Implementing Basic Arithmetic for Large Integers: Introduction

Nitin Verma
mathsanew.com

June 18, 2021

The computer hardware provides facility to store numbers and do some basic arithmetic operations on them like addition, subtraction, multiplication and division. But generally this facility is limited to numbers in a certain range; for example on some computers, integers only up to $2^{64} - 1$ can be stored and operated upon. If we need to handle any arbitrarily large integers, we have to implement it in a program above this hardware.

In this series of three articles, we will learn how to implement large integers along with operations: addition, subtraction, multiplication and division. This article provides introduction and implements addition, subtraction and some common internal methods. The two subsequent articles will be covering multiplication and division.

Representation

We first need a way to represent a large integer in the computer. We can learn from the well-known *Positional Numeral System*, of which the commonly used *Decimal Numeral System* is an example. Each large integer can be represented as a sequence of *digits* where each digit itself should fit within a hardware provided integer. We will choose a base B for our numeral system, and each digit can have value in range 0 to $B - 1$.

In our implementation, we will assume 64-bit computer system, where types "int" and "long" in C language use 32-bits and 64-bits respectively; but this may vary across platforms. Each digit will be stored as a 32-bit hardware integer, allowing us to choose base B as large as 2^{32}. The sequence of digits will be stored as an array of 32-bit integers, which are available as type "unsigned int" in C. The following structure represents a large integer:

```
#define B (0x100000000)  /* 2^32 */
#define WIDTH (200)

typedef unsigned int uint;
typedef unsigned long ulong;

typedef struct bigint
{
  uint digits[WIDTH];
  int  msd;           /* index of the MSD in the "digits" array */
} bigint;
```

For simplicity, we have chosen to use a fixed-width array, but this can be improved for memory-efficiency. The *least-significant-digit* (LSD) and *most-significant-digit* (MSD) are $digits[WIDTH-1]$ and $digits[msd]$ respectively. The value of a bigint is given by:

$$\sum_{i=msd}^{WIDTH-1} digits[i] \cdot B^{WIDTH-1-i}$$

Throughout our implementation, we enforce the condition that digit $digits[msd]$ is not 0. That is, there will be no redundant 0s at the beginning of the digits sequence. We will refer this condition as “no leading zeros”. The only exception is the bigint for representing integer 0 itself; which has $msd = WIDTH - 1$ and $digits[msd] = 0$.

Arithmetic Operations

The hardware already provides some basic arithmetic operations on the 32-bit uint integers, that is, on the individual digits of our bigint. We will be using these digit-level hardware operations to help us implement the bigint-level operations. For the addition, subtraction, multiplication and division operations, we will take the algorithms for doing the same on decimal numerals which we learned in school, and adapt them for numerals with any arbitrary base B. These algorithms are sometimes referred as “School-book” algorithms. Additionally, multiplication will also be implemented using *Karatsuba Multiplication Algorithm.*

We will denote the two operands by $a = (a_{n_a-1}a_{n_a-2}\dots a_1a_0)$ and $b = (b_{n_b-1}b_{n_b-2}\dots b_1b_0)$. Each digit a_i (or b_i) denotes the digit of a (or b) at *position* i; the LSD is at position 0. Since the base is B, the digits are in range 0 to $B-1$.

The correctness proofs of these schoolbook algorithms for any base B are similar to that for decimal numerals.

Common Implementation Notes

1. The main arithmetic methods frequently refer to some common internal methods. All of such internal methods have been collectively placed in the section "Common Internal Methods".

2. All the arithmetic methods and some internal methods return -1 on failure after printing an error message. To keep the source code simple, the return value of calls to these methods is not handled. But for any practical deployment, error handling must be added.

3. The bigint inputs to all the methods (including internal ones) must satisfy the condition of "no leading zeros" defined in section "Representation". Implementation of these methods then ensure that their constructed output bigints also satisfy this condition. Having all bigints strictly satisfying this condition allows us to depend upon their number of digits (as returned by $BINT\_LEN()$) for quick comparison of two bigints.

4. As we will use the hardware operations on our 32-bit uint integers, we need to make sure that all the intermediate and final results can fit in the data-type used for them. For example, if the result requires more than 32-bits, but can fit in 64-bits, we will need to use the "long" type provided by C. Note that, if a result is representable by 2 digits in our base B numeral system, with $B = 2^{32}$, we can be sure that it is less than 2^{64} and so can fit in the 64-bit "unsigned long" type.

Addition

Like the schoolbook method, we will iteratively process one digit each from a and b, starting at the LSD (position 0). Since,

$$a_0 + b_0 \leq (B-1) + (B-1) = (1)B^1 + (B-2)B^0$$

so the *carry* produced by position 0, c_0, can be maximum 1. For the addition at position 1 we have:

$$c_0 + a_1 + b_1 \leq 1 + (B-1) + (B-1) = (1)B^1 + (B-1)B^0$$

So, the carry produced by position 1, c_1, also can be maximum 1. Thus, by repeating such argument, we know that the carry produced by any position can never exceed 1.

Lemma 1. *In the schoolbook addition algorithm with two operands in any base B, the carry produced by any position cannot exceed 1.*

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.

Also, the addition of two digits and carry at any position can produce values of up to 2-digit numeral in base B. So, as observed in section "Common Implementation Notes" point 4, these values should fit in the "unsigned long" type. Below method implements addition for the bigint integers.

```
/* input struct r can also be one of a or b */
int add(bigint *a, bigint *b, bigint *r)
{
  int   i;
  uint  carry = 0;
  ulong d;

  /* internal swap to ensure 'b' is not wider than 'a' */
  if(BINT_LEN(a) < BINT_LEN(b))
    SWAP(a, b);

  for(i = WIDTH-1; i >= a->msd; i--)
  {
    d = a->digits[i];

    if(carry || i >= b->msd)
    {
      if(carry)
        d = d + 1;

      if(i >= b->msd)
        d = d + b->digits[i];

      if(d >= B)
      {
        carry = 1;
        d = d - B;
      }
      else
        carry = 0;
    }

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

  return 0;
}
```

Subtraction

Like the schoolbook method, we will iteratively process one digit each from a and b, starting at the LSD (position 0). Below method implements subtraction for the bigint integers. Note that in any iteration, the subtraction of 1 (if borrow) and b's digit (which is maximum $B - 1$) from variable d (initially at least 0) still keeps it at least $-B$ always. So, with $B = 2^{32}$, the 64-bit signed type of "long" should suffice for d.

This method has to perform $a - b$, so if $a < b$, it internally swaps the two operands to ensure that $a \geq b$. The absolute value of the subtraction is placed in result r, and the method's return values of 0 and 1 represent positive and negative result respectively.

```
/* input struct r can also be one of a or b */
int subtract(bigint *a, bigint *b, bigint *r)
{
  int  i, borrow = 0, neg = 0;
  long d;

  /* internal swap to ensure 'a' >= 'b' */
  if(compare(a, b) == -1)
  {
    neg = 1;
    SWAP(a, b);
  }

  for(i = WIDTH-1; i >= a->msd; i--)
  {
    d = a->digits[i];

    if(borrow || i >= b->msd)
    {
      if(borrow)
        d = d - 1;

      if(i >= b->msd)
        d = d - b->digits[i];

      if(d < 0)
      {
        borrow = 1;
        d = d + B;
      }
      else
        borrow = 0;
    }

    r->digits[i] = d;
  }

  r->msd = i + 1;
  rm_leading_0s(r);

  return neg;
}
```

Printing

Since each digit of a bigint is a 32-bit binary number, we can print it using 8 hexadecimal digits as follows:

```
void bint_print(bigint *a)
{
  int i;

  for(i = a->msd; i <= WIDTH-1; i++)
    printf("%.8X", a->digits[i]);

  printf("\n");
}
```

To print a bigint using decimal digits, observe how the base-10 and base-10^k representations (for any integer $k \geq 1$) of any integer are related. For example, the base-10 numeral $(75001568045304267)_{10}$ can be expressed in base-1000 simply as: $(075\ 001\ 568\ 045\ 304\ 267)_{1000}$, where each of 075, 001 etc are a digit's value in base-1000.

So, for any given bigint, if we have its representation in base-10^9, we can obtain its base-10 representation by converting each of the base-10^9 digit into a base-10 numeral of 9 digits. Note that, base-10^9 is chosen because its digit (value not exceeding $10^9 - 1$) can fit in 32-bits.

The following method makes use of this fact. It first obtains the base-10^9 representation of the bigint by iteratively dividing it by 10^9 (the bigint *divide*() method will be implemented in a subsequent article). And then it converts each base-10^9 digit (stored in binary 32-bits) into a base-10 numeral using *printf*().

In this method, if decimal digits are directly generated by iteratively dividing the bigint by 10, instead of 10^9, it will require one bigint division per decimal digit generated. This means more number of bigint divisions, which are expected to be less efficient than hardware divisions. On the other hand, the iterative hardware divisions by 10 to generate decimal digits, as possibly done by *printf*(), are more efficient.

Also note that, if we use base $B = 10^9$ numeral system for our bigints, it will be straightforward to print their decimal representation.

```
void bint_print10(bigint *a)
{
  int    i;
  uint   dec[WIDTH*2];
  bigint b, t1, t2, rm;
  bigint *am, *qt;      /* am: 'a' modified */

  am = &t1;
  qt = &t2;
  BINT_INIT(&b, 1000000000);
  copy(am, a);
  i = 0;

  do
  {
    /* divide am by b and obtain quotient qt and remainder rm */
    divide(am, &b, qt, &rm);
    dec[i++] = rm.digits[WIDTH-1];
    SWAP(am, qt);
  }
  while(!BINT_ISZERO(am));

  i--;

  while(i >= 0)
    printf("%.9d", dec[i--]);

  printf("\n");
}
```

Common Internal Methods

This section contains all the common internal methods which are used while implementing the main methods for addition, subtraction, (and in subsequent articles) multiplication and division.

```
/* i must be in the range 0 to B-1 */
#define BINT_INIT(a, i) {(a)->digits[WIDTH-1] = i; \
                         (a)->msd = WIDTH-1;}

#define BINT_LEN(a) (WIDTH - (a)->msd)

#define BINT_ISZERO(a) (((a)->msd == WIDTH-1) && \
                        ((a)->digits[WIDTH-1] == 0))

#define SWAP(a, b) {bigint *t; t = a; a = b; b = t;}
```

```
/* d: destination, s: source */
static void copy(bigint *d, bigint *s)
{
  d->msd = s->msd;

  memcpy(&d->digits[d->msd], &s->digits[s->msd],
         BINT_LEN(s)*sizeof(uint));
}

/* ensure condition "no leading zeros" is met */
static void rm_leading_0s(bigint *a)
{
  int i = a->msd;

  while((i < WIDTH-1) && (a->digits[i] == 0))
    i++;

  a->msd = i;
}

/* copy n most-significant digits (prefix) from s to d */
static void prefix(bigint *d, bigint *s, int n)
{
  d->msd = WIDTH-n;

  memcpy(&d->digits[d->msd], &s->digits[s->msd],
         n*sizeof(uint));
}

/* copy n digits from s starting at index i, to d */
static void portion(bigint *d, bigint *s, int i, int n)
{
  d->msd = WIDTH-n;

  memcpy(&d->digits[d->msd], &s->digits[i],
         n*sizeof(uint));

  rm_leading_0s(d);
}

/* copy n least-significant digits (suffix) from s to d */
static void suffix(bigint *d, bigint *s, int n)
{
  portion(d, s, WIDTH-n, n);
}
```

```
static int shift_left(bigint *a, int n)
{
  int l = BINT_LEN(a);

  if(a->msd < n)
  {
    printf("shift_left: overflow\n");
    return -1;
  }

  /* left-shifting a 0 bigint will result in an invalid bigint */
  if(BINT_ISZERO(a))
    return 0;

  memmove(&a->digits[a->msd - n], &a->digits[a->msd],
          l*sizeof(uint));

  memset(&a->digits[a->msd - n + l], 0, n*sizeof(uint));

  a->msd = a->msd - n;

  return 0;
}

/* for a > b, a < b and a = b, return 1, -1 and 0 respectively */
static int compare(bigint *a, bigint *b)
{
  int i;

  if(BINT_LEN(a) > BINT_LEN(b))
    return 1;
  else if(BINT_LEN(a) < BINT_LEN(b))
    return -1;
  else
  {
    for(i = a->msd; i < WIDTH; i++)
    {
      if(a->digits[i] > b->digits[i])
        return 1;
      else if(a->digits[i] < b->digits[i])
        return -1;
    }
    return 0;
  }
}
```

■