Algorithm Analysis I: Asymptotics

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Announcements

- Exercise #1 (midnight) and Quiz #0 (6pm) "due" tonight
- Project 1's Checkpoint is *due on Tuesday
  - *actually due, not airquotes "due"
  - Checkpoint is a Gradescope-administered survey
- You should have reached out and/or contacted your P1 partner by now!
  - If not, sending an email rn isn't a bad idea …
  - Don't forget to check your spam folder!
- Upvote questions at pollev.com/cse332

Learning Objectives

- Understand the different ways to analyze an algorithm (eg, space vs time; best vs worst)
- Describe *why* we use asymptotic analysis as a way of comparing two algorithms, and when we *do not* want to use it
- Be able to produce a O, θ, or Ω bound for any iterative algorithm

Lecture Outline

- **Comparing two algorithms**
- Analyzing Code
- When the Input is Large: Asymptotic Analysis
- Review: Logarithms and Exponents
- Big-Oh Definitions

What do we care about?

- Correctness:
  - Does the algorithm do what is intended.
- Performance:
  - Speed **time complexity**
  - Memory **space complexity**
- Why analyze?
  - To make good design decisions
  - Enable you to look at an algorithm (or code) and identify the bottlenecks, etc.

Q: How should we compare two algorithms?

A: How should we compare two algorithms?

- Uh, why NOT just run the program and time it??
  - Too much ***variability***, not reliable or ***portable***:
    - Hardware: processor(s), memory, etc.
    - OS, Java version, libraries, drivers
    - Other programs running
    - Implementation dependent
  - Choice of input
    - Testing (inexhaustive) may ***miss*** worst-case input
    - Timing does not ***explain*** relative timing among inputs (what happens when *n* doubles in size)

- Often want to evaluate an ***algorithm***, not an implementation
  - Even ***before*** creating the implementation ("coding it up")

Comparing Algorithms (1 of 2)

- When is one ***algorithm*** (not ***implementation***) better than another?
  - Various possible answers (clarity, security, …)
  - But a big one is ***performance***: for sufficiently large inputs, runs in less time (our focus) or less space than the other
- Large inputs (n) because probably any algorithm is "plenty good" for small inputs (if *n* is 10, probably anything is fast enough)
- Want a model that's:

  - ***Independent*** of CPU speed, programming language, coding tricks, etc.
  - General and rigorous, ***complementary*** to "coding it up and timing it on some test cases": can do analysis before coding!

Comparing Algorithms (2 of 2)

- Want a model that's:
  - ***Independent*** of CPU speed, programming language, coding tricks, etc.
  - General and rigorous, ***complementary*** to "coding it up and timing it on some test cases": can do analysis before coding!
  - ***Descriptive of large inputs***; most algorithms are "good enough" for small inputs
    - If *n* is 10, probably anything is fast enough
    - But what consider to be "large"?

Lecture Outline

- Comparing two algorithms
- **Analyzing Code**
- When the Input is Large: Asymptotic Analysis
- Review: Logarithms and Exponents
- Big-Oh Definitions

Analyzing Code (1 of 2)

- We abstract away the computer by counting "operations" (time complexity) and "elements" (space complexity)
  - Remember: "Independent of CPU speed, programming language, coding tricks, etc."

- Basic ***elements*** take "some amount of" ***constant space***
  - Integers in an array
  - Nodes in a linked list
  - Etc.
  - (This is an *approximation of reality*: a very useful "lie".)

Analyzing Code (2 of 2)

- Basic ***operations*** take "some amount of" ***constant time***
  - Arithmetic
  - Assignment
  - Access one Java field **or array index**
  - Etc.
  - (Again, this is an *approximation of reality*)

Consecutive statements	Sum of time of each statement
Loops	Num iterations * time for loop body
Recurrence	Solve recurrence equation
Function Calls	Time of function's body
Conditionals	Time of condition + time of {slower/faster} branch

Which Branch To Analyze?

- Case Analysis != Asymptotic Analysis

- We'll start by focusing on two cases:
  - ***Worst-case complexity***: max # steps algorithm takes on "most challenging" input of size N
  - ***Best-case complexity*****:** min # steps algorithm takes on "easiest" input of size N

- Unless otherwise stated, we usually refer to the ***worst case***
  - But there are uses for other types of analyses
  - So for now we'll analyze the ***slower branch***

Examples

① 6

```
b = b + 5
c = b / a
b = c + 100
```

② 1 + 4n

```
for (i = 0; i < n; i++) {
  sum++;
}
```

③ 1 + 1 + 4n

```
if (j < 5) {
  sum++;
} else {
  for (i = 0; i < n; i++) {
    sum++;
  }
}
```

Another Example

```
int coolFunction(int n, int sum) {
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      sum++;
    }
  }
  print "This program is great!";
  for (i = 0; i < n; i++) {
    sum++;
  }
  return sum
}
```

2

$16n^2 + 5$

$4n+1$

$4(4n+1)+1$

1

$4n+1$

1

Analyzing Loops, Formally

- We use summations to quantify the runtime

```
for (i = 0; i < n; i++) {
  sum++;
}
```

1 1 2

$$\sum_{i=0}^{n-1} 4 \Rightarrow 4n$$

Example Problem

Find an integer in a *sorted* array

2	3	5	16	37	50	73	75	126

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
   ???
}
```

Example Solution: Linear Search

Find an integer in a *sorted* array

2	3	5	16	37	50	73	75	126

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
  for(int i=0; i < arr.length; ++i)
    if(arr[i] == k)
      return true;
  return false;
}
```

5n

Best case: find(2)

Worst case: find(126)

Example Solution: Linear Search Runtimes

Find an integer in a *sorted* array

2	3	5	16	37	50	73	75	126

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
 for(int i=0; i < arr.length; ++i)
    if(arr[i] == k)
      return true;
 return false;
}
```

Best case: 6 "ish" steps = $O(1)$

Worst case: 5 "ish" * (arr.length) + 1
= O(arr.length)

Lecture Outline

- Comparing two algorithms
- Analyzing Code
- **When the Input is Large: Asymptotic Analysis**
- Review: Logarithms and Exponents
- Big-Oh Definitions

Remember a faster search algorithm?

From 143: "binary search is logN"

Ignoring Constant Factors (1 of 2)

- Binary search is $O(\log n)$ and linear is $O(n)$
  - But which is actually ***faster***?
  - Depending on ***constant factors*** and ***size of n***, in a particular situation, linear search could be faster!

- Depends on constant factors:
  - How *many* assignments, additions, etc. for each *n*

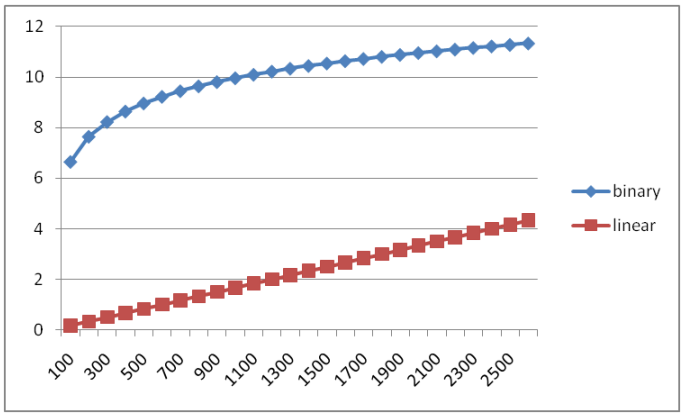
- Depends on size of *n:*
  - Remember: "Descriptive of large inputs; most algorithms are 'good enough' for small inputs"
  - Each data structure's and algorithm's behavior can vary for every finite N
  - So we pick N→∞ as our definition of "large"

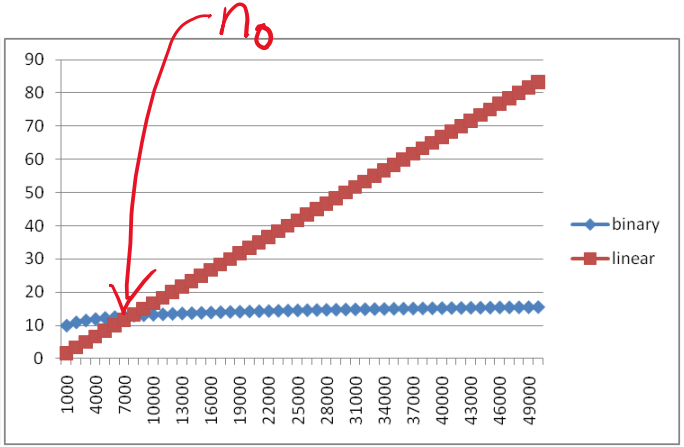
Ignoring Constant Factors (2 of 2)

- How formalize the intuitive idea of how an algorithm behaves as N→∞?
  - There exists some n_0 such that for all $n > n_0$ ***binary search "wins"***
- Let's play with a couple plots to get some intuition…

Example: Binary Search vs Linear Search

- Let's "help" linear search "win"
  - Run it on a computer 100x as fast (say 2018 model vs. 1990)
  - Use a new compiler/language that is 3x as fast
  - Be a clever programmer to eliminate half the work
  - Each iteration is 600x as fast as in binary search





When we're dealing with infinity, constants and lower-order terms don't meaningfully add to the final result

Intuitive Simplifications

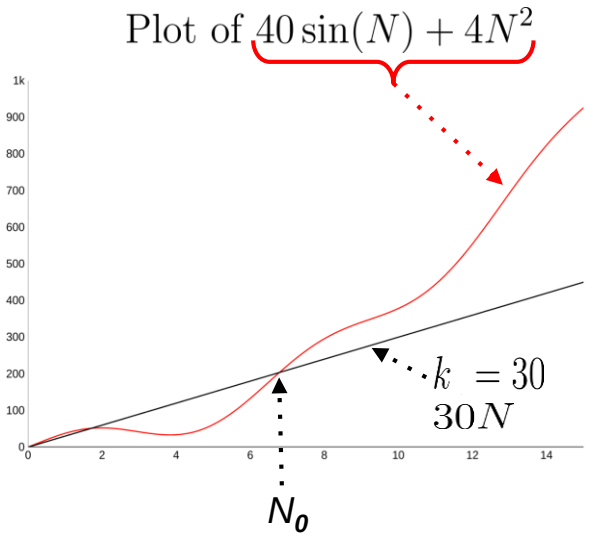
- When we're dealing with infinity, constants and lower-order terms don't meaningfully add to the final result

- (1) Eliminate lower-order terms
  - $6 + \frac{1}{2}N^2 + \frac{3}{2}N + 1 + \frac{1}{2}N^2 + \frac{1}{2}N + \frac{1}{2}N^2 - \frac{1}{2}N + N^2 + N$
  - $6 + \frac{1}{2}N^2 + \frac{3}{2}N + 1 + \frac{1}{2}N^2 + \frac{1}{2}N + \frac{1}{2}N^2 - \frac{1}{2}N + N^2 + N$
  - $\frac{5}{2}N^2$

- (2) Ignore multiplicative constants
  - $\frac{5}{2}N^2$
  - N^2

Why Does This Work?

Demo: https://www.desmos.com/calculator/rl25eewwe3



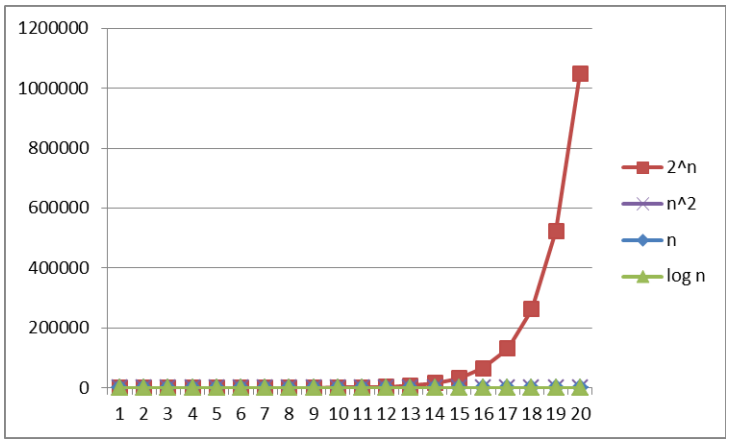
Lecture Outline

- Comparing two algorithms
- Analyzing Code
- When the Input is Large: Asymptotic Analysis
- **Review: Logarithms and Exponents**
- Big-Oh Definitions

Logarithms and Exponents

- Definition: $\log_2 x = y$ if $x = 2^y$
  - Note: since so much is binary in CS, $\log$ almost always means $\log_2$

- Just as exponents grow *very* quickly, logarithms grow *very* slowly
  - So, $\log_2$ 1,000,000 = "a little under 20"



Log base doesn't matter (much)

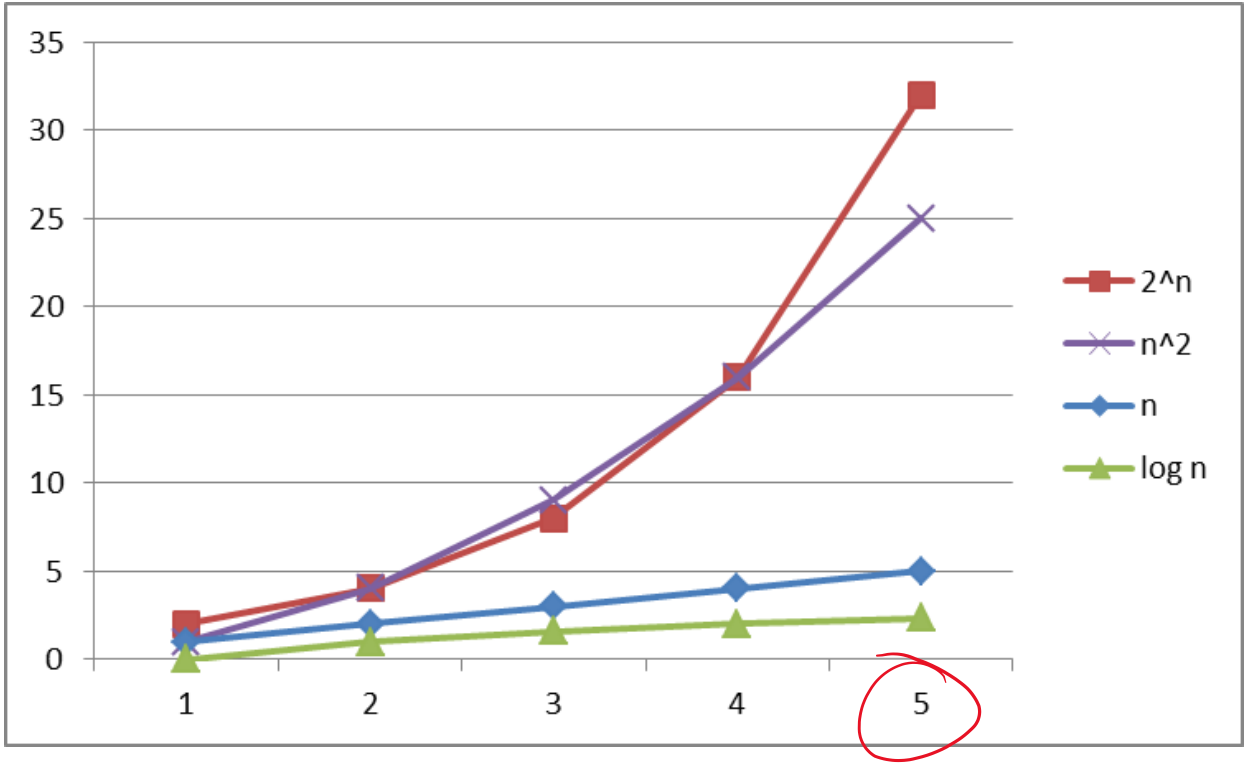
- "Any base *B* log is equivalent to base 2 log within a constant factor"
  - ***And we are about to prove constant factors don't matter!***
  - In particular, $\log_2 x = 3.22 \log_{10} x$

- Why a constant multiplier ?
  - $\log_B x = (\log_A x) / (\log_A B)$

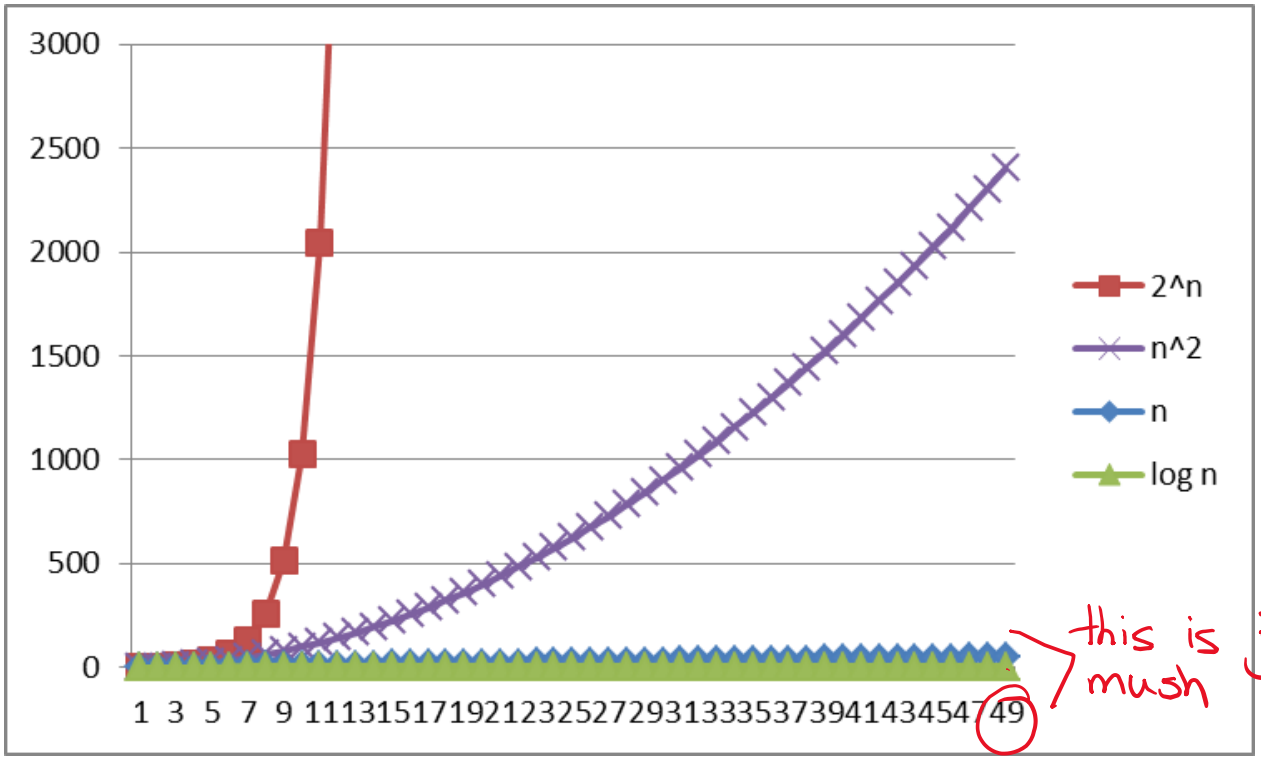
Review: Properties of logarithms

- **log(A*B) = log A + log B**
  - So **log(N^k) = k log N**
- **log(A/B) = log A – log B**
- **X =** $\log_2 2^x$
- **log(log x)** is written **log^y log x**
  - Grows as slowly as 2^2 grows fast
  - Ex: $\log_2 \log_2 4billion \sim \log_2 \log_2 2^{32} = \log_2 32 = 5$
- **(log x)(log x)** is written **$\log^2$x**
  - It is greater than **log x** for all **x > 2**

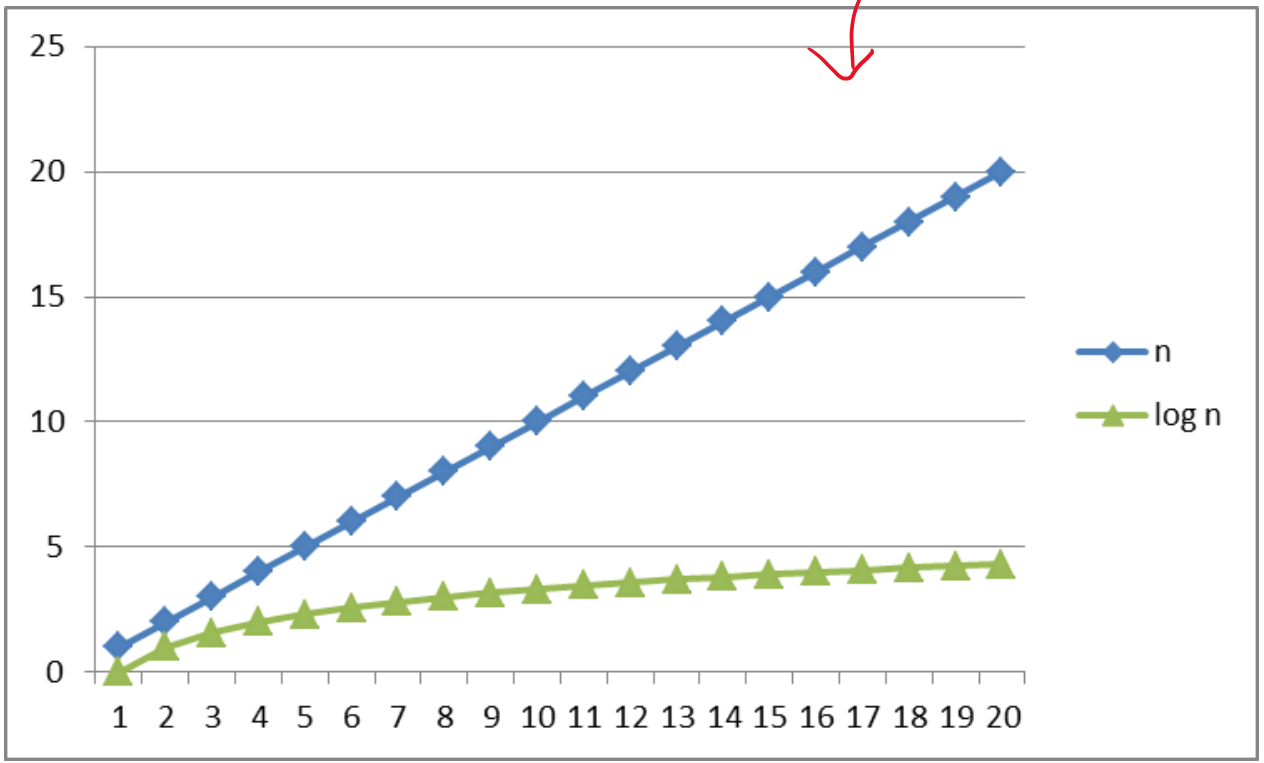
Logarithms and Exponents



Logarithms and Exponents



Logarithms and Exponents



Lecture Outline

- Comparing two algorithms
- Analyzing Code
- When the Input is Large: Asymptotic Analysis
- **Big-Oh Definition**

Introduction: Asymptotic Notation

- About to show formal definition, which amounts to our earlier intuitive simplifications:
  - Eliminate lower-order terms
  - Ignore multiplicative constants

- Examples:
  - $4n + 5$
  - $0.5n \log n + 2n + 7$
  - $n^3 + 2^n + 3n$
  - $n \log (10n^2)$

Big-Oh relates functions

- We use O on a function f(n) (for example n^2) to mean *the set of functions with asymptotic behavior* ***less than or equal to*** f(n)

- So $(3n^2+17)$ **is in** $O(n^2)$
  - $3n^2+17$ and n^2 have the same **asymptotic behavior**

- Confusingly, we also say/write:
  - $(3n^2+17)$ **is** $O(n^2)$
  - $(3n^2+17)$ **∈** $O(n^2)$
  - $(3n^2+17)$ **=** $O(n^2)$ ← *least ideal*

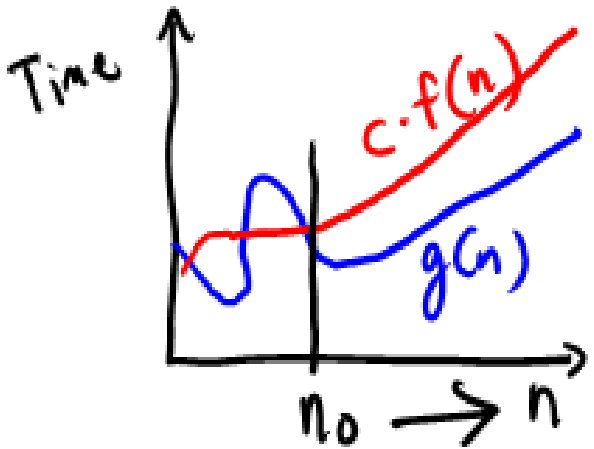
- But we would never say $O(n^2) = (3n^2+17)$

Big-Oh, Formally (1 of 3)

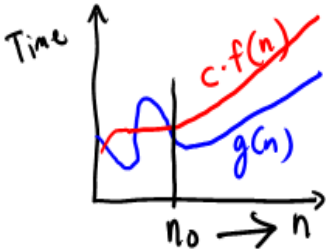
Definition: **g(*n*)** is in O(**f(*n*)**) iff there exist positive constants ***c*** and n_0 such that

$g(n) \leq c\, f(n)$ for all $n \geq n_0$

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$



Big-Oh, Formally (2 of 3)



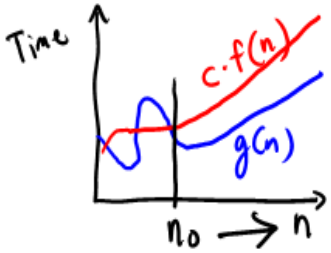
Definition: $g(n)$ is in $O(f(n))$ iff there exist positive constants c and n_0 such that

$g(n) \leq c\, f(n)$ for all $n \geq n_0$

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$

To show $g(n)$ is in $O(f(n))$, pick a c large enough to "cover the constant factors" and n_0 large enough to "cover the lower-order terms"

Big-Oh, Formally (3 of 3)



Definition: **g(*n*)** is in O(**f(*n*)**) iff there exist positive constants ***c*** and n_0 such that

$g(n) \leq c\, f(n)$ for all $n \geq n_0$

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$

Example: Let **g(*n*)** = 3*n* + 4 and **f(*n*)** = *n*
$3n+4 \leq 4n \quad \forall n \geq 5$
$\therefore 3n+4 \in O(n)$
c = 4 and n_0 = 5 is one possibility

Example: Let **g(*n*)** = 3*n* + 4 and **f(*n*)** = n^5
$3n+4 \leq 3n^5 \quad \forall n \geq 2$
$\therefore 3n+4 \in O(n^5)$
c = 3 and n_0 = 2 is one possibility

Example: Let **g(*n*)** = 3*n* + 4 and **f(*n*)** = 2^n
$3n+4 \leq 100000000 \cdot 2^n \quad \forall n \geq 1$
$\therefore 3n+4 \in O(2^n)$
c = 100000000 and n_0 = 1 is one possibility

Summary

- Complexity analyses use simplified cost models
- Asymptotic analysis can take liberties with mathematical expressions because it deals with infinity
  - Eg, dropping lower-order terms and constants
  - But it gives us a common "frame of reference" with which to compare algorithms, too!
- Case Analysis != Asymptotic Analysis
  - Case analysis is a different axis on which to evaluate runtime and space
- Review your log rules!