Lecture 2

Asymptotic Notation,

Worst-Case Analysis, and MergeSort

Announcements

- Please (continue to) send OAE letters to cs161-win2223-staff@lists.stanford.edu

161A (ACE)

The goal of ACE is to increase confidence and content knowledge through interactive small group sessions and additional academic resources. In CS161 ACE, you can expect an additional weekly section, ACE-specific office hours, and an extra community of people learning together and supporting each other.

- Fri 9:30am - 11:20am in 100-101K (taught by Lauren Saue-Fletcher)
- Short application (link on course website too): https://docs.google.com/forms/d/e/1FAIpQLSfz4xHbWbH_LZmn8PlQ9XB24OLynzTdmC5YvQVC6s04R0C6dA/viewform
- A fair amount of space for more students!
- Final application deadline: Friday, January 13th at 5:00pm (sooner is better – there is a quick meeting this Friday morning)
- Questions? Send an email to laurensauefletcher@stanford.edu

Homework!

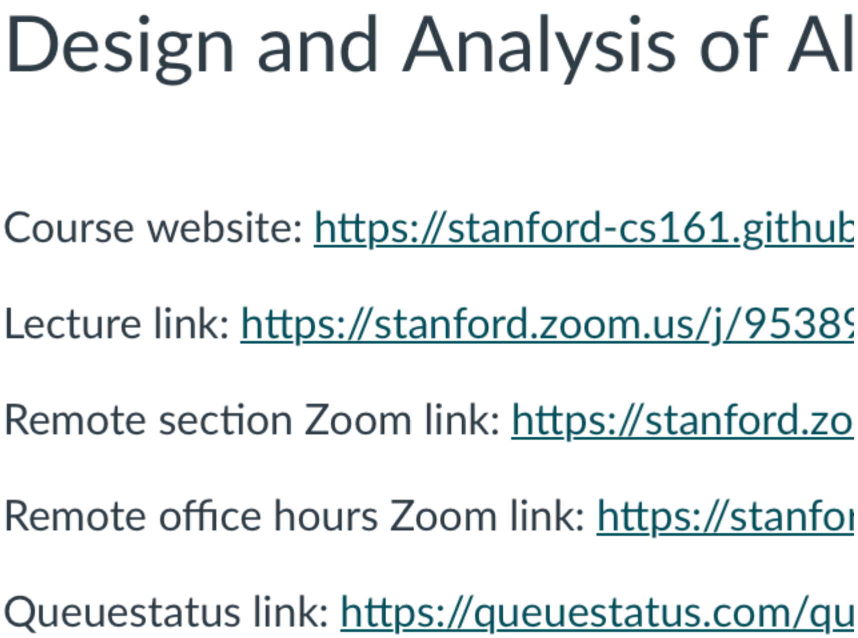
- HW1 will be released **today** (Wednesday).
- It is due the next **Wednesday, 11:59pm** (in one week), on Gradescope. As a reminder, HW1, HW2, and HW3 are solo submissions only.
- Homework comes in two parts:
  - Exercises:
    - More straightforward.
    - Try to do them on your own.
  - Problems:
    - Less straightforward.
    - Try them on your own first, but then collaborate!
- See the website for guidelines on homework:
  - Collaboration + late day policy (in the “Policies” tab)
  - Best practices (in the “Resources” tab)
  - Example homework (in the “Resources” tab)
  - LaTeX help (in the “Resources” tab)

Office Hours and Sections

- Office hours calendar is on the course website.
  - (under "Staff / Office Hours”)
  - Office hours start today
- Sections have been scheduled.
  - See course website
  - One will be recorded (and put on Canvas)
  - Don’t need to formally enroll in sections, just show up!

Huang basement

Links on Canvas



Design and Analysis of Al

Course website: https://stanford-cs161.github

Lecture link: https://stanford.zoom.us/j/95389

Remote section Zoom link: https://stanford.zo

Remote office hours Zoom link: https://stanfor

Queuestatus link: https://queuestatus.com/qu

End of announcements!

Last time

Philosophy

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

Technical content

- Grade-school integer multiplication
- Not-so-rigorous analysis
- Divide-and-conquer
- Karatsuba integer multiplication

will do now

Integer Multiplication

$$\begin{array}{lr} & 1234567895931413 \\ \times & 4563823520395533 \\ \hline \end{array}$$

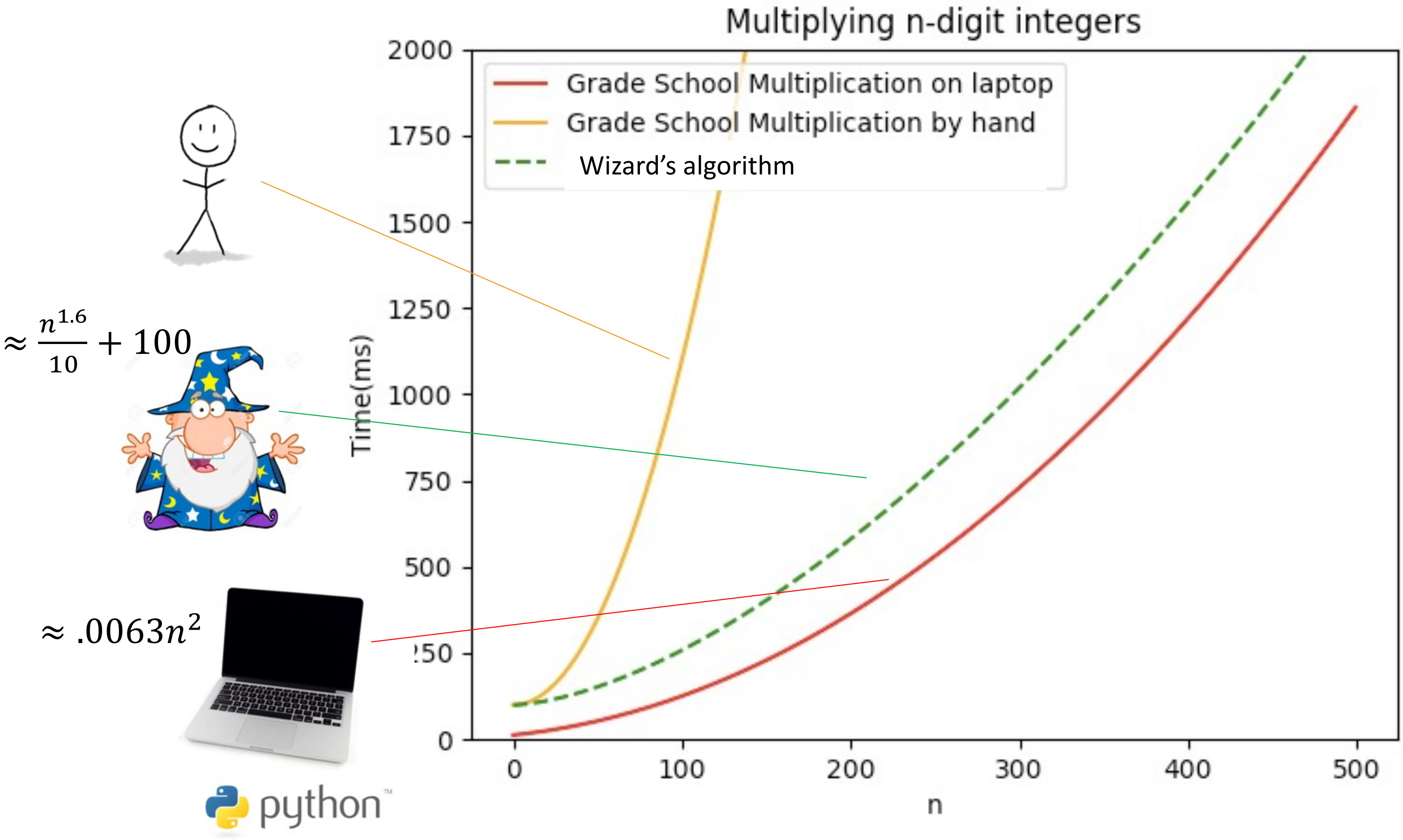
Big-Oh Notation

- We say that Grade-School Multiplication

  “runs in time $O(n^2)$”

- Formal definition later today!
- Informally, big-Oh notation tells us how the running time scales with the size of the input.

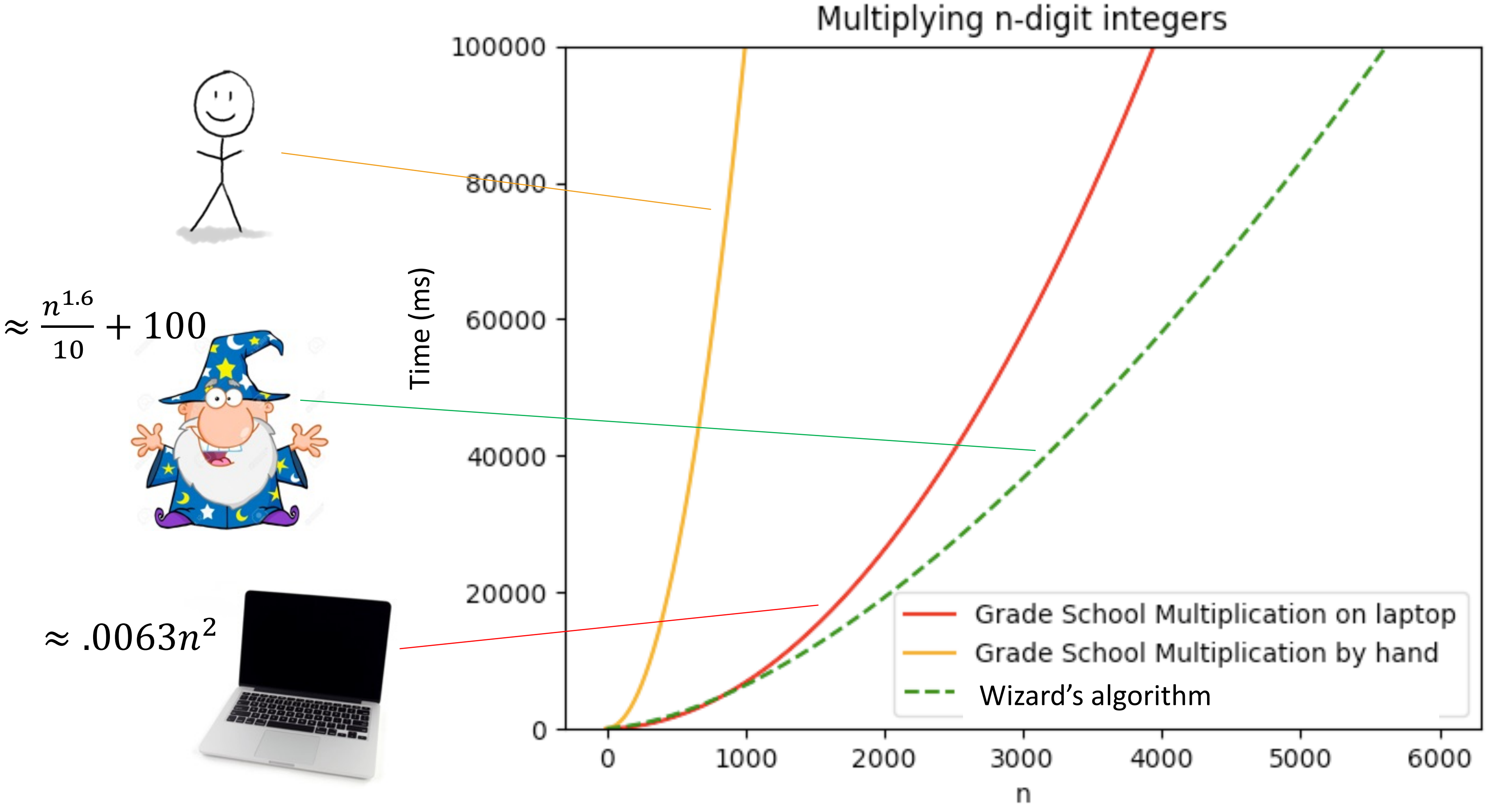
Why is big-Oh notation meaningful?



$\approx \frac{n^{1.6}}{10} + 100$

$\approx .0063n^2$

Let n get bigger…



$\approx \frac{n^{1.6}}{10} + 100$

$\approx .0063n^2$

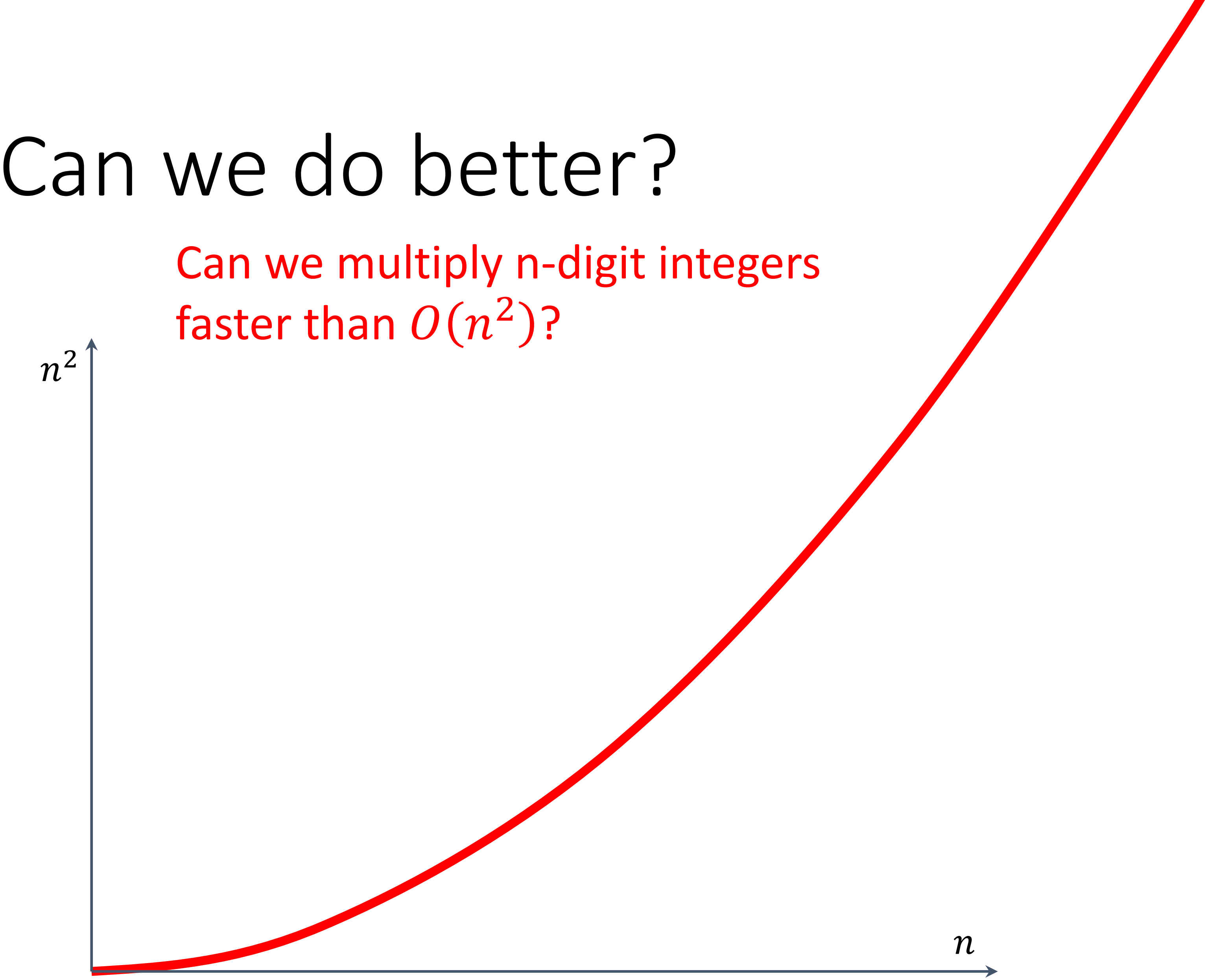
Take-away

- An algorithm that runs in time O($n^{1.6}$) is “better” than an algorithm that runs in time O(n^2).

- So the question is…

Can we do better?

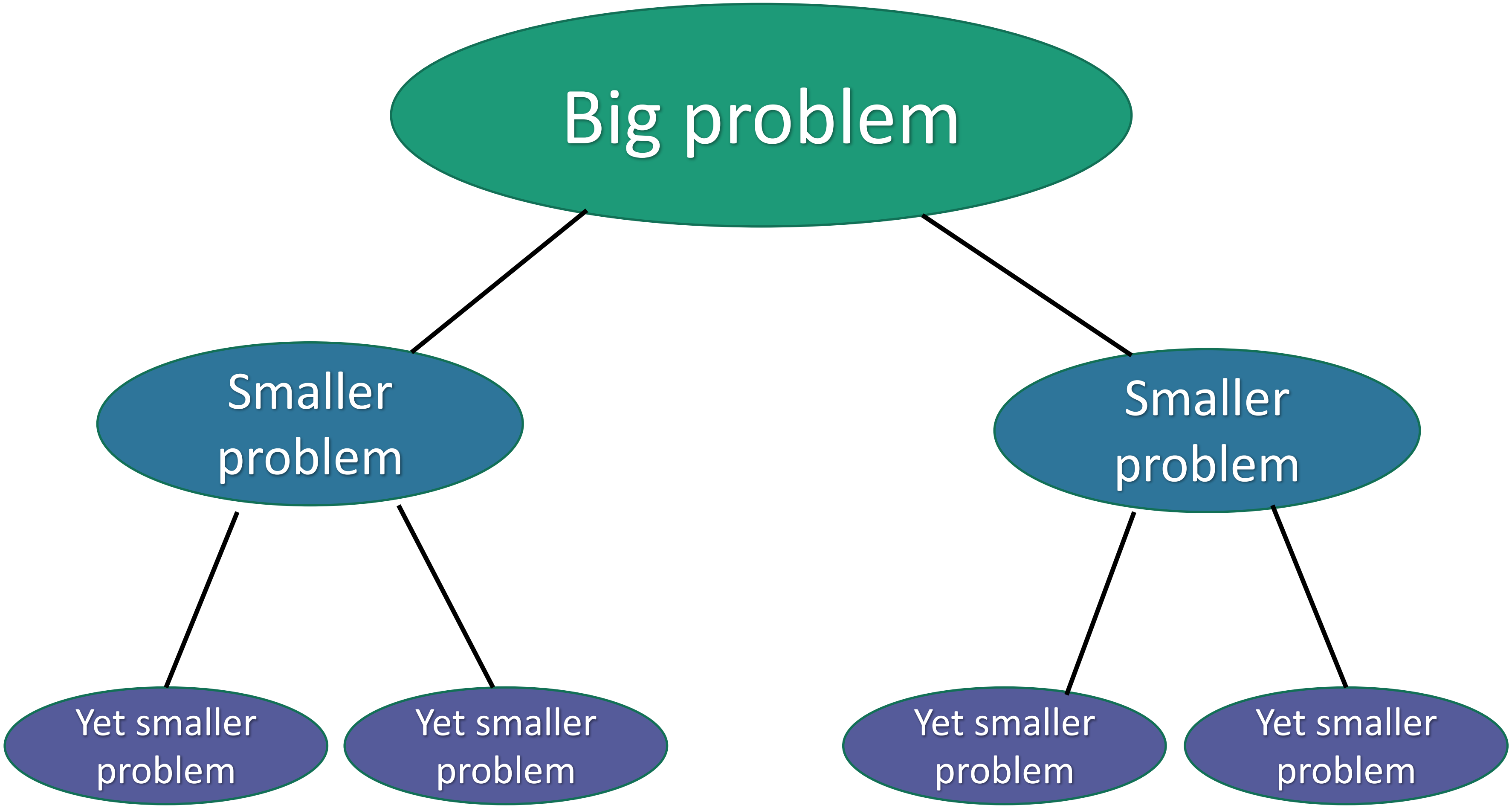
Can we multiply n-digit integers faster than $O(n^2)$?



Let’s dig into our algorithmic toolkit…

Divide and conquer

Break problem up into smaller (easier) sub-problems



Divide and conquer for multiplication

Break up an integer:

$$1234 = 12 \times 100 + 34$$

$$
\begin{aligned}
&1234 \times 5678 \\
&= (12 \times 100 + 34)(56 \times 100 + 78) \\
&= \underbrace{(12 \times 56)}_{\textcircled{1}} 10000 + \underbrace{(34 \times 56}_{\textcircled{2}} + \underbrace{12 \times 78)}_{\textcircled{3}} 100 + \underbrace{(34 \times 78)}_{\textcircled{4}}
\end{aligned}
$$

One 4-digit multiply → Four 2-digit multiplies

More generally

Suppose n is even

Break up an n-digit integer:

$$[x_1 x_2 \cdots x_n] = [x_1 x_2 \cdots x_{n/2}] \times 10^{n/2} + [x_{n/2+1} x_{n/2+2} \cdots x_n]$$

$$x \times y = (a \times 10^{n/2} + b)(c \times 10^{n/2} + d)$$

$$= \underbrace{(a \times c)}_{\textcircled{1}} 10^n + (\underbrace{a \times d}_{\textcircled{2}} + \underbrace{c \times b}_{\textcircled{3}}) 10^{n/2} + \underbrace{(b \times d)}_{\textcircled{4}}$$

One n-digit multiply → Four (n/2)-digit multiplies

Divide and conquer algorithm

not very precisely…

(Assume n is a power of 2…)

x,y are n-digit numbers

```
Multiply($x, y$):
  • If n=1:
      • Return xy
  • Write $x = a\,10^{\frac{n}{2}} + b$
  • Write $y = c\,10^{\frac{n}{2}} + d$
  • Recursively compute $ac, ad, bc, bd$:
      • ac = Multiply(a, c), etc..
  • Add them up to get $xy$:
      • xy = ac $10^n$ + (ad + bc) $10^{n/2}$ + bd
```

Base case: I've memorized my 1-digit multiplication tables…

a, b, c, d are n/2-digit numbers

Make this pseudocode more detailed! How should we handle odd n? How should we implement "multiplication by 10^n"?

Siggi the Studious Stork

See the Lecture 1 Python notebook for actual code!

Question

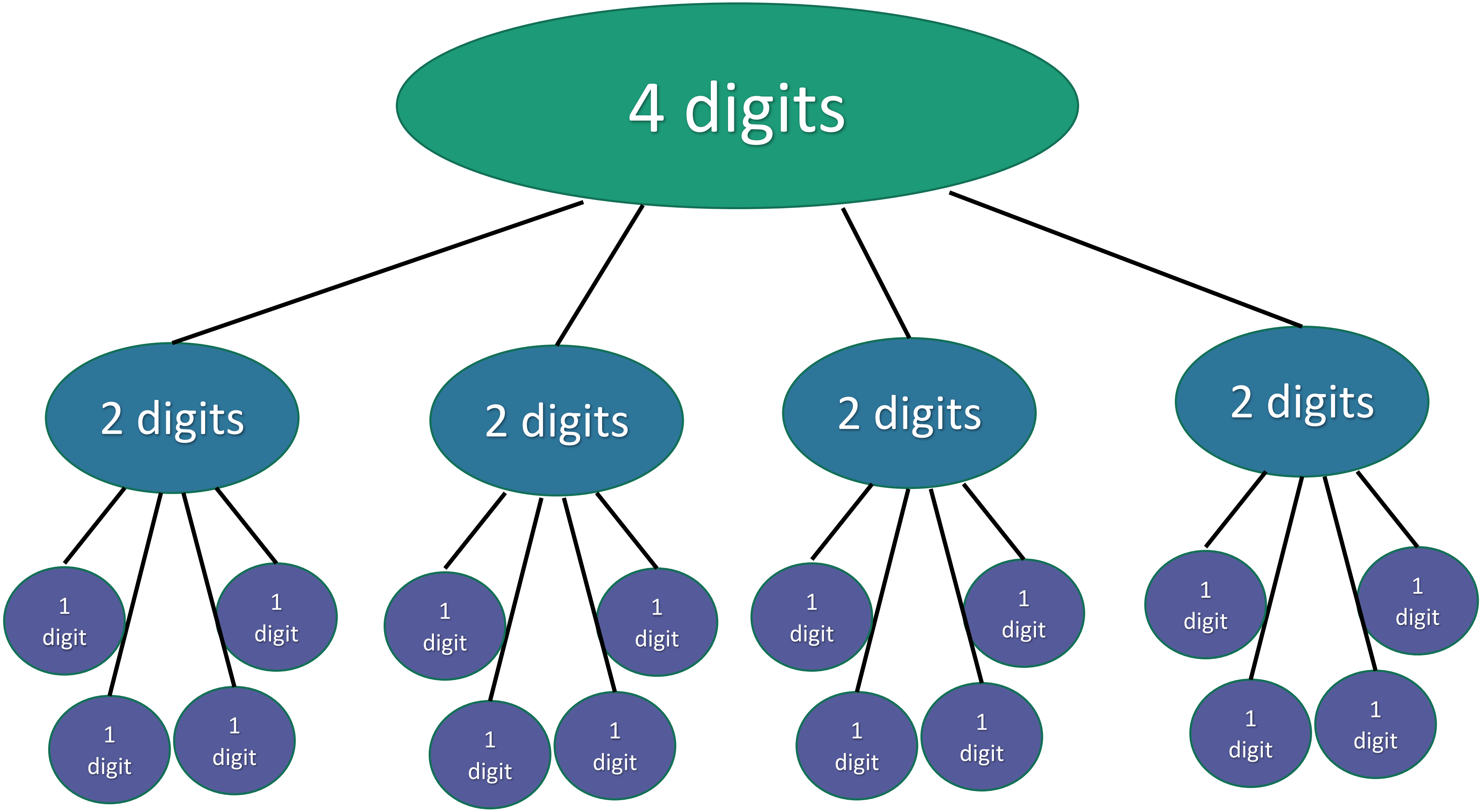
- We saw that 4-digit multiplication problem broke up into four 2-digit multiplication problems

$$1234 \times 5678$$

- If you recurse on those 2-digit multiplication problems, how many 1-digit multiplications do you end up with in total?

Recursion Tree

16 one-digit multiplies!



What is the running time?

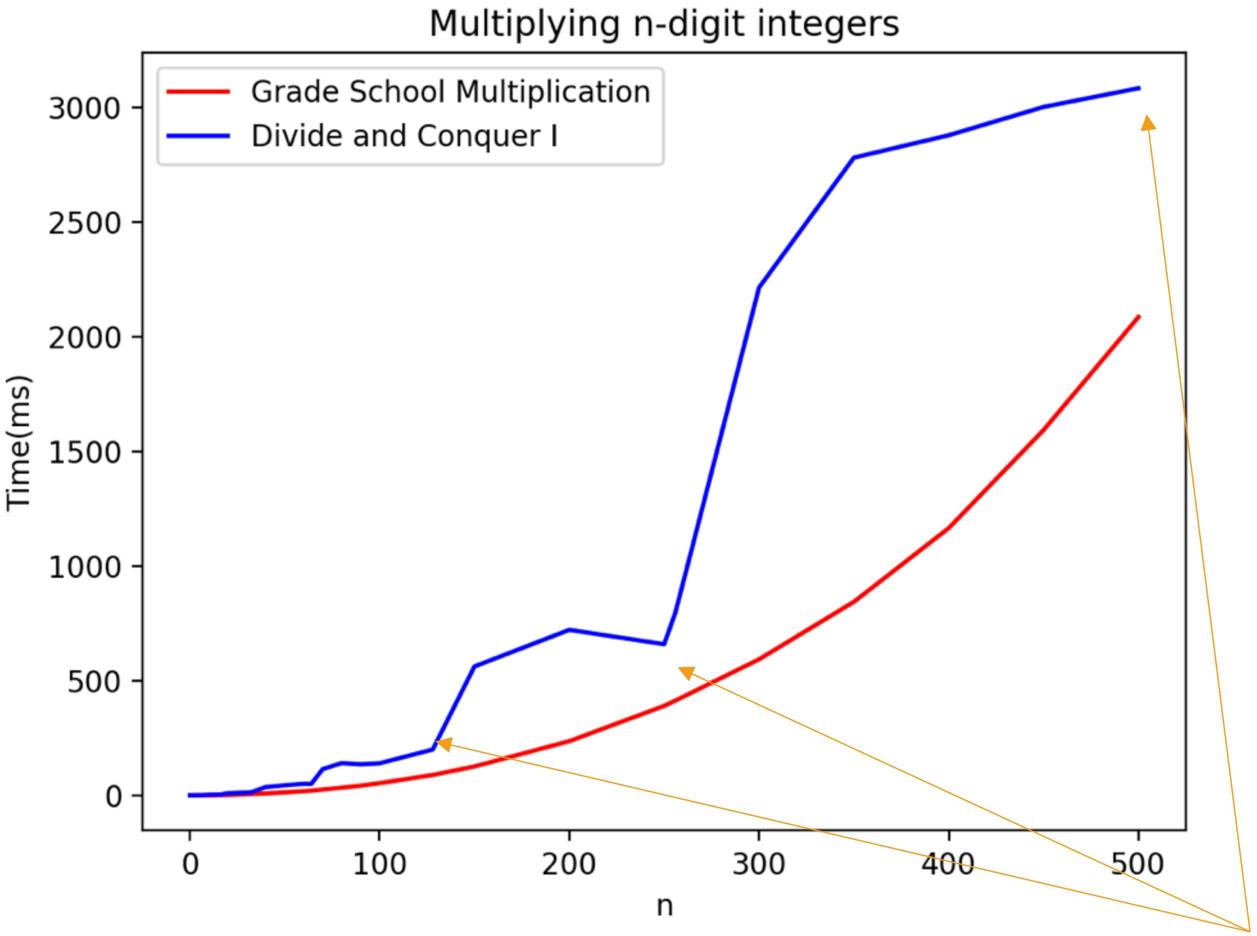
- Better or worse than the grade school algorithm?

- How do we answer this question?
  1. Try it.
  2. Try to understand it analytically.

1. Try it.

Check out the Lecture 1 IPython Notebook

Conjectures about running time?



Doesn’t look too good but hard to tell…

Maybe one implementation is slicker than the other?

Maybe if we were to run it to n=10000, things would look different.

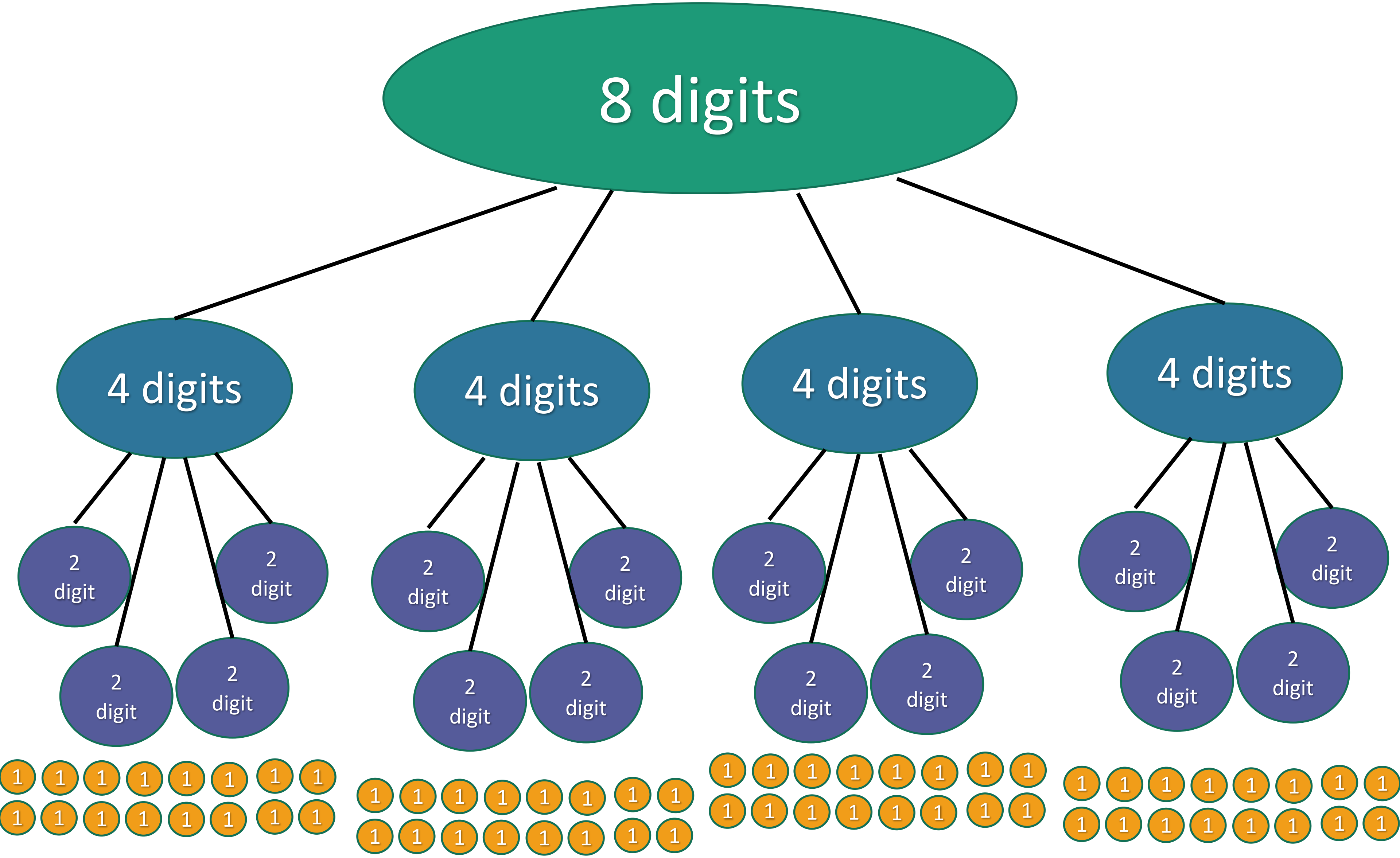
Something funny is happening at powers of 2…

2. Try to understand the running time analytically

- We saw that multiplying 4-digit numbers resulted in 16 one-digit multiplications.
- How about multiplying 8-digit numbers?
- What do you think about n-digit numbers?

Recursion Tree

64 one-digit multiplies!

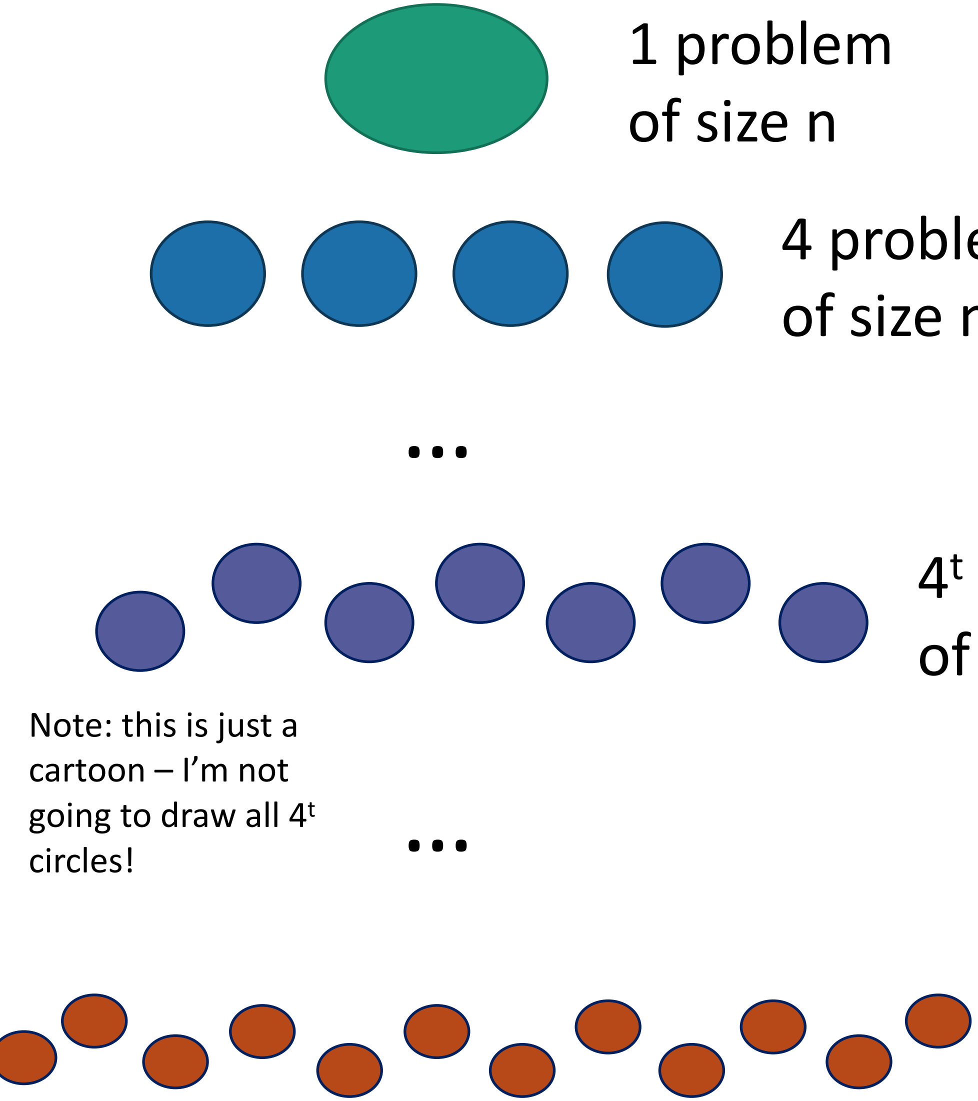


2. Try to understand the running time analytically

Claim:

The running time of this algorithm is AT LEAST n^2 operations.

There are n^2 1-digit problems



1 problem
of size n

4 problems
of size n/2

…

4^t problems
of size $n/2^t$

Note: this is just a cartoon – I'm not going to draw all 4^t circles!

…

$\underline{n^2}$ problems
of size 1

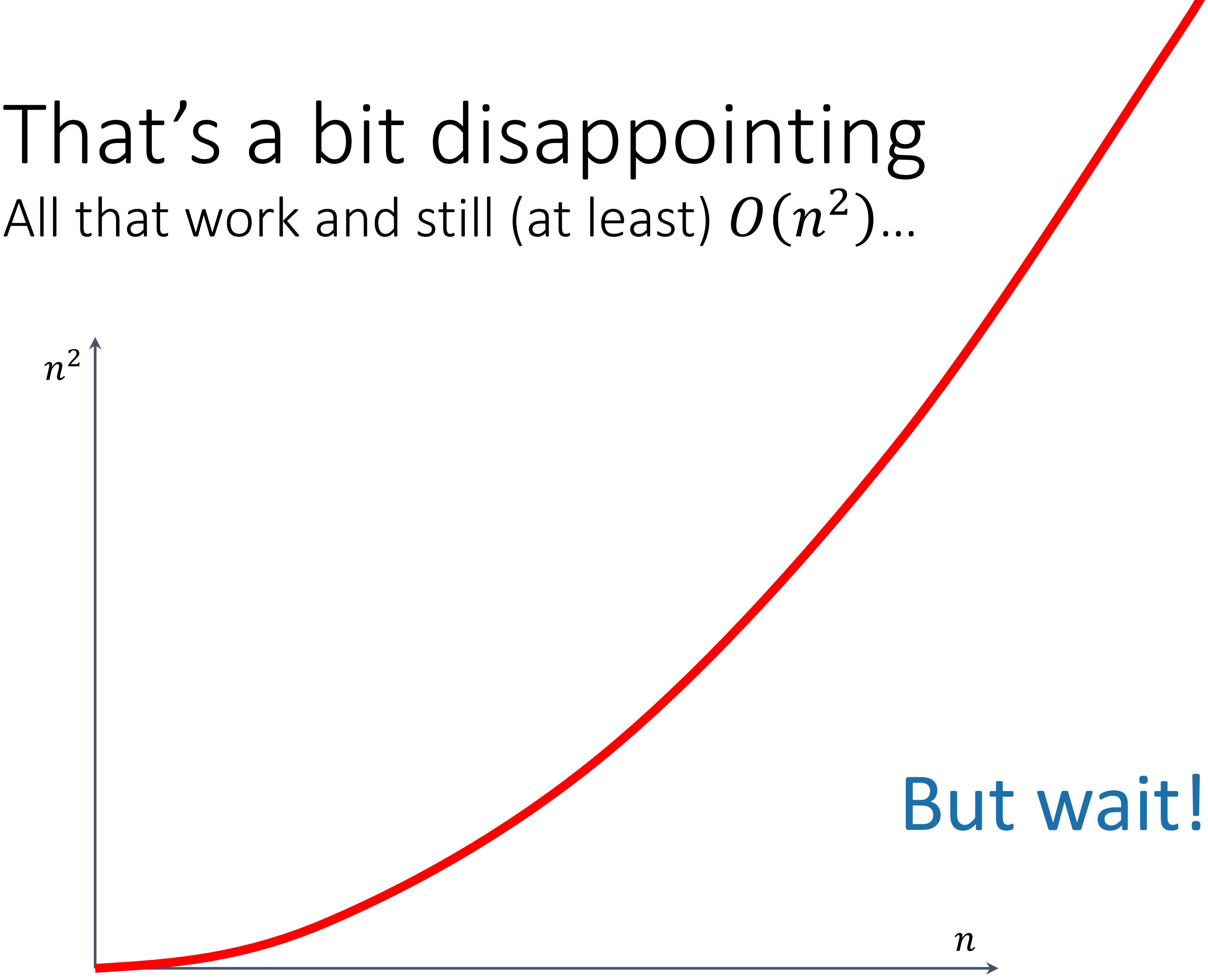
- If you cut n in half $\log_2(n)$ times, you get down to 1.
- So, at level
  $t = \log_2(n)$
  we get…

$$4^{\log_2 n} = n^{\log_2 4} = n^2$$

problems of size 1.

That’s a bit disappointing

All that work and still (at least) $O(n^2)$…



But wait!!

Divide and conquer can actually make progress

- Karatsuba figured out how to do this better!

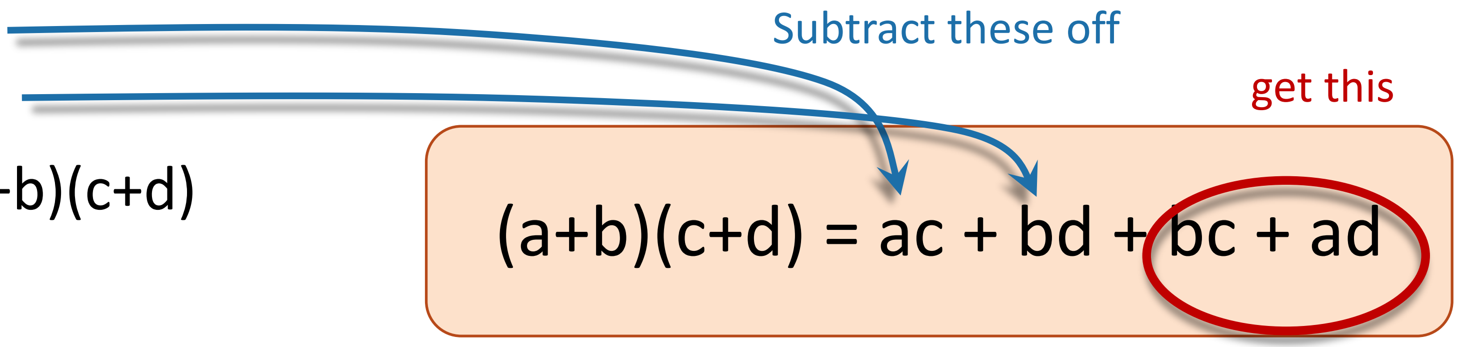
$$\begin{aligned} xy &= (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d) \\ &= ac \cdot 10^{n} + (ad + bc)10^{n/2} + bd \end{aligned}$$

Need these three things

- If only we could recurse on three things instead of four…

Karatsuba integer multiplication

- Recursively compute these THREE things:
  - ac
  - bd
  - (a+b)(c+d)



Subtract these off

get this

(a+b)(c+d) = ac + bd + bc + ad

- Assemble the product:

$$
\begin{aligned}
xy &= (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d) \\
&= ac \cdot 10^{n} + (ad + bc)10^{n/2} + bd
\end{aligned}
$$

✓ ✓ ✓

How would this work?

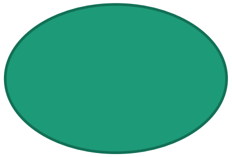
(Still not super precise, see IPython notebook for detailed code. Also, still assume n is a power of 2.)

x,y are n-digit numbers

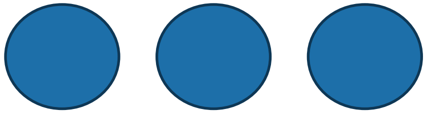
a, b, c, d are n/2-digit numbers

```
**Multiply**$(x, y)$:
  • **If** n=1:
      • **Return** xy
  • Write $x = a\,10^{\frac{n}{2}} + b$ and $y = c\,10^{\frac{n}{2}} + d$
  • ac = **Multiply**(a, c)
  • bd = **Multiply**(b, d)
  • z = **Multiply**(a+b, c+d)
  • xy = ac $10^n$ + (z – ac - bd) $10^{n/2}$ + bd
  • Return xy
```

What’s the running time?

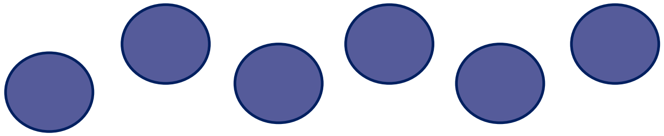


1 problem
of size n



3 problems
of size n/2

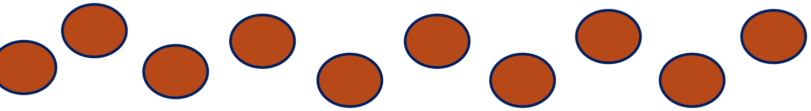
…



3^t problems
of size $n/2^t$

Note: this is just a cartoon – I’m not going to draw all 3^t circles!

…



$\underline{n^{1.6}}$ problems
of size 1

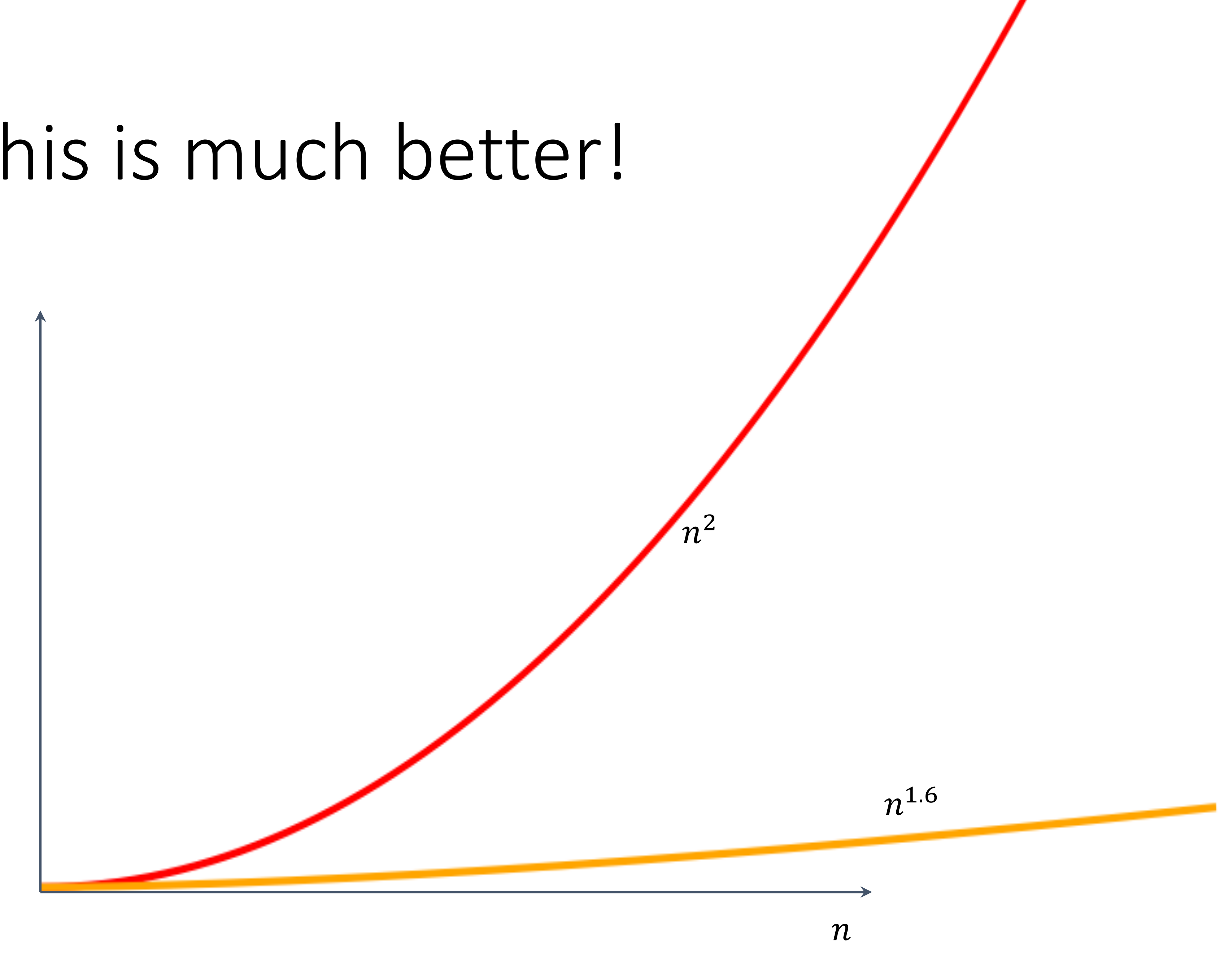
- If you cut n in half $\log_2(n)$ times, you get down to 1.
- So at level
  $t = \log_2(n)$
  we get…

$$3^{\log_2 n} = n^{\log_2 3} \approx n^{1.6}$$

problems of size 1.

We aren’t accounting for the work at the higher levels! But we’ll see next lecture that this turns out to be okay.

This is much better!



Can we do better?

- **Toom-Cook** (1963): instead of breaking into three n/2-sized problems, break into five n/3-sized problems.
  - Runs in time $O(n^{1.465})$

Try to figure out how to break up an n-sized problem into five n/3-sized problems! **(Hint: start with nine n/3-sized problems).**

Ollie the Over-achieving Ostrich

Given that you can break an n-sized problem into five n/3-sized problems, where does the 1.465 come from?

Siggi the Studious Stork

- **Schönhage–Strassen** (1971):
  - Runs in time $O(n \log(n) \log\log(n))$
- **Furer** (2007)
  - Runs in time $n \log(n) \cdot 2^{O(\log^*(n))}$
- **Harvey and van der Hoeven** (2019)
  - Runs in time $O(n \log(n))$

[This is just for fun, you don’t need to know these algorithms!]

Sorting

- 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

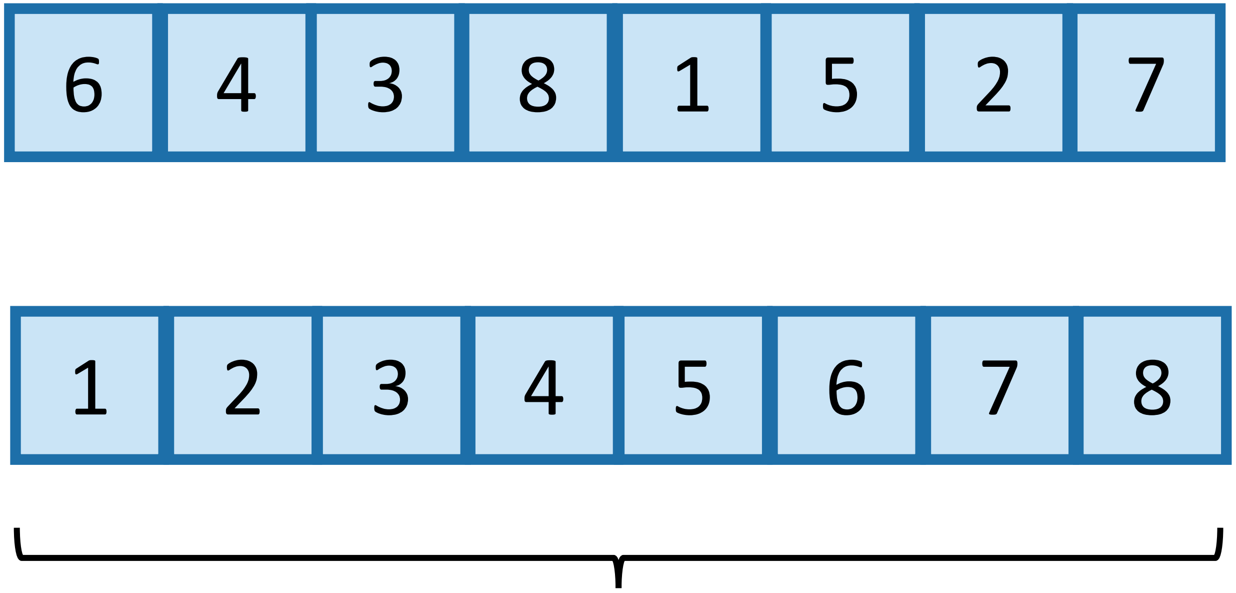
SortingHatSort not discussed

The Plan

- Sorting!
- Worst-case analysis
    - InsertionSort: Does it work?
- Asymptotic Analysis
    - InsertionSort: Is it fast?
- MergeSort
    - Does it work?
    - Is it fast?

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

Length of the list is n

Pre-lecture exercise:

What was the mystery sort algorithm?

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

```
def mysteryAlgorithmOne(A):
  B = [None for i in range(len(A))]
  for x in 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
```

Pre-lecture exercise:

What was the mystery sort algorithm?

1. MergeSort
2. QuickSort
3. [✓ InsertionSort]
4. BogoSort

```
def mysteryAlgorithmOne(A):
  B = [None for i in range(len(A))]
  for x in 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

6	4	3	8	5

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

6	**4**	3	8	5
4	6	3	8	5

Then move A[2]:

4	6	**3**	8	5
3	4	6	8	5

Then move A[3]:

3	4	6	**8**	5
3	4	6	**8**	5

Then move A[4]:

3	4	6	8	**5**
3	4	**5**	6	8

Then we are done!

Insertion Sort

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

What does that mean???

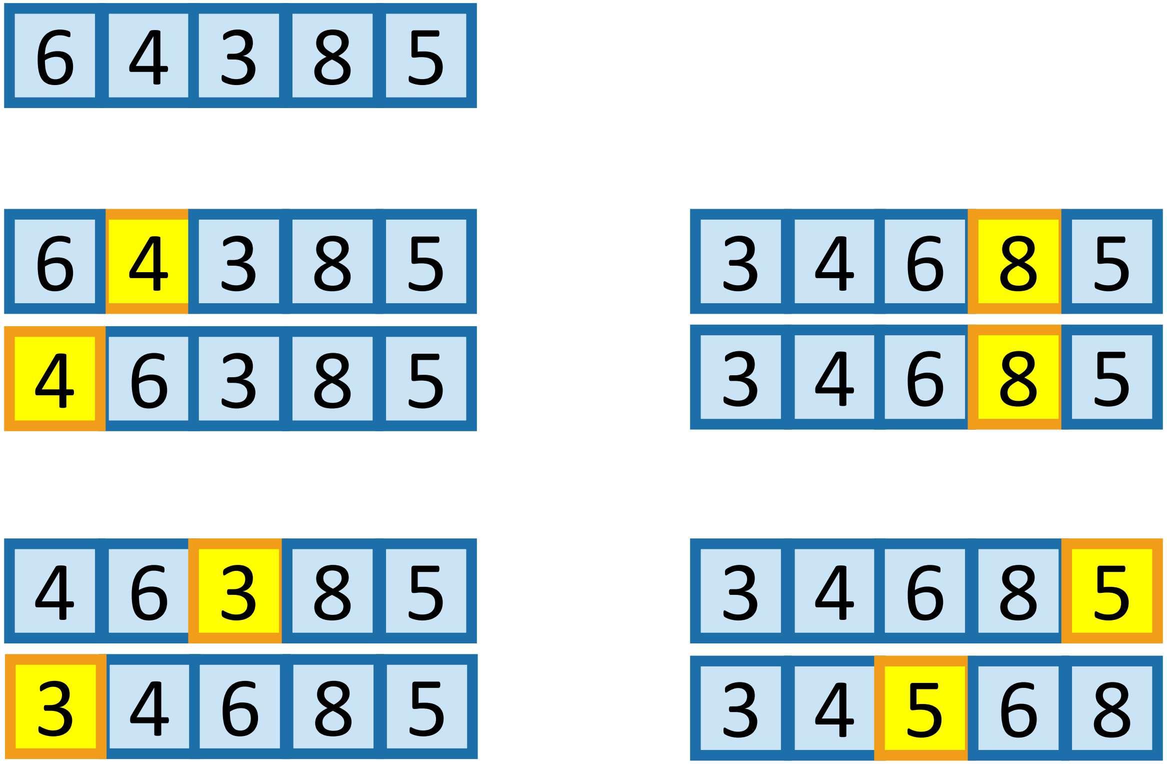
Plucky the
Pedantic Penguin

The Plan

- InsertionSort recap
- Worst-case Analysis
    - Back to InsertionSort: Does it work?
- Asymptotic Analysis
    - Back to InsertionSort: Is it fast?
- MergeSort
    - Does it work?
    - Is it fast?

Claim: InsertionSort “works”

- “Proof:” It just worked in this example:



6 4 3 8 5

6 4 3 8 5

4 6 3 8 5

4 6 3 8 5

3 4 6 8 5

3 4 6 8 5

3 4 6 8 5

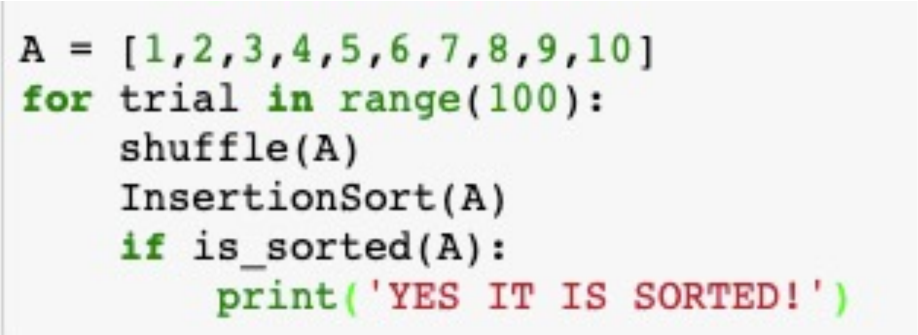
3 4 6 8 5

3 4 5 6 8

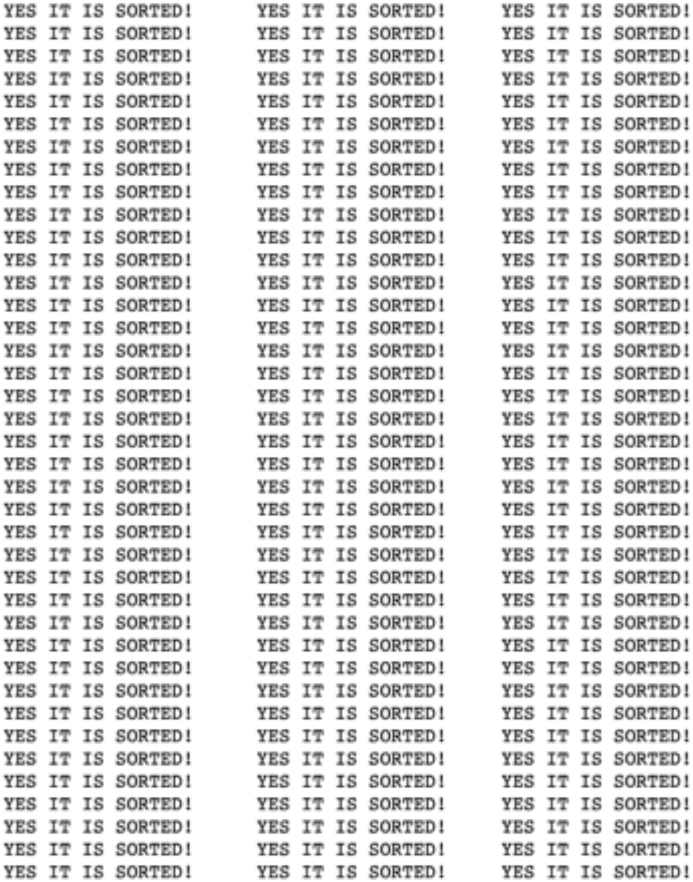
Sorted!

Claim: InsertionSort “works”

- “Proof:” I did it on a bunch of random lists and it always worked:



```
A = [1,2,3,4,5,6,7,8,9,10]
for trial in range(100):
    shuffle(A)
    InsertionSort(A)
    if is_sorted(A):
        print('YES IT IS SORTED!')
```



```
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
YES IT IS SORTED!    YES IT IS SORTED!    YES IT IS SORTED!
```

What does it mean to “work”?

- Is it enough to be correct on only one input?
- Is it enough to be correct on most inputs?

- In this class, we will use **worst-case analysis**:
  - An algorithm must be correct on **all possible** inputs.
  - The running time of an algorithm is the worst possible running time over all inputs.

Worst-case analysis

Think of it like a game:

Here is my algorithm!

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

Algorithm designer

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

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

Insertion Sort

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

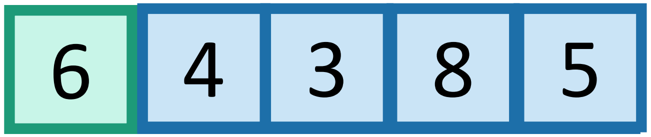
- Okay, so it’s pretty obvious that it works.

- 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.

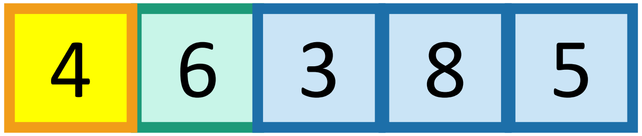
Why does this work?

- Say you have a sorted list, $\boxed{3}\boxed{4}\boxed{6}\boxed{8}$, 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: $\boxed{3}\boxed{4}\boxed{5}\boxed{6}\boxed{8}$

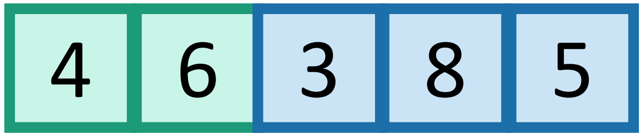
So just use this logic at every step.



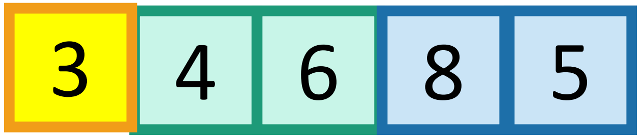
The first element, [6], makes up a sorted list.



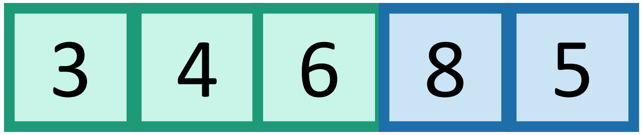
So correctly inserting 4 into the list [6] means that [4,6] becomes a sorted list.



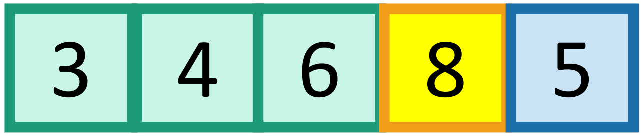
The first two elements, [4,6], make up a sorted list.



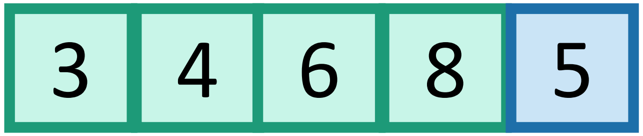
So correctly inserting 3 into the list [4,6] means that [3,4,6] becomes a sorted list.



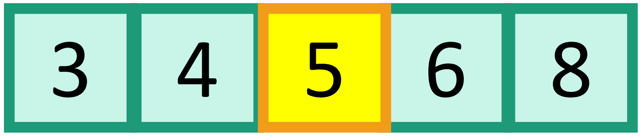
The first three elements, [3,4,6], make up a sorted list.



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.

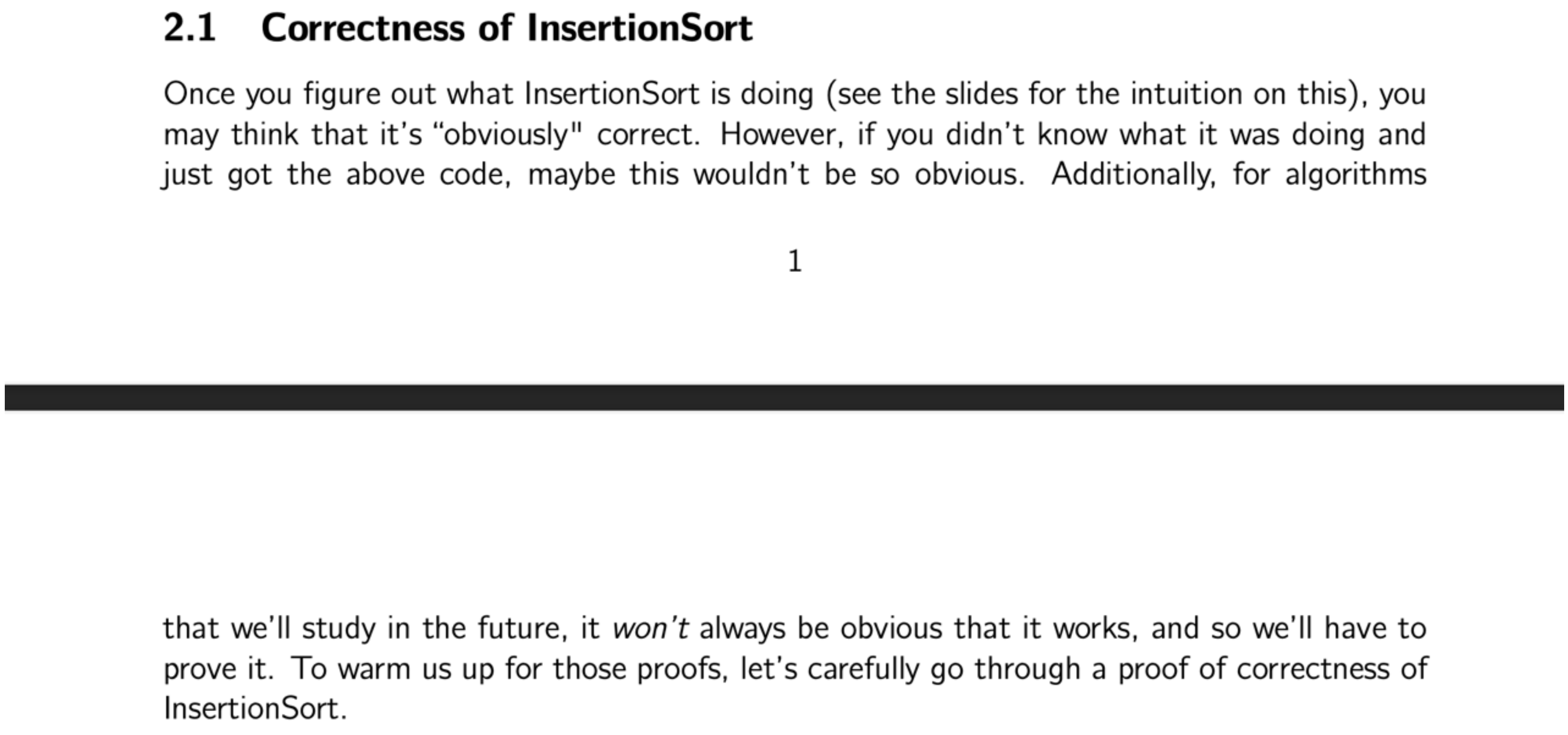
YAY WE ARE DONE!

This sounds like a job for…

Proof By Induction!

The notes contain the details!

- See website!



2.1 Correctness of InsertionSort

Once you figure out what InsertionSort is doing (see the slides 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. To warm us up for those proofs, let's carefully go through a proof of correctness of InsertionSort.

Outline of a proof by induction

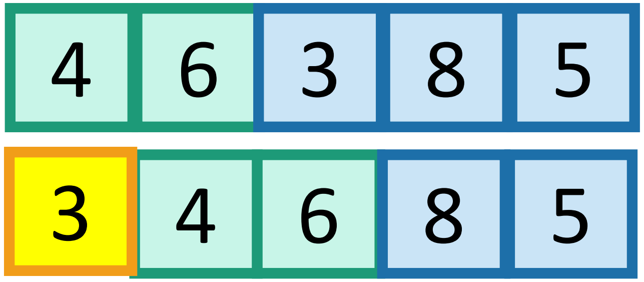
Let A be a list of length n

- Inductive Hypothesis:
  - A[:i+1] is sorted at the end of the i^{th} iteration (of the outer loop).
- Base case (i=0):
  - `A[:1]` is sorted at the end of the 0'th iteration. ✓
- Inductive step:
  - For any 0 < k < n, if the inductive hypothesis holds for i=k-1, then it holds for i=k.
  - Aka, if A[:k] is sorted at step k-1, then A[:k+1] is sorted at step k

This logic

(see notes for details)

- Conclusion:
  - The inductive hypothesis holds for i = 0, 1, …, n-1.
  - In particular, it holds for i=n-1.
  - 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! ✓



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.

This was iteration i=2.

Aside: proofs by induction

- We're going to 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:
  - GO TO SECTION
  - **GO TO SECTION**
  - Notes
  - References
  - Office hours

Make sure you really understand the argument on the previous slide! Check out the notes for a more formal write-up and go to the sections for an overview of what we are looking for in proofs by induction.

Siggi the Studious Stork

What have we learned?

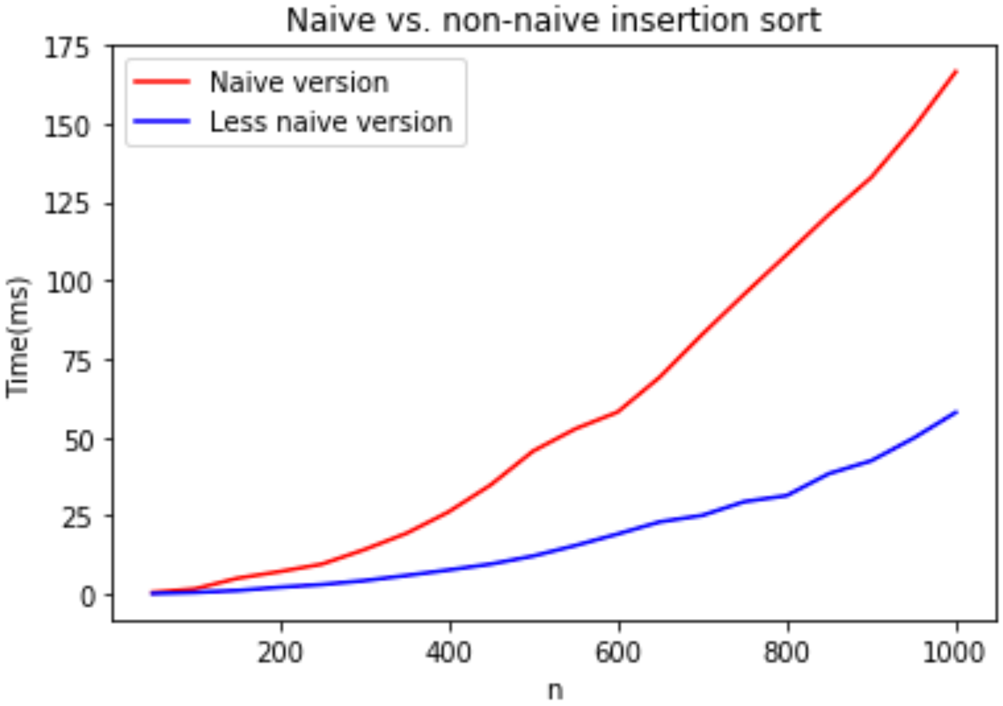
- In this class we will use worst-case analysis:
  - We assume that a “bad guy” produces a worst-case input for our algorithm, and we measure performance on that worst-case input.
- With this definition, InsertionSort “works”
  - Proof by induction!

The Plan

- InsertionSort recap
- Worst-case Analysis
  - Back to InsertionSort: Does it work?
- Asymptotic Analysis
  - Back to InsertionSort: Is it fast?
- MergeSort
  - Does it work?
  - Is it fast?

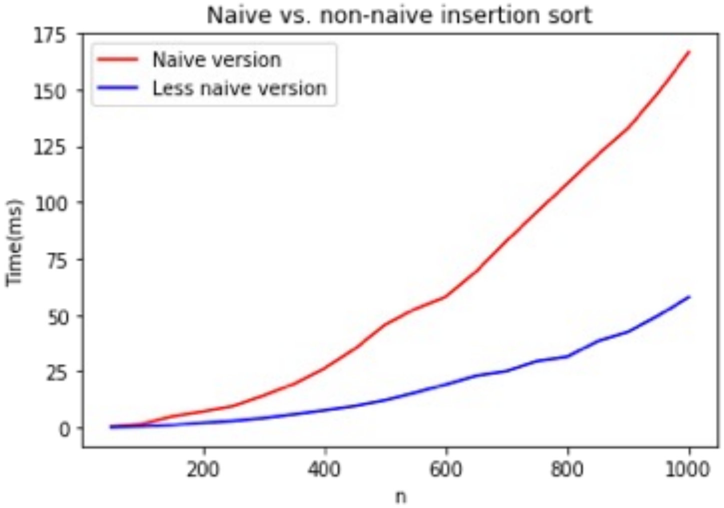
How fast is InsertionSort?

- This fast:



Issues with this answer?

- The “same” algorithm can be slower or faster depending on the implementations.
- It can also be slower or faster depending on the hardware that we run it on.



With this answer, “running time” isn’t even well-defined!

How fast is InsertionSort?

- Let’s count the number of operations!

```
def InsertionSort(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
```

By my count*…

- $2n^2 - n \ - 1$ variable assignments
- $2n^2 - n \ - 1$ increments/decrements
- $2n^2 - 4n + 1$ comparisons
- …

*Do not pay attention to these formulas, they do not matter. Also not valid for bug bounty (good citizenship) points.

Issues with this answer?

- It’s very tedious!
- In order to use this to understand running time, I need to know how long each operation takes, plus a whole bunch of other stuff…

```
def InsertionSort(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
```

Counting individual operations is a lot of work and doesn’t seem very helpful!

Lucky the lackadaisical lemur

In this class we will use…

- **Big-Oh notation!**
- Gives us a meaningful way to talk about the running time of an algorithm, independent of programming language, computing platform, etc., without having to count all the operations.

Main idea:

Pre-lecture exercise:

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

Some examples…

(Heuristically: only pay attention to the largest function of n that appears.)

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

We say this algorithm is "asymptotically faster" than the others.

Why is this a good idea?

- Suppose the running time of an algorithm is:

$$T(n) = 10n^2 + 3n + 7 \quad \text{ms}$$

This constant factor of 10 depends a lot on my computing platform…

These lower-order terms don't really matter as n gets large.

We're just left with the n^2 term! That's what's meaningful.

Pros and Cons of Asymptotic Analysis

Pros:

- Abstracts away from hardware- and language-specific issues.
- Makes algorithm analysis much more tractable.
- Allows us to meaningfully compare how algorithms will perform on large inputs.

Cons:

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

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

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

Informal definition for O(…)

- 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 $O\big(g(n)\big)$” if:

for all large enough n,

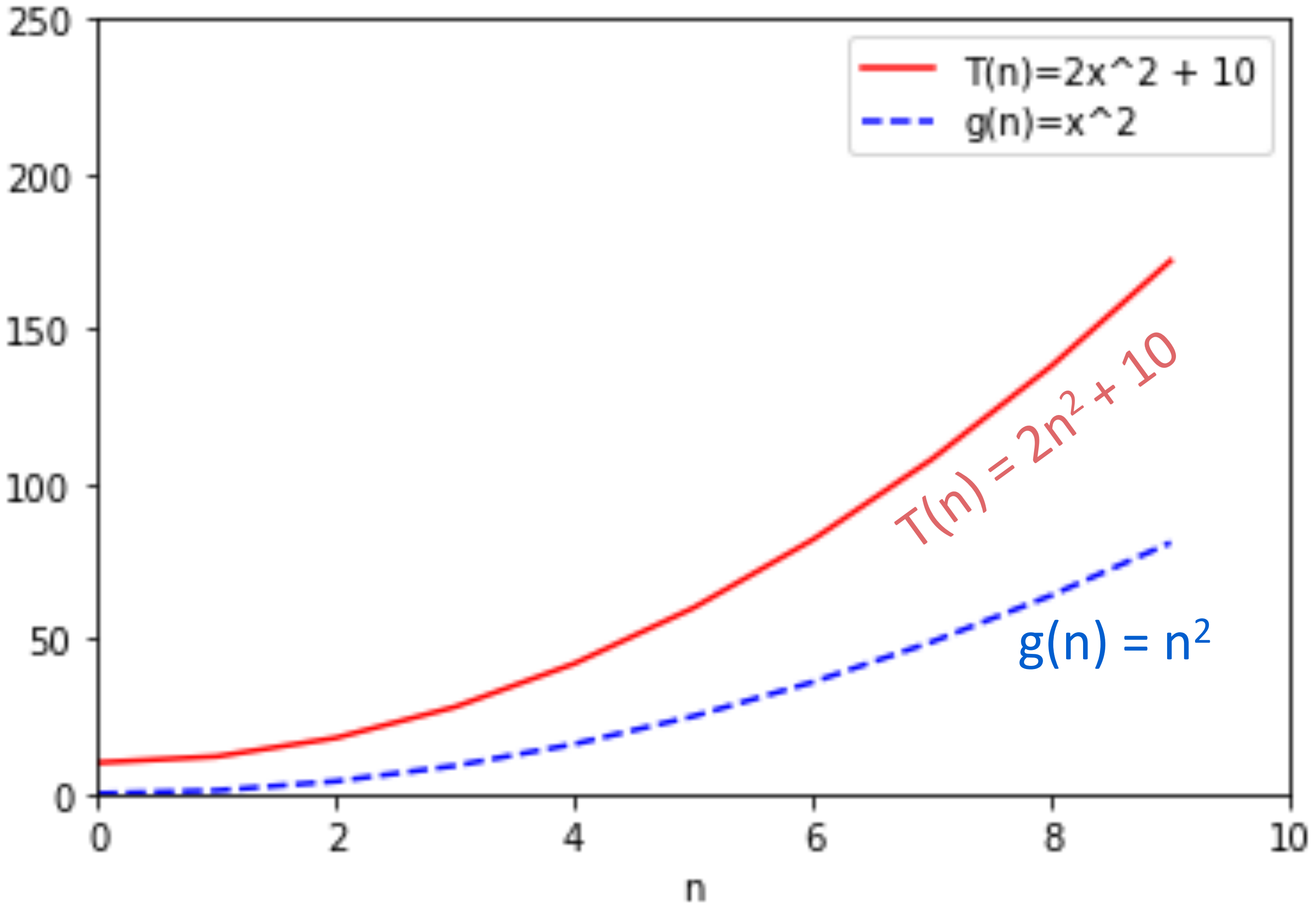
$T(n)$ is at most some constant multiple of $g(n)$.

Here, “constant” means “some number that doesn’t depend on n.”

Example

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

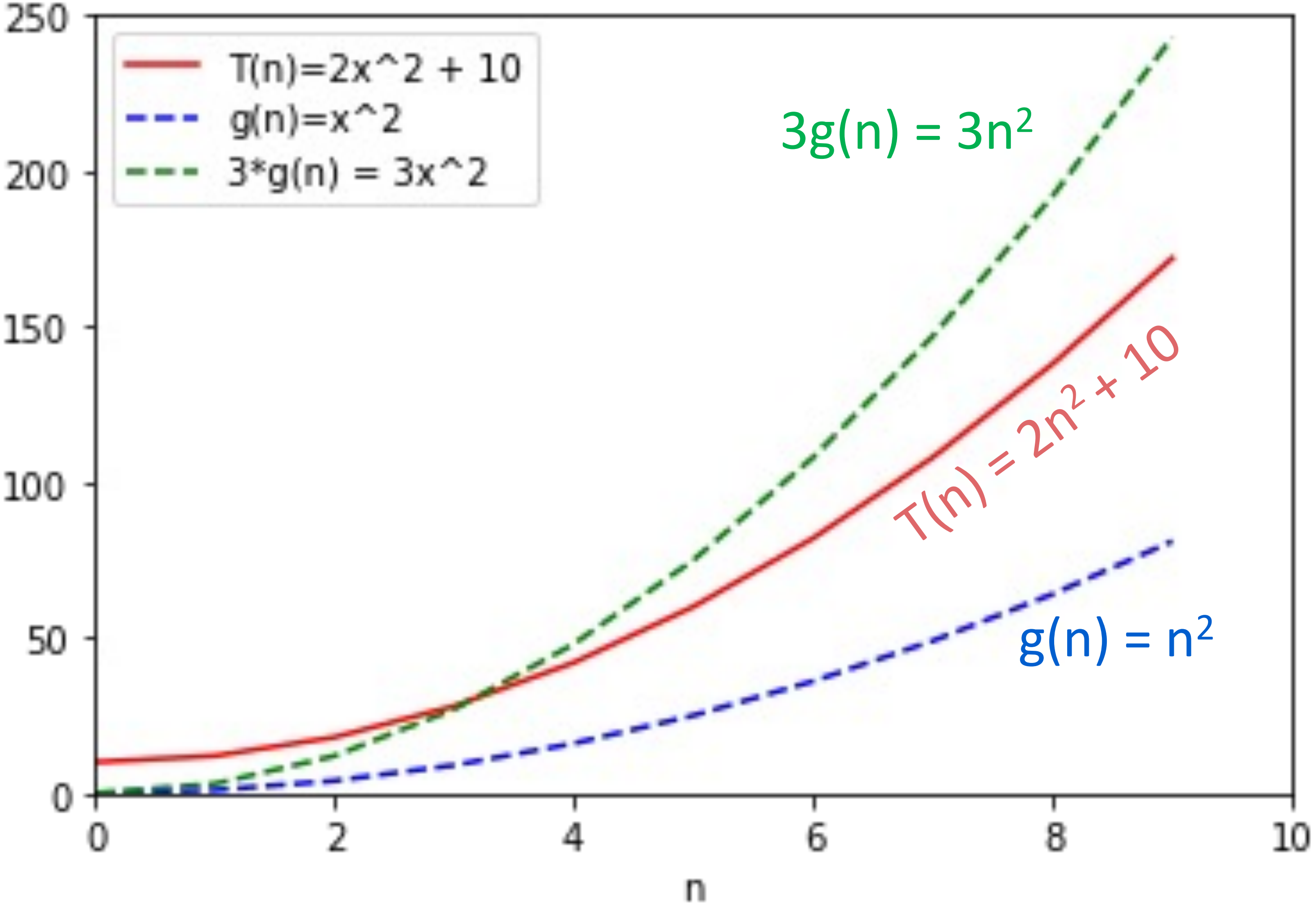
for large enough n, $T(n)$ is at most some constant multiple of $g(n)$.



Example

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

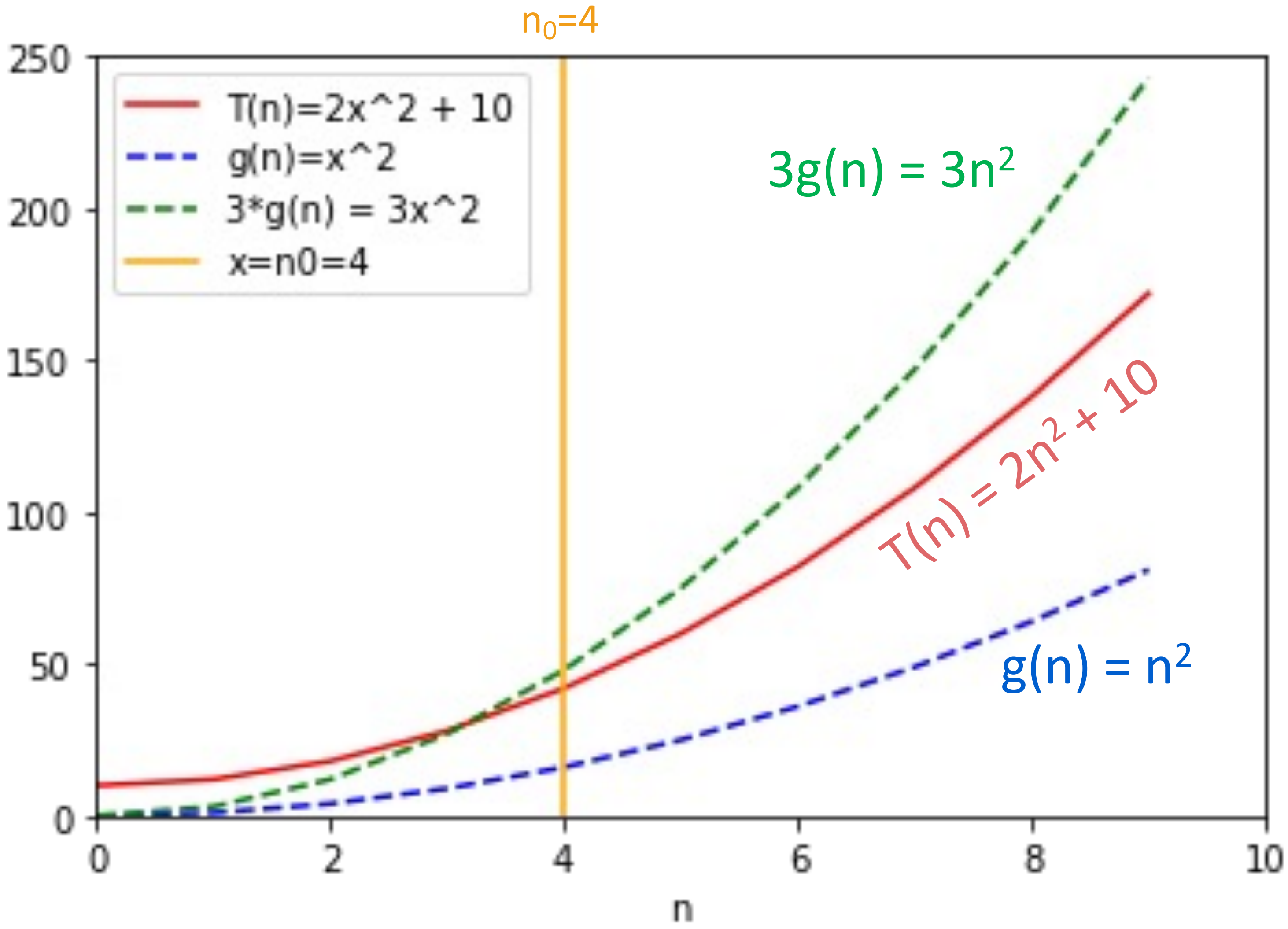
for large enough n,
$T(n)$ is at most some constant
multiple of $g(n)$.



Example

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

for large enough n,
$T(n)$ is at most some constant
multiple of $g(n)$.



Formal definition of O(...)

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

- Formally,

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

"If and only if"

$$\Leftrightarrow$$

"For all"

$$\exists c > 0, n_0 \;\; s.t. \;\; \forall n \geq n_0,$$

"There exists"

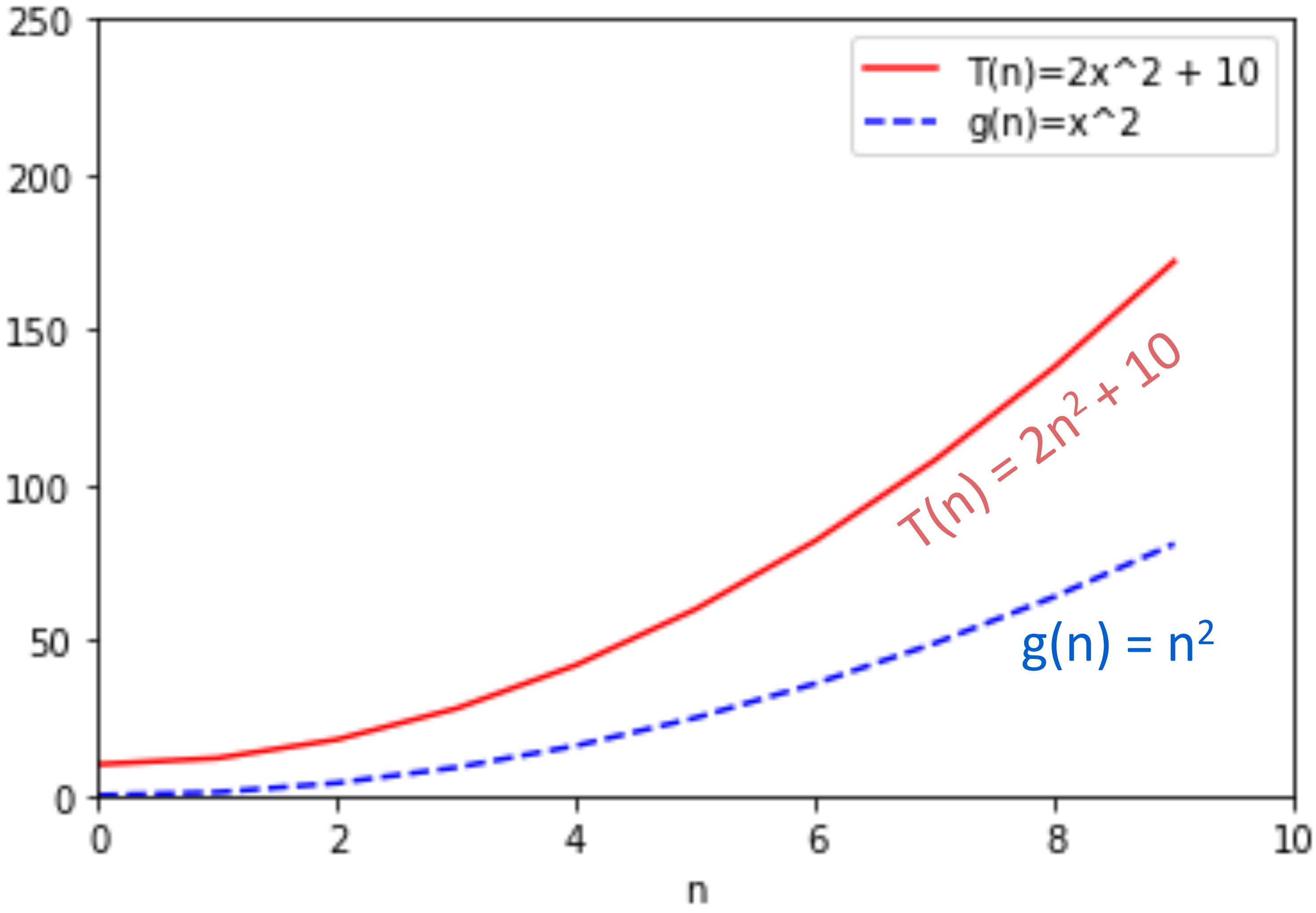
$$T(n) \leq c \cdot g(n)$$

"such that"

Example

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

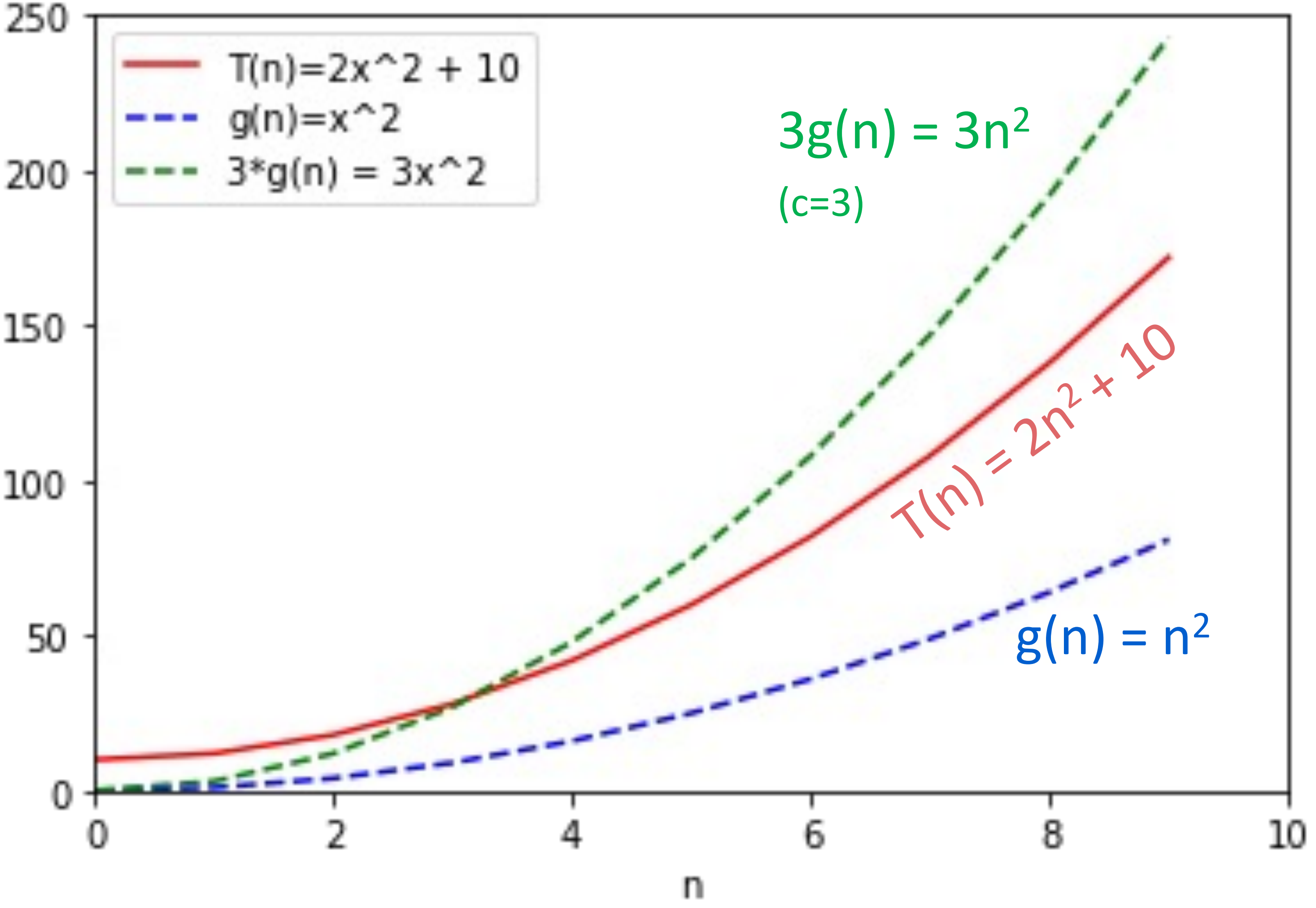
$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c > 0, n_0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Example

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

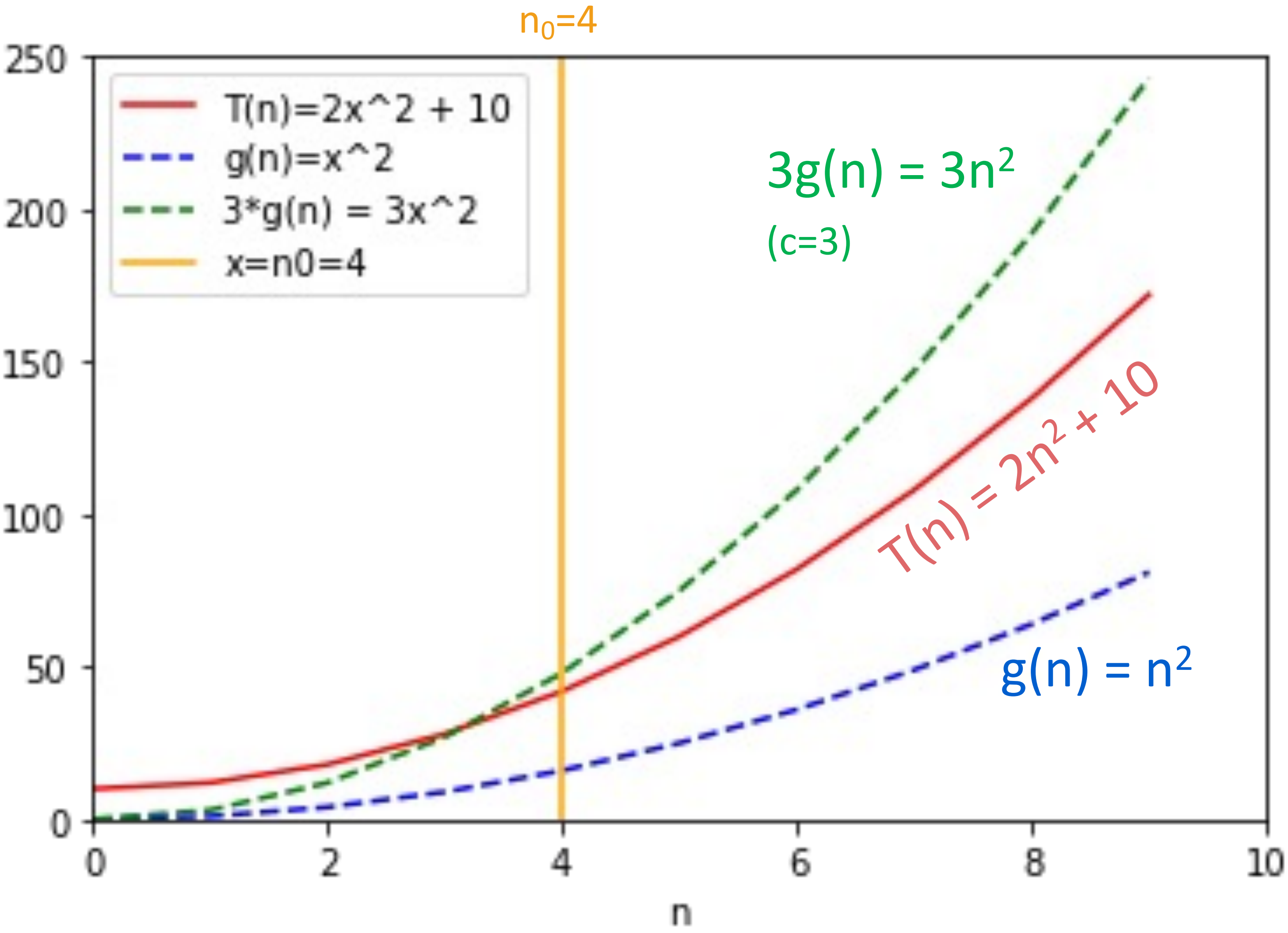
$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c > 0, n_0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Example

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

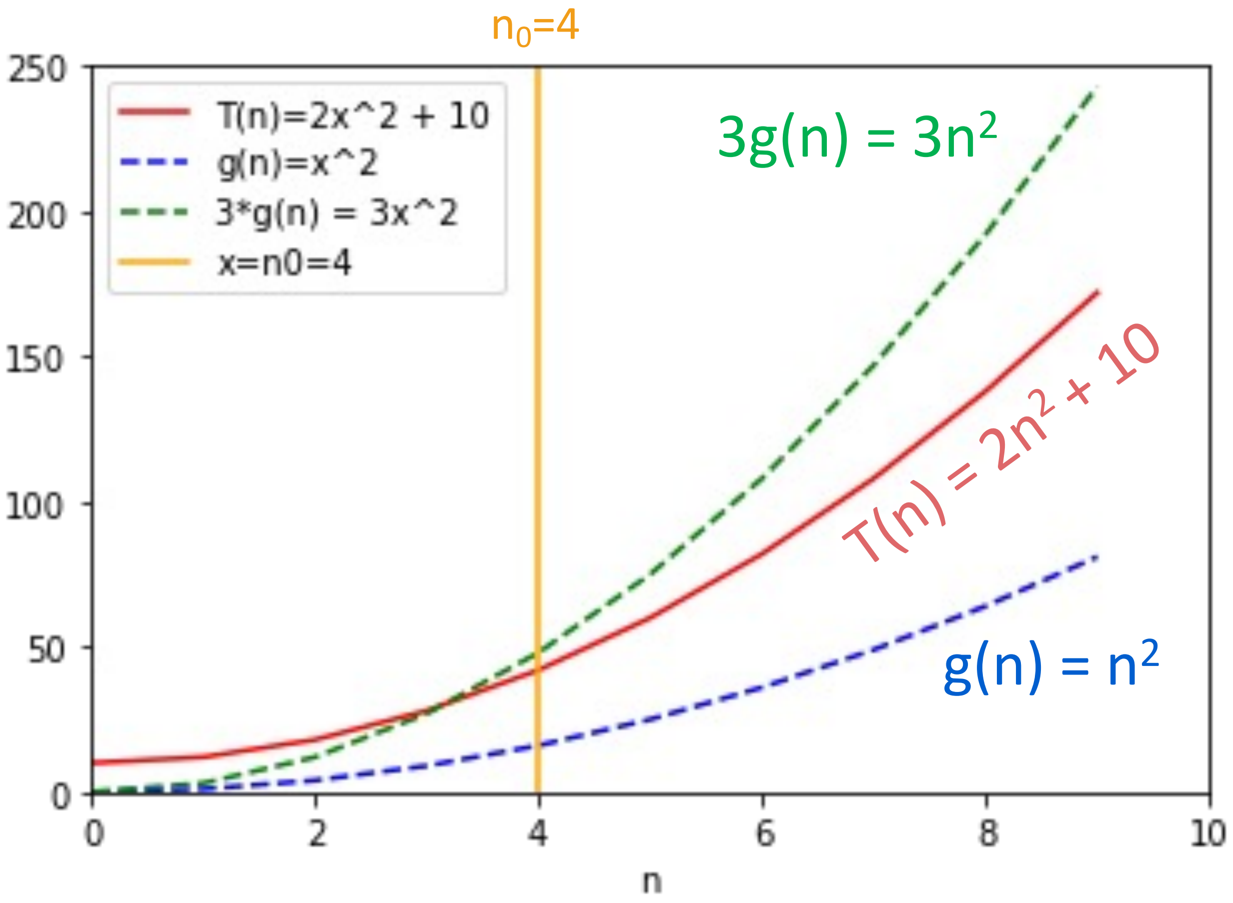
$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c > 0, n_0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Example

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

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



Formally:

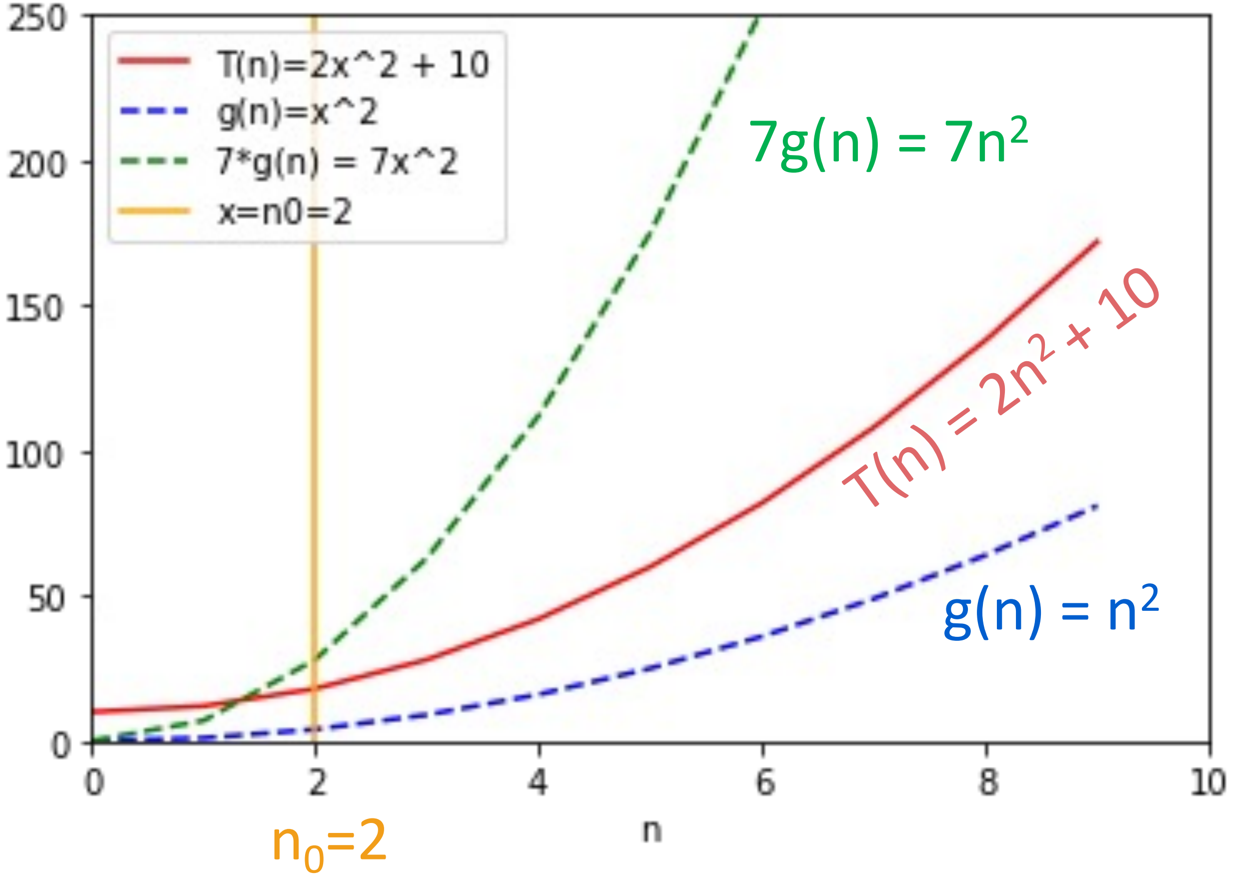
- Choose c = 3
- Choose n_0 = 4
- Then:

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

Same example

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

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



Formally:

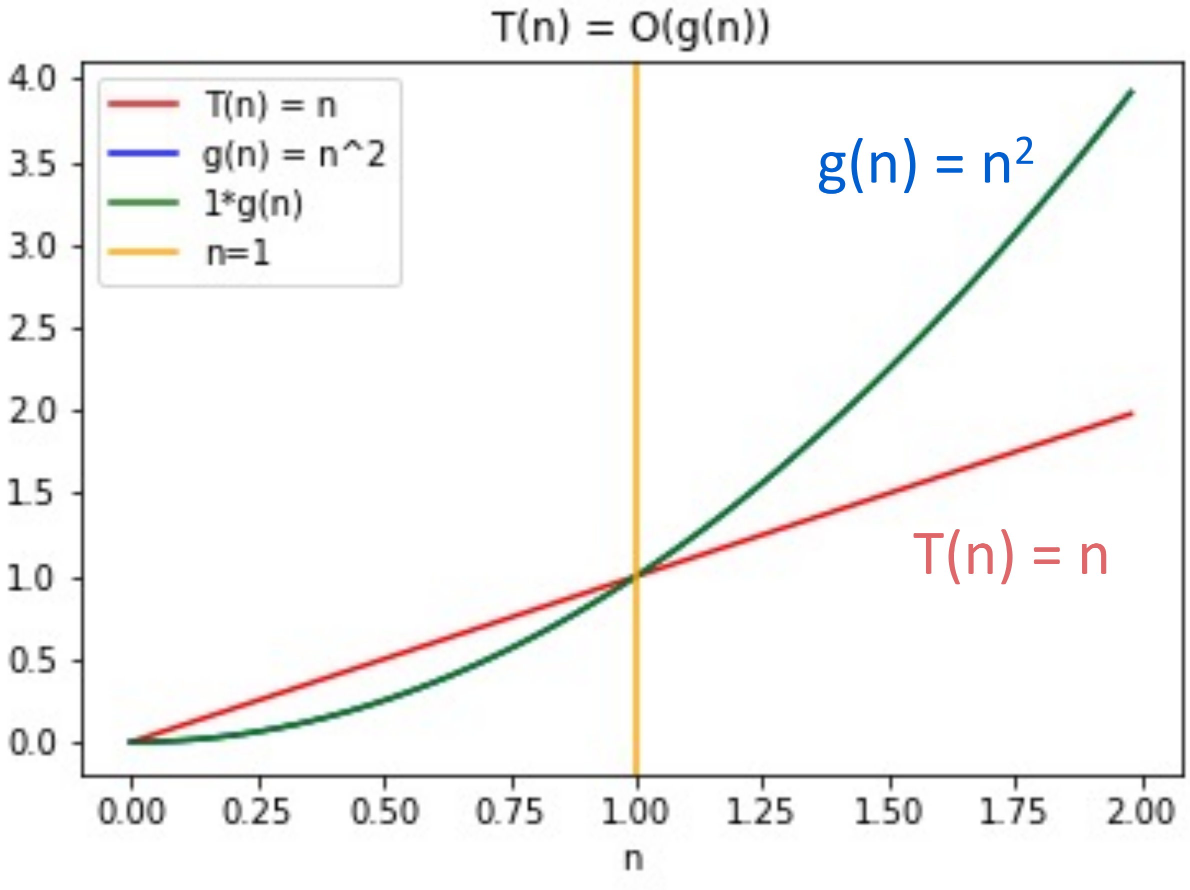
- Choose c = 7
- Choose n_0 = 2
- Then:

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

There is not a "correct" choice of c and n_0

O(…) is an upper bound: $n = O(n^2)$

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



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

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

Ω(…) means a lower bound

- We say "$T(n)$ is $\Omega\big(g(n)\big)$" if, for large enough n, $T(n)$ is at least as big as a constant multiple of $g(n)$.

- Formally,

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

$$\Longleftrightarrow$$

$$\exists c > 0\,, n_0 \;\; s.t. \;\; \forall n \geq n_0,$$

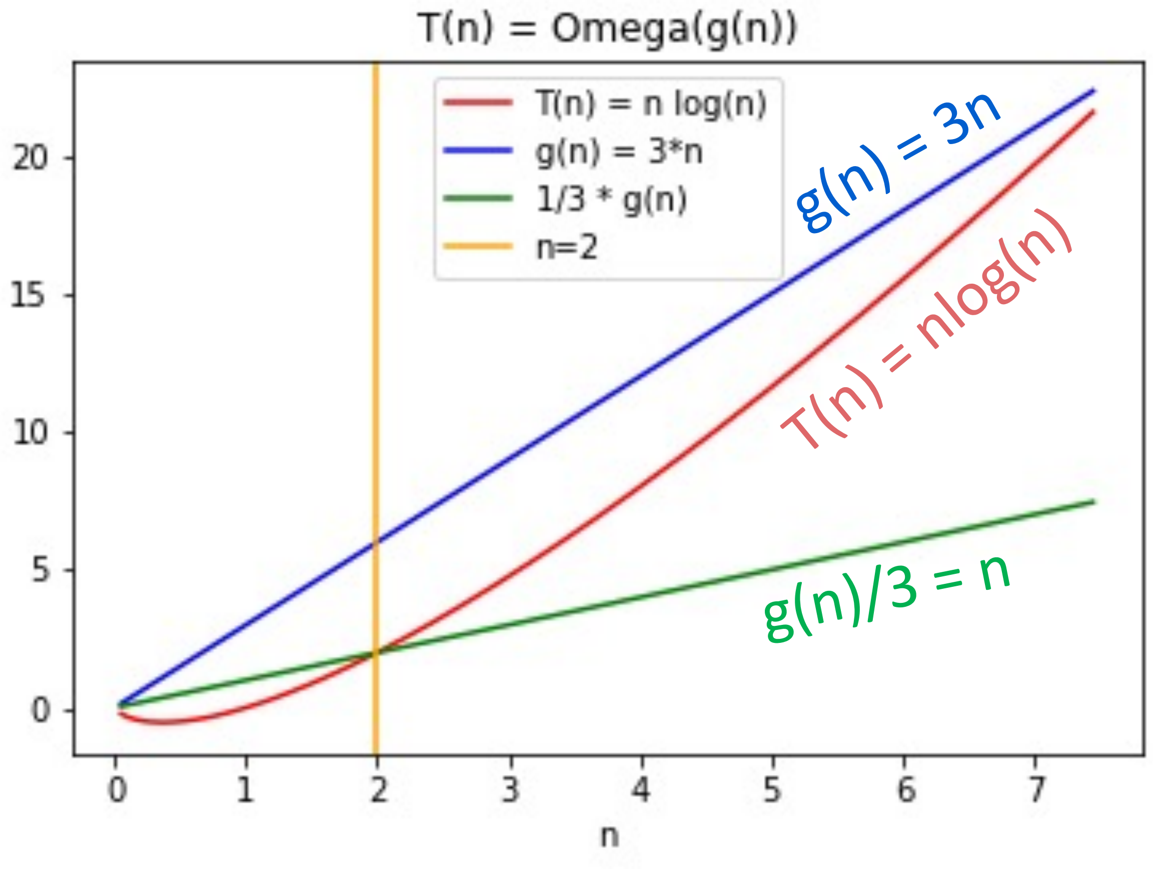
$$c \cdot g(n) \leq T(n)$$

Switched these!!

Example

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

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



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

$$\forall n \geq 2,$$

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

Θ(…) means both!

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

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

and

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

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

$\Leftrightarrow$

$\exists c > 0, n_0 \;\; s.t. \;\; \forall n \geq n_0,$

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

Non-Example:
n^2 is not $\mathrm{O}(n)$

- Proof by contradiction:
- Suppose that $n^2 = O(n)$.
- Then there is some positive c and n_0 so that:

$$\forall n \geq n_0, \qquad n^2 \leq c \cdot n$$

- Divide both sides by n:

$$\forall n \geq n_0, \qquad n \;\leq c$$

- That's not true!!! What about $\max(n_0, c+1)$?
  - Then $n \geq n_0$, but $n \; > c$.
- Contradiction!

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.

Another example: polynomials

- Say $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ is a polynomial of degree $k \geq 1$ and $a_k > 0$.

- Then:
  1. $p(n) = O\left(n^k\right)$
  2. $p(n)$ is **not** $O\left(n^{k-1}\right)$

- See the notes/references for a proof.

Try to prove it yourself first!

Siggi the Studious Stork

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** $O(2^n)$
- $\log_2(n) = \Omega(\ln(n))$
- $\log_2(n) = \Theta(2^{\log\log(n)})$

Work through these on your own! Also look at the examples in the reading!

Siggi the Studious Stork

Brainteaser

- Are there functions f, g so that NEITHER $f = O(g)$ nor $f = \Omega(g)$?

Ollie the Over-achieving Ostrich

Recap: Asymptotic Notation

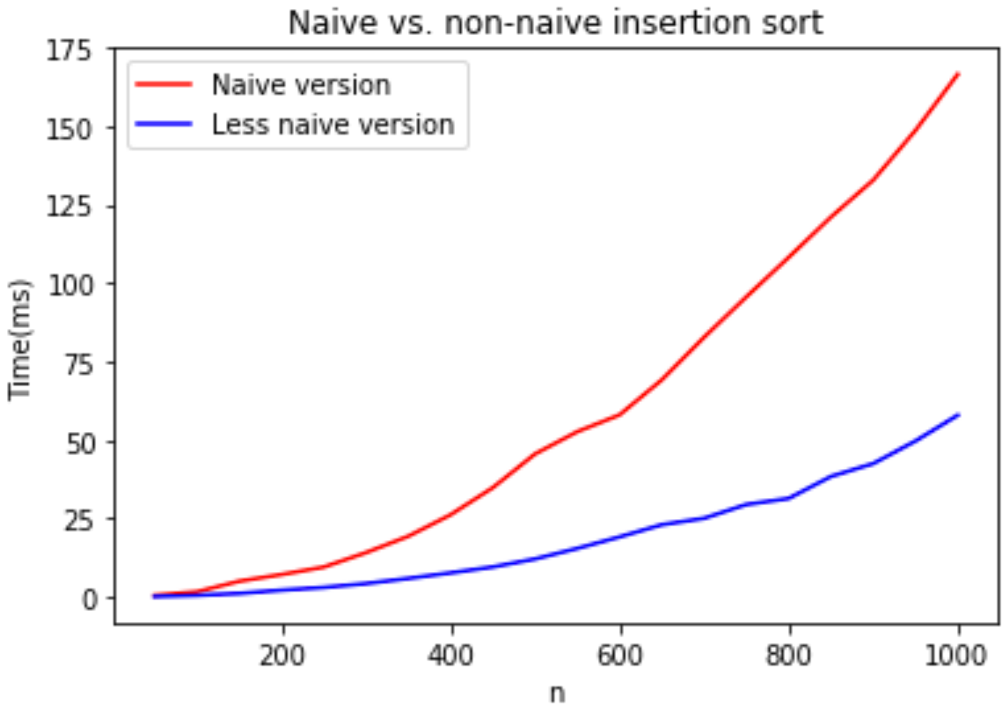
- This makes both Plucky and Lucky happy.
  - **Plucky the Pedantic Penguin** is happy because there is a precise definition.
  - **Lucky the Lackadaisical Lemur** is happy because we don’t have to pay close attention to all those pesky constant factors.

- But we should always be careful not to abuse it.

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

Back to Insertion Sort

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

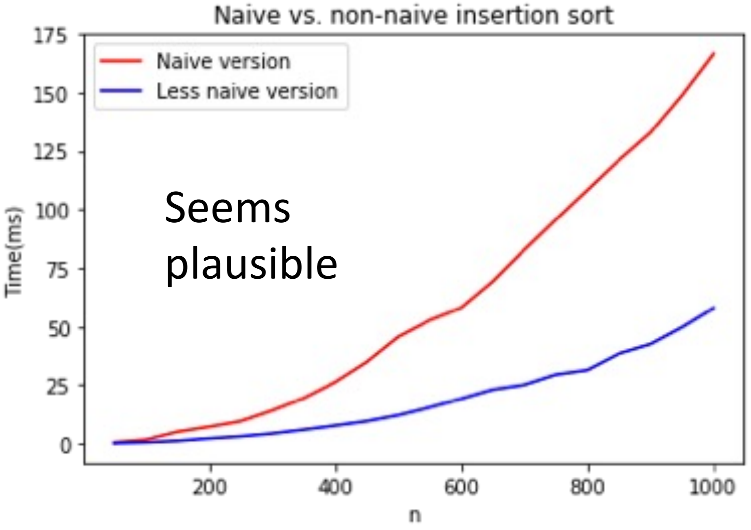


Insertion Sort: running time

- Operation count was:
    - $2n^2 - n - 1$ variable assignments
    - $2n^2 - n - 1$ increments/decrements
    - $2n^2 - 4n + 1$ comparisons
    - …

- The running time is $O(n^2)$

Go back to the pseudocode and convince yourself of this!



Seems plausible

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?

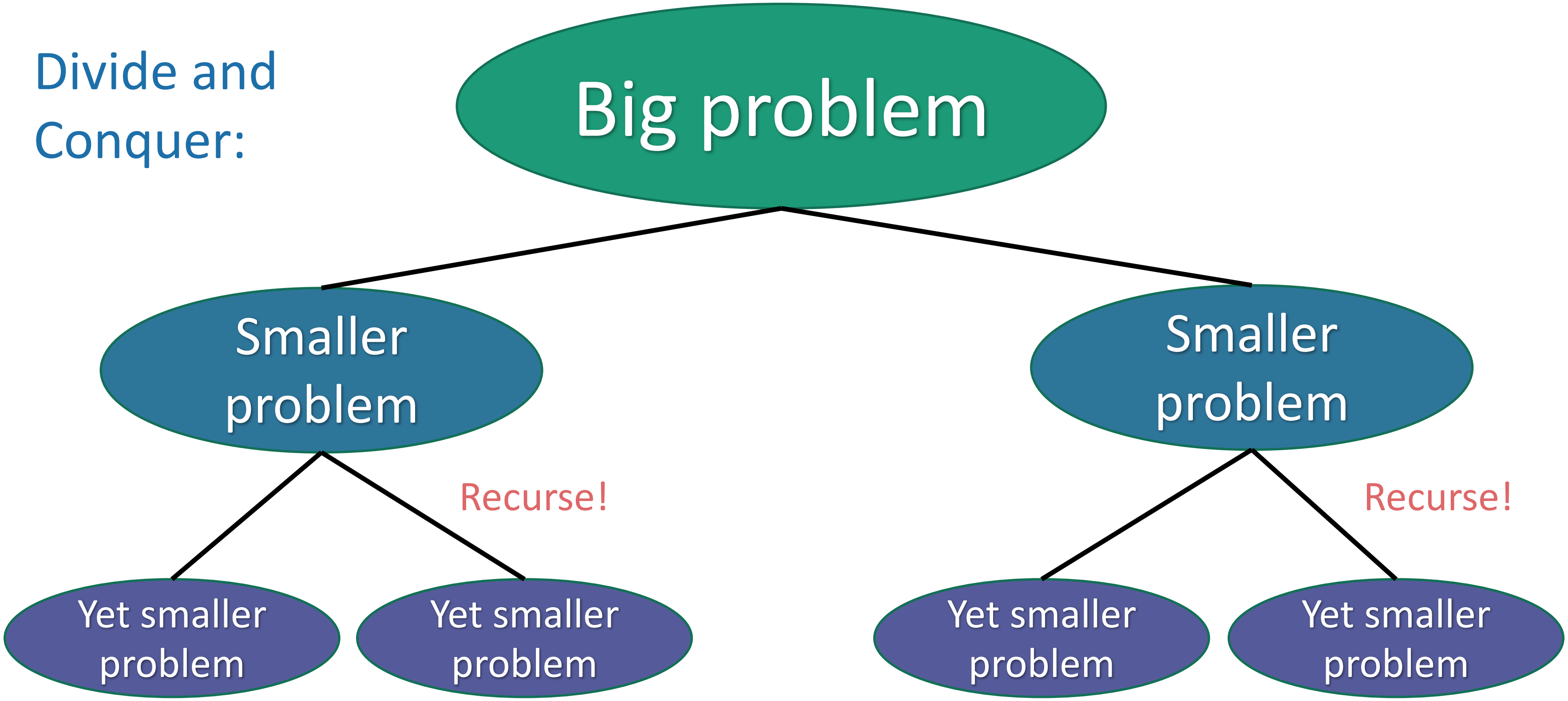
The Plan

- InsertionSort recap
- Worst-case analyisis
  - Back to InsertionSort: Does it work?
- Asymptotic Analysis
  - Back to InsertionSort: Is it fast?
- MergeSort
  - Does it work?
  - Is it fast?

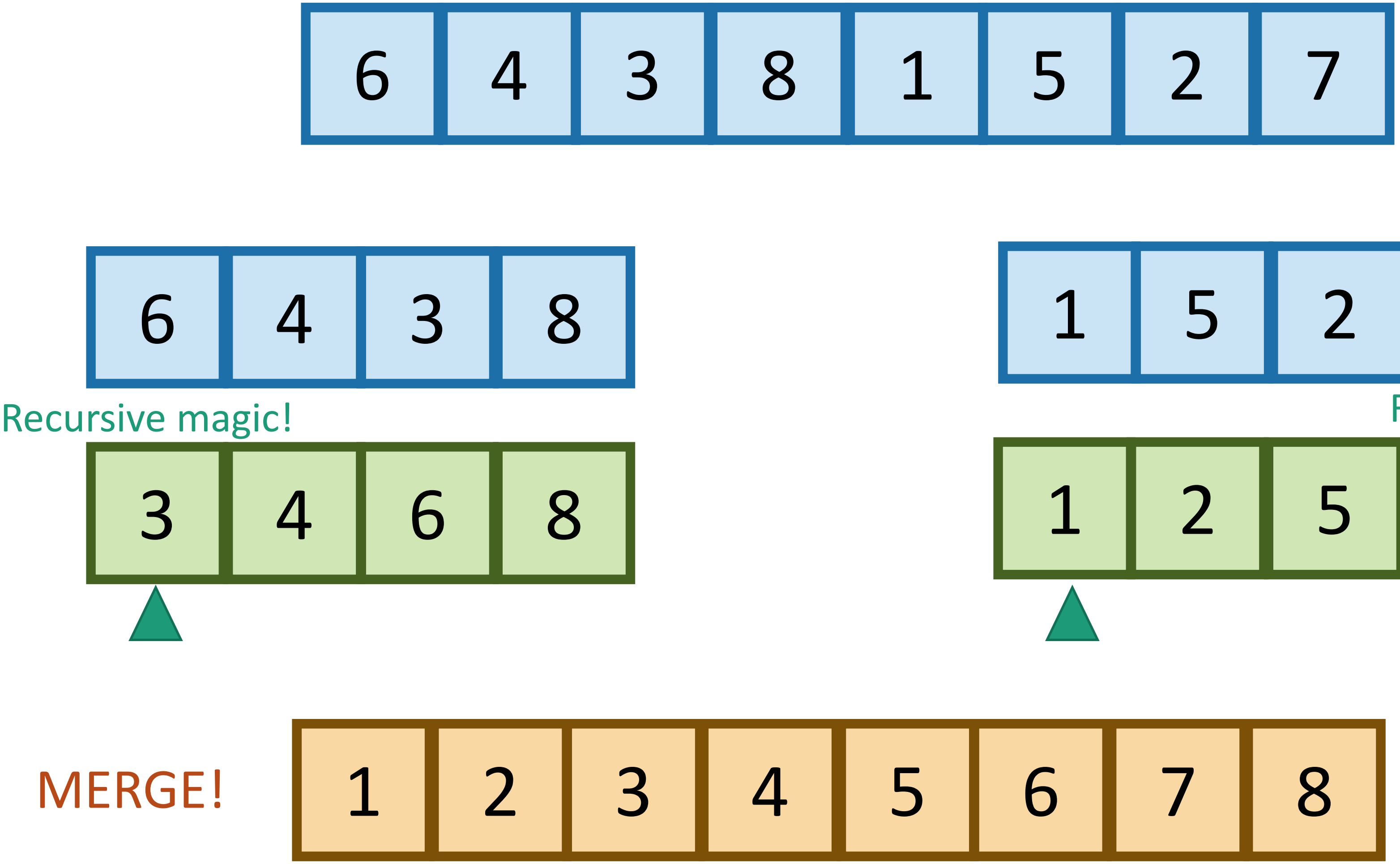
Can we do better?

- MergeSort: a divide-and-conquer approach
- Recall:

Divide and Conquer:



MergeSort



6 4 3 8 1 5 2 7

6 4 3 8

1 5 2 7

Recursive magic!

3 4 6 8

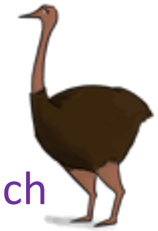
Recursive magic!

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

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

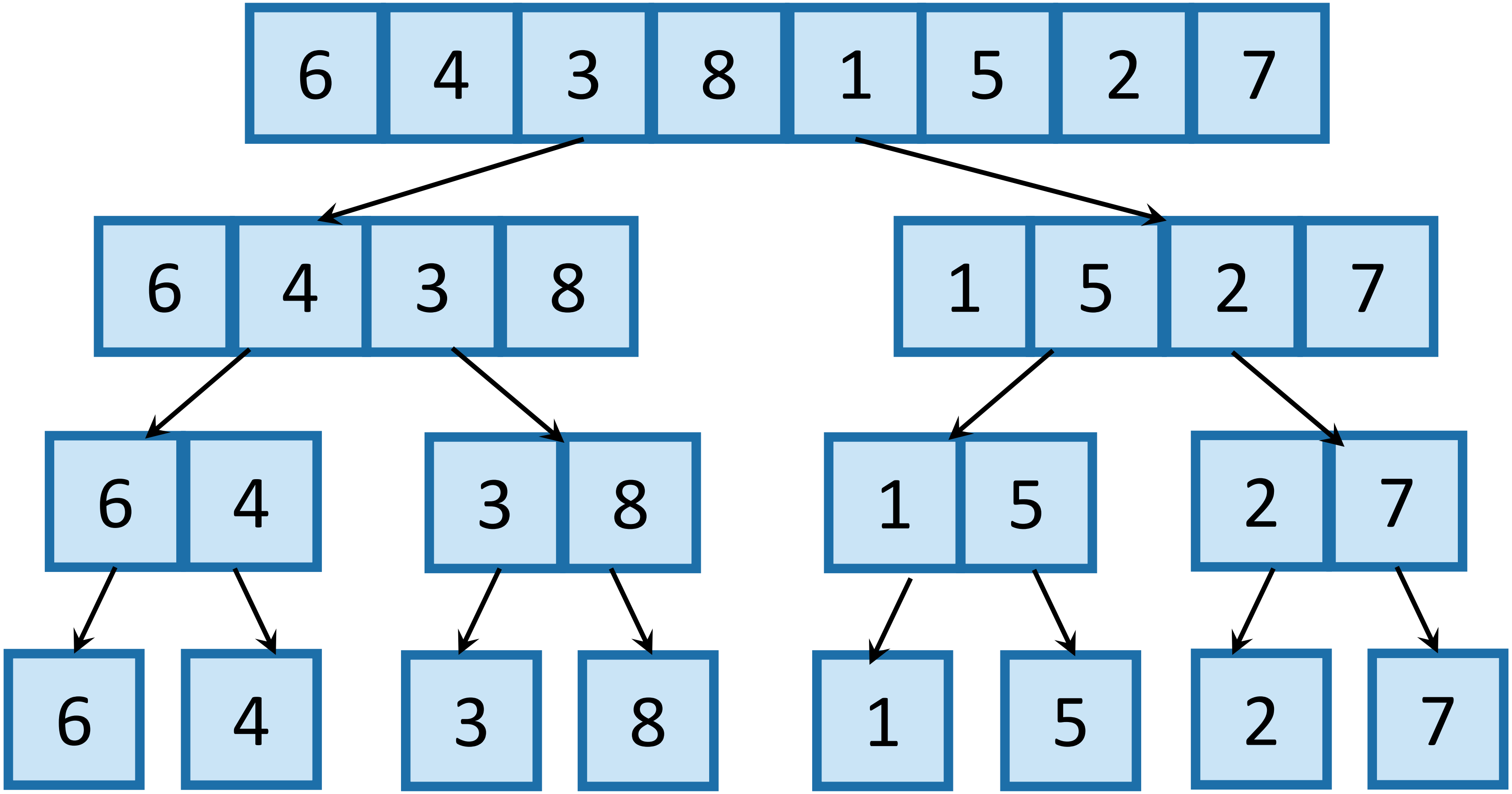
See Lecture 2 IPython notebook for MergeSort Python Code.

MergeSort Pseudocode

```
MERGESORT(A):
    • n = length(A)
    • if $n \leq 1$:                    If A has length 1,
                                        It is already sorted!
        • return A
    • 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?

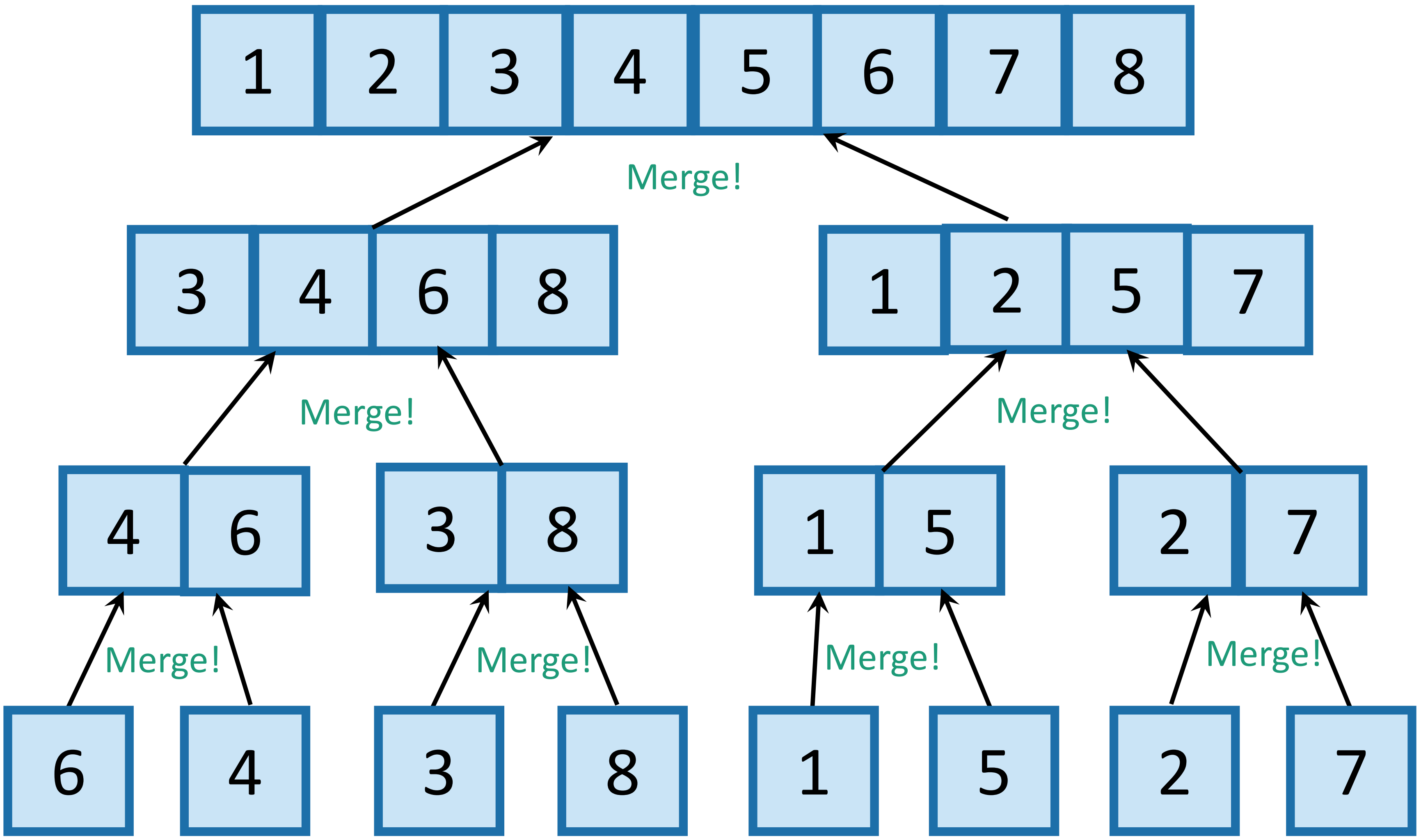
First, recursively break up the array all the way down to the base cases



This array of length 1 is sorted!

Then, merge them all back up!

Sorted sequence!



1 2 3 4 5 6 7 8

Merge!

3 4 6 8

1 2 5 7

Merge!

Merge!

4 6

3 8

1 5

2 7

Merge!

Merge!

Merge!

Merge!

6

4

3

8

1

5

2

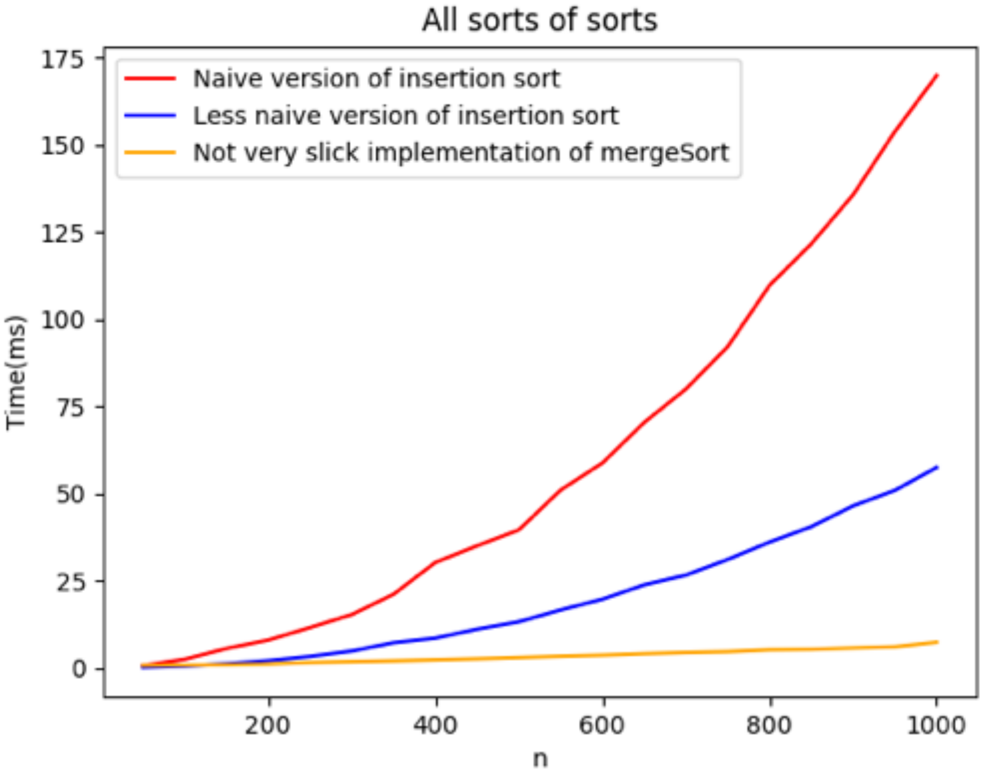
7

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

- Yet another job for…

Proof By Induction!

Work this out! There's a skipped slide with an outline to help you get started.

Assume that n is a power of 2 for convenience.

It’s fast

CLAIM:

MergeSort runs in time $O(n\log(n))$

- Proof coming soon.
- But first, how does this compare to InsertionSort?
  - Recall InsertionSort ran in time O(n^2).

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

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.

$32, \underbrace{16, 8, 4, 2, 1}_{\text{Halve 5 times}} \Rightarrow$ log(32) = 5

$64, \underbrace{32, 16, 8, 4, 2, 1}_{\text{Halve 6 times}} \Rightarrow$ log(64) = 6

log(128) = 7

log(256) = 8

log(512) = 9

….

log(# **particles in the universe**) < 280

- log(n) grows very slowly!

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

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

Punchline: A running time of O(n log n) is a lot better than $O(n^2)$!

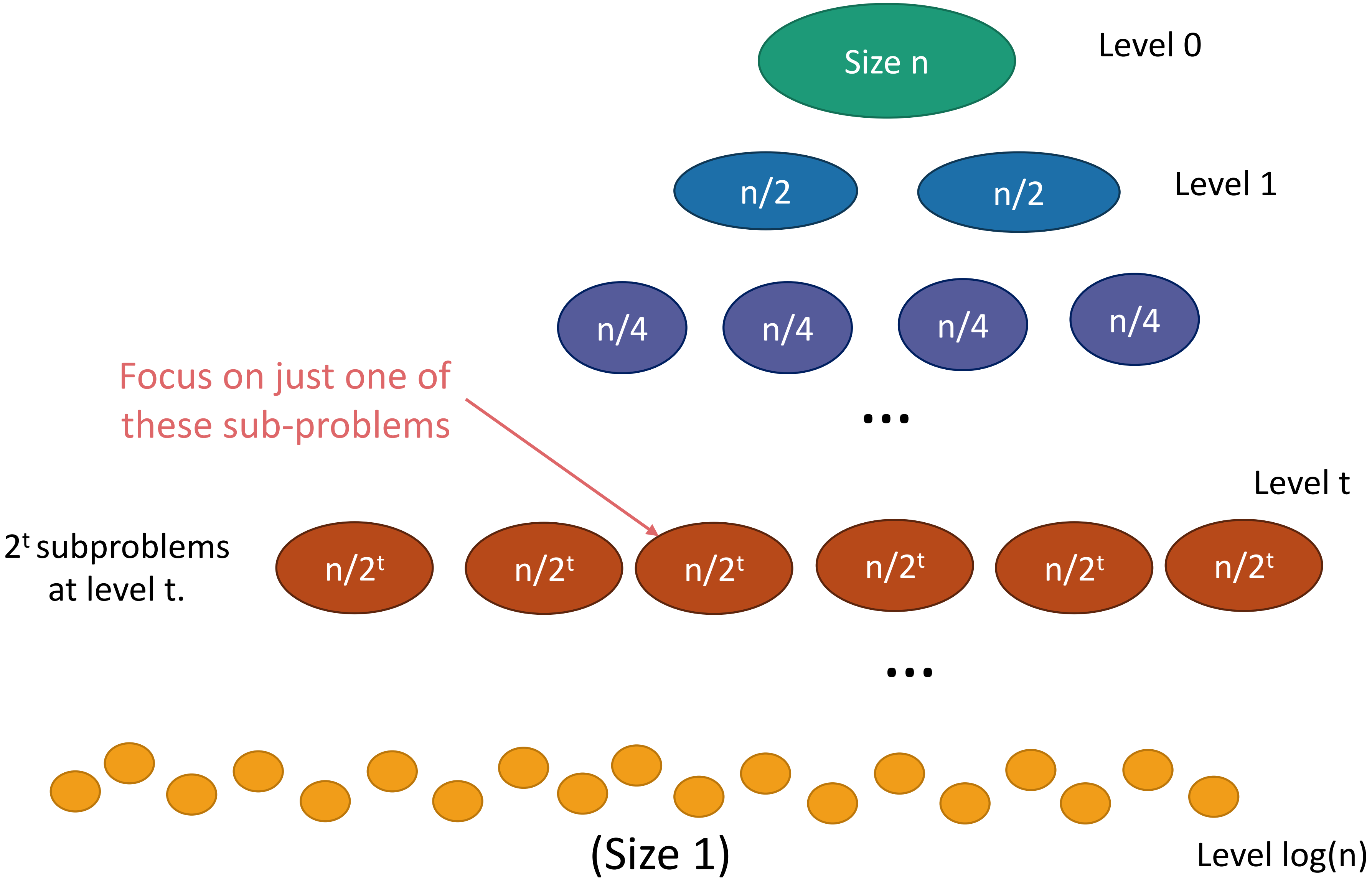
Assume that n is a power of 2 for convenience.

Now let's prove the claim

CLAIM:

MergeSort runs in time $O(n \log(n))$

Let’s prove the claim



Size n

Level 0

n/2

n/2

Level 1

n/4

n/4

n/4

n/4

…

Focus on just one of these sub-problems

Level t

2^t subproblems at level t.

$n/2^t$

$n/2^t$

$n/2^t$

$n/2^t$

$n/2^t$

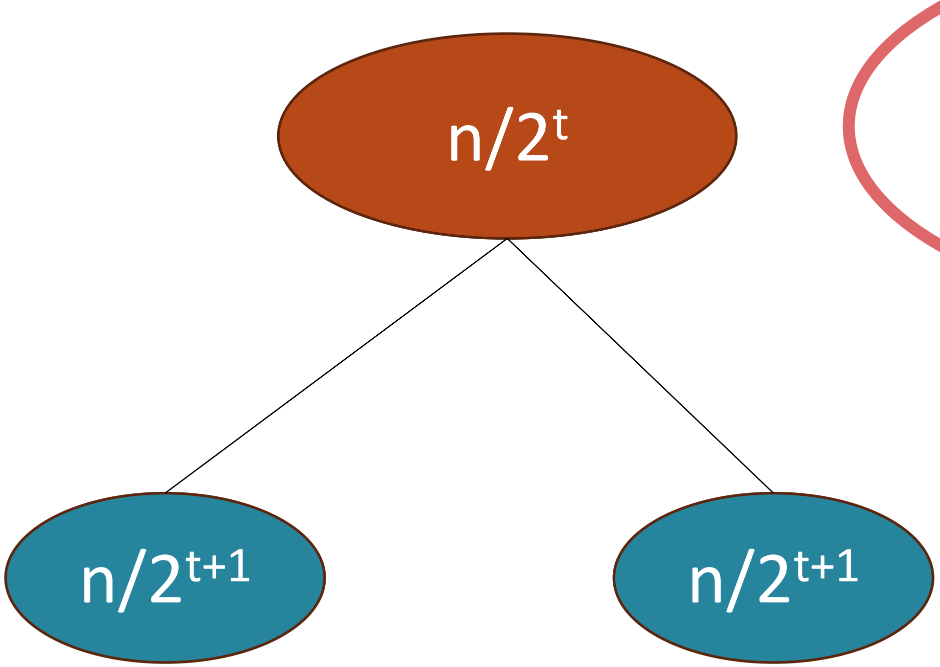
$n/2^t$

…

(Size 1)

Level log(n)

How much work in this sub-problem?



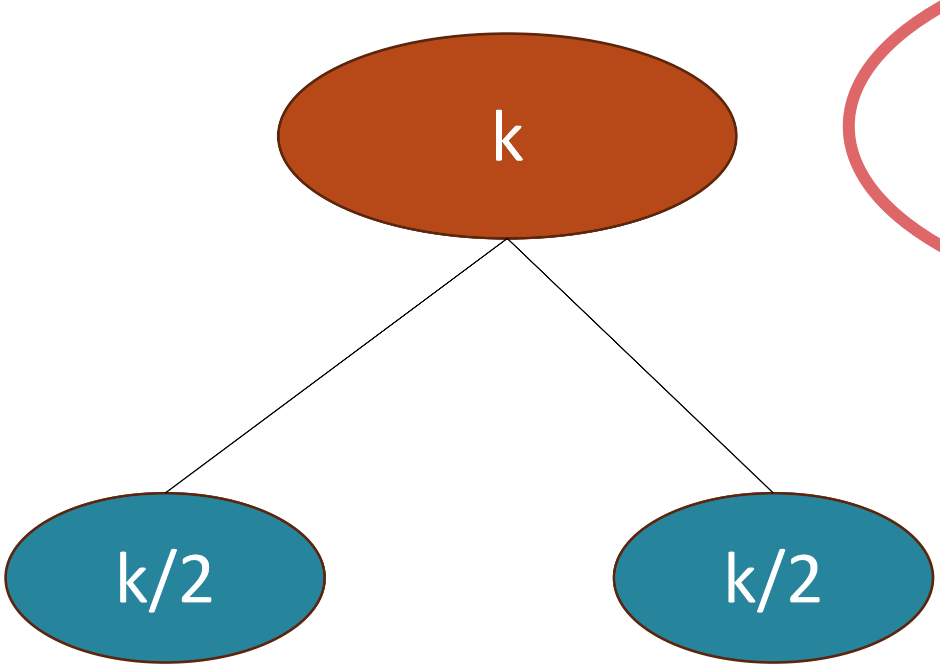
Time spent MERGE-ing the two subproblems

+

Time spent within the two sub-problems

How much work in this sub-problem?

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

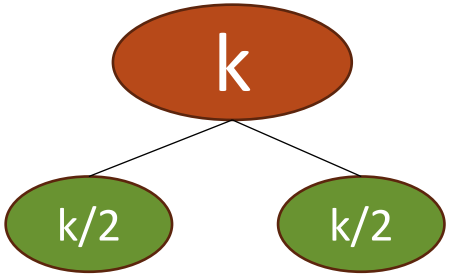


Time spent MERGE-ing the two subproblems

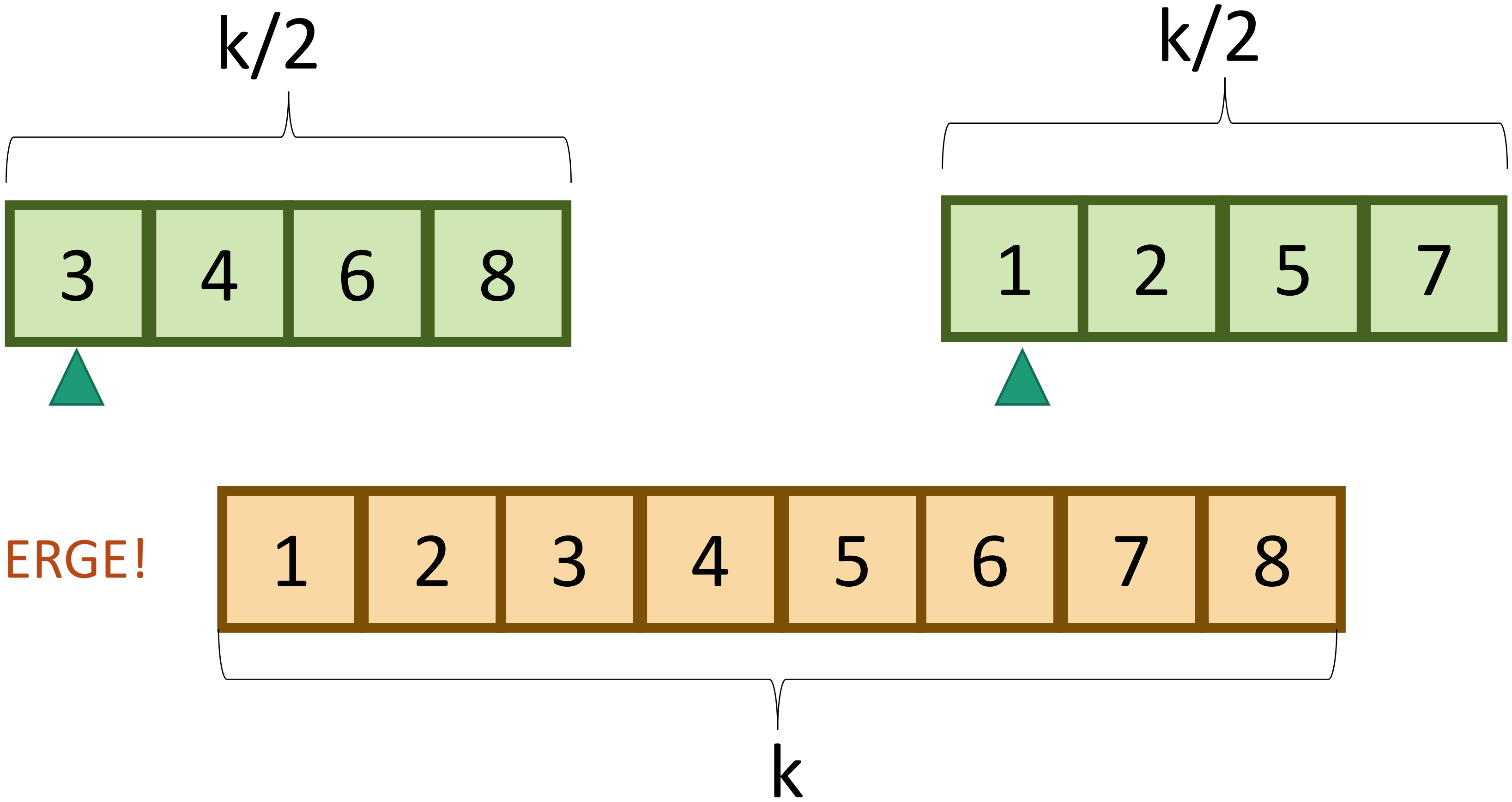
+

Time spent within the two sub-problems

How long does it take to MERGE?



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



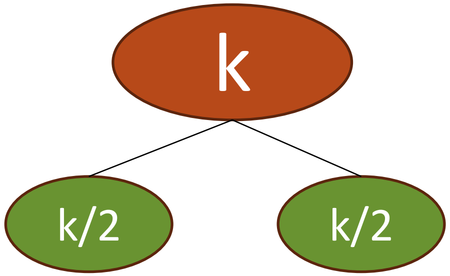
3 4 6 8

1 2 5 7

MERGE!

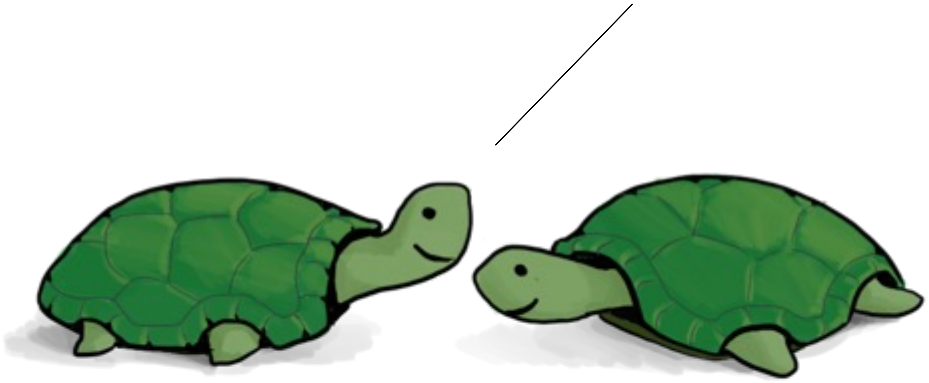
1 2 3 4 5 6 7 8

How long does it take to MERGE?



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

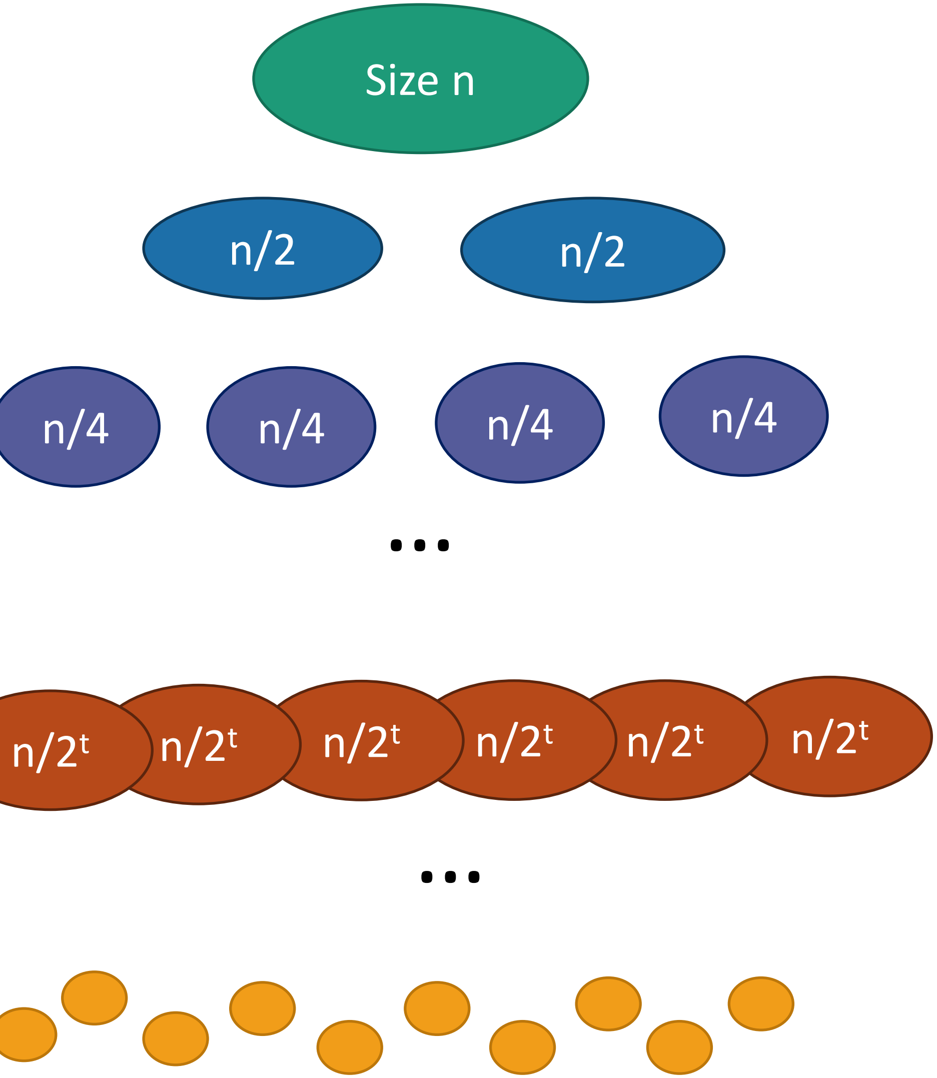
How long does it take to run MERGE on two lists of size k/2?



Think-Pair-Share Terrapins (if time)

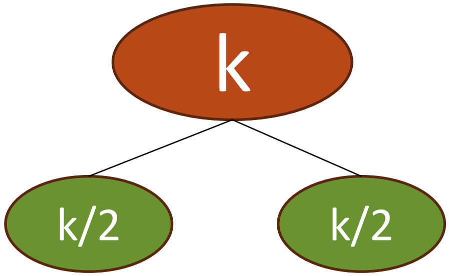
Answer: It takes time O(k), since we just walk across the list once.

Recursion tree

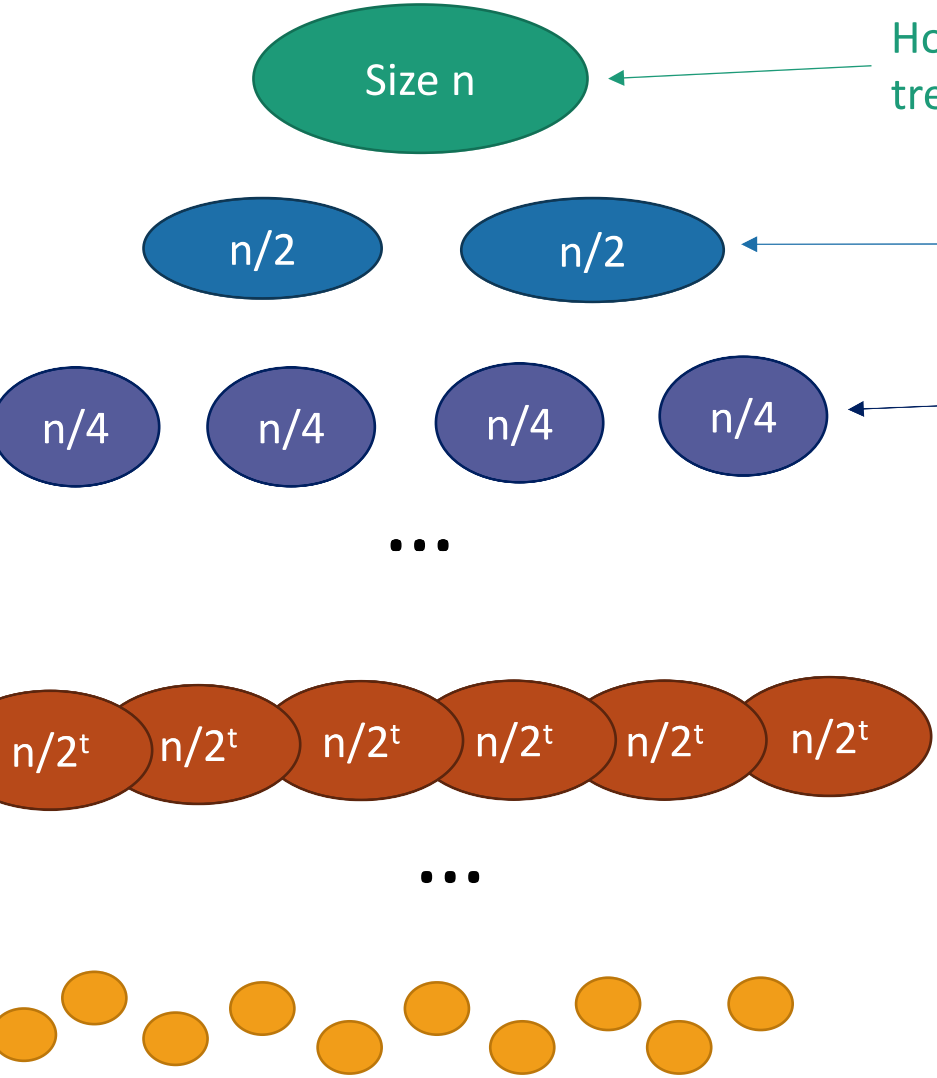


(Size 1)

There are O(k) operations done at this node.



Recursion tree



How many operations are done at this level of the tree? (Just MERGE-ing subproblems).

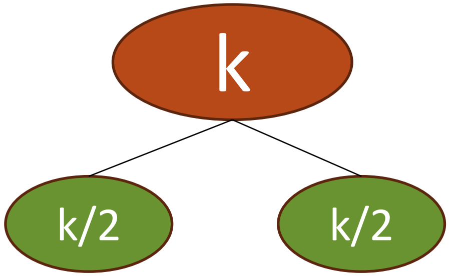
How about at this level of the tree? (between both n/2-sized problems)

This level?

This level?

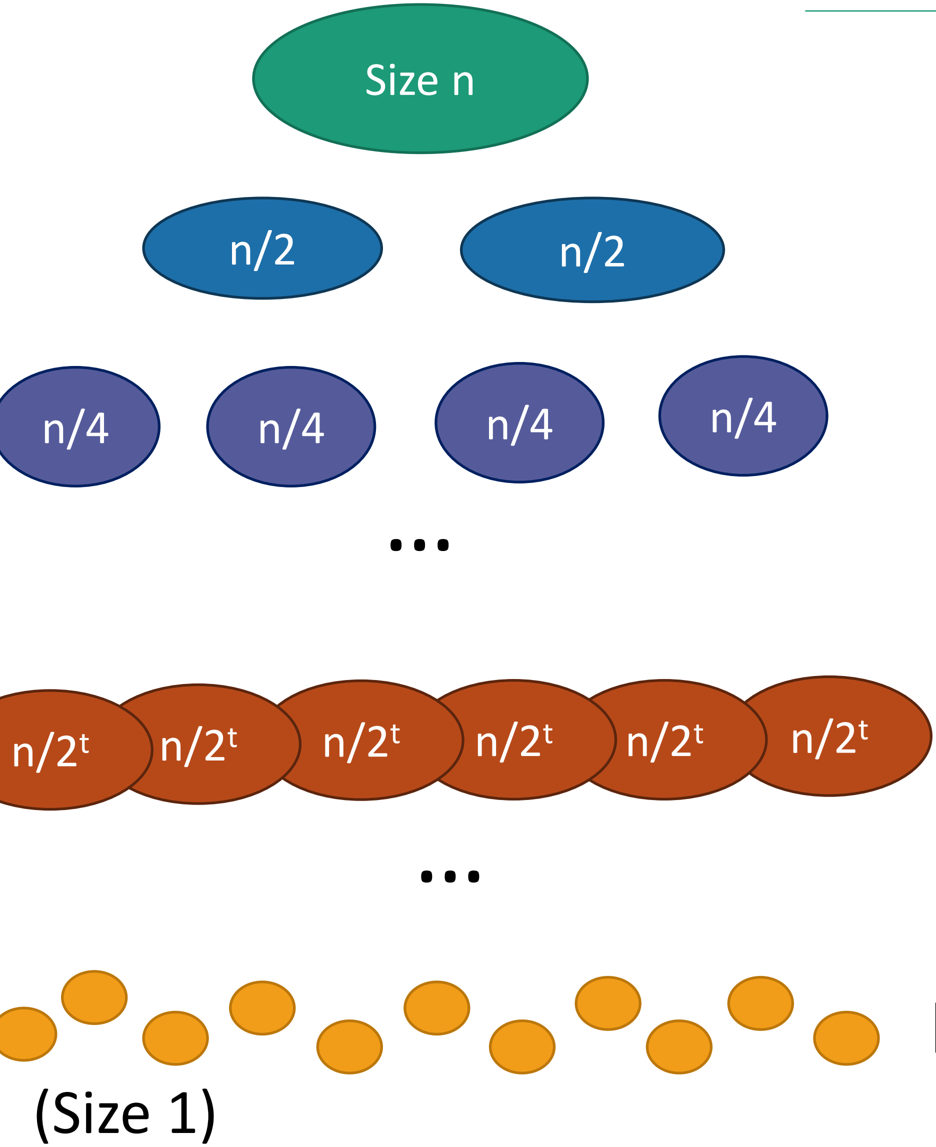
(Size 1)

There are O(k) operations done at this node.



Recursion tree

Work this out yourself!



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

Total runtime…

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

That was the claim!

What have we learned?

- MergeSort correctly sorts a list of n integers in time O(n log(n)).
- That’s (asymptotically) better than InsertionSort!

The Plan

- InsertionSort recap
- Worst-case analyisis
  - Back to InsertionSort: Does it work?
- Asymptotic Analysis
  - Back to InsertionSort: Is it fast?
- MergeSort
  - Does it work?
  - Is it fast?

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
- How do we analyze the running time of a recursive algorithm?
  - One way is to draw a recursion tree.

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)