DM - Épisode 2

Voici la deuxième partie du DM. Nous allons y voir un certain nombre d'algorithmes visant à synthétiser un code efficace de multiplication par une constante. Vos travaux doivent être envoyés à ***sylvain.chevillard@ens-lyon.fr*** *au plus tard le 19 avril 2009 à 23h59. Les retards seront sanctionnés, tout comme pour la première partie du DM. Les mêmes consignes générales que pour la première partie s'appliquent. N'hésitez pas à structurer votre code en écrivant des fonctions intermédiaires qui vous sembleraient pertinentes.*

Rappel des faits : M est une constante et on essaie d'écrire un programme `MulByM` en C qui imlémente la fonction $x \mapsto Mx$ en n'utilisant que des additions/soustractions et des décalages. Dans la première partie du DM, vous avez implanté les deux méthodes les plus simples, qui s'appuient directement sur l'écriture binaire ou le recodage de Booth.
Ces méthodes ne sont évidemment pas les plus efficaces. Par exemple, le nombre 165 (qui s'écrit 10100101 en binaire) peut être calculé en calculant d'abord 5 (101 en binaire) puis en écrivant $165 = 5 + (5 \cdot 2^5)$. Ainsi, un programme de multiplication par 165 possible est :

```
#include <gmp.h>

void MulBy165(mpz_t result, mpz_t x) {
  mpz_t x5;
  mpz_init(x5);

  mpz_set(x5, x);           /* x5 = x  */
  mpz_mul_2exp(x5, x, 2);   /* x5 = 4x */
  mpz_add(x5, x5, x);       /* x5 = 5x */

  mpz_set(result, x5);              /* result = 5x */
  mpz_mul_2exp(result, result, 5);  /* result = 160x */
  mpz_add(result, result, x5);      /* result = 165x */

  mpz_clear(x5);
  return;
}
```

Nous avons ici exploité le fait que 165 s'écrit $2^k c + c$ (donc $165x = 2^k(cx) + (cx)$) et que cx peut être stocké dans une variable intermédiaire.
Ce principe conduit à un algorithme connu sous le nom *d'algorithme de Bernstein* :
- si $M = 1$, il n'y a rien à faire ;
- si M est pair, $M = 2^k M'$ où M' est impair. Appeler l'algorithme de Bernstein pour générer un code de multiplication par M'. En déduire un code de multiplication par M par simple ajout d'un shift ;
- si M est impair, on peut essayer d'exprimer M sous l'une des formes suivantes :
  a) $M = M' + 1$;
  b) $M = M' - 1$;
  c) $M = 2^k M' + M'$;
  d) $M = 2^k M' - M'$.
  Essayer l'algorithme de Bernstein sur chacune de ces formes (il peut y avoir plusieurs valeur de k possibles), voir laquelle est la plus efficace, et l'utiliser pour en déduire un code de multiplication par M.

Il ne faut pas implémenter l'algorithme de Bernstein tel quel ; pour le rendre un peu plus efficace, il faut lui adjoindre une stratégie *branch and bound*. Si par exemple on a déjà déterminé qu'on peut calculer M par la méthode $M = M' + 1$ en n additions, et que, au cours de l'exploration de la méthode $M = 2^k M' + M'$ on dépasse n additions, il n'est pas nécessaire d'explorer cette piste plus longtemps. Ceci permet de couper des branches de l'arbre d'exploration, en maintenant pour chaque entier M un entier n_M qui indique qu'on connaît déjà une façon de multiplier par M en moins de n_M additions.

Question 1.
Écrire un programme `AlgoBernstein` qui prend en argument un grand entier M et synthétise un programme C de multiplication par M, en utilisant l'algorithme de Bernstein.

Algorithme de Lefèvre
L'algorithme de Bernstein n'est pas adapté pour les entiers M de grande taille, du fait de sa complexité. V. Lefèvre a proposé en 1999[1] un algorithme glouton mais polynomial pour traiter la multiplication par une constante M. En voici le principe : on regarde le recodage de booth modifié de M et on cherche un motif dedans. Par exemple, si $M = 20\,061$ ($M = 10100\bar{1}010\bar{1}00\bar{1}01$ après recodage), on peut remarquer que le motif $1000000010\bar{1}$ apparaît deux fois (l'une fois directement, et l'autre fois sous sa forme complémentée $\bar{1}000000\bar{1}01$) de façon entremêlée, de sorte que

$$\begin{array}{rr} & 1000000010\bar{1} \\ - & \quad 1000000010\bar{1} \\ + & 1000000000000 \\ \hline & 10100\bar{1}010\bar{1}00\bar{1}01 \end{array}$$

Plus précisément, l'algorithme repose sur l'idée suivante : pour toute distance d, on cherche :
a) le nombre d'occurrences de la sous-chaîne $(1 \underbrace{\star\cdots\star}_{d-1 \text{ chiffres}} 1)$ ou $(\bar{1} \underbrace{\star\cdots\star}_{d-1 \text{ chiffres}} \bar{1})$ dans la chaîne des bits du recodage de Booth de M ;
b) le nombre d'occurrences de la sous-chaine $(1 \underbrace{\star\cdots\star}_{d-1 \text{ chiffres}} \bar{1})$ ou $(\bar{1} \underbrace{\star\cdots\star}_{d-1 \text{ chiffres}} 1)$.
Dans l'exemple précédent, pour $d = 5$, il y a 3 occurrences du cas b) : deux occurrences de $10000\bar{1}$ et une occurrence de $\bar{1}00001$.

$$\underbrace{1}_{}0\underbrace{100\bar{1}}_{}0\underbrace{1}_{}0\overbrace{\bar{1}00\bar{1}}^{}01$$

Nous en déduisons un motif contenant 3 bits non nuls qui se répète deux fois. Remarque : si le premier bit d'une occurrence est le même que le dernier bit d'une autre occurrence, ça ne marche pas. Toujours dans notre exemple, pour $d = 2$, il y a 3 occurrences du cas b) : deux occurrences de $\bar{1}01$ et une de $10\bar{1}$. Mais deux d'entre elles partagent un 1 en commun.

$$\begin{array}{c} \overset{\text{(ii)}}{\overbrace{}} \\ 10100\underbrace{\bar{1}}_{\text{(i)}}0\underbrace{1}_{}0\bar{1}00\underbrace{\bar{1}}_{}01 \end{array}$$

[1] *Multiplication by an Integer Constant*, rapport de recherche du LIP n°1999-06, Ecole Normale Supérieure de Lyon

Il faut donc laisser tomber une des deux occurrences et on obtient donc un motif avec 2 bits non nuls seulemement : $\bar{1}000000\bar{1}$ si on garde l'occurrence (i) ou $10000\bar{1}$ si on garde l'occurrence (ii).
L'algorithme de Lefèvre consiste à utiliser cette technique pour trouver un motif ayant un nombre de bits non nuls maximal. Ceci conduit permet d'écrire M sous la forme

$$M = \pm(M' \pm 2^k M') + R$$

où M' est le nombre correspondant au motif et R correspond à tout ce qui reste. Dans notre exemple, on a $M' = 515$ et $R = 4096$ et $M = (2^5 M' - M') + R$. L'algorithme est appelé récursivement sur M' et R. Lorsqu'aucun motif n'apparaît, la méthode de base par addition/soustraction et décalages est appliquée.

Question 2.
Montrer que l'algorithme de Lefèvre est polynomial en la taille de M (c'est-à-dire la taille de son développement binaire).

Question 3.
Écrire un programme `AlgoLefevre1` qui synthétise un programme C de multiplication par M en utilisant l'algorithme de Lefèvre. Lorsque vous avez le choix entre plusieurs motifs ayant le même nombre de bits non nuls, vous êtes libres d'appliquer l'heuristique qui vous semblera la plus appropriée.

Algorithme de Lefèvre amélioré.
L'algorithme de Lefèvre a un défaut important : si un motif apparaît à la fois dans M' et dans R, l'algorithme ne s'en apercevra pas et le calcul de ce motif sera fait deux fois.
V. Lefèvre a donc amélioré son algorithme[2] en le généralisant. On ne cherche plus à calculer simplement Mx mais un ensemble de produits $M_1x, \ldots, M_nx$. On recherche désormais un motif qui soit présent dans deux constantes M_i et M_j. Bien sûr, on n'exclut pas le cas $i = j$ qui correspond à chercher un motif présent deux fois dans une même constante M_i, comme on le faisait dans l'algorithme précédent.
Nous définissons la distance entre deux chiffres d'un même nombre comme précédemment : par exemple, dans 10010000, le deux 1 sont à distance $d = 3$. Remarque : le premier 1 correspond à 2^7 et le second à 2^4 ; la distance entre deux bits est donc la différence de leur poids respectifs.
Pour deux nombres différents M_i et M_j, on définit aussi la distance entre deux chiffres comme la différence de leur poids respectifs dans les deux nombres. Par exemple, si $M_i = 10\bar{1}01$ et $M_j = \bar{1}0$, la distance entre le $\bar{1}$ de M_i (qui correspond à -2^2) et le $\bar{1}$ de M_j (qui correspond à -2^1) est 1. La distance dépend de l'ordre dans lequel on considère les deux nombres. Dans l'exemple précédent, la distance entre le $\bar{1}$ de M_j et le $\bar{1}$ de M_i est -1.
L'algorithme de Lefèvre devient donc le suivant : pour tout couple $i \leq j$, et toute distance d (d pouvant être négatif ou nul, sauf si $i = j$), on compte le nombre d'occurrences de couples $(1, 1)$ ou $(\bar{1}, \bar{1})$ à distance d entre M_i et M_j (cas a)) et le nombre d'occurrences de couples $(1, \bar{1})$ ou $(\bar{1}, 1)$ (cas b)). On en déduit un motif possèdant un nombre de bits non nul maximal, présent à la fois dans un nombre M_i et un nombre M_j (possiblement $i = j$).
On peut donc écrire $M_i = \pm R_i \pm 2^{k_i} N$ et $M_j = \pm R_j \pm 2^{k_j} N$ (ou bien $M_i = (\pm N \pm 2^{k_i} N) + R_i$ dans le cas $i = j$). On enlève alors M_i et M_j de l'ensemble des M_k, et on rajoute R_i R_j et N à la liste et on s'appelle récursivement.

[2] *Multiplication by an Integer Constant*, rapport de recherche n°4192, LORIA, Nancy. Mai 2001

Si aucun motif ayant au moins deux bits non nuls n'est trouvé, on calcule tous les $M_i x$ par l'algorithme d'additions/soustractions et décalage.

Question 4.
Écrire un programme `AlgoLefevre2` qui synthétise un programme C de multiplication par M en utilisant l'algorithme de Lefèvre amélioré.

Question 5.
Appliquer chacun des algorithmes que vous avez programmés (`AlgoNaif`, `AlgoBooth`, `AlgoBernstein`, `AlgoLefevre1` et `AlgoLefevre2`) aux constantes suivantes et donner le nombre d'additions/soustractions nécessaires :

$$M = 45$$
$$M = 113$$
$$M = 341$$
$$M = 861$$
$$M = 1717$$
$$M = 20061$$
$$M = 543413$$
$$M = 47804853381$$
$$M = 799144290325165979$$
$$M = 905502432259640355$$
$$M = 391745696951556693$$

Question 6.
Proposer des pistes pour améliorer les algorithmes vus dans ce DM, voire proposer d'autres idées d'algorithmes pour effectuer efficacement la multiplication par une constante. Si vous avez le temps, implémentez vos idées et comparez leur performances avec les autres algorithmes.