How to Multiply Polynomials Faster

Haoyuan Sun

2015-05-22

"FFT does not stand for fast and furious turtles" – Remy Lee

1 Introduction

The old school way of multiplying polynomials is too slow because it requires $O(N^2)$ operations. But is there any algorithms better than the plain long multiplications? There are several approaches to greatly increase the speed of polynomial multiplications, and these methods often have far greater applications than just manipulate 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, 3rd Ed.*

2 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 $x = 1$. But for the sake of clarity and simplicity, I will stick with integers.

2.1 Techniques

The first observation is that for large numbers, addition is many times faster than multiplications[1], 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.

2.2 First Attempt

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

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

[1] the naive way to add is $O(N)$, but the fastest known multiplication algorithm is $O(N \log N \log \log N)$

So we have:

$$x \times y = ac \cdot 10^N \tag{1}$$
$$+ (ad + cb) \cdot 10^{N/2} \tag{2}$$
$$+ bd \tag{3}$$

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 this seems like an unfortunate coincidence, can we save some recursive calls?

2.3 Improvement

We realize that $(a+b)(c+d) = (1)+(2)+(3)$. So instead of brute force $ad+cb$, we can compute $(a+b)(c+d)$, which saves us one multiplication operations. By just this one optimization, our runtime complexity becomes $O(N^{\log 3}) \approx O(N^{1.584})$.[2]

Here is Python implementation of the algorithm:[3]

```
1  def karatsuba(x, y):
2      if x < 100 or y < 100: return x*y
3
4      # calculates the size of the numbers in base 10
5      m = max( len(str(x)) ,  len(str(y)) ) // 2
6
7      # split the digit sequences about the middle
8      cut = pow(10, m)
9      a, b = x // cut, x % cut
10     c, d = y // cut, y % cut
11
12     # divide and conquer
13     z0 = karatsuba(a, c)
14     z1 = karatsuba( (a + b), (c + d) )
15     z2 = karatsuba(b, d)
16
17     return z0 * pow(10, 2*m) + (z1 - z0 - z2) * pow(10, m) + z2
```

3 Cooly-Tukey FFT Algorithm

3.1 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.

[2] Here we use 2 as the base for all logarithms, which is the most natural one in computer algorithm

[3] Proof of concept of only, not production quality

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 command and conquer to make our life easier.

3.2 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 $p_{odd}(x) = p(x) = a_1 + a_3 x + \cdots + a_{n-1} x^{n/2-1}$, and $p_{even}(x) = p(x) = a_0 + a_2 x + \cdots + a_{n-2} x^{n/2-1}$.

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):

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

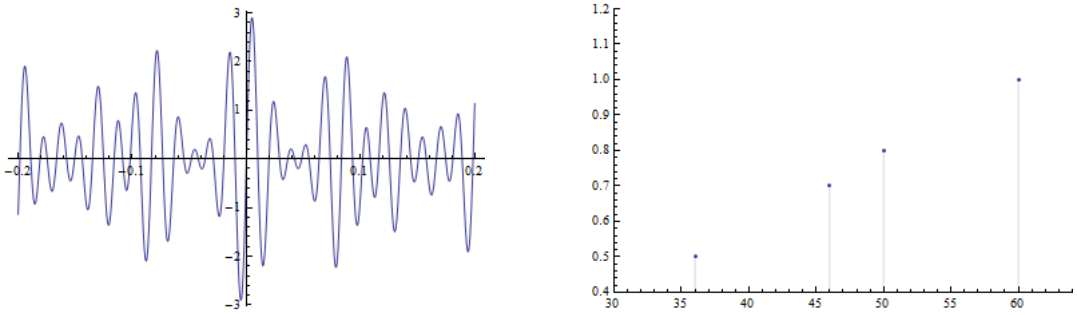
And it turns out that the inverse of this transformation matrix is super nice:

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

3.3 Fourier Transform

Simply speaking, a *Fourier transform* takes a function from time domain to frequency domain. A time domain is our familiar $x - y$ plane. Any smooth curve can be represented by a series of sinosoids, so the frequency domain is the plot of the frequency vs. amplitude of the series. The transformation matrix in the previous subsection is one type of Fourier transform called a *discrete Fourier transform (DFT)*. And *fast Fourier transform (FFT)* is the quick way of computing DFT.

Here is a simple graph that explains conceptually how Fourier transform works. In practice, we will deal with complex number instead of real number, and simply ignore the imaginary part if we only care about the real part.



(a) Time Domain (b) Frequency Domain

Figure 1: $f(x) = 0.8\sin(100\pi x) + 0.7\sin(92\pi x) + 1.0\sin(120\pi x) + 0.5\sin(72\pi x)$

3.4 Implementation

Now it is time for the actual implementation! We have a plan for the algorithm, but the implementation is very 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 feed it with appropriate ω)

```
1  def fft(n, w, p):
2
3      if n == 1: return p
4
5      a = fft(n//2, w*w, p[0::2]) # even powers
6      b = fft(n//2, w*w, p[1::2]) # odd powers
7
8      ret = [None]*n
9      x = 1
10     for k in range(n/2):
11         ret[k] = a[k] + x * b[k]      # p(a)
12         ret[k+n/2] = a[k] - x * b[k] # p(-a)
13         x *= w
14
15     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.

```
1  def multiply(n, p1, p2):
2      # nth principal root of unity
3      w = complex(cos(2*pi/n), sin(2*pi/n))
4
5      # forward transform to point value
6      a = fft(n, w, p1)
7      b = fft(n, w, p2)
8
9      # convolution
10     p3 = [a[i]*b[i] for i in range(n)]
11
12     # reverse transform to get back coefficients
13     final = fft(n, w**(-1), p3)
14     return [(x / n) for x in final]
```

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

3.5 Improvements

There can be 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 are have integer coefficients, but we need to make sure the mod is big enough.

2. There exists iterative implementation, consult chapter 30 from *Introduction to Algorithms*.

3. We made the algorithm nice by forcing n be a power of 2. There are ways to remove this sketchy constraint, but they are way beyond the scope of this lecture.