Long arithmetic

Long arithmetic - a set of tools (data structures and algorithms) that allow you to work with numbers much larger quantities than it allows standard data types.

Types of long integer arithmetic

Generally speaking, even just in the Olympiad set of problems is large enough, so we make a classification of different types of long arithmetic.

Classical long arithmetic

The basic idea is that the number is stored as an array of digits it.

Numbers can be used from one system or another value, commonly used decimal system and its power (ten thousand billion), or binary system.

Operations on numbers in the form of a long arithmetic made using "school" algorithms for addition, subtraction, multiplication, long division. However, they are also useful for fast multiplication algorithms: Fast Fourier Transformalgorithm and Karatsuba.

Described here only work with non-negative long numbers. Support for negative numbers must enter and maintain additional flag "negativity" numbers, or else work in complementary codes.

Data structure

Keep long numbers will be in the form of a Vector of Numbers $int$where each element - it is a single digit number.

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

To increase efficiency in the system will work in base billion, i.e. each element of the vector $lnum$contains not one, but 9numbers:

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

The numbers will be stored in a vector in such a manner that at first there are the least significant digit (ie, ones, tens, hundreds, etc.).

Furthermore, all the operations are implemented in such a manner that after any of them leading zeros (i.e. excess leading zeros) are not (of course under the assumption that prior to each leading zeros is also available). It should be noted that the implementation representation for the number zero is well supported just two representations: an empty vector of numbers and digits vector containing a single element - zero.

Output

The most simple - a conclusion of a long number.

First, we simply display the last element of the vector (or 0, if the vector is empty), and then derive all the remaining elements of the vector, adding zeros to their 9characters:

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

(Here a small fine point: not to forget to write down the cast (int), because otherwise the number $a.size()$will be unsigned, and if $a.size() \le 1$, then the subtraction overflows)

Reading

Reads a line in $string$, and then convert it into a vector:

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

If used instead of $string$the array $char$'s, the code will be even smaller:

```
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 number has leading zeros may be, they can remove, after reading the following way:

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

Addition

Adds to the number of $a$number $b$and stores the result in a:

```
int carry = 0;
for (size_t i=0; i<max(a.size(),b.size()) || carry; ++i) {
	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

Takes the number of $a$the number of b($a \geq b$) and stores the 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();
```

Here we are after subtracting remove leading zeros to maintain the predicate that they do not exist.

Multiplying the length of a short

Multiplies long $a$to short b($b < \text{base}$) and stores the 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();
```

Here we are after the division remove leading zeros to maintain the predicate that they do not exist.

(Note: The method **further optimization** . If speed is critical, you can try to replace two division one: count only the integer portion of a division (in the code is a variable $carry$), and then count on it the remainder of the division (with the help of one multiplication) . Typically, this technique allows faster code, but not very much.)

Multiplication of two long numbers

Multiplies $a$on $b$and results are stored 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() || carry; ++j) {
          long long cur = c[i+j] + a[i] * 1ll * (j < (int)
b.size() ? b[j] : 0) + carry;
          c[i+j] = int (cur % base);
          carry = int (cur / base);
     }
while (c.size() > 1 && c.back() == 0)
     c.pop_back();
```

Long division for a short

Divides long $a$for short b($b < \text{base}$), private stores a, balance $carry$:

```
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 arithmetic in factored form

The idea here is not to store the number itself, and its factorization, ie each incoming power it simple.

This method is also very simple to implement, and it is very easy to make multiplication and division, but it is impossible to perform the addition or subtraction. On the other hand, this method saves memory in comparison with a "classic" approach, and allows the multiplication and division significantly (asymptotically) faster.

This method is often used when you need to make on the delicate division module: then enough to store a number in the form of powers to the prime divisors of the module, and another number - balance on the same module.

Long arithmetic system of simple modules (Chinese theorem or scheme Garner)

The bottom line is that some selected system modules (usually small, fit into standard data types), and the number is stored as a vector of residuals from dividing it by each of these modules.

According to the Chinese Remainder Theorem, it is enough to uniquely store any number between 0 and the product of these modules minus one. Thus thereGarner algorithm that allows it to produce recovery from modular form into the usual "classical" form number.

Thus, this method saves memory compared to the "classic" long arithmetics (although in some cases not as radically as the factorization method). Furthermore, in a modular fashion, you can very quickly make addition, subtraction and multiplication, - all for adding identical time asymptotically proportional to the number of modules in the system.

However, given the price of all this is very time-consuming translation number of this modular form in the usual form, which, in addition to considerable time costs also need the implementation of "classical" long arithmetic multiplication.

In addition, to make **the division** of numbers in this representation system for simple modules is not possible.

Types of fractional long arithmetic

Operations on fractional numbers found in the Olympiad problems are much less common, and work with huge fractional numbers is much more complicated, so the competitions found only specific subset of fractional long arithmetic.

Long arithmetic in an irreducible fraction

The number appears as an irreducible fraction $\frac{a}{b}$, where $a$and b- integers. Then all operations on fractional numbers easily reduced to operations on the numerator and denominator of these fractions.

Usually, when this storage numerator and denominator have to use long arithmetic, but, however, it is the easiest form - "classic" long arithmetic, although sometimes is sufficiently embedded 64-bit numeric type.

Isolation of floating point position as a separate type

Sometimes the problem is required to make calculations with very large or very small numbers, but it does not prevent them from overflowing. Built $8-10$-byte type $double$, as is known, allows the exponent value in a range $[-308;308]$, which sometimes may be insufficient.

Reception, in fact, very simple - introduce another integer variable responsible for the exponent, and after each operation, the fractional number of "normal", iereturns to the segment $[0.1;1)$, by increasing or decreasing exponential.

When multiplying or dividing two such numbers should accordingly lay down or subtract their exponents. When adding or subtracting before proceeding number should lead to an exponential one, which one of them is multiplied by the 10difference in the degree exponents.

Finally, it is clear that it is not necessary to choose 10as the base of the exponent. Based on the embedded device floating-point types, the best seems to put an equal basis 2.