Multiplication par une constante entière

Vincent Lefèvre

Juin 2001

Introduction

But : générer du code optimal à l'aide d'opérations élémentaires (décalages vers la gauche, additions, soustractions).

Utile pour :

- la compilation, quand le processeur n'a pas d'instruction `MUL` ou si celle-ci est très lente (constante : au plus 32 ou 64 bits) ;
- un certain type de calcul matriciel, e.g. algo du style Toom-Cook pour la multiplication de grands entiers ;
- le calcul approché des valeurs successives d'un polynôme (constante : quelques centaines de bits).

Exemples

Calcul de $1997x$.

$$\begin{aligned} 17x &\leftarrow (x \ll 4) + x \\ 51x &\leftarrow (17x \ll 2) - 17x \\ 1997x &\leftarrow (x \ll 11) - 51x \end{aligned}$$

Calcul de $2001x$.

$$\begin{aligned} 2049x &\leftarrow (x \ll 11) + x \\ 3x &\leftarrow (x \ll 2) - x \\ 2001x &\leftarrow 2049x - (3x \ll 4) \end{aligned}$$

Formulation du problème

On se donne un entier positif impair n (notre constante).

On cherche une suite d'entiers positifs u_0, u_1, u_2, ..., u_q telle que :

- valeur initiale : $u_0 = 1$;
- pour $i > 0$, $u_i = |s_i\, u_j + 2^{\,c_i}\, u_k|$, avec

$$j < i, \quad k < i, \quad s_i \in \{-1, 0, 1\}, \quad c_i \in \mathbb{N};$$

- valeur finale : $u_q = n$.

$\longrightarrow$ calcul des $u_i\, x$ successifs.

Algorithme qui détermine une suite minimale ?

Complexité du problème : inconnue. NP-complet ?

Heuristiques ?

Note : si on se restreint à $s_i = 1$ et $c_i = 0$, c'est le problème des chaînes d'additions pour calculer une puissance (Knuth, vol. 2).

En pratique : décalages retardés.
$\rightarrow$ Cas n pair non traité.
$\rightarrow$ On choisira toujours $s_i \neq 0$.

Avec nos heuristiques : $c_i \neq 0$.
$\rightarrow \forall i,\ u_i$ impair.

Méthode binaire

Exemple : $113x$. $113 = 1110001_2$

$$\begin{array}{rcl} 3x & \leftarrow & (x \ll 1) + x \\ 7x & \leftarrow & (3x \ll 1) + x \\ 113x & \leftarrow & (7x \ll 4) + x \end{array}$$

Nombre d'opérations = (nombre de 1) − 1.

Si m est la taille de la constante ($m = \lfloor \log_2(n) \rfloor + 1$), on a $q_{\mathrm{av}} \sim m/2$.

Amélioration : chiffres signés et recodage de Booth.

On utilise la transformation

$$0\,\underbrace{1111\ldots1111}_{k\text{ chiffres}} \to 1\ \underbrace{0000\ldots000}_{k-1\text{ chiffres}}\ \bar{1}.$$

basée sur la formule : $2^{k-1}+2^{k-2}+\ldots+2^2+2^1+2^0=2^k-1$.

$113 = 1110001_2 = 100\bar{1}0001_2$ après recodage.

$$\begin{aligned} 7x &\leftarrow (x \ll 3) - x \\ 113x &\leftarrow (7x \ll 4) + x \end{aligned}$$

2 opérations au lieu de 3.

On peut montrer que $q_{\mathrm{av}} \sim m/3$.

Algorithme de Bernstein

Robert Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, vol. 16 (7), 641–652, juillet 1986.

On se restreint à $k = i - 1$, $j = 0$ ou $i - 1$, $s_i \neq 0$ et $c_i \neq 0$.

De manière plus générale : il est actuellement implémenté dans des compilateurs, avec la possibilité d'avoir des coûts différents pour l'addition, la soustraction, et le décalage.

Algorithme *branch-and-bound*.

Formules utilisées :

$$
\begin{aligned}
\mathrm{Cost}(1) &= 0 \\
\mathrm{Cost}(n \text{ pair}) &= \mathrm{Cost}(n/2^c \text{ impair}) + \mathtt{ShiftCost} \\
\mathrm{Cost}(n \text{ impair}) &= \min \begin{cases} \mathrm{Cost}(n+1) + \mathtt{SubCost} \\ \mathrm{Cost}(n-1) + \mathtt{AddCost} \\ \mathrm{Cost}(n/(2^c+1)) + \mathtt{ShiftCost} + \mathtt{AddCost} \\ \mathrm{Cost}(n/(2^c-1)) + \mathtt{ShiftCost} + \mathtt{SubCost} \end{cases}
\end{aligned}
$$

(avec diverses techniques permettant d'éviter des recherches inutiles dans l'arbre)

Complexité de cet algorithme : exponentielle.

Mon algorithme – idée

Basé sur la méthode binaire. On considère le nombre comme un vecteur de chiffres 0, $+1$, -1, notés `0`, `P` et `N` (après recodage de Booth).

Idée (appliquée récursivement) : on cherche des motifs qui se répètent, de façon à faire disparaître (apparaître) le plus de chiffres non nuls (`P` et `N`) en une seule opération.

Pour simplifier, on cherche seulement des motifs qui apparaissent deux fois (au moins).

Mon algorithme – exemple

$20061 = 100111001011101_2$, recodé en `P0P00N0P0N00N0P`.
L'utilisation du motif `P000000P0N` permet de faire disparaître 3 chiffres non nuls avec seulement une opération supplémentaire.

```
  P000000P0N
-      P000000P0N
+ 00P000000000000
------------------
  P0P00N0P0N00N0P
```

Sur cet exemple, on obtient 4 opérations (avec l'algorithme de Bernstein, on en obtient 5).

Recherche du motif

Problème : trouver rapidement un motif qui contient le nombre *maximal* de chiffres non nuls, que l'on appellera *poids* du motif.

Trouver un motif $\approx$ trouver un décalage et un signe $(+/-)$.

On tient compte du fait que la proportion des chiffres non nuls est faible en moyenne, surtout près des feuilles de l'arbre de calcul : $w(\text{père}) = 2\,w(\text{motif}) + w(\text{reste})$. Exemple : $7 = 2 \times 3 + 1$.

Solution : on calcule toutes les distances entre deux chiffres non nuls du nombre (en distinguant si ces chiffres sont identiques ou opposés). On obtient ainsi un majorant du poids du motif associé à chaque distance.

Recherche du motif – exemple

Positions : `432109876543210`

Nombre : `POPOONOPONOONOP`

Recherche du motif :

opération	distance	majorant	poids
$\lvert u_j - 2^2 u_k \rvert$	2 (P-N)	3	2
$\lvert u_j - 2^5 u_k \rvert$	5 (P-N)	3	3
$\lvert u_j + 2^7 u_k \rvert$	7 (P-P)	3	2

$\rightarrow$ décalage de 5 avec soustraction.

Complexité de mon algorithme (constr. du code) : $O(m^3)$ au pire.
$O(m^2)$ en moyenne ?

Améliorations possibles

– Recherche de sous-motifs communs, e.g. `PONONOOPONONOOOPON` avec motif `PONON` et reste `PON`.

– Essai de plusieurs motifs de même poids maximal.
Mais attention à la complexité !

– Transformation `PON` $\leftrightarrow$ `OPP` (et `NOP` $\leftrightarrow$ `ONN`), e.g. 11010101001_2 : `PONOPOPOPOOP` $\rightarrow$ 5 opérations, `POONONNOPOOP` $\rightarrow$ 3 opérations (motif `POONOON`). Mais attention à la complexité ! Possibilité de faire la transformation tant que le poids du motif courant est inférieur à une certaine valeur $\rightarrow$ toujours polynomial.

– Utilisation d'autres opérations ($\rightarrow$ autre formulation).

Nouvel algorithme (sous-motifs communs)

Généralisation naturelle de l'algo précédent à la multiplication d'un même nombre par plusieurs constantes : recherche d'un motif qui se répète dans une même constante ou dans 2 constantes différentes.

Au lieu de chercher une distance (décalage), on cherche un triplet (i, j, d) où i et j sont les numéros de deux constantes dans l'ensemble, et d un décalage, avec les restrictions suivantes :

1. $i \leqslant j$;
2. si $i = j$, alors $d > 0$.

Nouvel algorithme – exemple

$n = 47804853381$, qui s'écrit en binaire :

$$1011001000010110010000010110010000101_2$$

Après recodage de Booth :

`P0N0N00P000P0N0N00P000P0N0N00P0000P0P`

1^{re} itération :

Triplet (i, j, d) : $(0, 0, 11)$

Motif associé : `P0N0N00P`

Reste associé : `P0N0N00P00000000000000000000000000P0P`

$\rightarrow$ { `P0N0N00P00000000000000000000000000P0P`, `P0N0N00P` }

Rappel : { `P0N0N00P00000000000000000000000000P0P`, `P0N0N00P` }

2$^{\text{e}}$ itération :

Triplet (i, j, d) : $(0, 1, 29)$

Motif associé : `P0N0N00P`

Reste associé : `P0P`

$\rightarrow$ { `P0P`, `P0N0N00P` }

3$^{\text{e}}$ itération :

Triplet (i, j, d) : $(0, 1, -3)$

Motif associé : `P0P`

Reste associé : `P00000P`

$\rightarrow$ { `P00000P`, `P0P` }

Maintenant, on ne peut plus trouver de motif se répétant de poids supérieur ou égal à 2. On utilise donc la méthode binaire pour les constantes restantes.

$$
\begin{array}{rcrcl}
129x & \leftarrow & x & + & x \ll 7 \quad (\texttt{P000000P}) \\
5x & \leftarrow & x & + & x \ll 2 \quad (\texttt{P0P}) \\
89x & \leftarrow & 129x & - & 5x \ll 3 \quad (\texttt{P0N0N00P}) \\
47781511173x & \leftarrow & 5x & + & 89x \ll 29 \\
 & & \multicolumn{3}{r}{(\texttt{P0N0N00P000000000000000000000000P0P})} \\
47781522565x & \leftarrow & 47781511173x & + & 89x \ll 7 \\
 & & \multicolumn{3}{r}{(\texttt{P0N0N00P000000000000000P0N0N00P0000P0P})} \\
47804853381x & \leftarrow & 47781522565x & + & 89x \ll 18 \\
 & & \multicolumn{3}{r}{(\texttt{P0N0N00P000P0N0N00P000P0N0N00P0000P0P})}
\end{array}
$$

Recherche de coûts par arbre de calcul

Nous nous intéressons à la recherche des coûts optimaux.
$\rightarrow$ Utiliser des propriétés de la fonction de coût optimal q_{opt} provenant de propriétés algébriques de la multiplication.
$\rightarrow$ Définir une fonction f à valeurs dans $\mathbb{N}$, majorant q_{opt}.

f : plus grande fonction de $\mathbb{N}^*$ dans $\mathbb{N}$ s'annulant en 1 et en 2 telle que

- $\forall\, a \in \mathbb{N}^*,\ \forall\, b \in \mathbb{N}^*,\ a > b \Rightarrow f(a \pm b) \leqslant f(a) + f(b) + 1$ (distributivité).
- $\forall\, a \in \mathbb{N}^*,\ \forall\, b \in \mathbb{N}^*,\ f(ab) \leqslant f(a) + f(b)$ (associativité).

Calcul de f

- Déterminer les antécédents de 0 (ce sont les puissances de 2).
- Déterminer successivement les antécédents de $i = 1, 2, 3$, etc. en utilisant la 1re propriété avec a et b tels que $f(a) + f(b) = i - 1$ et la 2e propriété avec a et b tels que $f(a) + f(b) = i$.

Problème : les ensembles considérés sont infinis.

$\rightarrow$ On se limite à un intervalle fixé $[\![1, n]\!]$ et on trouve une fonction f_n telle que $f_n \geqslant f \geqslant q_{\mathrm{opt}}$.

Inconvénient de cette approche : pas de réutilisation de résultats (sauf sous une certaine forme, grâce à la propriété d'associativité).

Résultats et comparaison des algorithmes

Note concernant les implémentations :

- Algo de Bernstein : réimplémenté en C, bien optimisé.
- Algo basés sur des recherches de motifs : implémentés en Perl en utilisant des fonctions de haut niveau, non optimisés.

$\rightarrow$ Pas de temps d'exécution des algorithmes.

q	m	plus petite valeur	
1	2	3	11
2	4	11	1011
3	6	43	101011
4	8	213	11010101
5	11	1703	11010100111
6	14	13623	11010100110111
7	18	174903	101010101100110111
8	21	1420471	101011010110010110111
9	24	13479381	110011011010110111010101

Tab. 1 – Plus petites valeurs pour lesquelles le nouvel algorithme génère un code de longueur supérieure ou égale à q.

q	m	plus petite valeur	
1	2	3	11
2	4	11	1011
3	6	43	101011
4	10	683	1010101011
5	14	14709	11100101110101
6	20	699829	10101010110110110101
7	28	171398453	1010001101110101010100110101

Tab. 2 – Minorants de la plus petite valeur pour laquelle on obtient un code de longueur supérieure ou égale à q avec un algorithme optimal ($q = 6$ et 7 : résultats obtenus par Ross Donelly, postés dans le groupe `rec.puzzles`). Exactement les plus petites valeurs ?

[2 transparents suivants]

Tab. 3 – Nombre moyen d'opérations pour une constante de taille m avec
– l'algorithme de Bernstein,
– notre algorithme de recherche de sous-motifs communs (SMC),
– le même algorithme, mais en considérant toutes les transformations `PON` $\leftrightarrow$ `OPP` et `NOP` $\leftrightarrow$ `ONN` possibles (SMC+),
– une recherche de coûts par arbre de calcul (résultats sur f_n avec $n = 2^{29}$, donc non prouvés),
– une recherche exhaustive bornée à un milliard ($\rightarrow$ optimal ?).

m	Bernstein	SMC	SMC+	Arbre	DAG
2	1 (0.0 %)	1 (0.0 %)	1 (0.0 %)	1 (0.0 %)	1
3	1 (0.0 %)	1 (0.0 %)	1 (0.0 %)	1 (0.0 %)	1
4	1.5 (0.0 %)	1.5 (0.0 %)	1.5 (0.0 %)	1.5 (0.0 %)	1.5
5	1.75 (0.0 %)	1.75 (0.0 %)	1.75 (0.0 %)	1.75 (0.0 %)	1.75
6	2 (0.0 %)	2 (0.0 %)	2 (0.0 %)	2 (0.0 %)	2
7	2.281 (0.0 %)	2.313 (1.4 %)	2.281 (0.0 %)	2.281 (0.0 %)	2.281
8	2.563 (0.6 %)	2.578 (1.2 %)	2.547 (0.0 %)	2.547 (0.0 %)	2.547
9	2.758 (1.1 %)	2.836 (4.0 %)	2.75 (0.9 %)	2.727 (0.0 %)	2.727
10	3.047 (5.5 %)	3.066 (6.2 %)	2.945 (2.0 %)	2.887 (0.0 %)	2.887
11	3.287 (6.2 %)	3.311 (6.9 %)	3.176 (2.6 %)	3.096 (0.0 %)	3.096
12	3.534 (5.7 %)	3.532 (5.7 %)	3.406 (1.9 %)	3.343 (0.0 %)	3.343
13	3.765 (6.0 %)	3.762 (5.9 %)	3.621 (1.9 %)	3.554 (0.0 %)	3.553
14	4.009 (8.1 %)	3.985 (7.4 %)	3.818 (2.9 %)	3.724 (0.4 %)	3.710

m	Bernstein	SMC	SMC+	Arbre	DAG
15	4.246 (10.9%)	4.204 (9.8%)	4.011 (4.8%)	3.876 (1.3%)	3.828
16	4.479 (13.0%)	4.422 (11.5%)	4.209 (6.2%)	4.071 (2.7%)	3.964
17	4.712 (14.1%)	4.639 (12.3%)	4.405 (6.6%)	4.269 (3.3%)	4.131
18	4.944 (14.2%)	4.848 (12.0%)	4.588 (6.0%)	4.461 (3.1%)	4.329
19	5.174 (14.6%)	5.060 (12.1%)	4.769 (5.6%)	4.624 (2.4%)	4.514
20	5.403 (15.8%)	5.268 (12.9%)	4.953 (6.1%)	4.765 (2.1%)	4.667
21	5.630 (17.8%)	5.475 (14.5%)	5.132 (7.3%)	4.904 (2.6%)	4.780
22	5.858 (20.2%)	5.677 (16.5%)	5.312 (9.0%)	5.071 (4.1%)	4.871
23	6.083 (22.4%)	5.880 (18.3%)	5.487 (10.4%)	5.252 (5.6%)	4.972
24	6.309 (23.5%)	6.078 (18.9%)	5.657 (10.7%)	5.433 (6.3%)	5.110
25	6.532 (23.8%)	6.277 (19.0%)	5.827 (10.5%)	5.589 (6.0%)	5.274
26	6.755 (24.0%)	6.473 (18.8%)	5.999 (10.1%)	5.721 (5.0%)	5.447
27	6.978 (24.6%)	6.668 (19.1%)	6.170 (10.2%)	5.841 (4.3%)	5.599

m	$q_{\text{algo 1}}$	$q_{\text{algo 2}}$	gain
8	2.6	2.6	0.0%
16	4.4	4.4	0.0%
32	8.0	7.6	5.0%
64	14.5	13.4	7.6%
128	26.3	23.7	9.9%
256	47.6	42.2	11.3%
512	86.5	75.5	12.7%
1024	157.7	135.4	14.1%
2048	290.0	243.3	16.1%
4096	536.9	440.3	18.0%
8192	1001.9	802.8	19.9%

Tab. 4 – Nombre moyen d'opérations pour des constantes aléatoires de taille m avec les deux algo basés sur des recherches de motifs.

Questions ouvertes

- Complexité du problème et des différentes heuristiques (en temps et en espace) ?
- Comment améliorer les heuristiques ? Utilisation d'autres opérations élémentaires ?
- D'autres voies à explorer, par exemple comme écrire la constante dans un système de numération bien choisi.
- Étudier la relation suivante sur les entiers strictement positifs : $m \rightsquigarrow n$ si le calcul de n peut faire intervenir m comme résultat intermédiaire. Nombres ayant beaucoup de successeurs ?