CSE100: Design and Analysis of Algorithms
Lecture 01 – Introduction

Jan 19th 2021

Logistics and Introduction

Let's start with some logistics

- https://catcourses.ucmerced.edu/courses/19742
- Textbook(s):
  - CLRS: Introduction to Algorithms (main)
  - Kleinberg and Tardos: Algorithm Design (optional)
- Lab Assignments:
  - posted on Mondays, due in 7 days
  - See Late Policy in Syllabus for more details
- Exams:
  - Midterms Week 5, Week 9, Week 14
  - Final Week 17

The big questions

- Who are we?
  - Professor, TAs, students?
- Why are we here?
  - Why learn about algorithms?
- What is going on?
  - What is this course about?
  - Logistics?
- Wrap-up

Welcome to CSE 100!

Who are we?

- Instructor:
  - Santosh Chandrasekhar
- Awesome TAs:
  - Maryam Khazaei Pool
  - Shubham Rohal
  - Ghazal Zand

Who are you?

- CSE majors…
- Some Math

Why are you here?

You are better equipped to answer this question than I am, but I'll give it a go anyway…

- Algorithms are fundamental
- Algorithms are useful.
- Algorithms are fun!
- CSE100 is a required course.

Why is CSE100 required?

- Algorithms are fundamental.
- Algorithms are useful.
- Algorithms are fun!

Algorithms are fundamental

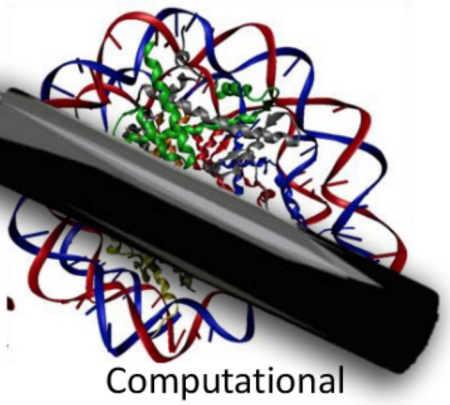
Operating Systems (CSE 150)

Machine learning (CSE 176)

Computers and Network Security (CSE 178)

Database Systems (CSE 111)

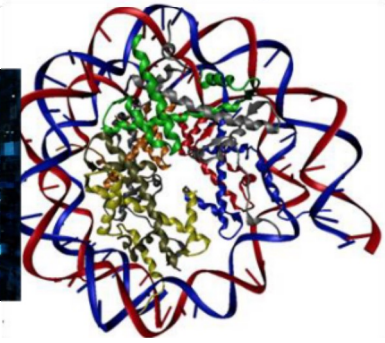
Networking (CSE 160)



Computational Neuroscience (CSE 173)

Algorithms are useful

- All those things, without UC Merced CSE course numbers
- As we get more and more data and problem sizes get bigger and bigger, algorithms become more and more important.
- Will help you get a job.



Algorithms are fun!

- Algorithm design is both an art and a science.
- Many surprises!
- A young field, many exciting research questions!

- (Will help you get a job you like!)

What's going on?

- Course goals/overview
- Logistics

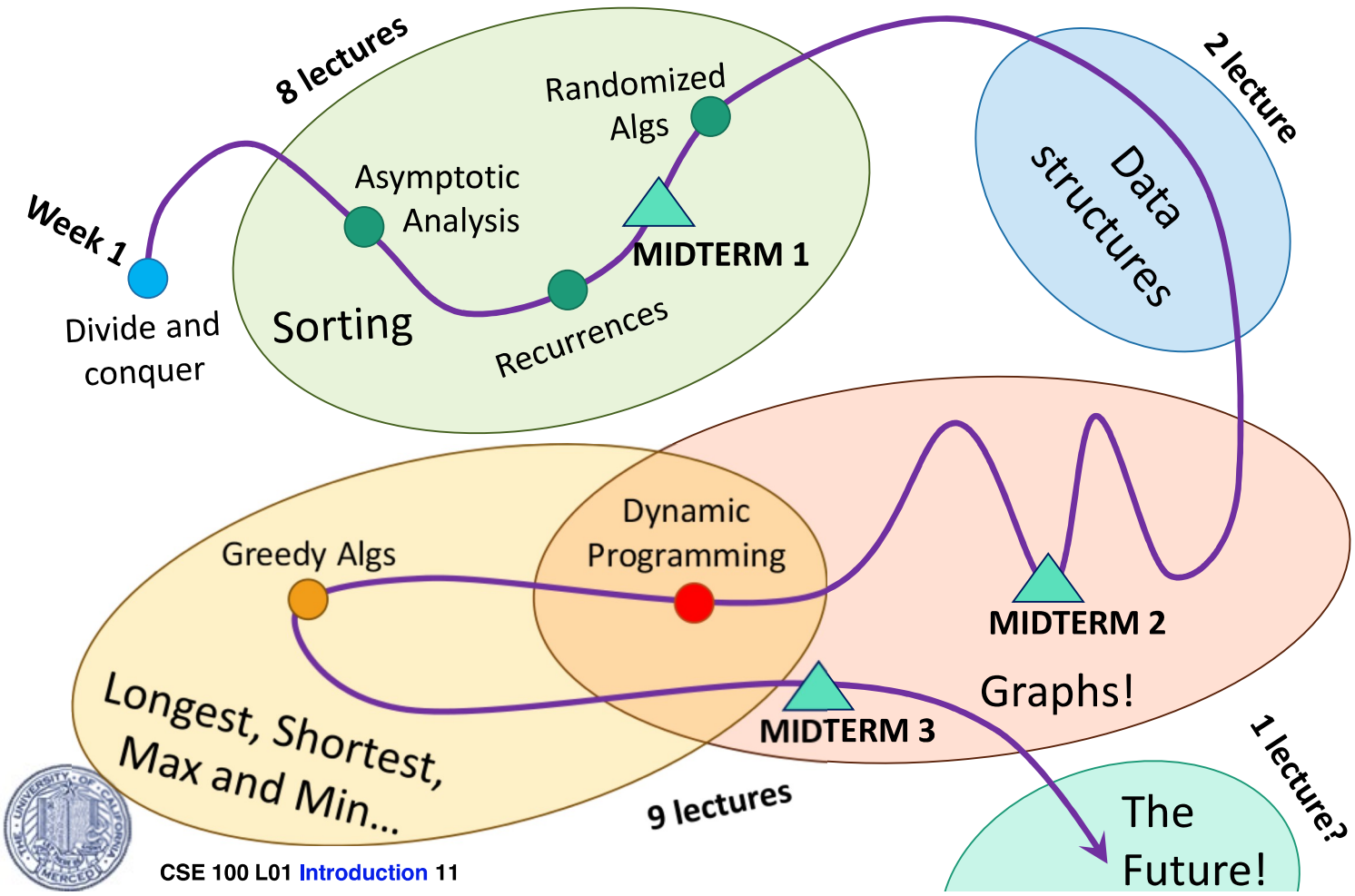
Course goals

- The design and analysis of algorithms
  - These go hand-in-hand

- In this course you will:
  - Learn to think analytically about algorithms
  - Flesh out an "algorithmic toolkit"
  - Learn to communicate clearly about algorithms

More detailed schedule on the website!

Roadmap

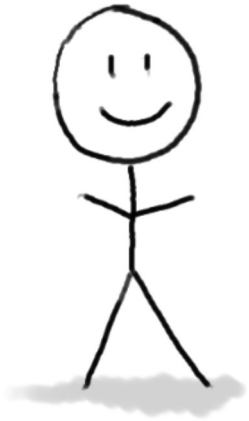


Our guiding questions:

Does it work?

Is it fast?

Can I do better?



Algorithm designer

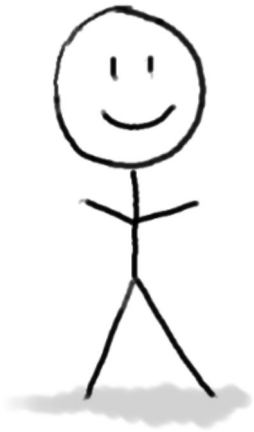
Our internal monologue…

What exactly do we mean by better? And what about that corner case? Shouldn't we be zero-indexing?

Plucky the Pedantic Penguin

Detail-oriented
Precise
Rigorous

Does it work?
Is it fast?
Can I do better?



Dude, this is just like that other time. If you do the thing and the stuff like you did then, it'll totally work real fast!

Lucky the Lackadaisical Lemur

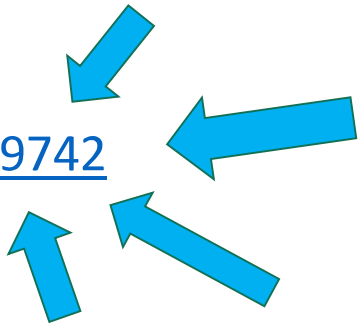
Big-picture
Intuitive
Hand-wavey

Both sides are necessary!

We will feel this tension throughout the course

- In lecture, I will channel Lucky maybe a bit more than I should.
- On Problems and Labs, you should lean a bit more towards Plucky.

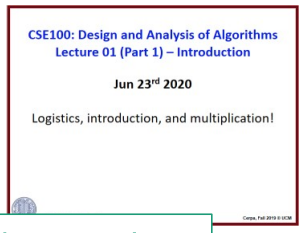
Course elements and resources



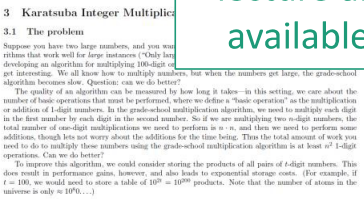
- Course website:
  - https://catcourses.ucmerced.edu/courses/19742
- Lectures and Notes
- Textbook
- Lab Discussions
- Lab Assignments
- Exams
- Office hours (posted by end of Week) or meetings by appointment.

Lectures

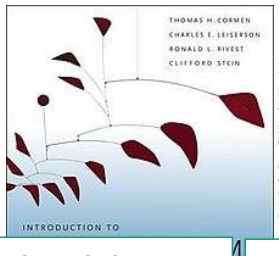
- TR 3:00-4:15pm via Zoom
- Resources available:
  - Slides, Notes, Book, Discussion, Lab Assignments



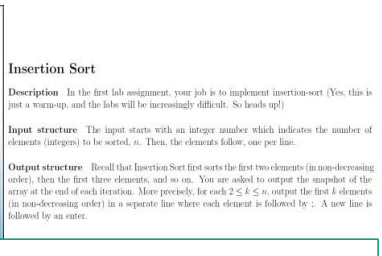
Slides are the slides from lecture.



Notes from lecture are available!



Textbook has mathy details that slides may omit

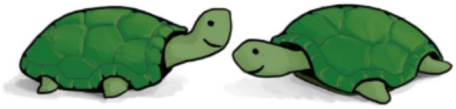


Discussion and Lab Assignments have implementation details that slides may omit.

- Goal of lectures:
  - Hit the most important points of the week's material
    - Sometimes high-level overview
    - Sometimes detailed examples

How to get the most out of lectures

- **During lecture:**
  - Show up or tune in, ask questions.
  - Engage with in-class questions.
- **Before lecture:**
  - Do ***the reading suggested*** on the website.
- **After lecture:**
  - Go through the exercises on the slides.



Think-Pair-Share Terrapins (in-class questions)

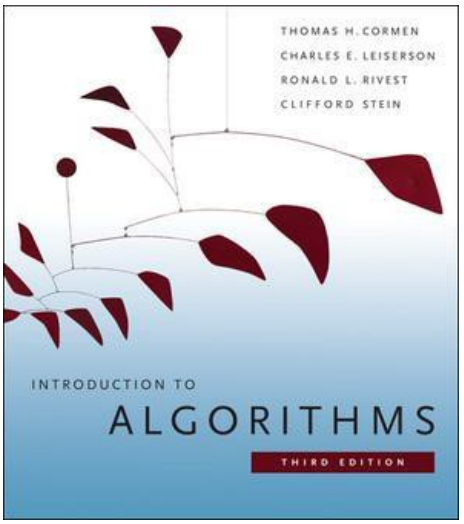
Siggi the Studious Stork (recommended exercises)

Ollie the Over-achieving Ostrich (challenge questions)

- ***Do the reading***
  - either before or after lecture, whatever works best for you.
  - **do not wait to "catch up" the week before the exam.**

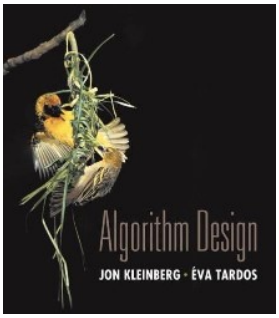
Textbook

- **CLRS**:
  - Introduction to Algorithms, by Cormen, Leiserson, Rivest, and Stein.

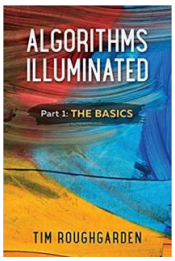


- **Notes**:
  - We will provide summary notes for many lectures
  - They are not intended to replace the book!

We will also sometimes refer to "Algorithm Design" by Kleinberg and Tardos



"Algorithms Illuminated" by Tim Roughgarden is also a great reference.



Lab Discussion

Weekly Lab Discussion in two parts:

1. Exercises:
   - Check-your-understanding and computations
   - Should be pretty straightforward
   - We recommend you do these on your own

2. Problems:
   - Proofs and algorithm design
   - Not straightforward
   - You may collaborate with your classmates
     (WITHIN REASON: See website for collaboration policy!)

They will help you prepare for the midterms

Lab Assignments

- Lab assignments are algorithm implementations in C++.
  - You will write C++ code, and test it with test cases.
- You will need to learn ***some*** C++ if you don't know.
  - For next week, the ***Lab Assignment 00*** is to get started with lab assignments and the grader system.
  - See course website for details.

- The goal is to make the algorithms (and their runtimes) more tangible.
- It is not the goal to become a super C++ programmer.
  - (Although if that happens that's cool).

How to get the most out of discussions

- Do the exercises on your own.
- Try the problems on your own **before** talking to a classmate.

- If you get help from TA:
  - **Try the problem first**.
  - Ask: **"I was trying this approach and I got stuck here."**
  - After you've figured it out, **write up your solution from scratch**, without the notes you took during office hours.

Exams

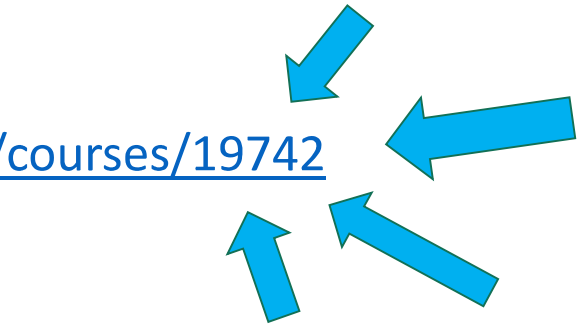
- There will be 3 **midterms** and a **final**. All take home (posted W, due F).
  - **MIDTERM 1:** Week 5
  - **MIDTERM 2:** Week 9
  - **MIDTERM 3:** Week 14
  - **FINAL:** Week 17

- We will release discussion solutions before each.

- Cannot be rescheduled!

Course elements and resources

- Course website:
  - https://catcourses.ucmerced.edu/courses/19742
- Lectures and Notes
- Textbook
- Lab Discussions
- Lab Assignments
- Exams
- Office hours (by appointment or dur[illegible] sessions)



Bug bounty!

- We hope all slides and notes will be bug-free.
- Howover, I sometmes maek typos.
- **If you find a typo** (that affects understanding*) on slides, notes, discussion or lab assignments:
  - Let us know! (Email schandrasekhar@ucmerced.edu).

Bug Bounty Hunter

*So, typos lke thees onse don't count, although please point those out too. Typos like 2 + 2 = 5 do count, as does pointing out that we omitted some crucial information.

A note on course policies

- Course policies are listed on the website.
- Read them and adhere to them.
- That's all I'm going to say about course policies.

The big questions

- Who are we?
  - Professor, TA's, students?
- Why are we here?
  - Why learn about algorithms?
- What is going on?
  - What is this course about?
  - Logistics?
- Wrap-up

Wrap up

- https://catcourses.ucmerced.edu/courses/19742
- Algorithms are fundamental, useful and fun!

- In this course, we will develop both algorithmic intuition and algorithmic technical chops

Next time

- Karatsuba Integer Multiplication
- Algorithmic Technique:
  - Divide and conquer
- Algorithmic Analysis tool:
  - Intro to asymptotic analysis

CSE100: Design and Analysis of Algorithms
Lecture 02 – Introduction (cont)

Jan 21st 2021

Multiplication

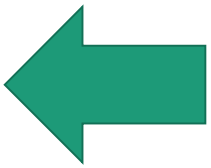
The big questions

- Who are we?
  - Professor, TA's, students?
- Why are we here?
  - Why learn about algorithms?
- What is going on?
  - What is this course about?
  - Logistics?
- Can we multiply integers?
  - And can we do it quickly?

Course goals

- Think analytically about algorithms
- Flesh out an "algorithmic toolkit"
- Learn to communicate clearly about algorithms

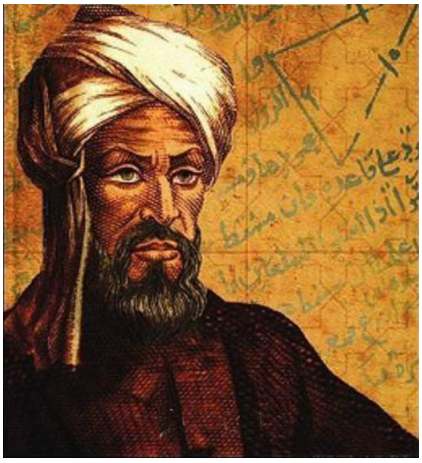
Today's goals

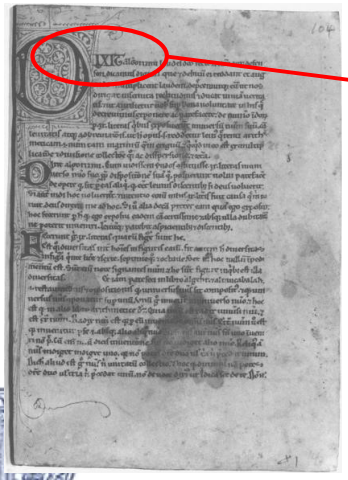
- Karatsuba Integer Multiplication 
- Algorithmic Technique:
  - Divide and conquer
- Algorithmic Analysis tool:
  - Intro to asymptotic analysis

Let’s start at the beginning

Etymology of “Algorithm”

- Al-Khwarizmi was a 9th-century scholar, born in present-day Uzbekistan, who studied and worked in Baghdad during the Abbassid Caliphate.
- Among many other contributions in mathematics, astronomy, and geography, he wrote a book about how to multiply with Arabic numerals.
- His ideas came to Europe in the 12th century.





Dixit algorizmi
(so says Al-Khwarizmi)

- Originally, “Algorisme” [old French] referred to just the Arabic number system, but eventually it came to mean “Algorithm” as we know today.

This was kind of a big deal

XLIV × XCVII = ?

$$\begin{array}{r} 44 \\ \times\ 97 \\ \hline \end{array}$$

A problem you all know how to solve:

Integer Multiplication

$$\begin{array}{r} 12 \\ \times\ 34 \\ \hline \end{array}$$

A problem you all know how to solve:

Integer Multiplication

$$
\begin{array}{r}
1234567895931413 \\
\times \quad 4563823520395533 \\
\hline
\end{array}
$$

A problem you all know how to solve:

Integer Multiplication

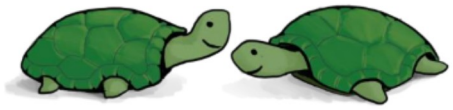
n

```
  12339257207527523846237642835683649183745238562 98
x 45623235823423952856234672350191307501353500137 53
```

???

How fast is the grade-school multiplication algorithm?

(How many one-digit operations?)



Think-pair-share Terrapins

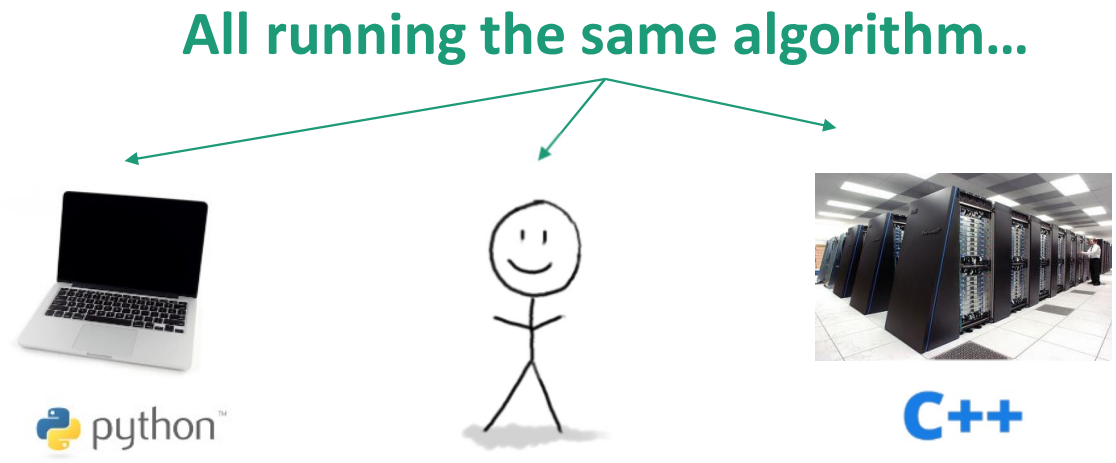
About n^2 one-digit operations

Plucky the Pedantic Penguin

At most n^2 multiplications,
and then at most n^2 additions (for carries)
and then I have to add n different 2n-digit numbers…

Does that answer the question?

- What does it mean for an algorithm to be “fast”?

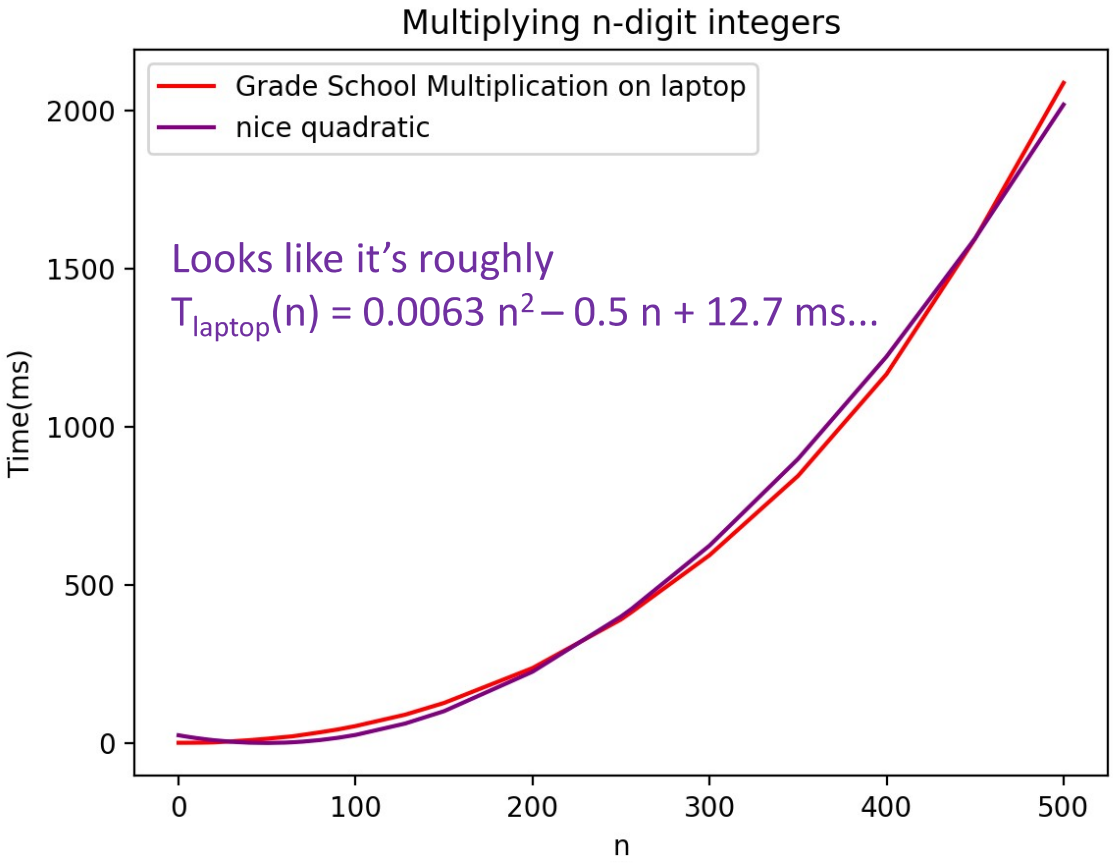


- When we say “About n^2 one-digit operations”, we are measuring how the runtime scales with the size of the input.

highly non-optimized

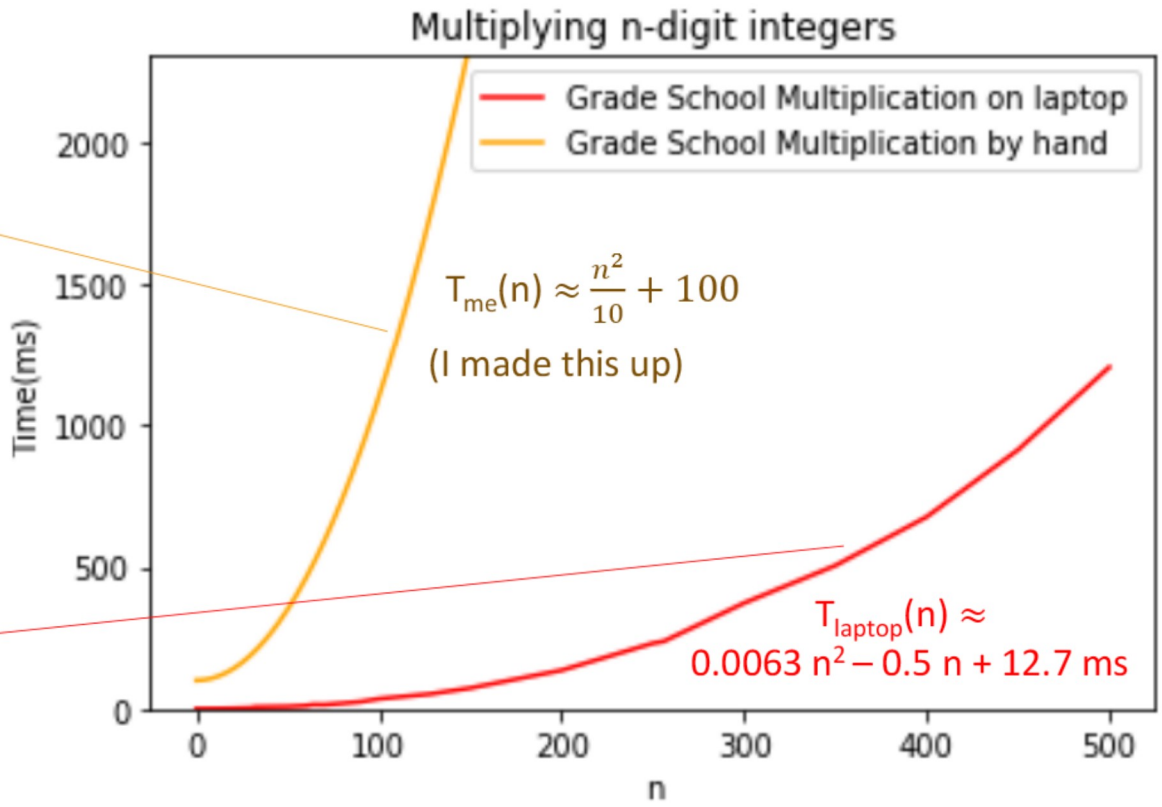
Implemented in Python, on my laptop

The runtime "scales like" n^2



Implemented by hand

The runtime still “scales like” n^2



python™

Super Informally…

Asymptotic Analysis

- We will say Grade School Multiplication "runs in time $O(n^2)$"
- Formalizes what it means to "scale like n^2"
- We will see a formal definition next lecture.
- Informally, definition-by-example:

Number of milliseconds on an input of size n	Asymptotic Running Time
$\frac{1}{10} \cdot n^2 + 100$	$O(n^2)$
$0.063 \cdot n^2 - .5\,n + 12.7$	$O(n^2)$
$100 \cdot n^2 - 10^{10000}\sqrt{n}$	$O(n^2)$
$\frac{1}{10}\, n^{1.6} + 100$	$O(n^{1.6})$

(Only pay attention to the largest power of n that appears.)

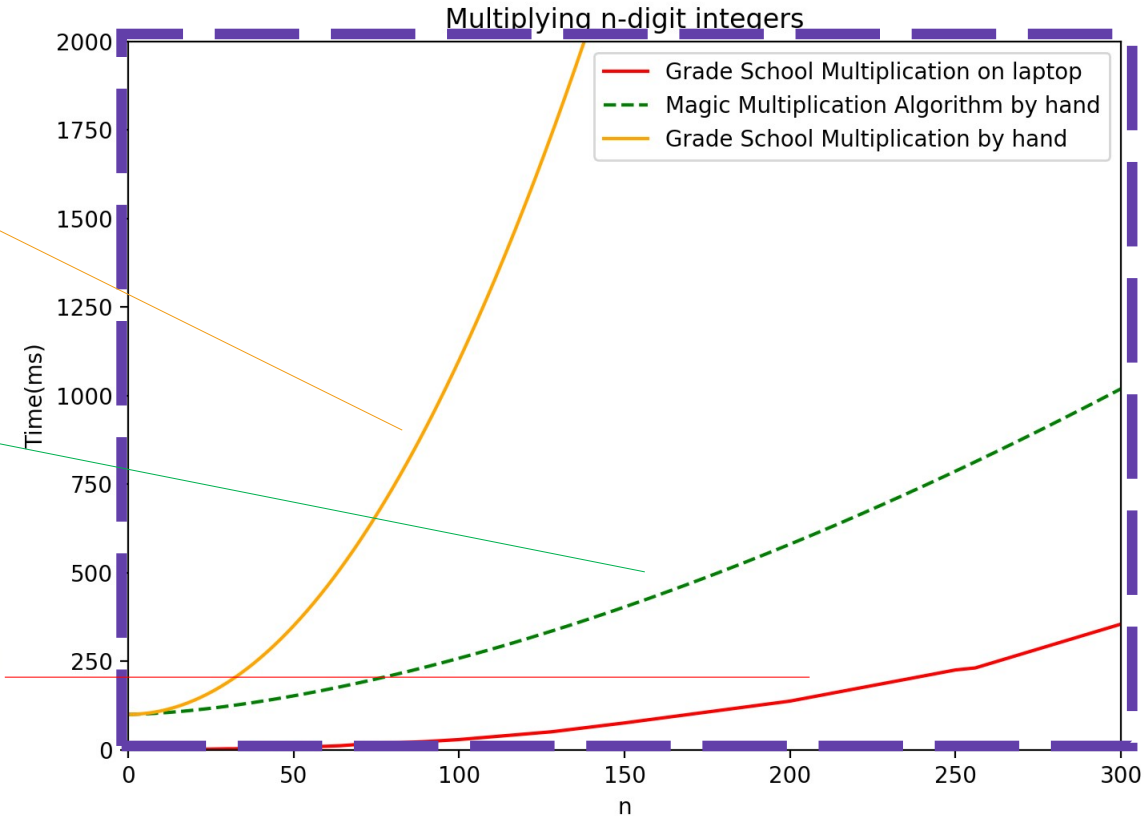
We say this algorithm is "asymptotically faster" than the others.

Why is asymptotic analysis meaningful?

$\approx \frac{n^2}{10} + 100$

$\approx \frac{n^{1.6}}{10} + 100$

$\approx .0063n^2$

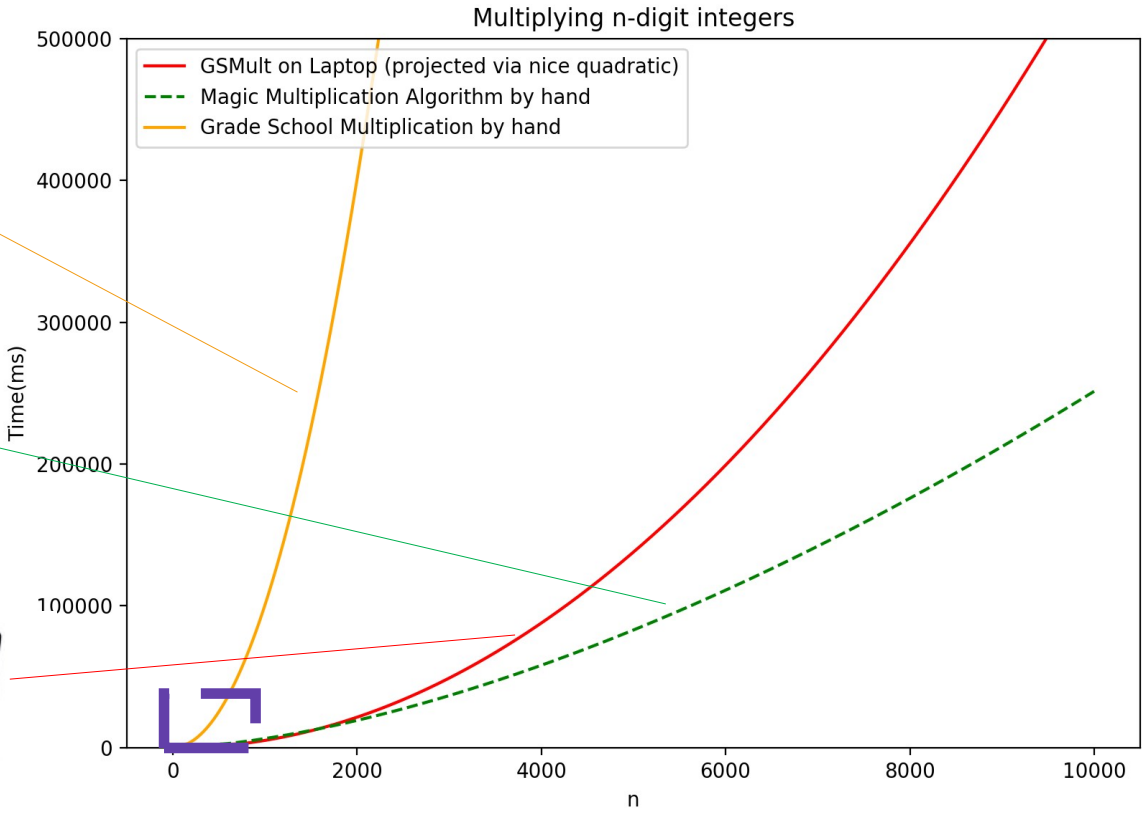


Let n get bigger…

$\approx \frac{n^2}{10} + 100$

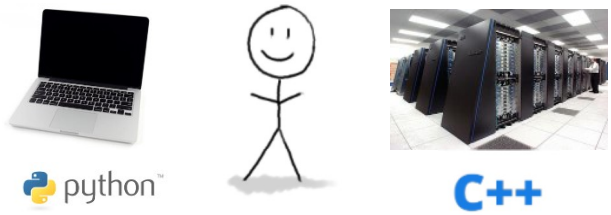
$\approx \frac{n^{1.6}}{10} + 100$

$\approx .0063n^2$



Asymptotic analysis is meaningful

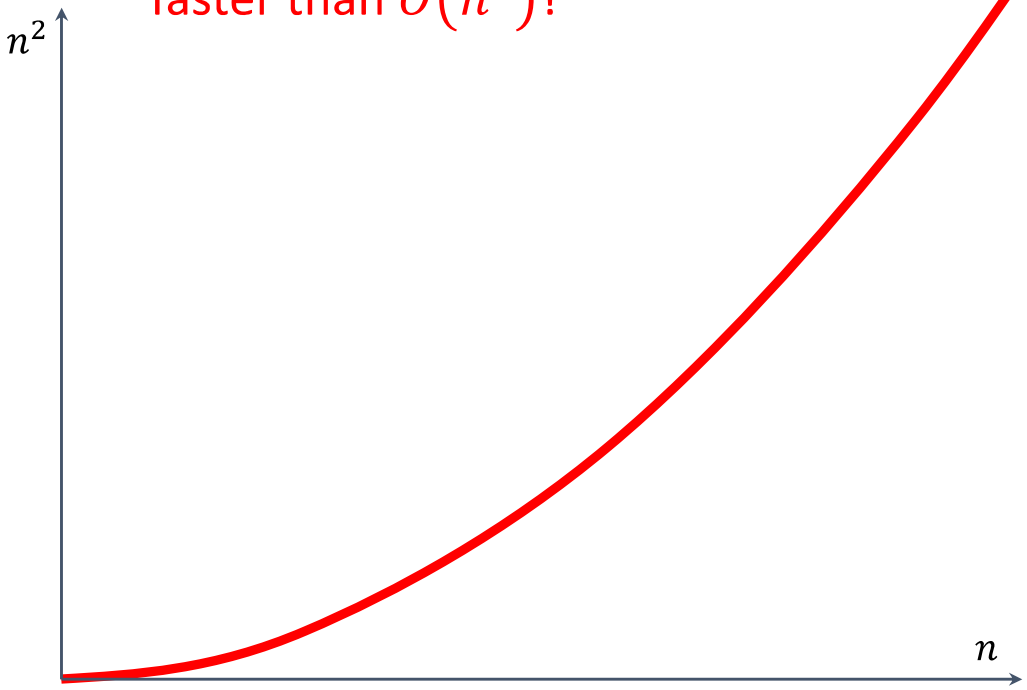
- For large enough input sizes, the “asymptotically faster” will be faster than the “asymptotically slower” one, no matter what your computational platform.



- So the question is…

Can we do better?

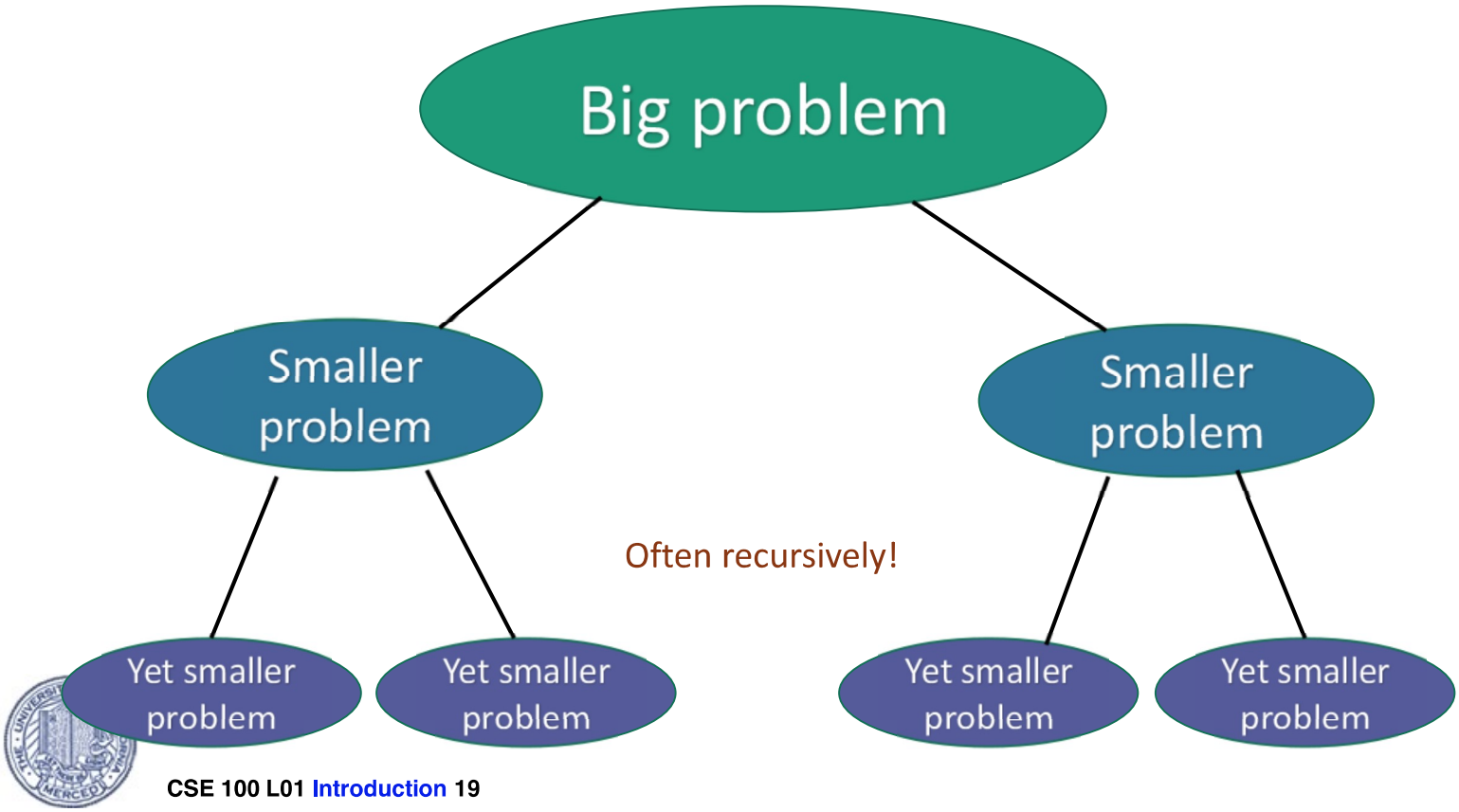
Can we multiply n-digit integers faster than $O(n^2)$?



Let’s dig in to our algorithmic toolkit…

Divide and conquer

Break problem up into smaller (easier) sub-problems



Often recursively!

Divide and conquer for multiplication

Break up an integer:

$$1234 = 12 \times 100 + 34$$

$$1234 \times 5678$$
$$= (12 \times 100 + 34) (56 \times 100 + 78)$$
$$= (12 \times 56)10000 + (34 \times 56 + 12 \times 78)100 + (34 \times 78)$$

1 2 3 4

One 4-digit multiply → Four 2-digit multiplies

More generally

Break up an n-digit integer:

$$[x_1 x_2 \cdots x_n] = [x_1 x_2 \cdots x_{n/2}] \times 10^{n/2} + [x_{n/2+1} x_{n/2+2} \cdots x_n]$$

$$\begin{aligned} x \times y &= (a \times 10^{n/2} + b)(c \times 10^{n/2} + d) \\ &= \underbrace{(a \times c)10^n}_{1} + \underbrace{(a \times d}_{2} + \underbrace{c \times b)10^{n/2}}_{3} + \underbrace{(b \times d)}_{4} \end{aligned}$$

One n-digit multiply → Four (n/2)-digit multiplies

Divide and conquer algorithm

not very precisely…

(Assume n is a power of 2…)

x,y are n-digit numbers

Multiply(x, y):

- **If** n=1:
  - **Return** xy

Base case: I've memorized my 1-digit multiplication tables…

- Write $x = a\ 10^{\frac{n}{2}} + b$
- Write $y = c\ 10^{\frac{n}{2}} + d$

a, b, c, d are n/2-digit numbers

- 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

Make this pseudocode more detailed! How should we handle odd n? How should we implement "multiplication by 10^n"?

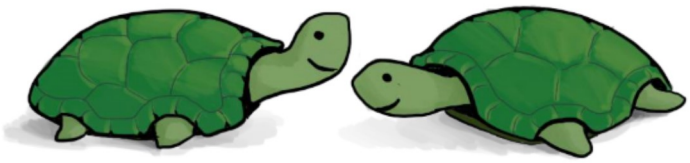
Siggi the Studious Stork

Think-Pair-Share

- We saw that this 4-digit multiplication problem broke up into four 2-digit multiplication problems

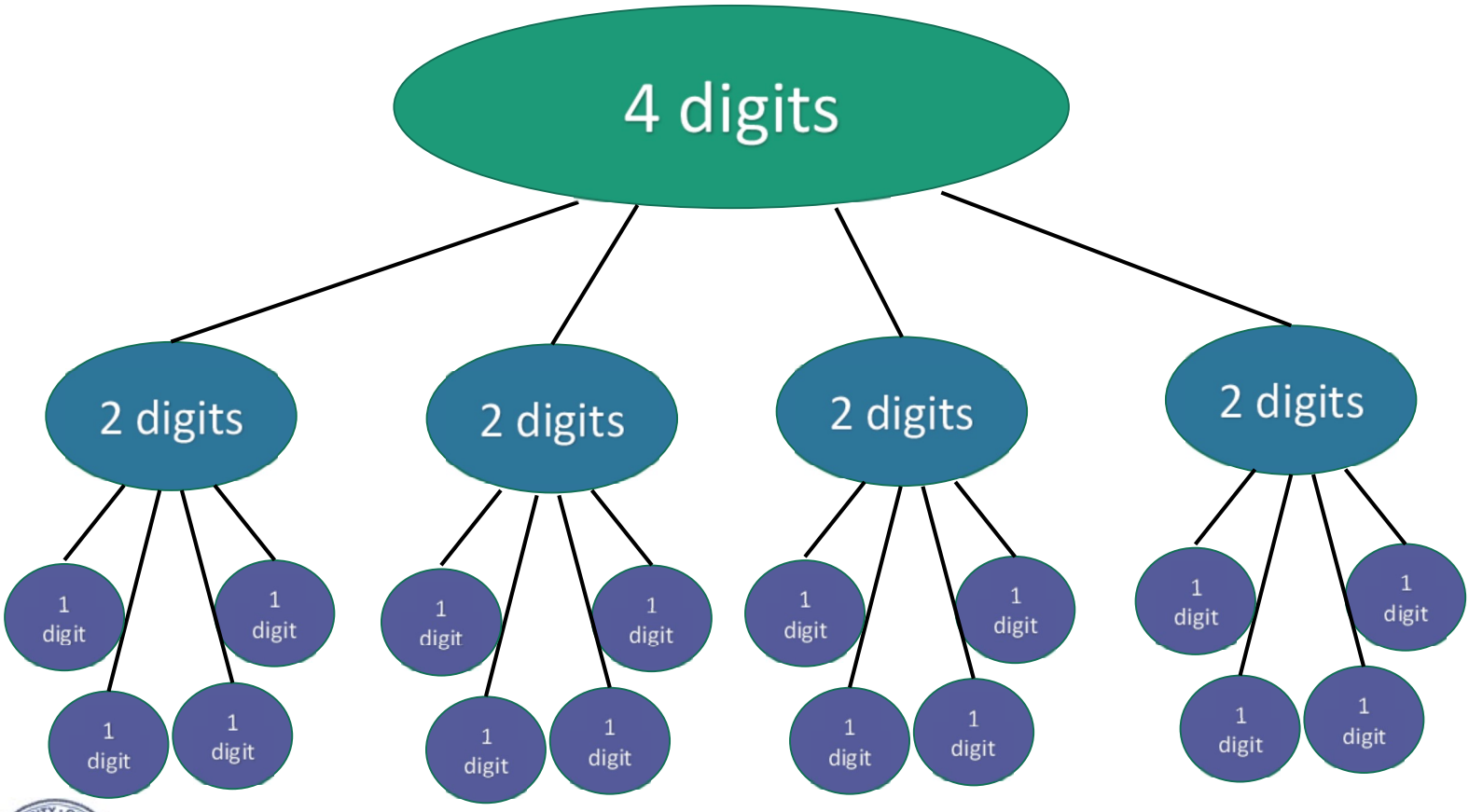
$$1234 \times 5678$$

- If you recurse on those 2-digit multiplication problems, how many 1-digit multiplications do you end up with total?



Recursion Tree

16 one-digit multiplies!

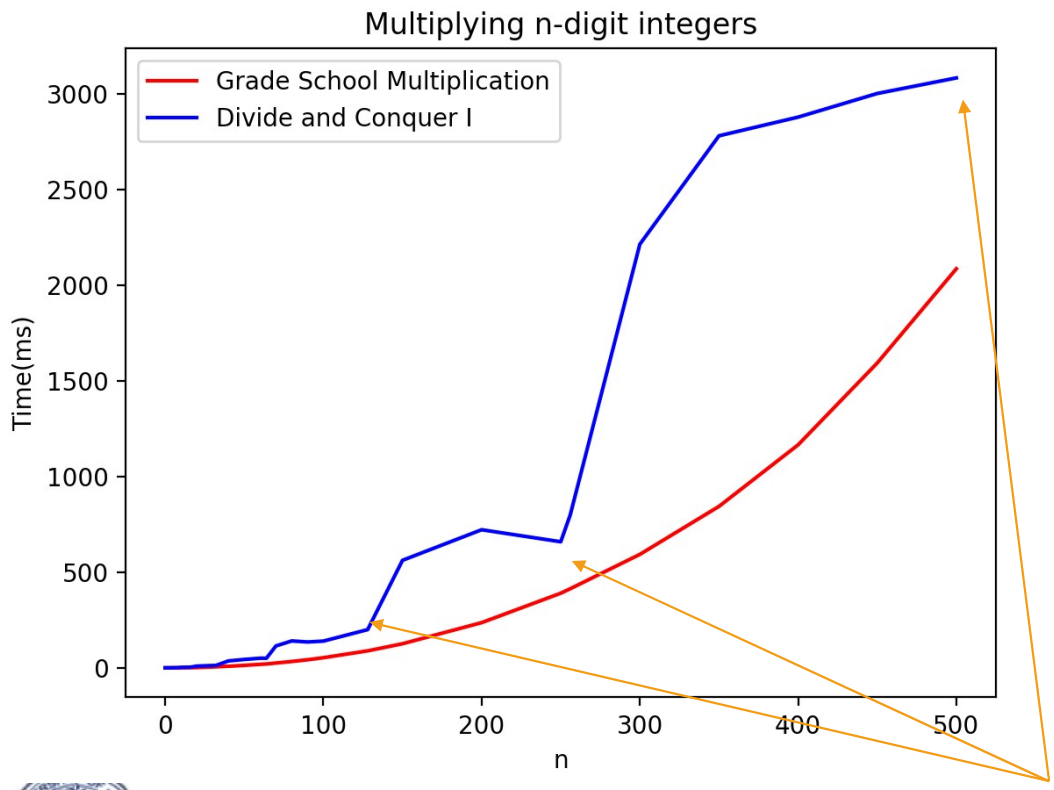


What is the running time?

- Better or worse than the grade school algorithm?

- How do we answer this question?
  1. Try it.
  2. Try to understand it analytically.

1. Try it.



Conjectures about running time?

Doesn't look too good but hard to tell…

Concerns with the conclusiveness of this approach?

Maybe one implementation is slicker than the other?

Maybe if we were to run it to n=10000, things would look different.

Something funny is happening at powers of 2…

2. Try to understand the running time analytically

- Proof by meta-reasoning:

It must be faster than the grade school algorithm, because we are learning it in an algorithms class.

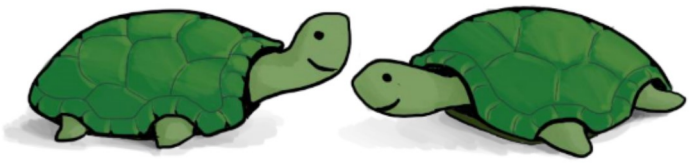
Not sound logic!

Plucky the Pedantic Penguin

2. Try to understand the running time analytically

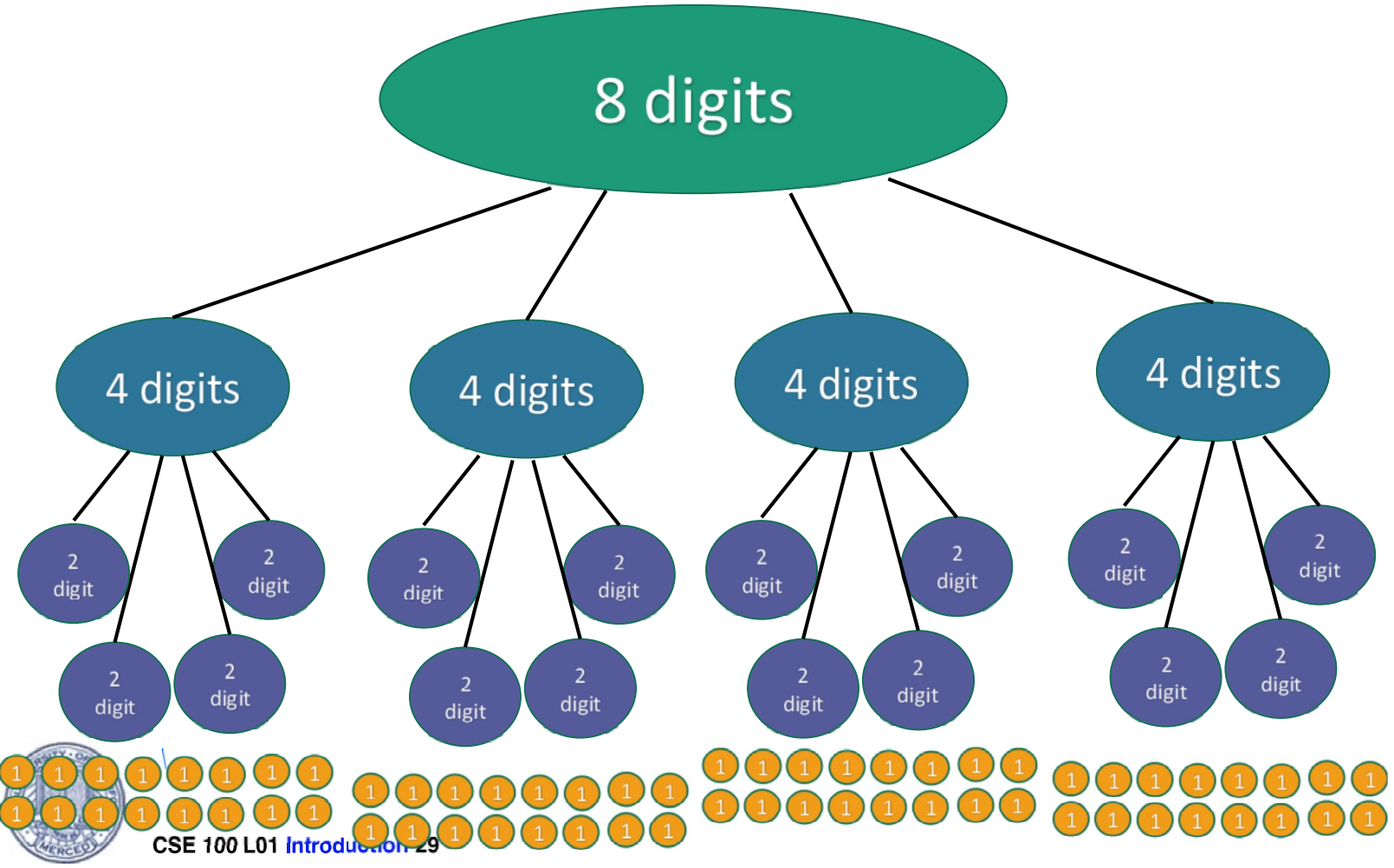
Think-Pair-Share:

- We saw that multiplying 4-digit numbers resulted in 16 one-digit multiplications.
- How about multiplying 8-digit numbers?
- What do you think about n-digit numbers?



Recursion Tree

64 one-digit multiplies!



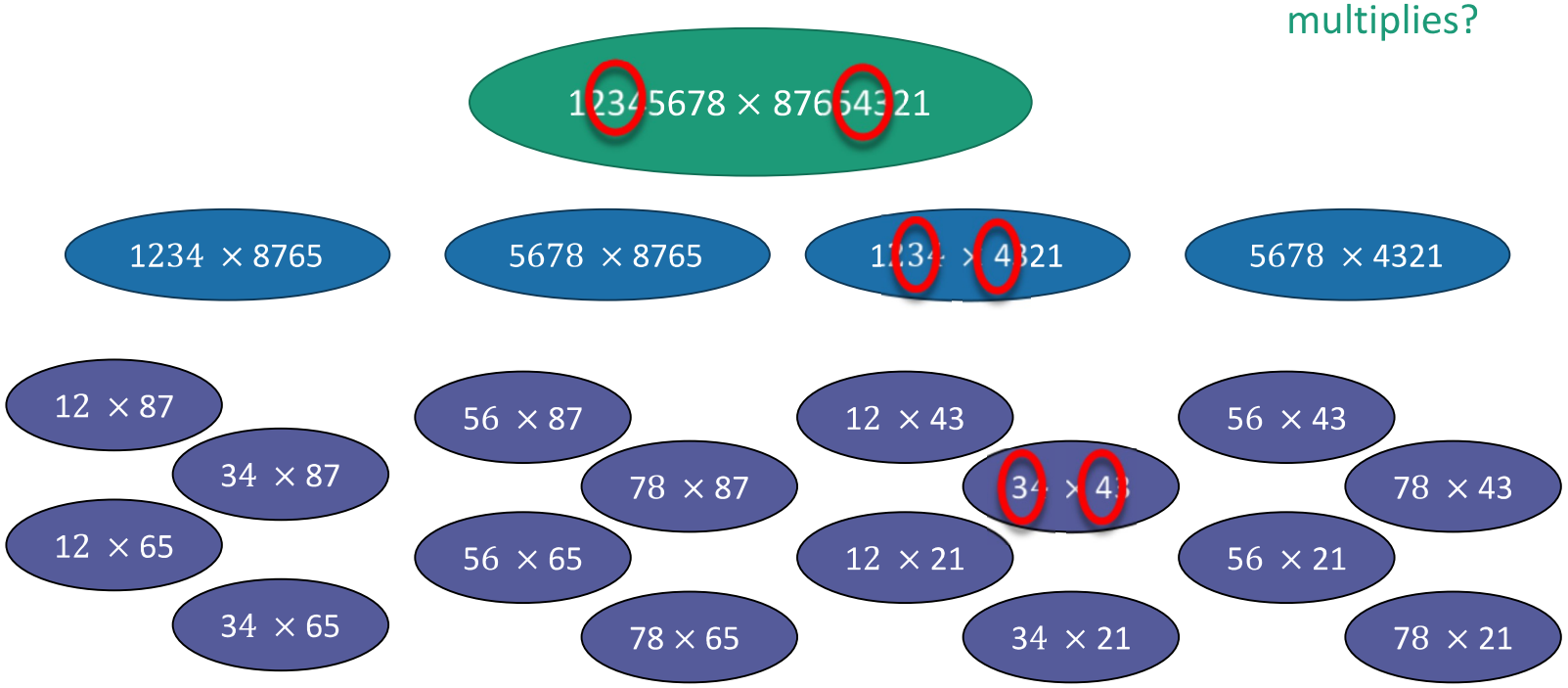
2. Try to understand the running time analytically

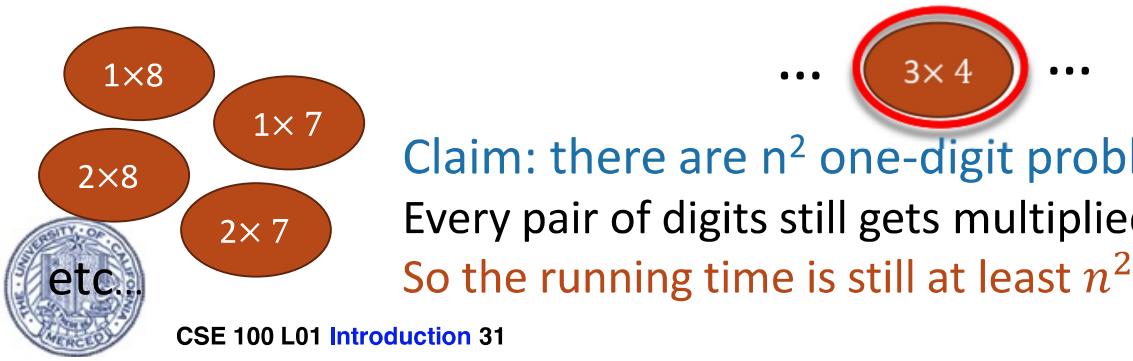
Claim:

The running time of this algorithm is AT LEAST n^2 operations.

Let's do an example

How many one-digit multiplies?





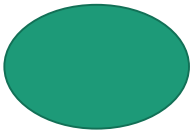
Claim: there are n^2 one-digit problems.

Every pair of digits still gets multiplied together separately.

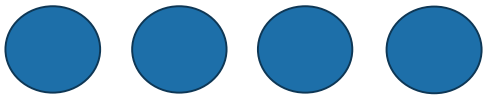
So the running time is still at least n^2.

There are n^2 1-digit problems*

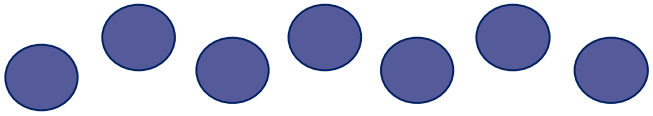
*we will come back to this sort of analysis later and still more rigorously.



1 problem of size n

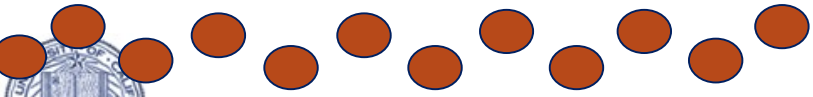


4 problems of size n/2



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

...



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

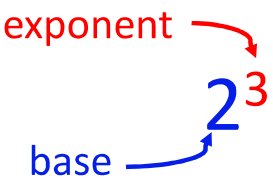
- If you cut n in half $\log_2(n)$ times, you get down to 1.
- So we do this $\log_2(n)$ times and get…

$4^{\log_2 n} = n^2$ problems of size 1.

What about the work you actually do in the problems?

Review of exponents & logarithms (1)

What is an Exponent?

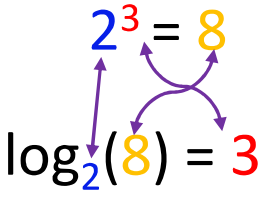


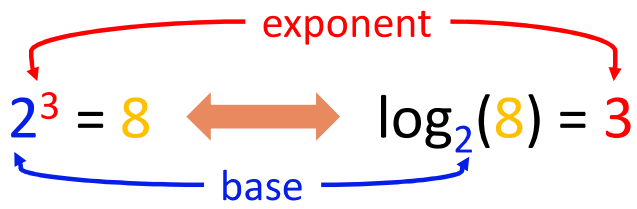
The exponent of a number says how many times to use the number in a multiplication
In this example: $2^3 = 2 \times 2 \times 2 = 8$

What is an Logarithm?

A Logarithm goes the other way. It asks the question "what exponent produced this?":

$2^? = 8$ and answers like this:

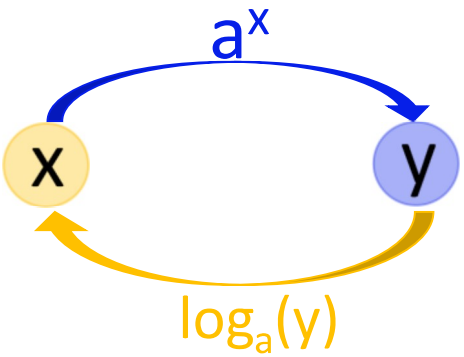




Review of exponents & logarithms (2)

Working Together

Exponents and Logarithms work well together because they "undo" each other (so long as the base "a" is the same):



The Logarithmic Function is "undone" by the Exponential Function.
(and vice versa)

Doing one, then the other, gets you back where you started:

- Doing $\mathbf{a^x}$ then $\mathbf{\log_a}$ gives you **x** back again: $\log_a(a^x) = x$
- Doing $\mathbf{\log_a}$ then $\mathbf{a^x}$ gives you **x** back again: $a^{\log_a(x)} = x$

Going back to the our problem

We have $4^{\log\_2(n)}$ problems of size 1, and we argue that this is n^2 problems of size 1

- $4^{\log\_2(n)} = n^2$
- $4^{\log\_2(n)} = 2^{2*\log\_2(n)} = [2^{\log\_2(n)}]^2$

Using the identity defined previously:

$a^{\log_a(x)} = x$

- $[2^{\log\_2(n)}]^2 = [n]^2 = n^2$ ✓

Yet another way to see this*

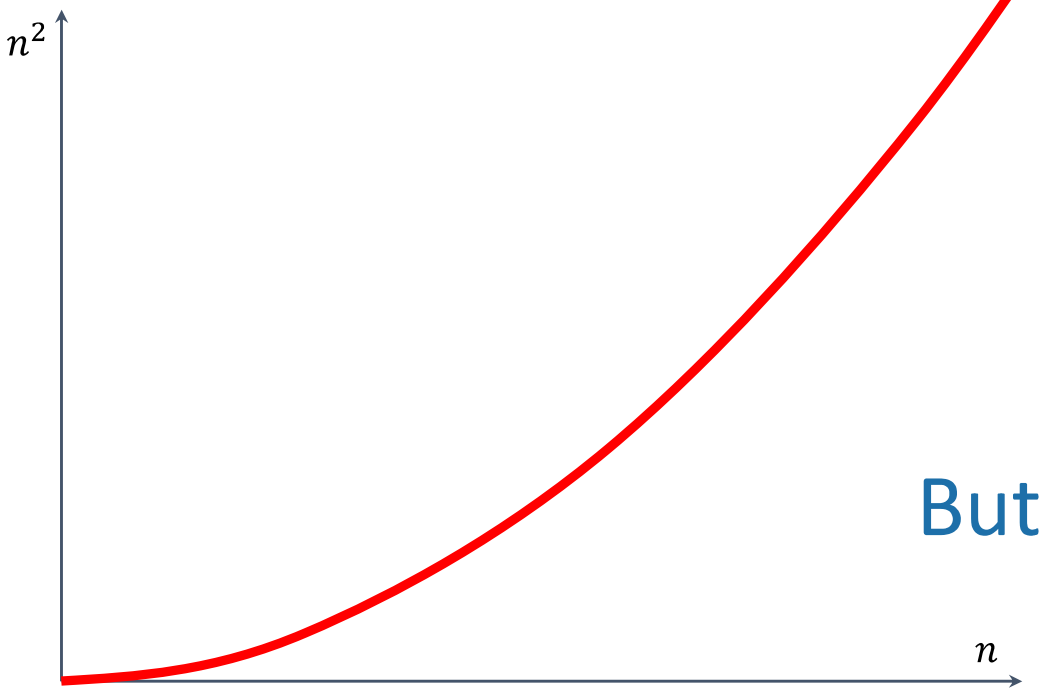
- Let T(n) be the time to multiply two n-digit numbers.
- Recurrence relation:
  - $T(n) = 4 \cdot T\left(\frac{n}{2}\right) +$ (about n to add stuff up)

Ignore this term for now…

$$
\begin{aligned}
T(n) &= 4 \cdot T(n/2) \\
&= 4 \cdot (4 \cdot T(n/4)) && \cdots\cdots 4^2 \cdot T(n/2^2) \\
&= 4 \cdot (4 \cdot (4 \cdot T(n/8))) && \cdots\cdots 4^3 \cdot T(n/2^3) \\
&\vdots \\
&= 2^{2t} \cdot T(n/2^t) && \cdots\cdots 4^t \cdot T(n/2^t) \\
&\vdots \\
&= n^2 \cdot T(1). && \cdots\cdots 4^{\log_2(n)} \cdot T(n/2^{\log_2(n)})
\end{aligned}
$$

That's a bit disappointing

All that work and still (at least) $O(n^2)$…



But wait!!

Divide and conquer can actually make progress

- Karatsuba figured out how to do this better!

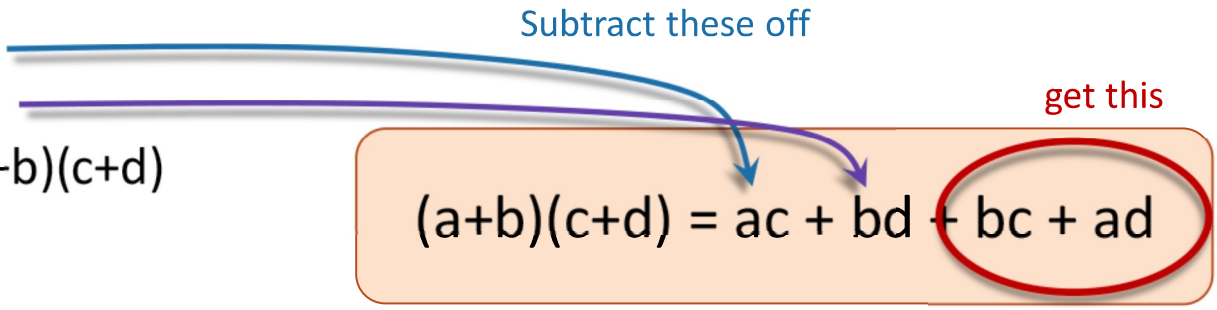
$$xy = (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d)$$
$$= ac \cdot 10^{n} + (ad + bc)10^{n/2} + bd$$

Need these three things

- If only we recurse three times instead of four…

Karatsuba integer multiplication

- Recursively compute these THREE things:
  - ac
  - bd
  - (a+b)(c+d)



- Assemble the product:

$$xy = (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d)$$
$$= ac \cdot 10^{n} + (ad + bc)10^{n/2} + bd$$

✓ ✓ ✓

How would this work?

x,y are n-digit numbers

(Still not super precise. Also, still assume n is a power of 2.)

Multiply(x, y):

- **If** n=1:
  - **Return** xy
- Write $x = a\ 10^{\frac{n}{2}} + b$ and $y = c\ 10^{\frac{n}{2}} + d$

a, b, c, d are n/2-digit numbers

- ac = **Multiply**(a, c)
- bd = **Multiply**(b, d)
- z = **Multiply**(a+b, c+d)

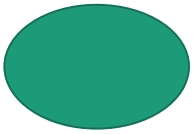
We can do the addition a+b and c+d in time O(n). This results in integers that are still roughly n/2 bits long.

- cross_terms = z - ac - bd

The quantity cross_terms is meant to be (ad + bc)

- xy = ac 10^n + (cross_terms) $10^{n/2}$ + bd
- Return xy

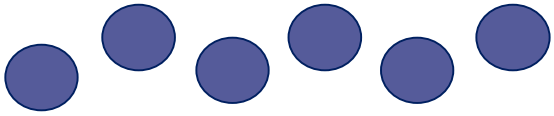
What's the running time?



1 problem of size n

3 problems of size n/2

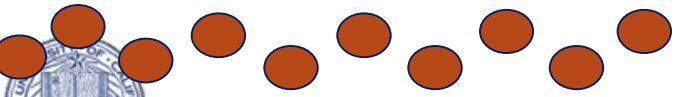
...



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

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

...



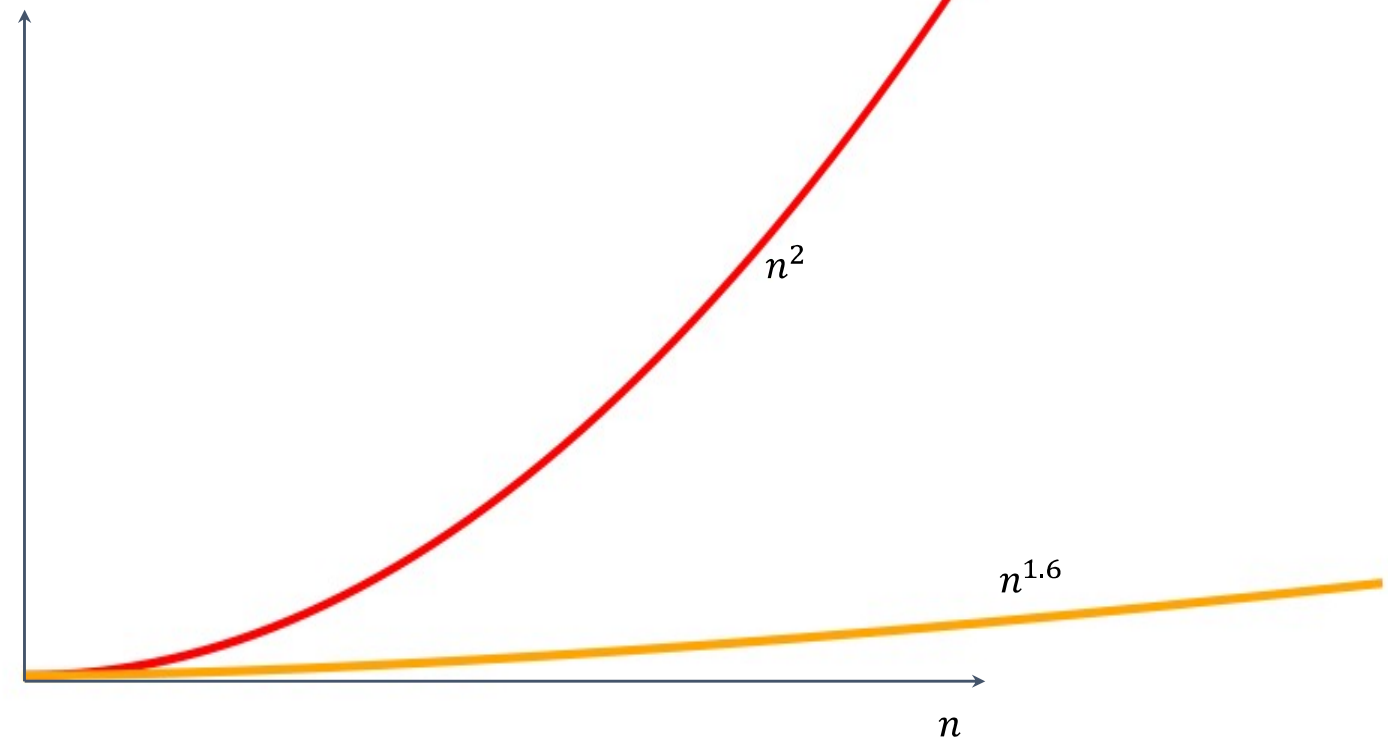
$\frac{n^{1.6}}{\_\_\_}$ problems of size 1

- 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.

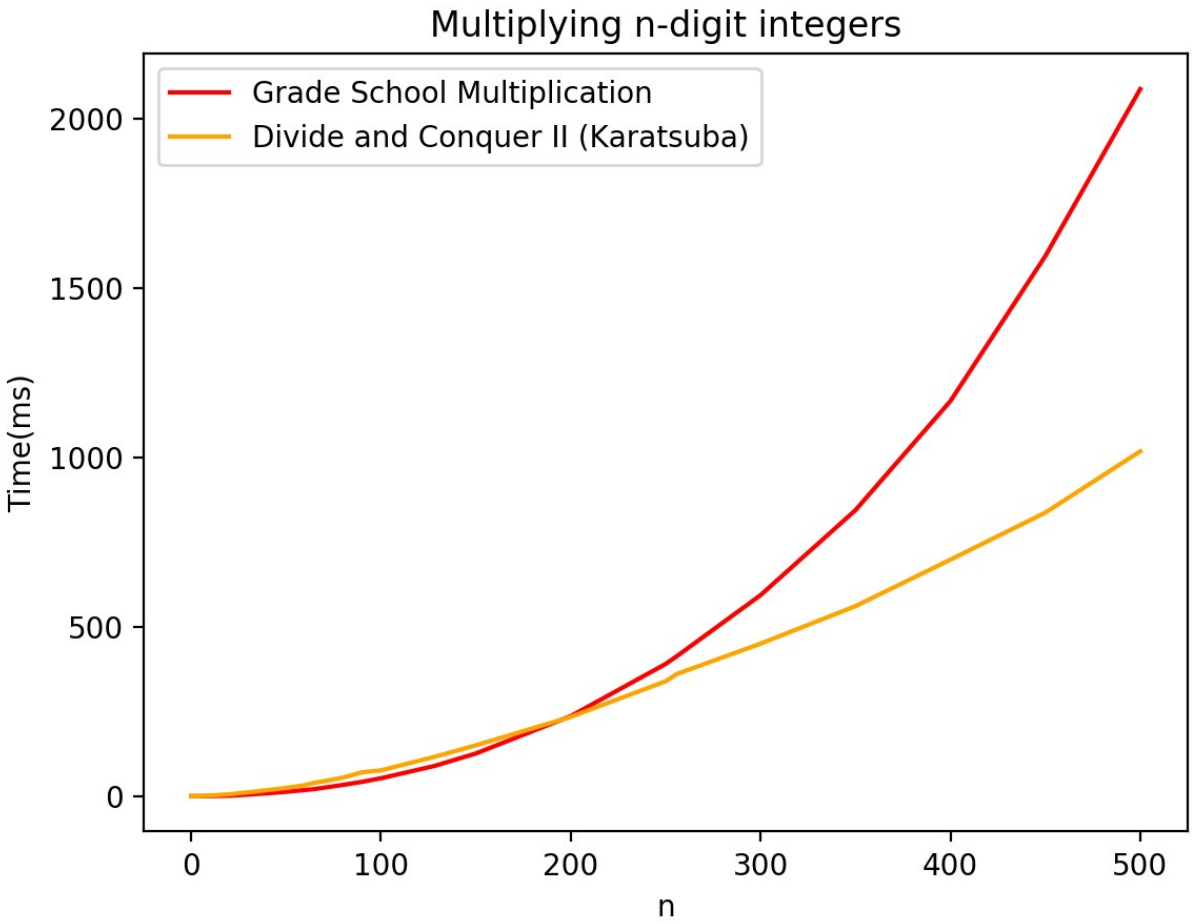
We aren't accounting for the work at the higher levels! But we'll see later that this turns out to be okay.

This is much better!



We can even see it in real life!

python



Can we do better?

- **Toom-Cook** (1963): instead of breaking into three n/2-sized problems, break into five n/3-sized problems.
  - Runs in time $O(n^{1.465})$

Try to figure out how to break up an n-sized problem into five n/3-sized problems! **(Hint: start with nine n/3-sized problems).**

Given that you can break an n-sized problem into five n/3-sized problems, where does the 1.465 come from?

Ollie the Over-achieving Ostrich

Siggi the Studious Stork

- **Schönhage–Strassen** (1971):
  - Runs in time $O(n \log(n) \log\log(n))$
- **Furer** (2007)
  - Runs in time $n \log(n) \cdot 2^{O(\log^*(n))}$

[This is just for fun, you don't need to know these algorithms!]

Course goals

- Think analytically about algorithms
- Flesh out an "algorithmic toolkit"
- Learn to communicate clearly about algorithms

Today's goals

- Karatsuba Integer Multiplication
- Algorithmic Technique:
  - Divide and conquer
- Algorithmic Analysis tool:
  - Intro to asymptotic analysis

Wrap up

- Karatsuba Integer Multiplication:
  - You can do better than grade school multiplication!
  - Example of divide-and-conquer in action
  - Informal demonstration of asymptotic analysis

Next Time

- Sorting!
- Divide and Conquer some more
- Asymptotics and (formal) Big-Oh notation

BEFORE Next Lecture

- ***Lab Assignments 00 and 01*** on the course website!

CSE100: Design and Analysis of Algorithms
Lecture 03 – Sorting

Jan 26th 2021

InsertionSort, Divide-and-conquer, MergeSort

Last Week

Philosophy

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

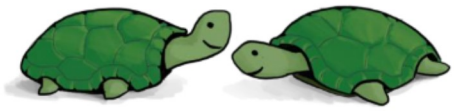
Technical content

- 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

Today

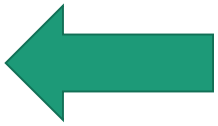
- Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?
- Return of divide-and-conquer with Merge Sort
- Skills:
  - Analyzing correctness of iterative and recursive algorithms.
  - Analyzing running time of recursive algorithms.

Next Time:

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

Today

- Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?



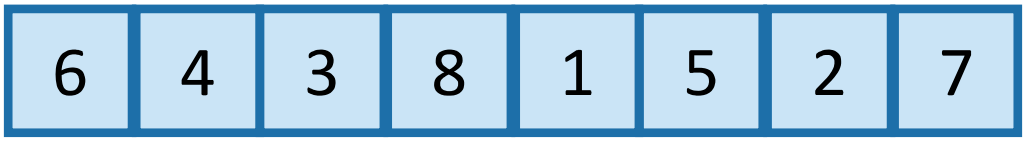
- Return of divide-and-conquer with Merge Sort
- Skills:
  - Analyzing correctness of iterative and recursive algorithms.
  - Analyzing running time of recursive algorithms.

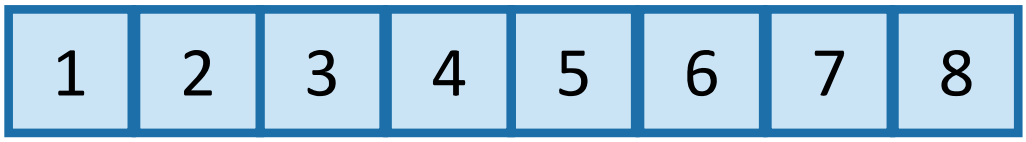
Next Time:

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

Sorting

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

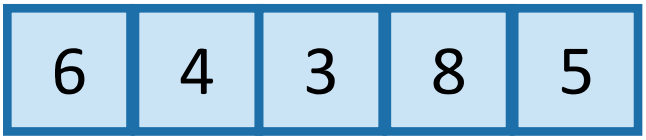




Benchmark: insertion sort

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

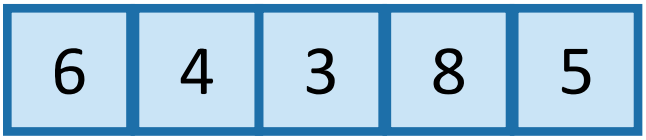
- Say we want to sort:



Benchmark: insertion sort

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

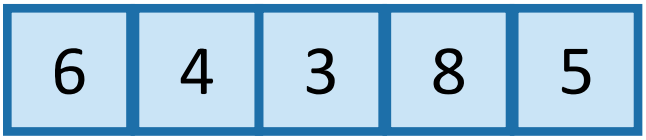
- Say we want to sort:
- How would you do it?



Benchmark: insertion sort

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

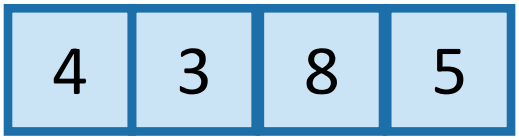
- Say we want to sort:
- How would you do it?
- Insert items one at a time.



Benchmark: insertion sort

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

- Say we want to sort:
- How would you do it?
- Insert items one at a time.



Benchmark: insertion sort

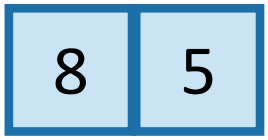
We're going to go through this in some detail – it's good practice!

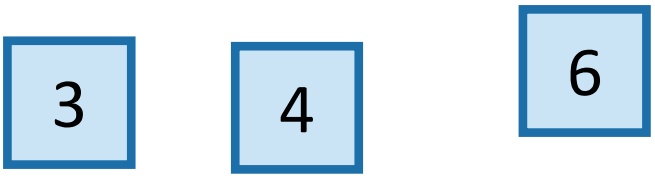
- Say we want to sort:
- How would you do it?
- Insert items one at a time.

Benchmark: insertion sort

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

- Say we want to sort:
- How would you do it?
- Insert items one at a time.





Benchmark: insertion sort

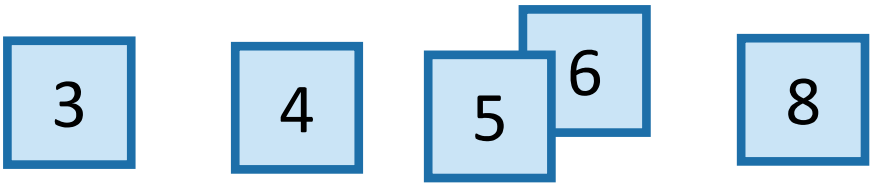
We're going to go through this in some detail – it's good practice!

- Say we want to sort:
- How would you do it?
- Insert items one at a time.

Benchmark: insertion sort

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

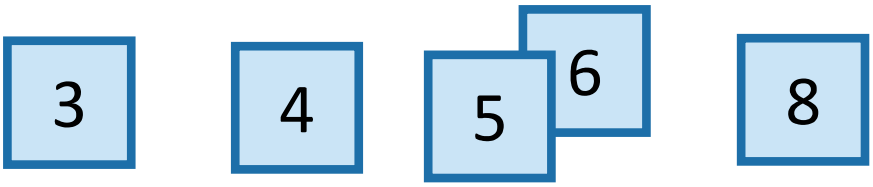
- Say we want to sort:
- How would you do it?
- Insert items one at a time.



Benchmark: insertion sort

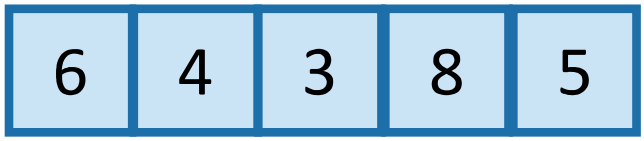
We're going to go through this in some detail – it's good practice!

- Say we want to sort:
- How would you do it?
- Insert items one at a time.

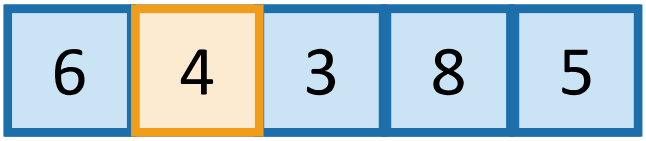


- How would we actually implement this?

Insertion sort example…

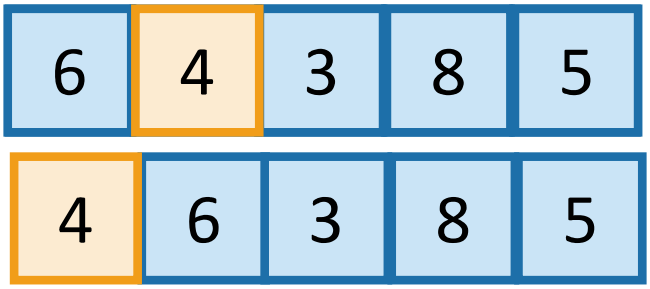


Insertion sort example…



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

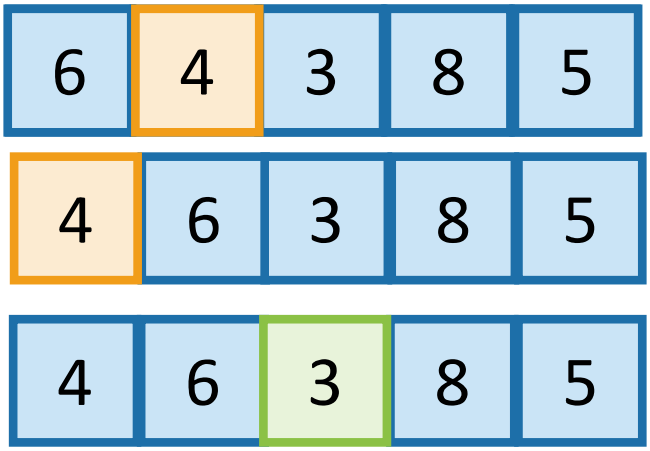
Insertion sort example…



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

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

Insertion sort example…

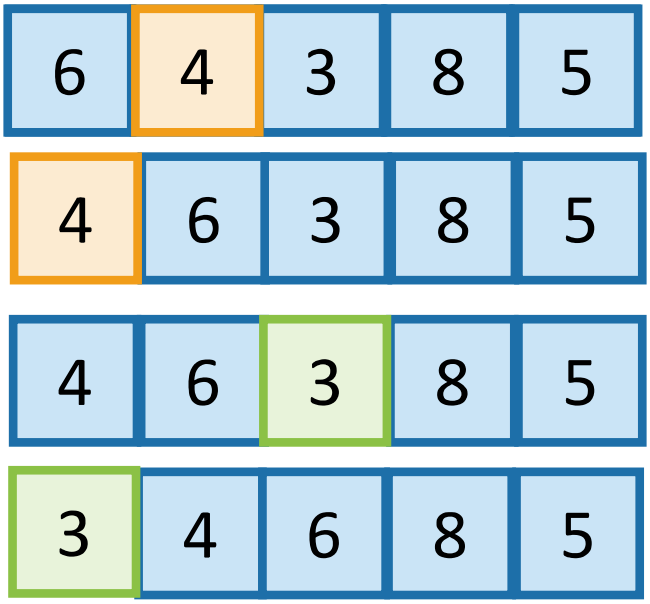


Start with the second element (the first element is sorted within itself…)

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

Now look at “3”

Insertion sort example…



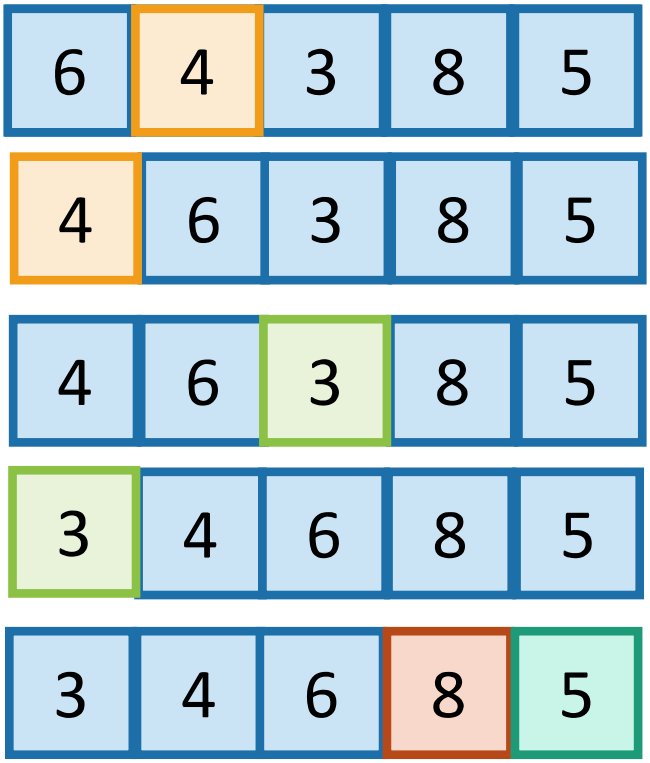
Start with the second element (the first element is sorted within itself…)

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

Now look at “3”

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

Insertion sort example…



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

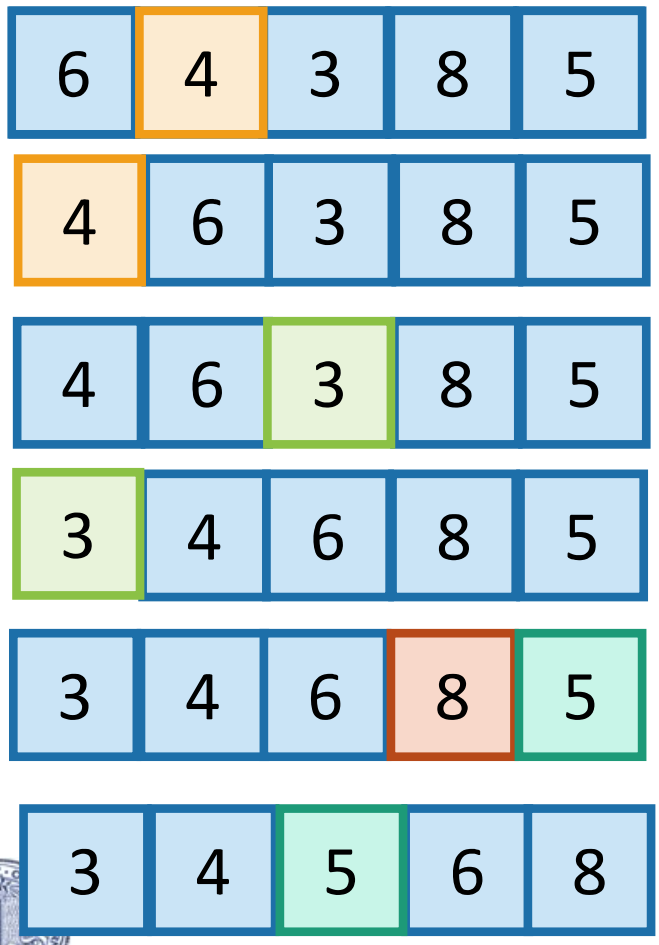
Pull “4” back until it’s in the right place.

Now look at “3”

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

“8” is good…look at 5

Insertion sort example…



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

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

Now look at “3”

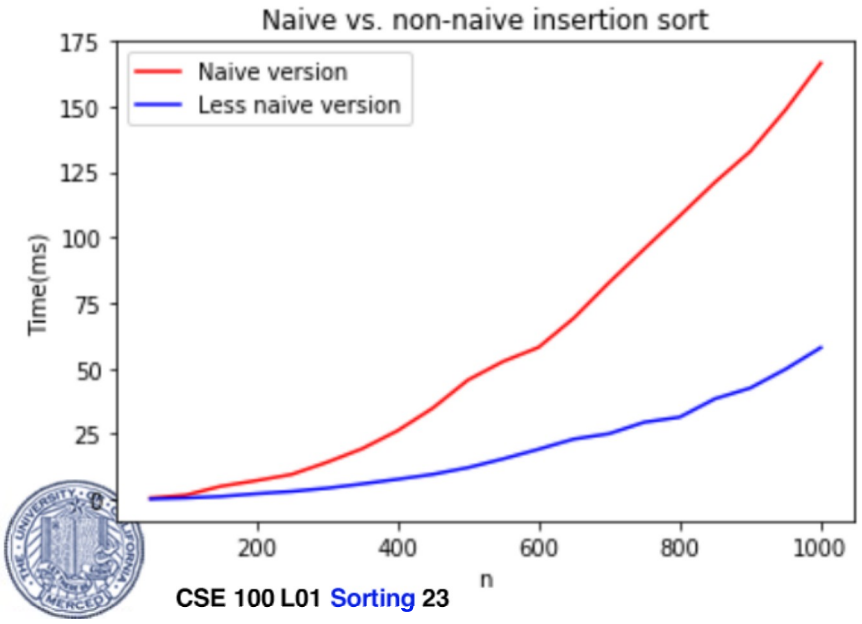
Pull “3” back until it’s in the right place.

“8” is good…look at 5

(then fix 5 and we’re done)

Insertion Sort

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



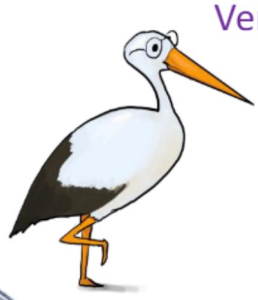
- The “same” algorithm can be faster or slower depending on the implementation…
- We are interested in how fast the running time scales with n, the size of the input.

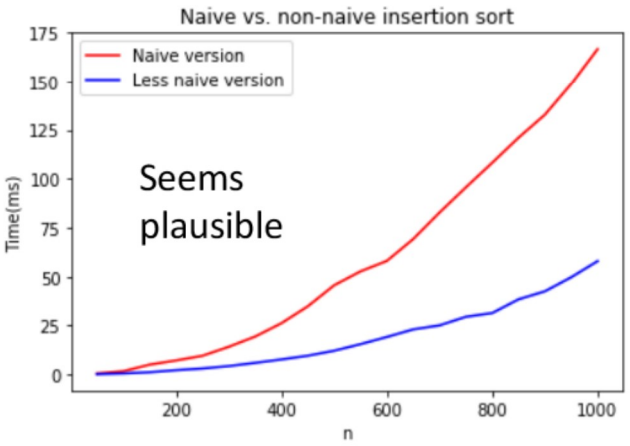
Insertion Sort: running time

Technically we haven't defined this yet…we'll do it later.

- Claim: The running time is $O(n^2)$
- I don't want to focus on this in lecture, but there's a hidden slide to help you verify this later. (Or see CLRS).

Verify this!





Insertion sort pseudocode

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

Lucky the lackadaisical lemur

Algorithm 1: INSERTIONSORT(A)

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

Plucky the pedantic penguin

- (Discussion on board)

Insertion sort: running time

Algorithm 1: INSERTIONSORT(A)

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

n-1 iterations of the outer loop

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

Running time is $O(n^2)$

Insertion Sort

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

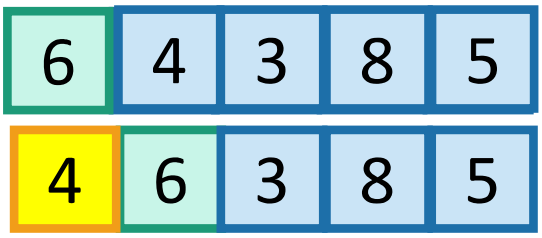
- Okay, so it's pretty obvious that it works.

- 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?

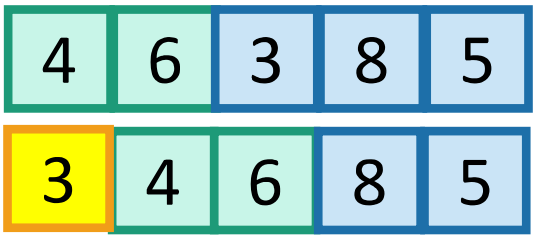
- Say you have a sorted list, 3 4 6 8, and another element 5.
- Insert 5 right after the largest thing that's still smaller than 5. (Aka, right after 4).
- Then you get a sorted list: 3 4 5 6 8

So just use this logic at every step.



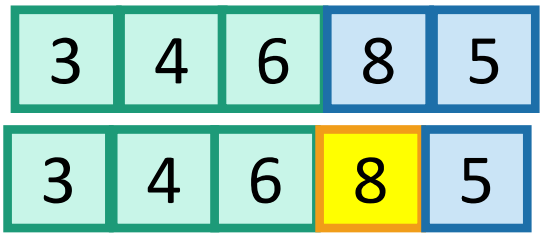
The first element, [6], makes up a sorted list.

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



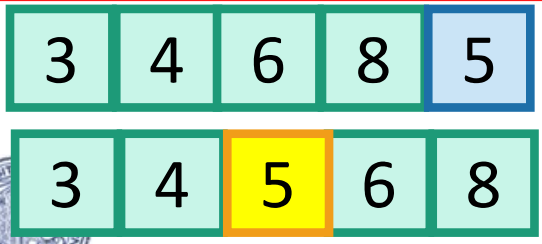
The first two elements, [4,6], make up a sorted list.

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!

Recall: proof by induction

- Maintain a loop invariant.
- Proceed by induction.

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

- **Four steps in the proof by induction:**
  - Inductive Hypothesis: The loop invariant holds after the i^{th} iteration.
  - Base case: the loop invariant holds before the 1^{st} iteration.
  - Inductive step: If the loop invariant holds after the i^{th} iteration, then it holds after the $(i+1)^{st}$ iteration
  - Conclusion: If the loop invariant holds after the last iteration, then we win.

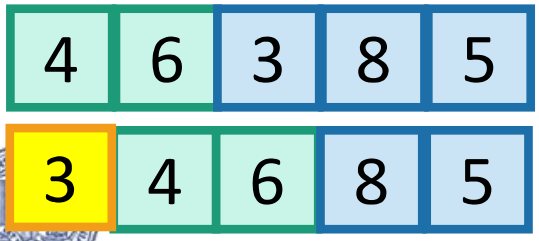
Formally: induction

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 handout for details)



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.

This was iteration i=2.

Aside: proofs by induction

- We're gonna see/do/skip over a lot of them.
- If that went by too fast and was confusing:
  - Slides [there's a hidden one with more info about induction]
  - Notes
  - Book
  - 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

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?

Today

- Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?
- Return of divide-and-conquer with Merge Sort
- Skills:
  - Analyzing correctness of iterative and recursive algorithms.
  - Analyzing running time of recursive algorithms.

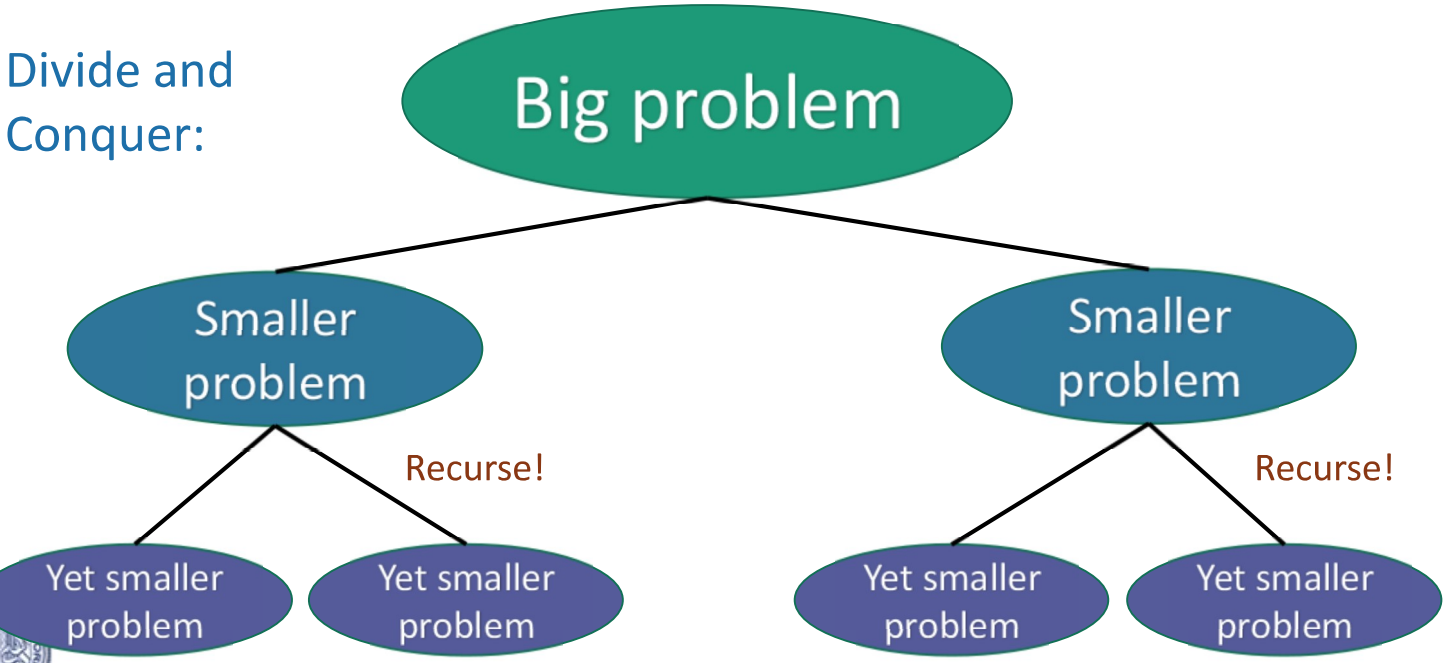
Next Time:

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

Can we do better?

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

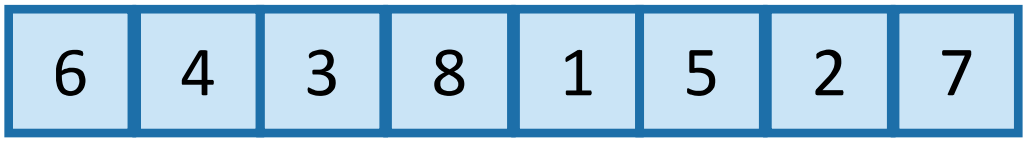
Divide and Conquer:

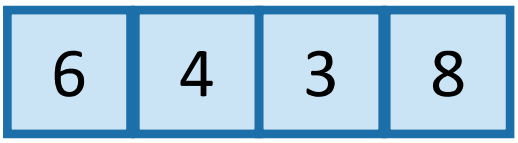


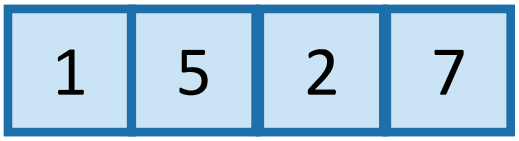
MergeSort

6	4	3	8	1	5	2	7

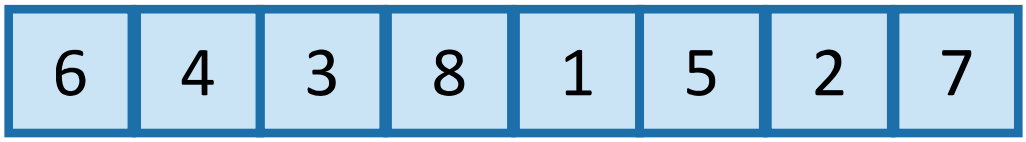
MergeSort

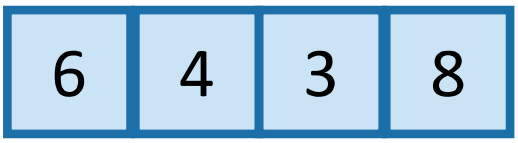




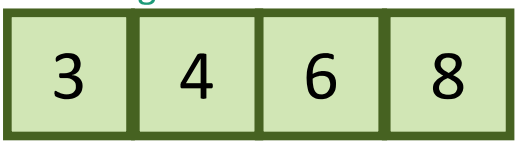


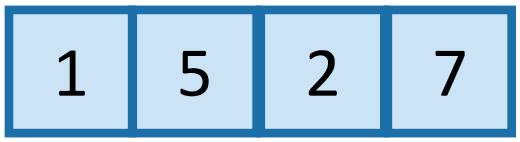
MergeSort



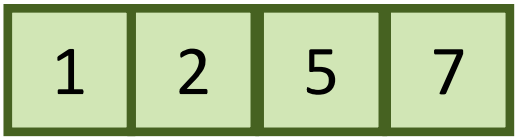


Recursive magic!

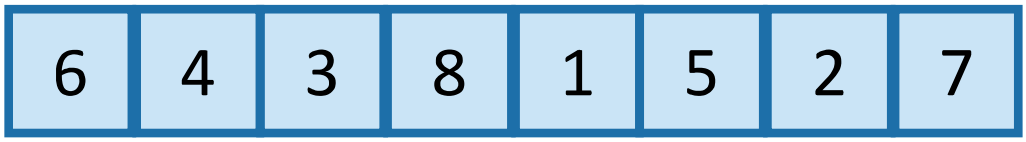


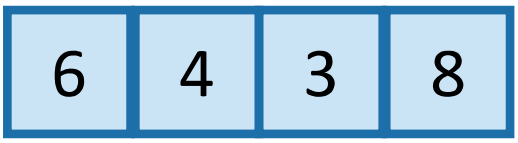


Recursive magic!

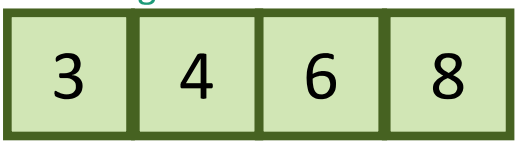


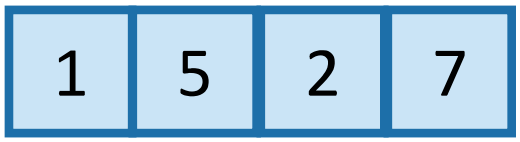
MergeSort



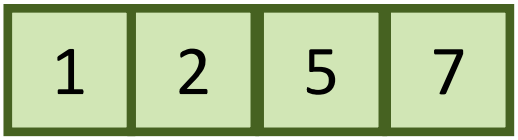


Recursive magic!



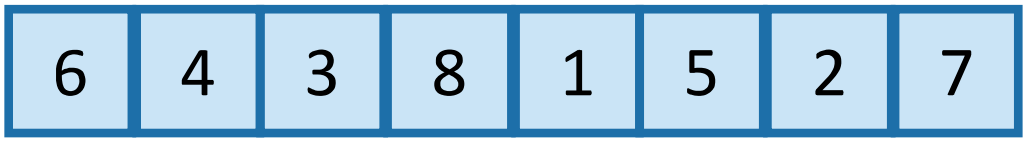


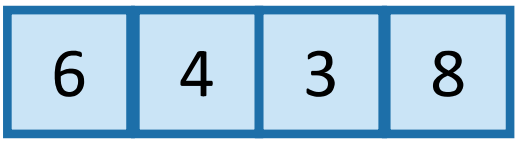
Recursive magic!



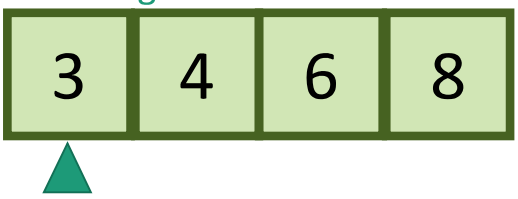
MERGE!

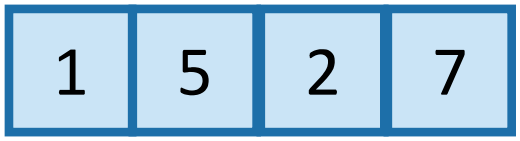
MergeSort



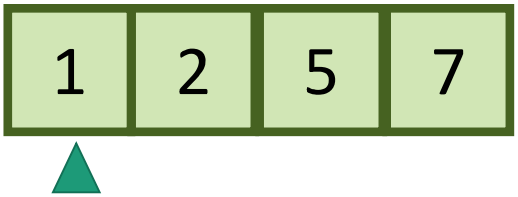


Recursive magic!



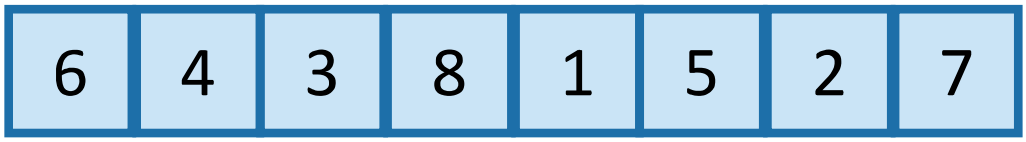


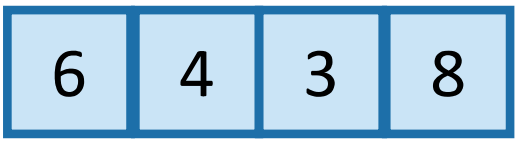
Recursive magic!



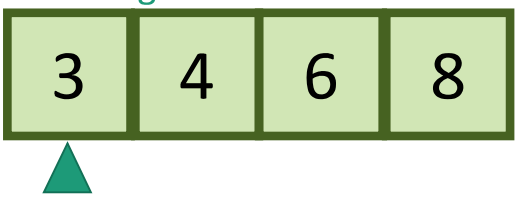
MERGE!

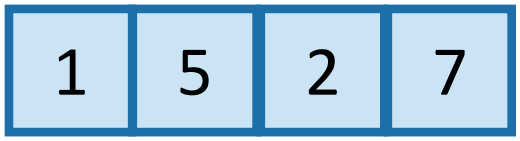
MergeSort



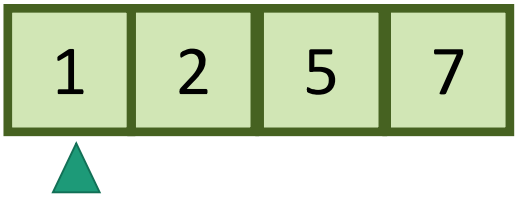


Recursive magic!



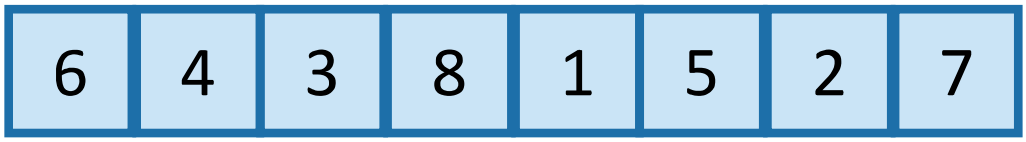


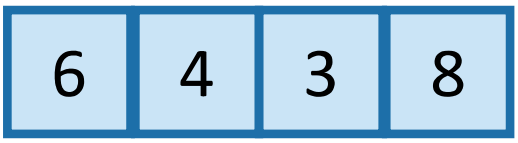
Recursive magic!



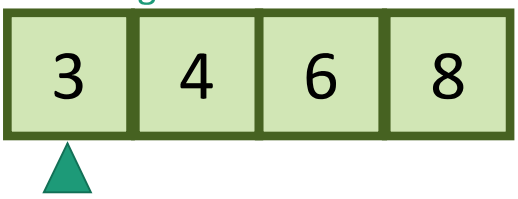
MERGE!

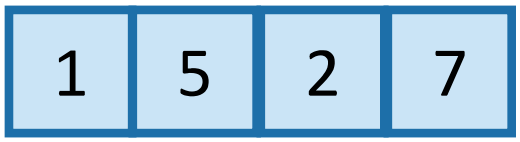
MergeSort



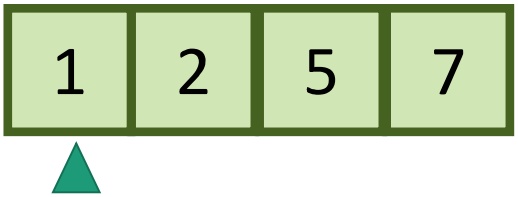


Recursive magic!



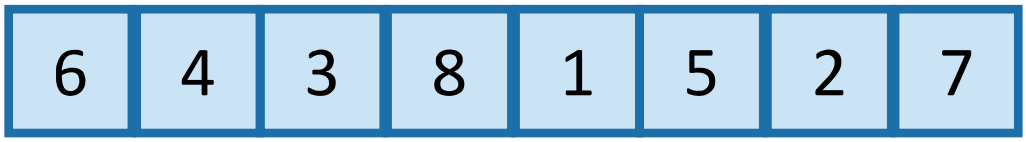


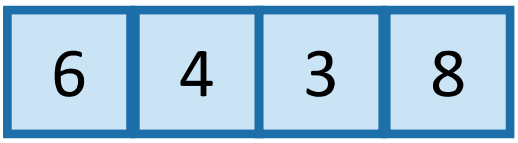
Recursive magic!



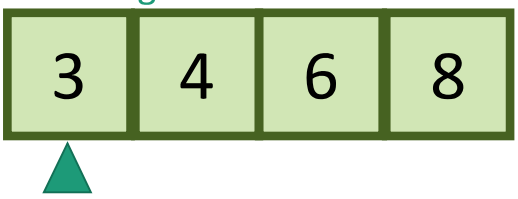
MERGE!

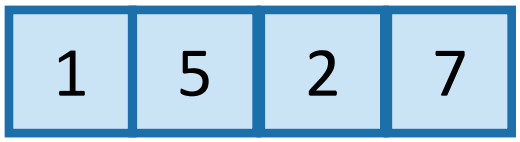
MergeSort



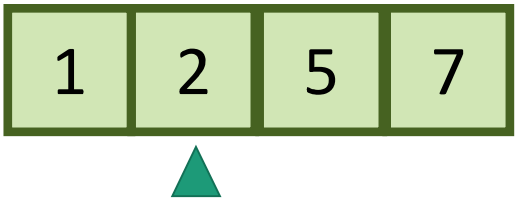


Recursive magic!



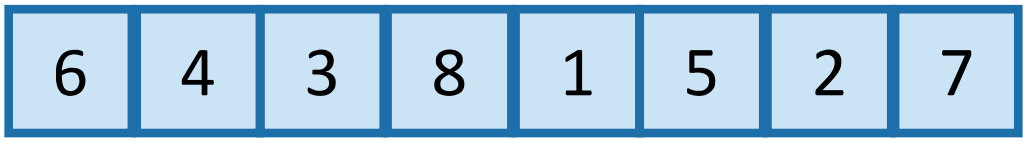


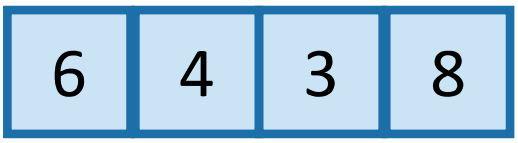
Recursive magic!



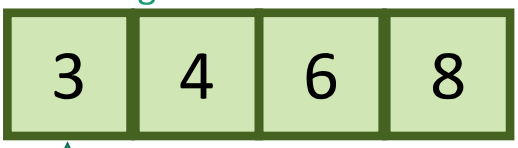
MERGE!

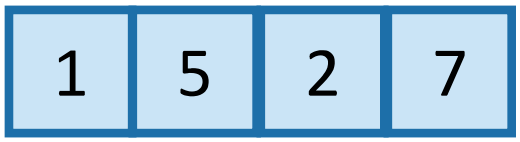
MergeSort



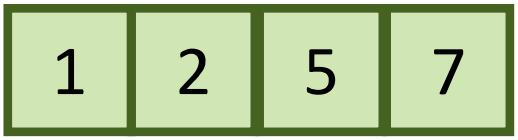


Recursive magic!



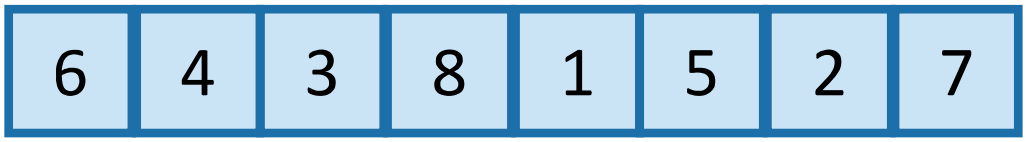


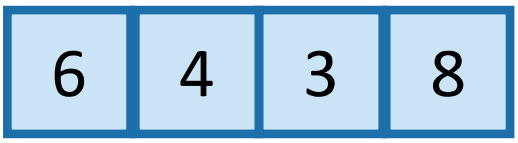
Recursive magic!



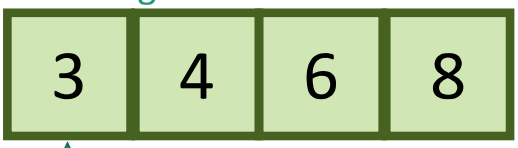
MERGE!

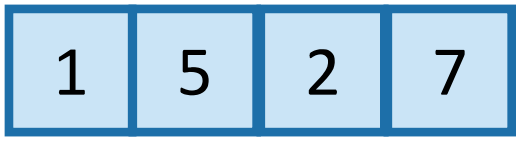
MergeSort



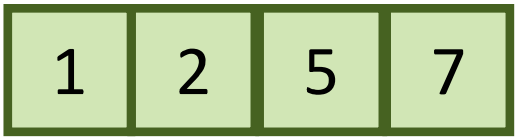


Recursive magic!



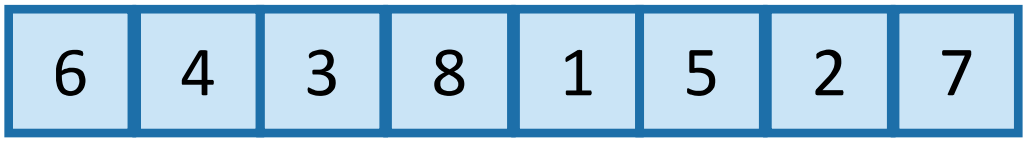


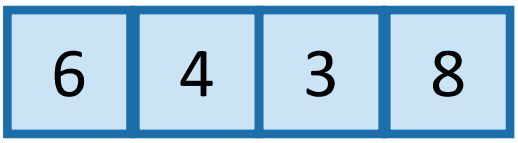
Recursive magic!



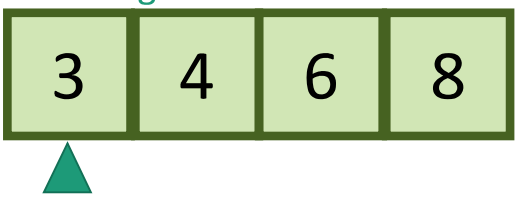
MERGE!

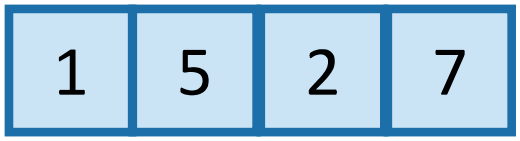
MergeSort



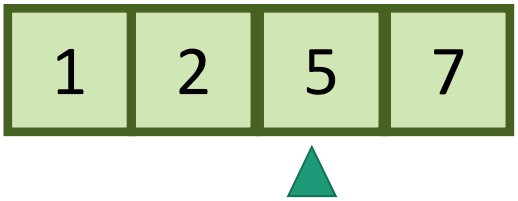


Recursive magic!



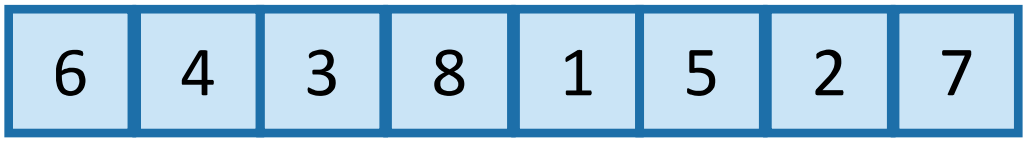


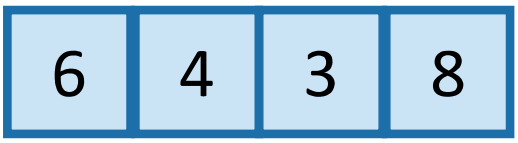
Recursive magic!



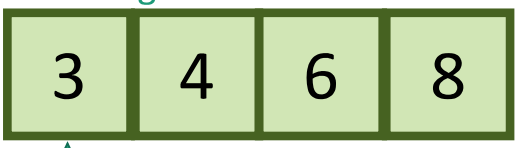
MERGE!

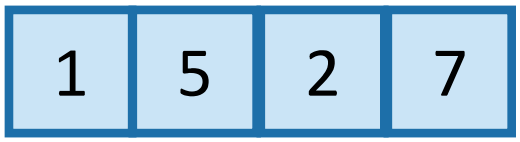
MergeSort



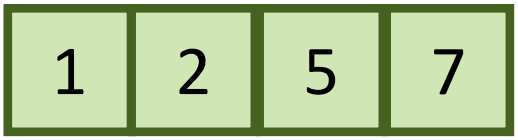


Recursive magic!



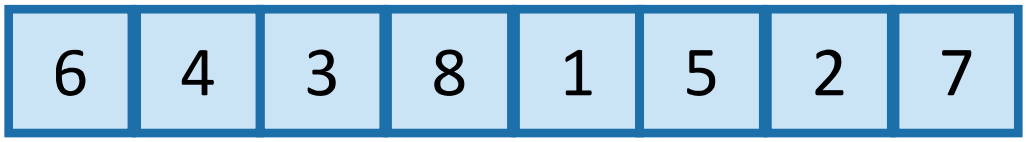


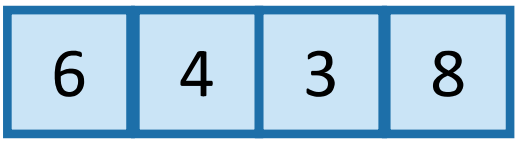
Recursive magic!



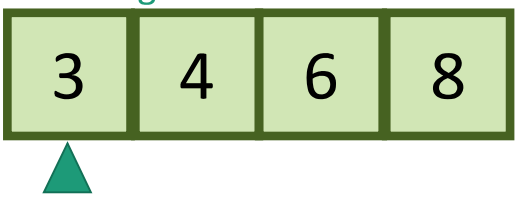
MERGE!

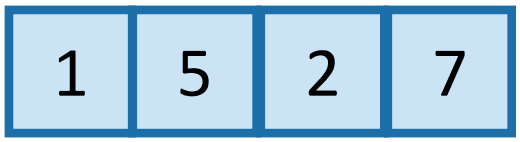
MergeSort



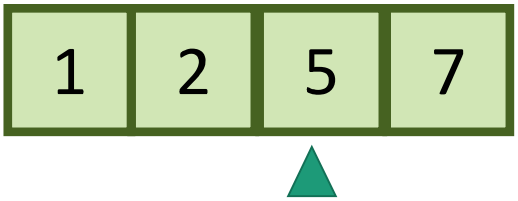


Recursive magic!



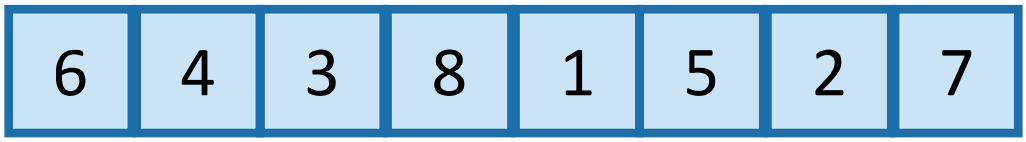


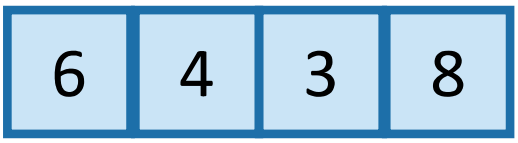
Recursive magic!



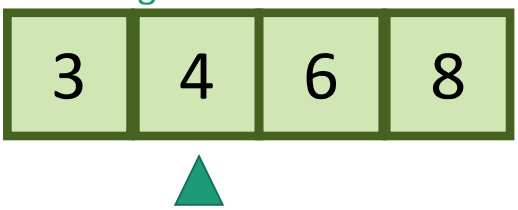
MERGE!

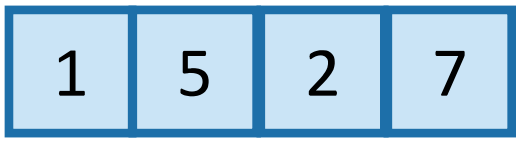
MergeSort



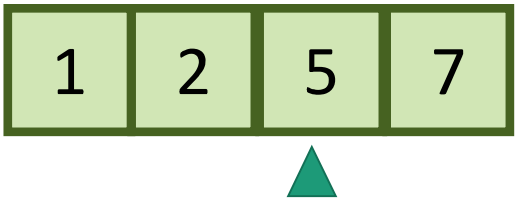


Recursive magic!



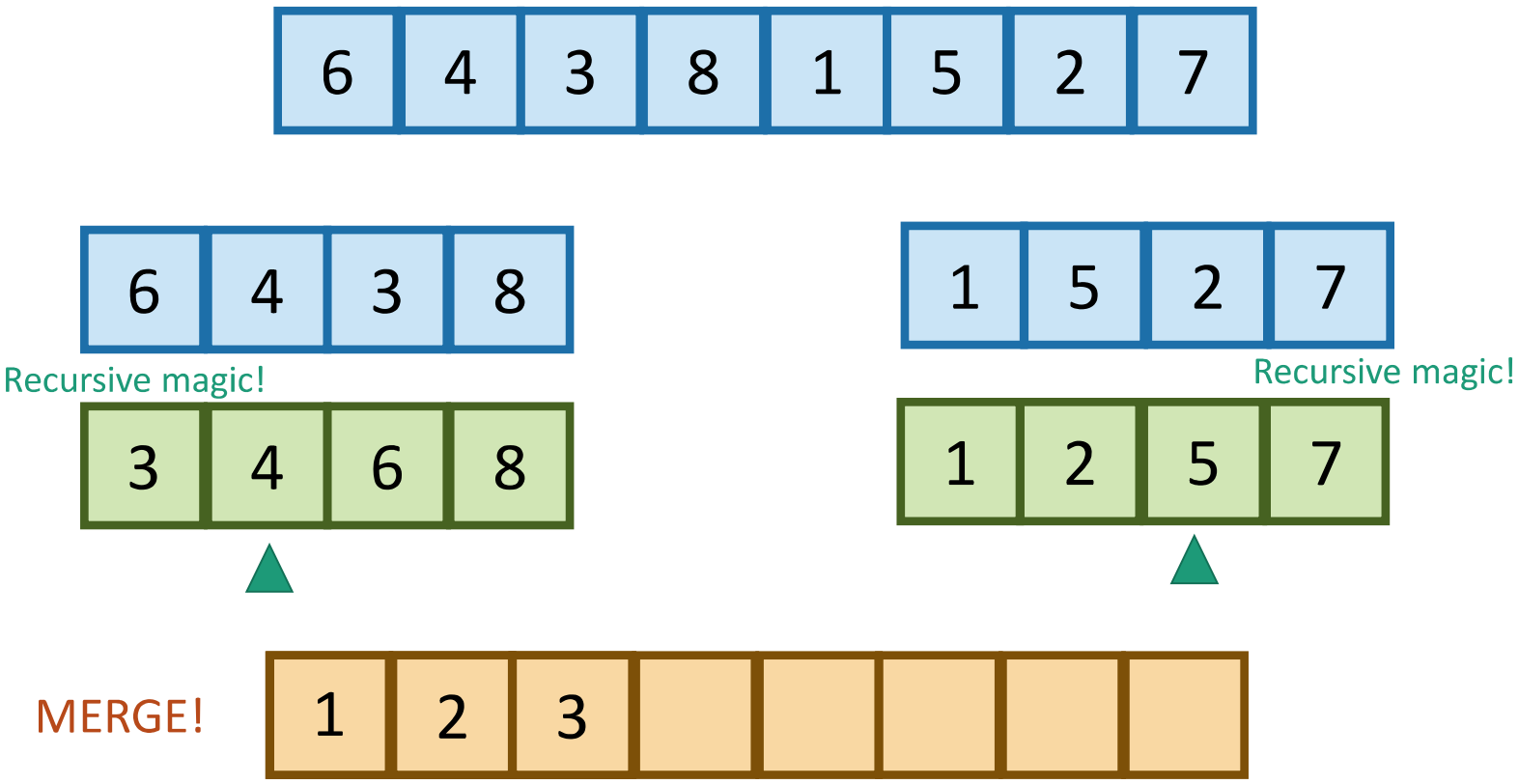


Recursive magic!

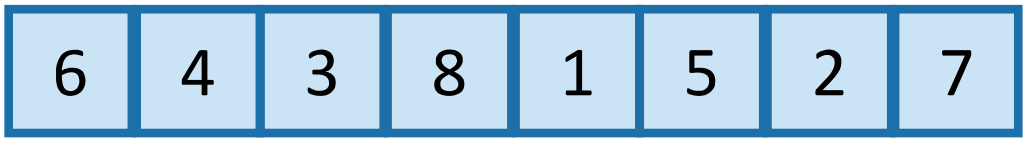


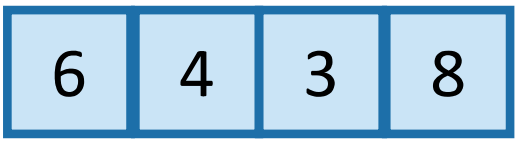
MERGE!

MergeSort

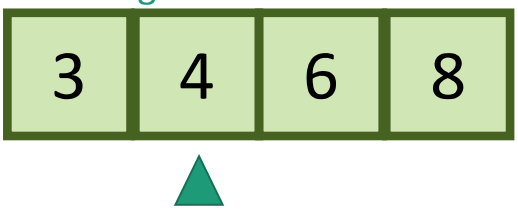

6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3

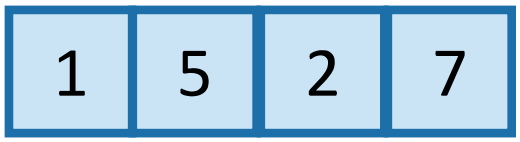
MergeSort



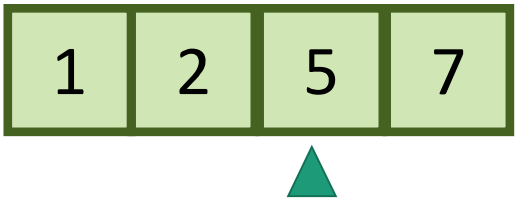


Recursive magic!



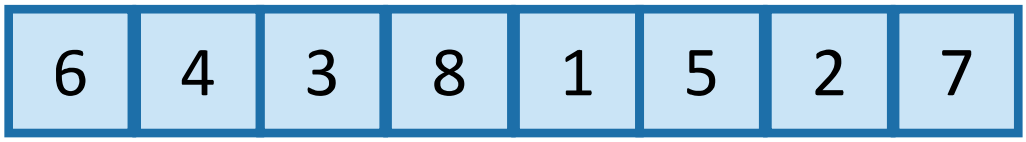


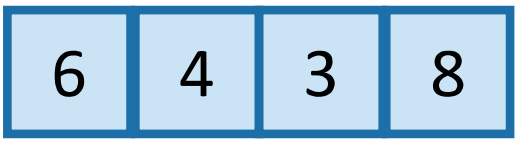
Recursive magic!



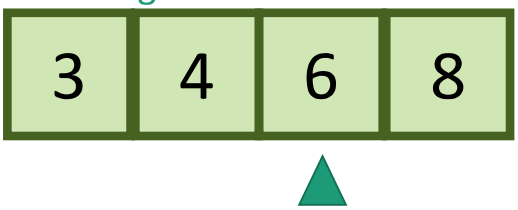
MERGE!

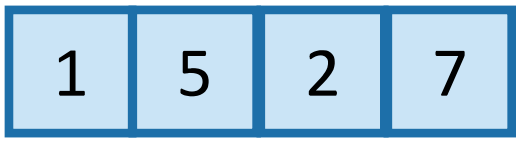
MergeSort



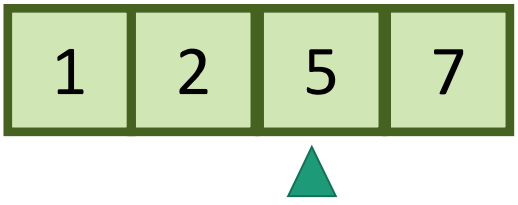


Recursive magic!



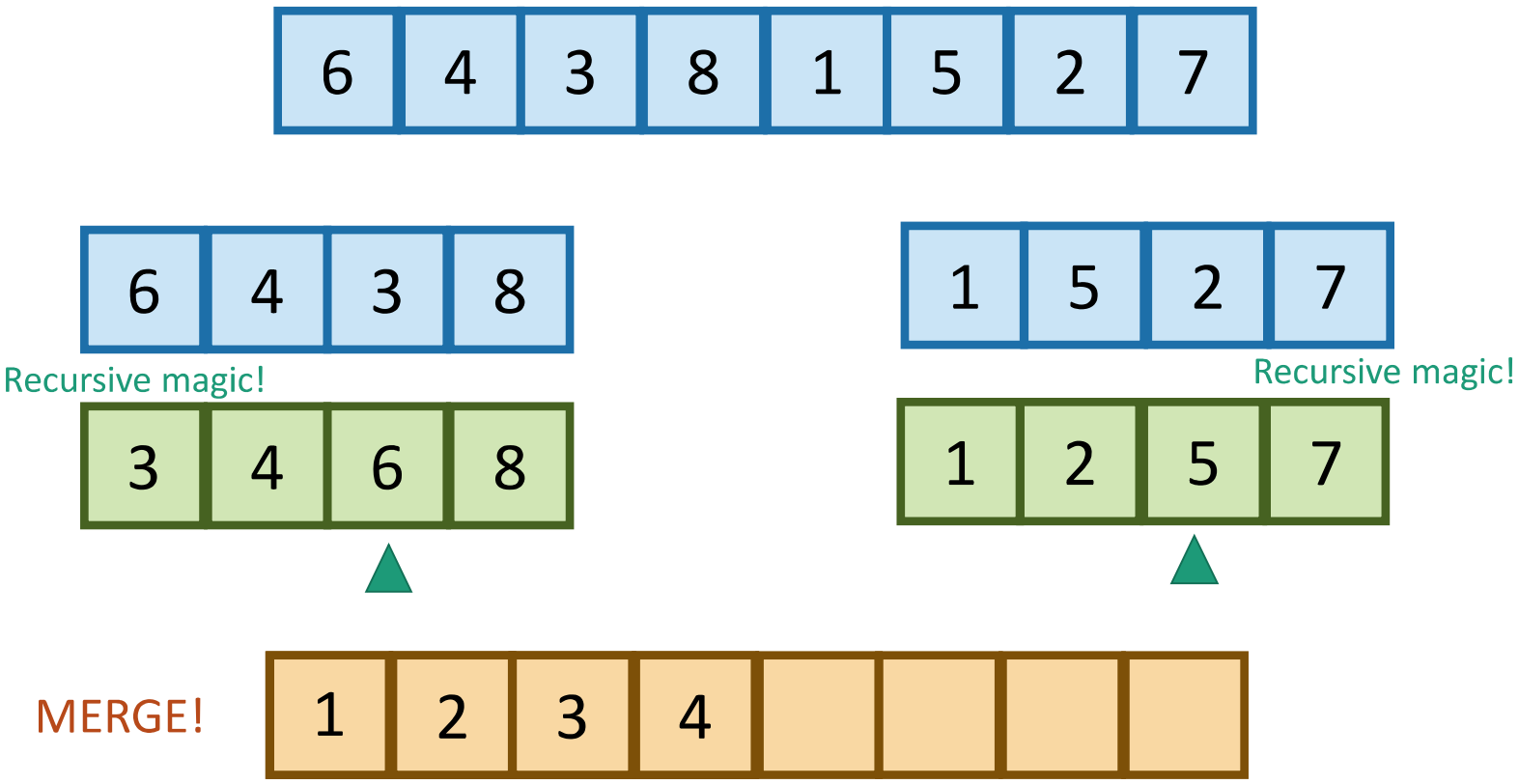


Recursive magic!

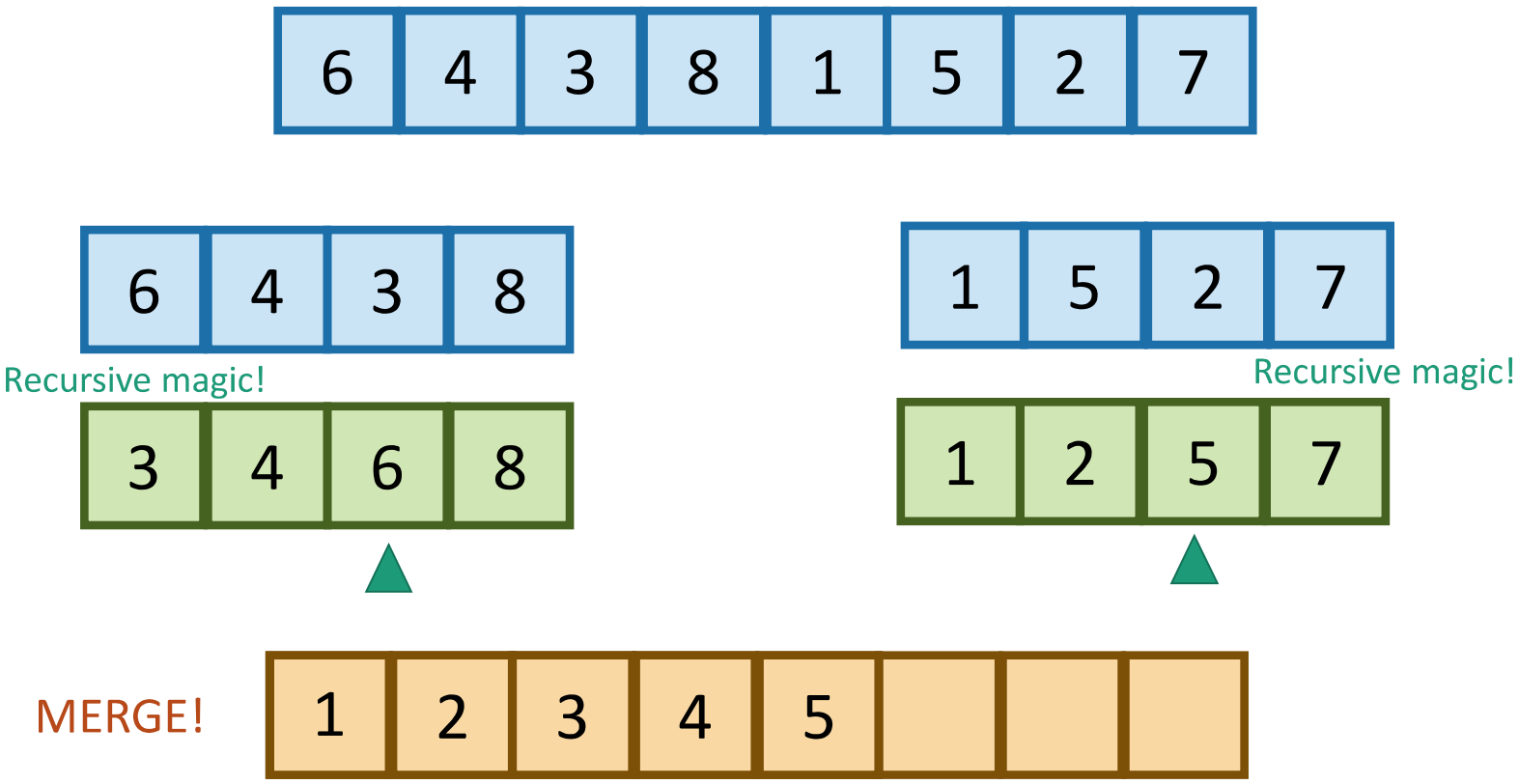


MERGE!

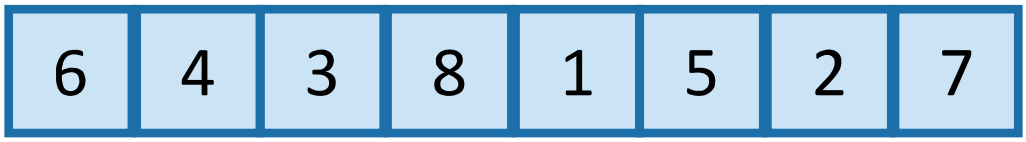
MergeSort

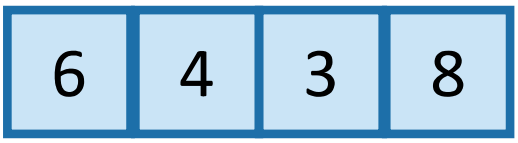

6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4

MergeSort

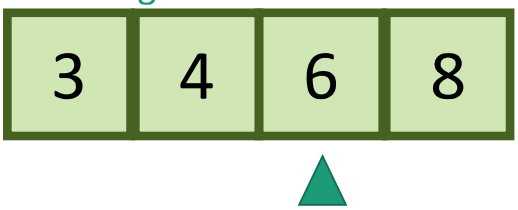

6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4 5

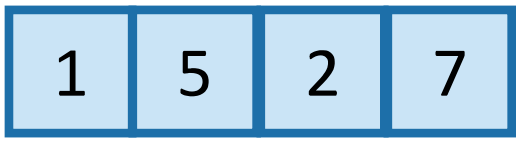
MergeSort



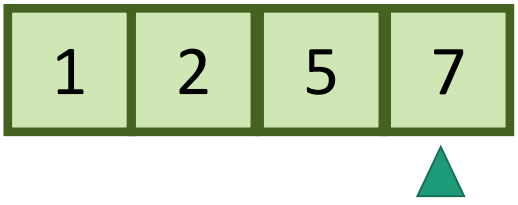


Recursive magic!

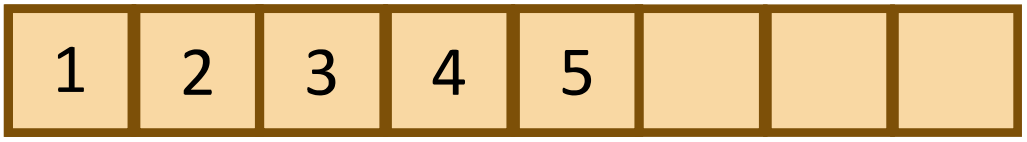




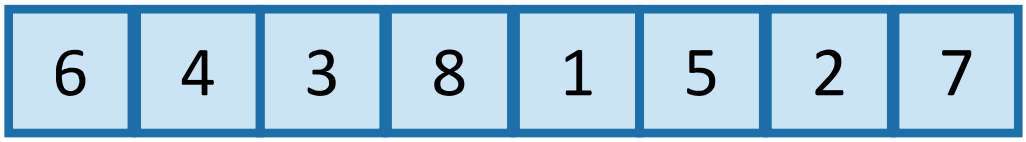
Recursive magic!

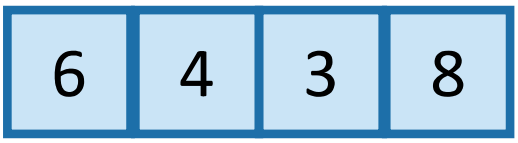


MERGE!

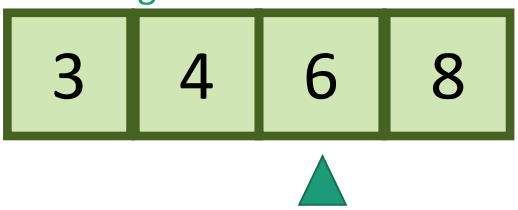


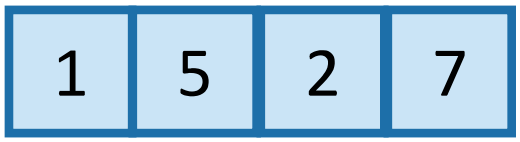
MergeSort



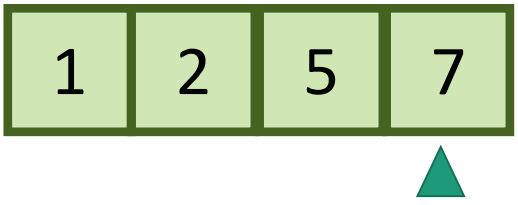


Recursive magic!

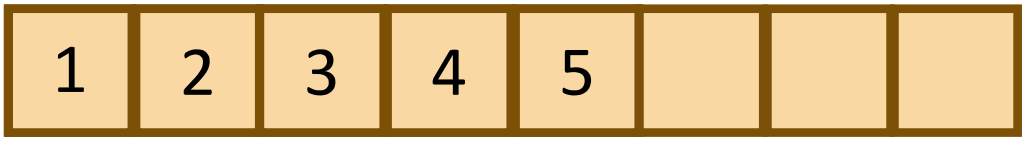




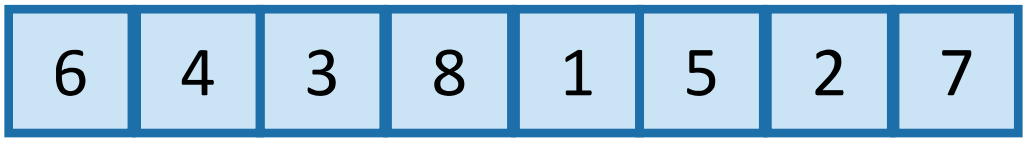
Recursive magic!

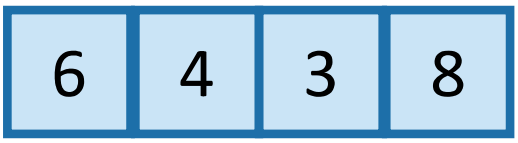


MERGE!

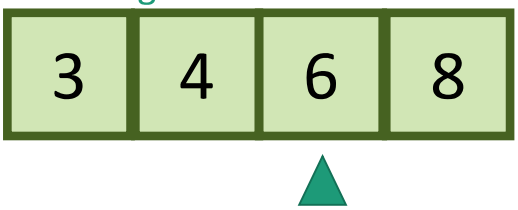


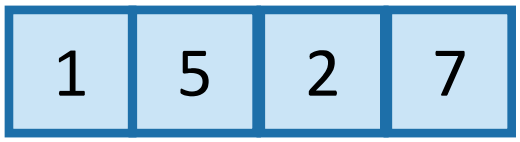
MergeSort



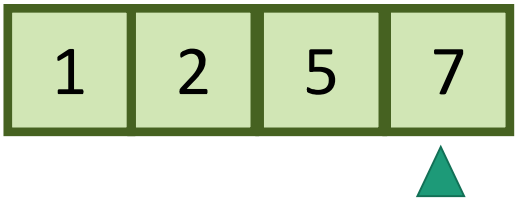


Recursive magic!

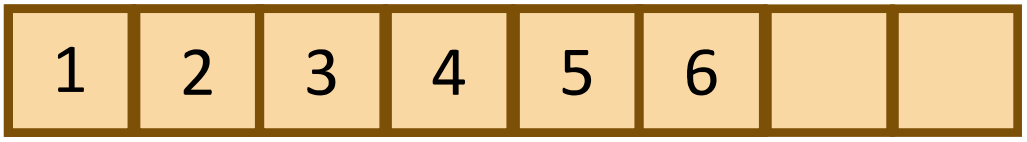




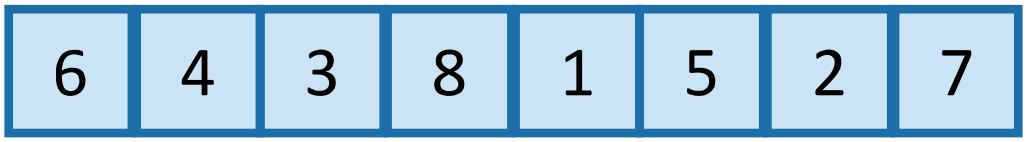
Recursive magic!

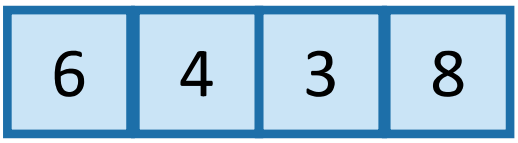


MERGE!

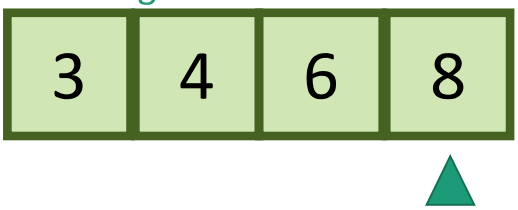


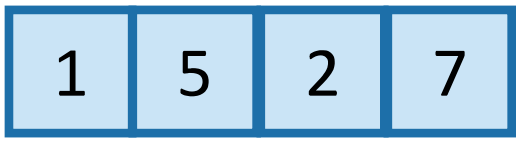
MergeSort



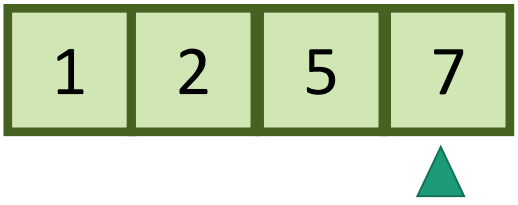


Recursive magic!

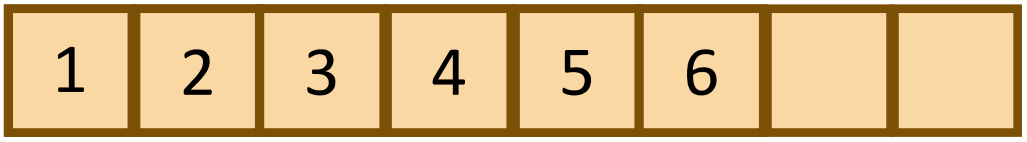




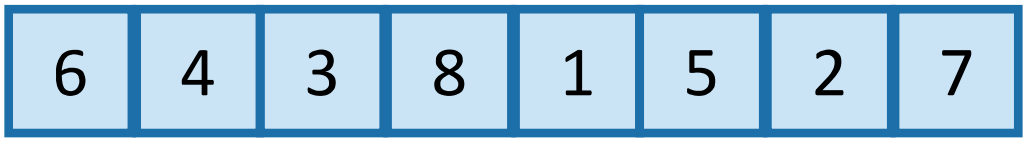
Recursive magic!

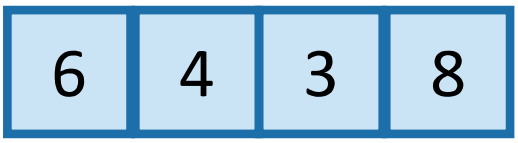


MERGE!

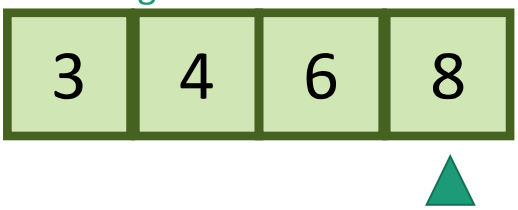


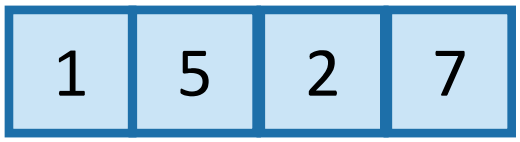
MergeSort



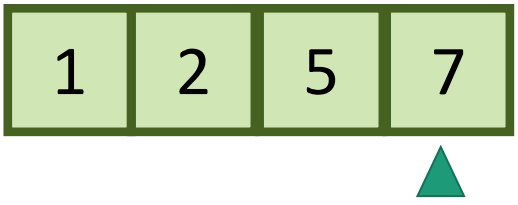


Recursive magic!

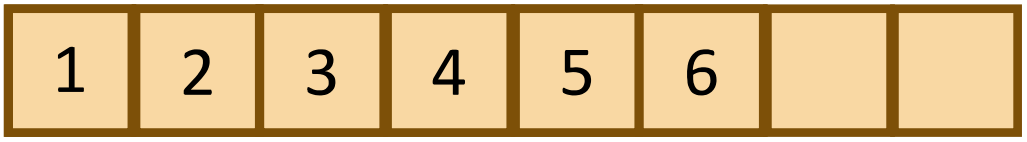




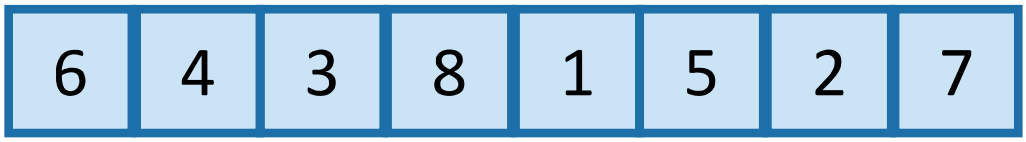
Recursive magic!

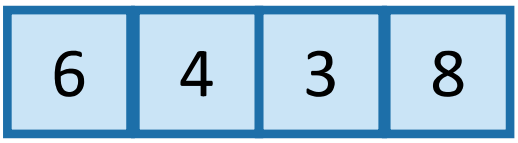


MERGE!

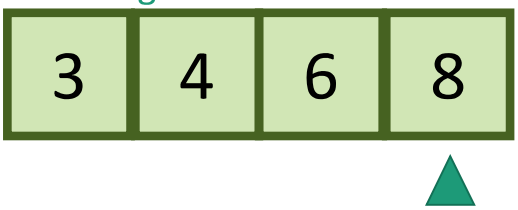


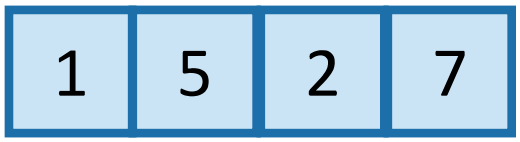
MergeSort



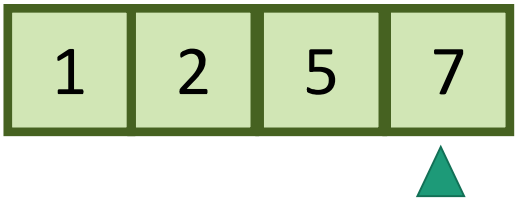


Recursive magic!

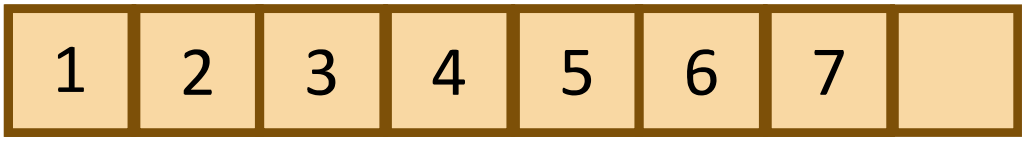




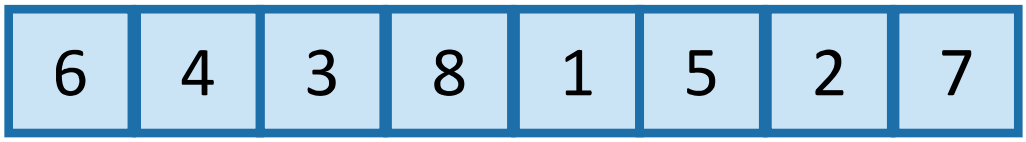
Recursive magic!

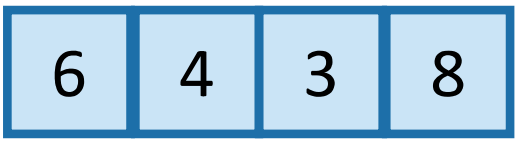


MERGE!

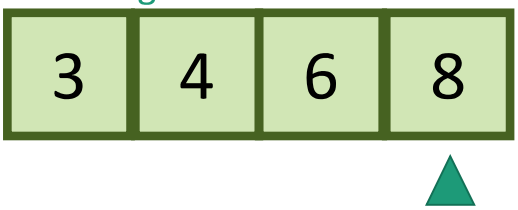


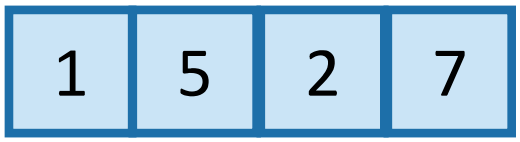
MergeSort



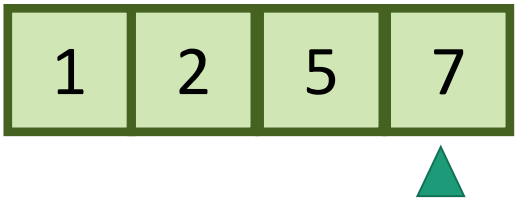


Recursive magic!

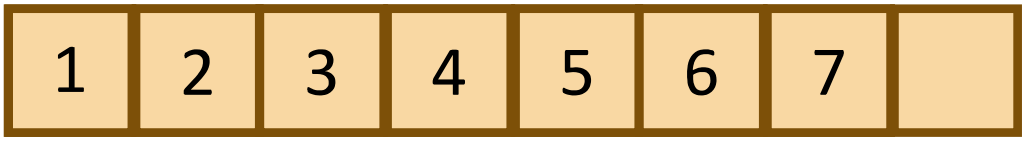




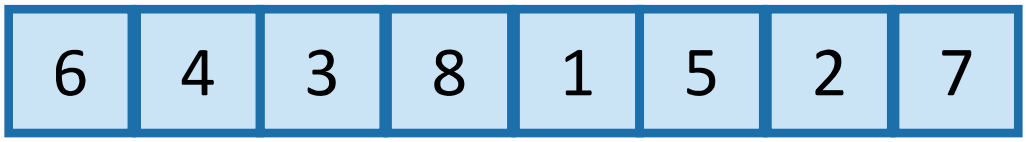
Recursive magic!

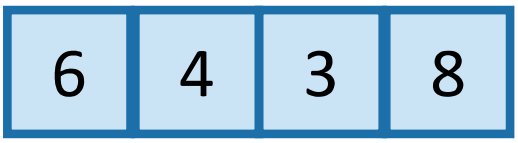


MERGE!

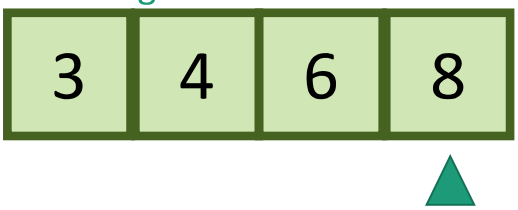


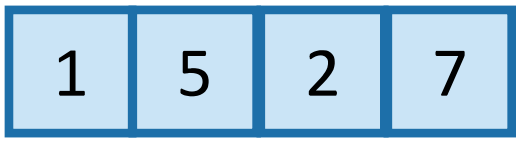
MergeSort



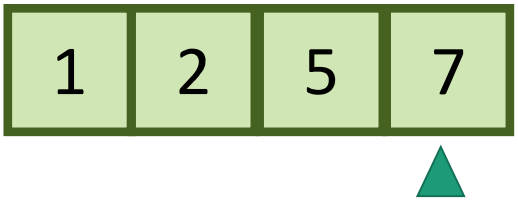


Recursive magic!

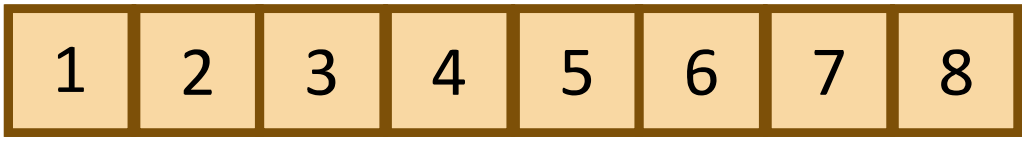




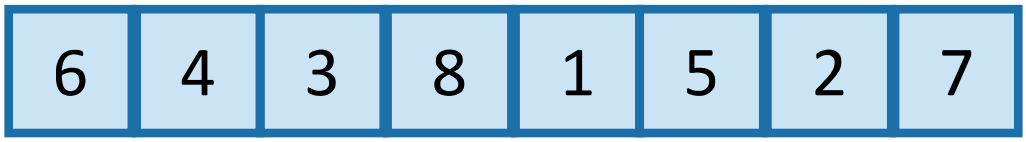
Recursive magic!

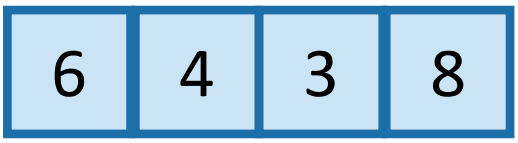


MERGE!

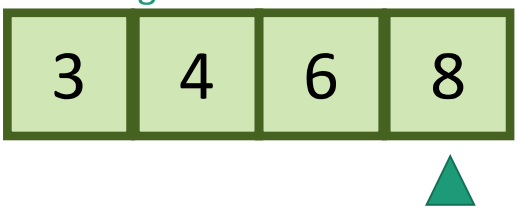


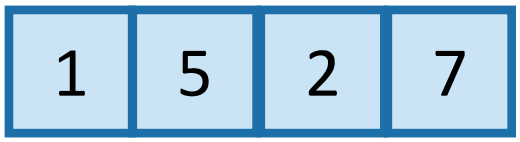
MergeSort



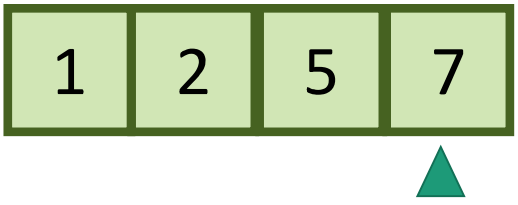


Recursive magic!

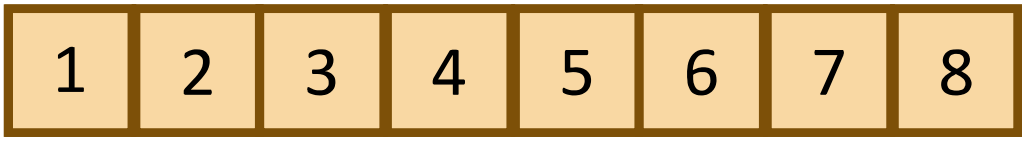




Recursive magic!

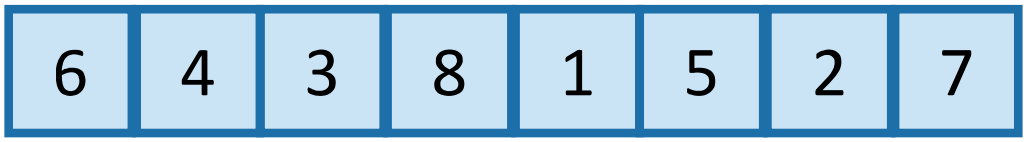


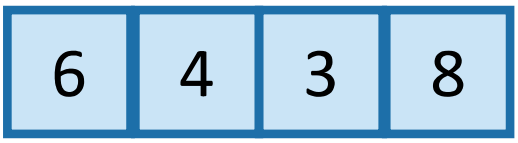
MERGE!



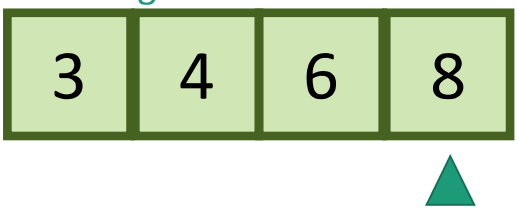
MERGE pseudocode + analysis on board, or CRLS

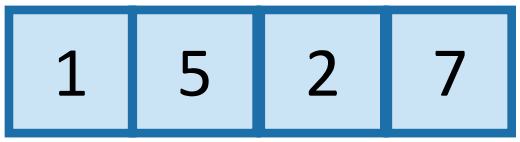
MergeSort



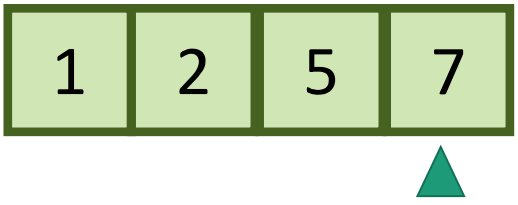


Recursive magic!



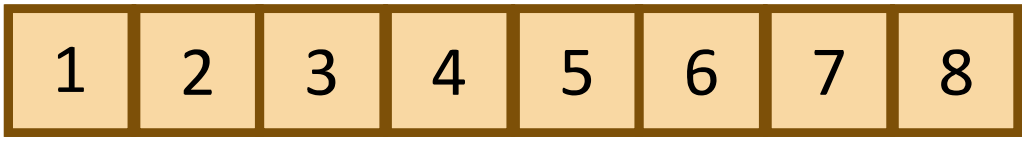


Recursive magic!



How would you do this in-place?

MERGE!



MERGE pseudocode + analysis on board, or CRLS

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[1 : n/2])

Sort the left half

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

Sort the right half

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

Merge the two halves