Programmation Avancée

TP n°3 : Représentation binaire

Simon Forest

28 janvier 2021

Exercice 1 : Décomposition binaire

1. Écrire une fonction qui prend en argument un `unsigned int` et en affiche la décomposition en base 2 en convention gros-boutiste. Faire la version itérative et récursive (faire cela sans regarder le cours si possible).
2. Comment tester si le $k$ème bit de la représentation binaire de l'entier `n` est à 1 sans utiliser les opérations sur les bits vues en cours ?
3. Comment tester si le $k$ème bit de la représentation binaire de l'entier `n` est à 1 en utilisant deux opérations sur les bits ?
4. En déduire un algorithme itératif permettant d'afficher la décomposition en base 2 en convention gros-boutiste sans utiliser de tableau ou de structure comme pour la version vue en cours.

Exercice 2 : Addition et multiplication

Dans cet exercice, on définit une structure permettant de représenter des nombres comme séquences d'un nombre fixé de bits, et on écrit les opérations d'addition et de multiplication pour cette représentation.

1. Dans un fichier `.h`, déclarer une structure `nombre` contenant un unique champ qui est un tableau statique de `unsigned char` de taille fixe égale à une constante `NB_BITS` introduite par une directive `#define`. Dans la suite, il ne faudra faire référence qu'à `NB_BITS` et pas sa valeur effective.
2. Dans un fichier `.c`, définir une fonction `from_uint_to_nombre` calculant la représentation d'un `unsigned int` en un `nombre`. Définir une fonction `from_nombre_to_uint` faisant la conversion dans l'autre sens (on ne considérera pour cela que les 32 premiers « chiffres » d'un `nombre`). La convention utilisée pour la représentation devra être *little endian*.
3. Définir une fonction `somme_nombre` calculant la somme de deux `nombre`. Cette fonction devra implémenter l'algorithme standard de l'addition vu en cours. Tester que votre addition fonctionne en utilisant les fonctions de conversion pour comparer à l'addition de `unsigned int`.
4. Définir une fonction `decaler_nombre` qui décale les « chiffres » d'un `nombre` de k cases vers la droite (rappel : on est en *little endian*) en remplissant par des 0.
5. Sur la base des deux fonctions précédentes, définir une fonction `mult_nombre` calculant la multiplication de deux `nombre`. Cette fonction implémentera l'algorithme classique pour faire la multiplication. Tester cette fonction sur des exemples comme pour `somme_nombre`.

Exercice 3 : Calcul du bit de poids fort

Le bit de poids fort (*most signifiant bit*, ou *MSB*), est le bit à 1 le plus à droite dans la représentation d'un entier `x` en utilisant une convention petit-boutiste. On s'intéresse à calculer efficacement le rang k de ce bit.

1. Écrire un algorithme simple permettant de faire ce calcul. Quelle est sa complexité (On suppose que le nombre N de bits du type que l'on utilise (`char`, `int`, *etc.*) est laissé en paramètre) ?
2. En utilisant les opérations bit-à-bit, comment décider efficacement si k est inférieur à une valeur l donnée en argument ?
3. En déduire un algorithme plus efficace pour calculer k. Quelle est sa complexité ?
4. Comment adapter l'algorithme pour calculer efficacement le rang du bit de poids faible (plus petit bit à 1 dans la représentation binaire) ?

Exercice 4 : Ouverture de fichiers par *flags*

La fonction `fopen` nécessite de passer un mode d'ouverture comme argument sous la forme d'une chaîne de caractères. Cela n'est pas pratique car cela ne permet pas de simplement combiner les options définissant le mode. Dans cet exercice, on écrit un *wrapper* autour de `fopen` permettant d'utiliser des *flags* pour pour la librairie SDL.

1. Définir avec des directives `#define` des constantes représentant les options suivantes :
   - une option `MYFOPEN_READ`, permettant d'ouvrir un fichier en lecture,
   - une option `MYFOPEN_WRITE`, permettant d'ouvrir un fichier en écrite,
   - une option `MYFOPEN_BINARY`, permettant d'ouvrir un fichier en mode binaire.

   Chaque constante devra avoir une représentation binaire avec un seul bit à 1 et ce bit devra être différent pour chaque option.
2. Un jeu d'options est alors obtenu en combinant zero ou plusieurs des options ci-dessus avec l'opérateur `|`. Comment tester si une option particulière a été activée à partir de l'entier représentant le jeu d'options ?
3. Écrire une fonction `FILE* myfopen(const char* filename, unsigned int flags)` permettant d'ouvrir un fichier en spécifiant un mode d'ouverture en utilisant les *flags* définis plus haut. Cette fonction utilisera `fopen` pour faire l'ouverture finale, une fois que le jeu d'options aura été traduit dans la chaîne de caractères correspondante.

Exercice 5 : Échange avec xor

Vous avez sûrement vu comment échanger le contenu de deux variables (par exemple des `int`) en en utilisant une troisième (sinon, comment on fait ?). Dans cet exercice, on va montrer qu'il est possible de faire un tel échange sans introduire de troisième variable. Pour cela, on va utiliser le ou-exclusif (xor en anglais), qui est calculé par l'opérateur `^` en `C`. Dans la suite, on suppose que `x` et `y` sont deux `int`.

1. Exprimer simplement la valeur de `(x ^ y) ^ y` et `(x ^ y) ^ x`.
2. En déduire une façon d'échanger `x` et `y` sans introduire de nouvelle variable.

Exercice 6 : Hashage par xor

Pour s'assurer qu'un fichier a été transmis ou conservé correctement, on en calcule un *hash* que l'on compare à la valeur que ce *hash* est censée avoir. Ici, on conçoit une méthode simple basée sur l'opération xor. Le *hash* calculé est la valeur `h = x1 ^ ... ^ xk` où `x1`,...,`xk` sont tous les octets du fichiers, numérotés de 1 à k.

1. Écrire un programme prenant en entrée un fichier (sur la ligne de commande) et en affiche le *hash* calculé par la méthode ci-dessus. On pourra utiliser le code d'affichage écrit pour l'exercice 1.
2. Tester le programme sur un fichier texte. Vérifier que le *hash* est différent si un caractère du fichier est modifié. Peut-on changer deux caractères et que le *hash* reste le même ?
3. Au lieu de ne produire *in fine* qu'un seul octet, on souhaite faire une fonction de hashage basée sur xor qui produise n octets à la fin. Comment modifier la méthode de calcul de *hash* et le code pour cela ?