Lecture 2

Divide-and-conquer, MergeSort, and Big-O notation

Today

- Things we want to know about algorithms:
  - Does it work?
  - Is it efficient?

- We'll start to see how to answer these by looking at some examples of sorting algorithms.
  - InsertionSort
  - MergeSort

SortingHatSort not discussed

The plan

- Part I: Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?
  - **Skills:**
    - Analyzing correctness of iterative and recursive algorithms.
    - Analyzing running time of recursive algorithms (part 1…more next time!)
- Part II: How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis

Sorting

- Important primitive
- **For today**, we'll pretend all elements are distinct.

6	4	3	8	1	5	2	7

1	2	3	4	5	6	7	8

Benchmark: insertion sort

- Say we want to sort: $A = (6,5,3,1,8,7,2,4)$

- "Algorithm": Insert items one at a time.

Student sorting experiment (pumpkins!)

Insertion Sort Algorithm:

```
InsertionSort(A):
  for i in [1:n]
       current ← A[i]
       j ← i-1
       while j >= 0 and A[j] > current:
            A[j+1] ← A[j]
            j ← j-1
       A[j+1] ← current
```

Insertion Sort Algorithm:

```
InsertionSort(A):
  for i in [1:n]
      current ← A[i]
      j ← i-1
      while j >= 0 and A[j] > current:
          A[j+1] ← A[j]
          j ← j-1
      A[j+1] ← current
```

6 5 3 1 8 7 2 4

Insertion Sort

1. Does it work?
2. Is it fast?

Insertion Sort: running time

```
InsertionSort(A):
  for i in [1:n]
    current ← A[i]
    j ← i-1
    while j >= 0 and A[j] > current:
      A[j+1] ← A[j]
      j ← j-1
    A[j+1] ← current
```

n iterations of the outer loop

In the worst case, about n iterations of this inner loop

Running time scales like n^2

Insertion Sort

1. Does it work?
2. Is it fast?

- Okay, so it's pretty obvious that it works.

- HOWEVER! In the future it won't be so obvious, so let's take some time now to see how we would prove this rigorously.

Why does this work?

- Say you have a sorted list, | 3 | 4 | 6 | 8 |, and another element $\boxed{5}$.

- Insert $\boxed{5}$ right after the largest thing that's still smaller than $\boxed{5}$. (Aka, right after $\boxed{4}$).

- Then you get a sorted list: | 3 | 4 | 5 | 6 | 8 |

So just use this logic at every step.

6	4	3	8	5

The first element, [6], makes up a sorted list.

4	6	3	8	5

So correctly inserting 4 into the list [6] means that [4,6] becomes a sorted list.

4	6	3	8	5

The first two elements, [4,6], make up a sorted list.

3	4	6	8	5

So correctly inserting 3 into the list [4,6] means that [3,4,6] becomes a sorted list.

3	4	6	8	5

The first three elements, [3,4,6], make up a sorted list.

3	4	6	8	5

So correctly inserting 8 into the list [3,4,6] means that [3,4,6,8] becomes a sorted list.

3	4	6	8	5

The first four elements, [3,4,6,8], make up a sorted list.

3	4	5	6	8

So correctly inserting 5 into the list [3,4,6,8] means that [3,4,5,6,8] becomes a sorted list.

YAY WE ARE DONE!

This slide skipped in class; for reference only.

Recall: proof by induction

A loop invariant is something that should be true at every iteration.

- Maintain a loop invariant.
- Proceed by induction.

- **Four steps in the proof by induction:**
  - Inductive Hypothesis: The loop invariant holds after the i^{th} iteration.
  - Base case: the loop invariant holds before the 1^{st} iteration.
  - Inductive step: If the loop invariant holds after the i^{th} iteration, then it holds after the $(i+1)^{st}$ iteration
  - Conclusion: If the loop invariant holds after the last iteration, then we win.

Formally: induction

A "loop invariant" is something that we maintain at every iteration of the algorithm.

- Loop invariant(i): `A[:i+1]` is sorted.

- Inductive Hypothesis:
  - The loop invariant(i) holds at the end of the i^{th} iteration (of the outer loop).
- Base case (i=0):
  - Before the algorithm starts, `A[:1]` is sorted. ✓
- Inductive step:
- Conclusion:
  - At the end of the n-1'st iteration (aka, at the end of the algorithm), `A[:n] == A` is sorted.
  - That's what we wanted! ✓

This logic (see Lecture Notes for details)

4	6	3	8	5

The first two elements, [4,6], make up a sorted list.

This was iteration i=2.

3	4	6	8	5

So correctly inserting 3 into the list [4,6] means that [3,4,6] becomes a sorted list.

Aside: proofs by induction

- We're gonna see/do/skip over a lot of them.
- I'm assuming you're comfortable with them from CS..

- If that went by too fast and was confusing:
  - Slides [there's a hidden one with more info]
  - Lecture notes
  - Book
  - Office Hours

Make sure you really understand the argument on the previous slide!

Siggi the Studious Stork

To summarize

InsertionSort is an algorithm that correctly sorts an arbitrary n-element array in time that scales like n^2.

Can we do better?

The plan

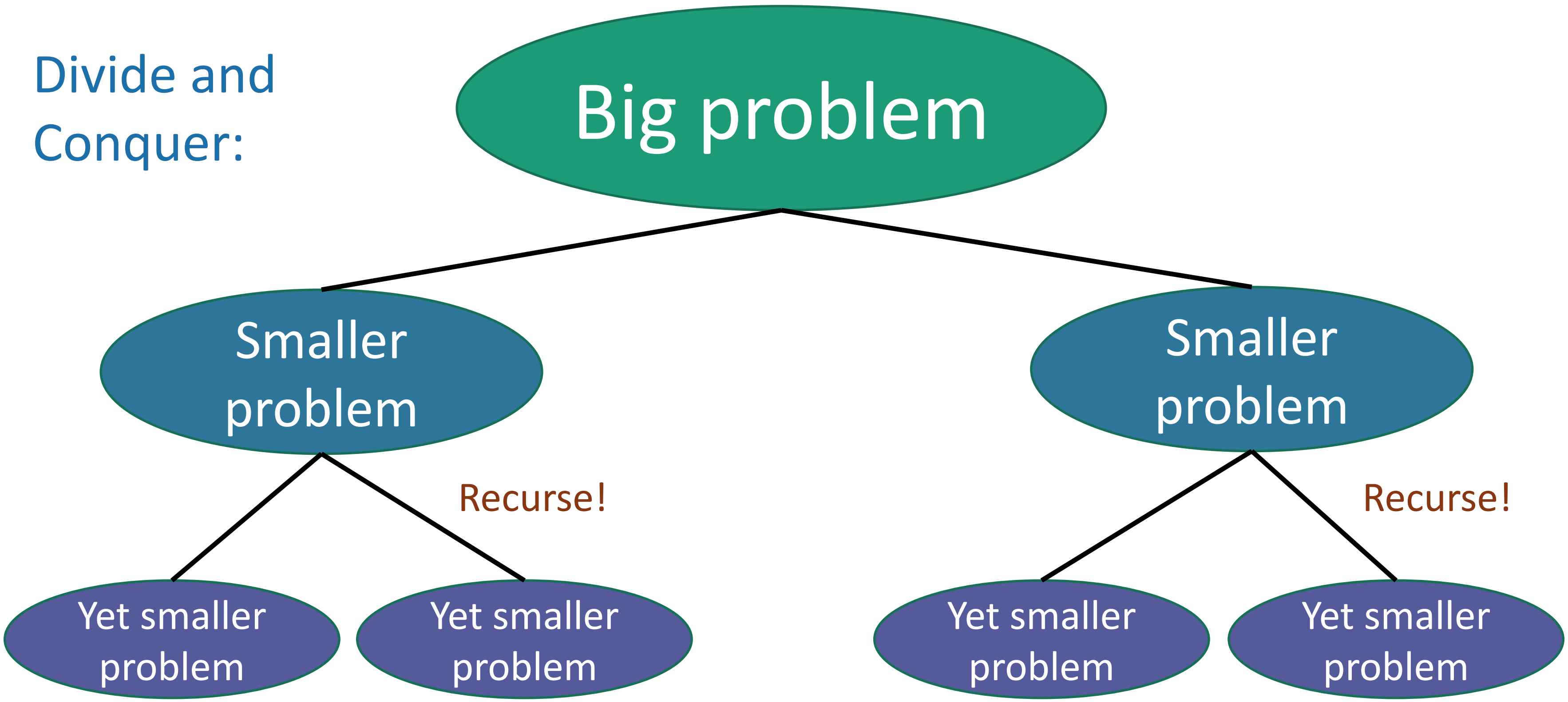
- Part I: Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?

  - **Skills:**
    - Analyzing correctness of iterative and recursive algorithms.
    - Analyzing running time of recursive algorithms (part A)

- Part II: How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis

Can we do better?

- MergeSort: a divide-and-conquer approach
- Recall from last time:



MergeSort

6	4	3	8	1	5	2	7

6	4	3	8

Recursive magic!

3	4	6	8

1	5	2	7

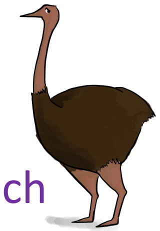
Recursive magic!

1	2	5	7

MERGE!

1	2	3	4	5	6	7	8

How would you do this in-place?



Ollie the over-achieving Ostrich

MergeSort Pseudocode

MERGESORT(A):

- n ← length(A)
- **if** n ≤ 1:
  - **return** A

If A has length 1,
It is already sorted!

- L ← MERGESORT(A[0 : n/2])

Sort the left half

- R ← MERGESORT(A[n/2 : n])

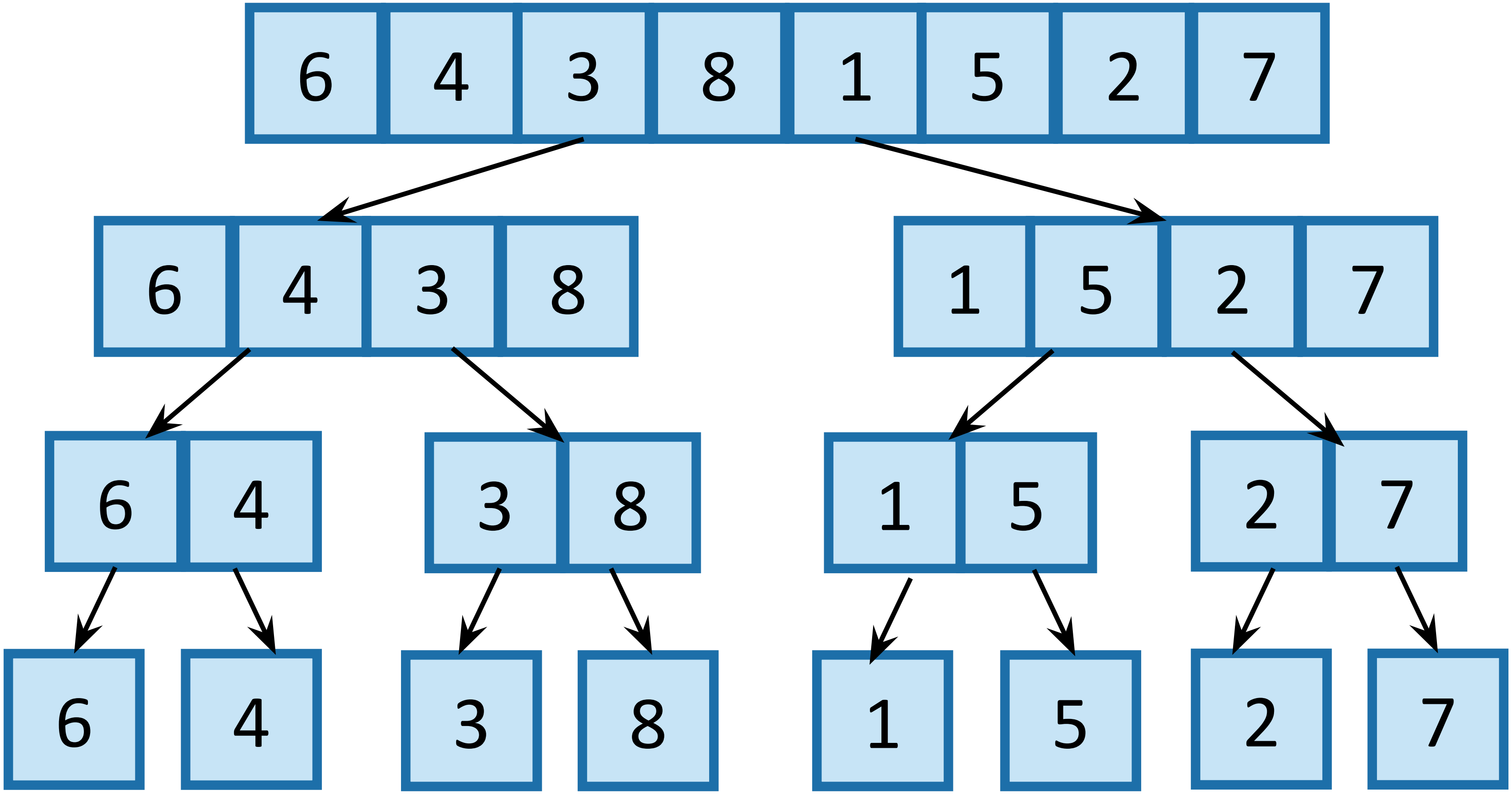
Sort the right half

- **return** MERGE(L,R)

Merge the two halves

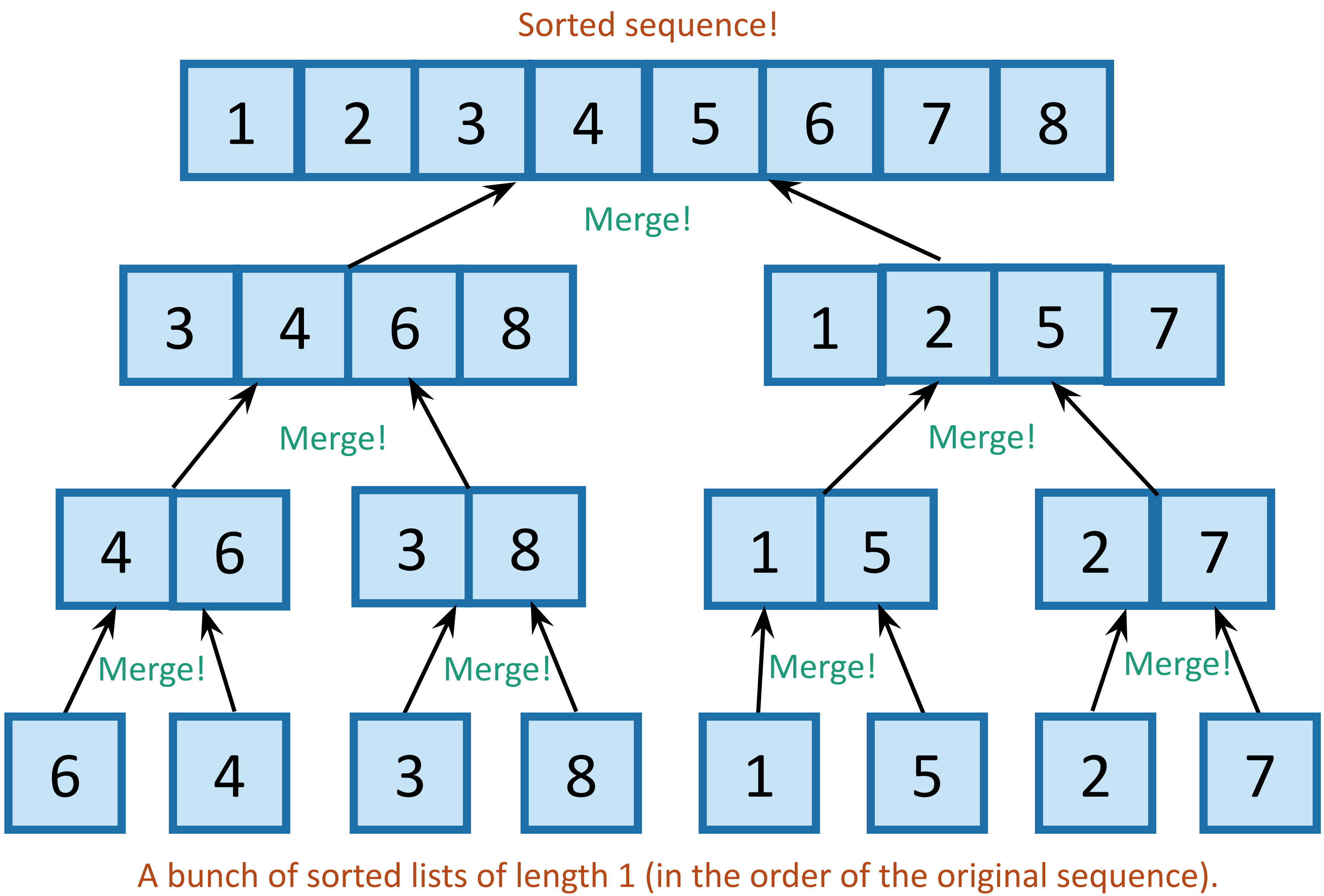
What actually happens?

First, recursively break up the array all the way down to the base cases



This array of length 1 is sorted!

Then, merge them all back up!



Two questions

1. Does this work?
2. Is it fast?

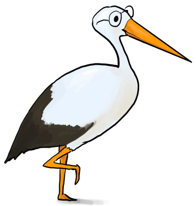
It works Let's assume $n = 2^t$

Again we'll use induction.
This time with an invariant
that will remain true after
every recursive call.

- Inductive hypothesis:

"In every recursive call,
MERGESORT returns a sorted array."

- Base case (n=1): a 1-element array is always sorted.
- Inductive step: Suppose that L and R are sorted. Then MERGE(L,R) is sorted.
- Conclusion: "In the top recursive call, MERGESORT returns a sorted array."

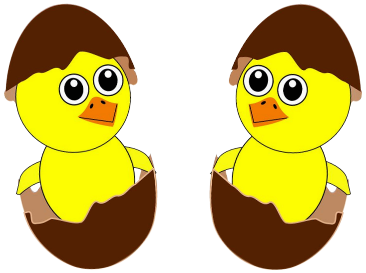


Fill in the inductive step! (Either do it yourself or read it in CLRS!)

- n ← length(A)
- **if** n ≤ 1:
  - **return** A
- L ← MERGESORT(A[1 : n/2])
- R ← MERGESORT(A[n/2+1 : n])
- **return** MERGE(L,R)

Two questions

1. Does this work?
2. Is it fast?



Think-Pair-Share:
(2 min: try to think- how fast is MergeSort?
2 min: what does the person next to you think? why?)

MergeSort Pseudocode

MERGESORT(A):

- n ← length(A)
- **if** n ≤ 1:
  - **return** A

If A has length 1,
It is already sorted!

- L ← MERGESORT(A[0 : n/2])

Sort the left half

- R ← MERGESORT(A[n/2 : n])

Sort the right half

- **return** MERGE(L,R)

Merge the two halves

It's fast Let's keep assuming $n = 2^t$

CLAIM:

MERGESORT requires at most $11n(\log(n) + 1)$ operations to sort n numbers.

What exactly is an "operation" here?
We're leaving that vague on purpose.
Also I made up the number 11.

How does this compare to InsertionSort?

Scaling like n^2 vs scaling like $n\log(n)$?

Quick log refresher

All logarithms in this course are base 2

- log(n) : how many times do you need to divide n by 2 in order to get down to 1?

32
16
8
4
2
1

log(32) = 5

64
32
16
8
4
2
1

log(64) = 6

log(128) = 7
log(256) = 8
log(512) = 9
.
.
.
log(**number of particles in the universe**) < 280

Moral: log(n) grows very slowly with n.

It's fast!

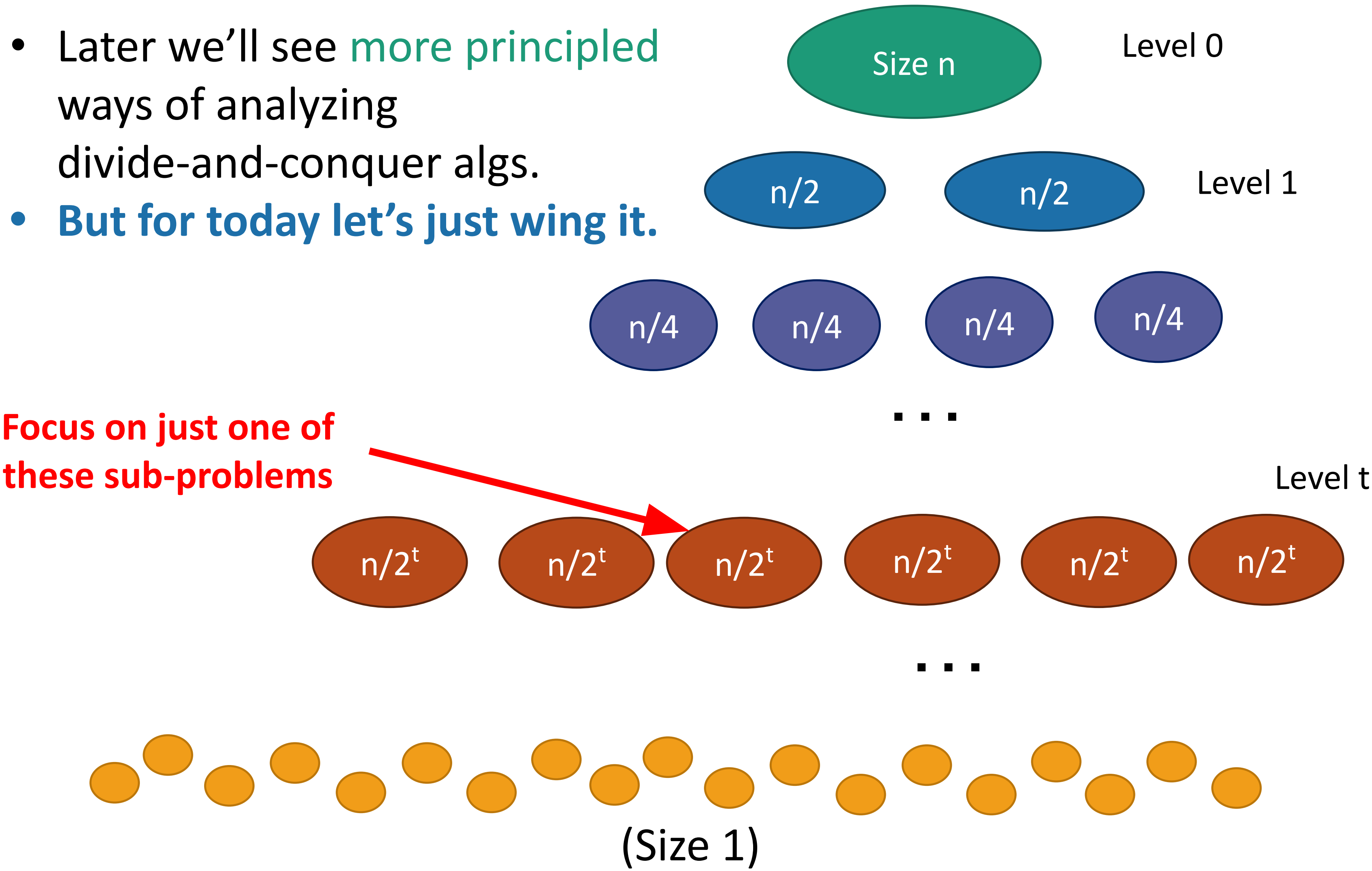
CLAIM:

MERGESORT requires at most $11n(\log(n) + 1)$ operations to sort n numbers.

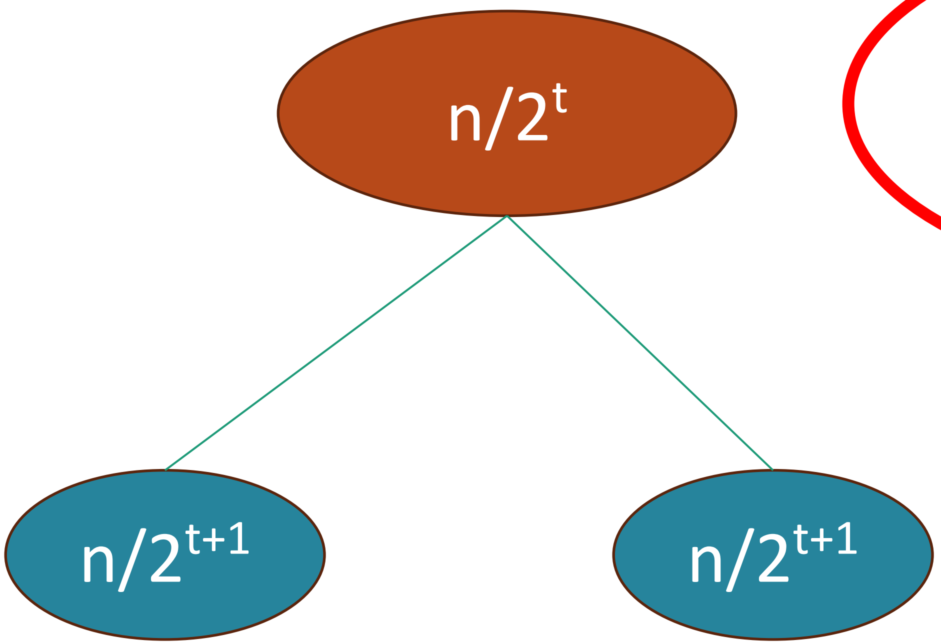
Much faster than InsertionSort for large n!

Let's prove the claim

- Later we'll see more principled ways of analyzing divide-and-conquer algs.
- **But for today let's just wing it.**



How much work in this sub-problem?



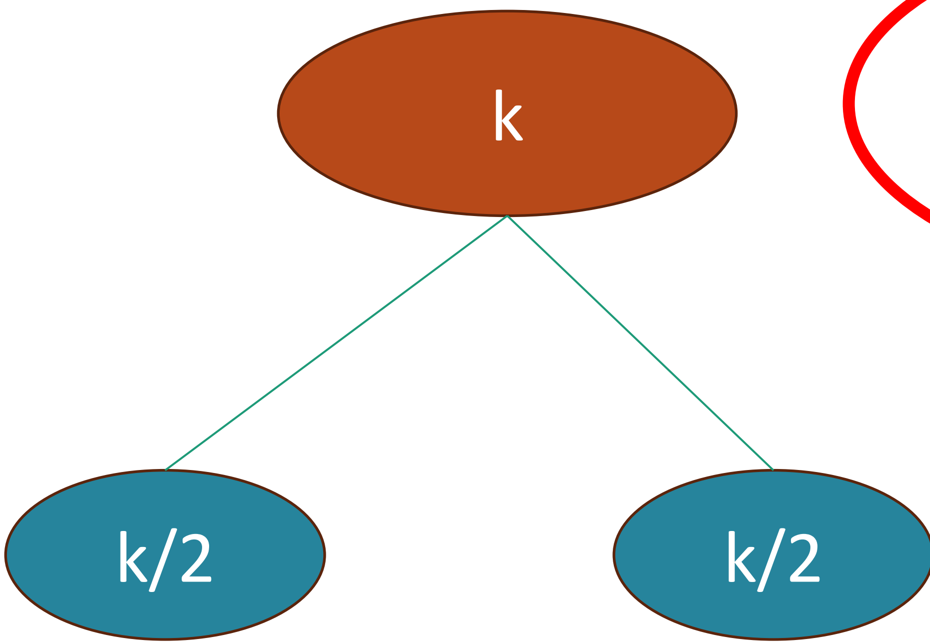
Time spent MERGE-ing the two subproblems

+

Time spent within the two sub-problems

How much work in this sub-problem?

Let $k=n/2^t$…

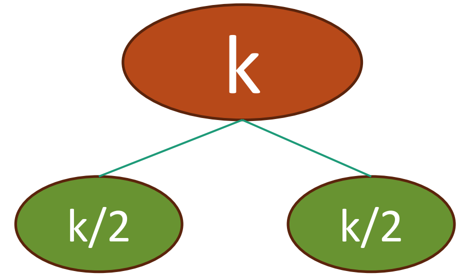


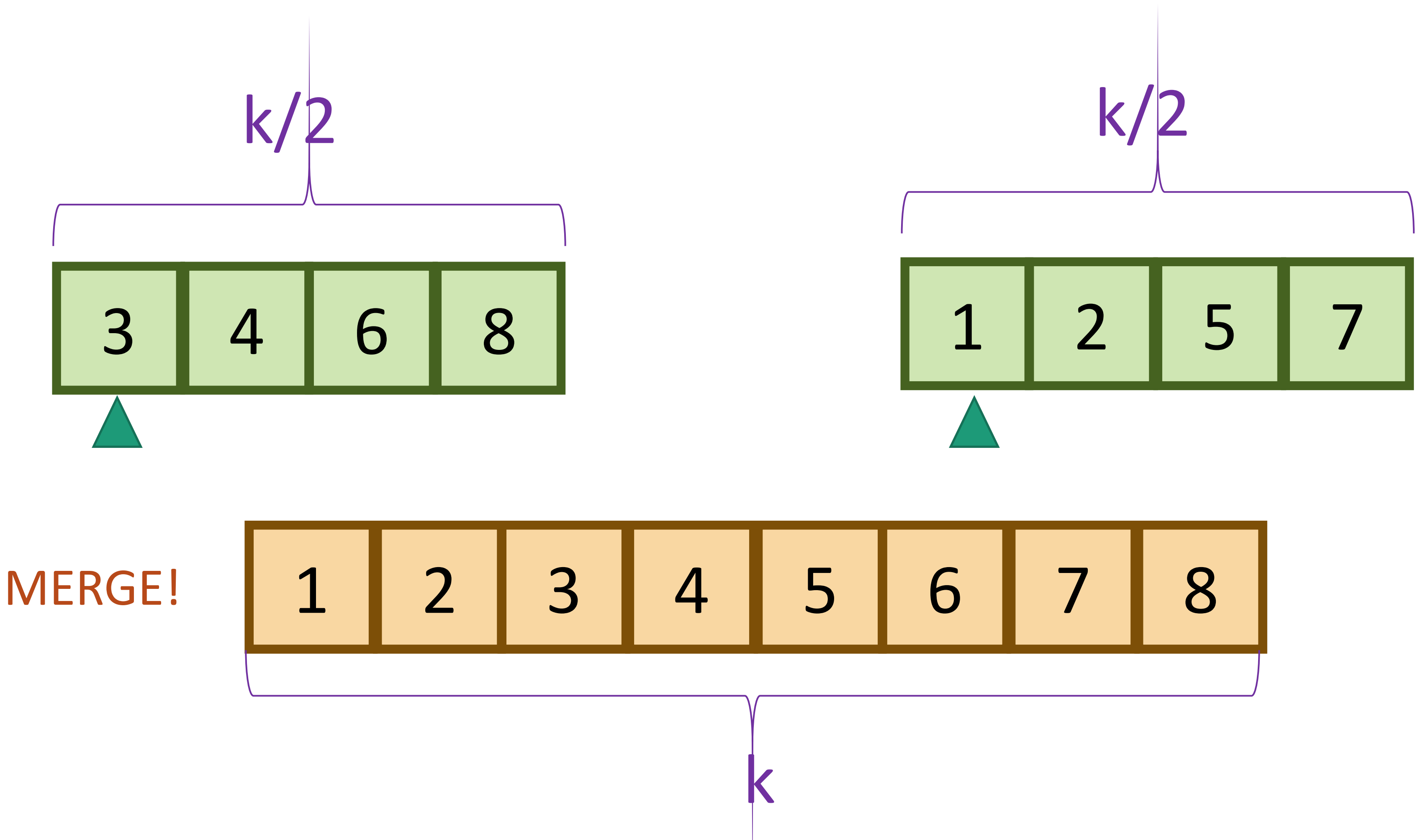
Time spent MERGE-ing the two subproblems

+

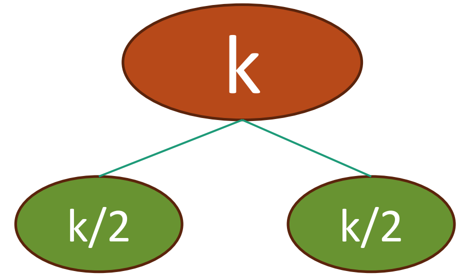
Time spent within the two sub-problems

How long does it take to MERGE?





How long does it take to MERGE?



- Time to initialize an array of size k
- Plus the time to initialize three counters
- Plus the time to increment two of those counters k/2 times each
- Plus the time to compare two values at least k times
- Plus the time to copy k values from the existing array to the big array.
- Plus…

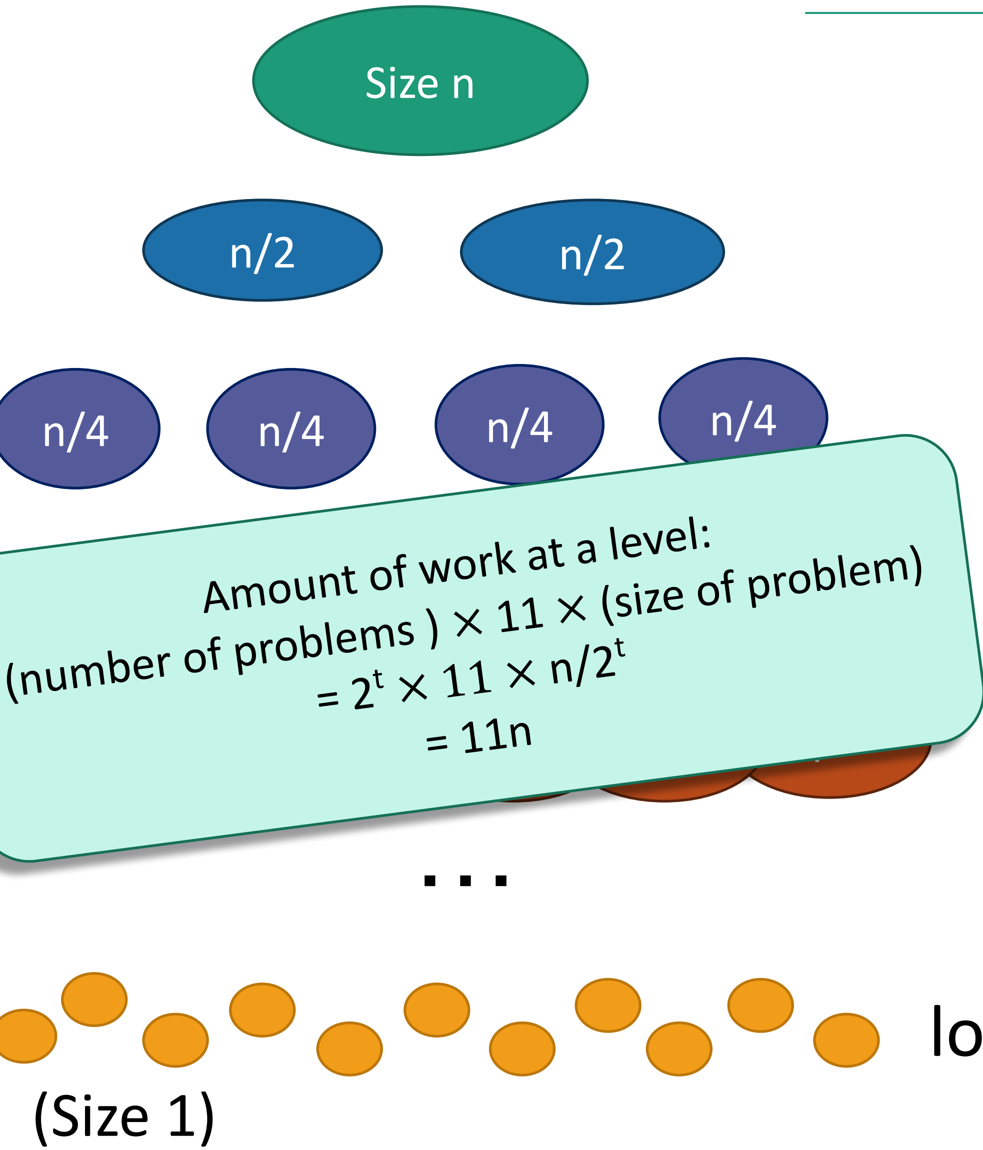
Plucky the Pedantic Penguin

Let’s say no more than 11k operations.

There’s some justification for this number “11” in the lecture notes, but it’s really pretty arbitrary.

Lucky the lackadaisical lemur

Recursion tree



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	11n
1	2	n/2	11n
2	4	n/4	11n
…			
t	2^t	$n/2^t$	11n
…			
log(n)	n	1	11n

Total runtime…

- 11n steps per level, at every level
- log(n) + 1 levels
- 11n (log(n) + 1) steps total

That was the claim!

A few reasons to be grumpy

- Sorting

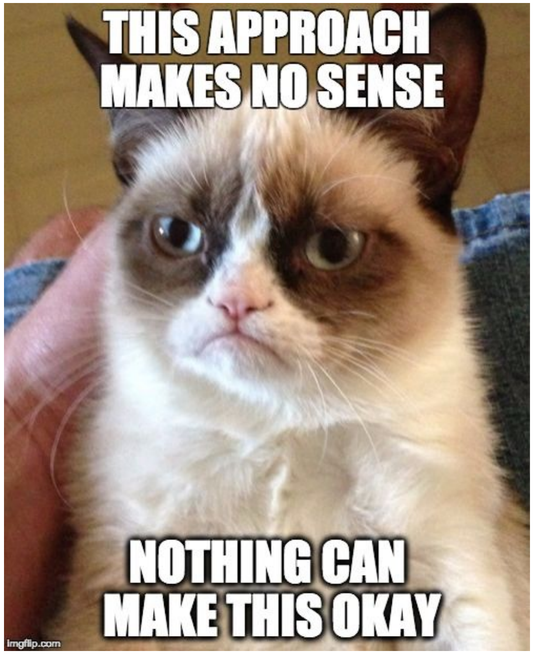
1	2	3	4	5	6	7	8

should take zero steps…

- What’s with this 11k bound?
  - You made that number “11” up.
  - Different operations don’t take the same amount of time.

How we will deal with **grumpiness**

- Take a deep breath…
- Worst case analysis
- Asymptotic notation



The plan

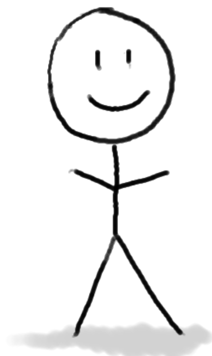
- Part I: Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?
  - **Skills:**
    - Analyzing correctness of iterative and recursive algorithms.
    - Analyzing running time of recursive algorithms (part A)
- Part II: How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis

Worst-case analysis

Sorting a sorted list
should be fast!!

1	2	3	4	5	6	7	8

- In this class, we will focus on worst-case analysis



Algorithm
designer

Here is my algorithm!

```
Algorithm:
   Do the thing
   Do the stuff
   Return the answer
```

Here is an input!

- Pros: very strong guarantee
- Cons: very strong guarantee

Why worst-case analysis?

The real reasons:

1. We don't really know anything much better
   - Very popular these days: "average case analysis"
   - Downside: we typically don't know what an average input looks like.

2. Best-case + worst-case $\neq$ average-case

   Best-case + worst case = worst-case

   $O(1) + O(n \log n) = O(n \log n)$

Big-O notation

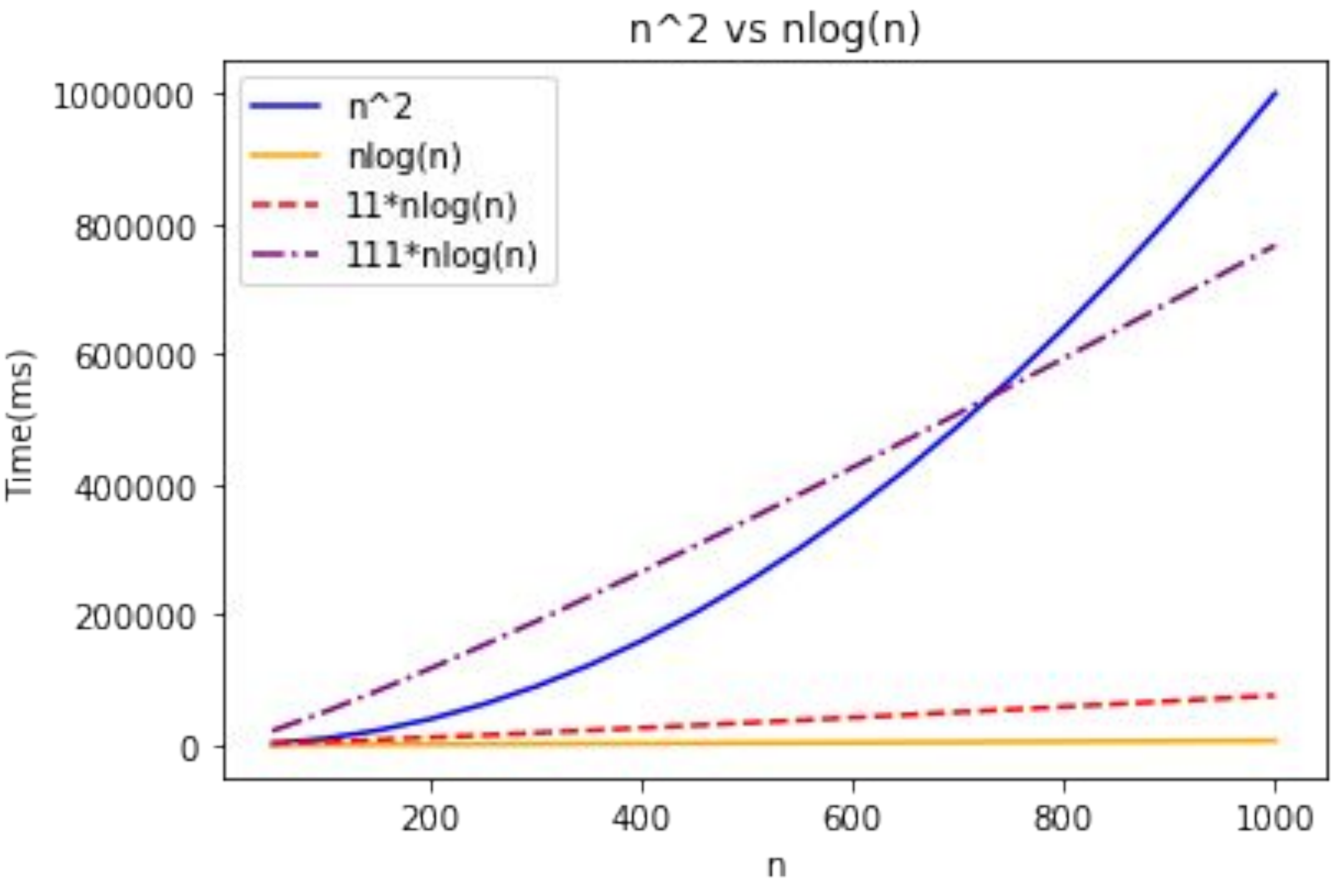
How long does an operation take? Why are we being so sloppy about that "11"?

- What do we mean when we measure runtime?
  - We probably care about wall time: how long does it take to solve the problem, in seconds or minutes or hours?

- This is heavily dependent on the programming language, architecture, etc.
- These things are very important, but are not the point of this class.
- We want a way to talk about the running time of an algorithm, independent of these considerations.

Main idea:

Focus on how the runtime **scales** with n (the input size).



Asymptotic Analysis

How does the running time scale as n gets large?

One algorithm is "faster" than another if its runtime scales better with the size of the input.

Pros:

- Abstracts away from hardware- and language-specific issues.
- Makes algorithm analysis much more tractable.

Cons:

- Only makes sense if n is large (compared to the constant factors).

$2^{100000000000000}\, n$
is "better" than n^2 ?!?!

pronounced “big-oh of …” or sometimes “oh of …”

O(…) means an upper bound

- Let T(n), g(n) be positive functions of positive integers.
  - Think of T(n) as being a runtime: positive and increasing in n.

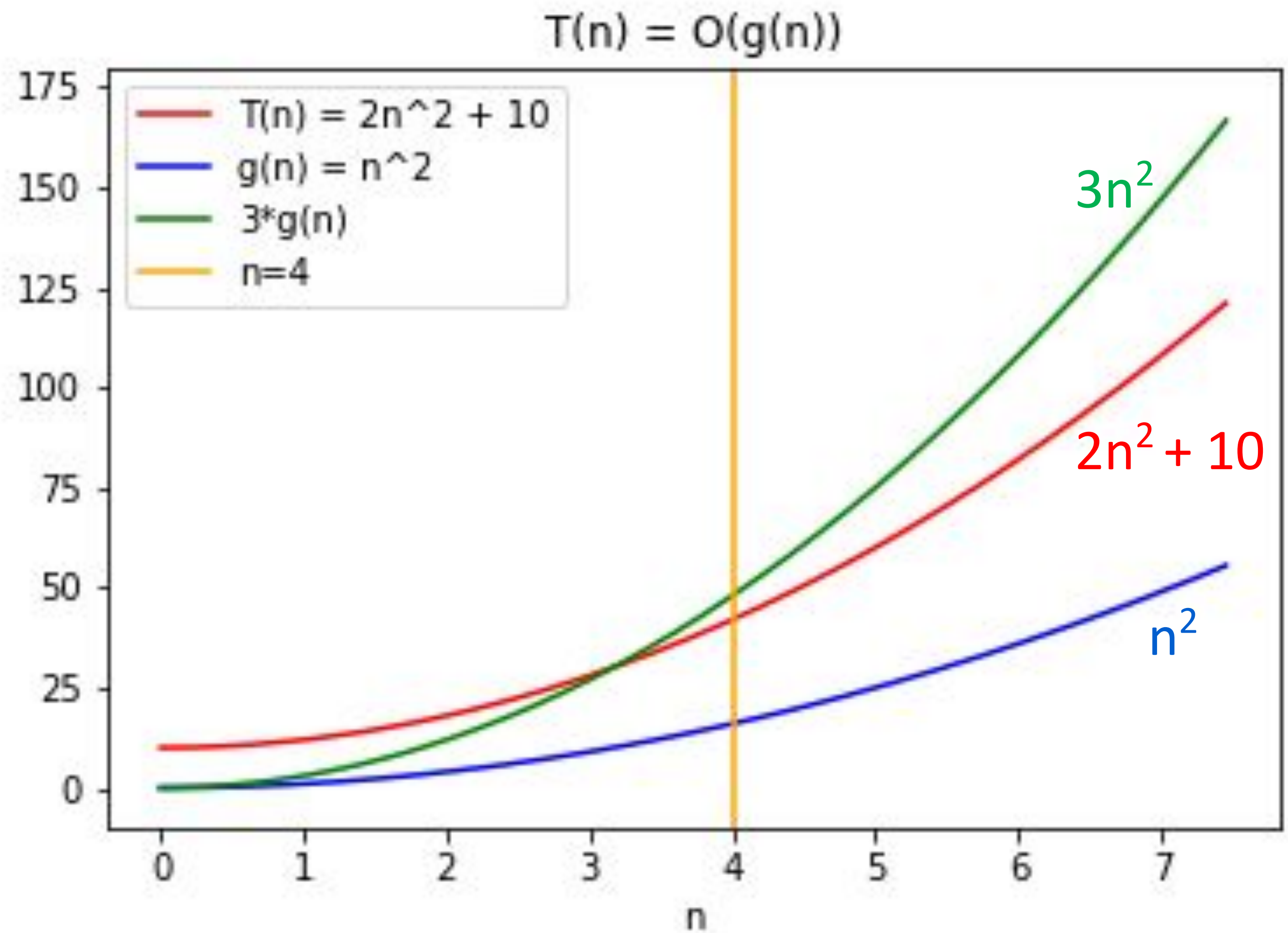
- We say “T(n) is O(g(n))” if g(n) grows at least as fast as T(n) as n gets large.

- Formally,

$$T(n) = O\big(g(n)\big)$$
$$\Longleftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$

Example
$2n^2 + 10 = O(n^2)$

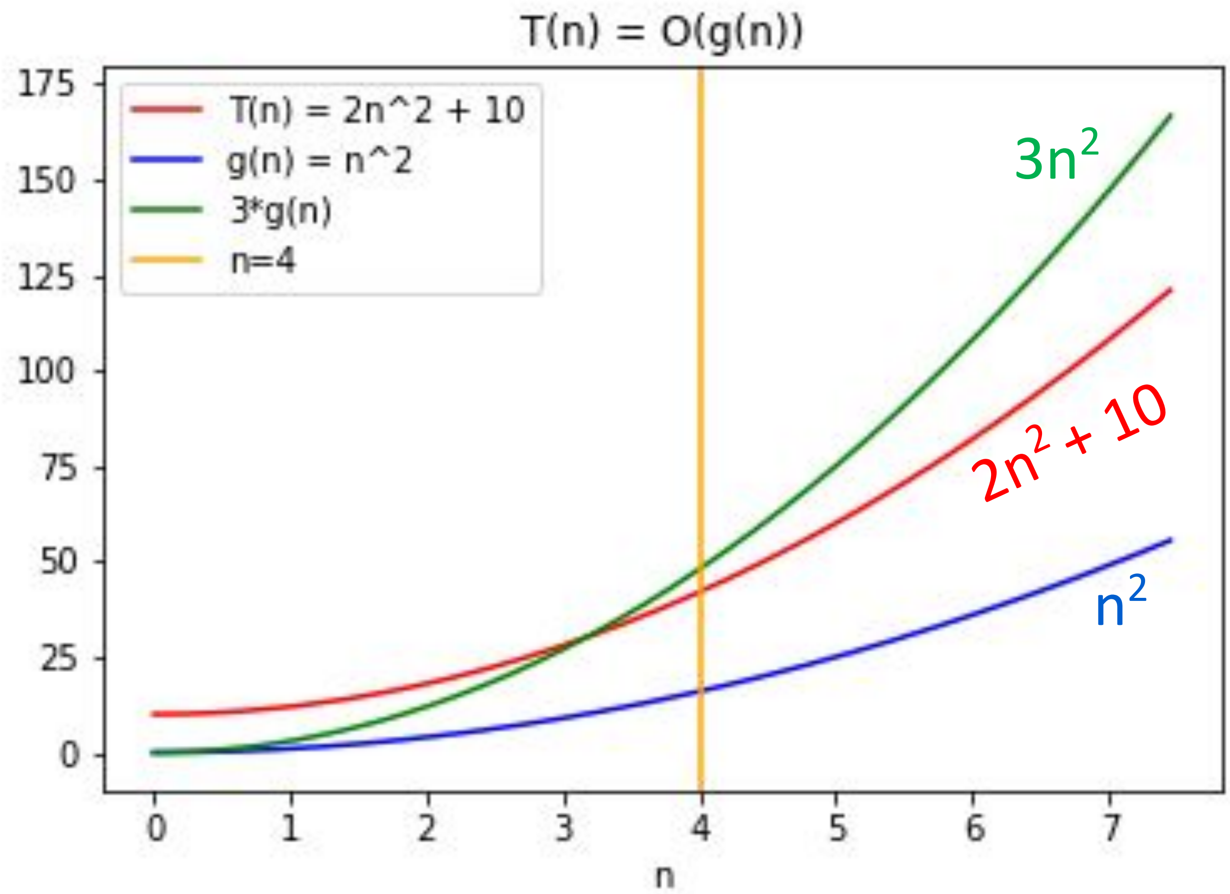
$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



Example

$2n^2 + 10 = O(n^2)$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



Formally:

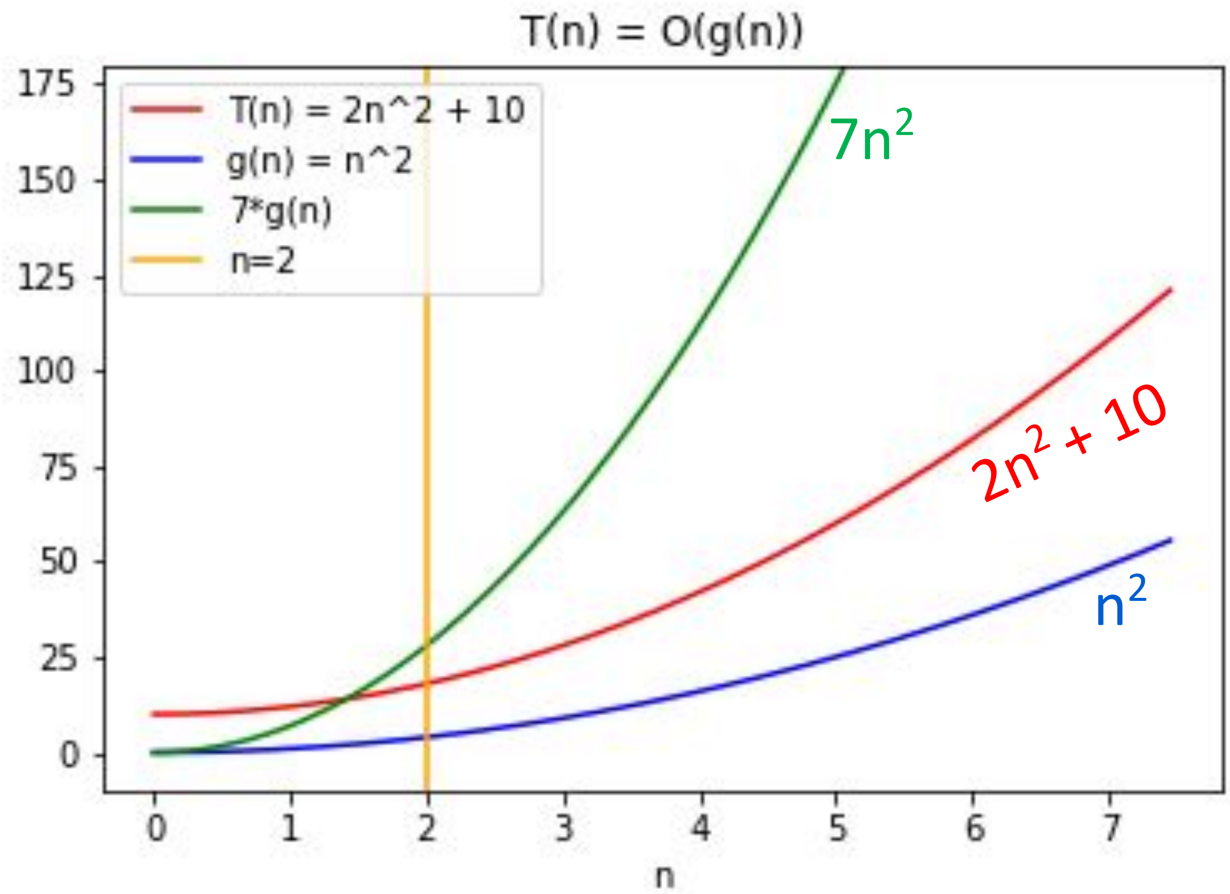
- Choose c = 3
- Choose n_0 = 4
- Then:

$$\forall n \geq 4,$$
$$0 \leq 2n^2 + 10 \leq 3 \cdot n^2$$

same Example

$2n^2 + 10 = O(n^2)$

$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



Formally:

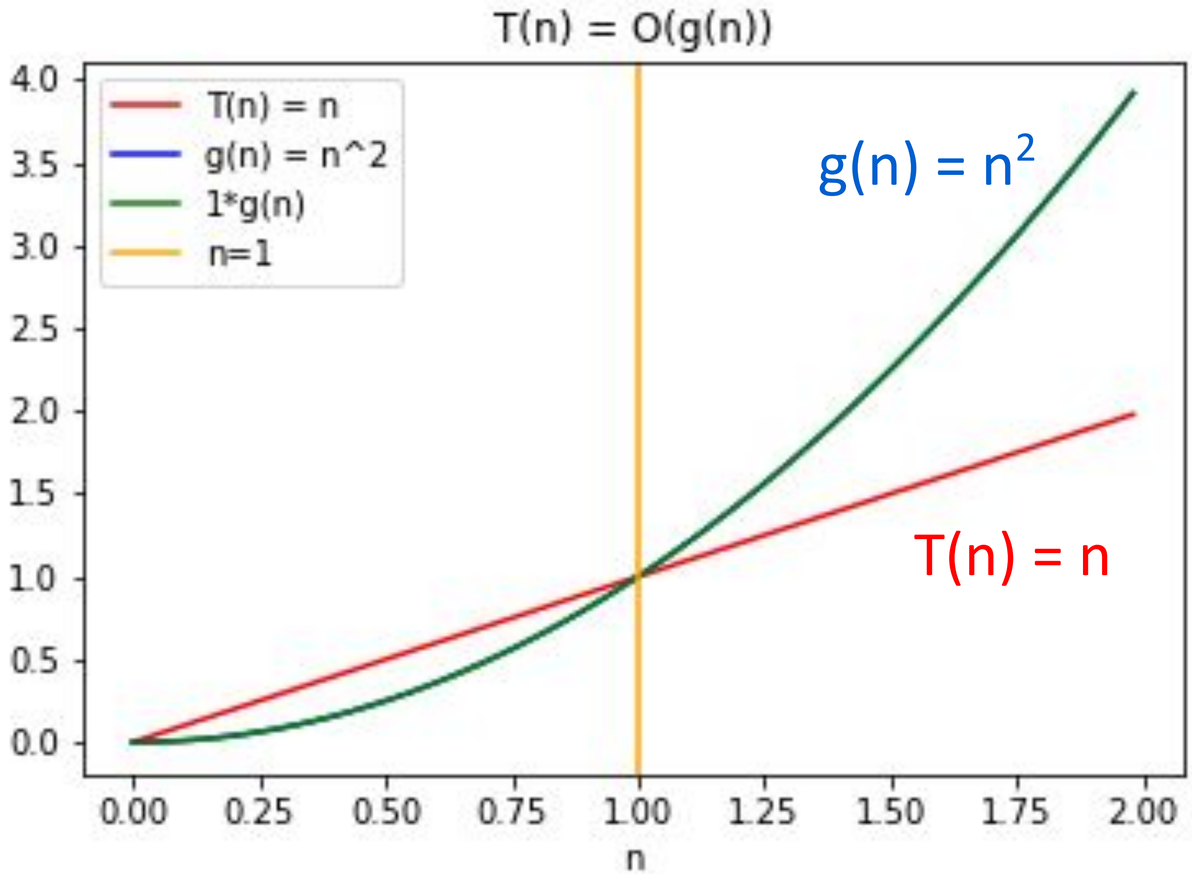
- Choose c = 7
- Choose n_0 = 2
- Then:

$$\forall n \geq 2,$$
$$0 \leq 2n^2 + 10 \leq 7 \cdot n^2$$

There isn't a unique "correct" choice of c and n_0

Another example:
$n = O(n^2)$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



- Choose c = 1
- Choose n_0 = 1
- Then

$$\forall n \geq 1,$$
$$0 \leq n \leq n^2$$

Ω(…) means a lower bound

- We say "T(n) is Ω(g(n))" if T(n) grows at least as fast as g(n), as n gets large.

- Formally,

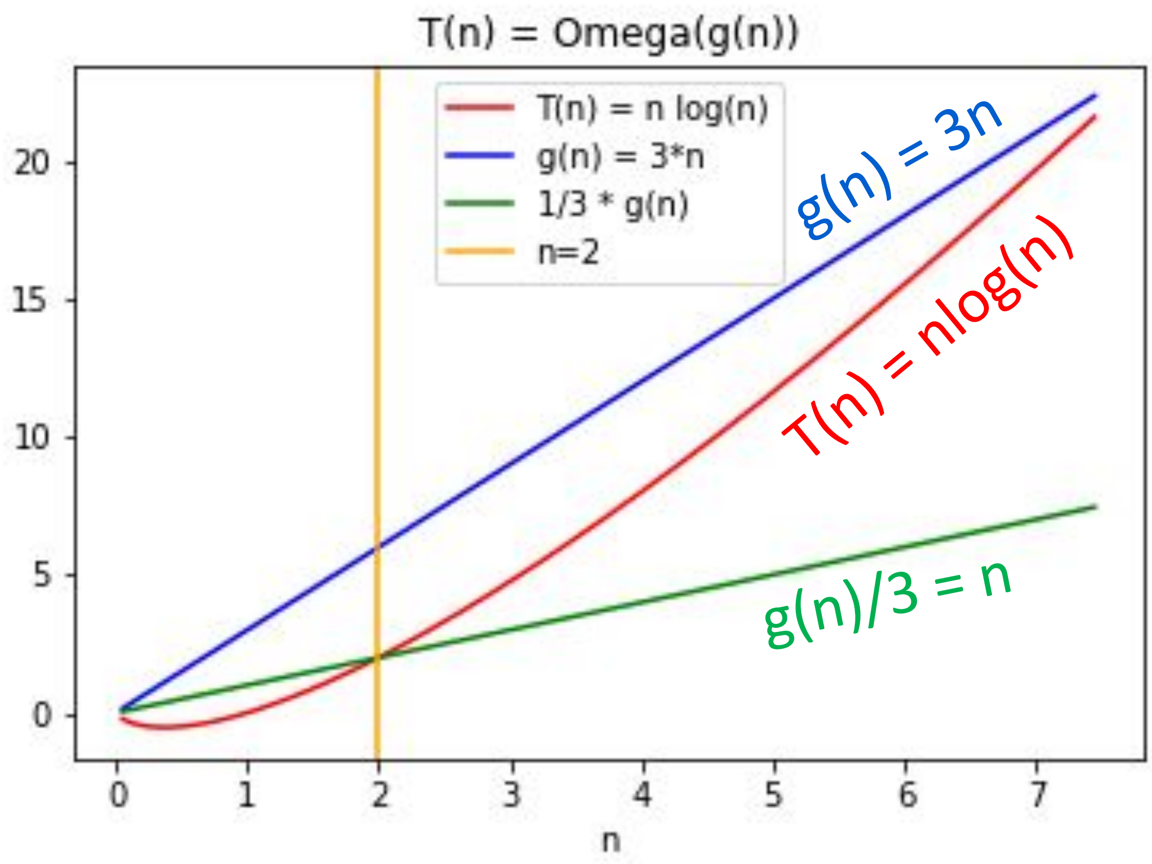
$$T(n) = \Omega\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq c \cdot g(n) \leq T(n)$$

Switched these!!

Example

$n \log_2(n) = \Omega(3n)$

$$T(n) = \Omega(g(n))$$
$$\Longleftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq c \cdot g(n) \leq T(n)$$



- Choose c = 1/3
- Choose n_0 = 2
- Then

$$\forall n \geq 2,$$

$$0 \leq \frac{3n}{3} \leq n \log_2(n)$$

$\Theta(\ldots)$ means both!

- We say "$T(n)$ is $\Theta(g(n))$" if:

$$T(n) = O(g(n))$$

-AND-

$$T(n) = \Omega(g(n))$$

Some more examples

- All degree k polynomials are $O(n^k)$
- For any $k \geq 1$, n^k is not $O(n^{k-1})$

(On the board if we have time… if not see the lecture notes!)

Take-away from examples

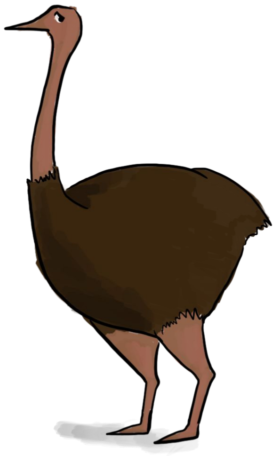
- To prove $T(n) = O(g(n))$, you have to come up with c and n_0 so that the definition is satisfied.

- To prove $T(n)$ is NOT $O(g(n))$, one way is **proof by contradiction**:
  - Suppose (to get a contradiction) that someone gives you a c and an n_0 so that the definition ***is*** satisfied.
  - Show that this someone must by lying to you by deriving a contradiction.

Some brainteasers

- Are there functions f, g so that NEITHER $f = O(g)$ nor $f = \Omega(g)$?
- Are there non-decreasing functions f, g so that the above is true?
- Define the n'th fibonacci number by $F(0) = 1$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$ for $n > 2$.
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

  True or false:
  - $F(n) = O(2^n)$
  - $F(n) = \Omega(2^n)$



Ollie the Over-achieving Ostrich

What have we learned?

Asymptotic Notation

- This makes both Plucky and Lucky happy.
  - **Plucky the Pedantic Penguin** is happy because there is a precise definition.
  - **Lucky the Lackadaisical Lemur** is happy because we don't have to pay close attention to all those pesky constant factors like "11".

- But we should be careful not to abuse it.

- In this course, (almost) every algorithm we see will be actually practical, without needing to take $n \geq n_0 = 2^{10000000}$.

The plan

- Part I: Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?
  - **Skills:**
    - Analyzing correctness of iterative and recursive algorithms.
    - Analyzing running time of recursive algorithms (part A)
- Part II: How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis

Wrap-Up

Recap

- InsertionSort runs in time **$O(n^2)$**
- MergeSort is a divide-and-conquer algorithm that runs in time **$O(n \log(n))$**

- How do we show an algorithm is correct?
  - Today, we did it by induction
- How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic analysis

Next time

- A more systematic approach to analyzing the runtime of recursive algorithms.