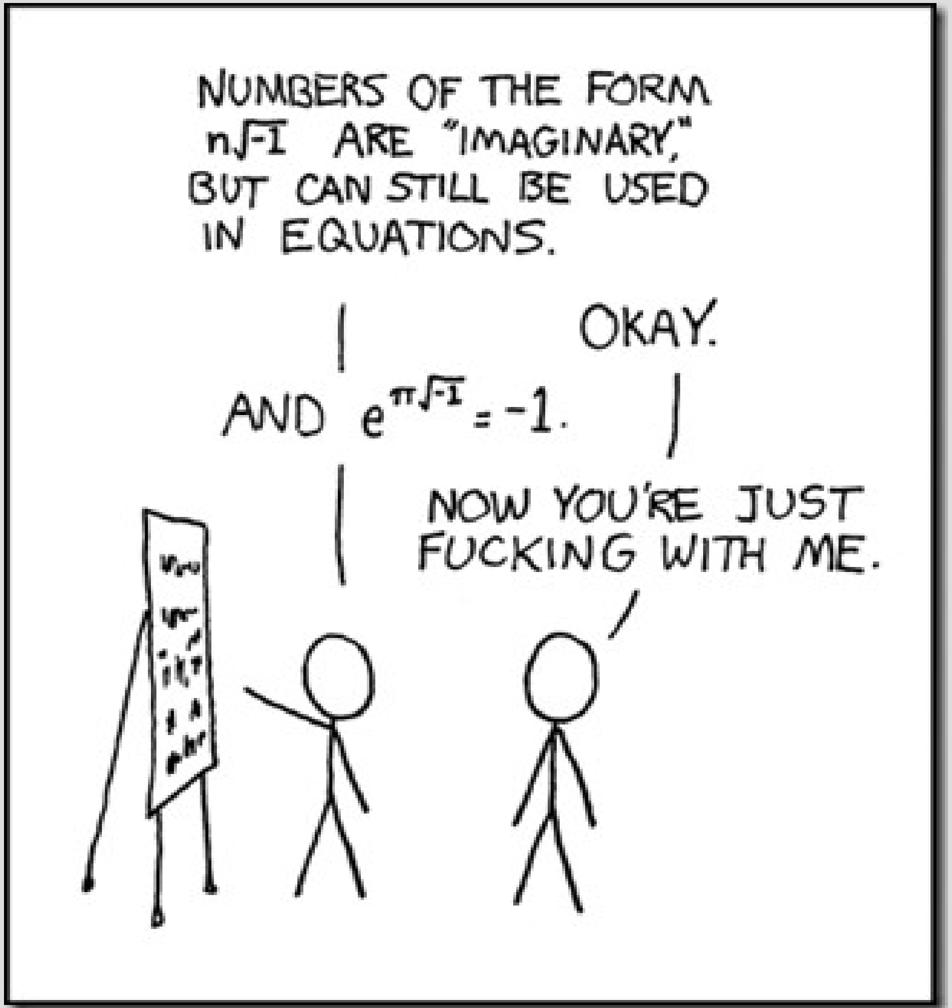
Fast Fourier Transform

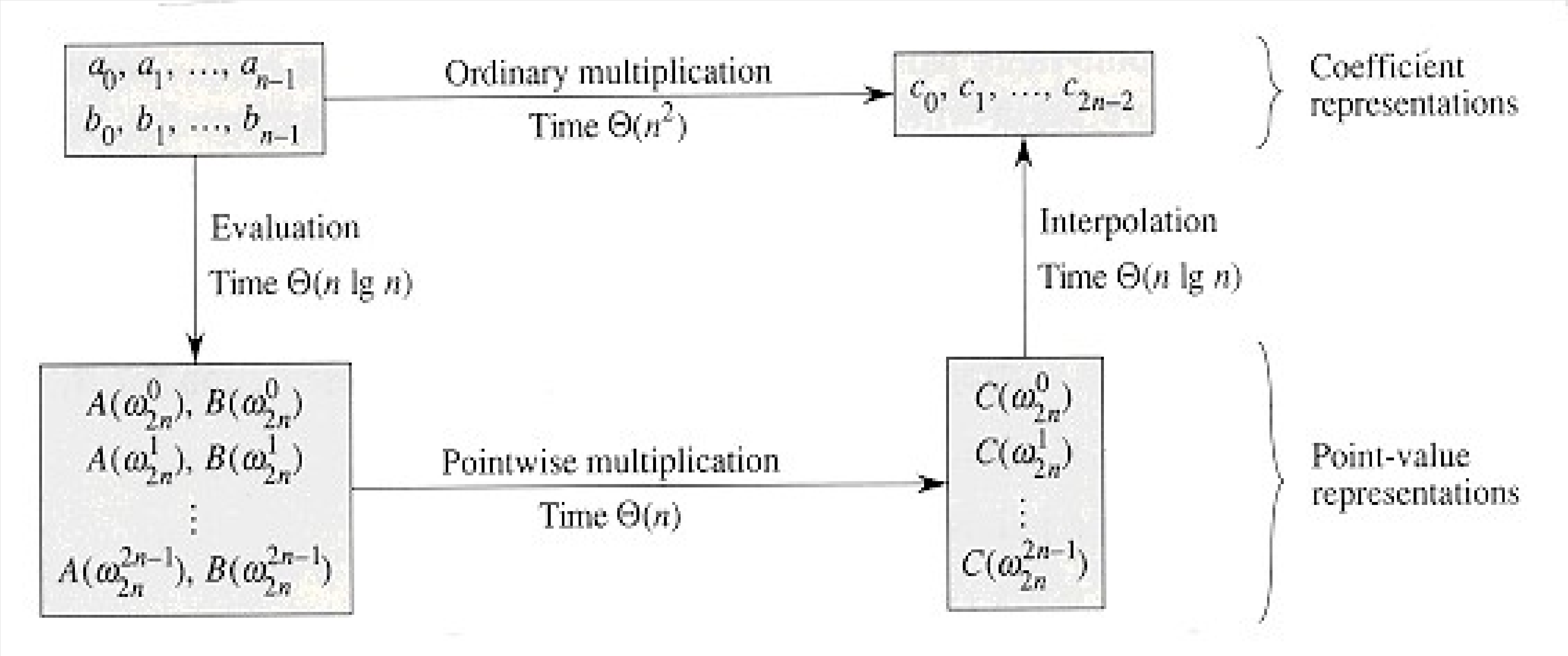


Announcements

PA2 due Sunday

HW 3 (last!) posted this weekend (will be 3 problems)

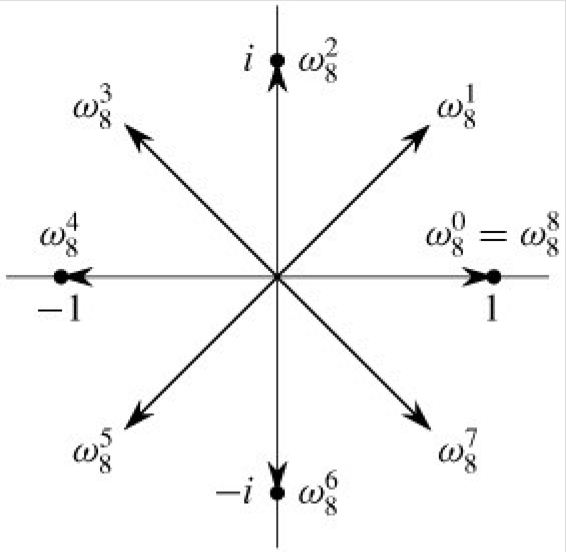
Fast Fourier Transform



Math ground work

Let $w_n^k = e^{2\pi i k/n}$

$= \cos(2\pi k/n) + i \sin(2\pi k/n)$



Called "n^th roots of unity"

We will prove/use:

1. $w_{dn}^{dk} = w_n^k \qquad (\forall n \geq 0, k \geq 0, d > 0)$
2. $w_n^{n/2} = -1 \qquad (\forall n > 0)$
3. $\sum_{j=0}^{n-1} (w_n^k)^j = 0 \qquad (\forall n > 0,\ k \text{ not divisible by } n)$
4. If $n > 0$ is even, square of nth unity roots are $n/2$ unity roots

Math ground work

Prove: $w_{dn}^{dk} = w_n^k$

By definition:

$$w_{dn}^{dk} = e^{2\pi i(dk)/(dn)} = e^{2\pi ik/n} = w_n^k$$

Prove: $w_n^{n/2} = -1$

Again, by definition:

$$w_n^{n/2} = e^{2\pi i(n/2)/n} = e^{2\pi i(1/2)}$$
$$= e^{\pi i} = -1$$

Math ground work

Prove: $$\sum_{j=0}^{n-1}(w_n^k)^j = 0$$

A geometric sum is known to be:

$$\sum_{j=0}^{n-1} ar^j = a\frac{1-r^n}{1-r}$$

... thus:

$$\sum_{j=0}^{n-1}(w_n^k)^j = \frac{1-(w_n^k)^n}{1-w_n^k} = \frac{1-(w_n^{nk})}{1-w_n^k} = \frac{1-1}{1-w_n^k} = 0$$

k not divisible by n, denominator ≠ 0

Math ground work

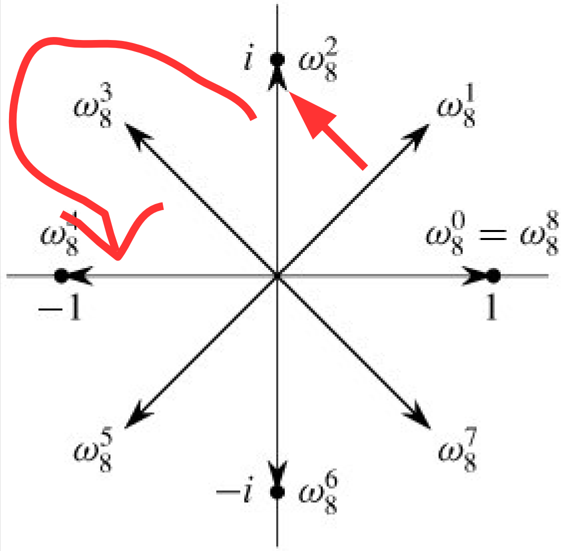
Prove: If $n > 0$ is even, square of nth unity roots are $n/2$ unity roots

Direct proof:

$(w_n^k)^2 = w_n^{2k} = w_{n/2}^k$ (using proof #1)

Picture proof:

$(w_n^k)^2 = (e^{2\pi ik/n})^2 = e^{2\pi i(2k)/n}$



Thus, twice the angle

Fast Fourier Transform

First, we need to efficiently go from coefficient to point form (n is even)

$$A(x) = \sum_{j=0}^{n-1} a_j \cdot x^j \rightarrow (x_0, y_0), \ldots (x_{n-1}, y_{n-1})$$

We will use the n roots of unity for xs

$$x_k = w_n^k, \qquad y_k = \sum_{j=0}^{n-1} a_j \cdot w_n^{k \cdot j}$$

Fast Fourier Transform

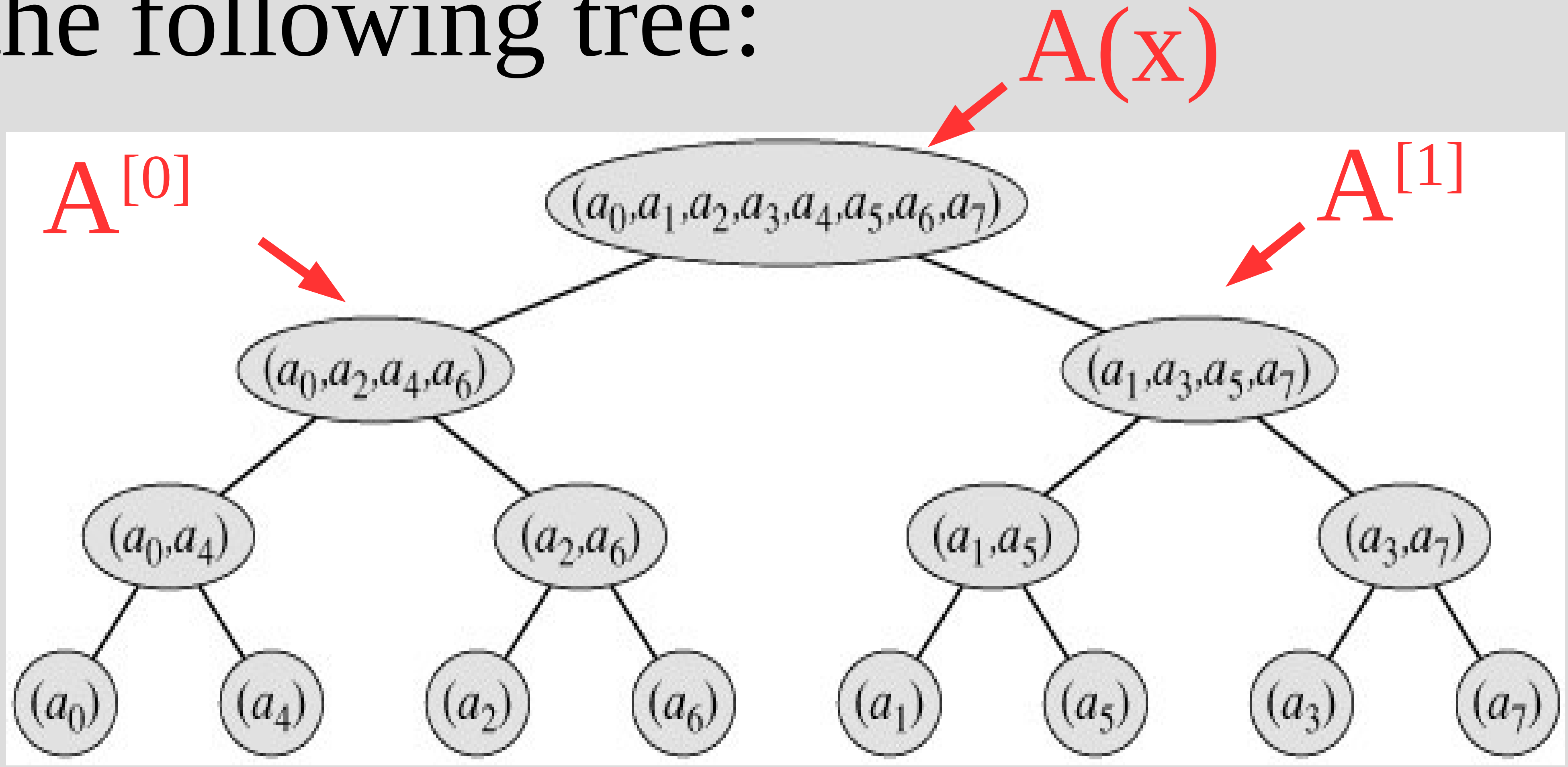
We can use the symmetry of the unity roots to divide & conquer:

First we break even and odd indexed coefficients into their own polys

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + ... + a_{n-2} x^{n/2-1}$$
$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + ... + a_{n-1} x^{n/2-1}$$

Fast Fourier Transform

By following this process, we get the following tree:



Fast Fourier Transform

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + ... + a_{n-2} x^{n/2-1}$$
$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + ... + a_{n-1} x^{n/2-1}$$

We then notice that:

$$A^{[0]}(x^2) = a_0 + a_2 x^2 + a_4 x^4 + ... + a_{n-2} x^{n-2}$$
$$A^{[1]}(x^2) = a_1 + a_3 x^2 + a_5 x^4 + ... + a_{n-1} x^{n-2}$$

Thus:

$$\mathrm{A(x)} = \mathrm{A}^{[0]}(x^2) + x A^{[1]}(x^2)$$

Fast Fourier Transform

By proof #4, computing A() as:

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2), \text{ with } x = w_n^k$$

... breaks down the problem into:
two parts, each with half the points

(as squaring nth unity roots gives n/2 unity roots)

Fast Fourier Transform

Recursive-FFT(a)

n = a.length (n assumed power of 2)

if (n == 1), return a

$w_n = e^{2\pi i/n}$, w = 1

$a[0] = (a_0, a_2, \ldots a_{n-2})$, $a[1] = (a_1, a_3, \ldots a_{n-1})$

y[0] = Recursive-FFT(a[0])

y[1] = Recursive-FFT(a[1])

for k = 0 to n/2 - 1

  $y_k = y[0]_k + w * y[1]_k$

  $y_{k+(n/2)} = y[0]_k - w * y[1]_k$

  $w = w * w_n$

return y

Fast Fourier Transform

For loop runs O(n) times with O(1) work inside each loop

2 recursive calls each size n/2, thus...

$$T(n) = \boxed{2 \cdot T(n/2)} + \boxed{O(n)}$$

$O(n^{log_2 2}) = O(n^1) = O(n)$, thus...

$\boxed{\text{between}}$ and $\boxed{\text{within}}$ recursion work the same

Thus, tack on a $\lg n$ to it: $O(n \lg n)$

Fast Fourier Transform

The first line of loop computes:

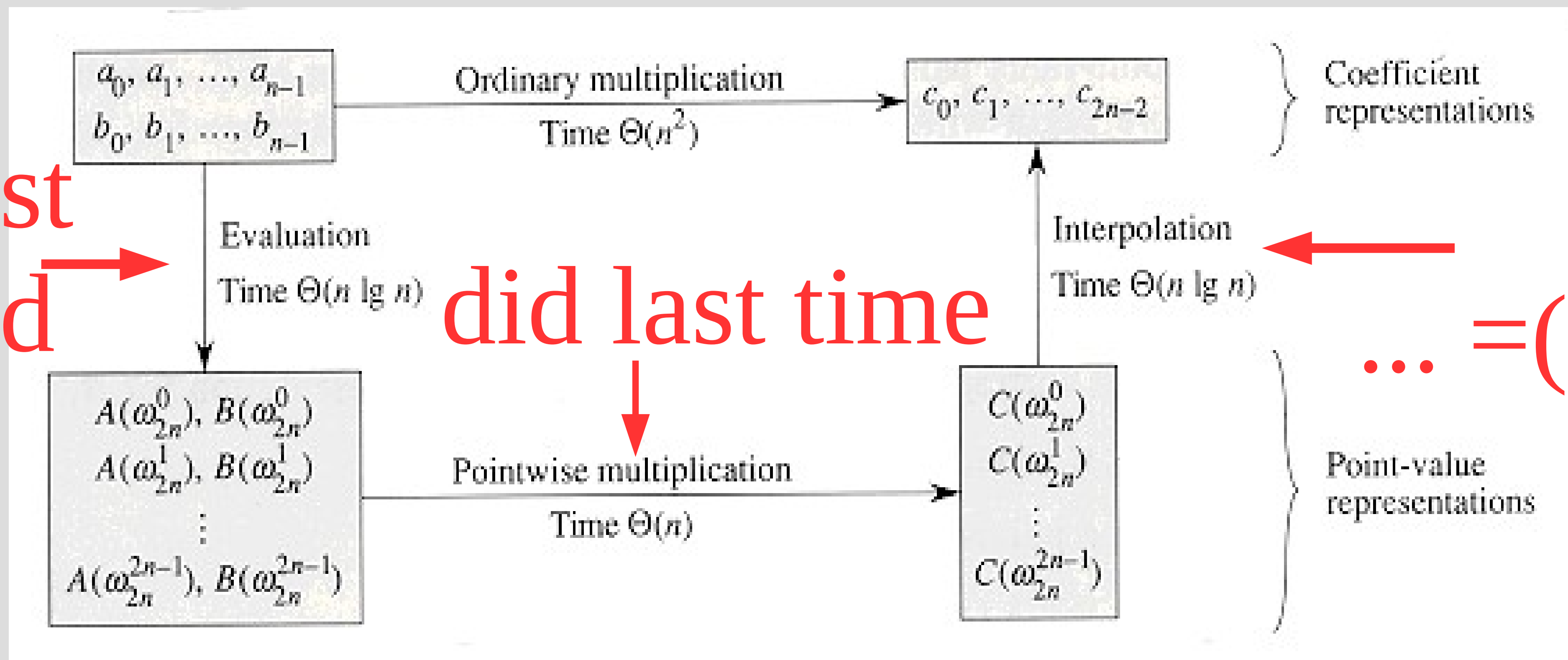
$$y_k = y_k^{[0]} + w_n^k \cdot y_k^{[1]}$$
$$= A^{[0]}(w_n^{2k}) + w_n^k \cdot A^{[1]}(w_n^{2k})$$
$$= A(w_n^k)$$

Similarly, the second finds:

proof #2

$$y_{k+(n/2)} = y_k^{[0]} - w_n^k \cdot y_k^{[1]}$$
$$= A^{[0]}(w_n^{2k}) + (-1) \cdot w_n^k \cdot A^{[1]}(w_n^{2k})$$
$$= A^{[0]}(w_n^{2k+n}) + (w_n^{n/2}) \cdot w_n^k \cdot A^{[1]}(w_n^{2k+n})$$
$$= A^{[0]}(w_n^{2k+n}) + w_n^{k+(n/2)} \cdot A^{[1]}(w_n^{2k+n})$$
$$= A(w_n^{k+(n/2)})$$

Fast Fourier Transform



just did

did last time

... =(

Fast Fourier Transform

If you remember from last time, we want to solve for a's in:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & w_n & w_n^2 & w_n^3 & \dots & w_n^{n-1} \\ 1 & w_n^2 & w_n^4 & w_n^6 & \dots & w_n^{2(n-1)} \\ 1 & w_n^3 & w_n^6 & w_n^9 & \dots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \dots & w_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

y V a

Fast Fourier Transform

To solve for a's in previous, we use the math magic below!

If we call V the previous square matrix, then the (j,k) entry in V^{-1} is: $\frac{1}{n} \cdot w_n^{-j \cdot k}$

The current (j,k) entry of V is: $w_n^{j \cdot k}$

Due to unity root magic

Fast Fourier Transform

Proof: (that this is V^{-1})

$$\text{Entry } (j,k) \text{ in } V \cdot V^{-1} = \sum_{x=0}^{n-1} w_n^{j \cdot x} (\tfrac{1}{n} w_n^{-x \cdot k})$$

$$= \frac{1}{n} \sum_{x=0}^{n-1} w_n^{x(j-k)} = \frac{1}{n} \sum_{x=0}^{n-1} (w_n^{(j-k)})^x$$

Using proof #3, if j ≠ k then this is 0

When j = k, we have $\frac{1}{n} \sum_{x=0}^{n-1} (1)^x = 1$

Fast Fourier Transform

Wait, a second... we basically just solved y = V a, with $V_{(j,k)} = w_n^{j \cdot k}$

Now we want to solve (knowing y not a) a = V^{-1} y, with $V^{-1}_{(j,k)} = \frac{1}{n} \cdot w_n^{-j \cdot k}$

This is a very similar problem!

Fast Fourier Transform

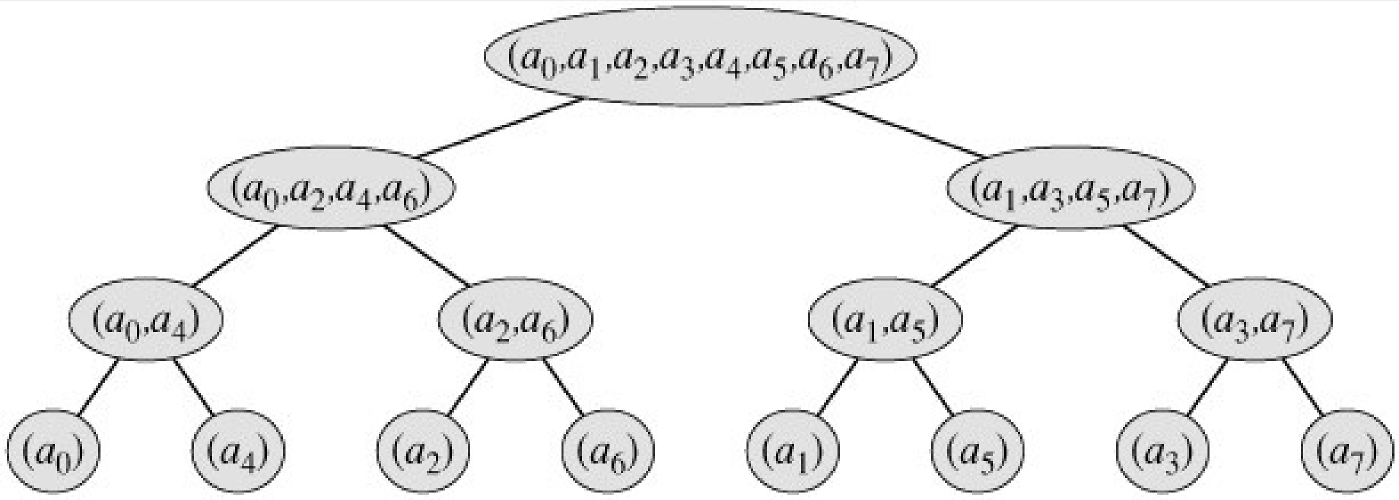
swap y and a

Recursive-FFT-backwards(y)
n = y.length (n assumed power of 2)
if (n == 1), return y
$w_n = e^{-2\pi i/n}$, w = 1 only added "-" to exponent
$y[0] = (y_0, y_2, \ldots y_{n-2})$, $y[1] = (y_1, y_3, \ldots y_{n-1})$
a[0] = Recursive-FFT(y[0])
a[1] = Recursive-FFT(y[1])
for k = 0 to n/2 - 1
    $a_k = a[0]_k + w * a[1]_k$
    $a_{k+(n/2)} = a[0]_k - w * a[1]_k$
    $w = w * w_n$
return a

after recursion, divide a by n

Fast Fourier Transform

Breaking down A(x) into $A^{[0]}(x)$ and $A^{[1]}(x)$ gives:



If we can get a_i in order of the bottom we can efficiently compute A

Fast Fourier Transform

Consider the order:

$[a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7]$

See a pattern?

Fast Fourier Transform

Consider the order:

$[a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7]$

See a pattern?

... what if I write it as:

[000,100,010,110,001,101,011,111]

These are just the bits inversed

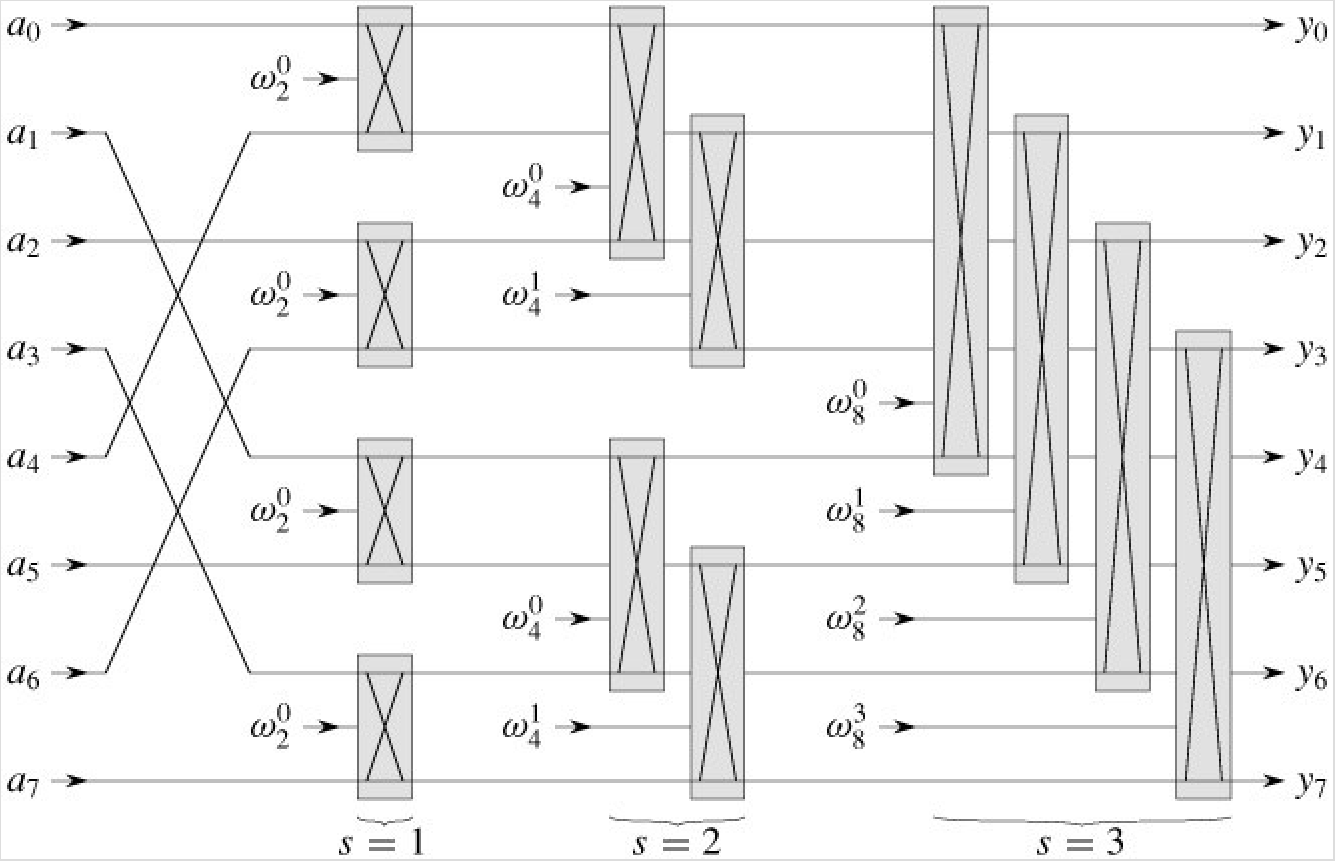
Fast Fourier Transform

Thus, if we initially swap the coefficient matrix in this order...

1. We can update the value in place
2. Each level of the tree, we compare coefficients twice as far as the previous

Fast Fourier Transform

Thus we can compute it iteratively as:



Good for parallel processing?

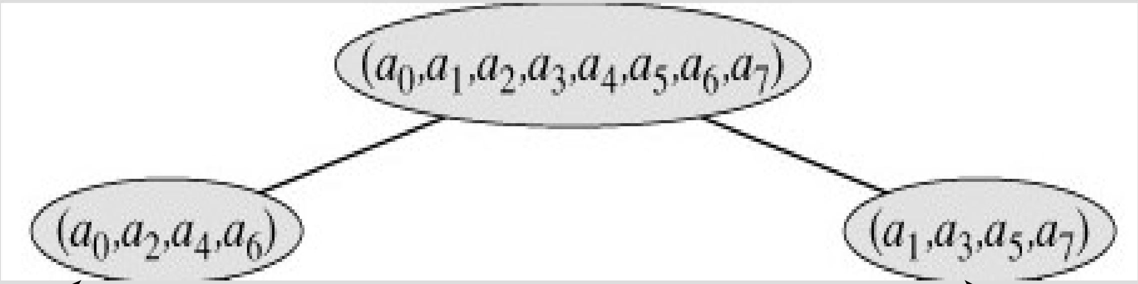
Fast Fourier Transform

This works well for a circuit,
but not so much for multi-core

The processes need to wait until all
previous level done to continue

Fast Fourier Transform

It might work just as well (or better) to parallelize the recursive calls



cpu #1 solves

cpu #2 solves

Easy ~2x speed up!