Polytech Nice Sophia
SI3 / 2021-2022
Erick Gallesio

Programmation Procédurale

Corrigé TD 4

Manipulations de bits

1 Impression de nombres entiers

1.1 Impression en base 10

Pour imprimer le nombre `123`, il faut imprimer le caractère `1` (càd le caractère dont le code est `'0'+ 1`), suivi de du caractère `2` (càd `'0'+ 2`) …

Pour cet exercice, on peut avoir deux types de solution:

- une solution itérative
- une solution récursive

1.1.1 Version itérative

Une première méthode pour imprimer un nombre consiste à trouver les chiffres à afficher par des divisions par 10 successives. Ainsi, pour afficher `123`, on affichera le reste de la division par 10 (⇒ 3) puis on affichera ensuite le quotient de `123` par 10 (⇒ 12).

Le problème avec cette méthode est que l'on obtient les chiffres dans l'ordre inverse. On passe donc par un tableau qui permet de stocker les restes de la division par 10. Ce tableau sera ensuite imprimé à l'envers.

Une solution possible:

```c
void print_iteratif(int n){
  int i, tab[50];           // 50 est largement suffisant (le plus
                            // grand nombre sur 64 bits nécessite 20 digits)

  // Impression du signe éventuel
  if (n < 0) {
    putchar('-');
    n = -n;
  }

  //  Calcul des restes de la division par 10
  i = 0;
  do {
    tab[i++] = n % 10;
    n /= 10;
  }
  while (n);    // ⇔ n != 0

  // Affichage des restes (entiers) avec putchar
  for (--i; i >= 0; i--) {
    putchar('0'+tab[i]);
  }
}
```

Notes:

1. La version donnée traite les nombres négatifs
2. On utilise ici un `do-while` car on aura toujours au moins un chiffre à afficher.

1.1.2 Version récursive

La version récursive permet d'obtenir directement les chiffres dans le bon ordre: en effet pour afficher le nombre 123, il faut d'abord afficher 12 avec une fonction qui sait afficher des nombres ≥ 10 (la fonction que l'on est en train d'écrire sait faire ça!) et ensuite on affiche les unités (ici 3). Si on ne se préoccupe pas du signe, on obtient:

```c
void print_recursif(int n) {
  if (n >= 10) print_recursif(n / 10);
  putchar('0' + n%10);
}
```

Comme on le voit, cette solution est beaucoup plus simple que la version itérative. Si on ajoute le traitement du signe, nous obtenons:

```c
void print_recursif(int n) {
  if (n < 0) {
    putchar('-');
    print_recursif(-n);
  } else {
    if (n >= 10) print_recursif(n / 10);
    putchar('0' + n%10);
  }
}
```

1.2 Impression en base quelconque

Pour imprimer dans une base donnée, il suffit de passer un paramètre supplémentaire pour avoir la base utilisée et de remplacer les divisions par 10 par des divisions par la base.

Le problème qui nous reste à résoudre est comment afficher un digit `d` lorsque `base` > 10 (parfois on utilise des chiffres et parfois des lettres).

En fait on a deux cas, qui sont traités ci-dessous:

```
  if (d < 10)
    putchar('0' + d);           // affichage d'un chiffre
  else
    putchar('A' + d - 10);      // d ≥ 10 ⇒ affichage d'une lettre
```

En utilisant l'opérateur ternaire de C, ce test peut s'écrire:

```
  putchar(d + ((d < 10)? '0': 'A' - 10))
```

Ainsi, la version récursive en base quelconque devient:

```
void print_base(int n, int base) {
  if (n < 0) {
    putchar('-');
    print_base(-n, base);
  } else {
    int d = n %base;
    if (n >= base) print_base(n / base, base);
    putchar(d + ((d < 10)? '0': 'A' - 10));
  }
}
```

Code complet du programme

1.3 Décomposition binaire

Ici les nombres ne sont pas signés. On a donc pas de nombres négatifs.

Pour la **version récursive**, on peut bien sûr s'inspirer de la version `print_base` définie dans l'exercice précédent:

```
void binaire_rec(unsigned int n) {
  if (n > 1)
    binaire_rec(n >> 1);       // ⇔ n / 2
  printf("%d", n & 1);         // ou putchar((n & 1) ? '1' : '0')
}
```

Pour la **version bit à bit**, il faut connaître le nombre maximum de bits utilisés pour représenter un `unsigned int` sur notre implémentation de C. Cela peut s'obtenir par le calcul suivant:

```
   int max = sizeof(unsigned int) * 8;  // 8 car un char occupe 8 bits
```

Ensuite, on positionne le bit de poids le plus fort à 1 (cela correspond au nombre 2^{max}-1) et on travaille en déplaçant ce bit à droite à chaque tour de boucle. L'inconvénient de cette version est que l'on a un résultat qui laisse les '0' de tête (mais cela peut se corriger facilement et est laissé en exercice):

```
void binaire_bit_a_bit(unsigned int n) {
  // nombre max de bits dans un int: (sizeof(unsigned int) * 8)
  unsigned masque = 1 << ((sizeof(unsigned int) * 8) - 1);  //  2**max-1

  while (masque > 0) {
    printf("%d", (masque & n) != 0);
    masque >>= 1;
  }
}
```

Code complet du programme

2 Grands ensembles

Pour cet exercice, la seule vraie difficulté est de trouver l'endroit où se trouve le bit qui nous intéresse (pour le tester ou pour le mettre à 1). En fait, on peut trouver ce bit avec le quotient et le reste de la division par 8.

Exemple:

Supposons que l'on veuille ajouter 12 à un ensemble E. Pour cela, il faut d'abord trouver la case du tableau où se trouve le bit correspondant à l'entier 12. La division entière `12/8` (résultat 1), nous indique que 12 est dans la case d'indice 1 du tableau. Le reste de la division (`12%8 ⇒ 4`) nous donne la position du bit dans cet octet. Nous devons donc mettre à 1 le bit n° 4 du caractère situé dans la case 1 du tableau représentant l'ensemble.

Si la case d'indice 1 contenait déjà la valeur `00100100` , pour mettre le bit n° 4 à 1, il suffit d'appliquer un OR (bit à bit) avec la valeur `1<<4` (c'est à dire `00010000`). En effet nous avons:

```
   00100100
OR 00010000
   --------
 = 00110100
```

De la même façon, pour tester si la valeur 12 est dans l'ensemble E, il suffit de tester si le $4^{ème}$ bit de `E[1]` est à 1. Pour cela, l'opération que nous devons utiliser est un AND (bit à bit) avec la valeur `1<<4` . En effet, nous avons:

```
    001X0100
AND 00010000
    --------
  = 000X0000
```

Ici, si X est égal à 0, la valeur du AND sera égale à 0 (qui, on le rappelle, veut dire *faux* en C) et si X est égal à 1, le résultat est `1<<4` , qui est différent de 0 (et donc *vrai*).

Donc pour résumer,

- ajouter l'élément 12 dans l'ensemble E: `E[1] = E[1] | (1<<4);`
- tester si l'élément 12 est dans l'ensemble: `if (E[1] & (1<<4)) ... else ...`

L'écriture de l'ajout d'un élément dans l'ensemble est donc:

```
void BIGSET_add(BIGSET bs, int i){
  int slot = i / CHAR_SIZE;  // Trouver l'indice de l'octet à modifier
  int num  = i % CHAR_SIZE;  // Trouver le bit à modifier

  bs[slot] |= 1 << num;      // ⇔ à bs[slot] = bs[slot] | (1 << num)
}
```

De la même façon si on veut tester si l'entier `i` est présent dans l'ensemble `bs` , nous devons:

1. calculer la position du bit qui nous intéresse avec le quotient et le reste de la division entière de `i` par `CHAR_SIZE` (comme précédemment)
2. tester si le bit `num` est à 1 dans la case `bs[slot]` . Pour ce test, nous allons ici utiliser un AND (bit à bit).

   Si la la case `bs[slot]` était égale à `00100100` et que `num` est égal à 4, nous avons:

```
     00100100
AND  00010000
     --------
   = 00000000
```

Ceci indique que le bit n° 4 n'est pas égal à 1 dans l'élément `slot` du tableau. Bien sûr si ce bit était présent, le AND précédent serait égal à `00010000` (càd `1<<numm`).

La fonction demandée peut donc être implémentée de la façon suivante.

```
int BigSet_is_in(BigSet bs, int i) {
 int slot = i / CHAR_SIZE;
 int num  = i % CHAR_SIZE;

 return bs[slot] & (1 << num);
}
```

Remarque:

Cette fonction renvoie 0 si l'élément n'est pas dans l'ensemble et une valeur différente de 0 sinon. Si on voulait renvoyer 0 et 1 seulement, on aurait pu écrire

```
return (bs[slot] & (1 << num))) != 0;
```

En général, le `return` donné dans la fonction précédente suffit puisque, en C, les tests sont vrais si la valeur testée est différente de 0. Ainsi, pour tester, si `i` est dans `bs`, on écrira:

```
if (BigSet_is_in(i, bs)) {    // ⇔ le resultat de BigSet_is_in est ≠ 0
  ...
}
```

Intersection et Union

Pour construire l'intersection de deux ensembles nous pourrions avoir l'écriture (*«naturelle»*) suivante:

```
void BigSet_inter(BigSet s1, BigSet s2, BigSet res) {
  BigSet_init(res);                              // forcer init de res = ∅

  for (int i = 0; i < MAX_VAL; i++)
    if (BigSet_is_in(s1, i) && BigSet_is_in(s2, i))  // si (i ∈ s1) ∧ (i ∈ s2)
      BigSet_add(res, i);                          //   res = res ∪ {i}
}
```

Cette boucle s'exécute `MAX_VAL` fois et fait un test asez compliqué à chaque fois (noter au passage que l'on calcule 3 fois la même position du bit à chaque tour de boucle).

Une version plus efficace pour calculer l'intersection de deux ensembles consiste à utiliser un **and bit à bit** sur des «paquets» de 8 valeurs. Ce sera bien plus efficace que de travailler sur les entiers un par un (puisqu'ici, en quelque sorte, on fait un test d'appartenance de 8 valeurs en une seule fois). Par ailleurs, cette boucle n'aura qu'à s'exécuter `MAX_SET` fois. Le code de l'intersection est donc:

```
void BigSet_inter(BigSet s1, BigSet s2, BigSet res) {
  BigSet_init(res);

  for (int i = 0; i < MAX_BIGSET; i++)
    res[i] = s1[i] & s2[i];
}
```

Code complet pour les grands ensembles