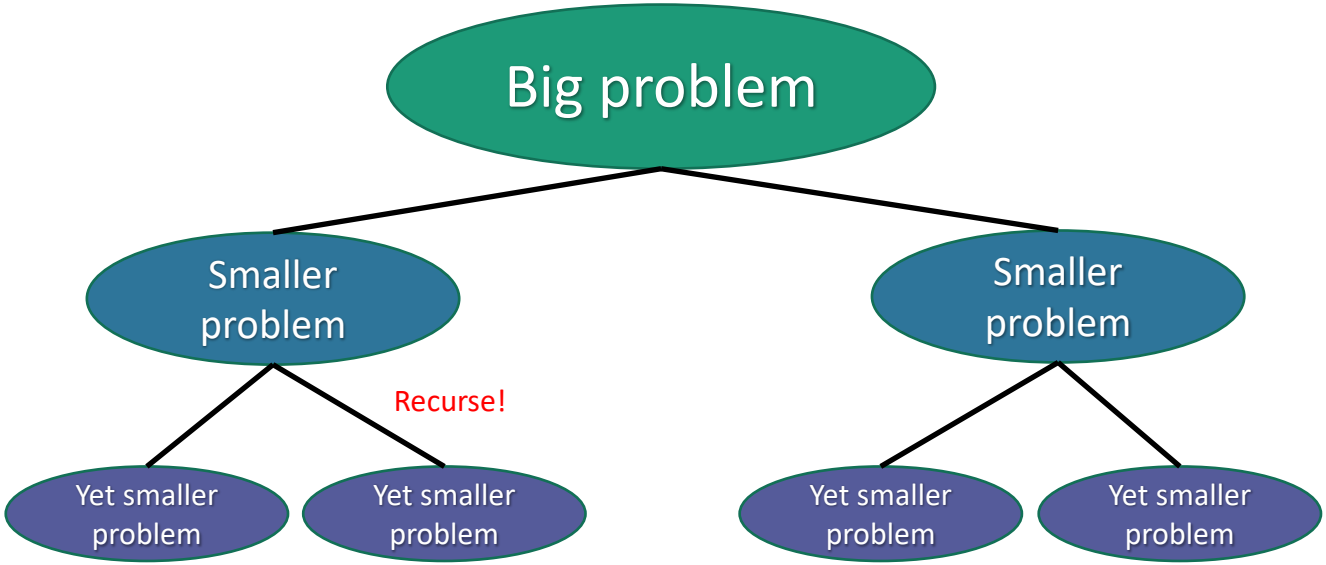


Lecture 2

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

Which of the following is the smallest?
$10n^2$, $100n^{1.6}$, or $33n \log n$

Thanks for the anonymous feedback!

- Enrollment is significantly higher than in previous years. (This is a good thing – *I want all of you here*!)
  - It is creating a bit of a mess with things like lecture seating and section rooms. We're working with the university to figure these things. Thank you for your patience.

- Please keep asking questions during lecture
  - It's also 100% OK to say "this last slide was too fast, could you go over it again?"

- Please keep the helpful feedback coming!
  - However, anonymous feedback form is not a great place for asking questions. (Because then we can't answer.)

[Recap slide 1/6]

Welcome to CS161

Philosophy

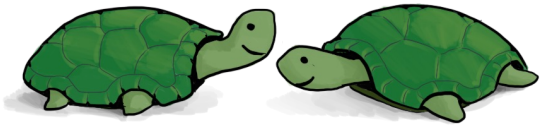
- Algorithms are awesome!
- Our motivating questions:
  - Does it work?
  - Is it fast?
  - Can I do better?

Cast

Plucky the pedantic penguin

Lucky the lackadaisical lemur

Ollie the over-achieving ostrich



Think-Pair-Share Terrapins

Pair-Explain Parrots

Siggi the studious stork

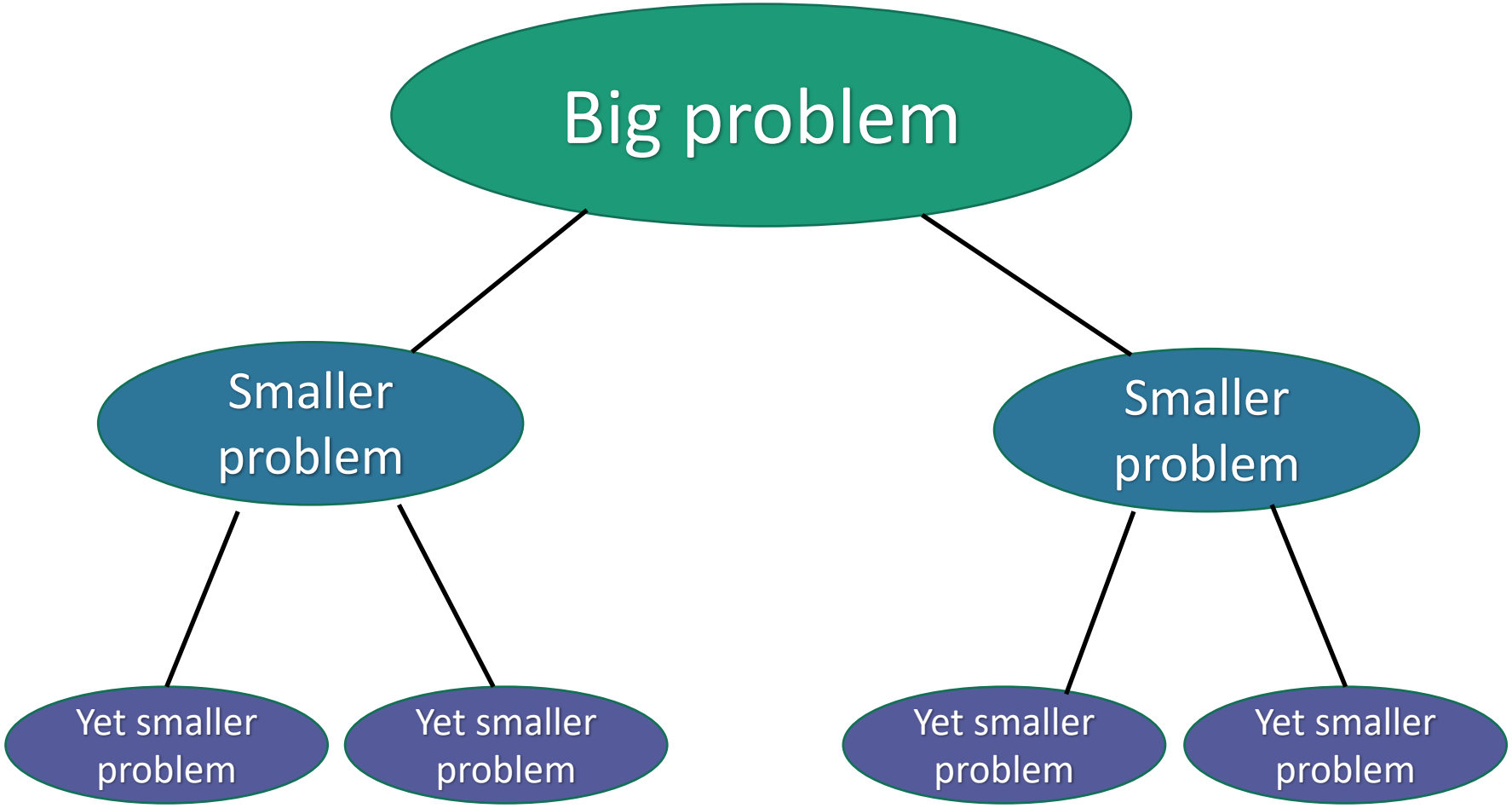
?

Mystery friend joining today!

[Recap slide 2/6]

Divide and conquer

Break problem up into smaller (easier) sub-problems



[Recap slide 3/6]

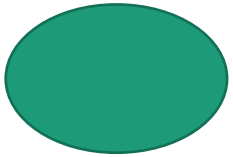
Slow integer multiplication w/ divide and conquer

Multiply(x, y):

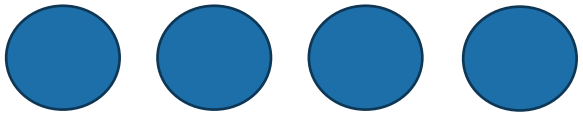
- **If** n=1:
  - **Return** xy
- Write $x = a\ 10^{\frac{n}{2}} + b$
- Write $y = c\ 10^{\frac{n}{2}} + d$
- Recursively compute ac, ad, bc, bd:
  - ac = **Multiply**(a, c), etc…
- Add them up to get xy:
  - xy = ac 10^n + (ad + bc) $10^{n/2}$ + bd

[Recap slide 4/6]

Running-time analysis of Multiply(,)

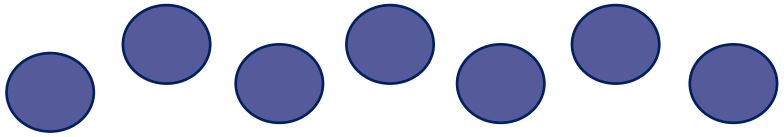


1 problem
of size n



4 problems
of size n/2

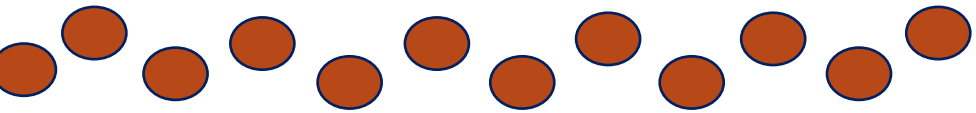
…



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

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

…



$\frac{n^2}{\text{____}}$ problems
of size 1

- If you cut n in half $\log_2(n)$ times, you get down to 1.
- So we do this $\log_2(n)$ times and get…

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

problems of size 1.

[Recap slide 5/6]

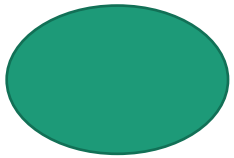
Faster integer multiplication w/ divide and conquer

Karatsuba(x, y):

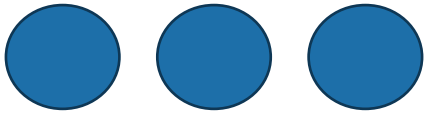
- **If** n=1:
  - **Return** xy
- Write $x = a\ 10^{\frac{n}{2}} + b$ and $y = c\ 10^{\frac{n}{2}} + d$
- ac = **Karatsuba**(a, c)
- bd = **Karatsuba**(b, d)
- z = **Karatsuba**(a+b, c+d)
- cross_terms = z - ac - bd
- xy = ac 10^n + (cross_terms) $10^{n/2}$ + bd
- Return xy

[Recap slide 6/6]

Running-time analysis of Karatsuba(,)

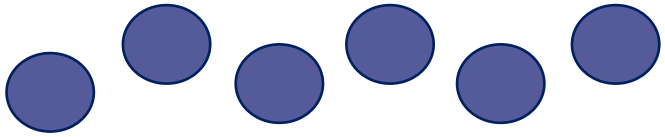


1 problem
of size n



3 problems
of size n/2

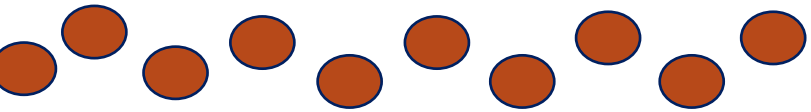
…



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

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

…



$n^{1.6}$ problems
of size 1

- If you cut n in half $\log_2(n)$ times, you get down to 1.
- So we do this $\log_2(n)$ times and get…

$3^{\log_2 n} = n^{\log_2 3} \approx n^{1.6}$
problems of size 1.

Today

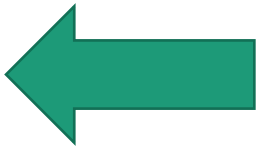
SortingHatSort not discussed

- 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

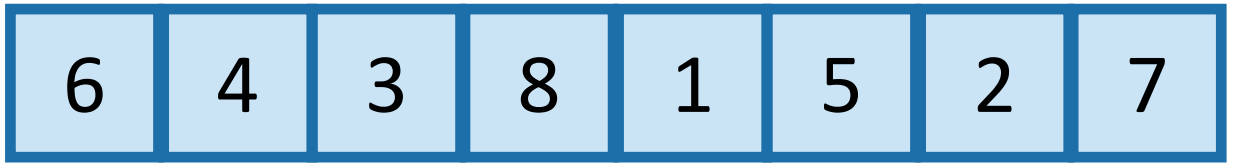
- Dr Benji Xie will tell us about EthiCS + CS161

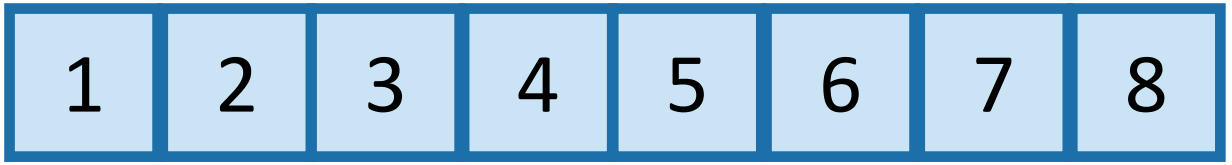
The plan

- Sorting! 
- InsertionSort Algorithm
  - Does it work? Is it fast?
  - Worst-case analysis
  - Skill: Analyzing correctness of iterative and recursive algorithms.
- MergeSort Algorithm
  - Does it work? Is it fast?
    - Skill: Analyzing running time of recursive algorithms (part 1…more next time!)
- Embedded Ethics

Today's goal: Sorting

- Important primitive
- For today, we'll pretend all elements are distinct.





Length of the list is n

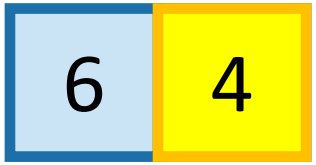
The plan

- Sorting!
- InsertionSort Algorithm
  - Does it work? Is it fast?
  - Worst-case analysis
  - Skill: Analyzing correctness of iterative and recursive algorithms.
- MergeSort Algorithm
  - Does it work? Is it fast?
    - Skill: Analyzing running time of recursive algorithms
      (part 1…more next time!)
- Embedded Ethics

Sorting – warmup of the warmup

- Sorting one element – that's really easy!

- Sorting two elements – still pretty simple…

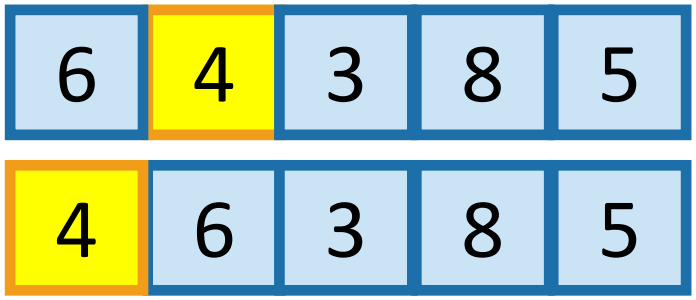


- Can you generalize to n elements?

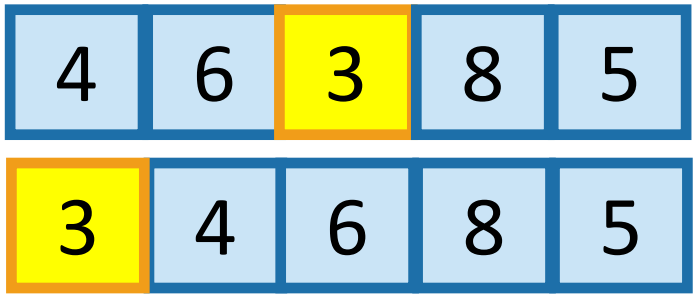
InsertionSort

example

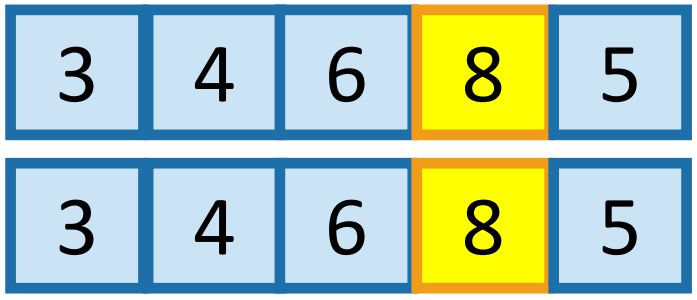
Start by moving A[1] toward the beginning of the list until you find something smaller (or can't go any further):



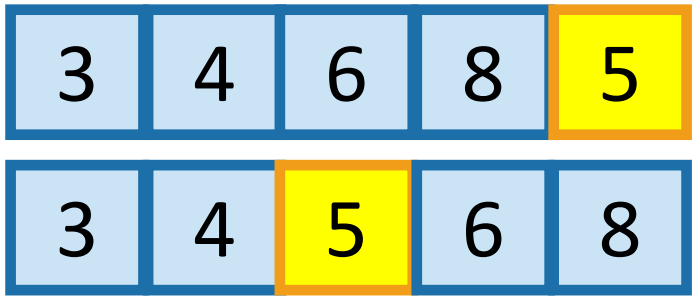
Then move A[2]:



Then move A[3]:

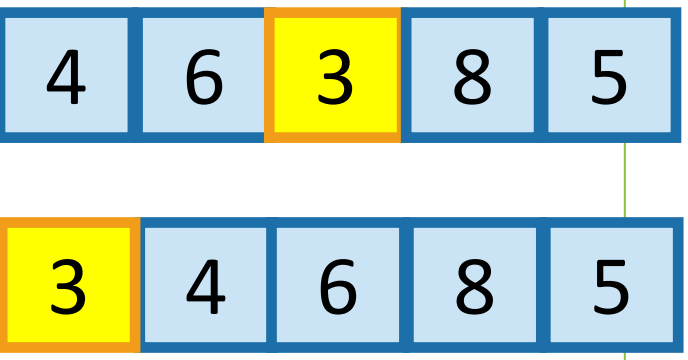


Then move A[4]:



Then we are done!

Insertion Sort (the actual algorithm)




```
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
```

Remember our two questions:

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

Insertion Sort

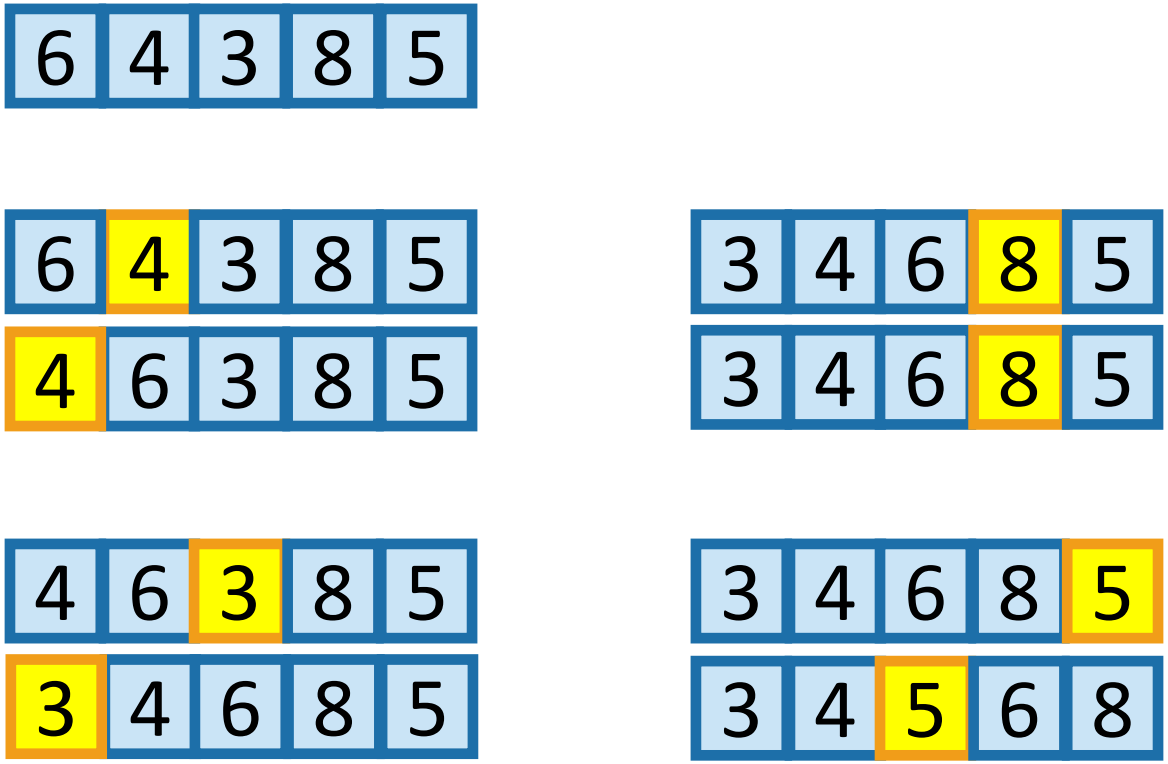
1. Does it work?
2. Is it fast?

- 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.

Claim: InsertionSort "works"

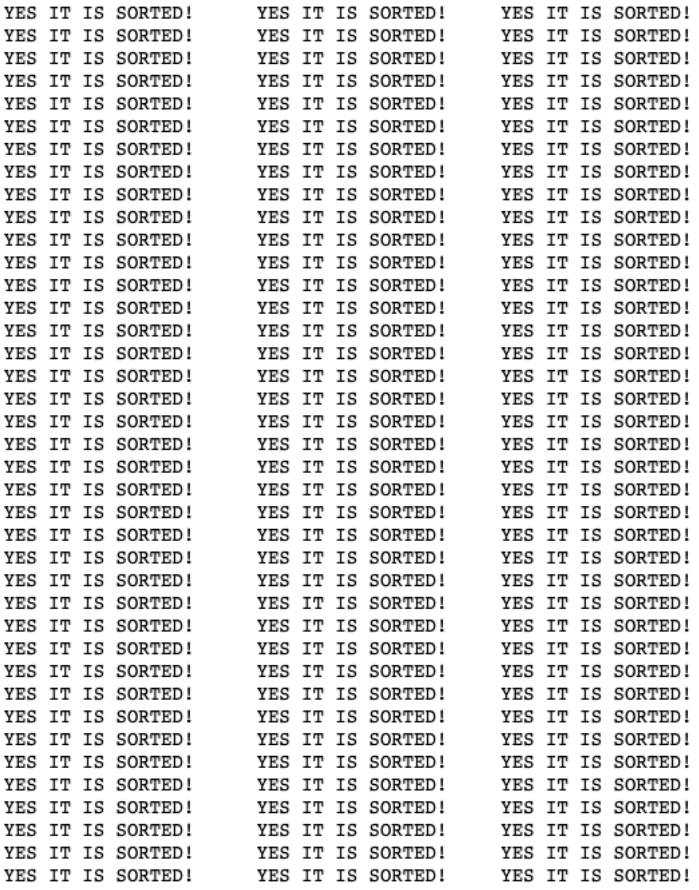
- "Proof:" It just worked in this example:



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!')
```



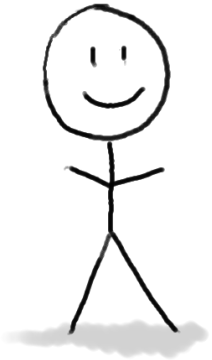
What does it mean to “work”?

- 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:



Algorithm designer

Here is my algorithm!

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

Anakin the adversarial aardvark

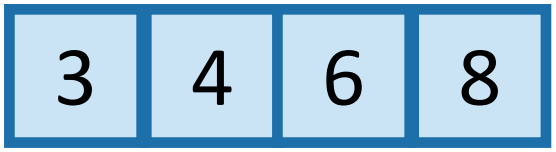
HERE IS AN INPUT!

(E.G. 8437)

WHICH I DESIGNED TO BE TERRIBLE FOR YOUR ALGORITHM!

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

Why does this work?

- Say you have a sorted list, 

, and another element 

.

- Insert 

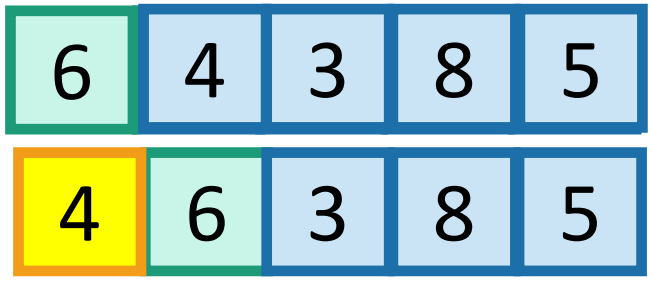
right after the largest thing that's still smaller than 

. (Aka, right after 

).

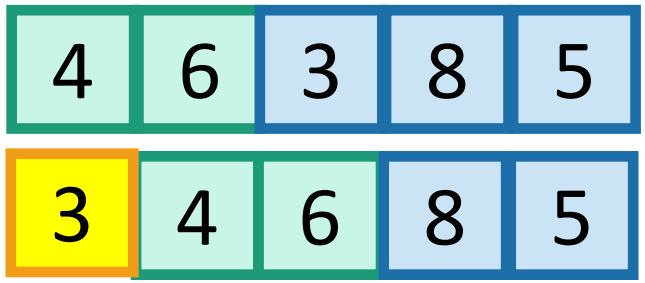
- Then you get a sorted list: 

So just use this logic at every step.



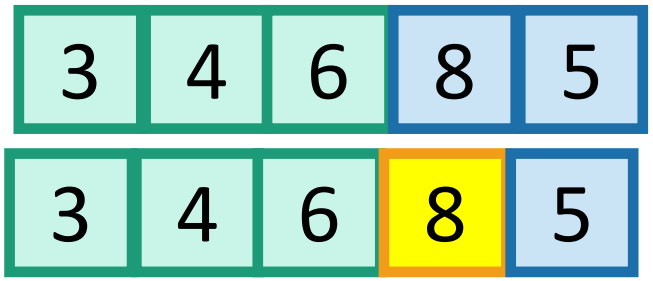
The first element, [6], makes up a sorted list.

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



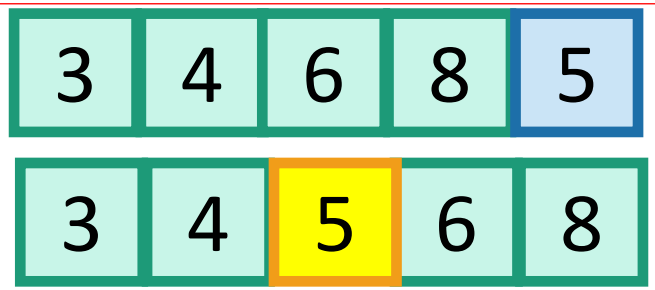
The first two elements, [4,6], make up a sorted list.

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



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

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



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

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

YAY WE ARE DONE!

This sounds like a job for…

Proof By Induction!

Formally: induction

A "loop invariant" is something that we maintain at every iteration of the algorithm.

- Loop invariant(i): `A[:i+1]` is sorted.

- Inductive Hypothesis:
  - The loop invariant(i) holds at the end of the i^{th} iteration (of the outer loop).
- Base case (i=0):
  - Before the algorithm starts, `A[:1]` is sorted. ✓
- Inductive step:
  - If the inductive hypothesis holds at step i, it holds at step i+1
  - Aka, if A[:i+1] is sorted at step i, then A[:i+2] is sorted at step i+1
- Conclusion:
  - 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! ✓

This logic (see lecture notes for details)

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

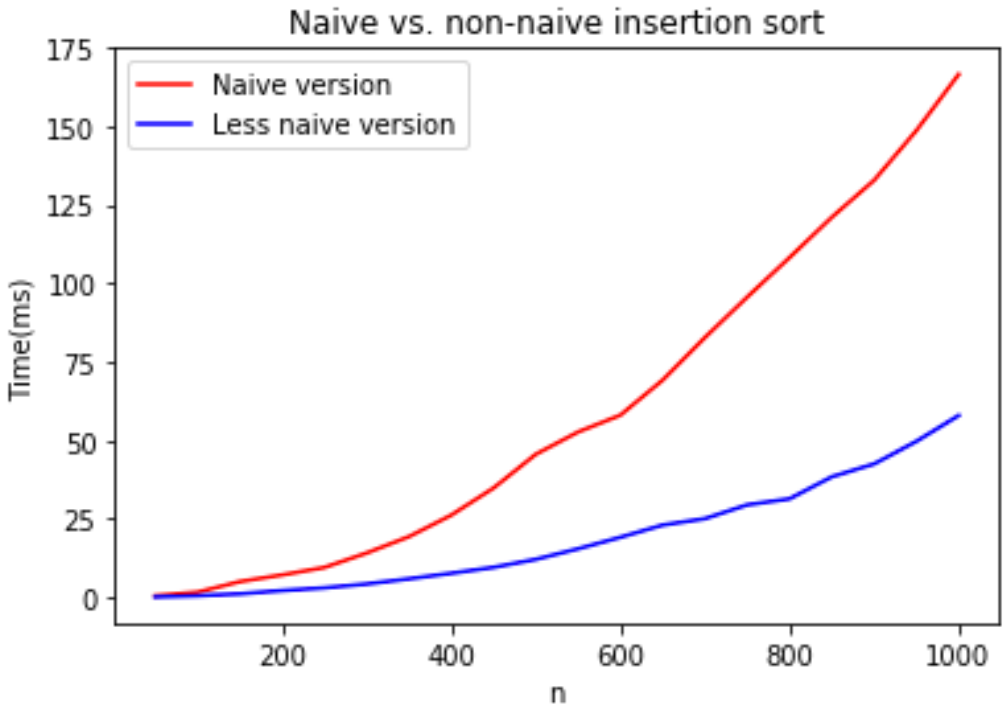
- We're gonna see/do/skip over a lot of them.
- I'm assuming you're comfortable with them from CS103.
  - When you assume…
- If that went by too fast and was confusing:
  - Slides
  - Lecture notes
  - Book
  - Ed
  - Office Hours

Make sure you really understand the argument on the previous slide! Check out the notes for a formal write-up.

Siggi the Studious Stork

InsertionSort

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



Hopefully you're comfortable with big-O notation after taking CS106B/X!
But just in case, we have refreshers in Section 1 and in HW1.

Insertion Sort: running time

```
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

Running time is $O(n^2)$

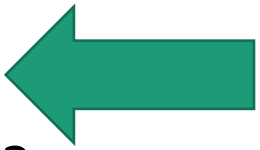
"There's O(1) stuff going on inside the inner loop, so each time the inner loop runs, that's O(n) work. Then the inner loop is executed O(n) times by the outer loop, so that's $O(n^2)$."

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?

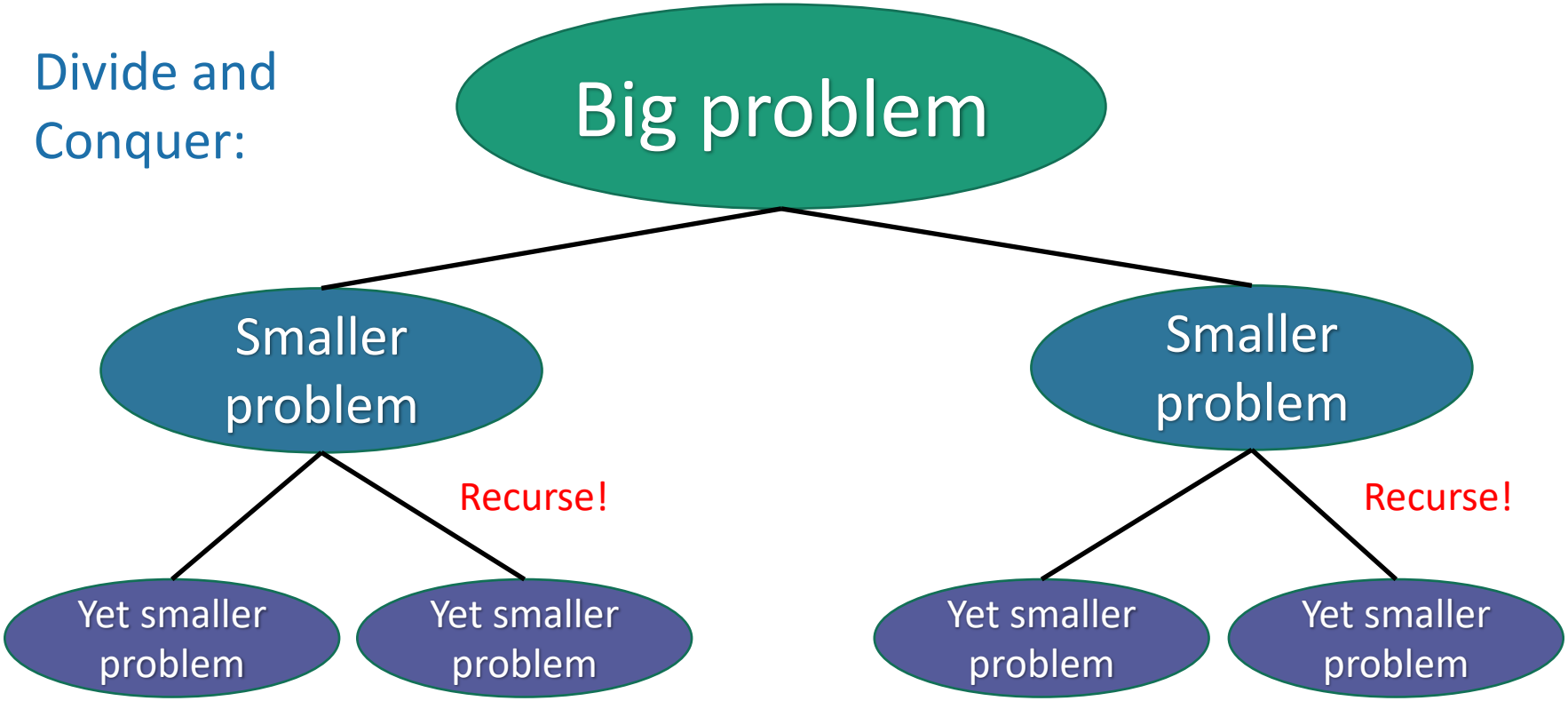
The plan

- Sorting!
- InsertionSort Algorithm
  - Does it work? Is it fast?
  - Worst-case analysis
  - Skill: Analyzing correctness of iterative and recursive algorithms.
- MergeSort Algorithm 
  - Does it work? Is it fast?
    - Skill: Analyzing running time of recursive algorithms (part 1…more next time!)
- Embedded Ethics

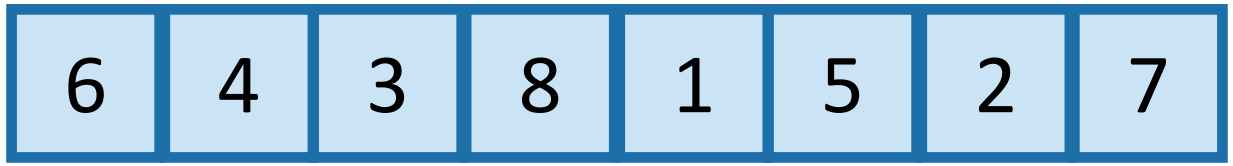
Can we do better?

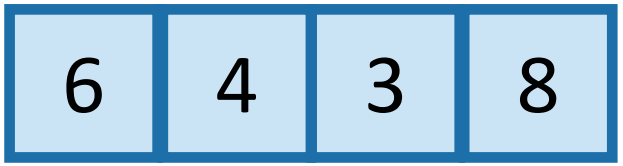
- MergeSort: a divide-and-conquer approach
- Recall from last time:

Divide and Conquer:

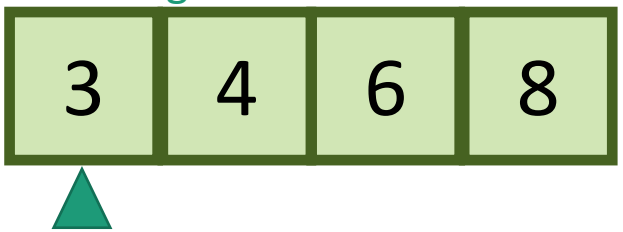


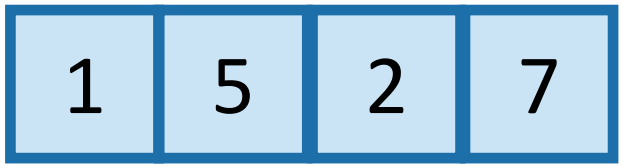
MergeSort



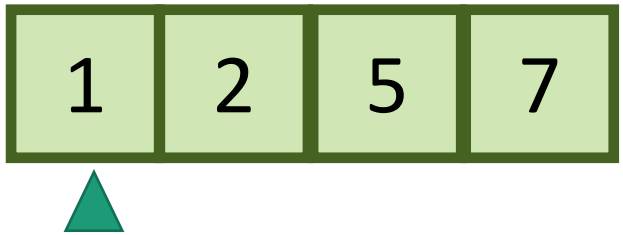


Recursive magic!

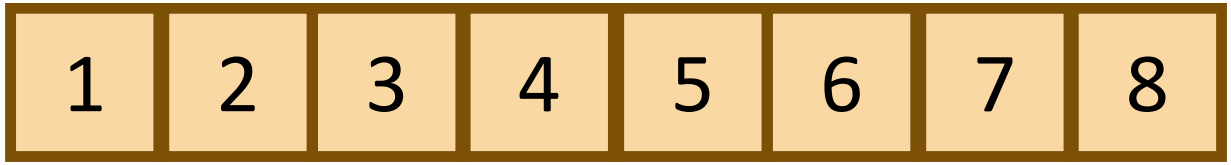




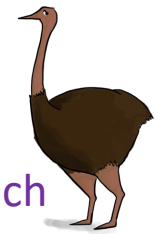
Recursive magic!



MERGE!



How would you do this in-place?



Ollie the over-achieving Ostrich

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])

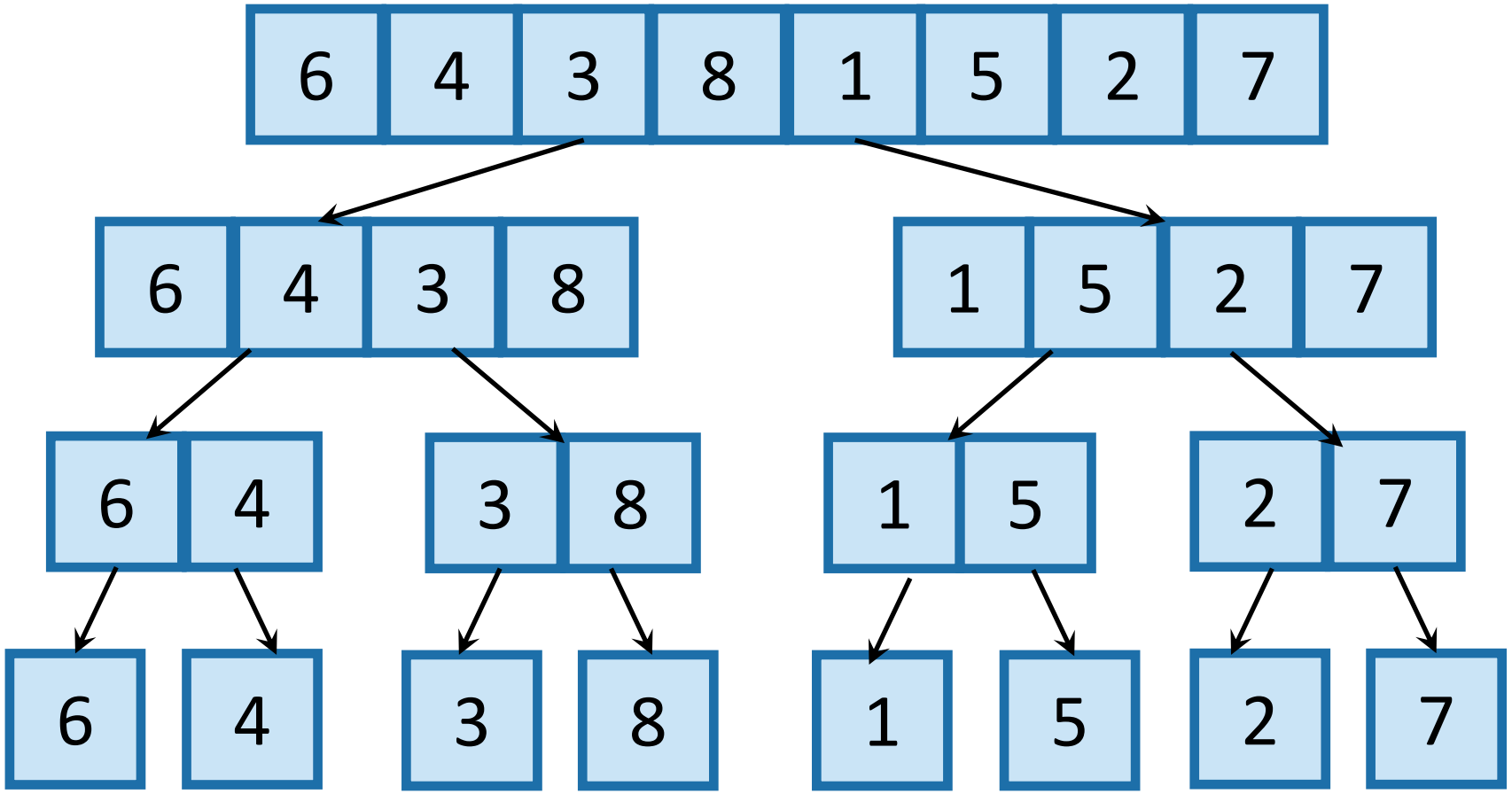
Sort the right half

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

Merge the two halves

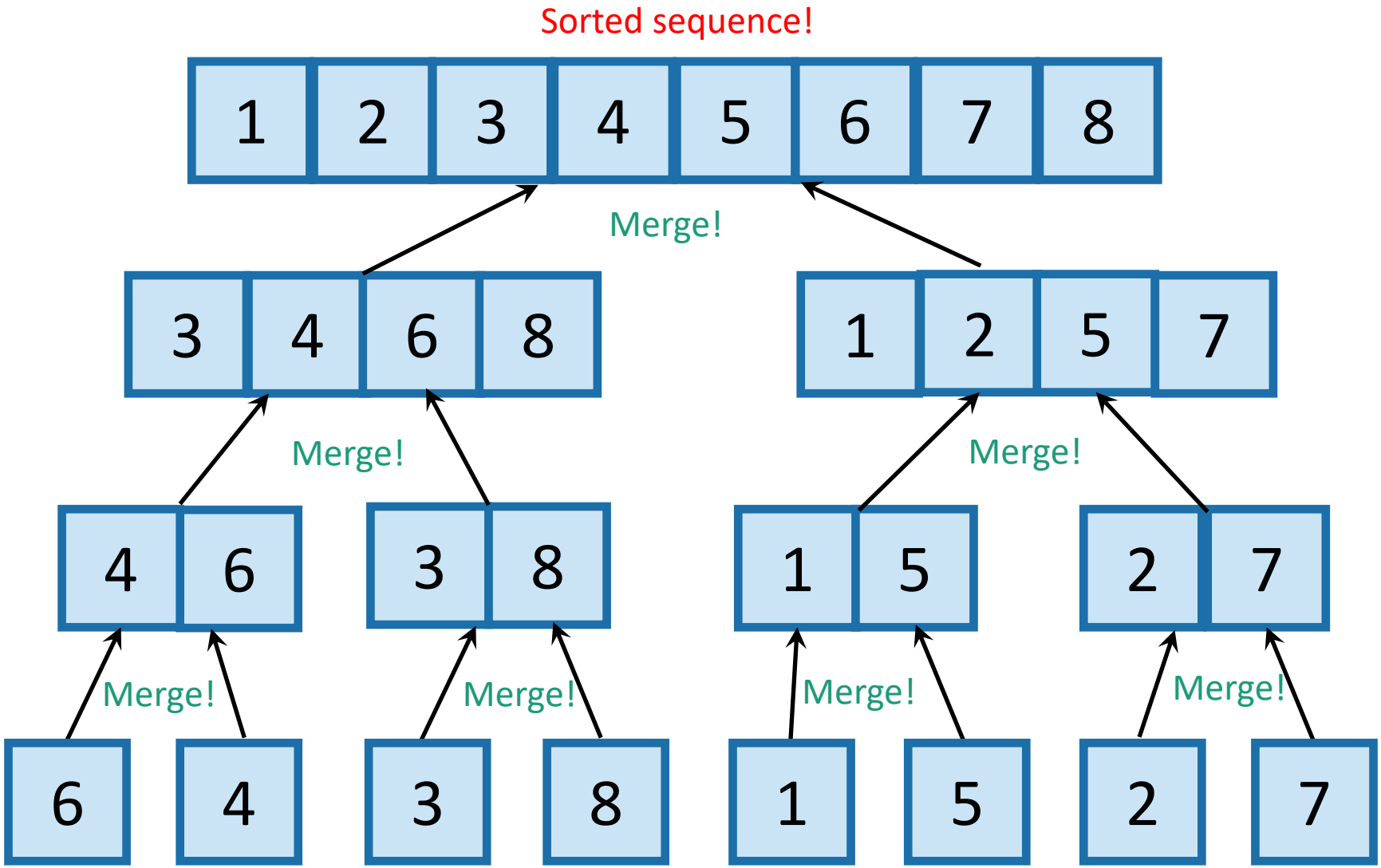
What actually happens?

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!



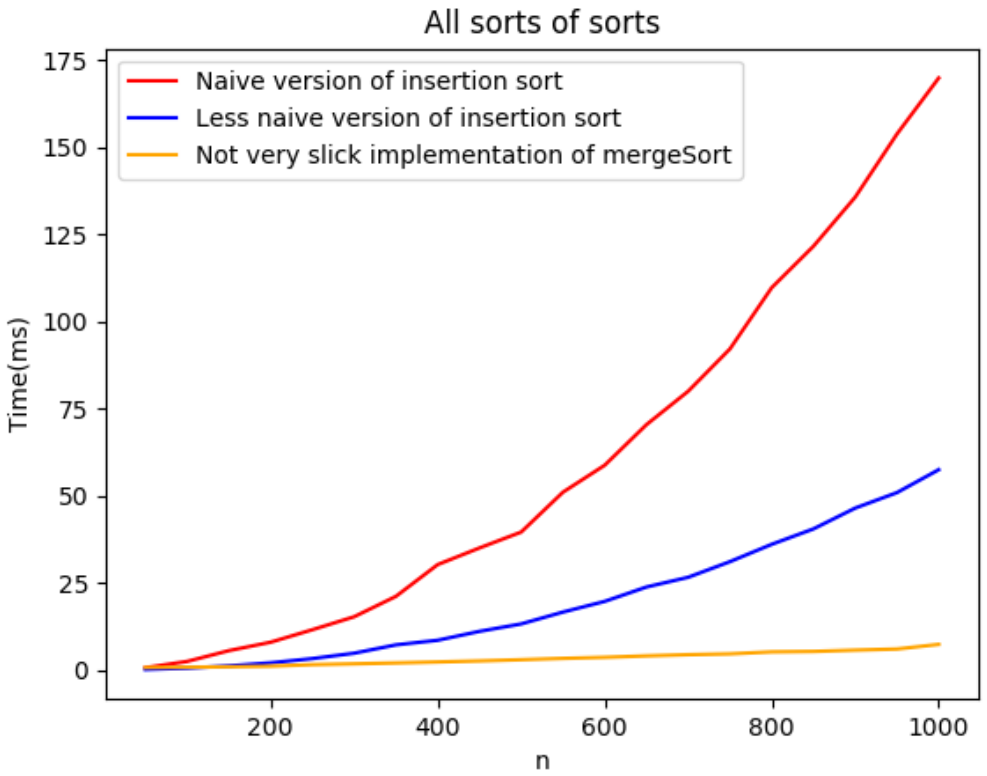
A bunch of sorted lists of length 1 (in the order of the original sequence).

Two questions

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

Empirically:

1. Seems to work.
2. Seems fast.



It works

- **Inductive hypothesis:**

  "In every iteration, the 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 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 ≤ 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! (Either do it yourself or read it in CLRS Section 2.3.1!)

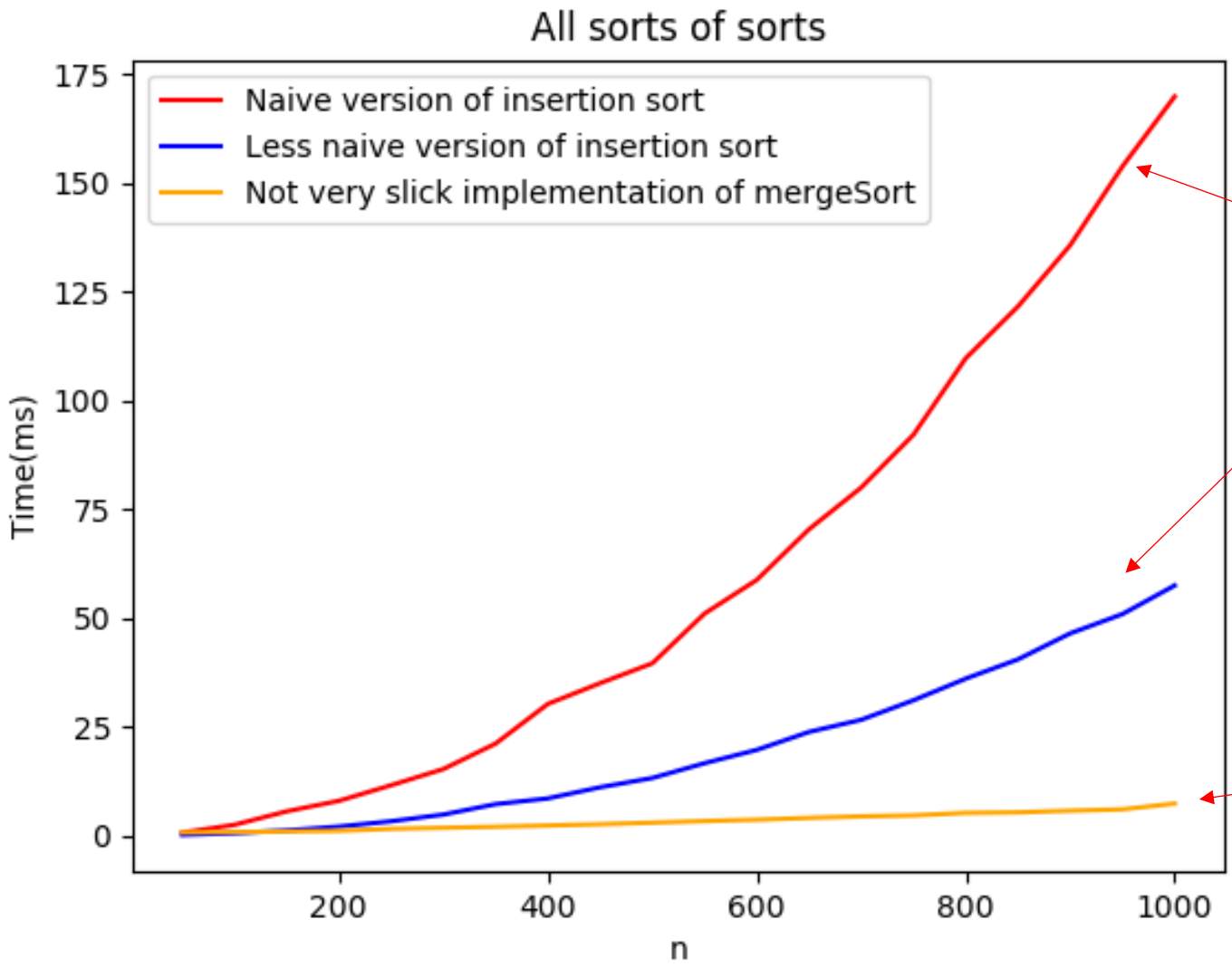
It's fast

CLAIM:

MergeSort requires at most O(n*log(n)) operations to sort n numbers.

- Proof coming soon.
- But first, how does this compare to InsertionSort?
  - Recall InsertionSort used on the order of n^2 operations.

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



Use something like n^2 operations.

Supposedly uses $\leq O(n\log(n))$ operations

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

Aside:

All logarithms in this course are base 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 ⇒ log(32) = 5

Halve 5 times

64, 32, 16, 8, 4, 2, 1 ⇒ 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. $\mathrm{O}(n^2)$? (Analytically)

- $\log(n)$ 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 $O(n^2)$!

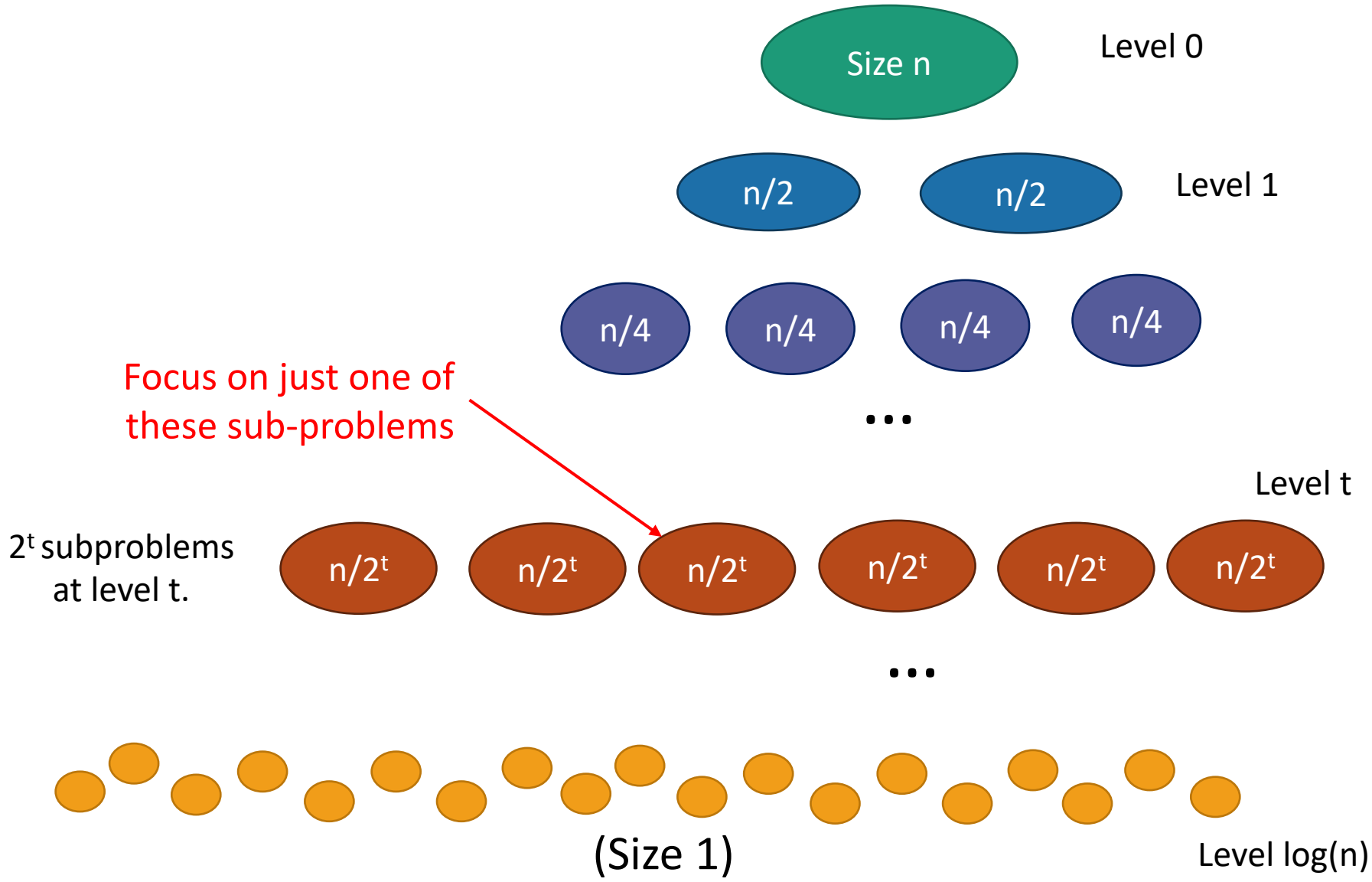
Now let's prove the claim

CLAIM:

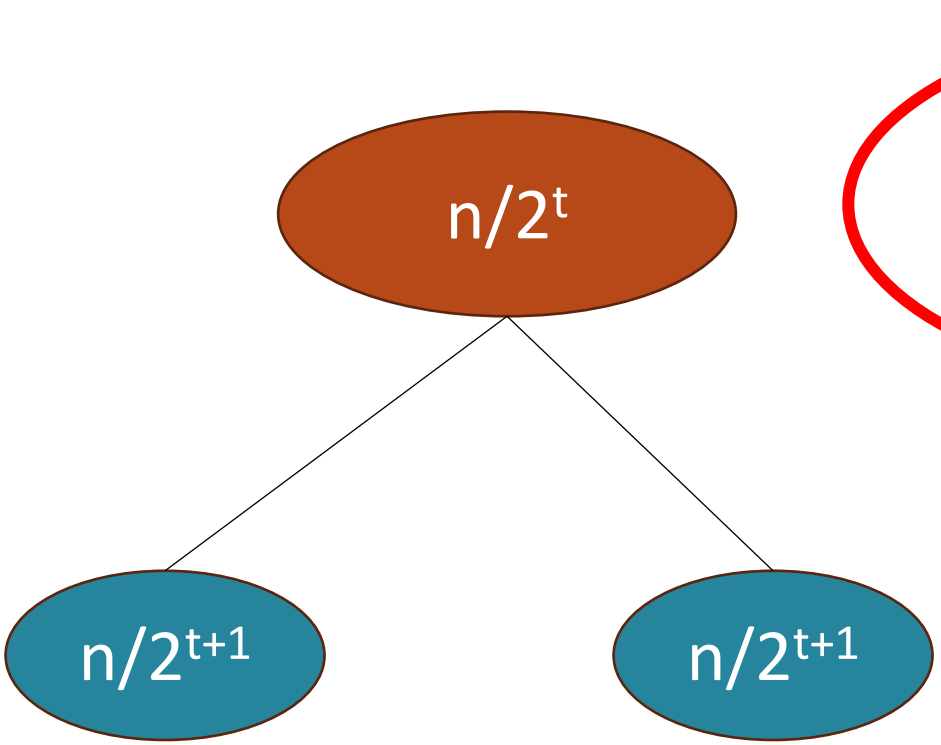
MergeSort runs in time $O(n \log(n))$

Let's prove the claim

Assume that n is a power of 2 for convenience.



How much work in this sub-problem?



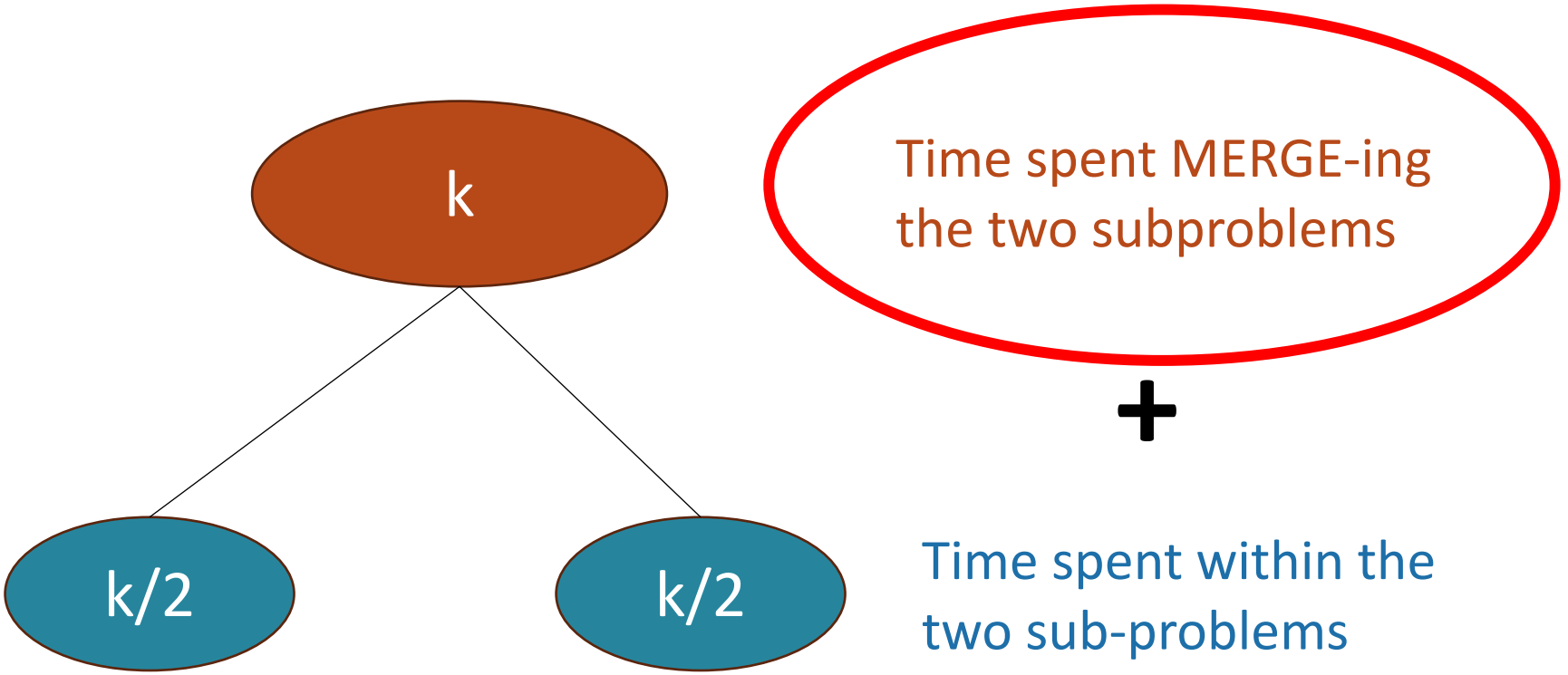
Time spent MERGE-ing the two subproblems

+

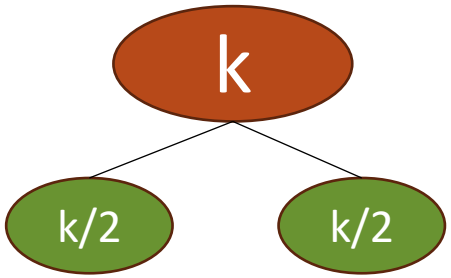
Time spent within the two sub-problems

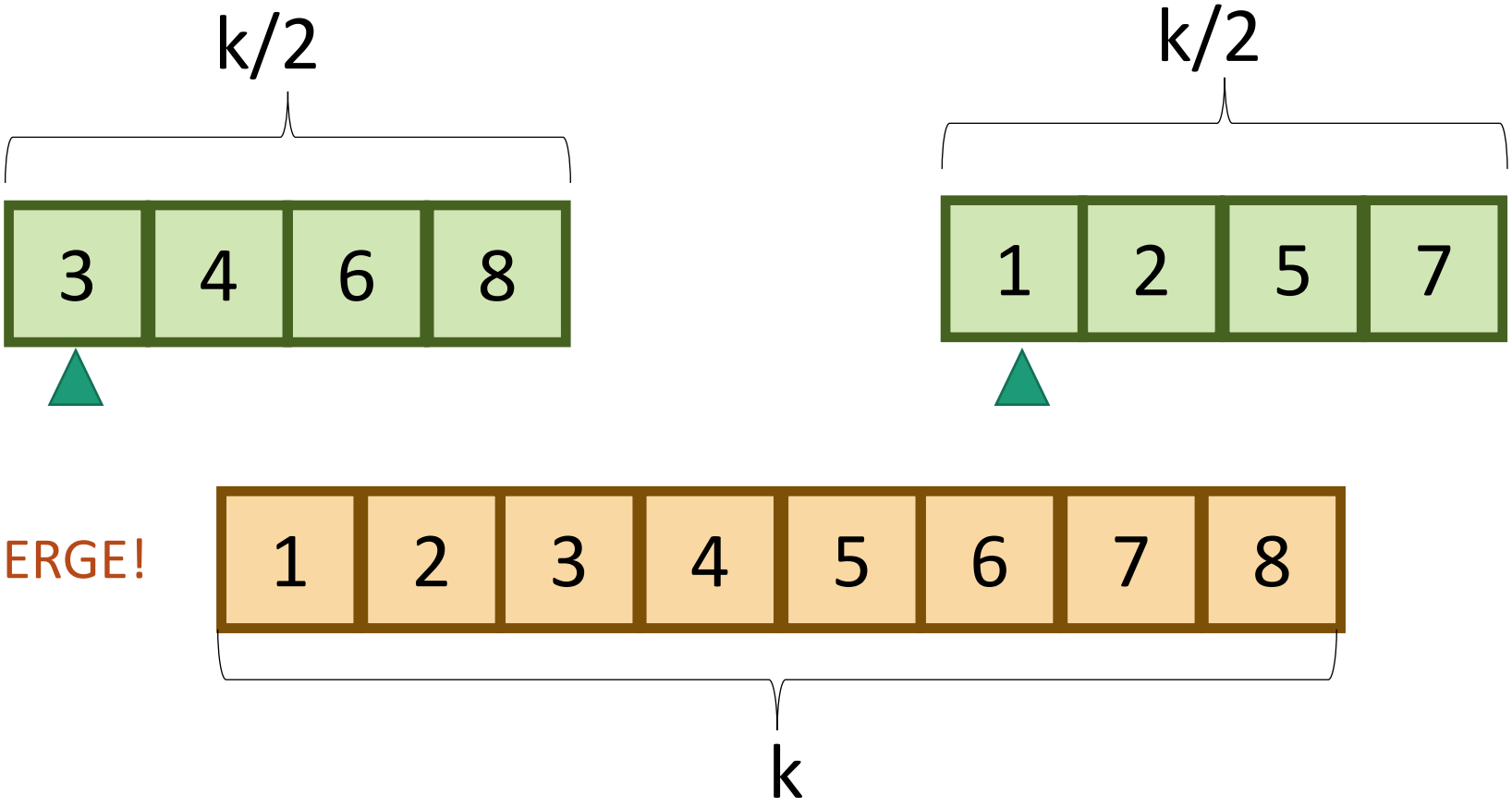
How much work in this sub-problem?

Let $k=n/2^t$…

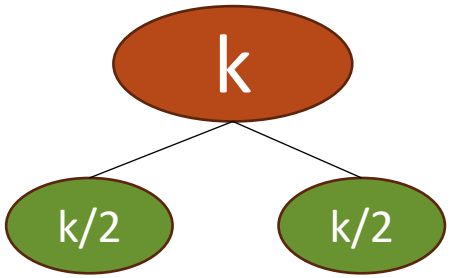


How long does it take to MERGE?

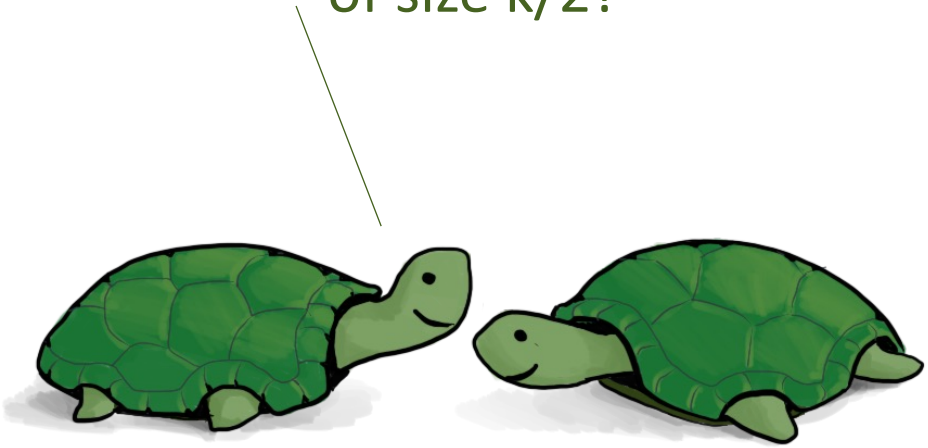




How long does it take to MERGE?



About how many operations does it take to run MERGE on two lists of size k/2?



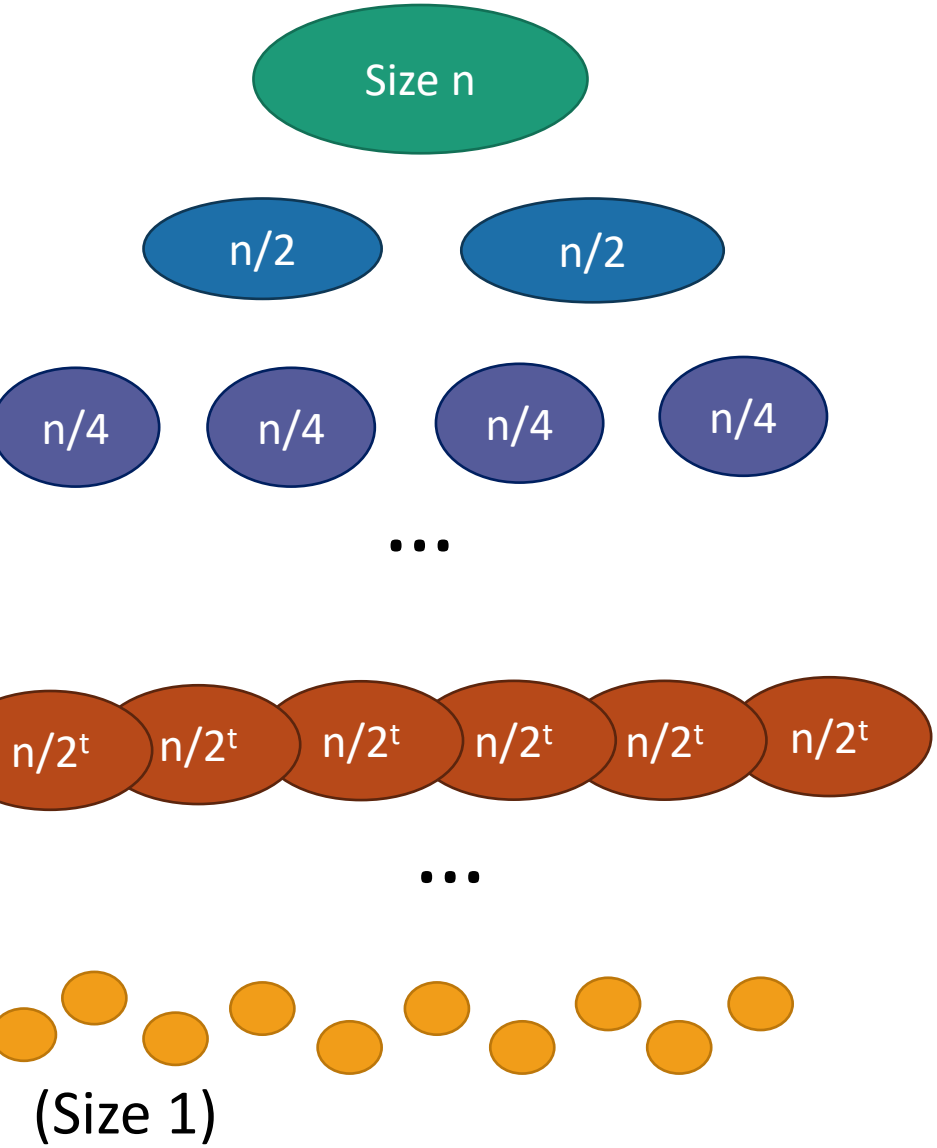
Think-Pair-Share Terrapins

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

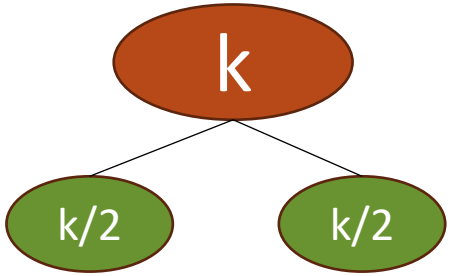
MERGE(L,R):

- n_L = length(L); n_R = length(R);
- i_L, i_R=0;
- **while** i_L < n_L **and** i_R < n_R:
  - A[i_L+i_R] = min{L[i_L],R[i_R]};
  - **if** L[i_L]<R[i_R] **then**
    - i_L++;
  - **else** i_R++;
- **while** i_L < n_L
  - A[i_L+i_R] = L[i_L];
  - i_L++;
- **while** i_R < n_R
  - A[i_L+i_R] = R[i_R];
  - i_R++;
- **return** A

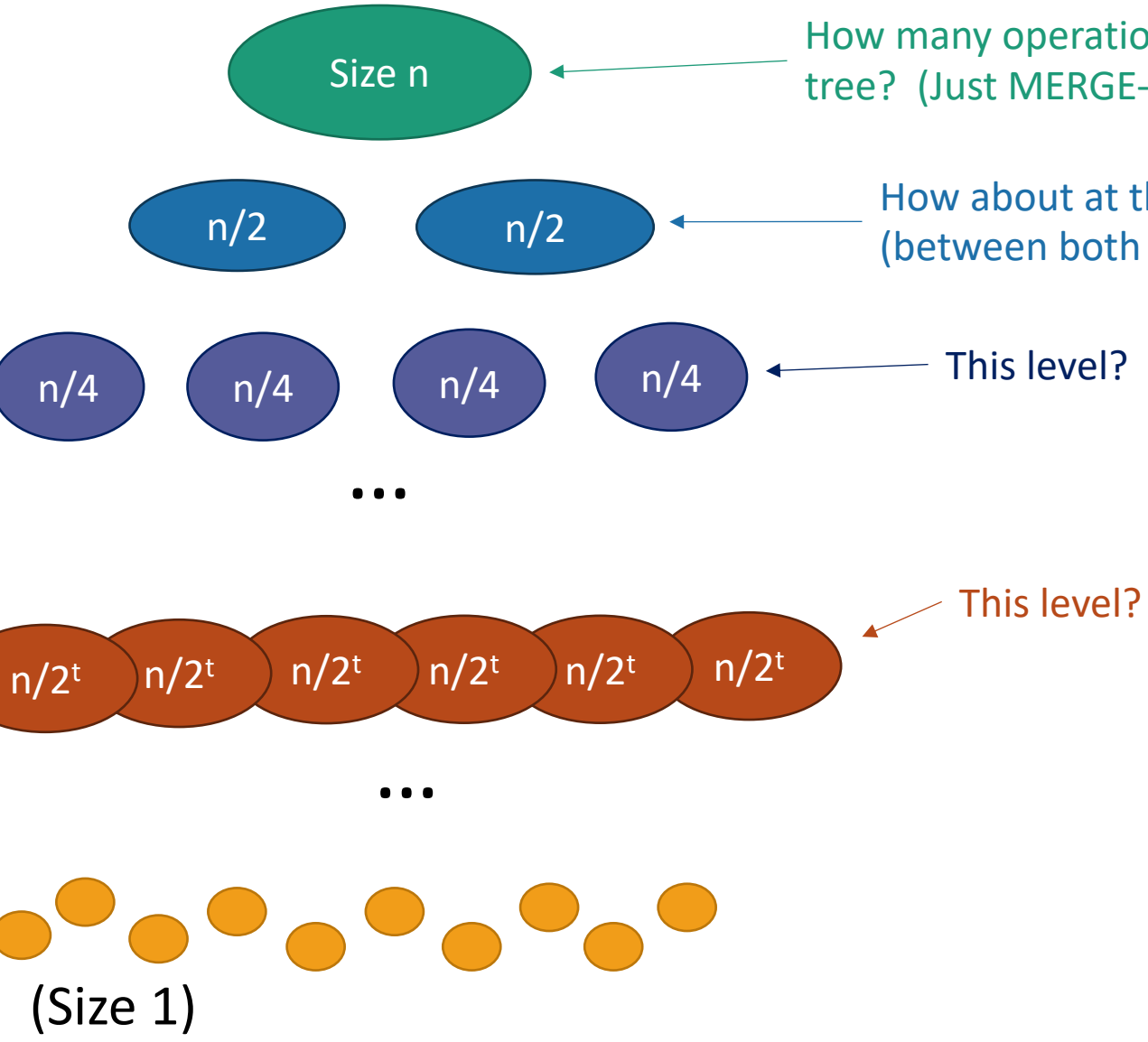
Recursion tree



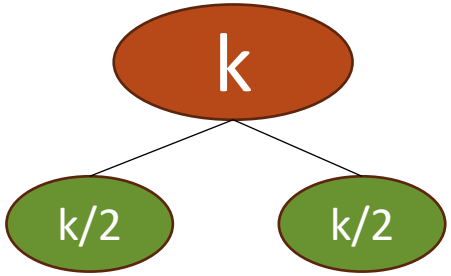
There are O(k) operations done at this node.



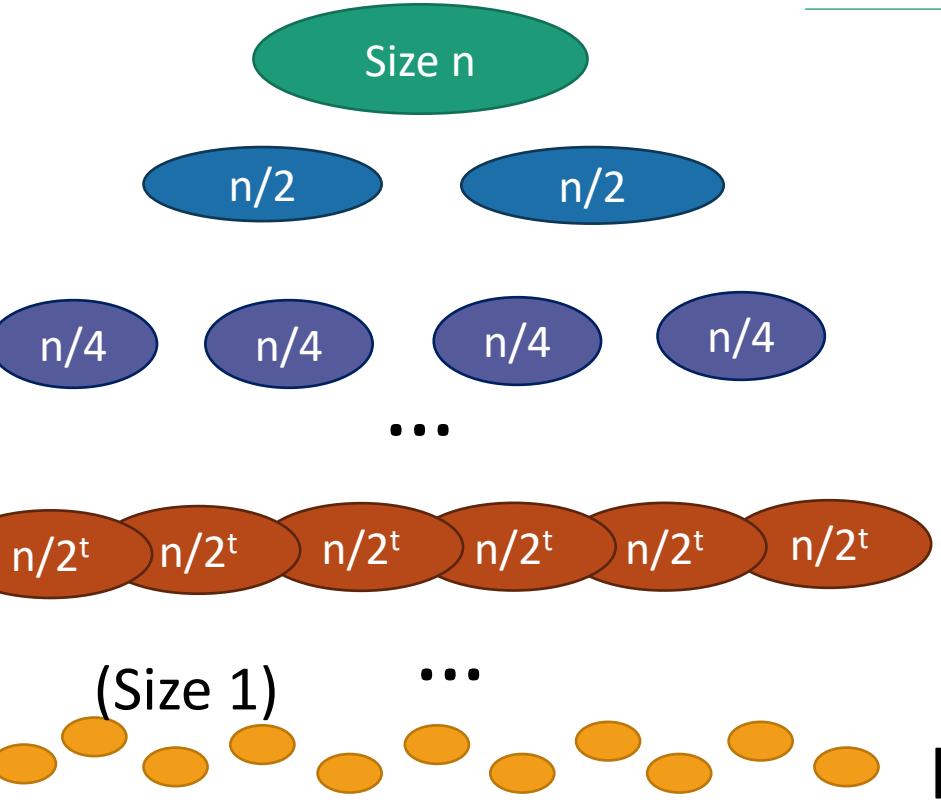
Recursion tree



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



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)

Pair-Explain Parrots

O(k) work at this node.

Total runtime…

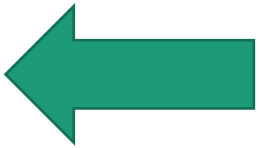
- O(n) steps per level, at every level
- log(n) + 1 levels
- O(n log(n)) total!

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

- Sorting!
- InsertionSort Algorithm
  - Does it work? Is it fast?
  - Worst-case analysis
  - Skill: Analyzing correctness of iterative and recursive algorithms.
- MergeSort Algorithm
  - Does it work? Is it fast?
    - Skill: Analyzing running time of recursive algorithms
      (part 1…more next time!)
- Embedded Ethics 

Welcome to Embedded EthiCS

Benjamin “Benji” Xie, Ph.D. (he|they)

benjixie@stanford.edu

Ece Korkmaz (she/her)

CS161 CA, Embedded EthiCS CA

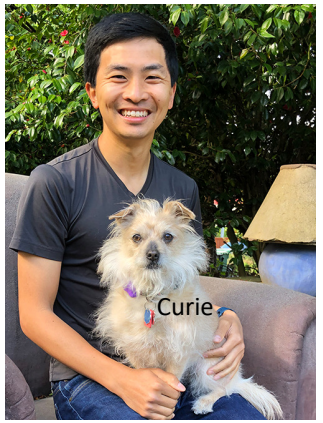
Ananya Karthik (she/her)

Embedded EthiCS Coordinator

The big questions (Redux!)

- Why are we here?
  - Who are we? Where did we come from?
- What is going on?
  - What is Embedded Ethics?
  - Why is learning about ethics part of learning about algorithms?

Hi! I'm Benjamin Xie (he|they)



- BS + MEng in CS at MIT
  - Research, industry, non-profit, & start-up experience in using data to understand ppl
- PhD in Information Science, Univ of Washington Seattle
  - human-computer interaction, computing education, critical data studies
  - Structural/societal power relationships as lens to interpret data
- Now I'm helping Stanford to develop Embedded Ethics program
  - Also research w/ community stakeholders to teach, advocate for ethical AI use

MIT EECS | Information School UNIVERSITY of WASHINGTON | HAI Stanford University Human-Centered Artificial Intelligence | Stanford McCoy Family Center for Ethics in Society

What is Embedded Ethics?

Training the next generation of computer scientists to "consider ethical issues from the outset rather than building technology and letting problems surface downstream" by integrating skills and habits of ethical analysis throughout the Stanford Computer Science curriculum.

Elan the Ethical Emu

Where is Embedded Ethics?

- Harvard (2017)
- Georgetown (2017)
- Brown (2019)
- Northeastern (2019)
- MIT (2020)
- … and many other places

- CS106A
- CS106B
- CS107
- CS109
- CS147
- CS161
- CS221
- CS234
- CS247B
- soon: CS107E, CS111, CS224n, more!

San Jose Mercury News

25 cents

Serving Northern California Since 1851

March 1, 1989

Enter. What is ethical in computer use? Return

By Tom Philp
Mercury News Staff Writer

A Stanford computer scientist and a philosopher are developing the university's first course to get students to examine the ethical implications of their use of computers.

The broad-ranging course, to be taught this spring, will deal with topics ranging from the outbreak of computer viruses to privacy issues of electronic bulletin boards. While some universities have developed courses to help students prepare for the rapidly changing computer world, no other university in Silicon Valley — or the Bay Area — now offers such a course.

"We're not trying to give them the answers," said Terry Winograd, the associate professor of computer science who is developing the course. "We're trying to get them to do good thinking."

Among the questions to be pondered: should students freely share copyrighted software? Should they be concerned if their work has military applications? Should they submit a project on deadline if they are concerned that potential bugs could ruin others' work?

For two years, Stanford has offered a seminar on computer ethics, but it was for fewer than a dozen students. But the new course, which can satisfy a curriculum requirement for computer science undergraduate students, will probably be several times larger.

"The hope is, we can take students who are currently more oriented in

See ETHICS, Page 8A

The course will address issues like invasion of privacy, ownership of computer programs, and the risks they are introducing to people's lives

For truly ethical AI, its research must be independent from big tech

Timnit Gebru

We must curb the power of Silicon Valley and protect those who speak up about the harms of AI

Mon 6 Dec 2021 08.33 EST

What do we teach?

- Issue spotting and ethical sensitivity.
- Recognizing values in **design** choices.
- Developing language to talk about moral choices.
- Professional responsibilities of computer scientists & software engineers.
- Important topics in technology ethics: bias & fairness, **inequity**, privacy, surveillance, data control & consent, trust, disinformation, participatory design, concentration of power.

Algorithms & the Good

Turn a real world problem into a formal (math) problem

Use an algorithm to solve the problem

Happiness ensues

Real world problem? I didn't see you all the way over there …

How do we make sure we aren't losing important features of the real world problem when we formalize it?

Turn a real world problem into a formal (math) problem

Use an algorithm to solve the problem

Happiness ensues

By the time you finish 161 you will have a tool kit stuffed with algorithms – which one is right for the job?

Turn a real world problem into a formal (math) problem

Use an algorithm to solve the problem

Happiness ensues

Socrates says happiness is in the life of virtue … oh okay, next slide.

Turn a real world problem into a formal (math) problem

Use an algorithm to solve the problem

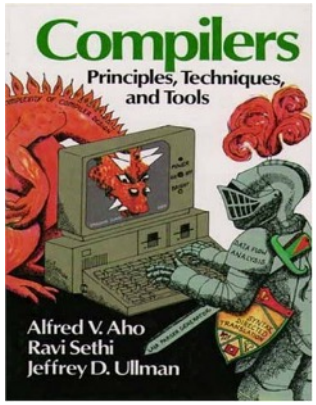
Happiness ensues

Algorithms are fundamental

Operating Systems (CS 140)

229)

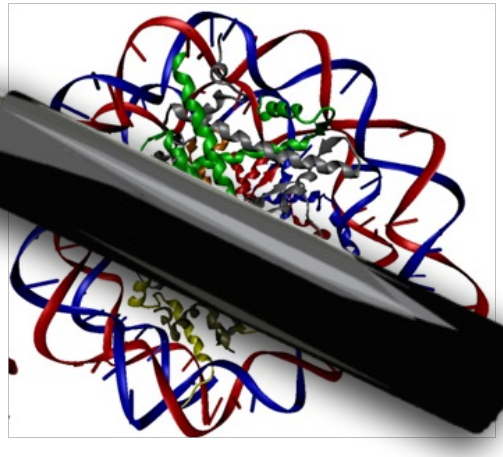
Cryptography (CS 255)



Compilers (CS 143)

The Algorithmic Lens

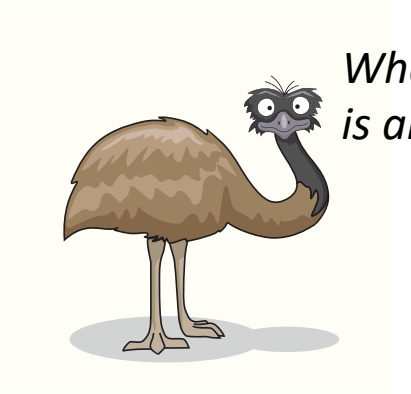
Networking (CS 144)



Computational Biology (CS 262)

If algorithms are fundamental (which they are) …

- Then some of the biggest and most consequential choices we will make as computer scientists are
  - Deciding *which problem* to solve
  - Deciding how to turn that problem into something *algorithmically tractable*
  - Deciding what to measure & optimize
  - Deciding *which algorithm* to use to solve it, tradeoffs



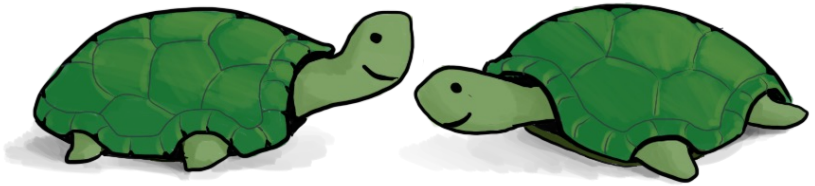
What if every decision is an ethical dilemma?!

Wait a minute ... If the thing we are doing isn't good, should we do it faster? If we could do the bad thing more efficiently, would we do more of it??

Most of the time we want our algorithms to be fast/efficient.

What is a real-world example of a case when we might we do *not* want them to be fast?

When we do not want an algorithm at all?



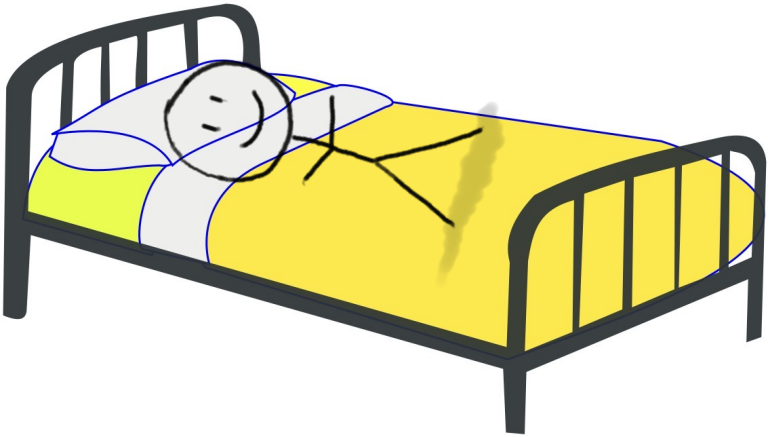
Think-Pair-Share Terrapins

Our nightmares:

Here's the first algorithm I thought of for this problem – it works okay for me!

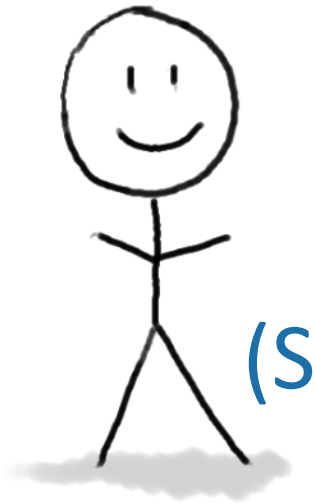
Here's a use case you didn't think of! Your nice little algorithm isn't so socially beneficial now, is it?

Anakin the adversarial aardvark



This juvenile sentencing algorithm is really inefficient – what if we made it faster and bundled it with an easy-to-use software package?

Our guiding questions:



Does it work?

Is it fast?

Can I do better?

Can I do it right?

(Should I do it all?)

Catch you soon!

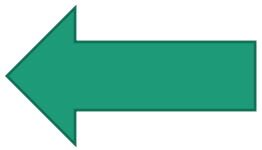
Benjamin Xie | benjixie@stanford.edu | benjixie.com | @benjixie

Want to talk about HAI/HCI/CSEd research, ethics questions, career choices, small dog meetups, etc.?

Set up a meeting at calendly.com/bxie

The plan

- Sorting!
- InsertionSort Algorithm
  - Does it work? Is it fast?
  - Worst-case analysis
  - Skill: Analyzing correctness of iterative and recursive algorithms.
- MergeSort Algorithm
  - Does it work? Is it fast?
    - Skill: Analyzing running time of recursive algorithms
      (part 1…more next time!)
- Embedded Ethics

Wrap-Up 

Next time

- A more systematic approach to analyzing the runtime of recursive algorithms.
- The Select Algorithm (this is a really cool one!)

Before next time

Friday quizzes will be in Bishop Auditorium

- Aviad's OH (Gates 164)
- ~~Here~~ on Friday: Quiz 1 (practice, but do it!)

- Please (continue to) send OAE requests to jchens@stanford.edu

Tips for quizzes in this class

- Show up on time!
  - A few minutes early is even better…
- We don’t answer questions during the quiz
  - (This policy is to make the quizzes as fair as possible)
  - We’ll do our best to make questions as clear as possible
  - If you’re still confused:
    - Do your best to find the most reasonable interpretation
    - Explicitly write any assumptions you had to make

Tips for challenge algo q on quiz

You don't need to prove correctness

We don't tell you what running time to aim for

- You're still **responsible for designing correct (and efficient) algorithms**
  - Using the critical skills you acquired by proving correctness in HW
- How good is your algorithm?
  - Incorrect = not so good…
  - Slow correct algorithms = better than incorrect
  - Fast correct algorithms = best!
- If absolutely you cannot find any correct algorithms:
  - Don't panic. These things happen.
  - You can still demonstrate knowledge of course material by **clearly stating** that your algorithm is incorrect, **explaining why** it's incorrect.

Tips for challenge algo q on quiz

You don't need to prove correctness

We don't tell you what running time to aim for

Q: Why don't we give you a target running time?

- We want you to try *your* best.
- Practice for the real world!
- Another important reason coming up in Lecture 5 ☺

Tips for challenge algo q on quiz

We'll give you an algorithms booklet with algorithms that you can use in your pseudocode

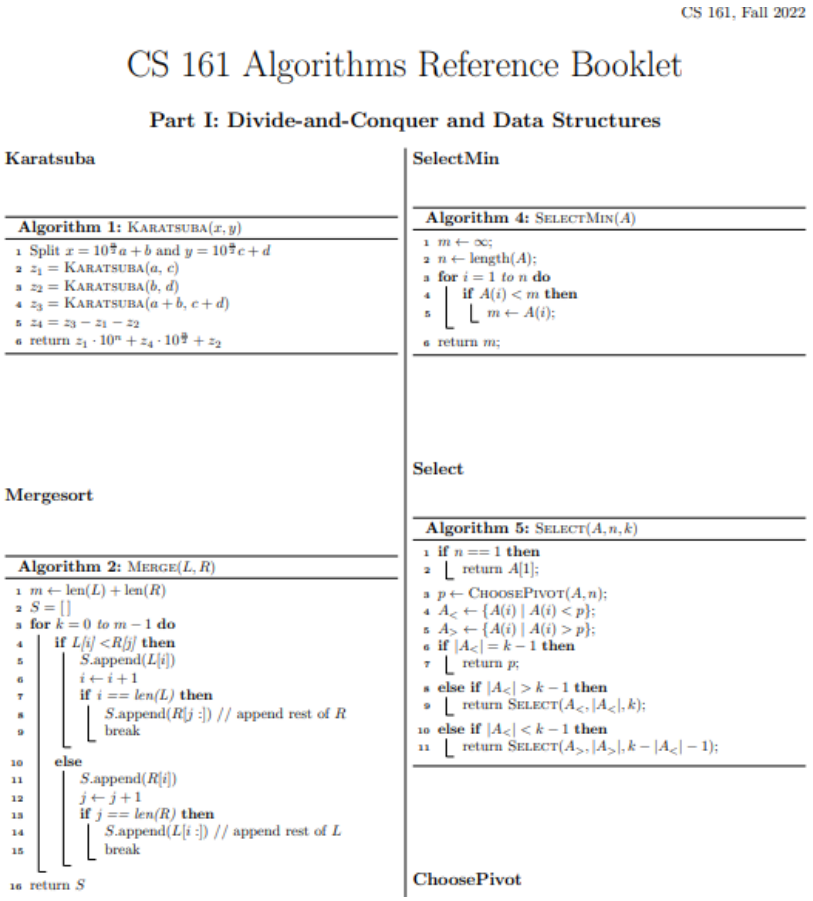
Save your time!

Your solution may call any f'n from the booklet

Your solution can even say:
"def Modified-MergeSort():
Replace Line 7 of MergeSort with
[1 clever line of pseudocode]

…

Call Modified-MergeSort()"



CS 161, Fall 2022

CS 161 Algorithms Reference Booklet

Part I: Divide-and-Conquer and Data Structures

Karatsuba

Algorithm 1: KARATSUBA(x, y)

1 Split $x = 10^{\frac{n}{2}}a + b$ and $y = 10^{\frac{n}{2}}c + d$
2 $z_1 = \text{KARATSUBA}(a, c)$
3 $z_2 = \text{KARATSUBA}(b, d)$
4 $z_3 = \text{KARATSUBA}(a + b, c + d)$
5 $z_4 = z_3 - z_1 - z_2$
6 return $z_1 \cdot 10^n + z_4 \cdot 10^{\frac{n}{2}} + z_2$

SelectMin

Algorithm 4: SELECTMIN(A)

1 $m \leftarrow \infty$;
2 $n \leftarrow \text{length}(A)$;
3 **for** $i = 1$ *to* n **do**
4 **if** $A(i) < m$ **then**
5 $m \leftarrow A(i)$;
6 return m;

Mergesort

Algorithm 2: MERGE(L, R)

1 $m \leftarrow \text{len}(L) + \text{len}(R)$
2 $S = [\,]$
3 **for** $k = 0$ *to* $m - 1$ **do**
4 **if** *L[i] <R[j]* **then**
5 *S*.append(*L[i]*)
6 $i \leftarrow i + 1$
7 **if** *i == len(L)* **then**
8 *S*.append(*R[j :]*) // append rest of *R*
9 break
10 **else**
11 *S*.append(*R[i]*)
12 $j \leftarrow j + 1$
13 **if** *j == len(R)* **then**
14 *S*.append(*L[i :]*) // append rest of *L*
15 break
16 return S

Select

Algorithm 5: SELECT(A, n, k)

1 **if** $n == 1$ **then**
2 return $A[1]$;
3 $p \leftarrow \text{CHOOSEPIVOT}(A, n)$;
4 $A_< \leftarrow \{A(i) \mid A(i) < p\}$;
5 $A_> \leftarrow \{A(i) \mid A(i) > p\}$;
6 **if** $|A_<| = k - 1$ **then**
7 return p;
8 **else if** $|A_<| > k - 1$ **then**
9 return SELECT$(A_<, |A_<|, k)$;
10 **else if** $|A_<| < k - 1$ **then**
11 return SELECT$(A_>, |A_>|, k - |A_<| - 1)$;

ChoosePivot

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.