УДК 681.142.2

Дихотомический поиск множителя целых чисел произвольного размера

А.Ф. Деон [1]

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

Рассмотрены вопросы формирования множителя целых чисел произвольного размера по алгоритму дихотомического поиска и анализа количества выполняемых компьютерных операций при определении скоростных свойств алгоритма.

E-mail: deonalex@mail.ru

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

В операции деления десятичных чисел произвольного размера необходимо последовательно находить очередную цифру в числе результата. Эту цифру можно получать перебором цифр 0, 1, … , 9 в десятичной системе счисления (десятичной арифметике). Заранее эта цифра неизвестна, поэтому для самого длительного последовательного перебора цифр потребуются 10 итераций. Поскольку при анализе быстродействия выполняемых программ часто приходится ориентироваться на самый длительный случай, то следует учитывать именно эти 10 итераций.

Число итераций можно сократить, если учесть свойство возрастающей упорядоченности десятичных цифр. Тогда поиск необходимой цифры можно выполнить дихотомическим способом. Число итераций в самом длительном случае оценивается с округлением в бóльшую сторону как $\lceil \log_2 9 \rceil = 4$. Целью исследования является построение и оценка быстродействия алгоритма дихотомического поиска множителя для целых десятичных чисел произвольного размера в байтовом представлении. В качестве языка программирования используется исторический процедурный диалект языка C. Такие диалекты языка C, как CLR (common language runtime) или C#, позволяют получить аналогичные реализации по технологии объектно-ориентированного программирования.

Для представления целых чисел произвольного размера можно использовать метод расположения каждой цифры целого числа в отдельном байте, представляя все число как массив цифр-байтов [1]. Поскольку далее потребуется выполнить операцию вычитания, то применяется обратный порядок расположения десятичных цифр в массиве [2]. В процессе вычитания положительных чисел возможно появление отрицательного результата, например (+5) – (+9) = –4, поэтому старший байт массива числа должен содержать код знака «+» или «–».

Исходные данные. Ниже представлены функции: *ByteBits*( ) — для двоичного представления цифр числа, *ByteBitPrint*( ) — для печа-

ти байтовоого массива числа и *ByteBitMass*( ), в которой символьный массив числа копируется в байтовый массив в обратном порядке:

```
Void inline ByteBits( unsigned char z )
{      unsigned char b = 0x80;                          // 1 в бите 7
       while( b > 0 )
       {      ( z & b ) == 0 ? cout << '0' : cout << '1';
              b >>= 1;
       }
}
//---------------------------------------------------------------
void inline ByteBitPrint( unsigned char* u, int nu )
{      for( int i = 0; i < nu; i++ )                    // листинг u
       {      ByteBits( u[i] );
              cout << "  ";
       }
       cout << endl;
}
//---------------------------------------------------------------
#include <string.h>                                      // strlen
void ByteBitMass( unsigned char* a, char* sa )
{      int n = strlen( sa );
       for( int i = 1; i < n; i++ )
              a[n-1-i] = (unsigned char)sa[i] - 48; //цифровой
                                                    /элемент
       a[n-1] = (unsigned char) sa[0];                // знак числа
}
```

Эти функции используются для ввода двух чисел *a* и *b* в функции *InputAB*( ):

```
#include <string.h>                                      // strlen
#include "ByteBits.h"
#include "ByteBitPrint.h"
#include "ByteBitMass.h"
void InputAB( char* sa, unsigned char* a,
                char* sb, unsigned char* b )
{      cout << "sa = ";
       cin >> sa;                          // символьный ввод числа a
       ByteBitMass( a, sa );       // преобразование в байтовый вид
       int na = strlen( sa );          // количество цифр в числе a
       cout << "a = "; ByteBitPrint( a, na );    // листинг числа a
       cout << "sb = ";
       cin >> sb;                          // символьный ввод числа b
       ByteBitMass( b, sb );       // преобразование в байтовый вид
       int nb = strlen( sb );          // количество цифр в числе b
       cout << "b = "; ByteBitPrint( b, nb );    // листинг числа b
}
```

В представленной ниже программе *HI01* функции, которые находятся в одноименных подключаемых файлах, позволяют ввести интересующие нас два числа с клавиатуры и просмотреть их на мониторе:

```
// Program HI01 (Win32)
// Исходные данные
#include <conio.h>                                       // _getch
#include <iostream>
using namespace std;
```

```
#include "InputAB.h"
void main(void)
{      char sa[256];              // символьный буфер целого числа a
       unsigned char a[256];       // байтовый массив целого числа a
       char sb[256];              // символьный буфер целого числа b
       unsigned char b[256];       // байтовый массив целого числа b
       InputAB( sa, a, sb, b );                  // ввод чисел a и b
       _getch();                               // просмотр результата
}
```

Если в программе *HI01* ввести число $a = 57$ и число $b = 8$, то на мониторе появится следующий результат:

```
sa = +57
a = 00000111  00000101  00101011
sb = +8
b = 00001000  00101011
```

Одноразрядное байтовое умножение. В дальнейшем для реализации поиска дихотомического множителя потребуется функция *SingleMultiply*( ), выполняющая умножение целого числа *b* произвольного размера на одноразрядное целое число *e*. Результат умножения размещается в массиве $c = b * e$. В функции *SingleMultiply*( ) параметр *int nb1* содержит количество байтов в числе *b* без учета знака числа. Функция возвращает число байтов в массиве *c* без учета знака результата.

```
int inline SingleMultiply( unsigned char* c, unsigned char* b,
                                  int nb1, int e )
{      c[nb1] = 0;             // позиция для старшей цифры переноса
       for( int i = 0; i < nb1; i++ )
              c[i] = b[i] * e;              // поразрядное умножение
       for( int j = 0; j < nb1; j++ )         // поразрядный перенос
       {      if( c[j] > 9 )   // двухзначное число после умножения
              {      c[j+1] += c[j] / 10;  // перенос старшей цифры
                     c[j] = c[j] % 10;   // младшая цифра умножения
              }
       }
       if( c[nb1] == 0 ) return nb1;//длина результата без переноса
       return nb1 + 1;              // длина результата с переносом
}
```

В представленной ниже программе *HI02* функция *SingleMultiply*( ) выполняет умножение $c = a * b$:

```
// Program HI02 (Win32)
// Умножение одноразрядного целого числа на байтовое число
#include <conio.h>                                          // _getch
#include <iostream>
using namespace std;
#include "InputAB.h"
#include "SingleMultiply.h"
void main(void)
{      char sa[256];              // символьный буфер целого числа a
       unsigned char a[256];         // байтовый массив целого числа
       char sb[256];              // символьный буфер целого числа b
       unsigned char b[256];       // байтовый массив целого числа b
       InputAB( sa, a, sb, b );                  // ввод чисел a и b
```

```
      int na = strlen( sa );              // байтовая длина числа a
//       Одноразрядное умножение c = a * b
      unsigned char c[256];              // для результата умножения
      int e = b[0];                       // одноразрядный множитель
      int nc = SingleMultiply( c, a, na-1, e );       // c = a * e
      cout << "c = a * b = "; ByteBitPrint( c, nc );  // результат
      _getch();                                  // просмотр результата
}
```

Если ввести числа $a = 58$ и $b = 3$, то получаем следующий результат:

```
sa = +58
a = 00001000  00000101  00101011
sb = +3
b = 00000011  00101011
c = a * b = 00000100  00000111  00000001
```

Скоростные свойства функции *SingleMultiply*( ) определяются количеством операций над целыми числами внутри тела функции. Поскольку в результате умножения количество цифр в итоге может быть на единицу больше, то в начале функции предполагается, что *c*[*nb1*] = *0*, пока нет переноса старшей цифры.

В теле функции *SingleMultiply*( ) на первом этапе выполняется инструкция *c*[*nb1*] = *0* с одной операцией индексирования и с одной операцией сохранения числа 0:

$$W_1 = 1 + 1 = 2.$$

На втором этапе выполняется цикл одноразрядного умножения *for*(*int i = 0; i < nb1; i++*) *c*[*i*] = *b*[*i*] ∗ *e*. Сначала производится одна операция в выражении *int i = 0*. Затем на каждой итерации выполняются одна операция проверки условия *i < nb1*, две операции для приращения *i++* и четыре операции в выражении *c*[*i*] = *b*[*i*] ∗ *e*. В общей сложности на втором этапе будет выполнено следующее количество операций:

$$W_2 = 1 + \sum_{i=0}^{nb1-1} \left(1 + 2 + 4\right) = 1 + 7nb1.$$

На третьем этапе возможен перенос цифр в старшие разряды результата под управлением заголовка цикла *for*(*int j = 0*; *j < nb1*; *j++*). Сначала выполняется одна операция в выражении *int j = 0*. На каждой итерации выполняются по три операции на условие *j < nb1* и приращение *j++*. В теле цикла заголовок условия *if*(*c*[*j*] > *9*) контролирует появление цифры переноса, если цифра результата представлена двузначным числом. При этом выполняется одна операция индексирования и одна операция проверки условия *>9*. Если переносы отсутствуют, то на этом этапе будет выполнено минимальное количество операций:

$$W_{3\min} = 1 + \sum_{j=0}^{nb1-1} \left(3 + 2\right) = 1 + 5nb1.$$

В теле инструкции условия находятся инструкции *c[j+1] + = = c[j] / 10; c[j] = c[j] % 10*, на выполнение которых затрачиваются девять операций. На третьем этапе будет выполнено максимальное число итераций, если перенос выполняется для каждого разряда в числе результата:

$$W_{3\max} = 1 + \sum_{j=0}^{nb1-1} (3+2+9) = 1 + 14nb1.$$

На последнем, четвертом этапе в минимальном случае, когда отсутствуют переносы, будут выполнены две операции в инструкции *if(c[nb1] == 0) return nb1*:

$$W_{4\min} = 2.$$

Если на старшей цифре результата происходит перенос, то выполняется одна операция в инструкции *return nb1 + 1*. Вместе с предыдущим условием *if(c[nb1] == 0)* получаем

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

Подводя итог, получаем минимальную и максимальную оценки числа операций при выполнении функции *SingleMultiply*( ):

$$W_{\min}^{SingleMultiply} = W_1 + W_2 + W_{3\min} + W_{4\min} =$$

$$= 2 + 1 + 7nb1 + 1 + 5nb1 + 2 =$$

$$= 6 + 12nb1;$$

$$W_{\max}^{SingleMultiply} = W_1 + W_2 + W_{3\max} + W_{4\max} =$$

$$= 2 + 1 + 7nb1 + 1 + 14nb1 + 3 =$$

$$= 7 + 21nb1.$$

Выровненное байтовое вычитание. При подборе соответствующего дихотомического множителя необходимо проводить вычитание, чтобы убедиться в том, что остаток от произведения предлагаемого дихотомического множителя на заданный множитель не превосходит значение заданного множителя. Такое вычитание выполняет функция *SingleSubtract*( ), в которой параметр *unsigned char*s* является указателем на массив побайтового результата вычитания, параметр *unsigned char* u* задает массив уменьшаемого, параметр *unsigned char* v* определяет массив вычитаемого, параметр *int nv* определяет количество байтов в выровненных по длине массивах *u* и *v* без учета знаков чисел:

```
int inline SingleSubtract( unsigned char* s, unsigned char* u,
                                    unsigned char* v, int nv )
{     unsigned char tranzit = 0;          // для занимаемой единицы
```

```
        for( int i = 0; i < nv; i++ )  // цикл побайтового вычитания
        {       signed char d = u[i] - tranzit - v[i];
                if( d < 0 ) { d = 10 + d; tranzit = 1; }
                else tranzit = 0;
                s[i] = d;
        }
        if( tranzit == 0 )                  // положительный результат
        {       s[nv] = 43;                                 // знак '+'
                return nv+1;
        }
        s[0] = 10 - s[0];  // формирование отрицательного результата
        for( int i = 1; i < nv; i ++ ) s[i] = 9 - s[i];
        s[nv] = 45;                                          // знак '-'
        return nv+1;
}
```

В представленной ниже программе *HI03* выполняется вычитание $s = a - b$ введенных чисел a и b. Функция вычитания *SingleSubtract*( ) находится в подключаемом файле *#include "SingleSubtract.h"*:

```
// Program HI03 (Win32)
// Выровненное вычитание байтовых целых чисел
#include <conio.h>                                          // _getch
#include <iostream>
using namespace std;
#include "InputAB.h"
#include "SingleSubtract.h"
void main(void)
{       char sa[256];              // символьный буфер целого числа a
        unsigned char a[256];       // байтовый массив целого числа a
        char sb[256];              // символьный буфер целого числа b
        unsigned char b[256];       // байтовый массив целого числа b
        InputAB( sa, a, sb, b );                  // ввод чисел a и b
        int na = strlen( sa );              // байтовая длина числа a
//        Выровненное вычитание s = a - b
        unsigned char s[256];             // для результата вычитания
        int ns = SingleSubtract( s, a, b, na-1 );       // s = a - b
        cout << "s = a - b = "; ByteBitPrint( s, ns );  // результат
        _getch();                                 // просмотр результата
}
```

Если при выполнении программы *HI03* ввести числа $a = 25$ и $b = 9$, то на мониторе появится следующий результат:

```
sa = +25
a = 00000010  00000101  00101011
sb = +09
b = 00001001  00000000  00101011
s = a - b = 00000110  00000001  00101011
```

В теле функции *SingleSubtract*( ) на первом этапе будет выполнена одна инструкция *unsigned char tranzit = 0*, которая занимает одну операцию:

$$W_1 = 1.$$

На втором этапе выполняется инструкция цикла с заголовком *for*(*int i = 0*; *i < nv*; *i++*). Выражение *int i = 0* занимает одну опера-

цию. На каждой итерации выполняются одна операция проверки условия $i < nv$ и две операции приращения i++. В теле цикла определены три инструкции. В первой инструкции *signed char d = u*[*i*] – – *tranzit* – *v*[*i*] проводится вычитание соответствующих разрядов чисел с учетом занимаемой единицы на предыдущей итерации, для чего будет затрачено пять операций. Во второй инструкции заголовок условия *if*(*d < 0*) проверяет, надо ли занимать единицу из следующего старшего разряда, на что уходит одна операция. Если условие истинно, выполняются инструкции *d = 10 + d; tranzit = 1*, затрачивая в общем три операции. В противном случае на выполнение инструкции *else tranzit = 0* затрачивается одна операция. В третьей инструкции *s*[*i*] = *d* выполняются две операции. Таким образом, оценка числа операций на втором этапе может быть минимальной и максимальной:

$$W_{2\,\min} = 1 + \sum_{i=0}^{nv-1} \left(1+2+5+1+1+2\right) = 1 + 12nv;$$

$$W_{2\,\max} = 1 + \sum_{i=0}^{nv-1} \left(1+2+5+1+3+2\right) = 1 + 14nv.$$

На третьем этапе проводится проверка, является ли результат вычитания положительным или отрицательным. Если результат *if*(*tranzit == 0*) положительный, то будут выполнены три операции в инструкциях *s*[*nv*] = *43; return nv+1*:

$$W_{3\,\min} = 1;$$

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

Если результат вычитания отрицательный, то выполняется четвертый этап: необходимо преобразовать полученный байтовый код в цифры, соответствующие отрицательному числу в евклидовой арифметике. Для этого сначала модифицируется младшая цифра результата в инструкции *s*[*0*] = *10* – *s*[*0*], затрачивая две операции. Затем преобразуются остальные цифры в цикле *for*(*int i = 1; i < nv; i ++*) *s*[*i*] = *9* – *s*[*i*]; на этот этап будет затрачено следующее число операций:

$$W_4 = 2 + 1 + \sum_{i=1}^{nv-1} \left(1+2+3\right) = 3 + 6\left(nv-1\right) = -3 + 6nv.$$

На последнем, пятом этапе выполняются инструкции *s*[*nv*] = *45; return nv+1*, для чего потребуются три операции:

$$W_5 = 3.$$

Оценка общего количества операций в функции *SingleSubtract*( ) может быть минимальной или максимальной в зависимости от количества занимаемых единиц при вычитании и от знака результата:

$$W_{\min}^{SingleSubtract} = W_1 + W_{2\min} + W_{3\max} = 1+1+12nv+4 = 6+12nv;$$

$$W_{\max}^{SingleSubtract} = W_1 + W_{2\max} + W_{3\min} + W_4 + W_5 =$$

$$= 1+1+14nv+1-3+6nv+3 = 3+20nv.$$

Дихотомический поиск одноразрядного множителя для чисел одной длины. Пусть заданы два числа a и b одной длины. Необходимо найти такое одноразрядное число $z \in \left[\overline{0,9}\right]$, чтобы выполнялось условие $a - b*z < b$. Это накладывает дополнительное ограничение на число $a \le b*9$. Такая задача возникает при делении в столбик из начального курса арифметики.

Поиск числа z можно выполнить, последовательно перебирая цифры 0, 1, 2, …, 9. Если в каком-либо случае окажется, что $z = 9$, то выполняется 10 итераций поиска числа 9: в произвольном случае это соответствует максимальному числу итераций поиска.

При дихотомическом поиске числа z учитывается арифметическая упорядоченность цифр 0, 1, 2, …, 9. Дихотомия заключается в том, что сначала проверяют число из середины интервала: полагают $z = \left[(0+9)/2\right] = 4$, отбрасывая дробную часть. Если для такого z условие $a - b*z < b$ не соблюдается, то ищут следующее число в интервале $[0,1,2,3]$ или $[5,6,7,8,9]$. Всего таких итераций будет $]\log_2 9[= 4$. В общем случае максимальное число итераций при дихотомическом поиске почти в два раза меньше, чем при последовательном поиске.

Одноразрядный дихотомический поиск выполняется в функции *DichMult*( ) при условии, что длина числа a равна длине числа b. В этой функции параметр *unsigned char* u* является указателем на байтовый массив числа a, параметр *unsigned char* v* задает байтовый массив числа b. Параметр *int nv* содержит байтовую длину массива v без учета знака числа, параметр *unsigned char* c* предназначен для байтового массива, в который помещается произведение $v*z$, параметр *unsigned char* s* обозначает байтовый массив, в котором находится остаток $s = a - s*z$. Функция *DichMult*( ) возвращает найденный дихотомический множитель z:

```
int inline DichMult( unsigned char* u, unsigned char* v,
                            int nv,
                            unsigned char* c, unsigned char* s )
{
//     cout << "Begin DichMult" << endl;
       int n1 = 0, n2 = 9;                  // начало и конец поиска
       int z;                                  // середина интервала
       while( n1 <= n2 )                              // цикл поиска
       {
```

```
//            cout << "n1 = " << n1 << "   n2 = " << n2 << endl;
              z = ( n1 + n2 ) >> 1; // z=(n1+n2)/2 середина интервала
//            cout << "z = " << z << endl;
              int nc = SingleMultiply( c, v, nv, z );  // c = v * z
//            cout << "c = v * z = "; ByteBitPrint( c, nc );
              if( nc > nv )                              // c > v * z
              {     n2 = z - 1;          // продолжить поиск слева
                    continue;
              }
              int ns1 = SingleSubtract( s, u, c, nc );//s = u - v * z
              if( s[ns1-1] == 45 )   // знак '-', z слишком большое
              {     n2 = z - 1;          // продолжить поиск слева
                    continue;
              }
              int ns2 = SingleSubtract( c, s, v, nc );   // c = s - v
              if( c[ns2-1] == 45 ) break;  // знак '-', найдено z * v
              n1 = z + 1;        // z мало, продолжить поиск справа
       }
//     cout << "End DichMult" << endl;
       return z;
}
```

В программе *HI04*, представленной ниже, выполняется дихотомический поиск одноразрядного множителя *dm* для введенных чисел *a* и *b* одинаковой длины. Функция *DichMult*( ) находится в подключаемом файле *#include "DichMult.h"*:

```
// Program HI04 (Win32)
// Выровненный одноразрядный дихотомический множитель
// для байтовых целых чисел
#include <conio.h>                                        // _getch
#include <iostream>
using namespace std;
#include "InputAB.h"
#include "SingleMultiply.h"
#include "SingleSubtract.h"
#include "DichMult.h"
void main(void)
{     char sa[256];             // символьный буфер целого числа a
      unsigned char a[256];       // байтовый массив целого числа
a
      char sb[256];             // символьный буфер целого числа b
      unsigned char b[256];       // байтовый массив целого числа
b
      InputAB( sa, a, sb, b );                 // ввод чисел a и b
      int nb = strlen( sb );             // байтовая длина числа b
//       Выровненный одноразрядный дихотомический поиск
      unsigned char c[256];        // для промежуточных вычислений
      unsigned char s[256];           // для результата вычитания
      int dm = DichMult( a, b, nb-1, c, s );             // b * dm
      cout << "dm = " << dm << endl;
      cout << "s = a - b * dm = "; ByteBitPrint( s, nb-1);
      _getch();                              // просмотр результата
}
```

Если при запуске программы *HI04* ввести числа $a = 41$ и $b = 12$ и прокомментировать отладочные распечатки в функции *DichMult*( ),

то на мониторе появится результат, позволяющий проследить последовательность дихотомического поиска множителя *dm*:

```
sa = +41
a = 00000001  00000100  00101011
sb = +12
b = 00000010  00000001  00101011
Begin DichMult
n1 = 0  n2 = 9
z = 4
c = v * z = 00001000  00000100
n1 = 0  n2 = 3
z = 1
c = v * z = 00000010  00000001
n2 = 2  n2 = 3
z = 2
c = v * z = 00000100  00000010
n1 = 3  n2 = 3
z = 3
c = v * z = 00000110  00000011
End DichMult
dm = 3
s = a - b * dm = 00000101  00000000
```

Оценка скоростных свойств функции *DichMult*( ) начинается на первом этапе с выполнения инструкции *int n1 = 0, n2 = 9*, содержащей две операции:

$$W_1 = 2.$$

На втором этапе находится инструкция цикла поиска множителя с заголовком *while*(*n1 <= n2*). На каждой итерации затрачивается одна операция для проверки условия *n1 <= n2*:

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

В теле цикла вычисляется дихотомический множитель *z* = (*n1* + + *n2*) >> *1*, на что приходятся три операции. Затем проводится умножение *v*z* с помощью инструкции *int nc = SingleMultiply*(*c, v, nv, z*). Оценка выполнения функции *SingleMultiply*( ) была приведена ранее. Учитывая одну операцию на запоминание в *nc*, всего будет затрачено следующее количество операций:

$$W_{2,1\min} = 3 + W_{\min}^{SingleMultiply} + 1 = 3 + 6 + 12nv + 1 = 10 + 12nv;$$

$$W_{2,1\max} = 3 + W_{\max}^{SingleMultiply} + 1 = 3 + 7 + 21nv + 1 = 11 + 21nv.$$

Следует проанализировать длину результата после умножения. Если получаемое число длиннее множителя *v*, то значение *z* не может быть результатом дихотомического поиска. Проверка выполняется с помощью инструкции условия с заголовком *if*(*nc > nv*). На выяснение условия затрачивается одна операция в выражении *nc > nv*. Если условие истинно, то для следующей итерации дихотомического по-

иска необходимо выбрать левый интервал $\left[\overline{n1, z-1}\right]$. Параметры нового интервала вычисляются в теле инструкции условия с помощью инструкции *n2 = z – 1*, на что затрачиваются две операции:

$$W_{2,2\ false} = 1;$$

$$W_{2,2\ true} = 1 + 2 = 3.$$

Далее с помощью вычитания *int ns1 = SingleSubtract(s, u, c, nc)* выполняется анализ, является ли полученный множитель *z* искомым дихотомическим множителем. Количество операций для функции *SingleSubtract( )* было вычислено ранее. Учитывая запоминание в *ns1*, получаем следующее количество операций:

$$W_{2,3\min} = W_{\min}^{SingleSubtract} + 1 = 6 + 12nv + 1 = 7 + 12nv;$$

$$W_{2,3\max} = W_{\max}^{SingleSubtract} + 1 = 3 + 20nv + 1 = 4 + 20nv.$$

После вычитания необходимо проанализировать знак результата с помощью заголовка инструкции условия *if(s[ns1–1] == 45)*. На это затрачиваются три операции. Если знак отрицательный, то предлагается продолжить поиск в левом интервале *n2 = z – 1*:

$$W_{2,4\min} = 3;$$

$$W_{2,4\ \max} = 3 + 2 = 5.$$

При положительном знаке после вычитания необходимо выполнить еще одно вычитание *int ns2 = SingleSubtract(c, s, v, nc)*, чтобы убедиться в том, что либо дихотомический множитель найден, либо его следует искать в правом интервале. Учитывая запоминание в *ns2*, получаем следующее количество операций:

$$W_{2,5\min} = W_{\min}^{SingleSubtract} + 1 = 6 + 12nv + 1 = 7 + 12nv;$$

$$W_{2,5\max} = W_{\max}^{SingleSubtract} + 1 = 3 + 20nv + 1 = 4 + 20nv.$$

Оценка знака результата второго вычитания проводится в заголовке инструкции условия *if(c[ns2–1] == 45)*. На это затрачиваются три операции. Истинность условия *c[ns2–1] == 45* завершает дихотомический поиск *break*. Ложное условие показывает, что необходимо продолжить поиск в правом интервале с помощью инструкции *n1 = z + 1*, на которую приходятся две операции:

$$W_{2,6\min} = 3;$$

$$W_{2,6\max} = 3 + 2 = 5.$$

Итак, минимальная оценка второго этапа с циклом *while*(*n1 <= n2*) получается тогда, когда выполняется одна итерация, т. е. результатом дихотомического поиска является число 4:

$$W_{2\min} = W_{2,0} + W_{2,1\min} + W_{2\,false} + W_{2,3\min} + W_{2,4\max} + W_{2,5\min} + W_{2,6\ \min} =$$

$$= 1 + 10 + 12nv + 1 + 7 + 12nv + 5 + 7 + 12nv + 3 =$$

$$= 34 + 36nv.$$

Максимальная оценка второго этапа получается тогда, когда выполняется $\left]\log_2 9\right[= 4$ итераций, т. е. результатом дихотомического поиска является число 0 или число 9:

$$W_{2\max} = 4\left(W_{2,0} + W_{2,1\max} + W_{2,2\,true} + W_{2,3\max} + W_{2,4\min} + W_{2,5\max} + W_{2,6\max}\right) =$$

$$= 4\left(1 + 11 + 21nv + 3 + 4 + 20nv + 3 + 4 + 20nv + 5\right) =$$

$$= 4\left(31 + 61nv\right) = 124 + 244nv.$$

Функция *DichMult*( ) завершена, можно записать минимальную и максимальную оценки ее выполнения:

$$W_{\min}^{DichMult} = W_1 + W_{2\min} = 2 + 34 + 36nv = 36 + 36nv;$$

$$W_{\max}^{DichMult} = W_1 + W_{2\max} = 2 + 124 + 244nv = 126 + 244nv.$$

Дихотомический поиск множителя для произвольных целых чисел. В некоторых случаях одноразрядный дихотомический поиск приходится выполнять, когда исходное число *a* содержит на одну цифру больше, чем число *b*, например $a = 12,\ b = 3,\ z = 4$, поскольку $b * z = 12$. Из арифметики следует, что число *a* может быть длиннее числа *b* не более, чем на одну цифру для одноразрядного множителя *z*. Если числа *a* и *b* произвольной длины при условии, что $b \le a$, то, выполняя одноразрядный дихотомический поиск и последовательно перебирая все цифры числа *a*, можно получить все цифры дихотомического множителя.

Дихотомический поиск множителя для произвольных чисел *a* и *b* выполняется в представленной далее функции *DMultiplier*( ). Алгоритм функции устроен таким образом, что сначала проверяется старшая цифра числа *a*. Если результат нулевой, то поиск осуществляется с двумя старшими цифрами числа *a*. Например, если $a = 12,\ b = 3$, то поиск с одной старшей цифрой 1 дает множитель $z = 0$. Следующий поиск с числом из двух цифр $a = 12$ дает правильный результат $z = 4$. В общем случае, продолжая перебирать все цифры числа *a*, получаем требуемый множитель *z*.

В функции *DMultiplier*( ) параметр *unsigned char* c* указывает на байтовый массив результата $b * z$, параметр *unsigned char* a* содер-

жит адрес байтового массива числа *a*, параметр *int na* содержит длину числа *a* с учетом знака. Аналогично параметры *unsigned char* b, int nb* содержат информацию числа *b*, параметр *unsigned char* s* предназначен для байтового массива остатка $a - b * z$. Функция *DMultiplier*( ) возвращает длину найденного дихотомического множителя:

```
// Файл DMultiplier.h (Win32)
// Дихотомический поиск множителя для произвольных целых чисел
int DMultiplier( unsigned char* c, unsigned char* a, int na,
                    unsigned char* b, int nb,
                      unsigned char* s )
{      unsigned char* u = new unsigned char[na];   // для цифр из a
       unsigned char* v = new unsigned char[na];   // для цифр из b
       unsigned char* w = new unsigned char[na];   // для дихотомии
       int na1 = na - nb;        // начало поиска со старших цифр a
       int na2 = na - 2;
       int nv = nb - 1;                          // длина b без знака
       for( int i = 0; i < nv; i++ )             // для начала поиска
       {      u[i] = a[na1+i];             // начальная порция из a
              v[i] = b[i];                     // исходный делитель
       }
       while( na1 >= 0 )                         // деление в столбик
       {      int dm = DichMult( u, v, nv, w, s );// дихотомия поиска
//            cout << "dm = " << dm << endl;
              c[na1] = dm;            // очередная цифра результата
              na1--;           // индекс следующей цифры делимого a
              if( na1 < 0 ) break;     // в делимом больше цифр нет
              for( int j = nv; j > 0; j-- ) u[j] = s[j-1];
              u[0] = a[na1];//следующая младшая цифра порции делимого
              v[nv++] = 0;               // выравнивание делителя v
              if( s[nv-2] == 0 ) nv = nv - 1;// длина порции делителя
       }
       delete [] u;               // освободить динамическую память
       delete [] v;
       delete [] w;
       return na - nb + 1;                       // длина c = b * dm
}
```

В программе *HI05*, приведенной ниже, выполняется дихотомический поиск множителя *c* для введенных чисел *a* и *b* произвольной длины. В подключаемом файле *#include "DMultiplier.h"* находится функция *DMultiplier*( ):

```
// Program HI05 (Win32)
// Дихотомический поиск целого множителя для произвольных чисел
#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"
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 );              // байтовая длина числа a
        int nb = strlen( sb );              // байтовая длина числа b
//        Дихотомический поиск целого множителя
        unsigned char c[256];             // для результата умножения
        unsigned char s[256];         // для промежуточного вычитания
        int nc = DMultiplier( c, a, na, b, nb, s );//поиск множителя
        cout << "c = "; ByteBitPrint( c, nc );          // результат
        cout << "s = "; ByteBitPrint( s, nb-1 );  // листинг остатка
        _getch();                                 // просмотр результата
}
```

Если при выполнении программы *HI05* ввести числа $a = 96918$ и $b = 999$, то на мониторе получается множитель $c = 97$ и остаток $s = 2$. Для просмотра поиска в функции *DMultiplier*( ) снят комментарий с инструкции *cout << "dm = " << dm << endl*:

```
sa = +96918
a = 00001000  00000001  00001001  00000110  00001001  00101011
sb = +999
b = 00001001  00001001  00001001  00101011
dm = 0
dm = 9
dm = 7
c = 00000111  00001001  00000000
s = 00000101  00000001  00000000
```

В теле функции *DMultiplier*( ) первые три инструкции позволяют выделить динамическую память для промежуточных вычислений. Не будем учитывать время их выполнения, поскольку в реальных условиях эта память может быть предоставлена заранее исходя из конкретных условий решаемых задач.

Оценка скоростных свойств функции *DMultiplier*( ) начинается на первом этапе с выполнения инструкций *int na1 = na – nb; int na2 = = na – 2; int nv = nb – 1*, в которых в общей сложности шесть операций:

$$W_1 = 6.$$

На втором этапе находится инструкция цикла для исходного копирования с заголовком *for(int i = 0; i < nv; i++)*, на выполнение которой затрачивается одна операция инициализации *int i = 0*:

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

В теле цикла на каждой итерации выполняется проверка условия $i < nv$ и приращение $i{+}{+}$, это занимает три операции:

$$W_{2,1} = \sum_{i=0}^{nv-1} 3 = 3nv.$$

Затем в теле цикла выполняются две инструкции $u[i] = a[na1+i]$; $v[i] = b[i]$, затрачивая семь операций:

$$W_{2,2} = \sum_{i=0}^{nv-1} 7 = 7nv.$$

В итоге на втором этапе затрачивается следующее количество операций:

$$W_2 = W_{2,0} + W_{2,1} + W_{2,2} = 1 + 3nv + 7nv = 1 + 10nv.$$

Третий этап занимает главный цикл поиска с заголовком *while* (*na1 >= 0*). В теле цикла на каждой итерации проверяется условие *na1 >= 0*. Число итераций определяется количеством цифр в числе *a* и в числе *b*, по которым осуществляется поиск дихотомического множителя:

$$W_{3,1} = \sum_{na1=0}^{na-nb} 1 = 1 + na - nb.$$

Далее в цикле поиска выполняется инструкция одноразрядного дихотомического поиска очередной цифры множителя *int dm = = DichMult*(*u, v, nv, w, s*). Оценка функции *DichMult*( ) была получена ранее. Учитывая операцию запоминания в *dm*, имеем следующий результат:

$$W_{3,2\min} = 1 + \sum_{na1=0}^{na-nb} W_{\min}^{DichMult} = 1 + \left(1 + na - nb\right)\left(36 + 36nv\right);$$

$$W_{3,2\max} = 1 + \sum_{na1=0}^{na-nb} W_{\max}^{DichMult} = 1 + \left(1 + na - nb\right)\left(126 + 244nv\right).$$

Полученная очередная цифра множителя запоминается в массиве $c[k{+}{+}] = dm$, а индекс старшей цифры сдвигается на одну позицию к младшим цифрам *na1--*. Всего потребуется выполнить шесть операций:

$$W_{3,3} = \sum_{na1=0}^{na-nb} 6 = 6\left(1 + na - nb\right).$$

Следует проверить, что все цифры числа *a* уже рассмотрены *if*(*na1 < 0*) *break* и множитель найден, тогда условие *na1 < 0* истинно:

$$W_{3,4} = \sum_{na1=0}^{na-nb} 1 = 1 + na - nb.$$

Если условие *na1 < 0* ложно, то необходимо увеличить порцию делимого на одну цифру. Для этого остаток порции делимого смеща-

ется на одну позицию с помощью цикла с заголовком *for*(*int j = nv; j > 0; j--*), затрачивается одна операция инициализации *int j = nv*:

$$W_{3,5,0} = \sum_{na1=0}^{na-nb} 1 = 1 + na - nb.$$

В теле этого цикла на каждой итерации выполняется одна операция проверки условия *j > 0* и две операции для декремента *j--* :

$$W_{3,5,1} = \sum_{na1=0}^{na-nb} \sum_{j=1}^{nv} (1+2) = 3nv(1 + na - nb).$$

Также на каждой итерации выполняется смещение элементов массива *u*[*j*] = *s*[*j–1*]:

$$W_{3,5,2} = \sum_{na1=0}^{na-nb} \sum_{j=1}^{nv} 4 = 4nv(1 + na - nb).$$

Всего на этот внутренний цикл уходит следующее количество операций:

$$W_{3,5} = W_{3,5,0} + W_{3,5,1} + W_{3,5,2} = (1 + 3nv + 4nv)(1 + na - nb) =$$
$$= (1 + 7nv)(1 + na - nb).$$

Остается добавить очередную младшую цифру в новую порцию делимого *u*[*0*] = *a*[*na1*] и выровнять делимое и делитель по количеству цифр *v*[*nv++*] = *0*. Последнее будет необходимо для функции вычитания байтовых чисел:

$$W_{3,6} = \sum_{na1=0}^{na-nb} (2+3) = 5(1 + na - nb).$$

Если в массиве *v* присутствует старший 0, то его следует убрать, используя *if*(*s[nv–2] == 0*) *nv = nv – 1*:

$$W_{3,7\,min} = \sum_{na1=0}^{na-nb} 3 = 3(1 + na - nb);$$

$$W_{3,7\,max} = \sum_{na1=0}^{na-nb} (3+2) = 5(1 + na - nb).$$

Итак, рассмотрены все инструкции третьего этапа в функции *DMultiplier*( ). Их общее количество определяется суммированием:

$$W_{3\,min} = W_{3,1} + W_{3,2\,min} + W_{3,3} + W_{3,4} + W_{3,5} + W_{3,6} + W_{3,7\,min} =$$

$$= (1 + (36 + 36nv) + 6 + 1 + (1 + 7nv) + 5 + 3)(1 + na - nb) =$$

$$= \left(53 + 43nv\right)\left(1 + na - nb\right);$$

$$W_{3\max} = W_{3,1} + W_{3,2\max} + W_{3,3} + W_{3,4} + W_{3,5} + W_{3,6} + W_{3,7\max} =$$

$$= \left(1 + \left(126 + 244nv\right) + 6 + 1 + \left(1 + 7nv\right) + 5 + 5\right)\left(1 + na - nb\right) =$$

$$= \left(145 + 251nv\right)\left(1 + na - nb\right).$$

В конце тела функции *DMultiplier*( ) присутствуют инструкции освобождения динамической памяти *delete* [ ] *u; delete* [ ] *v; delete* [ ] *w*. При оценке скоростных свойств их можно не учитывать, поскольку эту память всегда можно зарезервировать заранее, исходя из конкретных условий решаемых задач.

Таким образом, оценка скоростных свойств дихотомического поиска множителя определяется скоростью выполнения функции *DMultiplier*( ). Минимальная оценка $W_{\min}^{DMultiplier}$ получается, если одноразрядный множитель равен четырем:

$$W_{\min}^{DMultiplier} = W_1 + W_2 + W_{3\min} =$$

$$= 6 + \left(1 + 10nv\right) + \left(53 + 43nv\right)\left(1 + na - nb\right) =$$

$$= 7 + 10nv + \left(53 + 43nv\right)\left(1 + na - nb\right).$$

Максимальная оценка $W_{\max}^{DMultiplier}$ дихотомического поиска множителя получается, если одноразрядный поиск все время находит цифру 0 или цифру 9:

$$W_{\max}^{DMultiplier} = W_1 + W_2 + W_{3\max} =$$

$$= 6 + \left(1 + 10nv\right) + \left(145 + 251nv\right)\left(1 + na - nb\right) =$$

$$= 7 + 10nv + \left(145 + 251nv\right)\left(1 + na - nb\right).$$

Рассматривая определение функции *DichMult*( ) и параметры обращения к ней в программе *HI04*, имеем $nv = nb - 1$. Подставляя это выражение в предыдущие формулы, получаем окончательные оценки скоростных свойств функции *DMultiplier*( ):

$$W_{\min}^{DMultiplier} = 7 + 10\left(nb - 1\right) + \left(53 + 43\left(nb - 1\right)\right)\left(1 + na - nb\right) =$$

$$= 7 + 43nb + 10na + 43nb\left(na - nb\right);$$

$$W_{\max}^{DMultiplier} = 7 + 10\left(nb - 1\right) + \left(145 + 251\left(nb - 1\right)\right)\left(1 + na - nb\right) =$$

$$= 109 + 367nb - 106na + 251nb\left(na - nb\right).$$

Результаты показывают, что с возрастанием количества цифр *na* и *nb* в исходных числах *a* и *b* скорость вычисления дихотомического множителя *z* в соотношении $a \le b * z$ в основном зависит от произведения $nb(na - nb)$, если $na \ne nb$.

СПИСОК ЛИТЕРАТУРЫ

1. Окулов С.М. Основы программирования. М.: Лаборатория базовых знаний, 2002. 424 с.
2. 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