CS 3510 — Algorithms, Spring 2022

Lecture 22: Fast Fourier Transform

April 21, 2022

Lecturer: Aditya Diwakar *Scribe: Aditya Diwakar*

Remark: The following content is unrelated to complexity theory. It is being covered as the Fast Fourier Transform is a ubiquitous concept, it will only show up on Exam 4 as an extra credit question.

1 Motivation

The lecture today will be a trip down memory lane as we will be covering something similar to integer multiplication, but with polynomials instead. What is the best way to multiply two polynomials together? Take the following example:

$$f(x) = x^3 + 3x^2 + 2x + 4 \qquad g(x) = x^2 + 2x$$

The *grade school* multiplication for polynomials is to multiply polynomials at each term. To compute $f(x) \cdot g(x)$, we would do:

$$\begin{aligned} f(x) \cdot g(x) &= (x^3 + 3x^2 + 2x + 4)(x^2 + 2x) \\ &= x^3(x^2 + 2x) + 3x^2(x^2 + 2x) + 2x(x^2 + 2x) + 4(x^2 + 2x) \\ &= x^5 + 2x^4 + 3x^4 + 6x^3 + 4x^2 + 4x^2 + 8x \\ &= x^5 + 5x^4 + 6x^3 + 8x^2 + 8x \end{aligned}$$

Let's analyze how much work we just did. We had to multiply the second polynomial by every term in the first polynomial. If f had n terms and g had m terms, then this takes $\mathcal{O}(nm)$ time. This is the runtime for our grade school algorithm. Let's formalize what an algorithm would look like.

First, we need to see how we could store the polynomials in a program. The clearest way is to store the coefficients of the polynomial. For example, if $f = 3x^2 + 2$ can be represented as $[2, 0, 3]$ where the ith index corresponds to the coefficient of x_i. We also know that if we multiply two polynomials of degree n and m, the

resultant polynomial would be of degree $m + n$.

Since we know the coefficients of x^i in f and g, then we can compute the coefficient for x^{i+j} in $h(x) = f(x)g(x)$ using this algorithm:

```
Function SlowPolyMult(F = [f_0, ..., f_n], G = [g_0, ..., b_m]):
    /* array for h = fg coefficients                    */
    H ← []
    for f_i ∈ F do
        for g_j ∈ G do
            H[i + j] ← f_i · g_j
    return H
```

As mentioned before, this has 2 nested for-loops giving this a runtime of $O(nm)$ (assuming the coefficients are constant bounded). Can we do any better?

1.1 Another Polynomial Representation

Before, we represented polynomials by using their coefficients. However, we can actually uniquely represent polynomials in another, more powerful way.

Theorem. *An n degree polynomial can be uniquely determined with $n + 1$ points.*

Proof. An n degree polynomial has $n + 1$ coefficients that need be determined. With $n + 1$ points, we can construct a system of linear equations:

$$\begin{cases} a_n x_1^n + a_{n-1} x_1^{n-1} + \ldots + a_1 x_1 + a_0 & = y_1 \\ & \vdots \\ a_n x_{n+1}^n + a_{n-1} x_{n+1}^{n-1} + \ldots + a_1 x_{n+1} + a_0 & = y_{n+1} \end{cases}$$

Whenever we have a linear system of equations, we can put it into a matrix:

$$\begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ \vdots & \vdots & & \vdots & \\ x_{n+1}^n & x_{n+1}^{n-1} & \cdots & x_{n+1} & 1 \end{bmatrix} \begin{bmatrix} a_n \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_{n+1} \end{bmatrix}$$

This matrix (known as *Vandermonde*) has a unique solution when $x_0, \ldots, x_{n+1}$ are unique, hence we have a unique solution for the polynomial coefficients values and

therefore $n+1$ points uniquely determines a polynomial of degree n.

Here is the big idea, if we have some representation of a polynomial using the coefficients, we have another representation of the polynomial using $n+1$ points.

For example, a degree 1 polynomial (line) is uniquely determined with 2 points. A degree 2 polynomials (parabola) is uniquely determined with 3 points, etc. How can we use this idea to multiply polynomials?

Again, the degree of $f(x)g(x) = h(x)$ is $n+m$ if f has degree n and g has degree m meaning we need $n+m+1$ points to determine it.

Take the following polynomials as an example:

$$f(x) = x^2 + 2x + 1 \quad g(x) = x^2 - 2x + 1$$

We know that $h(x)$ will have degree 4 meaning we need 5 points to uniquely determine it, we also know that $h(x_i) = f(x_i)g(x_i)$ so if we evaluate f and g at 5 points, we can get 5 values of h to determine it. Let's do that with $x = 0, \pm 1, \pm 2$.

$$\begin{aligned} f(x) = x^2 + 2x + 1 &\implies [(-2,1),(-1,0),(0,1),(1,4),(2,9)] \\ g(x) = x^2 - 2x + 1 &\implies [(-2,9),(-1,4),(0,1),(1,0),(2,1)] \end{aligned}$$

Now, we can evaluate $h(x)$ at $0, \pm 1, \pm 2$ without knowing the actual formula for h:

$$\begin{aligned} h(x) &= [(-2, 1 \cdot 9), (-1, 0 \cdot 4), (0, 1 \cdot 1), (1, 4 \cdot 0), (2, 9 \cdot 1)] \\ &= [(-2, 9), (-1, 0), (0, 1), (1, 0), (2, 9)] \end{aligned}$$

Great, so we did all this work, but what is the running time of this algorithm? We are only performing $n+m$ multiplications, so this only takes $\mathcal{O}(n+m)$, right?

No.

If we already evaluated the polynomials and have a bunch of points, we can do the multiplication fast, but if the input is a bunch of coefficients, how do we quickly evaluate points?

Each evaluation of a polynomial f takes $O(n)$ and we need to evaluate n *distinct* points, hence evaluation takes $O(n^2)$ time. We did all this work for nothing, right?

No.

2 The Evaluation Problem

As mentioned, it takes way too long to evaluate n distinct points on a polynomial, so we need a better way. It might help to start with an example such as $f(x) = x^2$. Let's say we are multiplying it with a polynomial that has degree 5. Then, we would need to get $2 + 5 + 1 = 8$ unique evaluations of f.

We *could* evaluate f at $x = \pm 1, \pm 2, \pm 3, \pm 4$ but that has the same problem, it takes $O(n^2)$ time. But $f(x) = x^2$ is symmetrical, so we could save time by only evaluating $x = 1, 2, 3, 4$ and we automatically get 8 points.

What about x^3? The same is true, we can say that $f(x) = -f(-x)$ and only need to evaluate 4 points to get 8 total points.

Let's generalize the above example! Take the example $f(x) = 3x^5 + 2x^4 + x^3 + 7x^2 + 5x + 1$, we can split this up into even and odd degree terms like so:

$$\begin{aligned} f(x) &= (2x^4 + 7x^2 + 1) + (3x^5 + x^3 + 5x) \\ &= (2x^4 + 7x^2 + 1) + x(3x^4 + x^2 + 5x) \\ &= f_{even}(x^2) + x f_{odd}(x^2) \end{aligned}$$

We know that an even degree polynomial will be symmetric and an odd degree polynomial is simply flipped, so we can say that $f(x_i) = f_{even}(x^2) + x f_{odd}(x^2)$ and $f(-x_i) = f_{even}(x^2) - x f_{odd}(x^2)$. Notice however, this means we only actually have to compute $n/2$ points (since points are reused). Hence, we have reduced the number of terms to compute from $n \to n/2$.

The question is then, how do we compute $f_{even}(x^2)$ and $f_{odd}(x^2)$ efficiently? Thankfully, we are making an algorithm to do just that, so we can recursively solve!

However, we run into a slight challenge. The inputs to f_{even} and f_{odd} are squared so we cannot take advantage of the symmetry. What can we do to fix this?

3 The Symmetry Problem

We've seen that we cannot recursively solve this problem with just positive and negative reals as that fails on the first level of recursion. What other numbers do we know that have more positive/negative symmetry? ℂan you guess?

The complex numbers have a lot of natural symmetry in something we will later discuss. Remember! The intention was that x_1, x_2, x_3, x_4 should be paired in $\pm$ pairs such that we can get away with only computing $n/2$ values instead of n.

So instead of x_1, x_2, x_3, x_4, we instead have $\pm x_1, \pm x_2$. Now, when we recursively make a call, we want x_1^2, x_2^2 to also be positive/negative pairs. Hence, we want $x_2^2 = -x_1^2$. How is it possible that x_2^2 is a negative number? If we let $x_1 = 1$, then $-x_1 = -1$ which means that $x_2^2 = -1 \implies x_2 = i$ and $-x_2 = -i$.

Hence, we have to be very specific about the values we choose to evaluate at. We have to pick what are known as the roots of unity. Specifically, if we need k points, we need k roots of unity (and more precisely, we tend to pick the nearest power of 2 above k). For example, if we want $k = 6$, then we can pick the 8th roots of unity.

3.1 Roots of Unity

The roots of unity are what they sound like. They are the roots of the formula $z^k = 1$. For example. if $k = 2$, then $z^2 = 1 \implies z = \pm 1$. If $z = 4$, then $z^4 = 1 \implies z = \pm 1, \pm i$. It just so happens that all the roots of unity lay on the complex unit circle and can be written in the form: $\omega = e^{2\pi i/n}$

This works because w^j and $w^{j+n/2}$ are always going to be positive/ negative paired. As a result, we can implement an algorithm that reduces the number of evaluation points needed by $1/2$ at each recursion!

4 FFT Implementation

The below is an implementation of the FFT algorithm:

```
1  Function FFT(F = [f_0, ..., f_n]):
       /* base case for when polynomial degree is 0                    */
2      if |F| = 1 then
3          return F
       /* get the nearest power of 2 above degree of F                 */
4      n ← NearestPow2(|F|)
5      ω ← exp(2πi/n)
6      f_even ← FFT(F[:: 2])
7      f_odd ← FFT(F[1 :: 2])
       /* return array                                                 */
8      f_total ← []
9      for j ∈ 0 → n/2 do
10         f_total[j] = f_even[j] + ω^j f_odd[j]
11         f_total[j + n/2] = f_even[j] − ω^j f_odd[j]
12     return f_total
```

In the above, we make recursive calls on the even and odd degrees reducing the degree in half. The rest of the work is linear, hence we get a recurrence relation given by $T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n)$ which is the runtime for FFT.

5 Interpolation

Great! We've done half the task. Our original fast polynomial multiplication needed to convert coefficients to values, multiply these values, then convert those computed values to the coefficients of the resultant polynomial. How do we do this last bit?

This is what is known as interpolation. Remember from before, we could write

the polynomial as a system of linear equations like so:

$$\begin{cases} a_n x_1^n + a_{n-1} x_1^{n-1} + \ldots + a_1 x_1 + a_0 & = f(x_1) \\ & \vdots \\ a_n x_{n+1}^n + a_{n-1} x_{n+1}^{n-1} + \ldots + a_1 x_{n+1} + a_0 & = f(x_{n+1}) \end{cases}$$

Whenever we have a linear system of equations, we can put it into a matrix:

$$\begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ \vdots & \vdots & & \vdots & \\ x_{n+1}^n & x_{n+1}^{n-1} & \cdots & x_{n+1} & 1 \end{bmatrix} \begin{bmatrix} a_n \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_{n+1}) \end{bmatrix}$$

This above form relies on individual x_k values, but through our FFT journey, we now have specific values for these x_k values. We know that $x_k = (\exp(2\pi i/n))^{k-1} = \omega^{k-1}$. Hence, we can rewrite the above in the following form:

$$\begin{bmatrix} 1 & \cdots & 1 & 1 & 1 \\ \omega^{n-1} & \cdots & \omega^2 & \omega & 1 \\ \omega^{2(n-1)} & \cdots & \omega^4 & \omega^2 & 1 \\ \vdots & \iddots & \vdots & \vdots & \vdots \\ w^{n(n-1)} & \cdots & w^{2(n)} & w^n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} f(\omega^0) \\ f(\omega) \\ \vdots \\ f(\omega^{n-1} \\ f(\omega^n) \end{bmatrix}$$

This matrix is known as the DFT (Discrete Fourier Transform) matrix. One of the beautiful things is that this matrix has a nonzero determinant and therefore is invertible. This means we can simply invert this matrix to solve for the coefficients $a_0, \ldots, a_n$. Remember, if we have $Ax = b$, we can solve x by doing $x = A^{-1}b$.

$$\begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix} = \left(\begin{bmatrix} 1 & \cdots & 1 & 1 & 1 \\ \omega^{n-1} & \cdots & \omega^2 & \omega & 1 \\ \omega^{2(n-1)} & \cdots & \omega^4 & \omega^2 & 1 \\ \vdots & \iddots & \vdots & \vdots & \vdots \\ w^{n(n-1)} & \cdots & w^{2(n)} & w^n & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} f(\omega^0) \\ f(\omega) \\ \vdots \\ f(\omega^{n-1} \\ f(\omega^n) \end{bmatrix}$$

Now, we are curious about the inverse of this matrix. The inverse of this matrix is beautifully given by the following:

$$\left(\begin{bmatrix} 1 & \cdots & 1 & 1 & 1 \\ \omega^{n-1} & \cdots & \omega^2 & \omega & 1 \\ \omega^{2(n-1)} & \cdots & \omega^4 & \omega^2 & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ w^{n(n-1)} & \cdots & w^{2(n)} & w^n & 1 \end{bmatrix}\right)^{-1} = \frac{1}{n}\begin{bmatrix} 1 & \cdots & 1 & 1 & 1 \\ \omega^{-(n-1)} & \cdots & \omega^{-2} & \omega^{-1} & 1 \\ \omega^{-2(n-1)} & \cdots & \omega^{-4} & \omega^{-2} & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ w^{-n(n-1)} & \cdots & w^{-2(n)} & w^{-n} & 1 \end{bmatrix}$$

Notice here the inverse is quite literally simply the reciprocal of the matrix entry for the DFT matrix with a normalization factor of $1/n$. Hence, the interpolation or inverse of the FFT (to go from values to coefficients, rather than values to coefficients) can be implemented the same way but by changing out these matrix values).

In summary, the coefficients are given in this way:

$$\begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix} = \frac{1}{n}\begin{bmatrix} 1 & \cdots & 1 & 1 & 1 \\ \omega^{-(n-1)} & \cdots & \omega^{-2} & \omega^{-1} & 1 \\ \omega^{-2(n-1)} & \cdots & \omega^{-4} & \omega^{-2} & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ w^{-n(n-1)} & \cdots & w^{-2(n)} & w^{-n} & 1 \end{bmatrix}\begin{bmatrix} f(\omega^0) \\ f(\omega) \\ \vdots \\ f(\omega^{n-1} \\ f(\omega^n) \end{bmatrix}$$

The only difference between computing values and computing coefficients were the matrix values. Specifically, the definition for ω changed.

$$\text{FFT: } \omega = \exp\left(\frac{2\pi i}{n}\right) \rightarrow \text{IFFT: } \omega = \frac{1}{n}\exp\left(\frac{-2\pi i}{n}\right)$$

Finally, the inverse FFT algorithm can be implemented as such:

```
1  Function IFFT(F = [f(ω^0), ..., f(ω^{n-1})]):
       /* base case for when polynomial degree is 0                    */
2      if |F| = 1 then
3          return F
       /* get the nearest power of 2 above degree of F                 */
4      n ← NearestPow2(|F|)
5      ω ← (1/n) exp(−2πi/n)
6      f_even ← IFFT(F[:: 2])
7      f_odd ← IFFT(F[1 :: 2])
       /* return array                                                 */
8      f_total ← []
9      for j ∈ 0 → n/2 do
10         f_total[j] = f_even[j] + ω^j f_odd[j]
11         f_total[j + n/2] = f_even[j] − ω^j f_odd[j]
12     return f_total
```

The runtime of this algorithm is the same runtime as FFT since the only difference is the value ω. Hence, this runs in $\mathcal{O}(n \log n)$ time.

6 Summary

Hence, let us revisit polynomial multiplication. We can use the FFT to convert from the coefficient representations to value representations in $\mathcal{O}(n \log n)$ time. Using these values, we can compute the value representation of the multiplied polynomial in $O(n)$ time (assuming n is the larger of the two degrees), and finally take this value product representation and use the IFFT to get to the coefficients in polynomial time. Therefore, we now have polynomial multiplication in $\mathcal{O}(n \log n)$.

7 Challenge

Using this fact, design an algorithm that can multiply integers in $\mathcal{O}(n \log n)$ time. Hint: use the FFT and represent your number with polynomials.