Massachusetts Institute of Technology 6.046J/18.410J
Debayan Gupta, Aleksander Madry, and Bruce Tidor Handout 2

Recitation 2: Fast Fourier Transform

1 The Fast Fourier Transform

Recall the Fast Fourier Transform from lecture this week as an algorithm for multiplying two polynomials as follows. Let A and B be two degree bound n (degree $\leq n$) polynomials expressed as follows.

$$A(x) = a_0 + a_1 x + a_2 x^2 + .. + a_{n-1} x^{n-1} \tag{1}$$
$$B(x) = b_0 + b_1 x + b_2 x^2 + .. + b_{n-1} x^{n-1} \tag{2}$$

We call the length n vector $(a_0, a_1, .., a_{n-1})$ the coefficient vector representation for A and similarly, $(b_0, b_1, .., b_{n-1})$ is the coefficient vector representation for B. Equivalently, since any polynomial P is uniquely determined by its evaluations on $\geq degree(P) + 1$ points, A and B can also be represented by n points evaluated on the polynomial. We call this the point-value representation.

The product polynomial $C(x) = A(x) \cdot B(x)$ has degree $2 \cdot (n-1) < 2n$. Note that the first coefficient of $C(x)$ is $c_0 = a_0 b_0$, the second coefficient is $c_1 = a_0 b_1 + a_1 b_0$, ..., and the last coefficient is $c_{2n-2} = a_{n-1} b_{n-1}$. In general the coefficient c_k is given as follows.

$$c_k = \sum_{i=0}^{k} a_i b_{k-i} \tag{3}$$

Equation (3) is also known as the *convolution* of the coefficient vectors of A and B. Multiplying polynomials A and B, can now be expressed more mathematically as convolving their coefficient vectors. Note that trivially, it would take $O(n^2)$ time to compute this convolution. However, if A and B were evaluated at the same $2n$ points i.e. they were in point-value representation, multiplying them is easy. Simply do a point-wise multiplication in $O(n)$ time. Using this trick, as we saw in lecture, the Fast Fourier Transform allows us to compute this convolution in time $O(n \log n)$.

$$\begin{tikzcd}
A(x), B(x) \text{ coefficient representation} \arrow[r, "\text{Naively}", "O(n^2)"'] \arrow[d, Rightarrow, "\text{FFT}"', "O(n \log n)"] & C(x) \text{ coefficient representation} \\
A(x), B(x) \text{ point-value representation} \arrow[r, Rightarrow, "C(x) = A(x) \cdot B(x)", "O(n)"'] & C(x) \text{ point-value representation} \arrow[u, Rightarrow, "O(n \log n)", "\text{Inverse FFT}"']
\end{tikzcd}$$

Recall our algorithm for polynomial multiplication:

1. Evaluate polynomials $A(x)$ and $B(x)$ on the same $2n$ locations $\{x_0, x_1, .., x_{2n-1}\}$. ($O(n \log n)$ time using FFT)
2. Point-wise multiply the polynomials to obtain a point-value representation of $C(x) = A(x) \cdot B(x)$. ($O(n)$ time)
3. Compute the coefficient vector for C from the point-value representation. This is called *interpolation*. ($O(n \log n)$ time using Inverse FFT)

For step (1), we will choose the N-th roots of unity, where $N = 2^k$ is a power of 2 so that $2n <= N < 4n$ (Note that $N = O(n)$ and we could also have more than $2n$ points). The N-th roots of unity are given as $\{w^0, w^1, .., w^{N-1}\}$ where $w = e^{\frac{2\pi i}{N}}$. Note that $(w^k)^N = 1$ for $k \in \{0 .. N-1\}$.

Let y_k be value of A at $x_k = w^k$, we wish to evaluate A at all the roots of unity and find $y_0, y_1, .., y_{N-1}$ as follows:

$$y_k = \sum_{j=0}^{n-1} a_j e^{\frac{2\pi i k j}{N}} \tag{4}$$

Equation (4) is called the Discrete Fourier Transform (DFT) of A. The Fast Fourier Transform computes the DFT in $O(n \log n)$ time. The FFT to evaluate A at all the N-th roots of unity (also called DFT) is a divide and conquer algorithm given as follows:

1. Define

$$A_{even}(x) = a_0 + a_2 x + a_4 x^2 + .. + a_{n-1} x^{(n-1)/2} \tag{5}$$
$$A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + .. + a_{n-2} x^{(n-3)/2} \tag{6}$$

   Hence

$$A(x) = A_{even}(x^2) + x A_{odd}(x^2) \tag{7}$$

   Note that this is simply algebraic manipulation. Nothing special going on here. However, note that A_{even} and A_{odd} are degree bound $\frac{n}{2}$ polynomials.

2. Recursively evaluate $A_{even}(x)$ and $A_{odd}(x)$ on the *squares* of the N-th roots of unity.

$$\{(w^0)^2, (w^1)^2, .., (w_{N-1})^2\}$$

   But note that this set is the $\frac{N}{2}$-th roots of unity and has size $\frac{N}{2}$ (all the values in the second half of the set are a repeat of the first half, you can verify this easily). Hence, only $\frac{N}{2}$ evaluations need to be performed on $A_{even}(x)$ and $A_{odd}(x)$.

3. Evaluate $A(x)$ on all the N-th roots of unity using (7) and the recursively computed values for $A_{even}(x)$ and $A_{odd}(x)$.

We are now ready to compute the runtime for the FFT. Step (1) divides the problem into 2 problems of half the size (both in the degree of the polynomials and the number of evaluations). And it takes $O(N)$ time to perform the necessary additions in step (3). Hence, we have $T(N) = 2T(\frac{N}{2}) + O(N)$ which evaluates to $T(N) = \Theta(N \log N) = \Theta(n \log n)$ since $N = \Theta(n)$.

Hence, going back to our polynomial multiplication algorithm, we have performed step (1) in $O(n \log n)$ time. Step (2) is trivially $O(n)$ time to perform the point-wise multiplication. For step (3), we need to interpolate the evaluations of C to find the coefficient vector for C. The coefficient c_k can be obtained from the evaluations of C, y_t's, as follows:

$$c_k = \frac{1}{N} \sum_{t=0}^{N-1} y_t e^{\frac{-2\pi i t k}{N}} \tag{8}$$

This can easily be proven by plugging in the value of $y_t = \sum_{j=0}^{N-1} c_j e^{\frac{2\pi i j t}{N}}$ since we know the y_t's are the evaluations of C at the N-th roots of unity.

$$\begin{aligned}
c_k &= \frac{1}{N} \sum_{t=0}^{N-1} y_t e^{\frac{-2\pi i t k}{N}} && (9)\\
&= \frac{1}{N} \sum_{t=0}^{N-1} \left(\sum_{j=0}^{N-1} c_j e^{\frac{2\pi i j t}{N}} \right) e^{\frac{-2\pi i t k}{N}} && (10)\\
&= \frac{1}{N} \sum_{j=0}^{N-1} \left(c_j \sum_{t=0}^{N-1} e^{\frac{2\pi i t (j-k)}{N}} \right) && (11)\\
&= \frac{1}{N} c_k N = c_k && (12)
\end{aligned}$$

Note that in (11), $\sum_{t=0}^{N-1} e^{\frac{2\pi i t (j-k)}{N}} = N \cdot \delta_{j-k}$ is N if $j = k$ and 0 otherwise.

Hence, (8) is known as the Inverse DFT, which is the same as the DFT but with a scaling factor and a minus sign. This can also be used with the Fast Fourier Transform using as evaluation points $\{w^0, w^{-1}, \cdots, w^{-(N-1)}\}$ which is also a collapsing set of the roots of unity.

Hence, in step (3) of our polynomial multiplication algorithm, the interpolation takes $O(n \log n)$ to obtain the polynomial $C(x)$ in coefficient form using the Fast Fourier Transform. Hence overall, we can multiply the two polynomials in $O(n \log n)$ time.

1.1 Minkowski Sum using FFT

Let X and Y be length n sets of integers in the range $\{0, .., m-1\}$. The Minkowski Sum $X+Y$ is defined as the set:

$$X + Y = \{x + y \mid x \in X, y \in Y\} \tag{13}$$

We want to compute the size $|X+Y|$.

Naively, we can look through all n^2 pairs $(x, y)|x \in X, y \in Y$ and sum them up. We can then maintain a binary search tree of the values in order to avoid duplicate entries. Initialize a counter $i = 0$. Each time we insert a new value into the BST, increment i by 1. At the end of the process, i will be the size of $X+Y$. This process takes $O(n^2 \log n)$. If we use a different, perhaps randomized (Hash Table) method of avoiding duplicates, we could get a better runtime, but even this will take $\Omega(n^2)$ time.

We will now use the Fast Fourier Transform to solve this in $O(m \log m)$ time instead. The algorithm is as follows:

1. Construct a degree bound m polynomial $P_X(x)$ that has coefficient 1 for term x^k if $k \in X$, 0 otherwise.
2. Similarly, construct a degree bound m polynomial $P_Y(x)$ that has coefficient 1 for term x^k if $k \in Y$, 0 otherwise.
3. Multiply polynomials $P_{X+Y} = P_X(x) \cdot P_Y(x)$ using FFT.
4. Return the number of terms in P_{X+Y} that have a non-zero coefficient.

Note that by our construction, the coefficient of term $a_k x^k$ in P_{X+Y} is exactly the number of ways that numbers in X and Y can sum to k. If a_k is greater than 0, that means there exists values $x \in X$ and $y \in Y$ such that $x + y = k$. Hence, $|X+Y|$ is equal to the number of non-zero coefficients in P_{X+Y}.

Steps (1), (2) and (4) take $O(m)$ time and step (2) takes $O(m \log m)$ time. Hence, the overall cost of this algorithm is $O(m \log m)$. Note that if $m = O(n)$, this is $O(n \log n)$.

Note: One common mistake is to try to use FFT when the integers in X and Y are not bounded. If so, the integers in X and Y can get arbitrarily large, so the polynomials corresponding to X and Y will also be very large. Using FFT on those high degree polynomials could then take a lot of time.