Arbitrary-Precision Arithmetic

Table of Contents

Arbitrary-Precision arithmetic, also known as "bignum" or simply "long arithmetic" is a set of data structures and algorithms which allows to process much greater numbers than can be fit in standard data types. Here are several types of arbitrary-precision arithmetic.

Classical Integer Long Arithmetic

The main idea is that the number is stored as an array of its "digits" in some base. Several most frequently used bases are decimal, powers of decimal (10^4 or 10^9) and binary.

Operations on numbers in this form are performed using "school" algorithms of column addition, subtraction, multiplication and division. It's also possible to use fast multiplication algorithms: fast Fourier transform and Karatsuba algorithm.

Here we describe long arithmetic for only non-negative integers. To extend the algorithms to handle negative integers one has to introduce and maintain additional "negative number" flag or use two's complement integer representation.

Data Structure

We'll store numbers as a `vector<int>`, in which each element is a single "digit" of the number.

```
typedef vector<int> lnum;
```

To improve performance we'll use 10^9 as the base, so that each "digit" of the long number contains 9 decimal digits at once.

```
const int base = 1000*1000*1000;
```

Digits will be stored in order from least to most significant. All operations will be implemented so that after each of them the result doesn't have any leading zeros, as long as operands didn't have any leading zeros either. All operations which might result in a number with leading zeros should be followed by code which removes them. Note that in this representation there are two valid notations for number zero: and empty vector, and a vector with a single zero digit.

Output

Printing the long integer is the easiest operation. First we print the last element of the vector (or 0 if the vector is empty), followed by the rest of the elements padded with leading zeros if necessary so that they are exactly 9 digits long.

```
printf ("%d", a.empty() ? 0 : a.back());
for (int i=(int)a.size()-2; i>=0; --i)
    printf ("%09d", a[i]);
```

Note that we cast `a.size()` to integer to avoid unsigned integer underflow if vector contains less than 2 elements.

Input

To read a long integer, read its notation into a `string` and then convert it to "digits":

```
for (int i=(int)s.length(); i>0; i-=9)
    if (i < 9)
        a.push_back (atoi (s.substr (0, i).c_s
    else
        a.push_back (atoi (s.substr (i-9, 9).c
```

If we use an array of `char` instead of a `string`, the code will be even shorter:

```
for (int i=(int)strlen(s); i>0; i-=9) {
    s[i] = 0;
    a.push_back (atoi (i>=9 ? s+i-9 : s));
}
```

If the input can contain leading zeros, they can be removed as follows:

```
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

Addition

Increment long integer a by b and store result in a:

```
int carry = 0;
for (size_t i=0; i<max(a.size(),b.size()) || c
    if (i == a.size())
        a.push_back (0);
    a[i] += carry + (i < b.size() ? b[i] : 0);
    carry = a[i] >= base;
    if (carry)  a[i] -= base;
}
```

Subtraction

Decrement long integer a by b ($a \ge b$) and store result in a:

```
int carry = 0;
for (size_t i=0; i<b.size() || carry; ++i) {
    a[i] -= carry + (i < b.size() ? b[i] : 0);
    carry = a[i] < 0;
    if (carry)  a[i] += base;
}
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

Note that after performing subtraction we remove leading zeros to keep up with the premise that our long integers don't have leading zeros.

Multiplication by short integer

Multiply long integer a by short integer b ($b < base$) and store result in a:

```
int carry = 0;
for (size_t i=0; i<a.size() || carry; ++i) {
    if (i == a.size())
        a.push_back (0);
    long long cur = carry + a[i] * 1ll * b;
    a[i] = int (cur % base);
    carry = int (cur / base);
}
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

Additional optimization: If runtime is extremely important, you can try to replace two divisions with one by finding only integer result of division (variable `carry`) and then use it to find modulo using multiplication. This usually makes the code faster, though not dramatically.

Multiplication by long integer

Multiply long integers a and b and store result in c:

```
lnum c (a.size()+b.size());
for (size_t i=0; i<a.size(); ++i)
    for (int j=0, carry=0; j<(int)b.size() ||
        long long cur = c[i+j] + a[i] * 1ll *
        c[i+j] = int (cur % base);
        carry = int (cur / base);
    }
while (c.size() > 1 && c.back() == 0)
    c.pop_back();
```

Division by short integer

Divide long integer a by short integer b ($b < base$), store integer result in a and remainder in `carry`:

```
int carry = 0;
for (int i=(int)a.size()-1; i>=0; --i) {
    long long cur = a[i] + carry * 1ll * base;
    a[i] = int (cur / b);
    carry = int (cur % b);
}
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

Long Integer Arithmetic for Factorization Representation

The idea is to store the integer as its factorization, i.e. the powers of primes which divide it.

This approach is very easy to implement, and allows to do multiplication and division easily (asymptotically faster than the classical method), but not addition or subtraction. It is also very memory-efficient compared to the classical approach.

This method is often used for calculations modulo non-prime number M; in this case a number is stored as powers of divisors of M which divide the number, plus the remainder modulo M.

Long Integer Arithmetic in prime modulos (Garner Algorithm)

The idea is to choose a set of prime numbers (typically they are small enough to fit into standard integer data type) and to store an integer as a vector of remainders from division of the integer by each of those primes.

Chinese remainder theorem states that this representation is sufficient to uniquely restore any number from 0 to product of these primes minus one. Garner algorithm allows to restore the number from such representation to normal integer.

This method allows to save memory compared to the classical approach (though the savings are not as dramatic as in factorization representation). Besides, it allows to perform fast addition, subtraction and multiplication in time proportional to the number of prime numbers used as modulos (see Chinese remainder theorem article for implementation).

The tradeoff is that converting the integer back to normal form is rather laborious and requires implementing classical arbitrary-precision arithmetic with multiplication. Besides, this method doesn't support division.

Fractional Arbitrary-Precision Arithmetic

Fractions occur in programming competitions less frequently than integers, and long arithmetic is much trickier to implement for fractions, so programming competitions feature only a small subset of fractional long arithmetic.

Arithmetic in Irreducible Fractions

A number is represented as an irreducible fraction $\frac{a}{b}$, where a and b are integers. All operations on fractions can be represented as operations on integer numerators and denominators of these fractions. Usually this requires using classical arbitrary-precision arithmetic for storing numerator and denominator, but sometimes a built-in 64-bit integer data type suffices.

Storing Floating Point Position as Separate Type

Sometimes a problem requires handling very small or very large numbers without allowing overflow or underflow. Built-in double data type uses 8-10 bytes and allows values of the exponent in $[-308; 308]$ range, which sometimes might be insufficient.

The approach is very simple: a separate integer variable is used to store the value of the exponent, and after each operation the floating-point number is normalized, i.e. returned to $[0.1; 1)$ interval by adjusting the exponent accordingly.

When two such numbers are multiplied or divided, their exponents should be added or subtracted, respectively. When numbers are added or subtracted, they have to be brought to common exponent first by multiplying one of them by 10 raised to the power equal to the difference of exponent values.

As a final note, the exponent base doesn't have to equal 10. Based on the internal representation of floating-point numbers, it makes most sense to use 2 as the exponent base.

Practice Problems

- UVA - How Many Fibs?
- UVA - Product
- UVA - Maximum Sub-sequence Product
- SPOJ - Fast Multiplication
- SPOJ - GCD2
- UVA - Division
- UVA - Fibonacci Freeze
- UVA - Krakovia
- UVA - Simplifying Fractions
- UVA - 500!
- Hackerrank - Factorial digit sum
- UVA - Immortal Rabbits
- SPOJ - 0110SS
- Codeforces - Notepad