6.046 Lecture 3

Prof. Shafi Goldwasser
shafi@csail.mit.edu

Call for appointment
617-253-6054
Slides courtesy of

- Leiserson, Demaine, Kellis, Andrew Wayne, myself
How to Multiply

Integers, Polynomials and Matrices
Topics:

- Integer Multiplication
  - Karatsuba’s Algorithm 1962
  - Toom-Cook Algorithm 1963
- Strassen’s Matrix multiplication algorithm 1969
- Powering a number
- Computing the n-th Fibonacci Numbers
Divide-and-conquer design paradigm

1. *Divide* the problem (instance) into sub-problems.

2. *Conquer* the sub-problems by solving them recursively.

3. *Combine* sub-problems solutions.

Applied to numbers, polynomials, & matrices
• **Addition.** Given two $n$-bit integers $a$ and $b$, compute $a + b$.

• **Grade-school.** $\Theta(n)$ bit operations.

Grade-school addition algorithm is **optimal**.
Multiplying large integers

- Given \( n \)-bit integers \( a, b \) (in binary), compute \( c = ab \)
- **Naive (grade-school) algorithm:**
  - Write \( a, b \) in binary
  - Compute \( n \) intermediate products
  - Do \( n \) additions
  - Total work: \( \Theta(n^2) \)
Multiplying large Integers:
Divide and Conquer

Divide: Write \( a = A_1 \cdot 2^{n/2} + A_0 \)
\( b = B_1 \cdot 2^{n/2} + B_0 \)
for \( A_0, A_1, B_0, B_1 \)
n/2 bit integers
Assume \( n=2^k \) w.l.o.g

Conquer: \( ab = A_1 \cdot B_1 \cdot 2^n + A_1 \cdot B_0 \cdot 2^{n/2} + B_1 \cdot A_0 \cdot 2^{n/2} + A_0 \cdot B_0 \)
\( = A_1 \cdot B_1 \cdot 2^n + (A_1 \cdot B_0 + B_1 \cdot A_0) \cdot 2^{n/2} + A_0 \cdot B_0 \)

Reduces to 4 Multiplications of n/2 bit integers (done recursively), 3 additions of n bit integers, and 2 shifts.
Additions and shifts = \( \Theta(n) \)
Multiplying Large Integers: Recurrence for D & C

\[ T(n) = 4T(n/2) + \Theta(n) \]

- # subproblems
- subproblem size
- work dividing and combining
  - compute the products recursively
  - then perform \( \Theta(n) \) additions

\( L = n^{\log_b 4} = n^{\log_2 4} = n^2 \) > \( R = n \)

\( \Rightarrow \) **CASE 1** of Master Theorem

\( \Rightarrow T(n) = \Theta(n^2) \).

No better than the grade school algorithm ???
Multiplying large integers: Karatsuba’s Algorithm

Divide: Write
\[ a = A_1 2^{n/2} + A_0 \]
\[ b = B_1 2^{n/2} + B_0 \]

Conquer: \[ a \times b = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{n/2} + A_0 B_0 \]

Karatsuba’s observation: Only need the sum above, not the individual products, can get the sum via the equation
\[
(A_0 + A_1)(B_0 + B_1) = A_0 B_0 + A_1 B_1 + (A_0 B_1 + B_1 A_0)
\]
Can now get away with only 3 multiplications (in yellow) of \( n/2 \)-bit integers (computed recursively):

Compute \( x = A_1 B_1 \), \( y = A_0 B_0 \), \( z = (A_0 + A_1)(B_0 + B_1) \)
let \( ab = x 2^n + (z - y - x) 2^{n/2} + y \)
Karatsuba: Code

MULTIPLY \((n, a, b)\)

- \(a\) and \(b\) are \(n\)-bit integers
- Assume \(n\) is a power of 2 for simplicity

1. If \(n < 2\) then use grade-school algorithm else
2. \(A_1 \leftarrow a \ \text{div} \ 2^{n/2}\) ; \(B_1 \leftarrow b \ \text{div} \ 2^{n/2}\)
3. \(A_0 \leftarrow a \ \text{mod} \ 2^{n/2}\) ; \(B_0 \leftarrow b \ \text{mod} \ 2^{n/2}\)
4. \(x \leftarrow \text{MULTIPLY}(n/2, A_1, B_1)\)
5. \(y \leftarrow \text{MULTIPLY}(n/2, A_0, B_0)\)
6. \(z \leftarrow \text{MULTIPLY}(n/2, A_1+A_0, B_1+B_0)\)
7. Output \(x \ 2^n + (z-x-y)2^{n/2} + y\)
Recurrence: Magic Time

\[ T(n) = 3T(n/2) + \Theta(n) \]

# subproblems

subproblem size

work adding and subtracting

\[
L = n^{\log_b a} = n^{\log_2 3} = n^{1.58496\ldots} > (R = n)
\]

⇒ CASE 1 of MT

⇒ \( T(n) = \Theta(n^{1.58496}) \).

Much better than \( \Theta(n^2) \)!

Algorithm Folks are a greedy bunch
Ask: Can we do any better?
Toom-Cook Algorithm

Idea 1: Divide integers into 3 parts rather than 2

Write $a = A_2 \cdot 2^{2n/3} + A_1 \cdot 2^{n/3} + A_0$ for $A_i \cdot B_i$

$b = B_2 \cdot 2^{2n/3} + B_1 \cdot 2^{n/2} + B_0$ for $n/3$ bit integers

(assume wlog $a,b = 3^k$)

Naïve Multiplication: takes $9$ multiplications of $n/3$ bit integers, with $T(n) = 9T(n/3) + \Theta(n)$

Goal: Reduce From 9 to 5 multiplication

Idea 2: Represent Integers as polynomials & reduce integer multiplication to polynomial multiplication
Representing Integers as Polynomials

Take numbers $a = A_2 2^{2n/3} + A_1 2^{n/3} + A_0$ and $b = B_2 2^{2n/3} + B_1 2^{n/3} + B_0$ as before.

Define polynomials $A(x) = A_2 x^{2} + A_1 x + A_0$ and $B(x) = B_2 x^{2} + B_1 x + B_0$.

View numbers as $a = A(x)$, $b = B(x)$ where $x = 2^{n/3}$.

Alternative Strategy to compute $a \times b$:

- Compute the product polynomial $C(x) = A(x)B(x)$.
- Evaluate $C(x) = C(2^{n/3}) = A(2^{n/3})B(2^{n/3})$. 
Operations on Polynomials: Coefficient Representation

\[ A(x) = A_2 x^2 + A_1 x + A_0 \]
\[ B(x) = B_2 x^2 + B_1 x + B_0 \]

**Add:** \( O(n) \) arithmetic operations.

\[ A(x) + B(x) = (A_2 + B_2)x^2 + (A_1 + B_1)x + (A_0 + B_0) \]

**Multiply (convolve):** \( O(n^2) \).

\[ A(x) \ast B(x) = (A_2 x^2 + A_1 x + A_0) \ast (B_0 x^2 + B_1 x + B_0) \]

9 multiplications of \( n/3 \) bit coefficients….

Back to where we started? Not Quite

**Theorem** A polynomial of degree $t$ is uniquely specified by its evaluation at $t+1$ distinct points (2 points determine a line etc..)
Idea 3: Multiplication in the point-value representation is more efficient

Use it!

General Lesson: can use different representations where operations are more efficient
Operations on Polynomials: Point-Value Representation

A(x): \[ A(0)=A_0, \quad A(1)=A_1+A_2+A_0, \quad A(2)=4A_2+2A_1+A_0 \]
B(x): \[ B(0)=B_0, \quad B(1)=B_1+B_2+B_0, \quad A(2)=4B_2+2B_1+B_0 \]

Point wise add:
A(x)+B(x): \[ A(0)+B(0), \quad A(1)+B(1), \quad A(2)+B(2) \]

Point wise Multiply:
A(x)*B(x): has degree 4, must be specified at deg+1=5 pts
Evaluate A: \[ A(0), A(1), A(2), A(-1), A(-2) \]
Evaluate B: \[ B(0), B(1), B(2), B(-1), B(-2) \]
Multiply \[ C= A(i)B(i) \text{ for } i=0,1,2,-1,-2 \]

5 Multiplications of n/3 bit number
Rather than 9 !!!

Additions of n/3 bit numbers
From Point-Value to Coefficients: Interpolation

\[
\begin{align*}
\begin{pmatrix}
C(0) \\
C(1) \\
C(-1) \\
C(2) \\
C(-2)
\end{pmatrix}
\quad \begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 & 1 \\
1 & 2 & 4 & 8 & 16 \\
1 & -2 & 4 & -8 & 16
\end{pmatrix}
\begin{pmatrix}
C_0 \\
C_1 \\
C_2 \\
C_3 \\
C_4
\end{pmatrix}
\end{align*}
\]

Invert:

\[
\begin{align*}
\begin{pmatrix}
C_0 \\
C_1 \\
C_2 \\
C_3 \\
C_4
\end{pmatrix}
\quad \begin{pmatrix}
\text{Inverse} \\
\text{of above} \\
\text{Matrix} \\
\text{(which is non-} \\
\text{singular)}
\end{pmatrix}
\begin{pmatrix}
C(0) \\
C(1) \\
C(-1) \\
C(2) \\
C(-2)
\end{pmatrix}
\end{align*}
\]
Putting it all together:
Integer Multiplication

**DIVIDE:** Represent a,b n-bit integers as polynomials A, B of degree 2 such that $A(2^{n/3})=a$, $B(2^{n/3})=b$ and evaluate each at 5 points to a point wise representation [values are $n/3$ bit integers]

**CONQUER:** Multiply point-wise (in 5 points) polynomials A and B (recursively) to obtain poly $C=AB$ of degree 4 in point wise representation

**COMBINE** Interpolate to get polynomial C in coefficient form and evaluate it at $C(2^{n/3})=ab$. 
Putting it all together: Analysis

\[
T(n) = O(n^{1.465}) \text{ better than Karatsuba}
\]
Why stop here?

- We can obtain a sequence of asymptotically faster integer multiplication algorithms by splitting the inputs into more and more pieces

- If we split A and B into k equal parts then the corresponding multiplication algorithm is obtained from an interpolation based polynomial multiplication algorithm of two degree \((k-1)\) polynomials

- Since the product is of degree \(2(k-1)\), we need to evaluate at \(2k-1\) points. Thus, there are \(2k-1\) multiplies and Divide time is still \(O(n)\).

\[
T(n) = (2k-1)T(n/k) + O(n) = \Theta(n^{\log_k 2k-1}) \approx n^\varepsilon \text{ for any } \varepsilon > 1
\]
Fastest Algorithm: FFT based

- $T(n) = \Theta(n \log n \log \log n)$
- Based on the Fast Fourier Transform

- **Basic ideas:** Many similar ideas to Cook-Toom with methods for super fast evaluation and interpolation of polynomials
Matrix multiplication

**Input:** \( A = [a_{ij}] \), \( B = [b_{ij}] \).

**Output:** \( C = [c_{ij}] = A \times B \). \( i, j = 1, 2, \ldots, n \).

\[
\begin{bmatrix}
    c_{11} & c_{12} & \cdots & c_{1n} \\
    c_{21} & c_{22} & \cdots & c_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    c_{n1} & c_{n2} & \cdots & c_{nn}
\end{bmatrix} = \begin{bmatrix}
    a_{11} & a_{12} & \cdots & a_{1n} \\
    a_{21} & a_{22} & \cdots & a_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{n1} & a_{n2} & \cdots & a_{nn}
\end{bmatrix} \begin{bmatrix}
    b_{11} & b_{12} & \cdots & b_{1n} \\
    b_{21} & b_{22} & \cdots & b_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    b_{n1} & b_{n2} & \cdots & b_{nn}
\end{bmatrix}
\]

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}
\]
Standard algorithm

for $i = 1$ to $n$
    do for $j = 1$ to $n$
        do $c_{ij} = 0$
            for $k = 1$ to $n$
                do $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$

Running time = $\Theta(n^3)$

Ignored cost of inner multiplications
Divide-and-conquer algorithm

**Idea:**
n \times n matrix = 2 \times 2 matrix of \((n/2) \times (n/2)\) submatrices:

\[
\begin{bmatrix}
  r & s \\
  t & u \\
\end{bmatrix} = \begin{bmatrix}
  a & b \\
  c & d \\
\end{bmatrix} \cdot \begin{bmatrix}
  e & f \\
  g & h \\
\end{bmatrix}
\]

\[
C = A \times B
\]

\[
\begin{aligned}
  r &= ae + bg \\
  s &= af + bh \\
  t &= ce + dg \\
  u &= cf + dh
\end{aligned}
\]

8 mults of \((n/2) \times (n/2)\) submatrices
4 adds of \((n/2) \times (n/2)\) submatrices
Divide-and-conquer algorithm

**Idea:**

\( n \times n \) matrix = \( 2 \times 2 \) matrix of \((n/2) \times (n/2)\) submatrices:

\[
\begin{bmatrix}
  r & s \\
  t & u
\end{bmatrix} = \begin{bmatrix}
  a & b \\
  c & d
\end{bmatrix} \cdot \begin{bmatrix}
  e & f \\
  g & h
\end{bmatrix}
\]

\( C = A \times B \)

- \( r = ae + bg \) \( 8 \) mults of \((n/2) \times (n/2)\) submatrices
- \( s = af + bh \) recursive
- \( t = ce + dh \) \( 4 \) adds of \((n/2) \times (n/2)\) submatrices
- \( u = cf + dg \)
Analysis of D&C algorithm

\[ T(n) = 8T(n/2) + \Theta(n^2) \]
Analysis of D&C algorithm

\[ T(n) = 8T(n/2) + \Theta(n^2) \]

CASE 1

\[ n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3). \]

No better than the ordinary algorithm.
Strassen's idea

• Multiply $2x2$ matrices with only 7 recursive mults.
Strassen's idea:

- Multiply $2 \times 2$ matrices with only 7 recursive multiplies.

\[
\begin{align*}
P_1 &= a \times (f - h) \\
P_2 &= (a + b) \times h \\
P_3 &= (c + d) \times e \\
P_4 &= d \times (g - e) \\
P_5 &= (a + d) \times (e + h) \\
P_6 &= (b - d) \times (g + h) \\
P_7 &= (a - c) \times (e + f)
\end{align*}
\]
Strassen’s idea

- Multiply 2x2 matrices with only 7 recursive mults.

\[
\begin{align*}
P_1 &= a \times (f - h) \\
P_2 &= (a + b) \times h \\
P_3 &= (c + d) \times e \\
P_4 &= d \times (g - e) \\
P_5 &= (a + d) \times (e + h) \\
P_6 &= (b - d) \times (g + h) \\
P_7 &= (a - c) \times (e + f)
\end{align*}
\]

\[
\begin{align*}
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7
\end{align*}
\]
Strassen's idea

• Multiply 2x2 matrices with only 7 recursive mults.

\[
\begin{align*}
P_1 &= a \times (f - h) \\
P_2 &= (a + b) \times h \\
P_3 &= (c + d) \times e \\
P_4 &= d \times (g - e) \\
P_5 &= (a + d) \times (e + h) \\
P_6 &= (b - d) \times (g + h) \\
P_7 &= (a - c) \times (e + f)
\end{align*}
\]

\[
\begin{align*}
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7
\end{align*}
\]

7 mults, 18 adds/subs.

Note: No reliance on commutativity of mult!
Strassen's idea

- Multiply 2x2 matrices with only 7 recursive mults.

\[
P_1 = a \times (f - h) \\
P_2 = (a + b) \times h \\
P_3 = (c + d) \times e \\
P_4 = d \times (g - e) \\
P_5 = (a + d) \times (e + h) \\
P_6 = (b - d) \times (g + h) \\
P_7 = (a - c) \times (e + f) \\
\]

\[
r = P_5 + P_4 - P_2 + P_6 \\
= (a + d)(e + h) + d(g - e) - (a + b)h + (b - d)(g + h) \\
= ae + ah + de + dh + dg - de - ah - bh + bg + bh - dg - dh \\
= ae + bg
\]
Strassen’s algorithm

1. **Divide:** Partition $A$ and $B$ into $(n/2) \times (n/2)$ sub-matrices. Form terms to be multiplied using $+$ and $-$.

2. **Conquer:** Perform 7 multiplications of $(n/2) \times (n/2)$ sub-matrices recursively.

3. **Combine:** Form $C$ using $+$ and $-$ on $(n/2) \times (n/2)$ sub-matrices.
Strassen's algorithm

1. **Divide:** Partition $A$ and $B$ into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.

2. **Conquer:** Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively.

3. **Combine:** Form $C$ using $+$ and $-$ on $(n/2) \times (n/2)$ submatrices.

$$T(n) = 7 \ T(n/2) + \Theta(n^2)$$
Analysis of Strassen

\[ T(n) = 7 \, T(n/2) + \Theta(n^2) \]

\[ n^{\log_{ba}} = n^{\log_2 7} \gg n^{2.81} \Rightarrow \text{CASE 1} \Rightarrow T(n) = O(n^{{\log_2 7}}). \]
Analysis of Strassen

\[ T(n) = 7T(n/2) + \Theta(n^2) \]

\[ n^{\log_{27} n} \gg n^{2.81} \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^{\log_{27} 7}). \]

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen’s algorithm beats the ordinary algorithm on today’s machines for \( n > 32 \) or so.

Best Algorithm to date: Coppersmith-Winograd \( O(n^{2.376}) \)
Powering a number

**Problem:** Compute $a^b$, where $a, b$ are $n$-bit integers

**Naive algorithm:** $\Theta(b)$ multiplications
(ignoring for simplicity the size of intermediate results and the cost of multiplications) = $\Theta(2^n)$ multiplications
Powering a number

**Problem:** Compute $a^b$, a,b n-bit integers
How many multiplications?

**Divide-and-conquer algorithm:**

$$a^n = \begin{cases} a^{b/2} \cdot a^{b/2} & \text{if } b \text{ is even;} \\ a^{(b-1)/2} \cdot a^{(b-1)/2} \cdot a & \text{if } b \text{ is odd.} \end{cases}$$

$$T(n) = T(n-1) + \Theta(1) \implies T(n) = \Theta(n).$$