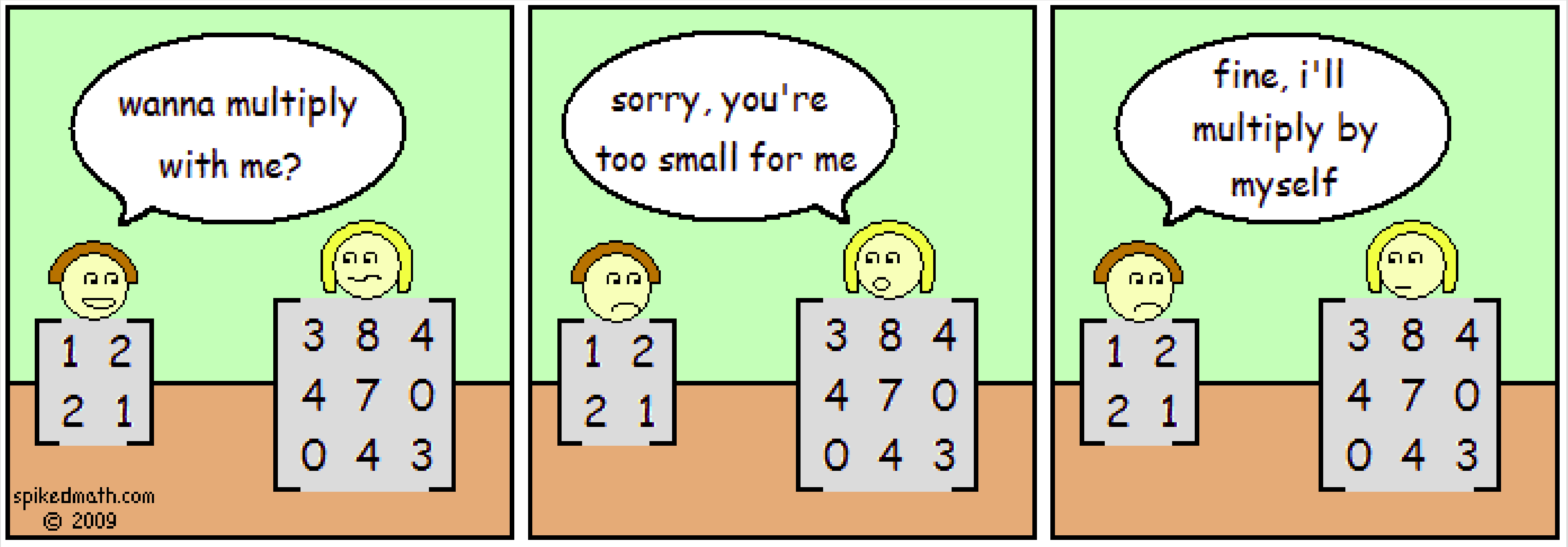
Efficient multiplication



Matrix multiplication

If you have square matrices A and B, then C = A*B is defined as:

$$c_{i,j} = \sum_{k=0}^{n} a_{i,k} \cdot b_{k,j}$$

For $\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 5 & 4 \\ -5 & 1 \end{bmatrix}$

$$\mathbf{AB} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 5 & 4 \\ -5 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 5 & 9 \end{bmatrix}$$

$$\mathbf{BA} = \begin{bmatrix} 5 & 4 \\ -5 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 13 & 4 \\ -3 & 1 \end{bmatrix}$$

Takes $O(n^3)$ time

Matrix multiplication

Can we do better?

What is the theoretical lowest running time possible?

Matrix multiplication

Can we do better?
Yes!

What is the theoretical lowest running time possible?

$O(n^2)$, must read every value at least once

Matrix multiplication

Block matrix multiplication says:

$$\left[\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right] \left[\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array}\right] = \left[\begin{array}{c|c} C_1 & C_2 \\ \hline C_3 & C_4 \end{array}\right]$$

Thus $C_1 = A_1 * B_1 + A_2 * B_3$,

We can use this fact to make a recursive definition

Matrix multiplication

Divide&conquer algorithm:

Mult(A,B)

If |A| == 1, return A*B (scalar)

else... divide A&B into 4 equal parts

  C1 = Mult(A1,B1) + Mult(A2,B3)

  C2 = Mult(A1,B2) + Mult(A2,B4)

  C3 = Mult(A3,B1) + Mult(A4,B3)

  C4 = Mult(A3,B2) + Mult(A4,B4)

Matrix multiplication

Running time:

- Base case is $O(1)$
- Recursive part needs to add two n/4 x n/4 matrices, so $O(n^2)$
- 8 recursive calls, each size n/2

$T(n) = 8\,T(n/2) + O(n^2)$

$T(n) = O(n^{\log_2 8}) = O(n^3)$

Strassen's method

Although the simple divide&conquer did not improve running time...

Can eliminate one recursive call to get $O(n^{\log_2 7})$ with fancy math

Has a much larger constant factor, so not useful unless matrix big

Strassen's method

Step 1: compute some S's (just 'cause!)

S1=B2-B4
S2=A1+A2
S3=A3+A4
S4=B3-B1
S5=A1+A4
S6=B1+B4
S7=A2-A4
S8=B3+B4
S9=A1-A3
S10=B1+B2

Strassen's method

Step 2: compute some P's (7 < 8)

P1=A1*S1

P2=S2*B4

P3=S3*B1

P4=A4*S4

P5=S5*S6

P6=S7*S8

P7=S9*S10

Strassen's method

Step 3:

C1 = P5 + P4 - P2 + P6

C2 = P1 + P2

C3 = P3 + P4

C4 = P5 + P1 - P3 - P7

(Book works out algebra for you)

Strassen's method

In practice, you should never use this on a matrix smaller than 16x16

The break-point is debatable, but Strassen's is better if over 100x100

Theoretical methods exist to reduce to $O(n^{2.3728639})$, but not practical at all

Fast Fourier Transform

The FFT is a very nice algorithm (ranks up there with bucket sort)

It has many uses, but we will use it to solve polynomial multiplication

Naive approach takes $O(n^2)$ time (i.e. FOIL)

Fast Fourier Transform

Assume we have polynomials:

$$A(x) = \sum_{j=0}^{n} a_j \cdot x^j,\ B(x) = \sum_{j=0}^{n} b_j \cdot x^j$$

C(x) = A(x) * B(x)

$$C(x) = \sum_{j=0}^{2 \cdot n} c_j x^j \qquad c_j = \sum_k a_k \cdot b_{j-k}$$

O(n) per c_j, up to 2n c_j's = $O(n^2)$

Fast Fourier Transform

Rather than directly computing C(x), map to a different representation

$A(x) = (x_0, y_0), (x_1, y_1), \ldots (x_n, y_n)$

Theorem 30.1: If $x_i \neq x_j$ for all $i \neq j$, then above gives a unique polynomial

Fast Fourier Transform

Proof: (direct)

Represent in matrix form:

$$
\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ & & \dots & & \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{bmatrix}
\begin{bmatrix} a_0 \\ a_1 \\ \dots \\ a_n \end{bmatrix}
=
\begin{bmatrix} y_0 \\ y_1 \\ \dots \\ y_n \end{bmatrix}
$$

The left matrix is invertible, done

Fast Fourier Transform

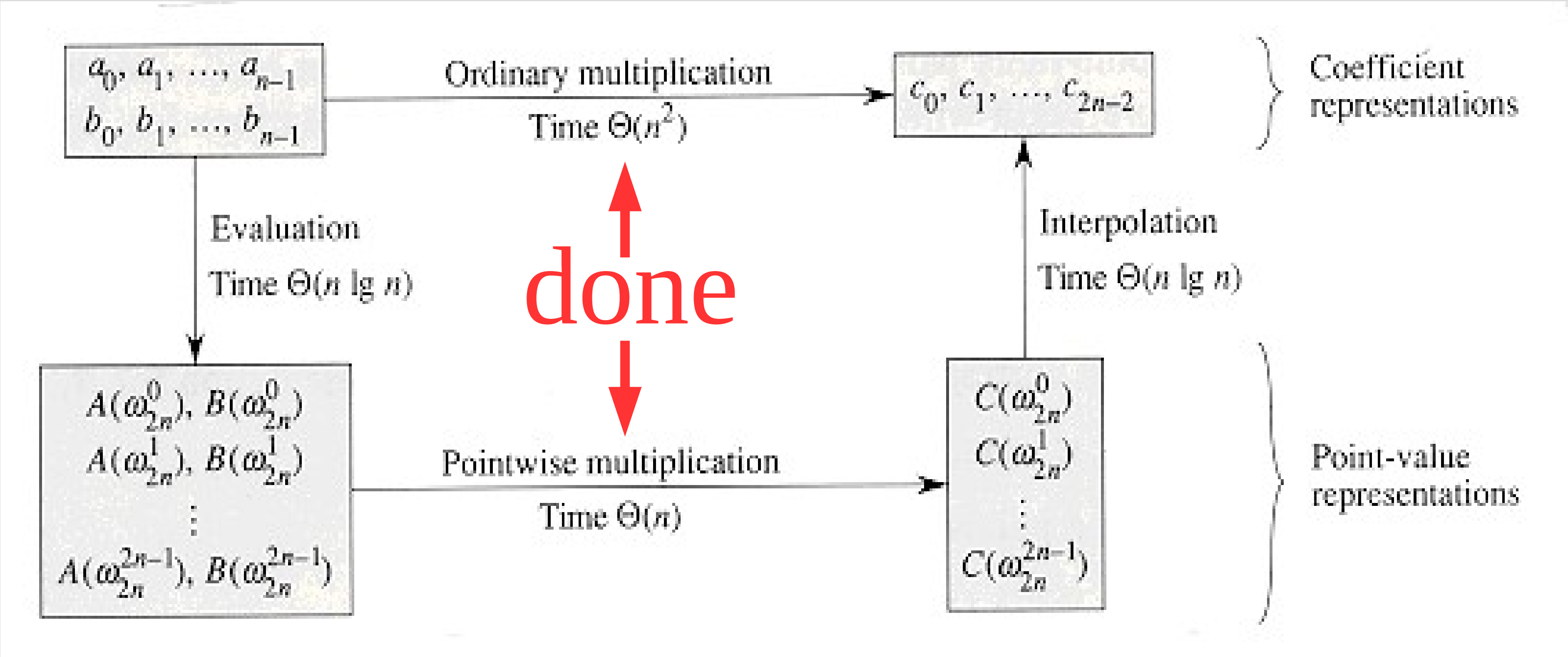
Q: Why bother with point-values?

A: We can do A(x) * B(x) in O(n) in this space

Namely, $(x_i, cy_i) = (x_i, ay_i * by_i)$

Need to get to point-value and back to coefficients in less than $O(n^2)$

Fast Fourier Transform



Coming soon! (next time)