

Computations with large numbers

Weihu Hong, Dept. of Math, Clayton State University, 2000 Clayton State Blvd,
Morrow, GA 30260,

Mingshen Wu, Dept. of Mathematics, Statistics, and Computer Science, University of
Wisconsin-Stout, Menomonie, WI 54751

Abstract. *Since any personal computer has a limited range of integer values, therefore, it will result in an integer overflow when a program tries to compute a value larger than machine's maximum value. We will discuss a workable algorithm that will be able to deal with any large numbers without getting an integer overflow.*

Keywords: Integer overflow.

§1. Introduction

Any programmer knows that when a variable is declared to be of integer type, it has a maximum value. It will result in an integer overflow when a program is to compute an integer value larger than your machine's maximum value. Some machine gives you an error message when overflow occurs, but others don't. Therefore, it is necessary to prevent it from occurring. It would be nice to find an algorithm that will be able to deal with any large number without getting an integer overflow on any machine. In the following, we will provide an algorithm that will fulfill such goal.

§2. Algorithm

Our approach is simple and easy for implementation. The idea is to take any integer input as a string, which can be as long as you wish. For instance, if you have to calculate
12345678912345678998162727272822221234 + 1234598726262626262626262626262681818819982
you can view both numbers as string, then manipulate them either digit by digit or block by block. The following algorithm Add (A, B) is an algorithm of digit by digit.

Algorithm Add (A, B):

$$A = A_n A_{n-1} A_{n-2} \cdots A_1 A_0,$$

where A_0 is the unit digit, A_1 is the tenth digit, and so on

$$B = B_n B_{n-1} B_{n-2} \cdots B_1 B_0,$$

where B_0 is the unit digit, B_1 is the tenth digit, and so on

(if not of the same length, fill it out by zeros to the left)

Initialize Carry = 0;

For ($i = 0; i < n; i++$)

if $A_i + B_i + \text{Carry} > 9$,

then $\{C_i = A_i + B_i + \text{Carry} - 10, \text{Carry} = 1\}$

else $\{C_i = A_i + B_i + \text{Carry}, \text{Carry} = 0\}$

if Carry > 0, then $C_{n+1} = \text{Carry}$ else $C_{n+1} = ""$;

Output: $C = C_{n+1} C_n C_{n-1} \cdots C_1 C_0$

Considering the efficiency for large numbers, we might divide each number into blocks of length k that can be determined by the machine's maximum integer, and then use regular operation of addition to carry out the sum of each block, and then concatenate the sums of blocks to get the sum. The following algorithm demonstrates the idea of block by block.

Algorithm Add2 (A, B)

Assume that k is a positive whole number;

$$A = A_p \times 10^{pk} + A_{p-1} \times 10^{(p-1)k} + A_{p-2} \times 10^{(p-2)k} + \cdots + A_1 \times 10^k + A_0;$$

$$B = B_q \times 10^{qk} + B_{q-1} \times 10^{(q-1)k} + B_{q-2} \times 10^{(q-2)k} + \cdots + B_1 \times 10^k + B_0;$$

where $A_j (0 \leq j \leq p)$ and $B_j (0 \leq j \leq q)$ are whole numbers of k or fewer digits;

Each sum $A_j + B_j + 1 (j = 0, 1, 2, \dots, \min(p, q))$ is less than the machine's max;

(1) Use regular addition to find each sum $S_j = A_j + B_j (j = 0, 1, \dots, \min(p, q))$;

If $S_0.length = k + 1$, then Carry = 1, otherwise, Carry = 0;

Set $S_j = A_j$ (if $p > q$) or B_j (if $p \leq q$) for $j = \min(p, q) + 1$ to $\max(p, q)$;

(2) If (Carry = 1) then $(S_0 = S_0 - 10^k$; $\text{pach}(k - S_0.length)$ 0s to the left of S_0 ;

$C = "" \& S_0$);

(3) For $(j = 1, 2, \dots, \max(p, q))$

{ $S_j = S_j + \text{Carry}$;

if $(S_j.length < k + 1)$ then

if $(j < \max(p, q))$ then $\text{pach}(k - S_j.length)$ 0s to the left of S_j ;

$C = S_j \& C$; Carry = 0;

else

{Carry = 1; $S_j = S_j - 10^k$;

$\text{pach}(k - S_j.length)$ 0s to the left of S_j ;

$C = S_j \& C$;

}

(4) If (Carry = 1) then $C = "1" \& C$;

(5) Output C as the sum $A + B$.

Likewise, we can do subtraction in a similar way. The following is an algorithm for subtraction in digit by digit.

Algorithm Subtract (A, B):
 $A = A_n A_{n-1} A_{n-2} \cdots A_1 A_0$,
 where A_0 is the unit digit, A_1 is the tenth digit, and so on
 $B = B_n B_{n-1} B_{n-2} \cdots B_1 B_0$,
 where B_0 is the unit digit, B_1 is the tenth digit, and so on
 (if not of the same length, fill it out by zeros to the left)
 If $A \geq B$ then sign = "+"
 else sign = "-";
 Initialize Borrow = 0;
 For ($i = 0; i < n; i++$)
 if $A_i - \text{Borrow} < B_i$,
 then $\{C_i = A_i - \text{Borrow} + 10 - B_i, \text{Borrow} = 1\}$
 else
 $\{C_i = A_i - \text{Borrow} - B_i, \text{Borrow} = 0\}$
 Output : if (sign == "-") then $-C_n C_{n-1} \cdots C_0$
 else $C_n C_{n-1} \cdots C_0$

We can also do subtraction block by block as shown in Algorithm Subtract2 (A, B).

Algorithm Subtract2 (A, B):
 Assume that k is a positive whole number;
 $A = A_p \times 10^{pk} + A_{p-1} \times 10^{(p-1)k} + A_{p-2} \times 10^{(p-2)k} + \cdots + A_1 \times 10^k + A_0$;
 $B = B_q \times 10^{qk} + B_{q-1} \times 10^{(q-1)k} + B_{q-2} \times 10^{(q-2)k} + \cdots + B_1 \times 10^k + B_0$;
 where $A_j (j = 0, 1, 2, 3, \dots, p-1)$ and $B_j (j = 0, 1, 2, 3, \dots, q-1)$ are
 whole numbers of k digits (pach 0s if necessary);
 A_p and B_q are whole numbers of k or fewer digits;
 A_i and B_j are less than the machine's max;
 (1) if ($A = B$) then
 output $D = 0$; Exit;
 else if ($A > B$) then sign = '+';
 else {sign = '-'; swap A and B;}

- (2) Assume $A > B$;
 Initialize $Borrow = 0$;
 Use regular subtraction to find each difference :
 For ($j = 0$ to $\max(p, q)$)
 if ($j \leq \min(p, q)$) then
 if ($A_j - Borrow > B_j$) then ($D_j = A_j - B_j - Borrow$; $Borrow = 0$;))
 else ($D_j = A_j + 10^k - B_j - Borrow$; $Borrow = 1$;))
 else
 if ($A_j - Borrow > 0$) then ($D_j = A_j - Borrow$; $Borrow = 0$;))
 else ($D_j = A_j + 10^k - Borrow$; $Borrow = 1$;))
- (3) Let $m = \max(p, q)$;
 (4) For ($j = 0$ to $m - 1$), patch ($k - D_j.length$) 0s to the left of D_j ;
 (5) $D = ""$;
 for ($j = 0$ to m) $D = D_j \& D$;
 (6) if ($sign = '-'$) then $D = sign \& D$;
 (7) Output D as the difference $A - B$.

For multiplication, we can do either digit wise or block wise. The following is an algorithm for multiplication in block wise.

Algorithm Multiply (A, B) :

Assume that

$$A = A_p \times 10^{pk} + A_{p-1} \times 10^{(p-1)k} + A_{p-2} \times 10^{(p-2)k} + \dots + A_1 \times 10^k + A_0;$$

$$B = B_q \times 10^{qk} + B_{q-1} \times 10^{(q-1)k} + B_{q-2} \times 10^{(q-2)k} + \dots + B_1 \times 10^k + B_0;$$

where A_i, B_j ($0 \leq i \leq p; 0 \leq j \leq q$) are whole numbers of k or fewer digits,

Each product $A_i B_j$ is less than the machine's max;

- (1) Use regular multiplication to find each product $A_i B_j$;
- (2) Attching $(i + j)k$ zeros to the end of the product $A_i B_j$;
- (3) Use the $Add(A, B)$ or $Add2(A, B)$ to sum up all products;
- (4) Output the sum as the product of A and B .

In order to check divisibility and primality of integers, we need an algorithm of division. We can proceed as follow:

Algorithm Divide (A, B) :

$$A = A_n A_{n-1} A_{n-2} \dots A_1 A_0,$$

where A_0 is the unit digit, A_1 is the tenth digit, and so on

$$B = B_m B_{m-1} B_{m-2} \dots B_1 B_0,$$

where B_0 is the unit digit, B_1 is the tenth digit, and so on

- (1). If $A < B$, then Output quotient = 0, and remainder = A.
- (2). If $A = B$, then Output quotient = 1, and remainder = 0.
- (3). If $A > B$, then

Dividend = A, divisor = B,
K = the length of B;

While (Dividend > divisor)

```

{
  D1 = the first k digits of Dividend from the left.
  D2 = the substring of Dividend starting from the (k+1) to the end
  If (D1 >= divisor) then
    {Find the greatest non-negative integer  $k$  such that  $k \times (\text{divisor}) \leq D1$ ;
      $q_{count} = k$ ;
    Update Dividend = Subtract (D1, Multiply ( $k$ , divisor)) & D2;

  }
  Else
    {  $q_{count} = 0$ ;
      D1 = the first  $k + 1$  digits of Dividend
      D2 = the substring of Dividend starting from the (k+2) to the end

      Find the greatest non-negative integer  $k$ 
          such that  $k \times (\text{divisor}) \leq D1$ ;

       $q_{count} = k$ ;
      Update Dividend = Subtract (D1, Multiply ( $k$ , divisor)) & D2;
    }
}
If (Dividend = divisor) then
{
   $q_{count} = 1$ ;    Remainder = 0;
}
Else
{
  Remainder = Dividend;
}

```

Output : quotient = $q_0q_1q_2 \cdots q_{count-1}$
remainder = Remainder

Once we implement Divide (A, B), we can define modulo function Mod (A, B), which returns the remainder when A is divided by B. Therefore, we can check divisibility and primality of integers.

§3. Applications

It is very common that whenever we teach Number Theory, we need display some large numbers such as Fibonacci primes and Mersenne numbers by using computer. If we are not careful, we will run into an integer overflow. We can use the algorithms developed in section 2 above to eliminate such errors. As a demonstration, we give two examples in the following. The first example is a comparison of displays of large numbers on computers with or without using our algorithms.

Example 1. It is well known^[2] that the 3rd, 4th, 7th, 11th, 13th, 17th, 23rd, 29th, 43rd, 47th, 83rd, 131st, 137th, 359th, 431st, 433rd, 449th, 509th, 569th, 571st, 2971st, 4723rd, 5387th, 9311th, 9677th, 14431st, 25561st, 30757th, 30757th, 35999th, and 81839th terms of Fibonacci sequence are primes. It is easy to display any of the first few small ones using a computer. However, it is impossible to display a large one without getting an integer overflow. In the table 1, it was shown that the display via JavaScript without using our algorithm will either loss precisions or get a scientific notation that unfortunately results in a real number instead of an integer. A live demonstration is available on our website at <http://cims.clayton.edu/whong/tools/Demo4BigNumber.htm>.

Table 1 Fibonacci Primes.

The first 22 primes in the Fibonacci Sequence	
The Display without using our algorithm	The Display using our algorithm
2	2
3	3
5	5
13	13
89	89
233	233
1597	1597
28657	28657
514229	514229
433494437	433494437
2971215073	2971215073
99194853094755490	99194853094755497
1.0663404174917107e+27	1066340417491710595814572169
1.9134702400093282e+28	19134702400093278081449423917
4.754204377346978e+74	475420437734698220747368027166749382927701417016557193662268716376935476241
5.298927110060951e+89	529892711006095621792039556787784670197112759029534506620905162834769955134424689676262369
1.3872771278047825e+90	1387277127804783827114186103186246392258450358171783690079918032136025225954602593712568353
3.0617199924845414e+93	3061719992484545030554313848083717208111285432353738497131674799321571238149015933442805665949
1.0597999265301477e+106	10597999265301490732599643671505003412515860435409421932560009680142974347195483140293254396195769876129909
3.668447431608094e+118	36684474316080978061473613646275630451100586901195229815270242868417768061193560857904335017879540515228143777781065869
9.604120061892246e+118	96041200618922553823942883360924865026104917411877067816822264789029014378308478864192589084185254331637646183008074629
Infinity	357103560641909860720907774139063454445569926582843306794041997476301071102767570483343563518510007800304195444080518562630900027386498933944619210192856768352683468831754423234217978525765921040747291316681576556861490773135214861782877716560879686368266117365351884926393775431925116896322341130075880287169244980698837941931247516010101631704349963583400361910809925847721300802741705519412306522941202429437928826033885416656967971559902743150263252229456298992263008126719589203430407385228230361628494860172129702271172926469500802342608722006420745586297267929052509059154340968348509580552307148642001438470316229

Example 2. It is well known^[1] that the 257th Mersenne number is a prime. To display the 257th Mersenne number, we can make a simple loop as follows:

```
Count = 0;
X = 1;
While (Count Not Equal to 257)
{ X = Multiply(X,"2");
  Z = Subtract(X,"1");
  Count++;
} Output Z;
```

The 257th Mersenne number is
231584178474632390847141970017375815706539969331281128078915168015826259279871.

§4. Conclusion

It is very common that an error of integer overflow occurs in scientific computation. To eliminate such an error, it is a good idea to implement algorithms as we did in this note in either digit by digit or block by block so that programs will be able to handle situations intelligently for larger integers. The algorithm given in this note can be easily applied to many scientific computations such as RSA^[3] encryption and searching for large primes.

Acknowledgments: We thank Dr. Aust and Dr. Ming-Jun Lai for their many suggestions that improved this note.

References

- [1] David M. Burton, 2007, Elementary Number Theory, the 6th Ed., McGraw Hill.
- [2] Chris K. Caldwell, Fibonacci Primes, <http://primes.utm.edu>
- [3]. Bruce Schneier, 1996, Applied Cryptography, Protocols, Algorithms, and Source Code in C, the 2nd Ed., Wiley.