Implementing Basic Arithmetic for Large Integers: Multiplication

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.

Now we look into the multiplication operation, and discuss two algorithms for the same: *Schoolbook Multiplication* and *Karatsuba Multiplication*. We will be frequently using the concepts and definitions from [1]. As done earlier, the two operands of arithmetic operations will generally be denoted by $a = (a_{n_a-1}a_{n_a-2}\dots a_1a_0)$ and $b = (b_{n_b-1}b_{n_b-2}\dots b_1b_0)$, in base B numeral system.

Schoolbook Multiplication Algorithm

Here is an example of multiplication of decimal numerals using the schoolbook method, which is also called "Long Multiplication":

			7	4	3	8	(upper operand)
				9	3	6	(lower operand)
		4	4	6	2	8	
	2	2	3	1	4		
6	6	9	4	2			
6	9	6	1	9	6	8	

We iteratively pick each digit b_i of the lower operand b, starting at the LSD (position 0), and multiply it with the complete upper operand a to produce one "row" R_i. Each such row R_i is left-shifted by i positions. The result of $(a \cdot b)$ is addition of these left-shifted rows.

Multiplication by a Digit

Let us first understand the multiplication of a by any single digit s of b. It is done iteratively by picking each digit a_i, starting at the LSD (position 0) and computing $a_i s$. Since,

$$a_0 s \leq (B-1)s = Bs - s + B - B = (s-1)B^1 + (B-s)B^0$$

so the *carry* produced by position 0, c_0, can be maximum $(s-1)$. For the computation at position 1, we have:

$$c_0 + a_1 s \leq (s-1) + (B-1)s = Bs - 1 + B - B = (s-1)B^1 + (B-1)B^0$$

So, the carry produced by position 1, c_1, also can be maximum $(s-1)$. Thus, by repeating such argument we know that the carry c_i produced by any position i can never exceed $(s-1)$.

Lemma 1. *In the schoolbook multiplication algorithm, when multiplying operand a with digit s in any base B, the carry produced by any position cannot exceed $(s-1)$. And since $s \leq (B-1)$, a carry can never exceed $(B-2)$.*

This allows us to conclude that such a carry can be stored and added like any other digit of our base B numeral system, using the *uint* type.

Note that the calculation at any position i is of the form (defining $c_{-1} = 0$): $c_{i-1} + a_i s$, which can produce values of up to 2-digit numeral in base B, because:

$$c_{i-1} + a_i s \leq (s-1) + (B-1)s = Bs - 1 + B - B = (s-1)B^1 + (B-1)B^0$$

[Side Note] At all positions except the MSD position of a (i.e. $i = n_a - 1$), this calculation contributes the lower digit of this 2-digit value in the result $(a \cdot s)$, sending the upper digit to carry. At position $i = n_a - 1$, it will contribute both of its digits in the result. Now we will write a lemma which will be useful later when discussing Division operation.

Lemma 2. *In the schoolbook multiplication algorithm, when multiplying operand $a = (a_{n_a-1} a_{n_a-2} \ldots a_1 a_0)$ with digit s in any base B, the result $(a \cdot s)$ has either n_a or $n_a + 1$ digits. In the result, the two digits at (most-significant) positions n_a, $n_a - 1$ are produced by the last iteration, which multiplies MSD of a with s. This 2-digit value is given by (c_{n_a-2} is the earlier produced carry):*

$$c_{n_a-2} + a_{n_a-1} s$$

[End of Side Note]

Since the calculation at any position can produce values only up to 2-digit, they should fit in the "unsigned long" type.

Implementation

Now we know how to produce each row R_i. Instead of first producing and storing all such rows, and then adding them all (after left-shifting), we will add each row to a common result r as and when it is produced. Thus we will avoid the need to store these rows for later processing.

Below method *multiply_digit*() implements multiplication of a bigint by a single digit, and *multiply*() implements multiplication of two bigints. Please refer to [1] for implementation of methods referred by these methods.

Note that each row R_i has up to $n_a + 1$ digits, and after left-shifting it by i positions, it effectively has $n_a + i + 1$ digits. When we add row R_i to the common result r, we will be performing an addition involving $n_a + i + 1$ digit-positions (the current r must not be having more digits than this); i.e. $n_a + i + 1$ iterations in the addition method. Starting with r as R_0, the addition of remaining $n_b - 1$ rows will require total such iterations as:

$$(n_a + 2) + (n_a + 3) + \ldots + (n_a + n_b)$$

So when $n_a < n_b$, we can reduce these iterations by swapping a and b. That is, for efficiency, the operand with more number of digits will always be chosen as the upper operand.

```
/* input struct r can also be same as struct a */
int multiply_digit(bigint *a, uint s, bigint *r)
{
  int   j;
  uint  carry = 0;
  ulong d;

  if(s == 0)
  {
    BINT_INIT(r, 0);
    return 0;
  }

  /* digits[] index j corresponds to position WIDTH-1-j in the
     positional-numeral representation */

  for(j = WIDTH-1; j >= a->msd; j--)
  {
    d = ((ulong)a->digits[j])*s + carry;
    carry = d / B;
    r->digits[j] = d % B;
  }

  if(carry)
  {
    if(j < 0)
    {
      printf("multiply_digit: overflow\n");
      return -1;
    }
    else
    {
      r->digits[j] = carry;
      r->msd = j;
    }
  }
  else
    r->msd = j + 1;

  return 0;
}
```

```
/* input r must be a separate struct than a or b */
int multiply(bigint *a, bigint *b, bigint *r)
{
  int    j;
  bigint R; /* row */

  if(BINT_LEN(a) < BINT_LEN(b))
    SWAP(a, b);

  /* digits[] index j corresponds to position WIDTH-1-j in the
     positional-numeral representation */

  multiply_digit(a, b->digits[WIDTH-1], r);

  for(j = WIDTH-2; j >= b->msd; j--)
  {
    multiply_digit(a, b->digits[j], &R);
    shift_left(&R, WIDTH-1-j);
    add(r, &R, r);
  }

  return 0;
}
```

Karatsuba Multiplication Algorithm

Now we will discuss one more algorithm for multiplication. We first consider the case when the two operands have same number of digits. Say, the two operands are $a = (a_{n-1}a_{n-2}\ldots a_1a_0)$ and $b = (b_{n-1}b_{n-2}\ldots b_1b_0)$, each having n digits in base B, where n is even. We can break each of these numbers into two components as below, where $k = n/2$:

$$a = (a_{n-1}a_{n-2}\ldots a_k)B^k + (a_{k-1}a_{k-2}\ldots a_0)$$
$$b = (b_{n-1}b_{n-2}\ldots b_k)B^k + (b_{k-1}b_{k-2}\ldots b_0)$$

We can denote these components as integers L_a, R_a, L_b, R_b (each having k-digits) and write:

$$a = L_aB^k + R_a$$
$$b = L_bB^k + R_b$$

The L and R symbols signify "Left" and "Right" half in the digit sequence. Now, the product of a and b can be expressed as:

$$\begin{aligned} ab &= (L_aB^k + R_a)(L_bB^k + R_b) \\ &= (L_aL_b)B^{2k} + (L_aR_b + R_aL_b)B^k + (R_aR_b)B^0 \end{aligned} \tag{1}$$

So, to obtain ab, one way is to first compute each of (L_aL_b), (L_aR_b), (R_aL_b) and (R_aR_b) and use them in above expression. Each of these 4 terms require multiplication of two k-digit numbers. Note that multiplication by B^{2k} and B^k can be simply done by left-shifting by $2k$ and k positions respectively. The problem of multiplying a and b has been reduced to the subproblems of these 4 multiplications, followed by left-shifts and additions. This is in fact a *Divide and Conquer* approach.

Karatsuba Multiplication Algorithm (also described in [2] by its inventor A. A. Karatsuba) is based on the observation that the term $(L_aR_b + R_aL_b)$ can be rewritten to give us something useful:

$$L_aR_b + R_aL_b = (L_a + R_a)(L_b + R_b) - L_aL_b - R_aR_b \tag{2}$$

Since L_aL_b and R_aR_b anyway need to be computed in (1), so just by doing one more multiplication $(L_a + R_a)(L_b + R_b)$, we can avoid doing the two multiplications L_aR_b and R_aL_b.

Since adding two k-digit numbers may produce a $(k + 1)$-digit number, so the multiplication $(L_a + R_a)(L_b + R_b)$ may involve $(k+1)$-digit operands. We will be referring to a multiplication of operands having x and y digits as "a (x, y)-digits multiplication".

Thus we are now required to perform overall two (k, k)-digits (L_aL_b, R_aR_b) and one $(k + 1, k + 1)$-digits multiplications, instead of four (k, k)-digits multiplications appearing in (1). Though, for this saving some extra work is required, like additions $(L_a + R_a)$ and $(L_b + R_b)$, and subtractions of L_aL_b and R_aR_b as seen in (2).

This same technique can be recursively applied while solving the three multiplication subproblems. If number of digits (n) is not even for any subproblem, we can choose $k = \lceil n/2 \rceil$.

Let us denote by $T(n)$ the time-complexity of this algorithm for (n, n)-digits multiplication, where n is even. For this complexity analysis, we will treat the $(k+1, k+1)$-digits multiplication as having complexity $T(k) + c_1k$ (for some constant c_1), which is achievable. The extra work required for additions and subtractions in (2) can be written as c_2n for some constant c_2. So the recurrence relation for $T(n)$ is, for some constant c:

$$T(n) = 2T(k) + (T(k) + c_1k) + c_2n = 3T(n/2) + cn$$

If we assume n to be a power of 2, $T(n)$ will come out to be $O(n^{\log_2 3})$. It is easy to see that the schoolbook multiplication algorithm has time-complexity $O(n^2)$ for (n, n)-digits multiplication.

Operands with Unequal Lengths

In above, we have assumed a and b to be having the same number of digits. Now, let us say, a has n_a digits and b has n_b digits, $n_a \geq n_b$. For their multiplication, how can we benefit from the above method?

Due to the *Division Theorem*, there must exist integers q, r with $q \geq 1$, $0 \leq r \leq n_b - 1$, such that:

$$n_a = qn_b + r$$

Starting from the least-significant-digit (LSD) of a, we can divide it into q chunks of n_b digits each, and one last chunk of r digits which begins at the most-significant-digit (MSD) of a. We can write a in terms of these chunks (components) as:

$$a = (C_q)B^{qn_b} + \ldots + (C_2)B^{2n_b} + (C_1)B^{n_b} + (C_0)B^0$$

where $C_0, C_1, \ldots, C_{q-1}$ have n_b digits each and C_q has r digits. Note that, because of some 0 digits within a, these chunks may have redundant 0s at the beginning positions (starting at their MSD).

The product ab will be computed as:

$$ab = (C_qb)B^{qn_b} + \ldots + (C_2b)B^{2n_b} + (C_1b)B^{n_b} + (C_0b)B^0$$

That is, we will multiply each component with b, left-shift the results as required, and add them all. Note that, there are q multiplication subproblems of (n_b, n_b)-digits each, and 1 subproblem of (r, n_b)-digits. These subproblems can be recursively solved in the same manner; the earlier described Karatsuba method will be used whenever the two operands have equal number of digits.

Implementation

Below method shows an implementation of this algorithm. Please refer to [1] for implementation of methods referred by it.

For operands with small number of digits, the savings provided by this algorithm may not exceed the extra work required; making it less efficient. So, in below method we delegate to the schoolbook method when number of digits of smaller operand is less than a threshold. The optimal value of this threshold depends upon factors including the computer system.

Also note that, within every call of method *multiply_k*(), we need memory to store a few intermediate numbers, as done using local bigint variables La, Ra, Lb, Rb and $t1$. With our current bigint definition, these 5 bigints will occupy memory of $(5 \cdot WIDTH)$ integers in each method call. But if the bigint struct allocates memory only for required digits (instead of the fixed $WIDTH$), we can save some memory. In fact, some intermediate numbers require almost half the number of digits than the two operands.

```
/* input r must be a separate struct than a or b */
int multiply_k(bigint *a, bigint *b, bigint *r)
{
#define MULTIPLY_K_THRSH 15
  int    na, nb, k;
  bigint La, Ra, Lb, Rb, t1;

  /* internal swap to ensure: na >= nb */
  if(BINT_LEN(a) < BINT_LEN(b))
    SWAP(a, b);

  na = BINT_LEN(a);
  nb = BINT_LEN(b);

  if(nb <= MULTIPLY_K_THRSH)
    return multiply(a, b, r);

  if(na != nb)
  {
    int cstart, cend = WIDTH-1;    /* chunk-start, chunk-end */

    BINT_INIT(r, 0);

    /* invariant: digits after index "cend" have been processed */
    while(cend >= a->msd)
    {
      cstart = cend-nb+1;
      if(cstart < a->msd)
        cstart = a->msd;

      portion(&La, a, cstart, (cend-cstart+1));
      multiply_k(&La, b, &t1);
      shift_left(&t1, WIDTH-1-cend);
      add(r, &t1, r);

      cend = cstart-1;
    }

    return 0;
  }
```

```
  /* na = nb */
  k = (na + 1)/2;

  /* L: La*Lb,  R: Ra*Rb,  X: (La + Ra)*(Lb + Rb) */

  /* compute L in variable t1 */
  prefix(&La, a, na-k);
  prefix(&Lb, b, na-k);
  multiply_k(&La, &Lb, &t1);

  /* compute R in variable r */
  suffix(&Ra, a, k);
  suffix(&Rb, b, k);
  multiply_k(&Ra, &Rb, r);

  /* compute X in variable Ra */
  add(&La, &Ra, &La);
  add(&Lb, &Rb, &Lb);
  /* Ra and Rb free now, can be reused */
  multiply_k(&La, &Lb, &Ra);

  /* compute X-L-R in variable Ra */
  subtract(&Ra, &t1, &Ra);
  subtract(&Ra, r, &Ra);

  /* compute L (<<k*2) + (X-L-R) (<<k) + R */
  shift_left(&t1, k*2);
  shift_left(&Ra, k);
  add(r, &Ra, r);
  add(r, &t1, r);

  return 0;
}
```

■

References

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

[2] A. A. Karatsuba. *The Complexity of Computations*. Proc Steklov Institute of Mathematics, Vol 211 (1995), 169–183. `http://www.ccas.ru/personal/karatsuba/divcen.pdf`.