Lecture 2

Asymptotic Notation,

Worst-Case Analysis, and MergeSort

Announcements

- Please (continue to) send OAE letters to cs161-win2122-staff@lists.stanford.edu

Homework!

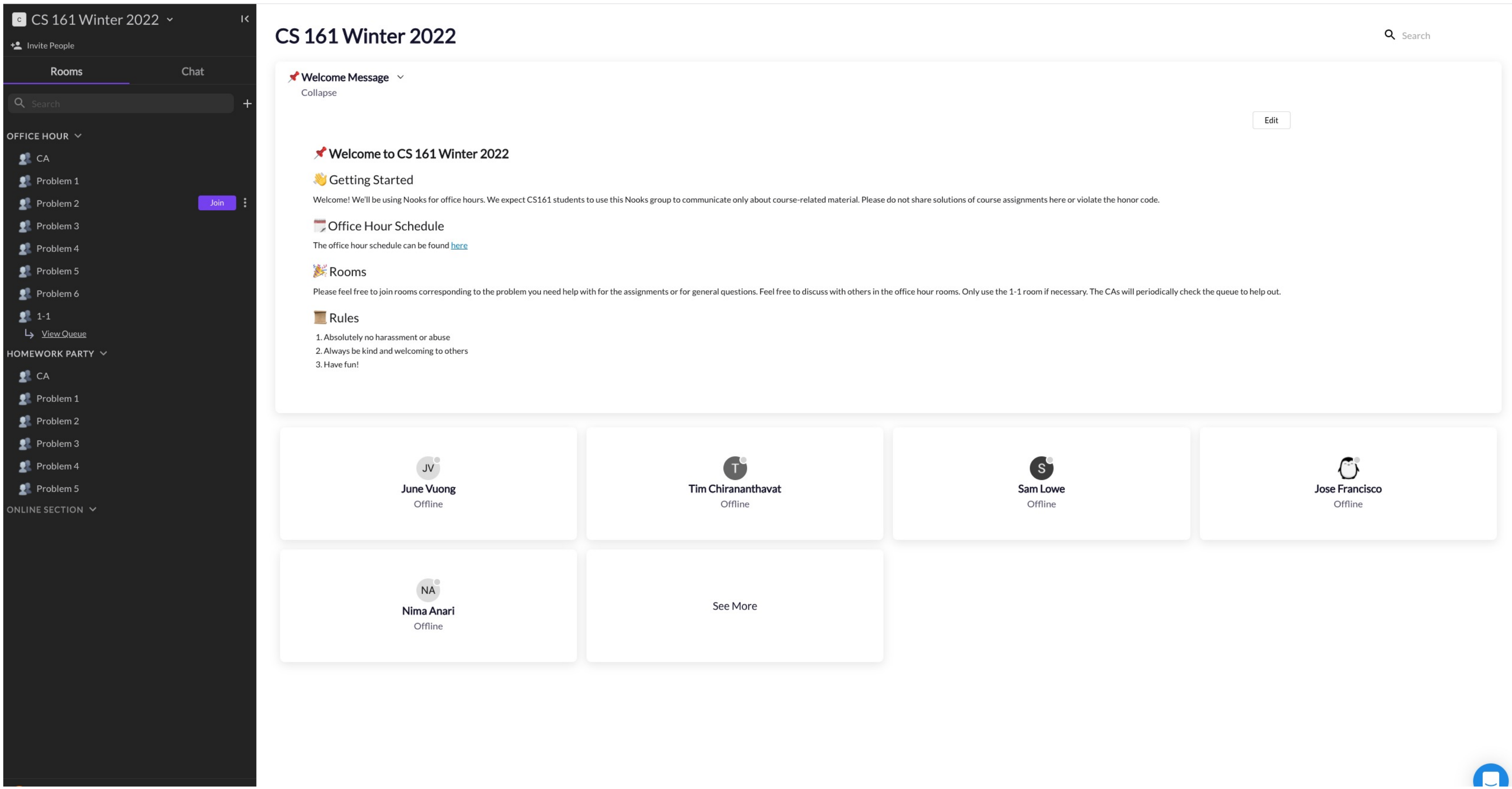
- HW1 will be released **today** (Wednesday).
- It is due the next **Wednesday, 11:59pm** (in one week), on Gradescope.
  - Gradescope link on Canvas
- Homework comes in two parts:
  - Exercises:
    - More straightforward.
    - Try to do them on your own.
  - Problems:
    - Less straightforward.
    - 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 tomorrow
- Homework parties: will be announced soon.
- Sections have been scheduled.
  - See course website
  - Thu 11:00am-12:00pm
  - Thu 1:30pm-2:30pm
  - Thu 5:30pm-6:30pm
  - Fri 11:00am-12:00pm
  - one will be recorded
  - Don't need to formally enroll in sections, just show up!

Huang basement

Nooks



CS 161 Winter 2022
Invite People
Rooms
Chat
Search
OFFICE HOUR
CA
Problem 1
Problem 2
Join
Problem 3
Problem 4
Problem 5
Problem 6
1-1
View Queue
HOMEWORK PARTY
CA
Problem 1
Problem 2
Problem 3
Problem 4
Problem 5
ONLINE SECTION
CS 161 Winter 2022
Search
Welcome Message
Collapse
Edit
Welcome to CS 161 Winter 2022
Getting Started
Welcome! We'll be using Nooks for office hours. We expect CS161 students to use this Nooks group to communicate only about course-related material. Please do not share solutions of course assignments here or violate the honor code.
Office Hour Schedule
The office hour schedule can be found here
Rooms
Please feel free to join rooms corresponding to the problem you need help with for the assignments or for general questions. Feel free to discuss with others in the office hour rooms. Only use the 1-1 room if necessary. The CAs will periodically check the queue to help out.
Rules
1. Absolutely no harassment or abuse
2. Always be kind and welcoming to others
3. Have fun!
JV
June Vuong
Offline
T
Tim Chirananthavat
Offline
S
Sam Lowe
Offline
Jose Francisco
Offline
NA
Nima Anari
Offline
See More

Links on Canvas

Winter 2022

- Home
- Gradescope
- Ed Discussion
- People
- **Syllabus**
- Panopto Course Videos
- Zoom

Design and Analysis of Algorithms

Jump to Today

Course website: https://stanford-cs161.github.io/winter2022/

Lecture link for first 2 weeks of quarter: https://stanford.zoom.us/j/99080790842?pwd=UElhemRNVWMrYUhZNEpCQzBJZWwrQT09

*Please make sure you are signing into zoom webinar link with your Stanford credentials.

Link to join **Nooks** for online office hours: https://spaces.nooks.in/goto/CS-161-Winter-2022~qeNoEoCiKZJ6LbPZ?pwd=iBwZyP

Ed & Gradescope is accessible via tab on the left pane of the course Canvas page.

End of announcements!

Last time

Philosophy

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

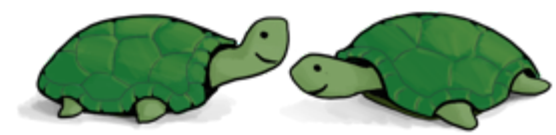
Technical content

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

Cast

Plucky the pedantic penguin

Lucky the lackadaisical lemur



Think-Pair-Share Terrapins

Ollie the over-achieving ostrich

Siggi 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

- 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

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

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???

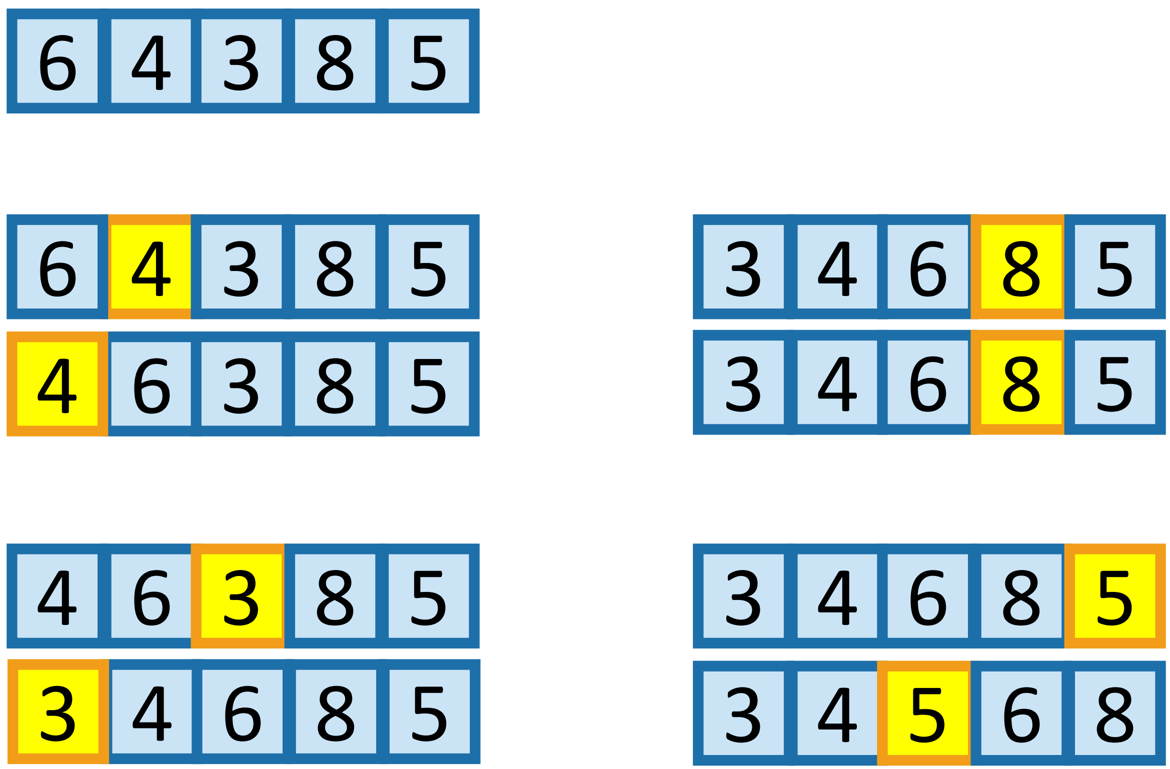
Plucky the
Pedantic Penguin

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:



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.

6 4 3 8 5

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

4 6 3 8 5

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

---

4 6 3 8 5

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

3 4 6 8 5

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

---

3 4 6 8 5

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

3 4 6 8 5

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

---

3 4 6 8 5

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

3 4 5 6 8

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

YAY WE ARE DONE!

This sounds like a job for...

Proof By Induction!

There is a handout with 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 will do the proof by induction on the number of iterations. Let's go over the informal idea first, and we'll do the formal proof below. Let `A` be our input list, and say that it has size n. Our inductive hypothesis will be that after iteration i of the outer loop, `A[:i+1]` is sorted.[1] This is obviously true after iteration 0 (aka, before the algorithm begins), because the one-element list $A[:1]$ is definitely sorted. Then we'll show that for any k with $0 < k < n$, if the inductive hypothesis holds for $i = k - 1$, then it holds for $i = k$. That is, if it is true

[1] An inductive hypothesis like this is sometimes called a *loop invariant*, because it's something that we want to hold (aka, be "invariant") at each iteration of the loop.

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 handout 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 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:
  - GO TO SECTION
  - **GO TO SECTION**
  - Handout
  - References
  - Office Hours

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

Siggi the Studious Stork

What have we learned?

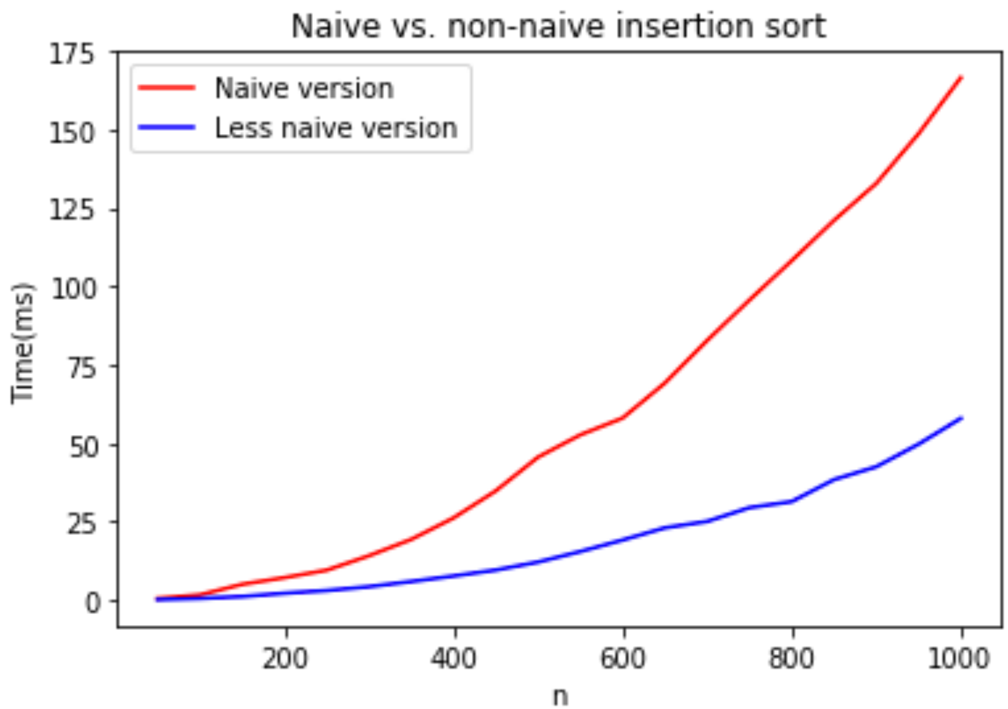
- In this class we will use worst-case analysis:
  - We assume that a "bad guy" comes up with a worst-case input for our algorithm, and we measure performance on that worst-case input.
- With this definition, InsertionSort "works"
  - Proof by induction!

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?

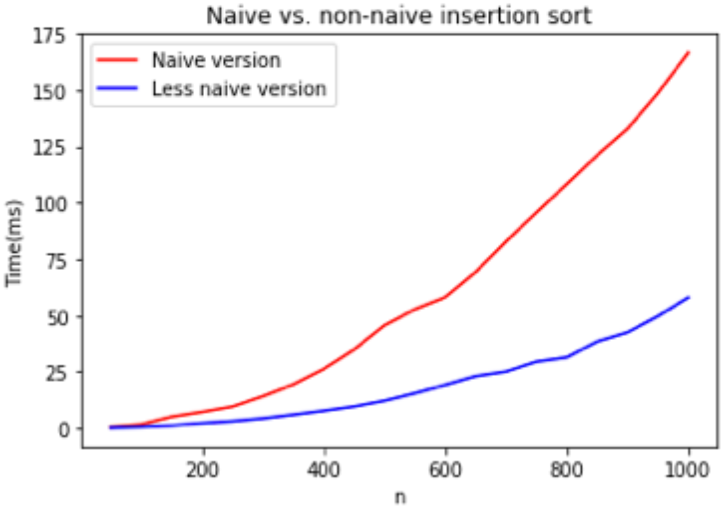
How fast is InsertionSort?

- 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 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:

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

Some examples…

(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 large enough n,

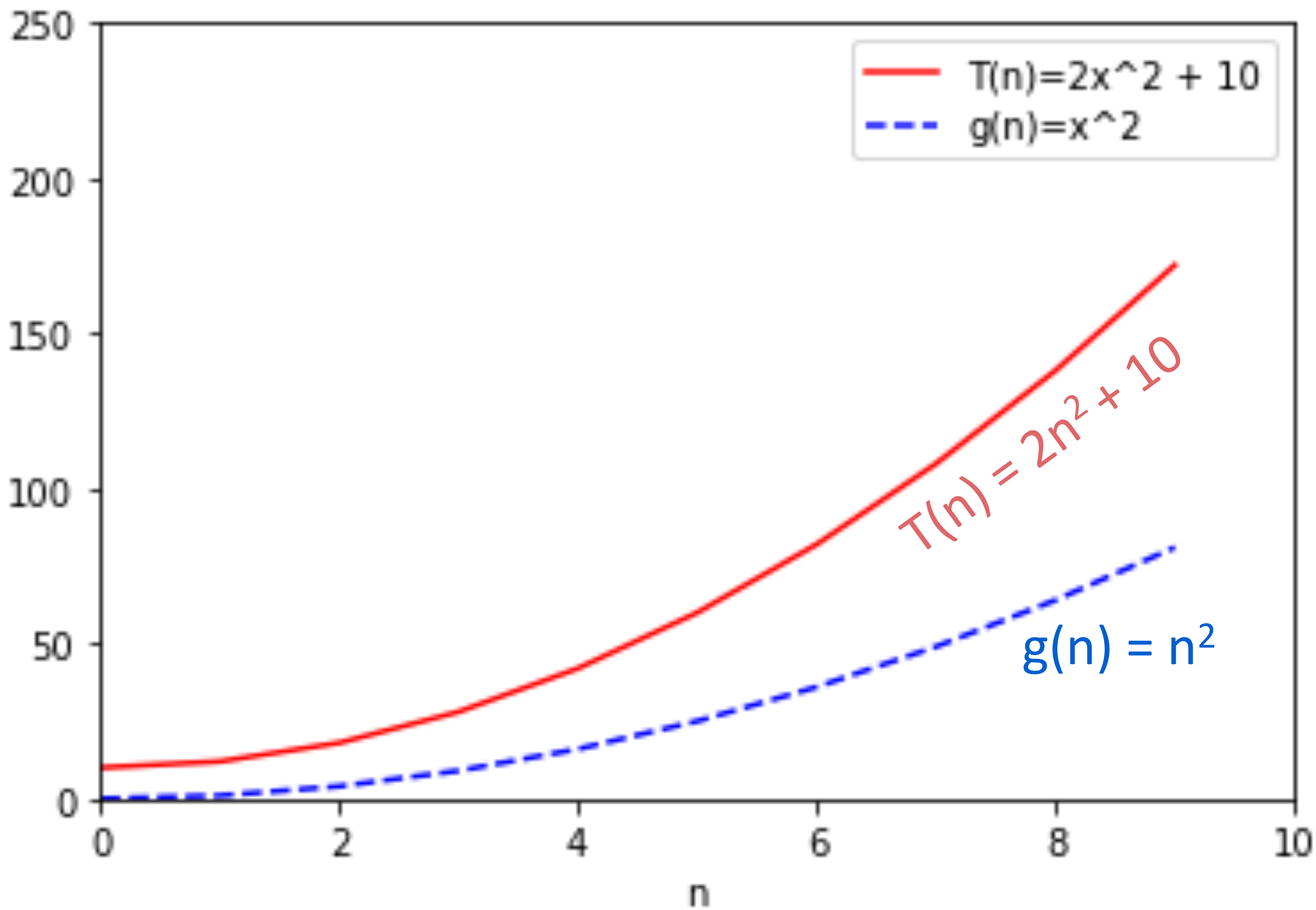
$T(n)$ is at most some constant multiple of $g(n)$.

Here, "constant" means "some number that doesn't depend on n."

Example

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

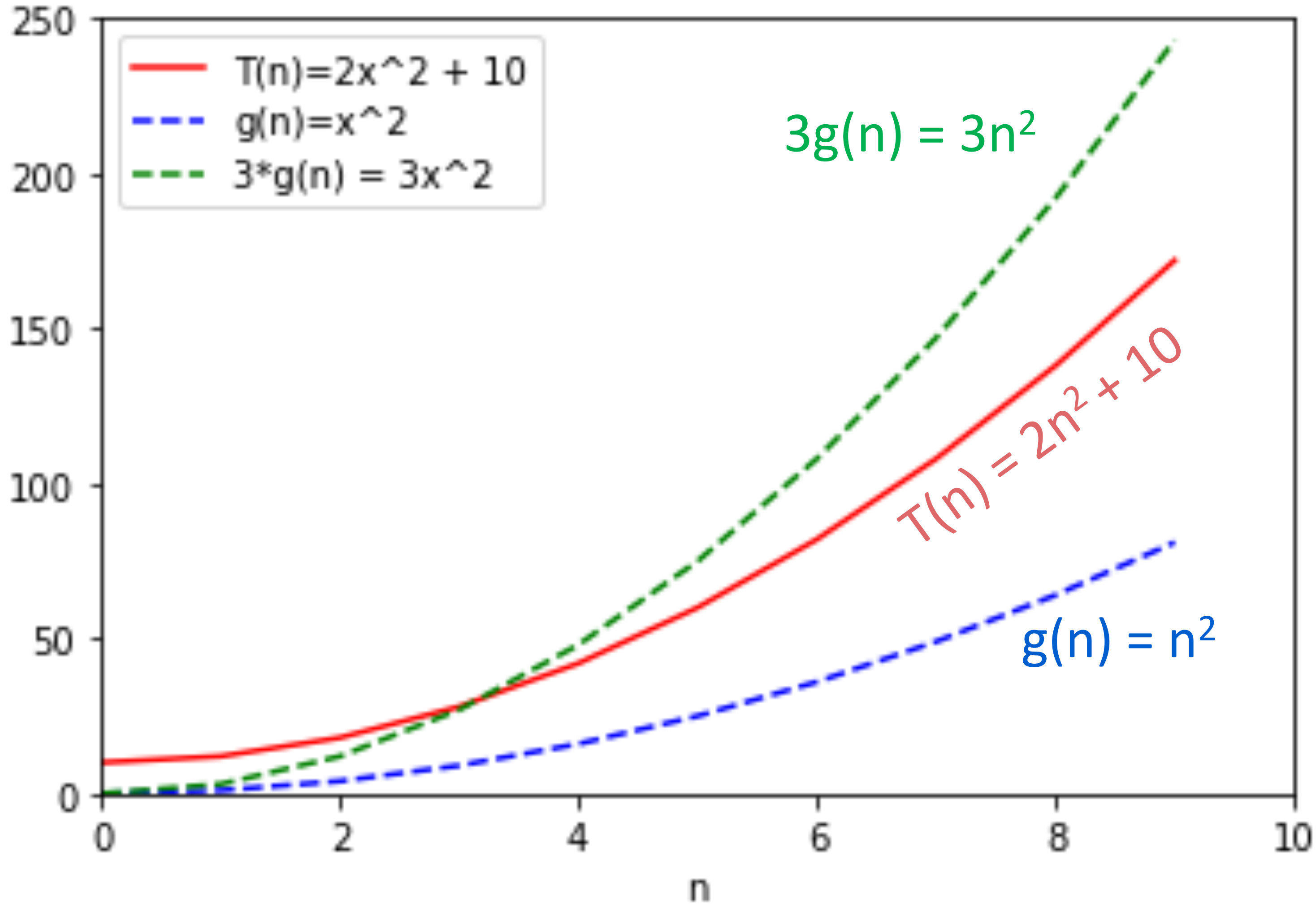
for large enough n,
$T(n)$ is at most some constant
multiple of $g(n)$.



Example

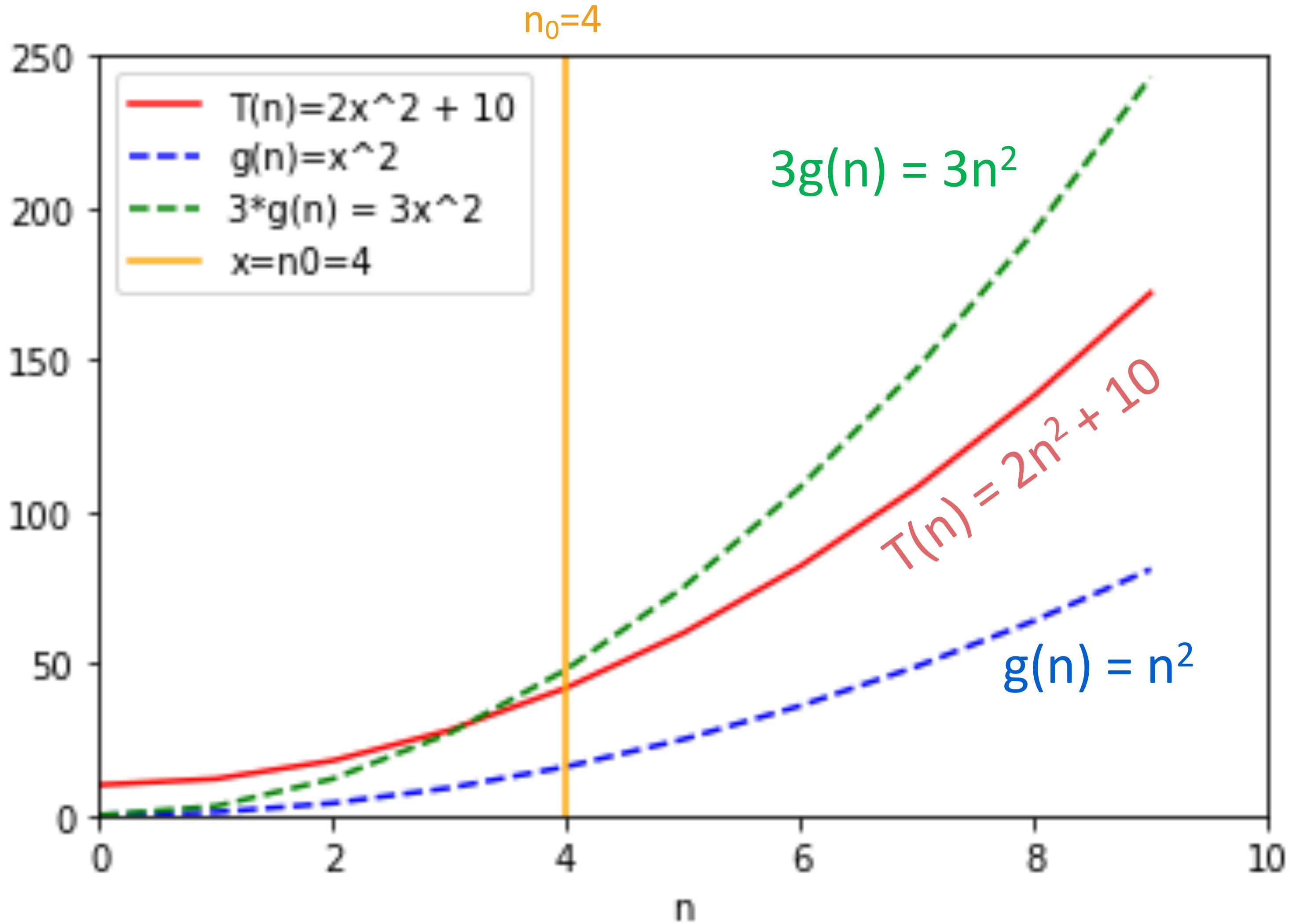
$$2n^2 + 10 = O(n^2)$$

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,

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

"If and only if"

$$\Longleftrightarrow$$

"For all"

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

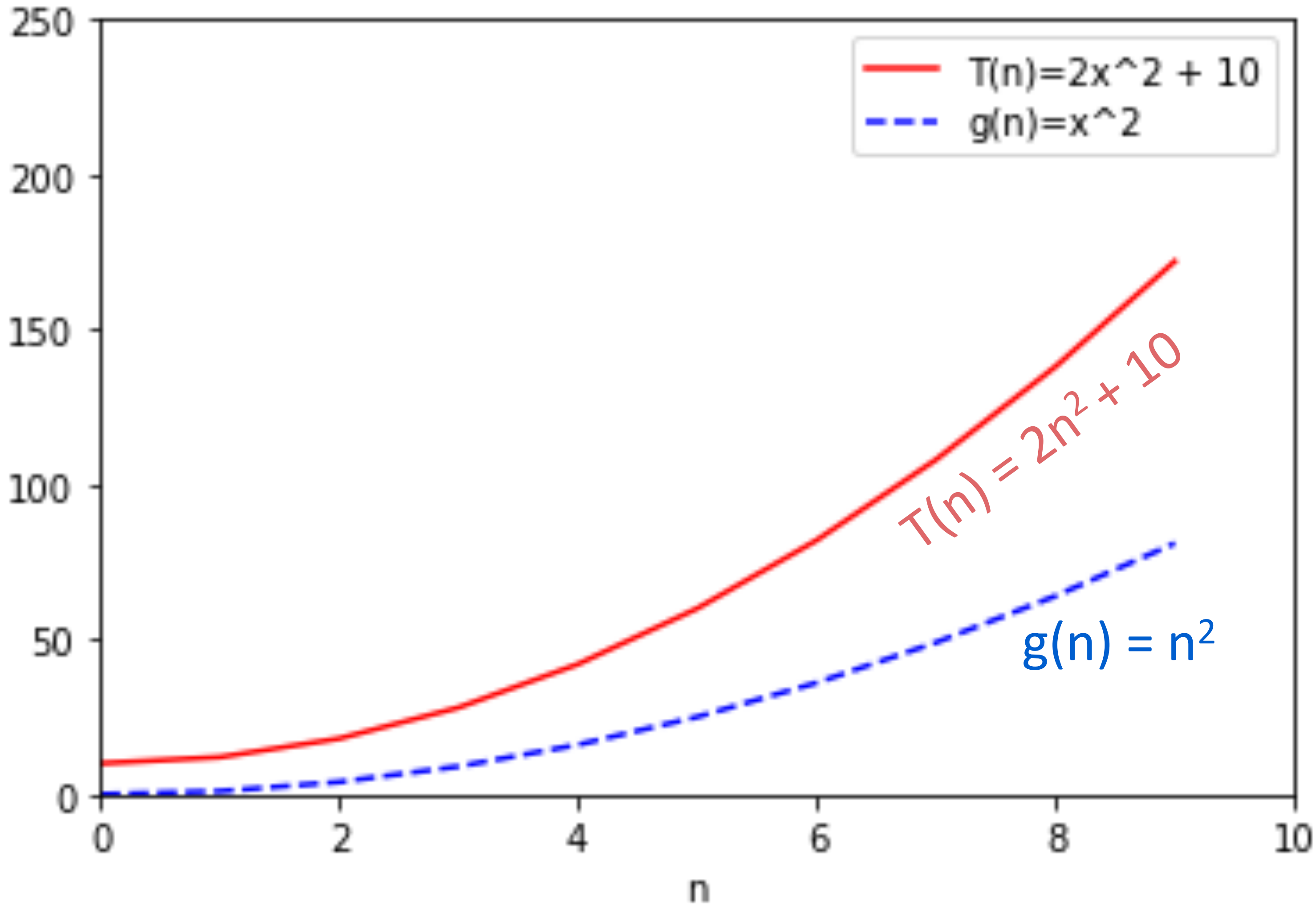
$$T(n) \leq c \cdot g(n)$$

"There exists"

"such that"

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

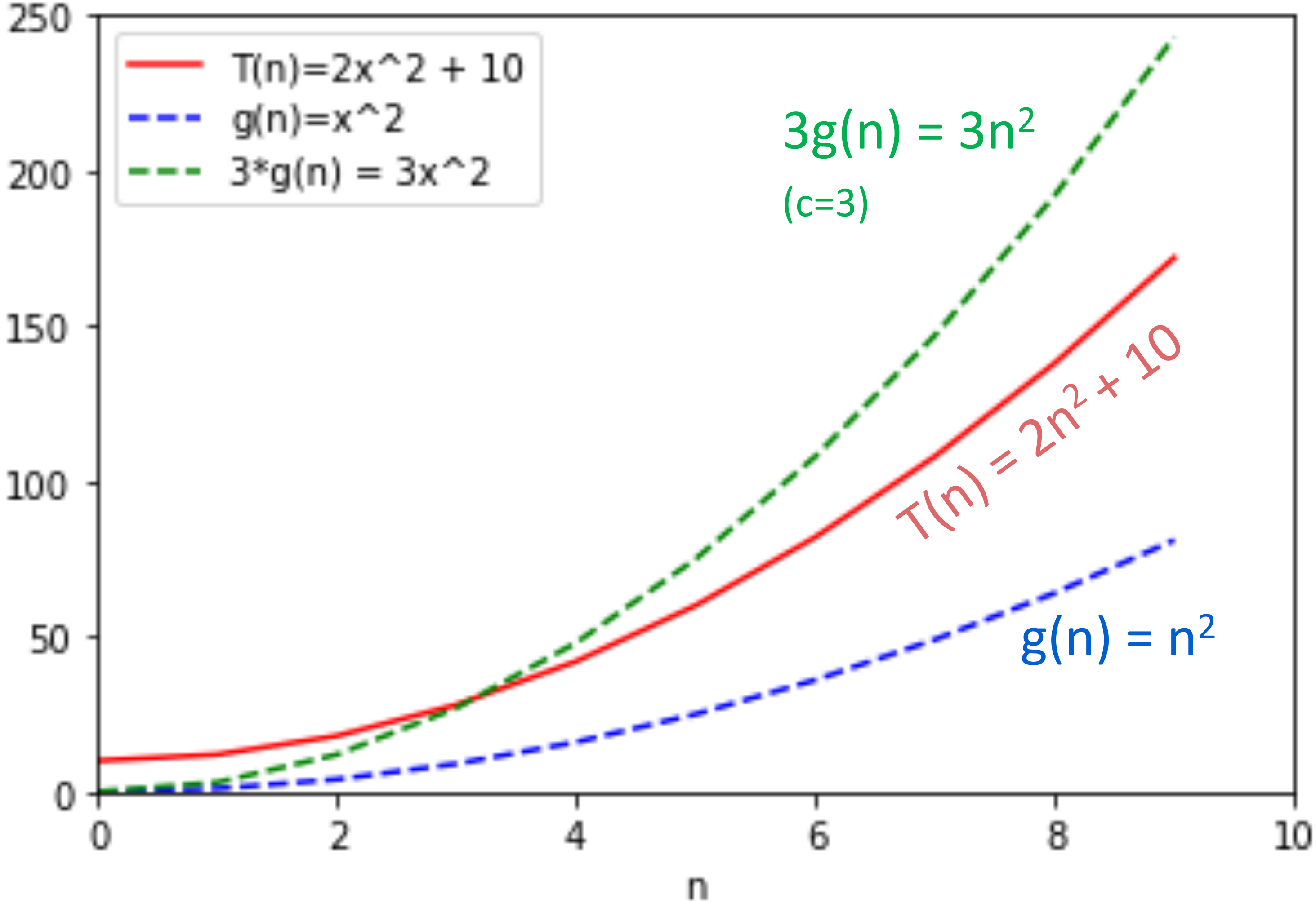
$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Example

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

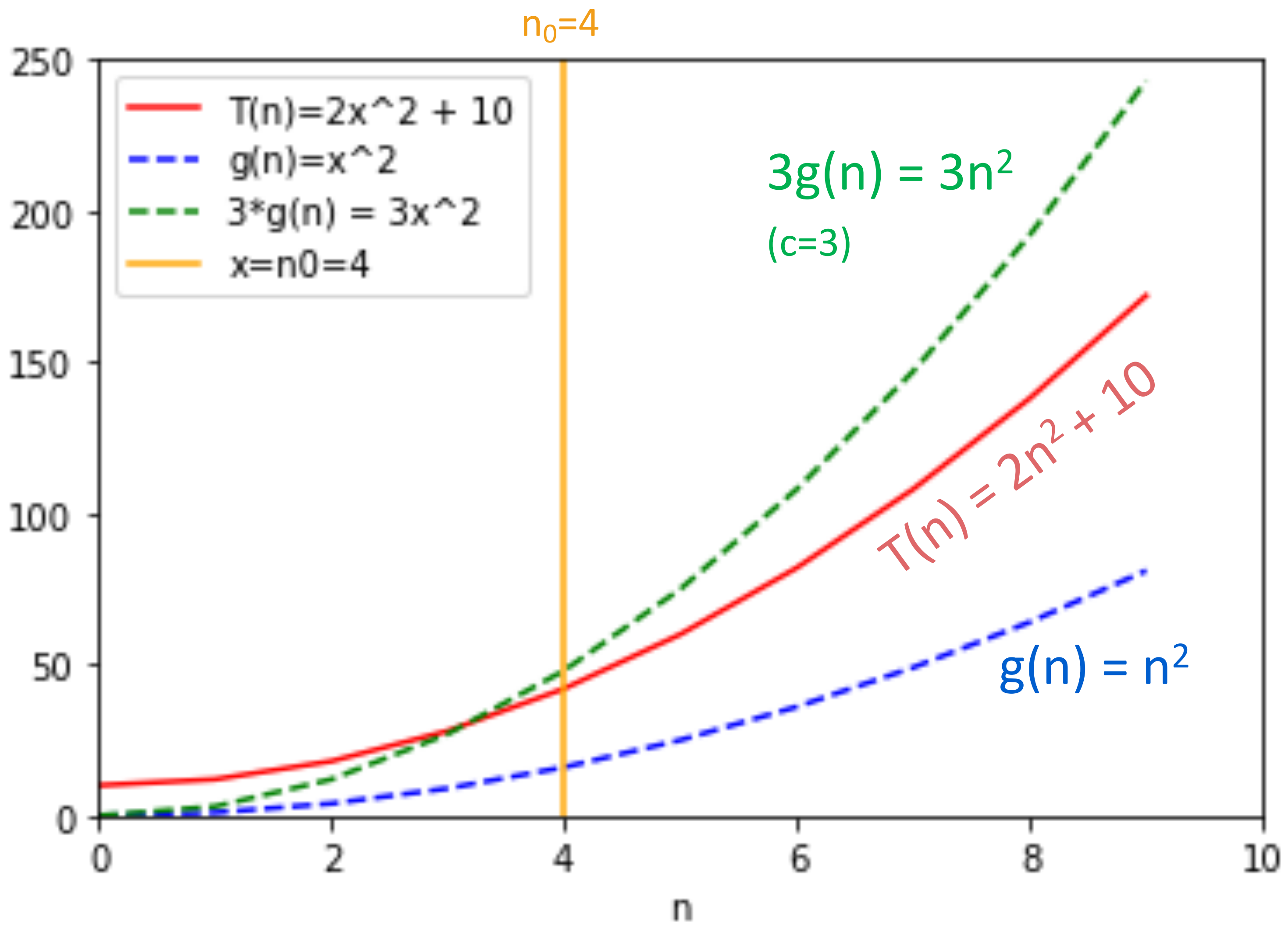
$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Example

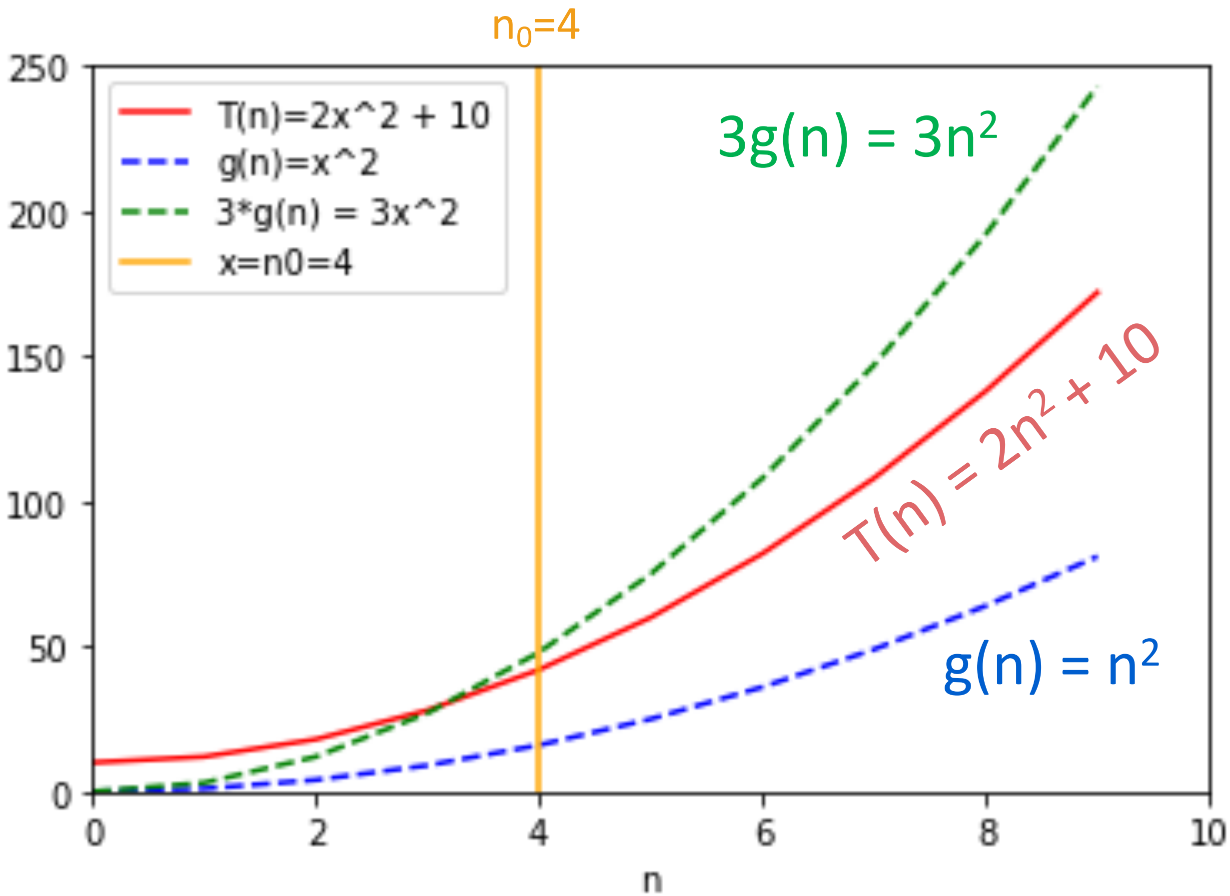
$2n^2 + 10 = O(n^2)$

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



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

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

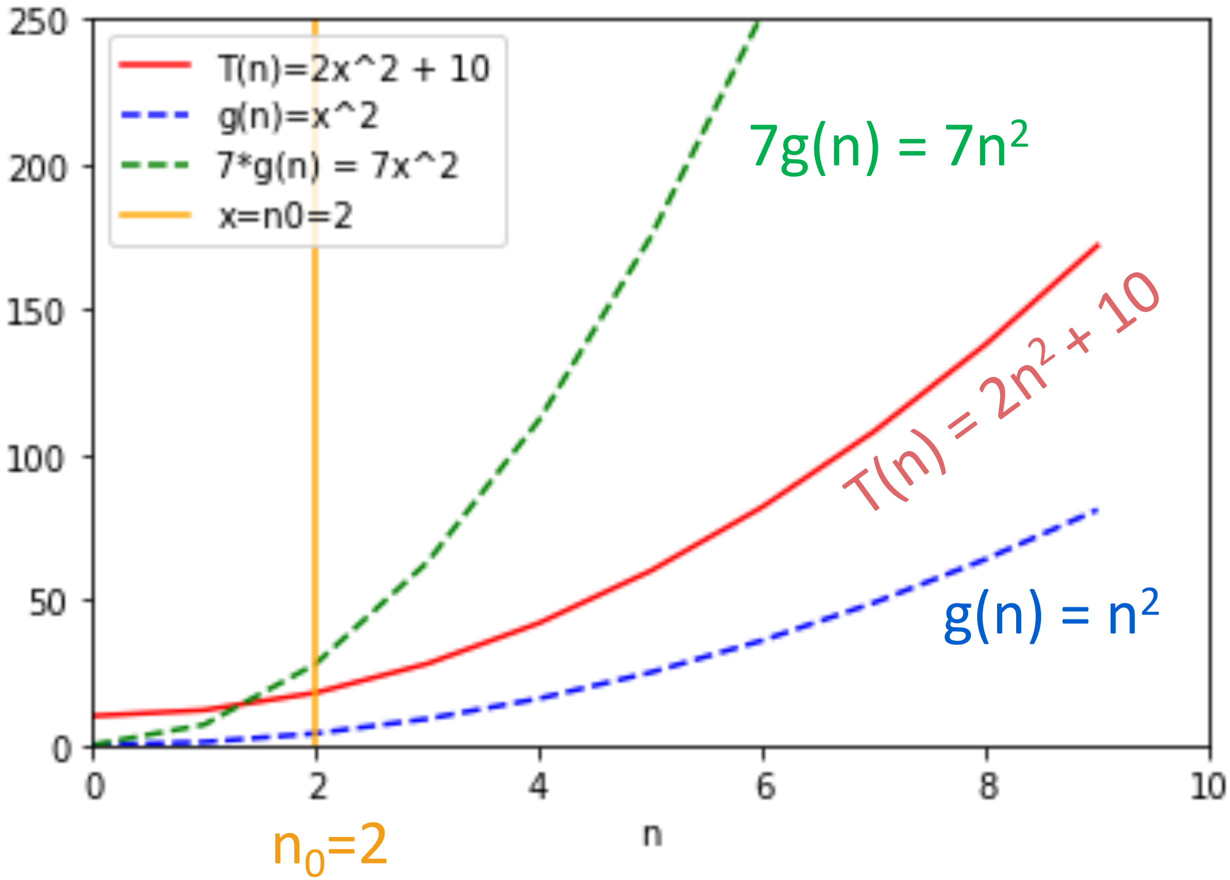


Formally:
- Choose c = 3
- Choose n_0 = 4
- Then:

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

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

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



Formally:

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

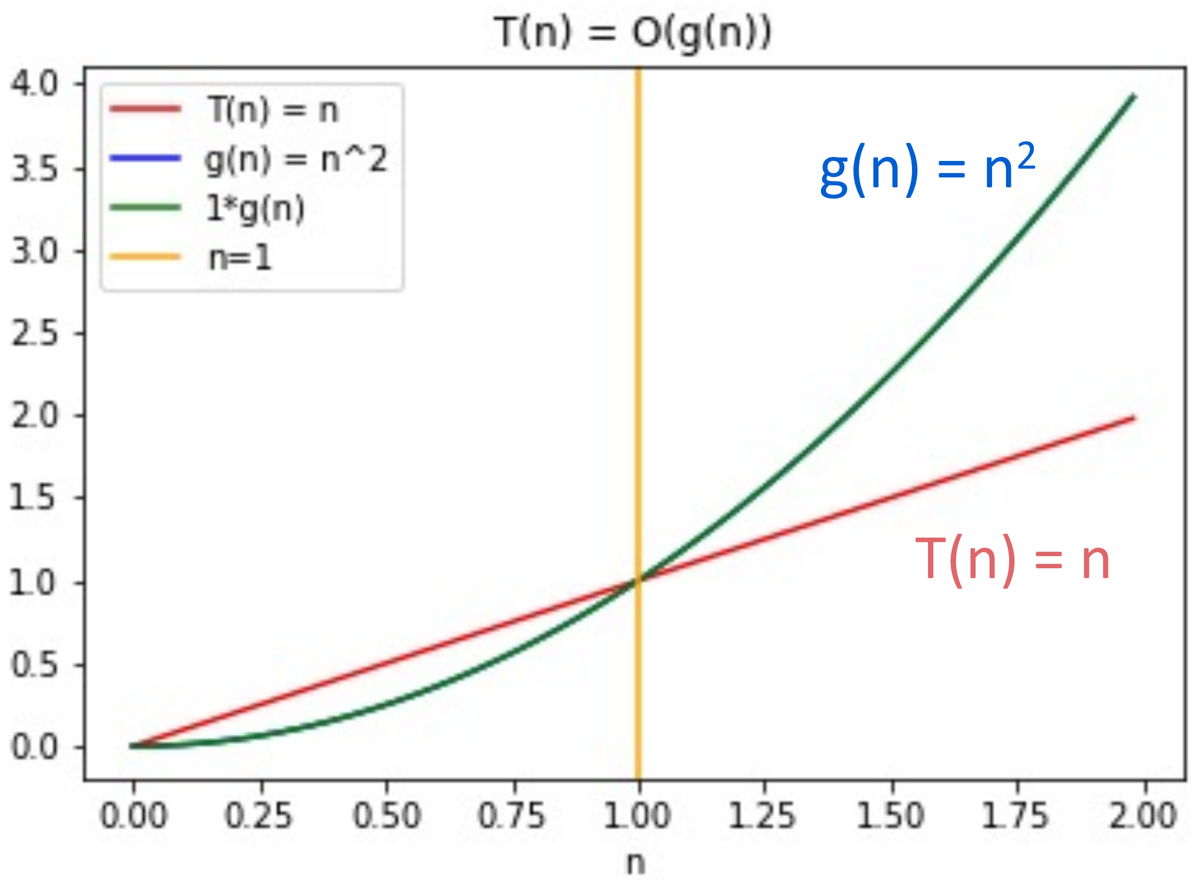
$$\forall n \geq 2,$$
$$2n^2 + 10 \leq 7 \cdot n^2$$

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

O(...) is an upper bound:

$n = O(n^2)$

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



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

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

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

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

- Formally,

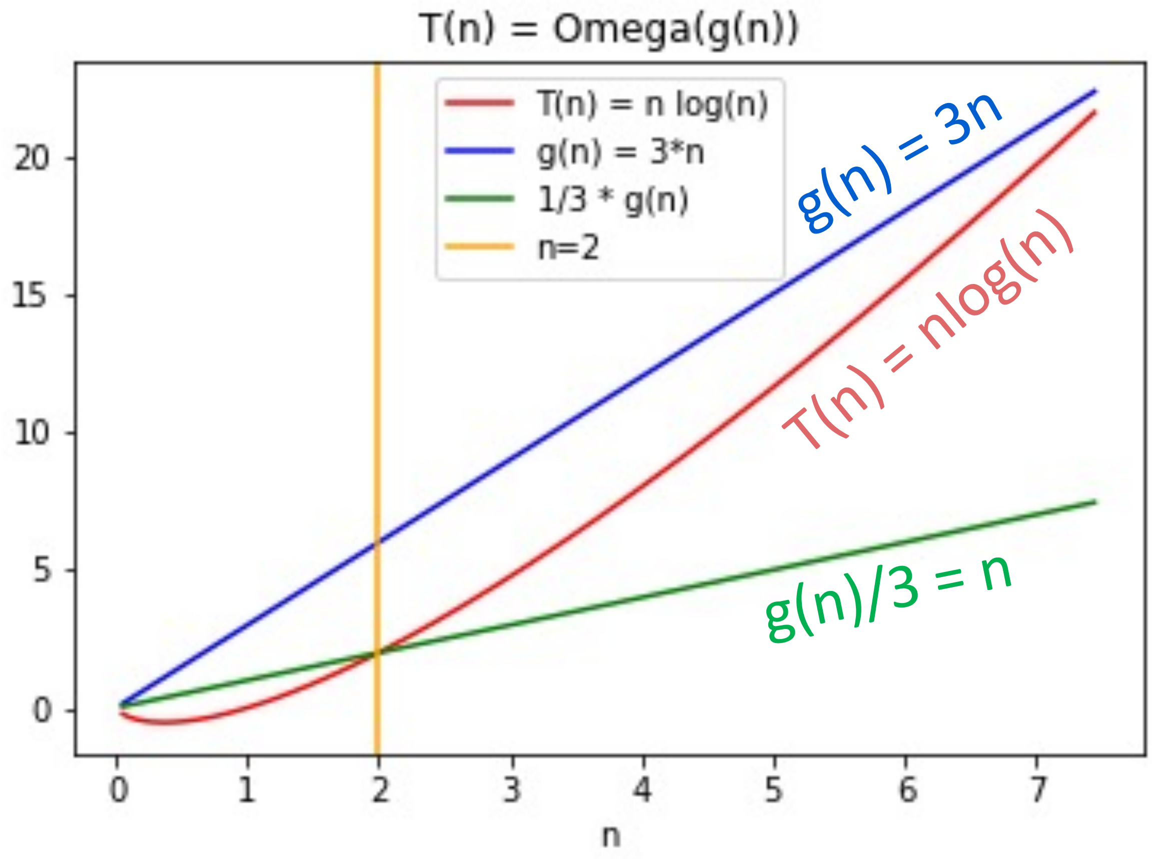
$$T(n) = \Omega(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$c \cdot g(n) \leq T(n)$$

Switched these!!

Example

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

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



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

$$\forall n \geq 2,$$

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

$\Theta(\ldots)$ means both!

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

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

and

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

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

Non-Example:
n^2 is not $O(n)$

- 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, say, $n_0 + \mathrm{c} + 1$?
  - 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 by 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 $k \geq 1$.

- 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.

Siggi the Studious Stork

SLIDE SKIPPED IN CLASS! (Note this is also in the reading).

Another example: polynomials

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

$$p(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_k n^k$$

- Then $p(n) = O\left(n^k\right)$
- Proof:
  - Choose $n_0 = 1.$
  - Choose $c = |a_0| + |a_1| + \cdots + |a_k|$
  - Then for all $n \geq n_0$:
  - $p(n) \leq |p(n)| \leq |a_0| + |a_1|n + \cdots + |a_k|n^k$
  - $\leq |a_0|n^k + |a_1|n^k + \cdots + |a_k|n^k$
  - $= c \cdot n^k$

Triangle inequality!

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

Definition of c

SLIDE SKIPPED IN CLASS! (Note this is also in the reading).

Example: more polynomials

- For any $k \geq 1, n^k$ is NOT $O\left(n^{k-1}\right)$.
- Proof:
  - Suppose that it were. Then there is some c, n_0 > 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 $n^k = O\left(n^{k-1}\right)$.

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

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?

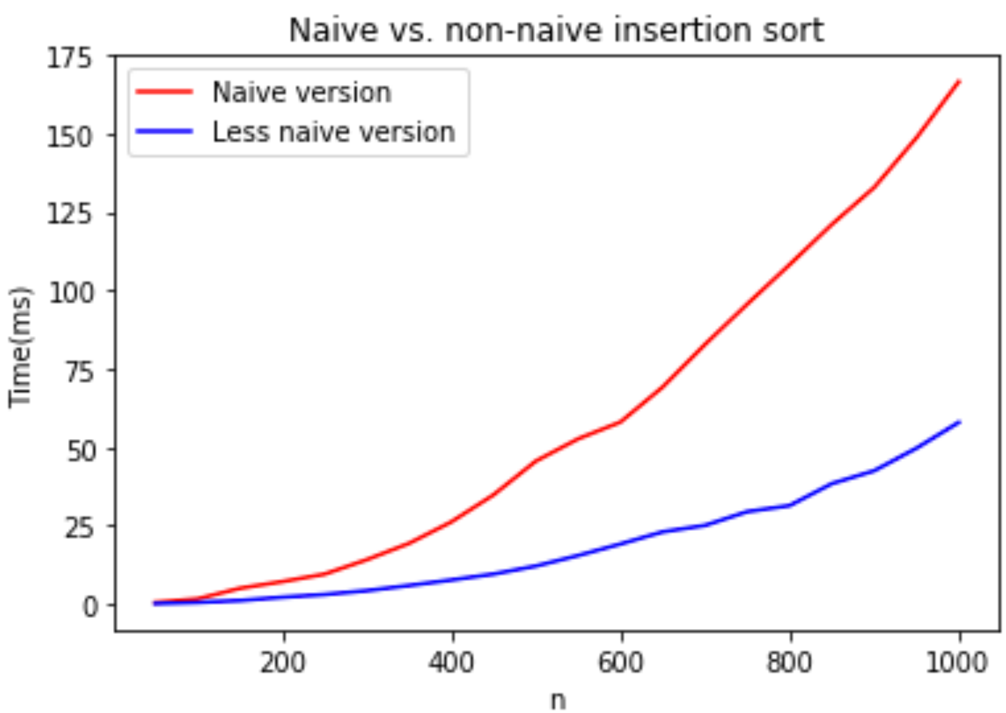
Ollie the Over-achieving Ostrich

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 actually practical, without needing to take $n \geq n_0 = 2^{10000000}$.

Back Insertion Sort

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

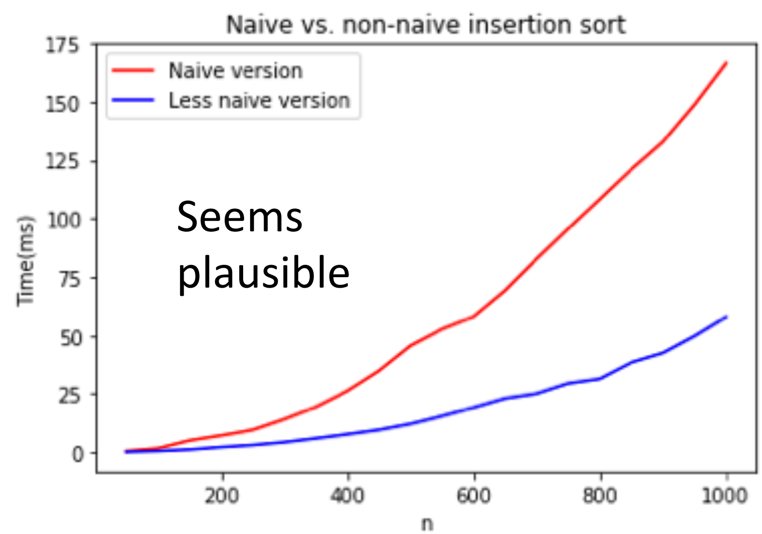


Insertion Sort: running time

- 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!



SLIDE SKIPPED IN CLASS

Insertion Sort: running time

As you get more used to this, you won't have to count up operations anymore. For example, just looking at the pseudocode below, you might think…

```
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
```

n-1 iterations of the outer loop

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

"There's O(1) stuff going on inside the inner loop, so each time the inner loop runs, that's O(n) work. Then the inner loop is executed O(n) times by the outer loop, so that's O(n^2)."

What have we learned?

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

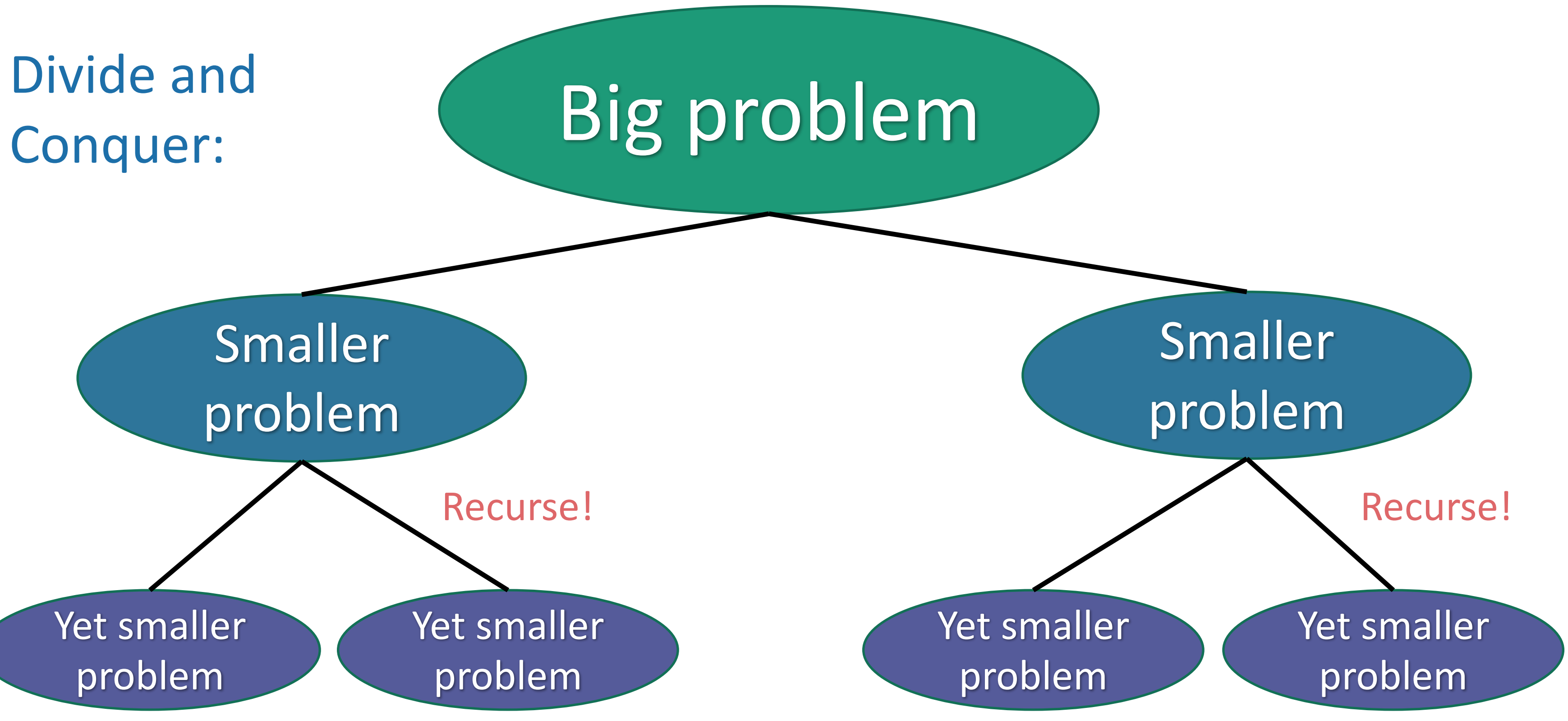
Can we do better?

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 from last time:



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?

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

Ollie the over-achieving Ostrich

See Lecture 2 IPython notebook for MergeSort Python Code.

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[0 : n/2])

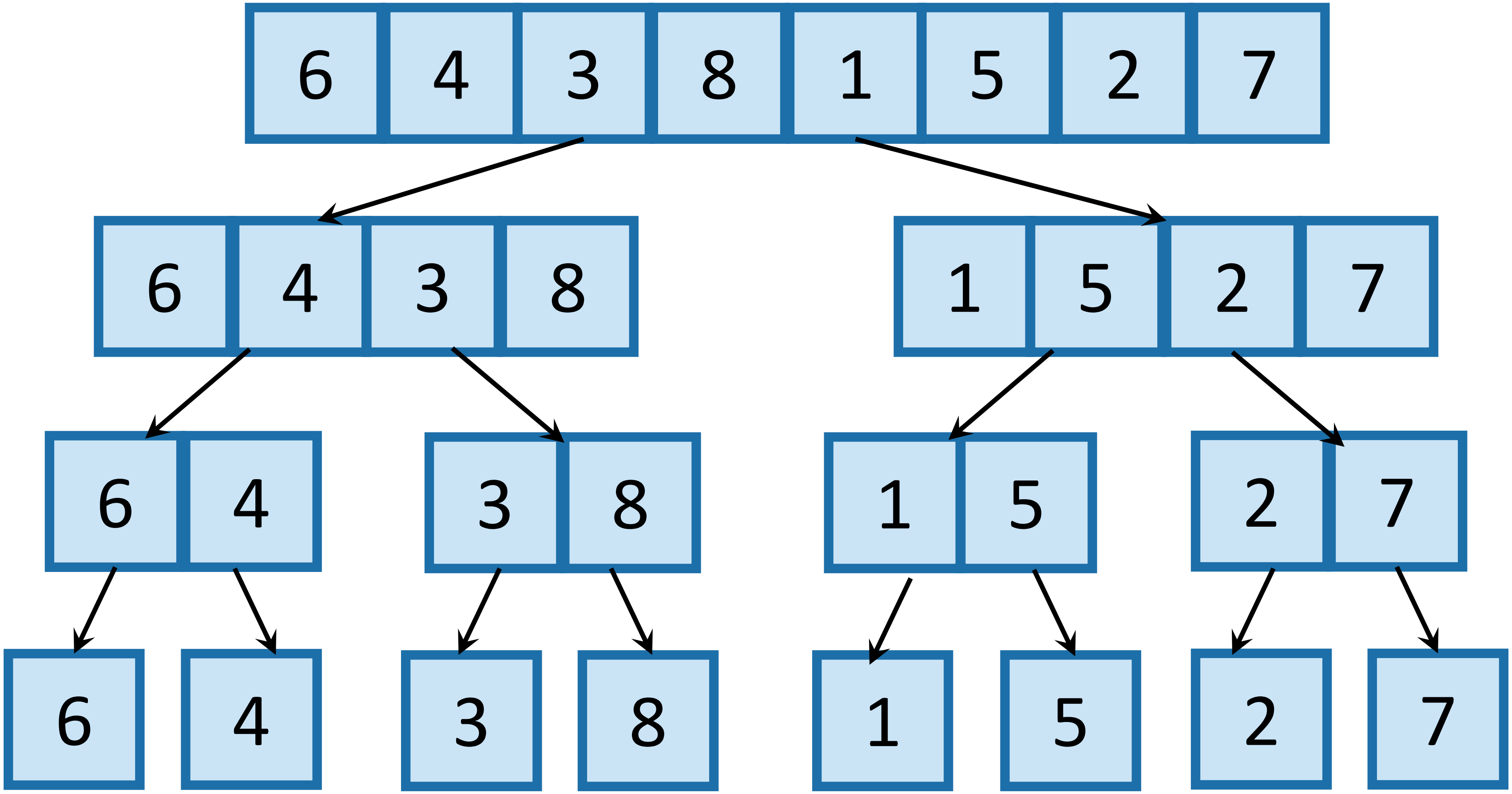
  Sort the left half
- R = MERGESORT(A[n/2 : n])

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

  Merge the two halves

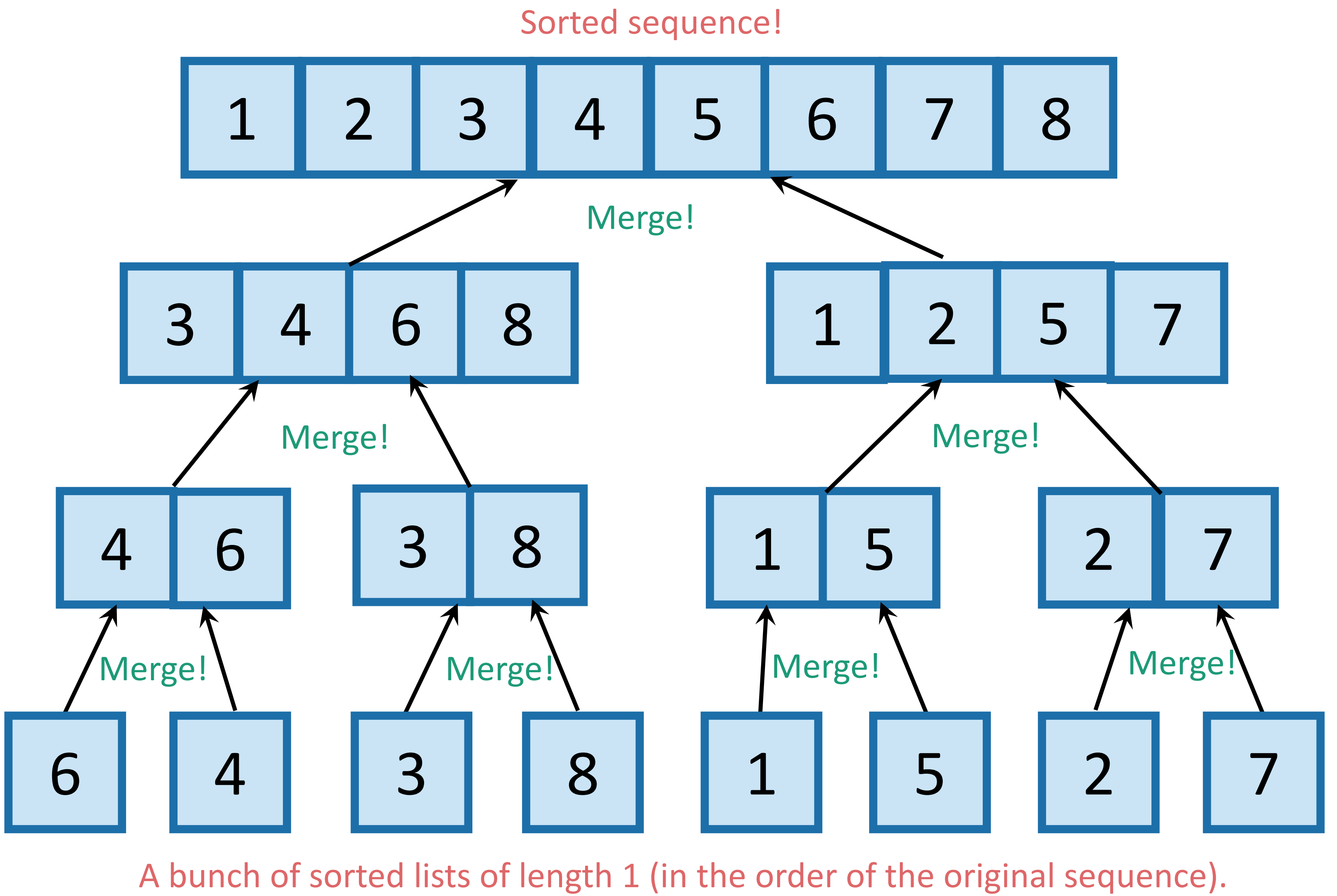
What actually happens?

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



This array of length 1 is sorted!

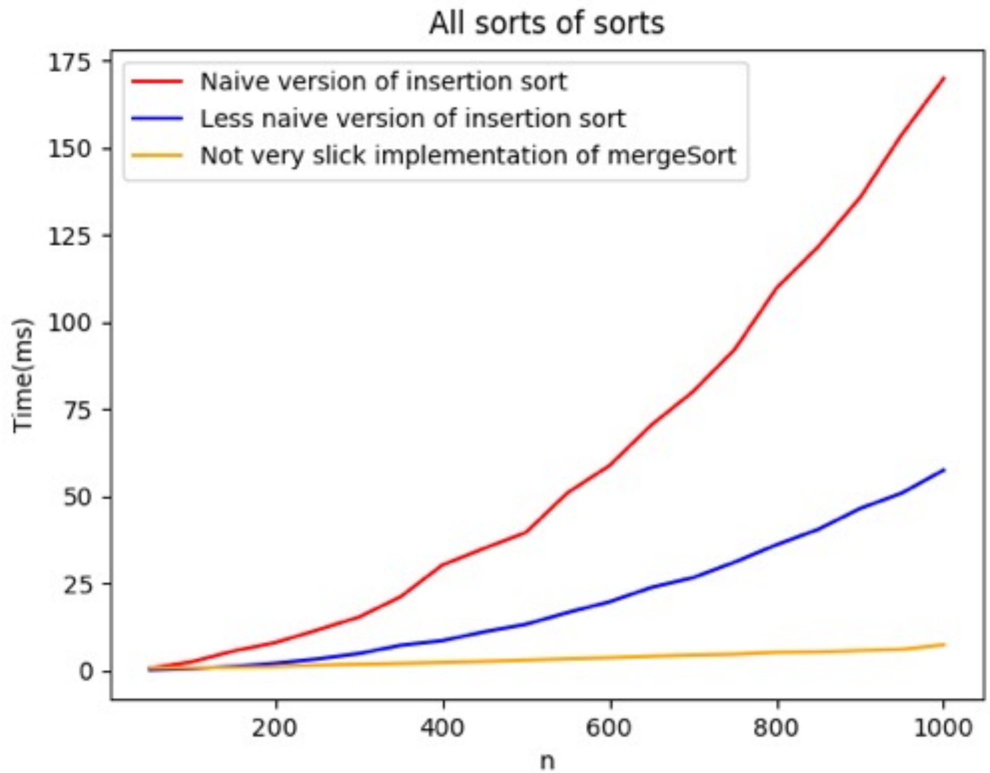
Then, merge them all back up!



Two questions

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

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.

THIS SLIDE SKIPPED IN CLASS

It works

Assume that n is a power of 2 for convenience.

- **Inductive hypothesis:**

"In every 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 the inductive hypothesis holds for k<i, then it holds for k=i.
- Aka, need to show that 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 = 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)

Fill in the inductive step!

HINT: You will need to prove that the MERGE algorithm is correct, for which you may need…another proof by induction!

Assume that n is a power of 2 for convenience.

It's fast

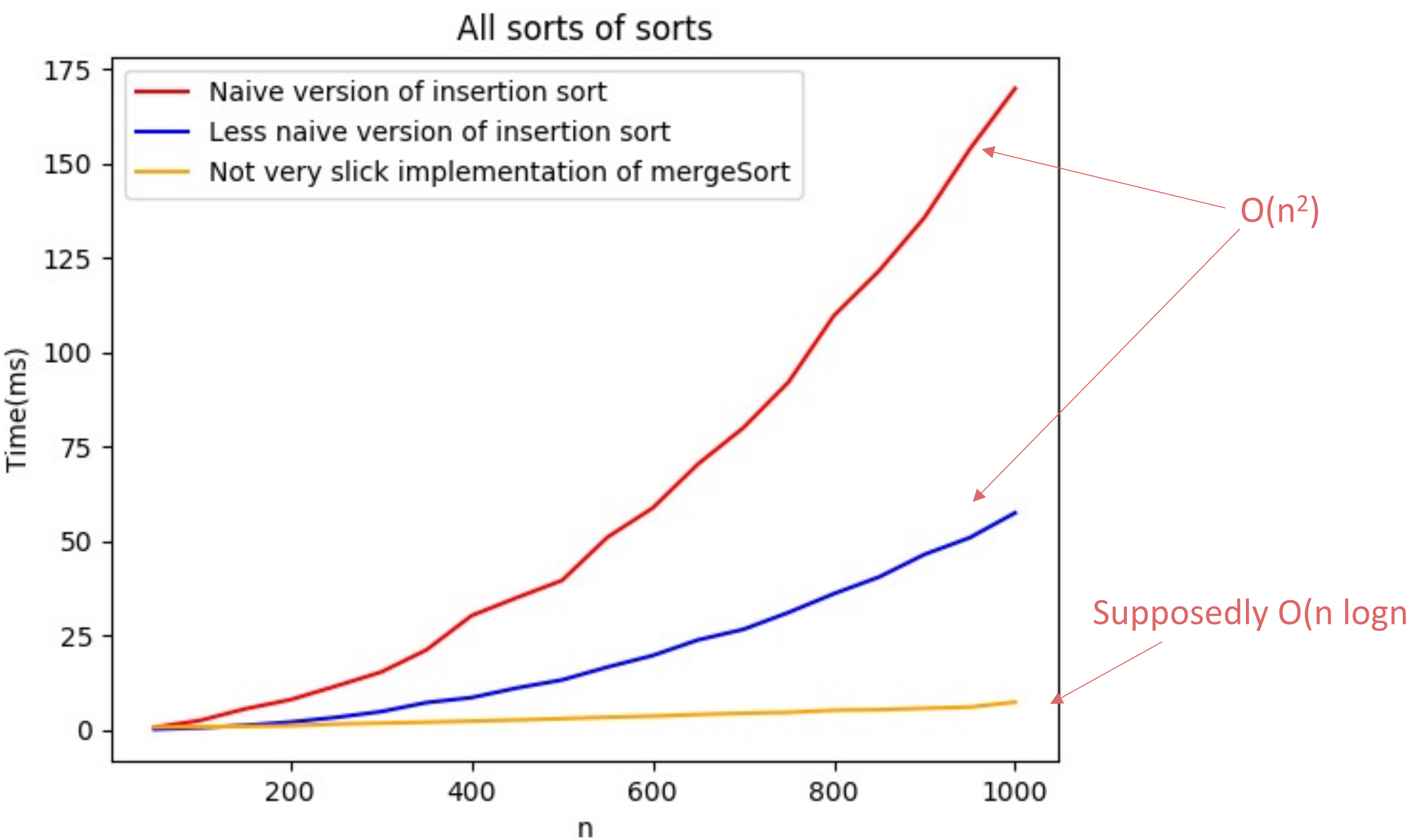
CLAIM:

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

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

[See Lecture 2 IPython Notebook for code]

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



$O(n\log(n))$ 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.

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

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

- log(n) grows very slowly!

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

$O(n \log\ n)$ 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)!

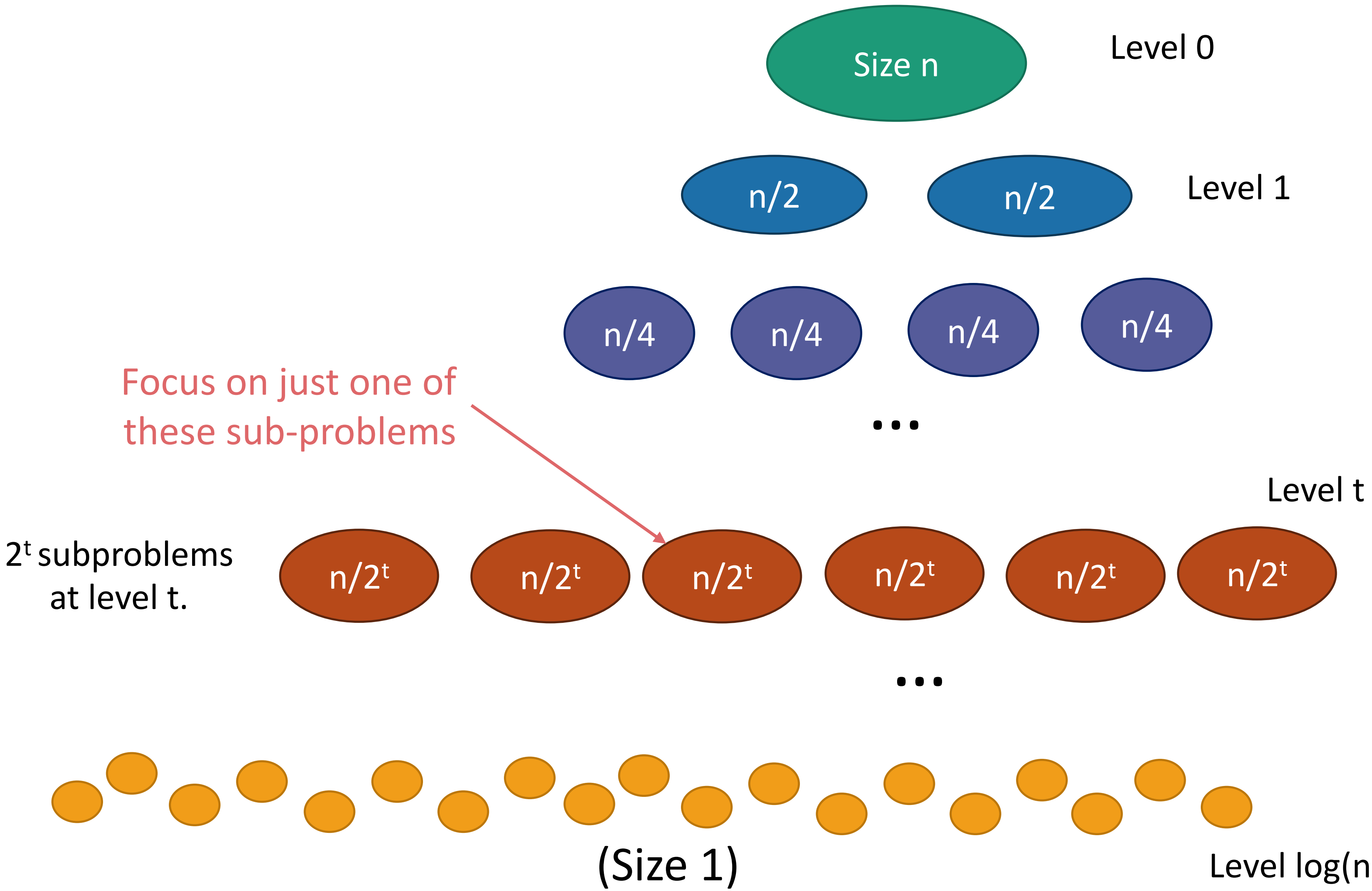
Assume that n is a power of 2 for convenience.

Now let's prove the claim

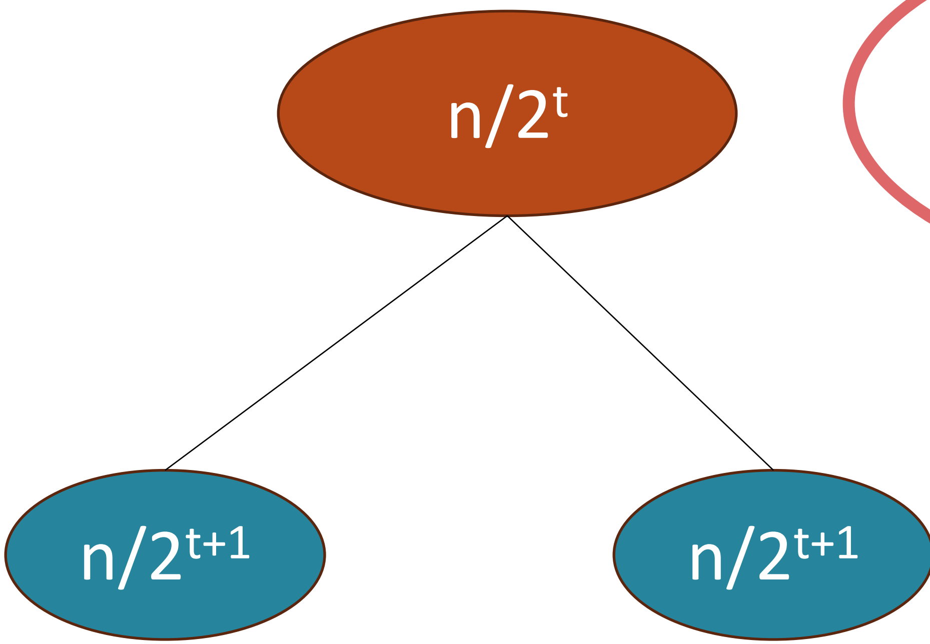
CLAIM:

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

Let’s prove the claim



How much work in this sub-problem?



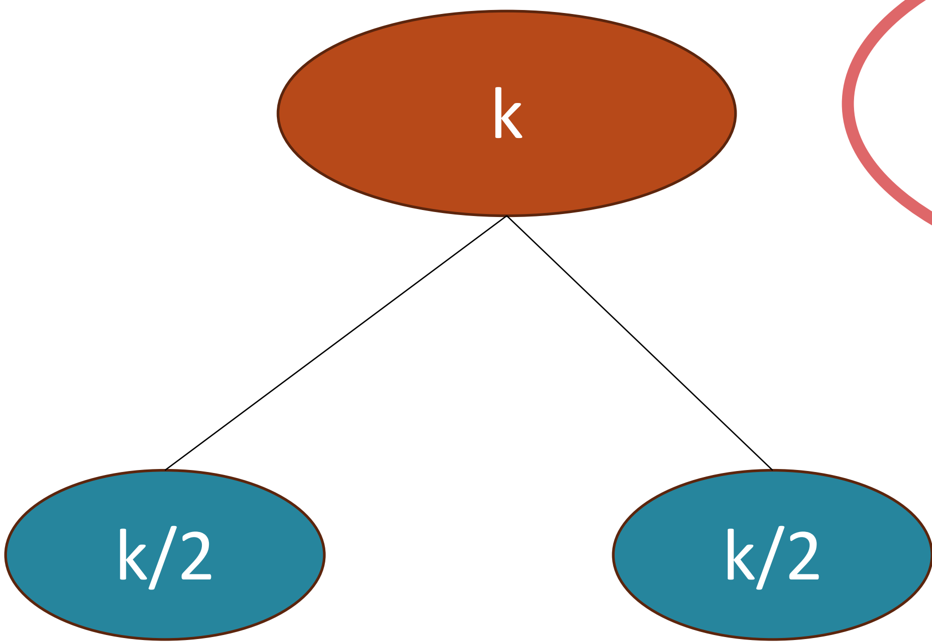
Time spent MERGE-ing the two subproblems

+

Time spent within the two sub-problems

How much work in this sub-problem?

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

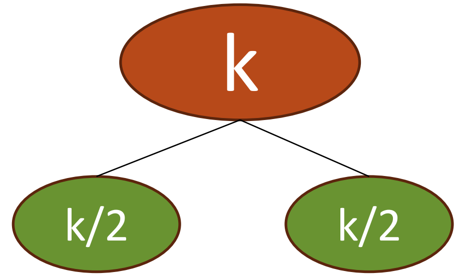


Time spent MERGE-ing the two subproblems

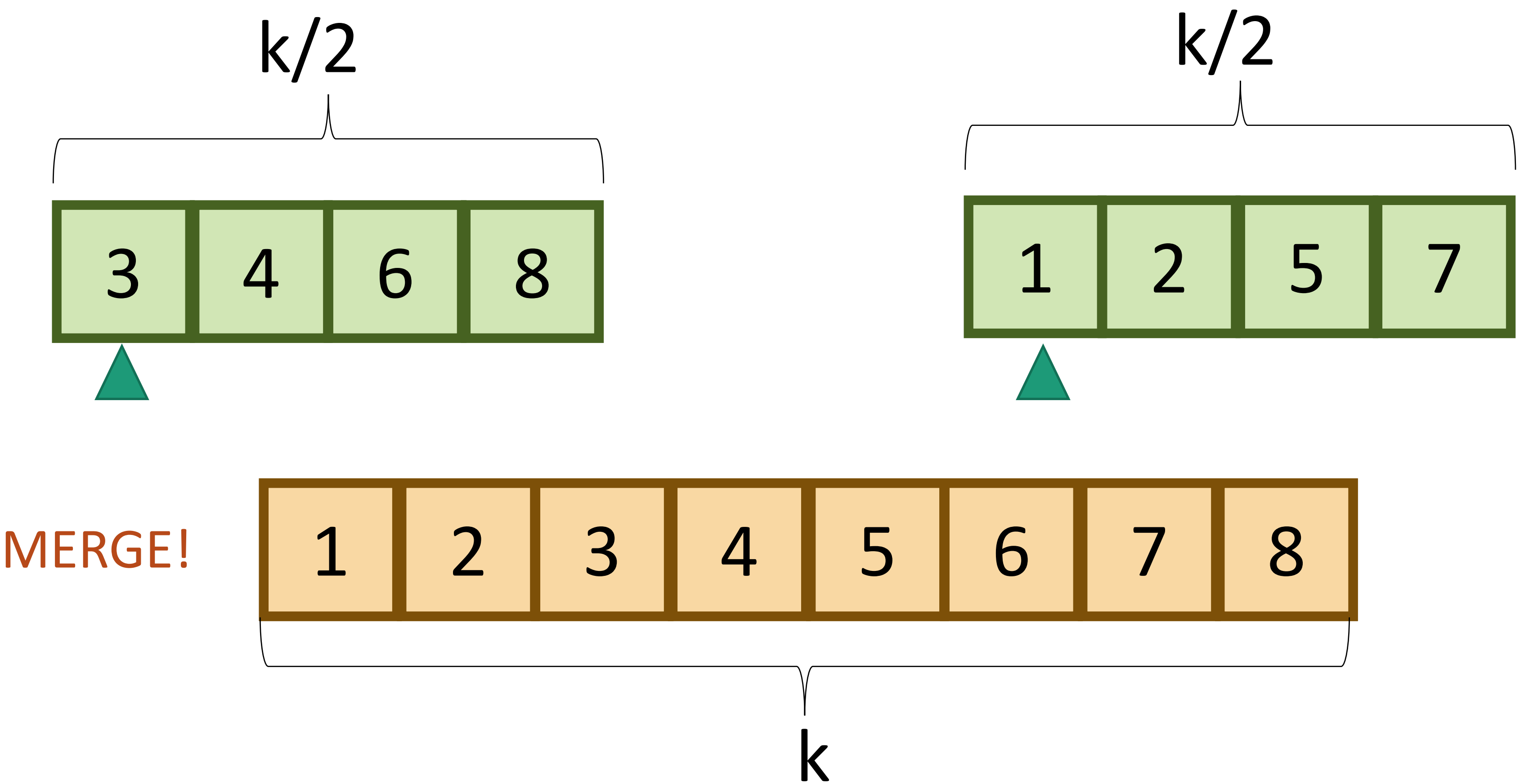
+

Time spent within the two sub-problems

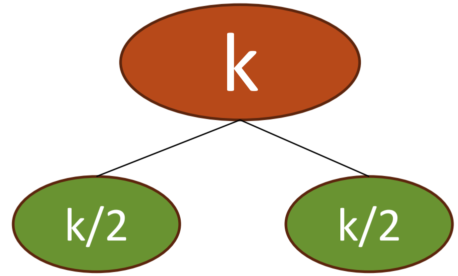
How long does it take to MERGE?



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

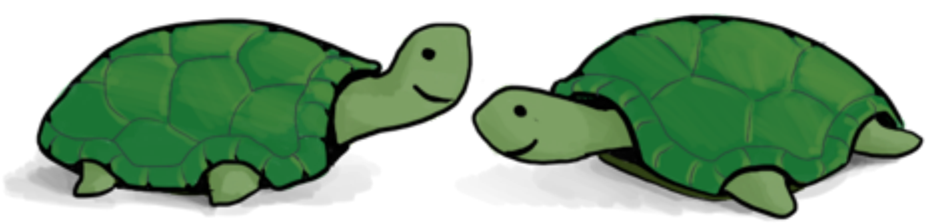


How long does it take to MERGE?



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

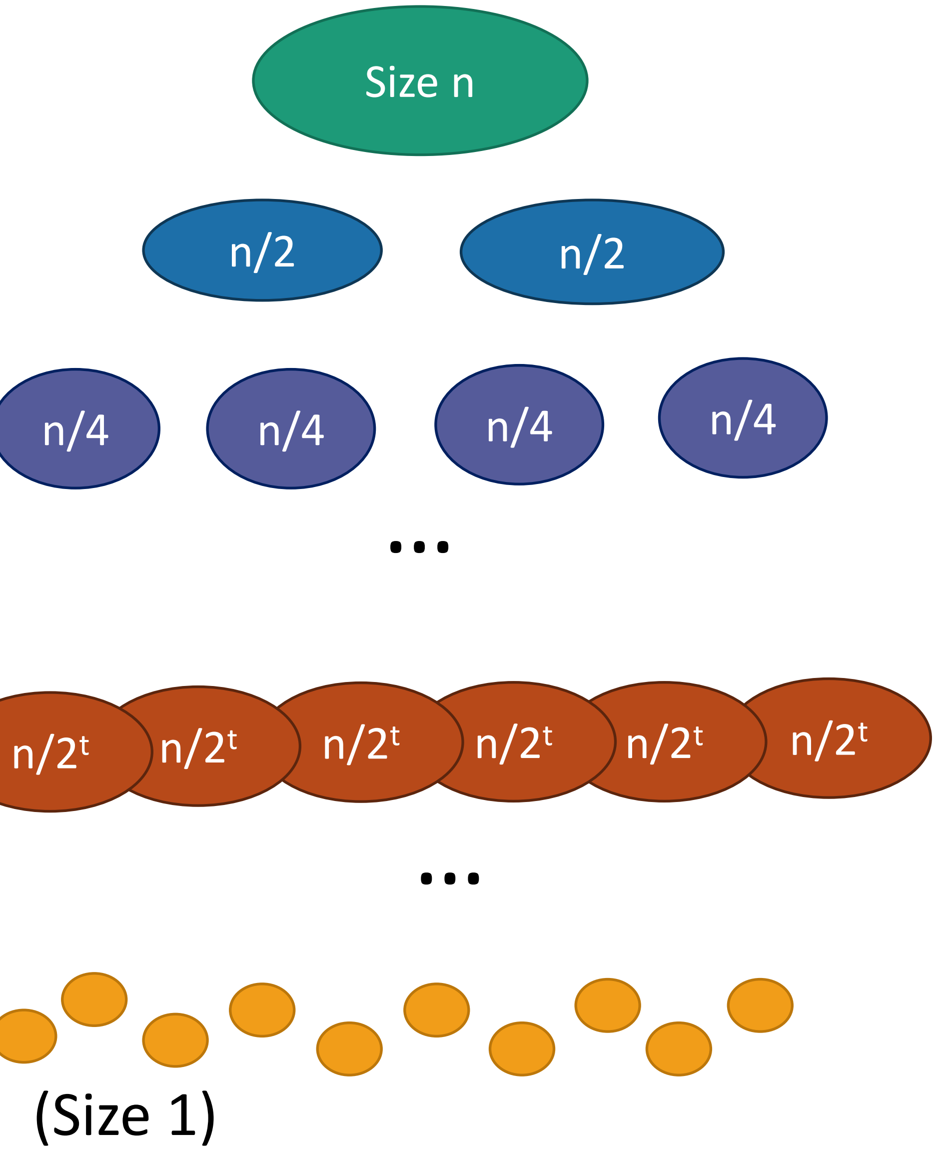
How long does it take to run MERGE on two lists of size k/2?



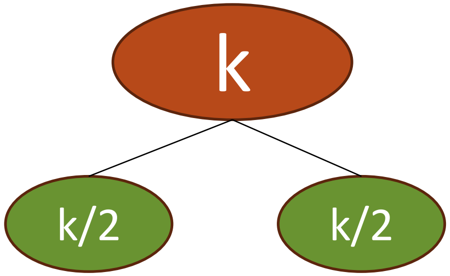
Think-Pair-Share Terrapins

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

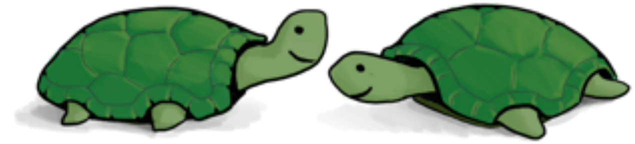
Recursion tree



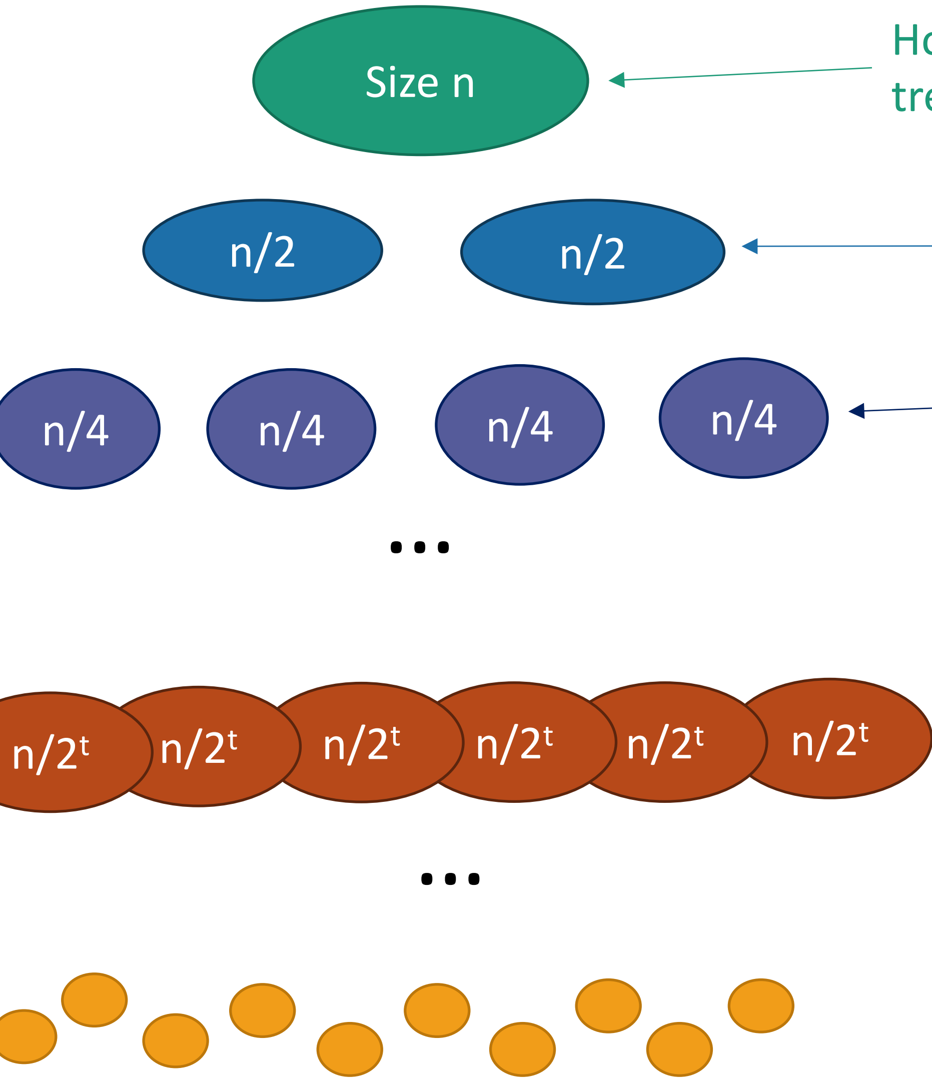
There are O(k) operations done at this node.



Recursion tree



Think, Pair, Share!



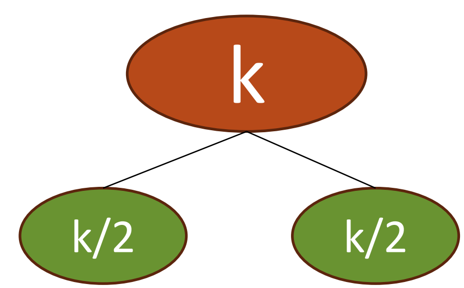
How many operations are done at this level of the tree? (Just MERGE-ing subproblems).

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

This level?

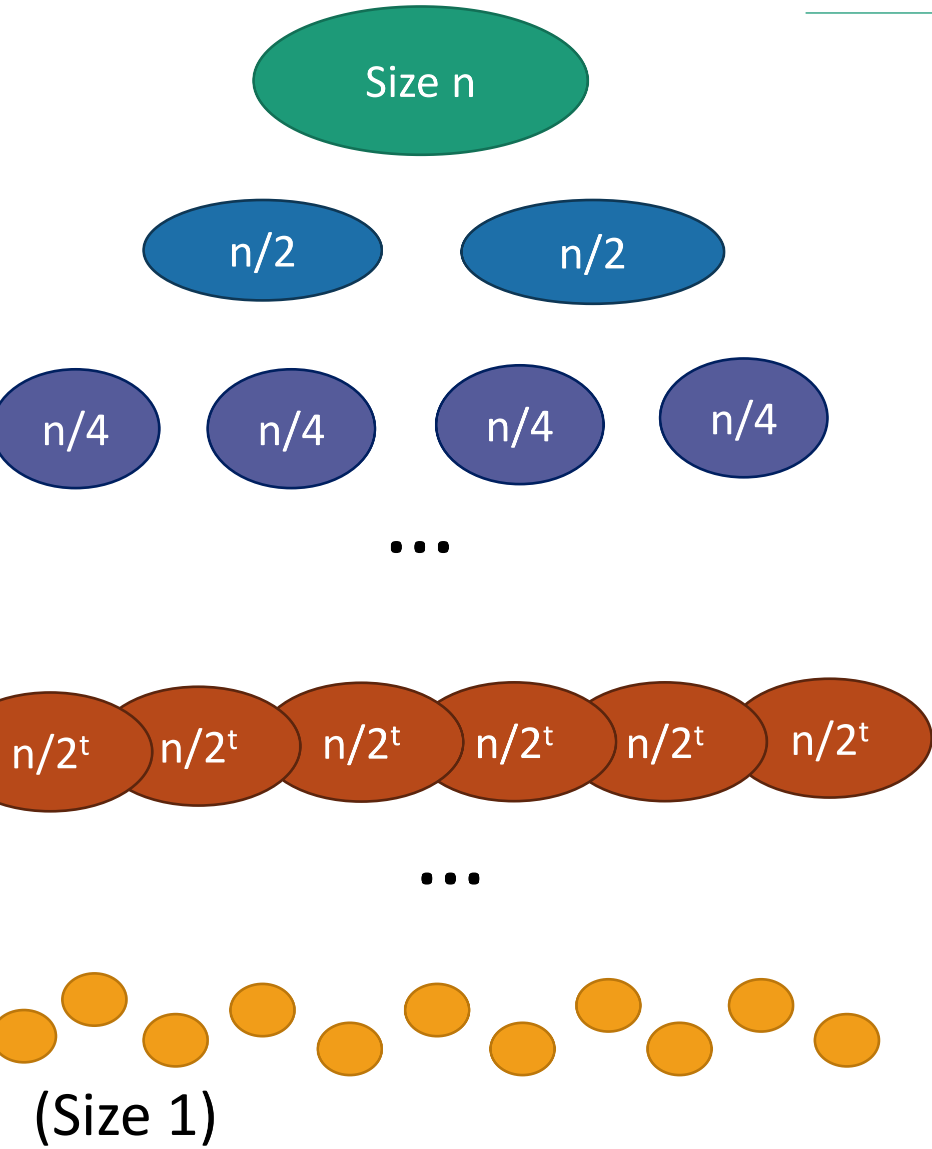
This level?

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)

BIG OMICRON AND BIG OMEGA AND BIG THETA

Donald E. Knuth
Computer Science Department
Stanford University
Stanford, California 94305

Most of us have gotten accustomed to the idea of using the notation $O(f(n))$ to stand for any function whose magnitude is upper-bounded by a constant times $f(n)$, for all large n. Sometimes we also need a corresponding notation for lower-bounded functions, i.e., those functions which are at least as large as a constant times $f(n)$ for all large n. Unfortunately, people have occasionally been using the O-notation for lower bounds, for example when they reject a particular sorting method "because its running time is $O(n^2)$." I have seen instances of this in print quite often, and finally it has prompted me to sit down and write a Letter to the Editor about the situation.