Encodage des nombres

Guillaume Rousseau
MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

26 septembre 2024

Nous avons vu en TP comment manipuler des nombres en C. Dans ce chapitre, on s'intéresse à la manière dont les nombres, et les autres informations, sont stockées sur un ordinateur. L'idée est que sur un ordinateur, tout est fait de suites de bits, c'est à dire de 0 et de 1. Des groupes de bits structurés permettent de stocker des informations.

1 Langage et alphabet

Définition 1. Soit Σ un ensemble fini, que l'on appellera ***alphabet***. On appelle ***mot de longueur*** l ***sur*** Σ tout élément $u = (u_0, u_1, \ldots, u_{l-1}) \in \Sigma^l$. On appelle ***mot sur*** Σ tout élément de l'ensemble suivant :

$$\Sigma^* = \bigcup_{l=0}^{\infty} \Sigma^l$$

L'unique élément de Σ^0 est noté ε, on l'appelle ***mot vide***.

Σ^* est donc l'ensemble des mots de tailles finies utilisant des lettres de Σ.
Attention : cet ensemble contient des mots de tailles aussi grandes que l'on veut, mais ne contient pas de mot infini.

Si $u = (u_0, u_1, \ldots, u_{l-1})$ est un mot sur Σ^*, on le notera simplement $u_0 u_1 \ldots u_{l-1}$

Exemple 1. BONJOUR est un mot sur l'alphabet latin, de longueur 7. 1983 est un mot sur l'alphabet $[\![0, 9]\!]$ de longueur 4

.

Définition 2. Soit E un ensemble.

— Un encodage de E sur l'alphabet Σ est un couple de fonctions $(\mathbf{enc}, \mathbf{dec})$ avec :

$$\begin{array}{lcl} \mathbf{enc} & : E & \to \Sigma^* \\ \mathbf{dec} & : \Sigma^* & \to E \end{array}$$

telles que pour tout $x \in E$, $\mathbf{dec}(\mathbf{enc}(x)) = x$

— Pour $l \in \mathbb{N}$, un encodage ($\mathbf{enc}$, $\mathbf{dec}$) de E sur l'alphabet Σ est dit **de longueur** l si pour tout $x \in E$, $\mathbf{enc}(x)$ est de longueur l.

Notons qu'en particulier l'encodage doit être injectif, et donc que E doit être de taille au plus $|\Sigma|^l$.

Un encodage de E permet donc de transformer tout élément de E en une suite de lettres, de façon à pouvoir décoder facilement la suite afin de retrouver l'élément d'origine. Dans un encodage de longueur l, tous les codes doivent avoir la même longueur.

La mémoire de l'ordinateur peut être vue comme une immense liste de 0 et de 1. Toute l'information contenue dans un ordinateur est donc encodée sur l'alphabet $\{0, 1\}$: le texte, les nombres, la musique, et même les programmes !

2 Codage des entiers

A Introduction aux bases

Retour à l'école primaire Revoyons le fonctionnement du système décimal, i.e. le système habituel pour compter. Le nombre 386 signifie "trois centaines plus huit dizaines plus six unités". Autrement dit :

$$386 = 3 \times 10^2 + 8 \times 10^1 + 6 \times 10^0$$

Le système décimal, aussi appelé base 10, comporte **10** chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. L'écriture en base 10 d'un nombre $n \in \mathbb{N}$ est une suite de chiffres, i.e. un mot sur l'alphabet $\{0, 1, \ldots, 9\}$. Il faut bien faire la différence entre un nombre entier, qui est un élément de $\mathbb{N}$ et son écriture en base 10, qui est une suite de chiffre : l'écriture en base 10 permet **d'encoder** les entiers naturels.

On introduit maintenant le système binaire, aussi appelé la base 2. Dans ce système, il n'y a que deux chiffres : 0 et 1. Pour faire la différence entre les nombres écrits en binaire et en décimal, on note $\overline{1101}^2$ les nombres binaires. Pour comprendre un nombre binaire, on procède comme en décimal : par exemple, pour $\overline{1101}^2$, on a :

$$\overline{1101}^2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 1 = 13$$

Ainsi, la suite de chiffre $(1, 1, 0, 1)$ en base 2 et la suite de chiffres $(1, 3)$ en base 10 représentent un seul et même entier, "treize".
Dans un ordinateur, les nombres sont donc écrits dans le système binaire, sur des ***bits***. Les bits sont souvent regroupés par 8, formant un ***octet*** / ***byte***, ou par groupes plus larges (16, 32, 64). Donc, souvent en binaire, contrairement au décimal, on écrit les nombres sur une longueur fixe. Par exemple, en binaire on pourra écrire $\overline{00001101}^2$ sur 8 chiffres, alors qu'en décimal on écrit rarement 00013 pour dire "treize".

Exercice 1.

1. Écrire les nombres de 0 à 15 en binaire sur 4 chiffres.

Décimal	Binaire
0	
1	
2	
3	

Décimal	Binaire
4	
5	
6	
7	

Décimal	Binaire
8	
9	
10	
11	

Décimal	Binaire
12	
13	
14	
15	

2. Peut-on écrire 16 en binaire sur 4 chiffres ? Combien de chiffres faut-il ?
3. Combien de nombres différents peut-on écrire dans un octet ?
4. Jusqu'à combien peut-on compter sur ses doigts (à 2 mains) ?

En généralisant ces idées, on obtient ***l'écriture en base*** B :

Définition 3. Soit $B \in \mathbb{N}$, $B \neq 0, 1$. L'écriture en base B d'un entier $n \in \mathbb{N}$ est un mot $a_{l-1} \dots a_0$ sur l'alphabet $[\![0, B-1]\!]$ satisfaisant :

$$\sum_{i=0}^{l-1} a_i B^i = n$$

On dit que $B \in \mathbb{N}$ est une ***base*** lorsque $B > 1$.

Dans la suite de ce chapitre, pour $B \in \mathbb{N}$, on note $\Sigma_B = [\![0, B-1]\!]$.

Pour $u = a_{l-1} \dots a_0$ un mot sur l'alphabet Σ_B, on note $\overline{u}^B = \overline{a_{l-1} \dots a_0}^B = \sum_{i=0}^{l-1} a_i B^i$. Ainsi, $\overline{u}^B$ est l'entier représenté par u en base B.

Remarque 1. Lorsque l'on parle d'écriture en base B, on note les mots $a_{l-1} \dots a_0$ et pas $a_0 \dots a_{l-1}$, de la même manière que l'on écrit "cinquante-trois" 53 et pas 35 : on dit que le chiffre le plus significatif est à gauche.

Remarque 2. Il faut bien faire la différence entre la valeur d'un entier $n \in \mathbb{N}$ et son écriture en base B. La fonction qui à n associe son écriture en base B est un **encodage**, et la fonction qui à une suite de chiffres en base B associe un nombre est un **décodage**.

B Quelques exemples

Exercice 2.

Question 1. Combien valent les nombres suivants ?

a) $\overline{10011}^2$ b) $\overline{30}^5$ c) $\overline{66}^7$ d) $\overline{01111111}^2$ e) $\overline{13}^{11}$

Question 2. Calculez la décomposition de n en base B pour :

a) $B = 2, n = 37$ b) $B = 7, n = 82$ c) $B = 16, n = 35$

Question 3. Écrire les nombres suivant en base 4 :

a) 11 b) 11×4 c) 11×4^2 d) 11×4^n

e) 73 f) $\lfloor \frac{73}{4} \rfloor$ g) $\lfloor \frac{73}{4^2} \rfloor$ h) $\lfloor \frac{73}{4^3} \rfloor$ i) 73 mod 4.

j) 86 k) $\lfloor \frac{86}{4} \rfloor$ l) $\lfloor \frac{86}{4^2} \rfloor$ m) $\lfloor \frac{86}{4^3} \rfloor$ n) 86 mod 4.

Question 4. Soit B une base, et $u_{l-1} \dots u_0$ l'écriture en base B d'un entier $n \in \mathbb{N}$. Donner l'écriture en base B de :

a) Le reste de n par B
b) Le quotient de n par B
c) Le quotient de n par B^2
d) Le quotient de n par B^k, pour $k \in \mathbb{N}$.

Question 5. Étant donné B une base, $n \in \mathbb{N}$ et $k \in \mathbb{N}$, donner une formule mathématique permettant d'obtenir le k-ème chiffre de n en base B.

C Base hexadécimale

On s'intéresse à trois bases particulières en informatique :

— La base 10 (décimale) car c'est la base utilisée par les humains.

— La base 2, (binaire) car c'est la base utilisée par les ordinateurs

— La base 16, (hexadécimale), car elle permet de compacter la base 2. En effet, $16 = 2^4$, donc un chiffre en base 16 correspondra à une suite de 4 chiffres en base 2.

Pour écrire des nombres en base 16, il nous faut 16 chiffres. En plus des chiffres 0, 1, ..., 9, on rajoute les lettres a, b, ..., f afin d'avoir 16 symboles distincts.

On peut donc étendre la table des nombres de 0 à 15 vue plus haut pour y rajouter les chiffres hexadécimaux :

Dec	Bin	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3

Dec	Bin	Hex
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Dec	Bin	Hex
8	1000	8
9	1001	9
10	1010	a
11	1011	b

Dec	Bin	Hex
12	1100	c
13	1101	d
14	1110	e
15	1111	f

Par exemple $\overline{b2e}^{16}$ vaut $11 \times 16^2 + 2 \times 16 + 14 \times 1$.

Exemple 2. Soit $n = \overline{1101001011110100}^2$. Pour décomposer n en base 16 à partir de la base 2, on peut repasser par la base 10, avec laquelle nous sommes plus familiers, puis chercher la décomposition en base 16. Cependant, on peut faire la traduction directement. Pour cela, on découpe l'écriture en base 2 par paquets de 4, puis l'on remplace chaque paquet par le chiffre en base 16 correspondant :

$$\begin{array}{rcccccc} n & = & \overline{1101} & \overline{0010} & \overline{1111} & \overline{0100} \\ & = & d & 2 & f & 4 \end{array}$$

Donc, $n = \overline{d2f8}^{16}$. On notera aussi `0xd2f8`, en préfixant le nombre par ***0x***, pour signifier que l'on parle en hexadécimal. Cette notation peut être utilisée en C :

```
1 int x = 0xd2f8;
```

Être à l'aise avec l'hexadécimal est très utile en informatique, car les adresses mémoires, les couleurs, et d'autres valeurs sont souvent affichées en base 16. De plus, un chiffre hexadécimal représentant 4 bits, un octet est représentable avec 2 chiffres en hexadécimal.

D Propriétés de l'écriture en base B

On admet pour l'instant que tout nombre entier admet une écriture en base B, pour tout $B \in \mathbb{N}$, $B \neq 0, 1$. Soit $l \in \mathbb{N}$. Regardons quels nombres peuvent être écrits sur l chiffres seulement en base B. On commence par un encadrement de ces nombres.

Théorème 1. Soit $l \in \mathbb{N}$. Soit $n \in \mathbb{N}$ avec $n = \overline{a_{l-1} \dots a_0}^B$. Alors $0 \leq n < B^l$.

Démonstration. Pour $i \in [\![0, l-1]\!]$, on a $0 \leq a_i < B$. Donc :

$$0 = \sum_{i=0}^{l-1} 0.B^i \leq \sum_{i=0}^{l-1} a_i B^i \leq \sum_{i=0}^{l-1} (B-1)B^i = (B-1)\frac{B^l - 1}{B-1} = B^l - 1$$

D'où $0 \leq n < B^l$. □

Remarquons que l'on dit "l'écriture" et pas "une écriture" en base B. Cela sous-entend l'**existence** et l'**unicité** : chaque n peut s'écrire en base B, et ce de manière unique. En réalité, nous avons déjà vu que l'écriture n'est pas unique, car on peut rajouter autant de 0 que l'on veut.

Cependant, on a un résultat de quasi unicité :

Théorème 2. Soit $B \in \mathbb{N}$, $B \neq 0, 1$, et $n \in \mathbb{N}$. Soient $a_{k-1}, \dots, a_0, b_{l-1}, \dots, b_0 \in [\![0, B-1]\!]$ avec $k \leq l$ tels que :

$$\sum_{i=0}^{k-1} a_i B^i = n = \sum_{i=0}^{l-1} b_i B^i$$

i.e. deux écritures de n en base B sur k et l chiffres. Alors :

$$\forall i \in [\![0, k-1]\!], a_i = b_i \tag{1}$$
$$\forall i \in [\![k, l-1]\!], b_i = 0 \tag{2}$$

Démonstration. Nous allons prouver directement la première égalité. L'idée de la preuve est d'exprimer chaque a_i et chaque b_i en fonction de n uniquement, en utilisant des divisions euclidiennes. Afin de mieux comprendre l'idée de la preuve, commençons par montrer que $a_0 = b_0$. On a :

$$n = a_0 \sum_{i=1}^{k-1} a_i B^i \tag{3}$$
$$= a_0 + B \sum_{i=1}^{k-1} a_i B^{i-1} \tag{4}$$

De plus, $a_0 \in [\![0, B-1]\!]$: on a simplement écrit la division euclidienne de n par B. Donc a_0 est le reste de la division euclidienne de n par B.

En appliquant le même raisonnement sur les b_i, on montre de même que b_0 est également le reste de la division euclidienne de n par B. Par unicité de la division euclidienne, $a_0 = b_0$.

Nous allons maintenant généraliser ce raisonnement et donner des expressions explicites des a_i en fonction de n. Pour $i \in [\![0, k-1]\!]$, on note q_i le quotient de la division euclidienne de n par B^i. Soit $i_0 \in [\![0, k]\!]$. Montrons que a_{i_0} est le reste de la division euclidienne de q_{i_0} par B.

On a :

$$n = \sum_{i=0}^{k-1} a_i B^i = \sum_{i=0}^{i_0-1} a_i B^i + B^{i_0} \sum_{i=i_0}^{k-1} a_i B^{i-i_0}$$

Or, on a vu :

$$\sum_{i=0}^{i_0-1} a_i B^i < B^{i_0}$$

Donc, $q_{i_0} = \sum_{i=i_0}^{k-1} a_i B^{i-i_0} = \sum_{j=0}^{k-i_0-1} a_{i_0+j} B^j$, et donc a_{i_0} est le reste de la division euclidienne de q_{i_0} par B par un raisonnement analogue à celui du début de la preuve. Comme précédemment, en appliquant le même raisonnement sur les b_i, on en déduit que b_{i_0} est également le reste de la division euclidienne de q_{i_0} par B. Donc, $a_{i_0} = b_{i_0}$.

Donc, $a_{k-1}, \dots, a_0 = b_{k-1}, \dots, b_0$. Comme de plus, $\sum_{i=0}^{k-1} a_i B^i = n$, on a $\sum_{i=0}^{k-1} b_i B^i = n = \sum_{i=0}^{l-1} b_i B^i$, et donc $b_k = \dots = b_{l-1} = 0$. □

Par exemple, les seules écritures en base 2 de 5 sont $\overline{101}^2, \overline{0101}^2, \overline{00101}^2, \ldots$. Une conséquence de ce théorème est qu'à longueur fixée, il y a bien unicité de l'écriture en base B.

Lorsque l'on parlera de l'écriture en base B d'un nombre, alors soit on sera sur un nombre de chiffres fixés, soit on sous-entendra que l'on parle de l'écriture de taille minimale.

Montrons maintenant l'existence de l'écriture en base B.

Théorème 3. Soit $n \in \mathbb{N}$ et B une base. Alors, n admet une écriture en base B.

Démonstration. Nous allons exhiber un algorithme calculant une décomposition dans une base B donnée d'un entier n donné, et montrer que cet algorithme est correct. Cela permettra ainsi de montrer l'existence de l'écriture en base B, et même de disposer d'un procédé de construction !

Algorithme 1 : Décomposition en base

Entrée(s) : $n \in \mathbb{N}^*$, B une base
Sortie(s) : $a_{l-1} \ldots a_0$ la décomposition de n en base B

```
1 N ← n ;
2 l ← 0 ;
3 tant que N > 0 faire
4 |   q, r ← la division euclidienne de N par B;
5 |   N ← q;
6 |   a_l ← r;
7 |   l ← l + 1;
8 retourner a_{l-1} ... a_0
```

On complète l'algorithme en posant que l'écriture en base B de 0 est $\overline{0}^B$.

Exécutons cet algorithme sur un exemple pour observer son fonctionnement :

On peut alors remarquer qu'à chaque étape, on a coupé n en deux parties : une partie qui a été encodée dans les a_i, et une partie qui est encore stockée dans N. On peut alors recoller les deux parties ensemble et trouver l'invariant de boucle I suivant :

$$NB^l + \sum_{i=0}^{l-1} a_i B^i = n$$

Montrons que c'est bien un invariant de boucle.

— En entrée de boucle, $l = 0$, et $N = n$. On a bien l'égalité demandée, et I est donc bien vérifiée.

— Supposons I vérifiée au début d'un passage de boucle. On pose N', l' les valeurs de N et l à la fin du passage. Posons q, r le quotient et le reste de la division euclidienne de N par B. On a $N' = q$, $l' = l + 1$, et $a_l = r$. De plus, par propriété de la division euclidienne, $N = qB + r$, i.e. $N = N'B + a_l$. Vérifions I en fin de passage :

$$\begin{aligned} N'B^{l'} + \sum_{i=0}^{l'-1} a_i B^i &= qB^{l+1} + \sum_{i=0}^{l} a_i B^i \\ &= qB^{l+1} + \sum_{i=0}^{l-1} a_i B^i + a_l B^l \\ &= B^l(qB + a_l) + \sum_{i=0}^{l-1} a_i B^i \\ &= B^l N + \sum_{i=0}^{l-1} a_i B^i \\ &= n \qquad \text{par hypothèse d'invariant} \end{aligned}$$

Ainsi, I se conserve au cours du passage.

I est bien un invariant de boucle, et en particulier est vrai en sortie de boucle, lorsque N vaut 0. On a alors précisément que $\sum_{i=0}^{l-1} a_i B^i = n$, d'où la correction de l'algorithme. □

En TD : algorithme de décodage

E Représentation des entiers naturels

Entiers non-signés En C, les types `unsigned char` (8 bits), `unsigned int` (32 bits), ainsi que `unsigned long int` (64 bits) sont des types ***non-signés***, ce qui signifie qu'ils représentent des nombres positifs uniquement : le signe n'est pas encodé ! Un type non-signé est simplement une écriture en base 2, et donc ne peut stocker que des nombres entre 0 et $2^l - 1$, où l est le nombre de bits du type. Par exemple, le type `unsigned int` fait 32 bits et peut donc stocker des nombres entre 0 et $2^{32} - 1$.

Remarque 3. En réalité, la longueur des types `unsigned int` et `unsigned long int` ne sont pas garantis. Sur certaines machines, ils peuvent valoir 16 et 32, ou bien tous les deux 32, etc... Il existe des types dont la taille est garantie, dans la librairie `stdint.h`. Ces types sont `uint8_t`, `uint16_t` et `uint32_t`, le nombre de bits qu'ils contiennent étant écrit dans le type.

Définition 4. Soit $n = \overline{a_{l-1} \dots a_0}^2$ un entier signé sur l bits. Le bit le plus à gauche, a_{l-1}, coefficient de 2^{l-1}, est appelé ***bit de poids fort***. Inversement, le bit le plus à droite, a_0, coefficient de $2^0 = 1$, est appelé ***bit de poids faible***.

Exercice 3.

Question 1. Donnez les bits de poids forts et de poids faible des entiers suivants, écrits sur 8 bits :

a) 9 b) 231 c) 128 d) 127

Question 2. Donnez une règle simple pour déterminer le bit de poids fort, et une pour déterminer le bit de poids faible.

Opérations bit à bit Dans cette partie, on identifie les entiers et leur encodage de longueur l sur $\{0, 1\}$, et on les voit comme des vecteurs de 0 et de 1. Sur ces vecteurs, on peut appliquer les opérations booléennes &, | et $\sim$, qui font respectivement un ET, un OU, et un NON sur chacun des bits.

Exemple 3. On se place sur 8 bits. Soit $a = 25 = \overline{00011001}^2$ et $b = 149 = 10010101$. Alors :

$$a\&b = \overline{00010001}^2 = 17$$
$$a|b = \overline{10011101}^2 = 157$$
$$\sim a = \overline{11100110}^2 = 230$$

On introduit également les opérateurs de ***décalage*** : >> et <<. Comme leur nom l'indique, ces opérateurs décalent les bits d'un nombre, vers la gauche ou la droite, d'un certain nombre de places, en remplaçant par des 0.

Exemple 4. Soit $a = 25 = \overline{00011001}^2$. Alors :

$$a >> 1 = \overline{00001100}^2 = 12$$
$$a << 1 = \overline{00111010}^2 = 50$$
$$a >> 4 = \overline{00000001}^2 = 1$$
$$a << 3 = \overline{11001000}^2 = 200$$

On remarque que $a >> k = \lfloor \frac{a}{2^k} \rfloor$. De plus, si l'on est sur l bits, $a << k = (a \times 2^k) \mod 2^l$.

Addition

Définition 5. L'addition binaire est une fonction $\mathbf{add} : \Sigma_2^* \times \Sigma_2^* \to \Sigma_2^*$ telle que :

$$\forall a, b \in \Sigma_2^*, \overline{(\mathbf{add}(a,b))}^2 = \overline{a}^2 + \overline{b}^2$$

Ici, l'addition est donc une fonction agissant sur les **écritures** des nombres, et pas directement sur les **nombres**.
L'algorithme classique d'addition en base 2 fonctionne selon le même principe qu'en base 10.

Exemple 5. Voici comment effectuer l'addition de $\overline{10110}^2$ et $\overline{1101}^2$:

Exemple 6. $\mathbf{add}(101, 111) = 1100$

Lorsque l'on additionne des nombres sur un nombre fixé de bits, il se peut que le résultat ne tienne pas dans le type considéré. Par exemple, si on prend a et b des `unsigned char`, tenant sur 8 bits, avec $a = 180$ et $b = 97$:

Dans ce cas, le bit de poids fort disparaît, ce qui revient à prendre le résultat modulo 2^l. Lorsque cela se produit, on parle de ***dépassement d'entier***, ou ***integer overflow*** en anglais.

Exercice 4. Faire les addition suivantes en binaire sur 8 bits, et dire lorsqu'il y a dépassement d'entier.

a) 95 + 132 b) 163 + 92 c) 128 + 167

d) 7 + 64 e) 184 + 233 f) 94 + 105

Successeur On peut s'intéresser à un type d'addition particulier : l'incrémentation de 1.

Exemple 7. Additions en binaire : 101111 et 1, puis 1010011 et 1 :

Soustraction On introduit la technique du ***complément à 9*** pour faire des soustractions en base 10. Cette technique permet de transformer les soustractions en additions.

On se donne a et b des entiers sur l chiffres en base 10, et l'on veut calculer $a - b$ en base 10. On utilise le résultat suivant :

$$\begin{aligned} a - b &= a - b + 10^l - 1 - 10^l + 1 \\ &= a + (10^l - 1 - b) - 10^l + 1 \\ &= a + (9...9 - b) - 10^l + 1 \end{aligned}$$

On remarque ensuite que soustraire 10^l et ajouter 1 sont deux opérations simples en base 10.

Enfin, pour calculer $9...9 - b$, en notant $b = \overline{b_{l-1} \dots b_0}^{10}$, on a :

$$9 \dots 9 - b = \overline{(9 - b_{l-1}) \dots (9 - b_0)}^{10}$$

Ainsi, la soustraction $9 \dots 9 - b$ n'implique aucune retenue, elle nécessite juste de remplacer chaque chiffre individuel par son complément à 9 : 6 par 3, 2 par 7, 9 par 0, etc...

Exemple 8. Utilisons le complément à 9 pour calculer une grosse soustraction :

Pour les nombres en base 2, cette méthode s'appelle méthode du ***complément à 1***. Si l'on se restreint à des entiers sur l bits, la méthode ci-dessus se simplifie car 2^l vaut 0 (on travaille modulo 2^l). En notant $!b$ la négation bit à bit de b, on a :

$$a - b = a + !b + 1$$

Lorsque $b > a$, le résultat de la soustraction dépasse 0 et va donc repartir de $2^l - 1$. On appelle encore ça un dépassement d'entier, et en anglais un *integer underflow*.

Exercice 5. Faites les soustractions suivantes en base 2 sur 8 bits, et notez celles qui causent un dépassement.

a) 163 - 92 b) 128 - 67 c) 95 - 132

d) 255 - 136 e) 128 - 36 f) 94 - 105

F Représentation des entiers relatifs

Entiers signés En C, les types `char` (8 bits), `int` (32 bits), `long int` (64 bits) sont des types ***signés***, ce qui signifie qu'ils représentent des entiers ayant un ***signe***.

On pourrait imaginer que dans un type signé sur l bits, on stocke d'abord le signe dans un bit, puis sur les $l-1$ bits restants la valeur absolue du nombre. Cependant, ce système a plusieurs inconvénients. Le plus gros est que 0 a deux représentations : $000\ldots0$ et $100\ldots0$. De plus, l'algorithme d'addition deviendrait bien plus compliqué.

En fait, un type non signé va presque être une décomposition en base 2. Seulement, le bit de poids fort ne correspondra pas à 2^{l-1} mais à -2^{l-1} Autrement dit, si $a_{l-1}a_{l-2}\ldots a_0$ est un entier signé sur l bits, alors sa valeur est :

$$\sum_{i=0}^{l-2} a_i 2^i - a_{l-1}2^{l-1}$$

Cette formule permet de ***décoder*** les entiers non signés, c'est à dire de passer de l'écriture à la valeur. Le plus petit nombre encodable est -2^{l-1}, qui s'écrit $100\ldots0$ et le plus grand est $1+2+\cdots+2^{l-2} = 2^{l-1}-1$, qui s'écrira donc $011\ldots1$.

Définition 6. Cette manière d'encoder les entiers relatifs s'appelle le ***complément à 2*** (à ne pas confondre avec le complément à 1, qui est une méthode de soustraction).

Étudions maintenant l'opération d'***encodage***, permettant de passer de la valeur à l'écriture.

Soit $n \in [\![-2^{l-1}, 2^{l-1}-1]\!]$. Alors :

— Si $n \geq 0$, alors l'encodage de n est son écriture en base 2 sur l bits.

— Si $n < 0$, alors l'encodage de n est l'écriture en base 2 de $2^l - |n| = 2^l + n$ sur l bits.

Exercice 6. Compléter le tableau suivant :

n	n sur un **char**
0	
	0000 1100
99	0110 0011
	0111 1111
	1111 1111
-99	
	1000 0000
-64	

Addition et négation Avec cette représentation des entiers signés, l'algorithme d'addition est exactement le même !

Démo : $11001001 + 00101101 = -55 + 46$

G Petit-boutisme et gros-boutisme

Le type int fait 32 bits, ce qui correspond à 4 octets. Donc, en mémoire, une variable de ce type est stockée dans 4 octets successifs. Instinctivement, on peut s'attendre à ce que ces octets soient stockés du plus important au moins important : l'octet de poids fort en premier et l'octet de poids faible en dernier. Cependant, presque toujours dans la mémoire d'un ordinateur, les octets sont stockés dans l'ordre inverse.

Définition 7. Le **petit-boutisme** est le fait de stocker les entiers avec l'octet de poids faible en premier. Le **gros-boutisme** est le fait de les stocker avec l'octet de poids fort en premier.

Le gros-boutisme est l'ordre qui semble le plus naturel, mais le petit-boutisme est plus utilisé en pratique.

En anglais, on parle de **little-endianness** et de **big-endianness**. Un système ou protocole peut donc être **little-endian** ou **big-endian**.

Exemple 9. On écrit un entier 32 bits : `0x1a56cc9d` et 0x5bfce306.

— Avec un protocole gros-boutiste, on écrirait les octets suivants :

```
1a 56 cc 9d 5b fc e3 06
```

— Avec un protocole petit-boutiste, on écrirait les octets suivants :

```
9d cc 56 1a 06 e3 ce 5b
```

Cependant, dans les deux protocoles, les octets sont écrits dans le même sens. Par exemple, l'octet `9d` serait écrit `10011110` dans les deux cas. Pour résumer :

— l'ordre des nombres est le même peu importe le protocole ;

— l'ordre des octets au sein de chaque nombre est inversé selon le protocole ;

— l'ordre des bits au sein de chaque octet est le même peu importe le protocole.

Lorsque l'on écrit un programme qui doit lire un format de fichier ou utiliser un canal de communication, il est important de connaître le boutisme utilisé, afin de lire les nombres correctement.

3 Nombres à virgule flottante

Les nombres à virgule flottante servent à représenter les nombres réels. L'idée est de faire comme pour l'écriture scientifique, et de stocker le signe, l'ordre de grandeur, et un nombre de chiffres significatifs.

Plus précisément, le codage d'un nombre flottant se compose de trois parties :

— Un ***signe*** s stocké sur 1 bit

— Un ***exposant*** $e_{k-1} \dots e_0$ stocké sur k bits

— Une ***mantisse*** $m_1 \dots m_l$ stockée sur l bits

Les valeurs de k et l sont fixées par le type que l'on utilise. En C, le type `float` tient sur 32 bits, avec $k = 8, l = 23$.

La valeur d'un nombre flottant se calcule comme suit : Soit $f = se_{k-1} \dots e_0 m_1 \dots m_l \in \{0,1\}^{1+k+l}$. Alors :

— Si $\forall i \in [\![0, k-1]\!], e_i = 1$, alors :

  — Si $\forall j \in [\![1, l]\!], m_j = 0$ alors f encode $(-1)^s \infty$

  — Sinon, f est invalide, et encode la valeur ***Not a Number*** (ou NaN).

— Si $\forall i \in [\![0, k-1]\!], e_i = 0$ alors la valeur encodée par f est dite ***dénormalisée***, et vaut :

$$(-1)^s 2^{-2^{k-1}+2} \sum_{i=1}^{l} \frac{m_i}{2^i}$$

— Sinon, alors la valeur encodée par f est dite ***normalisée*** et vaut :

$$(-1)^s 2^{-2^{k-1}+1+\overline{e_{k-1}\dots e_0}^2} (1 + \sum_{i=1}^{l} \frac{m_i}{2^i})$$

Exemple 10. Quelles sont les valeurs encodées par les flottants suivants avec l'exposant sur $k = 8$ bits et la mantisse sur $l = 23$ bits ? N'utilisez pas de calculatrice !

1. 0x00000000
2. 0x3f800000
3. 0x419a0000
4. 0x41980000

Remarque 4. Il y a un nombre fini de nombres flottants sur un nombre de bits fixés.

Donc, les flottants ne peuvent pas représenter de manière parfaitement fidèle tous les nombres réels. Tous nombre pouvant être écrit de manière exacte comme un flottant sera de la forme $\frac{n}{2^m}$ avec $n, m \in \mathbb{N}$. Par contraposée, tout nombre qui ne peut pas s'écrire sous cette forme ne peut **pas** être représenté de manière parfaite comme un flottant. Par exemple, $\frac{1}{3}$ et $\frac{1}{10}$ sont forcément stockés sous la forme d'approximations !

4 Encodage de données quelconques

Encodages de longueur fixe

Pour encoder un ensemble E fini quelconque, une méthode simple est d'énumérer E, c'est à dire de faire correspondre chaque élément de E à un élément de $[\![0, |E| - 1]\!]$. En notant $\varphi : E \to [\![0, |E| - 1]\!]$ cette correspondance (bijective), on choisit $l \in \mathbb{N}$ tel que $|E| \leq 2^l$ et on peut alors encoder les éléments de E sur l bits, en associant à chaque $x \in E$ l'écriture binaire de $\varphi(x)$.

Exemple 11. L'encodage ASCII est une méthode d'encodage du texte. Il permet d'encoder les lettres majuscules et miniscules, les chiffres, plusieurs signes de ponctuation ainsi que les espaces et retours à la ligne. Chacun de ces caractères est associé à un entier entre 0 et 127.

Le code ASCII tient donc sur 7 bits. Cependant, les briques de base des ordinateurs sont les octets. Il reste donc 128 valeurs inutilisées. Les ordinateurs utilisent généralement des variantes de l'encodage ASCII, où les 128 premières valeurs correspondent aux codes ASCII tandis que les 128 suivantes sont utilisées pour des caractères additionnels. Par exemple, l'encodage Windows-1258 encode certaines lettres du vietnamien, et l'encodage Windows-1252 encode les accents les plus communs (é, è, à, etc...) ainsi que quelques symboles additionnels.

On étudiera l'encodage ASCII de plus près en TP.

Les systèmes d'encodage les plus simples consistent donc à faire correspondre certains éléments de bases avec des entiers, puis à encoder ces entiers en base 2.

Exemple 12. Le modèle RGB (Red-Green-Blue) est une manière de représenter les couleurs en associant à chaque couleur un triplet (r, g, b) indiquant respectivement le niveau de rouge, de vert et de bleu. Typiquement, on a $r, g, b \in [\![0, 255]\!]$.

On peut alors encoder une couleur (r, g, b) sur 3 octets, chaque octet stockant respectivement l'écriture binaire de r, g et b. Lorsque l'on code en HTML/CSS/JavaScript, il est très courant d'utiliser des valeurs RGB.

On peut ensuite s'intéresser à l'encodage d'une image. On suppose que nos images ne peuvent pas faire plus de 2^{32} pixels de haut ou de large. On peut alors encoder une image de $n \times m$ pixels en utilisant $8 + 3nm$ octets comme suit :

— Tout d'abord, on écrit en base 2 sur 32 bits n et m

— Puis on liste les valeurs r, g, b de chaque pixel comme précédemment, ligne par ligne et colonne par colonne. Chaque pixel prend 3 octets.

Il est important d'écrire les valeurs de n et m, pour pouvoir différencier une image de taille $n \times m$ d'une image de taille $m \times n$. Ceci permet de reconstituer l'image, c'est à dire de **décoder** une suite de bits encodée selon ce protocole.

Encodages de longueur variables

Lorsque l'on écrit un nombre dans une variable C de type int, ou long int, on a un nombre de bits fixe. Cependant, lorsque l'on écrit dans un fichier on n'écrit que des suites de 0 et de 1. Rien n'empêche donc de concevoir des protocoles d'encodage pour lesquels tous les éléments ne sont pas encodés par des suites de bits de même longueur, du moment que cet encodage peut être décodé.

Exemple 13. On encode chaque $n \in \mathbb{N}$ par la suite de bits 1^n0, c'est à dire $11\ldots10$ avec n 1. Par exemple, un fichier contenant la suite de bits suivante encoderait les nombres 5 et 8 :

```
111110 11111110
```

On ne peut pas se passer du 0 car il **délimite les différents nombres** !

L'encodage précédent n'est pas très malin car pour encoder un entier n, on a besoin de $n+1$ bits, ce qui est bien plus que les $\lceil \log_2(n) \rceil$ bits nécessaires pour l'écriture de n en base 2.

On peut proposer l'encodage suivant, un peu plus malin :

Exercice 7. Soit $n \in \mathbb{N}$. On note $a_k \ldots a_0$ son écriture en base 2. On encode n en remplaçant chaque 1 dans son écriture binaire par 11 et chaque 0 par 10, et en ajoutant 00 à la fin.
Par exemple, pour encoder $21 = \overline{10101}^2$, on écrira :

```
11 10 11 10 11 00
```

Pour encoder 21 puis $14 = \overline{11100}^2$ on écrira :

```
11 10 11 10 11 00    11 11 11 10 10 00
```

Question 1. A quoi sert le doublet de bits 00 à la fin de chaque code ?

Question 2. De combien de bits a-t'on besoin pour encoder $n \in \mathbb{N}$?

En pratique, l'encodage précédent est difficilement utilisable pour des ordinateurs, car on ne peut pas lire ou écrire bit par bit dans un fichier, mais seulement octet par octet.

L'encodage de longueur variable le plus connu est UTF8, qui permet d'encoder les caractères latins, grecs, arabes, chinois, etc... Au total, plus d'un million de caractères différents sont encodés par UTF8, et les codes peuvent faire entre 1 et 4 octets. A noter que cet encodage est compatible avec l'ASCII, dans le sens où les caractères ayant un code ASCII auront exactement le même code en UTF8.

Cet encodage fonctionne sur un principe simple mais efficace, et est utilisé dans l'écrasante majorité des systèmes modernes, sur internet, etc... Le principe est que le premier octet d'un code indique le nombre total d'octets que le code contient :

- Si le premier octet est de la forme `0xxxxxxx` alors le code fait un octet, et les 7 valeurs suivantes forment un code ASCII ;
- Si le premier octet est de la forme `110xxxxx` alors le code fera deux octets et aura 11 bits libres (2048 valeurs possibles), ce qui suffit pour encoder la quasi totalité des alphabets hors chinois, japonais et coréen ;
- Si le premier octet est de la forme `1110xxxx` alors le code fera trois octets, aura 16 bits libres (65536 valeurs possibles), couvrant l'intégralité des alphabets modernes ;
- Si le premier octet est de la forme `11110xxx` alors le code fera quatre octets et aura 21 bits libres (plus de 2 millions de valeurs possibles).

L'encodage VLQ (Variable Length Quantity) utilise un système similaire pour encoder des entiers quelconques sans limite de taille. Cet encodage est utilisé en pratique, par exemple pour représenter des nombres dans les fichiers MIDI !