Lecture 2

Asymptotic Notation,

Worst-Case Analysis, and MergeSort

Announcements

- HW0 Due Tuesday!
- HW1 out soon – due a week from Friday.
- Homework Party!
  - Monday 5:30-7:30pm, CoDa Basement
  - There will be food! 🍕🍕🍕
- Sections!
  - Start today! Check out website for schedule/locations
  - One section will be recorded!
- Check the exam schedule and your calendar.
  - If you cannot take the exams, you should not take this class!

Last time

Philosophy

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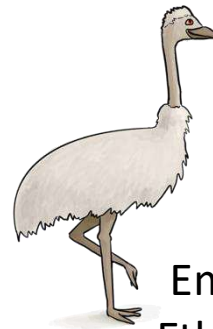 - Can I do better?

Technical content

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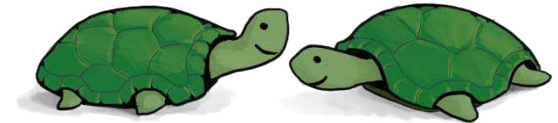- Example of "Divide and Conquer"
- Not-so-rigorous analysis
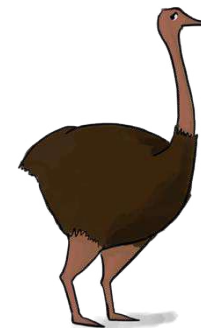
Cast

Plucky the pedantic penguin

Lucky the lackadaisical lemur

Emery the Ethical Emu

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.
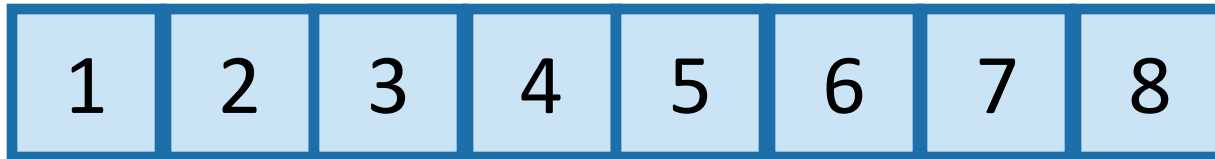 - InsertionSort
  - MergeSort

SortingHatSort not discussed

The Plan

- Sorting!
- Worst-case analyisis
  - 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
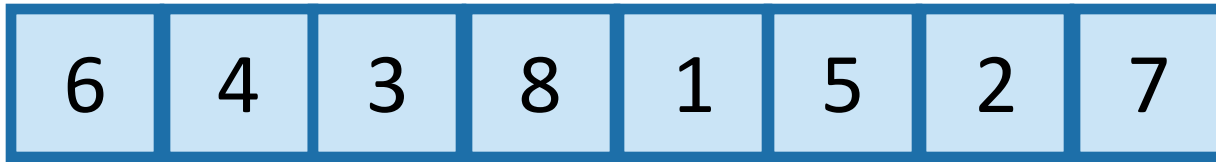
Length of the list is n

I hope everyone did the pre-lecture exercise!

What was the mystery sort algorithm?

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

```python
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
```

```python
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

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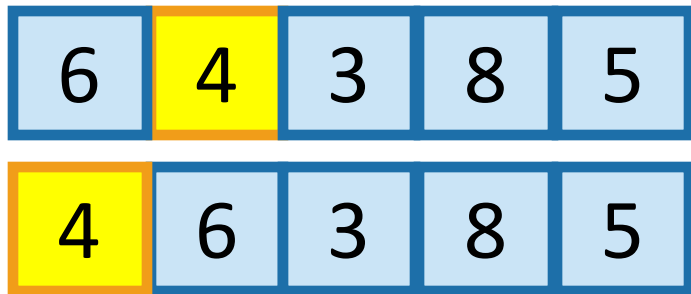
```python
def MysteryAlgorithmTwo(A):
    for i in range(1,len(A)):
        current = A[i]
        j = i-1
        while j >= 0 and A[j] > current:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = current
```

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
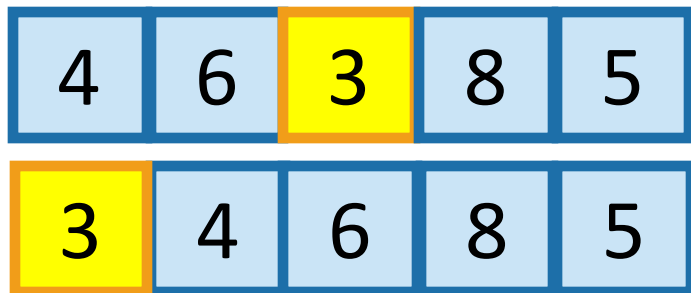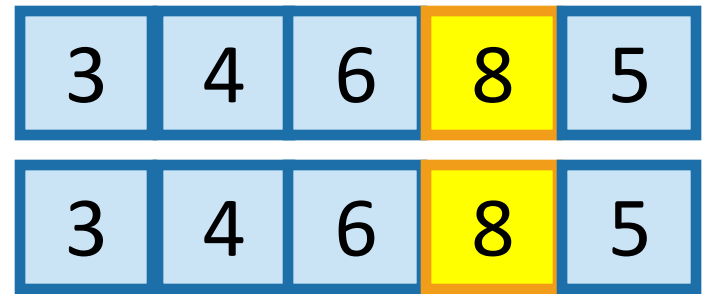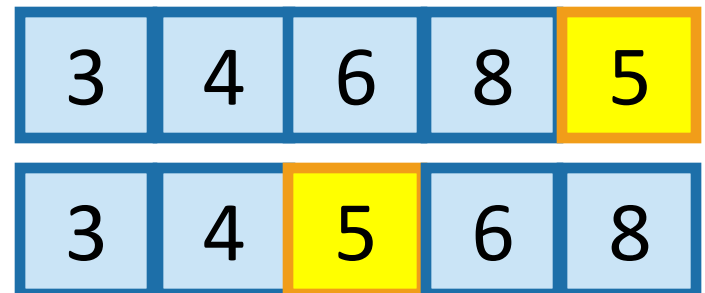
Then we are done!

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!
```

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:

Worst-case analysis guarantee:
`Algorithm` should work (and be fast) on that worst-case input.

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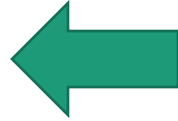
1. Does it work?
2. Is it fast?

- 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, | 3 | 4 | 6 | 8 | , and another element | 5 | .

- Insert | 5 | right after the largest thing that's still smaller than | 5 | . (Aka, right after | 4 |).

- Then you get a sorted list: | 3 | 4 | 5 | 6 | 8 |

So just use this logic at every step.

| 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'll do the proof by maintaining a *loop invariant,* in this case that after iteration i, then A[:i+1] is sorted. This is obviously true 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 this loop invariant holds for $k-1$, then it holds for k. That is, if it is true that A[:k] is sorted after the $k-1$'st iteration, then it is true that A[:k+1] is sorted after the k'th iteration. At the end of the day, we'll conclude that A[:n] (aka, the whole thing) is sorted after the $n-1$'st iteration, and we'll be done.

Formally, we will proceed by induction.

- **Inductive hypothesis.** After iteration i of the outer loop, A[:i+1] is sorted.

- **Base case.** After iteration 0 of the outer loop (aka, before the algorithm begins), the list A[:1] contains only one element, and this is sorted.

- **Inductive step.** Let k be an integer so that $0 < k < n$. Suppose that the inductive hypothesis holds for $k-1$, so A[:k] is sorted after the $k-1$'st iteration. We want to show that A[:k+1] is sorted after the k'th iteration.

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

  $$[A[0], A[1], \ldots, A[j^*], \ldots, A[k-1], A[k]]$$

  into

  $$[A[0], A[1], \ldots, A[j^*], A[k], A[j^* + 1], \ldots, A[k-1]].$$

Outline of a proof by induction

Let A be a list of length n

- Inductive Hypothesis:

- Base case:

- Inductive step:

- Conclusion:

| 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.

Outline of a proof by induction

Let A be a list of length n

- Inductive Hypothesis (inducting on i):
  - *A[:i+1] is sorted at the end of the ith iteration* (of the outer loop).

- Base case:

- Inductive step:

- Conclusion:

| 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.

Outline of a proof by induction

Let A be a list of length n

- Inductive Hypothesis (inducting on i):
  - *A[:i+1] is sorted at the end of the ith 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.
  - `A[:n]` *is sorted at the end of the n-1'st iteration*
  - Aka, A is sorted at the end of the algorithm! ✓

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 Office Hours / Section
  - Lecture 2 Handout
  - Book (appendix A)
  - Ian Tullis's notes (in the Resources tab)
  - 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 office hours if you have questions!

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?

I dunno…let's just try it out!

- This fast:

Naive vs. non-naive insertion sort
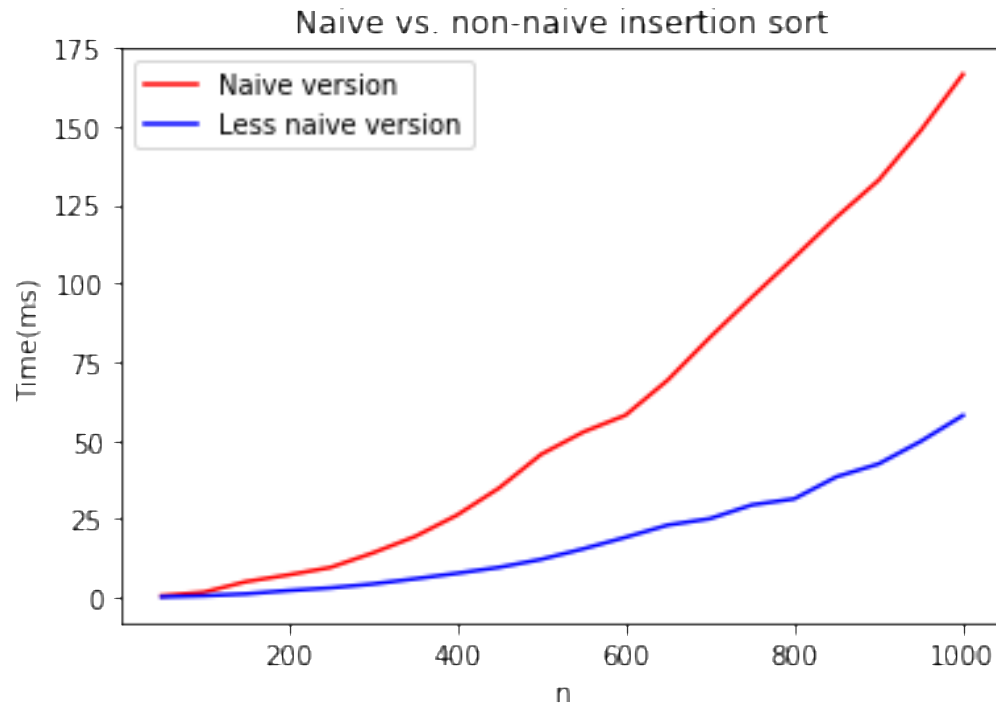
Issues with this answer?

- 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.

- It might also be slower or faster depending on the inputs we use.

- What if n=2000?

Naive vs. non-naive insertion sort

Naive version
Less naive version

Time(ms)

175
150
125
100
75
50
25
0

200   400   600   800   1000

n
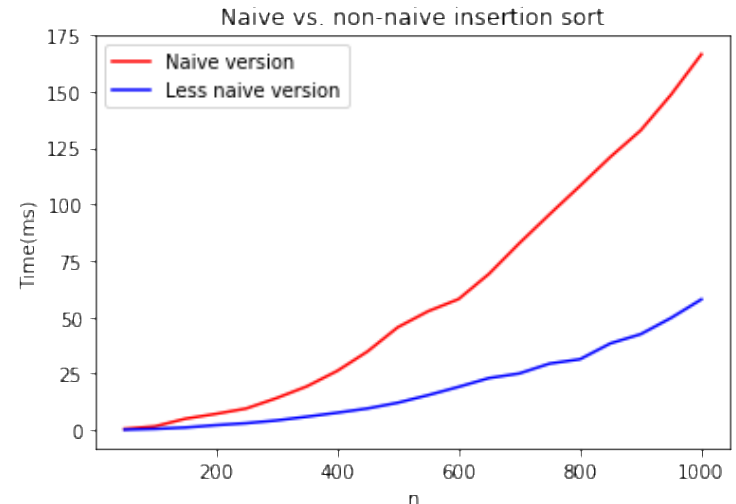
With this answer, "running time" isn't even well-defined!

How fast is InsertionSort?

Let's count the number of operations!

```python
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!

- Might be slightly different for slightly different implementations.

- 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
$\frac{1}{10} \cdot n^2 + 100$	$O(n^2)$
$0.063 \cdot n^2 - .5\,n + 12.7$	$O(n^2)$
$100 \cdot n^{1.5} - 10^{10000}\sqrt{n}$	$O(n^{1.5})$
$11 \cdot n\log(n) + 1$	$O(n\log(n))$
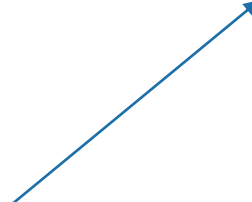
We say this algorithm is "asymptotically faster" than the others.

Why is this a good idea?

- Suppose the running time of an algorithm is:

$$T(n) = 10n^2 + 3n + 7 \text{ ms}$$
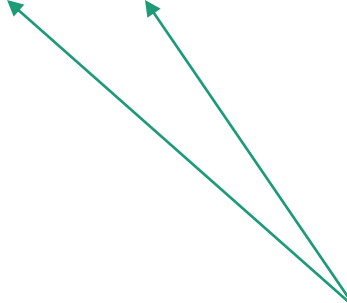
This constant factor of 10 depends a lot on my computing platform…

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

We're just left with the n² 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 ?!?!

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 $g(n)$.

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

$g(n) = n^2$

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 $g(n)$.

$3g(n) = 3n^2$

$n = 4$

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

$g(n) = n^2$
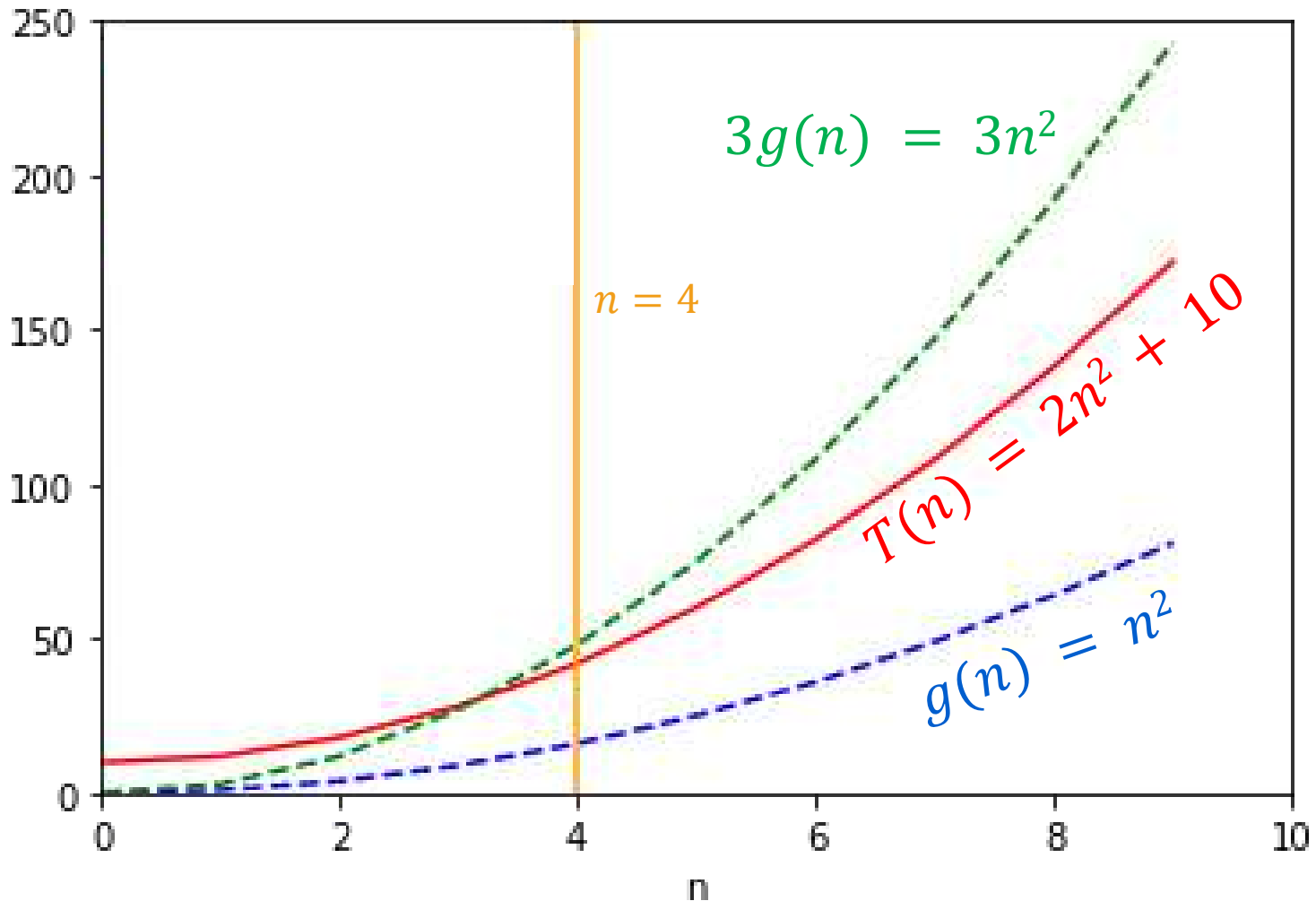
Formal definition of O(...)

- 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" $\Longrightarrow$ $\Longleftrightarrow$ "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

$$2n^2 + 10 = O(n^2)$$
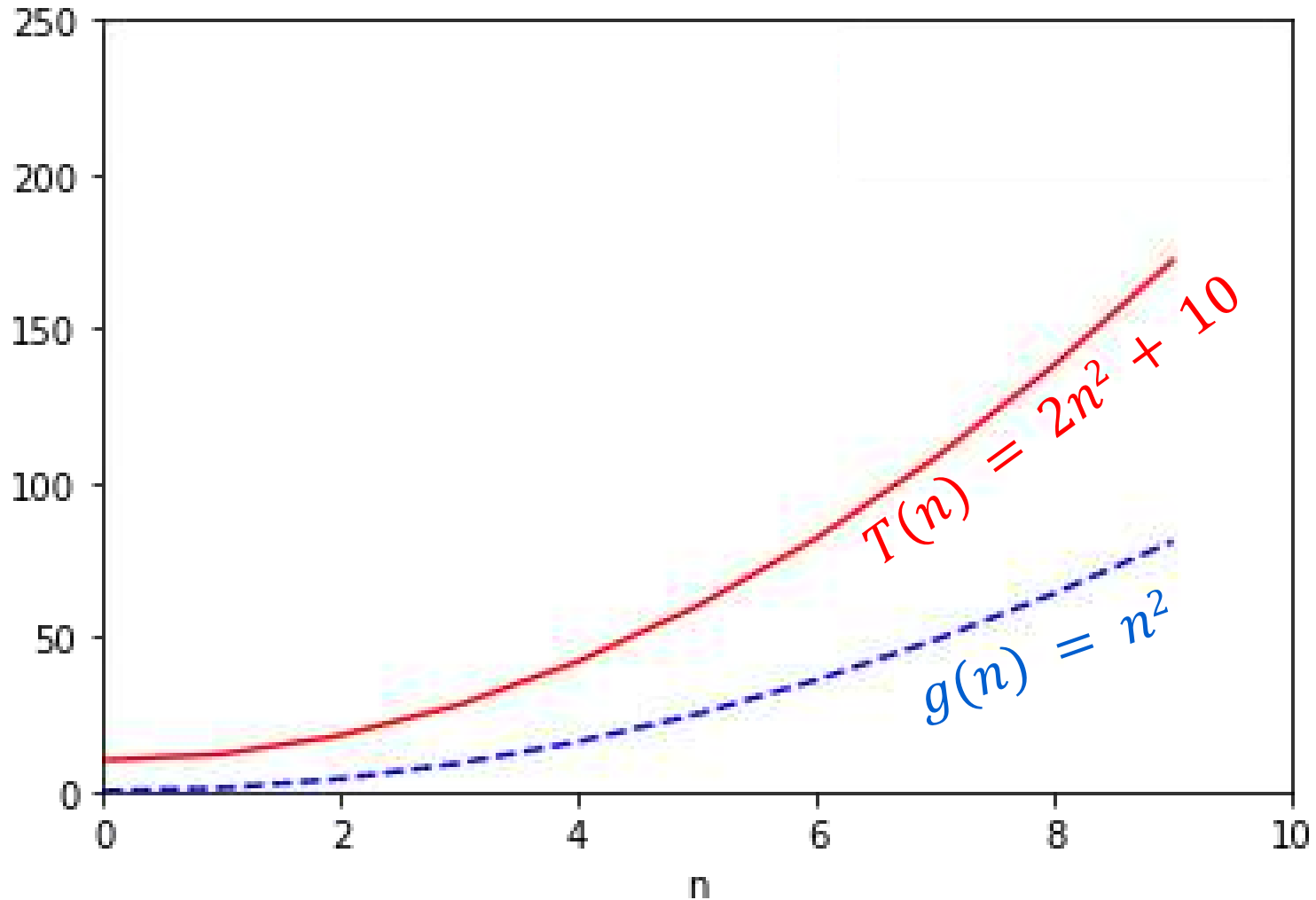
$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
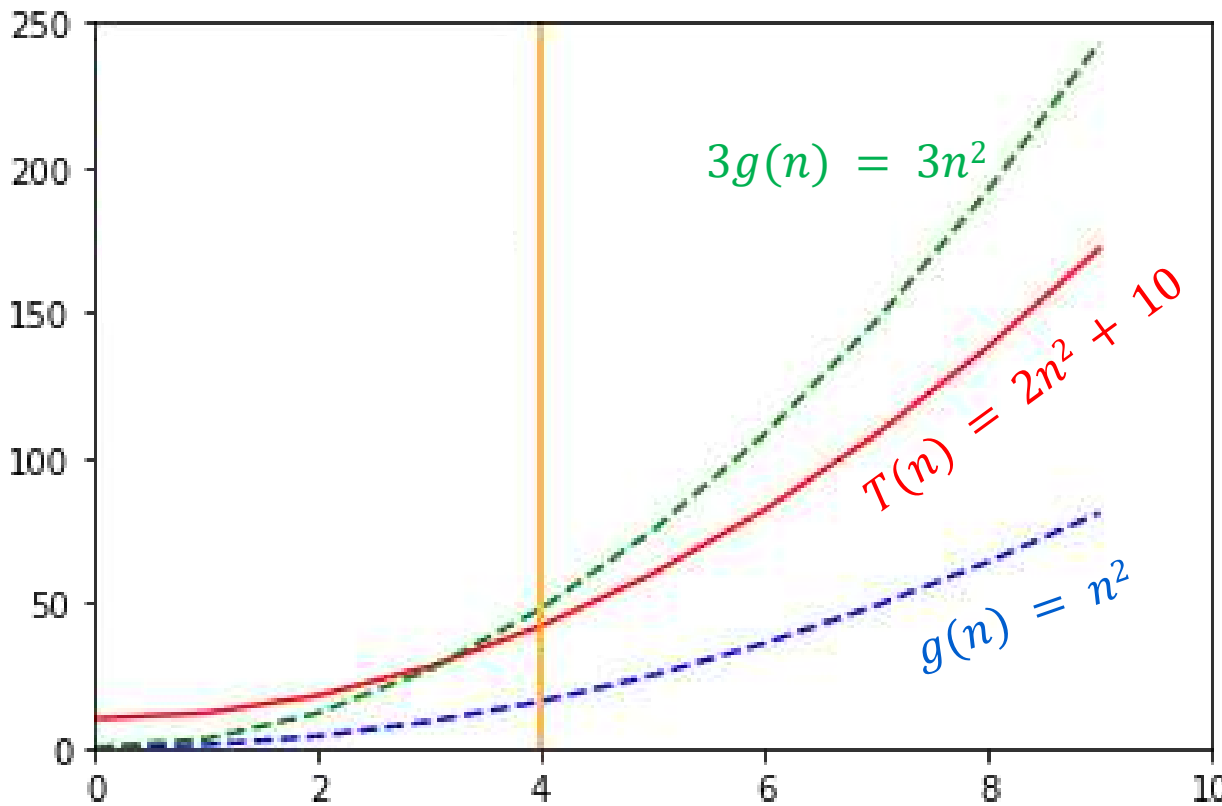$$\exists c, n_0 > 0 \ s.t. \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

Example

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

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

$3g(n) = 3n^2$
(Choose c=3)

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

$g(n) = n^2$

Example

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

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

(Choose $n_0=4$)

$3g(n) = 3n^2$

(Choose c=3)

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

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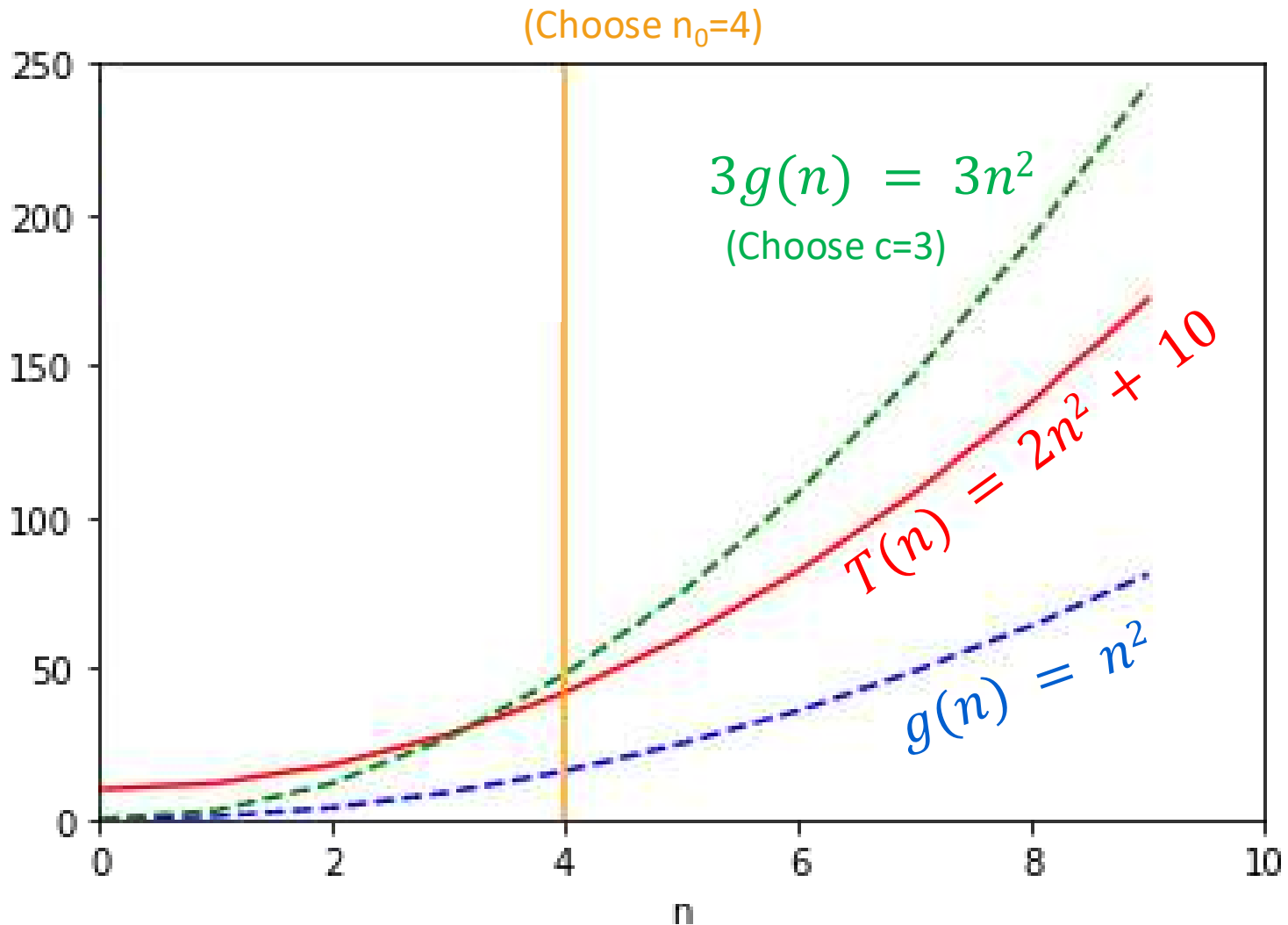
Example
$2n^2 + 10 = O(n^2)$

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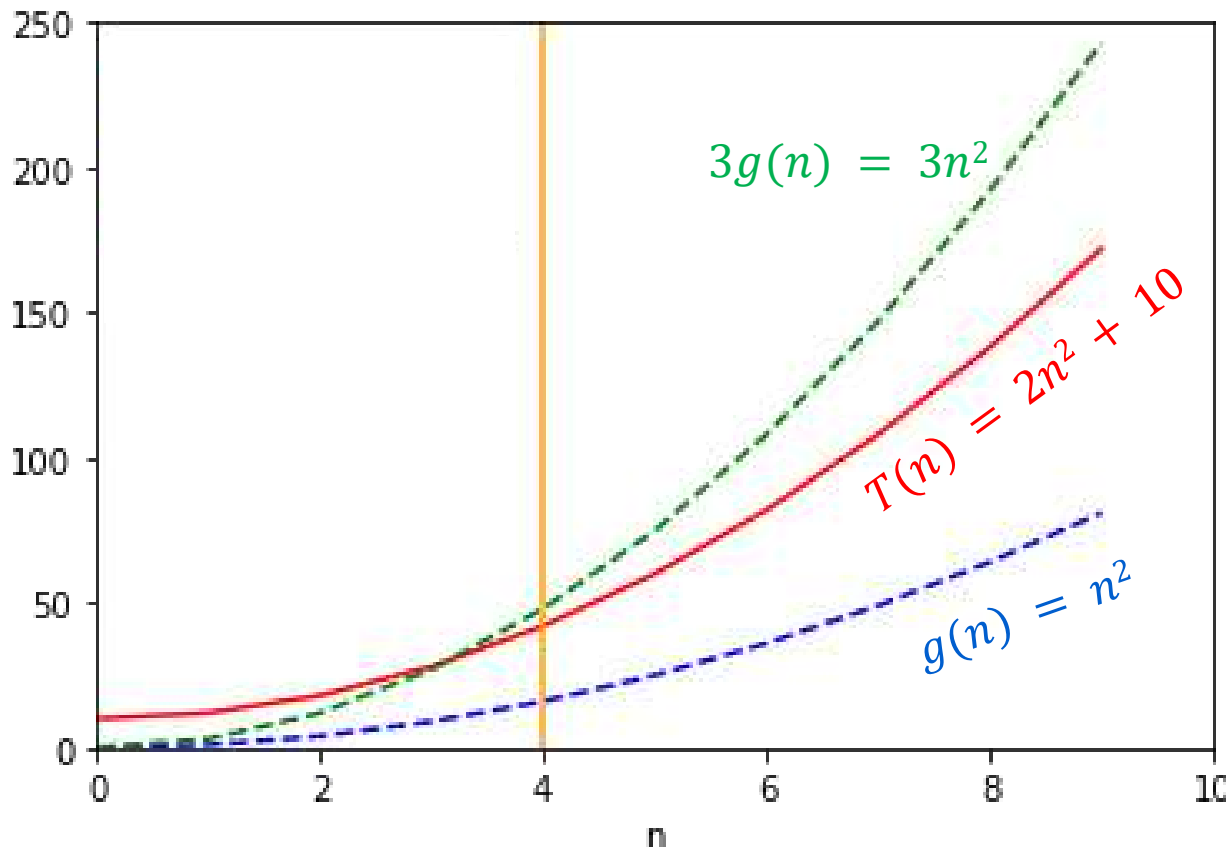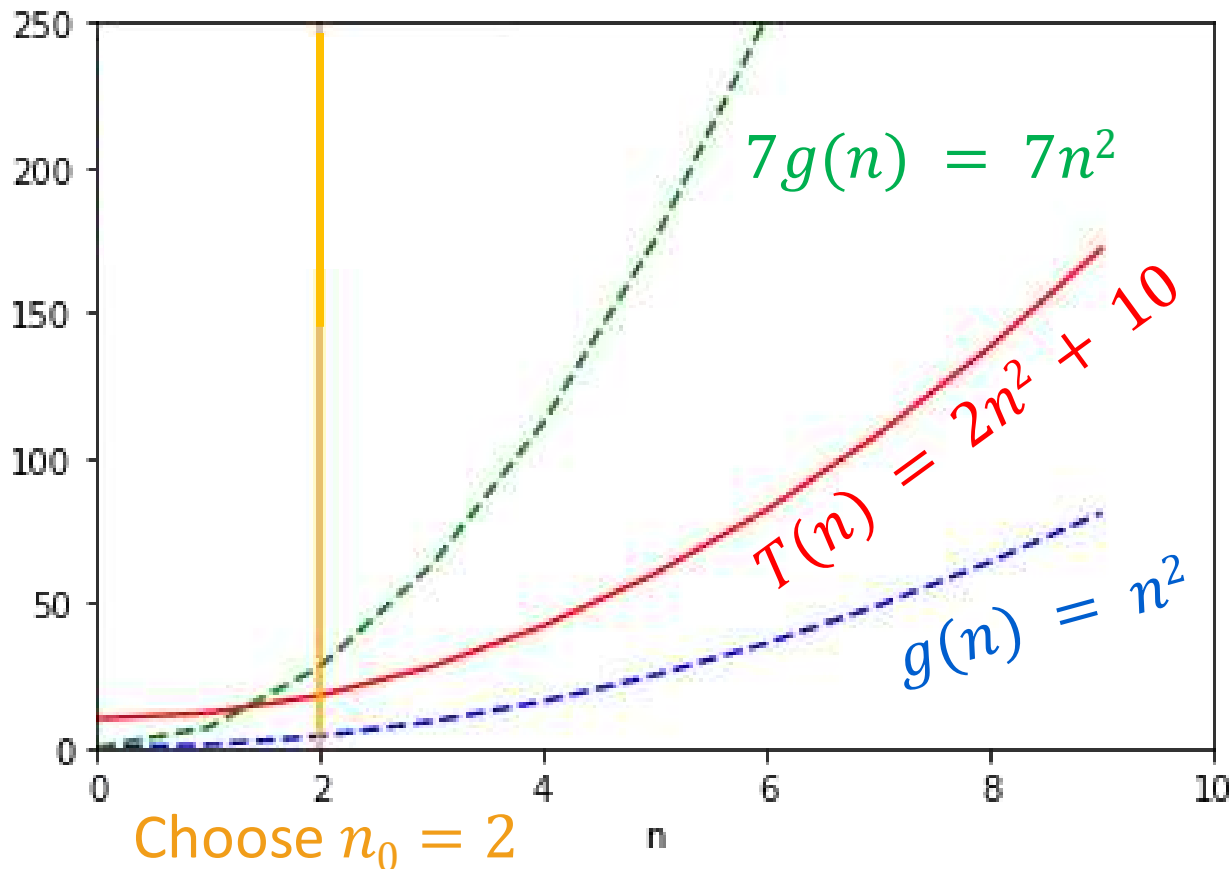$$T(n) \leq c \cdot g(n)$$

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

$$\forall n \geq 4,$$

$$2n^2 + 10 \leq 3n^2$$

We can see that this is true on the graph, or prove it with algebra:

$\forall n \geq 4,$ $\Leftrightarrow$ $\forall n \geq 4,$ $\Leftrightarrow$ $\forall n \geq 4,$

$2n^2 + 10 \leq 3n^2$ $10 \leq n^2$ $3.162 \leq n$ TRUE!

Example

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

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

$$3g(n) = 3n^2$$

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

$$g(n) = n^2$$

Formally:

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

$$\forall n \geq 4,$$

$$2n^2 + 10 \leq 3n^2$$

Q: Why did we pick $c = 3, n_0 = 4$? Why is this the "right" way to choose them?

A: There's not one "right" way!

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

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

$7g(n) = 7n^2$

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

$g(n) = n^2$

Choose $n_0 = 2$

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

$$\forall n \geq 2,$$

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

There's **not** a "correct" choice of c and n_0!
As long as you can find any c, n_0 that work, you'll have shown that $T(n) = O(g(n))$

What about:
$n = O(n^2)?$

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

O(...) is an **upper** bound:
$$n = O(n^2)$$

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

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

Legend:
- T(n) = n (red)
- g(n) = n^2 (blue)
- 1*g(n) (green)
- n=1 (orange)

$g(n) = n^2$

$T(n) = n$
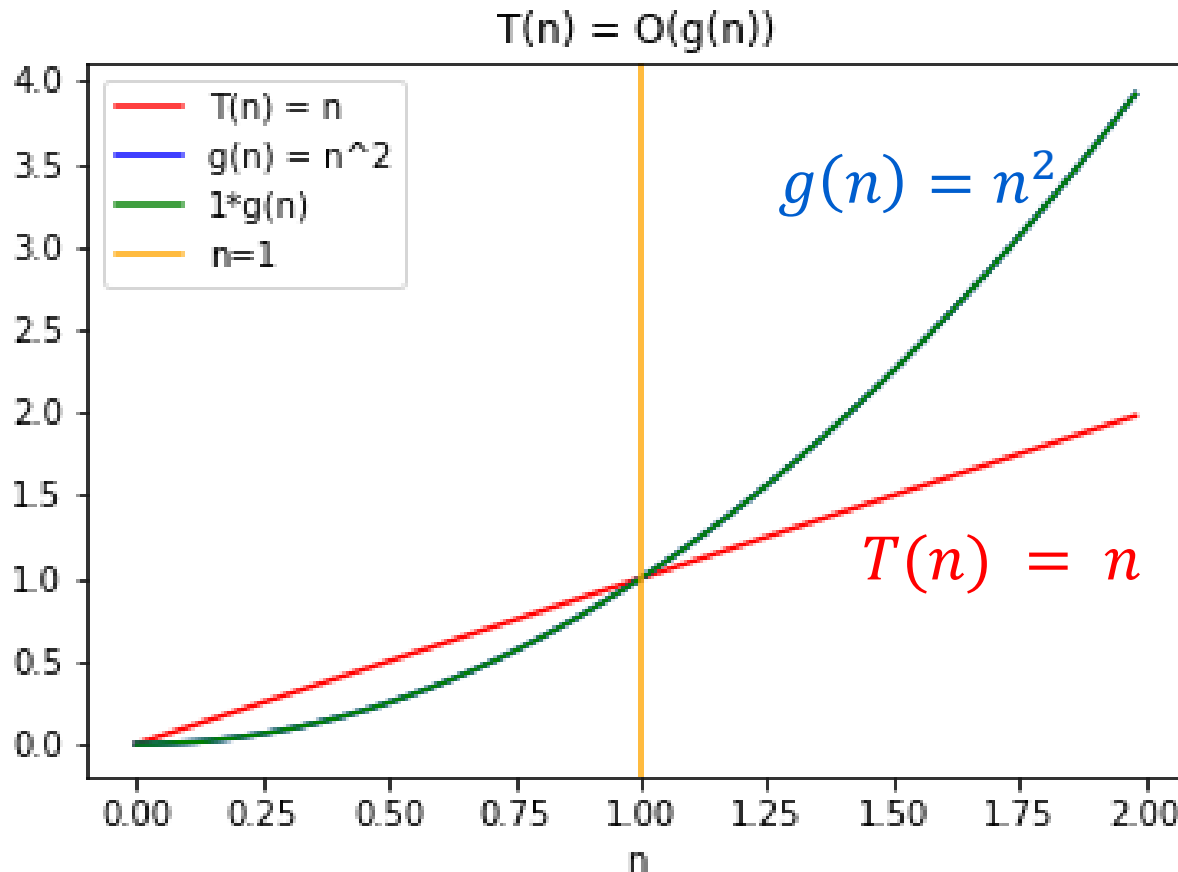
- Choose c = 1
- Choose n_0 = 1
- Then

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

$\Omega(\ldots)$ 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 $g(n)$.
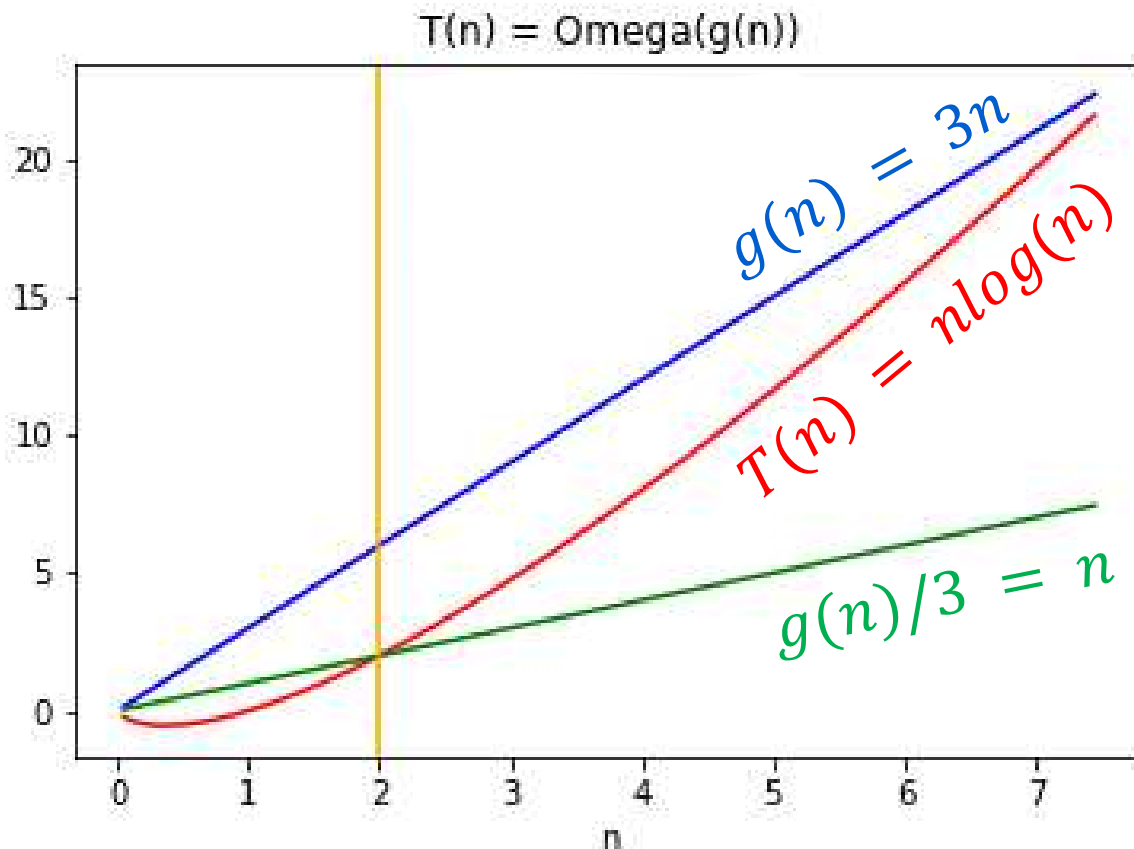
- Formally,

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

Switched these!!

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

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

T(n) = Omega(g(n))

$g(n) = 3n$

$T(n) = nlog(n)$

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

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

T(n) = Omega(g(n))

$g(n) = 3n$

$T(n) = n\log(n)$
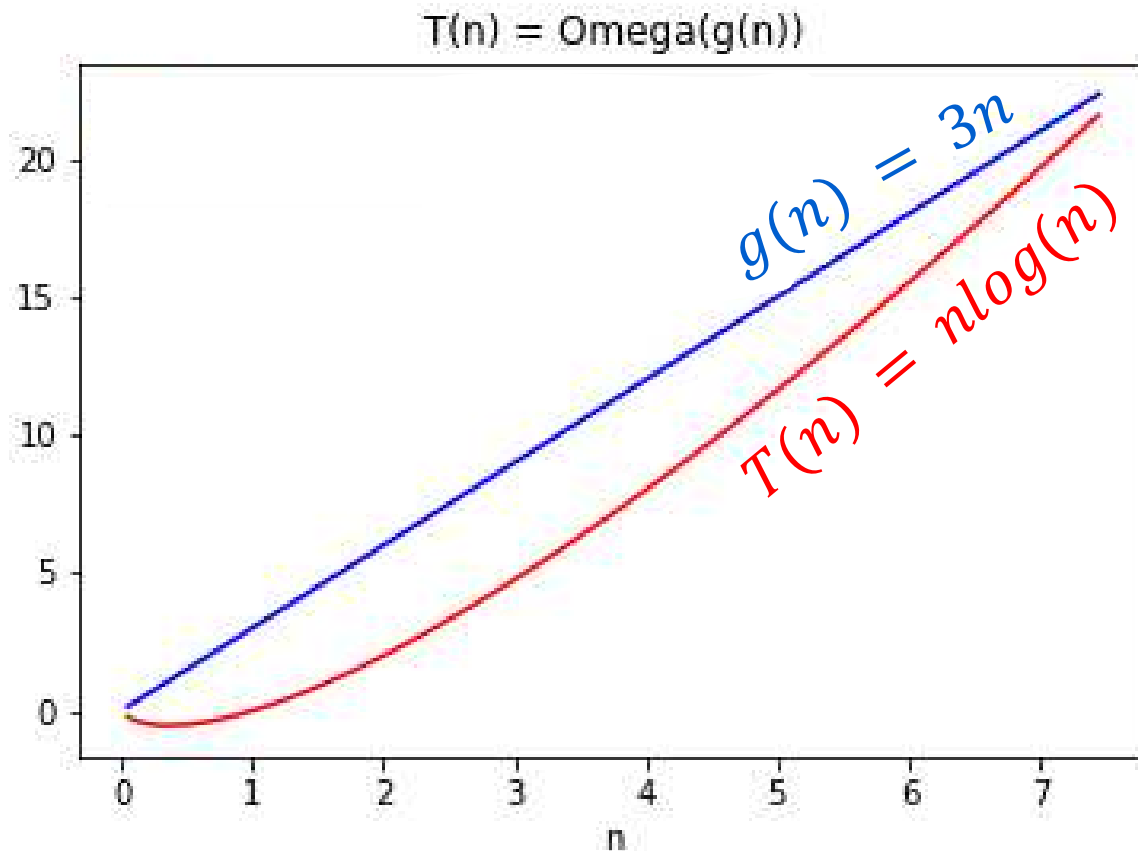
$g(n)/3 = n$

- 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

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

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

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

- 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, $n_0 + c + 1$?
  - 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. $p(n) = O(n^k)$
  2. $p(n)$ is **not** $O(n^{k-1})$

Try to prove it yourself first!

- See the book (AI Section 2.3.2) for a proof.

Siggi the Studious Stork

More examples

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

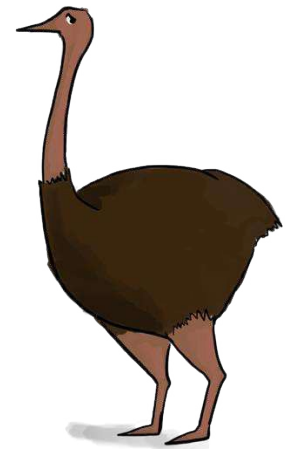- 3^n is **NOT** $O(2^n)$
- $\log_2(n) = \Omega(\ln(n))$
- $\log_2(n) = \Theta(\, 2^{\log\log(n)}\,)$

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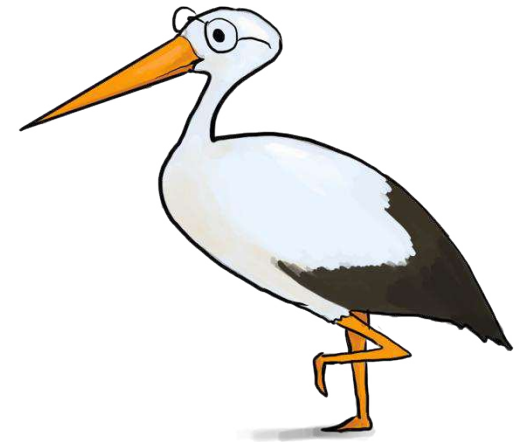
Siggi the Studious Stork

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 is my happy face!

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

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

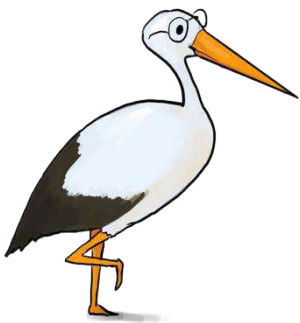- In the course, (almost) every algorithm we see will be actually practical, without needing to take $n \geq n_0 = 2^{10000000}$.

Back Insertion Sort

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

Naive vs. non-naive insertion sort
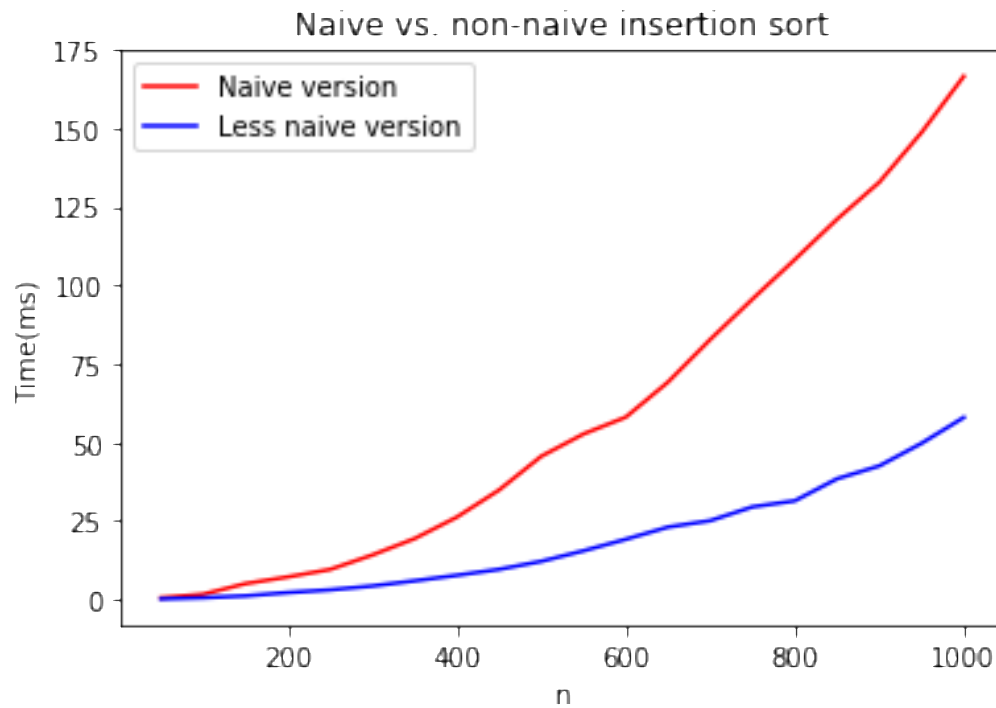
Insertion Sort: running time

- 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

Naive vs. non-naive insertion sort
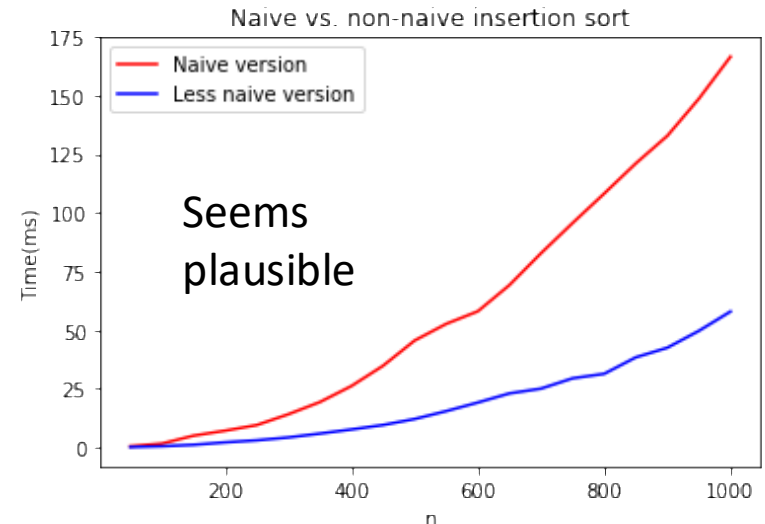
Insertion Sort: running time

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 $O(n^2)$.
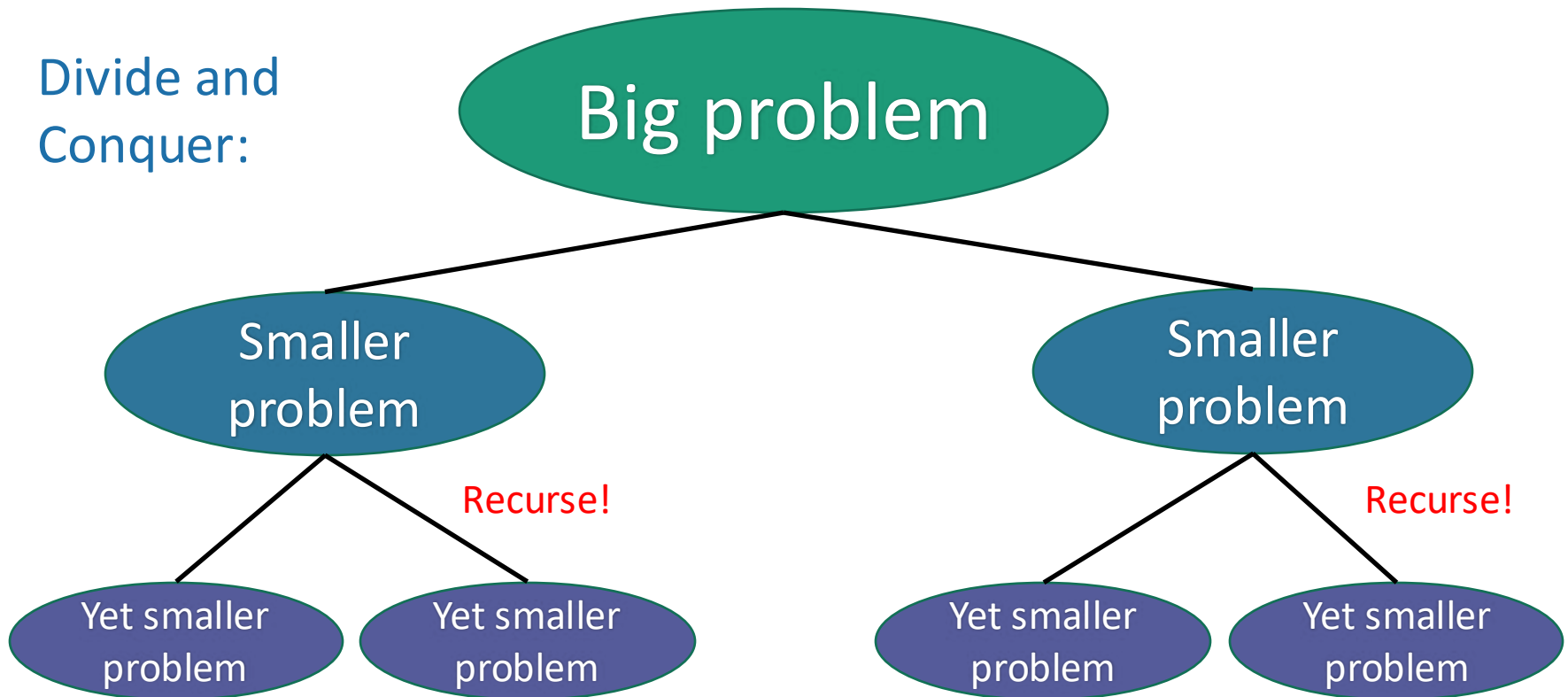
Can we do better?

The Plan

- InsertionSort recap

- Worst-case analyisis
  - Back to InsertionSort: Does it work?

- Asymptotic Analysis
  - Back to InsertionSort: Is it fast?

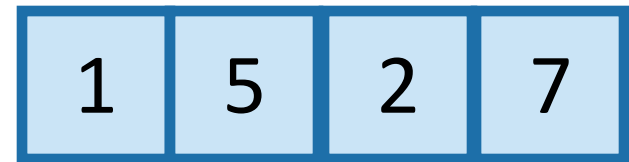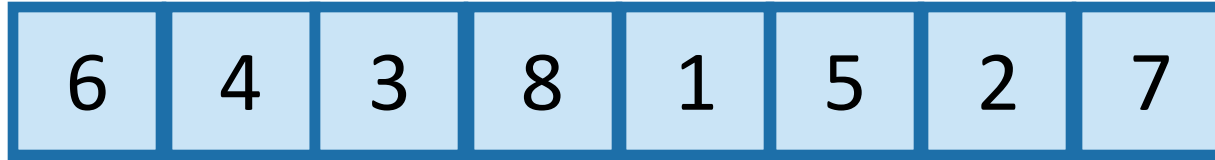- MergeSort
  - Does it work?
  - Is it fast?

Can we do better?

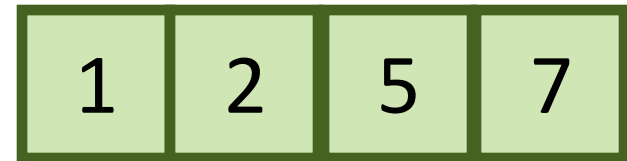- MergeSort: a divide-and-conquer approach
- Recall from last time:

Divide and Conquer:

Big problem

Smaller problem   **Smaller problem**

Recurse!   Recurse!

Yet smaller problem   Yet smaller problem   Yet smaller problem   Yet smaller problem

MergeSort

6	4	3	8	1	5	2	7

6	4	3	8

1	5	2	7

Recursive magic!

Recursive magic!

3	4	6	8
▲

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

MergeSort Pseudocode

MERGESORT(A):

- n = length(A)

- **if** n ≤ 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

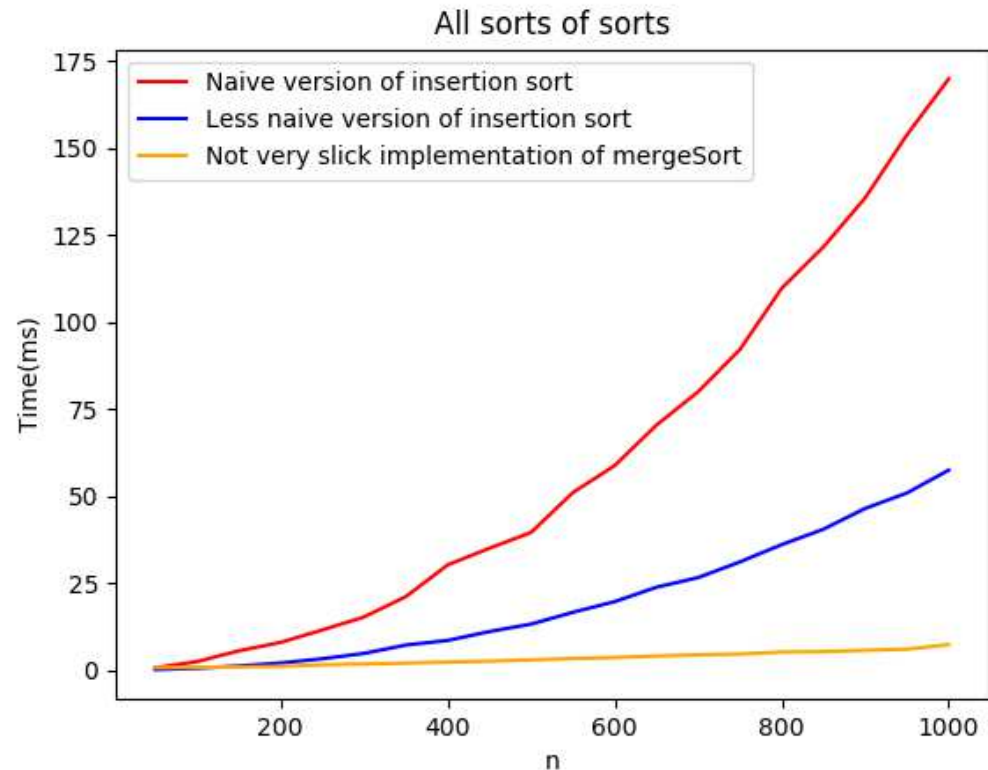# Two questions

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

IPython notebook says…

Empirically:
1. Seems to work.
2. Seems fast.

All sorts of sorts

Legend:
- Naive version of insertion sort
- Less naive version of insertion sort
- Not very slick implementation of mergeSort

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.

Outline!

Assume that n is a power of 2 for convenience.

- **Inductive hypothesis (IH):**

  "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, so IH holds for i=1.

- **Inductive step:** Need to show: if IH holds for all 0<i<k, then it holds for i=k.
- Aka, need to show that if L and R are sorted, then MERGE(L,R) is sorted.

- **Conclusion:** The IH holds for i=1,2,...,n, and in particular for n.
- Aka, In the top recursive call, MERGESORT returns a sorted array!

- MERGESORT(A):
  - n = length(A)
  - **if** $n \leq 1$:
    - **return** A
  - L = MERGESORT(A[0 : n/2])
  - R = MERGESORT(A[n/2 : 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!

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 $O(n^2)$.

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

Quick log refresher

- Def: $\log(n)$ is the number so that $2^{\log(n)} = n$.

- Intuition: $\log(n)$ is how many times you need to divide n by 2 in order to get down to 1.

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

Halve 5 times

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

Halve 6 times

log(128) = 7
log(256) = 8
log(512) = 9
….
log(**# 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(n²)!

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

$\cdots$

Level t

2^t subproblems at level t.

$n/2^t$ $n/2^t$ $n/2^t$ $n/2^t$ $n/2^t$ $n/2^t$

$\cdots$

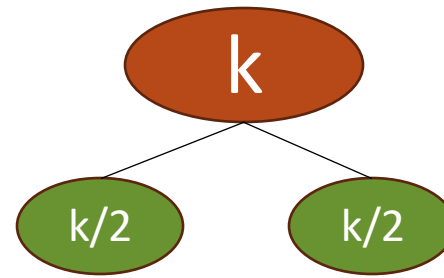(Size 1)

Level log(n)
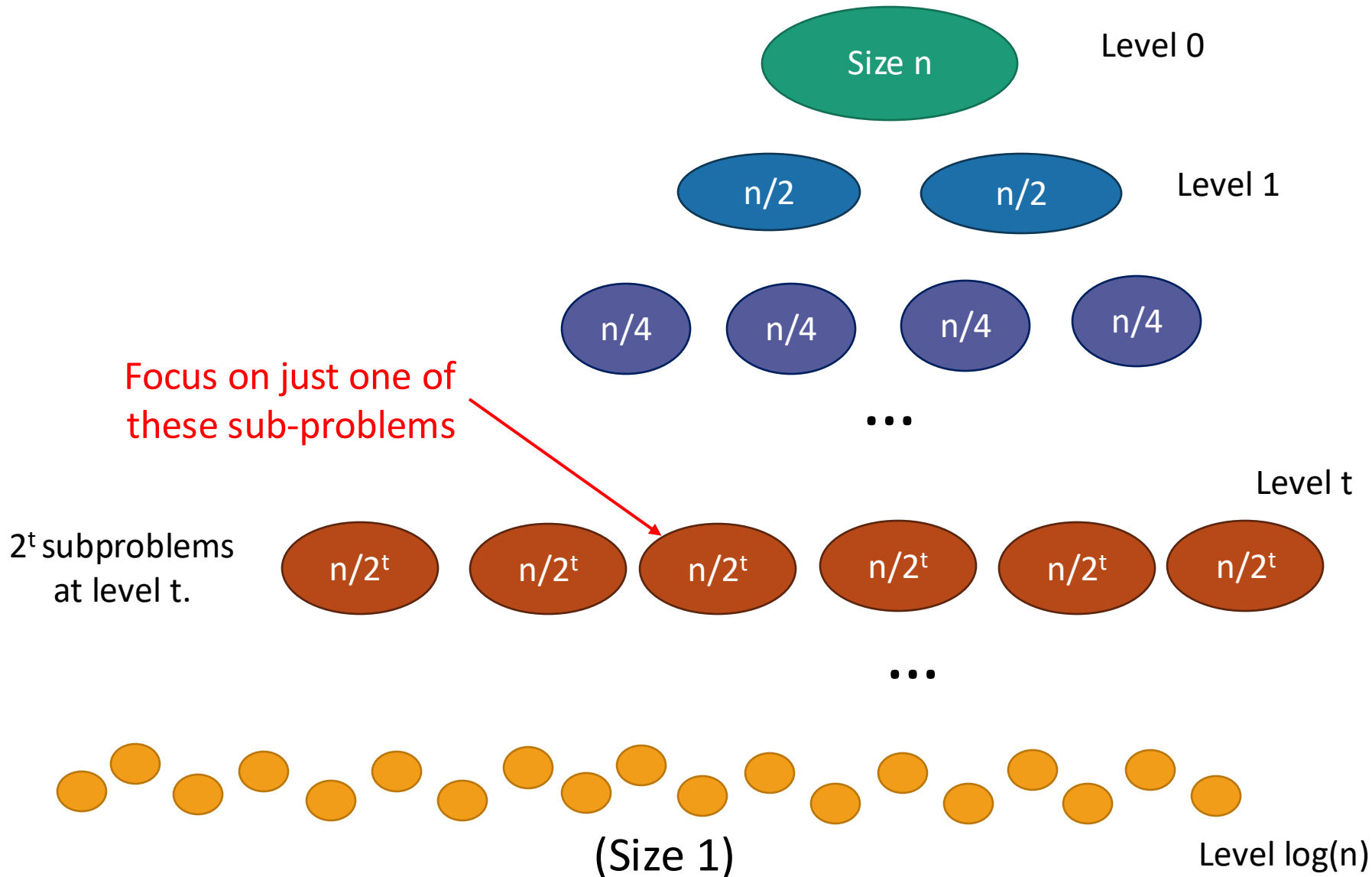
How much work in this sub-problem?
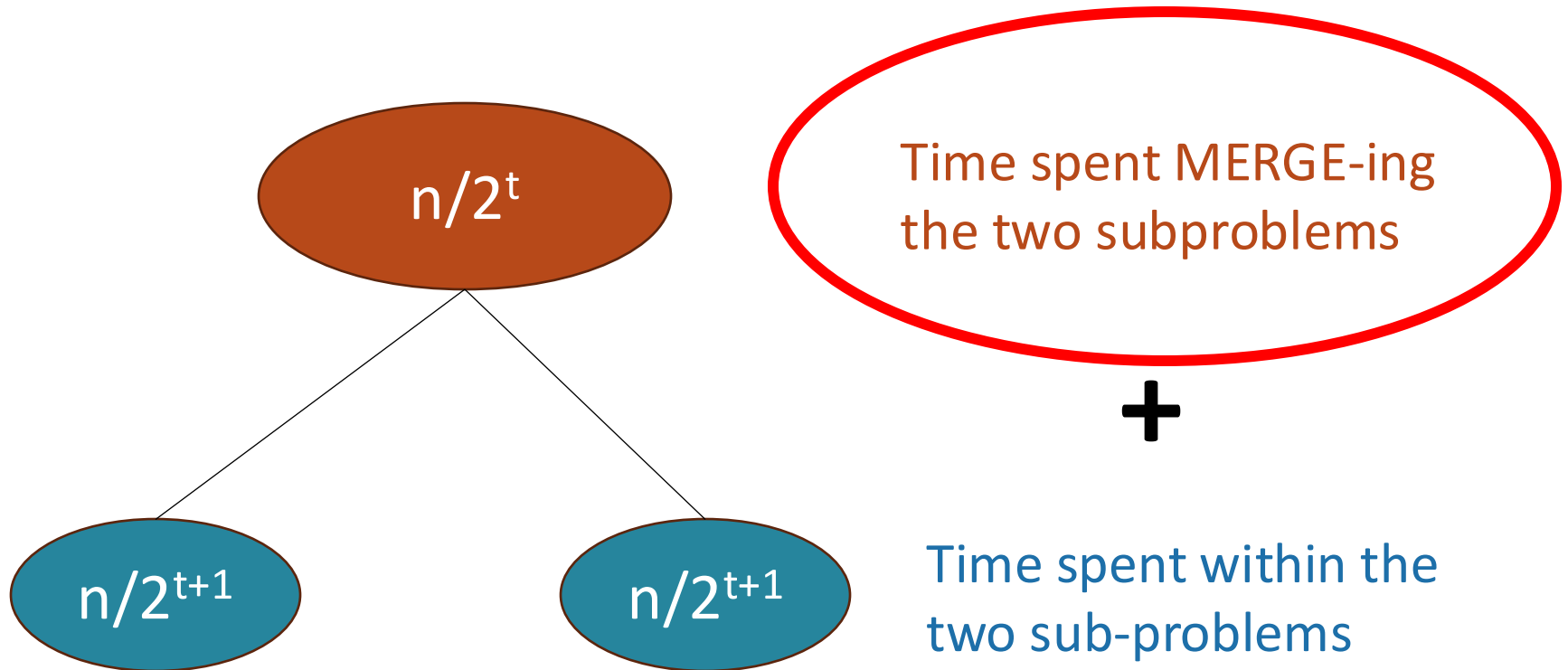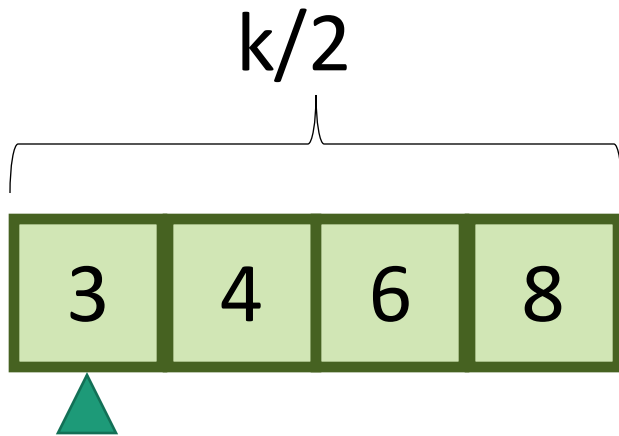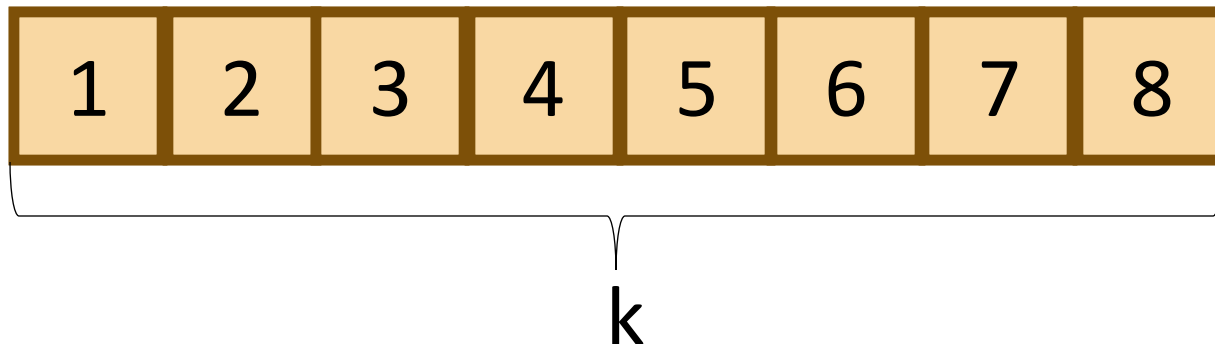
How much work in this sub-problem?

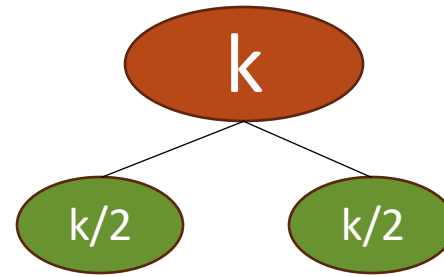Let $k = n/2^t$ ...

How long does it take to MERGE?

k

k/2 k/2

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

k/2

| 3 | 4 | 6 | 8 |

k/2

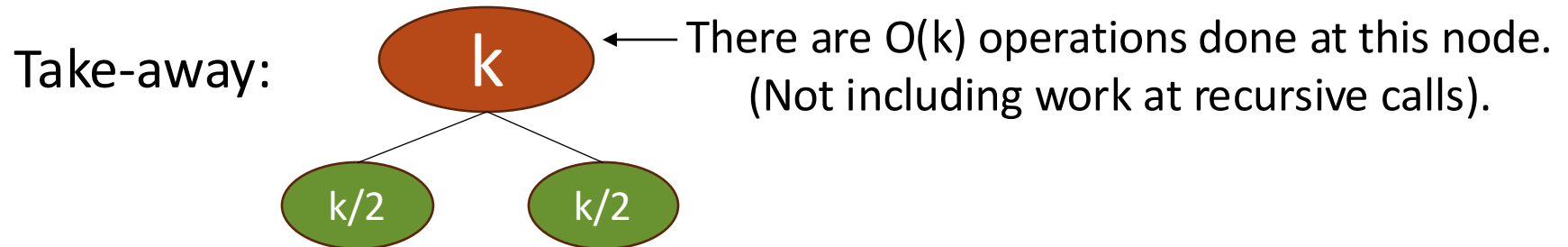| 1 | 2 | 5 | 7 |

MERGE!

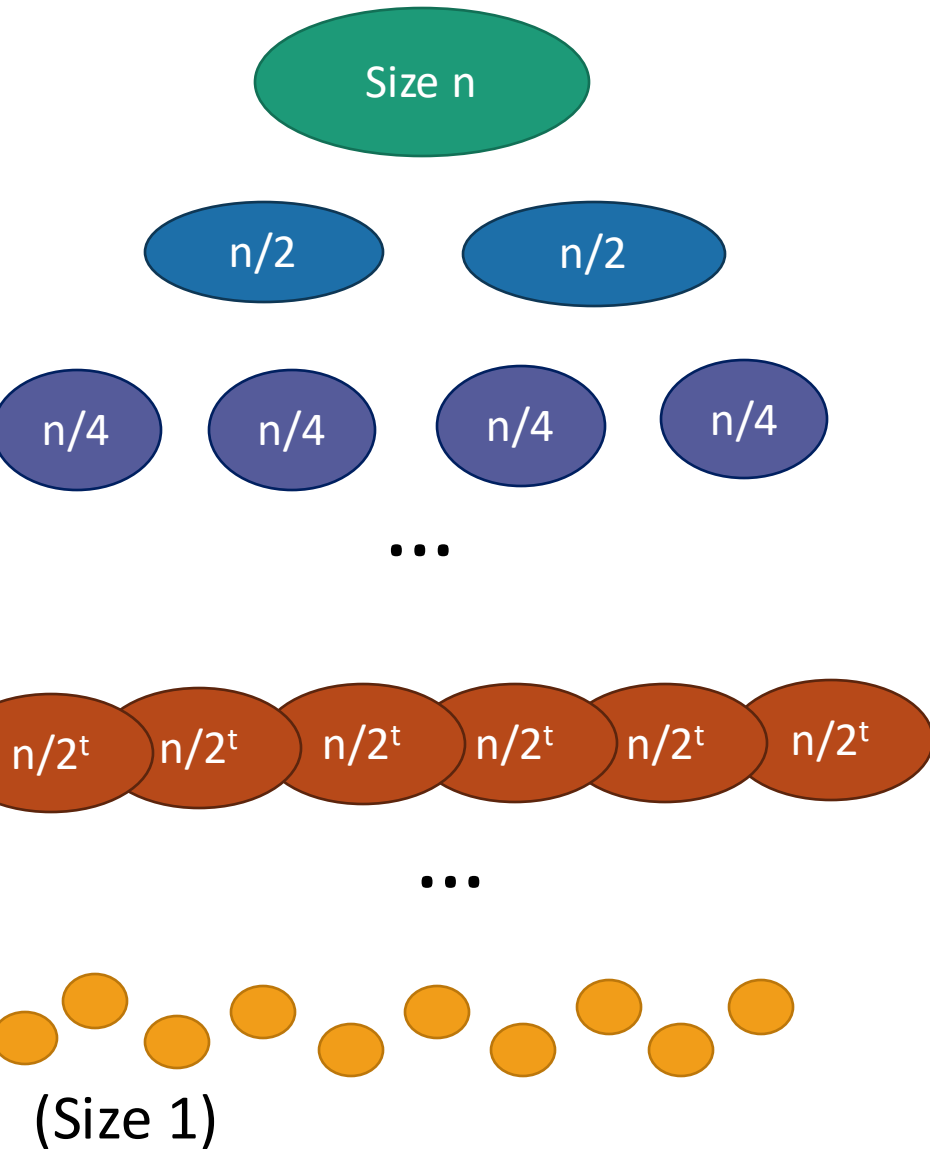| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

k

How long does it take to MERGE?

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

Question: in big-Oh notation, how long does it take to run MERGE on two lists of size k/2?
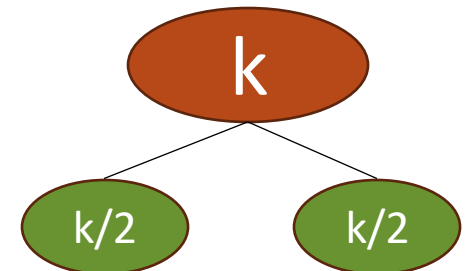
Answer: It takes time O(k), since we just walk across the list once.

Take-away:

← There are O(k) operations done at this node. (Not including work at recursive calls).

Recursion tree

Size n

n/2   n/2

n/4   n/4   n/4   n/4

...

$n/2^t$ $n/2^t$ $n/2^t$ $n/2^t$ $n/2^t$ $n/2^t$

...

(Size 1)

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

k

k/2   k/2

Recursion tree

Size n

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

n/2   n/2

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

n/4   n/4   n/4   n/4

...

This level?

$n/2^t$ $n/2^t$ $n/2^t$ $n/2^t$ $n/2^t$ $n/2^t$

...

This level?

(Size 1)

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

k

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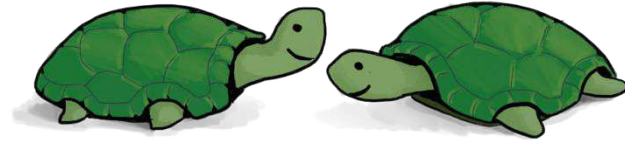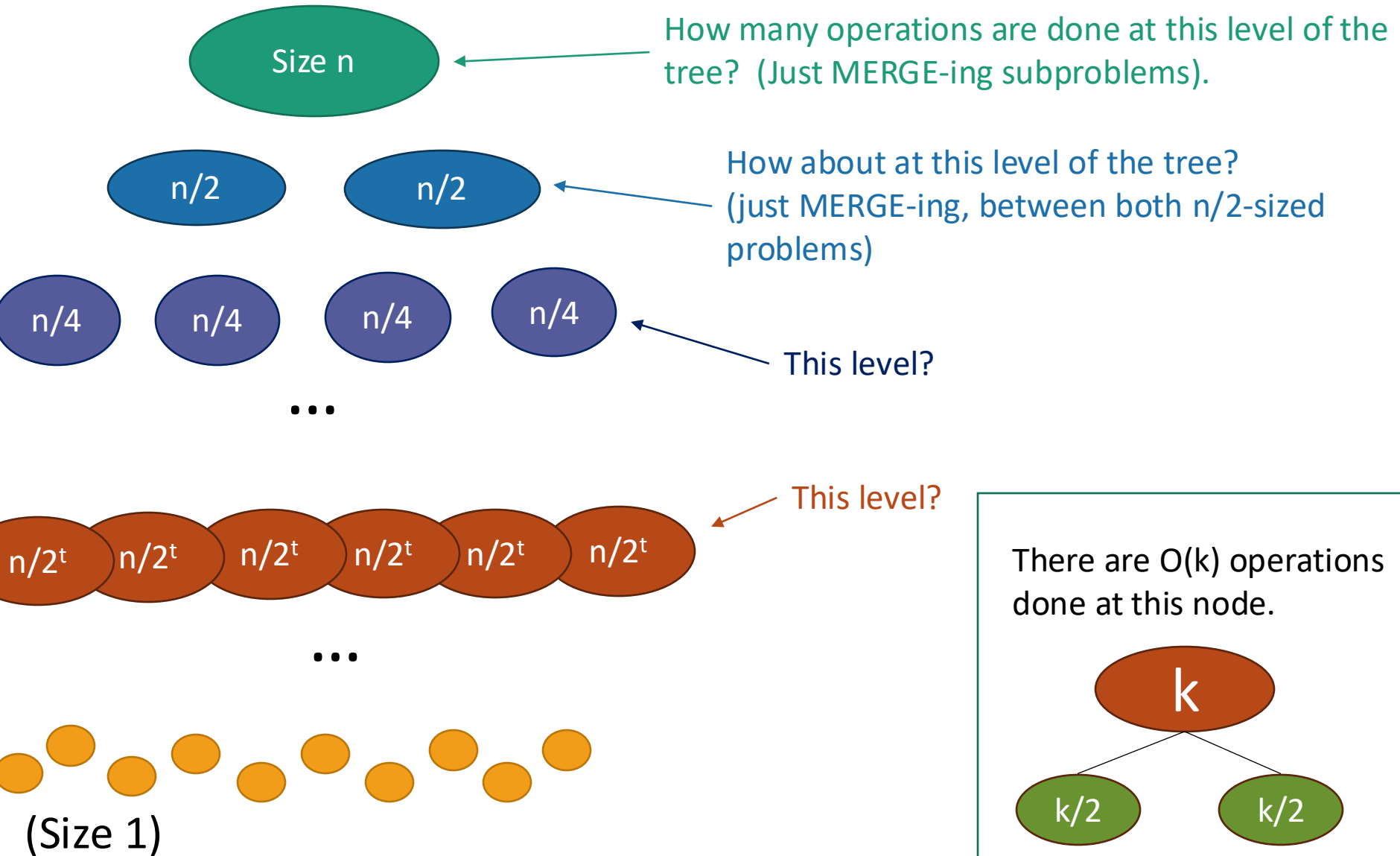
Recursion tree
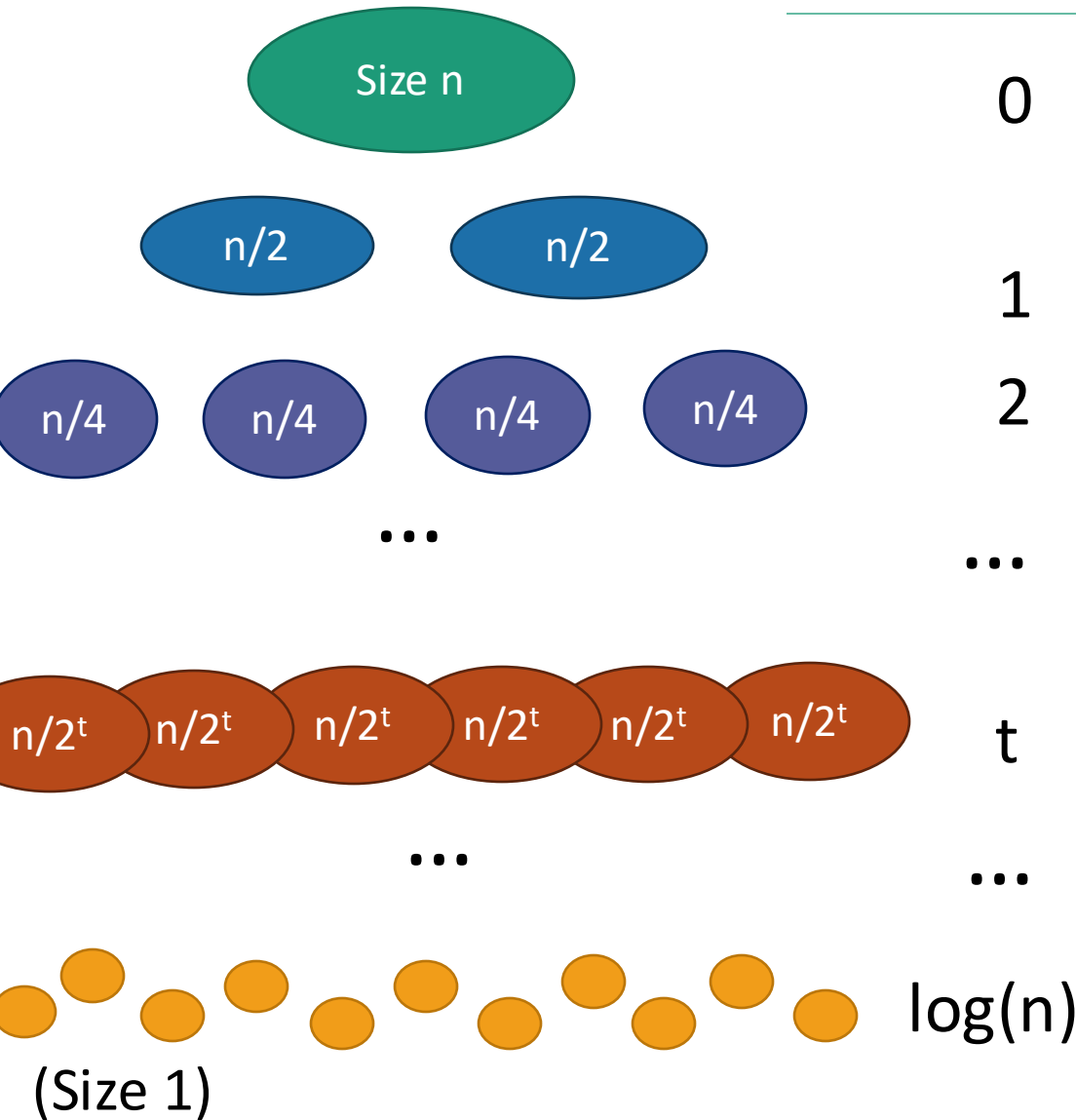
Level	# problems	Size of each problem	Amount of work at this level
0	1	n	$O(n)$
1	2	n/2	$O(n)$
2	4	n/4	$O(n)$
...			
t	2^t	$n/2^t$	$O(n)$
...			
$\log(n)$	n	1	$O(n)$

Ellipses in tree: Size n, n/2, n/2, n/4, n/4, n/4, n/4, ..., $n/2^t$, ..., (Size 1)

Explanation for this table done on the board!

Total runtime…

- $O(n)$ work 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

- Do HW0 (due Tuesday!)
- Get started on HW1
  - perhaps at the HW party on Monday!