Fast Multiplication: Karatsuba and FFT

Haoyuan Sun

May 2016

0 Introduction

The old school way of multiplying polynomials is too slow because it requires $O(N^2)$ operations. There are several approaches to speed up polynomial multiplications, and these methods often more extensive applications than just manipulating polynomials.

Given the time constraint, I will not be able to go very deep into the subject. However, there are tons of resources out there if you are interested in learning more. You will get many good results from a simple search using Google, and the best beginner resource is chapter 30 from *Introduction to Algorithms.*

1 Karatsuba's Algorithm

Karatsuba is the first multiplication algorithm with better time complexity than long multiplication. It was originally designed for integer multiplication, but this is just a special case of polynomial multiplication when $x = 10$. But for the sake of clarity and simplicity, I will stick with integers.

1.1 Techniques

The first observation is that for large numbers, addition is many times faster than multiplications, so we can trade one multiplication operations for several addition operations and still save tons of time.

The other crucial observation is that we may divide the problems into smaller subsets and then combine the results. This is known as divide and conquer (remember merge sort?). We can try to split an integer into two smaller integers and work from there.

1.2 First Attempt

Say we have two integers x and y where they each have N digits. Then we can split them like this:

$$x = a \cdot 10^{N/2} + b$$
$$y = c \cdot 10^{N/2} + d$$

So we have:

$$x \times y = ac \cdot 10^N + (ad + cb) \cdot 10^{N/2} + bd$$

Wait darn, we need to make four multiplications for each recursive call: $T(N) = 4T(N/2) + O(N)$. By Master's Theorem, we still have $O(N^2)$ for our runtime complexity. But can we save some recursive calls?

1.3 Improvement

We realize that $(a+b)(c+d) = ac + (ad + cb) + bd$. So instead of brute force $ad + cb$, we can compute $(a + b)(c + d)$, which saves us one multiplication operations. With this optimization, our runtime complexity becomes $O(N^{\log_2 3}) \approx O(N^{1.584})$.

Here is a sample Python implementation of the algorithm:

```python
def karatsuba(x, y):
    if x < 1000 or y < 1000: return x*y

    # determine the location of the cut
    m = int(max(log10(x), log10(y)) + 1) // 2
    cut = 10 ** m

    # split the digit sequences about the middle
    a, b = x // cut, x % cut
    c, d = y // cut, y % cut

    # divide and conquer
    z0 = karatsuba(a, c)
    z1 = karatsuba(a + b, c + d)
    z2 = karatsuba(b, d)

    return z0 * cut**2 + (z1 - z0 - z2) * cut + z2
```

2 Cooly-Tukey FFT Algorithm

2.0 Points Value Representation

The other way of approaching polynomial multiplication is to interpolate the polynomial. Every polynomial of degree k can be uniquely determined by $k + 1$ points. Then polynomial multiplication becomes multiplying matching points values.

The coefficient to point value transformation:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

In order to transform back to the familiar coefficient form of polynomial, we can either take the inverse or use the Lagrange Formula (not very useful):

$$p(x) = \sum_{i=1}^{n-1} y_i \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

But both operations take $O(N^2)$ time, which is too slow. So we are going to use divide and conquer to make our life easier.

2.1 Divide and Conquer (Again)

In order to make such transformation faster, we will employ divide and conquer again. But slightly different this time.

To make the math simple, assume the polynomial has a degree that is a power of 2. If $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$, then let $p_{odd}(x) = a_1 + a_3 x + \cdots + a_{n-1} x^{n/2-1}$, and $p_{even}(x) = a_0 + a_2 x + \cdots + a_{n-2} x^{n/2-1}$.

Since $p(x) = p_{even}(x^2) + x\, p_{odd}(x^2)$, if we find $p(x)$, then we can get $p(-x)$ without additional effort. Ideally, the squares of the points we pick should also be in pairs of additive inverses. This reminds us of the roots of unity, so we pick the nth roots of unity as starting point and recur. We basically found a quick way of mapping coefficients to point values where the transformation matrix looks like this (ω is a nth primitive root of unity):

$$\begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

And it turns out that the inverse of this transformation matrix is super nice:

$$\frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{pmatrix}$$

2.2 Implementation

Now it is time for the actual implementation! We have a plan for the algorithm, but the implementation is still fairly difficult.

We can represent polynomial as an array of coefficient, from lower to higher degree. Notice the similarity of forward and reverse transform, a single function is enough to handle both (by feeding it with appropriate ω)

```
def fft(n, w, p):
    if n == 1: return p

    a = fft(n//2, w*w, p[0: :2]) # even powers
    b = fft(n//2, w*w, p[1: :2]) # odd powers

    ret = [0 for _ in range(n)]
    x = 1
    for k in range(n//2):
        ret[k] = a[k] + x * b[k]      # p(a)
        ret[k+n/2] = a[k] - x * b[k] # p(-a)
        x *= w

    return ret
```

We are almost ready to take the final step towards the implementation, but we should make few more observations. In order to get enough information to uniquely determine the final polynomial, we need n greater than the degree of the result. Also, since we are trying to divide into two sub-problems with same size, we need to make n be a power of 2 (we can always add trailing zero to make things work).

The final step, with forward FFT, multiply the point values, and then the reverse FFT.

```
def multiply(n, p1, p2):
    # nth principal root of unity
    w = complex(cos(2*pi/n), sin(2*pi/n))

    # forward transform to point value
    a = fft(n, w, p1)
    b = fft(n, w, p2)

    # convolution
    p3 = [a[i]*b[i] for i in range(n)]

    # reverse transform to get back coefficients
    final = fft(n, w**(-1), p3)
    return [x/n for x in final]
```

This gives us running time of $O(n \log n)$ YEAH :)

2.3 Improvements

There are further enhancements to this algorithm:

1. Use modular arithmetic instead of complex numbers. For example, 2 is a primitive 8th root of unity mod 17. This effectively removes round off error if we have integer coefficients, but we need to very careful with pick an appropriate mod.

2. We made the algorithm nice by forcing n be a power of 2. There are ways to remove this condition without much sacrifice in performance, but they are way beyond the scope of this lecture.

3 Crash Course in Fourier Series and DFT

A Fourier series is an expansion of some periodic and integrable function $f(x)$ in terms of infinite sum of trigonometric functions of the same harmonic.

Fourier series depends on the following identities:

$$\int_{x_0}^{x_0+2\pi} \sin(mx)\sin(nx)dx = \delta_{m,n}\pi$$
$$\int_{x_0}^{x_0+2\pi} \cos(mx)\cos(nx)dx = \delta_{m,n}\pi$$
$$\int_{x_0}^{x_0+2\pi} \sin(mx)\cos(nx)dx = 0$$
$$\int_{x_0}^{x_0+2\pi} \sin(mx)dx = 0$$
$$\int_{x_0}^{x_0+2\pi} \cos(mx)dx = 0$$

for nonzero integers m, n and any real x_0, where $\delta_{m,n}$ is the Kronecker delta.

So we have $\{1, \sin(mx), \cos(nx)\}$ as pairwise orthogonal basis of an inner product space, where the inner product is defined as

$$\langle f, g\rangle = \frac{1}{P}\int_{x_0}^{x_0+P} \overline{g(x)}f(x)dx$$

where P is the period of the fundamental frequency.

Furthermore, this can be generalized to any complete orthogonal system, which formed the basis of generalized Fourier series.

3.1 Fourier Series and Transform Defined

Let $f(x)$ be a periodic function with period P that is integrable. Then it can be approximated by the partial sum of Fourier series $s_N(x)$.

Definition 1 (Real Fourier Series)

$$s_N(x) = \frac{a_0}{2} + \sum_{n=1}^{N}\left(a_n\cos\frac{2\pi nx}{P} + b_n\sin\frac{2\pi nx}{P}\right)$$

With identities we have earlier, we can find the coefficients (known as *Fourier coefficients*):

$$a_n = \frac{2}{P}\int_{x_0}^{x_0+2\pi} s(x)\cos(nx)dx$$
$$b_n = \frac{2}{P}\int_{x_0}^{x_0+2\pi} s(x)\sin(nx)dx$$

Such results can be easily extended to the realm of complex numbers, which is more versatile in many applications.

Definition 2 (Complex Fourier Series)

$$s_N(x) = \sum_{n=-N}^{N} c_n \cdot e^{2i\pi nx/P}$$

We observe that $c_n = \frac{1}{P}\int_{x_0}^{x_0+2\pi} s(x)e^{-2i\pi nx/P}dx$

Fourier series is very important because it is a lot easier to analyze the well-known functions like the exponential than some random periodic function. In practice, in particular signal processing, we are often limited to sample the values of a functions at equally spaced points. Transform such list of equally spaced sample of a function to the Fourier coefficients of the function is called *discrete Fourier transform*. When we apply DFT, the function is transformed from the *time space* to *frequency space*.

Definition 3 (Discrete Fourier Transform)
Given a sequence of N complex numbers $\{x_0, x_1, \ldots, x_N\}$. Then their corresponding values in the frequency space are defined to be:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2i\pi kn/N}$$

3.2 Cooley-Tukey in Full Glory

The specific polynomial case we covered was done by sampling the polynomial at roots of unity. Now it is time to see Cooley-Tukey in its most general form:

First, let:

$$E_k = \sum_{m=0}^{N/2-1} x_{2n} \cdot e^{-4i\pi kn/N}, O_k = \sum_{m=0}^{N/2-1} x_{2n+1} \cdot e^{-4i\pi kn/N}$$

Then we have $X_k = E_k + e^{-2i\pi kn/N}O_k$.

Because of the periodicity of DFT, we conclude that for $0 \leq k < N/2$:

$$X_k = E_k + e^{-2i\pi kn/N}O_k$$
$$X_{k+N/2} = E_k - e^{-2i\pi kn/N}O_k$$

Thus we reduced the original DFT to two smaller DFT, each with only half of the size of the original. The runtime complexity would be $O(N \log N)$ if N is a power of 2.