Lecture 2

More logistics

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

Some questions from last time

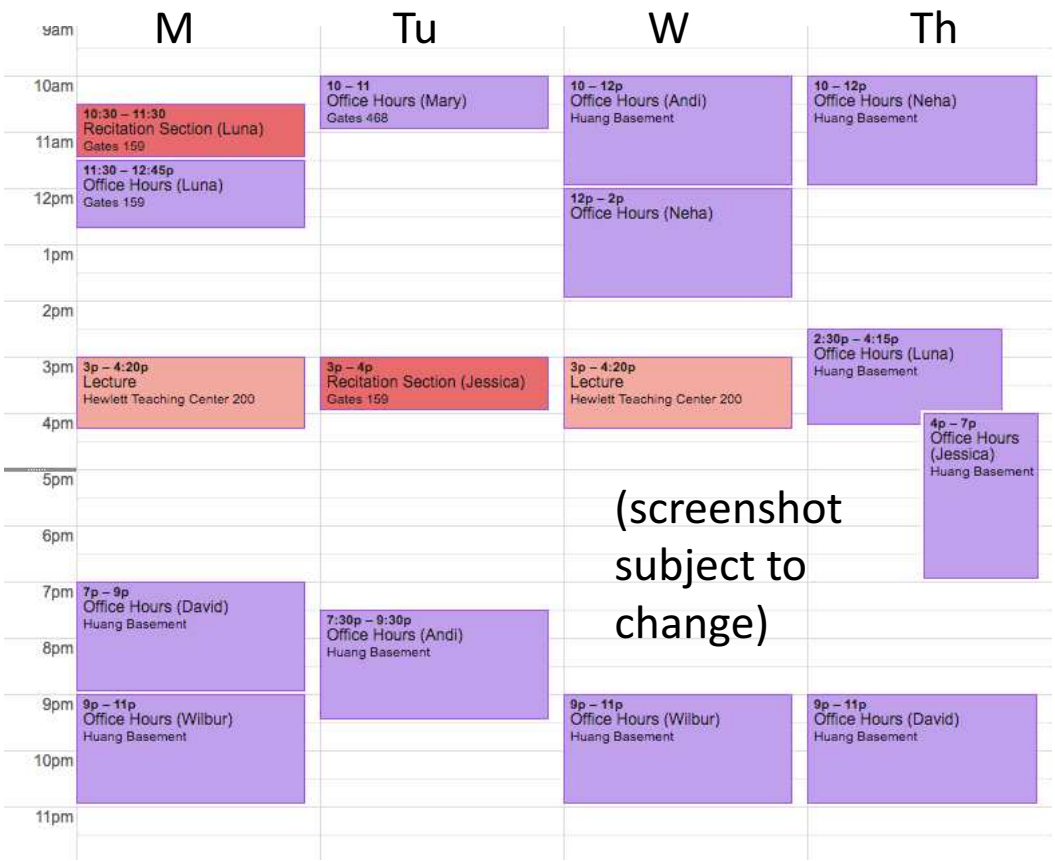
- What's the best way to take notes?
  - I will post slides ahead of time.
  - Sometimes the slides will refer to explanations on the board – I encourage you to take notes for that stuff.

- What's with the waitlist?
  - We got another TA! (Danny Wright). Enrollment will be opening up (**still limited**, but everyone who was on the waitlist at 8am this morning will get a spot).

Logistics some more!

Sign up for piazza

- As before: http://web.stanford.edu/class/cs161/
- There will be office hours! And recitation sections!



(screenshot subject to change)

The deal with sections:

- Held on Mondays and Tuesdays: go over content from previous week, with example problems!
- Don't need to sign up, go to any/either section!
- Problems/solutions from both sections will be posted!

Homework! We're gonna have it!

- Assignment will be posted FRIDAY. (3pm)
- Due the next FRIDAY. (3pm)
- See the COURSE WEBSITE for HW guidelines.
- Solutions most the following MONDAY.

Monday	Tuesday	Wednesday	Thursday	Friday
				HW ASSIGNED
				HW DUE
SOLUTIONS RELEASED				HW RETURNED (target)

Last time

Philosophy

- Algorithms are awesome and powerful!
- Algorithm designer's question: Can I do better?
- Interplay between rigor and intuition.

Technical content

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

Plucky the pedantic penguin

Lucky the lackadaisical lemur.

Today

- Sorting
- Return of divide-and-conquer with Merge Sort
- Skills:
  - Analyzing correctness of iterative and recursive algorithms.
  - Analyzing running time of recursive algorithms.

- 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

We're going to go through this in some detail – it's good practice!

- Say we want to sort:

6	4	3	8	5

- Insert items one at a time.

- How would we actually implement this?

Insertion sort example…

6	4	3	8	5

Start with the second element (the first element is sorted within itself…)

4	6	3	8	5

Pull “4” back until it’s in the right place.

4	6	3	8	5

Now look at “3”

3	4	6	8	5

Pull “3” back until it’s in the right place.

3	4	6	8	5

“8” is good…look at 5

3	4	6	5	8

(then fix 5 and we’re done)

Insertion sort pseudocode

Go one-at-a-time until things are in the right place.

Algorithm 1: $\textsc{InsertionSort}(A)$

```
for i = 2 → length(A) do
    key ← A[i];
    j ← i − 1;
    while j > 0 and A[j] > key do
        A[j + 1] ← A[j];
        j ← j − 1;
    A[j + 1] ← key;
```

Lucky the lackadaisical lemur

Plucky the pedantic penguin

- (Discussion on board)

Insertion sort: running time

Algorithm 1: INSERTIONSORT(A)

```
for i = 2 → length(A) do
    key ← A[i];
    j ← i − 1;
    while j > 0 and A[j] > key do
        A[j + 1] ← A[j];
        j ← j − 1;
    A[j + 1] ← key;
```

n-1 iterations of the outer loop

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

Running time is about n^2

Insertion sort: correctness

- Maintain a loop invariant.

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

- Initialization: the loop invariant holds before the first iteration.
- Maintenance: If it is true before the t'th iteration, it will be true before the (t+1)'st iteration
- Termination: It is useful to know that the loop invariant is true at the end of the last iteration.

(This is proof-of-correctness by induction)

Insertion sort: correctness

- Loop invariant: At the start of the t'th iteration (of the outer loop), the first t elements of the array are sorted.
- Initialization: At the start of the first iteration, the first element of the array is sorted. ✓
- Maintenance: By construction, the point of the t'th iteration is to put the (t+1)'st thing in the right place.

- Termination: At the start of the (len(A) + 1)'st iteration (aka, at the end of the algorithm), the first len(A) items are sorted. ✓

To summarize

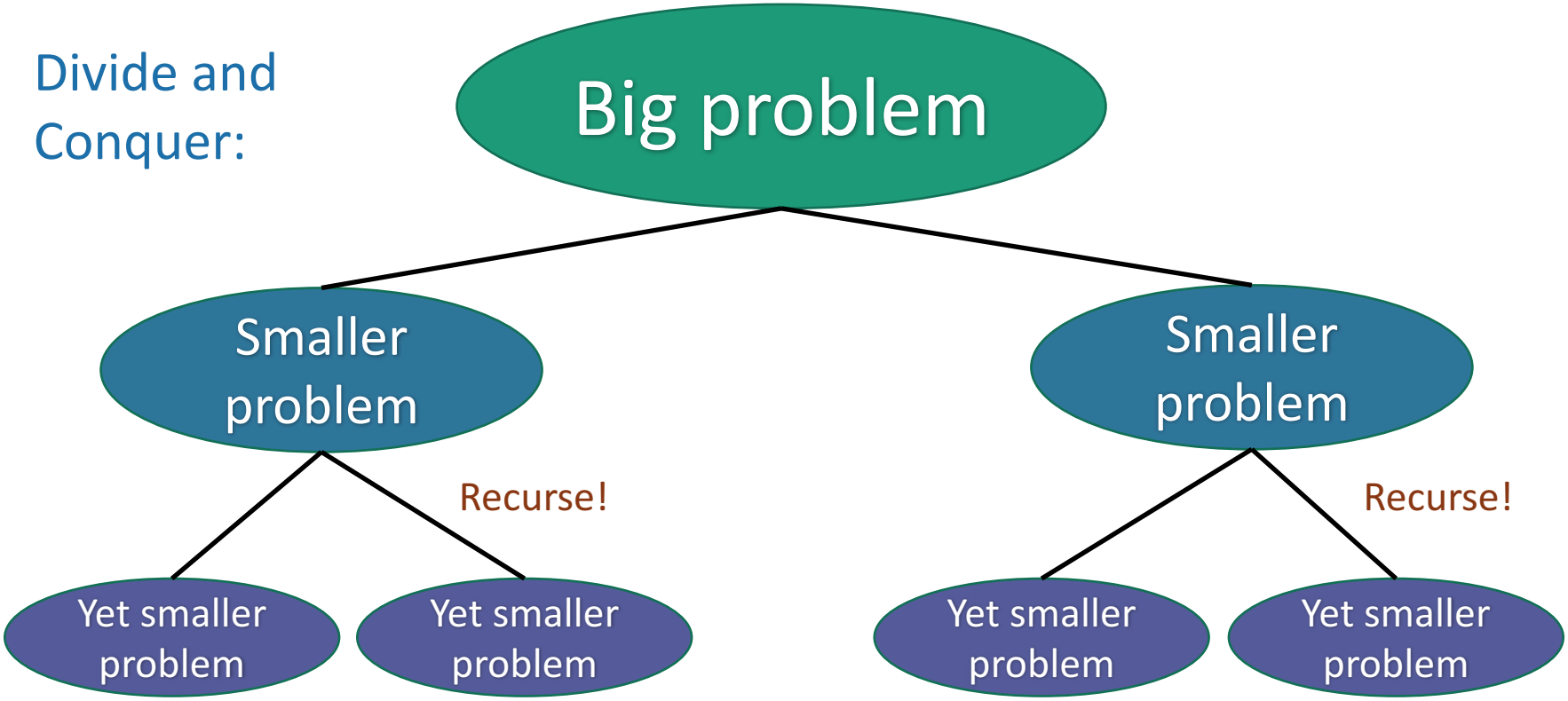
InsertionSort is an algorithm that correctly sorts an arbitrary n-element array in time about n^2.

Can we do better?

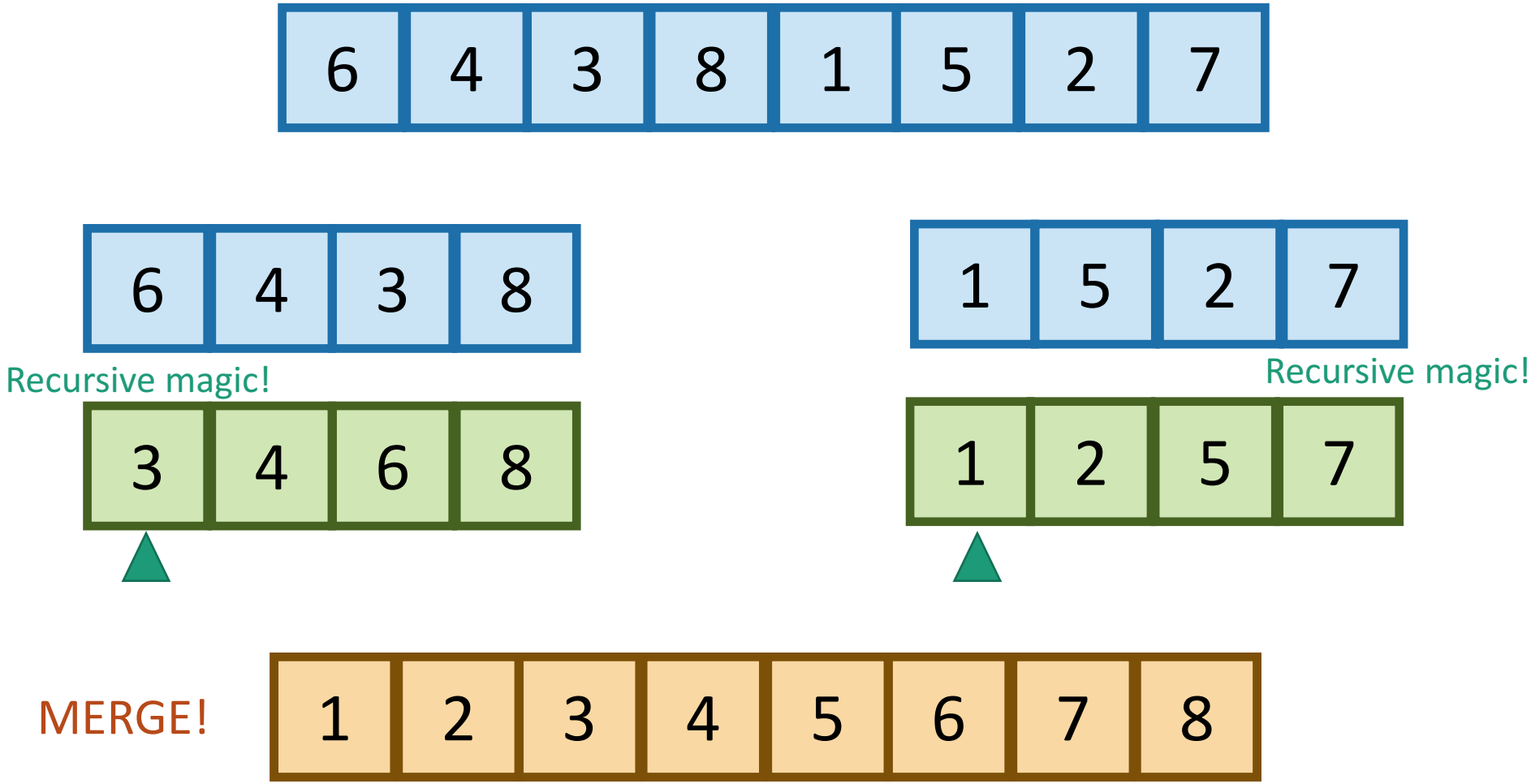
Can we do better?

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

Divide and Conquer:



MergeSort



(MERGE pseudocode + analysis on board)

(A good exercise: how would I do this in place?)

MergeSort Pseudocode

MERGESORT(A):

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

If A has length 1,
It is already sorted!

- L = MERGESORT(A[1 : n/2])

Sort the left half

- R = MERGESORT(A[n/2+1 : n])

Sort the right half

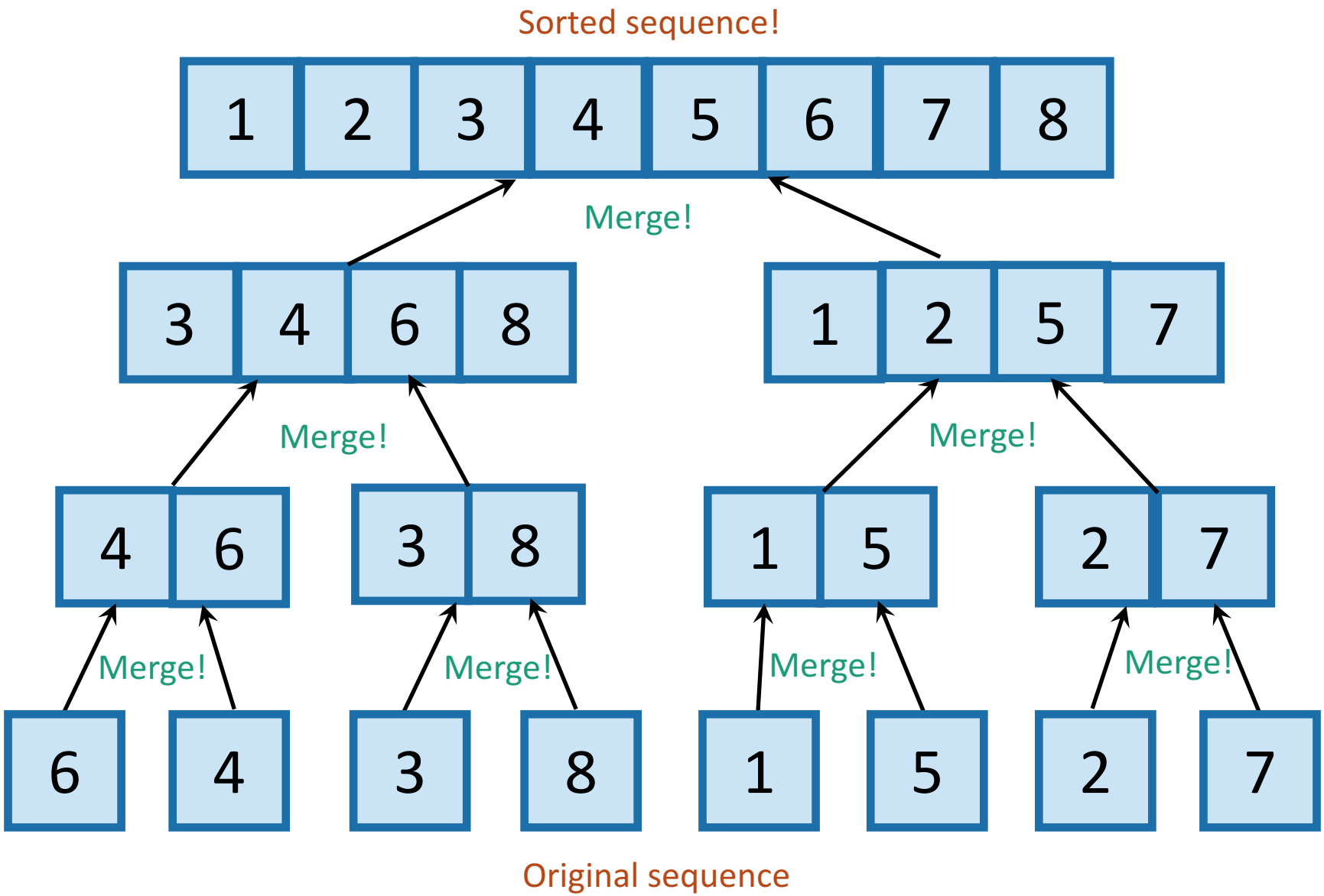
- **return** MERGE(L,R)

Merge the two halves

What actually happens when we run MergeSort?

- (on board)

Schematic of recursive calls



Two questions

- Does this work?
- Is it fast?

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

Not technically a "loop invariant," but a "recursion invariant," that should hold at the beginning of every recursive call.

- Invariant:

"In every recursive call,

MERGESORT returns a sorted array."

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

The maintenance step needs more details!! Why is this statement true?

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

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

CLAIM:

MERGESORT requires at most $6n \log(n) + 6n$ operations to sort n numbers.

Before we see why…

How does that compare to the $\approx n^2$ operations of INSERTIONSORT?

n log(n) vs n^2

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

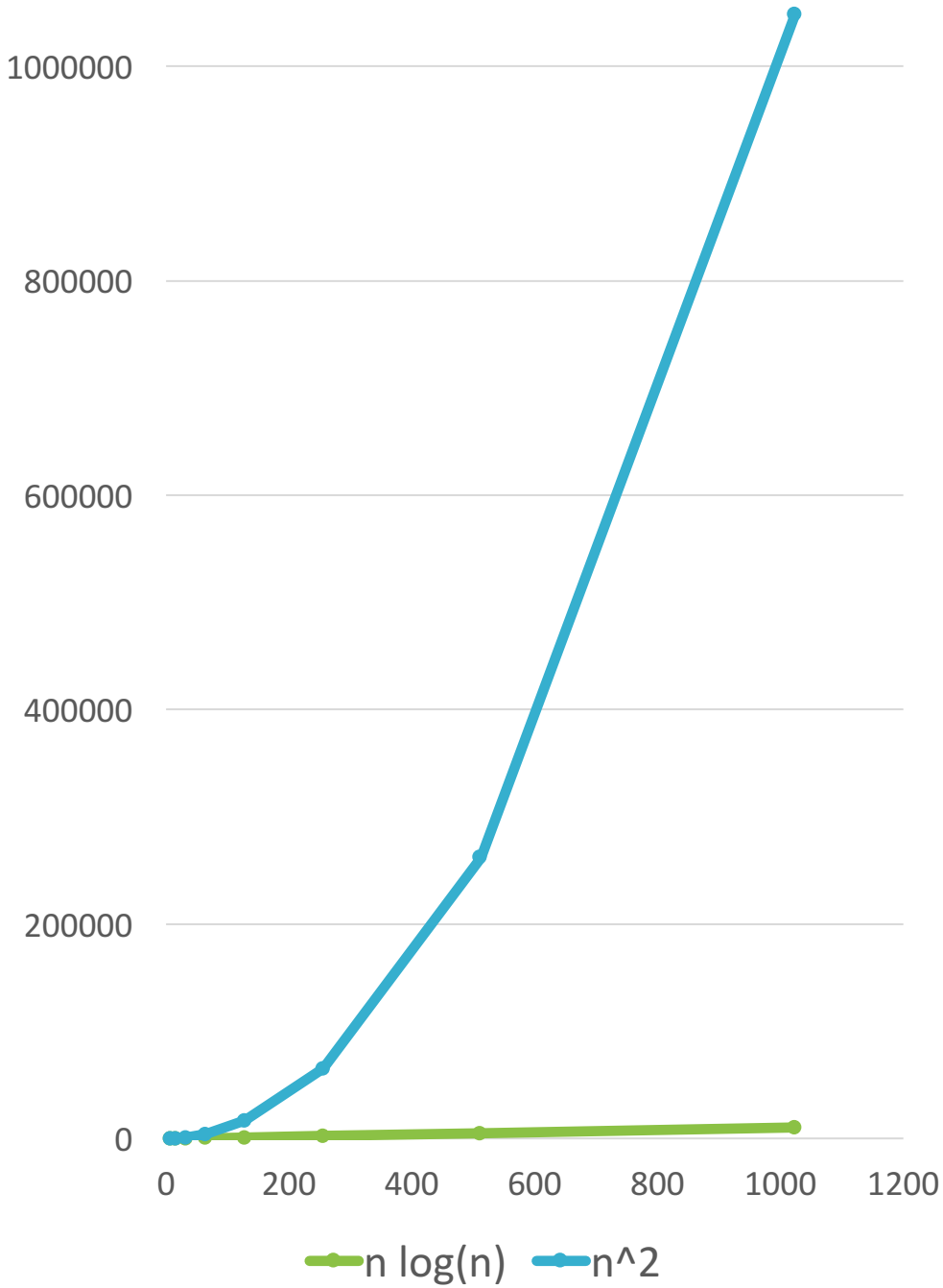
log(128) = 7
log(256) = 8
log(512) = 9
.
.
.
log(**number of particles in the universe**) < 280

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

n log(n) vs n^2

continued

n	n log(n)	n^2
8	24	64
16	64	256
32	160	1024
64	384	4096
128	896	16384
256	2048	65536
512	4608	262144
1024	10240	1048576



It's fast!

CLAIM:

MERGESORT requires at most $6n \log(n) + 6n$ operations to sort n numbers.

As n grows, that's much faster than the $\approx n^2$ operations of INSERTIONSORT!

Analysis

T(n) = time to run MERGESORT on a list of size n

This is called a recurrence relation: it describes the running time of a problem of size n in terms of the running time of smaller problems.

$$T(n) = T(n/2) + T(n/2) + T(\text{MERGE}) = 2T(n/2) + 6n$$

T(MERGE two lists of size n/2) is the time to do:

- 3 variable assignments (counters ← 1)
- n comparisons
- n more assignments
- 2n counter increments

So that's

2T(assign) + n T(compare) + n T(assign) + 2n T(increment)

or 4n + 2 operations

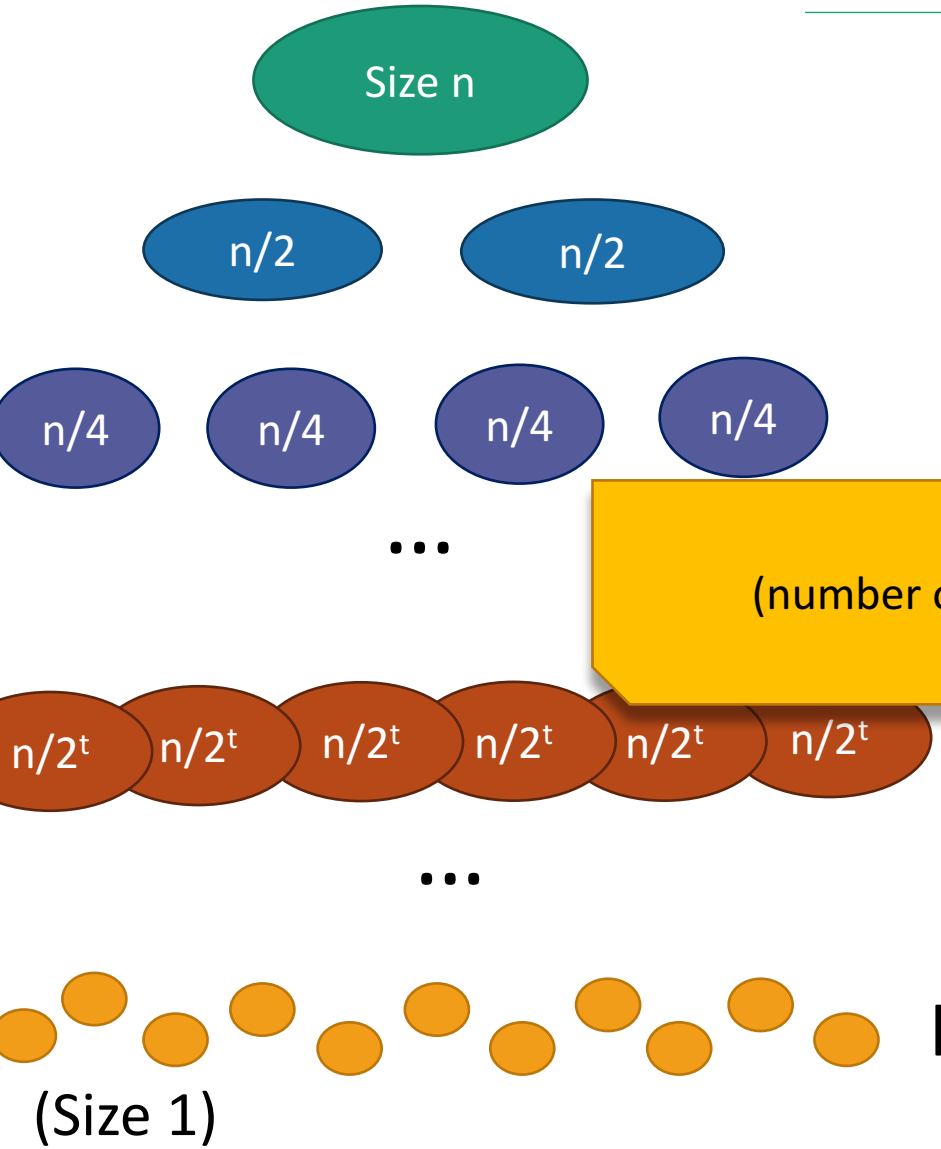
Let's say T(MERGE of size n/2) $\leq$ 6n operations

Plucky the pedantic penguin

Lucky the lackadaisical lemur

We will see later how to analyze recurrence relations like these automagically…but today we'll do it from first principles.

Recursion tree



Level	# problems	Size of each problem	Amount of work at this level (just MERGEing)
0	1	n	6n
1	2	n/2	6n
2	4	n/4	6n
…			
t	2^t	$n/2^t$	6n
…			
log(n)	n	1	6n

Amount of work at a level:
(number of problems) × 6 × (size of problem)
(explanation on board)

Total runtime…

- 6n steps per level, at every level
- log(n) + 1 levels
- 6n log(n) + 6n 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 T(MERGE) < 6n?
  - 2 + 4n < 6n is a loose bound.
  - Different operations don't take the same amount of time.

Today

- Sorting
- Return of divide-and-conquer with Merge Sort
- Skills:
  - Analyzing correctness of iterative and recursive algorithms.
  - Analyzing running time of recursive algorithms.

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

NOTE!!! WE ACTUALLY STOPPED HERE IN CLASS ON WEDNESDAY!

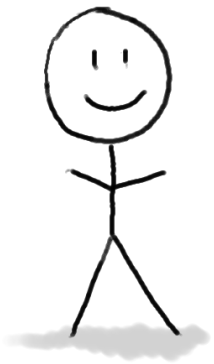
WE'LL PICK UP AFTER THIS NEXT TIME…

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



Here is my algorithm!

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

Algorithm designer

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

Big-O notation

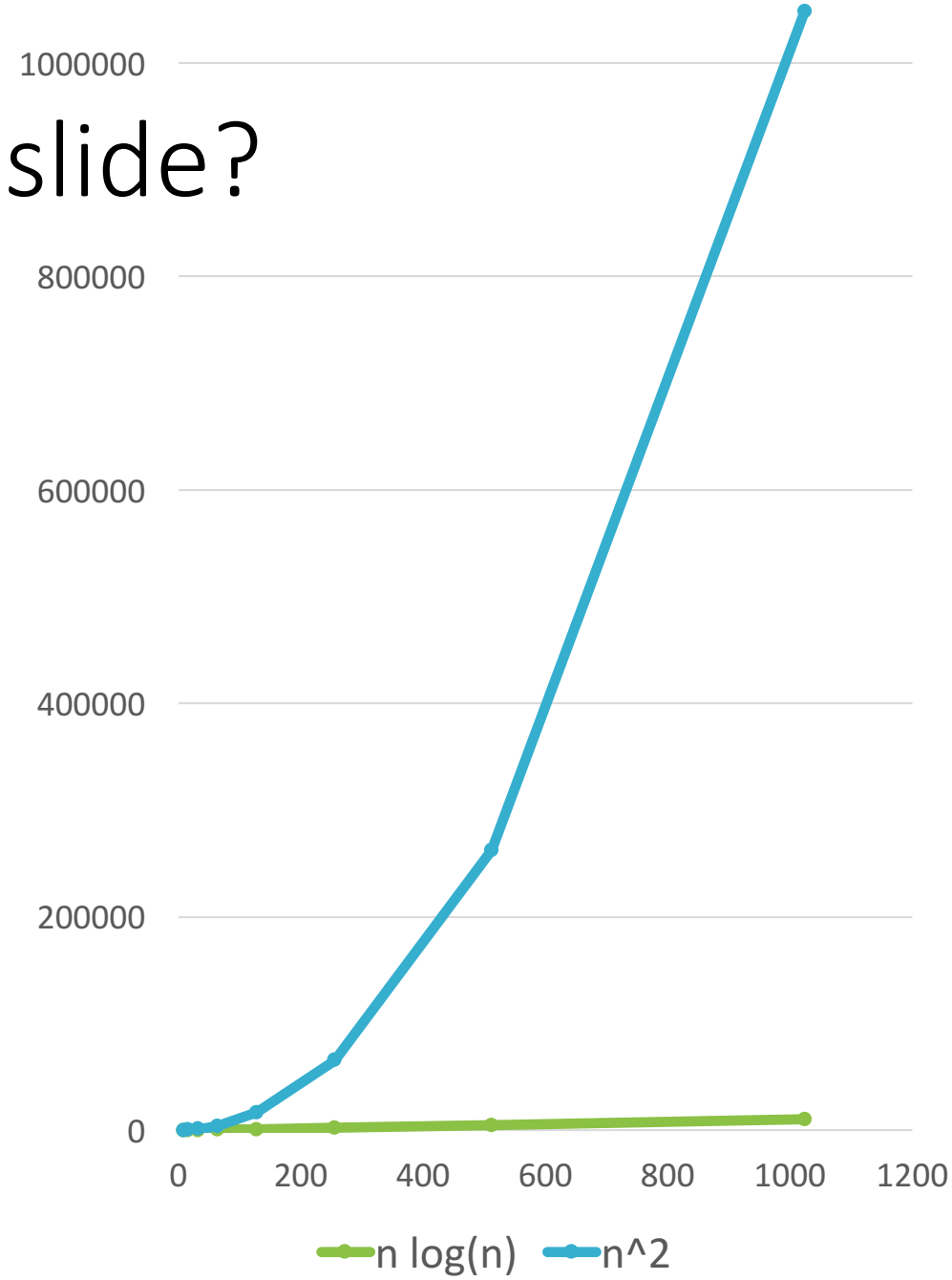
How long does an operation take? Why are we being so sloppy about that "6"?

- 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.

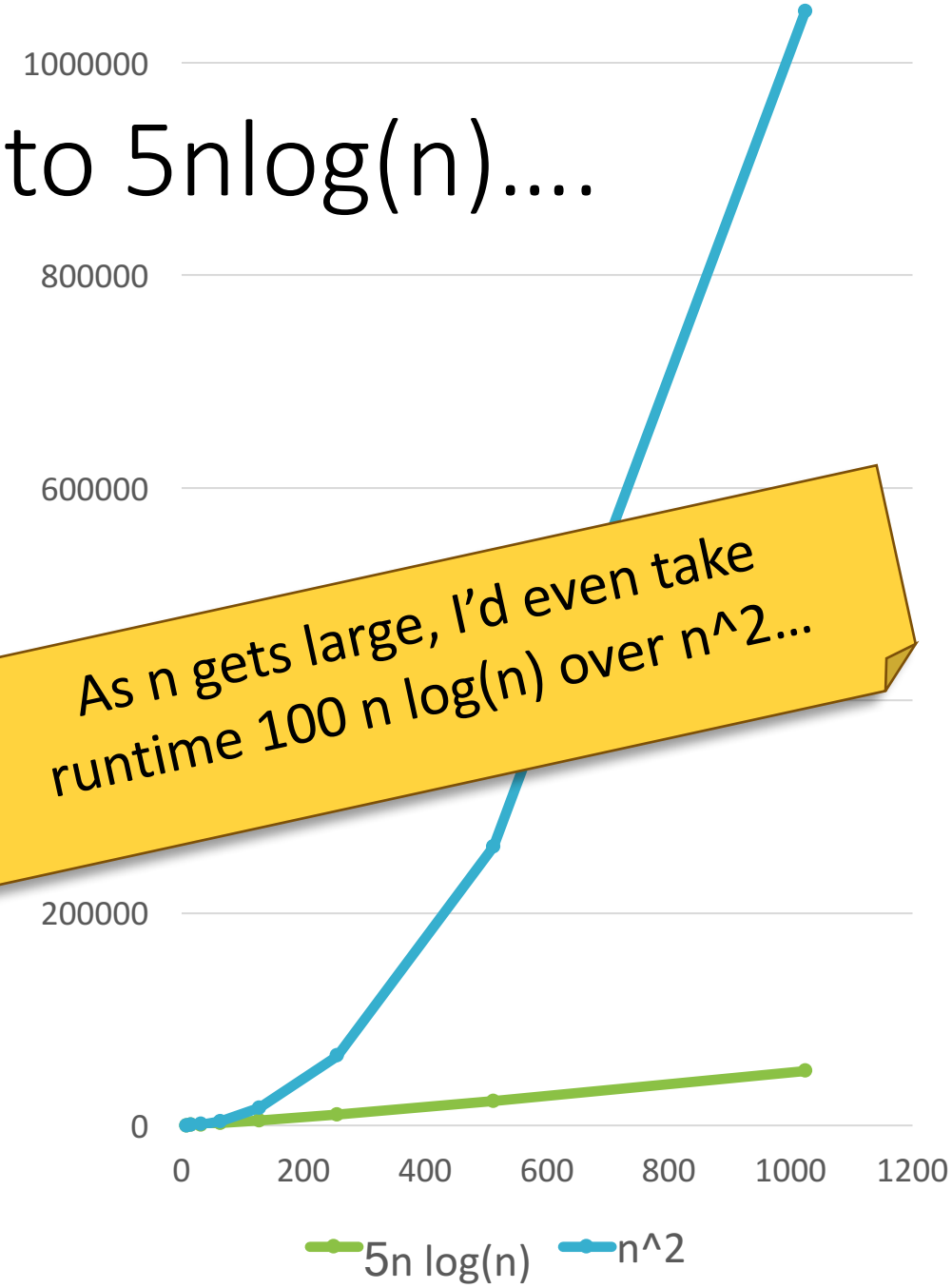
Remember this slide?

n	n log(n)	n^2
8	24	64
16	64	256
32	160	1024
64	384	4096
128	896	16384
256	2048	65536
512	4608	262144
1024	10240	1048576



Change nlog(n) to 5nlog(n)….

n	5n log(n)	n^2
8	120	64
16	320	256
32	800	1024
64	1920	4096
128	4480	16384
256	10240	65536
512	23040	262144
1024	51200	1048576



As n gets large, I'd even take runtime 100 n log(n) over n^2…

Asymptotic Analysis

How does the running time scale as n gets large?

One algorithm is "faster" than another if its runtime grows more "slowly" as n gets large.

Pros:

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

Without making Plucky grumpy!

Cons:

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

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

O(…) means an upper bound

- We say "T(n) is O(f(n))" if f(n) grows at least as fast as T(n) as n gets large.

- Formally,

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

(Explanation of what all these symbols mean on board)

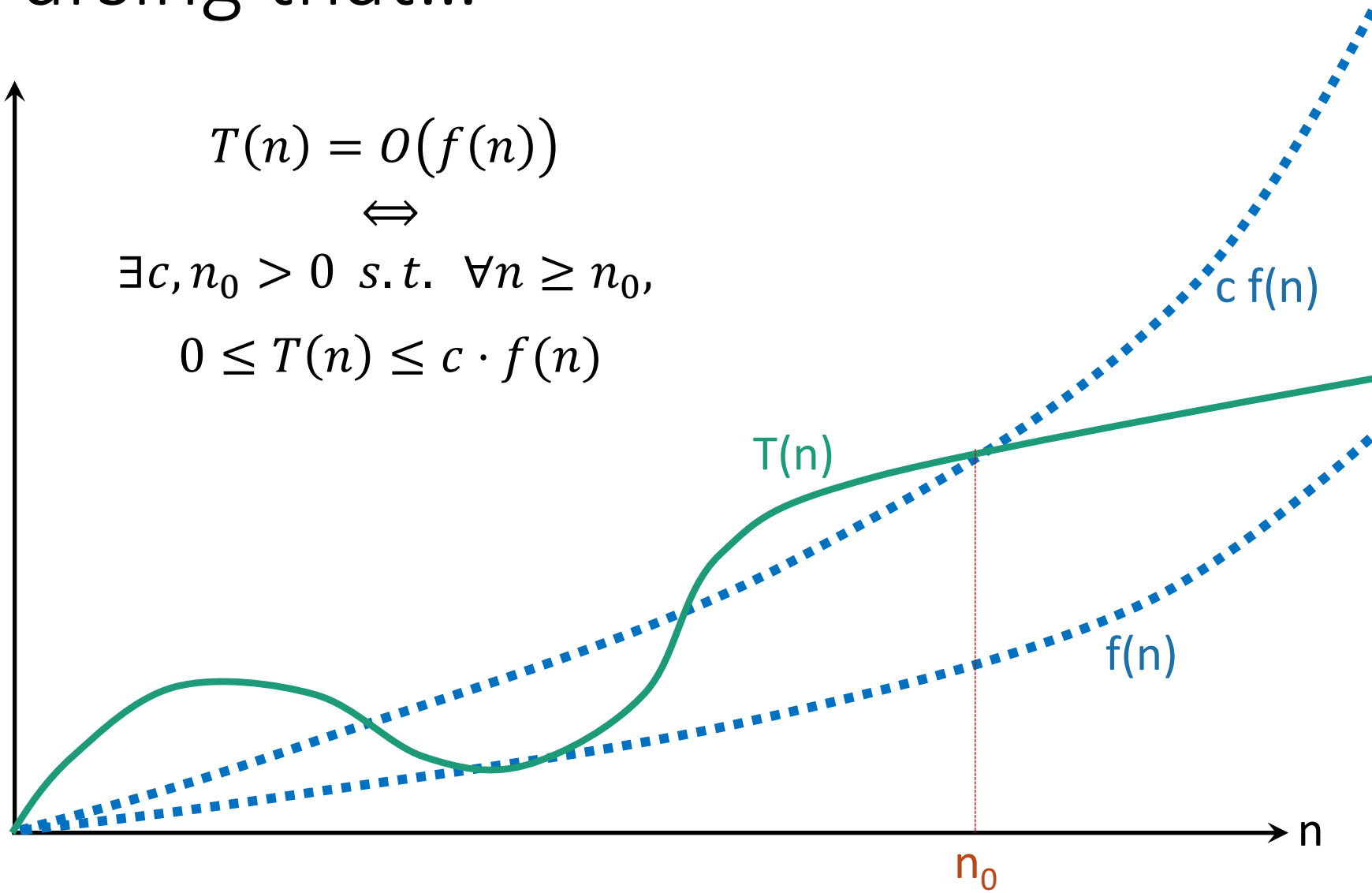
Parsing that…

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

$$\Leftrightarrow$$

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

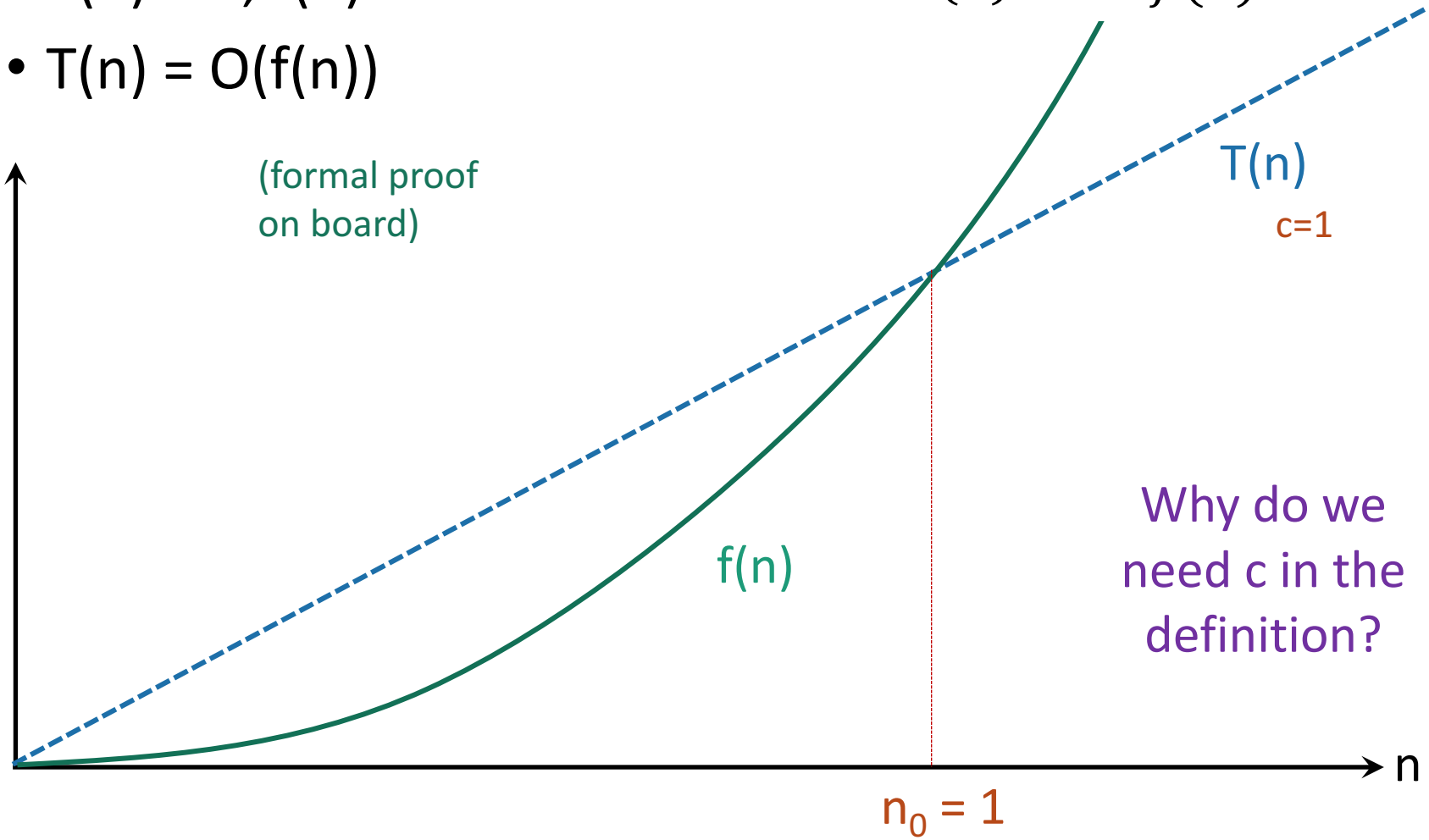
$$0 \leq T(n) \leq c \cdot f(n)$$



Example 1

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

- T(n) = n, f(n) = n^2.
- T(n) = O(f(n))

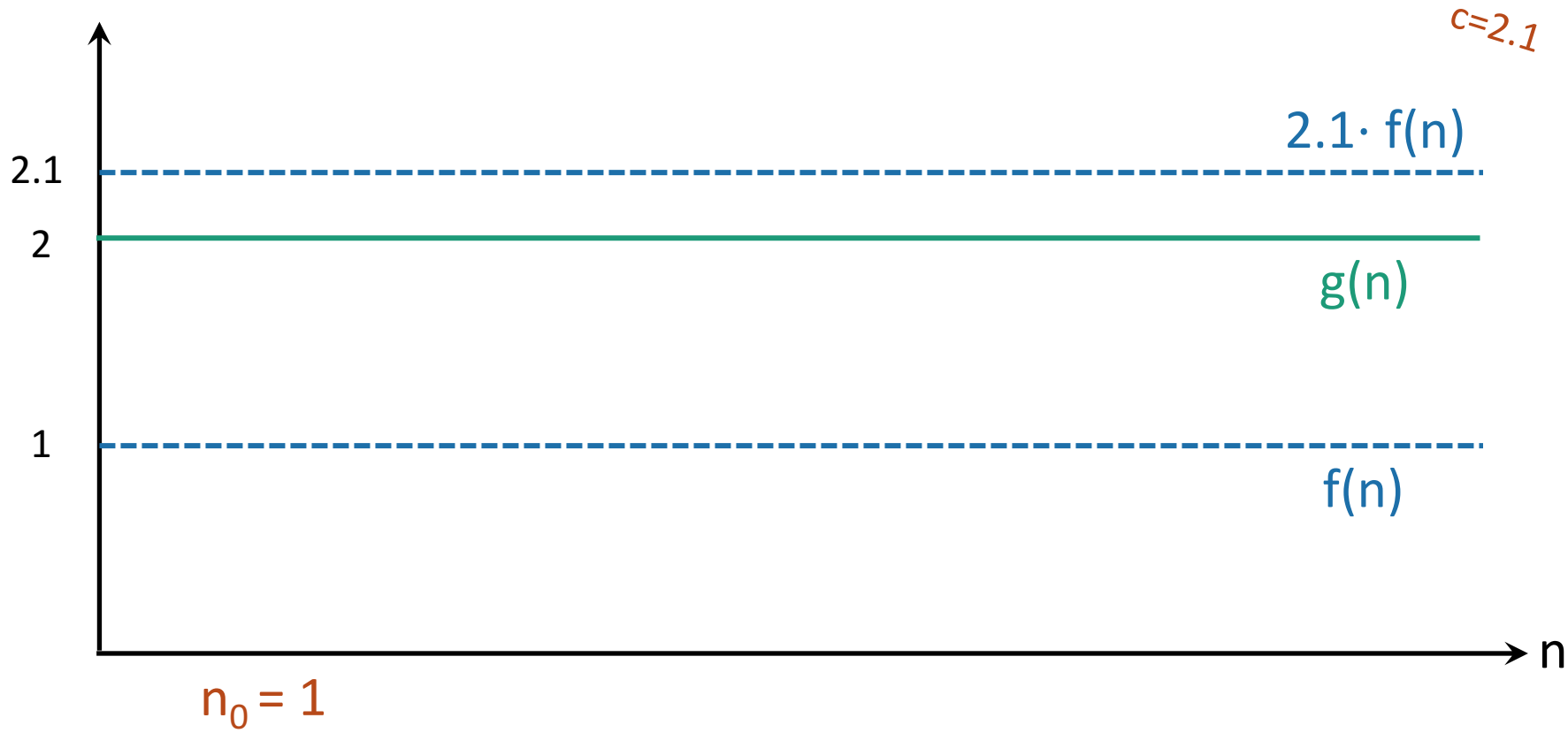


Example 2

(Need c but not really n_0)

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

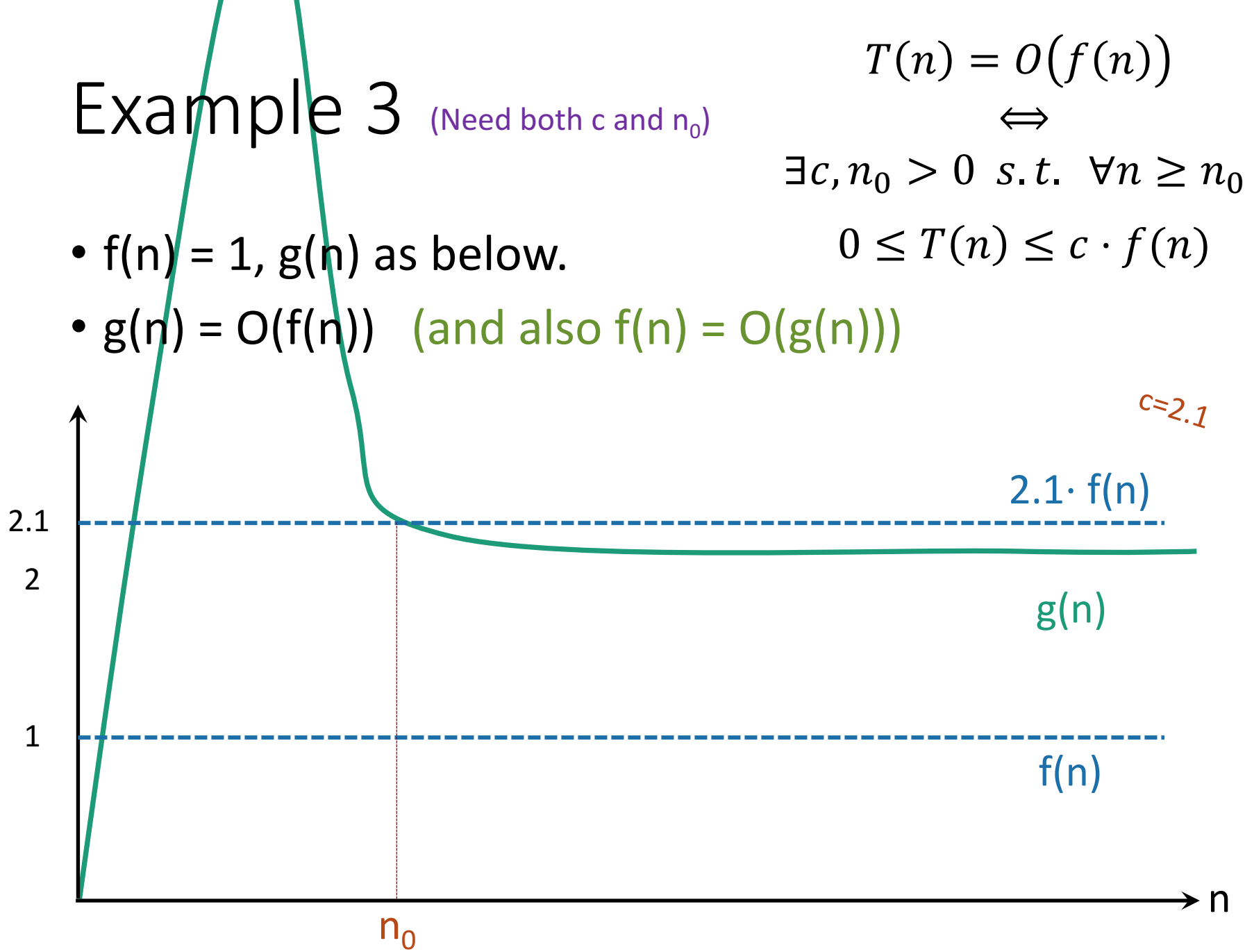
- g(n) = 2, f(n) = 1.
- g(n) = O(f(n)) (and also f(n) = O(g(n)))



Example 3 (Need both c and n_0)

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

- f(n) = 1, g(n) as below.
- g(n) = O(f(n)) (and also f(n) = O(g(n)))



Examples 4 and 5

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

(On the board)

Take-away from examples

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

- To prove $T(n)$ is NOT $O(f(n))$, one way is by contradiction:
  - Suppose 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.

Ω(…) means a lower bound

- We say "T(n) is Ω(f(n))" if f(n) grows at most as fast as T(n) as n gets large.

- Formally,

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

$$\Longleftrightarrow$$

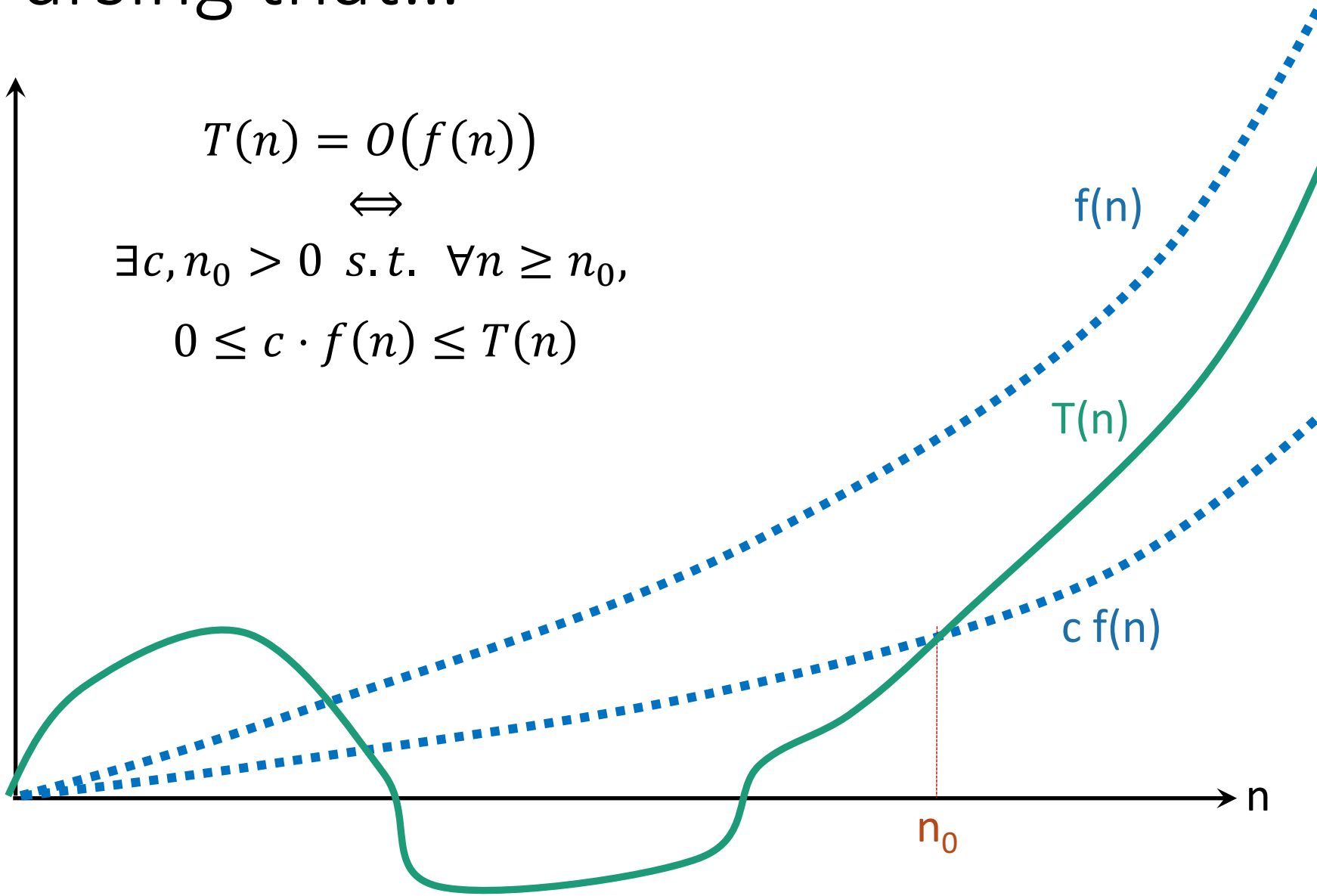
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$

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

Switched these!!

Parsing that…

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



$\Theta(\ldots)$ means both!

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

$$T(n) = O(f(n))$$

-AND-

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

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

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

We'll be using lots of asymptotic notation from here on out

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 $n \geq n_0 = 2^{10000000}$.

Recap

- Divide-and-conquer paradigm
- Sorting: Merge Sort

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

- Next time:
  - Integer multiplication part II
  - A more systematic approach to solving recurrences