FFT based multiplication of large numbers

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

Multiplication of large numbers of n digits can be done in time $O(n\log(n))$ (instead of $O(n^2)$ with the classic algorithm) thanks to the Fast Fourier Transform (FFT). This page presents this technique along with practical considerations.

Basic definitions and notations about large number representation can be found in *Arbitrary precision computation*.

1 Introduction

Two large integers X and Y of size at most $n-1$ can be written in the form

$$X = P(B), \quad Y = Q(B)$$

where B is the base (usually $B = 10$ or a power of 10) and P and Q two polynomials

$$P(z) = \sum_{j=0}^{n-1} x_j z^j, \quad Q(z) = \sum_{j=0}^{n-1} y_j z^j.$$

Denoting by $R(z)$ the polynomial obtained by the product of $P(z)$ by $Q(z)$, we have $XY = R(B)$ and a final rearrangement on the coefficients of $R(z)$ permits to obtain the product XY. Thus, we are lead to the problem of multiplying two polynomials of degree $< n$.

A polynomial of degree $< m$ is uniquely defined from its evaluations at m distincts points (with Lagrange interpolation for example). Thus, to obtain the product $R(z) = P(z)Q(z)$, it suffices to compute the values $R(w_k)$ at $2n$ distincts points w_k, that is computing $P(w_k)$ and $Q(w_k)$. The Fast Fourier Transform (FFT) idea consists in choosing for w_k the complex roots of unity

$$w_k = \exp\left(\frac{2ik\pi}{2n}\right) = \omega^k, \quad \omega = \exp\left(\frac{2i\pi}{2n}\right)$$

This choice of the w_k permits to have two properties :

- The set of values $(P(w_0), \ldots, P(w_{2n-1}))$ and $(Q(w_0), \ldots, Q(w_{2n-1}))$ can be computed in time $O(n \log n)$.
- From the values $R(w_k)$ for $i = 0, \ldots, 2n-1$, the polynomial $R(z)$ can be obtained in time $O(n \log n)$.

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

The second point results from the fact that the k-th coefficient r_k of $R(z)$ satisfy

$$r_k = \frac{1}{2n} T\left(\bar{w_k}\right), \qquad T(z) = \sum_{j=0}^{2n-1} R(w_j) z^j,$$

which easily follows from a double summation inversion. In Fourier transform words, the coefficients of $R(z)$ are obtained from the conjugate of the Fourier transform of $T(z)$.

1.1 Fourier transform

The previous tasks all finally reduces in the following problem : Given a sequence $A = (a_0, a_1, \ldots, a_{2n-1})$, compute its Fourier transform

$$\mathcal{F}_{2n}(A) = (b_0, b_1, \ldots, b_{2n-1}), \qquad b_k = \sum_{j=0}^{2n-1} a_j \omega^{jk}. \qquad (*)$$

The final property to retrieve $R(z)$ from the Fourier transform of its coefficients $R(\omega^j)$ writes in the form

$$\bar{\mathcal{F}}_{2n}(\mathcal{F}_{2n}(R)) = 2nR,$$

where the conjugate Fourier transform is

$$\bar{\mathcal{F}}_{2n}(A) = (b_0, b_1, \ldots, b_{2n-1}), \qquad b_k = \sum_{j=0}^{2n-1} a_j \omega^{-jk}.$$

1.2 Fast Fourier Transform

The Fast Fourier Transform (FFT) is a way to compute the Fourier transform of a sequence A in time $O(n \log n)$ instead of $O(n^2)$ with the classical way, when n is a power of 2.

With the notation of (*), the principle is to write

$$b_k = \sum_{j=0}^{2n-1} a_j \omega^{jk} = \sum_{j=0}^{n-1} a_{2j} (\omega^2)^{jk} + \omega^k \sum_{j=0}^{n-1} a_{2j+1} (\omega^2)^{jk}.$$

In other words, to compute the coefficients b_k of $\mathcal{F}_{2n}(A)$, perform the following steps :

- Define two subsequences of size n

$$A_0 = (a_0, a_2, \ldots, a_{2n-2}), \quad \text{and} \quad A_1 = (a_1, a_3, \ldots, a_{2n-1}).$$

- Compute the Fourier transforms

$$C = \mathcal{F}_n(A_0) = (c_0, c_1, \ldots, c_{n-1}) \quad \text{and} \quad D = \mathcal{F}_n(A_1) = (d_0, d_1, \ldots, d_{n-1}).$$

- Deduce the Fourier transform $B = (b_0, \ldots, b_{2n-1}) = \mathcal{F}_{2n}(A)$ with the formulas

$$b_k = c_k + \omega^k d_k, \quad b_{n+k} = c_k - \omega^k d_k, \qquad 0 \le k < n.$$

Thus the cost $T(2n)$ of computing $\mathcal{F}_{2n}(A)$ with FFT satisfy $T(2n) = 2T(n) + O(n)$. When n is a power of two, the process can be made recursive leading to the bound $T(n) = O(n \log n)$. The algorithm is of course similar for the conjuguate Fourier transform.

2 Multiplication with FFT

2.1 Algorithm

We now present formaly the algorithm to multiply big numbers with FFT (the method is called *Strassen multiplication* when it is used with floating complex numbers) :

Let n be a power of two. Let two big integers X and Y with less than n coefficients

$$X = \sum_{j=0}^{n-1} x_j B^j, \quad Y = \sum_{j=0}^{n-1} y_j B^j.$$

To compute $Z = XY$ in time $O(n \log(n))$, perform the following steps :

- Thanks to the FFT process, compute the Fourier transform X^* of size $2n$ of the sequence (x_j) :

$$X^* = (x_0^*, x_1^*, \ldots, x_{2n-1}^*) \equiv \mathcal{F}_{2n}(x_0, x_1, \ldots, x_{n-1}, 0, \ldots, 0)$$

- Also compute the Fourier transform Y^* of (y_j) :

$$Y^* = (y_0^*, y_1^*, \ldots, y_{2n-1}^*) \equiv \mathcal{F}_{2n}(y_0, y_1, \ldots, y_{n-1}, 0, \ldots, 0).$$

- Compute the product term by term of X^* by Y^* in Z^*

$$Z^* = (z_0^*, z_1^*, \ldots, z_{2n-1}^*), \qquad z_i^* \equiv x_i^* y_i^*.$$

- Compute the inverse Fourier transform Z of Z^* (with the conjugate FFT process)

$$Z = (z_0, z_1, \ldots, z_{2n-1}) \equiv \frac{1}{2n} \bar{\mathcal{F}}_{2n}(Z^*).$$

- Then after rearrangement of the coefficients z_i, the number

$$Z = \sum_{i=0}^{2n-1} z_i B^i$$

  is equal to the product of X by Y.

Notice that X^* is the Fourier transform of the sequence made with n consecutive zeros appended to $x_0, \ldots, x_{n-1}$ (the same is true with Y^*).

The algorithm consists in computing two FFTs of size $2n$, $2n$ products of basic data types (whose complexity is negligible), and one reverse FFT of size $2n$. Thus the product of two large integers with n digits has a complexity asymptotically equal to 3 FFTs.

To square a number of size n, only one direct FFT is needed, thus the squaring complexity is asymptotically equal to 2 FFTs.

2.2 Numerical errors

In the practice, one should use the basic data type `double` in C to handle floating point numbers appearing in the Fourier transform. Numerical errors appear during the computation. If the numerical errors are sufficiently small, they can be overpassed : each final value z_i after the reverse FFT should be an integer, thus we must take its nearest integer value, the fractionnal part giving the error obtained.

The bound on the numerical errors α on the z_i after the FFT process can be proved to be

$$\alpha \leq 6n^2B^2\log(n)\epsilon \tag{1}$$

where $\epsilon \simeq 1.\mathrm{e}-16$ is the machine precision on the double precision words. But this bound is a worst case one and is very pessimistic. In the practice, the numerical stability of the FFT is very good and the bound $\alpha = O(nB^2\epsilon)$ is fulfilled. To deduce correctly the true value of z_i by taking its nearest integer, we need α to be sufficiently small (α less than 0.5 at least, $\alpha < 0.25$ is better !). Choosing a base B of three or four digits usually suffices to have the desired accuracy for multiplying two large integers.

Other approaches of FFT consists in working in a finite field Z/pZ with p prime, choosing ω as a primitive n-th root of unity in Z/pZ (this approach is called NTT, for Number Theoretic Transform). This approach overpass numerical errors. But is seems that, except on special architectures, actual implementation of NTT approach are less efficient.

2.3 More on the complexity of multiplication with FFT

In fact, the time complexity of multiplication with FFT is a little bigger than $n\log(n)$.

Let us be more precise. To multiply two numbers of N digits, we write them in a base B which contains k digits (say $B = 10^k$), thus giving a number of coefficients equal to $n \simeq N/k$. The discussion above tells us that to multiply those two numbers, FFT permits to perform $O(n\log(n))$ operations on basic numbers (basic numbers express coefficients in the base B, they are usually basic numerical data types like `double` in C of Fortran). Because of the numerical error bound (1), these basic numbers should be precise enough to represent integers up to $6n^2B^2\log(n)$ (for example, working in double precision, the base

B should be choosen small enough so that the error bound is not too large... and this is not even possible if n is too large). Thus the number of digits of these basic numbers should be of the order of $\log(B)+\log(n)$. As a consequence, the basic operations on these numbers has cost $O((\log(B)+\log(n))^2)$ and the final cost is $O(n\log(n)(\log(B)+\log(n))^2)$. The base B is choosen so that $k=\log_{10}(B)$ is of the same order than $\log(n)$, and finally, multiplying those two numbers of $N\simeq kn$ digits have cost

$$O(n\log(n)^3)=O(N\log(N)^2).$$

Thus the cost of *Strassen multiplication* (this is the name of the floating-FFT we presented) to multiply numbers of N digits is $O(N\log^2(N))$. If the Strassen mulplication is also used to compute the operations on the basic numbers (one recursive level), the cost reduces to $O(N\log(N)\log\log^2(N))$. The best bound is obtained with complete recursive version and is $O(N\log(N)\ \log\log(N)\ \log\log\log(N)\ldots)$. The use of the Number Theoretic Transform (see above) leads to the same bounds.

The lowest known theoritical bound is obtained with the *Schönhage multiplication*, which generalizes the process by working in finite rings, and is $O(N\log(N)\log\log(N))$ (see *Multiplication en le Lisp* for a description of Schönhage multiplication). The *Nussbaumer multiplication* also reaches this complexity (see [2] for an explanation of various large-multiplication routines). Notice that on actual implementations, the Strassen approach is the fastest, even if asymptotically slightly inferior.

3 Practical implementation

The basic practical implementation of floating-FFT is not so difficult. To be really efficient, one should take into account the hermitian property of the problem (x^*_{2n-k} is the conjugate of x^*_k) to avoid to compute twice the same terms and storing minimal information. This problem, together with a maximal efficiency on the data storage to avoid recopy, is more difficult. In this context, the use of the Hartley transform (variant of the FFT) can be helpful : Hartley transform is involutive, avoiding writing a specific code for the inverse transform.

The practical timings of FFT are often far from the theoritical one $O(n\log n)$. This is essentially due to a sparse access to the memory during the FFT. Techniques exist that optimizes this aspect to remain in the cache memory (4-step or 6-step FFT).

A very careful implementation of the FFT make it more efficient than any other multiplication techniques with number with ten thousands digits ore more.

A very easy FFT `C` sample code can be found in *Easy programs for constants computation*.

References

[1] D.E. Knuth, *The Art of Computer Programming, Vol. II, Seminumerical Algorithms,* Addison Wesley, (1998)

[2] Crandall R. and Pomerance C, *Prime Numbers: a Computational Perspective*, Springer (2000)