Implementing Basic Arithmetic for Large Integers: Division

Nitin Verma
mathsanew.com

June 18, 2021

This article is a continuation of the earlier one titled *Implementing Basic Arithmetic for Large Integers: Introduction* [1], which introduced how arbitrarily large integers along with basic arithmetic operations can be implemented via a program. It discussed the *bigint* structure for such large integers, the operations of addition, subtraction and printing, and also included the source code of all common internal methods. Multiplication operation was discussed in a subsequent article [2].

Now we look into the division operation, using *Schoolbook Division Algorithm*. We will be frequently using the concepts and definitions from [1].

Example with Decimal Numerals

We first look at an example of division of decimal numerals using the schoolbook algorithm, also known as *Long Division Algorithm*.

$$
\begin{array}{cccccccccccc}
 & & & & & & & 0 & 4 & 0 & 3 & 6 \\
\cline{5-12}
3 & 6 & 5 &) & 0 & 1 & 4 & 7 & 3 & 4 & 9 & 1 \\
 & & & & & & & 0 & & & & \\
\cline{5-8}
 & & & & & 1 & 4 & 7 & 3 & & & \\
 & & & & & 1 & 4 & 6 & 0 & & & \\
\cline{6-9}
 & & & & & & 0 & 1 & 3 & 4 & & \\
 & & & & & & & & & 0 & & \\
\cline{7-10}
 & & & & & & & 1 & 3 & 4 & 9 & \\
 & & & & & & & 1 & 0 & 9 & 5 & \\
\cline{8-11}
 & & & & & & & & 2 & 5 & 4 & 1 \\
 & & & & & & & & 2 & 1 & 9 & 0 \\
\cline{9-12}
 & & & & & & & & & 3 & 5 & 1
\end{array}
$$

It has been presented slightly differently. First, the dividend has been prefixed with a 0. Second, the steps which generate a digit 0 in quotient have been shown explicitly like any other step; this is so even for the initial 0 in the quotient.

Each digit of the generated quotient will be referred as a "quotient-digit". After each subtraction, we bring down the next dividend-digit to append to the subtraction result. The value thus formed is divided by the divisor to obtain the next quotient-digit; so it is like "dividend" of a particular iteration. We will be referring such values as "Iteration's Dividend", abbreviated as "IDD". In this example 1473, 0134, 1349, 2541 are all IDD of their respective iteration.

To allow us to view this algorithm as an iterative process which repeats the same kind of generic step, we can also call the initial prefix digits taken from the input dividend as an IDD. So, in this example, 0147 is also an IDD.

Note that, in this example of a 3-digit divisor, all the IDDs are up to $3 + 1 = 4$ digits. Those IDDs which have less than 4 digits have been prefixed with 0s to make their width 4, e.g. 0134. In general, for a divisor b of n_b digits, we will always be treating the IDDs as having width $n_b + 1$, by prefixing some 0s if necessary. This convention will help us in our later discussion.

Note that the initial IDD always starts with a 0, followed by n_b digits taken from the dividend.

Long-Division as an Iterative Algorithm

Let us denote the dividend by $a = (a_{n_a-1}a_{n_a-2}\dots a_1a_0)$ having n_a digits, and the divisor by $b = (b_{n_b-1}b_{n_b-2}\dots b_1b_0)$ having n_b digits, in base B numeral system. The case when $n_a < n_b$ simply implies the quotient is 0 and remainder is a. So, we only need to consider the other case, $n_a \geq n_b$.

The long-division algorithm can be seen as an iterative process where each iteration generates exactly one quotient-digit. In each iteration, the "current" IDD x, having width $n_b + 1$ is considered and a digit d needs to be found such that:

$$\begin{aligned} bd &\leq x \\ b(d+1) &> x \end{aligned} \tag{1}$$

Once this quotient-digit d is found, subtraction $(x - bd)$ is done and the next digit from a is suffixed to the result, to form the next IDD. This continues until there are no more digits to bring down from a. At that point, the subtraction result, $(x - bd)$, is the remainder of this division.

Note that the result of subtraction $t = (x - bd)$ in any iteration follows:

$$t = x - bd < b(d + 1) - bd = b$$

i.e. $t \leq b - 1$. So, t has at most n_b digits.

The next IDD, which is formed by suffixing a digit (say, a_i) from a to t is $(tB + a_i)$, and must have at most $n_b + 1$ digits. Also, it must follow:

$$tB + a_i \leq (b - 1)B + (B - 1) = bB - 1 < bB$$

The initial IDD is also chosen as having n_b digits from the dividend, and must be less than bB, which has $n_b + 1$ digits. So, we can conclude that every IDD must have at most $n_b + 1$ digits, and must be less than bB. Though, as mentioned earlier, we will always be treating them as having width $n_b + 1$.

Further, any number d which satisfies (1) must follow $d \leq x/b < bB/b = B$. That is, $d \leq B - 1$, and so d must be a single digit number.

Now we look into the problem of computing quotient-digit d in each iteration.

Computing a Quotient-Digit

When we perform long-division manually for decimal numerals, we usually depend on trial-and-error to find a quotient-digit d. If the IDD is less than the divisor, d is simply 0. Otherwise, we can make a guess about d based on some initial digits of the divisor and the IDD. Then we need to multiply guess d with the divisor to check if the guess is correct. If incorrect, we make another guess and so on. In our earlier example, for IDD 1349, the guess of $d = 4$, based simply on 13/3 (13 from 1349, 3 from 365) is incorrect ($365 \cdot 4 = 1460$ exceeds 1349) and needs another try with $d = 3$.

We need to make this trial-and-error method efficient (requiring less number of corrections) and well-defined, for any base B numeral system. We will be discussing a method proposed by Pope and Stein in [3], which is also described in Knuth's book [4] (section *Multiple Precision Arithmetic*). To guess d, it uses the first digit of b and first two digits of the IDD (which, as mentioned earlier, is treated as having width $n_b + 1$).

A variant of this method which uses first two digits of b and first three digits of the IDD has been discussed by Hansen in [5], which is also a good resource on this topic. The theorems behind this variant have been attributed to Krishnamurthy and Nandi [6].

Let us denote the IDD of some iteration by $x = (x_{n_b}x_{n_b-1}\ldots x_1x_0)$, where x_{n_b}, x_{n_b-1} etc may be 0. In favor of simpler symbols, we will denote x_{n_b}, x_{n_b-1} as y, z and b_{n_b-1} (the MSD of b) as e. The 2-digit number formed by y and z will be denoted as (yz).

Before we proceed further, we will make some observations about multiplication of b by a single digit s, that will help us in the discussion below. Such multiplication will be referred as "multiplication $bs\{s = \ldots\}$" with s as specified.

We can use the schoolbook's multiplication method to help us understand this:

$$\begin{array}{cccccccc} & e & b_{n_b-2} & b_{n_b-3} & \ldots & b_1 & b_0 \\ & & & & & \times & s \\ \hline u & v & t_{n_b-2} & t_{n_b-3} & \ldots & t_1 & t_0 \end{array}$$

Please refer to article [2], section "Multiplication by a Digit", Lemma 2. The digits u, v at positions n_b, n_{b-1} of the result, will together form a number (uv) as given by (c_{n_b-2} is the carry from earlier position):

$$(uv) = c_{n_b-2} + es \tag{2}$$

So,

$$(uv) \geq es \tag{3}$$

Also, due to Lemma 1 in article [2], any carry cannot exceed $(s-1)$. So,

$$(uv) \leq (s-1) + es \tag{4}$$

Whenever we will need to compare bs (for any digit s) with the IDD x, we can first ignore their last $n_b - 1$ digits and simply compare the values formed by their two most significant digits: (uv) and (yz). Then we can use:

$$(uv) > (yz) \;\Rightarrow\; bs > x \tag{5}$$

$$(uv) < (yz) \;\Rightarrow\; bs < x \tag{6}$$

First Estimate

Let us say, we first estimate the desired quotient-digit d to be the following based on the initial digits of the divisor b and IDD x:

$$d_1 = min(\lfloor (yz)/e \rfloor, B-1) \tag{7}$$

Since $d_1 \leq \lfloor (yz)/e \rfloor$,

$$ed_1 \leq (yz) \tag{8}$$

Can this estimate d_1 be less than d? Let's assume the case when $d_1 < d$. Since $d \leq B-1$, we must have $d_1 < B-1$, i.e. $d_1 = \lfloor (yz)/e \rfloor$. But that means,

$$e(d_1 + 1) > (yz)$$

In the multiplication $bs\{s = d_1 + 1\}$ from the last section, the value (uv) in the result will follow, due to (3) and above:

$$(uv) \geq e(d_1 + 1) > (yz)$$

Thus, $b(d_1 + 1)$ would exceed the IDD x (using (5)), and so $d_1 + 1$ or any higher digit cannot be the desired quotient-digit d. This contradicts our assumption that $d_1 < d$. Hence,

$$d \leq d_1 \tag{9}$$

Intuitive Idea

Now we need to figure out, by how much d_1 may exceed d. Note that in the multiplication $bs\{s = d_1\}$, the value (uv) in the result follows (due to (2)):

$$(uv) = c_{n_b-2} + ed_1$$

where carry c_{n_b-2} cannot exceed $(d_1 - 1)$.

Due to (8), the ed_1 term is less than or equal to (yz), and the contribution by the carry term may further lead to (uv) exceeding (yz). If that happens, we definitely know that $bd_1 > x$ (using (5)) and d_1 is not the correct quotient-digit.

Intuitively, this contribution by carry term (up to $(d_1 - 1)$), which is roughly bounded by B, can be counterbalanced by reducing d_1 to $d_1 - \Delta$ in the ed_1 term, for some Δ. Then, ed_1 reduces by $e\Delta$, and so we can be sure to counterbalance the contribution of B by this $e\Delta$ if $e\Delta \geq B$.

We can see, the larger is e, the lesser Δ is required to ensure this counterbalancing. Now we will look at this idea formally.

Correction in the Estimate

Due to (9), we know that the desired d is of the form $d_1 - \Delta$, where $\Delta = 0, 1, 2, \ldots$. What happens when we do multiplication $bs\{s = d_1 - \Delta\}$ for any $\Delta = 0, 1, 2, \ldots$? The value (uv) in the result must follow (due to (4)):

$$(uv) \leq (d_1 - \Delta - 1) + e(d_1 - \Delta) \tag{10}$$

Suppose, for any given Δ, we can find out the condition C_Δ under which this (uv) is always less than (yz). Then, $b(d_1 - \Delta)$ must be less than x (using (6)). So, we can be sure that the desired d must be one among $d_1 - \Delta$, $d_1 - \Delta + 1$, ..., d_1, whenever condition C_Δ holds.

Let us try to find out such a condition. Equation (10) provides an upper-bound on (uv) and equation (8) provides a lower-bound on (yz). If we ensure that the (uv) upper-bound is less than (yz) lower-bound, we are guaranteed to get $(uv) < (yz)$. To do that:

$$\begin{aligned} & (d_1 - \Delta - 1) + e(d_1 - \Delta) < ed_1 \\ \Leftrightarrow \quad & (d_1 - \Delta - 1) < e\Delta \\ \Leftrightarrow \quad & e > (d_1 - \Delta - 1)/\Delta \end{aligned}$$

(Note that above is not possible with $\Delta = 0$). As $(B-1) > (d_1 - \Delta - 1)$, so, if we have $e \geq (B-1)/\Delta$, the above condition is guaranteed to be met. Thus, the desired condition C_Δ, which guarantees that the d is one among $d_1 - \Delta$, $d_1 - \Delta + 1$, ..., d_1 is:

$$C_\Delta \colon e \geq (B-1)/\Delta \tag{11}$$

For $\Delta = 1$, it looks difficult to achieve this condition. For $\Delta = 2$, it is achieved whenever $e \geq (B-1)/2$. Since the divisor b need not have its first digit e with $e \geq (B-1)/2$, so we will follow a process called *Normalization* to achieve this, whenever $e < (B-1)/2$. Note that $(B-1)/2$ itself is not an integer for even B.

Normalization

We will multiply both a and b by a *Scaling-Factor* f so that (bf)'s first digit is at least $(B-1)/2$. Then, we will perform the division process on these modified operands af (dividend) and bf (divisor). The quotient of this division must be same as the quotient with original operands a and b. The remainder needs to be divided by f to give the remainder for division with original operands.

We can find the scaling-factor f as follows. Let us try with f as a single digit number. In the multiplication $bs\{s=f\}$, the value (uv) in the result will follow, due to (3) and (4):

$$ef \leq (uv) \leq (f-1) + ef$$

We want this value (uv) to satisfy the constraints:

$$(B-1)/2 \leq (uv) \leq B-1$$

which can be achieved with:

$$\begin{array}{lrlr} & (B-1)/2 \leq ef & \text{and,} & (f-1)+ef \leq B-1 \\ \Leftrightarrow & f \geq (B-1)/(2e) & \text{and,} & f \leq B/(e+1) \end{array}$$

So, we can choose $f = \lfloor B/(e+1) \rfloor$.

Implementation

To summarize, if the divisor and dividend have been normalized, then the process of computing d is the following. Find the first estimate d_1 using (7). Since we use $\Delta = 2$, the desired d must be one among $d_1 - 2$, $d_1 - 1$, d_1. Specifically, d is the maximum among these three values with the property that $(d_1 - i)b \leq x$, $i = 0, 1, 2$.

Note that when b is a single digit divisor, IDD x is simply the two-digits (yz), and so the first estimate $d_1 = \lfloor (yz)/e \rfloor$ must be the correct digit d always. So, no normalization is needed in this case.

Below we implement the long-division for bigints, while using the above approach for computing a quotient-digit. Please refer to [1] and [2] for implementation of methods referred by these methods.

```
/* a: dividend, b: divisor, qt: quotient, rm: remainder */
int divide(bigint *a, bigint *b, bigint *qt, bigint *rm)
{
  int    na, nb, m;
  ulong  yz, d;
  bigint x, t1, t2, af, bf;
  uint   e, f = 1;

  if(BINT_ISZERO(b))
  {
    printf("divide: divisor is 0\n");
    return -1;
  }

  if(BINT_LEN(a) < BINT_LEN(b))
  {
    BINT_INIT(qt, 0);
    copy(rm, a);
    return 0;
  }

  e = b->digits[b->msd];

  if((BINT_LEN(b) > 1) && (e < B/2))
  {
    /* normalization */
    f = B/(e + 1);
    multiply_digit(a, f, &af);
    multiply_digit(b, f, &bf);
    a = &af;
    b = &bf;
    e = b->digits[b->msd];
  }

  na = BINT_LEN(a);
  nb = BINT_LEN(b);
  /* na >= nb holds. */

  /* quotient can have maximum (na-nb+1) digits  */
  qt->msd = WIDTH - (na-nb+1);

  /* loop-invariant P: first m digits of 'a' have been brought-down
     and processed. */

  m = 0;
```

```
  while(m < na)
  {
    if(m == 0)
    {
      m = nb;
      prefix(&x, a, m);
    }
    else
    {
      m++;
      shift_left(&x, 1);
      x.digits[WIDTH-1] = a->digits[a->msd + m-1];
    }

    yz = yz_from_x(&x, nb+1);
    d = yz/e;

    if(nb > 1)
      correct_d_and_subtract(&x, b, &d);
    else
    {
      /* we don't need correction in d if nb=1 */
      yz = yz - e*d;
      /* above remainder is less than e, so must be single digit */
      BINT_INIT(&x, yz);
    }

    qt->digits[qt->msd + m-nb] = d;
  }

  /* (loop-invariant P) AND (m=na) holds. */
  /* Now x contains the remainder. */

  rm_leading_0s(qt);

  if(f > 1)
  {
    BINT_INIT(&t1, f);
    divide(&x, &t1, rm, &t2);
  }
  else
    copy(rm, &x);

  return 0;
}
```

```
/* treat x as having width w, by prefixing some 0s if necessary */
static ulong yz_from_x(bigint *x, int w)
{
  int  iy = WIDTH-w;  /* index of y */
  uint y, z;

  if(iy >= x->msd)
    y = x->digits[iy];
  else
    y = 0;

  if(iy+1 >= x->msd)
    z = x->digits[iy+1];
  else
    z = 0;

  return ((ulong)y)*B + z;
}

static void correct_d_and_subtract(bigint *x, bigint *b, ulong *d)
{
  bigint t;

  if(*d > B-1)
    *d = B-1;

  multiply_digit(b, *d, &t);
  /* Now (t = bd) holds. */

  if(compare(&t, x) == 1)
  {
    *d = *d - 1;
    subtract(&t, b, &t);
    /* Now (t = bd) holds again. */

    if(compare(&t, x) == 1)
    {
      *d = *d - 1;
      subtract(&t, b, &t);
      /* Now (t = bd) holds again. */
      /* We must now have t <= x. */
    }
  }

  /* ((t = bd) AND (t <= x)) holds (implies bd <= x). */
  subtract(x, &t, x);
}
```

■

References

[1] Nitin Verma. *Implementing Basic Arithmetic for Large Integers: Introduction.* https://mathsanew.com/articles/implementing_large_integers_introduction.pdf (2021).

[2] Nitin Verma. *Implementing Basic Arithmetic for Large Integers: Multiplication.* https://mathsanew.com/articles/implementing_large_integers_multiplication.pdf (2021).

[3] D. A. Pope, M. L. Stein. *Multiple Precision Arithmetic.* C. ACM, Vol 3 (12) (1960), 652–654.

[4] D. E. Knuth. *The Art of Computer Programming*, Vol 2, Third Edition. Addison-Wesley (1997).

[5] P. B. Hansen. *Multiple-Length Division Revisited: A Tour of the Minefield.* Syracuse University EECS - Technical Reports, 166 (1992). https://surface.syr.edu/eecs_techreports/166.

[6] E. V. Krishnamurthy, S. K. Nandi. *On the Normalization Requirement of Divisor in Divide-and-Correct Methods.* C. ACM, Vol 10 (12) (1967), 809–813.