Algorithm Design V

Divide and Conquer II

Guoqiang Li
School of Software, Shanghai Jiao Tong University

SHANGHAI JIAO TONG UNIVERSITY

Complex Number

Complex Number

$z = a + bi$ is plotted at position (a, b).

Complex Number

$z = a + bi$ is plotted at position (a, b).

In its polar coordinates, denoted (r, θ), rewrite as

$$z = r(\cos\theta + i\sin\theta) = re^{i\theta}$$

- length: $r = \sqrt{a^2 + b^2}$.
- angle: $\theta \in [0, 2\pi)$.
- θ can always be reduced modulo 2π.

Complex Number

$z = a + bi$ is plotted at position (a, b).

In its polar coordinates, denoted (r, θ), rewrite as

$$z = r(\cos\theta + i\sin\theta) = re^{i\theta}$$

- length: $r = \sqrt{a^2 + b^2}$.
- angle: $\theta \in [0, 2\pi)$.
- θ can always be reduced modulo 2π.

Basic arithmetic:

- $-z = (r, \theta + \pi)$.
- $(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$.
- If z is on the unit circle (i.e., $r = 1$), then $z^n = (1, n\theta)$.

The n-th Complex Roots of Unity

Solutions to the equation $z^n = 1$

- by the multiplication rules: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$.

The n-th Complex Roots of Unity

Solutions to the equation $z^n = 1$

- by the multiplication rules: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$.
- It can be represented as

$$1, \omega, \omega^2, \ldots, \omega^{n-1}$$

where

$$\omega = e^{2\pi i/n}$$

The n-th Complex Roots of Unity

Solutions to the equation $z^n = 1$

- by the multiplication rules: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$.
- It can be represented as

$$1, \omega, \omega^2, \ldots, \omega^{n-1}$$

where

$$\omega = e^{2\pi i/n}$$

For n is even:

- These numbers are plus-minus paired.
- Their squares are the $(n/2)$-nd roots of unity.

Complex Conjugate

The complex conjugate of a complex number $z = re^{i\theta}$ is $z^* = re^{-i\theta}$.

Complex Conjugate

The complex conjugate of a complex number $z = re^{i\theta}$ is $z^* = re^{-i\theta}$.

The complex conjugate of a vector (or a matrix) is obtained by taking the complex conjugates of all its entries.

Complex Conjugate

The complex conjugate of a complex number $z = re^{i\theta}$ is $z^* = re^{-i\theta}$.

The complex conjugate of a vector (or a matrix) is obtained by taking the complex conjugates of all its entries.

The angle between two vectors $u = (u_0, \ldots, u_{n-1})$ and $v(v_0, \ldots, v_{n-1})$ in $\mathbb{C}^n$ is just a scaling factor times their inner product

$$u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \ldots + u_{n-1} v_{n-1}^*$$

Complex Conjugate

The complex conjugate of a complex number $z = re^{i\theta}$ is $z^* = re^{-i\theta}$.

The complex conjugate of a vector (or a matrix) is obtained by taking the complex conjugates of all its entries.

The angle between two vectors $u = (u_0, \ldots, u_{n-1})$ and $v(v_0, \ldots, v_{n-1})$ in $\mathbb{C}^n$ is just a scaling factor times their inner product

$$u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \ldots + u_{n-1} v_{n-1}^*$$

The above quantity is maximized when the vectors lie in the same direction and is zero when the vectors are orthogonal to each other.

The Fast Fourier Transform

Polynomial multiplication

If $A(x) = a_0 + a_1x + \ldots + a_dx^d$ and $B(x) = b_0 + b_1x + \ldots + b_dx^d$, their product

$$C(x) = c_0 + c_1x + \ldots + c_{2d}x^{2d}$$

has coefficients

$$c_k = a_0b_k + a_1b_{k-1} + \ldots + a_kb_0 = \sum_{i=0}^{k} a_ib_{k-i}$$

where for $i > d$, take a_i and b_i to be zero.

Polynomial multiplication

If $A(x) = a_0 + a_1x + \ldots + a_dx^d$ and $B(x) = b_0 + b_1x + \ldots + b_dx^d$, their product

$$C(x) = c_0 + c_1x + \ldots + c_{2d}x^{2d}$$

has coefficients

$$c_k = a_0b_k + a_1b_{k-1} + \ldots + a_kb_0 = \sum_{i=0}^{k} a_ib_{k-i}$$

where for $i > d$, take a_i and b_i to be zero.

Computing c_k from this formula take $O(k)$ step, and finding all $2d+1$ coefficients would therefore seem to require $\Theta(d^2)$ time.

Polynomial multiplication

If $A(x) = a_0 + a_1 x + \ldots + a_d x^d$ and $B(x) = b_0 + b_1 x + \ldots + b_d x^d$, their product

$$C(x) = c_0 + c_1 x + \ldots + c_{2d} x^{2d}$$

has coefficients

$$c_k = a_0 b_k + a_1 b_{k-1} + \ldots + a_k b_0 = \sum_{i=0}^{k} a_i b_{k-i}$$

where for $i > d$, take a_i and b_i to be zero.

Computing c_k from this formula take $O(k)$ step, and finding all $2d + 1$ coefficients would therefore seem to require $\Theta(d^2)$ time.

Q: Can we do better?

An alternative representation

SHANGHAI JIAO TONG UNIVERSITY

Fact: A degree-d polynomial is uniquely characterized by its values at any $d+1$ distinct points.

An alternative representation

SHANGHAI JIAO TONG UNIVERSITY

Fact: A degree-d polynomial is uniquely characterized by its values at any $d+1$ distinct points.

We can specify a degree-d polynomial $A(x) = a_0 + a_1x + \ldots + a_dx^d$ by either of the following:

- Its coefficients $a_0, a_1, \ldots, a_d$. (coefficient representation).

An alternative representation

Fact: A degree-d polynomial is uniquely characterized by its values at any $d + 1$ distinct points.

We can specify a degree-d polynomial $A(x) = a_0 + a_1 x + \ldots + a_d x^d$ by either of the following:

- Its coefficients $a_0, a_1, \ldots, a_d$. (coefficient representation).
- The values $A(x_0), A(x_1), \ldots A(x_d)$ (value representation).

An alternative representation

Fact: A degree-d polynomial is uniquely characterized by its values at any $d+1$ distinct points.

We can specify a degree-d polynomial $A(x) = a_0 + a_1 x + \ldots + a_d x^d$ by either of the following:

- Its coefficients $a_0, a_1, \ldots, a_d$. (coefficient representation).
- The values $A(x_0), A(x_1), \ldots A(x_d)$ (value representation).

$$\text{coefficient representation} \underset{\text{interpolation}}{\overset{\text{evaluation}}{\rightleftarrows}} \text{value representation}$$

An alternative representation

coefficient representation —evaluation→ value representation

value representation —interpolation→ coefficient representation

The product $C(x)$ has degree $2d$, it is determined by its value at any $2d+1$ points.

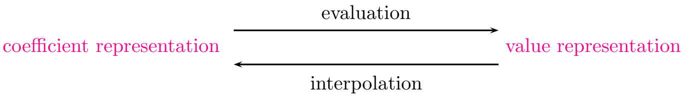
An alternative representation

$$\text{coefficient representation} \underset{\text{interpolation}}{\overset{\text{evaluation}}{\rightleftarrows}} \text{value representation}$$

The product $C(x)$ has degree $2d$, it is determined by its value at any $2d+1$ points.

Its value at any given point z is just $A(z)$ times $B(z)$.

An alternative representation



The product $C(x)$ has degree $2d$, it is determined by its value at any $2d + 1$ points.

Its value at any given point z is just $A(z)$ times $B(z)$.

Therefore, polynomial multiplication takes linear time in the value representation.

The algorithm

Input: Coefficients of two polynomials, $A(x)$ and $B(x)$, of degree d
Output: Their product $C = A \cdot B$

The algorithm

Input: Coefficients of two polynomials, $A(x)$ and $B(x)$, of degree d
Output: Their product $C = A \cdot B$

Selection
Pick some points $x_0, x_1, \ldots, x_{n-1}$, where $n \geq 2d + 1$.

The algorithm

Input: Coefficients of two polynomials, $A(x)$ and $B(x)$, of degree d
Output: Their product $C = A \cdot B$

Selection
Pick some points $x_0, x_1, \ldots, x_{n-1}$, where $n \geq 2d + 1$.

Evaluation
Compute $A(x_0), A(x_1), \ldots, A(x_{n-1})$ and $B(x_0), B(x_1), \ldots, B(x_{n-1})$.

The algorithm

Input: Coefficients of two polynomials, $A(x)$ and $B(x)$, of degree d
Output: Their product $C = A \cdot B$

Selection
Pick some points $x_0, x_1, \dots, x_{n-1}$, where $n \geq 2d + 1$.

Evaluation
Compute $A(x_0), A(x_1), \dots, A(x_{n-1})$ and $B(x_0), B(x_1), \dots, B(x_{n-1})$.

Multiplication
Compute $C(x_k) = A(x_k)B(x_k)$ for all $k = 0, \dots, n-1$.

The algorithm

Input: Coefficients of two polynomials, $A(x)$ and $B(x)$, of degree d
Output: Their product $C = A \cdot B$

Selection
Pick some points $x_0, x_1, \ldots, x_{n-1}$, where $n \geq 2d + 1$.

Evaluation
Compute $A(x_0), A(x_1), \ldots, A(x_{n-1})$ and $B(x_0), B(x_1), \ldots, B(x_{n-1})$.

Multiplication
Compute $C(x_k) = A(x_k)B(x_k)$ for all $k = 0, \ldots, n - 1$.

Interpolation
Recover $C(x) = c_0 + c_1 x + \ldots + c_{2d} x^{2d}$

Fast Fourier Transform

The selection step and the multiplications are just linear time:

Fast Fourier Transform

The selection step and the multiplications are just linear time:

- In a typical setting for polynomial multiplication, the coefficients of the polynomials are real number.

Fast Fourier Transform

The selection step and the multiplications are just linear time:

- In a typical setting for polynomial multiplication, the coefficients of the polynomials are real number.
- Moreover, are small enough that basic arithmetic operations take unit time.

Fast Fourier Transform

SHANGHAI JIAO TONG UNIVERSITY

The selection step and the multiplications are just linear time:

- In a typical setting for polynomial multiplication, the coefficients of the polynomials are real number.
- Moreover, are small enough that basic arithmetic operations take unit time.

Evaluating a polynomial of degree $d \leq n$ at a single point takes $O(n)$, and so the baseline for n points is $\Theta(n^2)$.

The selection step and the multiplications are just linear time:

- In a typical setting for polynomial multiplication, the coefficients of the polynomials are real number.
- Moreover, are small enough that basic arithmetic operations take unit time.

Evaluating a polynomial of degree $d \leq n$ at a single point takes $O(n)$, and so the baseline for n points is $\Theta(n^2)$.

The Fast Fourier Transform (FFT) does it in just $O(n \log n)$ time, for a particularly clever choice of $x_0, \ldots, x_{n-1}$.

Evaluation by divide-and-conquer

Q: How to make it efficient?

Evaluation by divide-and-conquer

Q: How to make it efficient?

First idea, we pick the n points,

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$$

then the computations required for each $A(x_i)$ and $A(-x_i)$ overlap a lot, because the even power of x_i coincide with those of $-x_i$.

Evaluation by divide-and-conquer

Q: How to make it efficient?

First idea, we pick the n points,

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$$

then the computations required for each $A(x_i)$ and $A(-x_i)$ overlap a lot, because the even power of x_i coincide with those of $-x_i$.

We need to split $A(x)$ into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)$$

Evaluation by divide-and-conquer

Q: How to make it efficient?

First idea, we pick the n points,

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$$

then the computations required for each $A(x_i)$ and $A(-x_i)$ overlap a lot, because the even power of x_i coincide with those of $-x_i$.

We need to split $A(x)$ into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)$$

More generally

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

where $A_e(\cdot)$, with the even-numbered coefficients, and $A_o(\cdot)$, with the odd-numbered coefficients, are polynomials of degree $\leq n/2 - 1$.

Evaluation by divide-and-conquer

Given paired points $\pm x_i$, the calculations needed for $A(x_i)$ can be recycled toward computing $A(-x_i)$:

$$\begin{aligned} A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2) \end{aligned}$$

Evaluation by divide-and-conquer

Given paired points $\pm x_i$, the calculations needed for $A(x_i)$ can be recycled toward computing $A(-x_i)$:

$$\begin{aligned} A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2) \end{aligned}$$

Evaluating $A(x)$ at n paired points $\pm x_0, \ldots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ at just $n/2$ points, $x_0^2, \ldots, x_{n/2-1}^2$.

Evaluation by divide-and-conquer

Shanghai Jiao Tong University

Given paired points $\pm x_i$, the calculations needed for $A(x_i)$ can be recycled toward computing $A(-x_i)$:

$$\begin{aligned} A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2) \end{aligned}$$

Evaluating $A(x)$ at n paired points $\pm x_0, \ldots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ at just $n/2$ points, $x_0^2, \ldots, x_{n/2-1}^2$.

If we could recurse, we would get a divide-and-conquer procedure with running time

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

How to choose n points?

Aim: To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be themselves plus-minus pairs.

How to choose n points?

Aim: To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be themselves plus-minus pairs.

Q: How can a square be negative?

How to choose n points?

Aim: To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be themselves plus-minus pairs.

Q: How can a square be negative?

- We use complex numbers.

How to choose n points?

Aim: To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be themselves plus-minus pairs.

Q: How can a square be negative?

- We use complex numbers.

At the very bottom of the recursion, we have a single point, 1, in which case the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$.

How to choose n points?

Aim: To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be themselves plus-minus pairs.

Q: How can a square be negative?

- We use complex numbers.

At the very bottom of the recursion, we have a single point, 1, in which case the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$.

The next level up then has $\pm\sqrt{+1} = \pm 1$, as well as the complex numbers $\pm\sqrt{-1} = \pm i$.

How to choose n points?

Aim: To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \dots, x_{n/2-1}^2$ to be themselves plus-minus pairs.

Q: How can a square be negative?

- We use complex numbers.

At the very bottom of the recursion, we have a single point, 1, in which case the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$.

The next level up then has $\pm\sqrt{+1} = \pm 1$, as well as the complex numbers $\pm\sqrt{-1} = \pm i$.

By continuing in this manner, we eventually reach the initial set of n points: the complex $n\,th$ roots of unity, that is the n complex solutions of the equation

$$z^n = 1$$

The n-th complex roots of unity

Shanghai Jiao Tong University

Solutions to the equation $z^n = 1$

- by the multiplication rules: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$.
- It can be represented as

$$1, \omega, \omega^2, \ldots, \omega^{n-1}$$

where

$$\omega = e^{2\pi i/n}$$

The n-th complex roots of unity

Solutions to the equation $z^n = 1$

- by the multiplication rules: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$.
- It can be represented as

$$1, \omega, \omega^2, \ldots, \omega^{n-1}$$

where

$$\omega = e^{2\pi i/n}$$

For n is even:

- These numbers are plus-minus paired.
- Their squares are the $(n/2)$-nd roots of unity.

The FFT algorithm


```
FFT(A, ω)
```

input : coefficient reprentation of a polynomial $A(x)$ of degree $\leq n-1$, where n is a power of 2;
ω, an n-th root of unity

output: value representation $A(\omega^0), \ldots, A(\omega^{n-1})$

if $\omega = 1$ **then** `return` $A(1)$;
express $A(x)$ in the form $A_e(x^2) + xA_o(x^2)$;
call `FFT(`A_e,ω^2`)` to evaluate A_e at even powers of ω;
call `FFT(`A_o,ω^2`)` to evaluate A_o at even powers of ω;
for $j = 0$ *to* $n-1$ **do**
  compute $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$;
end
`return(`$A(\omega^0), \ldots, A(\omega^{n-1})$`);`

Interpolation

$\texttt{FFT}$ moves from coefficients to values in time just $O(n \log n)$, when the points $\{x_i\}$ are complex n-th roots of unity $(1, \omega, \omega^2, \ldots, \omega^{n-1})$.

Interpolation

FFT moves from coefficients to values in time just $O(n \log n)$, when the points $\{x_i\}$ are complex n-th roots of unity $(1, \omega, \omega^2, \ldots, \omega^{n-1})$.

That is,

$$\langle value \rangle = \texttt{FFT}(\langle coefficients \rangle, \omega)$$

Interpolation

FFT moves from coefficients to values in time just $O(n \log n)$, when the points $\{x_i\}$ are complex n-th roots of unity $(1, \omega, \omega^2, \ldots, \omega^{n-1})$.

That is,

$$\langle value \rangle = \texttt{FFT}(\langle coefficients \rangle, \omega)$$

We will see that the interpolation can be computed by

$$\langle coefficients \rangle = \frac{1}{n}\texttt{FFT}(\langle values \rangle, \omega^{-1})$$

A matrix reformation

Let's explicitly set down the relationship between our two representations for a polynomial $A(x)$ of degree $\leq n-1$.

A matrix reformation

Let's explicitly set down the relationship between our two representations for a polynomial $A(x)$ of degree $\leq n-1$.

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

A matrix reformation

Let's explicitly set down the relationship between our two representations for a polynomial $A(x)$ of degree $\leq n-1$.

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Let M be the matrix in the middle, which is a Vandermonde matrix.

A matrix reformation

Let's explicitly set down the relationship between our two representations for a polynomial $A(x)$ of degree $\leq n-1$.

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Let M be the matrix in the middle, which is a Vandermonde matrix.

- If $x_0, x_1, \dots, x_{n-1}$ are distinct numbers, then M is invertible.

A matrix reformation

Let's explicitly set down the relationship between our two representations for a polynomial $A(x)$ of degree $\leq n-1$.

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Let M be the matrix in the middle, which is a Vandermonde matrix.

- If $x_0, x_1, \dots, x_{n-1}$ are distinct numbers, then M is invertible.
- evaluation is multiplication by M, while interpolation is multiplication by M^{-1}.

A matrix reformation

This reformulation of our polynomial operations reveals their essential nature more clearly.

A matrix reformation

This reformulation of our polynomial operations reveals their essential nature more clearly.

It justifies an assumption that $A(x)$ is uniquely characterized by its values at any n points.

A matrix reformation

This reformulation of our polynomial operations reveals their essential nature more clearly.

It justifies an assumption that $A(x)$ is uniquely characterized by its values at any n points.

Vandermonde matrices also have the distinction of being quicker to invert than more general matrices, in $O(n^2)$ time instead of $O(n^3)$.

A matrix reformation

This reformulation of our polynomial operations reveals their essential nature more clearly.

It justifies an assumption that $A(x)$ is uniquely characterized by its values at any n points.

Vandermonde matrices also have the distinction of being quicker to invert than more general matrices, in $O(n^2)$ time instead of $O(n^3)$.

However, using this for interpolation would still not be fast enough for us..

Interpolation resolved

In linear algebra terms, the `FFT` multiplies an arbitrary n-dimensional vector, which we have been calling the coefficient representation, by the $n \times n$ matrix.

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \dots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & x^{(n-1)(n-1)} \end{bmatrix}$$

Its (j, k)-th entry (starting row- and column-count at zero) is ω^{jk}

Interpolation resolved

The columns of M are orthogonal to each other, which is often called the Fourier basis.

Interpolation resolved

The columns of M are orthogonal to each other, which is often called the Fourier basis.

The FFT is thus a change of basis, a rigid rotation. The inverse of M is the opposite rotation, from the Fourier basis back into the standard basis.

Interpolation resolved

The columns of M are orthogonal to each other, which is often called the Fourier basis.

The FFT is thus a change of basis, a rigid rotation. The inverse of M is the opposite rotation, from the Fourier basis back into the standard basis.

Inversion formula

$$M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$$

Interpolation resolved

Take ω to be $e^{2\pi i/n}$, and think of M as vectors in $\mathbb{C}^n$.

Interpolation resolved

Take ω to be $e^{2\pi i/n}$, and think of M as vectors in $\mathbb{C}^n$.

Recall that the angle between two vectors $u = (u_0, \ldots, u_{n-1})$ and $v(v_0, \ldots, v_{n-1})$ in $\mathbb{C}^n$ is just a scaling factor times their inner product

$$u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \ldots + u_{n-1} v_{n-1}^*$$

where z^* denotes the complex conjugate of z.

Interpolation resolved

Take ω to be $e^{2\pi i/n}$, and think of M as vectors in $\mathbb{C}^n$.

Recall that the angle between two vectors $u = (u_0, \ldots, u_{n-1})$ and $v(v_0, \ldots, v_{n-1})$ in $\mathbb{C}^n$ is just a scaling factor times their inner product

$$u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \ldots + u_{n-1} v_{n-1}^*$$

where z^* denotes the complex conjugate of z.

The above quantity is maximized when the vectors lie in the same direction and is zero when the vectors are orthogonal to each other.

Interpolation resolved

Lemma

The columns of matrix M are orthogonal to each other.

Interpolation resolved

Lemma

The columns of matrix M are orthogonal to each other.

Proof.

Interpolation resolved

Lemma

The columns of matrix M are orthogonal to each other.

Proof.

- Take the inner product of of any columns j and k of matrix M,

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \ldots + \omega^{(n-1)(j-k)}$$

This is a geometric series with first term 1, last term $\omega^{(n-1)(j-k)}$, and ratio ω^{j-k}.

Interpolation resolved

Lemma

The columns of matrix M are orthogonal to each other.

Proof.

- Take the inner product of of any columns j and k of matrix M,

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \ldots + \omega^{(n-1)(j-k)}$$

  This is a geometric series with first term 1, last term $\omega^{(n-1)(j-k)}$, and ratio ω^{j-k}.
- Therefore, if $j \neq k$, it evaluates to

$$\frac{1 - \omega^{n(j-k)}}{1 - \omega^{(j-k)}} = 0$$

Interpolation resolved

Lemma

The columns of matrix M are orthogonal to each other.

Proof.

- Take the inner product of of any columns j and k of matrix M,

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \ldots + \omega^{(n-1)(j-k)}$$

  This is a geometric series with first term 1, last term $\omega^{(n-1)(j-k)}$, and ratio ω^{j-k}.

- Therefore, if $j \neq k$, it evaluates to

$$\frac{1 - \omega^{n(j-k)}}{1 - \omega^{(j-k)}} = 0$$

- If $j = k$, then it evaluates to n.

Interpolation resolved

SHANGHAI JIAO TONG UNIVERSITY

Corollary

$MM^* = nI$, *i.e.*,

$$M_n^{-1} = \frac{1}{n} M_n^*$$

The definitive FFT algorithm

The FFT takes as input a vector $a = (a_0, \ldots, a_{n-1})$ and a complex number ω whose powers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are the complex n-th roots of unity.

The definitive FFT algorithm

The FFT takes as input a vector $a = (a_0, \ldots, a_{n-1})$ and a complex number ω whose powers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are the complex n-th roots of unity.

It multiplies vector a by the $n \times n$ matrix $M_n(\omega)$, which has (j, k)-th entry ω^{jk}.

The definitive FFT algorithm

The FFT takes as input a vector $a = (a_0, \ldots, a_{n-1})$ and a complex number ω whose powers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are the complex n-th roots of unity.

It multiplies vector a by the $n \times n$ matrix $M_n(\omega)$, which has (j, k)-th entry ω^{jk}.

The potential for using divide-and-conquer in this matrix-vector multiplication becomes apparent when M's columns are segregated into evens and odds.

The definitive FFT algorithm

The FFT takes as input a vector $a = (a_0, \ldots, a_{n-1})$ and a complex number ω whose powers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are the complex n-th roots of unity.

It multiplies vector a by the $n \times n$ matrix $M_n(\omega)$, which has (j, k)-th entry ω^{jk}.

The potential for using divide-and-conquer in this matrix-vector multiplication becomes apparent when M's columns are segregated into evens and odds.

The product of $M_n(\omega)$ with vector $a = (a_0, \ldots, a_{n-1})$, a size-$n$ problem, can be expressed in terms of two size-$n/2$ problems: the product of $M_{n/2}(\omega^2)$ with $(a_0, a_2, \ldots, a_{n-2})$ and with $(a_1, a_3, \ldots, a_{n-1})$.

The definitive FFT algorithm

The FFT takes as input a vector $a = (a_0, \ldots, a_{n-1})$ and a complex number ω whose powers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are the complex n-th roots of unity.

It multiplies vector a by the $n \times n$ matrix $M_n(\omega)$, which has (j, k)-th entry ω^{jk}.

The potential for using divide-and-conquer in this matrix-vector multiplication becomes apparent when M's columns are segregated into evens and odds.

The product of $M_n(\omega)$ with vector $a = (a_0, \ldots, a_{n-1})$, a size-$n$ problem, can be expressed in terms of two size-$n/2$ problems: the product of $M_{n/2}(\omega^2)$ with $(a_0, a_2, \ldots, a_{n-2})$ and with $(a_1, a_3, \ldots, a_{n-1})$.

This divide-and-conquer strategy leads to the definitive FFT algorithm, whose running time is $T(n) = 2T(n/2) + O(n) = O(nlogn)$.

The general FFT algorithm

`FFT` (a, ω)
input : An array $a = (a_0, a_1, \ldots, a_{n-1})$ for n is a power of 2; ω, an n-th root of unity
output: $M_n(\omega)a$

if $\omega = 1$ **then** `return` a;
$(s_0, s_1, \ldots, s_{n/2-1})$=`FFT` $((a_0, a_2, \ldots, a_{n-2}), \omega^2)$;
$(s'_0, s'_1, \ldots, s'_{n/2-1})$=`FFT` $((a_1, a_3, \ldots, a_{n-1}), \omega^2)$;
for $j = 0$ *to* $n/2 - 1$ **do**
    r_j=$s_j + \omega^j s'_j$;
    $r_{j+n/2}$=$s_j - \omega^j s'_j$;
end
`return` $(r_0, r_1, \ldots, r_{n-1})$;

Top 10 algorithms of the 20th century

Top 10 algorithms of the 20th century

1946: The Metropolis Algorithm

1947: Simplex Method

1950: Krylov Subspace Method

1951: The Decompositional Approach to Matrix Computations

1957: The Fortran Optimizing Compiler

1959: QR Algorithm

1962: Quicksort

1965: Fast Fourier Transform

1977: Integer Relation Detection

1987: Fast Multipole Method

Homework

Homework

- Assignment 2 (1 week). Exercises 2.13, 2.19, 2.22, and 2.28.