CS 161: Design and Analysis of Algorithms

Mark Zhandry
Outline

• Why study algorithms?
• What makes a good algorithm?
• What We’ll See
  – Example: Multiplication
• Asymptotics review
• Course information
Why Study Algorithms?

• Required for CS majors
• Algorithms are everywhere in CS
  – OS (CS 140)
  – Compilers (CS 143)
  – Crypto (CS 155)
  – Etc.
• Algorithms important in other fields
  – Economics (Game Theory)
  – Biology
• Exciting!
What Makes a Good Algorithm

• Computes desired result
  – Always?
  – With high probability?
  – On real-world inputs?

• Uses resources efficiently
  – Time?
  – Space?
  – Disk I/O?
  – Programmer Effort?
CS 161 Concepts

• Data Structures
• Graph Algorithms
• Greedy Algorithms
• Divide & Conquer
• Dynamic Programming
• Linear Programming
• NP-Completeness
Algorithms We’ll See

- Integer/Matrix Multiplication
- Fast Fourier Transform (FFT)
- Shortest Paths
- Sequence Alignment
- Minimum Spanning Trees
- Maximum Flows
Integer Multiplication

• Grade-school algorithm

\[
\begin{array}{c}
3764 \\
\times 689 \\
\hline
33876 \\
\times 9 \\
\hline
33876 \\
\times 8 \\
\hline
301120 \\
\hline
+ 2258400 \\
\hline
2593396
\end{array}
\]

Can we do better? Yes!
Matrix Multiplication

\[
\begin{pmatrix}
a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\
a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n,1} & a_{n,2} & \cdots & a_{n,n}
\end{pmatrix}
\times
\begin{pmatrix}
b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\
b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
b_{n,1} & b_{n,2} & \cdots & b_{n,n}
\end{pmatrix}
= 
\begin{pmatrix}
c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\
c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
c_{n,1} & c_{n,2} & \cdots & c_{n,n}
\end{pmatrix}
\]

- Computing each term uses $n$ multiplications and $(n-1)$ additions
- Total: $\sim 2n^3$ operations

Can we do better? Yes!
Discrete Fourier Transform (DFT)

- Given a sequence \(a=(a_1, a_2, \ldots, a_n)\), the DFT of \(a\) is defined as \(A=(A_1, A_2, \ldots, A_n)\) where

\[
A_k = \sum_{i=1}^{n} a_i \omega_n^{ik}
\]

- Important in signal processing, solving differential equations, polynomial multiplication, and integer multiplication!
Discrete Fourier Transform (DFT)

\[ A_k = \sum_{i=1}^{n} a_i \omega_n^{ik} \]

- Appears to require \( \sim n^2 \) additions and multiplications.

Can we do better? Yes!
Shortest Paths

How do we determine the shortest path without exploring all paths?
Sequence Alignment

• How close is “snowy” to “sunny”?
  – Cost: 3 modifications:

  $\text{S ~ N ~ O ~ W ~ Y}$
  $\text{S U N N ~ Y}$

• Applications:
  – Spell-checkers: If I have a misspelled word, what did I mean?
  – Biology: Identify sections of DNA that are similar
Minimum Spanning Trees
Minimum Spanning Trees

- Applications to network design, approximation algorithms, and more.
Maximum Flows

Graph with labeled edges:
- Source (s) to a: 3
- a to b: 3
- a to d: 2
- b to a: 10
- b to c: 1
- c to b: 4
- c to e: 5
- d to a: 1
- d to e: 2
- e to t: 1
- e to c: 5
- t to e: 2
- t to d: 1
Maximum Flows

- Used in solutions for a host of different problems
Example: Integer Multiplication

- Algorithm 1: Grade-school algorithm

\[
\begin{array}{ccc}
3764 & 3764 & 3764 \\
\times 689 & \times 9 & \times 8 \\
33876 & 33876 & 30112 \\
301120 & & \\
+ 2258400 & & 3764 \\
\hline
2593396 & & 22584
\end{array}
\]
Example: Integer Multiplication

• Algorithm 1: Grade-school algorithm
  – For simplicity, assume running time dominated by single-digit multiplications
  – Number of single-digit multiplications: $n^2$
  – Can we do better?
Example: Integer Multiplication

- Algorithm 2: Recursive algorithm
  - Write n-digit number \( x \) as \( 10^{n/2} x_1 + x_2 \)
    - \( x_1, x_2 \) are \( n/2 \)-digit numbers
  - Write \( y \) as \( 10^{n/2} y_1 + y_2 \)
  - \( xy = 10^n x_1 y_1 + 10^{n/2}(x_1 y_2 + x_2 y_1) + x_2 y_2 \)
  - Can multiply \( n \)-digit numbers by performing 4 multiplications of \( n/2 \)-digit numbers.
Example: Integer Multiplication

- Algorithm 2: Recursive algorithm
  - \[ xy = 10^n x_1 y_1 + 10^{n/2}(x_1 y_2 + x_2 y_1) + x_2 y_2 \]
  - Claim: the number of single-digit multiplications for algorithm 2 is \( n^2 \).
    - True for \( n=1 \)
    - Assume true for \( n/2 \). Then \(#(\text{multiplications for } n\text{-bits}) = 4 \#(\text{multiplications for } n/2 \text{ bits}) = 4 \cdot (n/2)^2 = n^2.\)
Example: Integer Multiplication

• Which algorithm is better?
  – Both require $n^2$ single-digit multiplications, so running time almost the same
Example: Integer Multiplication

• Algorithm 3: Another Recursive Algorithm
  – We need the quantities $x_1y_1$, $(x_1y_2 + x_2y_1)$, and $x_2y_2$
  – Can we compute $x_1y_2 + x_2y_1$ using one multiplication?
  – Gauss: $x_1y_2 + x_2y_1 = (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2$
    • Already have $x_1y_1$ and $x_2y_2$!
Example: Integer Multiplication

- **Algorithm 3: Another Recursive Algorithm**
  - Compute $P = x_1 y_1$, $Q = x_2 y_2$.
  - Let $x_3 = x_1 + x_2$, $y_3 = y_1 + y_2$.
  - Compute $R = x_3 y_3$
  - Let $S = R - P - Q$.
  - $xy = 10^n P + 10^{n/2} S + Q$
Example: Integer Multiplication

• Algorithm 3: Another Recursive Algorithm
  – Replaced 4 multiplications with 3, seems like it should be faster.
  – Not so simple:
    • One of the multiplications is slightly larger.
    • Added some extra overhead (additions and subtractions)
  – Possible to show $< 4n^{1.6}$ single-digit multiplications
Example: Integer Multiplication

• Which algorithm is faster?
  – $4n^{1.6} > n^2$ for $n < 32$.
  – $4n^{1.6} < n^2$ for $n > 32$.

• Algorithm 3 slower for $n < 32$, faster for $n > 32$.

• How do we compare?
How to Compare Algorithms

• Say we want to compute a function $f(n)$
  – Algorithm 1 runs in $100,000n$ seconds
  – Algorithm 2 runs in $n^2$ seconds

• Which algorithm is better?
  – Algorithm 2 runs faster if $n<100,000$
  – Algorithm 1 runs faster if $n>100,000$
  – The ratio of run times goes to infinity
How to Compare Algorithms

• Say we want to compute a function $g(n)$
  – Algorithm 1 runs in $20n$ seconds
  – Algorithm 2 runs in $10n+30$ seconds

• Which algorithm is better?
  – Algorithm 2 is faster than Algorithm 1 when $n>3$
  – However, it is only twice as fast
Asymptotics/Big Oh

- We say Algorithm 1 is “at least as fast as” Algorithm 2 if, for large enough inputs, Algorithm 1 is at most a constant factor slower than Algorithm 2.

- For this class, we will compare algorithms using “at least as fast as”.
Big Oh

\[ O(f(n)) = \{ g(n) : \exists c, n_0 \text{ such that } g(n) \leq c f(n) \forall n \geq n_0 \} \]

- So \( g(n) \in O(f(n)) \) if
  - There are \( c \) and \( n_0 \) such that
  - \( g(n) \leq c f(n) \) for all \( n \geq n_0 \).
- Sometimes write \( g(n) = O(f(n)) \).
Big Oh Example 1

• \( f(n) = 10n+30, \ g(n) = 20n \)

• \( f(n) \in O(g(n)) \)
  – Proof: Let \( c = 1, \ n_0 = 3. \)
  – If \( n \geq n_0, \) then \( f(n)=10n+30 \leq 20n = c \ g(n). \)

• \( g(n) \in O(f(n)) \)
  – Proof: Let \( c = 2, \ n_0 = 1. \)
  – If \( n \geq n_0, \) then \( g(n)=20n \leq 2 \times (10n+30) = c \ f(n). \)
Big Oh Example 1

\[ f(n) \]
\[ g(n) \]
\[ 2f(n) \]
Big Oh Example 2

• $f(n) = 100,000n$, $g(n) = n^2$.

• $f(n) \in O(g(n))$
  – Proof: Let $c = 1$, $n_0 = 100,000$.
  – If $n \geq n_0$, then $f(n) = 100,000n \leq n^2 = c \cdot g(n)$.

• $g(n) \in O(f(n))$?

Big Oh Example 2

• \( f(n) = 100,000n, \ g(n) = n^2. \)
• If \( g(n) \in O(f(n)) \), then
  – There are \( c, n_0 \) such that
  – \( g(n) \leq c \ f(n) \) for all \( n \geq n_0. \)
• But, for any \( c \), suppose \( n > 100,000c. \)
  – Then \( g(n) = n^2 > 100,000c \ n=c \ f(n) \)
• So \( g(n) \notin O(f(n)). \)
Big Oh Example 2

The graph illustrates the growth of different functions as $n$ increases. The functions shown are:
- $g(n)$
- $16f(n)$
- $8f(n)$
- $4f(n)$
- $2f(n)$
- $f(n)$

The $x$-axis represents values of $n$ ranging from 500,000 to $2 \times 10^6$, and the $y$-axis shows the corresponding values ranging from $1 \times 10^{12}$ to $4 \times 10^{12}$. The lines show how each function grows relative to $f(n)$.
Big Oh Facts

- Big Oh allows us to ignore constant factors:
  - For any constant $c$, $c f(n) \in O(f(n))$
- Big Oh allows us to ignore lower order terms:
  - If $f(n) \in O(g(n))$, then $g(n)+f(n) \in O(g(n))$
Big Oh Facts

• If $a \leq b$, then $n^a \in O(n^b)$
  – Proof: $n^a \leq n^b$ for all $n \geq 1$.

• If $1 < a \leq b$, then $a^n \in O(b^n)$
  – Proof: $a^n \leq b^n$ for all $n > 0$.

• For any $a,b$, $\log_a(n) \in O(\log_b(n))$
  – Proof: $\log_a(n) = \log_b(n) \log_a(b)$ for all $n > 0$.

We can usually ignore the base for logarithms.
We will use $\log(n)$ to denote $\log_2(n)$
Big Oh Facts

• For any $a>0$, $b>1$, $n^a \in O(b^n)$
  – We’ll see a proof in a bit

• For any $a>0$, $\log n \in O(n^a)$
  – We’ll see a proof in a bit
Big Oh Facts

• If $f(n) \in O(g(n))$, and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$
  
  – Proof: There are $c$, $n_0$ such that $f(n) \leq c g(n)$ for $n \geq n_0$.
  – There are $c'$, $n_0'$ such that $g(n) \leq c' f(n)$ for $n \geq n_0'$.
  – Let $c''=c c'$, $n_0''=\text{Max}(n_0, n_0')$.
  – If $n \geq n_0''$, then $f(n) \leq c g(n) \leq c c' h(n) =c'' h(n)$.

• $f(n) \in O(f(n))$: let $c=1$, $n_0=1$.

Big Oh inclusion is a sort of “≤” on functions
Big Omega

$$\Omega(f(n)) = \{ g(n) : \exists c, n_0 \text{ such that } g(n) \geq c f(n) \forall n \geq n_0 \}$$

• So $$g(n) \in \Omega(f(n))$$ if
  – There are $$c$$ and $$n_0$$ such that
  – $$g(n) \geq c f(n)$$ for all $$n \geq n_0$$.

• Sometimes write $$g(n) = \Omega(f(n))$$. 
Big Omega

• $g(n) \in \Omega(f(n))$ is equivalent to $f(n) \in O(g(n))$
  – Proof: $f(n) \in O(g(n))$ means there is $c, n_0$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$.
  – Then $g(n) \geq (1/c) \cdot f(n)$ for $n \geq n_0$.
  – Similar proof for other direction.

Big Omega inclusion is a sort of “$\geq$” on functions
Big Theta

- \( g(n) \in \Theta(f(n)) \) if:
  - \( g(n) \in O(f(n)) \) and
  - \( g(n) \in \Omega(f(n)) \) (or equivalently \( f(n) \in O(g(n)) \))

- Note: the constants \( c, n_0 \) may be different for showing \( g(n) \in O(f(n)) \) and \( g(n) \in \Omega(f(n)) \)

- \( g(n) \in \Theta(f(n)) \) is equivalent to \( f(n) \in \Theta(g(n)) \)

Big Theta inclusion is a sort of “=” on functions
Little o, omega

• \( f(n) \in o(g(n)) \) if, for all \( c \), there is an \( n_0 \):
  \[ f(n) < c \ g(n) \text{ for all } n \geq n_0. \]

• \( f(n) \in \omega(g(n)) \) if, for all \( c \), there is an \( n_0 \):
  \[ f(n) > c \ g(n) \text{ for all } n \geq n_0. \]

• \( f(n) \in o(g(n)) \) is equivalent to \( g(n) \in \omega(f(n)) \)

Little o and omega are a sort of “<” and “>” for functions
Big Oh Is Not Total!

• It is not true that \( f(n) \in O(g(n)) \) or \( g(n) \in O(f(n)) \)
  \[- f(n) = n^{1+\cos(\pi n)}, \quad g(n) = n \]
• Similarly, \( f(n) \notin O(g(n)) \) does not imply that \( f(n) \in \omega(g(n)) \)
Limit Test

• Let $c = \lim_{n \to \infty} f(n)/g(n)$
  – If $c = 0$, then $f(n) \in o(g(n))$
  – If $0 < c < \infty$, then $f(n) \in \Theta(g(n))$.
  – If $c = \infty$, then $f(n) \in \omega(g(n))$
For any $a > 0$, $\log n \in o(n^a)$

- Proof: Let $c = \lim_{n \to \infty} \log n / n^a$
- L'Hôpital's rule:
  $$c = \lim_{n \to \infty} (1/n)/(a \cdot n^{a-1}) = \lim_{n \to \infty} 1/(a \cdot n^a) = 0.$$
Limit Test

• For any \( a > 0, \ b > 1, \ n^a \in o(b^n) \)
  – Proof: Let \( c = \lim_{n \to \infty} n^a / b^n \)
  – L'Hôpital's rule:
    \[
    c = \lim_{n \to \infty} \frac{(a \ n^{a-1})}{(b^n \ln b)} = \left(\frac{a}{\ln b}\right) \lim_{n \to \infty} n^{a-1} / b^n
    \]
  – Repeat until \( a \leq 0 \)
  – We have that \( c = \text{const} \times \lim_{n \to \infty} n^{a'}/b^n \) for \( a' \leq 0 \)
    • Numerator goes to 0, denominator goes to infinity
    • Thus \( c = 0 \).
Other Notation

• $f(n) \in g(n) + o(h(n))$ means $f(n) - g(n) \in o(h(n))$
  – Common example: $f(n) \in g(n) + o(1)$ means $f(n)$ converges to $g(n)$.

• $f(n) \in g(n)^{O(h(n))}$ means there is some function $h'(n) \in O(h(n))$ such that $f(n) = g(n)^{h'(n)}$.
  – Common example: $f(n) \in n^{O(1)}$ means $f(n) \in O(n^a)$ for some $a$. 
Course Information

• Lectures: WMF 2:15-3:30
• Text: *Algorithm Design* by Kleinberg and Tardos
• Piazza: [http://piazza.com/class#summer2012/cs161](http://piazza.com/class#summer2012/cs161)
• Staff Email: [cs161-summer2012-staff@lists.stanford.edu](mailto:cs161-summer2012-staff@lists.stanford.edu)
Grading

• Homeworks: 50%
  – Five weekly homeworks
• Midterm: 20%
  – Wednesday, July 25\(^{\text{th}}\) in class
• Final: 30%
  – Friday, August 17\(^{\text{th}}\), Location: TBA
Homework

• Assigned each Wednesday, due following Friday
  – Exceptions: no homework due week of midterm or final
  – Due at beginning of class (2:15PM)
• May work in groups of up to three
  – Write up solutions individually
• One 72-hour extension
• Can submit digitally to cs161-summer2012-submissions@lists.stanford.edu
• See website for policies