Arbitrary precision computation

Xavier Gourdon and Pascal Sebah
February 17, 2001[1]

1 Introduction

The basic numerical data types provided in most langages (C, C++, Fortran, ...) are limited to a given precision which depends on the data type itself and the machine. As a consequence, the computation of a numerical constant with more than (say) 10 or 20 digits needs a specific development to handle high precision numbers.

Several mathematical softwares (Maple, Mathematica, ...) directly provide the possibility to work with a non limited precision. They can be used to prototype algorithms to compute constants for example, but are highly non efficient since they are not dedicated to the numerical computation. Computer libraries (in C or C++ for example) exists that are much more efficient for an practical usage (see links below). Most of these are dedicated to the computation with a reasonable number of digits (say less than several thousands) and suffer from the fact that efficient algorithms on very large numbers are not implemented. Several isolated packages are dedicated to computation of huge numbers.

Netherveless, it remains of interest to know how to handle high precision numbers, either for writing your own package to have more control on it (local optimizations, memory management, ...) or to have an idea about the complexity of the problem.

2 Representation of high precision numbers

The arbitrary precision computational aspect depends on a choice of a representation for large numbers.

2.1 Big integers

Usually, a big integer N is handled thanks to it representation in a given base B (B usually depends on the maximal size of the basic data types) :

$$X = x_0 + x_1 B + x_2 B^2 + \cdots + x_n B^n. \qquad (*)$$

The *coefficients* x_i (also called *limbs*) are basic number data types (such as *long* or *double* in C) and satisfy $0 \leq x_i < B$.

The choice of the base B is important for several reasons :

- The base B must fit in a basic data type.

[1] This pages are from *//numbers.computation.free.fr/Constants/constants.html*

- The base B must be as large as possible to decrease the size of the representation of large numbers and to decrease the cost of the basic algorithms running on them (sum, product, ...), but sufficiently small such that internal operations running on the coefficients always fit in the basic data type choosen.

- The base B can be choosen as a power of 10 for practical reasons (final output, debugging) or as a power of 2 to benefit from very low level routines handling data bits. Nethertheless, switching from a representation in base of the form $B = 2^m$ to a base of the form $B = 10^n$ is not a trivial task if one wants to do it efficiently for huge numbers. Some problems like deciding the primality of a big number can use an implementation with a power-of-two base since the final output is only a boolean.

As for the choice of the basic data type used to store the coefficients x_i, it seems more natural to take the type `long` (in C). Netherveless, `long` type is usually a 32 bits data storage and the use of the C `double` type (usually 53 bits of precision) enables to have a larger basic storage. Moreover, some processors (like pentium) are often optimized for operations on `double`, making this choice better for such architectures.

2.2 Big relative integers

Big negative integers can be representated either as big integers with a sign associated to it, or only in the form (*) with $x_n < 0$ and $0 \leq x_i < B$ for $i < n$.

2.3 Big real numbers

Real numbers can be represented in two ways : either as a fixed point representation

$$X = x_0 + \frac{x_1}{B} + \frac{x_2}{B^2} + \cdots + \frac{x_n}{B^n},$$

or as a floating point representation in the form

$$x = y * B^e,$$

where e is an integer exponent and y a fixed point representation. The floating point representation is much more general but the presence of the exponent e leads to additionnal operations on them which are not so easy to implement.

3 Basic operations

The basic operations on large numbers are the sum, difference and product of large numbers, and the division by a short integer. The division by a large number is treated in the next section.

3.1 Rearrangement

Rearrangement is a basic operation which consists in transforming a *non canonical* representation

$$X = \sum_i x_i B^i,$$

with x_i eventually not between 0 and $B-1$, into a *canonical* one

$$X = \sum_i y_i B^i, \qquad 0 \leq y_i < B.$$

Starting from $i = 0$, it consists essentially in euclidian divisions of x_i by B and reporting the carry on x_{i+1}.

3.2 Sum or difference of large integers

Given two big integers X and Y in canonical form

$$X = \sum_{i=0}^{m} x_i B^i, \quad Y = \sum_{i=0}^{n} y_i B^i,$$

the big integer $Z = X + Y$ is obtained by adding x_i and y_i for all i and doing a final rearrangement on Z. The difference $Z = X - Y$ is obtained in a similar way.

3.3 Product by a small integer

Let q an integer such that qB fits in the basic storage data type of coefficients. The product of a large integer X by q is obtained by multiplying each x_i by q and reporting the carry, starting at $i = 0$.

3.4 Division by a small integer

Starting at $i = n$, the division of X by a small integer q is obtained by dividing x_i by q and reporting the carry multiplied by the base B on x_{i-1} and so on.

3.5 Product of two large integers

Given two big integers X and Y in canonical form

$$X = \sum_{i=0}^{m} x_i B^i, \quad Y = \sum_{i=0}^{n} y_i B^i,$$

the big integer $Z = X * Y$ can be obtained thanks to the formula

$$z_i = \sum_{k+l=i} x_k y_l,$$

with a final rearrangement. To make this operation valid, B^2 times $\sup(n, m)$ must fit in the basic storage data type. This constraint usually permits to choose the size of the base B.

4 Fast product of large integers

For large integers of size n, all the operations $+$, $-$ and product or division by a small integer gave a *complexity* of $O(n)$. That means that the number of basic operations on basic data storage type is proportionnal to n. As for the classical product, the complexity is $O(n^2)$ for multiplying two integers of size n. When n becomes big, this cost becomes very large compared to $O(n)$. When handling huge integers, it is then of interest to try to obtain a faster algorithm for multiplication.

4.1 Karatsuba multiplication

Karatsuba was the first to observe that multiplication of large integer can be made faster than $O(n^2)$.

The technique is recursive. Starting from two large integers X and Y of size n, we cut X and Y in two parts

$$X = X_0 + B^m X_1, \quad Y = Y_0 + B^m Y_1, \qquad m = \lfloor n/2 \rfloor.$$

To obtain XY, we write

$$XY = X_0Y_0 + B^m(X_0Y_1 + X_1Y_0) + B^{2m}X_1Y_1.$$

The classical approach would consists in computing the four products X_0Y_0, X_0Y_1, X_1Y_0 and X_1Y_1 to obtain XY. The idea of Karatsuba method is to perform three big products instead of these four, by writing

$$XY = P_0 + B^m(P_1 - P_2 - P_0) + B^{2m}P_2,$$

with

$$P_0 = X_0Y_0, \quad P_1 = (X_0 + X_1)(Y_0 + Y_1), \quad P_2 = X_1Y_1.$$

Two sums and one difference on integers of size $n/2$ are also needed.

The operation can be continued recursively, so finally the cost $C(n)$ satisfy

$$C(n) = 3C(n/2) + O(n),$$

which entails $C(2^k) = O(3^k)$ or

$$C(n) = O\left(n^{\log(3)/\log(2)}\right) = O\left(n^{1.5849\ldots}\right).$$

A variant of the method consists in writing

$$XY = P_0 + B^m(P_2 + P_0 - P_1) + B^{2m}P_2,$$

with

$$P_0 = X_0Y_0, \quad P_1 = (X_0 - X_1)(Y_0 - Y_1), \quad P_2 = X_1Y_1.$$

This latest approach can be used to avoid difficult carry treatment.

Some care should be taken in the implementation of Karatsuba multiplication in order to make it really efficient. The recursion should be stopped at a size m depending on the implementation and the machine. Under size m, the classical product is used. In the practice, Karatsuba multiplication starts to be better than the classical multiplication on integers with more than several hundred digits.

Karatsuba algorithm can be generalized : instead of cutting in two parts, one can cut in k parts and perform $2k-1$ products to recover the final product XY, leading to a complexity of $O(n^{\alpha})$ with $\alpha = \log(2k-1)/\log(k)$. Nethertheless, these generalizations are difficult to implement and often worst than the FFT method presented below.

4.2 FFT based multiplication

The product of two larges integers of size n can be done in time $O(n \log(n))$ thanks to Fast Fourier Transform techniques. See *FFT based multiplication of large numbers* for more details.