WEEK 3, LECTURE 5 ON 1 SEPTEMBER 2021 CS1.301.M21 ALGORITHM ANALYSIS AND DESIGN

FAST FOURIER TRANSFORM

The Problem: Given two d-degree polynomials, compute their product.

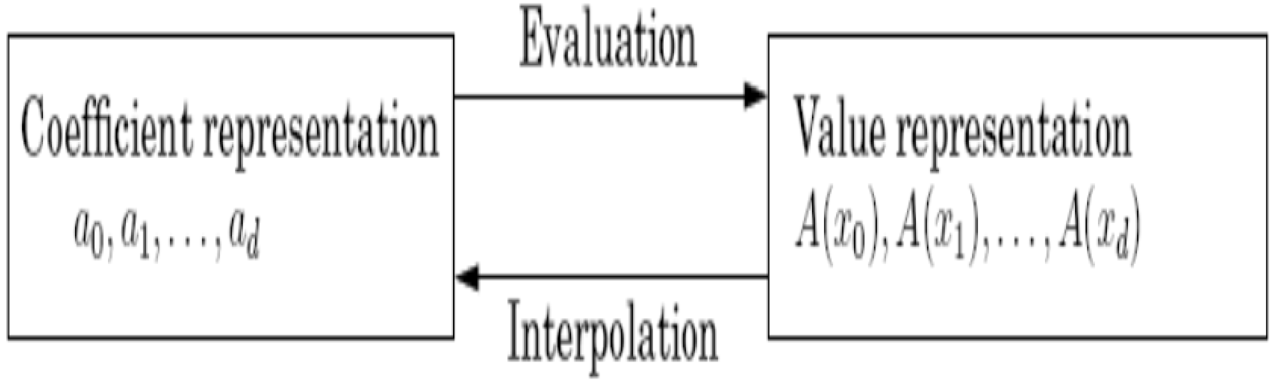
$$A(x) = a_0 + a_1 + \ldots + a_d x^d \times$$
$$B(x) = b_0 + b_1 + \ldots + b_d x^d$$

Naive Algorithm

Applying the formula directly will take $O(d^2)$

Can we do better?

There are other ways to represent polynomials:



Evaluating by divide and conquer

Divide the polynomial into parts that are odd and even coefficients

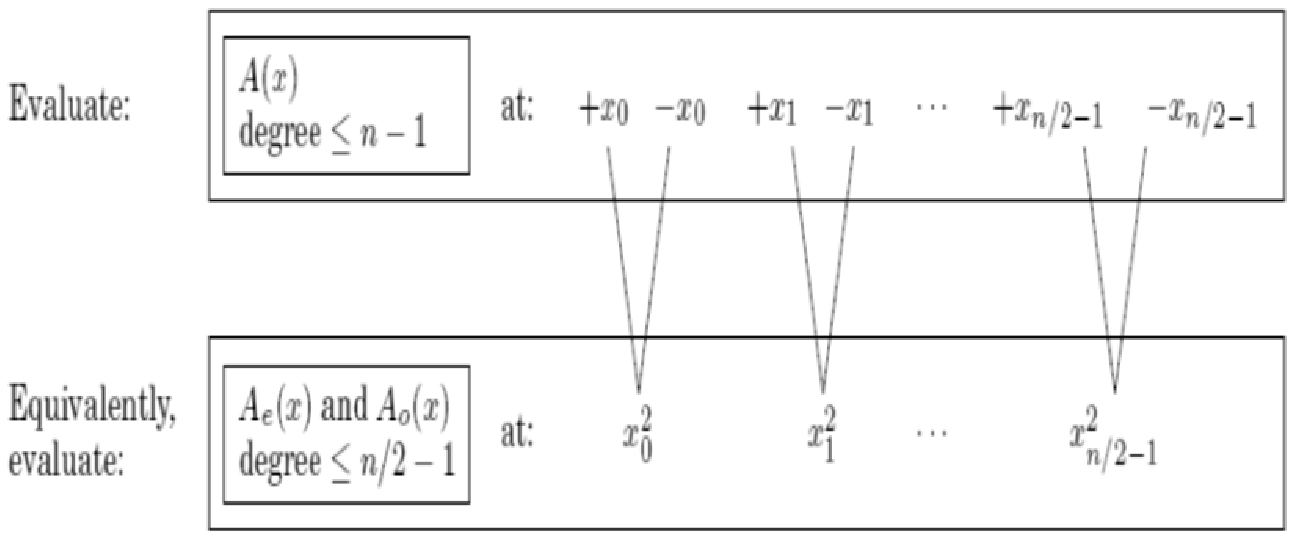
$$A(x) = A_e(x^2) + xA_o(x^2)$$

Example: $A(x) = 5 + 2x + 6x^2 + 3x^3 + 7x^4 + 8x^5$

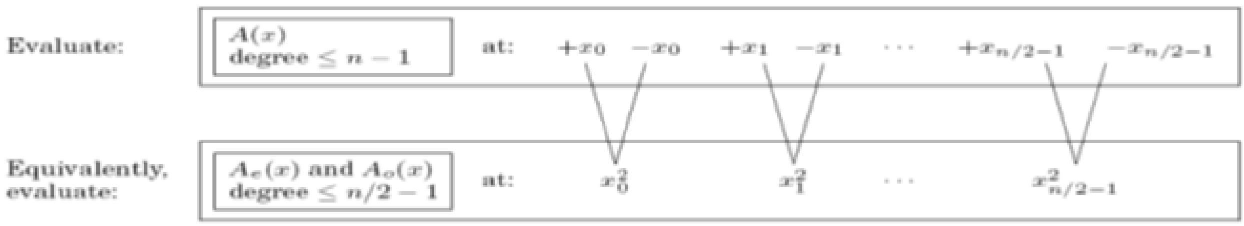
$A_e = (5 + 6x + 7x^2)$ and $A_0 = (2 + 3x + 8x^2)$

We can evaluate as two points namely x_i and $-x_i$.

Since $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ aren't plus minus pairs. So it only works at the first level of recursion.



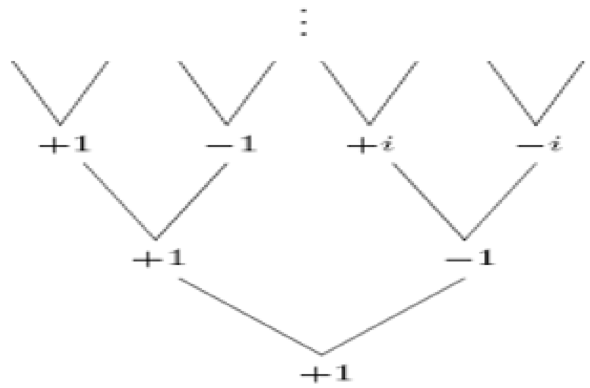
We need a square to be negative so we use complex numbers.



the complex *nth roots of unity*

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

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



Finally we arrive at the Fast Fourier Transform algorithm

<u>function FFT</u>(a, ω)
Input: An array $a = (a_0, a_1, \ldots, a_{n-1})$, for n a power of 2
A primitive nth root of unity, ω
Output: $M_n(\omega)\, a$

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