Polynomial Multiplication and Fast Fourier Transform

(Com S 477/577 Notes)

Yan-Bin Jia

Sep 17, 2020

In this lecture we will describe the famous algorithm of fast Fourier transform (FFT), which has revolutionized digital signal processing and in many ways changed our life. It was listed by the Science magazine as one of the ten greatest algorithms in the 20th century. Here we will learn FFT in the context of polynomial multiplication, and later on into the semester reveal its connection to Fourier transform.

Suppose we are given two polynomials:

$$\begin{aligned} p(x) &= a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}, \\ q(x) &= b_0 + b_1 x + \cdots + b_{n-1} x^{n-1}. \end{aligned}$$

Their product is defined by

$$p(x) \cdot q(x) = c_0 + c_1 x + \cdots c_{2n-2} x^{2n-2}$$

where

$$c_i = \sum_{\max\{0, i-(n-1)\} \le k \le \min\{i, n-1\}} a_k b_{i-k}.$$

In computing the product polynomial, every a_i is multiplied with every b_j, for $0 \le i, j \le n-1$. So there are at most n^2 multiplications, given that some of the coefficients may be zero. Obtaining every c_i involves one fewer additions than multiplications. So there are at most $n^2 - 2n + 1$ additions involved. In short, the number of arithmetic operations is $O(n^2)$. This is hardly efficient.

But can we obtain the product more efficiently? The answer is yes, by the use of a well-known method called *fast Fourier transform*, or simply, FFT.

1 Discrete Fourier Transform

Let us start with introducing the discrete Fourier transform (DFT) problem. Denote by ω_n an nth complex root of 1, that is, $\omega_n = e^{i\frac{2\pi}{n}}$, where $i^2 = -1$. DFT is the mapping between two vectors:

$$\boldsymbol{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \quad \longmapsto \quad \hat{\boldsymbol{a}} = \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_{n-1} \end{pmatrix}$$

such that

$$\hat{a}_j = \sum_{k=0}^{n-1} a_k \omega_n^{jk}, \qquad j = 0, \ldots, n-1.$$

It can also be written as a matrix equation:

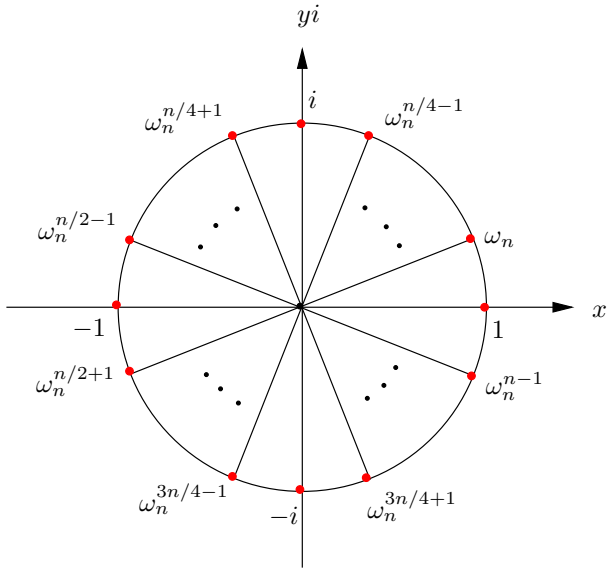
$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_{n-1} \end{pmatrix}.$$

The matrix above is a Vandermonde matrix and denoted by V_n.

Essentially, DFT evaluates the polynomial

$$p(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

at n points $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$; in other words, $\hat{a}_k = p(\omega_n^k)$ for $0 \le k \le n-1$. From now on we assume that n is a power of 2. If not, we can always add in higher order terms with zero coefficients $a_n = a_{n+1} = \cdots = a_{2^{\lceil \log_2 n \rceil} - 1} = 0$. The powers of ω_n are illustrated in the complex plane in the following figure.



The fast Fourier transform algorithm cleverly makes use of the following properties about ω_n:

$$\begin{aligned} \omega_n^n &= 1, \\ \omega_n^{n+k} &= \omega^k, \\ \omega_n^{\frac{n}{2}} &= -1, \\ \omega_n^{\frac{n}{2}+k} &= -\omega_n^k. \end{aligned}$$

It uses a divide-and-conquer strategy. More specifically, it divides $p(x)$ into two polynomials $p_0(x)$ and $p_1(x)$, both of degree $\frac{n}{2}-1$; namely,

$$\begin{aligned} p_0(x) &= a_0 + a_2 x + \cdots + a_{n-2} x^{\frac{n}{2}-1}, \\ p_1(x) &= a_1 + a_3 x + \cdots + a_{n-1} x^{\frac{n}{2}-1}. \end{aligned}$$

Hence

$$p(x) = p_0(x^2) + x p_1(x^2). \tag{1}$$

In this way the problem of evaluating $p(x)$ at $\omega_n^0, \ldots, \omega_n^{n-1}$ breaks down into two steps:

1. evaluating $p_0(x)$ and $p_1(x)$ at $(\omega_n^0)^2$, $(\omega_n^1)^2$, ..., $(\omega_n^{n-1})^2$,
2. combining the resulting according to (1).

Note that the list $(\omega_n^0)^2$, $(\omega_n^1)^2$, ..., $(\omega_n^{n-1})^2$ consists of only $\frac{n}{2}$ complex roots of unity, i.e., $\omega_n^0, \omega_n^2, \ldots, \omega_n^{n-2}$. So the subproblems of evaluating $p_0(x)$ and $p_1(x)$ have exactly the same form as the original problem of evaluating $p(x)$, only at half the size. This decomposition forms the basis for the recursive FFT algorithm presented below.

```
RECURSIVE-DFT(a, n)
1   if n = 1
2       then return a
3   ω_n ← e^{i 2π/n}
4   ω ← 1
5   a^[0] ← (a_0, a_2, ..., a_{n-2})
6   a^[1] ← (a_1, a_3, ..., a_{n-1})
7   â^[0] ← RECURSIVE-DFT(a^[0], n/2)
8   â^[1] ← RECURSIVE-DFT(a^[1], n/2)
9   for k = 0 to n/2 − 1 do
10      â_k ← â_k^[0] + ω â_k^[1]
11      â_{k+n/2} ← â_k^[0] − ω â_k^[1]
12      ω ← ω ω_n
13  return (â_0, â_1, ..., â_{n-1})
```

To verify the correctness, we here understand line 11 in the procedure RECURSIVE-DFT:

$$\hat{a}_{k+\frac{n}{2}} = \hat{a}_k^{[0]} - \omega \hat{a}_k^{[1]}.$$

At the kth iteration of the **for** loop of lines 9–12, $\omega = \omega_n^k$. We have

$$\begin{aligned} \hat{a}_{k+\frac{n}{2}} &= \hat{a}_k^{[0]} - \omega_n^k \hat{a}_k^{[1]} \\ &= \hat{a}_k^{[0]} + \omega_n^{k+\frac{n}{2}} \hat{a}_k^{[1]} \\ &= p_0\left(\omega_n^{2k}\right) + \omega_n^{k+\frac{n}{2}} p_1\left(\omega_n^{2k}\right) \\ &= p_0\left(\omega_n^{2k+n}\right) + \omega_n^{k+\frac{n}{2}} p_1\left(\omega_n^{2k+n}\right) \\ &= p\left(\omega_n^{k+\frac{n}{2}}\right), \qquad \text{from (1).} \end{aligned}$$

Let $T(n)$ be the running time of RECURSIVE-DFT. Steps 1–6 take time $\Theta(n)$. Steps 7 and 8 each takes time $T(\frac{n}{2})$. Steps 9–13 take time $\Theta(n)$. So we end up with the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n),$$

which has the solution

$$T(n) = \Theta(n \log_2 n).$$

2 Inverse DFT

Suppose we need to compute the inverse Fourier transform given by

$$\boldsymbol{a} = V_n^{-1}\hat{\boldsymbol{a}}.$$

Namely, we would like to determine the coefficients of the polynomial $p(x) = a_0 + \cdots + a_{n-1}x^{n-1}$ given its values at $\omega_n^0, \ldots, \omega_n^{n-1}$. Can we do it with the same efficiency, that is, in time $\Theta(n \log n)$? The answer is yes. To see why, note that the Vandermonde matrix V_n has inverse

$$V_n^{-1} = \frac{1}{n}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-(n-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \cdots & \omega_n^{-(n-1)^2} \end{pmatrix}$$

To verify the above, make use of the equation $\sum_{j=0}^{n-1}(\omega_n^k)^j = 0$ for non-negative integer k not divisible by n.

Based on the above observation, we can still apply RECURSIVE-DFT by replacing $\boldsymbol{a}$ with $\hat{\boldsymbol{a}}$, $\hat{\boldsymbol{a}}$ with $\boldsymbol{a}$, ω_n with ω_n^{-1} (that is, ω_n^{n-1}), and scaling the result by $\frac{1}{n}$.

3 Fast Multiplication of Two Polynomials

Let us now go back to the two polynomials at the beginning:

$$\begin{aligned} p(x) &= a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}, \\ q(x) &= b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}. \end{aligned}$$

Their product

$$(p \cdot q)(x) = p(x) \cdot q(x) = c_0 + c_1 x + \cdots c_{2n-2}x^{2n-2}$$

can be computed by combining FFT with interpolation. The computation takes time $\Theta(n \log n)$ and consists of the following three steps:

1. Evaluate $p(x)$ and $q(x)$ at $2n$ points $\omega_{2n}^0, \ldots, \omega_{2n}^{2n-1}$ using DFT. This step takes time $\Theta(n \log n)$.

2. Obtain the values of $p(x)q(x)$ at these $2n$ points through pointwise multiplication

$$\begin{aligned}(p \cdot q)(\omega_{2n}^0) &= p(\omega_{2n}^0) \cdot q(\omega_{2n}^0),\\ (p \cdot q)(\omega_{2n}^1) &= p(\omega_{2n}^1) \cdot q(\omega_{2n}^1),\\ &\vdots\\ (p \cdot q)(\omega_{2n}^{2n-1}) &= p(\omega_{2n}^{2n-1}) \cdot q(\omega_{2n}^{2n-1}).\end{aligned}$$

   This step takes time $\Theta(n)$.

3. Interpolate the polynomial $p \cdot q$ at the product values using inverse DFT to obtain coefficients $c_0, c_1, \ldots, c_{2n-2}$. This last step requires time $\Theta(n \log n)$.

We can also use FFT to compute the *convolution* of two vectors

$$a = (a_0, \ldots, a_{n-1}) \qquad \text{and} \qquad b = (b_0, \ldots, b_{n-1}),$$

which is defined as a vector $c = (c_0, \ldots, c_{n-1})$ where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}, \qquad j = 0, \ldots, n-1.$$

The running time is again $\Theta(n \log n)$.

4 History of FFT

Modern FFT is widely credited to the paper [2] by Cooley and Tukey. But the algorithm had been discovered independently by a few individuals in the past. Only the appearance of digital computers and the wide application of signal processing made people realize the importance of fast computation of large Fourier series. An incomplete list of pioneers includes

- Gauss (1805) — the earliest known origin of the FFT algorithm.
- Runge and König (1924) — the doubling algorithm.
- Danielson and Lanczos (1942) — divide-and-conquer on DFTs.
- Rudnick (1960s) — the first computer program implementation with $O(n \log n)$ time.

References

[1] T. H. Cormen *et al. Introduction to Algorithms*. McGraw-Hill, Inc., 2nd edition, 2001.

[2] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297-301, 1965.