Chapitre 1 : Numérotations

Le premier principe sur lequel repose la théorie de la calculabilité est le suivant :

Tout objet informatique est représentable (codable)
par un entier naturel.

En effet tout objet informatique, que ce soit une donnée (nombre, texte, image, et.) ou un programme, est une suite *finie* d'octets, qui peut être interprétée comme la représentation binaire d'un entier naturel.

Par conséquent la théorie de la calculabilité ne s'occupe, par convention, que de *calculs sur les entiers naturels* ; c'est une simple convention simplificatrice, qui n'implique aucune restriction, et ne signifie absolument pas que l'objet de cette théorie soit ce qu'on appelle la *théorie des nombres* (ou arithmétique) en mathématiques, qui traite en particulier des nombres premiers.

Cette correspondance entre objets informatiques et entiers souffre de deux bugs : ajouter des zéros en tête d'une représentation binaire ne change pas l'entier correspondant, et deux objets de *structure* différente peuvent avoir le même codage. Ces bugs sont mineurs, et facilement corrigés en pratique, en ajoutant au codage d'un objet, des informations sur sa structure.

Pour ce cours, et pour d'autres situations en informatique théorique, il est utile de connaître et d'étudier en détail quelques algorithmes simples de numérotation (autrement dit de codage par un entier) d'objets simples. Ces numérotations sont des *bijections* : deux objets distincts ont des numéros distincts (la numérotation est *injective*), et tout entier naturel est le numéro d'un objet (la numérotation est *surjective*). C'est l'objet de ce chapitre.

1. Entiers relatifs

A première vue il y a plus d'entiers relatifs que d'entiers naturels, et effectivement si l'on essaie de les numéroter en allant toujours dans la même direction, c'est un échec ! Il existe pourtant une méthode simple (et une infinité de variantes) pour numéroter les entiers relatifs :

x	0	1	-1	2	-2	3	-3	etc.
n (numéro)	0	1	2	3	4	5	6	etc.

Exercice. Ecrire les formules permettant de passer de x à n (fonction de numérotation), et inversement (comment retrouver l'entier relatif à partir de son numéro).

2. Couples d'entiers

Les remarques du paragraphe précédent s'appliquent a fortiori aux couples (x, y) d'entiers naturels : il semble qu'ils sont infiniment plus nombreux que les entiers naturels ! Il existe pourtant une méthode simple (et une infinité de variantes) pour numéroter les couples d'entiers naturels, en ordonnant les couples (x, y) suivant leur somme $x + y$:

(x, y)	(0, 0)	(1, 0)	(0, 1)	(2, 0)	(1, 1)	(0, 2)	(3, 0)	(2, 1)	etc.
x + y	0	1	1	2	2	2	3	3	etc.
n (numéro)	0	1	2	3	4	5	6	7	etc.

Exercice. Ecrire la formule permettant de passer de (*x*, *y*) à *n* (fonction de numérotation) ; conseil : commencer par traiter le cas des couples situés sur l'axe des abcisses. Pour retrouver inversement le couple à partir de son numéro, il n'existe pas de formule arithmétique utilisant simplement les quatre opérations (addition, soustraction, multiplication, division), un petit algorithme (qu'on écrira) est nécessaire.

Exercice. Etudier des méthodes de numérotation des triplets (*x*, *y*, *z*) d'entiers.

3. Mots

Les *mots* sont des suites finies de symboles, choisis dans un alphabet Σ de taille finie *t*. Pour les numéroter, le plus simple est de les ordonner selon leur longueur, et de classer alphabétiquement ceux de même longueur ; exemple avec un alphabet Σ = {a, b, c}, de taille *t* = 3 :

Mot *w*	ε (mot vide)	a	b	c	aa	ab	ac	ba	…	cc	aaa	aab	etc.
n (numéro)	0	1	2	3	4	5	6	7	…	12	13	14	etc.

Exercice. Vérifier que cette numérotation revient à écrire *n* en base *t* (taille de l'alphabet, ici 3), en utilisant les chiffres 1 (ici a) à *t* (ici c), au lieu des chiffres habituels 0 à *t* - *1* ; ce système de numération "sans zéro" fonctionne (sauf pour le nombre zéro, bien sûr), mais les opérations y sont beaucoup moins agréables (essayer quelques additions)…

4. Listes d'entiers

On a vu comment numéroter les couples d'entiers, puis (en exercice) les triplets, etc. Numéroter les listes (finies, mais de longueur variable) d'entiers, est un peu plus délicat ; en particulier il existe une infinité de listes dont la somme des éléments est nulle : [], [0], [0, 0], [0, 0, 0], etc. La clef du succès est d'ajouter à cette somme la taille (longueur) de la liste, soit :

$$\sigma\left([u_1, u_2, \ldots, u_t]\right) = t + u_1 + u_2 + \ldots + u_t$$

et d'ordonner les listes suivant ces sommes :

Liste	[]	[0]	[1]	[0, 0]	[2]	[1, 0]	[0, 1]	[0, 0, 0]	[3]	[2, 0]	[1, 1]	[0, 2]	[1, 0, 0]	…	[0, 0, 0, 0]	etc.
σ	0	1	2	2	3	3	3	3	4	4	4	4	4	…	4	etc.
n (numéro)	0	1	2	3	4	5	6	7	8	9	10	11	12	…	15	etc.

Exercice. On constate sur cette table que le nombre de listes avec σ constant vaut $2^{\sigma-1}$; donner une interprétation combinatoire de ce fait, et en déduire une numérotation (voisine de celle donnée ci-dessus) basée sur une correspondance directe entre une liste et la représentation binaire de son numéro. Conseil : utiliser une notation "unaire" (un entier est représenté par des "bâtonnets") pour les listes ; par exemple la liste [1, 3, 2] peut être représentée par `|,|||,||` .

Exercice. Il y a deux différences entre un ensemble et une liste : dans un ensemble, il n'y a pas d'élément répété, et l'ordre des éléments est sans importance. Ecrire les procédures permettant de numéroter les *ensembles finis* d'entiers, et inversement de retrouver un ensemble à partir de son numéro.

Exercice. Une idée totalement différente, pour numéroter les listes, et due à Gödel, utilise les nombres

premiers et attribue à la liste $[u_1, u_2, \ldots, u_t]$ le numéro

$$2^{u_1} \cdot 3^{u_2} \cdot 5^{u_3} \ldots p_t^{u_t}$$

où p_t désigne le $t^{ème}$ nombre premier. Utiliser cette formule pour numéroter les listes de la table ci-dessus ; proposer une correction pour le (léger) bug qui apparaît.

5. Procédures

Quel que soit le modèle de calcul envisagé, une *procédure* est un *texte*, qui a la caractéristique surprenante de pouvoir être *exécuté* ! D'autres noms possibles pour ce concept de base en informatique, sont :

- *Programme* : nous éviterons ce terme, car l'exécution d'un programme implique presque toujours une communication de ce programme avec "l'extérieur", et les problèmes d'*entrées/sorties* (input/output) concomitants ont traumatisé de nombreux programmeurs débutants, mais n'ont rien à voir avec la théorie de la calculabilité, bien que de nombreux textes à ce sujet ne semblent pouvoir s'en passer (conséquence probable du traumatisme subi par l'auteur dans son enfance, voir ce qui précède).

- *Algorithme* : ce terme est un peu imprécis.

- *Fonction* (langage C) : ce terme prête à confusion dans le contexte qui est le nôtre, voir ci-dessous.

Beaucoup de cours de calculabilité prennent comme premier modèle de calcul la *machine de Turing*, ce qui est à notre avis trompeur, en suggérant à tort que les deux sujets sont indissociables. En fait il n'y a aucune raison de ne pas utiliser un modèle de calcul familier, et dans le contexte de ce master d'informatique, nous utiliserons le langage C, connu de tous. Une procédure *p* sera donc décrite sous la forme :

```
int p (int x) {
  … ;
  return … ;
}
```

(nous utiliserons aussi des fonctions de plusieurs arguments lorsque cela sera commode, bien qu'une liste d'entiers puisse être codée par un seul entier, comme on l'a vu dans ce chapitre).

Ce choix pourra surprendre, car il a deux défauts évidents :

- Le type `int` désigne en C des entiers de taille fixe ; dans tout ce cours il faut comprendre que ce sont des entiers sans limitation de taille.

- Plus grave, les procédures (appelées *fonctions* en C), ne sont pas des "objets comme les autres" dans ce langage, contrairement aux langages fonctionnels.

Nous verrons à l'usage que le deuxième inconvénient n'est pas aussi catastrophique que certains pourraient le craindre. Bien sûr un langage fonctionnel (ou un langage de calcul formel comme Maple, qui manipule en outre des entiers de taille quelconque) serait plus approprié, et tout étudiant maîtrisant correctement un tel langage est encouragé à traduire dans ce langage les procédures écrites ici en C ; mais la calculabilité est une matière suffisamment riche (et déroutante au début) pour ne pas ajouter en prérequis la maîtrise d'un langage fonctionnel commun.

Les procédures, qui sont des cas particuliers de textes, et donc de mots, sont donc numérotables, par l'algorithme suivant :

- parcourir les mots (sur l'alphabet des 96 caractères "imprimables" ASCII) dans l'ordre de leur numérotation (voir section 3 de ce chapitre) ;

- ne retenir que ceux qui définissent des procédures, à l'aide d'un *analyseur syntaxique*, et les numéroter dans l'ordre de leur apparition.

Cet algorithme schématique est radicalement inefficace (voir chapitre 0, introduction), mais est logiquement cohérent. En particulier un langage informatique est inséparable d'un algorithme capable de distinguer les procédures syntaxiquement correctes. Une autre méthode plus proche de la pratique consiste à associer à toute procédure son *adresse* ; cette numérotation n'est pas surjective (un entier quelconque est rarement une adresse valide), dépend de la machine et en pratique n'attribue un numéro qu'à un petit nombre de procédures (celles définies au moment de la compilation) ; mais nous verrons qu'elle permet d'illustrer correctement, dans les chapitres suivants, des raisonnements fondamentaux.

Nous terminerons en insistant sur le résultat, qui peut être considéré comme un *axiome* de tout modèle de calcul :

Les procédures sont numérotables.

Annexe

On trouvera en annexe le texte intégral de quelques fonctions C qui réalisent les numérotations décrites dans ce chapitre ; ces algorithmes peuvent avoir leur propre intérêt dans d'autres branches de l'informatique.