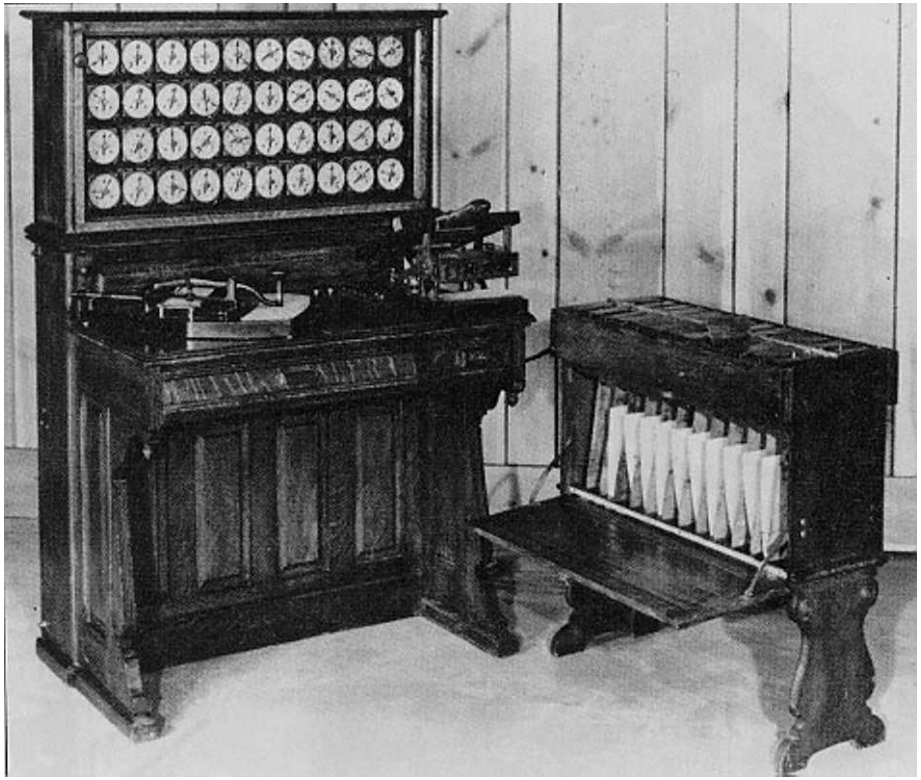
Lecture 3

Divide-and-conquer, Merge-Sort, and Big-O notation

Sorting (Hollerith machines)



1901 - Computer Tabulating Recording (CTR)

1924 - International Business Machines (IBM)

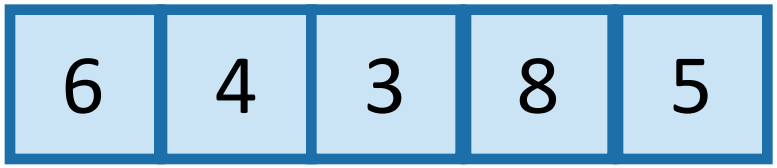
Outline

- Insertion Sort
- Merge Sort / Divide and Conquer
- Asymptotic Analysis
- Big O Notation
- The Master Theorem

Benchmark: insertion sort

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

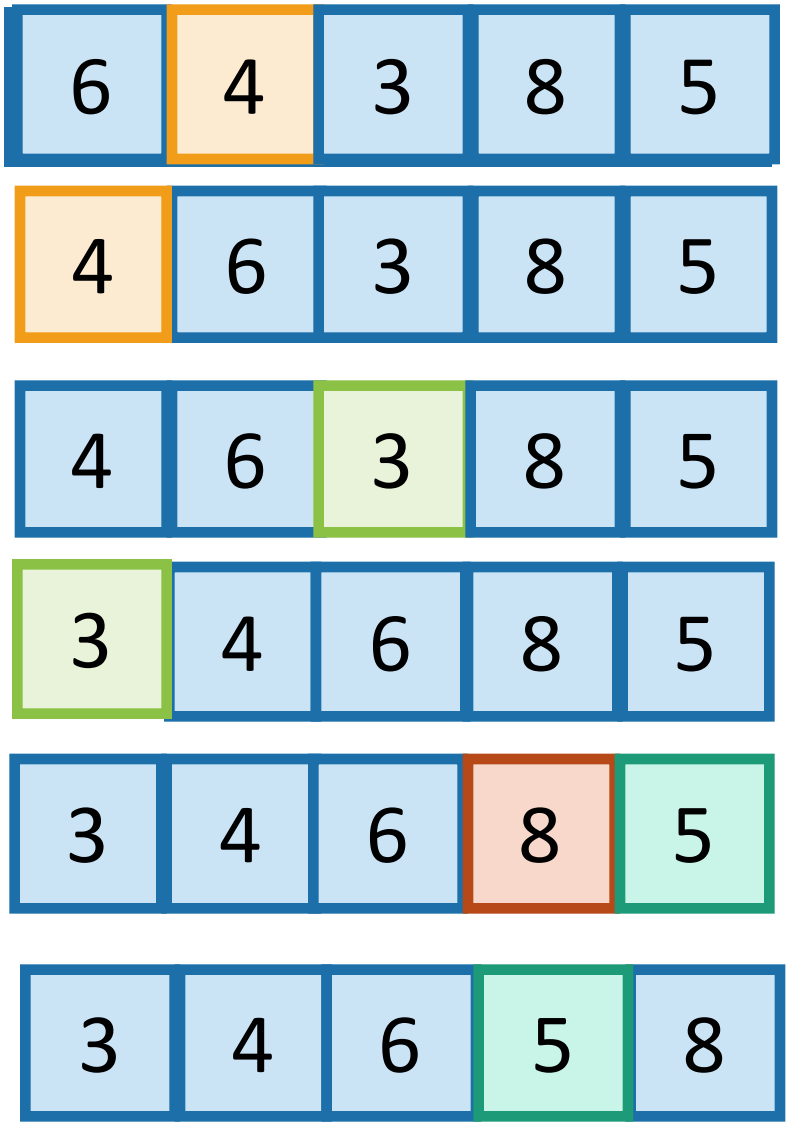
- Say we want to sort:



- Insert items one at a time.

- How would we actually implement this?

Insertion sort example…



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

Pull "4" back until it's in the right place.

Now look at "3"

Pull "3" back until it's in the right place.

"8" is good…look at 5

(then fix 5 and we're done)

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. ✓

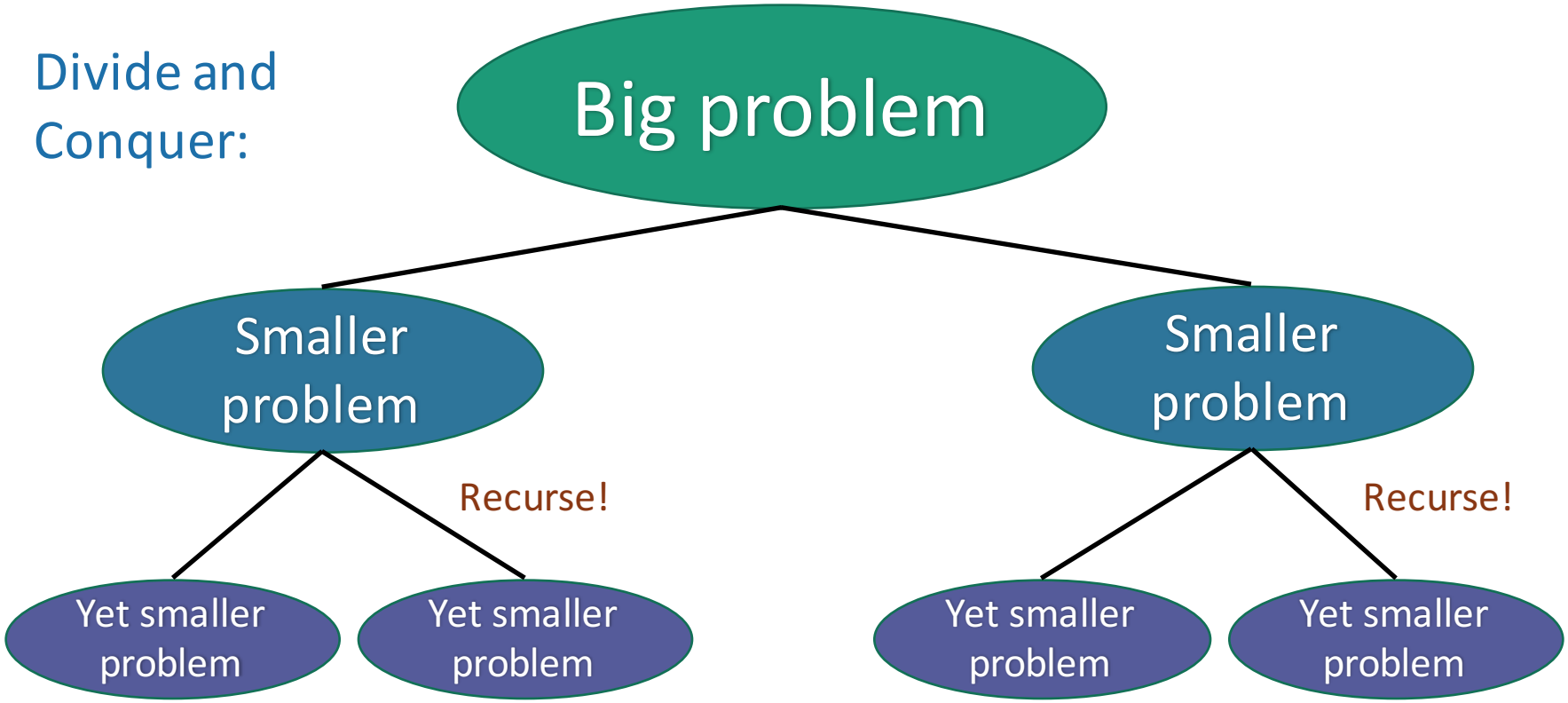
To summarize

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

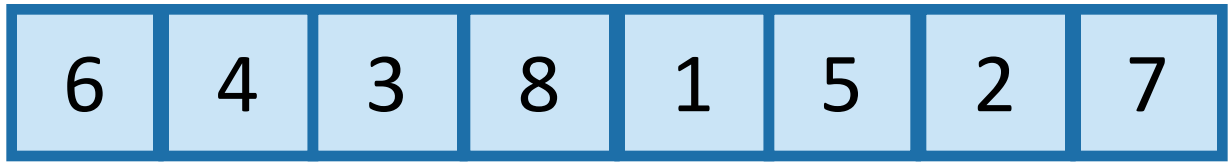
Can we do better?

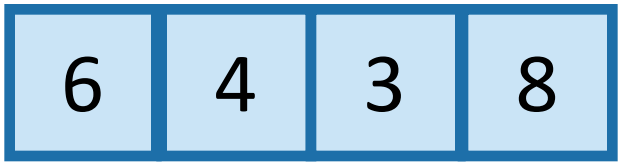
Can we do better?

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

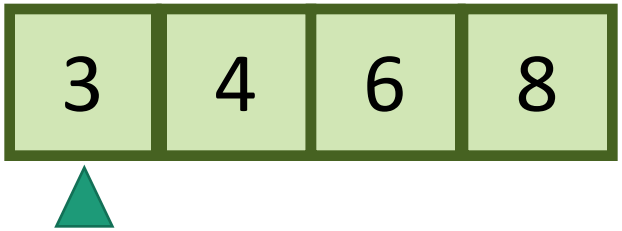


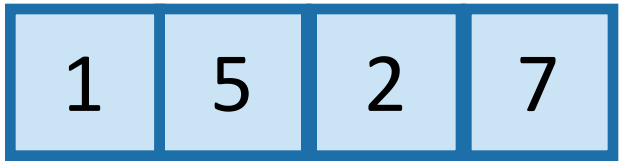
MergeSort



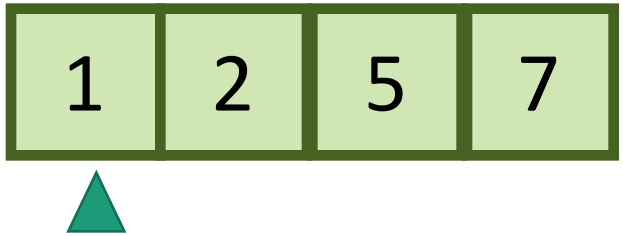


Recursive magic!

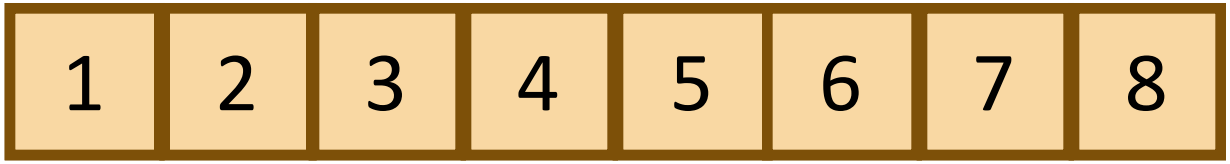




Recursive magic!



MERGE!



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

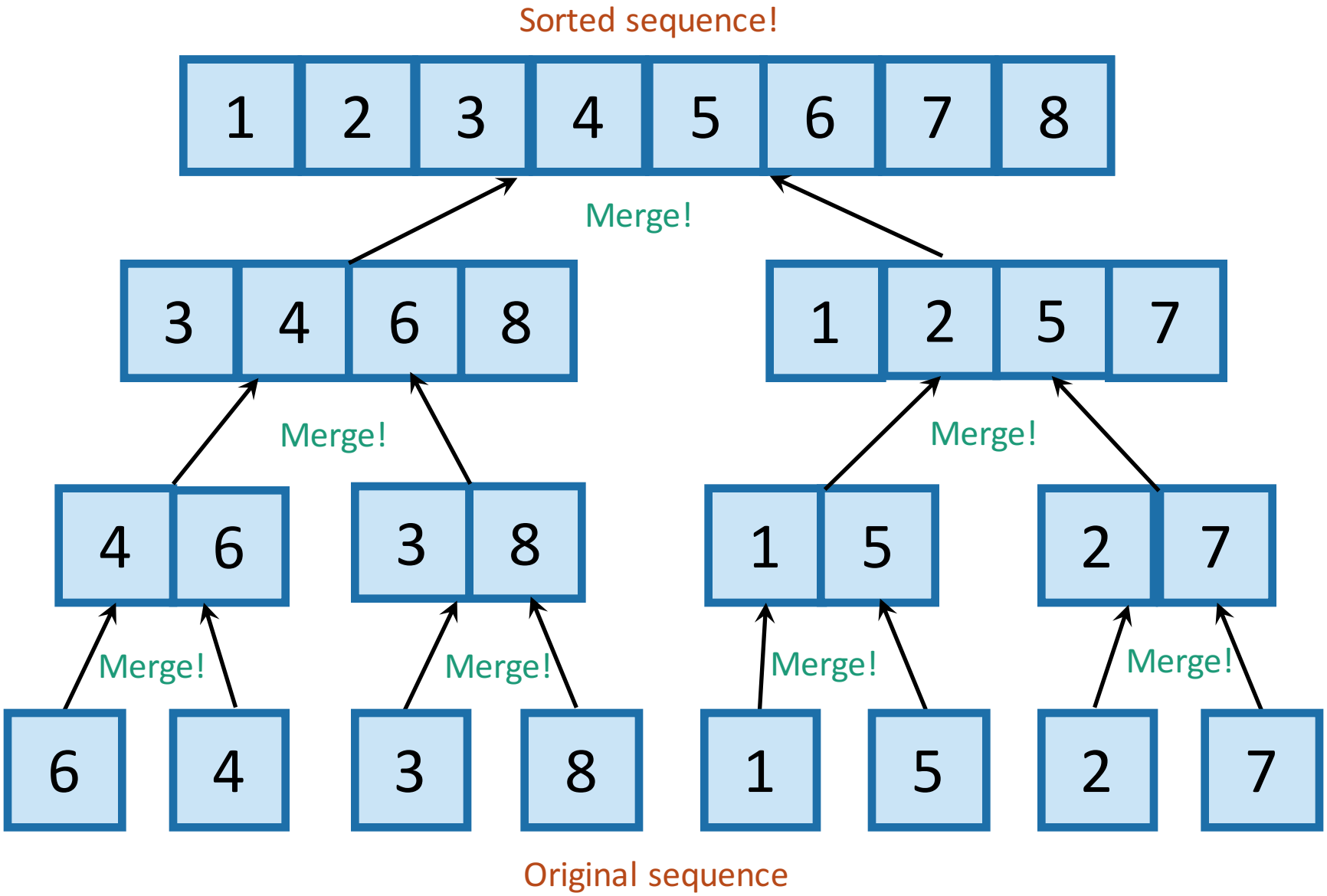
- R = MERGESORT(A[n/2+1 : n])

Sort the right half

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

Merge the two halves

Schematic of recursive calls



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:

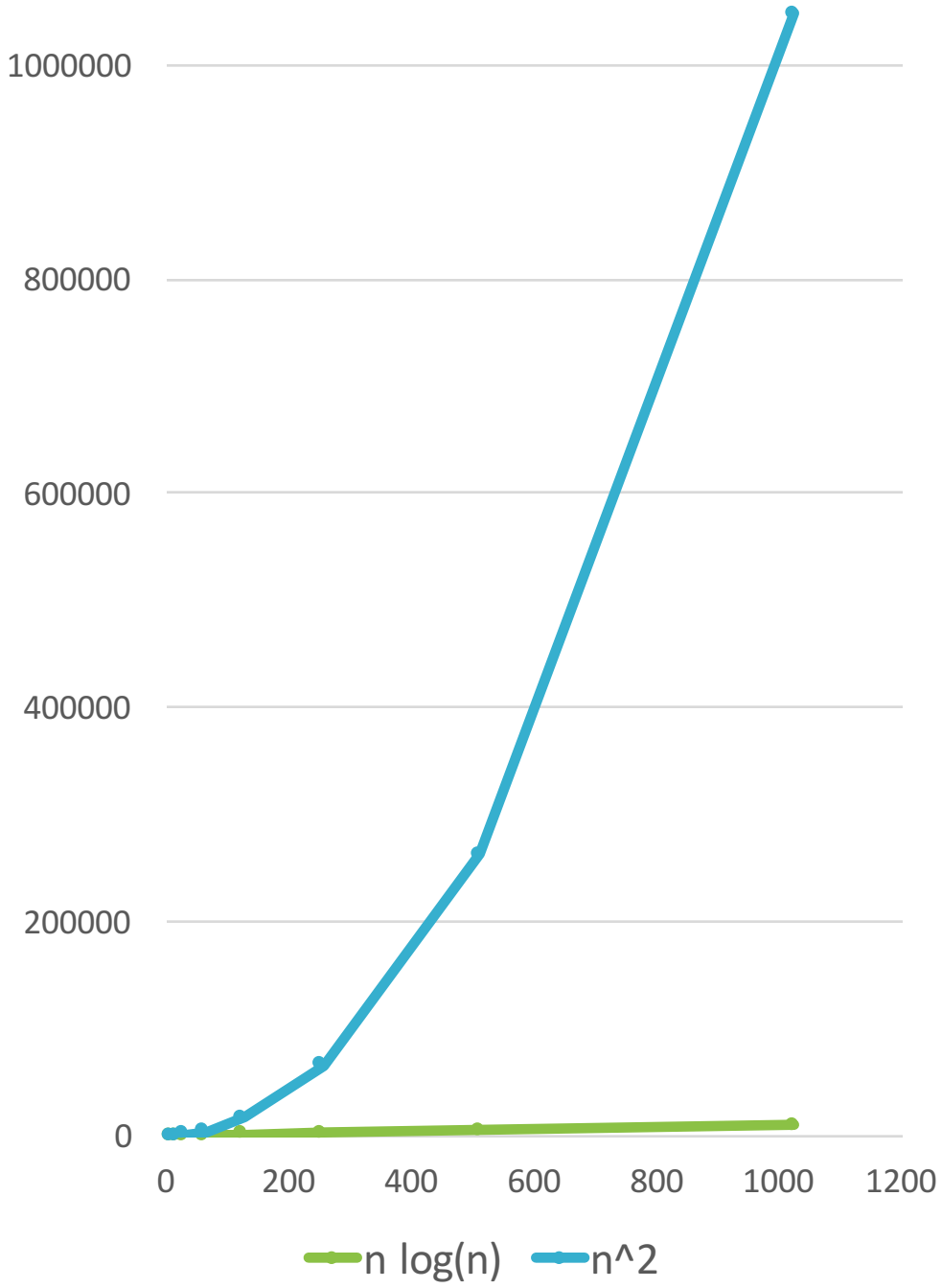
MERGESORT requires at most $6n \log(n) + 6n$ operations to sort n numbers.

Before we see why…

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

n log(n) vs n^2
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



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

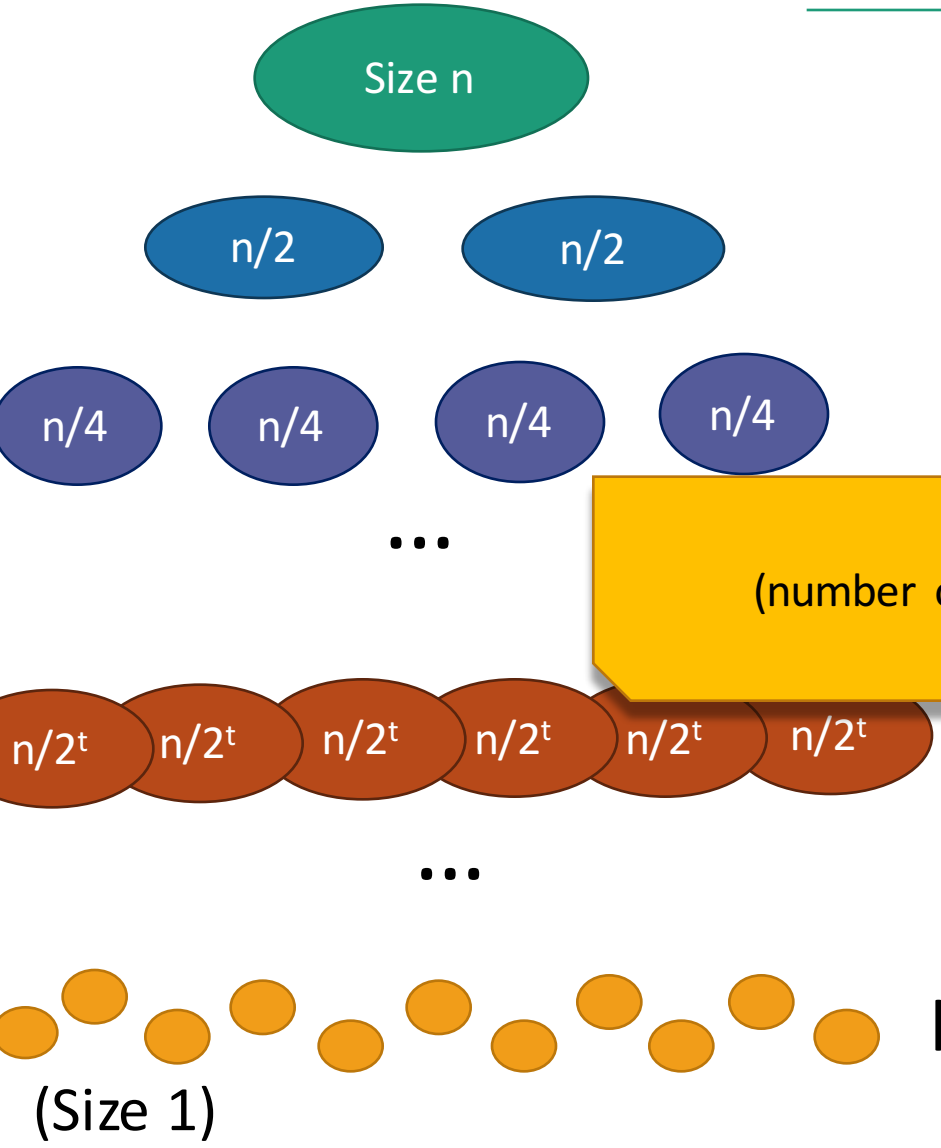
2T(assign) + n T(compare) + n T(assign) + 2n T(increment)

or 4n + 2 operations

Let's say T(MERGE of size n/2) ≤ 6n operations

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)

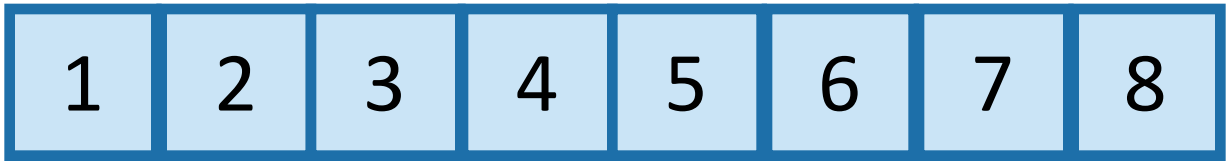
Total runtime…

- 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



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.

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.

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,

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

$$\Leftrightarrow$$

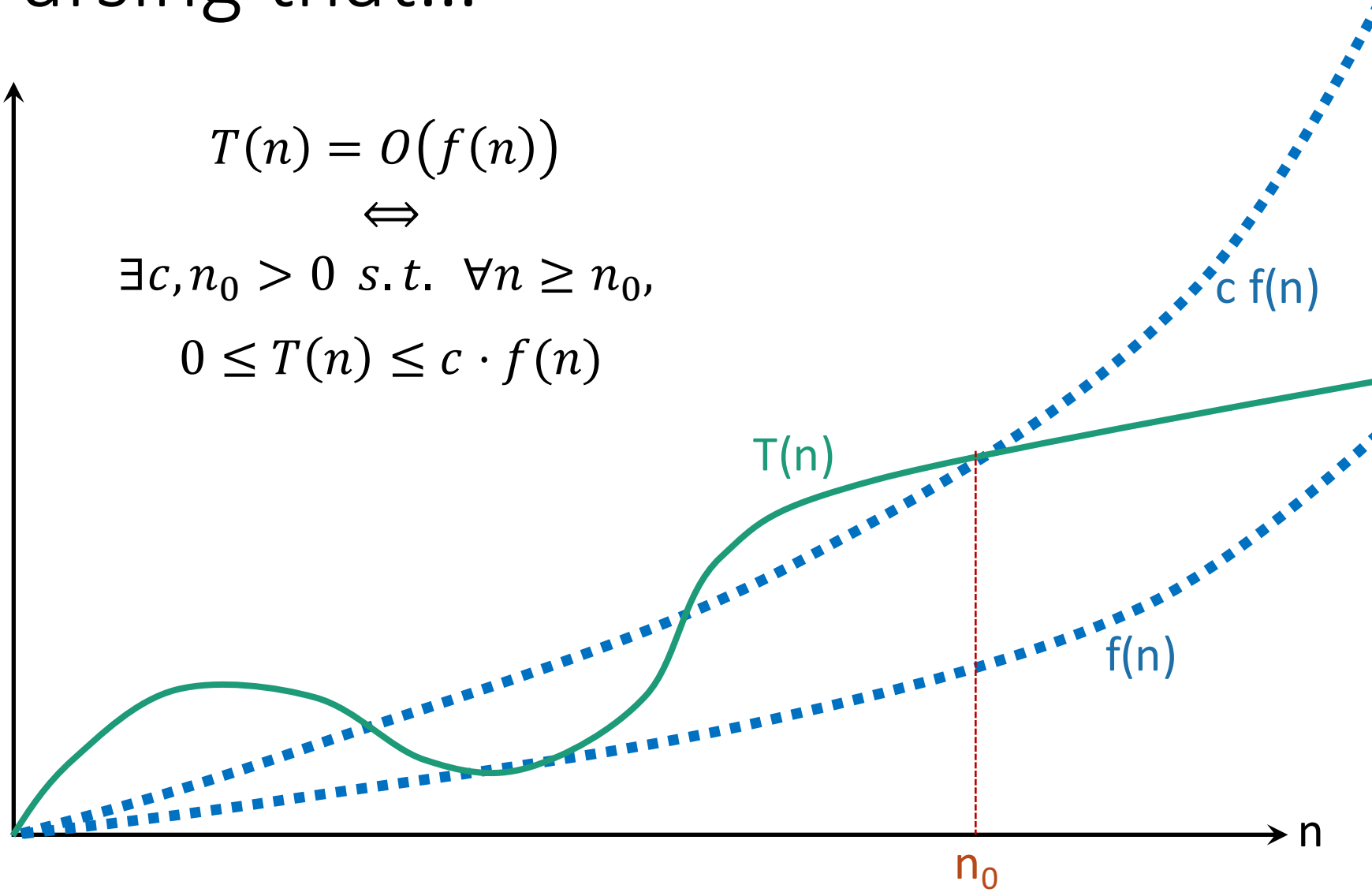
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$

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

(Explanation of what all these symbols mean on board)

Parsing that…

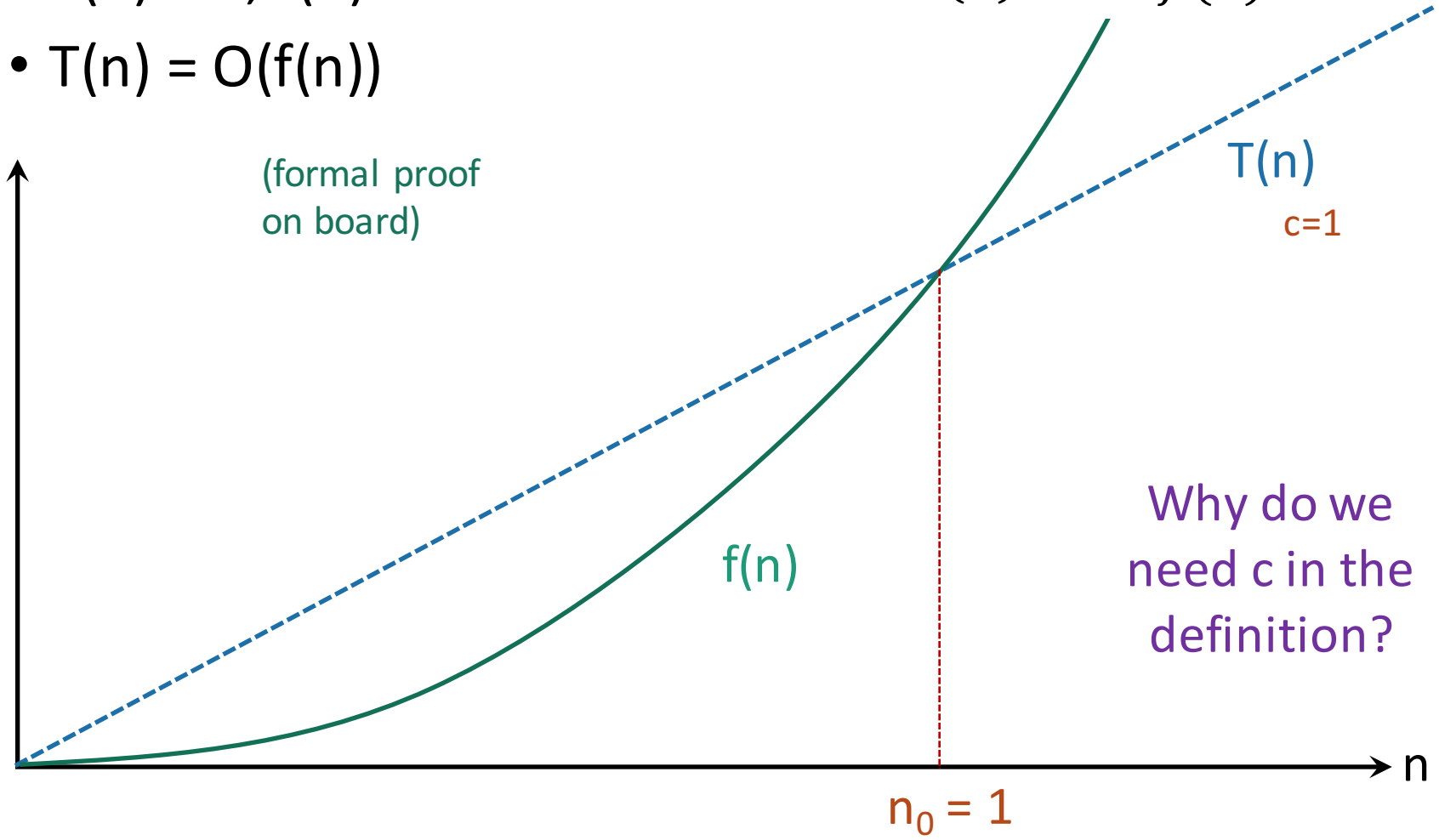
$$T(n) = O\big(f(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot f(n)$$



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))

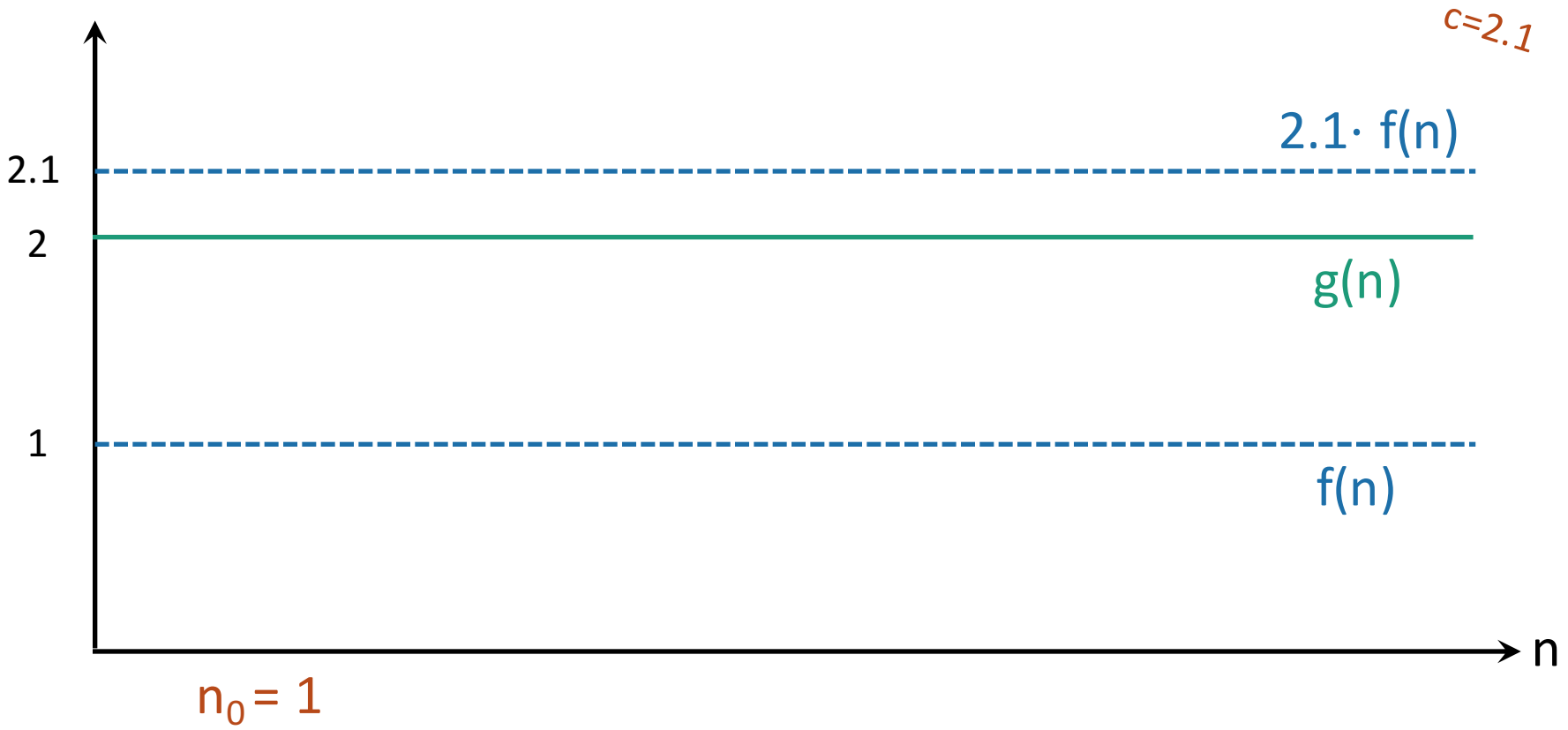


Example 2

(Need c but not really n_0)

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

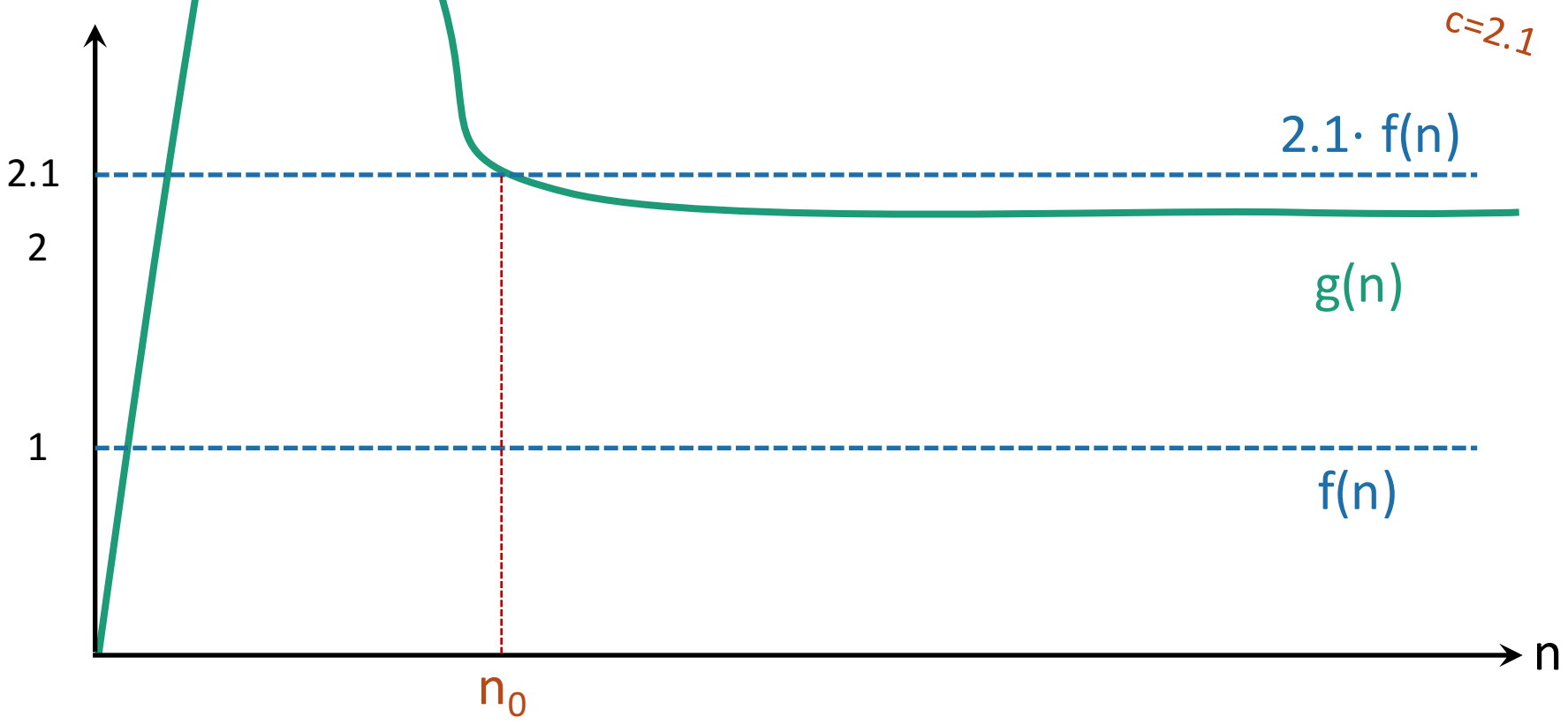
- g(n) = 2, f(n) = 1.
- g(n) = O(f(n)) (and also f(n) = O(g(n)))



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)))



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.

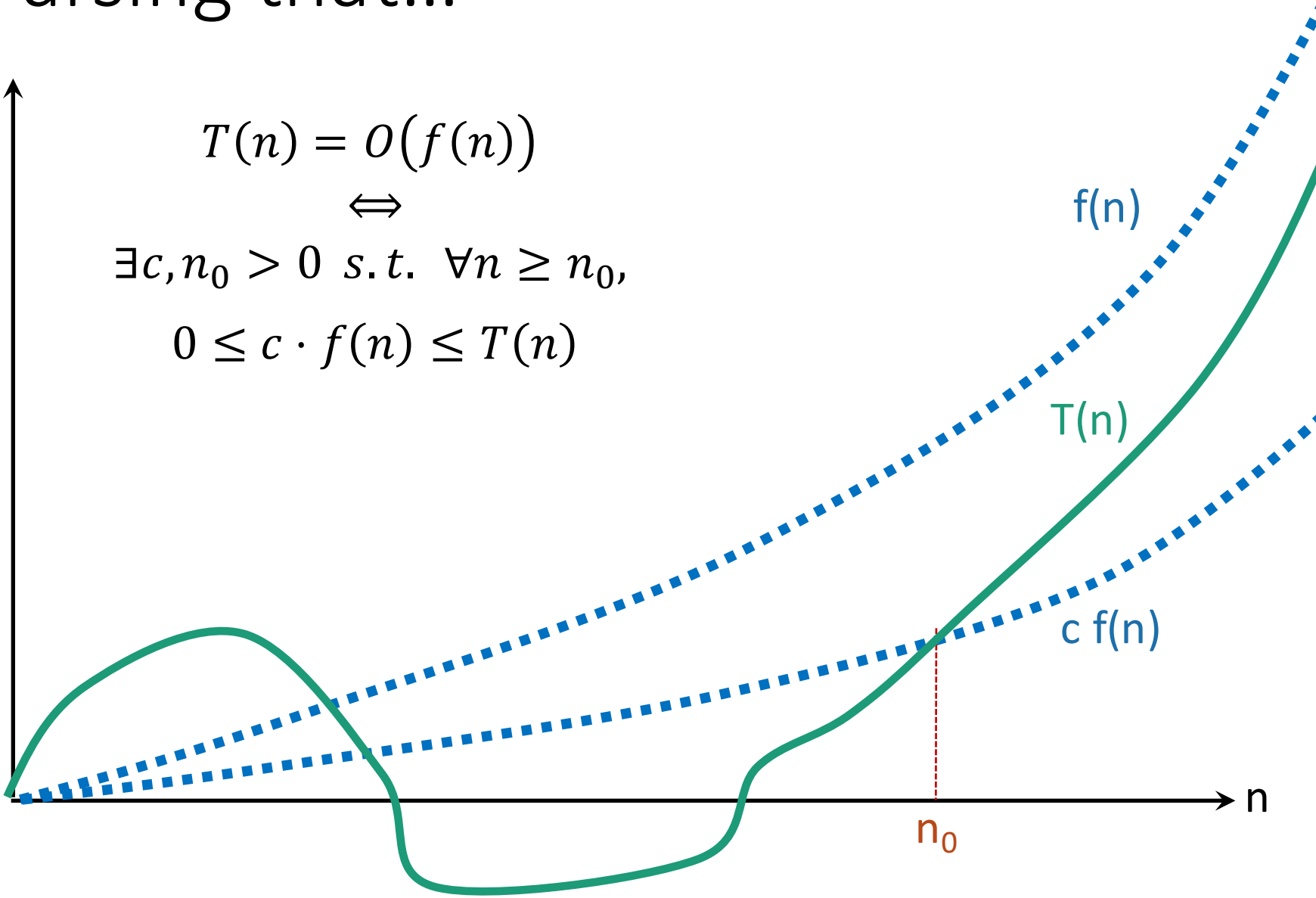
- Formally,

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

Switched these!!

Parsing that…

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



Θ(…) means both!

- We say "T(n) is Θ(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 $O(2^n)$
- $n \log(n) = \Omega(n)$
- $\log(n) = \Theta(2^{\log\log(n)})$

(on board if time – otherwise try them yourself!)

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

The master theorem

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Three parameters:

a : number of subproblems

b : factor by which input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions.

Jedi master Yoda

Summary

- Insertion Sort
- Merge Sort / Divide and Conquer
- Asymptotic Analysis
- Big O Notation
- The Master Theorem