УДК 681.142.2(075.8)

М. Ю. Барышникова, А. Ф. Деон,
А. В. Силантьева

СКОРОСТНЫЕ СВОЙСТВА АЛГОРИТМОВ СЛОЖЕНИЯ И ВЫЧИТАНИЯ ЦЕЛЫХ ЧИСЕЛ ПРОИЗВОЛЬНОГО РАЗМЕРА

Приведено описание подходов к оценке быстродействия алгоритмов, реализующих операции сложения и вычитания целых чисел произвольной размерности, на основе подсчета числа 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: deonalex@mail.ru

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

Операции компьютерного процессора стандартизованы выполнением действий с двоичными целыми числами размером n байт, где n — разрядность процессора. Количество различимых чисел находится в диапазоне $[0, 2^n - 1]$, т.е. от 0 до 4294967296 при 32-разрядном процессоре или от 0 до 18446744073709551616 — при 64-разрядном.

Этого достаточно для выполнения почти всех прикладных задач. В исключительных ситуациях, когда требуется оперировать числами большего размера, необходима реализация алгоритмических операций, которые выполняются процессором компьютера по специальным программам. К числу таких операций относятся операции сложения и вычитания целых чисел произвольного размера. Оценка быстродействия алгоритмов реализации этих операций и составляет суть данного исследования. В качестве языка программирования используется исторический C. Такой выбор был сделан исходя из понимания того, что такие диалекты этого языка, как CLR (common language runtime) или C#, предоставляют аналогичные реализации, но с возможностями объектно-ориентированного программирования.

Для представления целых чисел произвольного размера можно использовать битовую модель из нескольких байт, учитывая при сложении или вычитании возникающий особый характер переноса бит как внутри байт, так и между байтами. В другой модели предусматривается расположение каждой цифры целого числа в отдельном байте, представляя все число как массив цифр-байтов [1, 2]. В данной работе принят подход с представлением каждой десятичной цифры числа в отдельном байте. Поскольку при реализации операции сложения возможно появление дополнительной единицы переноса (например,

$5 + 9 = 14$), то используется обратный порядок расположения десятичных цифр в массиве, что упрощает реализацию сложения простым заполнением единицы в следующем старшем байте результата. Аналогично, поскольку в процессе вычитания положительных чисел возможно появление отрицательного результата (например, $(+5) - -(+9) = -4$), старший байт массива числа должен содержать код знака '+' или '–'.

Ниже представлены функции: *BinaryByte*( ) — для двоичного представления цифр числа; *BinaryPrint*( ) — для печати байтового массива числа, а также *BynaryMass*( ), в которой символьный массив числа копируется в байтовый массив, представляя число в обратном порядке:

```
void BinaryByte( unsigned char z )
{ unsigned char b = 0x80;
while( b > 0 )
{ ( z & b ) == 0 ? cout « '0' : cout « '1';
b »= 1;
}
}
void BinaryMass( 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
}
void BinaryPrint( unsigned char* u, int nu )
{ for( int i = 0; i < nu; i++ ) //печать u
{ BinaryByte( u[i] );
cout « " ";}
cout « endl;
}
```

Прежде чем непосредственно выполнять операции сложения или вычитания двух чисел, необходимо, чтобы они содержали одинаковое количество цифр, например,

(+15) + (+09) = +24. Ниже приведена программа AI03, в которой функции *Align*( ) и *Align2*( ) выполняют выравнивание количества цифр в двух числовых массивах, дополняя нули в старших разрядах числа с меньшим количеством цифр:

```
// Program AI03 (Win32)
// Выравнивание байтовых чисел в обратном порядке
```

```
#include <conio.h> //_getch
#include <iostream>
using namespace std;
#include <string.h> //strlen
#include "BinaryByte.h"
#include "BinaryPrint.h"
#include "BinaryMass.h"
inline int BinaryAlign( unsigned char* z, int nz, int k )
{ int n = nz + k; //новое количество байт в числе z
unsigned char s = z[nz-1]; //запомнить знак числа
for( int j = nz-1; j < n-1; j++ ) z[j] = 0; //выравнивание
z[n-1] = s; //приписать знак числа
return n;
}
int BinaryAlign2( unsigned char* u, int nu,
unsigned char* v, int nv )
{ int k = nu - nv; //число выравнивающих нулей
if( k > 0 ) //в числе a больше цифр
return BinaryAlign( v, nv, k ); //выравнивание числа v
if( k < 0 ) //в числе b больше цифр
return BinaryAlign( u, nu, -k ); //выравнивание числа u
return nu; //равное количество цифр в числах u, v
}
//————————————————————————
void main( void )
{ char sa[256]; //символьный буфер целого числа a
cout « "sa = ";
cin » sa; //символьный ввод числа a
unsigned char a[256];
BinaryMass( a, sa ); //преобразование в байтовый вид
int na = strlen( sa ); //количество цифр в числе a
BinaryPrint( a, na ); //печать a
char sb[256]; //символьный буфер целого числа b
cout « "sb = ";
cin » sb; //символьный ввод числа b
unsigned char b[256];
BinaryMass( b, sb ); //преобразование в байтовый вид
int nb = strlen( sb ); //количество цифр в числе b
BinaryPrint( b, nb ); //печать b
int m = BinaryAlign2( a, na, b, nb ); //выравнивание чисел
na = m;
```

```
nb = m;
BinaryPrint( a, na ); //печать a
BinaryPrint( b, nb ); //печать b
_getch(); //просмотр результата
}
```

Если после запуска программы AI03 ввести два числа +5 и +999, то получим результат выравнивания +005 и +999:

```
sa = +5
00000101 00101011
sb = +999
00001001 00001001 00001001 00101011
00000101 00000000 00000000 00101011
00001001 00001001 00001001 00101011
```

Оценка скоростных свойств [3] выравнивания двух чисел определяется количеством операций над целыми числами, выполняемых в теле функции *BinaryAlign*2( ). На первом этапе выполняется инструкция *int k = nu – nv*, в которой две операции (из перечня машинных команд сложения, вычитания, загрузки, сохранения и сравнения) вычитание ‘–‘ и сохранение ‘=’. Обозначим это как

$$W_1^{Aligm2} = 2.$$

Затем следует второй этап, на котором выполняется инструкция условия *if*$(k > 0)$ *return BinaryAlign* (v, nv, k), содержащая одну операцию сравнения $k > 0$ и инструкцию выполнения функции *BinaryAlign*( ). Объявление функции *BinaryAlign*( ) содержит модификатор *inline*, который реализуется компилятором как непосредственная подстановка тела функции, а не вызов с параметрами:

$$W_2^{Align2} = 2 + A.$$

Количество операций A в функции *Align*( ) определяется ее телом:

```
int n = nz + k; //новое количество байт в числе z
unsigned char s = z[nz-1]; //запомнить знак числа
int n1= n-1;
for( int j = nz-1; j < n1; j++ ) z[j] = 0; //выравнивание
z[n1] = s; //приписать знак числа
```

На выполнение инструкции $intn = nz + k$ затрачиваются две операции.

$$W_1^{Align} = 2.$$

Также по две операции занимают инструкции *unsigned char* $s = z[nz-1]$ с учетом расположения элемента в массиве z и инструкция $int\ n1 = n-1$:

$$W_2^{Allign} = 4.$$

На выполнение инструкции цикла $for\ (intj = nz-1;\ j < n1;\ j++)z[j] = 0$ приходится две операции для $int\ j = nz-1$, одна операция $j < n1$, и на каждой итерации выполняются: одна операция: проверки условия '<', две операции на увеличение и запоминание индекса $j++$, две операции на определение адреса элемента $z[j]$ и запоминания 0:

$$W_3^{Align} = 3 + \sum_{j=nz-1}^{n-1}(1+2+2) =$$
$$= 3 + 5(n-1-(nz-1)) = 3 + 5(n-nz).$$

Если выравнивать байтовый массив числа не следует, то $n = nz$. Максимальное выравнивание происходит, если $nz = 2$ и $n > 2$. Это дает возможность найти минимальную $A_{\min}$ и максимальную оценку выполнения функции *Align*( ):

$$A_{\min} = 3; \quad A_{\max} = 3 + 5(n-1).$$

Продолжим скоростной анализ второго этапа алгоритма функции *Align*2( ):

$$W_2^{Align2} = 4 + 3 + 5(n-2) = 7 + 5(n-2) = 5n - 3.$$

На последнем, третьем, этапе функции *Align*2( ) выполняется инструкция условия $if\ (k < 0)$ *return BinaryAlign* (v, nv, k). Анализ быстродействия данного этапа полностью совпадает с оценкой W_2^{Align2}:

$$W_3^{Align2} = 5n - 3.$$

Итак, функция *Align*2( ) делает минимальное количество операций, когда выравнивание не выполняется. Максимальное количество операций наступает, когда число из одной цифры дополняется в старших разрядах до длины второго числа, участвующего в арифметической операции:

$$A_{\min} = W_1^{Align2} + W_{2.\text{условие}}^{Align2} + W_{3.\text{условие}}^{Align2} = 3 + 1 + 1 = 5;$$

$$A_{\max} = W_1^{Align2} + W_{2.\text{условие}}^{Align2} + W_{3.\text{условие}}^{Align2} = 3 + 1 + 5n - 3 = 1 + 5n.$$

Сложение чисел без учета знаков. Выравнивание чисел по количеству цифр позволяет реализовать операцию сложения байтовых чисел. Учет единиц переноса в старшие разряды можно реализовать

с помощью проверки результата. Если результат сложения больше, чем 9, то результирующую цифру получают после вычитания числа 10, а единицу переноса прибавляют к результату сложения в следующем разряде. Если по окончании сложения всех цифр существует последняя единица переноса, то массив результирующего числа увеличивается на байт, в который записывается эта единица переноса. Этот подход реализован в функции *BinaryAdd*( ), приведенной ниже, которая возвращает количество цифр в полученном результате:

```
// Беззнаковое сложение байтовых чисел в обратном порядке
#include "BinaryAlign.h" //выравнивание байтовых целых чисел
int BinaryAdd( unsigned char* c,
unsigned char* a, int* pna,
unsigned char* b, int* pnb )
{ int na = *pna; //длина массива a
int nb = *pnb; //длина массива b
int m = BinaryAlign2( a, na, b, nb ); //выравнивание чисел
*pna = m; //число байт со знаком в слагаемом a
*pnb = m; //число байт со знаком в слагаемом b
unsigned char tranzit = 0; //единица переноса
int m1=m-1;
for( int i = 0; i < m1; i++ ) //цикл сложения байт
{ unsigned char d = a[i] + b[i] + tranzit;
if( d < 10 ) { c[i] = d; tranzit = 0; }
else { c[i] = d - 10; tranzit = 1; }
}
if( tranzit == 0 ) c[m1] = 43; //знак +
else { c[m1] = 1; c[m++] = 43; }//1 в новом старшем разряде
return m;
}
```

Если после запуска программы, включающей вызов этой функции, ввести числа $a = +5$ и $b = +999$, то на мониторе отображается байтовый массив числа $c = a + b = = (+5) + (+999) = +1004$:

```
sa = +5
00000101 00101011
sb = +999
00001001 00001001 00001001 00101011
00000101 00000000 00000000 00101011
00001001 00001001 00001001 00101011
00000100 00000000 00000000 00000001 00101011
```

Скоростные свойства функции *BinaryAdd*( ) определяются количеством операций над целыми числами внутри тела функции. На первом этапе выполняются инструкции $int\ na = *pna$; $int\ nb = *pnb$; $m =$ *BinaryAlign*$2(a, na, b, nb)$; $*pna = m$; $*pnb = m$; *unsigned char tranzit* = 0. Общее количество операций на данном этапе можно оценить как

$$W_{1\min}^{BinaryAdd} = 2 + 2 + A_{\min} + 2 + 2 = 4 + 5 + 4 = 13;$$
$$W_{1\max}^{BinaryAdd} = 2 + 2 + A_{\max} + 2 + 2 = 4 + 1 + 5n + 4 = 9 + 5n.$$

На втором этапе выполняется инструкция цикла:

```
Int m1=m-1;
for( int i = 0; i < m1; i++ ) //цикл сложения байт
{ unsigned char d = a[i] + b[i] + tranzit;
if( d < 10 ) { c[i] = d; tranzit = 0; }
else { c[i] = d - 10; tranzit = 1; }
}
```

До цикла выполняется инструкция $int\ m1 = m-1$, для которой требуются две операции. В заголовке цикла выполняется одна инструкция $int\ i = 0$. На каждой итерации выполняется одна операция проверки условия $i < m1$ и две операции на увеличение и запоминание индекса $i++$. Кроме того, в каждой итерации затрачивается пять операций на реализацию поразрядного сложения *unsigned char* $d = a[i] + b[i] + + transit$. Инструкция альтернативного условия $if(d < 10)$ $\{c[i] = d$; *tranzit* = 0; $\}$ $else\{c[i] = d - 10$; *tranzit* = 1; $\}$ занимает одну операцию на проверку условия. При истинности условия выполняются три операции на реализацию $c[i] = d$; *tranzit* = 0. При ложном условии выполняются четыре операции на вычисление $c[i] = d-10$; *tranzit* = 1. Таким образом, на втором этапе выполняется следующее количество операций:

$$W_{2\min}^{BinaryAdd} = 3 + \sum_{i=0}^{m-2}(3 + 5 + 1 + 3) = 3 + 12(m-1) = 12m - 9;$$
$$W_{2\max}^{BinaryAdd} = 3 + \sum_{i=0}^{m-2}(3 + 5 + 1 + 4) = 3 + 13(m-1) = 13m - 10.$$

На третьем этапе выполняется инструкция альтернативного условия:

```
if( tranzit == 0 ) c[m1] = 43; //знак +
else { c[m1] = 1; c[m++] = 43; } //1 в новом старшем разряде
```

Проверка условия *tranzit* == 0 занимает одну операцию. Если условие истинно, то выполняют две операции на реализацию выражения $c[m1] = 43$. При ложном условии выполняются пять операций для $c[m1] = 1$; и $c[m++] = 43$. Таким образом, на третьем этапе выполняется следующее количество операций:

$$W_{3\min}^{BinaryAdd} = 1 + 2 = 3;$$
$$W_{3\max}^{BinaryAdd} = 1 + 5 = 6.$$

Подводя итог оценки скоростных свойств функции сложения беззнаковых чисел, можно записать следующие формулы для чисел в n байт:

$$BA_{\min} = W_{1\min}^{BinaryAdd} + W_{2\min}^{BinaryAdd} + W_{3\min}^{BinaryAdd} =$$
$$= 13 + 12n - 9 + 3 = 7 + 12n;$$
$$BA_{\max} = W_{1\max}^{BinaryAdd} + W_{2\max}^{BinaryAdd} + W_{3\max}^{BinaryAdd} =$$
$$= 9 + 5n + 13n - 10 + 6 = 5 + 18n.$$

Вычитание чисел без учета знаков. Поразрядное вычитание чисел без учета знаков $c = a - b$ дает правильный результат, если $a > b$, например, $(+9) - (+5) = +4$. Необходимо предусмотреть заимствование из старших разрядов, если некоторый разряд в числе a меньше соответствующего разряда в числе b.

В случае, когда $a < b$, результат вычитания $c = a - b$ становится отрицательным, например, $(+5) - (+9) = -4$. Чтобы воспользоваться алгоритмом поразрядного вычитания, необходимо взять дополнение каждой цифры в результате поразрядного вычитания до ближайшего целого числа, кратного 10. Это означает, что занимаемая единица для старшего разряда не находится в числе a. Именно знак ‘–‘ результата указывает на недостаточное количество единиц, что можно определить через дополнение поразрядного вычитания. Реализацию этого алгоритма показывает функция *BinarySubtract*( ), которая выполняет вычитание чисел без учета знаков, выдавая на выходе соответствующий положительный или отрицательный результат. Функция *BinarySubtract*( ) возвращает количество байт в массиве результата $c = a - b$:

```
// Беззнаковое вычитание байтовых чисел в обратном порядке
#include "BinaryAlign.h" //выравнивание байтовых целых чисел
int BinarySubtract( unsigned char* c,
unsigned char* a, int* pna,
unsigned char* b, int* pnb )
{ int na = *pna; //длина массива a
```

```
int nb = *pnb; //длина массива b
int m = BinaryAlign2( a, na, b, nb ); //выравнивание чисел
*pna = m; //число байт со знаком в числе a
*pnb = m; //число байт со знаком в числе b
unsigned char tranzit = 0; //занимаемая единица
int m1=m-1;
for( int i = 0; i < m1; i++ ) //цикл вычитания байт
{ signed char d = a[i] - tranzit - b[i];
if( d < 0 ) { d = 10 + d; tranzit = 1; }
else tranzit = 0;
c[i] = d;
}
if( tranzit == 0 ) //положительный результат
{ c[m1] = 43; //знак '+'
return m;
}
c[0] = 10 - c[0]; //формирование отрицательного результата
for( int i = 1; i < m1; i ++ ) c[i] = 9 - c[i];
c[m1] = 45; //знак '-'
return m;
}
```

Если в программе ввести числа $a = +9$ и $b = +1005$, то на мониторе будет отображен результат вычитания $c = a - b = (+9) - -(+1005) = -996$:

```
sa = +9
00001001 00101011
sb = +1005
00000101 00000000 00000000 00000101 00101011
00001001 00000000 00000000 00000000 00101011
00000101 00000000 00000000 00000101 00101011
00000110 00001001 00001001 00000000 00101101
```

Скоростные свойства функции *BinsrySubtract*( ) определяются количеством операций над целыми числами внутри тела функции. На первом этапе выполняются инструкции $int\ na = *pna$; $int\ nb = *pnb$; $m =$ *BinaryAlign*2(*a, na, b, nb*); $*pna = m$; $*pnb = m$; *unsigned char* $tranzit = 0$. Общее количество операций на данном этапе можно оценить как

$$W_{1\min}^{BinarySubtract} = 2 + 2 + A_{\min} + 2 + 2 = 4 + 5 + 4 = 13;$$

$$W_{1\max}^{BinarySubtract} = 2 + 2 + A_{\max} + 2 + 2 = 4 + 1 + 5n + 4 = 9 + 5n.$$

На втором этапе выполняется инструкция цикла:

```
int m1=m-1;
for( int i = 0; i < m1; i++ ) //цикл вычитания байт
{ signed char d = a[i] - tranzit - b[i];
if( d < 0 ) { d = 10 + d; tranzit = 1; }
else tranzit = 0;
c[i] = d;
}
```

До цикла выполняется одна инструкция, реализуемая за две операции $int\ m1 = m - 1$. В заголовке цикла выполняется одна инструкция $int\ i = 0$. На каждой итерации выполняются одна операция проверки условия $i < m1$ и две операции на увеличение и запоминание индекса $i++$. Кроме того, в каждой итерации используются пять операций на реализацию поразрядного сложения *signed char d = a[i]- tranzit - b*$[i]$. Инструкция альтернативного условия $if\ (d < 0)\ \{d = 10 + d;$ *tranzit* = 1; $\}$ *else tranzit* = 0 занимает одну операцию на проверку условия. При истинности условия выполняются три операции на реализацию $d = 10 + d;$ *tranzit* = 1. При ложном условии выполняется одна операция *tranzit* = 0. В конце тела цикла находится инструкция $c[i] = d$ с выполнением в две операции. Таким образом, на втором этапе выполняется следующее количество операций:

$$W_{2\min}^{BinarySubtract} =$$

$$= 3 + \sum_{i=0}^{m-2} (3 + 5 + 1 + 1 + 2) = 3 + 12(m-1) = 12m - 9;$$

$$W_{2\max}^{BinarySubtract} =$$

$$= 3 + \sum_{i=0}^{m-2} (3 + 5 + 1 + 3 + 2) = 3 + 14(m-1) = 14m - 11.$$

На третьем этапе выполняется инструкция условия $if(tranzit == 0)$ $\{c[m1] = 43;\ returnm;\}$. При истинности условия *tranzit* == 0 реализуется одна операция. При ложном условии выполняются три операции:

$$W_{3\min}^{BinarySubtract} = 1;$$
$$W_{3\max}^{BinarySubtract} = 3.$$

На четвертом этапе происходит учет транзитной единицы для отрицательного результата:

```
c[0] = 10 - c[0]; //формирование отрицательного результата
for( int i = 1; i < m1; i++ ) c[i] = 9 - c[i];
c[m1] = 45; //знак '-'
```

В инструкции $c[0] = 10 - c[0]$ используются две операции. В заголовке цикла выполняется одна операция для *int i* = 1. На каждой итерации затрачивается по одной операции на проверку условия $i < m1$ и по две операции на увеличение индекса $i++$. В теле цикла необходимы три операции на выполнение инструкции $c[i] = 9 - c[i]$. По окончании цикла необходимо выполнить две операции для реализации инструкции $c[m1] = 45$:

$$W_4^{BinarySubtract} = 2 + 1 + \sum_{i=1}^{m-2} (1 + 2 + 3) + 2 = 5 + 6(m - 2) = 6m - 1.$$

Итак, для выполнения функции *BinarySubtract( )* необходимо следующее количество операций:

$$BS_{\min} =$$

$$= W_{1\min}^{BinarySubtract} + W_{2\min}^{BinarySubtract} + W_{3\min}^{BinarySubtract} + W_4^{BinarySubtract} =$$

$$= 13 + 12n - 9 + 1 + 6n - 1 = 4 + 18n;$$

$$BS_{\max} =$$

$$= W_{1\max}^{BinarySubtract} + W_{2\max}^{BinarySubtract} + W_{3\max}^{BinarySubtract} + W_4^{BinarySubtract} =$$

$$= 9 + 5n + 14n - 11 + 3 + 6n - 1 = 25n - 1.$$

Сложение произвольных целых чисел. В операции сложения могут участвовать как положительные, так и отрицательные числа. Сама операция обозначена явно как '+'. Знаки чисел скрыты в старших байтах соответствующих массивов. В табл. 1 показано, какую следует производить операцию над беззнаковыми числами $c = a + b$, если известны знаки чисел a и b.

Таблица 1

Выполняемые действия при сложении в зависимости от знаков слагаемых

a	b	Беззнаковая операция
+	+	$c = a + b$
+	–	$c = a - b$
–	+	$c = b - a$
–	–	$c = -(a + b)$

Ниже приведена функция *BigIntAdd*( ), которая управляет процессом сложения произвольных целых чисел с произвольными знаками

слагаемых:

```
// Сложение байтовых чисел в обратном порядке
#include "BinaryAlign.h" //выравнивание байтовых целых чисел
#include "BinaryAdd.h"
#include "BinarySubtract.h"
int BigIntAdd( unsigned char* c,
unsigned char* a, int* pna,
unsigned char* b, int* pnb )
{ int na = *pna; //длина числа a
int nb = *pnb; //длина числа b
int nc = 0;
if( a[na-1] == '+' && b[nb-1] == '+' ) //a >= 0, b >= 0
return nc = BinaryAdd( c, a, pna, b, pnb ); //c = a + b
if( a[na-1] == '-' && b[nb-1] == '-' ) //a < 0, b < 0
{ nc = BinaryAdd( c, a, pna, b, pnb ); //c = a + b
c[nc-1] = '-'; //c = -( a + b )
return nc;
}
if( a[na-1] == '+' && b[nb-1] == '-' ) //a >= 0, b < 0
return nc = BinarySubtract( c, a, pna, b, pnb );//c = a - b
if( a[na-1] == '-' && b[nb-1] == '+' ) //a <0 0, b >= 0
return nc = BinarySubtract( c, b, pnb, a, pna );//c = b - a
return nc;
}
```

Если ввести числа $a = +9$ и $b = -1005$, то на мониторе появится результат сложения $c = a + b = (+9) + (-1005) = 9 - 1005 = -0996$, полученный по алгоритму вычитания беззнаковых чисел:

```
sa = +9
00001001 00101011
sb = -1005
00000101 00000000 00000000 00000101 00101101
00001001 00000000 00000000 00000000 00101011
00000101 00000000 00000000 00000101 00101101
00000110 00001001 00001001 00000000 00101101
```

Скоростные свойства алгоритма функции сложения *BigIntAdd*( ) определяются набором выполняемых инструкций в теле функции. На первом этапе выполняются первые три инструкции $int\ na = *pna$;

$int\ nb = *pnb$; $int\ nc = 0$, которые используют три операции над целыми числами. Их количество обозначим как

$$W_1^+ = 3.$$

На втором этапе, выполняемом по инструкции выбора

```
if( a[na-1] == '+' && b[nb-1] == '+' ) //a >= 0, b >= 0
return nc = BinaryAdd( c, a, pna, b, pnb ); //c = a + b
```

затрачиваются четыре операции для $a[na-1] ==$ '+', затем четыре операции для $b[nb-1] ==$ '+' и одна операция для булевой дизъюнкции &&. Если условие истинно, то на выполнение операций внутри функции *BinaryAdd*(*c, a, pna, b, pnb*) используется количество операций, равное BA_2. Если условие функции выбора ложно, то функция *BinaryAdd*( ) не выполняется. Итак, при ложном или истинном условии осуществляется следующее количество операций:

минимальное

$$W_{2\min}^+ = 9;$$

максимальное

$$W_{2\max}^+ = 9 + BA_2.$$

Если на втором этапе не выполняется сложение, то следует третий этап по инструкции выбора:

```
if( a[na-1] == '-' && b[nb-1] == '-' ) //a < 0, b < 0
{ nc = BinaryAdd( c, a, pna, b, pnb ); //c = a + b
c[nc-1] = '-'; //c = -( a + b )
return nc;
}
```

Вычисление условия осуществляется, как и на предыдущем этапе, за девять операций. При истинности условия выполняется функция *BinaryAdd*(*c, a, pna, b, pnb*) за BA_3 операций. Инструкция присваивания $c[nc-1] =$ '–' применяет две операции. Итак, при ложном или истинном условии используется минимальное или максимальное количество операций соответственно:

$$W_{3\min}^+ = 9; \quad W_{3\max}^+ = 9 + BA_3 + 2 = 11 + BA_3.$$

Если на третьем этапе не выполняется сложение, то по инструкции выбора следует четвертый этап:

```
if( a[na-1] == '+' && b[nb-1] == '-' ) //a >= 0, b < 0
return nc = BinarySubtract( c, a, pna, b, pnb );//c = a - b
```

Анализ этого этапа совпадает с анализом второго этапа. В итоге получаем:

$$W^{+}_{4\min} = 9; \quad W^{+}_{4\max} = 9 + BS_4.$$

Если на четвертом этапе не выполняется сложение, то следует последний, пятый, этап по инструкции выбора:

```
if( a[na-1] == '-' && b[nb-1] == '+' ) // a <0 0, b >= 0
return nc = BinarySubtract( c, b, pnb, a, pna );// c = b - a
```

Анализ пятого этапа совпадает с анализом второго или четвертого этапа. В итоге получаем:

$$W^{+}_{5\min} = 9; \quad W^{+}_{5\max} = 9 + BS_5.$$

В целом минимальное количество операций на выполнение функции сложения целых чисел произвольного размера определяется на втором этапе:

$$W^{+}_{\min} = W^{+}_{1\min} + W^{+}_{1} + W^{+}_{2\min} = 3 + 9 + BA_2 =$$
$$= 3 + 9 + BA_{\min} = 3 + 9 + 7 + 12n = 19 + 12n.$$

Максимальное количество операций будет в случае, если проверки условий приведут к последнему пятому этапу:

$$W^{+}_{\max} = W^{+}_{1} + W^{+}_{2.\text{условие}} + W^{+}_{3.\text{условие}} + W^{+}_{4.\text{условие}} + W^{+}_{5\max} =$$
$$= 3 + 9 + BA_2 = 3 + 9 + 9 + 9 + 9 + BS_5 =$$
$$= 39 + BS_{\max} = 39 + 25n - 1 = 38 + 25n.$$

Вычитание произвольных целых чисел. В операции вычитания могут принимать участие положительные и отрицательные числа. Непосредственно операция обозначается явно как ‘–‘. Знаки чисел скрыты в старших байтах соответствующих массивов. В табл. 2 показано, какую следует производить операцию над беззнаковыми числами $c = a - b$, если известны знаки чисел a и b.

Таблица 2

Выполняемые действия при вычитании в зависимости от знаков операндов

a	b	Беззнаковая операция
+	+	$c = a - b$
+	–	$c = a + b$
–	+	$c = -(b + a)$
–	–	$c = a - b$

Представленная ниже функция *BigIntSubtract*( ) управляет процессом вычитания произвольных целых чисел с произвольными знаками:

```
// Вычитание байтовых чисел в обратном порядке
#include "BinaryAlign.h" //выравнивание байтовых целых чисел
#include "BinaryAdd.h"
#include "BinarySubtract.h"
int BigIntSubtract( unsigned char* c,
unsigned char* a, int* pna,
unsigned char* b, int* pnb )
{ int na = *pna; //длина числа a
int nb = *pnb; //длина числа b
int nc = 0;
if( a[na-1] == '+' && b[nb-1] == '+' ) //a >= 0, b >= 0
return nc = BinarySubtract( c, a, pna, b, pnb ); //c=a-b
if( a[na-1] == '-' && b[nb-1] == '+' ) //a < 0, b >= 0
{ nc = BinaryAdd( c, a, pna, b, pnb ); //c = a + b
c[nc-1] = '-'; //c = -( a + b )
return nc;
}
if( a[na-1] == '+' && b[nb-1] == '-' ) //a >= 0, b < 0
return nc = BinaryAdd( c, a, pna, b, pnb ); //c = a + b
if( a[na-1] == '-' && b[nb-1] == '-' ) //a < 0, b < 0
return nc = BinarySubtract( c, b, pnb, a, pna ); //c=b-a
return nc;
}
```

При вводе чисел $a = -9$ и $b = -1005$ на мониторе отобразится результат вычитания $c = a - b = (-9) - (-1005) = 1005 - 9 = +0996$, выполненный по алгоритму вычитания беззнаковых чисел:

```
sa = -9
00001001 00101101
sb = -1005
00000101 00000000 00000000 00000101 00101101
00001001 00000000 00000000 00000000 00101101
00000101 00000000 00000000 00000101 00101101
00000110 00001001 00001001 00000000 00101011
```

Скоростные свойства алгоритма функции вычитания *BigIntSubtract*( ) определяются набором выполняемых инструкций в теле функции. На первом этапе первые три инструкции $int\ na = *pna$;

$int\ nb = *pnb$; $int\ nc = 0$ используют три операции над целыми числами. Их количество обозначим как

$$W_1^- = 3.$$

На втором этапе, выполняемом по инструкции выбора

```
if( a[na-1] == '+' && b[nb-1] == '+' ) //a >= 0, b >= 0
return nc = BinarySubtract( c, a, pna, b, pnb ); //c=a-b
```

используются четыре операции для выражения $a[na-1] ==' +'$, затем четыре операции для $b[nb-1] ==' +'$ и одна операция для булевой дизъюнкции &&. Если условие истинно, то на выполнение операций внутри функции *BinarySubtract*(*c, a, pna, b, pnb*) затрачивается количество операций, равное BS_2. Если условие функции выбора ложно, то функция *BinarySubtract*( ) не выполняется. Итак, при ложном или истинном условии осуществляется следующее количество операций соответственно:

минимальное

$$W_{2\min}^+ = 9;$$

максимальное

$$W_{2\max}^+ = 9 + BS_2.$$

Если на втором этапе не выполняет вычитание, то следует третий этап по инструкции выбора:

```
if( a[na-1] == '-' && b[nb-1] == '+' ) //a < 0, b >= 0
{ nc = BinaryAdd( c, a, pna, b, pnb ); //c = a + b
c[nc-1] = '-'; //c = -( a + b )
return nc;
}
```

Вычисление условия занимает, как и на предыдущем этапе, девять операций. Если условие истинно, то выполняется сложение *BinaryAdd*( ) и занесение знака $c[nc-1] =' -'$. Таким образом, при ложном или истинном условии используется, соответственно, следующее количество операций:

минимальное

$$W_{3\min}^- = 9;$$

максимальное

$$W_{3\max}^- = 9 + 2 + BA_3.$$

Если на третьем этапе не выполняется сложение, то по инструкции выбора следует четвертый этап:

```
if( a[na-1] == '+' && b[nb-1] == '-' ) //a >= 0, b < 0
return nc = BinaryAdd( c, a, pna, b, pnb ); //c = a + b
```

Анализ этого этапа совпадает с анализом предыдущих этапов. В итоге получаем:

$$W^-_{4\min} = 9; \quad W^-_{4\max} = 9 + BA_4.$$

Если на четвертом этапе не выполняется сложение, то следует последний, пятый, этап по инструкции выбора:

```
if( a[na-1] == '-' && b[nb-1] == '+' ) //a <0 0, b >= 0
return nc = BinarySubtract( c, b, pnb, a, pna ); //c = b - a
```

Анализ пятого этапа совпадает с анализом предыдущих этапов. В итоге получаем:

$$W^-_{5\min} = 9; \quad W^-_{5\max} = 9 + BS_5.$$

В целом минимальное количество операций на выполнение функции вычитания целых чисел произвольного размера определяется на втором этапе:

$$W^-_{\min} = W^-_1 + W^-_{2\min} = 3 + 9 + BS_2 = 12 + BS_{\min} = \\ = 3 + 9 + 4 + 18n = 16 + 18n.$$

При вычитании целых чисел произвольного размера количество операций будет максимальным в случае, если проверки условий приведут к последнему, пятому, этапу:

$$W^-_{\max} = W^-_1 + W^-_{2.\text{условие}} + W^-_{3.\text{условие}} + W^-_{4.\text{условие}} + W^-_{5\max} = \\ = 3 + 9 + 9 + 9 + 9 + BS_5 = 39 + BS_{\max} = 39 + 25n - 1 = 38 + 25n.$$

Проведенный анализ показал, что при сложении и вычитании целых чисел произвольной размерности несмотря на очевидный линейный характер зависимости количества элементарных операций от разрядности числа n коэффициенты при n оказывают существенное влияние на быстродействие алгоритмов реализации данных операций. Таким образом, преимущества удобства и простоты реализации аддитивных операций над “длинными” числами, представленными в виде одномерных массивов, в которых каждая цифра числа занимает один байт, могут нивелироваться дополнительными временными затратами на выполнение указанных действий.

СПИСОК ЛИТЕРАТУРЫ

1. О к у л о в С. М. Основы программирования. – М.: Лаборатория базовых знаний, 2002. – 424 с.
2. О к у л о в С. М. Дискретная математика: Теория и практика решения задач по информатике: Учеб. пособие. – М.: БИНОМ; Лаборатория базовых знаний, 2008. – 422 с.
3. С е д ж в и к Р. Функциональные алгоритмы на С++: Анализ: Структуры данных: Сортировка: Поиск: Пер. с англ. – СПб.: ООО “ДианаСофЮП”, 2002. – 688 с.

Статья поступила в редакцию 10.05.2012