# DOCUMENTATION
# XLN

XLN - 8991778aa03f2e429ecd6ec91594864ebaf42bbb
0.2.6.
KI - 1

Likhachev Aleksei

28.11.2021

# Table of contents

# 1 Introduction

How quickly can we calculate the value of the expression 17^15 ? It will take time, but it is a trivial task. For example, can we calculate 17^15 digits of Pi with pencil and paper? Conditionally speaking yes, but it would take too much time. For this kind of calculations people invent various computing machines.

Modern processors work with 64 or 32 bit words. Undoubtedly, that's a lot and in everyday tasks is enough. But what if we need more? Any mathematical or physical calculations? In that case, the number that seemed very large recently will become incredibly small and useless at all. To get out of this unpleasant situation, we need to abandon the usual machine arithmetic and create our own abstraction on top of it - new algorithms, rules, structures. Then we will overcome the ceiling of 64 bits and be only limited by computer memory, which will already be enough for a very large class of different calculations.

Consider two ways to store large numbers. One is to store the digits of a number modulo 2^30 in separate memory cells. This will make our usual operations just a little bit more complicated. In this case, the maximum possible number is

$$\sum_{i=1}^{2^{32}} 2^{29} * 2^{30*k}$$

It is, who would have thought, very big. You would need about sixteen gigabytes of free space to store it.  And with the _XLN_SIZE64 flag this number will be

$$\sum_{i=1}^{2^{64}} 2^{29} * 2^{30*k}$$

And to store it already, one hundred and twenty-eight exabytes of free space would be required. As of this writing, the approximate price of such storage capacity is two and a half million dollars. So, this approach covers calculations of any complexity.

# 2 Theorems and Algorithms

## 2.1 Number representation and storage

We are used to the decimal number system, so it seems that taking 10 as the base of our number system would also be a good solution. But that's not true at all. This approach would require too many 4-bit memory cells to store the number, and is totally inefficient. We should use as large a machine word as possible, so that overflow will not occur when working with digits of a number. In this case, base 2^30 is good enough for us. We will store the number in cells of 32 bits each. This leaves us with two free bits, that we can use in any way. Note also that it is more efficient to store numbers starting from the lowest bits, as in this case, the change in the number will be carried out by shifts in the direction of increasing the index. Our number in this case can be represented as follows:

| **2^0** | **2^30** | **...** | **2^i** |
|---|---|---|---|
| $n_0$ | $n_1$ | … | $n_i$ |

Along the i index is the i digit of our number.

## 2.2 A - The algorithm for initializing the number of their string

For your comfort, the number initialization function takes a string with a number written in base 10^9 by default. The parsing algorithm is trivial and consists of sliding down the input string starting at the end of 9 characters. We save them into a temporary array, then normalize it and pass it for conversion to the number system modulo 2^30.

## 2.3 B - Algorithm for converting a number to a system modulo 2^30

## 2.4 $B^{-1}$ reverse conversion algorithm

Now that we got the number from the string by another modulo, we need to convert it. To do this we will divide it by 2^30 and the resulting remainder is the digit from the end we need. Let a be a number in another number system, and we calculate n

$$n[i] = \sum_{j=0}^{n-1} \frac{10^{9j} a[j]}{2^{30j}} \mod 2^{30}$$

And in the opposite direction obviously

$$a[i] = \sum_{j=0}^{n-1} \frac{2^{30j} n[j]}{10^{9j}} \mod 10^{9j}$$

## 2.5 Number output algorithm

As we store the digits of the least significant digits first, for the output we first need to check if the current number of digits is equal to zero, if so all we need to do is to output zero, otherwise the last digit, and then the cycle from obj->_current - 2 to zero will add the digits with the required

number of zeros. The difference in outputting numbers modulo 2^30 and 10^9 is only in necessary number of zeros for the addition. In the first case it is 10, in the second case it is 9.

## 2.6 C Algorithm of the sum

Suppose we want to add two numbers x and y and store the result in r. Solving the problem on paper, it would look like this.

$$x + y = \sum_{j=0}^{size(x)-1} \left(2^{30j} * x[j]\right) + \sum_{j=0}^{size(y)-1} \left(2^{30j} * y[j]\right) = \sum_{j=0}^{size(y)} 2^{30j} * (x[j] + y[j] + \text{carry})$$

Since the digit cannot be greater than the base of the system, it is transferred to a larger digit. It is calculated as follows.

$$\text{carry} = \begin{cases} 0 & \text{if j = 0} \\ \dfrac{x[j] + y[j]}{2^{30}} & \text{otherwise} \end{cases}$$

Note that

$$\frac{x[j] + y[j]}{2^{30}} \leq \frac{(2^{30} - 1) + (2^{30} - 1)}{2^{30}} = \frac{2^{30} - 1}{2^{29}} = 2 - \frac{1}{2^{29}} < 2 \Rightarrow \text{carry} = \mathbb{Z}/2\mathbb{Z}$$

## 2.7 D Subtraction algorithm

It is essentially just a slight modification of the addition algorithm. It is mandatory to assume that the left number is greater than the right one. Otherwise the result should be given a negative sign and the summands should be swapped. If j is greater than the size of any of the numbers, then we take zero instead.

$$x - y = \sum_{j=0}^{size(x)-1} \left(2^{30j} * x[j]\right) - \sum_{j=0}^{size(y)-1} \left(2^{30j} * y[j]\right) = \sum_{j=0}^{size(y)} 2^{30j} * (x[j] - y[j] - \text{carry})$$

$$\text{carry} = \begin{cases} 0 & \text{if j = 0} \\ \dfrac{x[j] - y[j]}{2^{30}} \mod 2 & \text{otherwise} \end{cases}$$

## 2.8 E Slow multiplication algorithm

Suppose we want to multiply two large numbers x and y. On paper again, the process would look like this

$$x \times y = \sum_{i=0}^{size(x)-1} \left(2^{30i} * x[i]\right) \times \sum_{j=0}^{size(y)-1} \left(2^{30j} * y[j]\right) =$$

$$\underbrace{\sum_{j=0}^{size(y)-1}(2^{30(i+n)}x[i]*y[j]) + \sum_{j=0}^{size(y)-1}(2^{30(i+n-1)}x[i-1]*y[j]) + \ldots + \sum_{k=0}^{size(y)-1}(2^{30n}x[0]*y[j])}_{(size(x)-1)*(size(y)-1)\text{ sums}}$$

Note that there can be several sums with multiplication by 2^30(i+j). As a result, the digits of the number are calculated as follows.

$$r[i+j] = \sum_{\alpha=0,\beta=j}^{\alpha\le i,\beta\ge 0}(2^{30(\alpha+\beta)}x[\alpha]*y[\beta] + \text{carry})$$

The asymptotic complexity of the algorithm is O(n*m), where n and m are the sizes of numbers, respectively.

## 2.9 D Slow division algorithm

This method of division should be used in the case of small numbers. Suppose we want to divide the numbers x and y

$$\frac{x}{y} = \frac{\sum_{i=0}^{size(x)-1}2^{30i}x[i]}{\sum_{i=0}^{size(y)-1}2^{30j}y[j]} = \frac{2^{30i}x[i] + 2^{30(i-1)}x[i-1] + \ldots + x[0]}{2^{30j}y[j] + 2^{30(j-1)}y[j-1] + \ldots + y[0]}$$

We could subtract the divisor from the divisor, but this algorithm is too slow as the quotient increases. Its complexity equals O(q*n), where q is the quotient and n the number of digits in the divisor.

Note that the values under the largest index contribute the most to the final result.

$$\frac{2^{30i}x[i]}{2^{30j}y[j] + 2^{30(j-1)}y[j-1] + \ldots + y[0]} \ge \frac{2^{30(i-1)}x[i-1]}{2^{30j}y[j] + 2^{30(j-1)}y[j-1] + \ldots + y[0]} \ge \ldots \ge \frac{2^{0}x[0]}{2^{30j}y[j] + 2^{30(j-1)}y[j-1] + \ldots + y[0]}$$

Then we can find the approximate quotient. From this point, without loss of generality, let us assume that the dimensions of the numbers are equal.

$$q' = \min(\frac{2^{30i}x[i] + x[i-1]}{y[i-1]}, 2^{30} - 1)$$

Let's prove that $q' \ge q$, at $q' = 2^{30} - 1$ this is correct. Otherwise we have

$$\frac{b*x[i] + x[i-1]}{y[i-1]} \ge \frac{\sum_{i=0}^{size(x)-1}2^{30i}x[i]}{\sum_{i=0}^{size(x)-1}2^{30i}y[i]} \to (b*x[i] + x[i-1])*y \ge y[i-1]*x$$

In the worst case we have

$b * y \geq x$ , which is understandably true. The theorem is proved.

The fact that if $y[i-1] \geq \frac{\text{base}}{2}$ , then $q'$ cannot be different from the true value

more than two will be omitted without proof. It will be described later in the next updates.

# 3 Function

xln* xln_alloc(uint32_t size);

This function allocates memory for the structure and "_size" cells to store it. Returns a pointer to the allocated memory or NULL in case of an error

xln* xln_realloc(xln* obj, uint32_t new_size);

This function reallocates memory for the structure and "_size" cells to store it. Returns a pointer to the allocated memory or NULL in case of an error

**void** xln_free(xln* obj);

This function frees the previously allocated memory for the object

xln* xln_normalize(xln* obj);

This function normalizes the number by removing leading zeros from

xln* xln_convert(xln* obj, uint64_t a[], uint32_t n);

This function converts a number obtained from a string by converting it from the modulo 10^9 number system to the modulo 2^30 number system.

xln* xln_convert_reverse(xln* obj);

This function converts a number obtained from a string by converting it from the modulo 2^30 number system to the modulo 10^9 number system.

xln* xln_init_string(xln* obj, **const char*** str);

This function initializes xln with a number from the string modulo 10^9,
normalizes it with the xln_norm function and returns a pointer to the object otherwise NULL

ln* xln_init_int(xln* obj, int32_t val);

This function initializes xln with a number from int, normalizes it with the xln_norm function and returns a pointer to the object otherwise NULL

**void** xln_print(xln* obj);

This function outputs to the console a large number modulo 2^30

**void** xln_print_r(xln* obj);

This function outputs to the console a large number modulo 10^9

## xln* xln_sum(xln* result, xln* left, xln* right);

This is a function that adds two large numbers element by element. If the pointer to the result is zero, it allocates memory for it. It returns a pointer to the sum of the two numbers or NULL in case of an error.

left->_mem[i] + right->_mem[i] + carry <= (2^30 - 1) + (2^30 - 1) + 1 = 2^31 -1 < 2^31

The addition algorithm is as follows:
1. Assign j , carry <- 0 where j is the bit index and carry the overflow index.
2. Add the digits modulo the system and calculate the overflow. [a[j] + b[j] + carry / base ]
3. Increment j by one, if we have passed all digits, assign the last value of the remainder and complete the execution.

## xln* xln_sub(xln* result, xln* left, xln* right);

This function subtracts one number from another with a sign. If no memory has been allocated for the resulting number, it is allocated and returns the result or NULL in case of an error.

## xln* xln_smul_int(xln* result, xln* left, uint32_t right);

A function for multiplying a large number by a small number smaller than the base of the system. If no memory has been allocated for the result, allocate it. Returns the result or zero in case of an error.

## xln* xln_smul(xln* result, xln* left, xln* right);

Slow multiplication function of two large numbers.
Automatically allocates memory for the result if the pointer is
zero. Returns the result or zero in case of an error. The asymptotic of the algorithm is equal to the product of their lengths n and m O(n*m). Suitable for small numbers.

## xln* xln_mul(xln* result, xln* left, xln* right);

In next update…

## xln* xln_sdiv_int(xln* result, xln* left, uint32_t right);

Dividing a large number by a number smaller than the base of the number system. Returns the result or NULL in case of an error.

## xln* xln_sdiv(xln* result, xln* left, xln* right);

Inefficient division algorithm. Suitable for small numbers.

The idea is that if the digit in the high-order is greater than half the base of the system, then q = (l[j+n]*base + l[j+n-1])/r[n-1] is between [
q_true, q_true + 2]. The division process itself is the division of l[j:j+n] by r, where j = l.size-r.size. We literally slide over the divisor.

Rough estimate of algorithm's complexity is O(n^2) where n is number of words in divisible.

## xln* xln_smod(xln* result, xln* left, xln* right);

An algorithm for finding the remainder modulo n based on the slow division algorithm for large numbers. Returns a pointer to the result or zero in case of error.

Rough estimate of algorithm's complexity is O(n^2) where n is number of words in divisible.

## lnbool xln_equal(xln* left, xln* right);

The function compares two numbers, starting from the highest digit. Returns 1 if the numbers are equal otherwise zero.

## lnbool xln_noequal(xln* left, xln* right);

The function compares two numbers, starting from the highest digit. Returns 1 if the numbers are not equal otherwise zero.

## lnbool xln_eqmore(xln* left, xln* right);

The function compares two numbers starting with the highest digit. Returns 1 if the left number is greater than or equal to the right number, otherwise zero.

## lnbool xln_eqless(xln* left, xln* right);

The function compares two numbers starting with the highest digit. Returns 1 if the left number is less than or equal to the right number, otherwise zero.

## lnbool xln_more(xln* left, xln* right);

The function compares two numbers starting with the highest digit. Returns 1 if the left number is greater than the right number otherwise zero.

## lnbool xln_less(xln* left, xln* right);

The function compares two numbers starting with the highest digit. Returns 1 if the left number is less than the right number otherwise zero.

# 4 Conclusion

Here are all the algorithms and theorems used to implement arbitrary-precision arithmetic. This library has great potential and will be useful in many theoretical and applied problems.

This product is implemented as a stand-alone part of the Xenon™ ecosystem. For best results, the XPU and X2M modules should also be used but can also work stand-alone.

As suggested by the latest version of the KI license under which this product is distributed, all necessary signatures should be checked to avoid any problems.