INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Multiplication by an Integer Constant: Lower Bounds on the Code Length

Vincent Lefèvre

N° 4493

Juillet 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4493--FR+ENG

Multiplication by an Integer Constant: Lower Bounds on the Code Length

Vincent Lefèvre

Thème 2 — Génie logiciel
et calcul symbolique

Projet SPACES

Rapport de recherche n° 4493 — Juillet 2002 — 15 pages

Abstract: The multiplication by a constant problem consists in generating code to perform a multiplication by an integer constant, using elementary operations, such as left shifts (multiplications by powers of two), additions and subtractions. This can also be seen as a method to compress (or more generally encode) integers. We will not discuss about the quality of this compression method, but this idea will be used to find lower bounds on the code length (number of elementary operations).

Key-words: integer multiplication, addition chains, compression

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Multiplication par une constante entière : minorants de la longueur du code

Résumé : Le problème de la multiplication par une constante consiste à générer du code effectuant une multiplication par une constante entière à l'aide d'opérations élémentaires, comme les décalages vers la gauche (multiplications par des puissances de deux), les additions et les soustractions. Ceci peut aussi être vu comme une méthode pour compresser (ou plus généralement encoder) des entiers. Nous ne discuterons pas de la qualité de cette méthode de compression, mais cette idée sera utilisée pour trouver des minorants de la longueur du code (nombre d'opérations élémentaires).

Mots-clés : multiplication entière, chaînes d'additions, compression

1 Introduction

The multiplication by an integer constant problem consists in generating code to perform a multiplication by an integer constant n, using elementary operations, such as left shifts (multiplications by powers of two), additions and subtractions. This problem has been studied and algorithms have been proposed in [3, 4, 7, 8, 9, 5]. Other operations like right shifts could be taken into account, but we will restrict to the above operations, as this was done in [8, 9]. However, the results presented in this paper could straightforwardly be generalized.

In this paper, we are interested in the relation between the multiplication by an integer constant problem and the compression (Section 4). This will allow us to deduce lower bounds on the length of the generated code, also called *program* (Section 5). But we first give a formulation of the problem (the same as in [8, 9]) in Section 2 and discuss on bounds on the shift counts in Section 3.

2 Formulation of the Problem

We now formulate the problem. We have made some assumptions and choices, which are not discussed here. The reader can find more details in [8, 9].

A number x left-shifted k bit positions (i.e., x multiplied by 2^k) is denoted $x \ll k$, and the shift has a higher precedence than the addition and the subtraction (contrary to the precedence rules of some languages, like C or Perl). We assume that the computation time of a shift does not depend on the value k, called the *shift count*.

In practice, shifts will generally be associated with another operation (addition or subtraction): shifts will always be delayed as much as possible. For instance, instead of performing $x \ll 3 + y \ll 8$, we will choose to perform $(x + y \ll 5) \ll 3$. As a consequence, we will choose to work with odd integers as much as possible (there are some exceptions, when the shift count is zero, but this never occurs in the code generated by most algorithms used in practice).

Thus the elementary operations will be additions and subtractions where one of the operands is left-shifted by a fixed number of bits (possibly zero), and these operations will take the same computation time, which will be chosen to be a time unit.

Let n be a nonnegative odd integer (this is our constant). A finite sequence of nonnegative integers $u_0, u_1, u_2, \ldots, u_q$ is said to be *acceptable* for n if it satisfies the following properties:

- initial value: $u_0 = 1$;
- for all $i > 0$, $u_i = |s_i\, u_j + 2^{c_i}\, u_k|$, with $j < i$, $k < i$, $s_i \in \{-1, 0, 1\}$ and $c_i \geq 0$;
- final value: $u_q = n$.

Thus, an arbitrary number x being given, the corresponding program iteratively computes $u_i x$ from already computed values $u_j x$ and $u_k x$, to finally obtain nx. Note that with this formulation, we are allowed to reuse any already computed value. Moreover, the absolute value is only here to make the notations simpler and no absolute value is actually computed: if $s_i = -1$, we always subtract the smaller term from the larger term.

The problem is to generate, from the number n, an acceptable sequence $(u_i)_{0 \le i \le q}$ that is as short as possible; q is called the *quality* (or *length*) of the program (it is the computation time when this program is executed under the condition that each instruction takes one time unit).

As n is odd, we have $s_i \neq 0$ for all i in a minimal acceptable sequence. This can be shown by delaying the shifts. Indeed, if the delayed shift associated with u_i is denoted δ_i, δ_0 being 0, we can write $d_i = \delta_k + c_i - \delta_j$ and u_i can be replaced by u'_i such that $u_i = 2^{\delta_i} u'_i$ with:

$$(u'_i, \delta_i) = \begin{cases} (u'_k, \delta_k + c_i) & \text{if } s_i = 0, \\ (\left|s_i\, u'_j + 2^{d_i}\, u'_k\right|, \delta_j) & \text{if } s_i \neq 0 \text{ and } d_i \geq 0, \\ (\left|s_i\, u'_k + 2^{-d_i}\, u'_j\right|, \delta_k + c_i) & \text{if } s_i \neq 0 \text{ and } d_i < 0, \end{cases}$$

and as n is odd, we have $\delta_q = 0$, therefore the sequence $(u'_i)_{0 \le i \le q}$ is acceptable for n. Operations $u'_i = u'_k$ (corresponding to $s_i = 0$) can be removed. Thus, if there was such an operation, the sequence was not minimal.

This problem consisting in generating optimal code is very difficult (`comp.compilers` contributors conjecture that it is NP-complete). Therefore one uses heuristics in practice; such heuristics have been described in [3, 4, 7, 8, 9, 5]. But we are here interested in lower bounds on the length q of any program: either an optimal program or the generated program for any heuristic. The idea is the following: we will see the program as a compressed form of the number n and the obvious lower bounds on the (worst or average) length of the compressed data will give us lower bounds on the length q of the program.

3 Bounds on the Shift Count

3.1 Considered Bounds

We need to know upper bounds on the shift count c_i in order to obtain upper bounds on the length of the compressed data. But we currently don't know any useful result about that. We will say that it is bounded by $S(m)$, where S is a function of the number of bits m of the constant n.

First, we must assume that $S(m) \geq m$ to be able to compute $2^m - 1$ in one elementary operation. With algorithms (heuristics) used in practice to generate the program, we can assume that $S(m) = m$. But what about optimal programs? To obtain n with an optimal program, it may be necessary to compute values u_i making u_i/n unbounded (this is suggested

by some computations, but not currently proved); indeed, two large values close to each other may be subtracted to give a much smaller value. However, we do not know any useful result about shift counts. So, we will also consider the over-pessimistic bound $S(m) = \alpha.m$ for some $\alpha > 1$, though it is not proved.

3.2 Shift Reduction

We now discuss about results that could allow us to get proved upper bounds on the shift count.

Let n be a nonnegative odd integer, q_{opt} the minimal integer such that there is a program that computes n in q_{opt} elementary operations and $\mathcal{P}$ the set of these minimal (or optimal) programs. We define: $c_{\text{sup}} = \sup_{P\in\mathcal{P},\ 1\le i\le q_{\text{opt}}} c_i$, i.e. the largest shift count that can occur in an optimal program that computes n.

In Section 3.3, we will prove that c_{sup} is finite using the following idea. If there are very large shift counts, this means that the binary representations of the values u_i could be split into at least two parts separated by a long sequence of zeros and the high parts will cancel each other. However, computing the high parts would take useless operations, thus increasing the value of q.

Here's an example with $n = 17$ and $q = 3$ where some shift counts can be as large as we want:

$$\begin{array}{lcllll} u_0 & = & 1 & & \\ u_1 & = & u_0 \ll (c-4) & + & u_0 \\ u_2 & = & u_0 \ll c & - & u_0 \\ u_3 & = & u_1 \ll 4 & - & u_2 \end{array}$$

with $c \ge 4$.

This can be expressed in the program in the following way. First, to make things clearer, the absolute values are removed: $|u - v|$ is written $u - v$ or $v - u$ depending on whether $u \ge v$ or $u < v$. Then, we will work on polynomials, where each monomial represents a part, the degree-0 monomial being the low part. Initially, we only have a low part: $u_0 = 1$. The operations remain as before, except for the large shifts, where a multiplication by X is performed. For instance, with the previous program, we would perform:

$$\begin{array}{lcllll} u_0 & = & 1 & & \\ u_1 & = & u_0X \ll (c-4) & + & u_0 \\ u_2 & = & u_0X \ll c & - & u_0 \\ u_3 & = & u_1 \ll 4 & - & u_2\,. \end{array}$$

Thus,

$$u_3 = 2^4u_1 - u_2 = 2^4(2^{c-4}X + 1) - (2^cX - 1) = 17.$$

Large shifts have canceled each other and we obtain a degree-0 monomial. By replacing X by 0, the program can be simplified and shortened.

Though having an optimal program means that the shift counts are bounded, it does not necessarily mean that the shift counts are as low as possible, i.e. considering $c_{\sup}$ only would not lead to optimal bounds on $S(m)$, because some optimal programs may use larger shift counts than others. There is the following trivial example:

$$\begin{aligned} 3 &= 1 \ll 2 - 1 \\ &= 1 \ll 1 + 1, \end{aligned}$$

which can be computed by an optimal program whose largest shift count is 1, though $c_{\sup} = 2$. But this is not the only case. Indeed, here is a more general example.

For $h \geq 2$, consider $n = (1 + 2^h)(1 + 2^{2h})(1 + 2^{4h}) - 2^{7h}$, which is a $6h + 1$-bit number that can be computed with 4 operations only, but using a shift count of $7h$:

$$\begin{aligned} u_0 &= 1 \\ u_1 &= u_0 \ll h + u_0 \\ u_2 &= u_1 \ll 2h + u_1 \\ u_3 &= u_2 \ll 4h + u_2 \\ u_4 &= u_3 - u_0 \ll 7h\,. \end{aligned}$$

We prove below that this number cannot be computed with fewer than 4 operations, thus $c_{\sup} \geq 7h$ for this number, though there exists a program that computes this number with 4 operations with a maximum shift count of $2h$ (so, not larger than $6h + 1$):

$$\begin{aligned} u_0 &= 1 \\ u_1 &= u_0 \ll h + u_0 \\ u_2 &= u_1 \ll h + u_0 \\ u_3 &= u_2 \ll 2h + u_1 \\ u_4 &= u_3 \ll 2h + u_1\,. \end{aligned}$$

To prove that $(1 + 2^h)(1 + 2^{2h})(1 + 2^{4h}) - 2^{7h}$ cannot be computed with fewer than 4 operations, we need the following lemma:

Lemma 1 *Let r be a nonnegative integer, $(s_i)_{1 \leq i \leq r}$ be r integers equal to ± 1 (signs) and $(c_i)_{1 \leq i \leq r}$ be r nonnegative integers. Consider*

$$n = \sum_{i=1}^{r} s_i \, 2^{c_i}\,.$$

Then there exists a representation of n in binary using signed digits that has no more than r nonzero digits.

The lemma can be proved by induction. It is true for $r = 0$. Assume that it is true up to $r - 1$. Let us prove it for the value r. If all the c_i's are different, then the representation

associated with the above sum is suitable. Otherwise there exist j and k ($j \neq k$) between 1 and r such that $c_j = c_k$. If $s_j \neq s_k$, we can remove the corresponding terms from the sum; this leads to a sum with $r-2$ terms, which can be written with no more than $r-2$ nonzero digits, therefore with no more than r digits. If $s_j = s_k$, we can replace the corresponding terms by a term having the same sign and a shift count equal to $c_j + 1$; this leads to a sum with $r-1$ terms, which can be written with no more than $r-1$ nonzero digits, therefore with no more than r digits. □

Now we can prove the following theorem.

Theorem 1 *For $h \geq 2$, the number*

$$(1+2^h)(1+2^{2h})(1+2^{4h}) - 2^{7h} = \sum_{i=0}^{6} 2^{ih}$$

cannot be computed with fewer than 4 elementary operations.

The number $n = (1+2^h)(1+2^{2h})(1+2^{4h}) - 2^{7h}$ is written in binary with 7 digits one, separated by at least a zero (since $h \geq 2$); it cannot be written with fewer digits (canonical Booth's recoding). As a consequence of the lemma, when n is expressed as a sum or difference of powers of two, there are at least 7 terms in this expression.

With only 1 elementary operation, we have at most 2 terms in the expression. With 2 elementary operations, we have at most 4 terms after expanding the expression. With 3 elementary operations, the values (j,k) associated with i in a program can be, up to an isomorphism of the corresponding DAG:

- $(0,0)\ (0,0)\ (1,2)\ \rightarrow$ 4 terms.
- $(0,0)\ (0,1)\ (0,2)\ \rightarrow$ 4 terms.
- $(0,0)\ (0,1)\ (1,2)\ \rightarrow$ 5 terms.
- $(0,0)\ (0,1)\ (2,2)\ \rightarrow$ 6 terms.
- $(0,0)\ (1,1)\ (0,2)\ \rightarrow$ 5 terms.
- $(0,0)\ (1,1)\ (1,2)\ \rightarrow$ 6 terms.
- $(0,0)\ (1,1)\ (2,2)\ \rightarrow$ 8 terms.

Therefore, if n can be computed with fewer than 4 operations, then it can be computed only with a DAG of the last form, i.e. we can write:

$$n = (2^a + s_a)\,(2^b + s_b)\,(2^c + s_c)$$

with $a, b, c \geq 1$ and $s_a, s_b, s_c \in \{1, -1\}$.

First, notice that n is congruent to 1 modulo 3:

$$n = (1 + 2^h)(1 + 2^{2h})(1 + 2^{4h}) - 2^{7h} \equiv (1 + 2^h) \times 2 \times 2 - 2^h \equiv 1 \pmod 3.$$

If $a = 1$, then $s_a \neq 1$ (because n is not divisible by 3) and $s_a \neq -1$ (otherwise, the expression can be written with at most 4 terms). Therefore $a \geq 2$. For the same reasons, $b \geq 2$ and $c \geq 2$. As $h \geq 2$, $n \equiv 1 \pmod 4$; therefore $s_a s_b s_c = 1$.

$2^a \not\equiv 0 \pmod 3$, therefore $2^a + s_a \not\equiv s_a \pmod 3$, and as $n \not\equiv 0 \pmod 3$, $2^a + s_a \not\equiv 0 \pmod 3$. The only possibility is: $2^a + s_a \equiv -s_a \pmod 3$. For the same reasons, $2^b + s_b \equiv -s_b \pmod 3$ and $2^c + s_c \equiv -s_c \pmod 3$. As a consequence,

$$n \equiv (-s_a)(-s_b)(-s_c) = -(s_a s_b s_c) = -1 \pmod 3,$$

which leads to a contradiction. This proves Theorem 1. □

3.3 Proof of a Large Upper Bound on $S(m)$

We now search for an upper bound on $S(m)$ by using the following idea: If the shift counts can be very large, then, as the number of elementary operations is limited to q, itself bounded by $\lfloor m/2 \rfloor$ (using Booth's recoding and the naive algorithm as described in [8, 9]), there will be at least a "hole" giving two parts in the binary representation.

Let m be an integer greater than 1, n a m-bit nonnegative odd integer (our constant) and $(u_i)_{0 \leq i \leq q}$ an associated minimal acceptable sequence. The associated shift counts are denoted c_i as in the formulation. Let σ be a permutation of the first q nonnegative integers 1, 2, ..., q that orders the shift counts, $d_i = c_{\sigma(i)}$ and $d_0 = -1$ so that for all $1 \leq j \leq q$, we have $d_j \geq d_{j-1}$.

Let j be an integer between 1 and q and let us mark (by multiplying by X) the shifts for which $c_i \geq d_j$, i.e. the shifts d_1, d_2, ..., d_{j-1} are not marked and the shifts d_j, d_{j+1}, ..., d_q are marked. The corresponding polynomial can be written: $A(X).X + b$, where A is a polynomial with integer coefficients and b an integer constant. We have: $A(1) + b = n$.

As q is minimal, we have $A(1) \neq 0$. And as in the computation, each coefficient of X is divisible by 2^{d_j}, then $A(1)$ is divisible by 2^{d_j} and we have: $|A(1)| \geq 2^{d_j}$.

Now, let us find an upper bound on $|b|$. To compute b, we ignore the shifts for which $c_i \geq d_j$, i.e. we write $X = 0$. Thus we have:

$$|b| \leq \prod_{k=1}^{j-1} \left(2^{d_k} + 1\right).$$

If we assume that $d_k \geq 2^{k-1}$ for $1 \leq k \leq j-1$ (this is not necessarily true in practice, but this will be satisfied by our bounds), then we have:

$$\prod_{k=1}^{j-1} \left(1 + 2^{-d_k}\right) \leq \prod_{k=0}^{j-2} \left(1 + 2^{-2^k}\right) = 2\left(1 - 2^{-2^{j-1}}\right) < 2.$$

Therefore $|b| < 2^{s_j}$ with:

$$s_j = 1 + \sum_{k=1}^{j-1} d_k.$$

As $A(1) + b = n$, we have $|A(1)| \leq n + |b|$. Therefore

$$2^{d_j} \leq |A(1)| \leq n + |b| < 2^m + 2^{s_j}$$

and $d_j \leq \max(m, s_j)$. From this inequality, we can deduce that the smallest shift count is bounded by m, the next one by $m+1$, then $2m+2$, then $4m+4$, and so on. By induction, we can prove that for $j \geq 2$, the j-th shift count can be bounded by $2^{j-2}(m+1)$. Note that $m \geq 2^0$ and for $j \geq 2$, $2^{j-2}(m+1) \geq 2^{j-1}$, justifying our above assumption. As a consequence, we have the following theorem.

Theorem 2 *Let m be an integer greater than 1 and n a m-bit nonnegative odd integer. Consider an optimal program that computes n. The largest shift count is smaller or equal to*

$$\begin{cases} m & \text{if } q = 1 \\ 2^{q-2}(m+1) & \text{if } q \geq 2. \end{cases}$$

As $q \leq \lfloor m/2 \rfloor$, we can take for $m \geq 4$:

$$S(m) = 2^{\lfloor m/2 \rfloor - 2}(m+1)$$

which is asymptotically equivalent to $2^{\lfloor m/2 \rfloor - 2}\, m$. Unfortunately, this upper bound is so large that it will not give us any interesting result in the following.

4 Compression

The program contains nonnegative integers. If we use the representation of the nonnegative integers in base 2, it will not be possible to know when the corresponding word ends; thus we need a prefix code of the integers. Of course, such a code should have a small length complexity (for instance, the well-known base-1 representation 1^n0 is not acceptable). So, we will first describe a method to encode the nonnegative integers (Section 4.1).

Then we will describe the encoding of an elementary operation and the whole program (Section 4.2), and finally we will deal with the size of the program (Section 4.3).

4.1 Prefix Code of the Nonnegative Integers

The problem of finding a prefix code of the nonnegative integers is related to the *unbounded search problem*, which has been studied by Bentley and Yao [2], Raoult and Vuillemin [10], Knuth [6], and Beigel [1]. However, to make things simpler in this paper, we will not define a prefix code as short as the ones dealt with in [1], in particular because our problem is more general: we need to encode several integers and taking that into account for the choice of the code could be preferable.

The conventional representation of the nonnegative integers in base 2 gives an encoding in $\lfloor \log_2(n) \rfloor + 1$ bits (for $n \geq 1$). So, we look for a prefix code that would be in slightly more than $\lfloor \log_2(n) \rfloor$ bits, i.e. something in $\log_2(n) + o(\log_2(n))$. The idea is to give the length of the word in an efficient way. We could give it in the base-1 representation, but this is not sufficient to achieve our goal. So, we will give it in base 2 and give the length of the length in base 1 (in fact, a variant of that).

Precisely, 0, 1, 2 and 3 will be respectively encoded as 000, 001, 010, 011. For $n \geq 4$, k denotes the number of bits of n without the first 1, i.e. n has $k+1$ bits ($k \geq 2$). For $k \geq 2$, h denotes the number of bits of k without the first 1, i.e. k has $h+1$ bits ($h \geq 1$). The code word of n will be h digits 1, a 0, the h bits of k without the first 1, and the k bits of n without the first 1. Table 1 gives the code words for a few integers.

integer	code word
0	000
1	001
2	010
3	011
4	10 0 00
5	10 0 01
6	10 0 10
7	10 0 11
8	10 1 000
15	10 1 111

integer	code word
16	110 00 0000
31	110 00 1111
32	110 01 00000
63	110 01 11111
64	110 10 000000
127	110 10 111111
128	110 11 0000000
255	110 11 1111111
256	1110 000 00000000
511	1110 000 11111111

Table 1: *Encoding of a few integers.*

The code word corresponding to a nonnegative integer n has the length

$$C(n) = \begin{cases} 3 & \text{if } n \leq 3, \\ \lfloor \log_2(n) \rfloor + 2\ \lfloor \log_2(\log_2(n)) \rfloor + 1 & \text{if } n \geq 4. \end{cases}$$

We have: $C(n) = \log_2(n) + o(\log_2(n))$, i.e. $C(n) \sim \log_2(n)$.

4.2 Encoding of the Program

An elementary operation has the form $u_i = |s_i\, u_j + 2^{c_i}\, u_k|$. Thus it suffices to encode s_i and the nonnegative integers c_i, j and k. The four words may be simply concatenated. As s_i can take three possible values (-1, 0 and 1), we use two bits[1] to encode s_i. The fourth combination of these two bits can be used to indicate the end of the program. The nonnegative integers c_i, j and k are encoded as described in Section 4.1.

The program is encoded by concatenating the code words of its elementary operations, and the two-bit stop word at the end.

4.3 Size of the Program

We are now interested in the size of the encoded program.

First, let us search for a bound on the size of the i-th elementary operation. The integer c_i is bounded by $S(m)$, as said in Section 3. The integers j and k are bounded by $i-1$. Thus, the i-th elementary operation can be encoded with at most $2 + C(S(m)) + 2\,C(i-1)$ bits and a program of length q, i.e. having q elementary operations, can be encoded with at most

$$\sum_{i=1}^{q} (2 + C(S(m)) + 2\,C(i-1)) + 2 = q\,(2 + C(S(m))) + 2\sum_{i=0}^{q-1} C(i) + 2$$

bits. The main term of $C(i)$ is $\lfloor \log_2(i) \rfloor$ (for $i \geq 4$), so we wish to evaluate the corresponding sum. We can prove by induction that

$$L(n) = \sum_{i=1}^{n} \lfloor \log_2(i) \rfloor = (n+1)\ \lfloor \log_2(n) \rfloor - 2^{\lfloor \log_2(n) \rfloor + 1} + 2.$$

There is not much difference with $n\ \lfloor \log_2(n) \rfloor$, and in particular, $L(n)$ is asymptotically equivalent to $n \log_2(n)$. So, without too much loss, we can bound i by q in the above formula.

Therefore a program of length q can be encoded with at most

$$B(m,q) = q\,(2 + C(S(m)) + 2\,C(q-1)) + 2$$

bits. This bound is asymptotically equivalent to $q\,(\log_2(S(m)) + 2\ \log_2(q))$, and if we assume that $S(m) = \alpha.m$ as in Section 3, we have:

$$B(m,q) \sim q\,(\log_2(m) + 2\ \log_2(q)).$$

[1] This is not necessarily the best choice, in particular knowing the fact that $s_i = 0$ can be avoided (except for the last elementary operation, if even integers are accepted as the input). But as this will not make a significant difference, we prefer to keep the general case.

5 Lower Bounds on the Length of the Program

First, we will define the following notation. If f and g are two nonnegative functions on some domain, we write $f(x) \gtrsim g(x)$ if there exists a function ε such that $|\varepsilon(x)| = o(1)$ and $f(x) \geq g(x)\,(1 + \varepsilon(x))$. This is also equivalent to say that there exists a function ε' such that $|\varepsilon'(x)| = o(1)$ and $f(x)\,(1 + \varepsilon'(x)) \geq g(x)$.

For each m-bit nonnegative odd integer n, we consider an optimal program that computes n, adding the following restriction on the shift counts to the formulation: $\forall i,\ c_i \leq S(m)$; of course, if $S(m)$ is large enough, this is no longer a restriction. For each n, the corresponding program length q_n is thus completely defined.

5.1 Worst Case

Let us consider the nonnegative odd integers having exactly m bits in their binary representation. They are 2^{m-2} such integers. As all the corresponding encoded programs must be different, there exists an integer whose code word has at least $m - 2$ bits. Therefore, for this integer, one has $B(m, q) \geq m - 2$. This gives a lower bound on the value q_{worst} of q in the worst case.

Asymptotically, under the $S(m) = \alpha.m$ assumption, we have:

$$B(m, q_{\text{worst}}) \sim q_{\text{worst}}\,(\log_2(m) + 2\,\log_2(q_{\text{worst}})).$$

Thanks to computations, we can guess that $\log_2(q_{\text{worst}}) \sim \log_2(m)$; therefore, without too much loss, we can bound $\log_2(q_{\text{worst}})$ by $\log_2(m)$ and write:

$$3\,q_{\text{worst}}\,\log_2(m) \gtrsim B(m, q_{\text{worst}}).$$

We can deduce: $3\,q_{\text{worst}}\,\log_2(m) \gtrsim m$. As a consequence, we obtain Theorem 3.

Theorem 3 *Let $m \geq 2$. For each positive odd integer n having exactly m bits in its binary representation, consider an acceptable sequence computing n and let q_n denote its length. Let $q_{\text{worst}} = \max_n q_n$ be the maximum length. Assume that the shift counts are bounded above by a function $S(m) = \alpha.m$ (α being a positive constant). Then*

$$q_{\text{worst}} \gtrsim \frac{m}{3\,\log_2(m)}.$$

Note that this also proves what we have guessed: $\log_2(q_{\text{worst}}) \sim \log_2(m)$.

We can also deduce an exact (instead of asymptotic) bound, for $m \geq 4$:

$$q_{\text{worst}} > \frac{m - 4}{2 + C(S(m)) + 2\,C(m)}$$

and

$$q_{\text{worst}} > \frac{m-4}{3\ \log_2(m) + 4\ \lfloor \log_2(\log_2(m)) \rfloor + 2\ \lfloor \log_2(\log_2(\alpha.m)) \rfloor + \log_2(\alpha) + 6}.$$

Of course, this bound can easily be (very slightly) improved. But this is not the goal of this paper.

We do not know if the lower bound given by Theorem 3 is reached. The only currently known upper bound for the worst case is $m/2$ (obtained with Booth's recoding).

5.2 Average Case

Now we wish to obtain similar results for the average case. Again, let us consider the set O_m of the 2^{m-2} nonnegative odd integers having exactly m bits in their binary representation. As all the corresponding encoded programs must be different, the average code word length is at least:

$$\frac{1}{2^{m-2}} \sum_{i=1}^{2^{m-2}} \lfloor \log_2 i \rfloor = \frac{L(2^{m-2})}{2^{m-2}} = m - 4 + \frac{m}{2^{m-2}},$$

(where L was defined in Section 4.3), i.e.

$$\frac{1}{2^{m-2}} \sum_{i \in O_m} B(m, q_i) \geq m - 4 + \frac{m}{2^{m-2}}.$$

Therefore

$$2 + (2 + C(S(m)) + 2\,C(m)) \frac{1}{2^{m-2}} \sum_{i \in O_m} q_i \geq m - 4 + \frac{m}{2^{m-2}}$$

and

$$q_{\text{av}} \geq \frac{m - 6 + m/2^{m-2}}{2 + C(S(m)) + 2\,C(m)}.$$

Asymptotically, under the $S(m) = \alpha.m$ assumption, we obtain Theorem 4, i.e. the same bound as with the worst case.

Theorem 4 *Let $m \geq 2$. For each positive odd integer n having exactly m bits in its binary representation, consider an acceptable sequence computing n and let q_n denote its length. Let $q_{\text{av}} = 2^{2-m} \sum_n q_n$ be the average length. Assume that the shift counts are bounded above by a function $S(m) = \alpha.m$ (α being a positive constant). Then*

$$q_{\text{av}} \gtrsim \frac{m}{3\ \log_2(m)}.$$

Again, we do not know if the lower bound given by Theorem 4 is reached. The only currently known upper bound for the average case is $m/3$ (obtained with Booth's recoding).

5.3 The Case of Bernstein's Algorithm

With Bernstein's algorithm (described in [3, 4, 8, 9]), an elementary operation can only be one amongst:

$$u_i = \begin{cases} 2^{c_i}\, u_{i-1} - 1 & \text{with } c_i \geq 1, \\ 2^{c_i}\, u_{i-1} + 1 & \text{with } c_i \geq 1, \\ (2^{c_i} - 1)\, u_{i-1} & \text{with } c_i \geq 2, \\ (2^{c_i} + 1)\, u_{i-1} & \text{with } c_i \geq 2, \end{cases}$$

and the shift count c_i is always bounded by $S(m) = m$. Contrary to the generic elementary operation, only one integer (c_i) needs to be encoded instead of 3. As a consequence, in the asymptotic lower bounds, instead of having a factor 3, we have a factor 1:

$$q_{\text{worst}} \gtrsim \frac{m}{\log_2(m)} \quad \text{and} \quad q_{\text{av}} \gtrsim \frac{m}{\log_2(m)}.$$

We may find larger lower bounds using the fact that the sum of the shift counts has the same magnitude as m.

References

[1] R. Beigel. Unbounded searching algorithms. *SIAM Journal on Computing*, 19(3):522–537, 1990.

[2] J. L. Bentley and A. C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, August 1976.

[3] R. Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7):641–652, July 1986.

[4] P. Briggs and T. Harvey. Multiplication by integer constants. ftp://ftp.cs.rice.edu/public/preston/optimizer/multiply.ps.gz, July 1994.

[5] R. Fredrickson. Constant coefficient multiplication. Master's thesis, Brigham Young University, December 2001.

[6] D. Knuth. Supernatural numbers. In D. A. Klarner, editor, *The Mathematical Gardner*, pages 310–325. Wadsworth International, Belmont, CA, 1981.

[7] V. Lefèvre. Multiplication by an integer constant. Research report RR1999-06, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1999.

[8] V. Lefèvre. Multiplication by an integer constant. Research report RR-4192, INRIA, May 2001.

[9] V. Lefèvre. Multiplication par une constante. *Réseaux et Systèmes Répartis, Calculateurs Parallèles*, 13(4–5):465–484, 2001.

[10] J.-C. Raoult and J. Vuillemin. Optimal unbounded search strategies. Research report 33, Laboratoire de Recherche en Informatique, Université de Paris-Sud, Orsay, France, 1979.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399