CSE515 Advanced Algorithms
Lecture 25
Fast Fourier Transform I

Antoine Vigneron

Ulsan National Institute of Science and Technology

July 27, 2021

Introduction

References:

- Section 2.6 of Algorithms by Dasgupta, Papadimitriou and Vazirani.
- Chapter 30 of Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein. (Available online from the UNIST library website.)

Polynomial Multiplication

- Consider the degree-2 polynomials $1 + 2x + 3x^2$ and $2 + x + 4x^2$.
- Their product is the polynomial

$$(1 + 2x + 3x^2) \cdot (2 + x + 4x^2) = 2 + 5x + 12x^2 + 11x^3 + 12x^4.$$

- More generally, if $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ and $B(x) = b_0 + b_1 x + \cdots + b_d x^d$ are degree-d polynomials, then their products is the degree-$2d$ polynomial $C(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d}$ with coefficients

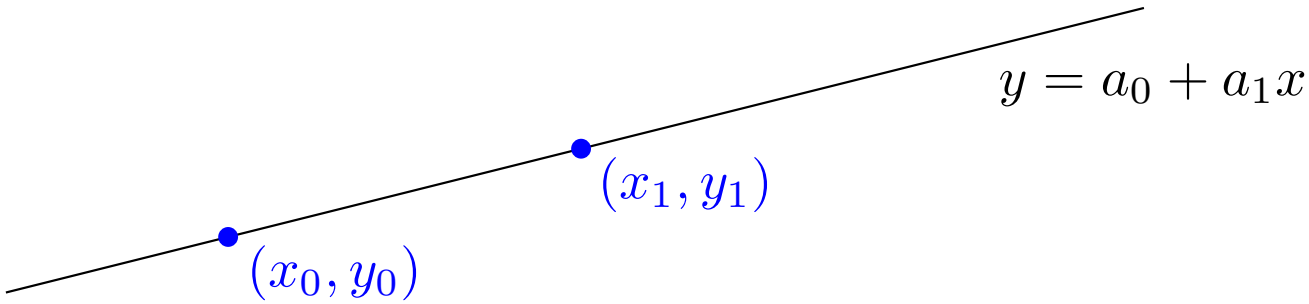
$$c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$

(We take $a_i = b_i = 0$ when $i > d$.)

Polynomial Multiplication

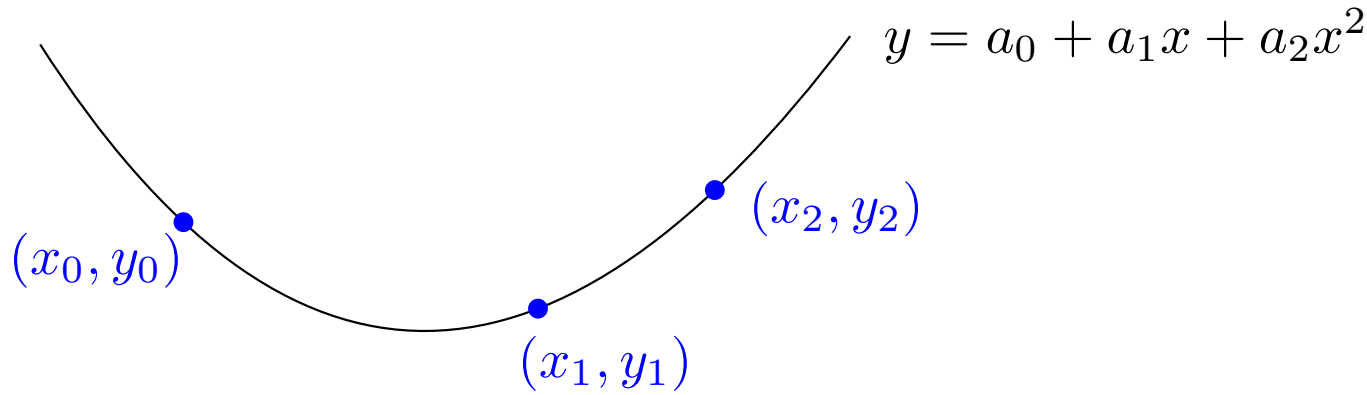
- The *polynomial multiplication problem* is to compute the polynomial $C(x) = A(x) \cdot B(x)$, that is, to compute all its coefficients c_0, c_1, ..., c_{2d}.
- How fast can it be done?
- Using the formula above, each coefficient is computed in $O(d)$ time, so it takes in total $O(d^2)$ time.
- In the coming lectures, we will see how to do it in time $O(d \log d)$ time using the *fast Fourier transform* (FFT).
- The FFT has other applications.

Representation of Polynomials



- Given any two points (x_0, y_0) and (x_1, y_1) such that $x_0 \neq x_1$, there is a unique line through these two points.
- In other words, there is a unique degree-1 polynomial $A(x) = a_0 + a_1 x$ such that $A(x_0) = y_0$ and $A(x_1) = y_1$.
- So if I choose x_0 and y_1, then $A(x_0)$ and $A(x_1)$ uniquely determine the polynomial $A(x)$.
- Hence $A(x)$ can be represented by $A(x_0)$ and $A(x_1)$, instead of its coefficients a_0 and a_1.

Representation of Polynomials



- Similarly, a parabola is uniquely defined by 3 points with different x-coordinates.
- So a degree 2 polynomial $A(x) = a_0 + a_1 x + a_2 x^2$ can either be represented by:
  - Its three coefficients a_0, a_1, a_2.
  - or its values $y_0 = A(x_0)$, $y_1 = A(x_1)$ and $y_2 = A(x_2)$ where x_0, x_1 and x_2 are distinct.

Representation of Polynomials

- More generally:

Proposition

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

- After we fix $d+1$ distinct points $x_0, \ldots, x_d$, we can specify a degree-d polynomial $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ in two different ways:
  - (1) using its coefficients $a_0, a_1, \ldots a_d$,
  - (2) or the values $A(x_0), A(x_1), \ldots, A(x_d)$.

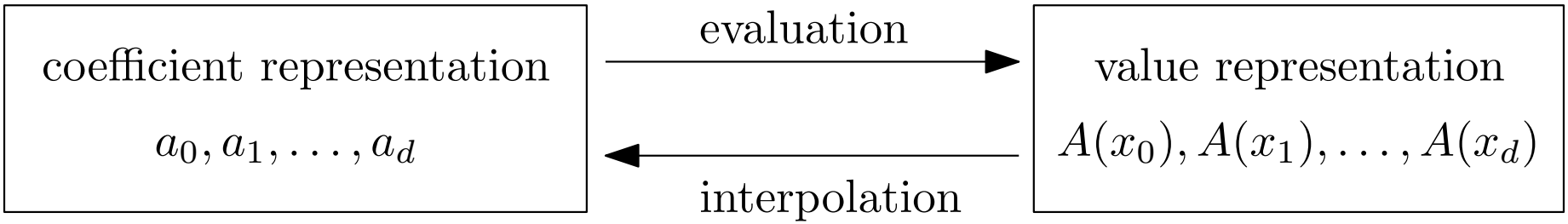
Representation of Polynomials

Example

Suppose $x_0 = 0$, $x_1 = 1$ and $x_2 = 3$. Let $A(x) = 3 - 2x + x^2$. Since $A(x_0) = 3$, $A(x_1) = 2$ and $A(x_2) = 6$, the two representations are:

- coefficients $(3, -2, 1)$, or
- values: $(3, 2, 6)$

Representation of Polynomials



- Polynomial *evaluation* converts from coefficient representation to value representation.
- Polynomial *interpolation* converts from value representation to coefficient representation.

Representation of Polynomials

Polynomial Multiplication

INPUT: coefficients of two polynomials $A(x)$ and $B(x)$ of degree $\leqslant d$
OUTPUT: coefficients of the product $C = A \cdot B$

Selection: pick distinct points $x_0, \ldots, x_{n-1}$ with $n \geqslant 2d + 1$

Evaluation: compute $A(x_0), \ldots, A(x_{n-1}), B(x_0), \ldots, B(x_{n-1})$

Multiplication: compute $C(x_k) = A(x_k) \cdot B(x_k)$ for $k = 0, \ldots, n-1$

Interpolation: recover the coefficients $c_0, \ldots, c_{2d}$ of $C(x)$

- The multiplication step can be performed in time $O(n)$.
- This is the point of using the representation by values: With the representation by coefficients, the naive algorithm takes quadratic time.
- FFT will allow us to perform evaluation and interpolation in $O(n \log n)$ time, hence the whole algorithm will take $O(n \log n)$ time.

Evaluation by Divide and Conquer

- Suppose we select the points $\pm x_0$, $\pm x_1$, $\ldots$, $\pm x_{n/2-1}$.
- We can write $A(x) = A_e(x^2) + xA_o(x^2)$.

Example

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

$$A(x) = \left(3 + 6x^2 + x^4\right) + x\left(4 + 2x^2 + 10x^4\right)$$

so $A_e(x) = 3 + 6x + x^2$ and $A_o(x) = 4 + 2x + 10x^2$.

- Then the degrees of A_o and A_e are $\leqslant n/2 - 1$.
- So our divide and conquer approach would recurse on A_o and A_e, whose degrees are at most half the degree of A.

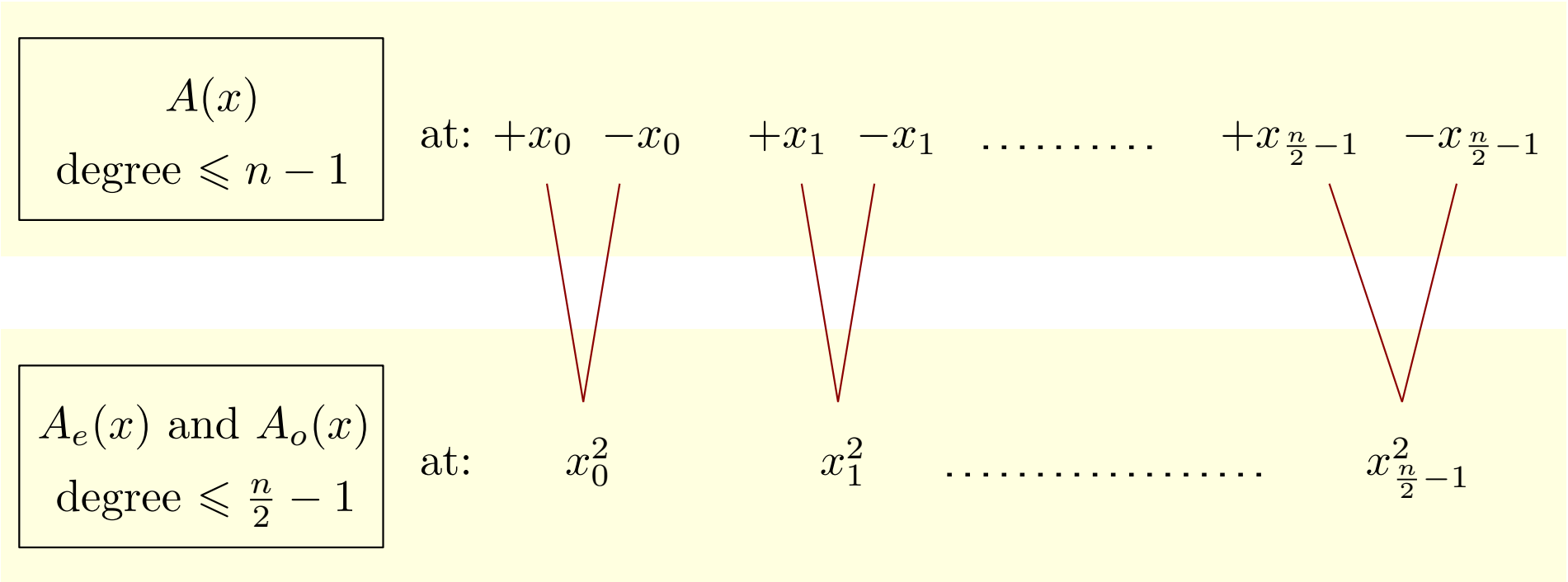
Evaluation by Divide and Conquer

- Observation:

$$A(x_i) = A_e(x_i^2) + x_i A_o(x_i^2)$$
$$A(-x_i) = A_e(x_i^2) - x_i A_o(x_i^2)$$

- So after evaluating $A_e(x_i^2)$ and $A_o(x_i^2)$, we get $A(x_i)$ and $A(-x_i)$ in *constant* time.

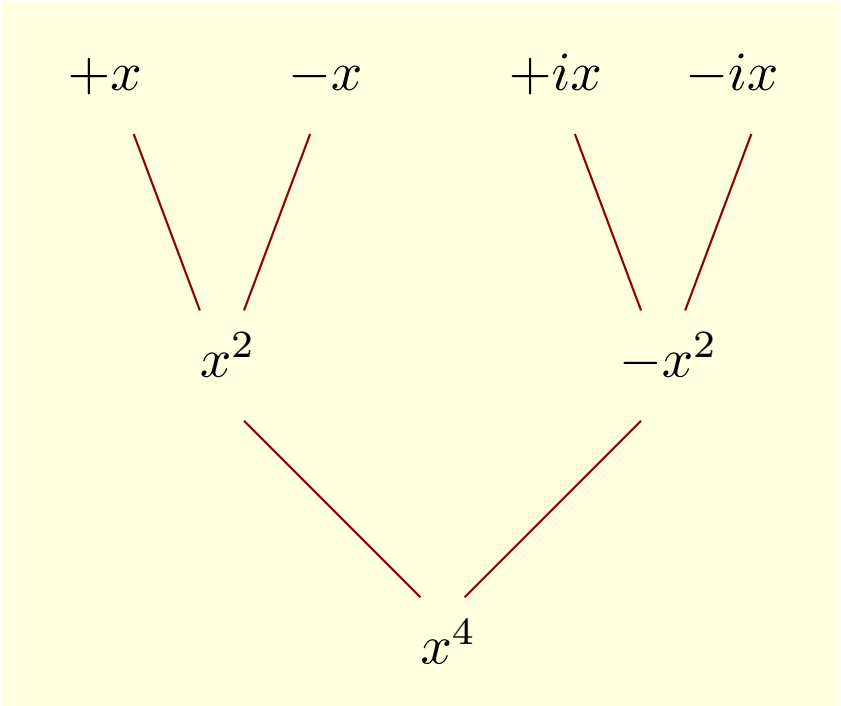
Evaluation by Divide and Conquer



- So we reduced the problem size by a factor 2: each pair of variables $\pm x_i$ has been reduced to one variable x_i^2, and the degree has been reduced by a factor at least 2.
- Problem: How can we go further?

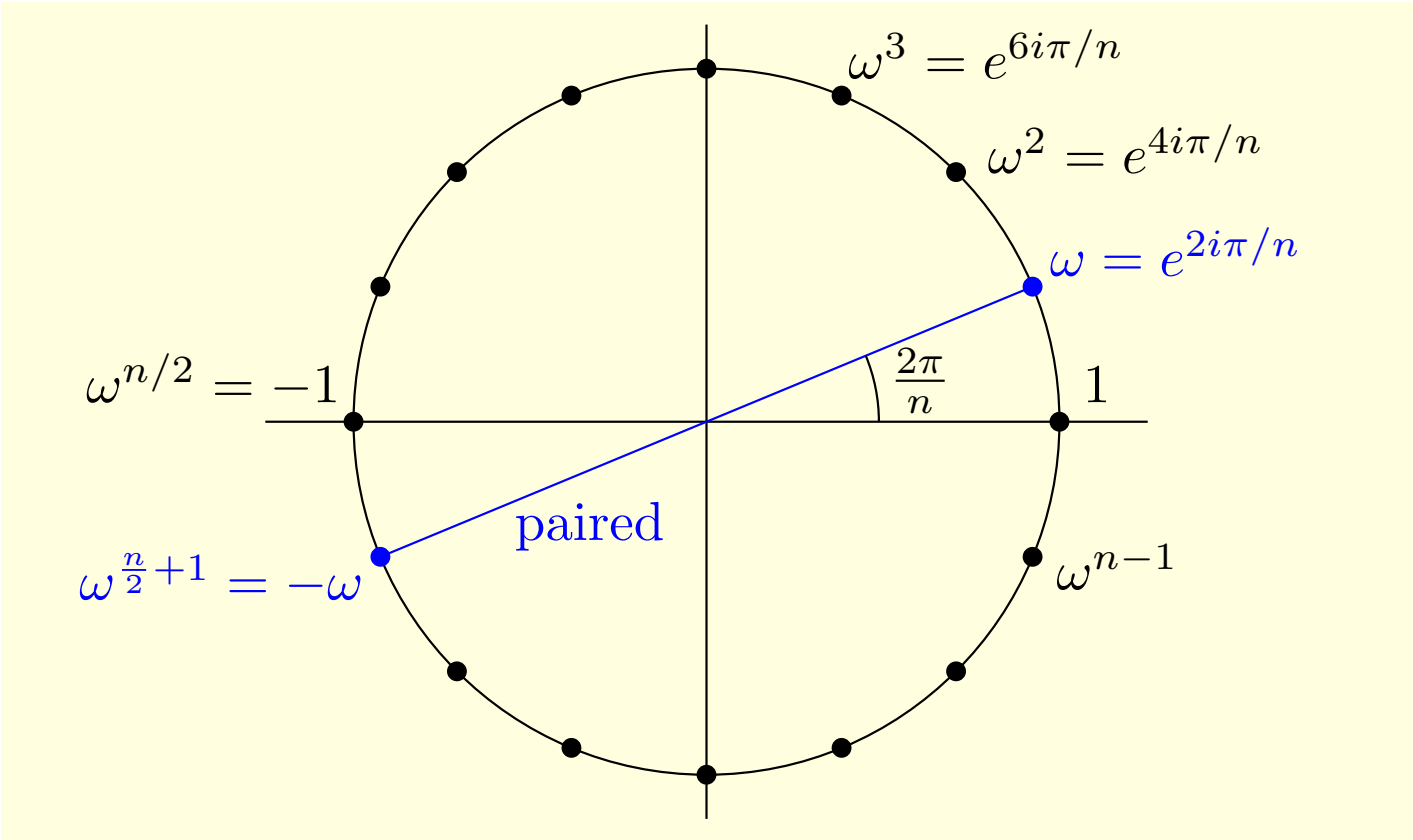
Evaluation by Divide and Conquer

- Idea: Use *complex* numbers.



- We multiply by 1, i, -1 and $-i$ instead of just 1 and -1.
- 1, i, -1 and $-i$ are the 4*th complex roots of unity*: they are the solutions of the equation $x^4 = 1$ in $\mathbb{C}$.

Complex *n*th Roots of Unity



- When n is even, the nth root of unity are as pictured above.

Complex nth Roots of Unity

- The complex nth root of unity are the solutions of the equation $x^n = 1$ in $\mathbb{C}$.
- They are of the form

$$e^{2ik\pi/n}$$

  where $k = 0, \ldots, n-1$.
- So if we let $\omega = e^{2i\pi/n}$, then they are

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

Complex nth Roots of Unity

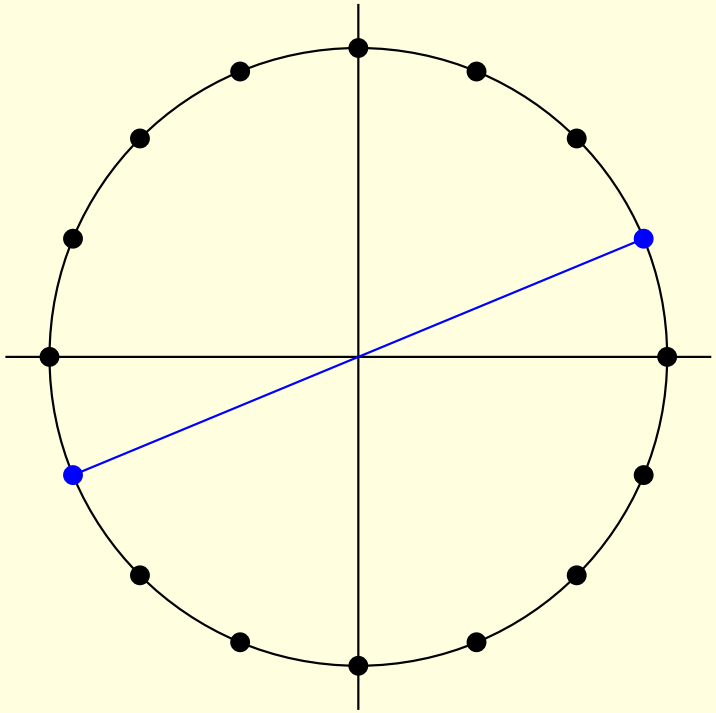
- Suppose n is even.
- Then the nth roots of unity are *paired*: for each nth root of unity ω^j, then its opposite $-\omega^j = \omega^{n/2+j}$ is also an nth root of unity.
- Also

$$1,\ \omega^2,\ \omega^4,\ \ldots \quad \text{and } \omega^{n-2}$$

  are the $(n/2)$th roots of unity.
- In other words, the $(n/2)$th roots of unity are the squares of the nth roots of unity.

Divide-and-Conquer Step

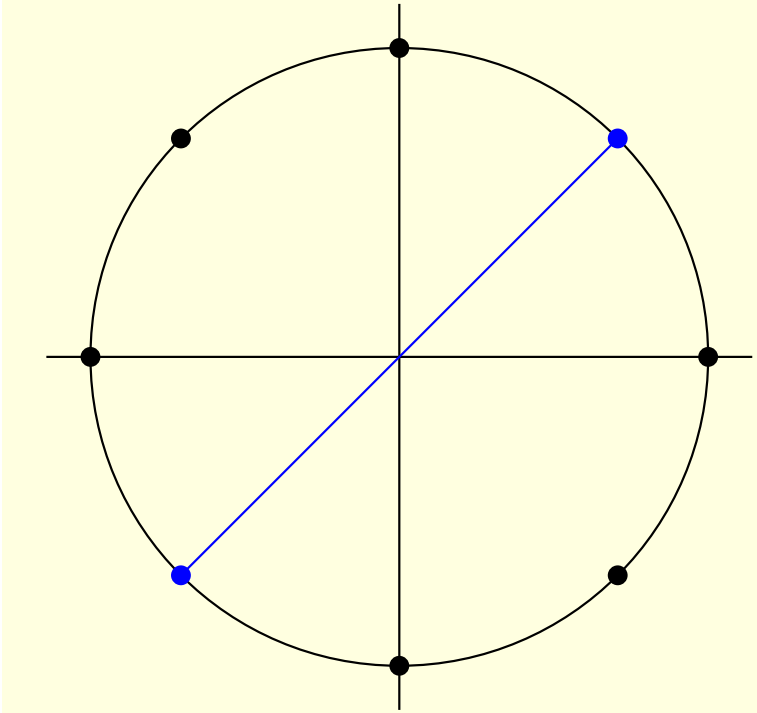
evaluate $A(x)$ at
nth roots of unity



n is a power of 2 $\Rightarrow$ paired

divide
and
conquer

evaluate $A_e(x)$ and $A_o(x)$ at
$(n/2)$th roots of unity



paired

Algorithm

- INPUT: coefficients of a polynomial $A(x)$ of degree $\leqslant n-1$, an nth root of unity ω, where n is a power of 2.

Pseudocode

1: **procedure** $\mathrm{FFT}(A, \omega)$
2: **if** $\omega = 1$ **then**
3: **return** $A(1)$
4: express $A(x)$ as $A_e(x^2) + xA_o(x^2)$
5: call $\mathrm{FFT}(A_e, \omega^2)$ to evaluate A_e at even powers of ω
6: call $\mathrm{FFT}(A_o, \omega^2)$ to evaluate A_o at even powers of ω
7: **for** $j \leftarrow 0, n-1$ **do**
8: compute $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$
9: **return** $A(\omega^0), A(\omega^1), \ldots, A(\omega^{n-1})$

Algorithm

Remark:

- The algorithm works because the roots are paired.
- In particular, since $\omega^{j+n/2} = -\omega^j$, we have

$$A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$$
$$A(\omega^{j+n/2}) = A_e(\omega^{2j}) - \omega^j A_o(\omega^{2j})$$

- This is why we can generate n values of A from $n/2$ pairs of value $A_e(\omega^{2j})$, $A_o(\omega^{2j})$.
- In other words, we can apply recursively the "trick" presented on Slide 14.

Analysis

- We recurse to two subproblems of size $n/2$, because A_e and A_o have degree twice smaller, and we use $(n/2)$th roots of unity.
- Ignoring recursive calls, FFT takes time $\Theta(n)$ due to the **for** loop.
- So the running time $T(n)$ satisfies the recurrence relation

$$T(n) = 2T(n/2) + \Theta(n).$$

- It is the same relation as for MERGE SORT, so it solves to

$$T(n) = \Theta(n \log n).$$

Concluding Remarks

- The FFT algorithm allows us to evaluate a polynomial of degree n at all the nth roots of unity in $O(n \log n)$ time in total, when n is a power of 2.
- It is non-trivial: a naive algorithm takes $O(n^2)$ time.
- We will see in next lecture that it also allows us to perform interpolation in $O(n \log n)$ time.
- So we will be able to multiply polynomials in $O(n \log n)$ time.