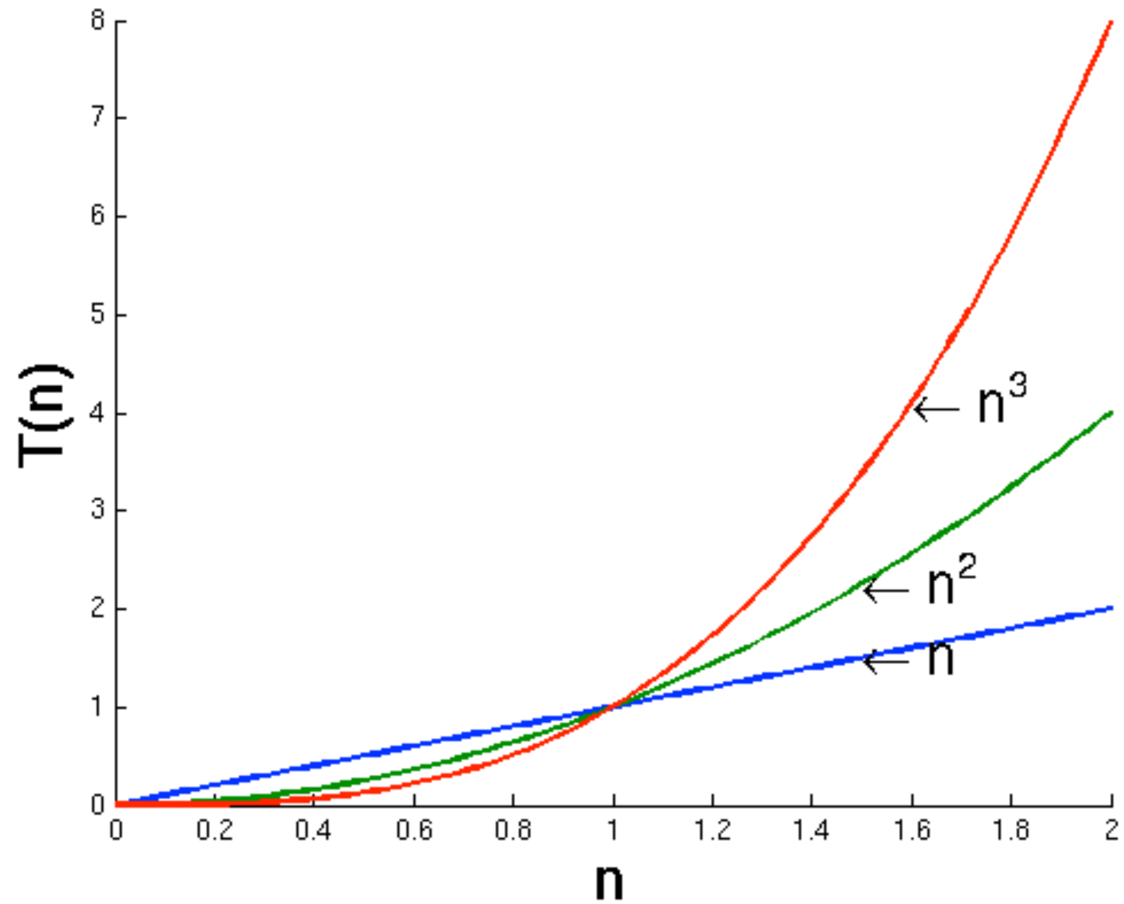
Asymptotic Running Time Analysis

Professor Kevin Gold

Asymptotic Analysis



- We think about running time based on how the number of operations scales with input size
- In the long run, some functions dominate others

Goals

- Our goals are to --
  - Make you an informed *consumer* of algorithms, interpreting how fast they are
  - Let you analyze the speed of *your own code*, so you can tell when a significant speedup may be possible
- Today we'll cover **big-O**, **big-Ω**, and **big-Θ** and the conventions by which computer scientists describe running times.

Asymptotic Growth

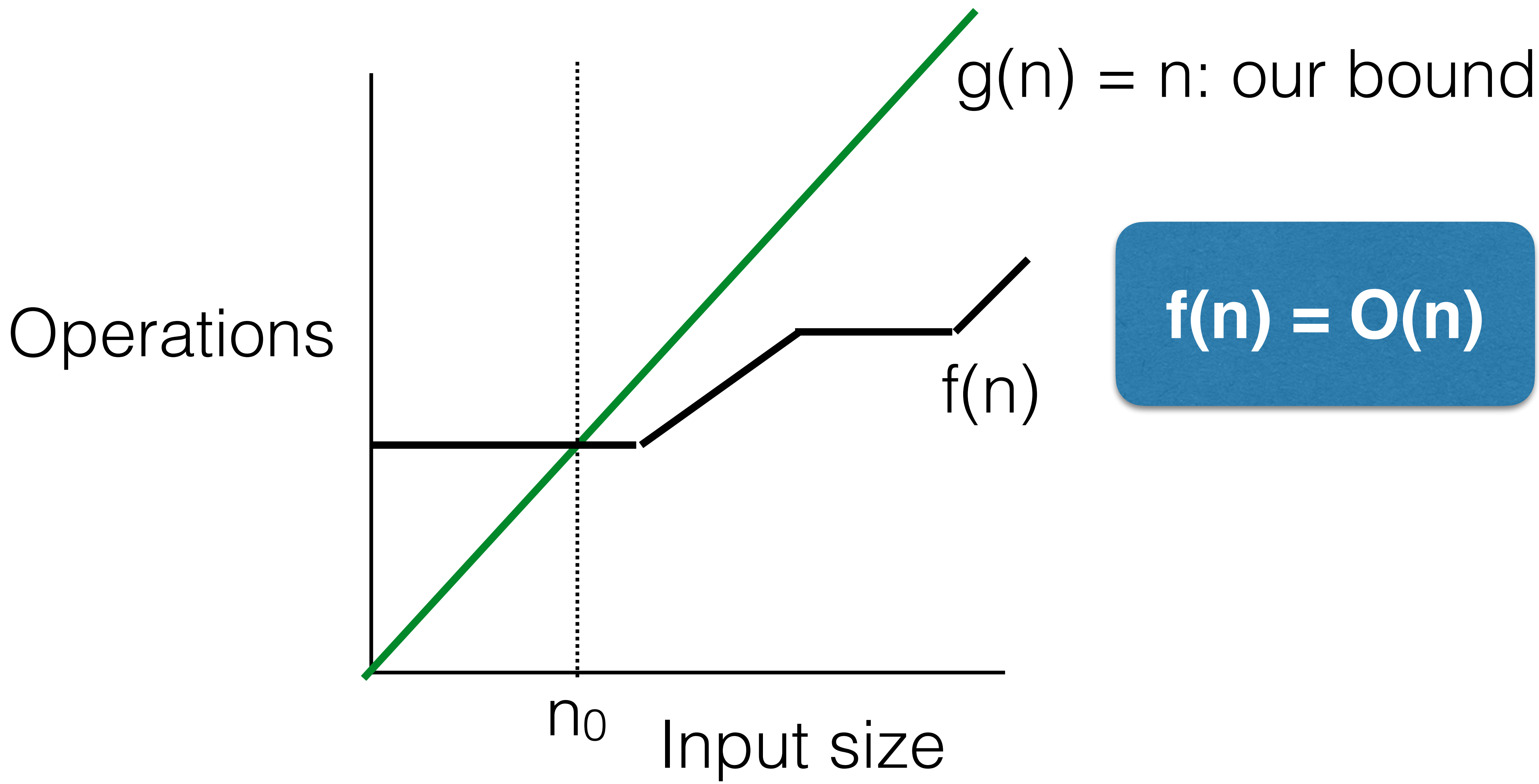
- We will generally only be concerned with asymptotic running times — **what happens when input size N gets large.**
- We want to ignore multiplication by constant factors.
    - If we *did* care about these, we'd need to know the exact time required for instructions — too machine-specific
    - The differences we'll care about are *bigger* than constant factors in the long run
- We'll show three ways of talking about function growth — an upper bound (O), a lower bound (Ω), and a tight bound (Θ)

Big-O: Intuitive Gloss

- Big-O separates functions into different worst-case running times - ignoring constants and in the long run
    - Constant: O(1)
    - Linear or better: O(N)
    - Quadratic or better: $O(N^2)$
    - Factorial or better: O(N!)
- Big-O is technically an **upper bound**, but is often used informally as if it were the real running time; why would you say it's $O(N^2)$ if it's really linear?

Big-O: A Definition

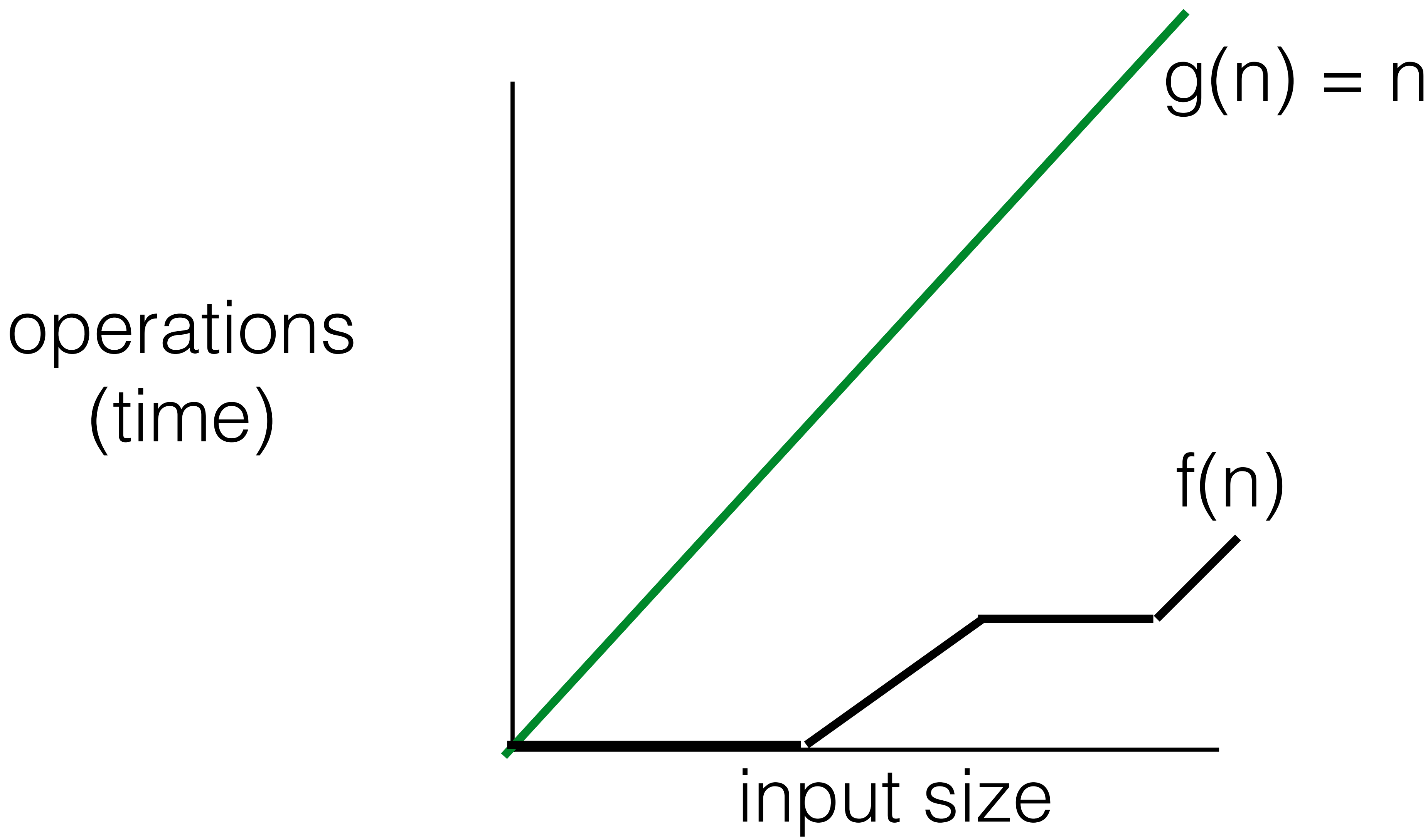
- Let f be a function of the natural numbers.
- f = O(g(n)) if, for some positive c and n_0, **$f(n) \leq cg(n)$ for all $n \geq n_0$**.



f(n) = O(n)

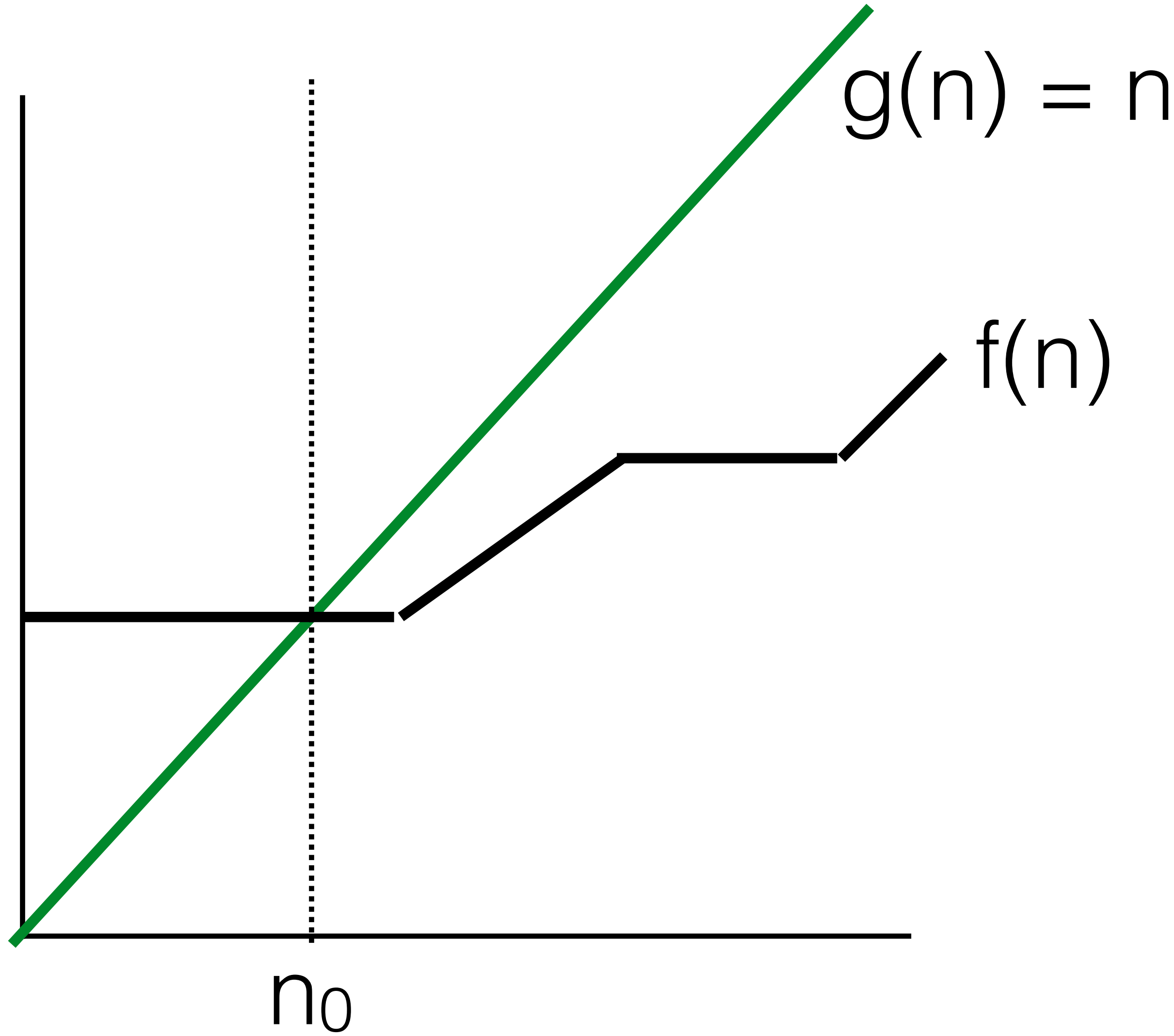
Breaking down the definition

- The core idea is $f(n) = O(g(n))$ if $f(n) \le g(n)$ as n gets large.



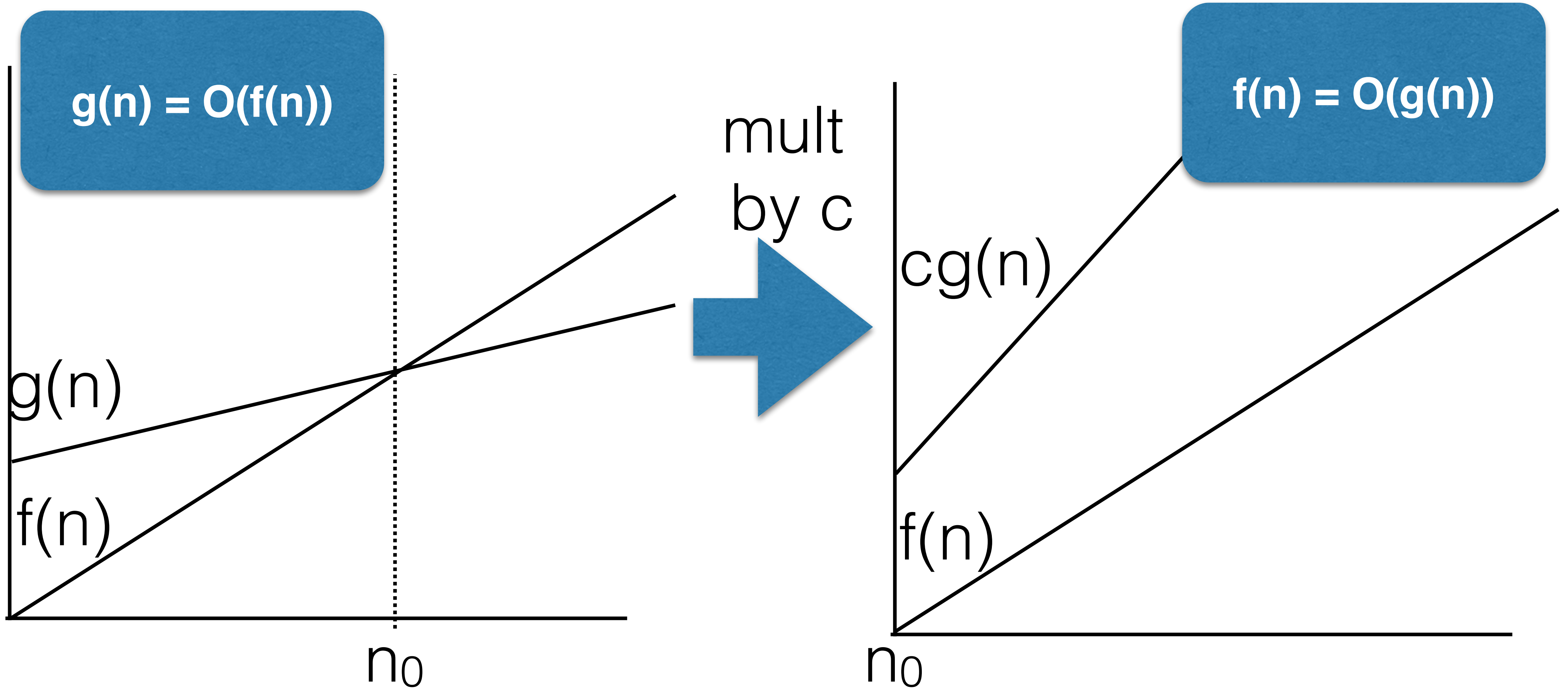
Breaking down the definition

- We allow f(n) to be greater than g(n) for a little while, as long as g(n) passes it in the long run (past some n_0).

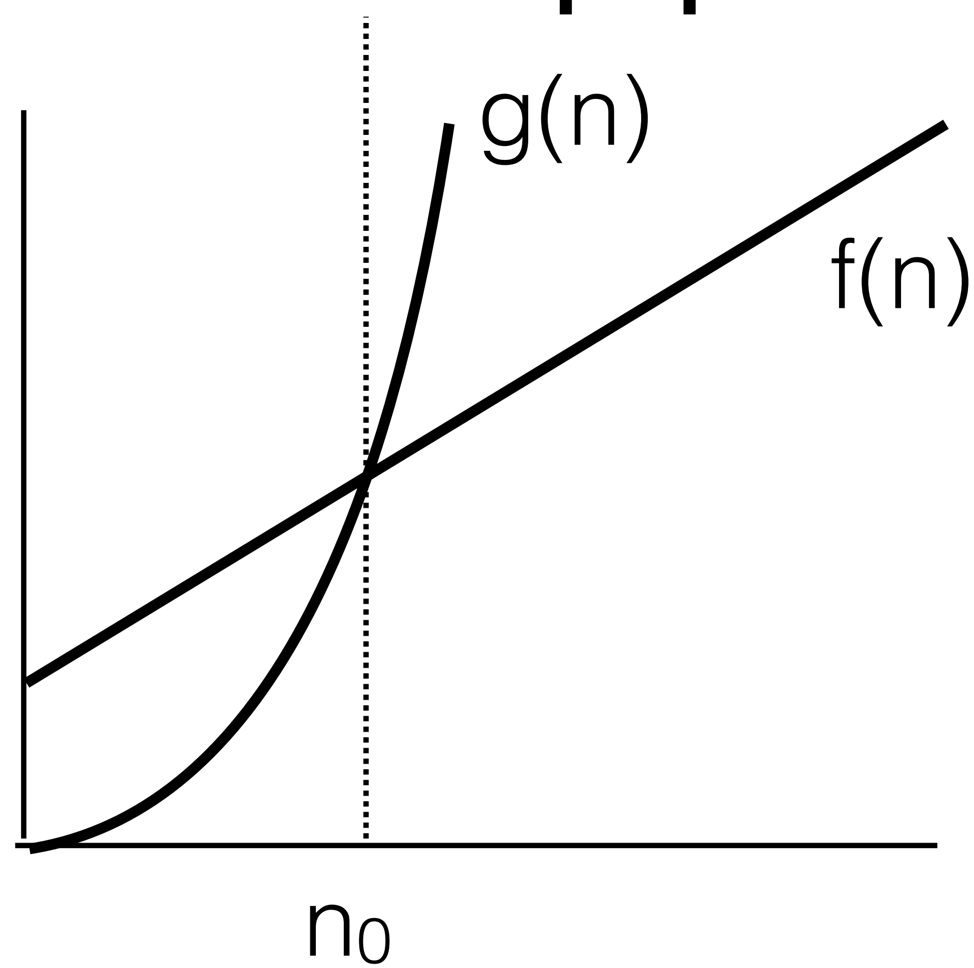


Breaking down the definition

- We also want constants to not matter. So if we can multiply by a constant c and get $f(n) \le cg(n)$, that should count as $f(n) = O(g(n))$.
- This will make functions with the same growth rate big-O of each other.



Big-O is an Upper Bound



- Here, **f(n) = O(g(n))** — they are not the same growth rate, but g(n) is always bigger after n_0 (c=1, n_0 = 0)

Why Use an Upper Bound? Uncertainty in the Analysis

- For i = 1 to N:
    If A[i] is odd:
        return A[i]
  return 0

- What is the running time of this? It looks linear in the worst case (no odd numbers). So, O(N)

- But what if that worst case is actually impossible because A[1] is *always odd*? Then the running time is actually O(1).

- We may not *know* whether this acts more like linear or constant time, but O(N) is a safe claim in either case

Big-O is the "≤" of Function Relations

- There are asymptotic operators we'll cover that are analogous to each of ≤, ≥, =, <, and >
- Of these, **Big-O is most similar to ≤**.
  - It holds when two growth rates are essentially equal: $N = O(2N)$. (Both linear)
  - It holds when the first growth rate is asymptotically less than the second: $N = O(2^N)$. (Linear vs exponential)
- It is the most commonly used of the bounds because with algorithms, we usually want an **upper bound on the worst case running time**.
  - "The worst case running time of my algorithm is $O(n^2)$"

A Big-O Claim is More Like Set Membership than Equality

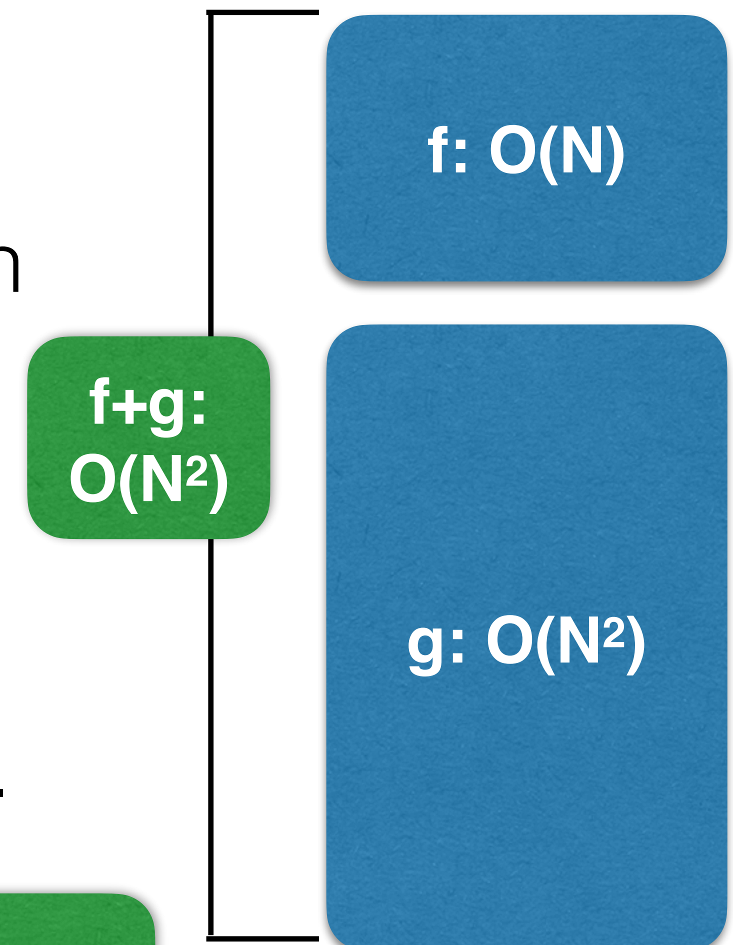
- Always put big-O on the right of the equals sign: $5n = O(n)$.

- The = sign isn't really symmetric, so you can't do algebra like "$5n = O(n) = 2n$ implies $5n = 2n$." You are claiming that the function on the left belongs to the category on the right.

    - For this reason, big-O is sometimes written with set inclusion: $\mathbf{f(n)} \in \mathbf{O(n)}$.

A Big-O Bound that Works for Both Functions is a Bound on their Sum

- **If f and g are both O(h), then f+g is O(h)**. (Where h is a function.)
- For example, $f(n) = n^2$ and $g(n) = n$ are both $O(n^2)$, so $f(n) + g(n) = n^2 + n$ is $O(n^2)$
- Proof: if $f \le c_1 h$ and $g \le c_2 h$, $f+g \le (c_1 + c_2)h$ and $(c_1 + c_2)$ is the constant required by the definition of big-O.



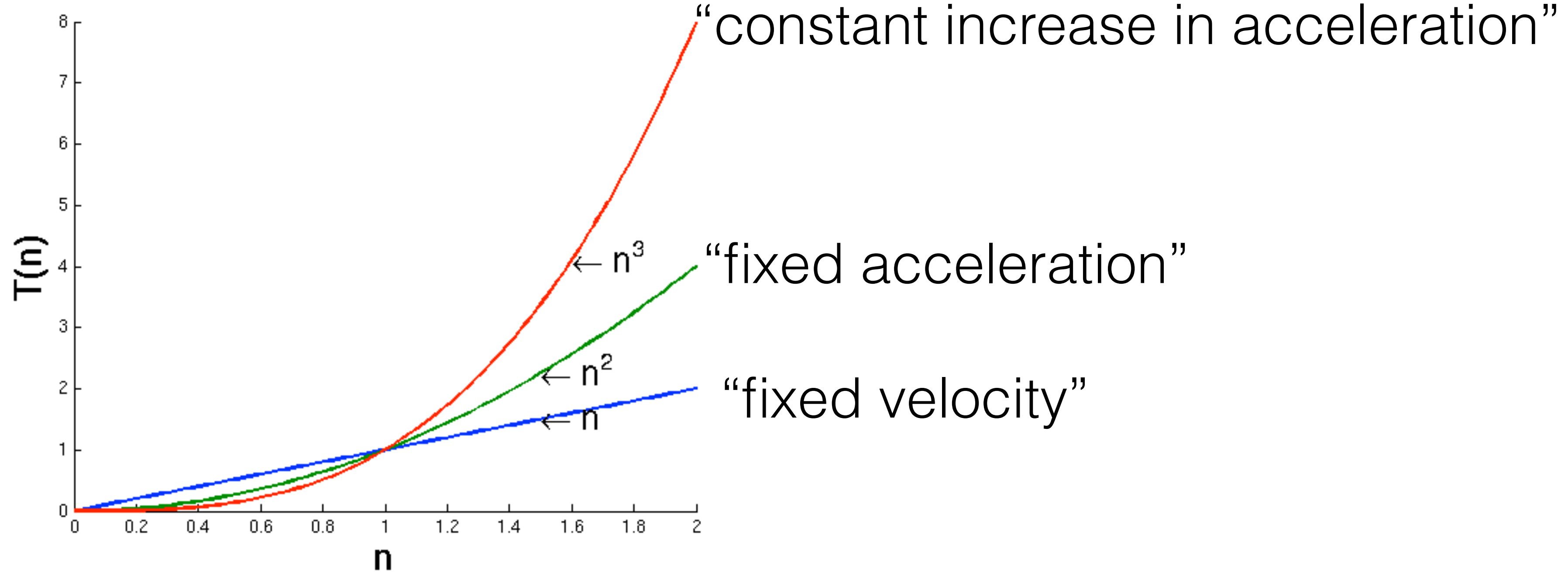
When calculating the asymptotic time of doing one subroutine, then another, the big-O is simply the same as the more expensive subroutine.

A Polynomial of Degree d is $O(n^d)$

- Given a polynomial $a_0n^d + a_1n^{d-1} + \ldots + a_d$, note that each term is $O(n^d)$
- So by the result on the previous slide, summing the terms results in a function that is still $O(n^d)$.

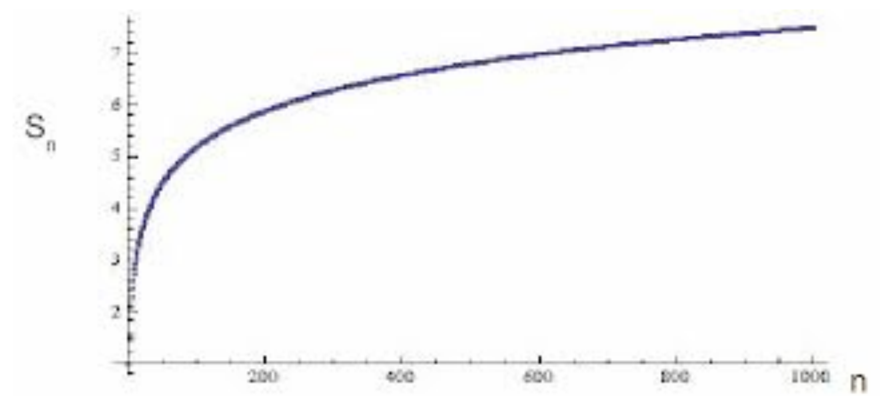
So always drop lower-order terms when describing a polynomial running time: $O(N^2)$, not $O(N^2 + N + 1)$

Increasing the Degree Increases the Big-O



- Every time we increase the degree, we get a factor *n* that can't be compensated for by multiplying by a constant
  - $n^3 \leq cn^2$ doesn't work, because we'd have $n \leq c$ and c is constant
- This reinforces the idea that *no matter what the constant factors are*, the bigger growth rate will eventually pass up the slower growth rate

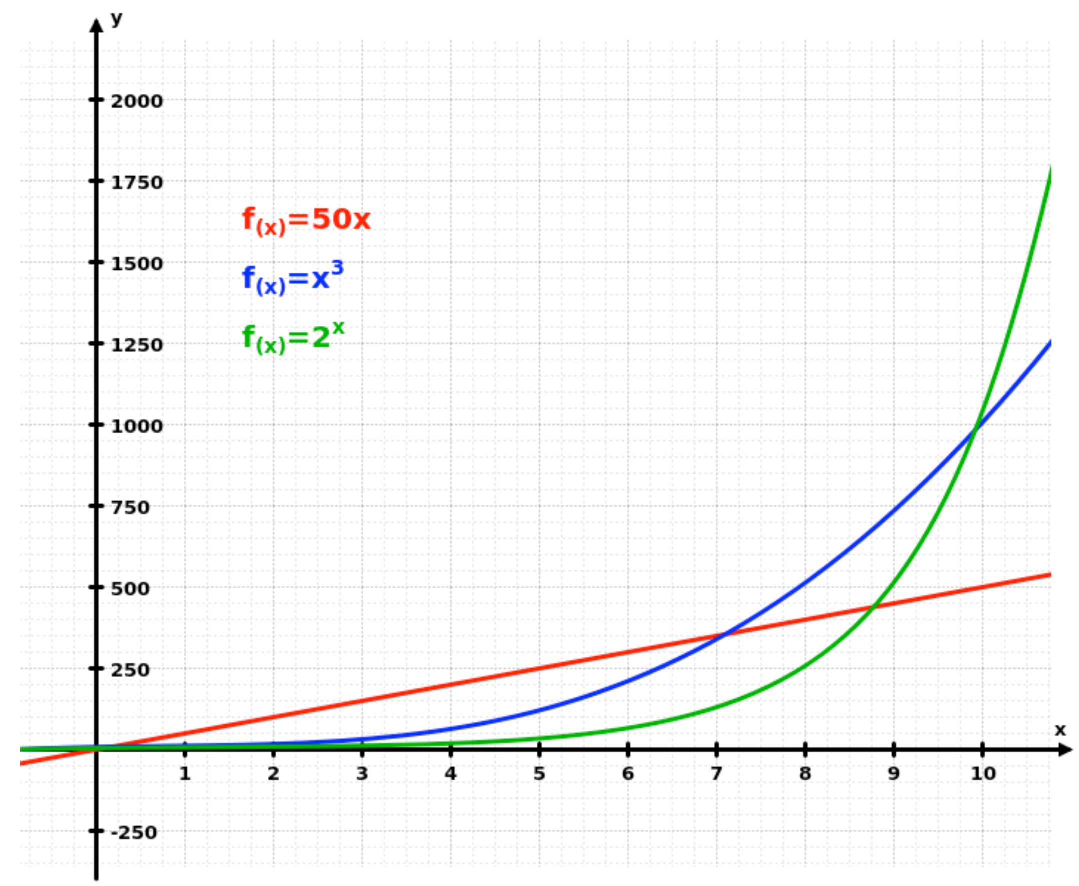
Every Logarithm Grows Slower Than Every Polynomial



- Logarithmic growth is always **slower any polynomial**
  - We can show this with limits: $\lim_{n\to\infty} \log n / n^x = \lim_{n\to\infty} (1/n)/xn^{x-1} = \lim_{n\to\infty} 1/xn^x = 0$
- **Logs of different bases are within constant factors of each other**: $\log_b n = (\log_c n) / (\log_c b)$ = a constant times $\log_c n$, for all bases b & c (for example, $\log_2 N = (\log_3 N)/(\log_3 2)$, and $1/(\log_3 2)$ is a constant)
- So we will often talk about **O(log n)** without specifying a base
  - Though occasionally we will speak of lg n (base 2) and ln n (base e)

Every Exponential Grows Faster Than Every Polynomial

- For every $r > 1$ and every $d > 0$, n^d grows slower than r^n
  - For example, n^{100} grows slower than 1.02^n (though n^{100} gets a nice head start)
- Unlike logs, exponentials with different bases have different big-O: 3^n is not $O(2^n)$, since $(3/2)^n$ isn't a constant



Exponential Time is Much Worse than Polynomial Time

- Time to process N inputs at 1 million instructions per second with the given running times (p. 34 of Kleinberg & Tardos)

N	n	n^3	2^n
10	< 1s	< 1s	< 1s
100	< 1s	< 1s	10^{17} years
1000	< 1s	18min	$>10^{25}$ years
10000	< 1s	12 days	$>10^{25}$ years

An Algorithm is Considered Tractable if it is Polynomial Time

- Where polynomial time means, $O(n^d)$ for some d
- Note that this includes logarithmic and constant time

It's Typically the Worst Case That is Analyzed

- Best cases don't say as much about how an algorithm performs typically - every algorithm can get lucky
- It's also usually easier to analyze the worst case than the average case over all inputs
    - No need to make assumptions about the distribution of input for worst case - just "anything that can go wrong, will"
- Only for *randomized* algorithms - with randomness built in to the algorithm - do we see heavy use of the average running time

Review of Big-O So Far: Practice Questions

- Identify whether $f(N) = O(g(N))$, $g(N) = O(f(N))$, or both.

1. $f(N) = 2N$, $g(N) = 4N$
2. $f(N) = N^2$, $g(N) = 2^N$
3. $f(N) = N^2$, $g(N) = N^2 + N + 100$
4. $f(N) = N$, $g(N) = \log N$
5. $f(N) = 2^N$, $g(N) = N2^N$
6. $f(N) = N \log N$, $g(N) = N$
7. $f(N) = \sqrt{N}$, $g(N) = \log N$

Review of Big-O So Far: Practice Questions

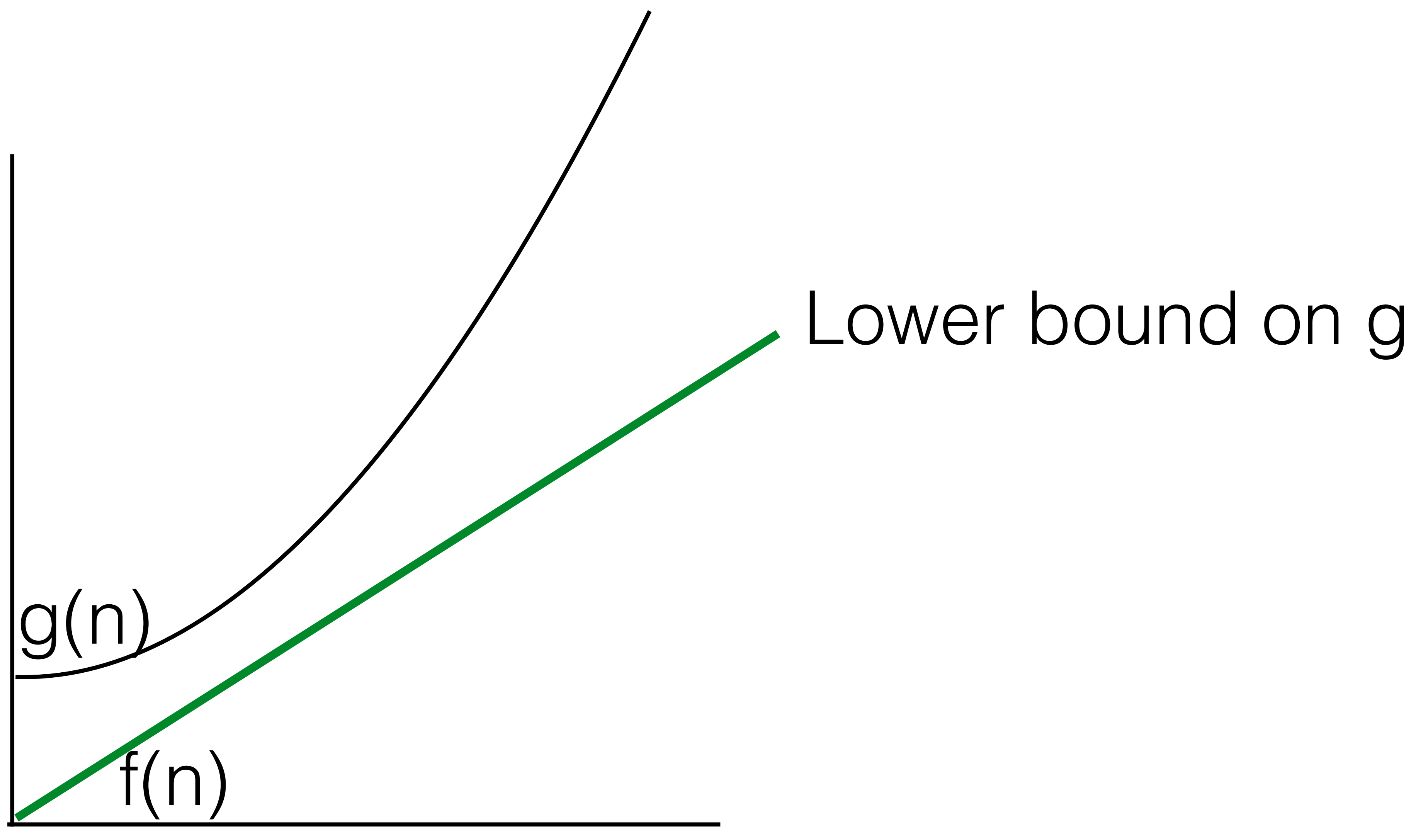
- Identify whether $f(N) = O(g(N))$, $g(N) = O(f(N))$, or both.

1. $f(N) = 2N$, $g(N) = 4N$ **both**
2. $f(N) = N^2$, $g(N) = 2^N$ **$N^2 = O(2^N)$**
3. $f(N) = N^2$, $g(N) = N^2 + N + 100$ **both**
4. $f(N) = N$, $g(N) = \log N$ **$\log N = O(N)$**
5. $f(N) = 2^N$, $g(N) = N2^N$ **$2^N = O(N2^N)$**
6. $f(N) = N \log N$, $g(N) = N$ **$N = O(N \log N)$**
7. $f(N) = \sqrt{N}$, $g(N) = \log N$ **$\log N = O(N^{0.5})$**

Lower Bounds: Ω

- Sometimes we may want to say, "This algorithm must take **at least** this much time"
    - Often to show a running time can't be improved further.
- Just as big-O serves as an upper bound on the running time, big-Ω serves as a lower bound, the ≥ to big-O's ≤
- The definition is identical to big-O with one inequality reversed
- $f(n) = \Omega(g(n))$ if, for some c and n_0, $f(n) \geq cg(n)$ for all $n \geq n_0$.

g is Ω(f) iff f is O(g)



Tight Bounds: Θ

- If $f = O(g)$ and $f = \Omega(g)$, then $f = \Theta(g)$.
- Unlike big-O, Θ means you are giving the actual growth rate (but still ignoring constants!) instead of an upper bound.
  - $n^2 = O(2^n)$ but $n^2 \neq \Theta(2^n)$
  - $n^2 + 1 = \Theta(n^2)$
- This is the asymptotic bound analogous to “=": growth rates must be effectively the same for one to be big-Θ of the other
- People often mean big-Θ when they say big-O! And they’re still telling the truth because $f = \Theta(g)$ implies $f = O(g)$!

Other Bounds: Little-o, Little-ω

- If we want to say one growth rate is strictly faster or slower than another, we use little-o and little-ω:
  - $n = o(n^2)$
  - $n = \omega(\log n)$
- These are analogous to < and > for growth rates.
- The technical definition of little-o is $f(n) = o(g(n))$ if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$
- Similarly, $f(n) = \omega(g(n))$ if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = +\infty$.
- We can now say "$\log_2 n$ is $o(n^d)$ for all $d > 0$" and "r^n is $\omega(n^d)$ for all $r > 1$, $d > 0$"

Other Bounds Practice

- Of o, ω, O, Θ, and Ω, which **best** describes the relationship between the functions, and which **also happen to apply**?
    - $f(n) = n^2$, $g(n) = n^2 + n$
    - $f(n) = 2^n$, $g(n) = 3^n$
    - $f(n) = n^2$, $g(n) = n \log n$

Other Bounds Practice

- Of o, ω, O, Θ, and Ω, which **best** describes the relationship between the functions, and which **also happen to apply**?
  - $f(n) = n^2$, $g(n) = n^2 + n$ **f(n) is O(g(n)), Ω(g(n)), Θ(g(n))**
  - $f(n) = 2^n$, $g(n) = 3^n$ **f(n) is o(g(n)), O(g(n))**
  - $f(n) = n^2$, $g(n) = n \log n$ **f(n) is Ω(g(n)), ω(g(n))**

In general, one of these three patterns must hold, corresponding to o, Θ, and ω

Analyzing Algorithms

- Analyzing an algorithm's runtime comes down to **counting operations** and characterizing how the number of operations scales with the input.
- We assume **elementary operations such as integer comparisons, additions, and assignment are constant time** (unless we expect to deal with numbers of unbounded size)
- Iterating over all the input once is linear time.
- If an algorithm does one subroutine and then another, the operations are summed, so use the larger big-O of the two.
- If operations happen in a loop, **multiply the number of operations that happen in one iteration by the number of times the loop executes.**

As usual, ignore constants and low-order terms.

Warm-Up

```
// Sum all entries in an N x N matrix A.
// Assume zero-indexing.
mySum(A):
Let sum = 0.
For i = 0 to N-1,
    For j = 0 to N-1,
        sum += A[i][j]
Return sum
```

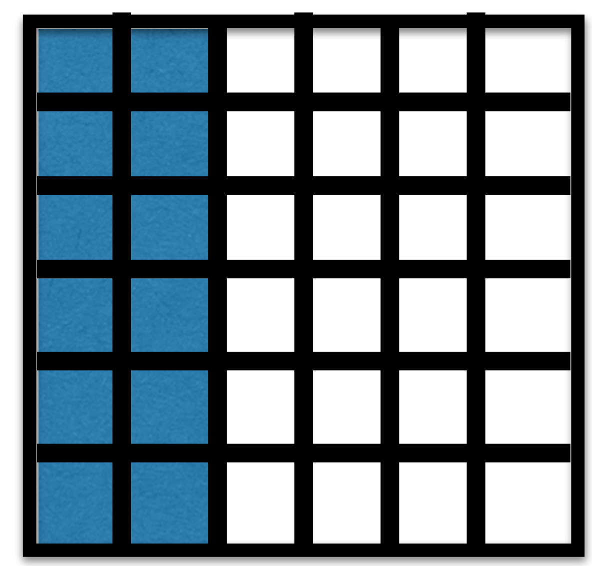
10	10	10
10	10	10
10	10	10

What’s the big-Θ running time?

Playing With Constants

// Sum **almost** all entries in an N x N matrix A.
// Assume zero-indexing.

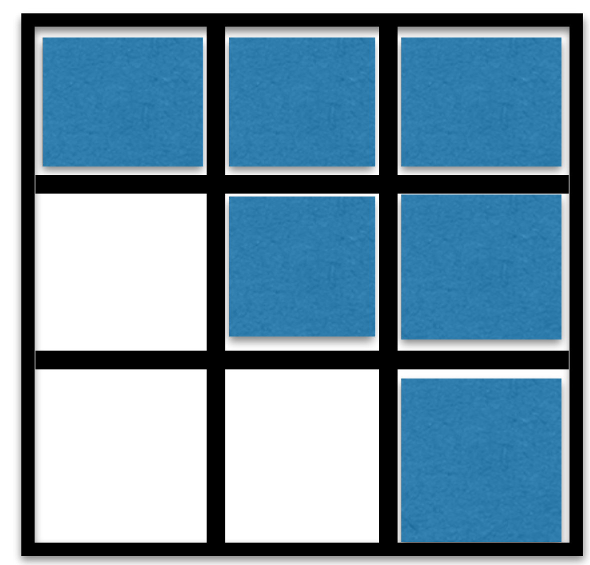
```
mySum(A):
Let sum = 0.
For i = 0 to N-1,
    For j = 0 to N-5,
        sum += A[i][j]
Return sum
```



What's the big-Θ running time?

Half the Matrix

// Sum the **upper right triangle of** an N x N matrix A.
// Assume zero-indexing.
mySum(A):
Let sum = 0.
For i = 0 to N-1,
    For j = **i** to N-1,
        sum += A[i][j]
Return sum



What's the big-Θ running time?

Don't Say N^2

```
// Sum an M x N matrix A.
// Assume zero-indexing.
mySum(A):
Let sum = 0.
For i = 0 to M-1,
    For j = 0 to N-1,
        sum += A[i][j]
Return sum
```

What's the big-Θ running time?

An Exponential Running Time

```
// Brute force approach to "subset-sum" problem.
// Given a set S of numbers, determine whether
// any subset of them could sum to a target number T.
mySubsetSum(S,T):
For all 2^|S| possible subsets mySubset of S,
    mySum = sum of all elements in mySubset
    if mySum == T
        Return mySubset
Return "not found"
```

What's the big-Θ worst case running time?
(Give it in terms of |S|)

An Exponential Running Time

```
// Brute force approach to "subset-sum" problem.
// Given a set S of numbers, determine whether
// any subset of them could sum to a target number T.
mySubsetSum(S,T):
For all 2^|S| possible subsets mySubset of S,
    mySum = sum of all elements in mySubset
    if mySum == T
        Return mySubset
Return "not found"
```

What's the big-Θ worst case running time?
$|S|2^{|S|}$

Think Worst Case

- // Sum a sorted left-to-right N x N matrix A **only in rows**
  // **containing a target element k**.
  // Assume zero-indexing.
  mySum(A,k):
  Let mySum = 0.
  For i = 0 to N-1,
      Perform binary search ($\Theta(\log n)$) to determine whether k is in row i
      If k is present in row i
          Sum row i and add the result to mySum
  Return mySum

What's the big-Θ worst case running time?

$\Theta(N^2)$

- Each iteration is $O(N)$: $O(\log N) + O(N) = O(N)$
  - Binary search for k is $O(\log N)$, summing the row is $O(N)$
- With N $O(N)$ iterations, the algorithm is $O(N^2)$
  - N times a polynomial that is $O(N)$ increases the degree by one
- The worst case is also $\Omega(N^2)$ - it must take at least that much time to sum all the elements in each row

Summary

- We describe algorithm speed by the growth in number of operations required as a function of N, the input size
  - We want that function to grow as slowly as possible!
- big-O: an upper bound on a growth rate, ignoring constants
- big-Ω: a lower bound that otherwise works like big-O
- big-Θ: if big-O and big-Ω apply - a tight bound
- o, ω: "strictly less than," "strictly greater than"
- Each polynomial degree $\Theta(n^k)$ is its own category of growth rate, and others are possible: $\Theta(n \log n)$, $\Theta(n!)$ …
- Your choice of data structures can affect this degree of efficiency