CSE 417
Divide & Conquer (pt 2)
Famous Examples

UNIVERSITY of WASHINGTON

Reminders

- **HW2 due Friday**

- **Problem 2 correction**
  - 7 week periods: 6 to fit model, last to test model
    - don’t want test your model on the same data
  - 10 periods (1-7, 2–8, …, 10-16) and 51 penalties ~> 510 combinations
  - scatter plot demo
  - HW2 ML approach (Lasso) is generally useful
    - your code only depends on #parameters and ranges

Divide & Conquer Review

- Apply the steps:
  1. Divide the input data into 2+ parts
  2. Recursively solve the problem on each part
  3. Combine the sub-problem solutions into a problem solution

- Key questions:
  1. Can you solve the problem by combining solutions from sub-problems?
  2. Is that easier than solving it directly?

- Use master theorem to calculate the running time

Outline for Today

> Integer multiplication
> Matrix multiplication
> Fast Fourier Transform
> Integer multiplication again

Integer Multiplication

- Processor provides ability to multiply small (<= 64 bit) numbers
- Multiplying arbitrary-size integers is a classic problem
  - doesn't come up often in real programming
  - and when it does, just use a library: `java.math.BigInteger`
- Does come up in coding interviews sometimes (sadly)
- These algorithms illustrate the techniques well
  - but all use *non-obvious* insights!

Representing Large Integers

- Q: How do we represent large numbers?

- A: We'll use a list of digits

  [5, 3, 6, 8] ~> $5 \; 10^3 + 3 \; 10^2 + 6 \; 10^1 + 8 \; 10^0 = 5368$

- Other options...
  - could store the digits in the opposite order (more natural)
  - could use base 10^9 with 32-bit coefficients
    - could still then multiply in 64 bits

Integer Addition: grade-school

> Add corresponding digits with carrying

```
    5 3 6 8
+     4 2 9
=
```

Integer Addition: grade-school

> Add corresponding digits with carrying

```
      1
  5 3 6 8
+   4 2 9
=       7
```

Integer Addition: grade-school

> Add corresponding digits with carrying

```
        1
    5 3 6 8
+     4 2 9
-----------
=       9 7
```

Integer Addition: grade-school

- Add corresponding digits with carrying

$$
\begin{array}{ccccc}
 & & & 1 & \\
 & 5 & 3 & 6 & 8 \\
+ & & 4 & 2 & 9 \\ \hline
= & & 7 & 9 & 7
\end{array}
$$

Integer Addition: grade-school

> Add corresponding digits with carrying

```
      1
  5 3 6 8
+   4 2 9
= 5 7 9 7
```

> Largest possible coefficient sum is 9 + 9 + 1 (carry) = 19
  - result is 1 digit + possible carry
  - this extends to higher bases

Integer Addition: grade-school

- Add corresponding digits with carrying

```
      1
  5 3 6 8
+   4 2 9
= 5 7 9 7
```

- Implement with list representation in O(n) time
  - loop from right to left, adding coefficients + carry
  - if you end up with an extra carry, insert 1 up front [O(n)]

Integer Multiplication: grade-school

> Multiply by performing a series of one-digit multiplications...

```
          5 3 6 8
        *   4 2 9
_________________
        4 8 3 1 2
      1 0 7 3 6
  + 2 1 4 7 2   _
_________________
    2 3 0 2 8 7 2
```

> Each row is c 10^k times first number for some c & k
  - this can be done in O(n) time

Integer Multiplication: grade-school

> Multiply by performing a series of one-digit multiplications...

```
       5 3 6 8
     *   4 2 9
______________
     4 8 3 1 2
   1 0 7 3 6
+ 2 1 4 7 2   _
______________
  2 3 0 2 8 7 2
```

> n – 1 additions of all the rows
  - each takes O(n) time

Integer Multiplication: grade-school

> Multiply by performing a series of one-digit multiplications...

> Total time is

$n * O(n) +$	(n single-digit multiplications)
$(n-1) * O(n)$	(n – 1 additions)
$= O(n^2)$	

> Q: Is this optimal?

> A: Far from it

Integer Multiplication: D & C

Apply divide and conquer…

1. Split the input in half…
   How?

$$[5, 3, 6, 8] \leadsto 5\ 10^3 + 3\ 10^2 + 6\ 10^1 + 8\ 10^0 = 5368$$

$$[5, 3] \leadsto 5\ 10^1 + 3\ 10^0 = 53$$
$$[6, 8] \leadsto 6\ 10^1 + 8\ 10^0 = 68$$

$$[5, 3, 6, 8] = [5, 3]\ 10^2 + [6, 8]$$

abuse of notation…
hopefully clear

Integer Multiplication: D & C

Apply divide and conquer…

1. Split the input in half…
   How?

   `[5, 3, 6, 8] = [5, 3]` 10^2 `+ [6, 8]`

   More generally:

   `A[0..n-1] = A[0..n/2-1]` $10^{n/2}$ `+ A[n/2..n-1]`

Integer Multiplication: D & C

Apply divide and conquer…

1. Split the **inputs** (A * B = ?) in half…

$$A[0..n-1] = A[0..n/2-1]\ 10^{n/2} + A[n/2..n-1]$$
$$B[0..m-1] = B[0..m/2-1]\ 10^{m/2} + B[m/2..m-1]$$

$$A * B = (A[0..n/2-1]\ 10^{n/2} + A[n/2..n-1]) * (B[0..m/2-1]\ 10^{m/2} + B[m/2..m-1])$$

Integer Multiplication: D & C

> Apply divide and conquer…

1. Split the **inputs** (A * B = ?) in half…

```
A[0..n-1] = A[0..n/2-1] 10^(n/2) + A[n/2..n-1]
B[0..m-1] = B[0..m/2-1] 10^(m/2) + B[m/2..m-1]

A * B = A[0..n/2-1] B[0..m/2-1] 10^(n/2+m/2) +
        A[0..n/2-1] B[m/2..m-1] 10^(n/2) +
        A[n/2..m-1] B[0..m/2-1] 10^(m/2) +
        A[n/2..m-1] B[m/2..m-1]
```

Integer Multiplication: D & C

Apply divide and conquer…

1. Split the **inputs** (A * B = ?) in half…

$$
\begin{aligned}
A * B = {} & A[0..n/2-1]\ B[0..m/2-1]\ 10^{n/2+m/2} + \\
& A[0..n/2-1]\ B[m/2..m-1]\ 10^{n/2} + \\
& A[n/2..m-1]\ B[0..m/2-1]\ 10^{m/2} + \\
& A[n/2..m-1]\ B[m/2..m-1]
\end{aligned}
$$

Perform 4 multiplications on data half as large

Integer Multiplication: D & C

Apply divide and conquer…

1. Split the **inputs** (A * B = ?) in half.
2. Perform 4 multiplications on data half as large

```
A * B = A[0..n/2-1] B[0..m/2-1] 10^(n/2+m/2) +
        A[0..n/2-1] B[m/2..m-1] 10^(n/2) +
        A[n/2..m-1] B[0..m/2-1] 10^(m/2) +
        A[n/2..m-1] B[m/2..m-1]
```

3. Combine by shifting (multiply by 10^k) and adding

Integer Multiplication: D & C

Apply divide and conquer…

1. Split the **inputs** (A * B = ?) in half.
2. Perform 4 multiplications on data half as large
3. Combine by shifting (multiply by 10^k) and adding
   - multiply by 10k is just moving positions in the array
     > recall that power of 10 comes from position in the array:

[5, 3, 6, 8] ~> 5 10^3 + 3 10^2 + 6 10^1 + 8 10^0 = 5368

Integer Multiplication: D & C

Apply divide and conquer…

1. Split the **inputs** (A * B = ?) in half.
2. Perform 4 multiplications on data half as large
3. Combine by shifting (multiply by 10^k) and adding
   - multiply by 10^k is just moving positions in the array (*shifting*)
     > takes O(n) time
   - addition takes O(n) time using grade-school algorithm

Integer Multiplication: D & C

> Apply divide and conquer...

1. Split the **inputs** (A * B = ?) in half.
   Perform 4 multiplications on data half as large

2. Combine by shifting (multiply by 10^k) and adding in $O(n+m)$ time

> Running time given by...

$$T(1) = O(1)$$

$$T(n) = 4\ T(n/2) + O(n)$$

simplify by assuming $m = n$
(two numbers of same size)

Integer Multiplication: D & C

- Apply divide and conquer...
- Running time given by...

$T(1) = O(1)$
$T(n) = 4\,T(n/2) + O(n)$

$C = \log_2 4 = 2$

Compare $O(n)$ to $O(n^C) = O(n^2)$

Running time is $O(n^2)$, same as grade-school version

Integer Multiplication: Karatsuba

- Need a smarter divide & combine approach... (and notation...)

  $A = A_1\ 10^k + A_0$ and $B = B_1\ 10^k + B_0$

- Consider computing...

  $(A_1 + A_0)\ (B_1 + B_0) = A_1\ B_1 + A_1\ B_0 + A_0\ B_1 + A_0\ B_0$

- If we also compute $A_1\ B_1$ and $A_0\ B_0$, then we also get $A_1\ B_0 + A_0\ B_1$ by subtraction

Integer Multiplication: Karatsuba

- Need a smarter divide & combine approach… (also notation…)

  $A = A_1\, 10^k + A_0$ and $B = B_1\, 10^k + B_0$

- Matters because $A\,B = (A_1\, 10^k + A_0)\,(B_1\, 10^k + B_0) =$
  $A_1\, B_1\, 10^{2k} + (A_1\, B_0 + A_0\, B_1)\, 10^k + A_0\, B_0$

- By computing $(A_1 + A_0)\,(B_1 + B_0)$, $A_1\, B_1$, and $A_0\, B_0$, we also get $A_1\, B_1 + A_1\, B_0$ in $O(n)$ time
  - only 3 multiplications
  - get A B from those 3 multiplications with adding & shifting

Integer Multiplication: Karatsuba

1. Divide into numbers of half the size
   - $A_1 + A_0$ and $B_1 + B_0$ have at most 1 extra digit
2. Recursively compute 3 multiplications
3. Combine by O(1) subtractions, shifts, and additions
   - takes O(n) time

> Running time given by

$$T(1) = O(1)$$

$$T(n) = 3\,T(n / 2) + O(n) \quad \text{for } n > 1$$

Integer Multiplication: Karatsuba

> Running time given by

$$T(1) = O(1)$$
$$T(n) = 3\,T(n / 2) + O(n) \quad \text{for } n > 1$$

> $C = \log_2 3 = 1.58496\ldots$
  – compare $O(n)$ to $O(n^{1.585})$

> Master theorem says it takes $O(n^{1.585})$ time

Outline for Today

> Integer multiplication
> Matrix multiplication
> Fast Fourier Transform
> Integer multiplication again

Matrix Multiplication (out of scope)

- Matrix is a table of numbers
  - group of linear transformations of a vector space
- Can think of it as an array of arrays
  - has **two indexes**: A[i][j]
- Multiplication defined by...

$$(A * B)[i][j] := \sum_{k=1}^{n} A[i][k] * B[k][j]$$

Matrix Multiplication (out of scope)

> Multiplication defined by...

$$(A * B)[i][j] := \sum_{k=1}^{n} A[i][k] * B[k][j]$$

> Direct implementation takes O(n^3) time

```
for i = 1 .. n
  for j = 1 .. n
    C[i][j] = 0
    for k = 1 .. n
      C[i][j] += A[i][k] * B[k][j]
```

Matrix Multiplication (out of scope)

- Divide & Conquer approach: Strassen's algorithm

  1. Split each matrix into 4 of size (n / 2) x (n / 2)
  2. Get the parts of A * B using 7 multiplications
     - also numerous additions etc.

- Running time satisfies T(1) = 1 and

  $$T(n) = 7\,T(n / 2) + O(n^2)$$

Matrix Multiplication (out of scope)

> Running time satisfies T(1) = 1 and

$$T(n) = 7\,T(n / 2) + O(n^2)$$

> Master theorem says to compare $O(n^2)$ to $O(n^C)$ with $C = \log_2 7 = 2.807...$
  – second is larger so $O(n^{2.8074})$ time

> Note: 8 multiplications would give $C = \log_2 8 = 3$
  – removing 1 extra multiplication gives the improvement

Matrix Multiplication (out of scope)

- Same idea was extended by others:
  - split matrix into **more pieces**
  - find ways to save multiplications
- As of 1990, best was approach of Coppersmith & Winograd
  - around 2.38
  - recently improved by Strothers then Williams then Le Gall
    - Williams used math + computer search
  - best now stands at 2.37286….

Matrix Multiplication (out of scope)

- In practice, only Strassen's $O(n^{2.807})$ is used
- To see why, suppose we split into k^2 pieces of size n/k x n/k
  - any reduction below k^3 multiplications is a speedup
  - (will be at least k^2)
- Take k = 100
  - we need to beat 100^3 = 1,000,000 multiplications
  - 50,000 multiplications would beat the best algorithm
  - will probably need 100k+ additions

Matrix Multiplication (out of scope)

> Take k = 100
  - we need to beat 100^3 = 1,000,000 multiplications
  - 50,000 multiplications would beat the best algorithm
  - will probably need 100k+ additions

> Suppose we had such an algorithm...
  It's running time satisfies T(1) = 1 and

$$T(n) = 50{,}000\ T(n/100) + O(n^2)$$

  - hidden constant in $O(n^2)$ is 100k+

Matrix Multiplication (out of scope)

- Suppose we had such an algorithm...
  It's running time satisfies $T(1) = 1$ and

  $$T(n) = 50{,}000\, T(n/100) + O(n^2)$$

  - hidden constant in $O(n^2)$ is 100k+

- Analysis says it is $O(n^{2.373})$,
  but the hidden constant on the $O(n^2)$ part is huge
  - matrices large enough to make the $O(n^2)$ term smaller are too big for the memory of modern computers

Matrix Multiplication (out of scope)

- In theory, matrix multiplication is extremely important
- Many other problems reduce to multiplication or use matrix multiplication as a key component
  - single-source shortest paths (Sankowski 2005)
    - also all-pairs shortest paths
  - perfect matching (Harvey 2006)
  - weighted linear matroid intersection (Harvey 2007)
  - …
- We use ω to indicate best exponent, $O(n^{\omega})$
  - algorithms get faster each time ω is improved

Matrix Multiplication (out of scope)

> Still an open question whether ω can be arbitrarily close to 2

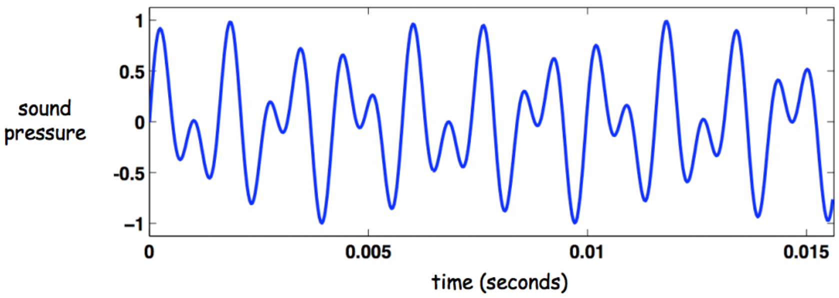
> Latest result shows can get as close as you want to 2 provided certain certain algebraic / combinatoric conjectures are true
  - result of Cohn, Kleinberg, Szegedy, and Umans
  - **open research problem**

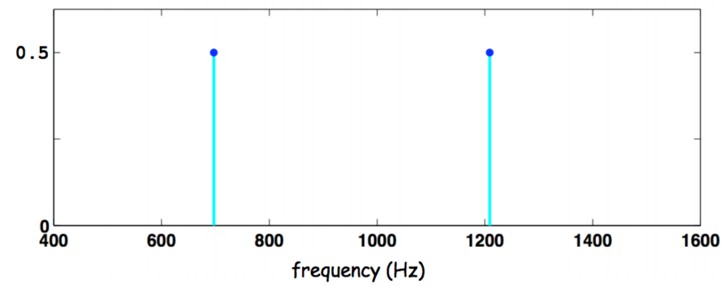
Outline for Today

> Integer multiplication
> Matrix multiplication
> Fast Fourier Transform
> Integer multiplication again

Fourier Transform

- Fourier Transform converts information in the time domain to information in the frequency domain
  - air pressure at time t to tones / notes
  - light intensity at time t to colors
  - simpler properties to more meaningful ones





Fourier Transform

- Let $A[t]$ be the amplitude at time t
  - we can assume a discrete signal without loss of generality (Shannon)
- Fourier transform of A, often written $\hat{A}$, is defined by

$$\hat{A}[k] = A[0] + A[1]\,\sigma^{k} + A[2]\,\sigma^{2k} + \ldots + A[n-1]\,\sigma^{(n-1)k}$$

  where $\sigma = \exp(-2\pi i/n)$, a complex number

- Naive implementation runs in $\Theta(n^2)$ time
  - for each $k = 0 .. n-1$, evaluate formula in $O(n)$ time

Fourier Transform

> Wide-spread applications
  - signal processing, telecommunications
    > cell phones, music
  - image and video compression
  - speech recognition
  - medical imaging
    > MRI, CT, PET scans
  - optics
  - radar
  - seismology
  - ...

we hear in frequency domain
not time domain

Fast Fourier Transform (FFT)

> FFT implements the Fourier Transform in O(n log n) time

> Previous applications would not be possible otherwise
  – huge difference between $O(n^2)$ and O(n log n)
  – if n = 100,000, then n^2 = 10,000,000,000 and n log n = 1,609,640

> "The FFT is one of the truly great computational developments of the century. It has changed the face of science and engineering so much that it is not an exaggeration to say that life as we know it would be very different without it."
  — Charles van Loan

Fast Fourier Transform (FFT)

- Cannot speed up the calculation of Â[k]
  - have to read all the inputs, which takes $\Theta(n)$ time
- We can can speed up computation of all Â[k]'s together
  - key is to recognize repeated work in the calculation of different Â[k]'s
- Algorithm is credited to Cooley and Tukey (1965)
  - was actually discovered by Gauss (1805)

Fast Fourier Transform (FFT)

Apply divide and conquer…

1. Split the data into evens and odds…

$$A_{even} = [A[0], A[2], \ldots, A[n-2]]$$
$$A_{odd} = [A[1], A[3], \ldots, A[n-1]]$$

generalize the problem slightly (?)

assume n is even

2. Call FFT recursively *using σ² instead of σ* to get…

$$\hat{A}_{even}[k] = A_{even}[0] + A_{even}[1]\,(\sigma^2)^k + \ldots + A_{even}[n/2-1]\,(\sigma^2)^{(n/2-1)k}$$
$$\hat{A}_{odd}[k] = A_{odd}[0] + A_{odd}[1]\,(\sigma^2)^k + \ldots + A_{odd}[n/2-1]\,(\sigma^2)^{(n/2-1)k}$$

Fast Fourier Transform (FFT)

Apply divide and conquer...

1. Split the data into evens and odds...

$$A_{even} = [A[0], A[2], \ldots, A[n-2]]$$
$$A_{odd} = [A[1], A[3], \ldots, A[n-1]]$$

apply definition of A_{even} & A_{odd}

2. Call FFT recursively *using σ^2 instead of σ* to get...

$$\hat{A}_{even}[k] = A_{even}[0] + A_{even}[1]\,\sigma^{2k} + \ldots + A_{even}[n/2-1]\,\sigma^{(n-2)k}$$
$$\hat{A}_{odd}[k] = A_{odd}[0] + A_{odd}[1]\,\sigma^{2k} + \ldots + A_{odd}[n/2-1]\,\sigma^{(n-2)k}$$

Fast Fourier Transform (FFT)

Apply divide and conquer…

1. Split the data into evens and odds…

$$A_{even} = [A[0], A[2], \ldots, A[n-2]]$$
$$A_{odd} = [A[1], A[3], \ldots, A[n-1]]$$

2. Call FFT recursively *using σ^2 instead of σ* to get…

$$\hat{A}_{even}[k] = A[0] + A[2]\,\sigma^{2k} + \ldots + A[n-2]\,\sigma^{(n-1)k}$$
$$\hat{A}_{odd}[k] = A[1] + A[3]\,\sigma^{2k} + \ldots + A[n-1]\,\sigma^{(n-2)k}$$

Fast Fourier Transform (FFT)

Apply divide and conquer…

1. Split the data into evens and odds…
2. Call FFT recursively *using σ^2 instead of σ* to get…

$$\hat{A}_{even}[k] = A[0] + A[2]\,\sigma^{2k} + \ldots + A[n-2]\,\sigma^{(n-2)k}$$
$$\hat{A}_{odd}[k] = A[1] + A[3]\,\sigma^{2k} + \ldots + A[n-1]\,\sigma^{(n-2)k}$$

3. Combine using the formula

$$\hat{A}[k] = \hat{A}_{even}[k] + \hat{A}_{odd}[k]\,\sigma^k$$

Fast Fourier Transform (FFT)

$$\hat{A}_{even}[k] = A[0] + A[2]\,\sigma^{2k} + \ldots + A[n-2]\,\sigma^{(n-2)k}$$
$$\hat{A}_{odd}[k] = A[1] + A[3]\,\sigma^{2k} + \ldots + A[n-1]\,\sigma^{(n-2)k}$$

3. Combine using the formula

$$\begin{aligned}\hat{A}[k] &= \hat{A}_{even}[k] + \hat{A}_{odd}[k]\,\sigma^{k} \\ &= (A[0] + A[2]\,\sigma^{2k} + \ldots + A[n-2]\,\sigma^{(n-2)k}) + \\ &\quad (A[1] + A[3]\,\sigma^{2k} + \ldots + A[n-1]\,\sigma^{(n-2)k})\,\sigma^{k} \\ &= A[0] + A[1]\,\sigma^{k} + A[2]\,\sigma^{2k} + \ldots A[n-1]\,\sigma^{(n-1)k}\end{aligned}$$

Fast Fourier Transform (FFT)

3. Combine using the formula

$$\hat{A}[k] = \hat{A}_{even}[k] + \hat{A}_{odd}[k]\ \sigma^k$$

EXCEPT that only works if $k \leq n/2$

- otherwise there is no such index in $\hat{A}_{even}$ and $\hat{A}_{odd}$

Fast Fourier Transform (FFT)

3. Combine using the formula

$$\hat{A}[k] = \hat{A}_{even}[k] + \hat{A}_{odd}[k]\ \sigma^{k}$$

For k + n/2, we want (just apply the formulas)

$$\hat{A}_{even}[k+n/2] = A[0] + A[2]\ \sigma^{2(k+n/2)} + \ldots + A[n-2]\ \sigma^{(n-2)(k+n/2)}$$
$$\hat{A}_{odd}[k+n/2] = A[1] + A[3]\ \sigma^{2(k+n/2)} + \ldots + A[n-1]\ \sigma^{(n-2)(k+n/2)}$$

Fast Fourier Transform (FFT)

3. Combine using the formula

$$\hat{A}[k] = \hat{A}_{even}[k] + \hat{A}_{odd}[k]\, \sigma^k$$

even number x (n/2)
= multiple of n

For k + n/2, we want

$$\hat{A}_{even}[k+n/2] = A[0] + A[2]\, \sigma^{2k+2n/2} + \ldots + A[n-2]\, \sigma^{(n-2)k+(n-2)n/2}$$
$$\hat{A}_{odd}[k+n/2] = A[1] + A[3]\, \sigma^{2k+2n/2} + \ldots + A[n-1]\, \sigma^{(n-2)k+(n-2)n/2}$$

Now use fact that $\sigma^n = 1$ (so $\sigma^{an} = 1$)

> for everything else any number would have worked!

Fast Fourier Transform (FFT)

> Apply divide and conquer...

1. Split the data into evens and odds...
2. Call FFT recursively *using σ^2 instead of σ* to get...
3. Combine using the formulas

$$\hat{A}[k] = \hat{A}_{even}[k] + \hat{A}_{odd}[k]\,\sigma^k \quad \text{and} \quad \hat{A}[k+n/2] = \hat{A}_{even}[k] + \hat{A}_{odd}[k]\,\sigma^{k+n/2}$$

> Split and combine in O(n) time

- total time is O(n log n) by master theorem

Quantum Fourier Transform (QFT) (waaaaay out of scope)

- FFT computes the Fourier transform in O(n) time
- QFT computes the Fourier transform in time…

O(log n)

- This can’t possibly be true
  - takes O(n) time just to write the output!
    - instead, the output is a quantum superposition of the Â[k]’s
  - takes O(n) time just to read the input!
    - (and we need to read all the input to get the right answer)
    - instead, the input must be a quantum superposition of the A[k]’s

Quantum Fourier Transform (QFT) (waaaaay out of scope)

- QFTs has inputs and outputs that are quantum super-positions
- No longer clear that you can use this for anything useful!
  - getting the usual output would take at least O(n) time
  - key will be to apply this where the input is exponentially large (and output isn't)

Quantum Fourier Transform (QFT) (waaaaay out of scope)

- Shor (1994) showed that you can use the QFT to efficiently...
  - factor numbers
  - compute discrete logarithms
- Means quantum computers could break cryptography
  - RSA & Diffie-Hellman: both widely used and broken
  - movie Sneakers (1992) considers a similar scenario

- Many non-scary applications also
  - quantum simulation to develop new drugs
  - quantum machine learning

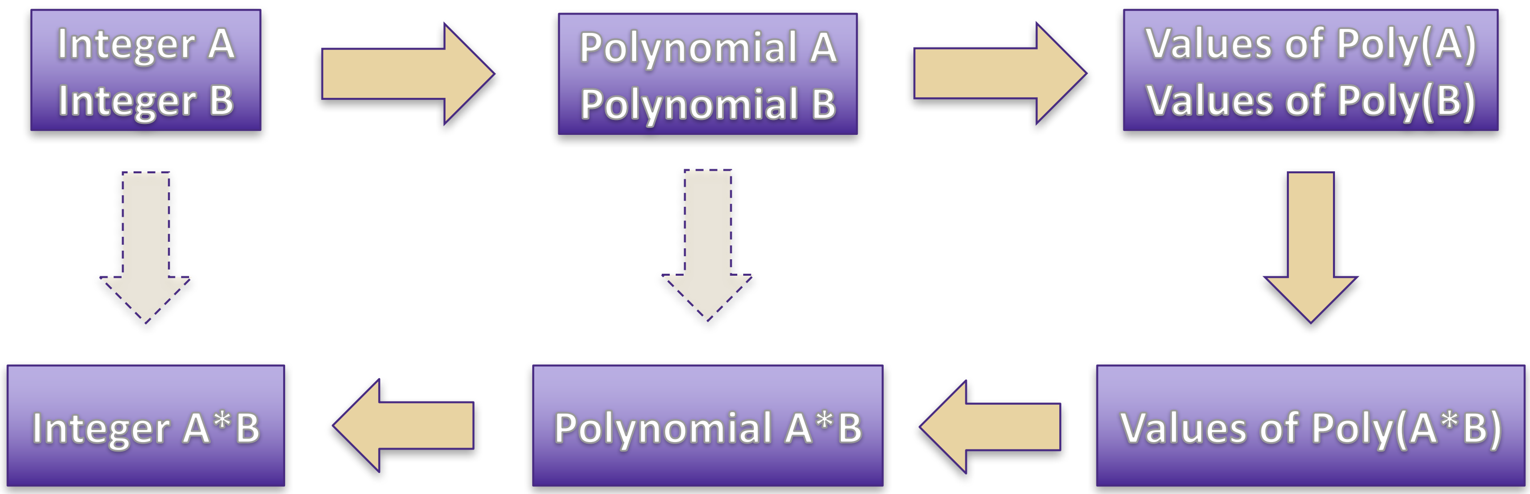
Quantum Fourier Transform (QFT) (waaaaay out of scope)

- QFT generalizes further using group theory
  - exactly how far is an open question
  - applications of this to other problems from Le Gall, Z, and many others

Outline for Today

> Integer multiplication
> Matrix multiplication
> Fast Fourier Transform
> Integer multiplication again

Integer Multiplication: Schönhage-Strassen (out of scope)



Integer Multiplication: Schönhage-Strassen (out of scope)

> Switch from integers to polynomial

$$5368 * 235 =$$
$$(5\ 10^3 + 3\ 10^2 + 6\ 10^1 + 8)\ (2\ 10^2 + 3\ 10 + 5)$$

versus

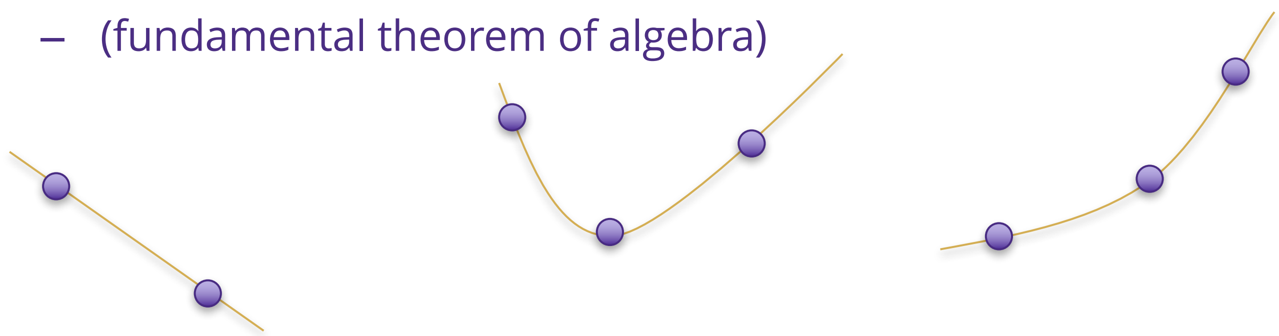
$$(5\ x^3 + 3\ x^2 + 6\ x^1 + 8)\ (2\ x^2 + 3\ x + 5)$$

> Difference is integer multiplication requires carrying
  - if a coefficient is too large, move part into the next coefficient…
  - can do this in O*(n) time

Integer Multiplication: Schönhage–Strassen (out of scope)

> Can represent a degree-n polynomial by n+1 coefficients or by its value at n+1 **distinct points**

- *exactly one line* goes through any two points
- *exactly one parabola* goes through any three points
- …
- (fundamental theorem of algebra)



Integer Multiplication: Schönhage-Strassen (out of scope)

- Switch to polynomial representations from list of coefficients to list of the function values at specific points

- Now, to multiply the polynomials, just multiply the values at those points
  - this is the definition of function multiplication
  - $\underbrace{(a_n\, x^n + \ldots + a_1 + a_0)}_{f(x)}\ \underbrace{(b_n\, x^n + \ldots + b_1 + b_0)}_{g(x)}$
  - if f(x) = a and g(x) = b, then (f * g)(x) = f(x) g(x) = a b

Integer Multiplication: Schönhage-Strassen (out of scope)

- Can pick *any* n+1 points: $x_0, x_1, \ldots, x_n$

- Value of the polynomial at x_k is

$$f(x_j) = a_n\, x_k^n + \ldots + a_1\, x_k + a_0$$

- Unfortunately, this takes O(n) time per point, so $O(n^2)$ to evaluate at all the points
  - if we want an O(n2) algorithm, grade-school works fine

Integer Multiplication: Schönhage-Strassen (out of scope)

> Pick the right n+1 points...

> Take $x_k = \sigma^k$

$$f(x_k) = a_n \sigma^{kn} + \ldots + a_1 \sigma^k + a_0$$

> Function evaluation becomes the FT
  - FFT evaluates the polynomial at n+1 points in O(n log n) time
  - going from points to the coefficients is the inverse FFT, which also takes O(n log n) time (same algorithm)

Integer Multiplication: Schönhage-Strassen (out of scope)

> Running time (O* = ignores exponentially smaller factors)
  1. O*(n) to convert between integers and polynomials
  2. O*(n log n) to evaluate
  3. O*(n) to multiply pointwise

> Total is O*(n log n). In fact, O(n log n log log n)

> Many more details…
  – particular difficulty is how to represent numbers exactly
  – floats would have potential round-off errors