

INF280

Manipulation de bits

Antoine Amarilli

23 juin 2014

Introduction

- Données représentées sous la forme de **bits**
- C++ et Java permettent de les manipuler **directement**
- Parfois plus rapide à **exécuter**
- Parfois plus rapide à **écrire**
- **C++ ici**, la plupart s'applique à Java

Table des matières

1 Introduction

2 Bases

3 Ensembles

4 Nombres

5 Autres

Opérations bit-à-bit

$a \& b$

- **ET** bit-à-bit
- Exemple :
01011101
& 00110101
= 00010101

$a | b$

- **OU** bit-à-bit
- Exemple :
01011101
| 00110101
= 01111101

$a \wedge b$

- **XOR** bit-à-bit
- Exemple :
01011101
^ 00110101
= 01101000

$\sim a$

- **NOT** bit-à-bit
- Exemple :
 \sim 00110101
= 11001010

Décalage

$a \ll i$ ● Décalage vers la **gauche** (poids forts)

● **Complète** avec 0

● Peut **tronquer**

● Exemple :

$$\begin{aligned} & 01011101 \ll 2 \\ & = 01110100 \end{aligned}$$

$a \gg i$ ● Décalage vers la **droite** (poids faibles)

● **Complète** avec 0 (sauf signe, voir plus tard)

● Peut **tronquer**

● Exemple :

$$\begin{aligned} & 01011101 \gg 2 \\ & = 00010111 \end{aligned}$$

Builtins gcc (non portable)

`__builtin_popcount(s)` Nombre de 1

`__builtin_ffs(s)` Index du 1 le plus à droite (à partir de 1)

- `ffs(00000001) = 1`
- `ffs(00000110) = 2`
- `ffs(00000000) = 0`

`__builtin_clz(s)` Index du 1 le plus à gauche (indéfini pour 0)

- `clz(10000001) = 0`
- `clz(01000110) = 1`
- `clz(00000000) = ?`

Pièges

- Attention à la **priorité** !
 - $a \& b == 1$
 - ⇒ $a \& (b == 1)$
- Attention aux **grands shifts** !
 - $a \ll 1337$ est **indéfini** !
- Attention aux **entiers signés** !

Table des matières

1 Introduction

2 Bases

3 Ensembles

4 Nombres

5 Autres

Principe

- Stocker un **petit ensemble** dans les bits d'un entier non-signé
- Le **bit** i est à 1 ssi l'**élément** i est dans l'ensemble
- ⇒ **Énumérer** les ensembles en énumérant les entiers
- ⇒ Plus **compact** que set
- ⇒ Parfois **nécessaire** pour passer en temps/mémoire !

Types

- Dépend de la **taille** de l'ensemble :
 - unsigned long long garantit **64 bits**
 - unsigned long garantit **32 bits**
 - unsigned garantit **16 bits**
- Aussi : uint64_t, uint32_t, uint16_t, uint8_t.

Lecture de l'ensemble

`s & t` intersection

`s | t` union

`s ^ t` différence symétrique

`s & (1 << i)` teste si l'élément i est dans l'ensemble

`__builtin_popcount(s)` nombre d'éléments

`__builtin_ffs(s)` index du premier élément

`__builtin_clz(s)` index du dernier élément

Modification de l'ensemble

`s | (1 << i)` ajouter l'élément i

`s & ~(1 << i)` retirer l'élément i

`s ^ (1 << i)` basculer l'élément i

`s & (s-1)` retirer le plus petit élément

Ensembles plus gros

- `bitset<N>`, **taille fixe**
 - ⇒ remplace les entiers quand 64 ne suffit pas
- `vector<bool>`, **taille variable**
 - ⇒ permet de changer la taille
 - ⇒ overhead de vector
- En **Java** : `BitSet`, taille variable

Sous-ensemble

- Énumérer les **sous-ensembles** de s
 - Soit n le sous-ensemble **courant**
 - **Idée :**
 - $| \sim s$ pour **mettre à 1** les bits inutiles
 - $+1$ pour **propager** une retenue
 - $\& s$ pour **remettre à 0** les bits inutiles
- ⇒ **Prochain sous-ensemble** : $((n | \sim s) + 1) \& s$

Paires

- $a * N + b$ pour encoder une paire
- p/N et $p\%N$ pour décoder la paire
- Évidemment il faut $b < N$
- Attention à la capacité!

(Plutôt arithmétique que bit-à-bit...)

n -uplets

Plus de deux éléments :

```
// encode
long long v = 0;
for (int i = 0; i < n; i++) {
    v *= N;
    v += p[i];
}

// décode
for (int i = n-1; i >= 0; i--) {
    q[i] = v % N;
    v /= N;
}
```


Table des matières

1 Introduction

2 Bases

3 Ensembles

4 Nombres

5 Autres

Complément à deux

- Entier **non signé** de n bits : de 0 à 2^n exclu
- Entier **signé** : représenter des positifs et négatifs
- Premier bit : champ du **signe** (0 pour positif)
- **Positifs** de 0 à 2^{n-1} exclu
- **Négatifs** :
 - De -1 à -2^{n-1} **inclus**
 - Bit de **signe** à 1
 - Autres bits : $2^n - \text{abs}(i)$

⇒ Vrai en pratique mais **non garanti** par le standard C++

Exemple de complément à deux

Valeur	Non-signé	Signé
0111 1111	127	127
0111 1110	126	126
...
0000 0001	1	1
0000 0000	0	0
1111 1111	255	-1
1111 1110	254	-2
...
1000 0001	129	-127
1000 0000	128	-128

Puissances de deux

- Tester si **divisible par 2** : $!(x \ \& \ 1)$
 - **Calculer** 2^i : $(1 \ \ll \ i)$
 - Tester si **puissance de 2** : $x \ \&\& \ !(x \ \& \ x-1)$
 - Doit contenir ≥ 1 bit à 1
 - Doit être **nul** si premier bit à 1 est mis à 0
- ⇒ Doit contenir **exactement** 1 bit à 1

Table des matières

1 Introduction

2 Bases

3 Ensembles

4 Nombres

5 Autres

Code de Gray

- **Énumérer** les valeurs de 0 à 2^n exclu
- **Changer** un seul bit à la fois
- **Cas 1** : 0, 1
- **Cas 2** : 00, 01, 11, 10
- **Récurrance** :
 - **Construire** le code C_{n-1}
 - **Construire** le code miroir $\overline{C_{n-1}}$
 - C'est également un code !
 - C_n^1 est C_{n-1} en préfixant avec un 0
 - C_n^2 est $\overline{C_{n-1}}$ en préfixant avec un 1
 - C_n est $C_n^1 C_n^2$
- **Application** : codeurs rotatifs (suivi des rotations d'un disque)