Applying Big-Oh notation

William Hendrix

Lecture 6

Today

- Review
- Big-Oh practice
- Big-Oh motivation
- Applying Big-Oh
- Recurrences

Formal definitions

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

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

$f(n) = \Theta(g(n))$ 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 $n \ge n_0$.

- *Analogy:* *O, Ω,* and *Θ* act like ≤, ≥, and =
- Most algorithms we discuss will belong to a few classes:
  $O(1) \ll O(\lg n) \ll O(n) \ll O(n \lg n) \ll O(n^2) \ll O(n^3) \ll O(2^n) \ll O(n!)$

Properties of Big-Oh notation

- Transitivity

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

- Equivalence rules

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

- Reflexivity and symmetry

$$f(n) = O(f(n)),\ f(n) = \Omega(f(n)), \text{ and } f(n) = \Theta(f(n))$$
$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

- All three ignore constant coefficients

$$\forall x > 0, xf(n) = O(f(n)),\ xf(n) = \Omega(f(n)), \text{ and}$$
$$xf(n) = \Theta(f(n))$$

- Only the largest term matters

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

Another important property

- Envelopment
  - Addition

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

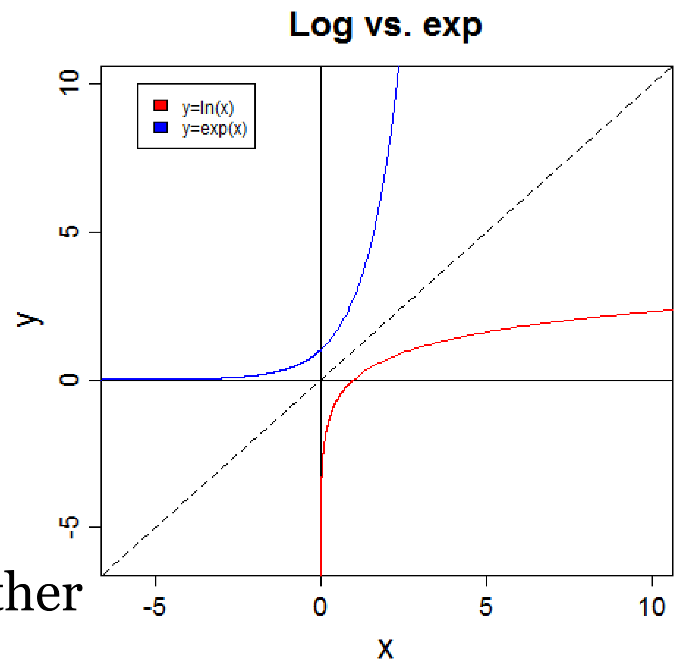
  - Multiplication

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

- Informally, you can move sums and products inside Big-Oh

Logarithms review

- **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**

Because

Base 2 → $\lg(ab) = \lg(a) + \lg(b)$	$2^A 2^B = 2^{A+B}$
$\lg(a^b) = b \lg(a)$	$\left(2^A\right)^b = 2^{Ab}$
$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\lg n)$	$\int_1^n \frac{1}{x} dx = \ln n$
$\lg(n!) = O(n \lg n)$	Properties 1 above

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	664 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			

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	664 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!	trillions	millions	10k	40ish	16ish

Lesson: on large data, coefficients are not likely to matter

Analysis of Big-Oh

Pros

- Provides a useful summary of the growth rate of the complexity
- Growth rate is *very* important
- Compact
- Simple: eight classes cover most useful algorithms

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 very large coefficients, making them impractical for many purposes

Proving Big-Oh

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

- **In symbols:**

  $$f(n) = O(n) \Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n \geq n_0, f(n) \leq cg(n)$$

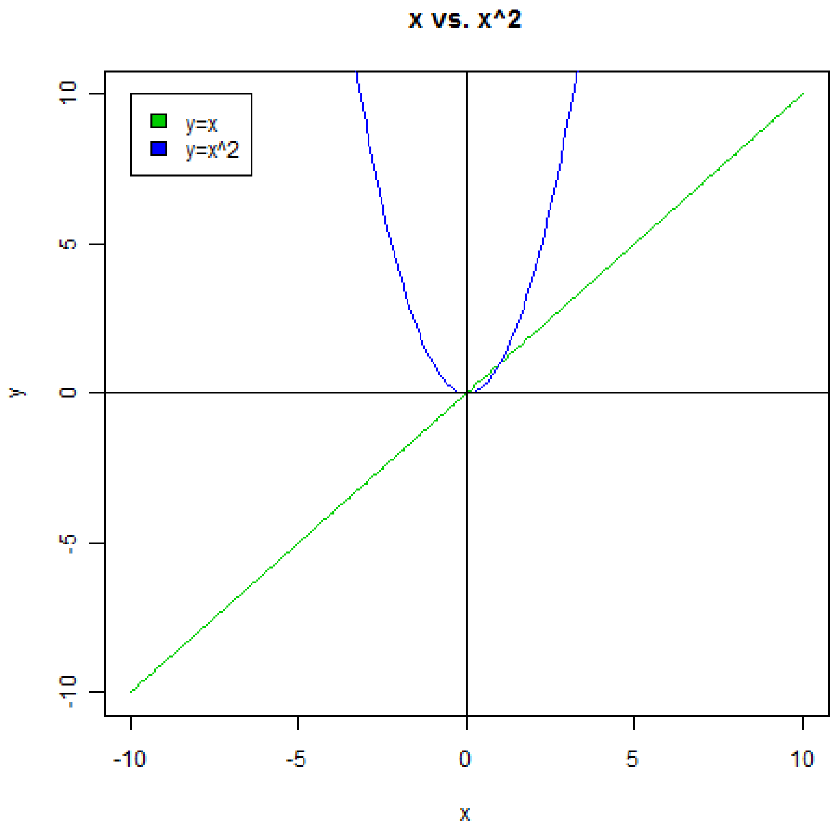
- To prove existence, we generally *find* the variable values that satisfy the equation
  - How big does multiple of *g(n)* need to be?
  - When does *g(n)* pass *f(n)*?
- **Basic strategy**
  - Write out assumptions
  - Think about what properties might apply, or what you know about functions (draw a picture!)
  - Manipulate until you get to the form

    $$f(n) \leq (\underline{\qquad})g(n), \forall n \geq (\underline{\qquad})$$

  - Let $c = \underline{\qquad}$ and $n_0 = \underline{\qquad}$ and conclude proof

Big-Oh example

- Prove that $n = O(n^2)$.



$$\textit{Goal:}\ n \le (\underline{\hspace{2cm}})n^2, \forall n \ge (\underline{\hspace{2cm}})$$

Big-Oh example

- Prove that $n = O(n^2)$.

Obviously true!

$1 \leq n, \forall n \geq 1$ — Divide by n on both sides

$n \leq n^2, \forall n \geq 1$

$n \leq (1)n^2, \forall n \geq (1)$ — *Hypothesis:* $c = 1$, $n_0 = 1$

Goal: $n \leq (\underline{\hspace{2cm}})n^2, \forall n \geq (\underline{\hspace{2cm}})$

Proof. Note that $1 \leq n$ for all $n \geq 1$.

Since $n \geq 1$, we can multiply both sides by n: $n \leq n^2$, for all $n \geq 1$.

Let $c = 1$ and $n_0 = 1$.
So, $n \leq cn^2$, for all $n \geq n_0$.

Since there are positive constants c and n_0 such that $n \leq cn^2$ for all $n \geq n_0$, $n = O(n^2)$. $\square$

More complex example

- Prove that $n = O(n^k)$, $\underbrace{\text{for all } k \geq 1}_{\text{Induction keywords!}}$.

Proof sketch:

$$(n = O(n^1)) \quad \text{Reflexive property}$$

$$(n = O(n^2)) \quad \text{Previous proof}$$

$$(n = O(n^k) \to n = O(n^{k+1})) \quad n \leq cn^k, \forall n \geq n_0$$

$$\text{Show: } n \leq (\underline{\qquad\quad})n^{k+1}, \forall n \geq (\underline{\qquad\quad})$$

Since $n \geq 1$, multiply RHS by n

$$n \leq cn^{k+1}, \forall n \geq n_0 \qquad \square$$

Back to our earlier example...

```
data:  an array of integers to find the min
n:     the number of values in data
```

Min algorithm:

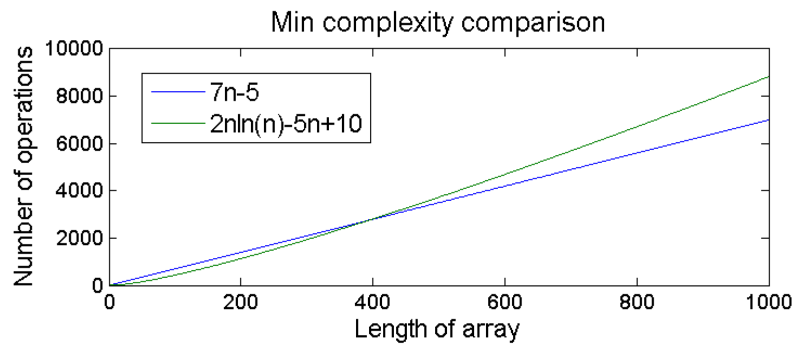
```
min = 1
for i = 2 to n
  if data[i] < data[min]
    min = i
  end
end
return min
```

Ops per line | Times executed
... ...

Total ops: $\leq$ 7(n-1) + 2 = 7n - 5

Question: is this better or worse than an algorithm that takes at most $2n \ln n - 5n + 10$ ops?

Better unless n < 396



Worst case: *O(n)*

Don't need to count instructions!

Other algorithm: *O(n*lg*(n))*

Conclusion: Our algorithm is better for sufficiently large *n*.

Algorithm analysis

- Identify loops and function calls
  - Everything else is *O(1)*
- *For loops*:
  - Estimate loop body running time, $O(f(n,i))$
  - Estimate number of iterations, $O(g(n))$
  - Actual time taken: $O\left(\sum_{i=1}^{g(n)} f(n,i)\right)$
  - Rough estimate: $O\left(g(n)\max_i\{f(n,i)\}\right)$
- Instructions in sequence: add up the complexity
  - Blocks of *O(1)* instructions: *O(1)* time
  - Overall complexity: largest loop or function call complexity
- Analyze helper functions separately
- Recursive functions: set up a recurrence and solve

Loop example

- **Algorithm:** Selection Sort

Input:

```
data:  an array of integers to sort
n:     the number of values in data
```

Output: permutation of `data` such that `data[1] ≤ ... ≤ data[n]`

Pseudocode:

```
for i = 1 to n
  Let m be the location of the min value in the array data[i..n]
  Swap data[i] and data[m]
end
return data
```

Loop example solution

- Selection Sort is $O(n^2)$.

Proof. Note that the **for** loop in lines 1–4 will iterate n times. On iteration i, line 2 calls the `min` function on `data[i..n]`. As we have already proven, `min` takes $O(k)$ time, where k is the size of the input array. Since `data[i..n]` has length $n-i+1$, this call will take $O(n-i+1)$ time. Line 3 can be solved with three assignment statements, which will take $O(1)$ time. Thus, iteration i of the loop will take $O(n-i+1)+O(1)=O(n-i)$ time. In total, the loop will take $O(\sum_{i=1}^{n} n-i) = (n-1)+(n-2)+...+1+0$ time. This sum equals $\sum_{i=1}^{n-1} i$, which is $O(n^2)$. Since the **for** loop takes $O(n^2)$ time total and the **return** statement in line 5 takes $O(1)$ time, the total time for Selection Sort is $O(n^2)+O(1)=O(n^2)$. $\square$

Review: recurrences

Linear nonhomogeneous recurrences with constant coefficients

$$T(n) = c_1T(n-1) + c_2T(n-2) + \ldots + c_kT(n-k) + f(n)s^n$$

1. Write down the characteristic polynomial

Coming up

- Finish Big-Oh
- Data structures

- **Homework 4** is due tonight
- **Homework 5** is due Thursday

- **Recommended readings:** Chapter 2
- **Practice problems:** Any from section 2.10 (p. 57)