FAST FOURIER TRANSFORM

GLENN SUN
OLGA RADKO MATH CIRCLE ADVANCED 3

OCTOBER 24, 2021

Today we will study the fast Fourier transform, an extremely useful tool in computer science and engineering. It allows you to magically do things faster than seems possible.

1 Asymptotic Growth Rate

If you were part of Advanced 3 last year, you might recall that we studied big-O notation to talk about how quickly functions grow. In this section, we'll review those concepts for those who are seeing them for the first time.

Problem 1. Use Desmos or a graphing calculator to assist you in the following questions. You may need to adjust the scale of the axes to see everything. Although you may not need a graph to answer some questions, the graphs should help you feel the ideas.

1. Graph $f(n) = n$ and $g(n) = 10n$. Then graph $f(n) = n^2$ and $g(n) = 10n$. For large values of n, which function is larger? Is it somewhat larger or a lot larger?
2. Graph $f(n) = n^a$ and $g(n) = \log(n)$ for various values of $a \in (0, \infty)$. For (very) large values of n, which function is larger? Is it somewhat larger or a lot larger?

To formally capture the idea that some functions grow similarly yet others grow a lot faster, we use big-O notation.

Definition 1. A function $f(n)$ is said to be $O(g(n))$ (pronounced "big-oh of $g(n)$") if there exists N and c such that for all $n > N$, we have $|f(n)| \leq c|g(n)|$.

Problem 2. Let's formally justify the conclusions from Problem 1.

1. Show that $f(n) = n$ and $g(n) = 10n$ are both $O(n)$.
2. Translate "$f(n)$ grows a lot faster than $g(n)$" into big-O notation. Then show that $f(n) = n^2$ grows a lot faster than $g(n) = 10n$. This proves Problem 1.1.
3. (Challenge) Prove Problem 1.2, that $f(n) = n^a$ grows a lot faster than $g(n) = \log(n)$ for all $a \in (0, \infty)$.

Problem 3. Assuming the result of Problem 2.3, show that $f(n) = n^2$ grows a lot faster than $g(n) = n \log(n)$.

In areas of mathematics such as theoretical computer science or numerical methods, we use asymptotic growth rate to study the how the *running time of algorithms* grows as function of the input size. That is, n represents the size of the input, and $f(n)$ is the number of steps the algorithm takes to produce the answer.

The fast Fourier transform is a tool that solve problems that look like they might need around n^2 steps at first glance, but only using $O(n \log(n))$ time. As we noted in Problem 3, that's a huge difference and that's why it's called the fast Fourier transform!

2 Polynomial Multiplication

Although the fast Fourier transform is a very general tool, let's first specialize by looking at one particular use of it. We will use it to multiply polynomials fast.

To determine how fast we can do something, we count the number of operations. The only two operations we'll worry about today are addition and multiplication of two numbers. Computers have hardware circuitry to add and multiply in constant time, pretty much no matter how large the numbers are![1]

Problem 4. Describe the standard way to multiply two degree n polynomials through distributing. Show that it takes $O(n^2)$ operations to compute the product.

It's not obvious how to do this faster. Let's first develop a key idea.

Problem 5. In this problem, we will prove that given any n points in the plane $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ with the x_i's distinct, there exists a unique degree $n-1$ polynomial that passes through all the points. This is known as *Lagrange's interpolation theorem.*

1. (Warmup) Prove the statement in the case $n = 2$.
2. Lagrange's key idea was to look at the following polynomial for every $0 \leq j \leq n-1$:

$$p_j(x) = \prod_{i \neq j} \frac{x - x_i}{x_j - x_i}.$$

Quickly say why the denominators are non-zero and why $p_j(x)$ is a polynomial. What is the degree of $p_j(x)$? Then evaluate $p_j(x_k)$ for every $0 \leq j, k \leq n-1$.

[1] Okay... this is not 100% true, but it's true in 99.999% of real-world cases. Modern 64-bit computers represent numbers in 64 bits in base-2 scientific notation as follows: 1 bit indicates the sign (positive or negative), 52 bits indicate the coefficient (a fraction between 1 and 2 with denominator 2^{52}), and 11 bits indicate the exponent part (a power of two between 2^{-1023} and 2^{1024}). This is called *floating point* and it lets you represent numbers up to about 10^{308} with high precision, more than you'll probably ever need.

3. Write a degree $n-1$ polynomial that passes through $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$. This proves the existence half of the theorem.

4. For the uniqueness part, recall the Fundamental Theorem of Algebra: a non-zero degree $n-1$ polynomial has at most $n-1$ roots. Use this to prove that there cannot be two distinct degree $n-1$ polynomials that pass through all the points.

Now, we're ready to present a high-level overview of the fast multiplication algorithm! The input is two degree $n-1$ polynomials $p(x)$ and $q(x)$.

1. Choose points $x_0, \ldots, x_{2n-2}$ and evaluate both p and q at all of those points.
2. Compute the points $(x_0, p(x_0)q(x_0)), \ldots, (x_{2n-2}, p(x_{2n-2})q(x_{2n-2}))$.
3. Output the unique polynomial of degree at most $2n-2$ passing through the points computed in step 2, which exists by Lagrange's interpolation theorem.

Problem 6. Let's analyze the above algorithm.

1. Explain why the above algorithm works. In particular, explain the significance of choosing $2n-1$ points.
2. Give a way to do step 1 using $O(n^2)$ time. (Remember that our only operations are addition and multiplication, so taking an exponent is multiple operations.)
3. How much time does step 2 take? As usual, express your answer in big-O.
4. Convince yourself that step 3 takes a very long time, if you use the construction from the proof of Lagrange's interpolation theorem.

Clearly, steps 1 and 3 take too long using the methods we've discussed so far, since we're aiming for a total of $O(n\log(n))$ time. In the next two sections, we'll show how to do both both steps in $O(n\log(n))$ time each. Then, adding together all the steps, we will have multiplied the two polynomials in $O(n\log(n))$ time.

3 Evaluating and Recovering Polynomials

First, let's tackle evaluating p and q at points $x_0, \ldots, x_{2n-2}$. Before we get into the details, let's make one quick simplification.

Problem 7. Suppose for all n of the form $n = 2^m$ where $m \in \mathbb{N}$, you can evaluate $a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ on n points using $O(n\log(n))$ time. Prove that for all $n \in \mathbb{N}$, you can evaluate $b_0 + b_1x + \cdots + b_{n-1}x^{n-1}$ on $2n-1$ points using $O(n\log(n))$ time.

(Hint: Of course the second takes longer, but big-O notation hides constants!)

From now on, we'll always assume that n is a power of 2, which will simplify things a lot. This is a common trick in computer science: we can assume this in many different problems because we only care about big-O asymptotics, and powers of 2 have nice properties.

The key insight for multiplying polynomials fast is that if you choose the right points, a lot of the computation when evaluating polynomials can be reused.

Problem 8. Let's investigate how to choose points to reuse computation.

1. Recall that a function f is called even if $f(-x) = f(x)$ for all x, and odd if $f(-x) = -f(x)$ for all x. Show that you can write any polynomial $p(x)$ as the sum of an even polynomial $p_e(x)$ and an odd polynomial $p_o(x)$.

   (Challenge: Show this is true for all functions $f : \mathbb{R} \to \mathbb{R}$, not just polynomials.)

2. Using the decomposition above, explain how to reuse some computation when you evaluate a polynomial on both a point x_i and its opposite $-x_i$.
3. Say how you can view your answer to the previous part as involving two degree $n/2 - 1$ polynomials, $\widetilde{p_e}(x)$ and $\widetilde{p_o}(x)$. Write $p(x_i)$ and $p(-x_i)$ in terms of these smaller degree polynomials.
4. It would be nice if we could continue reusing computation when evaluating these smaller degree polynomials too. Assuming our inputs are real numbers, can we?

If we could resolve the issue in the last part of the previous problem, we could reuse a lot of computation and save a lot of time. The issue hinges on the fact that we restricted ourselves to real numbers. Luckily, it turns out that if we switch to complex numbers, all of our problems will be solved!

Recall that a complex number can be written in rectangular form $a + bi$, or in polar form $r(\cos(\theta) + i\sin(\theta))$. We often use the shorthand notation $e^{i\theta}$ for $\cos(\theta) + i\sin(\theta)$. Of course, raising e to an imaginary power sounds like nonsense, but we use this shorthand because it neatly encodes true facts about the polar form of complex numbers. For example, recall that when multiplying two complex numbers in polar form, we add their angles. With complex exponentials, this is obvious: $e^{i\theta}e^{i\varphi} = e^{i(\theta+\varphi)}$. But if you tried to show this by multiplying $(\cos(\theta) + i\sin(\theta))(\cos(\varphi) + i\sin(\varphi))$, you'd need to remember a bunch of trig identities.[2]

Definition 2. The nth roots of unity are the complex numbers $\{\omega_n^k : k = 0, \ldots, n-1\}$, where $\omega_n = e^{2\pi i/n} = \cos(2\pi/n) + i\sin(2\pi/n)$. (The letter ω is "omega.")

Problem 9. Let's investigate the roots of unity.

1. Compute and draw all 4th roots of unity in the complex plane. In a separate picture, repeat for 8th roots of unity.
2. Show that when n is even, the negation of every nth root of unity is another nth root of unity.

[2] A good question now is why base e, instead of any other base that also makes multiplication seamless? The cop-out answer is that it plays well with calculus: check that $\frac{d}{d\theta}\cos(\theta) + i\sin(\theta) = i(\cos(\theta) + i\sin(\theta))$, just like how $\frac{d}{d\theta}e^{i\theta} = ie^{i\theta}$ by chain rule. However now it seems like there is deeper relationship going on, and indeed there is, but that is beyond the scope of this worksheet.

3. Show that every nth root of unity has a square root, which is a mth root of unity for some m. (A square root of $z \in \mathbb{C}$ is a number $w \in \mathbb{C}$ such that $z = w^2$.)

Recall that in Problem 8, we tried to reuse computation as follows: to evaluate $p(x)$ at points x_i and $-x_i$,

1. Split the polynomial into odd and even parts as $p(x) = x\widetilde{p_o}(x^2) + \widetilde{p_e}(x^2)$, where $\widetilde{p_o}(x)$ and $\widetilde{p_e}(x)$ have degree $n/2 - 1$.
2. Evaluate $\widetilde{p_o}(x)$ and $\widetilde{p_e}(x)$ at $x_i^2 = (-x_i)^2$.
3. Compute

$$p(x_i) = x_i\widetilde{p_o}(x_i^2) + \widetilde{p_e}(x_i^2) \quad \text{and} \quad p(-x_i) = -x_i\widetilde{p_o}(x_i^2) + \widetilde{p_e}(x_i^2).$$

This gave us two evaluations for pretty much the price of one. But we wanted to further reuse computation by doing step 2 through the same procedure recursively. Then, we couldn't because we didn't have a negative counterpart for x_i^2 that was useful for us. Now, by choosing to evaluate $p(x)$ at roots of unity, we can!

Problem 10. It's finally time to evaluate $p(x)$ at n points in $O(n\log(n))$ time.

1. Following the above idea, evaluate $p(x) = x^3 + 2x^2 + 3x + 4$ at the 4th roots of unity, reusing as much computation as possible. In other words, keep splitting polynomials into odd and even parts until the parts are constant (degree 0).
2. Describe how to evaluate a general degree $n - 1$ polynomial at the nth roots of unity reusing as much computation as possible.
3. Show that your process takes $O(n\log(n))$ time.

   (Hint: Write a recursive formula that expresses $T(n)$, the amount of time it takes to evaluate a degree $n-1$ polynomial at these n points, in terms of $T(n/2)$. Then write $T(n/2)$ in terms of $T(n/4)$, and continue expanding until you reach $T(1)$, which is a constant. Don't worry about getting exact constant factors, because they'll be hidden in big-O anyway.)

We've finished describing how to evaluate polynomials in $O(n\log(n))$ time! This was step 1 of our fast polynomial multiplication algorithm. Recall that step 2 was just to multiply the evaluations together, which we discussed a while ago. Now, for step 3, we need to describe how to recover the product polynomial's coefficients from its values at $\omega_n^0, \ldots, \omega_n^{n-1}$. Surprisingly, all the hard work is already done! It turns out that this reverse operation is no different than the forward operation.

Problem 11. Let's figure out how to efficiently recover a polynomial from its values.

1. Show that when n is even, $\omega_n^0 + \cdots + \omega_n^{n-1} = 0$.

   (Challenge: Show this is also true for odd n.)

2. Let $p(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$. To reduce clutter, denote $p_k = p(\omega_n^k)$. In the previous problem, we showed how to evaluate

$$\begin{aligned} p_0 &= a_0\omega_n^0 + a_1\omega_n^0 + \cdots + a_{n-1}\omega_n^0 \\ p_1 &= a_0\omega_n^0 + a_1\omega_n^1 + \cdots + a_{n-1}\omega_n^{n-1} \\ &\vdots \\ p_{n-1} &= a_0\omega_n^0 + a_1\omega_n^{n-1} + \cdots + a_{n-1}\omega_n^{(n-1)^2}. \end{aligned}$$

   Using the above equations, solve for $a_0, \ldots, a_{n-1}$ in terms of $p_0, \ldots, p_{n-1}$.

   (Hint: To get a_0, add $p_0 + \cdots + p_{n-1}$ using part 1. Can you do something similar to get the other coefficients?)

3. Explain how to find $a_0, \ldots, a_{n-1}$ from knowing $p_0, \ldots, p_{n-1}$, using the exact same algorithm that we used to evaluate $p(x)$.

Congratulations! This completes our $O(n \log n)$ algorithm for multiplying two polynomials. At this point, we should briefly mention what the general fast Fourier transform is, and hopefully it should be clear how it's essentially exactly what we just did.

Definition 3. The fast Fourier transform is an algorithm that computes the function $\mathcal{F}(a_0, \ldots, a_{n-1}) = (b_0, \ldots, b_{n-1})$ where

$$b_k = a_0\omega_n^0 + a_1\omega_n^k + \cdots + a_{n-1}\omega_n^{k(n-1)},$$

and takes $O(n \log n)$ time. The inverse fast Fourier transform computes $\mathcal{F}^{-1}$, that is, $\mathcal{F}^{-1}(b_0, \ldots, b_{n-1}) = (a_0, \ldots, a_{n-1})$ where b_k is defined as above, and again takes $O(n \log n)$ time.