CS 3510 Algorithms 1/23/2023

Lecture 5: Fast Fourier Transform

Lecturer: Abrahim Ladha *Scribe(s): Abrahim Ladha*

1 Operations in Coefficient Form

Lets consider arithmetic on polynomials. We may represent a polynomial as

$$A(x) = a_0 + a_1 x + ... + a_{n-1}x^{n-1}$$

Algorithm wise, we have an array of of bounded coefficients, stored as $[a_0, ..., a_{n-1}]$. We are concerned with the run time of our algorithms as a function of the number of coefficients, the degree.

Three elementary operations we can concern ourselves with are addition, multiplication, and evaluation. Notice first that addition and multiplication of two polynomial results in a polynomial, and evaluation of a polynomial results in a value.

1. Addition of two polynomials can be done by adding the terms.

$$A(x) + B(x) = \sum_{j=0}^{n-1} a_j x^j + \sum_{j=0}^{n-1} b_j x^j = \sum_{j=0}^{n-1} (a_j + b_j) x^j$$

   We essentially add two arrays together pairwise. Each operation takes constant time, and we have a linear number of them to do, so the run time is $O(n)$.

2. Multiplication of two polynomials is a little trickier. Recall the FOIL method. Where

$$(a + b)(c + d) = ac + ad + bc + bd$$

   Generalizing this, we consider worst case one multiplication for every pair of coefficients. This gives us naively a run-time of $O(n^2)$.

3. Evaluation of a polynomial is to substitute in some constant for every x. There are some naive ways to do this, but using Horner's method[1] allows us to do this in $O(n)$ time. Essentially, we represent our polynomial in a nested format, like

$$1 + x + x^2 + x^3 = 1 + x(1 + x(1 + x))$$

   . This evaluation occurs inside out.

[1]Horner's method

2 Operations in Sample Form

The set of coefficients is not the only way to represent a polynomial. We may also represent it as a set of "samples" or "values". Recall that two points uniquely define a straight line. If you have one point, there are infinitely many possible lines which may intersect that point. However, if you have a second point, the set of possibilities reduces just to one line. This generalizes, so that three points uniquely define a quadratic, and $n+1$ points unique define a degree n polynomial.

Lets consider polynomial arithmetic under this representation. Instead of inputting and returning an array of coefficients, lets input and return our polynomial as a set of samples. Instead of $[a_0, ..., a_{n-1}]$, we have $[A(x_0), ..., A(x_{n-1})]$ for some chosen values of $x_0, ..., x_{n-1}$ with distinct[2] x values.

1. Adding two polynomials represented as samples is as easy as pairwise adding the samples if they are sampled in the same place. Suppose we have $A(x)+B(x) = C(x)$. Then certainly it is true that $A(x_0) + B(x_0) = C(x_0)$. The sum of those two samples will give us a sample of the polynomial of C. We can perform this n times to get n samples of C, which uniquely defines it. We get to return an array of just the samples, so we only have to do a linear amount of additions, and this can be done in $O(n)$ time.

2. Multiplication is done by a similar trick, where we pairwise product the samples. It holds that $A(x_0)B(x_0) = C(x_0)$. But wait, C will have maximum degree equal to the sum of the max degrees of A, B[3]. We would have n points, which don't uniquely define C, a polynomial of degree $(n-1)+(n-1) = 2n-2$. We need to make sure we initially oversample A, B with $x_0, ..., x_{2n-1}$. We still have a linear amount of multiplication of samples, giving us a run-time of $O(n)$

3. If we want to sample a polynomial at a point its already evaluated at, then great! If not, this is non-trivial, so lets not get too deep into it. Just know its like, $O(n^2)$.

3 Big Picture

We want to show polynomial multiplication can be done in $O(n \log n)$ time from coefficients. We can't naively multiply them together in $O(n^2)$ time. We are going to transform our two input polynomials from coefficient representation to samples, then we will do $O(n)$ time sample multiplication, and then transform back from samples to coefficients. For now, lets just consider the problem of first getting samples, and then performing sample multiplication. We will worry about interpolation, going from samples to coefficients, later.

The naive way to transform from coefficients to samples is to just perform a linear amount of evaluations on some chosen $x_0, ..., x_{2n-1}$. Each evaluation takes linear time, and we have a linear amount of them. So we would have $O(n^2)$ time. This is not better than the method of coefficient multiplication. The main idea behind FFT is that if we choose

[2]If two points did have the same x value, then they would be above and below each other vertically in the cartesian plane, and polynomial intersecting both would not be a function

[3]if we have $C = (x^n + ...)(x^m + ...)$, then the leading term of C will be x^{n+m}

$x_0, ...x_{2n}$ in a specific way, their information overlaps quite a bit, and getting our samples won't take $O(n^2)$ but $O(n \log n)$.

4 First try of Divide and Conquer

Notice that for any polynomial, we can represent it as two smaller polynomials of even and odd terms. If we have

$$A(x) = 1 + x + x^2 + x^3 + x^4 + x^5$$

We can separate as

$$A(x) = (1 + x^2 + x^4) + (x + x^3 + x^5)$$

If we pull out an x from the odd terms, we get

$$A(x) = (1 + x^2 + x^4) = x(1 + x^2 + x^4)$$

We have now broken up a polynomial into two smaller polynomials of even degree. We can "unevaluate" these two subproblems for x^2 as

$$A(x) = (1 + (x^2)^1 + (x^2)^2) + x(1 + (x^2)^1 + (x^2)^2)$$

and naming our subproblems, we get

$$A(x) = A_e(x^2) + xA_o(x^2)$$

for

$$A_e(x) = (1 + x + x^2)$$

and

$$A_o(x) = (1 + x + x^2)$$

.

This is useful to us in a divide and conquer setting, if we choose our points to be positive and negative pairs. Suppose we chose $x_0, ..., x_{2n-1} = 1, -1, 2, -2, ...$ Then we would get that

$$A(x) = A_e(x^2) + xA_o(x^2)$$

and

$$A(-x) = A_e(x^2) - xA_o(x^2)$$

Then instead of four recursive calls for two samples, we have two recursive calls for two samples by reusing $A_e(x^2)$ and $A_o(x^2)$. We have a new problem. This only works for the top level recursion, as for the next level, there won't be any positive and negative pairs, only positive numbers. We need somehow to have a square to be negative. The trick is to exit the real numbers, and cast ourselves into the complex plane.

5 Complex Numbers

We define i to be a special made up[4] number such that $i^2 = -1$. We then define

$$\mathbb{C} = \{a + bi \mid a, b \in \mathbb{R}\}$$

As a warm up, convince yourself that the sum and product of two complex numbers is complex.

These complex numbers have two parts, so it may be convient to represent them as vectors in a two dimensional plane. Take the number $a+bi$ and it is equivalently represented by the vector (a, b). If we switch to polar coordinates, this complex number is representable by a length of the vector, and the angle counter-clockwise from the origin, (r, θ). For what we need, lets only study complex numbers whose vectors have length one. We can then represent these numbers as just an angle. Euler's formula gives us a way to go from the angle to the components as

$$e^{i\theta} = \cos\theta + i\sin\theta$$

Geometrically it makes sense. $(\cos\theta, \sin\theta)$ are the x, y components respectfully. We can prove its equivalent to the left hand side using a Taylor series expansion.

$$\sum_{k=0}^{\infty} \frac{(i\theta)^k}{k!} = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k)!}\theta^{2k} + i\sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!}\theta^{2k+1}$$

As i goes through successive powers on the left hand side like $..., i^2, i^3, i^4, ...$, it will alternate between real and imaginary. If we group the real terms, we get a Taylor series expansion of $\cos\theta$. If we group the imaginary terms and undistribute an i, then we get a Taylor series expansion of $\sin\theta$. Do not think of $e^{i\theta}$ as a number, algebraically. This is not something you do in a calculator. Just think of it as a funny way we represent a complex number on the unit circle in the complex plane.

We define the roots of unity to be the complex roots of $z^n - 1 = 0$ for n a power of two. We can denote these as $\omega^0, \omega^1, ..., \omega^{n-1}$. If you plot these on our complex plane, you will get equidistant points on the unit circle! Here, unity just means one. For example, lets compute the roots of unity of $z^4 = 1$. One way to think about it is what four[5] complex numbers can we raise to the fourth power to equal one. Another way is to think of four equidistant points on the unit circle, each 90 degrees from each other on the axii. The first root of unity will always be one, $\omega^0 = 1$. The second root should be 90 degrees counter-clockwise from it, and so on. We find these to be $\omega^0, \omega, \omega^2, \omega^3 = 1, i, -1, -i$. We have a general form of the roots of unity as

$$\omega^0, \omega, \omega^2, ..., \omega^{n-1} = 1, e^{2\pi i/n}, e^{4\pi i/n}, ..., e^{2(n-1)\pi i/n}$$

They have some interesting mathematical properties we won't get into. What we need them for is as points to sample! They have two desirable properties. First is that

$$-\omega^j = \omega^{n/2+j}$$

[4]all numbers are made up

[5]By the fundamental theorem of arithmetic, every polynomial has as many roots over $\mathbb{C}$ as its degree.

. Second is that

$$\omega^2 \text{ is a root of unity for } \frac{n}{2}$$

What this means is that if we choose to sample at roots of unity, then they behave like $-1, 1, -2, 2, ...$ pairs all the way down the recurrence! We are now ready to present FFT.

6 The Fast Fourier Transform

Algorithm 1 FFT

```
Input: $(A, \omega)$ where $A$ is a polynomial of degree $n-1$ as a list of coefficients and $\omega$ is a
    root of unity of $n$ where $n$ is a power of two
Output: The samples $A(w^0), ..., A(w^{n-1})$
1: if $\omega = 1$ return $A(1)$
2: compute $A_e, A_o$ such that $A(x) = A_e(x^2) + xA_o(x^2)$
3: $[A_e(\omega^0), A_e(\omega^2), ...] \leftarrow \texttt{FFT}(A_e, \omega^2)$
4: $[A_o(\omega^0), A_o(\omega^2), ...] \leftarrow \texttt{FFT}(A_o, \omega^2)$
5: for $j = 0, ..., n-1$ do
6:     $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$
7: end for
8: return $[A(\omega^0), ..., A(\omega^{n-1})]$
```

Note that when we want samples, it doesn't particular matter what points we sample at. But if we want to perform this fast algorithm, why not choose then to sample at the roots of unity. We have two recursive calls at each level. A_e, A_o have maximum degree half the original problem. We do a linear amount of work to split and recombine A into A_e, A_o, so this work is linear. This gives us our recurrence as

$$T(n) = 2T(n/2) + O(n)$$

You could plug this into the Master theorem to get $O(n \log n)$, or you could recognize its the same as the mergesort recurrence.

7 Multiplying Polynomials

Recall our original motivation was multiplying polynomials. What a "Fourier transform" even is, is unimportant to us. It has unlimited applications in signal processing and so on, but we are concerned with using it to quickly convert a polynomial from coefficients to samples. Recall that polynomial multiplication of two polynomials in coefficient form takes $O(n^2)$ time. Our fast polynomial multiplication would then be as follows

1. Convert coefficients of $A(x)$ to samples $[A(\omega^0), ..., A(\omega^{n-1})]$ with FFT
2. Convert coefficients of $B(x)$ to samples $[B(\omega^0), ..., B(\omega^{n-1})]$ with FFT
3. Compute $[A(\omega^0)B(\omega^0), ..., A(\omega^{n-1})B(\omega^{n-1})]$. Note that these are samples of $A(x)B(x)$

4. Interpolate samples back to coefficients for $A(x)B(x)$

Steps 1, 2 each take $O(n \log n)$ time. Step 3 takes $O(n)$ time. I haven't given any justification yet on how we can do interpolation. Previously, we have used FFT to compute samples as

$$[\texttt{samples}] = \texttt{FFT}([\texttt{coeffs}], \omega)$$

The final trick is that its true we can also use FFT to reverse this process as

$$[\texttt{coeffs}] = \frac{1}{n}\texttt{FFT}([\texttt{samples}], \omega^{-1})$$

This gives our final run time of polynomial multiplication to be

$$O(n \log n) + O(n \log n) + O(n) + O(n \log n) = O(n \log n)$$

8 A quick note on the Inversion

I won't prove why this inversion is true. You have to reformulate using linear algebra. The FFT operation can be redefined as multiplication by a special matrix defined in terms of ω, we can denote as V_ω. Inversion is then multiplying by the inverse of this matrix, and it will turn out that

$$nV_\omega^{-1} = V_{\omega^{-1}}$$

. You would need to prove this matrix is invertible, and its multiplication is $O(n \log n)$ because its special, and so on. I will refer you to the book.

9 Further Reading (Watching)

FFT is one of the few things which enters popular science enough for attempted Layman explanations. These may be hit or miss, but have nice graphics.

- The Remarkable Story Behind The Most Important Algorithm Of All Time
- But what is the Fourier Transform? A visual introduction.

There also exist previous recorded lectures on this topic

- Jake Abernathy at GT
- Eric Vigoda previous at GT
- Erik Demaine at MIT