Fast Integer Multiplication

Sai Sandeep

February 13, 2020

1 Introduction

Suppose that we are multiplying two integers - what is the algorithm that we use? The traditional algorithm that we learn in "grade school" is the following -

	30 +	7
20 +	600	140
3	90	21

Table 1: Adding all the products, we get that that $23 \times 37 = 851$.

Even though it is excellent for pedagogical purposes, it has a problem - if the integers are large, then the algorithm is too slow. To multiply two n bit integers, the above algorithm requires $\Theta(n^2)$ operations. Can we do faster?

For a long time, it was believed that this might be tight. There was even a seminar organized in 1960 in Moscow to prove that integer multiplication (among other similar computational problems) requires $\Omega(n^2)$ operations. Surprisingly, during the seminar, Karatsuba came up with an algorithm that runs in $O(n^{\log_2^3}) \approx O(n^{1.58})$ time. The algorithm is based on a clever divide and conquer idea, similar to the Strassen algorithm for matrix multiplication. The next question is, can we do even faster?

It turns out that we can solve integer multiplication in $O(n \log n)$ time[1]. This is achieved using fast polynomial multiplication algorithm.

2 Fast Polynomial multiplication

Suppose that we are given two polynomials $A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1 x + \ldots + b_{n-1}x^{n-1}$. Can we find an efficient algorithm to compute the polynomial $P(x) = A(x)B(x)$? The naive algorithm that is similar to the grade school multiplication algorithm achieves $O(n^2)$ time complexity. Our goal is to find a faster algorithm.

Before delving into the algorithmic question, we first look into the problem of representing the polynomials. An obvious way to represent a polynomial is to write down all the coefficients. However, there is a different way to represent the polynomials: storing the evaluation of the polynomial at various points.

We first recall certain basic facts about polynomials:

Theorem 1. *(Fundamental Theorem of Algebra) A degree n polynomial with complex coefficients has exactly n complex roots.*

As a corollary, we get the following:

Corollary 1.1. *A degree n polynomial with coefficients from $\mathbb{C}$ is uniquely determined by its evaluation at $n+1$ points in $\mathbb{C}$.*

[1] $O(n \log n)$ complex arithmetic operations, to be precise.

Thus, we can also represent the polynomials $A(x)$ and $B(x)$ by their evaluations as $A(q_1), A(q_2), \ldots, A(q_n)$, $B(q_1), B(q_2), \ldots, B(q_n)$. Note that this representation has a clear advantage: it is very easy to multiply the polynomials in this representation, just multiply the evaluations at respective points! But, this also has a disadvantage: evaluating the polynomial at a different point is hard, if we just store the evaluations at n points. This gives rise to the following question - can we efficiently transfer between the two representations? This way, we can be in best of the two worlds: first, convert the two polynomials A and B to their evaluation representation, multiply them in the evaluation representation, and finally convert back to the coefficient representation. A priori, this doesn't seem to be getting us any savings as the naive method of evaluating a degree n polynomial at n points requires $\Theta(n^2)$ time.

There is one final missing piece: we can choose the n points that we evaluate the polynomial as we wish. It turns out that if we choose the evaluation points to be the nth roots of unity, we can indeed evaluate the polynomials fast!

A small technical detail: henceforth, we assume that n is a power of 2. This is okay because we can achieve this by adding zero coefficients by at most doubling the degree.

Theorem 2. *(Discrete Fourier Transform) Suppose that P is a degree $n-1$ polynomial. Then, we can evaluate P at all the nth roots of unity in total $O(n \log n)$ time.*

Furthermore, we can also invert this operation efficiently:

Theorem 3. *(Inverse Discrete Fourier Transform) Suppose that P is a degree $n-1$ polynomial. Given the evaluation of P at the nth roots of unity, we can compute all the coefficients of P in total $O(n \log n)$ time.*

We first recall certain basic properties of complex numbers:

Definition 4. *An nth root of unity is a complex number x such that $x^n = 1$.*

Fact 5. *There are n roots of unity, $1, \omega, \omega^2, \ldots, \omega^{n-1}$, where $\omega = e^{\frac{2\pi i}{n}}$.*

Proof. Note that Theorem 1 implies that there are exactly n complex numbers that are nth roots of unity. $(\omega^k)^n = e^{2\pi i k} = (e^{2\pi i})^k = 1^k = 1$. □

Fact 6. *The $\frac{n}{2}$nd roots of unity are $1, \zeta, \zeta^2, \ldots, \zeta^{\frac{n}{2}-1}$, where $\zeta = \omega^2$.*

2.1 DFT

Let $A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$ be the input polynomial. Our goal is to evaluate A at all the nth roots of unity. The idea is to use divide and conquer paradigm. First, we can break the polynomial A into A_{even} and A_{odd}, where $A_{even}(x) = a_0 + a_2 x + \ldots$, where $A_{odd}(x) = a_1 + a_3 x + \ldots$. We get $A(x) = A_{even}(x^2) + xA_{odd}(x^2)$. Thus, we can recursively compute A_{even} and A_{odd} at all the $\frac{n}{2}$nd roots of unity, and then use Fact 6 to evaluate the polynomial A.

Fast Fourier Transform (A, n)

1. Compute A_{even}, A_{odd}.
2. Compute Fast Fourier Transform $(A_{even}, \frac{n}{2})$, Fast Fourier Transform $(A_{odd}, \frac{n}{2})$.
3. For $0 \leq k < \frac{n}{2} : A(\omega^k) = A_{even}(\omega^{2k}) + \omega^k A_{odd}(\omega^{2k}) = A_{even}(\zeta^k) + \omega^k A_{odd}(\zeta^k)$.
4. For $\frac{n}{2} \leq k < n : A(\omega^k) = A_{even}(\omega^{2k}) + \omega^k A_{odd}(\omega^{2k}) = A_{even}(\zeta^{k-\frac{n}{2}}) + \omega^k A_{odd}(\zeta^{k-\frac{n}{2}})$.

The time complexity $T(n)$ is bounded as $T(n) = 2T(\frac{n}{2}) + O(n)$. By solving this recurrence, we obtain $T(n) = O(n \log n)$.

2.2 Inverse DFT

Our goal is to compute the unique polynomial A given the evaluations of A at $A(1), A(\omega), \ldots, A(\omega^{n-1})$. Note that we have the following matrix equation:

$$\begin{bmatrix} A(1) \\ A(\omega) \\ \vdots \\ A(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

where the $n \times n$ matrix is denoted as the Fourier matrix F_n. To compute A from the evaluation, we need to find the inverse of the matrix F_n.

In fact, the inverse of the matrix F_n is the following:

$$G_n = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \ldots & \omega^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \ldots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

Lemma 7. *The above defined matrix satisfies $G_n F_n = I_n$.*

Proof. The proof follows from the following observation:

$$\sum_{j=0}^{n-1} \omega^{kj} = \begin{cases} n, \text{ if } k \equiv 0 \bmod n \\ 0, \text{ otherwise} \end{cases}$$

The case when k is a multiple of n is trivial, as each summand on the left is equal to 1. When k is not a multiple of n, $x = \omega^k$ is an nth root of unity, and it satisfies $0 = x^n - 1 = (x-1)(1 + x + x^2 + \ldots + x^{n-1})$. As $x \neq 1$, $\sum_{j=0}^{n-1} x^j = 0$. □

Thus, in order to compute $G_n[A(1)A(\omega)\ldots A(\omega^{k-1})]$, we repeat the same algorithm as DFT, except with ω replaced by $\frac{1}{\omega}$.

2.3 Final Algorithm

We describe the algorithm to multiply the two polynomials A and B.

Multiplying the polynomials A, B.

1. Let $1, \omega, \ldots, \omega^{2n-1}$ be the $2n$th roots of unity.
2. Compute $A(1), A(\omega), \ldots, A(\omega^{2n-1})$ and $B(1), B(\omega), \ldots, B(\omega^{2n-1})$ using the DFT algorithm.
3. Multiply them pointwise to compute $AB(1), AB(\omega), \ldots, AB(\omega^{2n-1})$.
4. Using the inverse DFT algorithm, compute the polynomial AB.

The total running time of the above algorithm is $O(n \log n)$.

3 Fast Integer Multiplication

By using the fast polynomial multiplication algorithm, we can multiply two n bit integers using $O(n \log n)$ complex arithmetic operations, simply by representing the integers as polynomials with $x = 2$. Note that all the complex numbers involved can be bounded by polynomial in n. Thus, we can do any arithmetic operations of them in $\text{poly}(\log n)$ time. Thus, in time $O(n\text{poly}(\log n))$, we can multiply two integers. Schönhage and Strassen use these ideas together with a modification of FFT where they use a finite ring in stead of complex numbers, to achieve $O(n \log n \log\log n)$ time complexity. After a series of works, finally the time complexity $O(n \log n)$ was achieved last year by David Harvey and Joris van der Hoeven.