.

1. Quelles sont les plus petite et plus grande valeurs représentables ?
2. Quelle est la plus petite valeur absolue non nulle représentable ?
3. Quel est le plus grand $a \in \mathbf{N}$ tel que tous les éléments de $[\![-a{:}a]\!]$ soient représentables ?
4. À quelles conditions sur n et k peut-on trouver une représentation canonique équivalente (n', k'), qui respecte les contraintes de taille sur la mantisse et l'exposant ? ◇

Les opérations d'addition, soustraction et multiplication peuvent produire des dépassements. Pire, la division peut produire des résultats non représentables : typiquement si n n'est pas une puissance de 2, $1/n$ n'est pas un rationnel dyadique. Plutôt que de produire des erreurs, la convention est plutôt de chercher la « meilleure approximation » du résultat souhaité : ce comportement est subtil et propice aux erreurs, et la norme IEEE-754[22] qui le standardise est assez complexe. On ne rentrera pas dans les détails, mais cela explique pourquoi, par exemple :

- `1-1/3 != 1/3+1/3`
- `(2.0**1023 + 1)-2.0**1023 == 0.0`.

4.4.3. *En Python* . Contrairement aux entiers, la représentation par défaut de nombres en virgule flottante de Python est exactement celle de l'architecture sous-jacente : c'est le type `float`, avec les comportements bizarres qu'on vient juste d'exposer. Il est toutefois possible de

19. https://fr.wikipedia.org/wiki/algorithme_du_PGCD_binaire[W]
20. https://en.wikipedia.org/wiki/Lehmer's_GCD_algorithm
21. https://github.com/python/cpython/blob/3.13/Objects/longobject.c#L5630
22. https://fr.wikipedia.org/wiki/IEEE-754[W]

travailler avec une représentation décimale en précision arbitraire (mais toujours fixée) grâce au module `decimal`. [23]

23. https://docs.python.org/fr/3/library/decimal.html

Solutions des exercices

Solution de l'exercice 1.1.1.

1. L'ensemble $\mathbf{B}^2$ a $2^2 = 4$ éléments, donc l'ensemble des fonctions $\mathbf{B}^2 \to \mathbf{B}$ a $2^4 = 16$ éléments.

2. On peut déjà noter que, pour tous $x, y \in \mathbf{B}$, $0 = x \wedge \neg x$, $1 = x \vee \neg x$, $x \Rightarrow y = \neg x \vee y$, $x \oplus y = (x \wedge \neg y) \vee (\neg x \wedge y)$ et $x \Leftrightarrow y = (\neg(x \vee y)) \vee (x \wedge y)$. On peut donc utiliser les opérations binaires constantes 0 et 1, et les opérations binaires $(x, y) \mapsto x \Rightarrow y$, $\oplus$ et $\Leftrightarrow$, en plus de x, y, $\wedge$, $\vee$ et $\neg$. On donne des expressions pour chacune des tables possibles :

x	y	0	$x \wedge y$	$x \wedge \neg y$	x	$\neg x \wedge y$	y	$x \oplus y$	$x \vee y$
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

x	y	$\neg(x \vee y)$	$x \Leftrightarrow y$	$\neg y$	$y \Rightarrow x$	$\neg x$	$x \Rightarrow y$	$\neg(x \wedge y)$	1
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

3. Il suffit de regarder dans la table : ce sont celles qui ont les mêmes valeurs pour $(x, y) = (0, 1)$ et $(x, y) = (1, 0)$. Il y en a 8 : les constantes 0 et 1, les opérations $\wedge$, $\vee$, $\Leftrightarrow$, $\oplus$, et les négations $\neg(x \vee y)$ et $\neg(x \wedge y)$.

4. Les fonctions constantes sont évidemment associatives, et les opérations $\wedge$, $\vee$ et $\oplus$ sont aussi bien connues pour l'être (c'est le minimum, le maximum et la somme modulo 2). Pour $\Leftrightarrow$, on vérifie que $x \Leftrightarrow y = x + y + 1 \mod 2$, qui est à nouveau associative. (Une autre méthode serait de calculer les tables de $x \wedge (y \wedge z)$ et $(x \wedge y) \wedge z$, $x \vee (y \vee z)$ et $(x \vee y) \vee z$, *etc.*, pour vérifier que ce sont chaque fois les mêmes.)

   Il suffit donc de voir que les négations $\neg(x \vee y)$ et $\neg(x \wedge y)$ ne le sont pas : $\neg(\neg(1 \vee 0) \vee 0) = \neg(0 \vee 0) = 1$ mais $\neg(1 \vee \neg(0 \vee 0)) = \neg 1 = 0$; de même $\neg(\neg(0 \wedge 1) \wedge 1) = \neg(1 \wedge 1) = 0$ mais $\neg(0 \wedge \neg(1 \wedge 1)) = \neg 0 = 1$.

   De plus $x \wedge 1 = x \Leftrightarrow 1 = x$ et $x \vee 0 = x \oplus 0 = x$, par simple application des tables de vérité.

Solution de l'exercice 1.2.6. On reprend les notations de la proposition 1.2.2, et on note $p(u) \subseteq X$ la partie de X associée au mot $u \in \mathbf{B}^n$. Soient également $v \in \mathbf{B}^n$ et $x \in X$. Par définition, on a :

$$\begin{aligned} x \in p(\neg u) \quad &\text{ssi} \quad x \notin p(u) \\ x \in p(u \wedge v) \quad &\text{ssi} \quad x \in p(u) \text{ et } x \in p(v) \\ x \in p(u \vee v) \quad &\text{ssi} \quad x \in p(u) \text{ ou } x \in p(v) \end{aligned}$$

donc $p(\neg u) = X \setminus p(u)$, $p(u \wedge v) = p(u) \cap p(v)$, et $p(u \vee v) = p(u) \cup p(v)$.

En remarquant (comme dans la question 2 de l'exercice 1.1.1) que les autres opérations s'écrivent avec $\neg$, $\wedge$ et $\vee$, on obtient que : $p(u \Rightarrow v) = \big(X \setminus p(u)\big) \cup p(v)$, et $p(u \Leftrightarrow v) = \big(X \setminus (p(u) \cup p(v))\big) \cup \big(p(u) \cap p(v)\big)$, et $p(u \oplus v) = \big(p(u) \setminus p(v)\big) \cup \big(p(v) \setminus p(u)\big)$ (c'est la différence symétrique).

Solution de l'exercice 1.3.1.

1. Comme il y a 2^n mots binaires de longueur n il y a 2^{2^n} opérations booléennes n-aires (on retrouve $2^4 = 16$ dans le cas $n = 2$ des connecteurs binaires). De même, il y a $(2^k)^{2^n} = 2^{k2^n}$ fonctions $\mathbf{B}^n \to \mathbf{B}^k$. C'est beaucoup !
2. On raisonne par récurrence sur n. Si $n = 0$, les seules opérations sont 0 et 1. Supposons le résultat établi pour n et considérons une opération $f : \mathbf{B}^{n+1} \to \mathbf{B}$. On définit deux opérations n-aires par : $f_0 : b_1 \cdots b_n \mapsto f(b_1 \cdots b_n 0)$ et $f_1 : b_1 \cdots b_n \mapsto f(b_1 \cdots b_n 1)$. On a alors $f(b_1 \cdots b_n b_{n+1}) = ((\neg b_{n+1}) \wedge f_0(b_1 \cdots b_n)) \vee (b_{n+1} \wedge f_1(b_1 \cdots b_n))$: en effet, quand $b_{n+1} = 0$, $((\neg b_{n+1}) \wedge f_0(b_1 \cdots b_n)) \vee (b_{n+1} \wedge f_1(b_1 \cdots b_n)) = (1 \wedge f_0(b_1 \cdots b_n)) \vee (0 \wedge f_1(b_1 \cdots b_n)) = f_0(b_1 \cdots b_n) \vee 0 = f_0(b_1 \cdots b_n)$; et quand $b_{n+1} = 1$, $((\neg b_{n+1}) \wedge f_0(b_1 \cdots b_n)) \vee (b_{n+1} \wedge f_1(b_1 \cdots b_n)) = (0 \wedge f_0(b_1 \cdots b_n)) \vee (1 \wedge f_1(b_1 \cdots b_n)) = 0 \vee f_1(b_1 \cdots b_n) = f_1(b_1 \cdots b_n)$. On conclut en appliquant l'hypothèse de récurrence.

Solution de l'exercice 1.3.2. *L'énoncé original ne considérait que le cas de $n = 2^p$, ce qui était inutilement compliqué, et comportait une première question qui n'avait pas beaucoup d'intérêt (ça avait du sens pour une version intermédiaire supprimée depuis, désolé pour ça).* La preuve est par récurrence forte sur k. Soit e une fonction booléenne construite avec k connecteurs binaires : si $k = 0$, il est évident que e ne peut dépendre que d'un seul de ses arguments ; si $e = e_1 * e_2$ avec $*$ un connecteur binaire, et si e_1 utilise k_1 connecteurs binaires et e_2 en utilise k_2, alors $k = k_1 + k_2 + 1$. Par hypothèse de récurrence, e_1 ne dépend que d'au plus $k_1 + 1$ arguments, et e_2 d'au plus $k_2 + 1$, donc e dépend d'au plus $k_1 + k_2 + 2 = k + 1$ arguments.

Solution de l'exercice 2.0.2.

1. Il y a un codage sur n bits ssi il existe une injection $\gamma : X \to \mathbf{B}^n$, c'est-à-dire si X a au plus 2^n éléments.
2. Il suffit de fixer un élément arbitraire $x \in X$ et de poser $\delta(u) = x$ dans tous les cas où δ n'était pas définie.
3. Si δ est totale et injective, comme elle est aussi surjective, c'est une bijection : donc X a exactement 2^n éléments. Réciproquement, si X a 2^n éléments, on obtient une bijection entre X et $\mathbf{B}^n$, qui définit un codage formé de deux bijections inverse l'une de l'autre.

Solution de l'exercice 2.1.3. L'explication est donnée après l'exercice. Une autre manière de se convaincre est de vérifier que la fonction calcule bien dans les 16 cas :

$0000 \mapsto 00$	$0001 \mapsto 01$	$0010 \mapsto 10$	$0011 \mapsto 11$
$0100 \mapsto 01$	$0101 \mapsto 10$	$0110 \mapsto 11$	$0111 \mapsto 00$
$1000 \mapsto 10$	$1001 \mapsto 11$	$1010 \mapsto 00$	$1011 \mapsto 01$
$1100 \mapsto 11$	$1101 \mapsto 00$	$1110 \mapsto 01$	$1111 \mapsto 10$

Solution de l'exercice 2.2.5.

```
def cesar1(c,k):
    """Retourne le caractère associé à `c` par le chiffre de César de clé `k`.
    La chaîne `c` doit être réduite à un caractère et `k` doit être un entier.
    Si `c` est une lettre de l'alphabet (de 'a' à 'z', ou de 'A' à 'Z')
    le résultat est obtenu en renvoyant la `k`-ième lettre après `c`,
    quitte à revenir au début de l'alphabet,
    en conservant la casse (majuscule ou minuscule).
    Dans les autres cas, on renvoie `c`. """
```

```
    if 'a' <= c <= 'z':
        a = 'a'
    elif 'A' <= c <= 'Z':
        a = 'A'
    else:
        return c
    ord_a = ord(a) # code de `a` ou `A`
    rang_c = ord(c) - ord_a # place de `c` dans l'alphabet (de 0 à 25)
    rang = (rang_c + k) % 26 # place du code dans l'alphabet (de 0 à 25)
    return chr(rang + ord_a)
```

Solution de l'exercice 3.1.8.

1. Si $a \neq b \in A$, en posant $u = a$ et $v = b$ deux mots réduits à une lettre, on a $u \cdot v = ab \neq ba = v \cdot u$.
2. On a évidemment $|u \cdot v| = |u| + |v|$ et $|\varepsilon| = 0$. Si $a \neq b \in A$, $|ab| = 2 = |aa|$ donc le morphisme n'est pas injectif. Si $A = \emptyset$, il n'y a pas de mot u tel que $|u| \neq 0$, donc le morphisme n'est pas surjectif. Si $A = \{a\}$ est un singleton, on a bien un isomorphisme : $a^n \mapsto n$.
3. On a $A^* = \bigcup_{n\in\mathbb{N}} A^n$. Si A est dénombrable, A^* est l'union d'une famille dénombrable d'ensembles dénombrables. Si A est indénombrable, comme on a une injection $A \to A^*$, A^* est indénombrable.

Solution de l'exercice 3.1.11.

1. On propose plusieurs versions possibles, dans des styles différents. On peut tout faire à la main :

```
def gonfle(s,k):
    retour = ""
    for c in s:
        for i in range(k):
            retour = retour+c
    return retour
```

ou bien se souvenir que `k*c` retourne la concaténation de `k` copies de `c` :

```
def gonfle(s,k):
    retour = ""
    for c in s:
        retour = retour + k*c
    return retour
```

ou encore on peut utiliser un peu de magie Python, si on la connait :

```
def gonfle(s,k):
    return "".join((k*c for c in s))
```

(on aura l'occasion de revenir sur la définition en compréhension et la méthode `join()` plus tard).

2. Là encore, plusieurs styles sont possibles. Tout à la main :

```
def palindrome(u):
    l = len(u)
    for i in range(l//2):
```

```
        if u[i] != u[l-i-1]:
            return False
    return True
```

ou bien en utilisant la fonction `reversed` (la conversion en `tuple` des deux côtés de l'égalité est nécessaire pour obtenir des séquences du même type) :

```
def palindrome(u):
    return tuple(reversed(u)) == tuple(u)
```

mais c'est moins efficace parce qu'on fait deux fois plus de tests (on parcourt toute la liste) ; ou bien avec un usage astucieux des tranches [24] :

```
def palindrome(u):
    l = len(u)
    return u[:l//2] == u[:-1-l//2:-1]
```

ou encore comme dans la première version, mais en une ligne avec de la magie :

```
def palindrome(u):
    return all(u[i] == u[-i-1] for i in range(len(u)))
```

Solution de l'exercice 3.3.6.

1. Le fait que la fonction γ^* est l'unique morphisme de monoïdes $X^* \to A^*$ qui coïncide avec γ sur $X = X^1$ est à nouveau une conséquence directe des définitions. Il n'y a qu'à vérifier son injectivité. Remarquons d'abord que $\gamma^*(u) = \varepsilon$ ssi $u = \varepsilon$. On montre que $\gamma^*(u) = \gamma^*(v)$ implique $u = v$ par récurrence forte sur $|u|$.
   Si $|u| = 0$ alors $u = \varepsilon$ donc $\gamma^*(u) = \gamma^*(v) = \varepsilon$ donc $v = \varepsilon = u$. Sinon, $\gamma^*(u) = \gamma^*(v)$ est non vide, et donc les deux mots sont non vides, et on pose $u = a \cdot u'$ et $v = b \cdot v'$. En appliquant γ^*, on obtient $\gamma(a) \cdot \gamma^*(u') = \gamma(b) \cdot \gamma^*(v')$. Si $|\gamma(a)| \leq |\gamma(b)|$, $\gamma(a)$ est un préfixe de $\gamma(b)$ et donc $a = b$ puisque γ est un codage préfixe ; on montre de même que $a = b$ quand $|\gamma(a)| > |\gamma(b)|$. On a donc $a = b$ et donc $\gamma(a) = \gamma(b)$ dans tous les cas : nécessairement, on a aussi $\gamma^*(u') = \gamma^*(v')$. Comme $\gamma(a) \neq \varepsilon$, $|u'| < |u|$, et l'hypothèse de récurrence donne $u' = v'$. Finalement, $u = a \cdot u' = b \cdot v' = v$.
2. Il suffit de remarquer que si $|u| = |v|$ alors u est un préfixe de v ssi $u = v$. Donc pour un codage de longueur fixe, $\gamma(a)$ est un préfixe de $\gamma(b)$ ssi $\gamma(a) = \gamma(b)$, c'est-à-dire ssi $a = b$ vu que γ est injective.
3. Encore une fois, plusieurs styles possibles :

```
def prefixe(u,v):
    l = len(u)
    if l>len(v):
        return False
    for i in range(l):
        if u[i] != v[i]:
            return False
    return True
```

```
def prefixe(u,v):
    return len(u) <= len(v) and u == v[:len(u)]
```

24. Voir `https://docs.python.org/fr/3.13/glossary.html#term-slice` et `https://docs.python.org/fr/3.13/library/stdtypes.html#common-sequence-operations` (en particulier la partie sur les « tranches de `s` de `i` à `j` avec un pas de `k` »).

```python
def prefixe(u,v):
    return len(u) <= len(v) and all(u[i] == v[i] for i in range(len(u)))
```

Notez que suivant les versions, on accepte que des mots de types différents soient préfixe l'un de l'autre ou non : avec la première et la dernière version, `prefixe("a",["a","b"]) == True` mais pas avec la deuxième (parce que pour Python une chaîne n'est jamais égale à une liste).

4. Si un code $\gamma_{\texttt{UTF-8}}(a)$ est un préfixe d'un autre $\gamma_{\texttt{UTF-8}}(b)$, ils ont le même premier octet, donc la même longueur (1 si ce premier octet commence par $\underline{0}$, $k>1$ si ce premier octet commence par $\underline{1^k0}$), donc ils sont égaux, et donc $a=b$ puisqu'évidemment $\gamma_{\texttt{UTF-8}}$ est injective. De plus, un suffixe strict non vide d'un code débute nécessairement par un octet commençant par $\underline{10}$, or le premier octet d'un code commence nécessairement par $\underline{0}$ ou par $\underline{11}$.

Solution de l'exercice 4.1.4. Le fichier source `baseB.py` est déposé sur Amétice.

Solution de l'exercice 4.1.7.

1. On montre l'équivalence par récurrence sur k. Si $k=0$ alors $m=n=0$ et l'équivalence est triviale. Supposons le résultat établi au rang k. Notons $\gamma_{B,k+1}(n)=c_0\cdots c_k$ et $\gamma_{B,k+1}(m)=b_0\cdots b_k$. On rappelle que $\sum_{i=0}^{k-1}c_iB^i\leq\sum_{i=0}^{k-1}B^i=\frac{B^k-1}{B-1}<B^k$. Si $c_k<b_k$ alors

$$n=\sum_{i=0}^{k}c_iB^i<B^k+c_kB^k\leq b_kB^k\leq m$$

par le rappel précédent. Si $c_k>b_k$ on obtient $n>m$ de la même manière. Reste le cas où $c_k=b_k$: on pose $n'=n-c_kB^k$ et $m'=m-c_kB^k$ de sorte que $\gamma_{B,k}(n')=c_0\cdots c_{k-1}$ et $\gamma_{B,k}(m')=b_0\cdots b_{k-1}$. En appliquant l'hypothèse de récurrence, on obtient : $n\leq m$ ssi $n'\leq m'$ ssi $\gamma_{B,k}(n')\leq_{\overline{\text{lex}}}\gamma_{B,k}(m')$ ssi $\gamma_{B,k+1}(n)\leq_{\overline{\text{lex}}}\gamma_{B,k+1}(m)$.

2. Si $|n|_B<|m|_B$ alors en posant $k=|m|_B$ et $\gamma_{B,k}(m)=b_0\cdots b_{k-1}$ on a $k>0$ et $b_{k-1}>0$ et donc

$$n<B^{|n|_B}\leq b_{k-1}B^{k-1}\leq m\,.$$

Si $|n|_B>|m|_B$ on a de même $n>m$. Enfin si $|n|_B=|m|_B$ a on a $n\leq m$ ssi $\gamma_B(n)\leq_{\overline{\text{lex}}}\gamma_B(n)$ par la question précédente.

Solution de l'exercice 4.1.10. Le code Python des fonctions demandées (sauf la partie optionnelle) se trouve à la suite du fichier `baseB.py` déposé sur Amétice. Pour l'addition en taille fixe, on a une simple boucle donc le temps nécessaire est de l'ordre de `k` fois une constante. De même pour l'addition en général : le nombre d'itérations est égal à la longueur de la plus longue écriture.

Solution de l'exercice 4.2.5.

1. Si $\sigma(n)=1$ et $\sigma(m)=0$, on a $n<0\leq m$. Si $\sigma(n)=\sigma(m)$ alors $n\leq m$ ssi $\gamma_{B,k}(n)\leq_{\overline{\text{lex}}}\gamma_{B,k}(m)$ par l'exercice 4.1.7.

2. Pour les deux questions suivantes, le code se trouve à la suite du fichier `baseB.py` déposé sur Amétice.

Solution de l'exercice 4.3.3. Soient $a \neq b \in A$ (on rappelle qu'en général on suppose que A est fini et a au moins deux éléments). On peut alors poser par exemple :

$$\begin{aligned}\gamma : A \cup \{S\} &\to A^* \\ a &\mapsto aa \\ S &\mapsto ab \\ c &\mapsto c \quad \text{quand } c \notin \{a, S\}.\end{aligned}$$

Solution de l'exercice 4.4.2.

1. La plus grande valeur (positive) est $(2^{m-1}-1) \times 2^{2^{e-1}-1}$, et la plus petite valeur (négative) est $-2^{m-1} \times 2^{2^{e-1}-1} = -2^{m-2+2^{e-1}}$.
2. L'énoncé original ne demandait pas une valeur absolue non nulle, ce qui rend la question triviale. La plus petite valeur absolue non nulle pour la mantisse est 1. La plus petite valeur absolue pour 2^k est obtenue pour la plus petite valeur de k possible. On obtient $2^{-2^{e-1}}$ comme plus petite valeur absolue non nulle représentable.
3. Tous les entiers de $[\![-2^{m-1}{:}2^{m-1}]\!]$ sont représentables avec $k = 0$. Le nombre 2^{m-1} est également représentable avec $n = 2^{m-2}$ et $k = 1$ (il faut supposer $e > 0$ évidemment). Par contre, ni $2^{m-1} + 1$ ni son opposé ne sont représentables : avec $k > 0$, on ne représente que des entiers pairs ; et avec $k \leq 0$, on ne représente que des nombres rationnels q avec $-2^{m-1} \leq q < 2^{m-1}$. On a donc $a = 2^{m-1}$.
4. *L'observation sur les formes canoniques était trompeuse (être premier avec B^k n'a pas beaucoup de sens si $k < 0$; et surtout c'est plutôt une question de divisibilité par B), ce qui n'aidait pas à la compréhension de cet exercice… C'est corrigé dans cette nouvelle version.*

   Si $n = 0$, on obtient une écriture canonique $(0, 0)$. Sinon, on peut écrire de manière unique $n = n' \times 2^l$ avec $l \in \mathbf{N}$ et n' impair (donc non multiple 2). La représentation canonique associée est alors (n', k') avec $k' = k + l$. On a automatiquement $n' = n/2^l \in [\![-2^{m-1}{:}2^{m-1}]\!]$ et $k' \geq k \geq -2^{e-1}$. La seule contrainte restante est $k' < 2^{e-1}$ et donc $l < 2^{e-1} - k$. La condition à vérifier est donc : $n = 0$ ou l'exposant l de 2 dans n vérifie $l < 2^{e-1} - k$ (par exemple si $k = 2^{e-1} - 1$ ce n'est possible que si n est nul ou impair).