Algorithm analysis

William Hendrix

Lecture 1

Today

- Syllabus
- Proof review
- RAM model of computation
- Big-Oh notation
  - Motivation
  - Formal definitions
  - Properties
- Analyzing algorithms
- Summations
- Recursive analysis

Syllabus

- Read at: sit.instructure.com
- Contact: whendrix@stevens.edu
- Office hours:
  - TR 12:30-1:30 pm
  - GS 251
- CAs: TBD
- Office hours: TBD

- Course objectives, grading scale, exams, etc.
- Class participation
- Feedback form
- Slides

What will we learn in this class?

- **How to determine if an algorithm is efficient**
  - RAM model
  - Big-Oh definition and properties
- **How to improve your algorithms by organizing data**
  - Stacks and queues
  - Binary search trees
    - Balanced BSTs
  - Priority queues and heaps
  - Hash tables
- **How to develop your own algorithms**
  - Greedy algorithms
  - Divide-and-conquer
  - Dynamic programming
  - Graph traversals
- **Classical sort, search, and graph algorithms**

RAM model of computation

- Set of assumptions that make analysis more reasonable

Assumptions

1. All "basic" operations (assignment, arithmetic, branching, memory access, etc.) take 1 operation
   - Loops and functions do not qualify
2. We have "infinite" memory

Cons

- Different operations take different number of clock cycles
  - Cache locality has significant impact
- Virtual memory can slow performance

Pros

- Can actually analyze algorithms

RAM model example

Input: $data$: array of integers
Input: n: size of $data$
Output: index min such that $data[min] \leq data[j]$, for all j from 1 to n

```
Algorithm: FindMin
min = 1;
for i = 2 to n do
    if data[i] < data[min] then
        min = i;
    end
end
return min;
```

Big-Oh notation

- Technique for *abstracting away details* of complexity
  - Can be used for time complexity, space complexity, etc.

- **Main idea:** most important aspect of complexity is *how fast it grows* relative to input size
  - Focus on asymptotic (eventual) growth rate
  - "Fast" functions will eventually pass "slow" functions for large *n*
  - Coefficients only matter if growth rate is similar
  - Predicting behavior for small *n* is difficult and often pointless

- Big-Oh notation
  - Organizes growth rates into classes
  - Three main symbols: $O(f(n)), \Omega(f(n)), \Theta(f(n))$
    - Analogous to "at most", "at least", and "similar to" *f(n)*

Justification of Big-Oh

- Algorithm runtime with c=1, running at 1 GHz:

n=	$\lg(n)$	n	$n\lg(n)$	n^2	n^3	2^n	$n!$
10	3 ns	10 ns	33 ns	100 ns	1 µs	1 µs	3.6 ms
20	4 ns	20 ns	86 ns	400 ns	8 µs	1 ms	77 yrs
30	5 ns	30 ns	147 ns	900 ns	27 µs	1 s	
40	5 ns	40 ns	213 ns	1.6 µs	64 µs	18.3 min	
50	6 ns	50 ns	282 ns	2.5 µs	125 µs	13 days	
100	7 ns	100 ns	644 ns	10 µs	1 ms		
1,000	10 ns	1 µs	9.97 µs	1 ms	1 s		
1,000,000	20 ns	1 ms	19.9 ms	16.7 min	31.7 yrs		
1,000,000,000	30 ns	1 s	29.9 s	31.7 yrs			

"Fails" at: Never! billions millions 10k 40ish 16ish

Lesson: on large data, coefficients not as important

Big-Oh

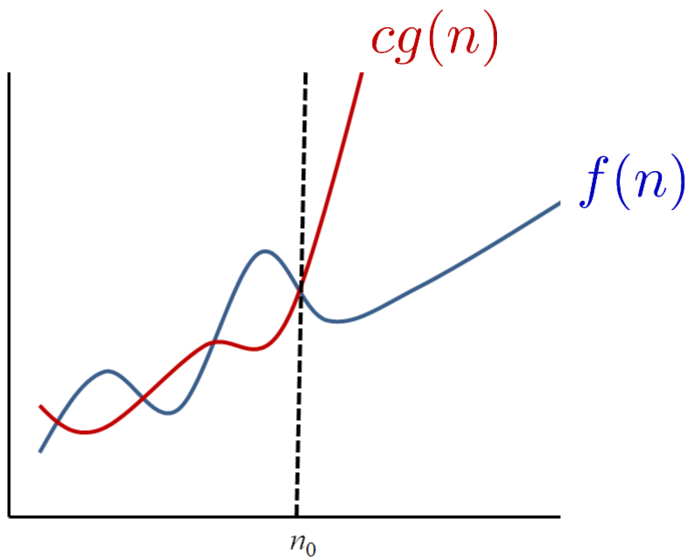
- Upper bound *("at most")*

$f(n) = O(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Big-Oh in pictures

- Upper bound *("at most")*

$f(n) = O(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.



Translation: *f* is smaller than some multiple of *g* <u>eventually</u> (and <u>stays smaller</u>)

Small values of *n* don't matter

f isn't growing faster than *g*

Big-Oh

- Upper bound *("at most")*

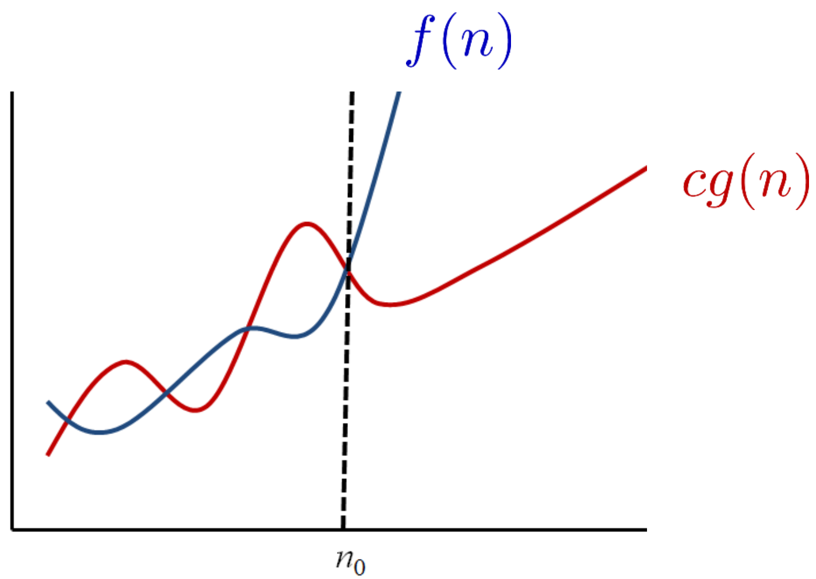
$f(n) = O(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

- We say "*g(n)* dominates *f(n)*" when $f(n) = O(g(n))$
- Notation weirdness:
  - O, Ω, and Θ are classes (sets) of functions
  - BUT: we use = to assign class, not $\in$
- **Example**
  - Prove that $7n^2 + 19n - 4444 = O(n^2)$.

Big-Omega picture

- Lower bound *("at least")*

$f(n) = \Omega(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.



Translation: f is bigger than some multiple of g eventually (and stays bigger)

Small values of n don't matter

g isn't growing faster than f

Big-Omega

- Lower bound *("at least")*

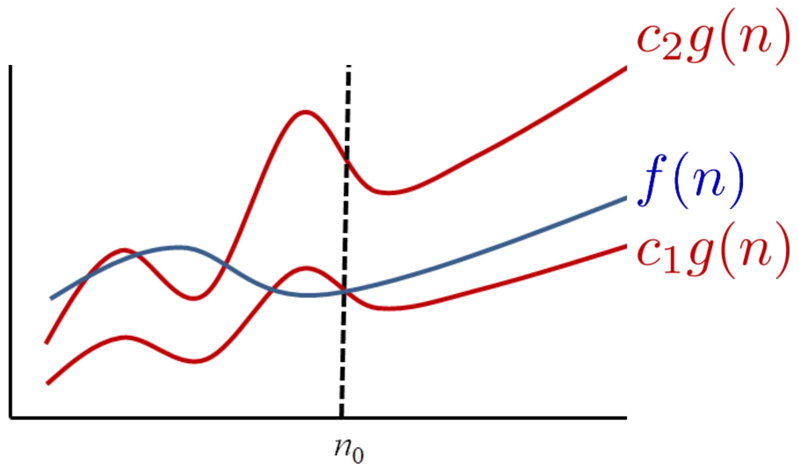
$f(n) = \Omega(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

- **Example**
  - Prove that $7n^2 + 19n - 4444 = \Omega(n^2)$.

Big-Theta picture

- Upper *and* lower bound *("same rate as")*

$f(n) = \Theta(g(n))$ if and only if there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.



Translation: *f* can be sandwiched between two multiples of *g* eventually (and stays between them)

f and *g* are growing at the same rate

Small values of n don't matter

Big-Theta

- Upper *and* lower bound *("same rate as")*

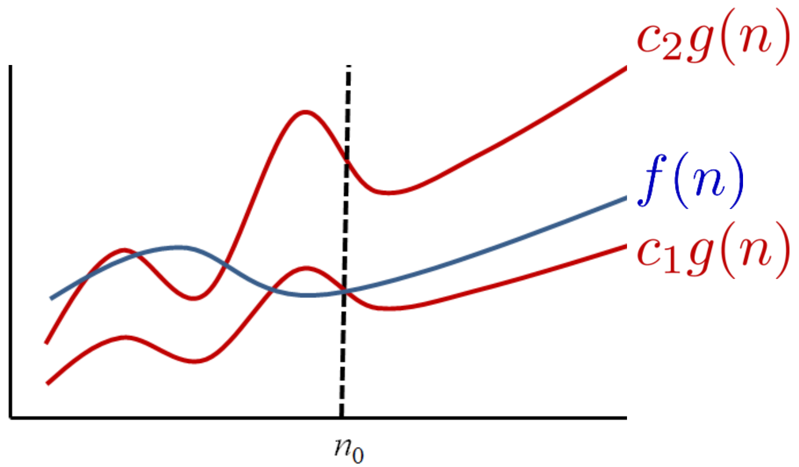
$f(n) = \Theta(g(n))$ if and only if there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$.

- **Example**
  - Prove that $7n^2 + 19n - 4444 = \Theta(n^2)$.

Big-Theta picture

- Upper *and* lower bound *("same rate as")*

$f(n) = \Theta(g(n))$ if and only if there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.



Translation: *f* can be sandwiched between two multiples of *g* eventually (and stays between them)

f and *g* are growing at the same rate

Small values of *n* don't matter

Big-Oh example

- Use the *formal definition* of Big-Oh to prove:

$$\sum_{i=1}^{n} i = O(n^2)$$

Big-Oh example (series)

- Prove that $\sum_{i=1}^{n} i = \Omega(n^2)$.

Analysis of Big-Oh

Pros

- Provides a useful summary of the growth rate of the complexity
- Compact
- Simple: eight classes cover most useful algorithms

$$O(1) \ll O(\lg n) \ll O(n) \ll O(n \lg n) \ll O(n^2) \ll O(n^3) \ll O(2^n) \ll O(n!)$$

Cons

- Ignores contributions from coefficients and lower-order terms
- Doesn't rank algorithms with same growth rate
- Doesn't rank algorithms on small inputs
- Some of the "best" algorithms have extremely large coefficients, making them impractical for many purposes

Connection to calculus

- You can also determine O, Ω, and Θ by limits:

g grows faster ⟶ $$\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0 \quad \to f(n) = O(g(n))$$

Actually $f(n) = o(g(n))$

Same growth rate ⟶ $$\lim_{n\to\infty}\frac{f(n)}{g(n)} \in (0,\infty) \to f(n) = \Theta(g(n))$$

g grows slower ⟶ $$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \infty \quad \to f(n) = \Omega(g(n))$$

Actually $f(n) = \omega(g(n))$

- Standard rules for taking limits apply
  - Including L'Hôpital's Rule

Formal definition extra practice

- Use the *formal definition* of Big-Theta to prove:

  For any $x > 0$, if $f(n) = \Theta(g(n))$, then $xf(n) = \Theta(g(n))$

Properties of Big-Oh notation

- Reflexivity

  $f(n) = O(f(n))$, $f(n) = \Omega(f(n))$, and $f(n) = \Theta(f(n))$

- Antisymmetry

  $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

  $f(n) = O(g(n))$ and $f(n) = \Omega(g(n)) \Leftrightarrow f(n) = \Theta(g(n))$

- Symmetry (Θ only)

  $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

- Transitivity

  $f(n) = O(g(n))$ and $g(n) = O(h(n)) \rightarrow f(n) = O(h(n))$
  $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \rightarrow f(n) = \Omega(h(n))$
  $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \rightarrow f(n) = \Theta(h(n))$

- Alternately:

  $O(O(h(n))) = O(h(n))$
  $\Omega(\Omega(h(n))) = \Omega(h(n))$
  $\Theta(\Theta(h(n))) = \Theta(h(n))$

Combination properties

- Envelopment
  - Addition

$$O(f(n)) + O(g(n)) = O(f(n) + g(n))$$
$$\Omega(f(n)) + \Omega(g(n)) = \Omega(f(n) + g(n))$$
$$\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$$

  - Multiplication

$$O(f(n))O(g(n)) = O(f(n)g(n))$$
$$\Omega(f(n))\Omega(g(n)) = \Omega(f(n)g(n))$$
$$\Theta(f(n))\Theta(g(n)) = \Theta(f(n)g(n))$$

- All three ignore constant coefficients

$$\forall x > 0, \quad \begin{array}{l} f(n) = O(g(n)) \rightarrow xf(n) = O(g(n)) \\ f(n) = \Omega(g(n)) \rightarrow xf(n) = \Omega(g(n)) \\ f(n) = \Theta(g(n)) \rightarrow xf(n) = \Theta(g(n)) \end{array}$$

- Only the largest term matters

$$f(n) = O(g(n)) \rightarrow O(f(n) + g(n)) = O(g(n))$$
$$f(n) = O(g(n)) \rightarrow \Omega(f(n) + g(n)) = \Omega(g(n))$$
$$f(n) = O(g(n)) \rightarrow \Theta(f(n) + g(n)) = \Theta(g(n))$$

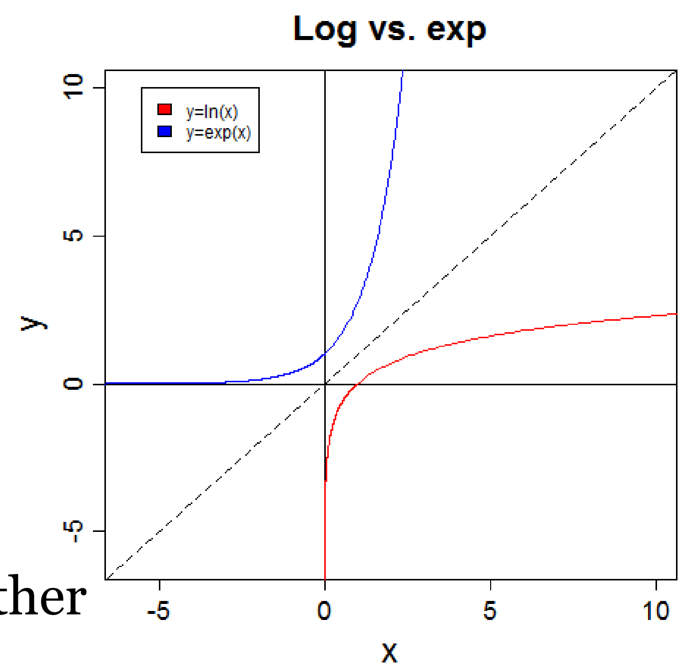
Big-Oh properties example

- Use Big-Oh properties to establish the following:

1. If $f(n) = 13n^2 + 1234n + 91.2n\sqrt{n}$, then $f(n) = \Theta(n^2)$. Use the facts that $n = O(n^2)$ and $\sqrt{n} = O(n)$.

Revenge of the logarithms

- **Logarithm:** inverse exponential function
$$y = \ln x \Leftrightarrow x = e^y$$
- Natural log (ln): inverse of e^x
- Logarithms of other base: $\log_b(x)$
  – $\log_2(x)$ is very common in algorithms
- Computing logs of other bases
  – $\log_b(x) = \frac{\ln x}{\ln b}$
  – All logs are *scalar multiples* of one another



- **Log properties**

Base 2 → $\lg(ab) = \lg(a) + \lg(b)$

$$\lg(a^b) = b \lg(a)$$

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\lg n)$$

Because

$$2^A 2^B = 2^{A+B}$$

$$\left(2^A\right)^b = 2^{Ab}$$

$$\int_1^n \frac{1}{x} dx = \ln n$$

Logarithm property example

- Prove that $\lg(n!) = \Theta(n \lg(n))$.

Coming up

- Big-Oh properties
- Algorithm analysis
- Recursive analysis
- Data structures

- **Recommended reading (today):** Sections 1.1 and 1.2
  – *Practice problems:* R-1.3, R-1.7, R-1.19, R-1.21, C-1.8
- **Recommended reading (next week):** Section 1.3
  – *Practice problems:* R-1.11, R-1.15, R-1.26, C-1.3, A-1.4