Lecture Overview

In this lecture we will discuss two main related topics

1. How to multiply two large polynomials (and therefore integers) using Fourier transforms
2. How to implement the fast Fourier transform (FFT) from scratch

- By doing this we will reduce the time complexity of polynomial multiplication from $O(n^2)$ to $O(n \log n)$.
- Perhaps more importantly, you will also learn one of the most useful and widely deployed computational tools in science and engineering.

Lecture Overview

In this lecture we will discuss two main related topics

1. How to multiply two large polynomials (and therefore integers) using Fourier transforms
2. How to implement the fast Fourier transform (FFT) from scratch

- By doing this we will reduce the time complexity of polynomial multiplication from $O(n^2)$ to $O(n \log n)$.
- Perhaps more importantly, you will also learn one of the most useful and widely deployed computational tools in science and engineering.

Lecture Overview

In this lecture we will discuss two main related topics

1. How to multiply two large polynomials (and therefore integers) using Fourier transforms
2. How to implement the fast Fourier transform (FFT) from scratch

- By doing this we will reduce the time complexity of polynomial multiplication from $O(n^2)$ to $O(n \log n)$.
- Perhaps more importantly, you will also learn one of the most useful and widely deployed computational tools in science and engineering.

Polynomials (1)

- A degree $n-1$ polynomial in x can be seen as a function:

$$A(x) = \sum_{i=0}^{n-1} a_i \cdot x^i.$$

- Any integer greater than the degree of a polynomial is a degree-bound of that polynomial.
- The polynomial A in x is:

$$a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 + \cdots + a_{n-1} x^{n-1}.$$

- The values a_i are the coefficients, the degree is $n-1$ and n, for example, is a degree-bound
- We can express any integer as a kind of polynomial by setting x to some base, say for decimal numbers:

$$A = \sum_{i=0}^{n-1} a_i \cdot 10^i.$$

Polynomials (1)

- A degree $n-1$ polynomial in x can be seen as a function:

$$A(x) = \sum_{i=0}^{n-1} a_i \cdot x^i.$$

- Any integer greater than the degree of a polynomial is a degree-bound of that polynomial.
- The polynomial A in x is:

$$a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 + \cdots + a_{n-1} x^{n-1}.$$

- The values a_i are the coefficients, the degree is $n-1$ and n, for example, is a degree-bound
- We can express any integer as a kind of polynomial by setting x to some base, say for decimal numbers:

$$A = \sum_{i=0}^{n-1} a_i \cdot 10^i.$$

Polynomials (1)

- A degree $n-1$ polynomial in x can be seen as a function:

$$A(x) = \sum_{i=0}^{n-1} a_i \cdot x^i.$$

- Any integer greater than the degree of a polynomial is a degree-bound of that polynomial.
- The polynomial A in x is:

$$a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 + \cdots + a_{n-1} x^{n-1}.$$

- The values a_i are the coefficients, the degree is $n-1$ and n, for example, is a degree-bound
- We can express any integer as a kind of polynomial by setting x to some base, say for decimal numbers:

$$A = \sum_{i=0}^{n-1} a_i \cdot 10^i.$$

Polynomials (1)

- A degree $n-1$ polynomial in x can be seen as a function:

$$A(x) = \sum_{i=0}^{n-1} a_i \cdot x^i.$$

- Any integer greater than the degree of a polynomial is a degree-bound of that polynomial.
- The polynomial A in x is:

$$a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 + \cdots + a_{n-1} x^{n-1}.$$

- The values a_i are the coefficients, the degree is $n-1$ and n, for example, is a degree-bound
- We can express any integer as a kind of polynomial by setting x to some base, say for decimal numbers:

$$A = \sum_{i=0}^{n-1} a_i \cdot 10^i.$$

Polynomials (1)

- A degree $n-1$ polynomial in x can be seen as a function:

$$A(x) = \sum_{i=0}^{n-1} a_i \cdot x^i.$$

- Any integer greater than the degree of a polynomial is a degree-bound of that polynomial.
- The polynomial A in x is:

$$a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 + \cdots + a_{n-1}x^{n-1}.$$

- The values a_i are the coefficients, the degree is $n-1$ and n, for example, is a degree-bound
- We can express any integer as a kind of polynomial by setting x to some base, say for decimal numbers:

$$A = \sum_{i=0}^{n-1} a_i \cdot 10^i.$$

Polynomials (2)

- The variable x allows us to evaluate the polynomial at a point:
- Evaluation just means plugging a value into the variable x.
- For example $A(3) = a_0 \cdot 3^0 + a_1 \cdot 3^1 + a_2 \cdot 3^2 \cdots + a_{n-1}3^{n-1}$.
- A fast way to evaluate a polynomial is using Horner's Rule.
  - Instead of computing all the terms individually, we do

$$A(3) = a_0 + 3 \cdot (a_1 + 3 \cdot (a_2 + \cdots + 3 \cdot (a_{n-1})))$$

  - This method requires $O(n)$ operations:

Polynomials (2)

- The variable x allows us to evaluate the polynomial at a point:
- Evaluation just means plugging a value into the variable x.
- For example $A(3) = a_0 \cdot 3^0 + a_1 \cdot 3^1 + a_2 \cdot 3^2 \cdots + a_{n-1}3^{n-1}$.
- A fast way to evaluate a polynomial is using Horner's Rule.
  - Instead of computing all the terms individually, we do

$$A(3) = a_0 + 3 \cdot (a_1 + 3 \cdot (a_2 + \cdots + 3 \cdot (a_{n-1})))$$

  - This method requires $O(n)$ operations:

Polynomials (2)

- The variable x allows us to evaluate the polynomial at a point:
- Evaluation just means plugging a value into the variable x.
- For example $A(3) = a_0 \cdot 3^0 + a_1 \cdot 3^1 + a_2 \cdot 3^2 \cdots + a_{n-1}3^{n-1}$.
- A fast way to evaluate a polynomial is using Horner's Rule.
  - Instead of computing all the terms individually, we do

$$A(3) = a_0 + 3 \cdot (a_1 + 3 \cdot (a_2 + \cdots + 3 \cdot (a_{n-1})))$$

  - This method requires $O(n)$ operations:

Polynomials (2)

- The variable x allows us to evaluate the polynomial at a point:
- Evaluation just means plugging a value into the variable x.
- For example $A(3) = a_0 \cdot 3^0 + a_1 \cdot 3^1 + a_2 \cdot 3^2 \cdots + a_{n-1}3^{n-1}$.
- A fast way to evaluate a polynomial is using Horner's Rule.
  - Instead of computing all the terms individually, we do

$$A(3) = a_0 + 3 \cdot (a_1 + 3 \cdot (a_2 + \cdots + 3 \cdot (a_{n-1})))$$

  - This method requires $O(n)$ operations:

Polynomials (2)

```
EVALUATE-HORNER(A, n, x)
begin
    t ← 0
    for i = n − 1 downto 0 step −1 do
        t ← (t · x) + a_i
    return t
end
```

Example

Consider $A(x) = 2 + 3x + 1.x^2$
We can evaluate this as

$$A(x) = 2 + x(3 + 1.x)$$

Coefficient Based Polynomial Arithmetic

- Once we have our polynomial representations, we might want to do some arithmetic with them.
- For a coefficient representation, the addition $C = A + B$ constructs C as the vector:

$$(a_0 + b_0, a_1 + b_1, a_2 + b_2, \ldots, a_{n-1} + b_{n-1}).$$

- Strictly speaking, A and B should have the same length but in practice we can just pad with zero coefficients to make this so.

Coefficient Based Polynomial Arithmetic

- Once we have our polynomial representations, we might want to do some arithmetic with them.
- For a coefficient representation, the addition $C = A + B$ constructs C as the vector:

$$(a_0 + b_0, a_1 + b_1, a_2 + b_2, \ldots, a_{n-1} + b_{n-1}).$$

- Strictly speaking, A and B should have the same length but in practice we can just pad with zero coefficients to make this so.

Coefficient Based Polynomial Arithmetic

- Once we have our polynomial representations, we might want to do some arithmetic with them.
- For a coefficient representation, the addition $C = A + B$ constructs C as the vector:

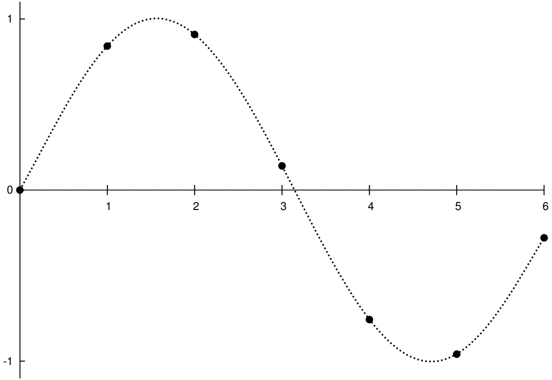
$$(a_0 + b_0, a_1 + b_1, a_2 + b_2, \ldots, a_{n-1} + b_{n-1}).$$

- Strictly speaking, A and B should have the same length but in practice we can just pad with zero coefficients to make this so.

Point Value Representation of Polynomials

Fact

Given n points (x_i, y_i), with all x_i distinct, there is a unique polynomial $A(x)$ of degree-bound n such that $y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$.



Point Value Polynomial Arithmetic

- For a point-value representation, the addition $C = A + B$ constructs C as:

  $$\{(x_0, y_0 + z_0), (x_1, y_1 + z_1), (x_2, y_2 + z_2), \ldots, (x_{n-1}, y_{n-1} + z_{n-1})\}$$

  where x_i is a point, $y_i = A(x_i)$ and $z_i = B(x_i)$.

- Note that the two point-value representations must use the same evaluation points.

- Both these operations are $O(n)$ in terms of the time they take.

Polynomial Multiplication

- Computing a polynomial multiplication, sometimes called convolution, is a little bit harder than addition.
- For a coefficient representation, the product $C = A \times B$ can be calculated with school-book long multiplication:

$$C(x) = \sum_{i=0}^{2n-2} c_i x_i$$

where

$$c_i = \sum_{j=0}^{i} a_j \cdot b_{i-j}$$

- To do now: multiply $7x^2 - 10x + 9$ and $2x^2 + 4x - 5$
- For a point-value representation, $C = A \times B$ is a bit easier:

$$\{(x_0, y_0 \cdot z_0), (x_1, y_1 \cdot z_1), (x_2, y_2 \cdot z_2), \ldots, (x_{n-1}, y_{n-1} \cdot z_{n-1})\}$$

where x_i is a point, $y_i = A(x_i)$ and $z_i = B(x_i)$.
- The first method is $O(n^2)$, the second method is $O(n)$!

Polynomial Multiplication

- Computing a polynomial multiplication, sometimes called convolution, is a little bit harder than addition.
- For a coefficient representation, the product $C = A \times B$ can be calculated with school-book long multiplication:

$$C(x) = \sum_{i=0}^{2n-2} c_i x_i$$

where

$$c_i = \sum_{j=0}^{i} a_j \cdot b_{i-j}$$

- To do now: multiply $7x^2 - 10x + 9$ and $2x^2 + 4x - 5$
- For a point-value representation, $C = A \times B$ is a bit easier:

$$\{(x_0, y_0 \cdot z_0), (x_1, y_1 \cdot z_1), (x_2, y_2 \cdot z_2), \ldots, (x_{n-1}, y_{n-1} \cdot z_{n-1})\}$$

where x_i is a point, $y_i = A(x_i)$ and $z_i = B(x_i)$.
- The first method is $O(n^2)$, the second method is $O(n)$!

Polynomial Multiplication

- Computing a polynomial multiplication, sometimes called convolution, is a little bit harder than addition.
- For a coefficient representation, the product $C = A \times B$ can be calculated with school-book long multiplication:

$$C(x) = \sum_{i=0}^{2n-2} c_i x_i$$

where

$$c_i = \sum_{j=0}^{i} a_j \cdot b_{i-j}$$

- To do now: multiply $7x^2 - 10x + 9$ and $2x^2 + 4x - 5$
- For a point-value representation, $C = A \times B$ is a bit easier:

$$\{(x_0, y_0 \cdot z_0), (x_1, y_1 \cdot z_1), (x_2, y_2 \cdot z_2), \ldots, (x_{n-1}, y_{n-1} \cdot z_{n-1})\}$$

where x_i is a point, $y_i = A(x_i)$ and $z_i = B(x_i)$.
- The first method is $O(n^2)$, the second method is $O(n)$!

Polynomial Multiplication

- Computing a polynomial multiplication, sometimes called convolution, is a little bit harder than addition.
- For a coefficient representation, the product $C = A \times B$ can be calculated with school-book long multiplication:

$$C(x) = \sum_{i=0}^{2n-2} c_i x_i$$

where

$$c_i = \sum_{j=0}^{i} a_j \cdot b_{i-j}$$

- To do now: multiply $7x^2 - 10x + 9$ and $2x^2 + 4x - 5$
- For a point-value representation, $C = A \times B$ is a bit easier:

$$\{(x_0, y_0 \cdot z_0), (x_1, y_1 \cdot z_1), (x_2, y_2 \cdot z_2), \ldots, (x_{n-1}, y_{n-1} \cdot z_{n-1})\}$$

where x_i is a point, $y_i = A(x_i)$ and $z_i = B(x_i)$.
- The first method is $O(n^2)$, the second method is $O(n)$!

Polynomial Multiplication

- Actually, we can do a bit better than the $O(n^2)$ case using the divide and conquer method due to Karatsuba.

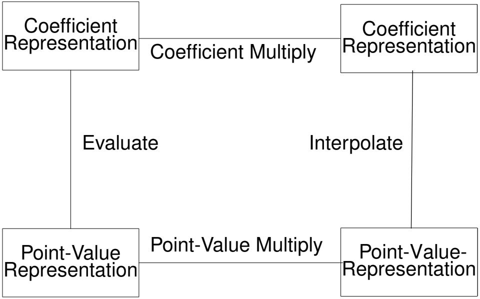
- This gives an algorithm with time complexity:

$$O(n^{\log_2 3}) = O(n^{1.59})$$

which is better than our previous method which took $O(n^2)$ operations.

Polynomial Multiplication

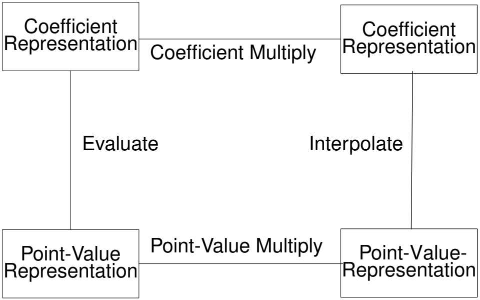
- The problem is that even this is too slow ... we know that using a point-value representation is $O(n)$!
- So a better technique would be to traverse around this diagram:



- Note that the opposite of evaluation is called interpolation.
  - So we evaluate to a point-value representation, multiply and then interpolate back again.
  - The question is, are we quicker than the normal multiply ?

Polynomial Multiplication

- The problem is that even this is too slow ... we know that using a point-value representation is $O(n)$!
- So a better technique would be to traverse around this diagram:



- Note that the opposite of evaluation is called interpolation.
  - So we evaluate to a point-value representation, multiply and then interpolate back again.
  - The question is, are we quicker than the normal multiply ?

The Main Idea -Part 1

Develop two fast algorithms that for any polynomial:

$$A(x) = \sum_{i=0}^{n-1} a_i \cdot x^i,$$

and a preselected set $x_0, x_1, \ldots, x_{n-1}$ of numbers (to be specified before we know which polynomials we will have),

- Evaluate $A(x_0), A(x_1), \ldots, A(x_{n-1})$ (evaluate)
- Given $A(x_0), A(x_1), \ldots, A(x_{n-1})$, reconstruct A's coefficients $a_0, a_1, \ldots a_{m-1}$ (interpolate)

The Main Idea -Part 2

The main steps for fast multiplication of two polynomials A and B each of degree n are:

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each
2. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of the FFT of order $2n$.
3. *Pointwise multiply:* Compute a point-value representation of $C(x) = A(x)B(x)$ by multiplying the values pointwise
4. *Interpolate:* Create a coefficient representation of $C(x)$ through a single application of the *inverse* FFT.

The first and third steps are easy to perform in $O(n)$ time. The claim is that if we evaluate at the complex roots of unity then we can perform steps 2 and 4 in $O(n \log n)$ time.

The Main Idea -Part 2

The main steps for fast multiplication of two polynomials A and B each of degree n are:

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each
2. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of the FFT of order $2n$.
3. *Pointwise multiply:* Compute a point-value representation of $C(x) = A(x)B(x)$ by multiplying the values pointwise
4. *Interpolate:* Create a coefficient representation of $C(x)$ through a single application of the *inverse* FFT.

The first and third steps are easy to perform in $O(n)$ time. The claim is that if we evaluate at the complex roots of unity then we can perform steps 2 and 4 in $O(n \log n)$ time.

The Main Idea -Part 2

The main steps for fast multiplication of two polynomials A and B each of degree n are:

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each
2. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of the FFT of order $2n$.
3. *Pointwise multiply:* Compute a point-value representation of $C(x) = A(x)B(x)$ by multiplying the values pointwise
4. *Interpolate:* Create a coefficient representation of $C(x)$ through a single application of the *inverse* FFT.

The first and third steps are easy to perform in $O(n)$ time. The claim is that if we evaluate at the complex roots of unity then we can perform steps 2 and 4 in $O(n \log n)$ time.

The Main Idea -Part 2

The main steps for fast multiplication of two polynomials A and B each of degree n are:

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each
2. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of the FFT of order $2n$.
3. *Pointwise multiply:* Compute a point-value representation of $C(x) = A(x)B(x)$ by multiplying the values pointwise
4. *Interpolate:* Create a coefficient representation of $C(x)$ through a single application of the *inverse* FFT.

The first and third steps are easy to perform in $O(n)$ time. The claim is that if we evaluate at the complex roots of unity then we can perform steps 2 and 4 in $O(n \log n)$ time.

Evaluation at Roots of Unity

- First let's address evaluation:
  - We need to evaluate a polynomial of degree n at n different points (ignore the degree-bound doubling for the moment).
  - Appears complexity of our method will be $O(n^2)$.
  - Is there a faster way of doing this than just using Horner's Rule ?
- Yes there is, we select the points we evaluate at to be special.
- These special points are chosen to be the N-th Complex Roots of Unity:
  - That is, the values $\omega_N = e^{2\pi ij/N}$ for $j = 0, 1, \ldots, N-1$.
  - Say we are evaluating at N points so we take the N-th complex roots of unity ω_N.
  - That is, we evaluate the polynomial at the points:

$$\omega_N^0, \omega_N^1, \omega_N^2, \ldots, \omega_N^{N-1}.$$

Evaluation at Roots of Unity

- First let's address evaluation:
  - We need to evaluate a polynomial of degree n at n different points (ignore the degree-bound doubling for the moment).
  - Appears complexity of our method will be $O(n^2)$.
  - Is there a faster way of doing this than just using Horner's Rule ?
- Yes there is, we select the points we evaluate at to be special.
- These special points are chosen to be the N-th Complex Roots of Unity:
  - That is, the values $\omega_N = e^{2\pi ij/N}$ for $j = 0, 1, \ldots, N-1$.
  - Say we are evaluating at N points so we take the N-th complex roots of unity ω_N.
  - That is, we evaluate the polynomial at the points:

$$\omega_N^0, \omega_N^1, \omega_N^2, \ldots, \omega_N^{N-1}.$$

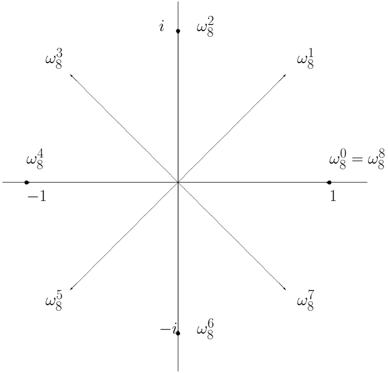
Evaluation at Roots of Unity

- First let's address evaluation:
  - We need to evaluate a polynomial of degree n at n different points (ignore the degree-bound doubling for the moment).
  - Appears complexity of our method will be $O(n^2)$.
  - Is there a faster way of doing this than just using Horner's Rule ?
- Yes there is, we select the points we evaluate at to be special.
- These special points are chosen to be the N-th Complex Roots of Unity:
  - That is, the values $\omega_N = e^{2\pi ij/N}$ for $j = 0, 1, \ldots, N-1$.
  - Say we are evaluating at N points so we take the N-th complex roots of unity ω_N.
  - That is, we evaluate the polynomial at the points:

$$\omega_N^0, \omega_N^1, \omega_N^2, \ldots, \omega_N^{N-1}.$$

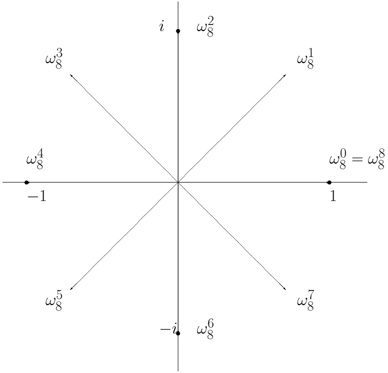
Evaluation at Roots of Unity

- What the hell am I talking about ? Try an example:
  - We know that $\omega_N^j = e^{2\pi ij/N}$ for $j = 0, 1, \ldots, N-1$.
  - So given the well known identity $e^{iu} = \cos(u) + i\sin(u)$, we can draw the values ω_N^j.
  - An easy one to draw is for $N = 8$.



Evaluation at Roots of Unity

- What the hell am I talking about ? Try an example:
  - We know that $\omega_N^j = e^{2\pi ij/N}$ for $j = 0, 1, \ldots, N-1$.
  - So given the well known identity $e^{iu} = \cos(u) + i\sin(u)$, we can draw the values ω_N^j.
  - An easy one to draw is for $N = 8$.



Discrete Fourier Transform

We want to evaluate a polynomial A at the n roots of unity.

- Therefore we evaluate

$$A(x) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

for every $k = 0, 1, \ldots, n-1$.

- Let's define the vector of results of these evaluations as

$$y_k = A(\omega_n^k)$$

- This vector $y = (y_0, \ldots, y_{n-1})$ is the Discrete Fourier Transform (DFT) of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$.

Example

The discrete Fourier transform of $0 + 0x + x^2 - x^3$ is $0, -1 + i, 2, -1 - i$

Discrete Fourier Transform

We want to evaluate a polynomial A at the n roots of unity.

- Therefore we evaluate

$$A(x) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

  for every $k = 0, 1, \ldots, n-1$.

- Let's define the vector of results of these evaluations as

$$y_k = A(\omega_n^k)$$

- This vector $y = (y_0, \ldots, y_{n-1})$ is the Discrete Fourier Transform (DFT) of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$.

Example

The discrete Fourier transform of $0 + 0x + x^2 - x^3$ is $0, -1+i, 2, -1-i$

Discrete Fourier Transform

We want to evaluate a polynomial A at the n roots of unity.

- Therefore we evaluate
$$A(x) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$
for every $k = 0, 1, \ldots, n-1$.
- Let's define the vector of results of these evaluations as
$$y_k = A(\omega_n^k)$$
- This vector $y = (y_0, \ldots, y_{n-1})$ is the Discrete Fourier Transform (DFT) of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$.

Example

The discrete Fourier transform of $0 + 0x + x^2 - x^3$ is $0, -1 + i, 2, -1 - i$

A Couple of Lemmas

Lemma

The Cancellation Lemma: $\omega_{dN}^{dk} = \omega_N^k$.

Lemma

The Halving Lemma: If $N > 0$ is even then the squares of the N complex N-th roots of unity are the $N/2$ complex $N/2$-th roots of unity.

Proof.

By the cancellation lemma, we have $(\omega_n^k)^2 = \omega_{n/2}^k$, for any nonnegative integer k. □

A Couple of Lemmas

Lemma

The Cancellation Lemma: $\omega_{dN}^{dk} = \omega_N^k$.

Lemma

The Halving Lemma: *If* $N > 0$ *is even then the squares of the* N *complex* N*-th roots of unity are the* $N/2$ *complex* $N/2$*-th roots of unity.*

Proof.

By the cancellation lemma, we have $(\omega_n^k)^2 = \omega_{n/2}^k$, for any nonnegative integer k. $\square$

A Couple of Lemmas

Lemma

The Cancellation Lemma: $\omega_{dN}^{dk} = \omega_N^k$.

Lemma

The Halving Lemma: *If $N > 0$ is even then the squares of the N complex N-th roots of unity are the $N/2$ complex $N/2$-th roots of unity.*

It follows from the Halving Lemma that if we square all the nth roots of unity, then each $(n/2)$th root of unity is obtained exactly twice. In other words,

$$(\omega_N^0)^2, (\omega_N^1)^2, (\omega_N^2)^2, \ldots, (\omega_N^{N-1})^2$$

consists not of n distinct values but only of $n/2$ values, each of which occurs exactly twice.

Fast Fourier Transform

- The basic idea of the Fast Fourier Transform (FFT), a fast version of the DFT, is define two new polynomials:

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + \cdots + a_{N-2} x^{N/2-1} \\ A^{[1]}(x) &= a_1 + a_3 x + \cdots + a_{N-1} x^{N/2-1} \end{aligned}$$

  and use these to divide and conquer the problem.
- From the above, we have:

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2).$$

- So the problem of evaluating A at $\omega_N^0, \omega_N^1, \ldots, \omega_N^{N-1}$ is reduced to evaluating $A^{[0]}$ and $A^{[1]}$ at the points:

$$(\omega_N^0)^2, (\omega_N^1)^2, (\omega_N^2)^2, \ldots, (\omega_N^{N-1})^2$$

  then combining the results.
- The Halving Lemma tells us there are only $N/2$ complex $N/2$-th roots of unity, each one must occur twice !

Fast Fourier Transform

- The basic idea of the Fast Fourier Transform (FFT), a fast version of the DFT, is define two new polynomials:

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + \cdots + a_{N-2} x^{N/2-1} \\ A^{[1]}(x) &= a_1 + a_3 x + \cdots + a_{N-1} x^{N/2-1} \end{aligned}$$

  and use these to divide and conquer the problem.
- From the above, we have:

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2).$$

- So the problem of evaluating A at $\omega_N^0, \omega_N^1, \ldots, \omega_N^{N-1}$ is reduced to evaluating $A^{[0]}$ and $A^{[1]}$ at the points:

$$(\omega_N^0)^2, (\omega_N^1)^2, (\omega_N^2)^2, \ldots, (\omega_N^{N-1})^2$$

  then combining the results.
- The Halving Lemma tells us there are only $N/2$ complex $N/2$-th roots of unity, each one must occur twice !

Fast Fourier Transform

- The basic idea of the Fast Fourier Transform (FFT), a fast version of the DFT, is define two new polynomials:

$$
\begin{aligned}
A^{[0]}(x) &= a_0 + a_2 x + \cdots + a_{N-2} x^{N/2-1} \\
A^{[1]}(x) &= a_1 + a_3 x + \cdots + a_{N-1} x^{N/2-1}
\end{aligned}
$$

  and use these to divide and conquer the problem.
- From the above, we have:

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2).$$

- So the problem of evaluating A at $\omega_N^0, \omega_N^1, \ldots, \omega_N^{N-1}$ is reduced to evaluating $A^{[0]}$ and $A^{[1]}$ at the points:

$$(\omega_N^0)^2, (\omega_N^1)^2, (\omega_N^2)^2, \ldots, (\omega_N^{N-1})^2$$

  then combining the results.
- The Halving Lemma tells us there are only $N/2$ complex $N/2$-th roots of unity, each one must occur twice !

Fast Fourier Transform

- The basic idea of the Fast Fourier Transform (FFT), a fast version of the DFT, is define two new polynomials:

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + \cdots + a_{N-2} x^{N/2-1} \\ A^{[1]}(x) &= a_1 + a_3 x + \cdots + a_{N-1} x^{N/2-1} \end{aligned}$$

  and use these to divide and conquer the problem.
- From the above, we have:

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2).$$

- So the problem of evaluating A at $\omega_N^0, \omega_N^1, \ldots, \omega_N^{N-1}$ is reduced to evaluating $A^{[0]}$ and $A^{[1]}$ at the points:

$$(\omega_N^0)^2, (\omega_N^1)^2, (\omega_N^2)^2, \ldots, (\omega_N^{N-1})^2$$

  then combining the results.
- The Halving Lemma tells us there are only $N/2$ complex $N/2$-th roots of unity, each one must occur twice !

Example of Divide Step

Example

Consider $A[x] = 0 + 0x + 1.x^2 - x^3$ again.

$$A^{[0]}[x] = a_0 + a_2 x = 0 + 1.x$$
$$A^{[1]}[x] = a_1 + a_3 x = 0 - 1.x$$

We can check by seeing that

$$\begin{aligned} A[x] &= A^{[0]}[x^2] + x.A^{[1]}[x^2] \\ &= 0 + 1.x^2 + x(0 - 1.x^2) \\ &= x^2 - x^3 \end{aligned}$$

as required.

Fast Fourier Transform

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + \cdots + a_{N-2} x^{N/2-1} \\ A^{[1]}(x) &= a_1 + a_3 x + \cdots + a_{N-1} x^{N/2-1} \end{aligned}$$

Polynomials $A^{[0]}$ and $A^{[1]}$ of degree-bound $n/2$ are recursively evaluated at the $n/2$ complex $(n/2)$th roots of unity

- These subproblems have exactly the same form as the original problem
- However, they are *half* the size because of the Halving Lemma
- So we can divide an n-element DFT computation into two $n/2$-element DFT computations and combine the results in linear time.
- This sort of divide and conquer strategy should remind you of merge sort, for example.

Fast Fourier Transform

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + \cdots + a_{N-2} x^{N/2-1} \\ A^{[1]}(x) &= a_1 + a_3 x + \cdots + a_{N-1} x^{N/2-1} \end{aligned}$$

Polynomials $A^{[0]}$ and $A^{[1]}$ of degree-bound $n/2$ are recursively evaluated at the $n/2$ complex $(n/2)$th roots of unity

- These subproblems have exactly the same form as the original problem
- However, they are *half* the size because of the Halving Lemma
- So we can divide an n-element DFT computation into two $n/2$-element DFT computations and combine the results in linear time.
- This sort of divide and conquer strategy should remind you of merge sort, for example.

Fast Fourier Transform

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + \cdots + a_{N-2} x^{N/2-1} \\ A^{[1]}(x) &= a_1 + a_3 x + \cdots + a_{N-1} x^{N/2-1} \end{aligned}$$

Polynomials $A^{[0]}$ and $A^{[1]}$ of degree-bound $n/2$ are recursively evaluated at the $n/2$ complex $(n/2)$th roots of unity

- These subproblems have exactly the same form as the original problem
- However, they are *half* the size because of the Halving Lemma
- So we can divide an n-element DFT computation into two $n/2$-element DFT computations and combine the results in linear time.
- This sort of divide and conquer strategy should remind you of merge sort, for example.

Fast Fourier Transform

$$
\begin{aligned}
A^{[0]}(x) &= a_0 + a_2 x + \cdots + a_{N-2} x^{N/2-1} \\
A^{[1]}(x) &= a_1 + a_3 x + \cdots + a_{N-1} x^{N/2-1}
\end{aligned}
$$

Polynomials $A^{[0]}$ and $A^{[1]}$ of degree-bound $n/2$ are recursively evaluated at the $n/2$ complex $(n/2)$th roots of unity

- These subproblems have exactly the same form as the original problem
- However, they are *half* the size because of the Halving Lemma
- So we can divide an n-element DFT computation into two $n/2$-element DFT computations and combine the results in linear time.
- This sort of divide and conquer strategy should remind you of merge sort, for example.

Fast Fourier Transform

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + \cdots + a_{N-2} x^{N/2-1} \\ A^{[1]}(x) &= a_1 + a_3 x + \cdots + a_{N-1} x^{N/2-1} \end{aligned}$$

Polynomials $A^{[0]}$ and $A^{[1]}$ of degree-bound $n/2$ are recursively evaluated at the $n/2$ complex $(n/2)$th roots of unity

- These subproblems have exactly the same form as the original problem
- However, they are *half* the size because of the Halving Lemma
- So we can divide an n-element DFT computation into two $n/2$-element DFT computations and combine the results in linear time.
- This sort of divide and conquer strategy should remind you of merge sort, for example.

Fast Fourier Transform

- The final recursive algorithm looks something like this:

FFT(A, N)
begin
    if $N = 1$ **then**
        return A
    else
        $\omega_N \leftarrow e^{2\pi i/N}$
        $\omega \leftarrow 1$
        $A^{[0]} \leftarrow (a_0, a_2, a_4, \ldots, a_{N-2})$
        $A^{[1]} \leftarrow (a_1, a_3, a_5, \ldots, a_{N-1})$
        $y^{[0]} \leftarrow FFT(A^{[0]}, N/2)$
        $y^{[1]} \leftarrow FFT(A^{[1]}, N/2)$
        for $k = 0$ **upto** $N/2 - 1$ **step** 1 **do**
            $y_k \leftarrow y_k^{[0]} + \omega \cdot y_k^{[1]}$
            $y_{k+N/2} \leftarrow y_k^{[0]} - \omega \cdot y_k^{[1]}$
            $\omega \leftarrow \omega \cdot w_N$
        return y
end

- Simply put, we first define $A^{[0]}$ and $A^{[1]}$ and then recursively evaluate them.

Fast Fourier Transform

1 FFT(A, N)
2 **begin**
3   **if** $N = 1$ **then**
4   **return** A
5   **else**
6         $\omega_N \leftarrow e^{2\pi i/N}$
7         $\omega \leftarrow 1$
8         $A^{[0]} \leftarrow (a_0, a_2, a_4, \dots, a_{N-2})$
9         $A^{[1]} \leftarrow (a_1, a_3, a_5, \dots, a_{N-1})$
10         $y^{[0]} \leftarrow FFT(A^{[0]}, N/2)$
11         $y^{[1]} \leftarrow FFT(A^{[1]}, N/2)$
12   **for** $k = 0$ **upto** $N/2 - 1$ **step** 1 **do**
13             $y_k \leftarrow y_k^{[0]} + \omega \cdot y_k^{[1]}$
14             $y_{k+N/2} \leftarrow y_k^{[0]} - \omega \cdot y_k^{[1]}$
15             $\omega \leftarrow \omega \cdot w_N$
16   **return** y
17 **end**

- Lines 3-4 are the base case for the recursion

Fast Fourier Transform

1 FFT(A, N)
2 **begin**
3   **if** $N = 1$ **then**
4   **return** A
5   **else**
6         $\omega_N \leftarrow e^{2\pi i/N}$
7         $\omega \leftarrow 1$
8         $A^{[0]} \leftarrow (a_0, a_2, a_4, \ldots, a_{N-2})$
9         $A^{[1]} \leftarrow (a_1, a_3, a_5, \ldots, a_{N-1})$
10         $y^{[0]} \leftarrow FFT(A^{[0]}, N/2)$
11         $y^{[1]} \leftarrow FFT(A^{[1]}, N/2)$
12   **for** $k = 0$ **upto** $N/2 - 1$ **step** 1 **do**
13             $y_k \leftarrow y_k^{[0]} + \omega \cdot y_k^{[1]}$
14             $y_{k+N/2} \leftarrow y_k^{[0]} - \omega \cdot y_k^{[1]}$
15             $\omega \leftarrow \omega \cdot w_N$
16   **return** y
17 **end**

- Lines 10-11 perform the recursive calls

Fast Fourier Transform

```
FFT(A, N)
begin
    if N = 1 then
        return A
    else
        ω_N ← e^{2πi/N}
        ω ← 1
        A^[0] ← (a_0, a_2, a_4, ..., a_{N−2})
        A^[1] ← (a_1, a_3, a_5, ..., a_{N−1})
        y^[0] ← FFT(A^[0], N/2)
        y^[1] ← FFT(A^[1], N/2)
        for k = 0 upto N/2 − 1 step 1 do
            y_k ← y_k^[0] + ω · y_k^[1]
            y_{k+N/2} ← y_k^[0] − ω · y_k^[1]
            ω ← ω · w_N
        return y
end
```

For $y_0, y_1, \ldots, y_{n/2-1}$,

$$
\begin{aligned}
y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\
&= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\
&= A(\omega_n^k)
\end{aligned}
$$

Fast Fourier Transform

```
12      for k = 0 upto N/2 − 1 step 1 do
13          y_k ← y_k^[0] + ω · y_k^[1]
14          y_{k+N/2} ← y_k^[0] − ω · y_k^[1]
15          ω ← ω · w_N
16      return y
```

For $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$, line 14 gives

$$\begin{aligned}
y_{k+n/2} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\
&= y_k^{[0]} + \omega_n^{k+n/2} y_k^{[1]} \\
&= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+n/2} A^{[1]}(\omega_n^{2k}) \\
&= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+n/2} A^{[1]}(\omega_n^{2k+n}) \\
&= A(\omega_n^{k+n/2})
\end{aligned}$$

Fast Fourier Transform

1 FFT(A, N)
2 **begin**
3   **if** $N = 1$ **then**
4   **return** A
5   **else**
6         $\omega_N \leftarrow e^{2\pi i/N}$
7         $\omega \leftarrow 1$
8         $A^{[0]} \leftarrow (a_0, a_2, a_4, \ldots, a_{N-2})$
9         $A^{[1]} \leftarrow (a_1, a_3, a_5, \ldots, a_{N-1})$
10         $y^{[0]} \leftarrow FFT(A^{[0]}, N/2)$
11         $y^{[1]} \leftarrow FFT(A^{[1]}, N/2)$
12   **for** $k = 0$ **upto** $N/2 - 1$ **step** 1 **do**
13             $y_k \leftarrow y_k^{[0]} + \omega \cdot y_k^{[1]}$
14             $y_{k+N/2} \leftarrow y_k^{[0]} - \omega \cdot y_k^{[1]}$
15             $\omega \leftarrow \omega \cdot w_N$
16   **return** y
17 **end**

- Lines 6, 7 and 15 simply keep ω updated to save having to recompute ω_n^k in every iteration of the *for* loop.

FFT - Worked Example

Example

Consider $A[x] = 0 + 0x + 1.x^2 - x^3$ once more. $N = 4$ so we need to compute the four 4th roots of unity. Line 7 sets $\omega \leftarrow 1$, as 1 is always the first root. $\omega_4 \leftarrow \cos(\pi/2) + i\sin(\pi/2) = i$.

$$A^{[0]} \leftarrow (0, 1)$$
$$A^{[1]} \leftarrow (0, -1)$$
$$y^{[0]} \leftarrow \mathrm{FFT}((0, 1), 2)$$
$$y^{[1]} \leftarrow \mathrm{FFT}((0, -1), 2)$$

What is $\mathrm{FFT}((0, 1), 2)$? It's simply the two squares roots of unity. I.e. $(1, -1)$. Similarly, $\mathrm{FFT}((0, -1), 2)$ is simply $(-1, 1)$

FFT - Worked Example contd.

Example

The first iteration of the loop from line 12 gives us

$$y_0 = 1 + (-1) = 0$$

$$y_2 = 1 - (-1) = 2$$

Now we update $\omega \leftarrow i$ on line 15 and perform the second loop

$$y_1 = -1 + i$$

$$y_3 = -1 - i$$

So the 4 point FFT of $A[x] = x^2 - x^3$ is $0, -1 + i, 2, -1 - i$ as we showed before.

Fast Fourier Transform - Analysis

To analyse the time complexity of the FFT we observe that:

- Each recursive call in Lines 10-11 calls FFT with a coefficient vector of length $n/2$.
- Lines 13-14 take $\Theta(n)$ time to compute in total.

The running time of the FFT can therefore be expressed as

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Polynomial Evaluation - Summary

Remember that our aim was to evaluate two polynomials A and B of degree-bound n at the roots of unity. We also needed to double the degree-bound to help us perform the multiplication later on.

1. Pad the coefficient vector for A with zeros so that their length is $2n$ (assume n is a power of two).
2. Define $A(x) = \sum_{i=0}^{2n-1} a_i x^i$ (half of the coefficients are zero).
3. Define $y_j = A(\omega_n^j) = \sum_{i=0}^{n-1} a_i \omega_n^{ji}$.
4. Then the vector $y = (y_0, y_1, y_2, \ldots, y_{2n-1})$ is the $2n$-element DFT of A.
5. We have our point-value representation as:

$$\{(\omega_{2n}^0, y_0), (\omega_{2n}^1, y_1), (\omega_{2n}^2, y_2), \ldots, (\omega_{2n}^{2n-1}, y_{2n-1})\}$$

Repeat the same process for polynomial B.

Inverse Fourier Transform - Interpolation

- We will use the Inverse DFT to interpolate polynomials. This relies on a Theorem that shows how to invert the DFT:

$$a_i = \frac{1}{n}\sum_{j=0}^{n-1} y_j w_n^{-ji}$$

  Although we won't prove this here, this allows us to convert the point-value representation of a polynomial to coefficient form.

- So if we can compute the DFT, the Inverse DFT simply does the same thing with a few amendments:
  1. Switching roles of a and y
  2. Replace ω_n by ω_n^{-1},
  3. Divide the final result by n.
- The Inverse DFT can therefore be computed in the same time complexity as the DFT. I.e. both take $O(n \log n)$ time.

Inverse Fourier Transform - Interpolation

- We will use the Inverse DFT to interpolate polynomials. This relies on a Theorem that shows how to invert the DFT:

$$a_i = \frac{1}{n}\sum_{j=0}^{n-1} y_j w_n^{-ji}$$

  Although we won't prove this here, this allows us to convert the point-value representation of a polynomial to coefficient form.

- So if we can compute the DFT, the Inverse DFT simply does the same thing with a few amendments:
  1. Switching roles of a and y
  2. Replace ω_n by ω_n^{-1},
  3. Divide the final result by n.
- The Inverse DFT can therefore be computed in the same time complexity as the DFT. I.e. both take $O(n \log n)$ time.

Inverse Fourier Transform - Interpolation

- We will use the Inverse DFT to interpolate polynomials. This relies on a Theorem that shows how to invert the DFT:

  $$a_i = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-ji}$$

  Although we won't prove this here, this allows us to convert the point-value representation of a polynomial to coefficient form.
- So if we can compute the DFT, the Inverse DFT simply does the same thing with a few amendments:
  1. Switching roles of a and y
  2. Replace ω_n by ω_n^{-1},
  3. Divide the final result by n.
- The Inverse DFT can therefore be computed in the same time complexity as the DFT. I.e. both take $O(n \log n)$ time.

Polynomial Multiplication - Summary

We have shown how to perform the main steps involved in polynomial multiplication

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each. $O(n)$ time.
2. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of the FFT of order $2n$. $O(n \log n)$ time.
3. *Pointwise multiply:* Compute a point-value representation of $C(x) = A(x)B(x)$ by multiplying the values pointwise. $O(n)$ time
4. *Interpolate:* Create a coefficient representation of $C(x)$ through a single application of the *inverse* FFT. $O(n \log n)$ time.

Therefore the overall time complexity of polynomial multiplication is $O(n \log n)$. This is a lot better than the $O(n^2)$ time we started with!

Polynomial Multiplication - Summary

We have shown how to perform the main steps involved in polynomial multiplication

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each. $O(n)$ time.
2. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of the FFT of order $2n$. $O(n \log n)$ time.
3. *Pointwise multiply:* Compute a point-value representation of $C(x) = A(x)B(x)$ by multiplying the values pointwise. $O(n)$ time
4. *Interpolate:* Create a coefficient representation of $C(x)$ through a single application of the *inverse* FFT. $O(n \log n)$ time.

Therefore the overall time complexity of polynomial multiplication is $O(n \log n)$. This is a lot better than the $O(n^2)$ time we started with!

Polynomial Multiplication - Summary

We have shown how to perform the main steps involved in polynomial multiplication

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each. $O(n)$ time.
2. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of the FFT of order $2n$. $O(n \log n)$ time.
3. *Pointwise multiply:* Compute a point-value representation of $C(x) = A(x)B(x)$ by multiplying the values pointwise. $O(n)$ time
4. *Interpolate:* Create a coefficient representation of $C(x)$ through a single application of the *inverse* FFT. $O(n \log n)$ time.

Therefore the overall time complexity of polynomial multiplication is $O(n \log n)$. This is a lot better than the $O(n^2)$ time we started with!

Polynomial Multiplication - Summary

We have shown how to perform the main steps involved in polynomial multiplication

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each. $O(n)$ time.
2. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of the FFT of order $2n$. $O(n \log n)$ time.
3. *Pointwise multiply:* Compute a point-value representation of $C(x) = A(x)B(x)$ by multiplying the values pointwise. $O(n)$ time
4. *Interpolate:* Create a coefficient representation of $C(x)$ through a single application of the *inverse* FFT. $O(n \log n)$ time.

Therefore the overall time complexity of polynomial multiplication is $O(n \log n)$. This is a lot better than the $O(n^2)$ time we started with!

Conclusions

- We are able to multiply two polynomials of degree-bound n in $O(n \log n)$ operations.
- Therefore we can multiply polynomials faster than even the Karatsuba approach.
  - The extra operations mean this method is only faster for reasonably large polynomials.
- Consider the context:
  - Graphics and signal processing applications use FFT a lot on very large data sets.
  - Even a small improvement in asymptotic time complexity will help massively when the input size is large.
  - FFTs can also be used for string matching problems.
- We've taken a step in the right direction using two underlying principles:
  - Divide and conquer is a very powerful tool.
  - A little mathematics can go a very long way.

Conclusions

- We are able to multiply two polynomials of degree-bound n in $O(n \log n)$ operations.
- Therefore we can multiply polynomials faster than even the Karatsuba approach.
  - The extra operations mean this method is only faster for reasonably large polynomials.
- Consider the context:
  - Graphics and signal processing applications use FFT a lot on very large data sets.
  - Even a small improvement in asymptotic time complexity will help massively when the input size is large.
  - FFTs can also be used for string matching problems.
- We've taken a step in the right direction using two underlying principles:
  - Divide and conquer is a very powerful tool.
  - A little mathematics can go a very long way.

Conclusions

- We are able to multiply two polynomials of degree-bound n in $O(n \log n)$ operations.
- Therefore we can multiply polynomials faster than even the Karatsuba approach.
  - The extra operations mean this method is only faster for reasonably large polynomials.
- Consider the context:
  - Graphics and signal processing applications use FFT a lot on very large data sets.
  - Even a small improvement in asymptotic time complexity will help massively when the input size is large.
  - FFTs can also be used for string matching problems.
- We've taken a step in the right direction using two underlying principles:
  - Divide and conquer is a very powerful tool.
  - A little mathematics can go a very long way.

Conclusions

- We are able to multiply two polynomials of degree-bound n in $O(n \log n)$ operations.
- Therefore we can multiply polynomials faster than even the Karatsuba approach.
  - The extra operations mean this method is only faster for reasonably large polynomials.
- Consider the context:
  - Graphics and signal processing applications use FFT a lot on very large data sets.
  - Even a small improvement in asymptotic time complexity will help massively when the input size is large.
  - FFTs can also be used for string matching problems.
- We've taken a step in the right direction using two underlying principles:
  - Divide and conquer is a very powerful tool.
  - A little mathematics can go a very long way.

Further Reading

- Introduction to Algorithms
  T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein.
  MIT Press/McGraw-Hill, ISBN: 0-262-03293-7.
  - Chapter 30 – Polynomials and the FFT
- Algorithm Design
  J. Kleinberg and É.Tardos.
  Pearson/Addison-Wesley, ISBN: 0-321-29535-8.
  - Chapter 5 – Divide and Conquer