Obvious Algorithm:
- Compute
- Given:
  - Degree $n-1$ polynomials $P$ and $Q$
  - $P = a_0 + a_1 x + a_2 x^2 + ... + a_{n-2} x^{n-2} + a_{n-1} x^{n-1}$
  - $Q = b_0 + b_1 x + b_2 x^2 + ... + b_{n-2} x^{n-2} + b_{n-1} x^{n-1}$
- Compute:
  - Degree $2n/2$ Polynomial $PQ$
  - $PQ = (a_{n-2} b_{n-2} + (a_{n-1} b_0) x + (a_{n-1} b_1 + a_{n-2} b_1) x^2$ $+ ... + (a_0 b_{n-2} + a_{n-1} b_{n-1}) x^{2n/2}$ $+ a_{n-2} b_{n-1} x^{2n-2}$
- Obvious Algorithm:
  - Compute all $a_i b_j$ and collect terms
  - $\Theta(n^2)$ time

Another Divide & Conquer Example: Multiplying Faster
- On the first HW you analyzed our usual algorithm for multiplying numbers
  - $\Theta(n^2)$ time
- On real machines each “digit” is, e.g., 32 bits long but still get $\Theta(n^2)$ running time with this algorithm when run on n-bit multiplication
- We can do better!
  - We’ll describe the basic ideas by multiplying polynomials rather than integers
  - Advantage is we don’t get confused by worrying about carries at first

Polynomial Multiplication
- Given:
  - Degree $n-1$ polynomials $P$ and $Q$
- Compute:
  - Degree $2n/2$ Polynomial $PQ$
- Obvious Algorithm:
  - Compute all $a_i b_j$ and collect terms
  - $\Theta(n^2)$ time

Naive Divide and Conquer
- Assume $n=2k$
  - $P = (a_0 + a_1 x + a_2 x^2 + ... + a_{k-2} x^{k-2} + a_{k-1} x^{k-1}) +$ $(a_0 + a_1 x + ... + a_{k-2} x^{k-2} + a_{k-1} x^{k-1}) x^k$ $= P_0 + P_1 x^k$ where $P_0$ and $P_1$ are degree $k-1$ polynomials
  - Similarly $Q = Q_0 + Q_1 x^k$
  - $PQ = (P_0 + P_1 x^k)(Q_0 + Q_1 x^k)$
    - $= P_0 Q_0 + (P_0 Q_1 + P_1 Q_0) x^k + P_1 Q_1 x^{2k}$
  - 4 sub-problems of size $k=n/2$ plus linear combining
- $T(n)$=4 $T(n/2)$+cn $\quad$ Solution $T(n) = \Theta(n^2)$

Master Divide and Conquer Recurrence
- If $T(n) = a T(n/b) + cn^k$ for $n>b$ then
  - if $a>b^k$ then $T(n) = \Theta(n^k)$
  - if $a=b^k$ then $T(n) = \Theta(n^k \log n)$
  - Works even if it is $[n/b]$ instead of $n/b$. 

Notes on Polynomials
- These are just formal sequences of coefficients
- when we show something multiplied by $x^k$ it just means shifted $k$ places to the left – basically no work

Usual polynomial multiplication

<table>
<thead>
<tr>
<th>4x^2 + 2x + 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x^2 - 3x + 1</td>
</tr>
<tr>
<td>4x^2 + 2x + 2</td>
</tr>
<tr>
<td>-12x^2 - 6x</td>
</tr>
<tr>
<td>4x^2 - 10x + 2</td>
</tr>
</tbody>
</table>
Karatsuba’s Algorithm

A better way to compute the terms

Compute

\[ A \leftarrow P_0 Q_0 \]
\[ B \leftarrow P_1 Q_1 \]
\[ C \leftarrow (P_0 + P_1)(Q_0 + Q_1) = P_0 Q_0 + P_0 Q_1 + P_1 Q_0 + P_1 Q_1 \]

Then

\[ P_0 Q_1 + P_1 Q_0 = C - A - B \]

So

\[ P_0 Q_1 = \frac{C - A - B}{2} \cdot \text{shift}(n, 1) \]

3 sub-problems of size \( n/2 \) plus \( O(n) \) work

\[ T(n) = 3 T(n/2) + O(n) \] where \( a = \log_2 3 \approx 1.59... \)

Karatsuba: Details

PolyMul(P, Q):

// P, Q are length \( n \rightarrow 2k \) vectors, \( \|P[k], Q[k]\| \) being
// the coefficient of \( x^k \) in polynomials P, Q respectively.
// Let Pzero be elements \( 0..k-1 \) of P, Pone be elements \( k..n-1 \)
// Qzero, Qone: similar
A ← PolyMul(Pzero, Qzero); // result is a \( (2k-1) \)-vector
B ← PolyMul(Pone, Qone);   // ditto
Psum ← Pzero + Pone; // add corresponding elements
Qsum ← Qzero + Qone;       // ditto
C ← polyMul(Psum, Qsum); // another \( (2k-1) \)-vector
Mid ← C – A – B; // subtract corresponding elements
R ← A + \text{shift}(n/2) + \text{shift}(B, n)  // a \( (2n-1) \)-vector
Return( R );

Multiplication

Polynomials

- Naive: \( \Theta(n^2) \)
- Karatsuba: \( \Theta(n^{1.58}) \)
- Best known: \( \Theta(n \log n) \) \( \approx \) "Fast Fourier Transform"
- FFT widely used for signal processing

Integers

- Similar, but some ugly details re: carries, etc. gives \( \Theta(n \log n \log \log n) \), mostly unused in practice except for symbolic manipulation systems like Maple

Hints towards FFT: Interpolation

Given set of values at 5 points Can find unique degree 4 polynomial going through these points
Karatsuba’s algorithm and evaluation and interpolation

- Strassen gave a way of doing 2x2 matrix multiplies with fewer multiplications.
- Karatsuba’s algorithm can be thought of as a way of multiplying degree 1 polynomials (which have 2 coefficients) using fewer multiplications.
  - \( P_0 = (P_0^2) + (P_0 Q_0) \)
  - \( P_1 = (P_1 Q_0) + (P_0 Q_1) \)
  - Evaluate at 0,1,-1 (Could also use other points)
    - \( A = P(0) Q(0) = P_0 \)
    - \( C = P(1) Q(1) = (P_0 P_1)(Q_0 + Q_1) \)
    - \( D = P(-1) Q(1) = (P_1 P_0)(Q_0 - Q_1) \)
  - Interpolating, Karatsuba’s Mid=(C-D)/2 and \( B = (C+D)/2 - A \)

Fun facts about \( \omega = e^{2\pi i/n} \) for even \( n \)

- \( \omega^n = 1 \)
- \( \omega^{n/2} = -1 \)
- \( \omega^{kn/2} = \omega^k \) for all values of \( k \)
- \( \omega^2 = e^{\pi i m/n} \) where \( m \equiv n/2 \)
- \( \omega^k = \cos(2k\pi/n) + i\sin(2k\pi/n) \) so can compute with powers of \( \omega \)

The recursive idea for \( n \) a power of 2

Also
  - \( P_{even} \) and \( P_{odd} \) have degree \( n/2 \) where
    - \( P(\omega^k) = P_{even}(\omega^{2k}) + \omega^k P_{odd}(\omega^{2k}) \)
    - \( P(-\omega^k) = P_{even}(\omega^{2k}) - \omega^k P_{odd}(\omega^{2k}) \)

Recursive Algorithm
  - Evaluate \( P_{even} \) at \( 1, \omega^n, \omega^{2n}, \ldots, \omega^{(n/2-1)n} \)
  - Evaluate \( P_{odd} \) at \( 1, \omega^n, \omega^{2n}, \ldots, \omega^{(n/2-1)n} \)
  - Combine to compute \( P \) at \( 1, \omega^n, \omega^{2n}, \ldots, \omega^{(n/2-1)n} \)
  - Combine to compute \( P \) at \( -1, \omega^n, \omega^{2n}, \ldots, \omega^{(n/2-1)n} \)

Hints towards FFT: Evaluation at Special Points

- Evaluation of polynomial at 1 point takes \( O(n) \)
  - So 2n points (naively) takes \( O(n^2) \)—no savings
- Key trick:
  - use carefully chosen points where there’s some sharing of work for several points, namely various powers of \( \omega = e^{2\pi i/n} \), \( i = \sqrt{-1} \)
  - Plus more Divide & Conquer.
- Result:
  - both evaluation and interpolation in \( O(n \log n) \) time

The key idea for \( n \) even

- \( P(\omega) = a_0 + a_1 \omega + a_2 \omega^2 + \cdots + a_{n-1} \omega^{n-1} \)
  - \( = a_0 + a_2 \omega^2 + a_4 \omega^4 + \cdots + a_{n-2} \omega^{n-2} \)
  - \( = P_{even}(\omega^2) + \omega P_{odd}(\omega^2) \)
  - \( P(\omega^2) = a_0 + a_1 \omega^2 + a_2 \omega^4 + \cdots + a_{n-1} \omega^{n-2} \)
  - \( = a_0 + a_2 \omega^2 + a_4 \omega^4 + \cdots + a_{n-2} \omega^{n-4} \)
  - \( = P_{even}(\omega^4) + \omega^2 P_{odd}(\omega^4) \)

where \( P_{even}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n-2} \)
and \( P_{odd}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n-2} \)

Analysis and more

- Run-time
  - \( T(n) = 2 T(n/2) + cn \) so \( T(n) = O(n \log n) \)
- So much for evaluation ... what about interpolation?
  - Given
    - \( r_0, R(1), r_1 R(\omega), r_2 R(\omega^2), \ldots, r_{n-1} R(\omega^{n-1}) \)
  - Compute
    - \( c_0, c_1, \ldots, c_{n-1}, \) s.t. \( R(x) = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1} \)
Interpolation = Evaluation: strange but true

- Weird fact:
  - If we define a new polynomial \( S(x) = r_0 + r_1x + r_2x^2 + \ldots + r_nx^n \) where \( r_0, r_1, \ldots, r_n \) are the evaluations of \( R \) at \( 1, \omega, \ldots, \omega^{n-1} \)
  - Then \( c_k = S(\omega^k)/n \) for \( k=0,\ldots,n-1 \)

- So...
  - evaluate \( S \) at \( 1, \omega, \ldots, \omega^{n-1} \) then divide each answer by \( n \) to get \( c_0, \ldots, c_{n-1} \)
  - \( \omega^k \) behaves just like \( \omega \) did so the same \( O(n \log n) \) evaluation algorithm applies!

Divide and Conquer Summary

- Powerful technique, when applicable
- Divide large problem into a few smaller problems of the same type
- Choosing sub-problems of roughly equal size is usually critical
- Examples:
  - Merge sort, quicksort (sort of), polynomial multiplication, FFT, Strassen's matrix multiplication algorithm, powering, binary search, root finding by bisection, …

Why this is called the discrete Fourier transform

- Real Fourier series
  - Given a real valued function \( f \) defined on \([0, 2\pi]\)
  - the Fourier series for \( f \) is given by
    \[
    f(x) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) \cos(mx) \, dx
    \]
  - is the component of \( f \) of frequency \( m \)
  - In signal processing and data compression one ignores all but the components with large \( a_m \) and there aren’t many since

- Complex Fourier series
  - Given a function \( f \) defined on \([0, 2\pi]\)
  - the complex Fourier series for \( f \) is given by
    \[
    f(z) = b_0 + b_1 e^{iz} + b_2 e^{2iz} + \ldots
    \]
  - is the component of \( f \) of frequency \( m \)
  - If we discretize this integral using values at \( 2\pi/n \) equally spaced points between \( 0 \) and \( 2\pi \), we get
    \[
    b_m = \frac{1}{2\pi} \int_{0}^{2\pi} f(z) e^{-imz} \, dz
    \]
  - where \( f_k = f(2k\pi/n) \) just like interpolation!