

Lecture 2

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

Announcements

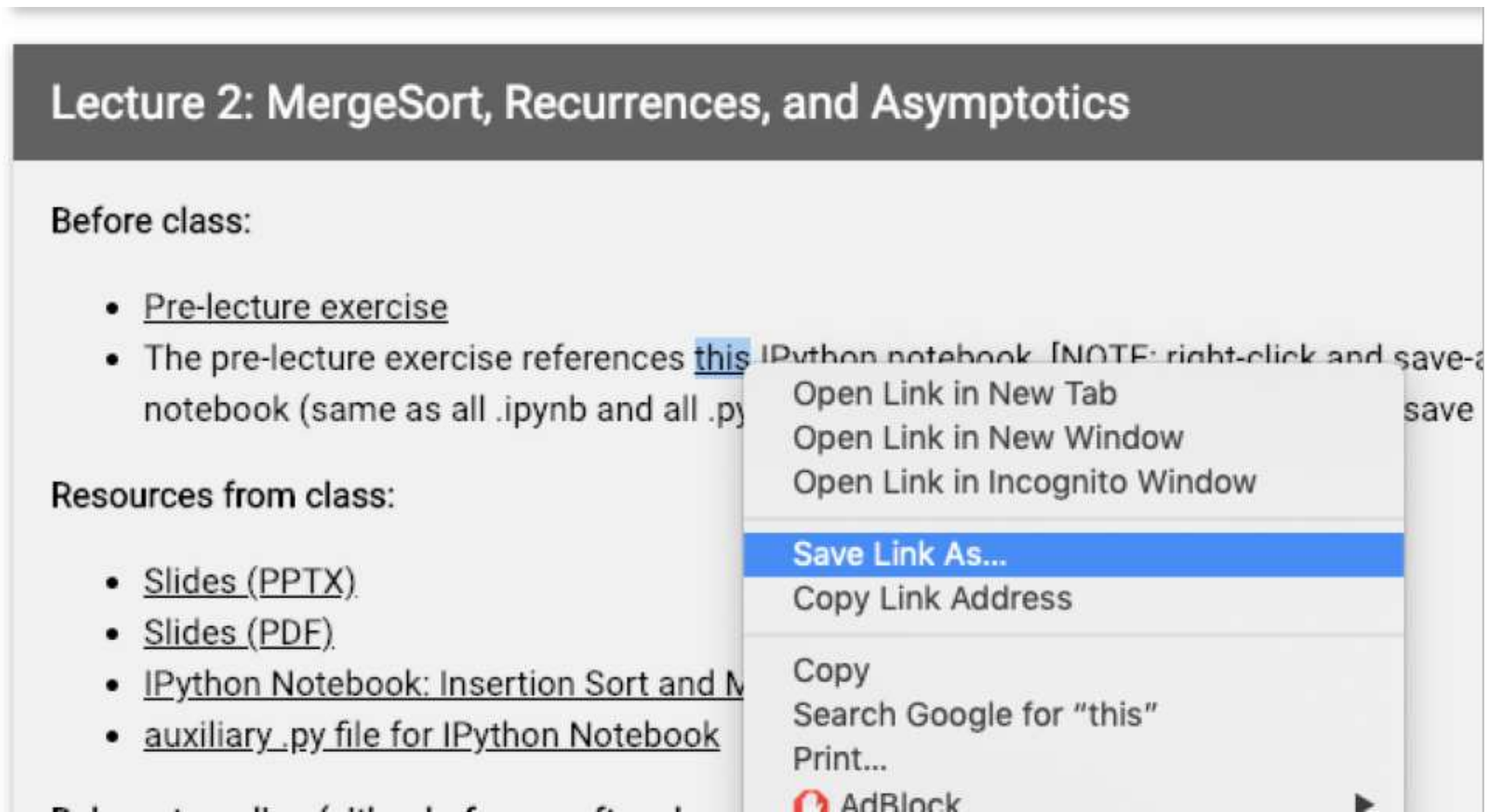
- Sign up for Piazza!
 - Connect with course staff!
 - Connect with your colleagues!
 - Form study groups!
 - Receive course announcements!
- Please (continue to) send OAE requests and exam conflicts to cs161-win1819-staff@lists.stanford.edu

More Announcements

- Homework!
 - HW1 will be released **Friday**.
 - It is due the following **Friday**.
- See the website for guidelines on homework:
 - Collaboration policy
 - Best practices
 - Example Homework!!!!
 - LaTeX help.

How to download IPython notebooks

- Right-click and “Save link as” (or equivalent):



The screenshot shows a web browser window with a dark header bar containing the text "Lecture 2: MergeSort, Recurrences, and Asymptotics". Below the header, the text "Before class:" is followed by a bulleted list. The second item in the list is "The pre-lecture exercise references [this](#) IPython notebook (same as all .ipynb and all .py)". A right-click context menu is open over the word "this", with the option "Save Link As..." highlighted in blue. Other options in the menu include "Open Link in New Tab", "Open Link in New Window", "Open Link in Incognito Window", "Copy Link Address", "Copy", "Search Google for 'this'", and "Print...". At the bottom of the menu, there is a red shield icon and the text "AdBlock".

Lecture 2: MergeSort, Recurrences, and Asymptotics

Before class:

- [Pre-lecture exercise](#)
- The pre-lecture exercise references [this](#) IPython notebook (same as all .ipynb and all .py)

Resources from class:

- [Slides \(PPTX\)](#)
- [Slides \(PDF\)](#)
- [IPython Notebook: Insertion Sort and Merge Sort](#)
- [auxiliary .py file for IPython Notebook](#)

Right-click context menu options:

- Open Link in New Tab
- Open Link in New Window
- Open Link in Incognito Window
- Save Link As...**
- Copy Link Address
- Copy
- Search Google for "this"
- Print...
- AdBlock

Office Hours and Sections

- We are still working on scheduling these!
 - We will post an announcement on Piazza soon when the schedule is finalized.
 - Don't worry, there will be plenty of office hours!
- You don't need to formally enroll in sections.

Piazza Heroes


- Thanks for helping out your colleagues!

Top Piazza Answerers:

Name, Email	number of answers
Ashish Paliwal	7
Esther Cherin Kim	2
Pranav Jain	2
Richard Lin	2
Avery Wang	1
Trenton Chang	1

Bonus slide on Karatsuba

- I realized based on the questions after class on Monday that I was unclear about how the Karatsuba recursion worked – sorry!
- I added a bonus slide with more details, you can take a look at that or at the python code.
- If you are still confused, ask on Piazza or OH!



THIS SLIDE ADDED FOR CLARIFICATION AFTER LECTURE

How would this work?

x, y are n-digit numbers

(Still not super precise, see [IPython notebook](#) for detailed code. Also, still assume n is a power of 2.)

Multiply(x, y):

- If $n=1$:
 - Return xy
- Write $x = a 10^{\frac{n}{2}} + b$ and $y = c 10^{\frac{n}{2}} + d$ *a, b, c, d are n/2-digit numbers*
- $ac = \text{Multiply}(a, c)$
- $bd = \text{Multiply}(b, d)$
- $z = \text{Multiply}(a+b, c+d)$ *We can do the addition $a+b$ and $c+d$ in time $O(n)$. This results in integers that are still roughly $n/2$ bits long.*
- $\text{cross_terms} = z - ac - bd$ *The quantity cross_terms is meant to be $(ad + bc)$*
- $xy = ac 10^n + (\text{cross_terms}) 10^{n/2} + bd$
- Return xy

Thanks for the anonymous feedback!

- Keep it coming!
- Where are the videos?
 - On canvas (link from website), or mvideox (link from website)
 - But! SCPD warns that it will be moving exclusively to Canvas and suggests you use Canvas.
- “I’m worried that I’ll slow the class down if I ask questions.”
 - Don’t worry! It’s my job to keep the class on pace, it’s your job to let me know if you are confused.
 - (I know it can be intimidating...but please know that I really value your questions!)

End of announcements!

Cast

Last time

Philosophy

- Algorithms are awesome!
- Our motivating questions:
 - Does it work?
 - Is it fast?
 - Can I do better?

Technical content

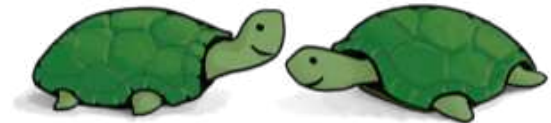
- Karatsuba integer multiplication
- Example of “Divide and Conquer”
- Not-so-rigorous analysis



Plucky the pedantic
penguin



Lucky the
lackadaisical lemur



Think-Pair-Share
Terrapins



Ollie the
over-achieving ostrich



Siggie the
studious stork

Today

- We are going to ask:
 - Does it work?
 - Is it fast?
- We'll start to see how to answer these by looking at some examples of sorting algorithms.
 - InsertionSort
 - MergeSort

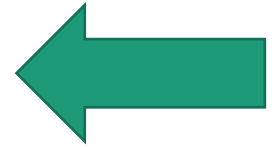


SortingHatSort not discussed

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
---	---	---	---	---	---	---	---

I hope everyone did the pre-lecture exercise!

What was the mystery sort algorithm?

1. MergeSort
2. QuickSort
3. InsertionSort
4. BogoSort

```
def mysteryAlgorithmOne(A):  
    for x in A:  
        B = [None for i in range(len(A))]  
        for i in range(len(B)):  
            if B[i] == None or B[i] > x:  
                j = len(B)-1  
                while j > i:  
                    B[j] = B[j-1]  
                    j -= 1  
                B[i] = x  
                break  
    return B
```

```
def MysteryAlgorithmTwo(A):  
    for i in range(1, len(A)):  
        current = A[i]  
        j = i-1  
        while j >= 0 and A[j] > current:  
            A[j+1] = A[j]  
            j -= 1  
        A[j+1] = current
```

I hope everyone did the pre-lecture exercise!

What was the mystery sort algorithm?

1. MergeSort
2. QuickSort
3. InsertionSort
4. BogoSort

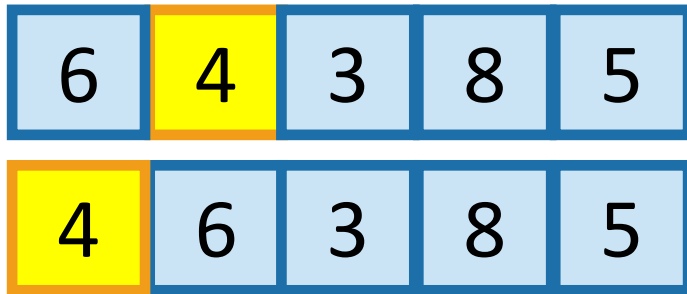
```
def mysteryAlgorithmOne(A):  
    for x in A:  
        B = [None for i in range(len(A))]  
        for i in range(len(B)):  
            if B[i] == None or B[i] > x:  
                j = len(B)-1  
                while j > i:  
                    B[j] = B[j-1]  
                    j -= 1  
                B[i] = x  
                break  
    return B
```

```
def MysteryAlgorithmTwo(A):  
    for i in range(1, len(A)):  
        current = A[i]  
        j = i-1  
        while j >= 0 and A[j] > current:  
            A[j+1] = A[j]  
            j -= 1  
        A[j+1] = current
```

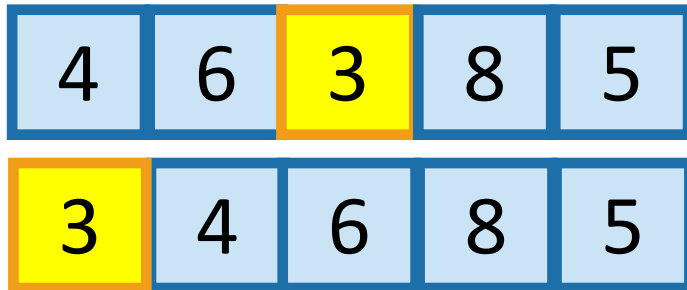
InsertionSort

example

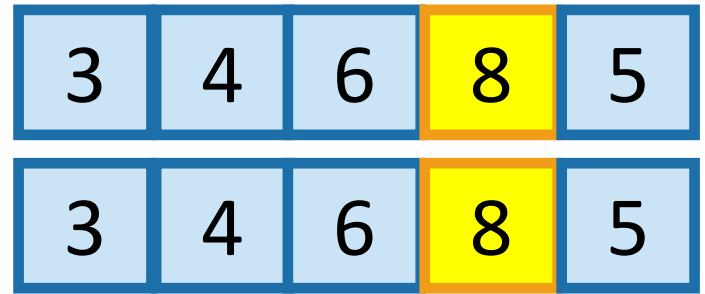
Start by moving $A[1]$ toward the beginning of the list until you find something smaller (or can't go any further):



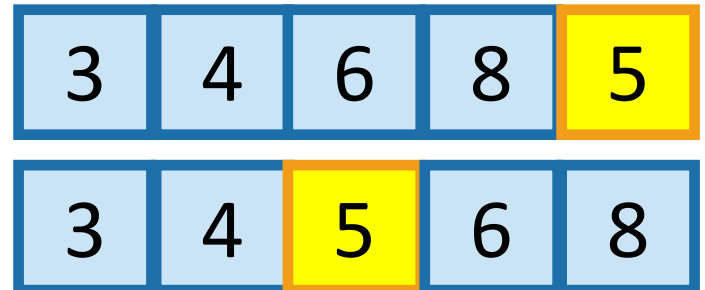
Then move $A[2]$:



Then move $A[3]$:



Then move $A[4]$:



Then we are done!

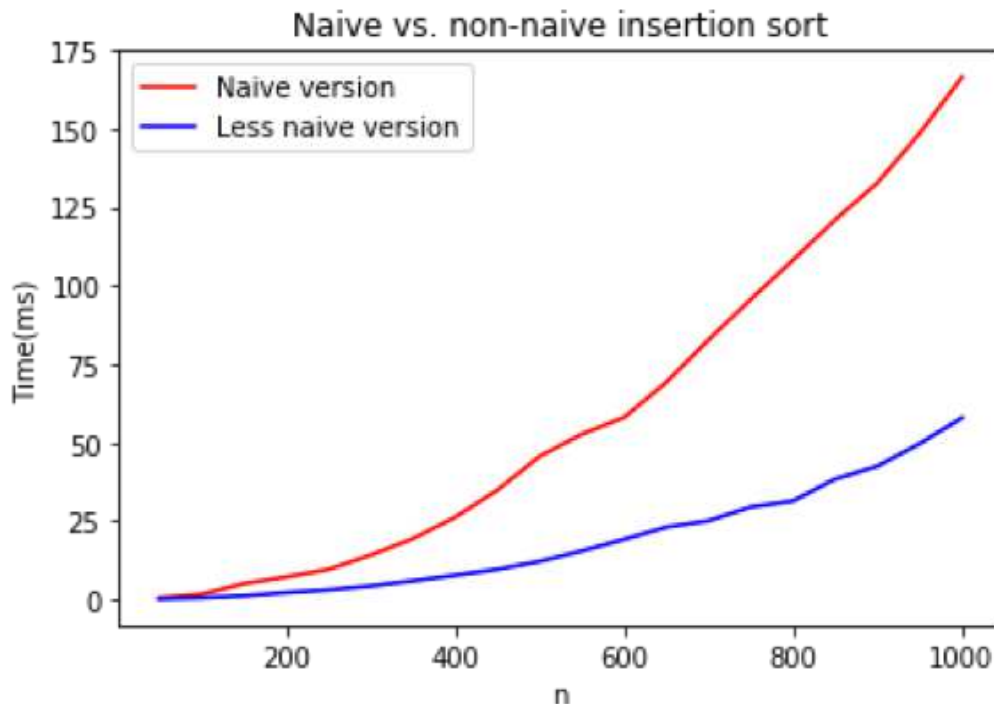
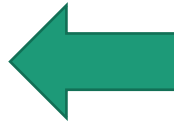
Insertion Sort

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

Insertion Sort

1. Does it work?

2. Is it fast?

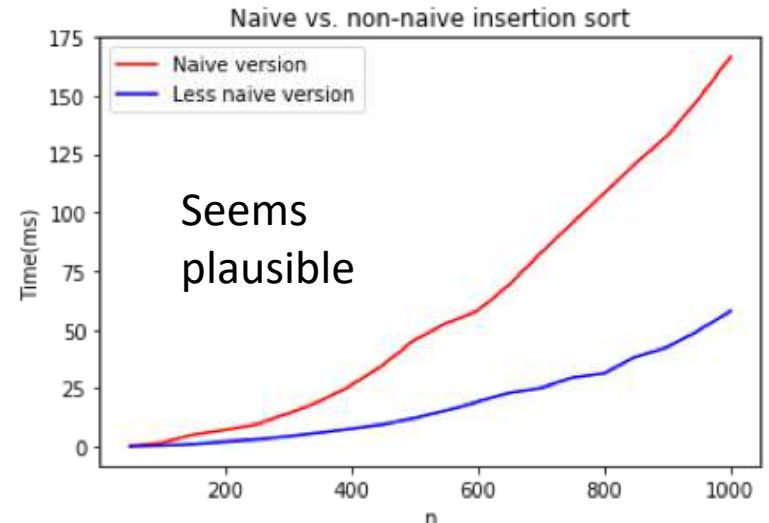


- The “same” algorithm can be faster or slower depending on the implementation...
- We are interested in how fast the running time scales with n , the size of the input.

Insertion Sort: running time

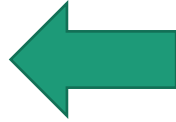
Technically we haven't defined this yet...we'll do it later in this lecture.

- Claim: The running time is $O(n^2)$
- I don't want to focus on this in lecture, but there's a hidden slide to help you verify this later. (Or see CLRS).



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

5

.

- Insert

5

 right after the largest thing that's still smaller than

5

. (Aka, right after

4

).

- Then you get a sorted list:

3	4	5	6	8
---	---	---	---	---

So just use this logic at every step.



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



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



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



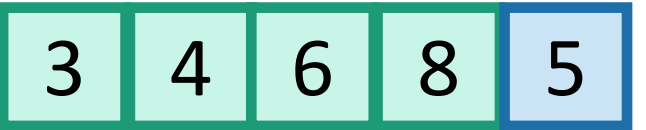
So correctly inserting 3 into the list [4,6] means that [3,4,6] becomes a sorted list.



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



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



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



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


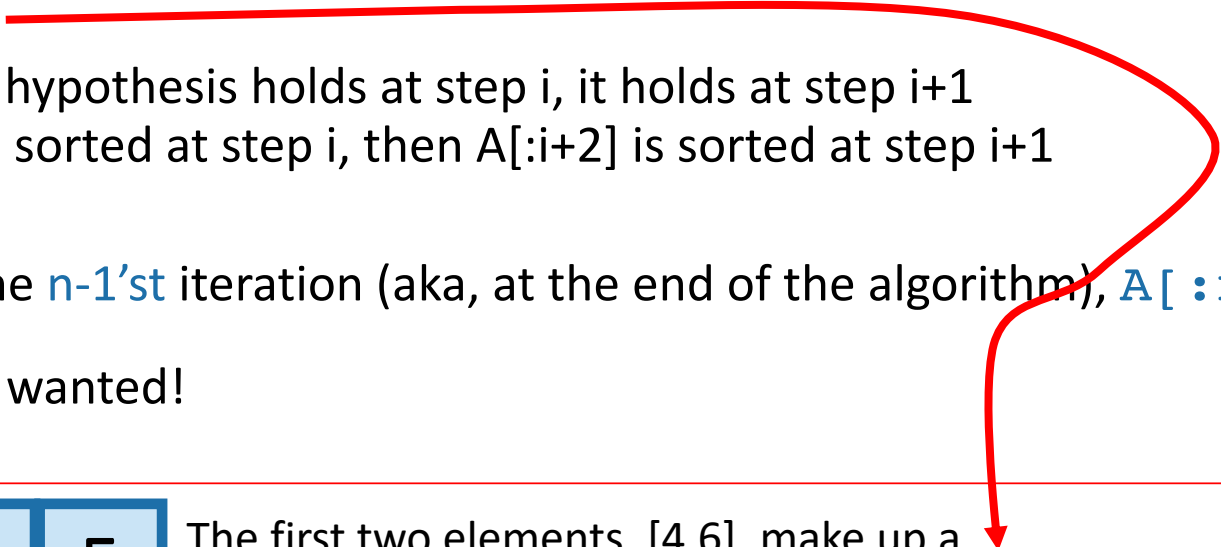

YAY WE ARE DONE!

This sounds like a job for...

**Proof By
Induction!**

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: 
 - If the inductive hypothesis holds at step i , it holds at step $i+1$
 - Aka, if $A[:i+1]$ is sorted at step i , then $A[:i+2]$ is sorted at step $i+1$
- Conclusion:
 - At the end of the $n-1^{\text{st}}$ iteration (aka, at the end of the algorithm), $A[: n] = A$ is sorted.
 - That's what we wanted! 

This logic (see handout for details)



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



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

This was iteration $i=2$.

There is a handout with the details!

- See website!

2 Correctness of InsertionSort

Once you figure out what INSERTIONSORT is doing (see the slides/lecture video for the intuition on this), you may think that it's "obviously" correct. However, if you didn't know what it was doing and just got the above code, maybe this wouldn't be so obvious. Additionally, for algorithms that we'll study in the future, it *won't* always be obvious that it works, and so we'll have to prove it. So in this handout we'll carefully go through a proof that INSERTIONSORT is correct.

We'll do the proof by maintaining a *loop invariant*, in this case that after iteration i , then $A[:i+1]$ is sorted. This is obviously true when $i = 0$ (because the empty list $A[:1] = []$ is definitely sorted) and then we'll show that for any $i > 0$, if it's true for $i - 1$, then it's true for i . At the end of the day, we'll conclude that $A[:n]$ (aka, the whole thing) is sorted and we'll be done.

Formally, we will proceed by induction.

- **Inductive hypothesis.** After iteration i of the outer loop, $A[:i+1]$ is sorted.
- **Base case.** When $i = 0$, $A[:1]$ contains only one element, and this is sorted.
- **Inductive step.** Suppose that the inductive hypothesis holds for $i - 1$, so $A[:i]$ is sorted after the $i - 1$ 'st iteration. We want to show that $A[:i+1]$ is sorted after the i 'th iteration.

Suppose that j^* is the largest integer in $\{0, \dots, i - 1\}$ so that $A[j^*] < A[i]$. Then the effect of the inner loop is to turn

$$[A[0], A[1], \dots, A[j^*], \dots, A[i - 1], A[i]]$$

into

$$[A[0], A[1], \dots, A[j^*], A[i], A[j^* + 1], \dots, A[i - 1]].$$

We claim that this second list is sorted. This is because $A[i] > A[j^*]$, and by the inductive hypothesis,

Aside: proofs by induction

- We're gonna see/do/skip over a lot of them.
- I'm assuming you're comfortable with them from CS103.
 - When you assume...
- If that went by too fast and was confusing:
 - Slides [there's a hidden one with more info about induction]
 - Re-watch the lecture video
 - Handout
 - Book
 - Office Hours

Make sure you really understand the argument on the previous slide! Check out the handout for a formal write-up.



Siggi the Studious Stork

What have we learned?

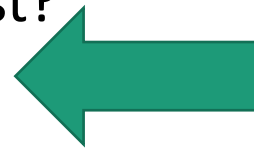
InsertionSort is an algorithm that correctly sorts an arbitrary n -element array in time $O(n^2)$.

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:

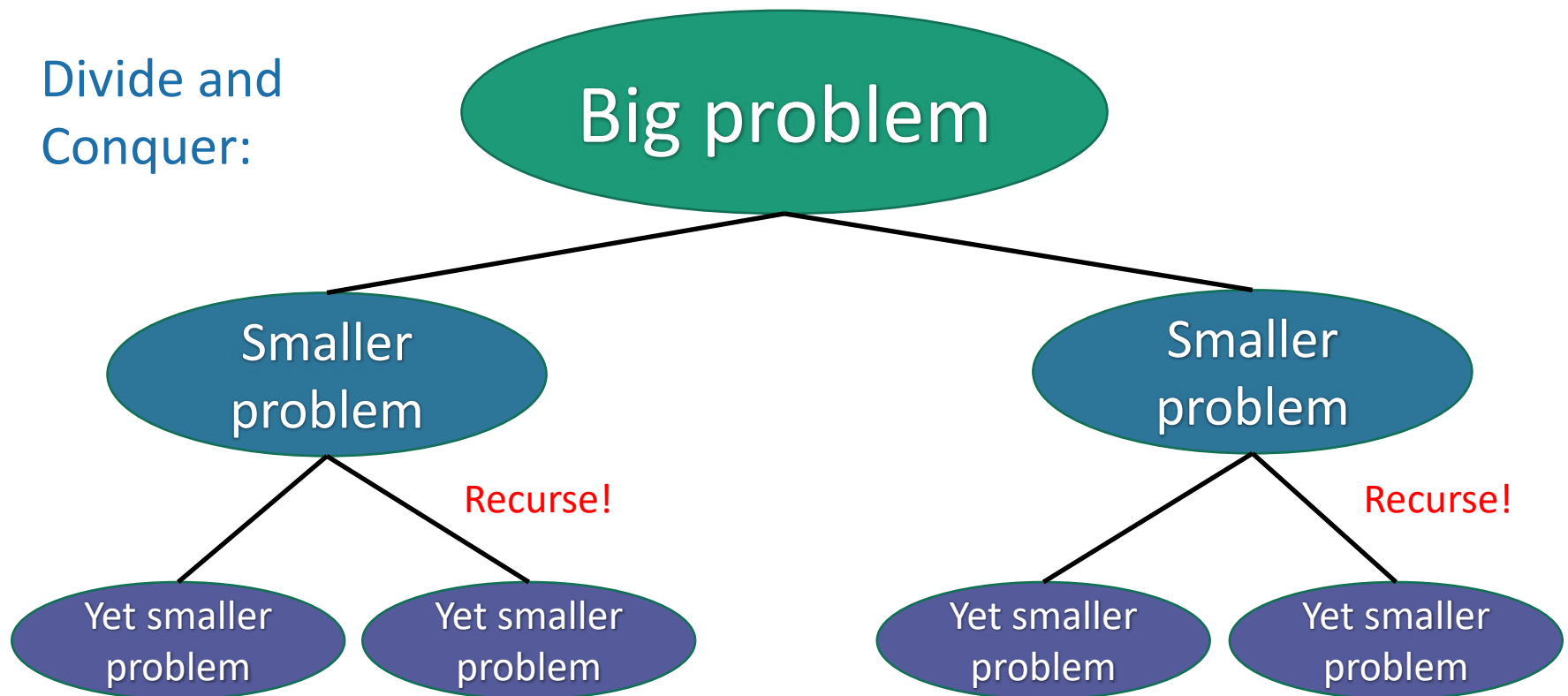
- Analyzing correctness of iterative and recursive algorithms.
- Analyzing running time of recursive algorithms (part A)

- Part II: How do we measure the runtime of an algorithm?

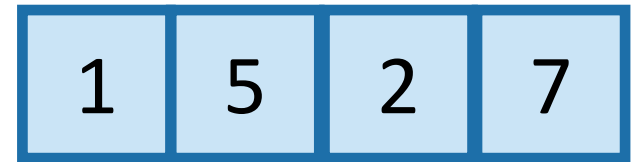
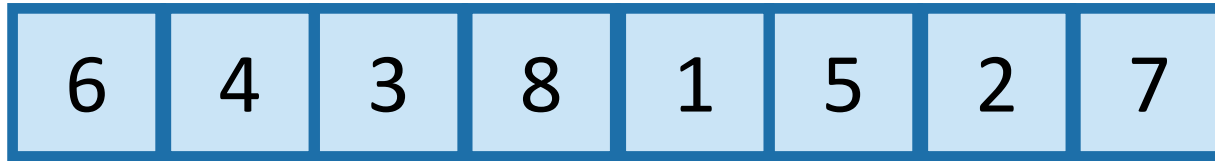
- Worst-case analysis
- Asymptotic Analysis

Can we do better?

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

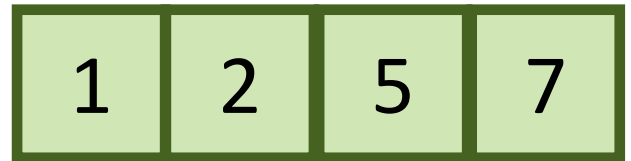
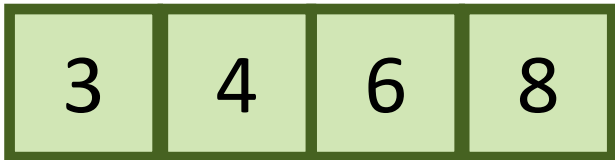


MergeSort



Recursive magic!

Recursive magic!



MERGE!



How would you do this in-place?

Code for the **MERGE** step is given in the Lecture2 IPython notebook, or CRLS

Ollie the over-achieving Ostrich



MergeSort Pseudocode

MERGESORT(A):

- $n = \text{length}(A)$
- **if** $n \leq 1$:
 - **return** A

If A has length 1,
It is already sorted!
- $L = \text{MERGESORT}(A[0 : n/2])$

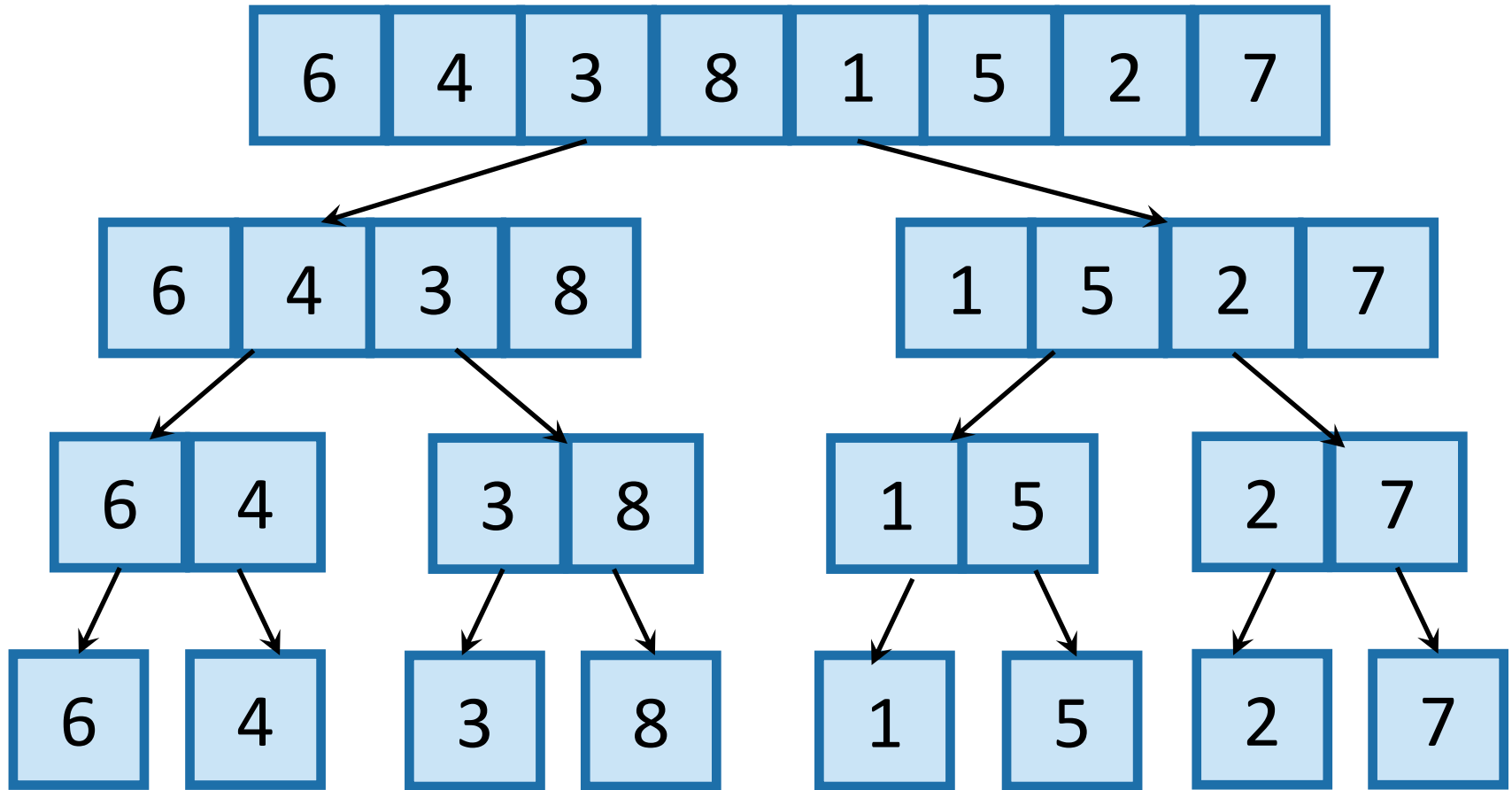
Sort the left half
- $R = \text{MERGESORT}(A[n/2 : n])$

Sort the right half
- **return** **MERGE**(L,R)

Merge the two halves

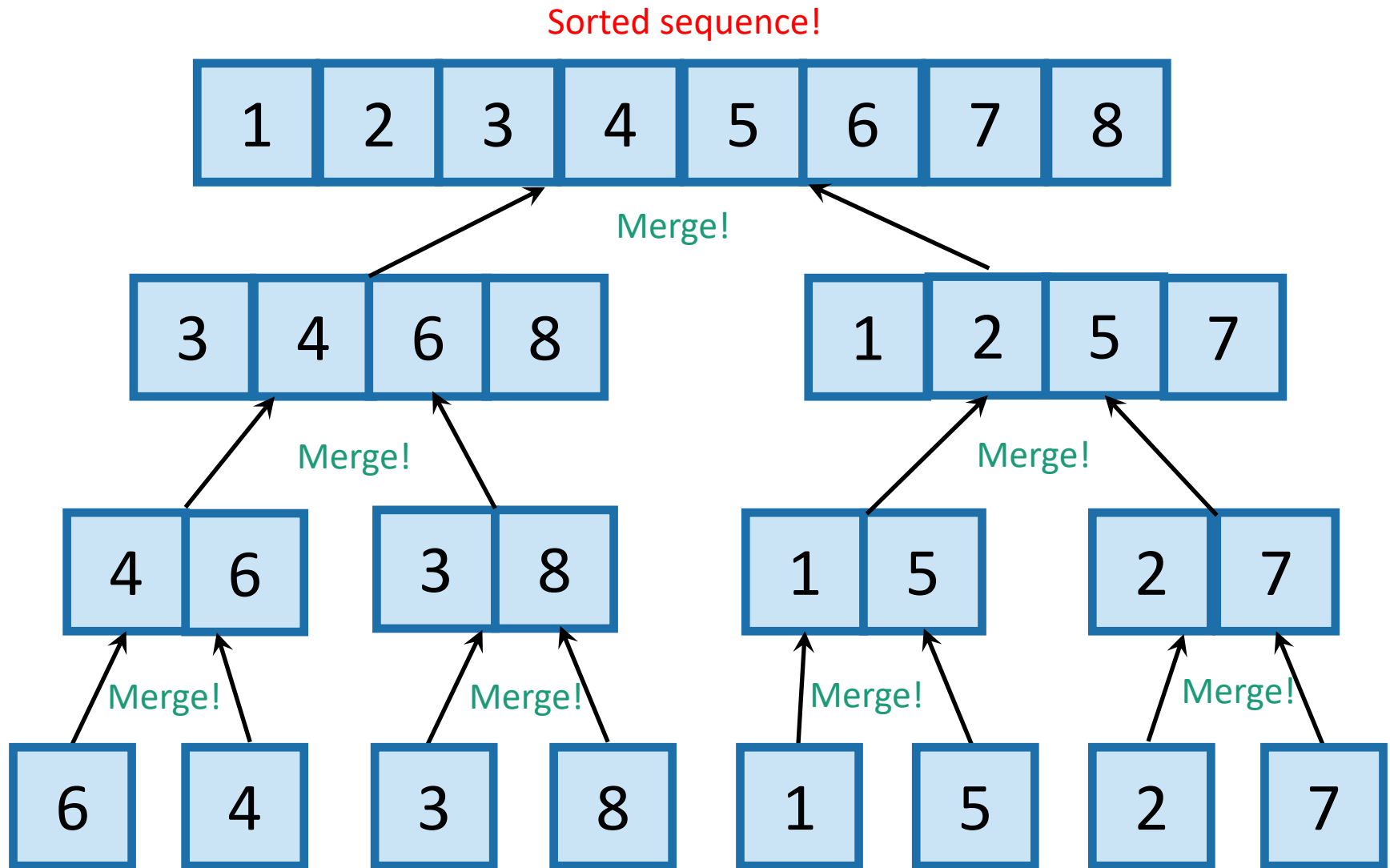
What actually happens?

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!



A bunch of sorted lists of length 1 (in the order of the original sequence).

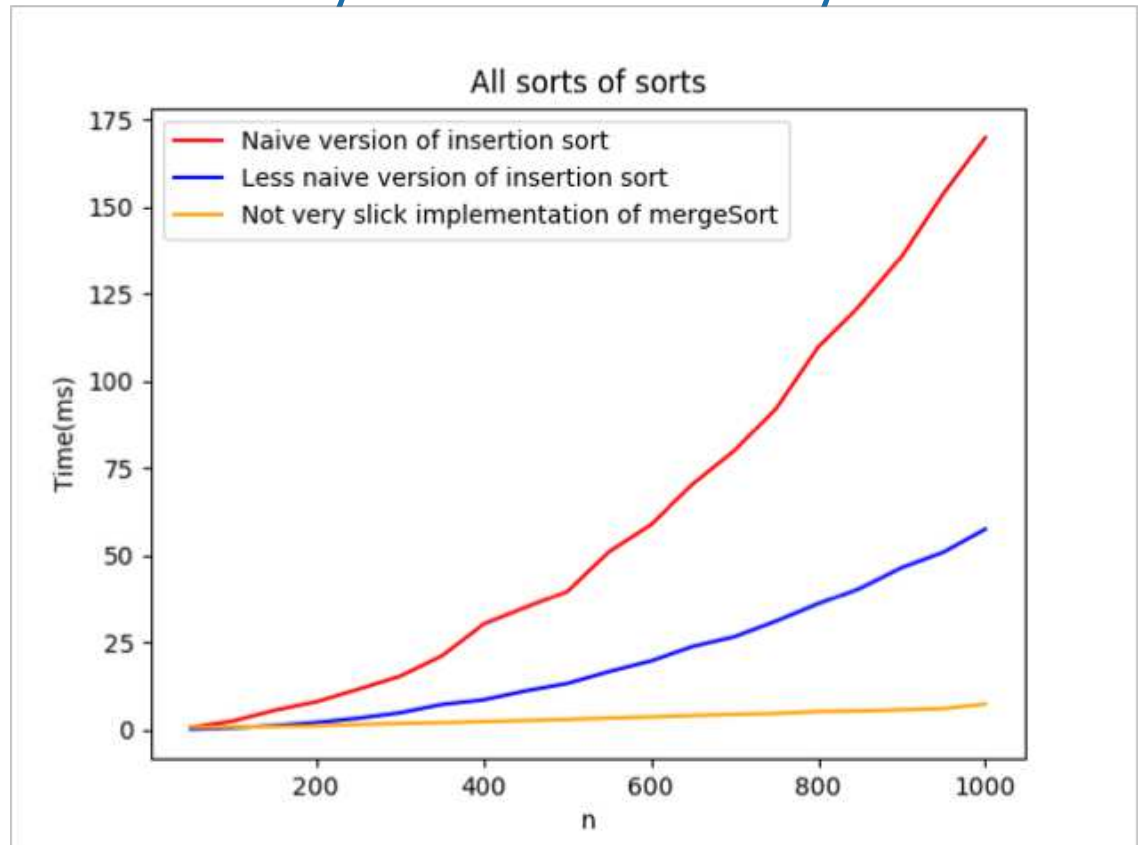
Two questions

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

IPython notebook says...

Empirically:

1. Seems to work.
2. Seems fast.



It works

Assume that n is a power of 2
for convenience.

- **Inductive hypothesis:**

“In every the recursive call on an array of length at most i , MERGESORT returns a sorted array.”

- **Base case ($i=1$):** a 1-element array is always sorted.
- **Inductive step:** Need to show:
If L and R are sorted, then $MERGE(L,R)$ is sorted.
- **Conclusion:** In the top recursive call, MERGESORT returns a sorted array.

- **MERGESORT(A):**
 - $n = \text{length}(A)$
 - **if** $n \leq 1$:
 - **return** A
 - $L = \text{MERGESORT}(A[1 : n/2])$
 - $R = \text{MERGESORT}(A[n/2+1 : n])$
 - **return** $\text{MERGE}(L,R)$



Fill in the inductive step! (Either do it yourself or read it in CLRS Section 2.3.1!)

Assume that n is a power of 2
for convenience.

It's fast

CLAIM:

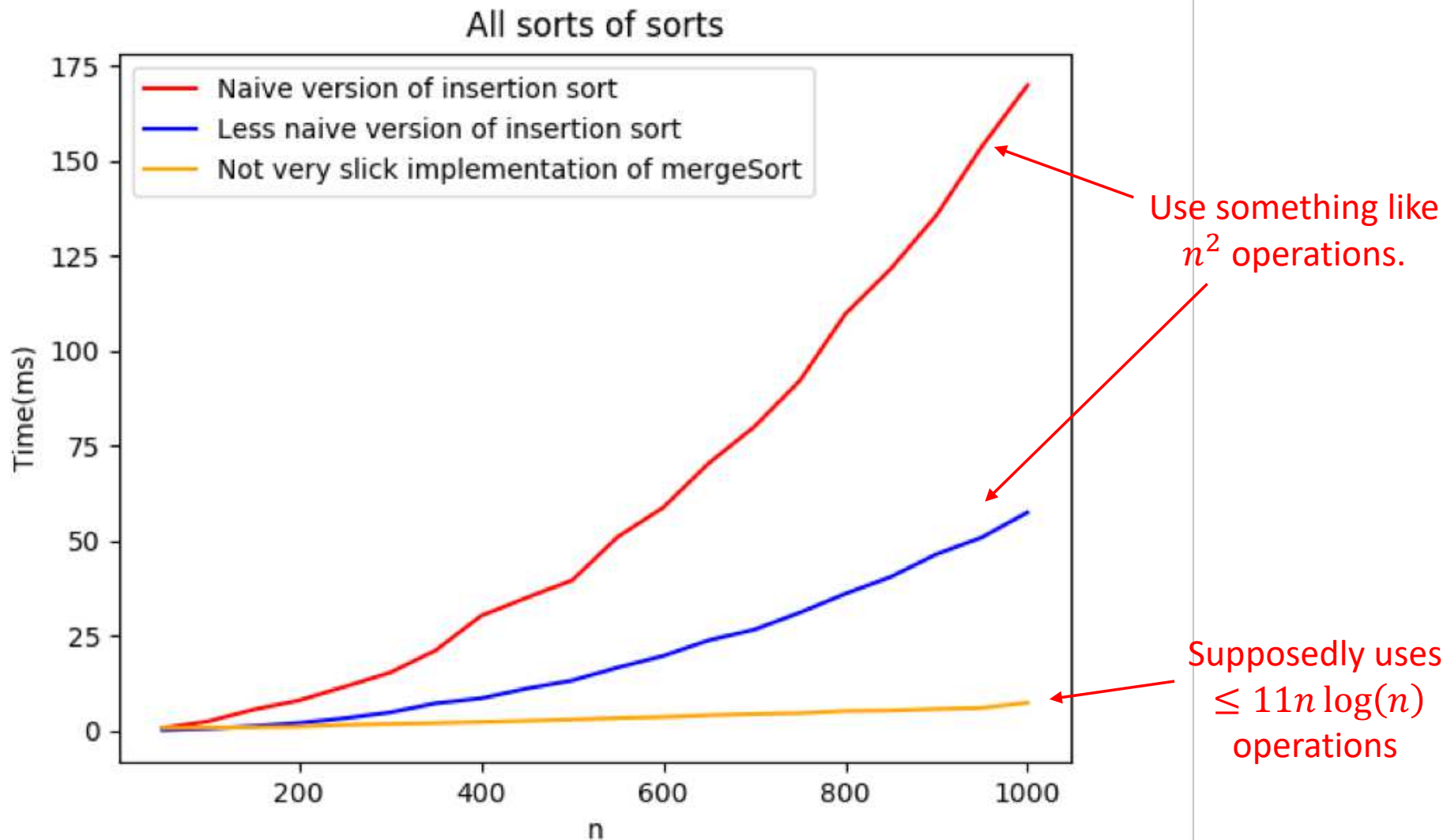
MergeSort requires at most $11n (\log(n) + 1)$
operations to sort n numbers.

- Proof coming soon.
- But first, how does this compare to InsertionSort?
 - Recall InsertionSort used on the order of n^2 operations.

What exactly is an “operation” here?
We’re leaving that vague on purpose.
Also the number 11 is sort of made-up.



$n \log(n)$ vs. n^2 ? (Empirically)



$n \log(n)$ vs. n^2 ? (Analytically)

All logarithms in this course are base 2

Aside:



Quick log refresher

- **Def:** $\log(n)$ is the number so that $2^{\log(n)} = n$.
- **Intuition:** $\log(n)$ is how many times you need to divide n by 2 in order to get down to 1.

$$32, 16, 8, 4, 2, 1 \Rightarrow \log(32) = 5$$

Halve 5 times

$$64, 32, 16, 8, 4, 2, 1 \Rightarrow \log(64) = 6$$

Halve 6 times

$$\log(128) = 7$$

$$\log(256) = 8$$

$$\log(512) = 9$$

....

$$\log(\text{\# particles in the universe}) < 280$$

- $\log(n)$ grows very slowly!

$n \log(n)$ vs. n^2 ? (Analytically)

- $\log(n)$ “grows much more slowly” than n
- $n \log(n)$ “grows much more slowly” than n^2

We will be more precise
about this in part II of the
lecture!



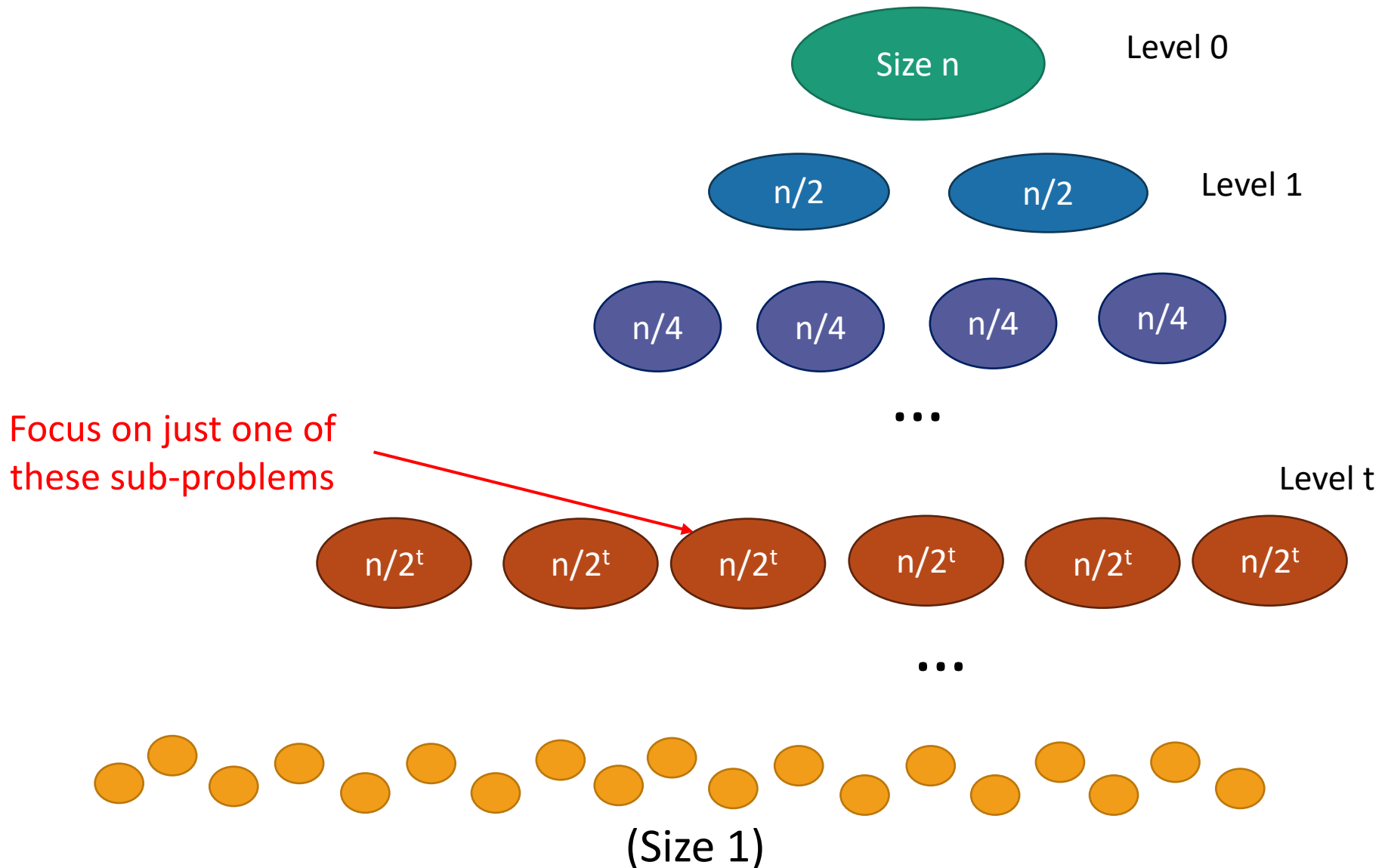
Assume that n is a power of 2
for convenience.

Now let's prove the claim

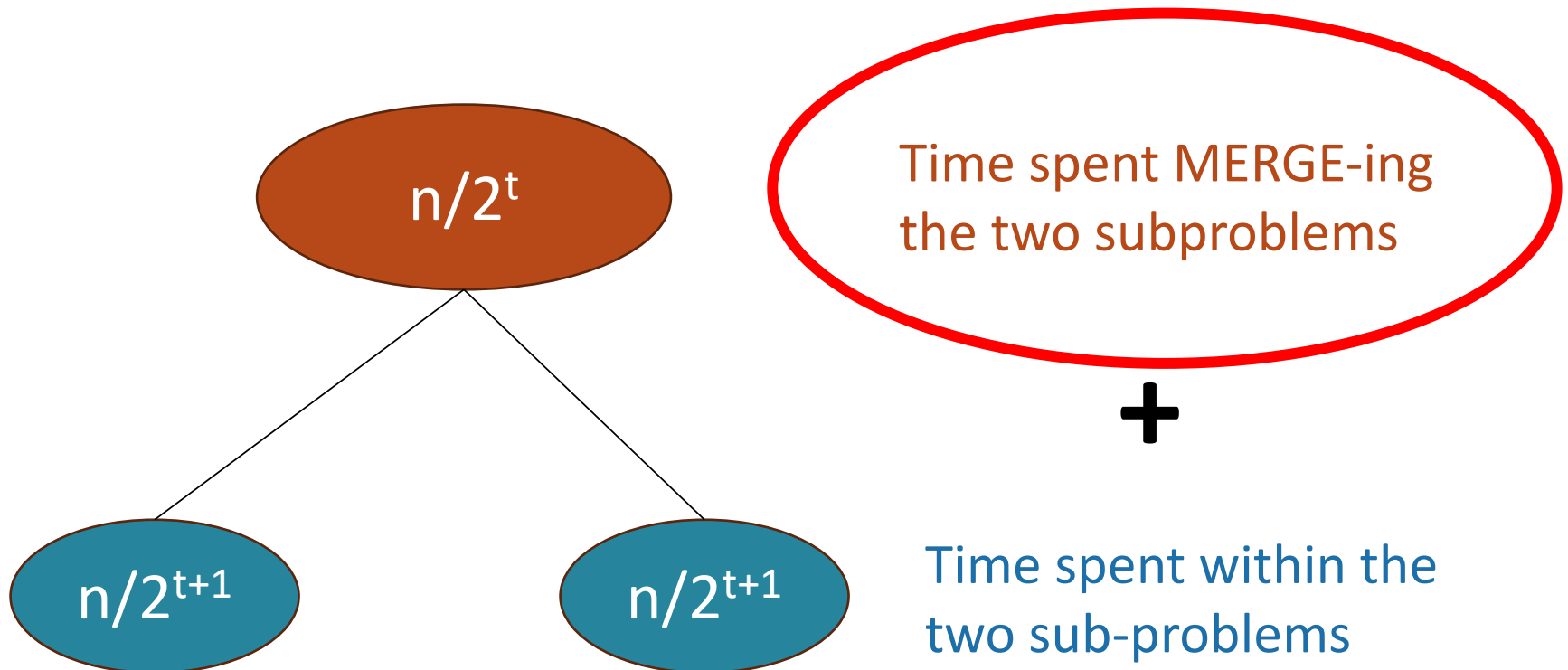
CLAIM:

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

Let's prove the claim

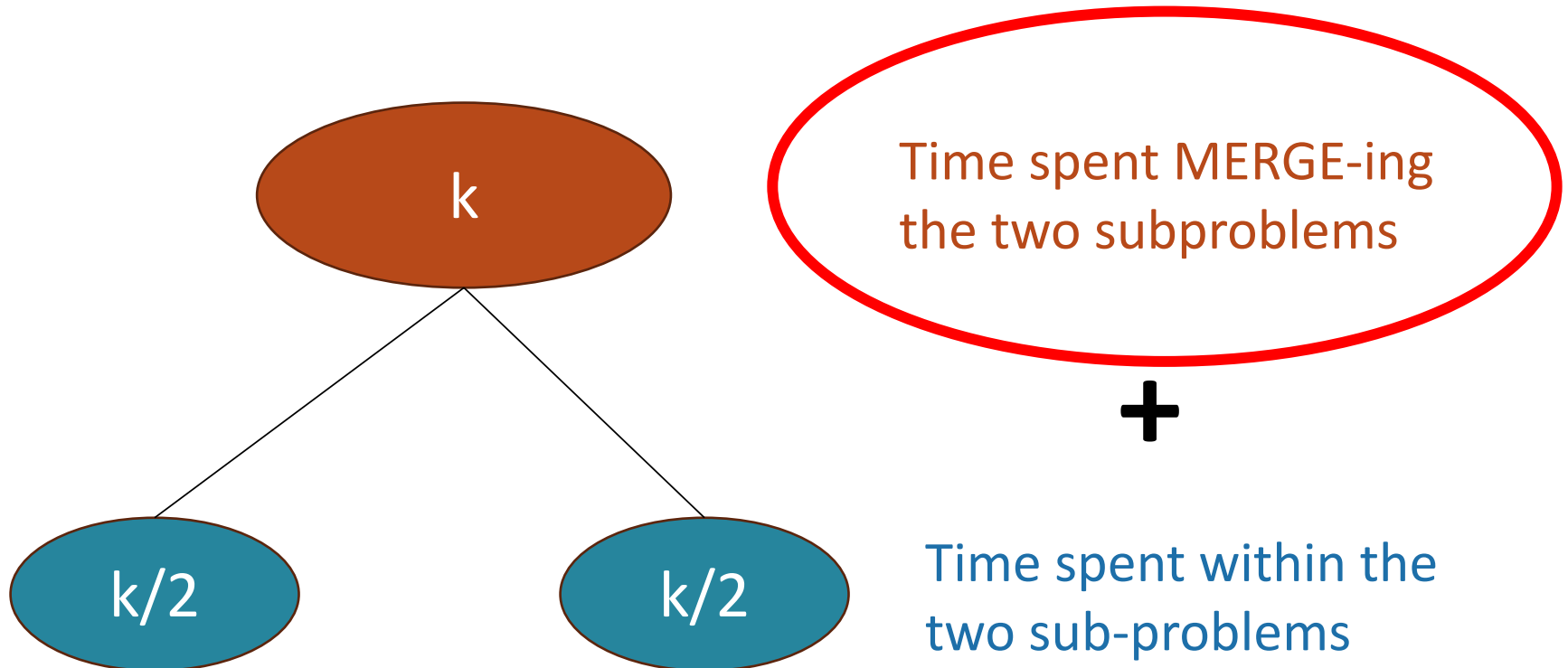


How much work in this sub-problem?

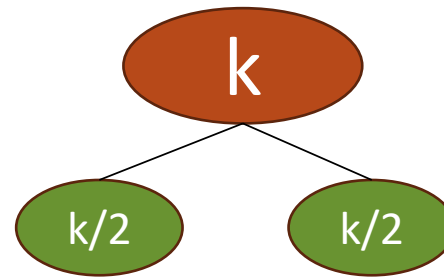


How much work in this sub-problem?

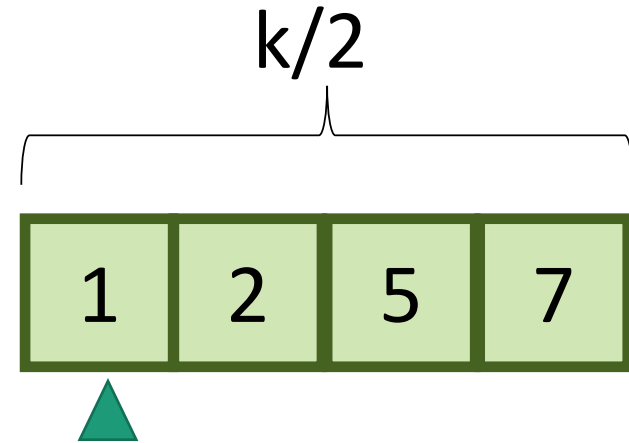
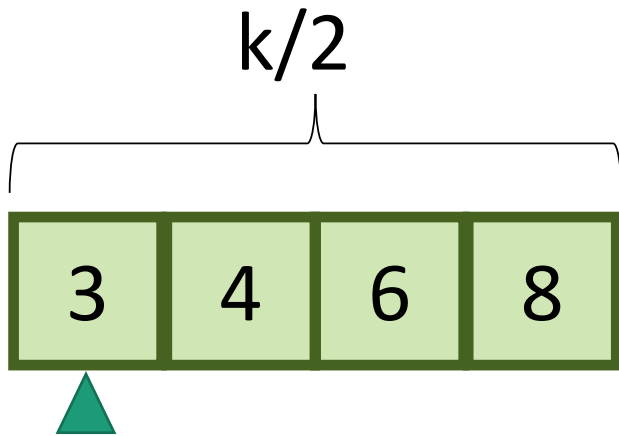
Let $k=n/2^t$...



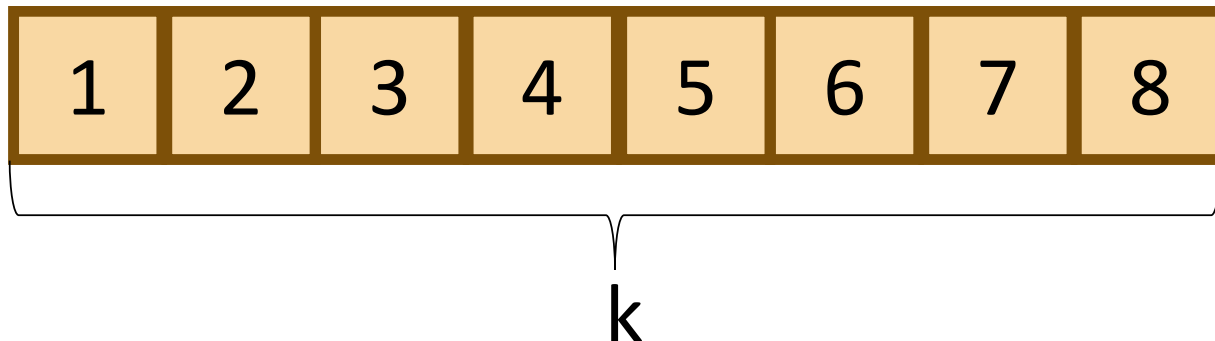
How long does it take to MERGE?



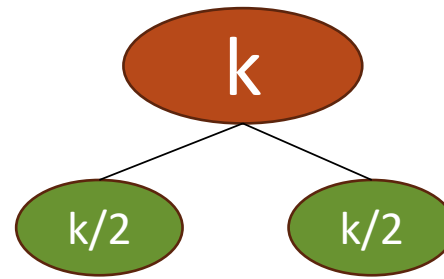
Code for the **MERGE** step is given in the Lecture2 notebook.



MERGE!

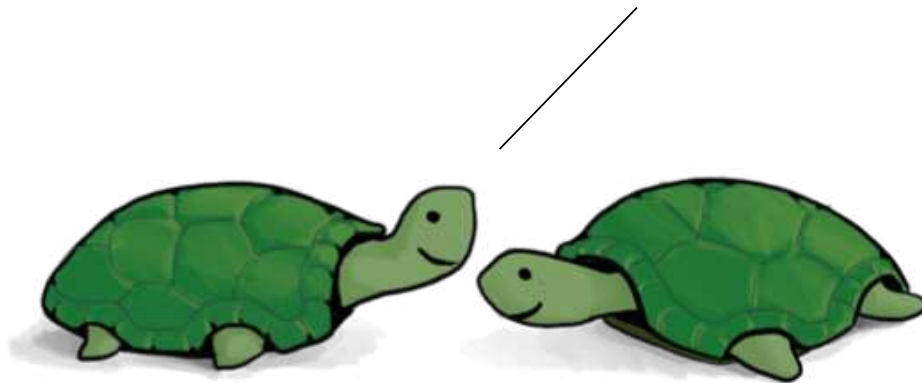


How long does it take to MERGE?



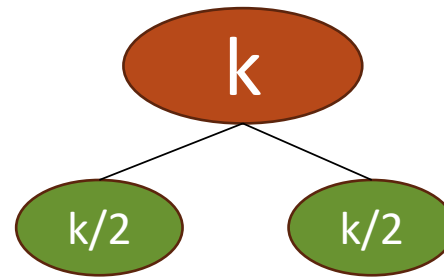
Code for the **MERGE** step is given in the Lecture2 notebook.

About how many operations does it take to run MERGE on two lists of size $k/2$?



Think-Pair-Share Terrapins

How long does it take to MERGE?



Code for the **MERGE** step is given in the Lecture2 notebook.

- 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...

Let's say no more than **11k** operations.

There's a hidden slide which sort of explains this number "11," but it's a bit silly and we'll see in a little by why it doesn't matter.

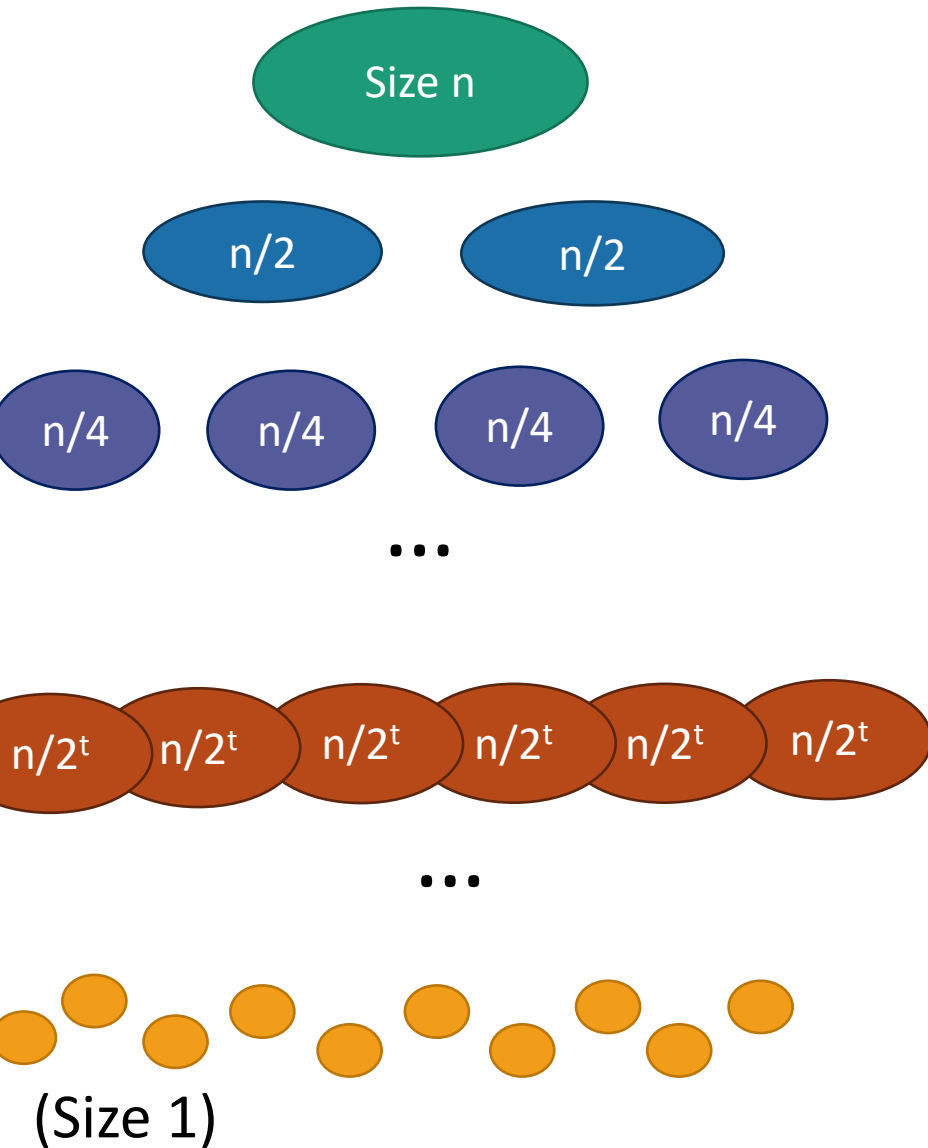


Plucky the
Pedantic Penguin

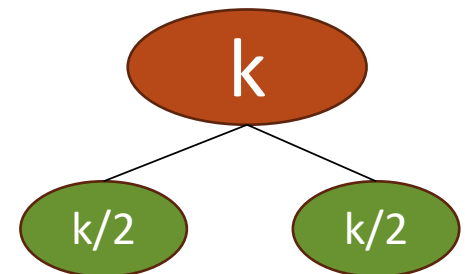


Lucky the
lackadaisical lemur

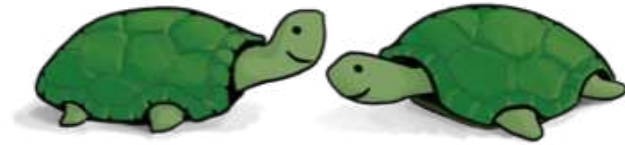
Recursion tree



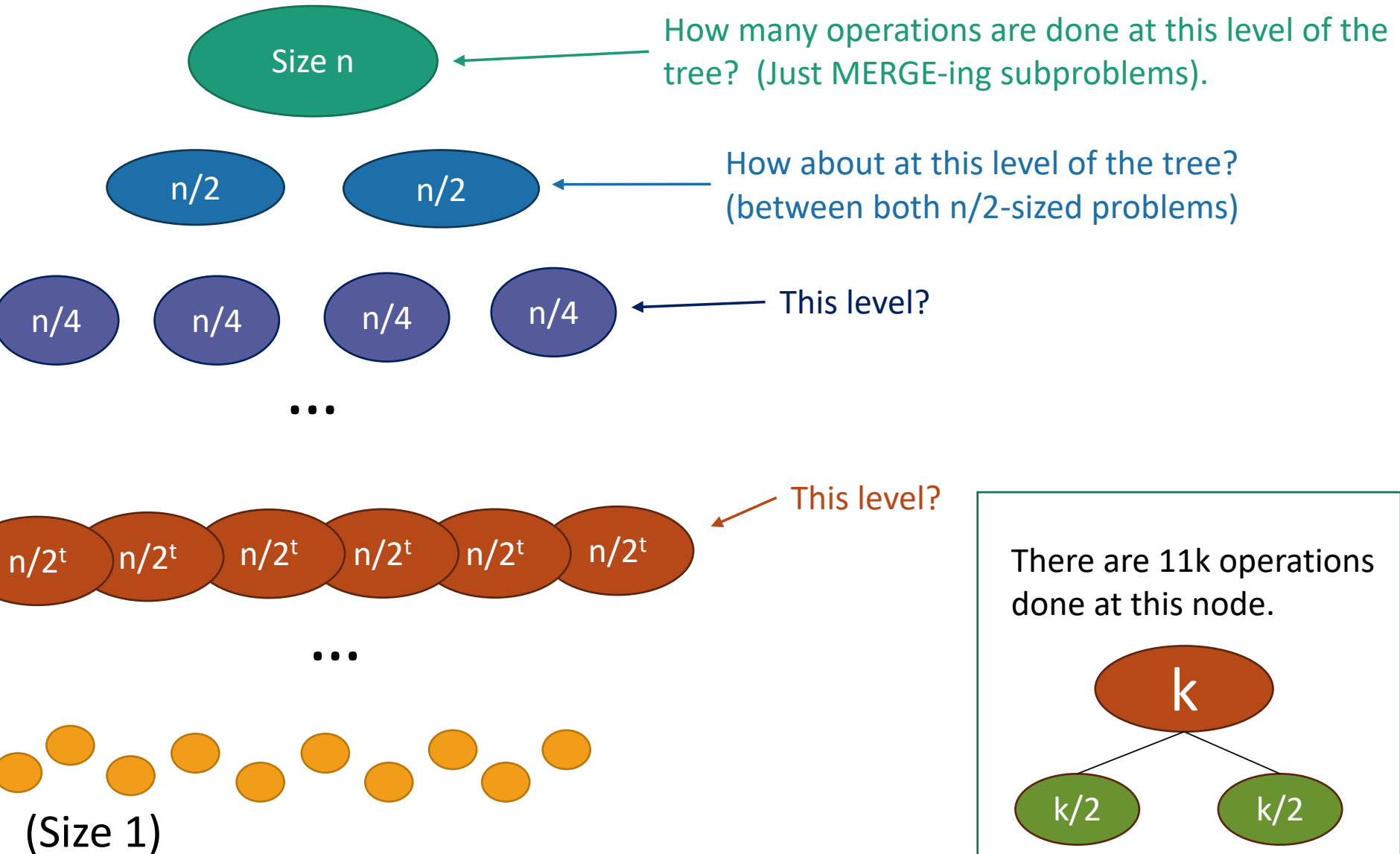
There are 11k operations done at this node.



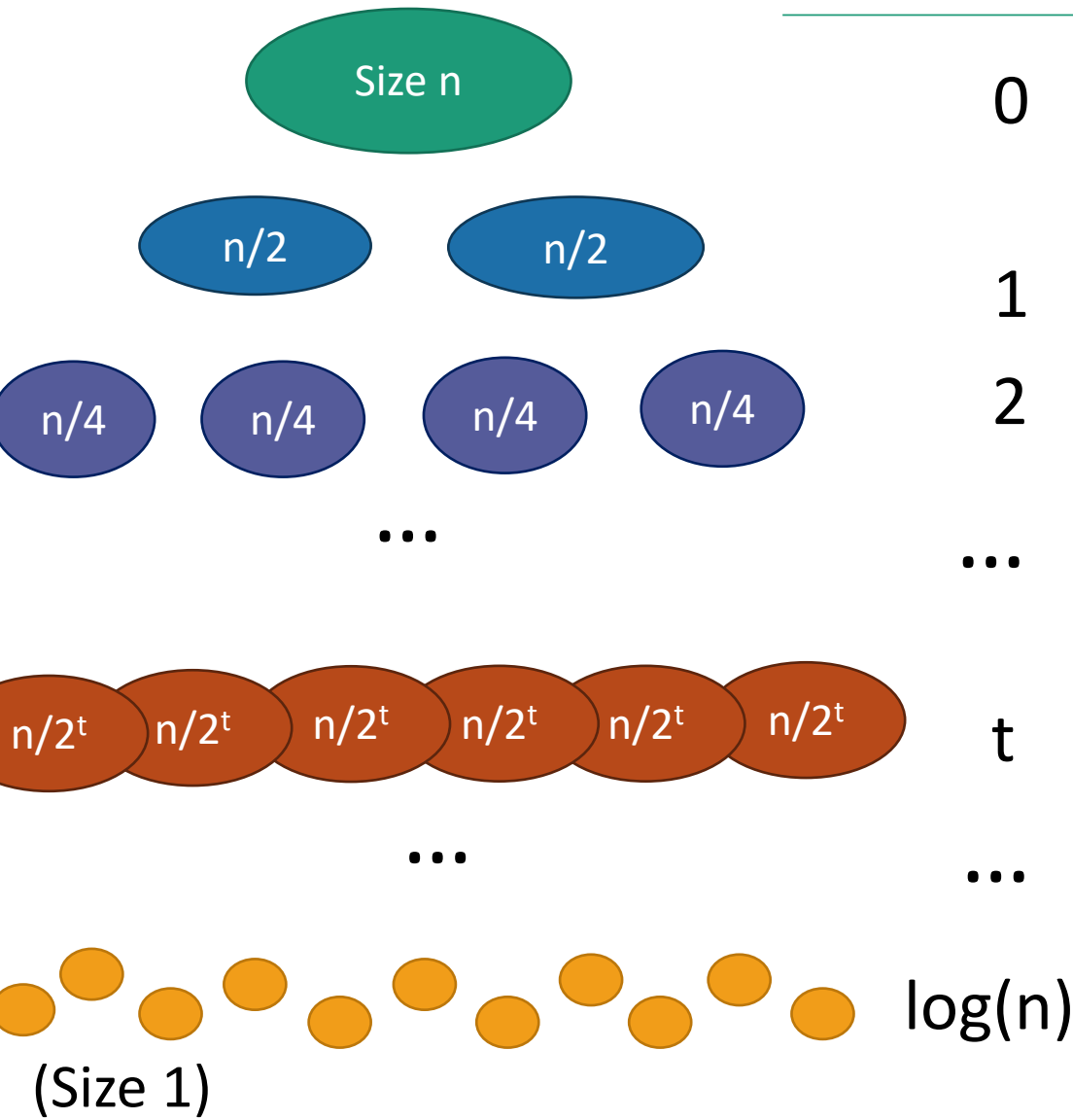
Recursion tree



Think, Pair,
Share!



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	$2n \leq 11n$

Explanation for this table done on the board!

Note: At the lowest level we only have two operations per problem, to get the length of the array and compare it to 1.

Total runtime...

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

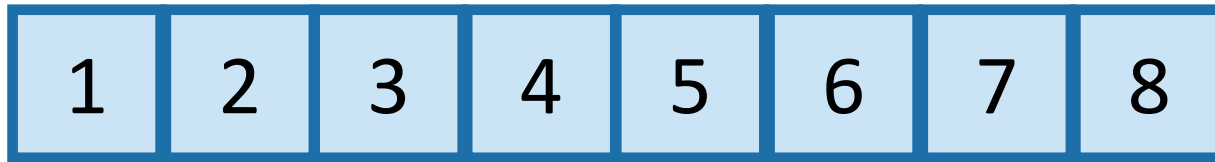
That was the claim!

What have we learned?

- MergeSort correctly sorts a list of n integers in at most $11n(\log(n) + 1)$ operations.

A few reasons to be grumpy

- Sorting



should take zero steps...

- What's with this 11k bound?
 - You (Mary/Lucky) 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)



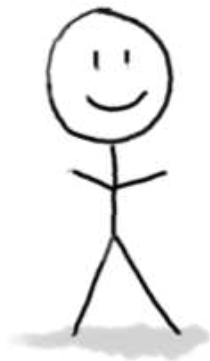
- Part II: How do we measure the runtime of an algorithm?

- Worst-case analysis
- Asymptotic Analysis

Worst-case analysis

Sorting a sorted list
should be fast!!

The “running time” for an algorithm is its running time on the **worst possible input**.



Algorithm
designer

Here is my algorithm!

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

HERE IS AN INPUT!
(WHICH I DESIGNED
TO BE TERRIBLE FOR
YOUR ALGORITHM!)



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

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).

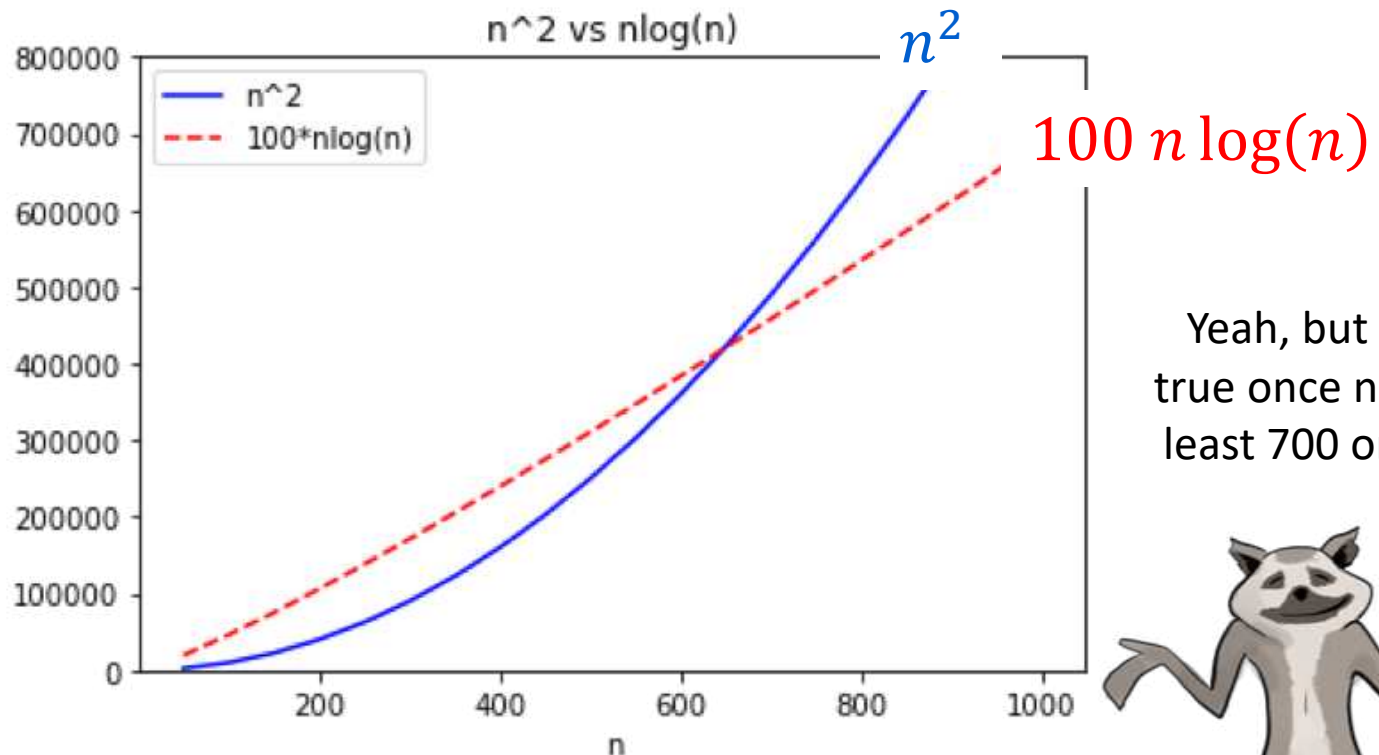
Informally....

(Only pay attention to the largest function of n that appears.)

Number of operations	Asymptotic Running Time
$\frac{1}{10} \cdot n^2 + 100$	$O(n^2)$
$0.063 \cdot n^2 - .5n + 12.7$	$O(n^2)$
$100 \cdot n^{1.5} - 10^{10000} \sqrt{n}$	$O(n^{1.5})$
$11 \cdot n \log(n) - 1$	$O(n \log(n))$

We say this algorithm is “asymptotically faster” than the others.

So $100 n \log(n)$ operations is
“better” than n^2 operations?



But when
 $n=200$, that's
not true at all!



Yeah, but it's
true once n is at
least 700 or so.



Asymptotic Analysis

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).

$1000000000n$
is “better” than n^2 ?!?!

Without making Plucky grumpy!

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

$O(\dots)$ means an upper bound

- Let $T(n)$, $g(n)$ be functions of positive integers.
 - Think of $T(n)$ as a runtime: positive and increasing in n .
- We say “ $T(n)$ is $O(g(n))$ ” if $T(n)$ grows no faster than $g(n)$ as n gets large.
- Formally,

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

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq T(n) \leq c \cdot g(n)$$

Example

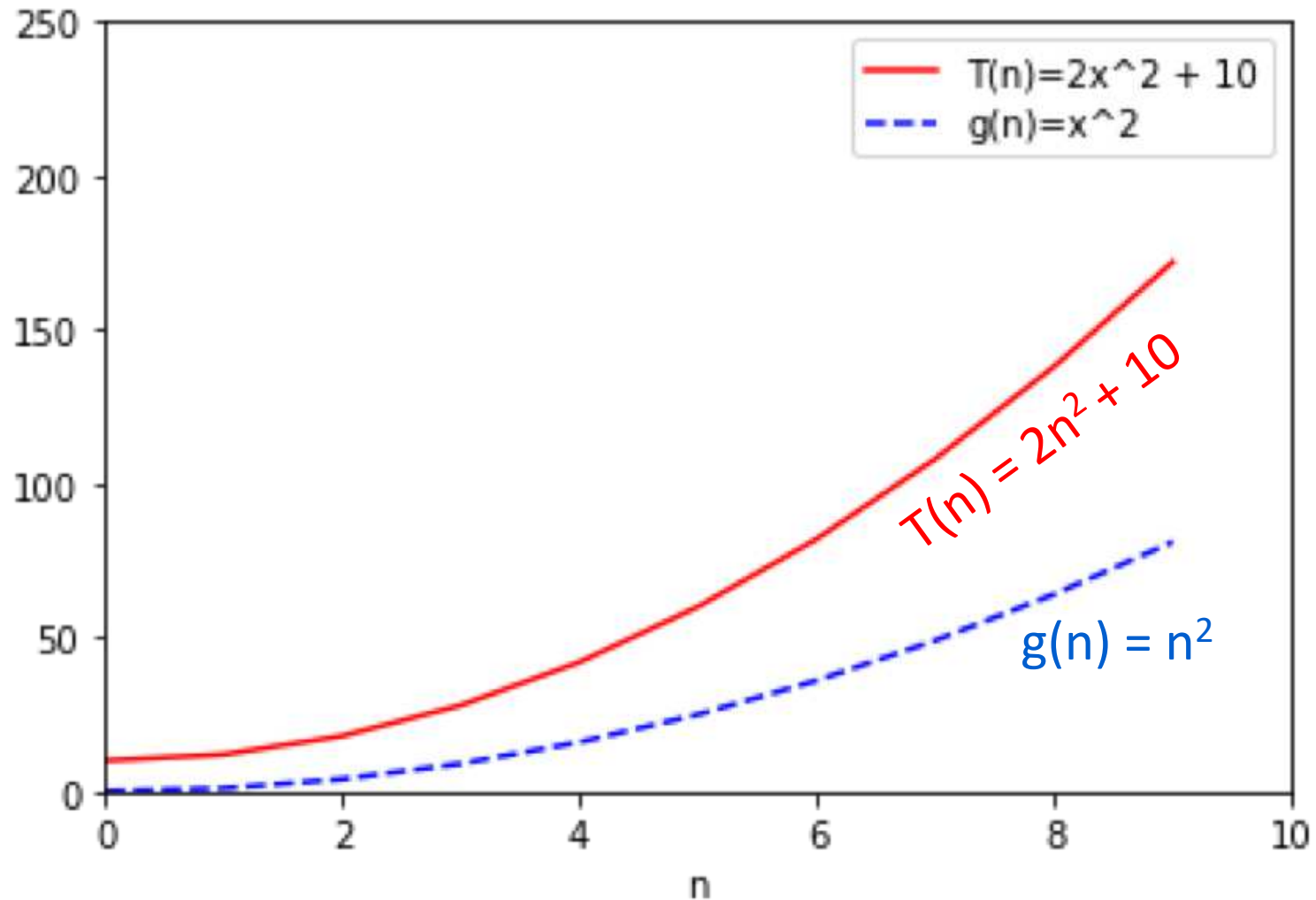
$$2n^2 + 10 = O(n^2)$$

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

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq T(n) \leq c \cdot g(n)$$



Example

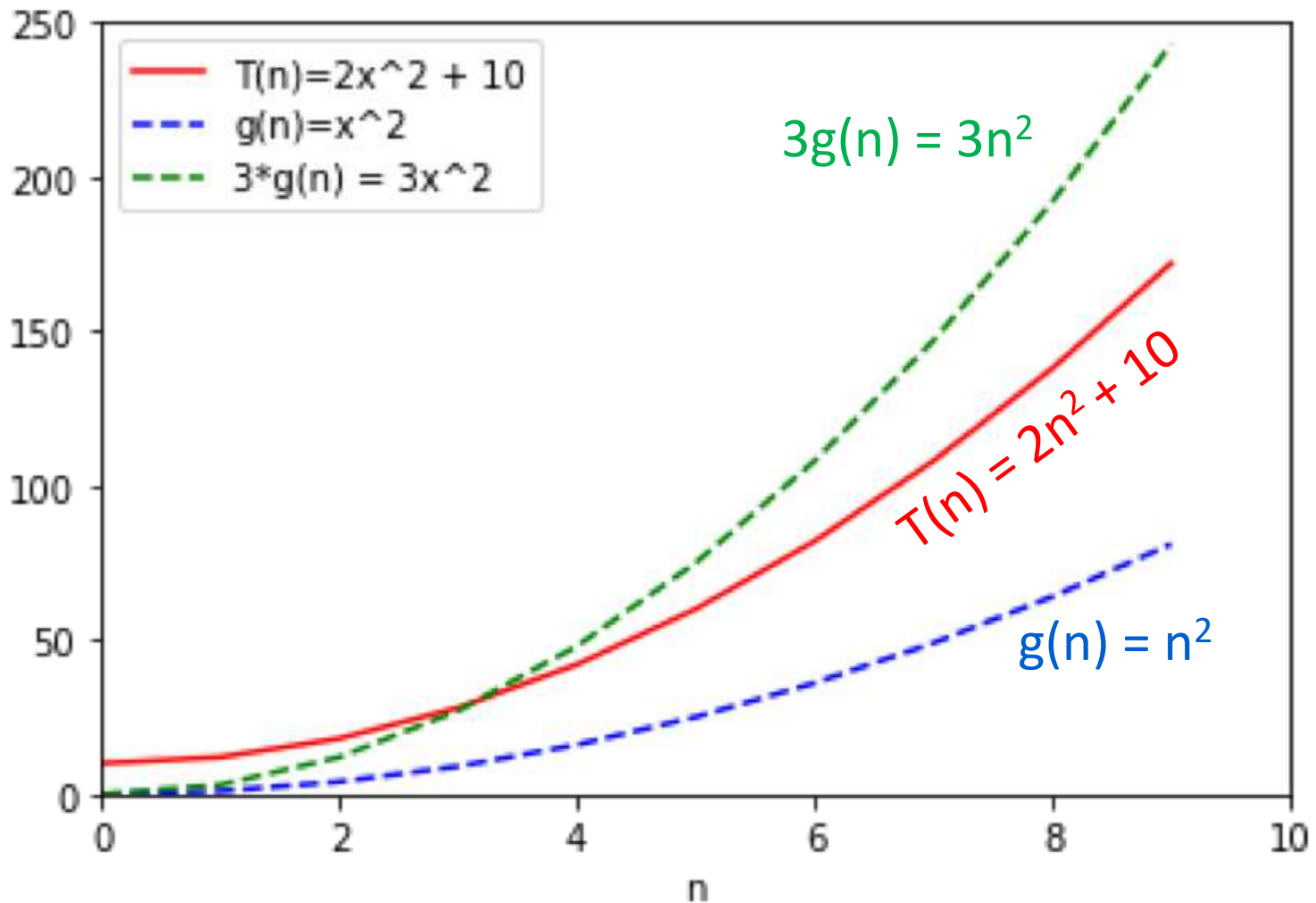
$$2n^2 + 10 = O(n^2)$$

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

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq T(n) \leq c \cdot g(n)$$



Example

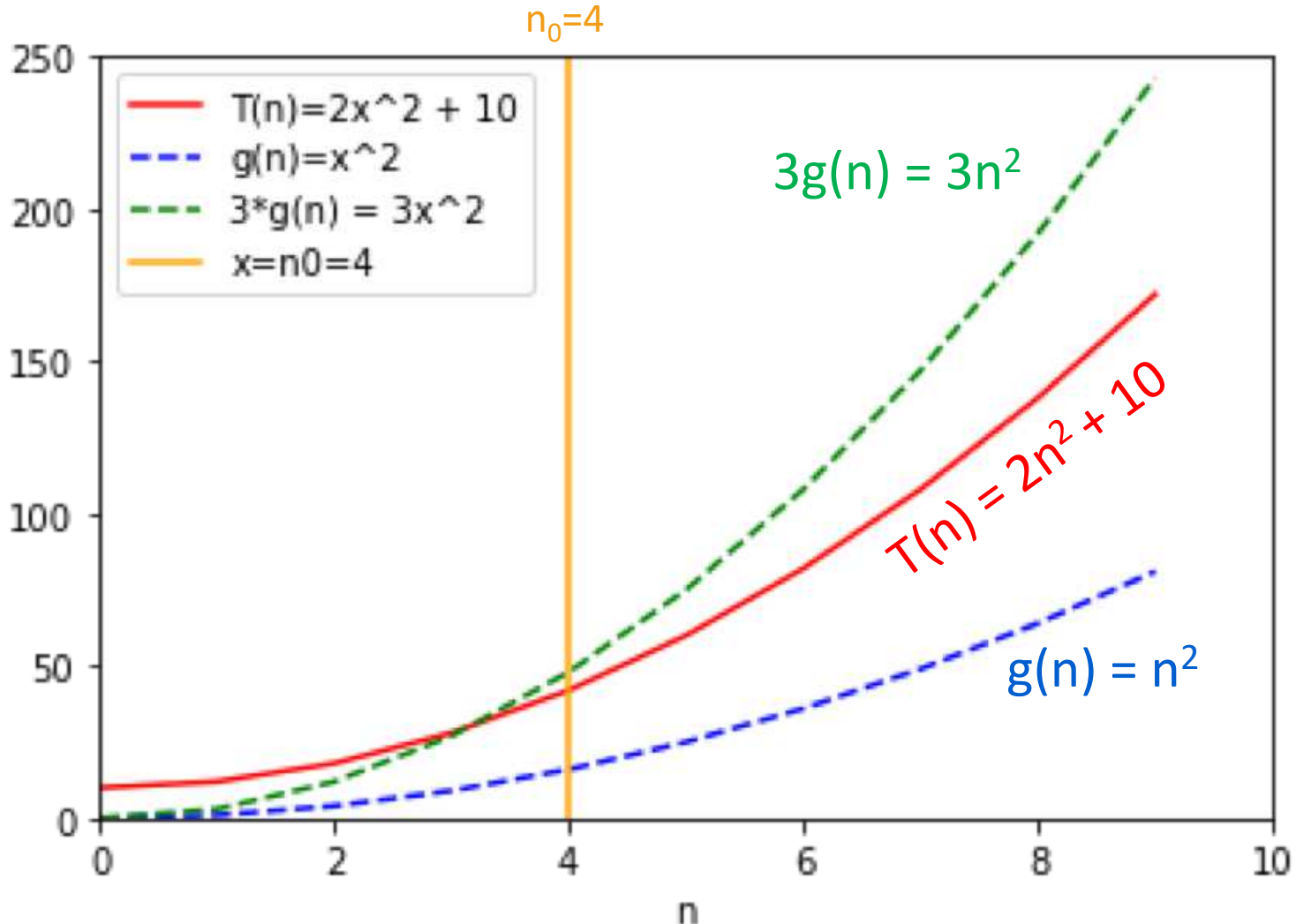
$$2n^2 + 10 = O(n^2)$$

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

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq T(n) \leq c \cdot g(n)$$



Example

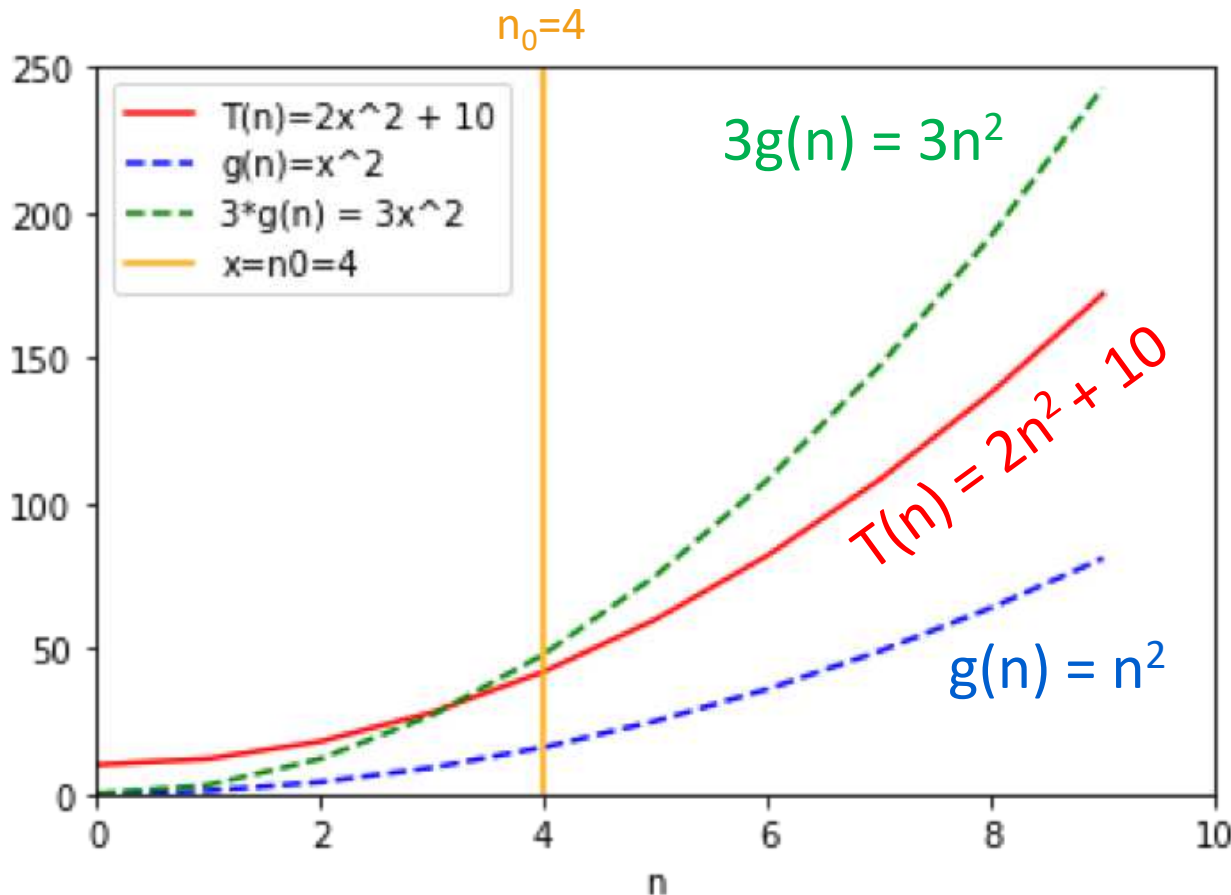
$$2n^2 + 10 = O(n^2)$$

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

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq T(n) \leq c \cdot g(n)$$



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

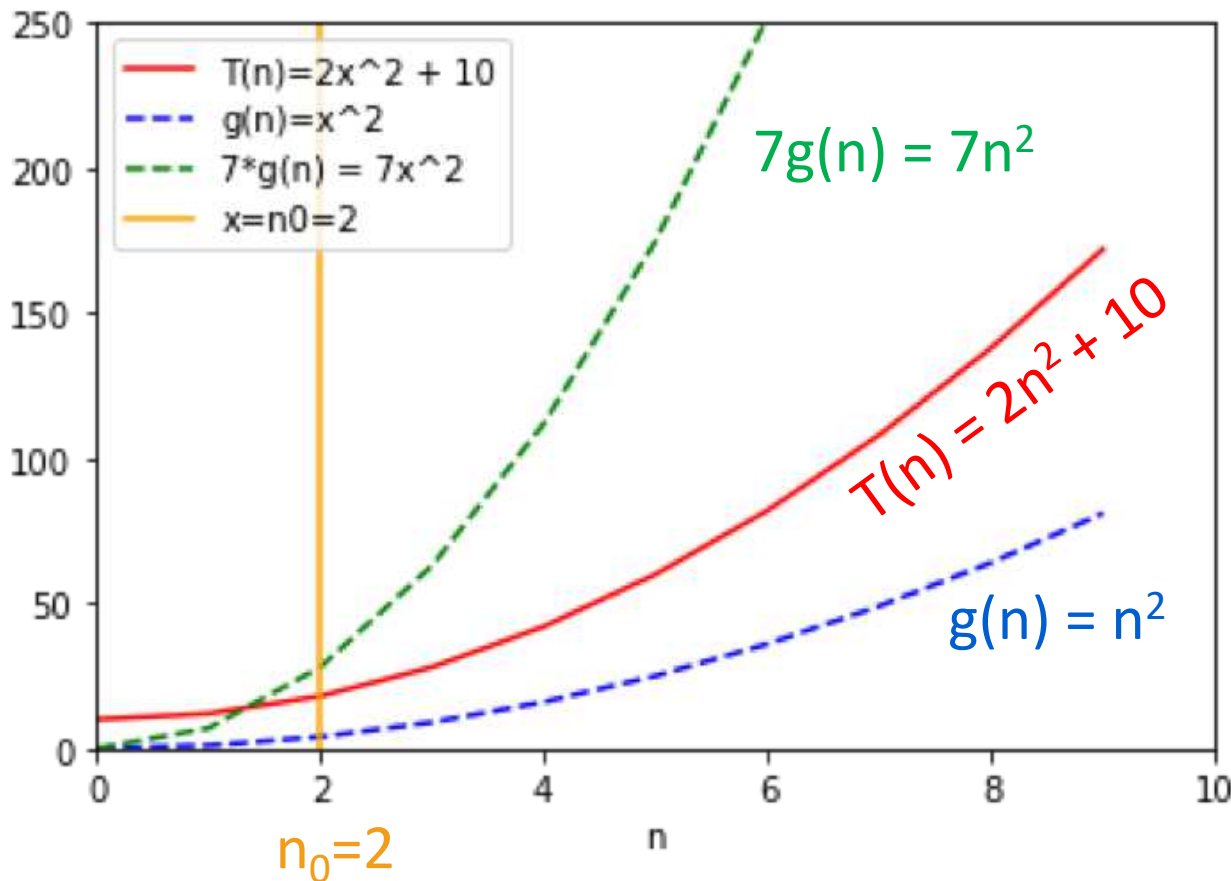
$2n^2 + 10 = O(n^2)$

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

\Leftrightarrow

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq T(n) \leq c \cdot g(n)$$



Formally:

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

$$\forall n \geq 2,$$

$$0 \leq 2n^2 + 10 \leq 7 \cdot n^2$$

There is not a
“correct” choice
of c and n_0

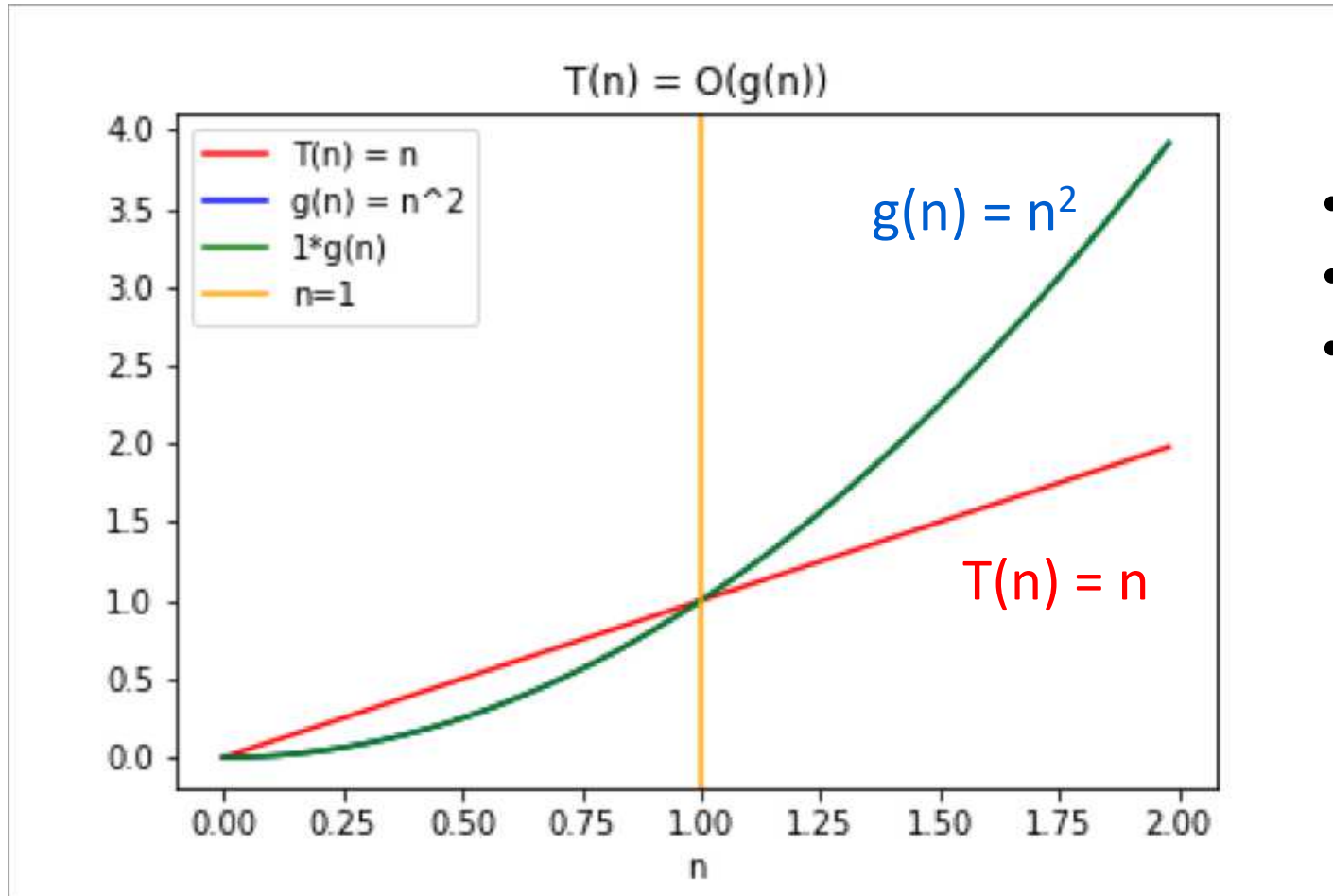
Another example:
 $n = O(n^2)$

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

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq T(n) \leq c \cdot g(n)$$



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

$$\forall n \geq 1,$$

$$0 \leq n \leq n^2$$

$\Omega(\dots)$ means a lower bound

- We say “ $T(n)$ is $\Omega(g(n))$ ” if $T(n)$ grows at least as fast as $g(n)$ as n gets large.
- Formally,

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

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq c \cdot g(n) \leq T(n)$$

Switched these!!

Example

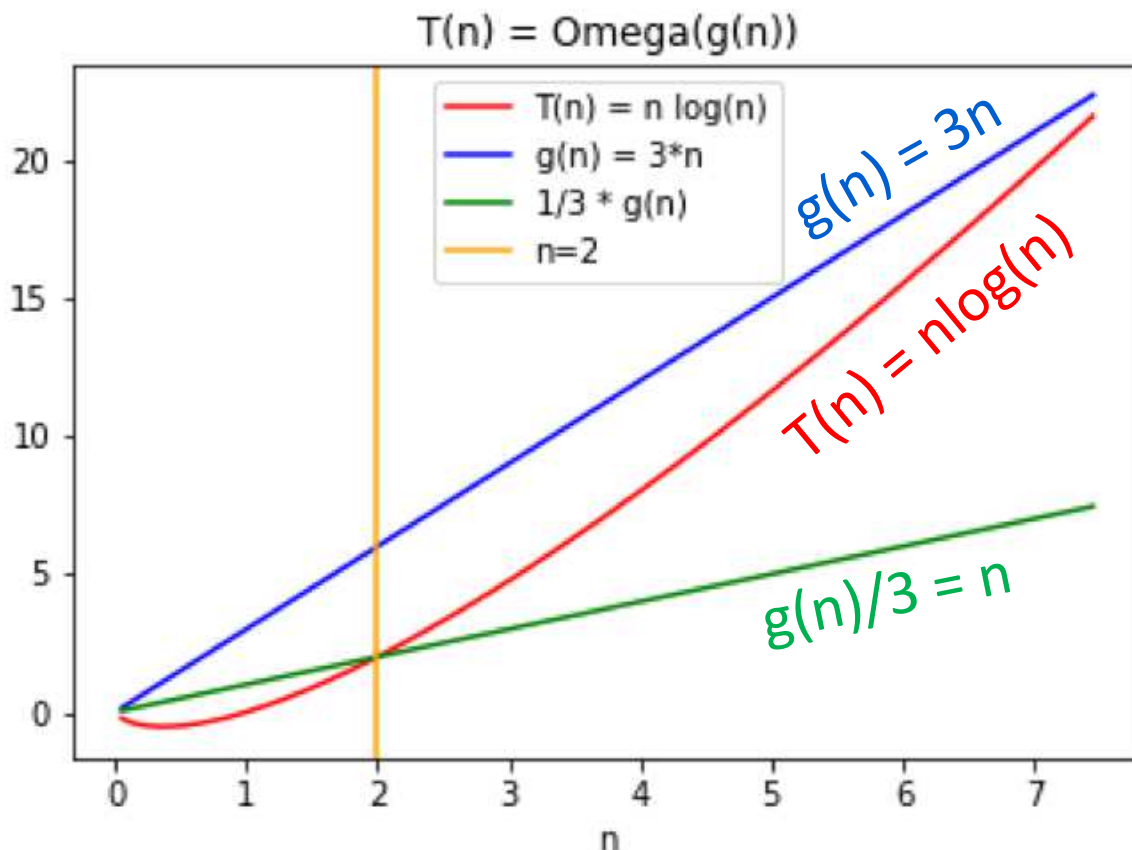
$n \log_2(n) = \Omega(3n)$

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

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq c \cdot g(n) \leq T(n)$$



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

$$\forall n \geq 2,$$

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

$\Theta(\dots)$ means both!

- We say “ $T(n)$ is $\Theta(g(n))$ ” iff both:

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

and

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

Example: polynomials

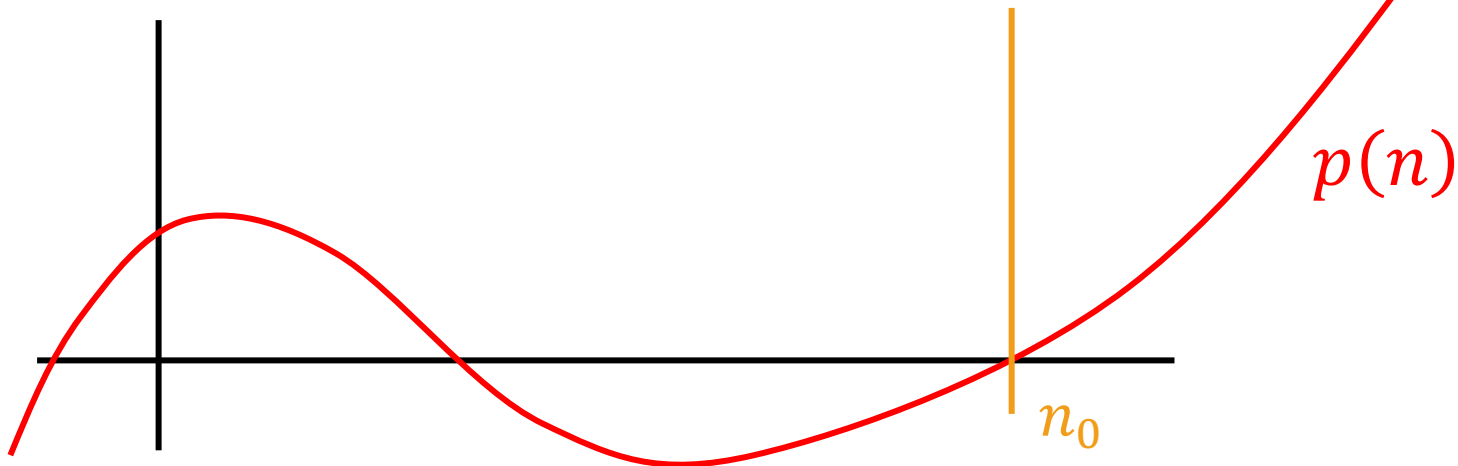
- Suppose the $p(n)$ is a polynomial of degree k :

$$p(n) = a_0 + a_1n + a_2n^2 + \cdots + a_kn^k \text{ where } a_k > 0.$$

- Then $p(n) = O(n^k)$

- Proof:

- Choose $n_0 \geq 1$ so that $p(n) \geq 0$ for all $n \geq n_0$.
- Choose $c = |a_0| + |a_1| + \cdots + |a_k|$



Example: polynomials

- Suppose the $p(n)$ is a polynomial of degree k :

$$p(n) = a_0 + a_1n + a_2n^2 + \cdots + a_kn^k \text{ where } a_k > 0.$$

- Then $p(n) = O(n^k)$

- Proof:

- Choose $n_0 \geq 1$ so that $p(n) \geq 0$ for all $n \geq n_0$.

- Choose $c = |a_0| + |a_1| + \cdots + |a_k|$

- Then for all $n \geq n_0$:

$$\begin{aligned} 0 \leq p(n) = |p(n)| &\leq |a_0| + |a_1|n + \cdots + |a_k|n^k \\ &\leq |a_0|n^k + |a_1|n^k + \cdots + |a_k|n^k \\ &= c \cdot n^k \end{aligned}$$

Triangle
inequality!

Definition of c

Because $n \leq n^k$
for $n \geq n_0 \geq 1$.

Example: more polynomials

- For any $k \geq 1$, n^k is **NOT** $O(n^{k-1})$.
- Proof:
 - Suppose that it were. Then there is some c, n_0 so that
$$n^k \leq c \cdot n^{k-1} \text{ for all } n \geq n_0$$
 - Aka, $n \leq c$ for all $n \geq n_0$
 - But that's not true! What about $n = n_0 + c + 1$!?
 - We have a contradiction! It *can't* be that $n^k = O(n^{k-1})$.

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 be lying to you by deriving a contradiction.

Yet more examples

- $n^3 + 3n = O(n^3 - n^2)$
- $n^3 + 3n = \Omega(n^3 - n^2)$
- $n^3 + 3n = \Theta(n^3 - n^2)$

- 3^n is **NOT** $O(2^n)$
- $\log(n) = \Omega(\ln(n))$
- $\log(n) = \Theta(2^{\log \log(n)})$

remember that $\log = \log_2$ in this class.

Work through these
on your own!



Siggi the Studios Stork

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

Recap: Asymptotic Notation

This is my
happy face!



- 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 always be careful not to abuse it.
- In the course, (almost) every algorithm we see will be actually practical, without needing to take $n \geq n_0 = 2^{100000000}$.

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.

Before next time

- Pre-Lecture Exercise:
 - A few recurrence relations (see website)