

Corrigé TD2: Encodage des nombres

MP2I Lycée Pierre de Fermat
guillaume.rousseau@ac-toulouse.fr

Exercice 1.

Q1. a) 19

b) 15

c) 48

d) 127

e) 14

Q2. a) 100101

b) 145

c) 23

Q3. a) $B^k = \overline{10 \dots 0}^B$ avec k zéros. En effet, les 0 correspondront aux puissances $B^0, B^1, \dots, B^{k-1}$, et le 1 correspondra alors à B^k .

b) $B^k - 1 = \overline{(B-1) \dots (B-1)}^B$ avec k $(B-1)$.

Q4. a) 23

b) 230

c) 2300

d) $230 \dots 0$ avec k zéros

e) 1021

f) 102

g) 10

h) 1

i) 1

j) 1112

k) 111

l) 11

m) 1

n) 2

Q5. a) $n \bmod B = u_0$. En effet :

$$\begin{aligned} n &= \sum_{i=0}^{l-1} u_i B^i \\ &= B(\sum_{i=1}^{l-1} u_i B^{i-1}) + u_0 \end{aligned}$$

et $u_0 \in \llbracket 0, B-1 \rrbracket$, donc l'égalité ci-dessus est la division euclidienne de n par B , le reste est donc u_0 , et c'est aussi $n \bmod B$.

- b) La même preuve montre que $\lfloor \frac{n}{B} \rfloor = \overline{u_{l-1} \dots u_0}^B$, car le quotient de la division de n par B est

$$\sum_{i=1}^{l-1} u_i B^{i-1}$$

ce qui vaut bien $\overline{u_{l-1} \dots u_0}^B$.

- c) Diviser n par B^2 revient à supprimer les deux premiers chiffres, et à décaler le reste :

$$\lfloor \frac{n}{B^2} \rfloor = \overline{u_{l-1} \dots u_2}^B$$

- d) De manière générale, diviser n par B^k supprime les k premiers chiffres, et décale le reste, ce qui met le chiffre u_k au niveau des unités :

$$\lfloor \frac{n}{B^k} \rfloor = \overline{u_{l-1} \dots u_k}^B$$

En effet, coupons l'écriture de n en base B en deux :

$$\begin{aligned} n &= \sum_{i=0}^{l-1} u_i B^i \\ &= \sum_{i=0}^{k-1} u_i B^i + \sum_{i=k}^{l-1} u_i B^i \\ &= \sum_{i=0}^{k-1} u_i B^i + B^k \sum_{i=k}^{l-1} u_i B^{i-k} \end{aligned}$$

On reconnaît la division euclidienne de n par B^k , mais il faut montrer que $\sum_{i=0}^{k-1} u_i B^i < B^k$ pour que ce soit bien le reste euclidien. Or, chaque chiffre est inférieur à $B - 1$, d'où :

$$\begin{aligned} \sum_{i=0}^{k-1} u_i B^i &\leq \sum_{i=0}^{k-1} (B-1) B^i \\ &\leq (B-1) \sum_{i=0}^{k-1} B^i \\ &\leq (B-1) \frac{B^k - 1}{B-1} \\ &\leq B^k - 1 \end{aligned}$$

Donc c'est bien le reste euclidien. En identifiant le quotient, on a

$$\lfloor \frac{n}{B^k} \rfloor = \sum_{i=k}^{l-1} u_i B^{i-k}$$

soit $\lfloor \frac{n}{B^k} \rfloor = \overline{u_{l-1} \dots u_k}^B$

- e) La question précédente montre aussi que

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

soit $n \bmod B^k = \overline{u_{k-1} \dots u_0}^B$.

Q6. En combinant les formules de la question précédente, on peut :

1. Partir de $n = \overline{u_{l-1} \dots u_0}^B$
2. Décaler de k chiffres pour obtenir $\lfloor \frac{n}{B^k} \rfloor = \overline{u_{l-1} \dots u_k}^B$
3. Extraire l'unité et avoir $\lfloor \frac{n}{B^k} \rfloor \bmod B = u_k$

Exercice 2.

Q1. $\overline{5a}^{16} = 5 \times 16 + 10 = 170$

Q2. $172 = 10 \times 16 + 12$ donc $172 = \overline{ac}^{16}$

Q3. Un octet contient 8 bits, donc 2 blocs de 4 bits, donc 2 chiffres hexadécimaux.

Q4. Un entier 32 bits contient 4 octets, donc 8 chiffres hexadécimaux.

Q5. Tableau rempli :

Dec	15	735	493	2657	3404
Bin	1111	0010 1101 1111	0001 1110 1101	1010 0110 0001	1101 0100 1100
Hex	f	2df	1ed	a61	d4c

Exercice 3.

Exercice 4.

Q1. Le principe de cet algorithme est de simplement calculer la somme

$$\sum_{j=0}^{l-1} a_j B^j$$

terme par terme. Ainsi, on a l'invariant suivant :

$$n = \sum_{j=0}^{i-1} a_j B^j = \overline{a_{i-1} \dots a_0}^B$$

ce qui exprime qu'à chaque passage sur la condition de boucle, on a lu les chiffres a_0 jusqu'à a_{i-1} .

On ne détaille pas la preuve ici, mais elle n'est pas compliquée : à chaque tour on incrémente i et on ajoute le terme de la somme manquant.

En supposant que l'on a bien montré l'invariant annoncé, on voit que la boucle termine puisque c'est en réalité une boucle "pour i allant de 0 à $l-1$ ". En sortie de boucle, on a $i = l-1$ et l'invariant dit alors précisément que $n = \overline{a_{l-1} \dots a_0}^B$, d'où la correction.

Q2. A chaque passage de boucle, l'algorithme calcule B^i , ce qui demande $i-1$ multiplications (ou 0 multiplications pour $i=0$), puis multiplie le résultat avec a_i (ce que l'on ne comptera pas comme une multiplication pour $i=0$). L'algorithme fait donc i multiplications à chaque passage, soit au total $0 + 1 + \dots + l-1 = \frac{l(l-1)}{2}$.

Q3. On modifie l'algorithme en suivant ce que propose la question :

On voit que l'algorithme ne fait plus que 2 multiplications par tour. Ce nouvel algorithme demande alors seulement $2l$ multiplications (et le premier tour ne fait que des multiplications par 1) !

Q4. En faisant tourner l'algorithme sur des exemples, on voit que l'algorithme de Horner consiste à construire le nombre lu en partant des chiffres de poids forts :

- Après le premier passage, n vaut a_{l-1} ,
- Après le deuxième passage, n vaut $a_{l-1}B + a_{l-2} = \overline{a_{l-1}a_{l-2}}^B$,
- Après le troisième passage, n vaut $\overline{a_{l-1}a_{l-2}a_{l-3}}^B$

```

1   $n \leftarrow 0;$ 
2   $l \leftarrow 0;$ 
3   $p \leftarrow 1;$ 
4  tant que  $i < l$  faire
5  |    $n \leftarrow n + a_i \times p;$ 
6  |    $p \leftarrow B \times p;$ 
7  |    $i \leftarrow i + 1;$ 

```

On peut donc essayer de montrer l'invariant suivant :

$$n = \overline{a_{l-1} \dots a_{i+1}}^B$$

On n'inclus pas a_i dans l'invariant car au début du tour, on n'a pas encore regardé a_i .

A nouveau, la preuve n'est pas compliquée, on vérifie l'hérédité sans difficulté en posant directement les calculs.

Ainsi, en fin de boucle, on a $i = -1$, l'invariant donne alors directement la correction.

Cet algorithme fait une seule multiplication par tour de boucle (et pour $i = l - 1$, on fait une multiplication par 1 que l'on ne compte pas). On a donc au total $l - 1$ multiplications.

Exercice 5.

Q1. 1. $100 \& 52 = 36$

2. $31 \& 64 = 0$

3. $95 | 44 = 127$

4. $31 | 64 = 95$

5. $\sim 216 = 39$

6. $(\sim 100) | 100 = 255$

Q2. Soit $x \in \mathbb{N}_l$, et $a_{l-1} \dots a_0$ ses bits. Notons $b_{l-1} \dots b_0$ les bits de $x | \sim x$. Pour $i \in \llbracket 0, l-1 \rrbracket$, on a $b_i = a_i | (1 - a_i)$. Qu'on ait $a_i = 0$ ou 1, on a forcément soit $a_i = 1$ soit $(1 - a_i) = 1$. Ainsi, $b_i = 1$, et ce pour tout i . Donc $x | \sim x = \overline{11 \dots 1}^2 = 2^l - 1$

Q3. De même, $x \& \sim x = 0$, car pour chaque position de bit i , soit $a_i = 0$ soit $(1 - a_i) = 0$.

Q4. $x \& y$ est un nombre dont les bits à 1 sont ceux qui étaient à 1 dans x ET dans y . Autrement dit, en notant $x = \overline{a_{l-1} \dots a_0}^2$, $y = \overline{b_{l-1} \dots b_0}^2$, $x \& y = \overline{c_{l-1} \dots c_0}^2$, on a, pour tout $i \in \llbracket 0, l-1 \rrbracket$, $c_i = a_i \& b_i \leq a_i$, et donc en sommant $x \& y \leq x$. De même, on a $x \& y \leq y$. D'où $x \& y \leq \min(x, y)$.

Pour qu'il y ait égalité, il faut que l'un des nombres soit "inclus" dans l'autre, par exemple $x = 5 = 00000101$ et $y = 11010111$: tous les bits de x qui sont à 1 le sont dans y , donc $x \& y = x = 5$.

Un cas où l'inégalité est stricte : $x = 4 = 00000100$ et $y = 35 = 00100001$: aucun bit en commun, donc $x \& y = 0$.

Q5. 1. $7 << 3 = 56$

2. $73 << 5 = 32$ (sur 8 bits, les puissances de 2 au dessus de 256 disparaissent).

3. $32 \gg 3 = 4$

4. $55 \gg 2 = 13$

Q6. $a \gg k = \lfloor \frac{a}{2^k} \rfloor$

Q7. $a \ll k = (a \times 2^k) \bmod 2^l$

Q8. On a vu en cours qu'en base B , le i -ème chiffre de n s'obtient avec $\lfloor \frac{n}{B^i} \rfloor \bmod B$. Donc, $a_i = (n \gg i) \& 1$.

Q9. $n' = n \mid (1 \ll i)$ est n où le i -ème bit a été mis à 1.

Q10. En n'utilisant que les trois opérateurs vus en cours, on peut proposer la formule suivante :

$$(n \& \sim (1 \ll i)) \mid (\sim n \& (1 \ll i))$$

Vérifions : posons $n = \overline{a_{l-1} \dots a_0}^2$ et calculons les bits de $n' = (n \& \sim (1 \ll i)) \mid (\sim n \& (1 \ll i))$

— Pour j différent de i :

— le j -ème bit de $(\sim n \& (1 \ll i))$ est 0 (car $1 \ll i$ a un 0 en j).

— le j -ème bit de $(n \& \sim (1 \ll i))$ est a_j (car $\sim (1 \ll i)$ a un 1 en j).

Donc, le j -ème bit de n' est a_j .

— Pour i :

— le i -ème bit de $(\sim n \& (1 \ll i))$ est $1 - a_i$ (car $1 \ll i$ a un 1 en i).

— le i -ème bit de $(n \& \sim (1 \ll i))$ est 0 (car $\sim (1 \ll i)$ a un 0 en i).

Donc, le i -ème bit de n' est $1 - a_i$.

Ainsi, la formule pour n' a été choisie précisément pour laisser les bits autre que i intact, et pour inverser i .

Notons qu'avec l'opérateur XOR, noté $\wedge$, de l'exercice 5, on a une formule plus simple et beaucoup plus immédiate : $n' = n \wedge (1 \ll i)$.

Exercice 6.

Q1. ...

Q2. Il y a plusieurs formules valides. Les deux plus simples sont :

— $a \oplus b = (a \mid b) \& \& (!a \mid !b)$ (au moins l'un est vrai, au moins l'un est faux)

— $a \oplus b = (a \& \& !b) \mid (!a \& \& b)$

Q3. Il faut montrer que pour x, y, z trois booléens, on a $x \oplus (y \oplus z) = (x \oplus y) \oplus z$. On peut remplir une table de valeur, où l'on calcule $x \oplus (y \oplus z)$ et $(x \oplus y) \oplus z$ pour les 8 valeurs possibles de x, y, z , et constater l'égalité.

Une solution un peu plus jolie : on remarque que $\oplus$ est simplement une addition dans $\mathbb{Z}/2\mathbb{Z}$: elle est directement associative.

Q4. a) $9 \wedge 20 = 29$

b) $91 \wedge 107 = 48$

c) $255 \mid 83 = \sim 183 = 72$

Q5. Le calcul du XOR ne change pas peu importe le nombre de bits sous-jacent. Notons que c'est aussi le cas pour le ET et le OU bit à bit, mais pas pour le NON : sur 4 bits, $\sim 0 = \overline{1111}^2 = 15$ mais sur 8 bits, $\sim 0 = \overline{11111111}^2 = 255$.

Q6. a) $n \wedge 0 = n$

b) $n \wedge (2^l - 1) = \sim n$

c) $n \wedge n = 0$: tous les bits s'annulent entre eux.

d) Vu ce qui a été dit à la fin de la question 3, calculer $n \wedge m$ revient à faire une addition en binaire, où l'on oublie toutes les retenues. Donc, on a $n \wedge m = n + m$ si et seulement si aucun bit n'est à 1 à la fois dans n et dans m , si et seulement si $n \& m = 0$. Notons que dans ce cas, on a alors aussi $n | m = n + m$.

e) Notons x_0, y_0 les valeurs initiales de x et y , puis x_i, y_i les valeurs après avoir exécuté la ligne i . On a :

$$\text{--- } x_1 = x_0 \wedge y_0, y_1 = y_0$$

$$\text{--- } y_2 = x_1 \wedge y_1 = x_0 \wedge y_0 \wedge y_0 = x_0, x_2 = x_1 = x_0 \wedge y_0$$

$$\text{--- } x_3 = x_2 \wedge y_2 = x_0 \wedge y_0 \wedge x_0 = y_0, y_3 = y_2 = x_0.$$

Finalement, le code a **échangé** les valeurs de x et y . C'est ce que l'on appelle un **xor-swap** : ça permet d'échanger deux variables sans faire intervenir une troisième variable comme on le ferait normalement :

```
1 int temporaire = x;
2 x = y;
3 y = x;
```