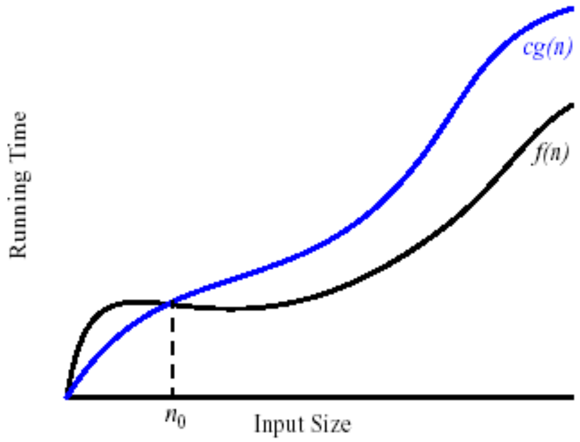
CSSE 230 Day 2



Growable Arrays Continued

Big-Oh notation

Submit Growable Array exercise

Agenda and goals

- Growable Array recap
- Big-Oh definition

- After today, you'll be able to
  - Use the term *amortized* appropriately in analysis
  - State the formal definition of big-Oh notation

Announcements and FAQ

- You will not usually need the textbook in class
- All should do piazza introduction post (a few students left)
- Turn in GrowableArrays now.
- Quiz problems 1-5. Do on your own, then compare with a neighbor.

You must demonstrate programming competence on exams to succeed

- See syllabus for exam weighting and caveats.
- Evening exams (Mon Week 3, Tue Week 8)
- Think of every program you write as a practice test
  - For example, HW4 → Exam 2

Review these as needed

- Logarithms and Exponents
  - properties of logarithms:

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b(x/y) = \log_b x - \log_b y$$

$$\log_b x^\alpha = \alpha \log_b x$$

$$\log_b x = \frac{\log_a x}{\log_a b}$$

  - properties of exponentials:

$$a^{(b+c)} = a^b a^c$$

$$a^{bc} = (a^b)^c$$

$$a^b/a^c = a^{(b-c)}$$

$$b = a^{\log_a b}$$

$$b^c = a^{c*\log_a b}$$

Practice with exponentials and logs

(Do these with a friend after class, not to turn in)

Simplify: Note that log n (without a specified) base means $\log_2 n$.

Also, log n is an abbreviation for log(n).

1. $\log (2\ n \log n)$
2. $\log(n/2)$
3. $\log (\text{sqrt}(n))$
4. $\log (\log (\text{sqrt}(n)))$

5. $\log_4 n$
6. $2^{2 \log n}$
7. if $n=2^{3k} - 1$, solve for k.

Where do logs come from in algorithm analysis?

Solutions

No peeking!

Simplify: Note that log n (without a specified) base means $\log_2 n$.
Also, log n is an abbreviation for log(n).

1. $1+\log n + \log \log n$
2. $\log n - 1$
3. $\frac{1}{2} \log n$
4. $-1 + \log \log n$
5. $(\log n) / 2$
6. n^2
7. $n+1=2^{3k}$
   $\log(n+1)=3k$
   $k= \log(n+1)/3$

A: Any time we cut things in half at each step (like binary search or mergesort)

Questions?

- About Homework 1?
  - Aim to complete tonight, since it is due after next class
  - It is substantial
  - The last problem (the table) is worth lots of points!

- About the Syllabus?

Homework 1 help

How many times does sum++ run?

```
for (i = 4; i < n; i++)
    for (j = 0; j <= n; j++)
        sum++;
```

Why is this one so easy? (does the inner loop depend on outer loop?)

What if inner were (j = 0; j <= i ; j++) ?

Homework 1 help

How many times does sum++ run?

```
for (i = 1; i <= n; i *= 2)
   sum++;
```

Be precise, using floor/ceiling as needed, to get full credit.

Warm Up and Stretching thoughts

- Short but intense! ~50 lines of code total in our solutions

- Be sure to read the description of how it will be graded. Note how style will be graded.

- **Demo:** Use Git to check out the project

- **Demo:** Running the JUnit tests for test, file, package, and project

Growable Arrays Exercise

Daring to double

Growable Arrays Table

N	E_N	**Answers for problem 2**
4	0	0
5	0	0
6	5	5
7	5	5 + 6 = 11
10	5	5 + 6 + 7 + 8 + 9 = 35
11	5 + 10 = 15	5 + 6 + 7 + 8 + 9 + 10 = 45
20	15	sum(i, i=5..19) = 180 **using Maple**
21	5 + 10 + 20 = 35	sum(i, i=5..20) = 200
40	35	sum(i, i=5..39) = 770
41	5 + 10 + 20 + 40 = 75	sum(i, i=5..40) = 810

Doubling the Size

- Doubling each time:
  - Assume that $N = 5(2^k) + 1$.
- Total # of array elements copied:

k	N	#copies
0	6	5
1	11	5 + 10 = 15
2	21	5 + 10 + 20 = 35
3	41	5 + 10 + 20 + 40 = 75
4	81	5 + 10 + 20 + 40 + 80 = 155
k	$= 5(2^k) + 1$	$5(1 + 2 + 4 + 8 + \ldots + 2^k)$

Express as a closed-form expression in terms of K, then express in terms of N

Doubling the Size (solution)

- Assume that $N = 5 (2^k) + 1$.
- Total # of array elements copied
  $= 5(1 + 2 + 4 + 8 + \ldots + 2^k)$
- **Do in terms of k, then in terms of N**

Adding One Each Time

- Total # of array elements copied:

N	#copies
6	5
7	5 + 6
8	5 + 6 + 7
9	5 + 6 + 7 + 8
10	5 + 6 + 7 + 8 + 9
N	???

Express as a closed-form expression in terms of N

- Q6-7

Conclusions

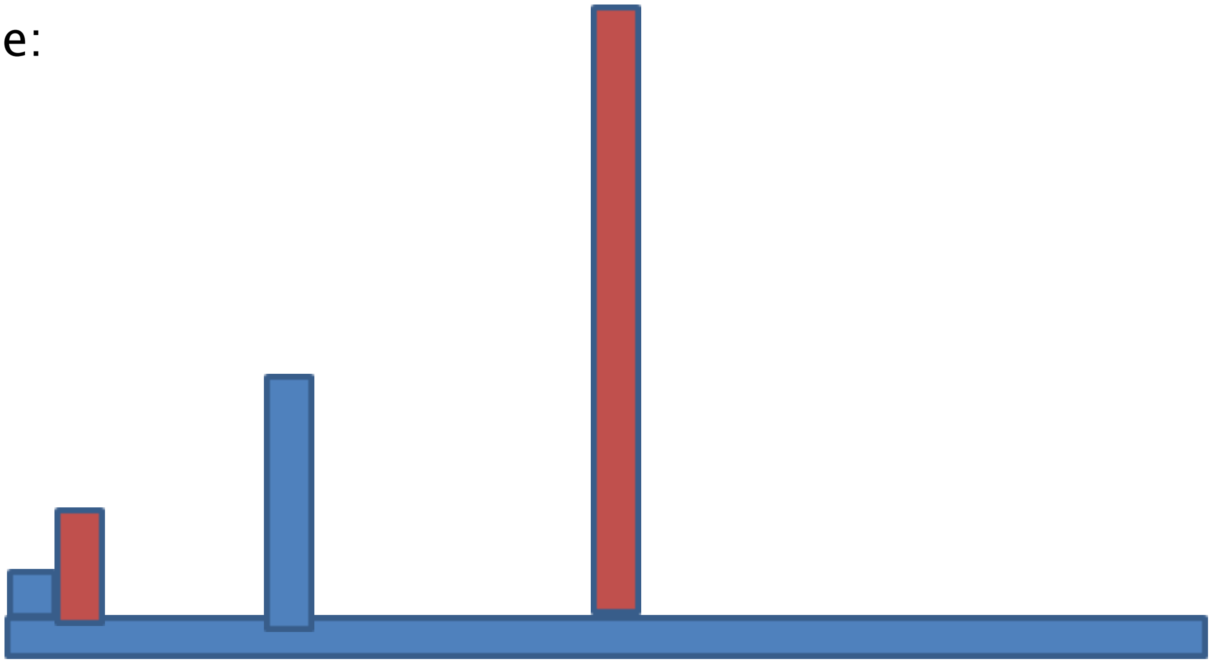
- What's the **amortized** cost of adding an additional string…
  - in the doubling case?
  - in the add-one case?

Amortized cost means the "average per-operation cost" while adding to a single GrowableArray many times.

- So which should we use?

Worst-case vs amortized cost for adding an element to an array using the doubling scheme

Worst-case:
O(n)



amortized:
O(1)

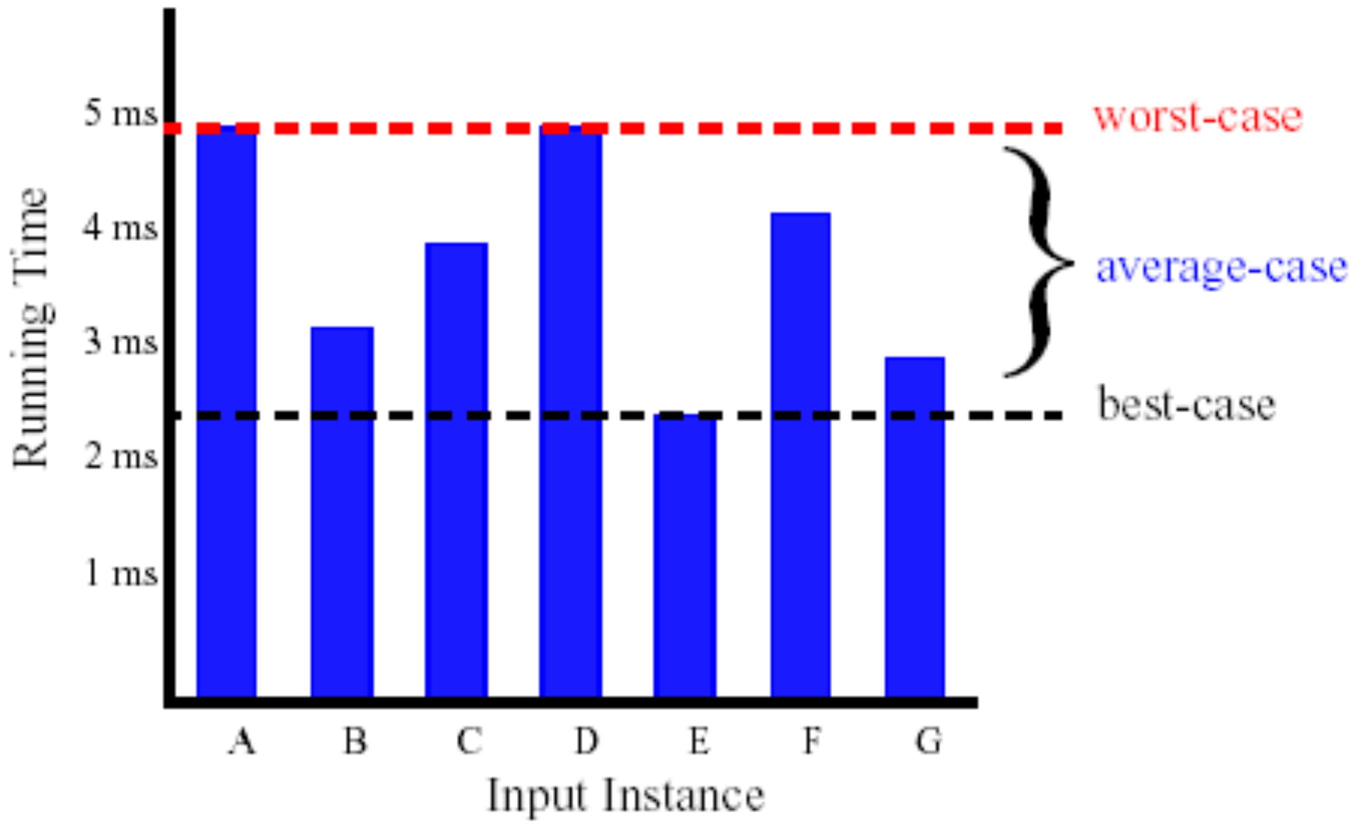
Note: average case means averaged over *input domain*, amortized cost means averaged over *many uses*.

Algorithm Analysis: Running Time

Running Times

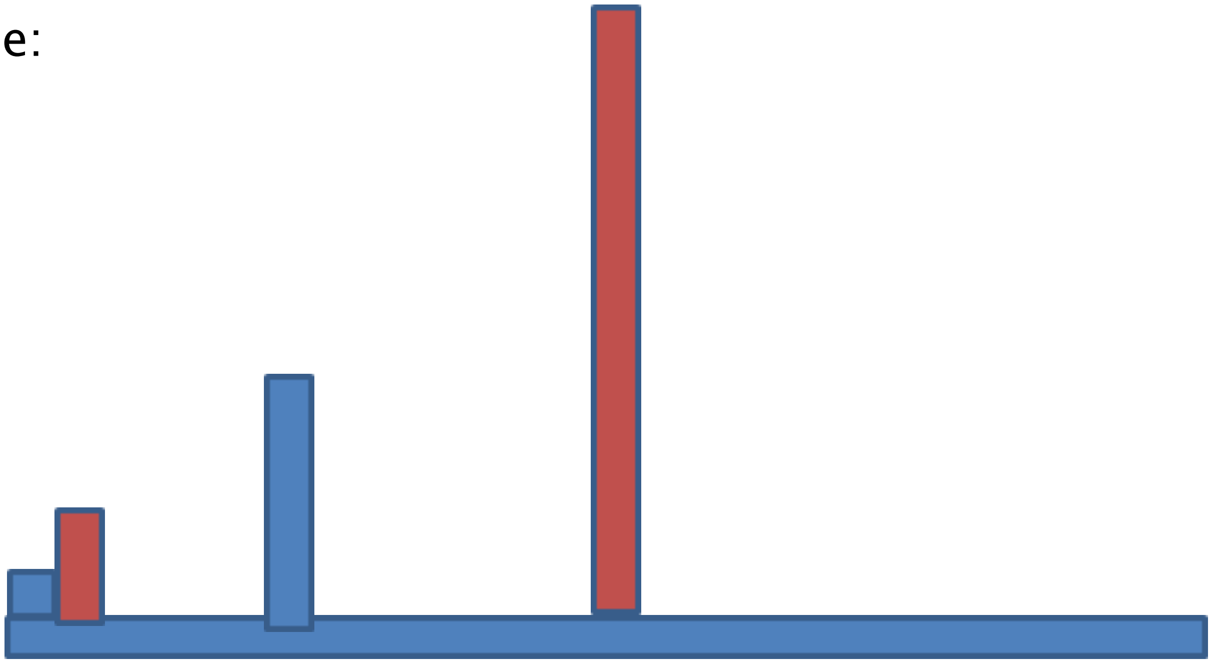
- Algorithms may have different *time complexity* on different data sets
- What do we mean by "Worst Case"?
- What do we mean by "Average Case"?
- What are some application domains where knowing the Worst Case time complexity would be important?
- http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

Average Case and Worst Case



Worst-case vs amortized cost for adding an element to an array using the doubling scheme

Worst-case:
O(n)



amortized:
O(1)

Note: average case means averaged over *input domain*, amortized cost means averaged over *many uses*.

Notation for Asymptotic Analysis

Big-Oh

Asymptotic Analysis

- We only care what happens when N gets large
- Is the function linear? quadratic? exponential?

Figure 5.1

Running times for small inputs

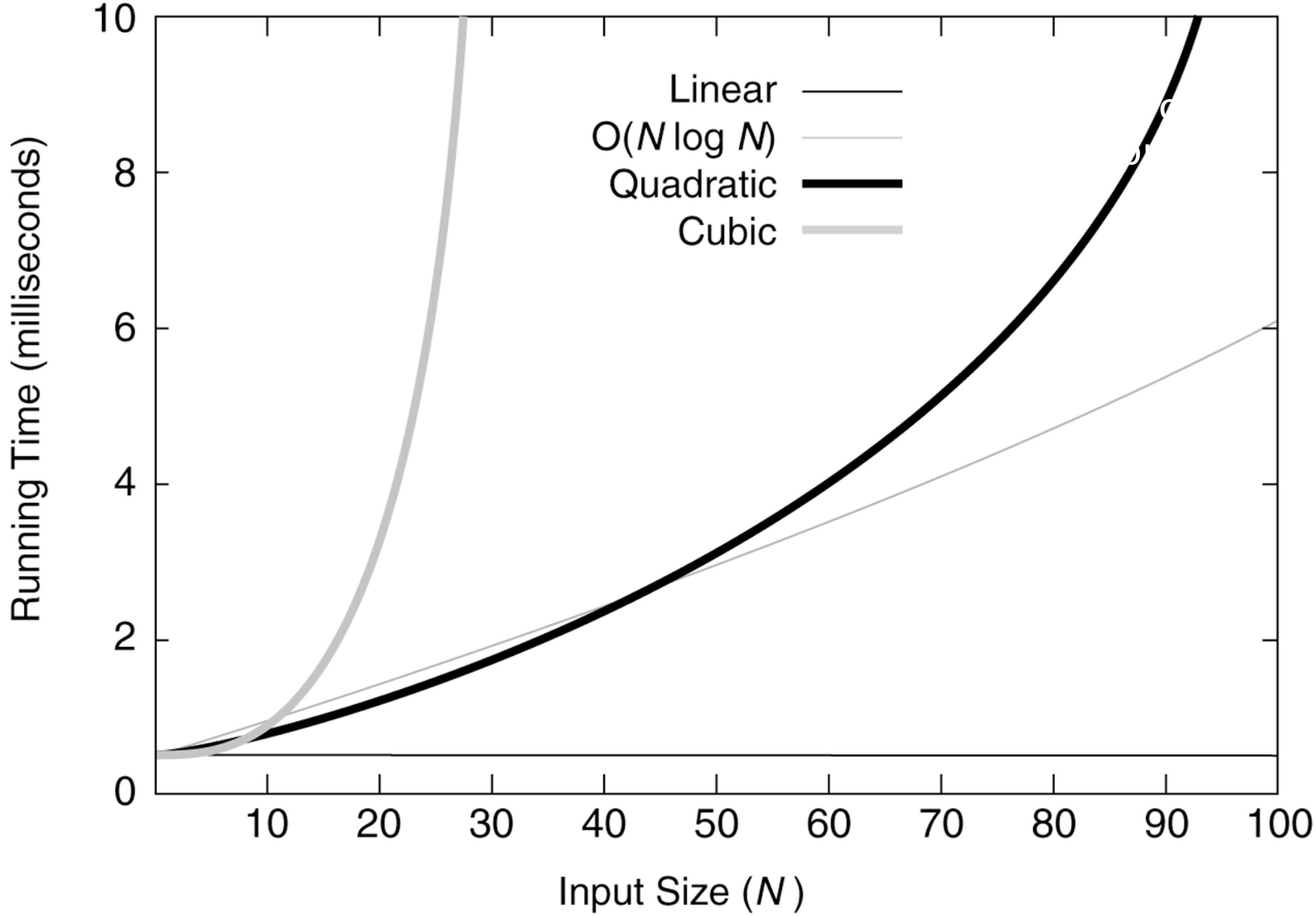


Figure 5.2

Running times for moderate inputs

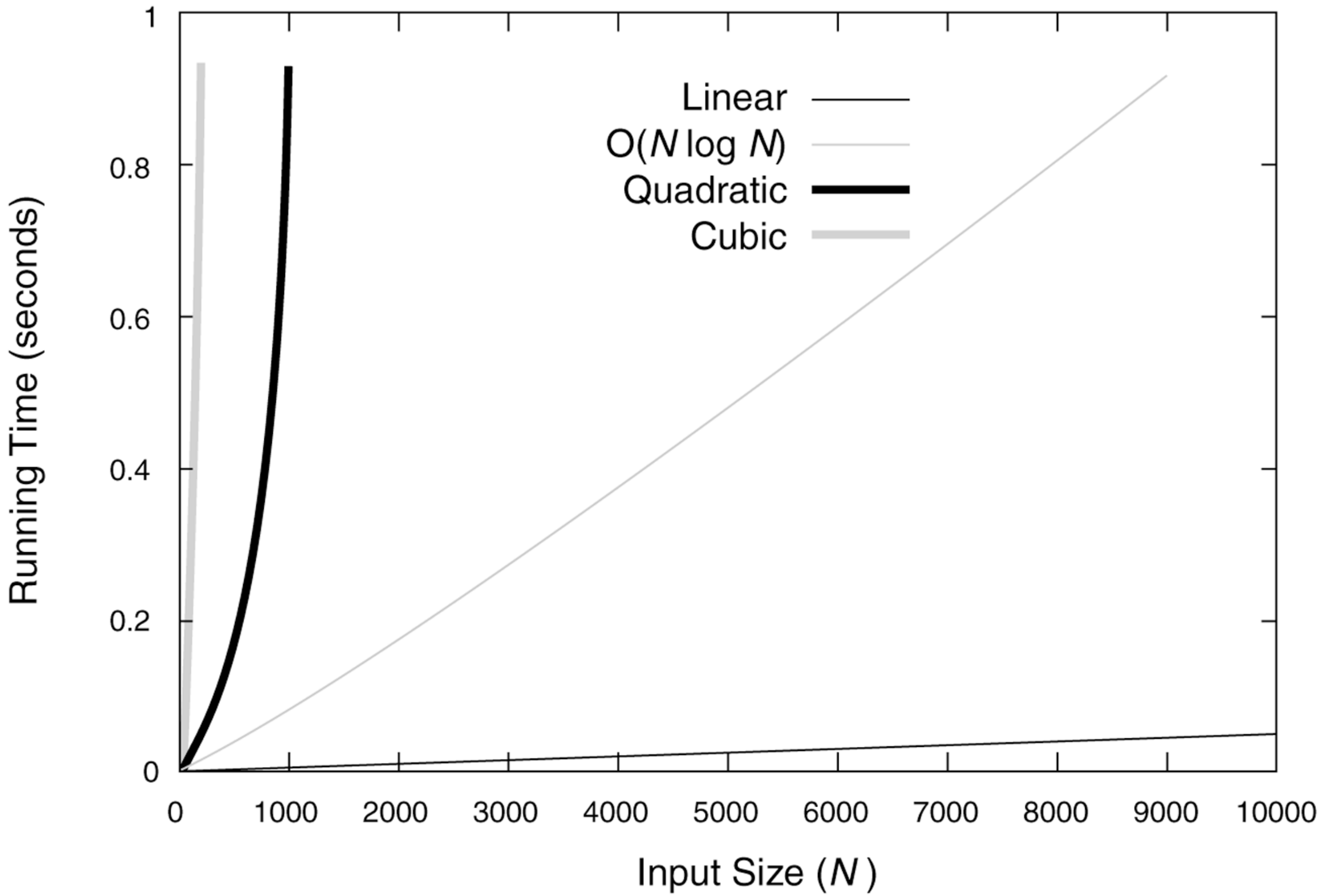


Figure 5.3

Functions in order of increasing growth rate

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

The answer to most big-Oh questions is one of these functions

a.k.a "log linear"

Simple Rule for Big-Oh (informal)

- Drop lower order terms and constant factors
- $7n - 3$ is $O(n)$
- $8n^2\log n + 5n^2 + n$ is $O(n^2\log n)$

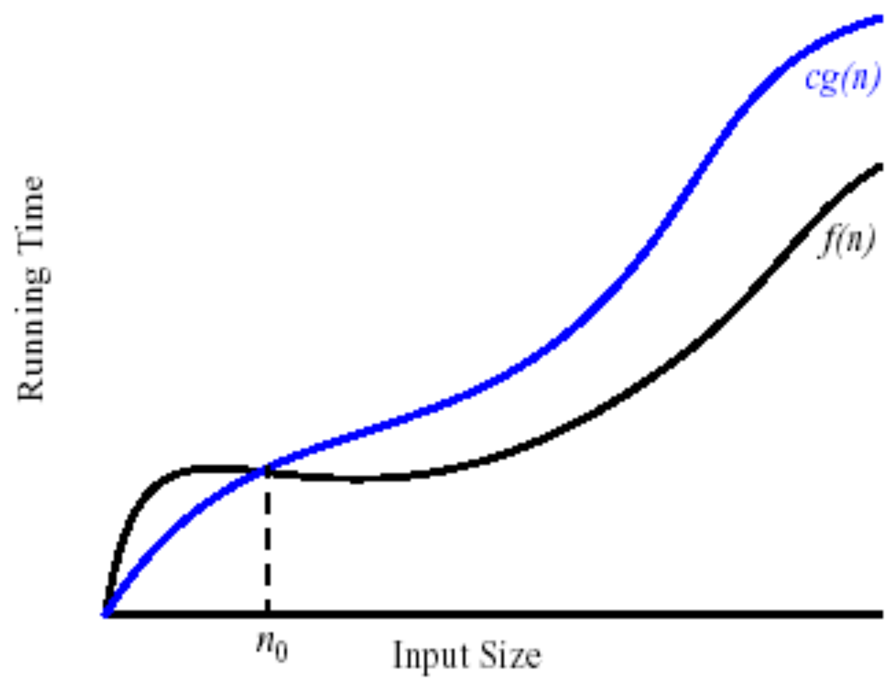
Q8

Formal Definition of Big-Oh

- Given functions f(n) and g(n), we say that f(n) is O(g(n)) if and only if there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$f(n) \leq c\, g(n) \text{ for all } n \geq n_0.$$

- For this to make sense, f(n) and g(n) should be functions over non-negative integers.



To *prove* a Big-O relationship, find 2 constants and show they work

Q9

- A function f(n) is (in) O(g(n)) if there exist two positive constants **c** and $\mathbf{n_0}$ such that *for all* $n \geq n_0$, $f(n) \leq c\, g(n)$
- **Q: How to prove that f(n) is O(g(n))?**
  A: Give c and $\mathbf{n_0}$

Assume that all functions have non-negative values, and that we only care about n≥0. For any function g(n), O(g(n)) is a set of functions.

- Ex: f(n) = 4n + 15, g(n) = ???.

To *prove* Big Oh, find 2 constants and show they work

- A function f(n) is (in) O(g(n)) if there exist two positive constants **c** and $\mathbf{n_0}$ such that for all $n \geq n_0$, $f(n) \leq c\, g(n)$
- **Q: How to prove that f(n) is O(g(n))?**
  A: Give c and $\mathbf{n_0}$
- Ex 2: f(n) = n + sin(n), g(n) = ???