EasyChair Preprint

№ 5157

Arithmetic Operations on Multiple Byte Integers and an Introduction to Multiple Byte Fractions

Ishraga Mustafa Awad Allam

May 17, 2021

Arithmetic Operations on Multiple Byte Integers and an Introduction to Multiple Byte Fractions

Ishraga Mustafa Awad Allam

University of Khartoum, Sudan

IshragaAllam@Gmail.com OR imallam@uofk.edu

Abstract

Big integers are very essential in many applications. Cryptography is one of these applications. There is an introduction to multiple byte fractions which can do good for accountants. In this study, the objective is to create a multiple byte integer type, with its arithmetic operations defined. The operations are: addition, subtraction, multiplication, division and modular exponentiation are overloaded, to work on this multiple byte integer type.

The creation of the multiple byte integer is done by using doubly linked lists, a well known technique in data structure. The reason is that doubly linked lists enable us to create integer of unlimited size. That is, you do not have to pre-specify the size of the arrays storing these integers.

This is done by dynamically allocating the memory to store the digits constructing the integers.

The operations on these integers are defined using the simple and straight forward techniques, learnt in school.

The results obtained are satisfactory and reliable. The type could be extended to help define multiple byte floating point numbers.

In this work, an improvement has been made to the work of BH Flowers.

Introduction:

The computer consists of bits and bytes. The important thing in computer is its word size which can be data or some computer operation, represented by an integer for everything. The largest integers we have been able to work with have been confined to short, an int or a long. In C++ language, for example, these are one byte, two bytes and four bytes long, respectively. For all types, advanced arithmetical operations are made standard for only that domain of type, otherwise the compiler just ignore the function output, no overflow message.

In principle, however, there is no reason to confine an integer to a specific number of bytes: the concept of a list allows us to work with any number of bytes, dynamically determined according to the transient needs of the program.

Integers limit in size made files vulnerable to attacks, even thought they were encrypted, because the size of the private key is limited. Also in real life, the size of amounts of large integers is limited to just more than 4 millions.

Multiple Byte Integers

The lists we have discussed so far have been singly linked: each node contains a single pointer to the succeeding node in the list. Multiple byte integers, however, are not efficiently dealt with by means of single linkages: we shall find it necessary to scan a list both forwards and backwards, and this is most easily done if each node contains a pointer to the previous node as well as a pointer to the succeeding node. Morever, it is no longer desirable to work with pointers to the elements of a fixed data array: since we do not know in advance the number of bytes required by an integer, it would be most inconvenient to work with fixed arrays; we need to work with the bytes themselves.

An 8-bit byte, unsigned, can accommodate the integers 0, 1, …, 255. If we set B = 256, we may represent any non-negative integer N as a polynomial in B:

$$N = a_m B_m + \ldots + a_1 B + a_0,$$

Or $N = (a_m, \ldots, a_1, a_0)B,$

Where, in the second formula, we have recognized B as the radix, and the integer coefficients a_i, which must obey the restriction $0 < a_i < B$, as the digits of the integer to the base B. There may in principle be any number digits. If B were equal to 10, we should have the usual decimal representation, but we shall take B to be 256.

If we define a digit to be a node containing *a* byte b and two pointers, a multiple byte integer (which we shall call mult) may be represented by a list of digits. In diagrammatic form as shown in figure 3.1.[1]:

Computer Operations with Integers:

Before computers were invented mathematicians did computations either by hand or by using mechanical devices. Either way, they were only able to work with integers of limited size. Many number theoretic computations, such as factoring and primality testing, require computations with integers with as many as 50 or even 100 digits. In this section we will study some of the basic algorithms for doing computer arithmetic. Then we will study the number of basic computer operations required to carry out these algorithms.

It had been mentioned that computers internally represent numbers using bits, or binary digits. Computers have a built-in limit on the size of integers that can be used in machine arithmetic. This upper limit is called the word size, which we denote by w. The word size is usually a power of 2, such as 2^{35}, although sometimes the word size is a power of 10[5].

To do arithmetic with integers larger than the word size, it is necessary to devote more than one word to each integer. To store an integer n > w, we express n in base w notation, and for each digit of this expansion we use one computer word. For instance, if the word size is 2^{35}, using ten computer words we can store integers as large as 2^{350} - 1, since integers less than 2^{350} have no more than ten digits in their base 2^{35} expansions. Also note that to find the base 2^{35} expansion of an integer, we need only group together blocks of 35 bits[5].

The first step in discussing computer arithmetic with large integers is to describe how the basic arithmetic operations are methodically performed.

We will describe the classical methods for performing the basic arithmetic operations with integers in base r notation where r > 1 is an integer. These methods are examples of algorithms (The word "algorithm" has an interesting history, as does the evolution of the concept of an algorithm. "Algorithm" is a corruption of the original term"algorism" which originally comes from the book Kitab Al Jabr w'Al-Maqabala{Rules of Restoration and Reduction} written by Abu Ja'far Mohammed ibn Musa Al-Khowarizmi, in the ninth century. The word "algorism" originally referred only to the rules of performing arithmetic using Arabic numerals. The word "algorism" evolved into "algorithm" by the eighteenth century. With growing interest in computing machines, the concept of an algorithm became more general, to include all definite procedures for solving problems, not just the procedures for performing arithmetic with integers expressed in Arabic notation).

Definition: An algorithm is a specified set of rules for obtaining a desired result from a set of input[5].

We will describe for performing addition, subtraction, and multiplication of two n-digit integers $a = (a_{n-1}\, a_{n-2} \ldots a_1\, a_0)_r$ and $b = (b_{n-1}\, b_{n-2} \ldots b_1\, b_0)_r$. The algorithms described are used both for binary arithmetic with integers less than the word size of a computer, and for multiple precision arithmetic with integers larger than the word size w, using w as the base[5].

The implementation of asymmetrical cryptographic schemes often requires the use of numbers that are many times larger than the integer data types that are supported natively by the compiler. In this article, we give an introduction to the implementation of arithmetic operations involving large integers.

Implementation Details: The most common way of representing numbers is by using the positional notation system. Numbers are written using digits to represent multiples of powers of the specified base. The base that we are most familiar with and use every day, is base 10. When we write the number 12345 in base 10, it actually means

$$12345_{10} = 1x10^4 + 2x10^3 + 3x10^2 + 4x10^1 + 5x10^0$$

Similarly, if the same number is specified in base 16, then

$$12345_{16} = 1x16^4 + 2x16^3 + 3x16^2 + 4x16^1 + 5x16^0$$

Hence, it is important to know the base that the number is specified in, since the same representation could represent different values when interpreted in different bases. In general, a positive number can be written as a polynomial of order k, where k is non-negative and a_k != 0.

$$n = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b^1 + a_0$$

b represents the base with $0 <= a <= b-1$.

For base 10 representation, b = 10 and the digits allowed at each position is $0 <= a <= 9_4$. But to get full advantage of these multiple byte integers, as the weight is set as byte, the best radix (b) is 256 and the digits allowed at each node is $0 <= a <= 255$[1].

Addition of Multiple Byte Integers:

The addition of two numbers can be represented as the addition of two lists (polynomials) as shown below.

Let

$$n = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b^1 + a_0 \quad m = c_k b^k + c_{k-1} b^{k-1} + \ldots + c_1 b^1 + c_0$$

Then

$n + m = (a_k + c_k)b^k + (a_{k-1} + c_{k-1})b^{k-1} + ... + (a_1 + c_1)b^1 + (a_0 + c_0)$[1]

They don't have to be equal in length; i.e., n has k while m has j, and k != j. But the radix is the same, for all arithmetical operations done.

However, it is often easier to visualize the addition of two large numbers as the digit-by-digit addition at each position. When we add the digits at a particular position, the largest resulting value is 2b – 2[4].

Proof: Since the maximum value of each digit is b - 1, we have

$$(b - 1) + (b - 1) = 2b - 2$$

Since 2b - 2 < 2b, the maximum value that we have to carry over to the next higher position is 1. For example, if we add 9 and 9 in base 10, we get 18, where 8 remains in the current position and the 1 gets *carried on* to the next higher position. When adding the two digits in the next position, we must remember to add the carry as well4.

We first discuss the algorithm for addition. When we add *a* and b, we obtain the sum

$$a + b = \sum_{j=0}^{n-1} a_j r^j + \sum_{j=0}^{n-1} b_j r^j = \sum_{j=0}^{n-1} (a_j + b_j) r^j,$$

To find the base r expansion of $a + b$, first note that by the division algorithm, there are integers C_0 and s_0 such that

$$a_0 + b_0 = C_0 r + s_0, 0 \leq s_0 < r.$$

Because a_0 and b_0 are positive integers not exceeding r, we know that $0 \leq a_0 + b_0 \leq 2r - 2$, so that $C_0 = 0$ or 1;

here C_0 is the carry to the next place. Next, we find that there are integers C_1 and s_1 such that

$$a_1 + b_1 + C_0 = C_1 r + s_1, 0 \leq s_1 < r.$$

Since $0 \leq a_1 + b_1 + C_0 \leq 2r - 1$, we know that $C_1 = 0$ or 1. Proceeding inductively, we find integers C_i and s_i for $1 \leq I \leq n - 1$ by

$$a_i + b_i + C_{i-1} = C_i r + s_i, 0 \leq s_i < r,$$

with $C_i = 0$ or 1. Finally, we let $s_n = C_{n-1}$, since the sum of two integers with n digits has n + 1 digits when there is a carry in the n^{th} place. We conclude that the base r expansion for the sum is $a + b = (s_n s_{n-1} \ldots s_1 s_0)_r$.

When performing base r addition by hand, we can use the same familiar technique as is used in decimal addition[5].

The two number lists must be traversed from right to left; i.e., from the smallest weight, adding the two corresponding nodes and the carry digit, what is greater than the radix is then added to the next nodes sum. Every addition result is inserted into a node of the resultant number list. Finally if there is also a carry, a node is again inserted and attached at the front of the list representing the sum of the two number lists.

This is precisely what happens in the definition of the operator+().

```
mult& mult::add(mult ms1, mult ms2)
    {
        int a;                  // The value of adding the two digits.
        int carry = 0;          // The carry of if the sum of the two digits is greater than the radix, initially is zero.
        mult ms3; // The product list of addition.
                multscan thisptr(ms1, NEXT);   // Traverse from right to left, the first is the most smallest weight
                dgtp pr1;
                multscan thatptr(ms2, NEXT);
                dgtp pr2;
    while(((pr1 = thisptr()) != 0) && ((pr2 = thatptr()) != 0))
        // While both are equal in length.
                {
        a = int(pr1 -> dgt) + int(pr2 -> dgt) + carry;
        if(((pr1 -> dgt) + (pr2 -> dgt) + carry) >= radix)
                                {
                                        carry = a / radix;
                                    a = a % radix;
                                        }
                                else carry = 0;
                                ms3.append(a);
                                }
            if(ms1.numdgt > ms2.numdgt)  // If I didn't do this loop, the next digit of the first list will be lost.
                                {
                                a = int(pr1 -> dgt) + carry;
```

```
if(((pr1 -> dgt) + carry) >= radix)
                                        {
carry = a / radix;
a = a % radix;
                                        }
                    else carry = 0;
                    ms3.append(a);
                    }
 while((pr1 = thisptr()) != 0)      // While the first list is greater in length than the second list.
                    a = int(pr2 -> dgt) + carry;
                    if(a >= radix)
                                        {
                    carry = a / radix;
                    a = a % radix;
                                        }
                                  else carry = 0;
                    ms3.append(a);
                    }
    if(carry >= 1)  ms3.append(carry);    // If the last digit addition carry is not zero, an attached node.
                    // Is appended to the product list.
  if((ms1.sgn == -1) && (ms2.sgn == +1))    // Checking if the two lists, doesn't have the same sign.
                    {
                    ms1.sgn = ms2.sgn = +1;  // Changing the signs first.
                    ms3 = ms2 - ms1;
                    ms1.sgn = -1;
                // The product will have a negative sign.
                    cout << "  The Result value : ";
                    printmult(ms3);
                    exit(0);
                    }
    if((ms1.sgn == +1) && (ms2.sgn == -1))
 // Checking if the two lists, doesn't have the same sign.
                    {
                    ms1.sgn = ms2.sgn = +1;
                // Changing the signs first.
                    ms3 = ms1 - ms2;
          // A subtraction is done here.
                    ms2.sgn = -1;
          // The product will have a negative sign.
                    cout << "  The Result value : ";
                    printmult(ms3);
                    exit(0);
                    }
          cout << "\n The addition : \n  ";
          rs = ms3;          // I need two variables, because sometimes in an addition, I also perform a subtraction.
          printmult(ms3);  // One is local and the other is global to return the result.
          return rs;
         }
```

The inline operator does the inline + operator between two multiple-Byte integers, where m1 is the first number list, m2 is the second number list, pa is a temporary resultant number list.

```
inline mult operator + (mult m1, mult m2) {    return (pa.add(m1, m2)); }
```

The lists in number1 and number2 are traversed from right to left using list <Byte int> reverse iterators that are moved through the lists synchronously by using the increment operator. The carry digit and the blocks at each position of the iterator are added, the new carry digit and sum block are calculated, and the sum block inserted at the front of the list in sum using list's insert() operator.

Sign Test for Addition:

a. If the sign of the first number list is positive and the sign of the second number list is positive, a normal addition will be done with the resultant sign, will be positive.

b. If the sign of the first number list is negative and the sign of the second number list is positive, a subtraction operation is done using the subtraction operator, after treating that both signs are positive and the first number list will be the second number list for the subtraction operation; i.e., reversing numbers**.**

c. If the sign of the first number list is positive and the sign of the second number list is negative, a subtraction operation is done after treating that both signs are positive.

d. The length of numbers can be tested here as addition is no good as subtraction for the subtraction has a fixed rule the state:

(i). If first's sign is negative and second's sign is negative for for subtraction, the resultant sign will be negative and the operation will be subtraction.

(ii). If first's sign is negative and second's sign is positive for subtraction, the resultant sign will be negative and the operation will be addition.

(iii). If first's sign is positive and second's sign is negative for subtraction, the resultant sign will be positive and the operation will be addition.

(iv). If first's sign is positive and second's sign is positive for subtraction, the resultant operation will depend on the length of each of the number lists size. If the second length is greater, the resultant sign will be negative otherwise it will be positive.

.Subtraction of Multiple Byte Integers:

The subtraction of two numbers is very similar to addition and can be easily implemented as byte-by-byte subtraction. However, when subtracting two digits, we often encounter the situation where the first digit is smaller than the second. In this case, we have to borrow 1 from the digit in the next position. This is actually done by subtracting 1 from the digit in the next higher position and adding the value of the base to the current digit[4].

We consider

$$a - b = \sum_{j=0}^{n-1} a_j r^j - \sum_{j=0}^{n-1} b_j r^j = \sum_{j=0}^{n-1} (a_j - b_j) r^j,$$

Where we assume that $a > b$. Note that by the division algorithm, there are integers B_0 and d_0 such that

$$a_0 - b_0 = B_0 r + d_0, 0 \leq d_0 < r,$$

and since a_0 and b_0 are positive integers less than r, we have

$$-(r - 1) \leq a_0 - b_0 \leq r - 1.$$

When $a_0 - b_0 \geq 0$, we have $B_0 = 0$. Otherwise, when $a_0 - b_0 < 0$, we have $B_0 = -1$; B_0 is the borrow, from the next place of the base r expansion of a. We use the division algorithm again to find integers B_1 and d_1 such that

$$a_1 - b_1 + B_0 = B_1 r + d_1, 0 \leq d_1 < r.$$

From this equation, we see that the borrow $B_1 = 0$ as long as $a_1 - b_1 + B_0 \geq 0$, and $B_1 = -1$ otherwise, since $-r \leq a_1 - b_1 + B_0 \leq r - 1$. We proceed inductively to find integers B_i and d_i, such that

$$a_i - b_i + B_{i-1} = B_i r + d_i, 0 \leq d_i \leq r$$

with $B_i = 0$ or -1, for $1 \leq i \leq n - 1$. We see that B_{n-1}. We see that $B_{n-1} = 0$, since $a > b$. We can conclude that

$$a - b = (d_{n-1} d_{n-2} \ldots d_1 d_0)_r.$$

When performing base r subtraction by hand, we use the same familiar technique as is used in decimal subtraction[5].

The two number lists must be traversed from right to left; i.e., from the smallest weight, subtracting the two corresponding nodes and the borrow digit if the result is negative, will be taken a radix in size from the exceeding first number digit; i.e., will be subtracted from the next subtraction of the other two corresponding digits.

```
mult& mult::sub(mult mt1, mult mt2)
    {
    int borrow = 0;   // The borrow if the first list digit was smaller than the second list digit, initially is zero.
    int at;// The value of subtracting the two digits.
        mult mt3, buf;          // The buf is needed when the second list length is greater than the first list.
        multscan thisptr(mt1, NEXT);   //  Traverse from right to left, the first is the most smallest weight
        dgtp pt1;
        multscan anotherptr(mt2, NEXT);
        dgtp pt2;
while(((pt1 = thisptr()) != 0) && ((pt2 = anotherptr()) != 0))
                {
            at = int(pt1 -> dgt) - int(pt2 -> dgt) - borrow;
    if((pt1 -> dgt) < (pt2 -> dgt))
                        {
                at = at + radix;
                borrow = 1;
                                }
                        else borrow = 0;
                        mt3.append(at);
            }
        if(mt1.numdgt > mt2.numdgt)
                {
                at = int(pt1 -> dgt) - borrow;
                if(at == 0)
                                {
                                        at = at
                + radix;
                        borrow = 1;
                                }
        else borrow = 0;
                        if(at <= 0) borrow = 1;
                        mt3.append(at);
                }
        while((pt1 = thisptr()) != 0)  //  While the first list is greater in length than the second list.
                {
                at = int(pt1 -> dgt) - borrow;
                if(at <= 0) borrow = 1;
                mt3.append(at);
                }
        mt3.sgn = +1;
if((mt1.sgn == -1) && (mt2.sgn == -1)) mt3.sgn = -1;
    // The product will have a negative sign.
                        mt3 = rs;
                }
        if((mt1.sgn == +1) && (mt2.sgn == -1))  // Checking if the two lists, doesn't have the same sign.
                {
                mt1.sgn = mt2.sgn = +1;
            // Changing the signs first.
                mt3 = mt1 + mt2;
    // Addition is done here.
                mt3.sgn = +1;
            // The product will have a positive sign.
                mt3 = rs;
                }
```

```
                if(mt1.numdgt < mt2.numdgt)  //  If the second list length is greater than the first list. SWAP
                        {
                                mt3 = mt2 - mt1;
                                mt3.sgn = -1;
        // The product will have a negative sign.
                                mt3 = rs;
                        }
        cout << "\n The subtraction : \n  ";
        rs = mt3;
        printmult (mt3);
        return rs;
}
```

Sign Test for Subtraction:

If the sign of the first number list is negative and the sign of the second number list is positive, an addition operation is done after treating both signs as positive. The resultant sign will be set to negative.

a. If both signs were negative, then the sign of the resultant is also negative.

b. If the sign of the first number list is positive and the sign of the second number list is negative, an addition operation is done after treating both signs as positive. The sign of the resultant is set to positive.

c. If the length of the first number list is less than that of the first, then a message will appear and the numbers will be swapped; i.e., the first will be the second for the subtraction operation. The resultant will have a negative sign.

d. If the length of the first number list is less than that of the first, then a message will appear and the numbers will be swapped; i.e., the first will be the second for the subtraction operation. The resultant will have a negative sign.

Multiplication of Multiple Byte Integers:

The simplest way to implement multiplication is by using repeated addition. To compute a * b, we set the result to 0, then we repeatedly add a to the result for b number of times. Although this approach is simple, its complexity is O(b) large integer additions and is equivalent to O(bn) byte-by-byte additions, where n is the number of digits in the large integer. For large b, O(bn) > $O(n^2)$. Hence, this approach does not scale to large multipliers.

The next easiest approach is by using techniques that we have learnt in elementary schools. More precisely, we multiply a by the each digit of b and sum the partial results[4]. It is clear that the time complexity of multiplying two large integers is $O(n^2)$ byte multiplications where n is the number of digits in the number. The process of adding the partial results has a complexity of O(n) large integer additions, equivalent to $O(n^2)$ byte additions. Thus, the total complexity is $O(n^2)$ byte multiplications and $O(n^2)$ byte additions[4]. But before discussing multiplication, we describe shifting. To multiply $(a_{n-1} \ldots a_1 a_0)_r$ by r^m, we need only shift the expansion left m places, appending the expansion with m zero digits.

Example: To multiply $(101101)_2$ by 2^5, we shift the digits to the left five places and append the expansion with five zeros, obtaining $(10110100000)_2$. □

To deal with multiplication, we first discuss the multiplication of an n-place integer by a one-digit integer. To multiply $(a_{n-1} \ldots a_1 a_0)_r$, we first note that

$a_0 b = q_0 r + p_0$, $0 \le p_0 < r$, and $0 \le q_0 \le r - 2$, since $0 \le a_0 b \le (r-1)^2$. Next, we have

$a_1 b + q_0 = q_1 r + p_1$, $0 \le p_1 < r$, and $0 \le q_1 \le r - 1$. In general, we have

$a_i b + q_{i-1} = q_i r + p_i$, $0 \le p_i < r$ and $0 \le q_i \le r - 1$. Furthermore, we have $p_n = q_{n-1}$. This yields $(a_{n-1} \ldots a_1 a_0)_r (b)_r = (p_n p_{n-1} \ldots p_1 p_0)_r$.

To perform a multiplication of two n-place integers we write

$$ab = a \sum_{j=0}^{n-1} b_j r^j = \sum_{j=0}^{n-1} (ab_j) r^j,$$

For each j, we first multiply a by the digit b_1, then shift to the left j places, and finally add all of the n integers we have obtained to find the product.

We multiplying two integers with base r expansion, we use the familiar method of multiplying decimal integers by hand[5].

Again the two number lists were traversed from right to left, taking the first digit of the second number list and multiply it with each digit of the first number list. Each multiplication if greater than the radix, a carry digit will be added to the next proceeding nodes' result. A node will be created when the last digit of the first number list is reach and there was a carry, at the front of the list. The next multiplication of the next digit of the second number list will be done in the same way except that a zero digit will be inserted first at the most right node of the result. The next other ones, every time an increment is done; i.e., two zero's will be inserted in the most right nodes of the result, and so one, using an iterator. Every resultant of each multiplication of a digit of the second number list is added to the one before it, and the sum will be the final resultant of the multiplication operation. The addition operator will be used here, but without the sign test. Both signs will be treated as positive.

The Sign Test results only in the final resultant sign, and if the sign of the two numbers lists were different, then the resultant sign will be negative, otherwise it is positive.

```
mult& mult::mul(mult m1, mult m2)
  {
        int carrym;
        long unsigned int x, count = 0;
        long int ssum;
        mult mmp, mb;        // mb to store each last addition of the multiplications done.
        multscan anotherptr(m2, NEXT);// Traverse from right to left, the first is the most smallest weight
        dgtp p2;
        while((p2 = anotherptr()) != 0)
// For every digit of the second list, it multiple all the first list.
        {
        carrym = 0;
        mult ma;  // Each time, a new initiated variable list is needed.
  for(x = 0; x < count; x++)  ma.append(0);  //  Shifting the first leading zeros.
        multscan otherptr(m1, NEXT);
        dgtp p1;
        while((p1 = otherptr()) != 0)
        {
ssum = int(p1 -> dgt) * int(p2 -> dgt) + carrym;
        if(ssum >= radix)
        {
        carrym = ssum / radix;
        ssum = ssum % radix;
        }
    else carrym = 0;
 ma.append(ssum);
        }
if(carrym >= 1) ma.append(carrym);        // Attaching the last carry to the product list.
if(count == 0) mmp = ma;        //  Initiale addition to the multiplication.
        else
// Else adding every last multiplications to the next one.
 {
        mmp = ma + mb;
        mmp = rs;
        }
        count++;
// The shifting of the leading zeros is incremented.
        mb = mmp;
// The value of storing additions, is updated.
```

```
                printmult(mmp);

                }

            mmp.sgn = +1;

        if((m1.sgn == -1) && (m2.sgn == +1))    mmp.sgn = -1;
                        // Only the sign will be negative.

         if((m1.sgn == +1) && (m2.sgn == -1))   mmp.sgn = -1;

         if((m1.sgn == -1) && (m2.sgn == -1))    mmp.sgn = +1;

          cout << "\n The multiplication : \n  ";

          rs = mmp;

          printmult(mmp);

          return rs;

    }
```

Division of Integers:

We now discuss integer division. We wish to find the quotient q in the division algorithm

$$a = bq + R,\ 0 \le R < b.$$

If the base r expansion of q is $q = (q_{n-1}\, q_{n-2} \ldots q_1\, q_0)_r$, then we have

$$a = b \sum_{j=0}^{n-1} q_j r^j + R,\ 0 \le R < b.$$

To determine the first digit q_{n-1} of q, notice that

$$a - bq_{n-1}\, r^{n-1} = b \sum_{j=0}^{n-2} q_j r^j + R$$

The right-hand side of this equation is not only positive, but also less than br^{n-1},

Since $\sum_{j=0}^{n-2} q_j r^j \le \sum_{j=0}^{n-2} (r-1) r^j = \sum_{j=0}^{n-1} r^j - \sum_{j=0}^{n-2} r^j = r^{n-1} - 1$.

Therefore, we know that

$$0 \le a - bq_{n-1}\, r^{n-1} < br^{n-1}.$$

This tells us that

$$q_{n-1} = [a / (br^{n-1})].$$

We can obtain q_{n-1} by successively subtracting br^{n-1} from a until a negative result is obtained, and then q_{n-1} is one less than the number of subtractions.

To find the other digits of q, we define the sequence of partial remainders R_i by $R_0 = a$ and

For i = 0, this is clearly correct, since $R_0 = a = qb + R$. Now assume that

$$R_k = \sum_{j=0}^{n-k-1} q_j r^j\ b + R.$$

$$R_i = R_{i-1} - bq_{n-1}\, r^{n-i}$$

For i = 1, 2, …, n. By mathematical induction, we show that

$$R_i = \sum_{j=0}^{n-i-1} q_j r^j\ b + R.$$

Then

$$R_{k+1} = R_k - bq_{n-k-1}\, r^{n-k-1}$$

$$= \sum_{j=0}^{n-k-1} q_j r^j\ b + R - bq_{n-k-1}\, r^{n-k-1}$$

$$= \sum_{j=0}^{n-(k+1)-1} q_j r^j\ b + R.$$

establishing *, which by it, we see that $0 \le R_i < r^{n-i}\, b$, for i = 1, 2, …, n, since $\sum_{j=0}^{n-i-1} q_j r^j \le r^{n-i} - 1$. Consequently, since $R_i = R_{i-1} - bq_{n-i}\, r^{n-i}$ and $0 \le R_i < r^{n-1}\, b$, we see that the digit q_{n-i} is given by $[R_{i-1} / (br^{n-i})]$ and can be obtained by successively subtracting br^{n-i} from R_{i-1} until a negative result is obtained, and then q_{n-i} is one less than the number of subtractions. This is how we find the digits of q[5].

Again the two number lists must be traversed from right to left, subtracting the two corresponding digits. Here I'm not using the subtraction operator because I need a flag, which after each subtraction if there was a negative result, the flag is set to positive, indicating the end of the operation. The second number list is then subtracted from the resultant of the last subtraction until the flag is positive. The number of these subtractions while the flag is negative is the final resultant, been converted to a list itself; i.e., each time this iterator which count these subtraction is reach the radix size, a node of the final resultant is created and the modulo of the division of this iterator is inserted into a new node, creating the final division number list. The iterator was declared as long as to be able to present as large as possible for a really big integers as it can be for the resultant.

```
mult& mult::dvd(mult mr1, mult mr2)

  {

        int borrow = 0, d2, d1;

        long unsigned int n = 0;

        int ar, ps = 0;

        mult mr3, mdval, buff = mr1;

        if(mr2.numdgt > buff.numdgt)  //  If  the
second list length was greater than the first list.

                {
```

```
fp = 1;      // This is a flag to show that the result will contain floating points.
            cout << "\n Float division not integers \n";
            mr3 = mdval = 0;      // The result will be zero.
            ps = 1;               // To terminate this operation, and not to go to next loop.
            n = 0;
            }
while(ps == 0)
            {                     // Traverse from right to left, the first is the most smallest weight
                        multscan thisptr(mr1, NEXT);
                          dgtp ptr1;
                          multscan anotherptr(mr2, NEXT);
                dgtp ptr2;        // I didn't use the subtraction operator because I needed a flag.
          while(((ptr1 = thisptr()) != 0) && ((ptr2 = anotherptr()) != 0))
                                    {
                                    d2 = int(ptr2 -> dgt);
                                    d1 = int(ptr1 -> dgt);
                                    ar = d1 - d2 - borrow;
                                    if((ptr1 -> dgt) < (ptr2 -> dgt))
                        {
                        ar = ar + radix;
            if(ptr1 -> next == 0) ar = 0;
            borrow = 1;
            }
                                    else borrow = 0;
                                    ptr1 -> dgt = ar;
                        if(ar < 0) ps = 1;      // The flag is on if the result is negative.
            }
            if(mr1.numdgt > mr2.numdgt)    // When the first list is greater than the second.
                        {  // Not to lost the next digit of the first list.
                                    ar = int(ptr1 -> dgt) - borrow;
                                    if((ptr1 -> next != 0) && (ar == 0))
                                    {
                                                ar = ar + radix;
                                                if(ptr1 -> next == 0) ar = 0;
                                                borrow = 1;
                                    }
                         else borrow = 0;
                        if(ar <= 0) borrow = 1;
                ptr1 -> dgt = ar;    // Change the first list digit.
                        if(ar < 0) ps = 1;
                        }
while((ptr1 = thisptr()) != 0)       // When the first list is greater than the second.
 {
                        ar = int(ptr1 -> dgt) - borrow;
                        ptr1 -> dgt = ar;
                         borrow = 0;
                        }
if(fp == 1)                  // A suggested loop for Floating Point operation.
  {
                        mult buf;
                        multscan thisptr(mr1, NEXT);
                        dgtp ptrf;
                              buf.sgn = mr2.sgn;
                              buf.append(0);          // Shifting the first list to increase it.
                              while((ptrf = thisptr()) != 0)
                                  buf.append(int(ptrf -> dgt));
                                   mr1 = buf;
```

```
                    cout << "Fractional mult is ";
                    printmult(mr1);
                    //if(n == 65000) ps = 1;
                    }
                n = n + 1;          // Counting
the number of subtractions.
          if((mr1.numdgt == mr2.numdgt) && (d2 > ar))  ps = 1;  //
Floating Point.
               if((buff.numdgt == mr2.numdgt) && (d2 > d1))

               {
                    ps = 1;
                    n = 0;
                                   }
          if(ar <= 0) mr1.numdgt = mr1.numdgt - 1;          //
If last digit is less than zero,
          if(mr1.numdgt == 0) ps = 1;                       //
then the length of it decreases.
          if(mr1.numdgt < mr2.numdgt) ps = 1;
          // Floating Point result...
                    }
               mdval = mr1;
                 mdval.sgn = +1;                            //
          The sign of the modulo is always positive.
               pr = mdval;
               mr1 = buff;
          while(n > 0)          // Converting the
          decimal number of the counting into a
          multiple Byte List.
          {
                    ar = n % radix;
                    n = n / radix;
               mr3.append(ar);
               }
```

```
                    mr3.sgn = +1;
     if((mr1.sgn == -1) && (mr2.sgn == +1)) mr3.sgn = -1;//
Only the sign changes.
   if((mr1.sgn == +1) && (mr2.sgn == -1)) mr3.sgn = -1;
      if((mr1.sgn == -1) && (mr2.sgn == -1)) mr3.sgn = +1;
                    cout << "\n The module : \n  ";
                    //if(fp == 1) mdval = 0;
                    printmult (mdval);
                    cout << "\n The dividor : \n  ";
                    rs = mr3;
                    if(fp == 1)          // A suggested
               loop for Floating Point operation.
                         {
               cout << "  This is a fraction : 0.";
                         for(n  =  0;  n  <  fc;  n++)
               cout << "0";
                         }
               printmult(mr3);
               if(fp  ==  1)  cout  <<  "  Sorry  but  this
               fraction  can  not  be  returned.  This
               opens up a new research. \n";
               return rs;
          }
```

A test is done first to see if the second number length of nodes is greater than the first. If it is so, a message will appear, telling that

there will be a Floating-Point result, which is another research to be done. If both numbers' lengths are equal, traverse from left to right, here is the advantage or 'use' of the doubly-linked list structure reason of being there, to check if the second number is greater than the first; again there is the message of a Floating-Point research.

The Sign Test results only in the final resultant sign, and if the signs of the two numbers' lists were different, then the resultant sign will be negative, otherwise it is positive. Just like multiplication.

Exponential Operation:

Here the first number list is multiplied by itself, using the multiplication operator, according to the second number digits times. The second number is traverse from right to left, each digit is multiplied by the radix raised to its weight, and this result number control the number of iteration of the multiplication of the first number list. This number if it is greater that the maximum of integers for the mathematic function then there will be a problem as these functions were built on integers, log or pow, which raise the radix to its coefficient.

```
mult& mult::expo(mult me1, mult me2)
{       long unsigned int x, count = 0;
        mult me3;
        me3 = me1;
        multscan thatptr(me2, NEXT);
        // Traverse from right to left, the first is the most smallest weight
        dgtp pe2;
        while((pe2 = thatptr()) != 0)

        {
                for(x = 0; x < ((int(pe2 -> dgt) * (exp((count) * (log(radix))))) - 1); x++)
                me3 = me3 * me1;
                count++;
        }
        if(me2.sgn == -1)
        {
                me3 = 0;
                cout << "This is a fractional, beyond this research \n";
        }
        cout << "\n The exponential : \n ";
        rs = me3;
        printmult(me3);
        return rs;
    }
```

Warning: The second number digit if it is zero and length one, is trivially known that any number, raised to zero is one. No need to do it here.

The second number sign can not be negative as Floating-Point numbers are outside this scope of research.

Comparison Operation:

Here, the lengths of the two numbers are compared, if their signs are the same. The larger is the big one.

```
mult& mult::cmp(mult mc1, mult mc2)
  {
        int pr = 0, d2, d1;
        mult mc3;
          multscan (mc2, LAST);          // Traverse from left to right, the first is the most greatest weight
          dgtp p2;                       // Here shows why it was a doubly-linked list
          multscan otherptr(mc1, LAST);
          dgtp p1;
                if((mc1.sgn == +1) && (mc2.sgn == -1))
                                        {
pr = ptrgt = 1;
// A flag so as not to perform the next loop.
```

```
                mc3 = mc1;
// The greatest will be the first list.
   }
if((mc1.sgn == -1) && (mc2.sgn == +1)
                        {
                                pr = ptrlt = 1;
// A flag of the less than is set on.
                                        mc3 = mc2;
// The greatest will be the second list.
                        }
            if((mc1.numdgt < mc2.numdgt) && (mc1.sgn == mc2.sgn)
                    {
                                pr = ptrlt = 1;
                                mc3 = mc2;
                    }
                if((mc1.numdgt > mc2.numdgt) && (mc1.sgn == mc2.sgn))
                    {
                                pr = ptrgt = 1;
                                mc3 = mc1;
                    }
            if((mc1.numdgt == mc2.numdgt) && (mc1.sgn == mc2.sgn))
                    {
 while(((p1 = otherptr()) != 0) && ((p2 = anotherptr()) != 0))
                                        {
                    d2 = int(p2 -> dgt);
                    d1 = int(p1 -> dgt);
                cout << "D1 " << d1 << "  D2 " << d2 << "\n";
             if(d2 > d1)
            {
                    pr = ptrlt = 1;
                mc3 = mc2;
                cout << "M2 > M1 \n";
                                }
                                        else if(d2 < d1)
                                                {
pr = ptrgt = 1;
 mc3 = mc1;
 cout << "M1 > M2 \n";
}
}
if(pr == 0)
{
  cout << "They are Equal 'But' : ";
  if(mc1.sgn == mc2.sgn) mc3 = 0;
  ptreq = 1;
// A flag of the equal than is set on.
}
                        cout << "\n The Greater is : \n  ";
                                if(mc3.sgn == +1)
                                ch = '+';
                                else ch = '-';
                                rs = mc3;
                                printmult(mc3);
                                return rs;
                                }
```

If one is negative, then it is trivially that the other number is the bigger whatever is the length is of its nodes.

If the lengths are equal, same sign, here the number lists are traversed from left to right, digits are compared, and the first one which is bigger than the other, then this one is the bigger. If there are all the same, then these numbers are equal, totally.

Results:

These are some examples done by my program:

Addition:

{+, 56, 78, 98, 245, 160} + {+, 45, 3, 68} = {+, 101, 81, 166, 245, 160}

In decimal = + 435161593248.

{+, 34, 120, 36, 86} + {-, 100, 200} = {+, 190, 175, 35, 86} ≡ + 3199148886.

{-, 34, 25, 78, 90, 200} + {+, 200, 220, 98} = {-, 90, 60, 235, 89, 200} ≡ - 387569113544. {-, 120, 38, 98} + {-, 36, 200} = {+, 156, 238, 98} ≡ + 10284642.

Subtraction:

{+, 200, 39, 87, 65, 55} – {+, 48, 98, 120} = {+, 152, 197, 222, 64, 55} ≡ + 656154705975. {-, 46, 24, 33, 78, 200} – {+, 47, 75, 98} = {-, 93, 99, 131, 78, 200} ≡ - 401101508296.

{+, 40, 39, 28, 36, 20} – {-, 200, 180} = {+, 240, 219, 28, 36, 20} ≡ + 1034468205588.

{-, 230, 48, 170, 78} – {-, 49, 160, 120, 35, 84} = {-, 75, 111, 206, 212, 83} ≡ - 323998372947.

Multiplication:

{+, 160, 147, 222} * {+, 250, 59} = {+, 64, 10, 98, 37, 52} ≡ + 275052111156.

{+, 20, 49, 120, 223} * {-, 98, 78, 57} = {-, 168, 225, 106, 24, 47, 6, 50}

{-, 206, 187, 135} * {+, 24, 69, 47} = {-,

80, 33, 102, 203, 146, 51} ≡ -

88104388760115.

{-, 201, 167, 96, 58} * {-, 222, 189, 49} =

{+, 78, 229, 41, 25, 204, 87, 11}

Division & Modulus:

{+, 222, 190, 211} / {+, 208, 199} = {+, 16, 1} ≡ + 4097.

{+, 222, 190, 211} % {+, 208, 199} = {+, 221, 113, 0} ≡ + 14512384.

Exponential:

{+, 87, 75, 129, 89, 221} ^ {+, 4} = {+, 33, 96, 31, 76, 246, 132, 214, 67, 181, 98, 89, 82, 83,

145, 234, 146, 69, 220, 24, 143} ≡ + 19054029194483739602405178337373074533045056859 1.

Comparison:

{+, 39, 67, 128, 226} < {-, 87, 75, 129, 89, 221} = Greater {+, 39, 67, 128, 226}

< False.

Related Work:

BH Flowers's work:

Double linked list were used to store the digits, used to define the integers. The power of C++ is used to define functions to construct list. A decimalizing constructor is made that convert any stored list in decimal number. This number is returned as strings as the limit of integers were exceeded. There are also forward and backward constructors, traversing of these lists.

Future Work:

(i). Arithmetic operators on multiple byte integer.

(ii). Multiple byte float point numbers and they arithmetic operations.

Conclusion:

The sign character is not showing in the final result of the decimal string. The Boolean type BH Flowers made, did not work for me. If the first digit of the string is zero, the decimal will be zero. The error message function contains errors on the standard.h header file.

Chew Keong Tan's Work:

The implementation of asymmetrical cryptographic schemes often requires the use of numbers that are many times larger than the integer data types that are supported natively by the compiler. In this article, an introduction was given to the implementation of arithmetic operations involving large integers. No attempt was tried to give a full coverage of this topic since it is both complex and lengthy. For a more detailed treatment, the reader is referred to the listed references of his.

The source code that accompanies his article implements the `BigInteger` class supporting large integer arithmetic operations. Overloaded operators includes +, -, *, /, %, >>, <<, ==, !=, >, <, >=, <=, &, |, ^, ++, -- and ~. Other additional features such as modular exponential, modular inverse, pseudoprime generation and probabilistic primality testing are also supported.

Features:

a. Arithmetic operations involving large signed integers in 2's complement representation.
b. Prime number tests using Fermat's Little Theorem, Rabin Miller's method and Solovay Strassen's method.
c. Modular exponential with Barrett reduction.
d. Modular inverse.
e. Random Pseudoprime generation.
f. Random Coprime generation.
g. Greatest common divisor.

Future Work:

(i). Faster implementation of arithmetic operations.

(ii). More robust primality testing methods.

(iii). Faster pseudo prime generation

Conclusion : In his article, he has provided a short introduction to the topic of large integer arithmetic. Then he has looked at how large integer addition, subtraction and multiplication can be implemented. Also he examined the problem of primality testing and introduced the concept of primality testing based on Fermat's Little Theorem. His implementation of BigInteger class can be downloaded from his page and provides the overloading of most arithmetic operators. He has pointed out the limitations of his implementations of primality testing and is working towards more robust primality testing methods and faster implementation of arithmetic operators.

References

[**1**]. B.H. Flowers, **An Introduction to Numerical Methods in C++,**

Oxford University Press, 1995.

[**2**]. **Bits and Bytes**, Sun Microsystems, at http://www.java.sun.com, accessed on 27 March 2007.

[**3**]. Brian Brown, **Data Structures and Number Systems,** 1984-2000, at

http://goforit.unk.edu, accessed on 3 December 2007.

[**4**]. Chew Keong Tan, The Code Project – **C# BigInteger Class** – C# Programming, 2002, at

http://www.codeproject.com, accessed on 6 August 2005.

[**5**]. Kenneth H. Rosen, **Elementary Number Theory and its Applications,** AT&T Bell laboratories and Kenneth H. Rosen, 1993.

[**6**]. Ellis Horowitz and Sartaj Sahri, **Fundamentals of Data Structures**, Computer Science Press, Inc., 1983.

[**7**]. Greg Goebel, **Computer Numbering Formats,** at

http://www.vertorsite.net, accessed on 12 June 2005.

[**8**]. Neil Dale, **C++ Plus Data Structures**, By Jones and Bartlett Publishers, inc., 1999.

[**9**]. Paul N. Hilfinger, **Numbers,** University of California, Department of Electrical Engineering and computer sciences, computer sciences division, C561B, Fall 1999.

[**10**]. Wikipedia, the free encyclopedia, **Computer numbering formats** – at

http://en.wikipedia.org, accessed on 3 May 2007.

[**11**]. A research on **The Development of Arithmetic Operations on Multiple Byte Integers**, done by *Ishraga Mustafa Awad Allam* at University of Khartoum, Faculty of Mathematical Sciences, edited 2007, defied 2009 and published 2010. It is also available as a book **Arithmetic Operations on Multiple Byte Integers**, done by *Ishraga* Allam, published by Lambert Academic Publishing. Telefax: +371 686 20455, email: info@omniscriptum.com, www.lap-publishing.com. Isbn 978-620-0-47586-2.