УДК 681.142.2

Скоростные свойства алгоритмов умножения и деления целых чисел произвольного размера

М.Ю. Барышникова [1], А.Ф. Деон [1], А.В. Силантьева [1]

[1] МГТУ им. Н.Э. Баумана, Москва, 105005, Россия

Приведено описание подходов к оценке быстродействия алгоритмов реализации операций умножения и деления целых чисел произвольной размерности на основе подсчета количества операций, выполняемых в ходе их обработки. Это позволяет определить границы применимости формы представления «длинных» чисел в виде одномерных массивов, в которых каждая цифра занимает один байт.

E-mail: baryshnikovam@mail.ru, deonalex@mail.ru, av_sil@mail.ru

Ключевые слова: *быстродействие, алгоритм, числа, размерность.*

Компьютерные операции процессора ориентированы на выполнение действий с двоичными целыми числами размером n байтов, где n — разрядность процессора. Количество различимых чисел находится в диапазоне $[0, 2^n - 1]$, т. е. от 0 до 4 294 967 296 при 32-разрядном процессоре или от 0 до 18 446 744 073 709 551 616 — при 64-разрядном.

В исключительных ситуациях, когда требуется оперировать числами большего размера, необходима реализация алгоритмических операций, выполняемых по специальным программам. К таким операциям относятся умножение и деление целых чисел произвольного размера. Оценка быстродействия этих алгоритмов составляет суть данного исследования. В качестве единицы измерения быстродействия в рамках этой работы используются операции — элементарные команды из перечня машинных команд сложения, вычитания, сохранения, сравнения и пересылки данных. При этом предполагается, что эти команды выполняются за одну операцию. В качестве языка программирования используется исторический язык C, хотя такие его диалекты, как CLR (common language runtime) или C#, предоставляют аналогичные реализации, но с возможностями объектно-ориентированного программирования.

Для представления целых чисел произвольного размера используется модель, согласно которой каждая цифра целого числа располагается в отдельном байте, представляя все число как массив цифр-байтов [1]. Поскольку возможно появление дополнительных старших цифр при выполнении операции умножения, то десятичные цифры в массиве располагаются в обратном порядке, что упрощает реализацию умножения простым заполнением цифр в следующих старших байтах результата. Старший байт массива числа должен содержать код знака «+» или «–». В представленных далее программах исполь-

зуются функции *ByteBits*( ), *ByteBitPrint*( ), *ByteBitMass*( ) и *InputAB*( ) для преобразования обычного строкового вида вводимых чисел к байтовым массивам хранения и отображения этих чисел. Тексты этих функций приведены в [2].

Умножение. Для реализации умножения целых десятичных чисел в байтовом представлении ниже приведена программа *MI01*, в которой вычисляется произведение $c = a * b$. Числа a и b вводятся со знаком с клавиатуры в функции *InputAB*( ). В подключаемом файле *#include MultCarry.h* находится функция *MultCarry*( ) для переноса цифр, возникающих в результате поразрядного умножения. В подключаемом файле *ByteBitMult.h* находится сама функция умножения *ByteBitMult*( ), которую рассмотрим далее.

```
// Program MI01 (Win32)
// Умножение целых байтовых чисел в обратном порядке
#include <conio.h>                                          // _getch
#include <iostream>
using namespace std;
#include <string.h>                                         // strlen
#include "InputAB.h"
#include "MultCarry.h"
#include "ByteBitMult.h"
void main( void )
{      char sa[256];              // символьный буфер целого числа a
       unsigned char a[256];
       char sb[256];              // символьный буфер целого числа b
       unsigned char b[256];
       InputAB( sa, a, sb, b);                    // ввод чисел a и b
       int na = strlen( sa );                        // длина числа a
       int nb = strlen( sb );                        // длина числа b
//       Умножение байтовых целых чисел
       unsigned char c[256];             // для результата умножения
       int nc = ByteBitMult( c, a, na, b, nb );         // c = a * b
       cout << "c = a * b = "; ByteBitPrint( c, nc );  // листинг c
       _getch();                                  // просмотр результата
}
```

Если при выполнении программы *MI01* ввести числа $a = 59$ и $b = 999$, то на монитор будет выведен результат $c = a * b = 58\,941$:

```
sa = +59
a = 00001001  00000101  00101011
sb = +999
b = 00001001  00001001  00001001  00101011
c = 00000001  00000100  00001001  00001000  00000101  00101011
```

Перенос цифр при одноразрядном умножении. При реализации умножения могут появляться цифры переноса в старшие разряды. Обычно умножение многоразрядных чисел проводится поразрядно. Поскольку одна десятичная цифра занимает в двоичном представлении младшие 4 бита в байте, то одноразрядное умножение не может выйти за пределы байта. Перенос цифр заключается в том, чтобы оставить в разряде только биты, относящиеся к младшим цифрам из

интервала $[\overline{0;9}]$. Старшая цифра прибавляется к цифре следующего старшего разряда.

Перенос цифр выполняется в функции *MultCarry*( ). В интерфейсе функции параметр *unsigned char* c* является указателем байтового массива с числом, полученным после поразрядного умножения. Параметр *int nc* содержит длину числа без знака. Параметр *int k* задает индекс байта, с которого следует оценивать перенос. Функция возвращает новую длину числа после реализации поразрядных переносов:

```
// Файл MultCarry.h (Win32)
// Перенос в цифрах байтового числа
int inline MultCarry( unsigned char* c, int nc, int k )
{      c[nc] = 0;
       for( int j = k; j < nc; j++ )
       {      if( c[j] > 9 )   // двухзначное число после умножения
              {      c[j+1]+=c[j]/10;//перенос старшей цифры умножения
                     c[j] = c[j] % 10;   // младшая цифра умножения
              }
       }
       if( c[nc] == 0 ) return nc; // длина результата без переноса
       return nc+1;                 // длина результата с переносом
}
```

В теле функции *MultCarry*( ) анализ скоростных свойств [3] начинается на первом этапе с инструкции *c*[*nc*] = *0*, в которой предполагается, что пока в старшем разряде числа отсутствует какая-либо цифра переноса. В этой инструкции выполняются две операции:

$$W_1 = 2.$$

На втором этапе выполняется цикл под управлением заголовка *for*(*int j = k; j < nc; j++*). Инициализация *int j = k* занимает одну операцию:

$$W_{2,0} = 1.$$

На каждой итерации происходит проверка условия $j < nc$ и выполнение приращения j++, затрачивая в общей сложности три операции:

$$W_{2,1} = \sum_{j=k}^{nc-1} 3 = 3\left(nc - k\right).$$

В теле цикла заголовок инструкции условия *if*(*c*[*j*] > *9*) осуществляет контроль каждого разряда числа, при этом затрачиваются две операции. Если условие *c*[*j*] > 9 истинно, то производится перенос с помощью инструкций *c*[*j+1*]+=*c*[*j*]*/10; c*[*j*] = *c*[*j*] *% 10* с общим числом операций $6 + 4 = 10$:

$$W_{2,2\min} = \sum_{j=k}^{nc-1} 2 = 2\left(nc - k\right);$$

$$W_{2,2\max} = \sum_{j=k}^{nc-1} \left(2+10\right) = 12\left(nc-k\right).$$

Таким образом, на втором этапе выполняется следующее количество операций:

$$W_{2\min} = W_{2,0} + W_{2,1} + W_{2,2\min} =$$

$$= 1 + 3\left(nc-k\right) + 2\left(nc-k\right) =$$

$$= 1 + 5\left(nc-k\right);$$

$$W_{2\max} = W_{2,0} + W_{2,1} + W_{2,2\max} =$$

$$= 1 + 3\left(nc-k\right) + 12\left(nc-k\right) =$$

$$= 1 + 15\left(nc-k\right).$$

Третий этап включает выполнение одной инструкции *if*(*c*[*nc*] == *0*) *return nc*:

$$W_3 = 2.$$

Если условие *c*[*nc*] == *0* ложно, то была цифра переноса в старший разряд результата, а само число стало на один разряд длиннее. Следовательно, на четвертом этапе будет выполнена инструкция *return nc+1*:

$$W_4 = 1.$$

Функция *MultCarry*( ) завершена. Ее быстродействие можно оценивать по количеству операций на каждом этапе:

$$W_{\min}^{MultCarry} = W_1 + W_{2\min} + W_3 =$$

$$= 2 + 1 + 5\left(nc-k\right) + 2 =$$

$$= 5 + 5\left(nc-k\right);$$

$$W_{\max}^{MultCarry} = W_1 + W_{2\max} + W_3 + W_4 =$$

$$= 2 + 1 + 15\left(nc-k\right) + 2 + 1 =$$

$$= 6 + 15\left(nc-k\right).$$

Функция умножения. Умножение целых чисел произвольного размера в байтовом представлении проводится в приведенной ниже функции *ByteBitMult*( ). Параметр *unsigned char* c* является указате-

лем на массив результата $c = a * b$, параметр *unsigned char* a* — на байтовый массив сомножителя *a* со знаком. Длина массива находится в параметре *int na*. Параметры *unsigned char* b, int nb* содержат аналогичную информацию для сомножителя *b*.

```
// Файл ByteBitMult.h (Win32)
// Умножение целых десятичных чисел произвольного размера
int ByteBitMult( unsigned char* c, unsigned char* a, int na,
                      unsigned char* b, int nb )
{      int m = na + nb - 2;                // длина результата c=a*b
       for( int i = 0; i < m; i++ ) c[i] = 0;           // чистка c
       int na1 = na - 1;                          // длина a без знака
       int nb1 = nb - 1;                          // длина b без знака
       for( int j = 0; j < nb1; j++ )       // сомножитель позиций b
       {      for( int i = 0; i < na1; i++)//сомножитель повторения a
              {      int k = i + j;          // позиция в результате
                     c[k] += a[i] * b[j];
              }
              m = MultCarry( c, na+j, j );    // перенос в умножении
       }
       for( ; m > 0; m-- )
              if( c[m] != 0 ) break;                // старшая цифра
       m = m + 1;                                    // позиция знака
       if( a[na-1] == b[nb-1] ) c[m] = 43;               // знак '+'
       else c[m] = 45;                                   // знак '-'
       return m+1;                                  // длина со знаком
}
```

В теле функции *ByteBitMult*( ) на первом этапе выполняется инструкция *int m = na + nb – 2*, затрачивая при этом три операции:

$$W_1 = 3.$$

Второй этап содержит цикл чистки массива *for*(*int i = 0; i < m; i++*) *c*[*i*] = *0*. Одна операция затрачивается на инициализацию *int i = = 0*. На каждой итерации проверяется условие $i < m$ за одну операцию, создается приращение *i++* (две операции), выполняется инструкция *c*[*i*] = *0* (две операции):

$$W_2 = 1 + \sum_{i=0}^{m-1}(1+2+2) = \sum_{i=0}^{na+nb-2-1} 5 = 5(na + nb - 2).$$

На третьем этапе выполняются две инструкции — *int na1 = na – 1; int nb1 = nb – 1*, затрачивая четыре операции:

$$W_3 = 4.$$

Четвертый этап содержит главный цикл умножения под управлением заголовка *for*(*int j = 0; j < nb1; j++*). Одна операция затрачивается на инициализацию *int j = 0*:

$$W_{4,0} = 1.$$

На каждой итерации выполняется проверка условия $j < nb1$ (одна операция), и две операции затрачиваются на приращение *j++*:

$$W_{4,1} = \sum_{j=0}^{nb-2} (1+2) = 3(nb-1).$$

В теле главного цикла выполняется внутренний цикл под управлением заголовка *for*(*int i = 0; i < na1; i++*). Одна операция затрачивается на инициализацию *int i = 0*:

$$W_{4,2,0} = \sum_{j=0}^{nb-2} 1 = nb-1.$$

На каждой итерации внутреннего цикла выполняется проверка условия *i < na1* (одна операция), и две операции отводятся для приращения *i++*:

$$W_{4,2,1} = \sum_{j=0}^{nb-2} \sum_{i=0}^{na-2} (1+2) = \sum_{j=0}^{nb-2} 3(na-1) =$$

$$= 3(na-1)(nb-1).$$

Для выполнения двух инструкций *int k = i + j; c*[*k*] *+= a*[*i*] ** b*[*j*] в теле внутреннего цикла требуются восемь операций:

$$W_{4,2,2} = \sum_{j=0}^{nb-2} \sum_{i=0}^{na-2} (2+6) = \sum_{j=0}^{nb-2} 8(na-1) =$$

$$= 8(na-1)(nb-1).$$

В общей сложности на выполнение внутреннего цикла затрачивается следующее количество операций:

$$W_{4,2} = W_{4,2,0} + W_{4,2,1} + W_{4,2,2} =$$

$$= (nb-1) + 3(na-1)(nb-1) + 8(na-1)(nb-1) =$$

$$= (nb-1)(1+11(na-1)) =$$

$$= (nb-1)(11na-10).$$

После внутреннего цикла поразрядного умножения в главном цикле четвертого этапа выполняется поразрядная корректировка переноса цифр с помощью инструкции *m = MultCarry*(*c, na+j, j*):

$$W_{4,3\min} = \sum_{j=0}^{nb-2} \left(1 + W_{\min}^{MultCarry}\right) = \sum_{j=0}^{nb-2} (1+5+5(na+j-j)) =$$

$$= (nb-1)(6+5na);$$

$$W_{4,3\max} = \sum_{j=0}^{nb-2}\left(1 + W_{\max}^{MultCarry}\right) = \sum_{j=0}^{nb-2}\left(1 + 6 + 13\left(na + j - j\right)\right) =$$

$$= \left(nb - 1\right)\left(7 + 13na\right).$$

Цикл четвертого этапа завершен. Можно оценить его минимальные и максимальные скоростные свойства:

$$W_{4\min} = W_{4,1} + W_{4,2} + W_{4,3\min} =$$

$$= 3\left(nb - 1\right) + \left(nb - 1\right)\left(11na - 10\right) + \left(nb - 1\right)\left(6 + 5na\right) =$$

$$= \left(nb - 1\right)\left(3 + 11na - 10 + 6 + 5na\right) =$$

$$= \left(-1 + 16na\right)\left(nb - 1\right);$$

$$W_{4\max} = W_{4,1} + W_{4,2} + W_{4,3\max} =$$

$$= 3\left(nb - 1\right) + \left(nb - 1\right)\left(11na - 10\right) + \left(nb - 1\right)\left(7 + 13na\right) =$$

$$= \left(nb - 1\right)\left(3 + 11na - 10 + 7 + 13na\right) =$$

$$= 24na\left(nb - 1\right).$$

Пятый этап характеризуется циклом под управлением заголовка *for*(*; m > 0; m--*). На каждой итерации проходит проверка условия *m > 0* (одна операция) и обеспечение декремента *m--* (две операции). Поскольку числа *a* и *b* минимально содержат по одной цифре и знак, то хотя бы одна итерация будет выполнена:

$$W_{5,1\min} = 1 + 2 = 3;$$

$$W_{5,1\max} = \sum_{m=1}^{na+nb-2}\left(1 + 2\right) = 3\left(na + nb - 2\right).$$

В теле цикла выполняется инструкция условия *if*(*c*[*m*] *! = 0*) *break*. На проверку условия *c*[*m*] *! = 0* затрачиваются две операции:

$$W_{5,2\min} = 1 \cdot 2 = 2;$$

$$W_{5,2\max} = \sum_{m=1}^{na+nb-2} 2 = 2\left(na + nb - 2\right).$$

Таким образом, на пятом этапе затрачивается следующее количество операций:

$$W_{5\min} = W_{5,1\min} + W_{5,2\max} = 3 + 2 = 5;$$

$$W_{5,2\max} = W_{5,1\max} + W_{5,2\max} =$$

$$= 3\left(na + nb - 2\right) + 2\left(na + nb - 2\right) = 5\left(na + nb - 2\right).$$

Шестой этап завершает функцию *ByteBitMult*( ). Инструкция *m = m + 1* определяет место знака в массиве результата, затрачивая две операции. Следующая инструкция альтернативного условия *if(a[na – 1] == b[nb – 1]) c[m] = 43; else c[m] = 45* ставит код знака (пять операций), выход из функции *return m+1* занимает одну операцию:

$$W_6 = 2 + 5 + 1 = 8.$$

Функция *ByteBitMult*( ) завершена. Общее количество ее операций определяется следующим образом:

$$W_{\min}^{ByteBitMult} = W_1 + W_2 + W_3 + W_{4\min} + W_{5\min} + W_6 =$$

$$= 3 + 5\left(na + nb - 2\right) + 4 + \left(nb - 1\right)\left(-6 + 16na + \frac{5}{2}nb\right) + 2\left(na + nb - 2\right) + 8 =$$

$$= 15 + 8\left(na + nb - 2\right) + \left(-1 + 16na\right)\left(nb - 1\right) =$$

$$= -8na + 7nb + 16na \cdot nb;$$

$$W_{\max}^{ByteBitMult} = W_1 + W_2 + W_3 + W_{4\max} + W_{5\max} + W_6 =$$

$$= 3 + 5\left(na + nb - 2\right) + 4 + 24na\left(nb - 1\right) + 5\left(na + nb - 2\right) + 8 =$$

$$= 3 - 14na + 10nb + 24na \cdot nb\,.$$

Деление. В представленной далее программе *DI01* проводится деление $c = a / b$ целых десятичных чисел *a* и *b* в байтовом представлении. Числа вводятся со знаком с клавиатуры в функции *InputAB*( ) [2].

В подключаемых файлах *#include "SingleMultiply.h"* и *#include "SingleSubtract.h"* находятся функции *SingleMultiply*( ) и *SingleSubtract*( ), которые используются в определении частного. Их тексты приведены в [2]. Вычисление частного проводится по алгоритму дихотомического поиска множителя с помощью функций *DichMult*( ) и *DMultiplier*( ), находящихся в подключаемых файлах *#include "DichMult.h"* и *#include "DMultiplier.h"*, тексты которых также представлены в [2]. Функция деления *ByteBitDiv*( ), чей текст будет показан далее, находится в подключаемом файле *ByteBitDiv.h*:

```
// Program DI01 (Win32)
// Деление целых байтовых чисел в обратном порядке
#include <conio.h>                                    // _getch
#include <iostream>
using namespace std;
#include <string.h>                                   // strlen
#include "InputAB.h"
```

```
#include "SingleMultiply.h"
#include "SingleSubtract.h"
#include "DichMult.h"
#include "DMultiplier.h"
#include "ByteBitDiv.h"
void main(void)
{
char sa[256];                    // символьный буфер целого числа a
unsigned char a[256];
char sb[256];                    // символьный буфер целого числа b
unsigned char b[256];
InputAB( sa, a, sb, b );                        // ввод чисел a и b
int na = strlen( sa );                // количество цифр в числе a
int nb = strlen( sb );                // количество цифр в числе b
//       Деление байтовых целых чисел
unsigned char c[256];                  // для результата умножения
unsigned char s[256];              // для промежуточного вычитания
int nc = ByteBitDiv( c, a, na, b, nb, s );            // c = a / b
cout << "c = a / b = "; ByteBitPrint( c, nc );        // результат
cout << "s = "; ByteBitPrint( s, nb-1 );        // листинг остатка
_getch();                                     // просмотр результата
}
```

Если при запуске программы *DI01* ввести числа $a = 96918$ и $b = -999$, то на мониторе окажется результат $c = a / b = -97$ с остатком $s = 15$:

```
sa = +96918
a = 00001000  00000001  00001001  00000110  00001001  00101011
sb = -999
b = 00001001  00001001  00001001  00101101
c = a / b = 00000111  00001001  00101101
s = 00000101  00000001  00000000
```

Деление целых чисел произвольного размера в байтовом представлении проводится в приведенной ниже функции *ByteBitDiv*( ). Параметр *unsigned char* c* является указателем на массив результата частного $c = \frac{a}{b}$, параметр *unsigned char* a* — на байтовый массив делимого *a* со знаком. Длина массива находится в параметре *int na*. Параметры *unsigned char* b, int nb* содержат аналогичную информацию для делителя *b*. Байтовый массив *unsigned char* s* содержит остаток без знака. Функция *ByteBitDiv*( ) возвращает длину массива *c*:

```
// Файл ByteBitDiv (Win32)
// Деление байтовых целых чисел
int ByteBitDiv( unsigned char* c, unsigned char* a, int na,
                    unsigned char* b, int nb,
                    unsigned char* s )
{    int nc = DichMult2( c, a, na, b, nb, s );
      int k = nc - 1;                     // позиция старшей цифры
      if( c[k] != 0 ) k += 1;
      if( a[na-1] == b[nb-1] ) c[k] = 43;              // знак '+'
      else c[k] = 45;                                  // знак '-'
      return k + 1;                    // длина c = a / b со знаком
}
```

На первом этапе в теле функции *ByteBitDiv*( ) выполняется поиск частного от деления с использованием функции дихотомического поиска множителя *DMultiplier*( ) [2]. Результат поиска помещается в переменную *nc = DMultiplier*(*c, a, na, b, nb, s*), для чего затрачивается еще одна операция:

$$W_{1\min} = 1 + W_{\min}^{DMultiplier} = 1 + 7 + 43nb + 10na + 43nb\left(na - nb\right) =$$

$$= 8 + 47nb + 10na + 43na\left(na - nb\right);$$

$$W_{1\max} = 1 + W_{\max}^{DMultiplier} = 1 + 109 + 367nb - 106na + 251nb\left(na - nb\right) =$$

$$= 110 + 367nb - 106na + 251nb\left(na - nb\right).$$

На втором этапе вычисляется позиция старшей цифры частного *int k = nc – 1*, для чего требуется выполнить две операции:

$$W_2 = 2.$$

На третьем этапе проводится корректировка позиции знака с помощью инструкции условия *if*(*c*[*k*] *!= 0*) *k += 1*. По две операции затрачиваются на проверку условия *c*[*k*] *!= 0* и на корректировку *k += 1*:

$$W_{3\min} = 2;$$

$$W_{3\max} = 2 + 2 = 4.$$

На четвертом этапе частное получает соответствующий знак *if*(*a*[*na – 1*] == *b*[*nb – 1*]) *c*[*k*] = *43*; *else c*[*k*] = *45*, что занимает семь операций:

$$W_4 = 7.$$

На заключительном, пятом этапе выполняется инструкция *return k + 1*:

$$W_5 = 1.$$

Теперь можно получить общую оценку количества операций, выполняемых при делении целых чисел произвольного размера:

$$W_{\min}^{ByteBitDiv} = W_{1\min} + W_2 + W_{3\min} + W_4 + W_5 =$$

$$= 8 + 47nb + 10na + 43nb\left(na - nb\right) + 2 + 2 + 7 + 1 =$$

$$= 20 + 47nb + na + 43nb\left(na - nb\right);$$

$$W_{\max}^{ByteBitDiv} = W_{1\max} + W_2 + W_{3\max} + W_4 + W_5 =$$

$$= -82 + 337nb - 84na + 243na \cdot nb - 243nb^2 + 2 + 4 + 7 + 1 =$$

$$= 110 + 367nb - 106na + 251na\left(na - nb\right).$$

Таким образом, проведенный анализ подтвердил фактически квадратичную зависимость операций умножения и деления целых чисел от размерности обрабатываемых чисел. В то же время, максимальный коэффициент при произведении размерностей $\left(na \cdot nb\right)$ обрабатываемых чисел равен 24, а при делении — 251, что говорит о десятикратной длительности операции деления по сравнению с операцией умножения целых чисел произвольной размерности. При этом для деления чисел использована функция дихотомического поиска множителя, что позволило сократить число итераций поиска с 10 до четырех $(\log 29 = 4)\left]\log_2 9\right[= 4$ и, в свою очередь, ускорить операцию деления.

СПИСОК ЛИТЕРАТУРЫ

1. Окулов С.М. Основы программирования. М.: Лаборатория базовых знаний, 2002. 424 с.
2. Деон А.Ф. Дихотомический поиск множителя целых чисел произвольного размера // Вестник МГТУ им. Н.Э. Баумана: электронное научно-техническое издание. 2013.
3. Седжвик Р. Функциональные алгоритмы на C++: Анализ/Структуры данных/Сортировка/Поиск/: пер. с англ. СПб.: ООО «ДианаСофЮП», 2002. 688 с.

Статья поступила в редакцию 25.10.2012