MAXimal

added: 10 Jun 2008 18:06
Edited: 23 Mar 2012 3:45

Long arithmetic

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

Contents [hide]

Types of long integer arithmetic

Generally speaking, even if only in the Olympiad set of problems is large enough, so proizvedëm classification of different types of long arithmetic.

Classical long arithmetic

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

The 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 produced by a "school" of algorithms for addition, subtraction, multiplication, long division. However, they are also useful algorithms for fast multiplication: Fast Fourier transform and the Karatsuba algorithm.

Described here only work with non-negative long numbers. To support 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 - a single digit number.

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

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

```
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., extra zeros at the beginning number) does not (of course, on the assumption that, before each operation the leading zeros, also available). It should be noted that the implementation representation for the number zero is well supported from two views: the empty vector numbers and vector numbers containing a single element - zero.

Conclusion

The most simple - it's the 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 $9$characters:

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

(Here, a little subtle point: not to forget to write down the cast (int), because otherwise the number $a.size()$will be unsigned, and if $a.size() \leq 1$, it will happen in the subtraction overflow)

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 more compact:

```
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 $a$the 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 after subtraction remove leading zeros in order to maintain a predicate that they do not exist.

Multiplying the length of a short

Multiplies long $a$to short b($b < \mathrm{base}$) 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 after dividing remove leading zeros in order to maintain a predicate that they do not exist.

(Note: The method **further optimization** . If performance is critical, you can try to replace the two division one: to 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($b < \mathrm{base}$), private preserves 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

Here the idea is to keep not the number itself, and its factorization, ie power of each of her children simple.

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

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

Long arithmetic in simple modules (Chinese theorem or scheme Garner)

The bottom line is that you choose a certain system modules (usually small, fit into standard data types), and the number is stored as a vector of residuals from his division to 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 there Garner algorithm , which allows to make a recovery from the modular form in the usual "classical" form number.

Thus, this method saves memory compared to the "classical" long arithmetic (although in some cases not as radically as the factorization method). In addition, 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, all this is very time consuming given the price of the translation of this modular form in the usual form, which, in addition to considerable time-consuming, require also the implementation of "classical" long arithmetic multiplication.

In addition, to make **the division** of numbers in this representation in 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 large fractional numbers is much more difficult, so in the Olympic Games found only a specific subset of the fractional long arithmetic.

Long arithmetic in an irreducible fraction

Number is represented 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 the fractions.

Normally this storage numerator and denominator have to use long arithmetic, but, however, it is the simplest form - the "classical" 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 $8-10$-baytovy type $double$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 that is responsible for the exponential, and after each operation, the fractional number of "normal", ie, returns to the segment $[0.1;1)$, by increasing or decreasing exponential.

When multiplying or dividing two such numbers it is necessary to lay down accordingly 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 of exponents.

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

22 Комментариев e-maxx Войти

Лучшее вначале Поделиться Избранный

Присоединиться к обсуждению...

Vudu4 • 2 года назад
А есть ли на сайте деление длинного на длинное желательно не сам код, а пояснения к нему?
18 • Ответить • Поделиться ›

Pavel Martynov → Vudu4 • 8 месяцев назад
Вот вариант http://cppalgo.blogspot.com/20...
2 • Ответить • Поделиться ›

Hero • 2 года назад
А как можно сравнить длинные числа?

6 ^ | ˅ • Ответить • Поделиться ›

e_maxx Модератор ➦ Hero • 2 года назад

Сначала удалить лидирующие нули из обоих чисел. Затем, если длины векторов не совпали, то число с меньшим количеством цифр меньше. Если же длины совпали, то идём по элементам векторов от конца к началу и сравниваем соответствующие элементы из первого и второго векторов: если они не равны, то одно из чисел меньше другого.

5 ^ | ˅ • Ответить • Поделиться ›

hunt • 2 года назад

что такое 1ll?

5 ^ | ˅ • Ответить • Поделиться ›

e_maxx Модератор ➦ hunt • 2 года назад

единица в 64-битном типе данных (long long).

6 ^ | ˅ • Ответить • Поделиться ›

fiona • год назад

Как возводить в степень длинное число и как извлекать корень из длинного числа?

3 ^ | ˅ • Ответить • Поделиться ›

Igor Adamenko ➦ fiona • год назад

В степень — умножение длинного на длинное.
Корень думаю никак.

2 ^ | ˅ • Ответить • Поделиться ›

Nicușor Chiciuc ➦ Igor Adamenko • 10 месяцев назад

There are algorithms for extracting square roots by hand. It could be possible to implement them for this type of data types also, I guess.

2 ^ | ˅ • Ответить • Поделиться ›

Андрей Самсонов ➦ fiona • месяц назад

корень можно извлечь, разложив его в ряд тейлора.

^ | ˅ • Ответить • Поделиться ›

Ivan Nikulin • 2 года назад

А зачем нам убирать ведущие нули при умножении, если их все равно не будет никогда? Или контрпример в студию :)

3 ^ | ˅ • Ответить • Поделиться ›

e_maxx Модератор ➦ Ivan Nikulin • 2 года назад

Длинное число умножить на ноль, например.

8 ^ | ˅ • Ответить • Поделиться ›

rasul • 2 года назад

А В СЛОЖЕНИИ ОТКУДА МЫ БЕРЕМ BASE?

2 ^ | ˅ • Ответить • Поделиться ›

e_maxx Модератор ➦ rasul • 2 года назад

base - это константа, общая для всех функций, см. раздел "Структура данных" (для примера в статье эта константа равна миллиарду).

4 ^ | ˅ • Ответить • Поделиться ›

Botan • 2 года назад

В примечании по производительности вы предлагали заменить два деления делением и умножением
А можно ли просто использовать "div" или "ldiv" из "stdlib.h"? Они ищут одновременно и делимое, и остаток.
Если я не ошибаюсь, это будет работать немного быстрее, чем предложенный вам вариант.

2 ^ | ˅ • Ответить • Поделиться ›

e_maxx Модератор ➦ Botan • 2 года назад

Да, вы правы, так можно сделать, и наверняка это будет работать быстрее.

5 ^ | ˅ • Ответить • Поделиться ›

elluZion • год назад

Не могли бы вы подробнее пояснить, как работает преобразование строки в длинное число, если строка представлена массивом символов? В частности, что вот это означает:

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

1 ^ | ˅ • Ответить • Поделиться ›

Svyat • 2 года назад

Можно использовать вместо
c[i+j] = int (cur % base);
carry = int (cur / base);
следующее:
carry = int (cur / base);
c[i+j] = int (cur - carry * base);

У меня работает немного быстрее

1 • Ответить • Поделиться ›

e_maxx Модератор → Svyat • 2 года назад

Да, такой приём есть, он обычно действительно немного ускоряет. Только в произведении carry и base надо не забыть умножить на 1LL, чтобы не произошло переполнения типа int.

1 • Ответить • Поделиться ›

BorisKozhyhovskiy • 15 дней назад

котр тест на алгоритм сложения: 10000000001 + 100000000
программа выдаёт 11099000001

• Ответить • Поделиться ›

Андрей • 5 месяцев назад

Возможна ли реализация с основанием сс 2^32, т.е. хранить не по 9, а по 32 элемента? И много ли придется поменять в арифметических операциях?

• Ответить • Поделиться ›

steve • 6 месяцев назад

Здравствуйте.
А можно пример кода, чтобы заполнять таблицу длинными числами (ДП про количество способов, когда надо хранить сразу несколько значений, а не только 2).
Заранее спасибо.

• Ответить • Поделиться ›