Erick Gallesio, S. Lavirotte
Polytech Nice Sophia
SI3

TD séances n° X
Manipulation de bits

1 Impression de nombres entiers

Nous allons ici traiter de l'impression des nombres entiers. Dans un premier temps, on imprimera des nombres dans une base quelconque. Ensuite, on s'occupera de l'impression des nombres en binaire.

1.1 Impression en base quelconque

1. Écrire tout d'abord une fonction permettant d'imprimer en base 10 un nombre entier (éventuellement négatif), en utilisant seulement la fonction `putchar`.

2. Généraliser votre fonction pour qu'elle puisse imprimer un nombre dans une base `b` quelconque ($2 \leq b \leq 36$).

1.2 Décomposition binaire

Écrire la fonction `en_binaire` qui affiche en binaire le nombre `n` qui lui est passé en paramètre.

1. Écrire une première version récursive travaillant par divisions successives (cette version affichera `n` avec le nombre minimal de bits nécessaires pour représenter `n`).

2. Écrire une version utilisant les opérateurs sur les bits de C (cette fonction affichera n sur le nombre de bits nécessaire pour représenter un `int` sur votre machine.

2 Grands ensembles

On décide de définir un type abstrait de données permettant de représenter de grands ensembles d'entiers (compris entre 0 et 999). Pour l'implémentation de ce type, on décide de représenter un grand ensemble par un tableau de bits (si un nombre est présent dans l'ensemble, le bit correspondant à ce nombre sera mis à 1, sinon le bit sera à 0). Pour cela, nous avons les définitions suivantes:

```
#define CHAR_SIZE  8                        /* nombre de bits dans un char */
#define MAX_BIGSET 125                      /* nombre de cellules dans un ensemble
*/
#define MAX_VAL    (CHAR_SIZE * MAX_BIGSET)

typedef unsigned char BigSet[MAX_BIGSET];  /* un ensemble dans [0 .. MAX_VAL[  */
```

Ecrire les fonctions suivantes.

- `void BigSet_init(BigSet s) /* initialiser s à l'ensemble vide */`
- `void BigSet_add(BigSet s, int i) /* ajouter i dans s */`
- `int BigSet_is_in(BigSet s, int i) /* 0 si i ∉ s et ≠ 0 sinon */`
- `void BigSet_print(BigSet s) /* afficher les éléments de s */`
- `void BigSet_inter(BigSet s1, BigSet s2, BigSet res) /* range dans res le résultat de l'intersection de s1 et s2 */`

Exemple de code utilisant les fonctions sur les ensembles

```
int main(void) {
  BigSet e1, e2, e3;

  BigSet_init(e1); BigSet_init(e2);

  for (int i = 0; i < 140; i += 12) BigSet_add(e2, i);
  for (int i = 0; i < 140; i += 9) BigSet_add(e1, i);

```

TD séances n° X
Manipulation de bits

```
  BigSet_inter(e1, e2, e3);
  printf("e1 = "); BigSet_print(e1);
  printf("e2 = "); BigSet_print(e2);
  printf("e3 = "); BigSet_print(e3);

  return 0;
}
```

L'exécution du programme précédent devra produire en sortie:

```
e1 = {0, 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99, 108, 117, 126, 135}
e2 = {0, 12, 24, 36, 48, 60, 72, 84, 96, 108, 120, 132}
e3 = {0, 36, 72, 108}
```

3 Chiffrement avec des `xor`

Dans cette question, nous allons utiliser une propriété intéressante du **ou exclusif** (opérateur `^` en C) pour faire du chiffrement réversible. La propriété en question est la suivante: soient deux caractères quelconques `a` et `b`, et `c = a ^ b`, alors `c ^ b == a`. Dans cet exemple, `c` constitue donc la version chiffrée du caractère `a` avec la clé `b`.

Pour retrouver le caractère original `a`, c'est simple: il suffit d'appliquer une nouvelle fois la clé `b` sur le caractère `c` (sommairement, cette propriété peut être résumée à: quels que soient les caractères `a` et `b`, alors `a^b^b == a`)

Nous allons utiliser cette propriété de l'opérateur `^` (**ou exclusif**) dans la fonction `xorcrypt`. Cette fonction renvoie une chaîne ou chaque caractère du message original a été crypté avec le caractère correspondant de la clé (note: si la clé est trop courte, on repart de son premier caractère).

Ainsi, si on veut chiffrer le message `"HELLO, WORLD"` avec la clé `"abcde"`, on obtient la suite de caractères `)'/(*MB4+7-\&` puisqu'on a:

msg	H	E	L	L	O	,		W	O	R	L	D
key	a	b	c	d	e	a	b	c	d	e	a	b
msg ^ key	)	'	/	(	*	M	B	4	+	7	-	&

Écrire la fonction `void xorcrypt(const char msg[], char key[])` qui permet de chiffrer le message `msg` avec la clé `key`.

4 Calcul de x^n (faultatif)

La méthode de calcul $x^n = x * x * … * x$ est peu efficace car elle requiert `n` multiplications (et les multiplications sont des opérations coûteuses).

Une méthode rapide se base sur le fait qu'en mémoire, un entier `n` est codé en binaire sous la forme:

$n \equiv b_k b_{k-1} … b_1 b_0$ (avec $b_i = 0$ ou 1)

et

$n = \Sigma_i 2^i b_i = b_0+2b_1+4b_2+…+2^k b_k$

2 exposant k * b indice k, soit : $2^k b_k$

On exprime donc x^n par $x^n = x^{b_0+2b_1+4b_2+…+2kb_k}$

Cela revient à calculer $xn=x^{b_0} \; . \; x^{2b_1} \; . \; x^{4b_2} \; … \; x^{2kb_k}$

Par exemple, $13=8+4+1=1_3 1_2 0_1 1_0$ en binaire, donc $x^{13}=x^8 \times x^4 \times x^1$.

En introduisant la variable `z` valant successivement x, x^2, x^4, x^8, et ^{y} le produit des « bons » z, on obtient la méthode suivante:

Erick Gallesio, S. Lavirotte
Polytech Nice Sophia
SI3

TD séances n° X
Manipulation de bits

> Si n se décompose en $b_k b_{k-1} \ldots b_0$
> `y` ← `1`
> `z` ← `x`
> pour i ← `0` à `k`: si b_i=`1` alors `y` ← `yz` et `z` ← z^2
> Quand on termine, $y = x^n$

Coder la fonction

```
double Puissance(double x, unsigned int n);
```

qui implémente cette méthode. On pourra utiliser les opérateurs C de manipulation de bits `<<`, `>>`, `&`, `|` …