# CSE100: Design and Analysis of Algorithms
# Lecture 03 – Sorting

**Jan 25th 2022**

Multiplication (cont.)
InsertionSort, Divide-and-conquer, MergeSort

# Last Week

## Philosophy

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

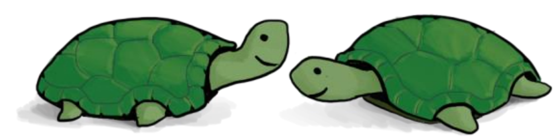
## Technical content

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

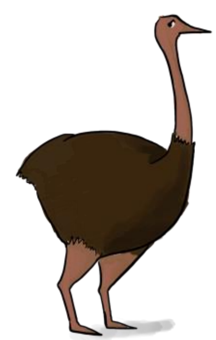
## Cast

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?
- Integer Multiplication (cont.)
  - Karatsuba Integer Multiplication
- We'll start to see how to answer these by looking at some examples of sorting algorithms.
  - InsertionSort
  - MergeSort

SortingHatSort not discussed

# Today

- Integer Multiplication (wrap up)
- 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

# Divide & conquer Integer Multiplication (review)

Break up an n-digit integer:

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

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

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

# Divide and conquer algorithm (review)

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

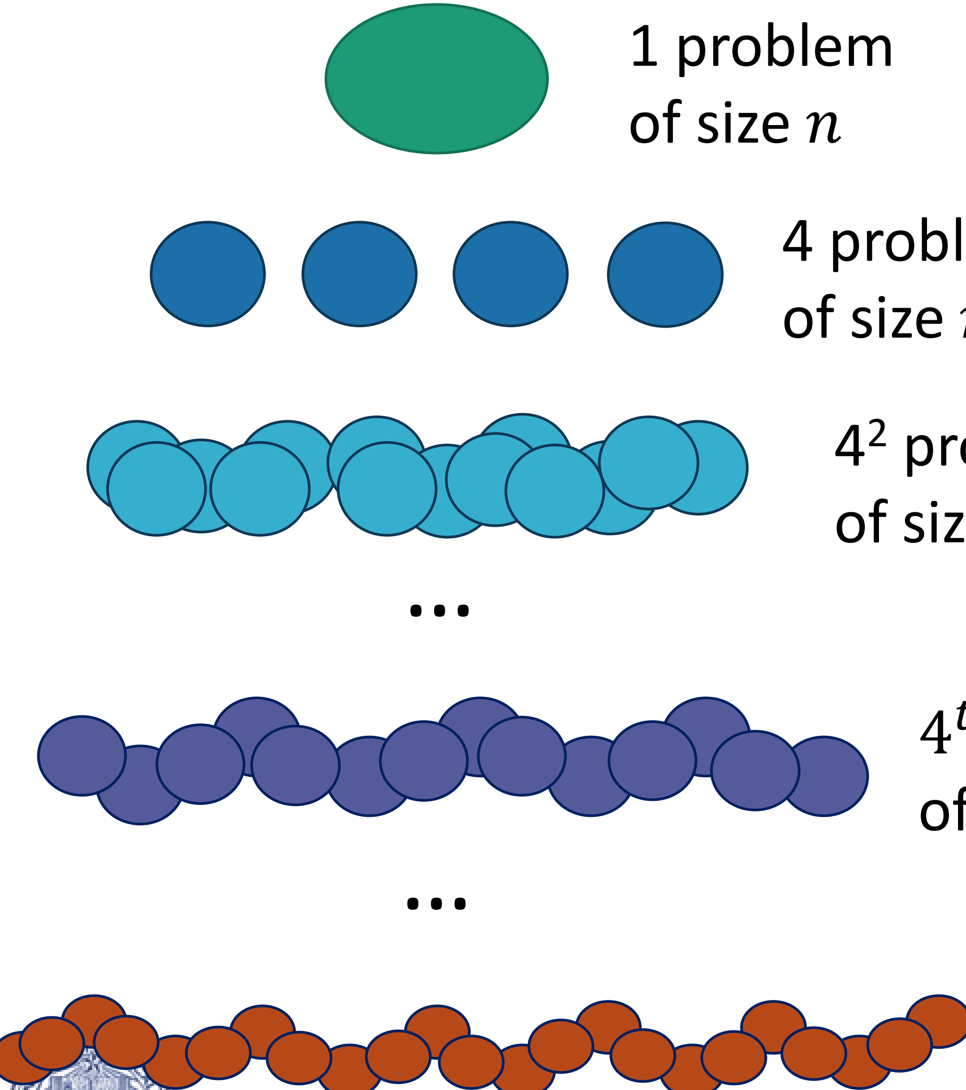
- Recursively compute $ac, ad, bc, bd$:
  - $ac$ = **Multiply**$(a, c)$, etc…
- Add them up to get $xy$:
  - $xy = ac10^n + (ad + bc)10^{n/2} + bd$

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

Siggi the Studious Stork

# There are $n^2$ 1-digit problems*

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




$\underline{n^2}$ 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?

exponent

$2^3$

base

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:

$$2^3 = 8$$

$$\log_2(8) = 3$$

exponent

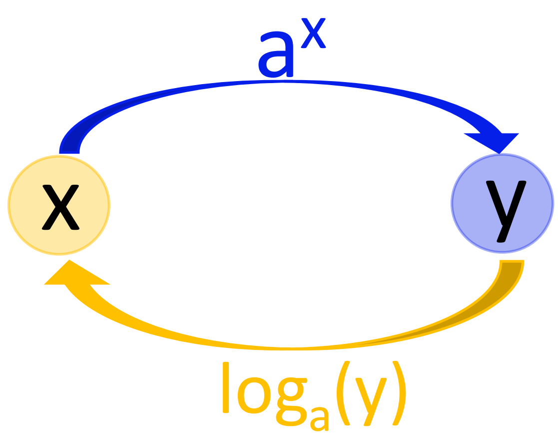
$$2^3 = 8 \quad \Longleftrightarrow \quad \log_2(8) = 3$$

base

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

# Going back to 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*

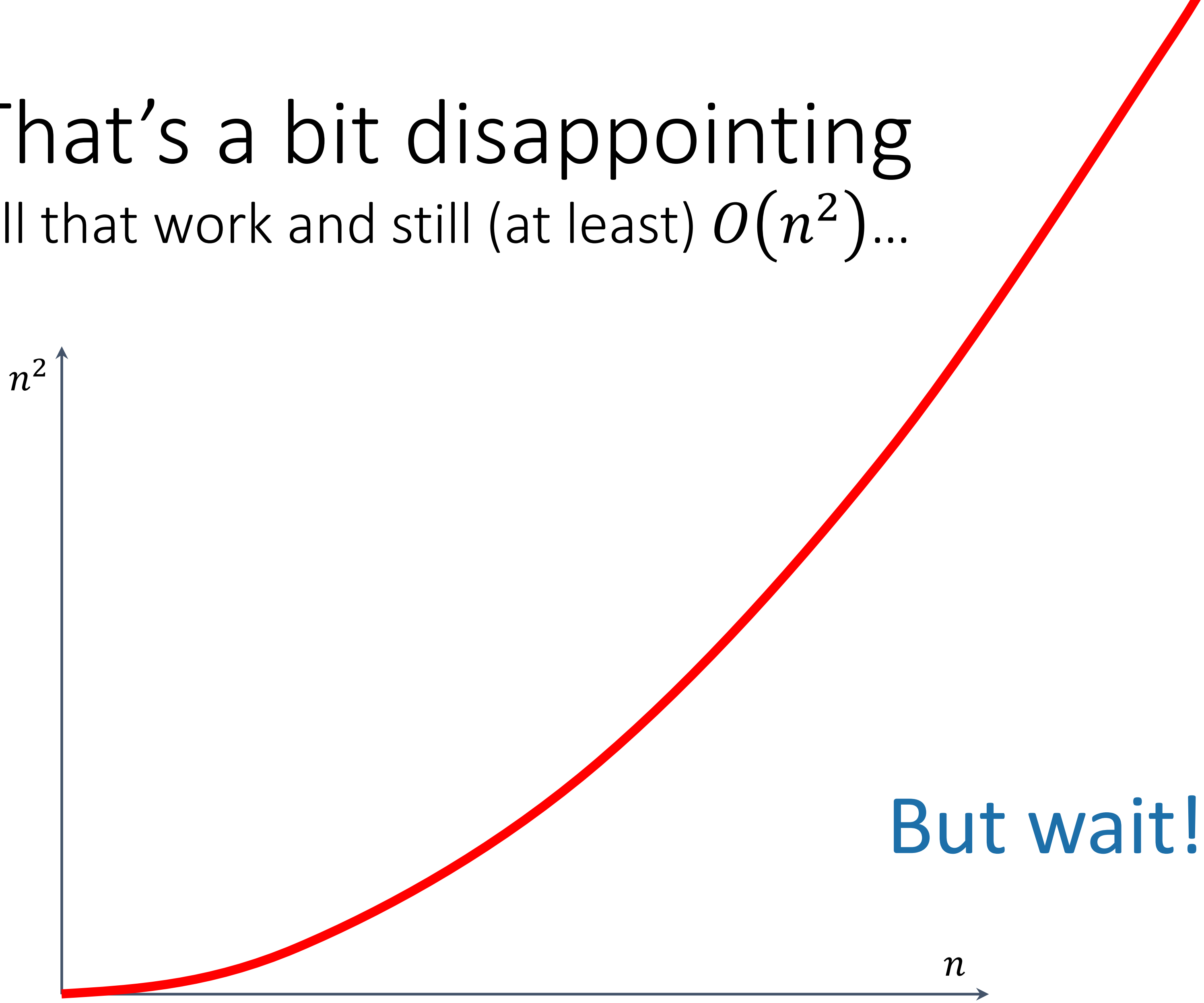
- Let T(n) be the time to multiply two n-digit numbers.
- Recurrence relation:
  - $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + \text{(about n to add stuff up)}$

Ignore this term for now…

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

# 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!

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

Need these three things

- If only we recurse three times instead of four…

# Karatsuba integer multiplication

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

Subtract these off

get this

(a+b)(c+d) = ac + bd + bc + ad

- 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?

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

x,y are n-digit numbers

**Multiply**$(x, y)$:

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

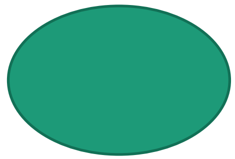
a, b, c, d are n/2-digit numbers

- $ac$ = **Multiply**$(a, c)$
- $bd$ = **Multiply**$(b, d)$
- $z$ = **Multiply**$(a + b, c + d)$
- cross_terms = $z - ac - bd$
- $xy = ac10^{n} +$ (cross_terms) $10^{n/2} + bd$
- Return $xy$

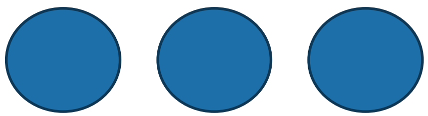
We can do the addition $a + b$ and $c + d$ in time $O(n)$. This results in integers that are still roughly $n/2$ bits long.

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

# What's the running time?

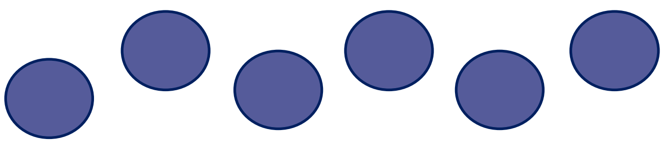


1 problem
of size n



3 problems
of size n/2

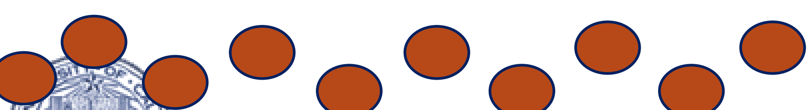
...



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

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

...



$n^{1.6}$ 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...

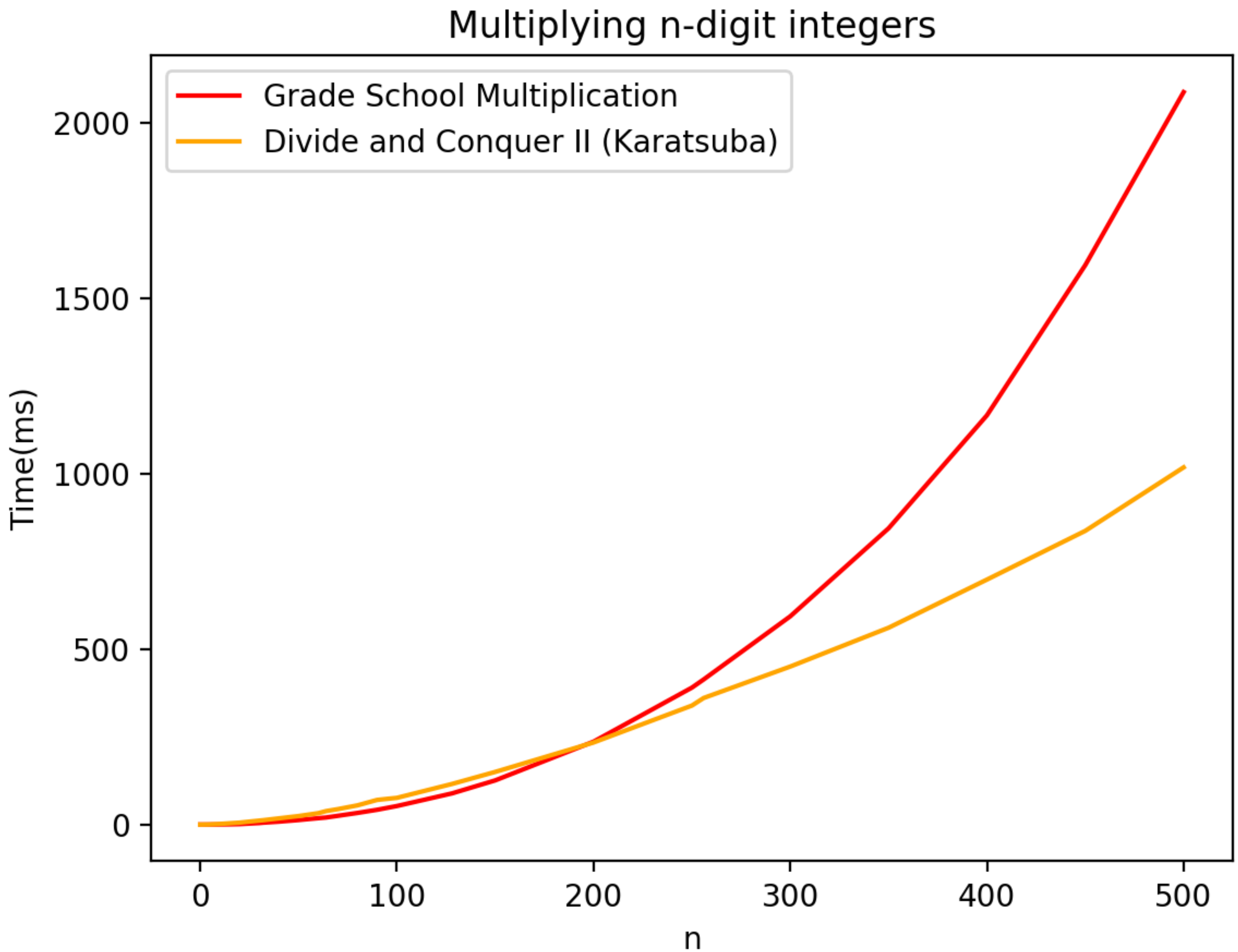
$$3^{\log_2 n} = n^{\log_2 3} \approx n^{1.6}$$

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!



# 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!]

# 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

# Today

- Integer Multiplication (wrap up)
- 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.

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

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

# Benchmark: insertion sort

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

- Say we want to sort:

| | | | | |
|---|---|---|---|---|
| 6 | 4 | 3 | 8 | 5 |

# 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?

| | | | | |
|---|---|---|---|---|
| 6 | 4 | 3 | 8 | 5 |

# Benchmark: insertion sort

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

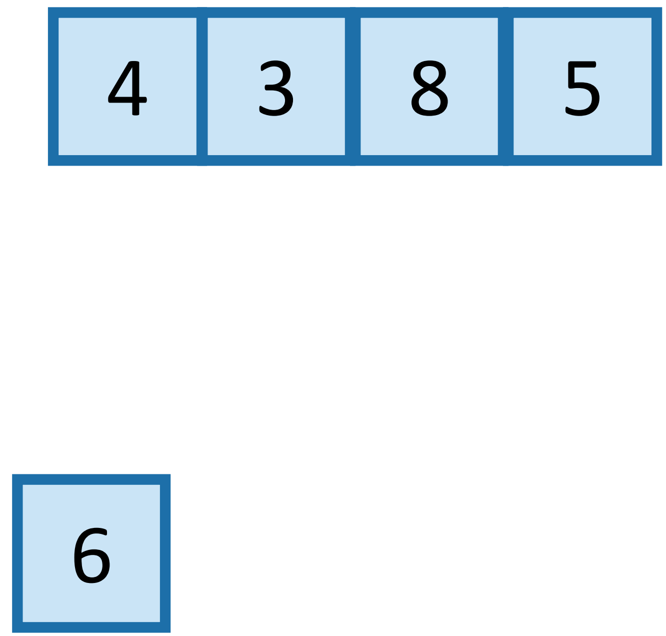
- Say we want to sort:
- How would you do it?
- Insert items one at a time.

| | | | | |
|---|---|---|---|---|
| 6 | 4 | 3 | 8 | 5 |

# 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.

| | | |
|---|---|---|
| 3 | 8 | 5 |

4

6

# 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.

8 5

3 4 6

# 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.

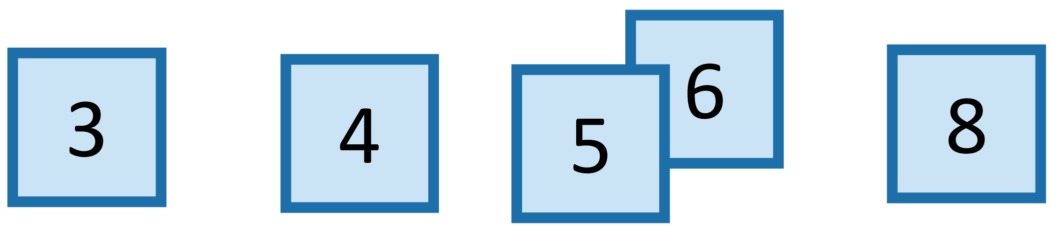
5

3 4 6 8

# Benchmark: insertion sort

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

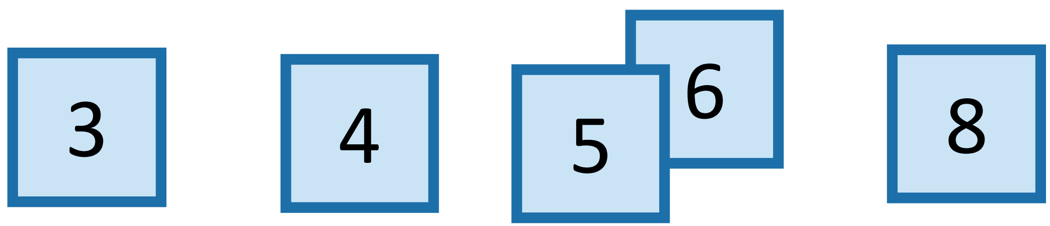
- Say we want to sort:
- How would you do it?
- Insert items one at a time.



# 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…

| | | | | |
|---|---|---|---|---|
| 6 | 4 | 3 | 8 | 5 |

# Insertion sort example…

| | | | | |
|---|---|---|---|---|
| 6 | 4 | 3 | 8 | 5 |

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

# Insertion sort example…

| | | | | |
|---|---|---|---|---|
| 6 | 4 | 3 | 8 | 5 |
| 4 | 6 | 3 | 8 | 5 |

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…

| | | | | |
|---|---|---|---|---|
| 6 | 4 | 3 | 8 | 5 |

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

| | | | | |
|---|---|---|---|---|
| 4 | 6 | 3 | 8 | 5 |

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

| | | | | |
|---|---|---|---|---|
| 4 | 6 | 3 | 8 | 5 |

Now look at “3”

# Insertion sort example…

| | | | | | |
|---|---|---|---|---|---|
| 6 | 4 | 3 | 8 | 5 | Start with the second element (the first element is sorted within itself…) |
| 4 | 6 | 3 | 8 | 5 | Pull “4” back until it’s in the right place. |
| 4 | 6 | 3 | 8 | 5 | Now look at “3” |
| 3 | 4 | 6 | 8 | 5 | Pull “3” back until it’s in the right place. |

# Insertion sort example…

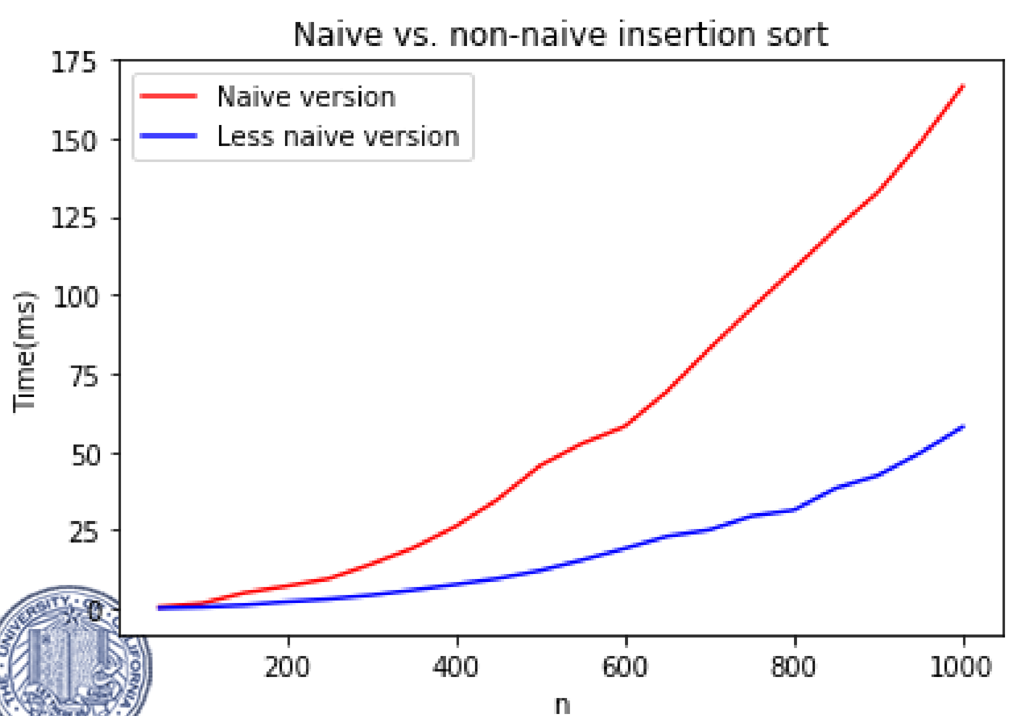
| | | | | | |
|---|---|---|---|---|---|
| 6 | 4 | 3 | 8 | 5 | Start with the second element (the first element is sorted within itself…) |
| 4 | 6 | 3 | 8 | 5 | Pull “4” back until it’s in the right place. |
| 4 | 6 | 3 | 8 | 5 | Now look at “3” |
| 3 | 4 | 6 | 8 | 5 | Pull “3” back until it’s in the right place. |
| 3 | 4 | 6 | 8 | 5 | “8” is good…look at 5 |

# Insertion sort example...

| | | | | | |
|---|---|---|---|---|---|
| 6 | 4 | 3 | 8 | 5 | Start with the second element (the first element is sorted within itself...) |
| 4 | 6 | 3 | 8 | 5 | Pull "4" back until it's in the right place. |
| 4 | 6 | 3 | 8 | 5 | Now look at "3" |
| 3 | 4 | 6 | 8 | 5 | Pull "3" back until it's in the right place. |
| 3 | 4 | 6 | 8 | 5 | "8" is good...look at 5 |
| 3 | 4 | 5 | 6 | 8 | (then fix 5 and we're done) |

# Insertion Sort

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



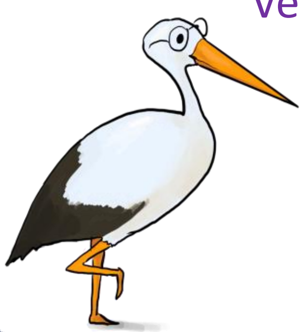

- The "same" algorithm can be faster or slower depending on the implementation…
- We are interested in how fast the running time scales with n, the size of the input.

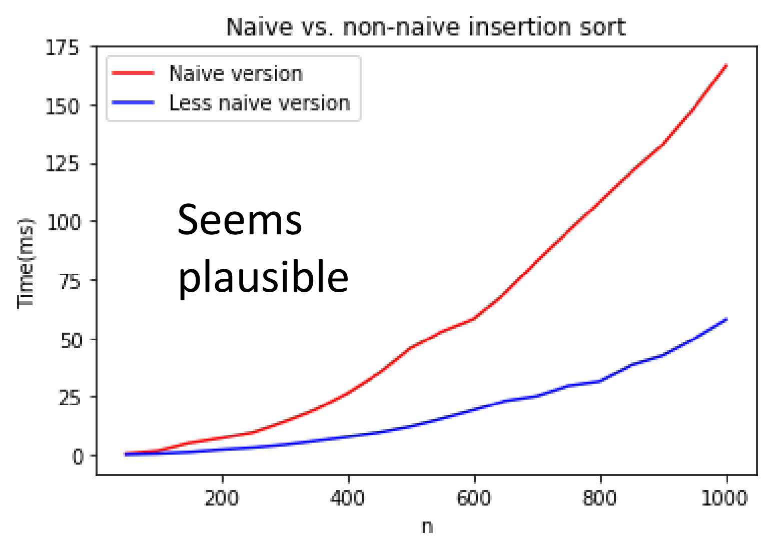
# Insertion Sort: running time

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!






Seems plausible

# Insertion sort pseudocode

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

**Algorithm 1:** $\textsc{InsertionSort}(A)$

**for** $i = 2 \to length(A)$ **do**
- $key \leftarrow A[i]$;
- $j \leftarrow i - 1$;
- **while** $j > 0$ *and* $A[j] > key$ **do**
  - $A[j+1] \leftarrow A[j]$;
  - $j \leftarrow j - 1$;
- $A[j+1] \leftarrow key$;

Lucky the lackadaisical lemur

Plucky the pedantic penguin

- (Discussion on board)

# Insertion sort: running time

**Algorithm 1:** $\textsc{InsertionSort}(A)$

$$
\begin{array}{l}
\textbf{for } i = 2 \rightarrow \mathit{length}(A) \textbf{ do} \\
\quad \mathit{key} \leftarrow A[i]; \\
\quad j \leftarrow i - 1; \\
\quad \textbf{while } j > 0 \;\mathit{and}\; A[j] > \mathit{key} \textbf{ do} \\
\quad\quad A[j+1] \leftarrow A[j]; \\
\quad\quad j \leftarrow j - 1; \\
\quad A[j+1] \leftarrow \mathit{key};
\end{array}
$$

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.

| | | | | |
|---|---|---|---|---|
| 6 | 4 | 3 | 8 | 5 |

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

| | | | | |
|---|---|---|---|---|
| 4 | 6 | 3 | 8 | 5 |

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

| | | | | |
|---|---|---|---|---|
| 4 | 6 | 3 | 8 | 5 |

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

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 6 | 8 | 5 |

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

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 6 | 8 | 5 |

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

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 6 | 8 | 5 |

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

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 6 | 8 | 5 |

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

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 8 |

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

**YAY WE ARE DONE!**

# This sounds like a job for…

**Proof By Induction!**

# 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 CLRS for details)

| | | | | |
|---|---|---|---|---|
| 4 | 6 | 3 | 8 | 5 |

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

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 6 | 8 | 5 |

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

This was iteration i=2.

# Aside: proofs by induction

- We’re gonna see/do/skip over a lot of them.
- If that went by too fast and was confusing:
  - Slides
  - Notes
  - Book
  - Office Hours

Make sure you really understand the argument on the previous slide! Check out CLRS 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

- Integer Multiplication (wrap up)
- 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.

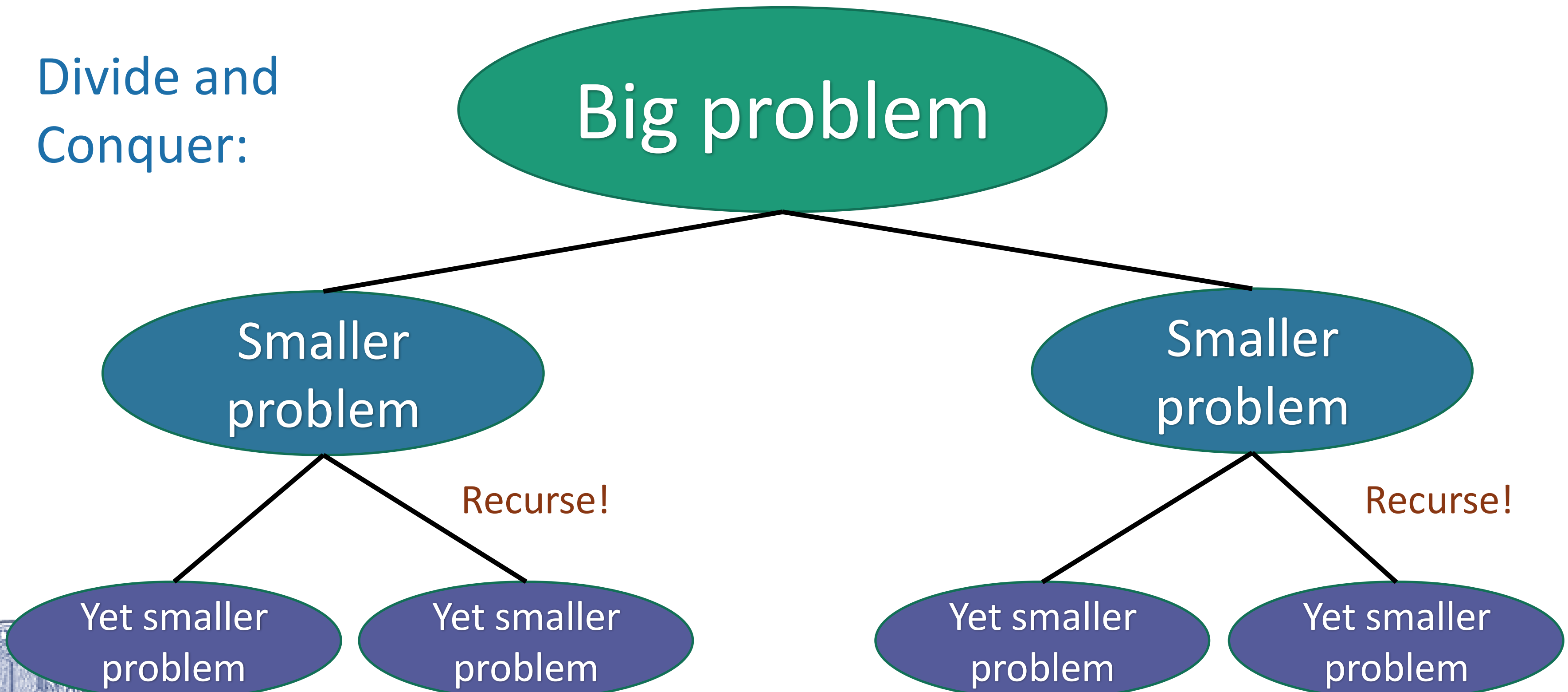
## Next Time:

- 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 |

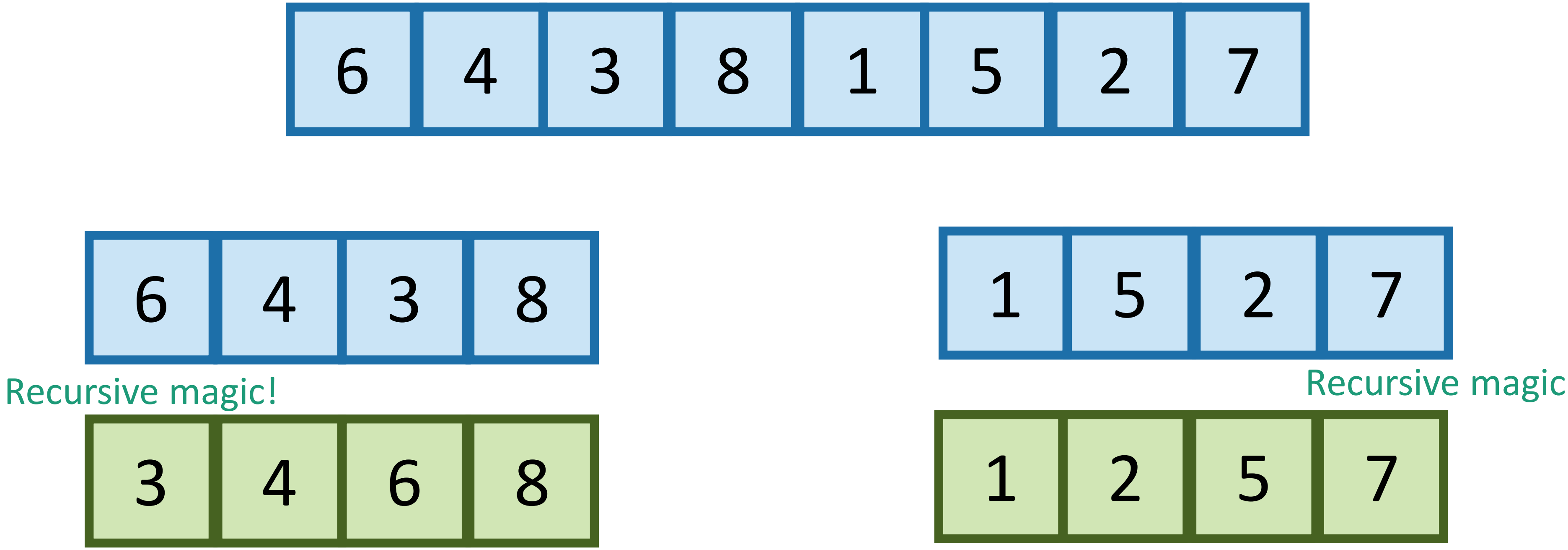
# MergeSort

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

| | | | |
|---|---|---|---|
| 6 | 4 | 3 | 8 |

| | | | |
|---|---|---|---|
| 1 | 5 | 2 | 7 |

# MergeSort



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

# MergeSort

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

| | | | |
|---|---|---|---|
| 6 | 4 | 3 | 8 |

Recursive magic!

| | | | |
|---|---|---|---|
| 3 | 4 | 6 | 8 |

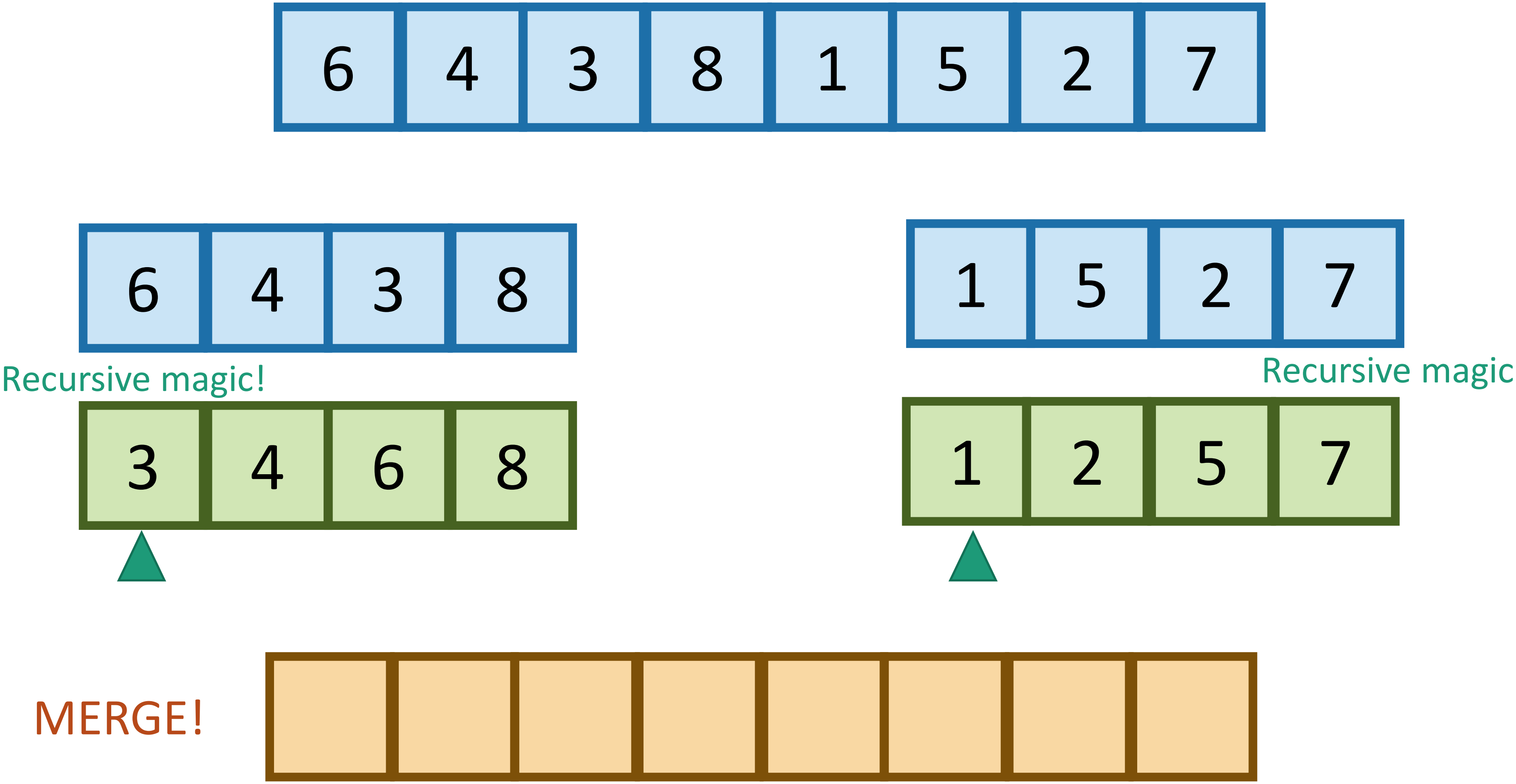
| | | | |
|---|---|---|---|
| 1 | 5 | 2 | 7 |

Recursive magic!

| | | | |
|---|---|---|---|
| 1 | 2 | 5 | 7 |

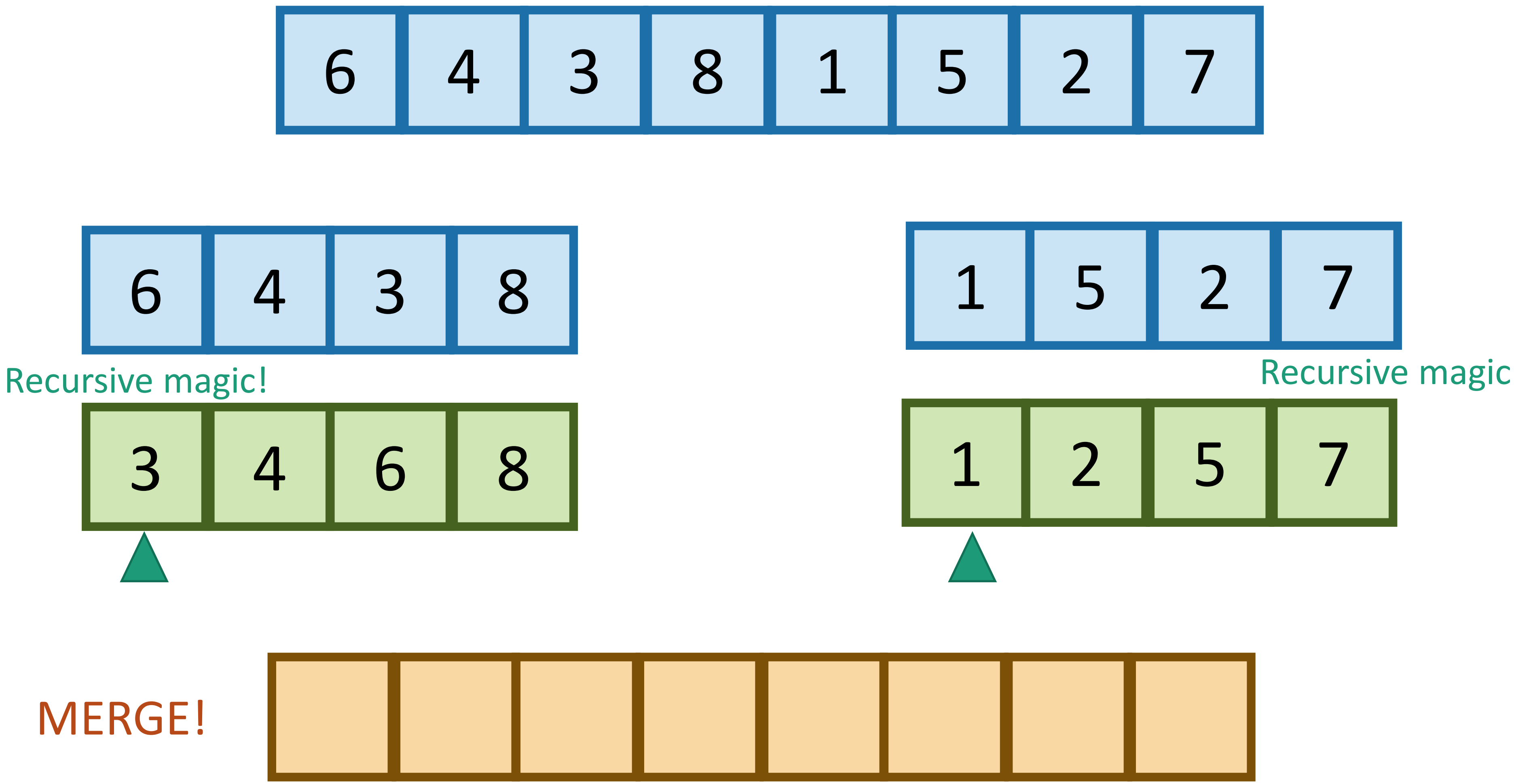
MERGE!

# MergeSort

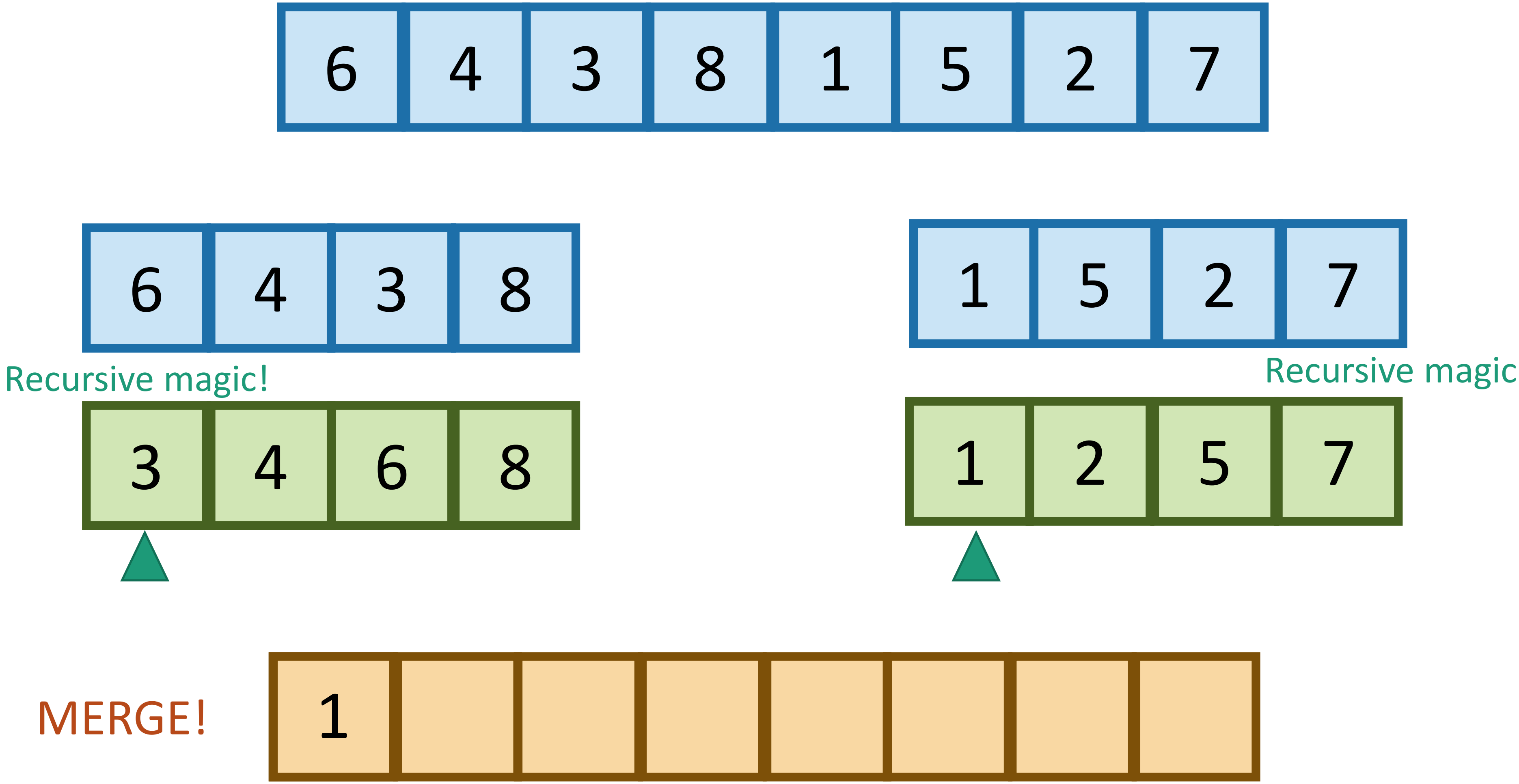


6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!

# MergeSort

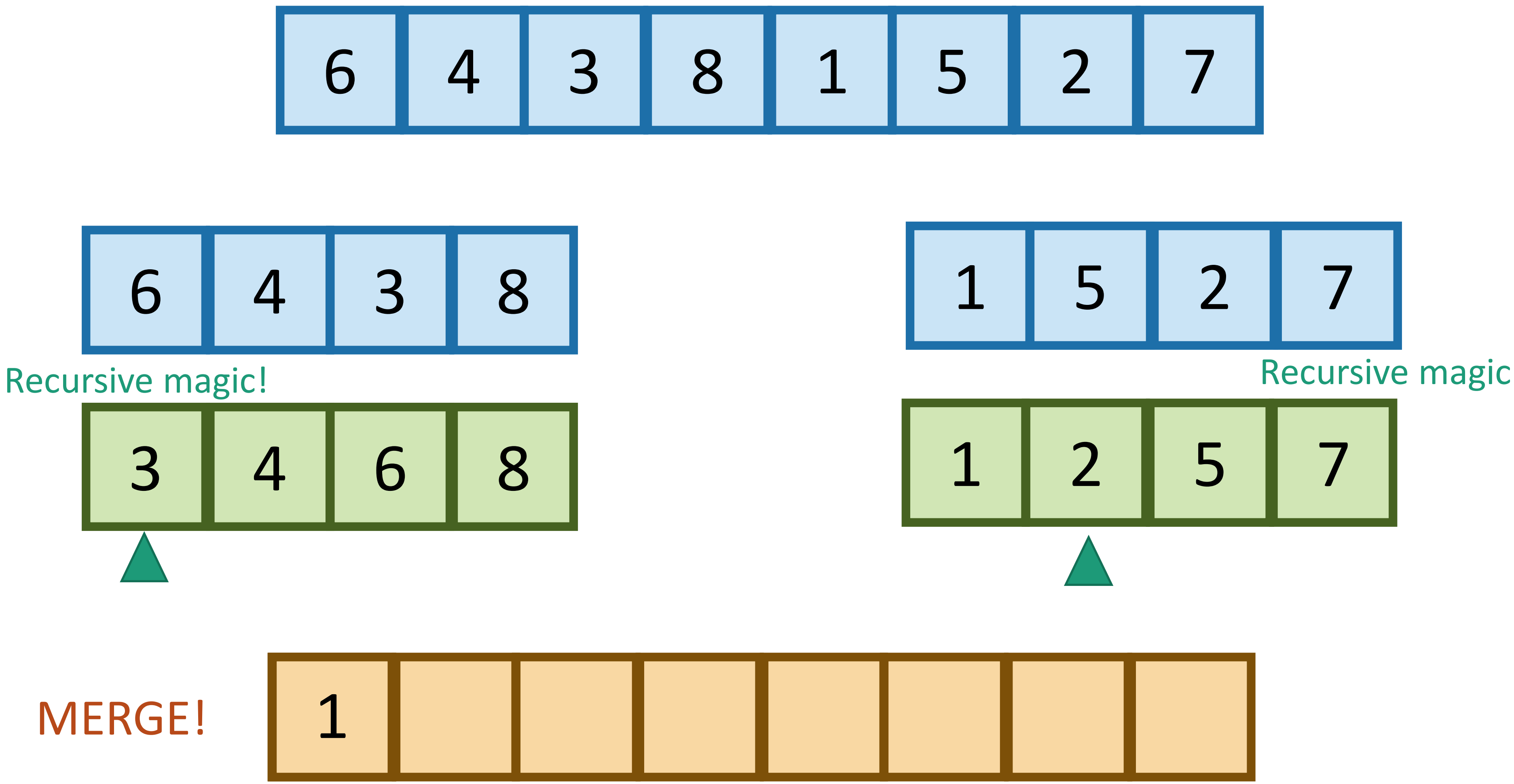

6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!

# MergeSort

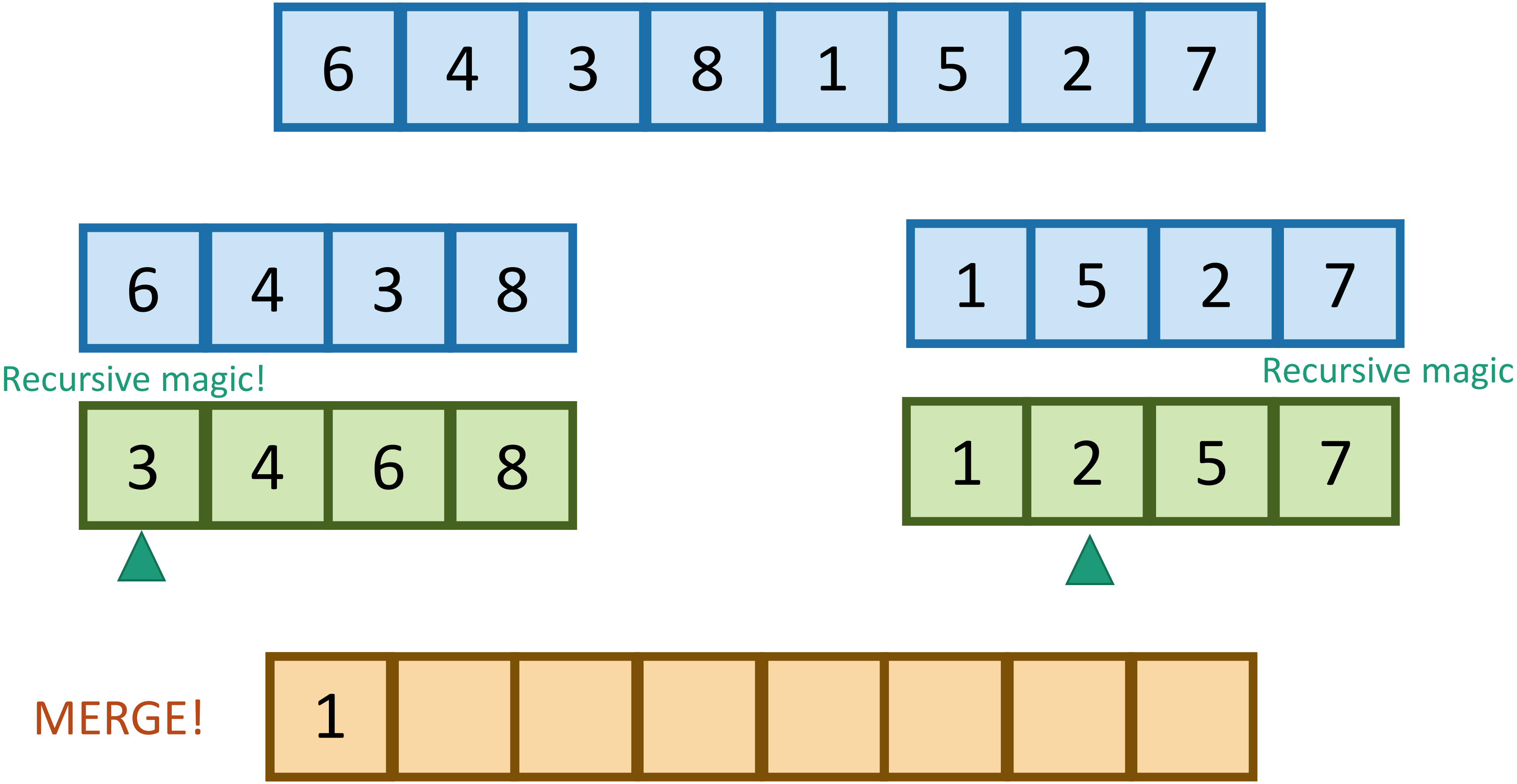


6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1

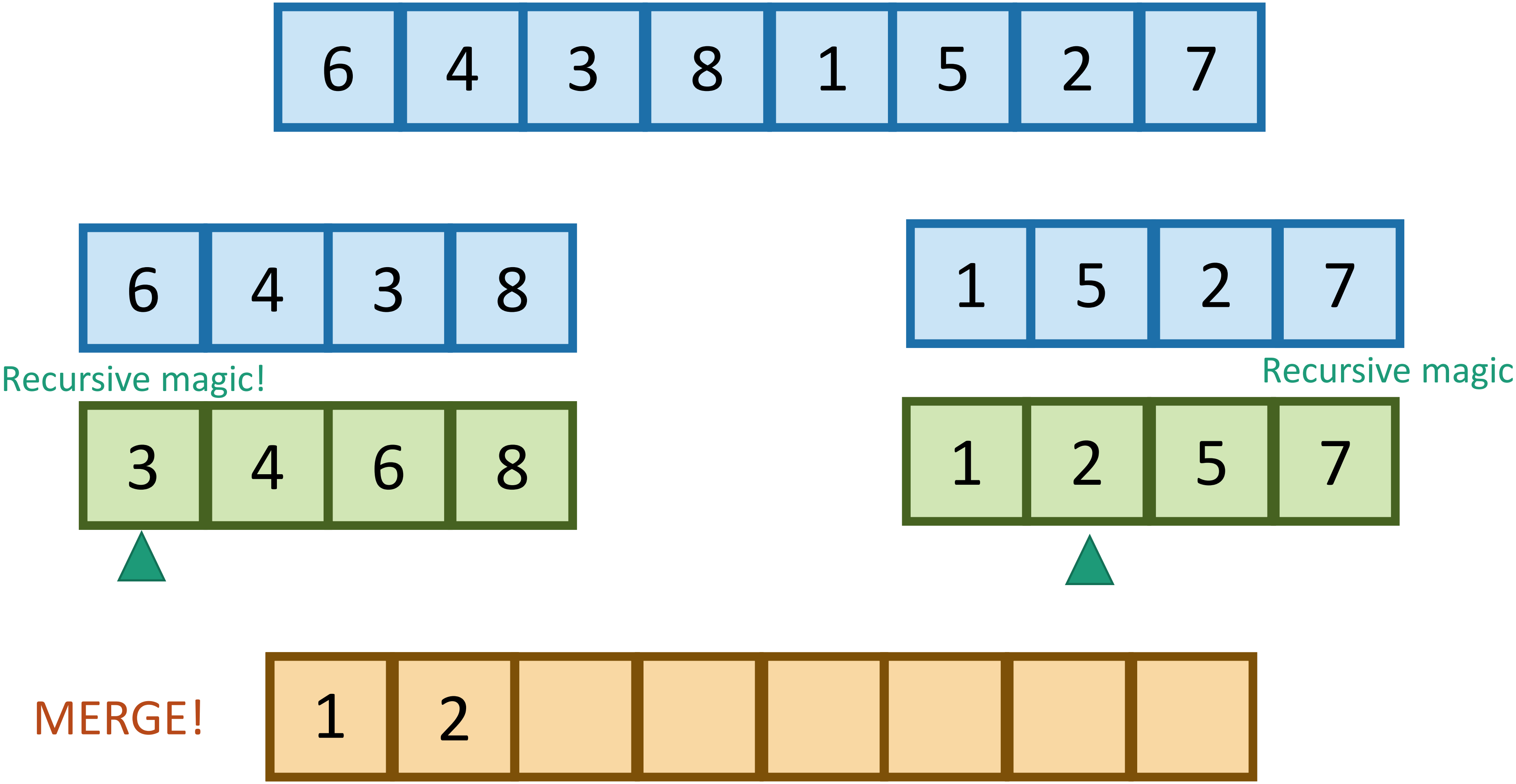
# MergeSort


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

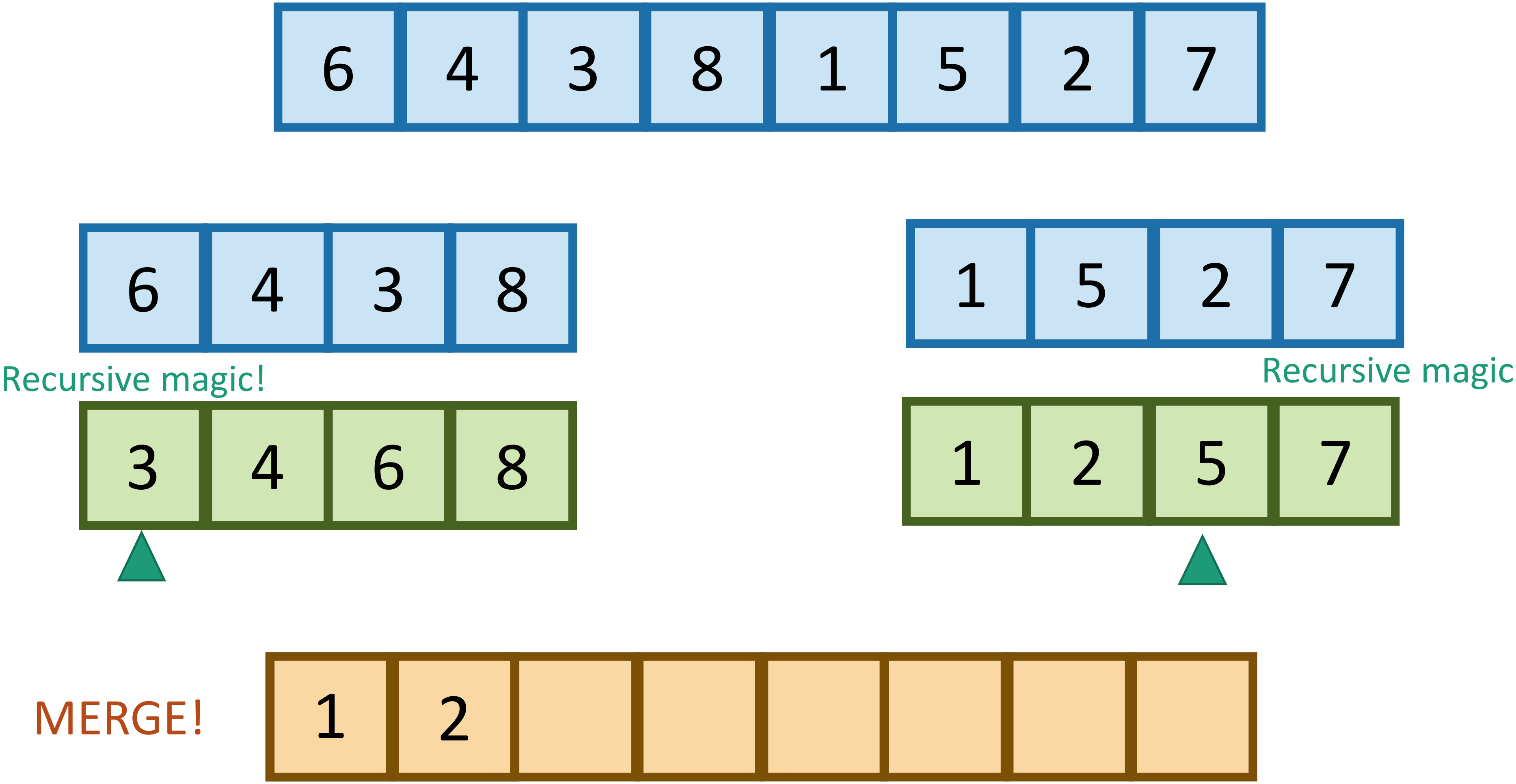
# MergeSort


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

# MergeSort

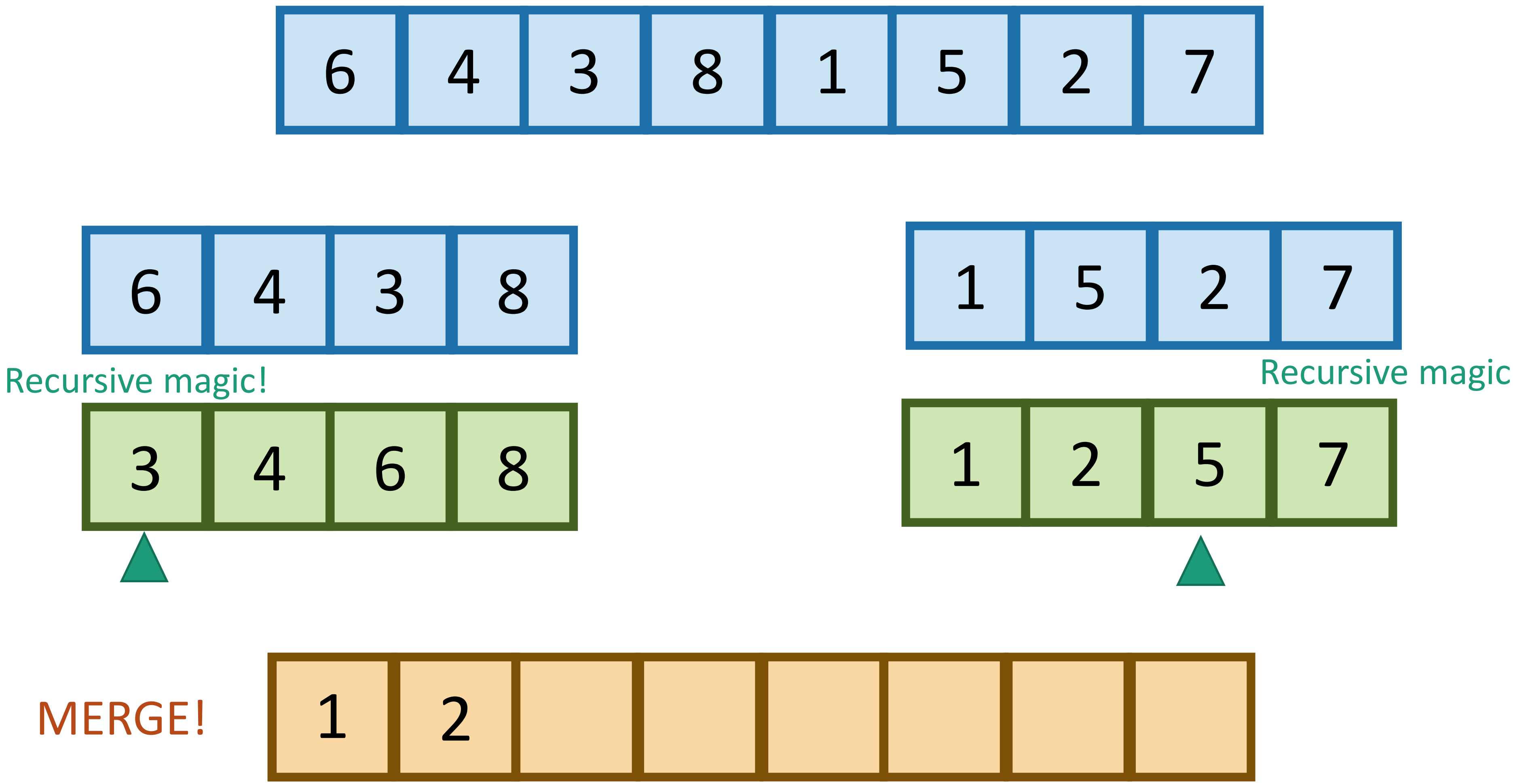

6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2

# MergeSort



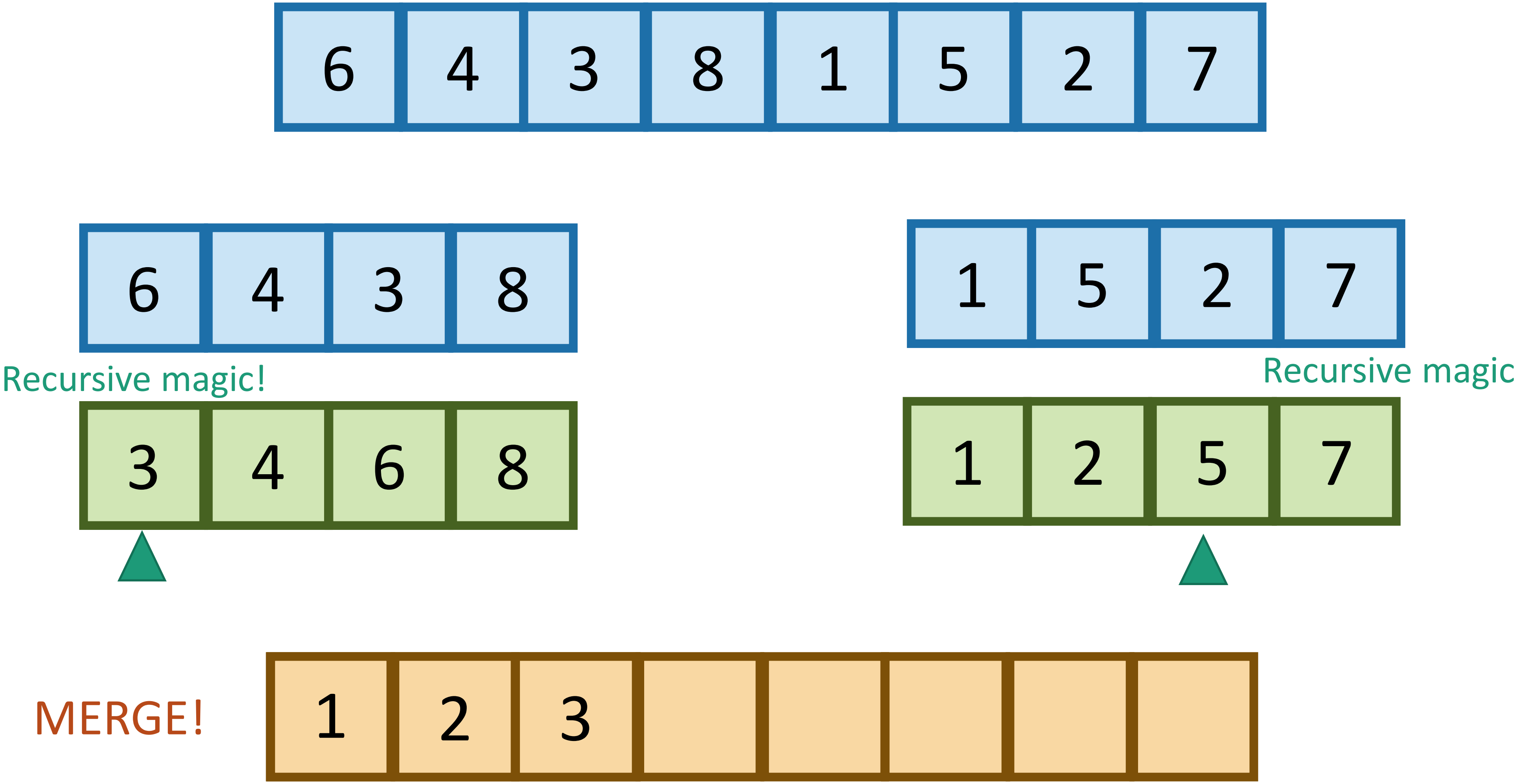
6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2

# MergeSort



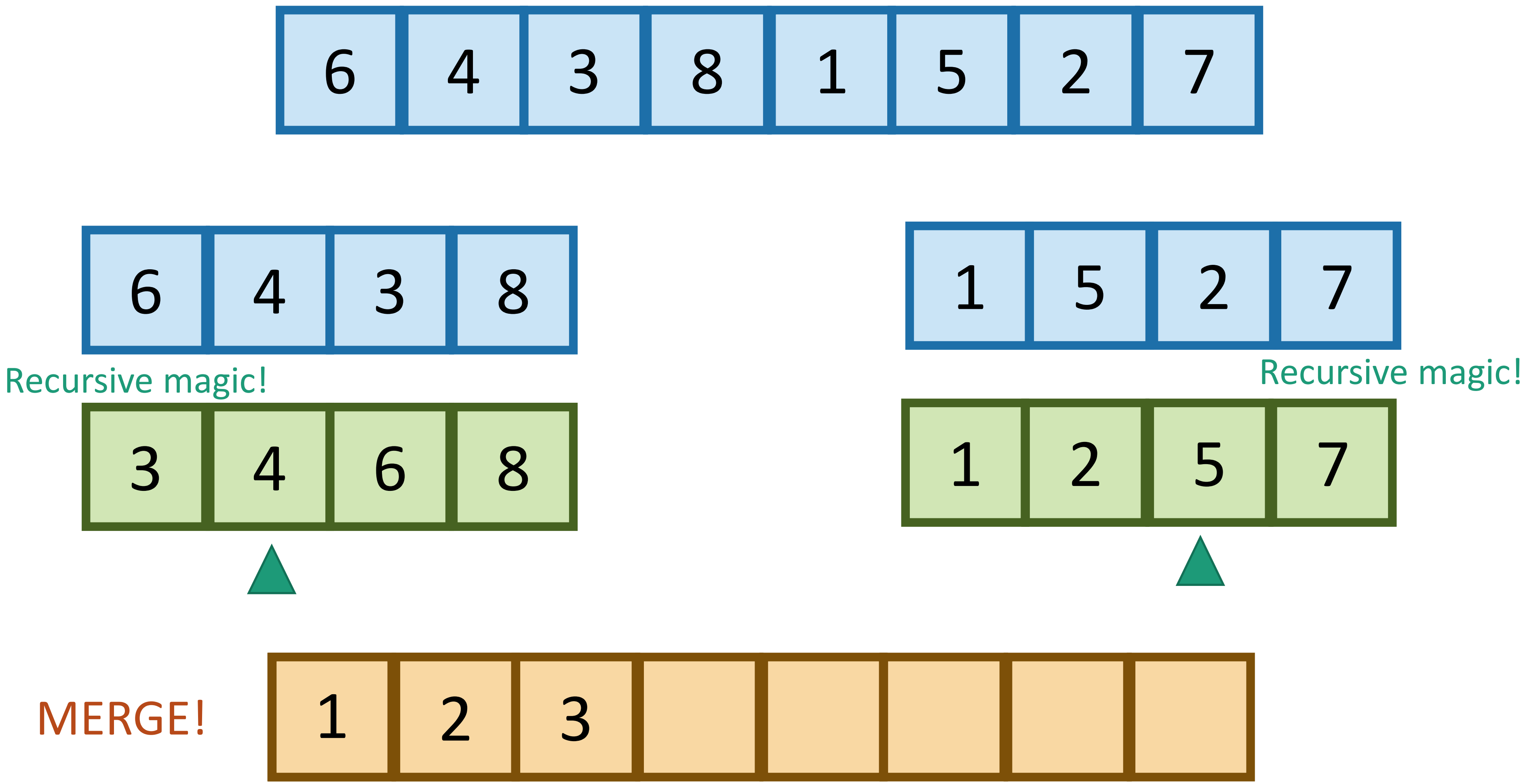
6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2

# MergeSort



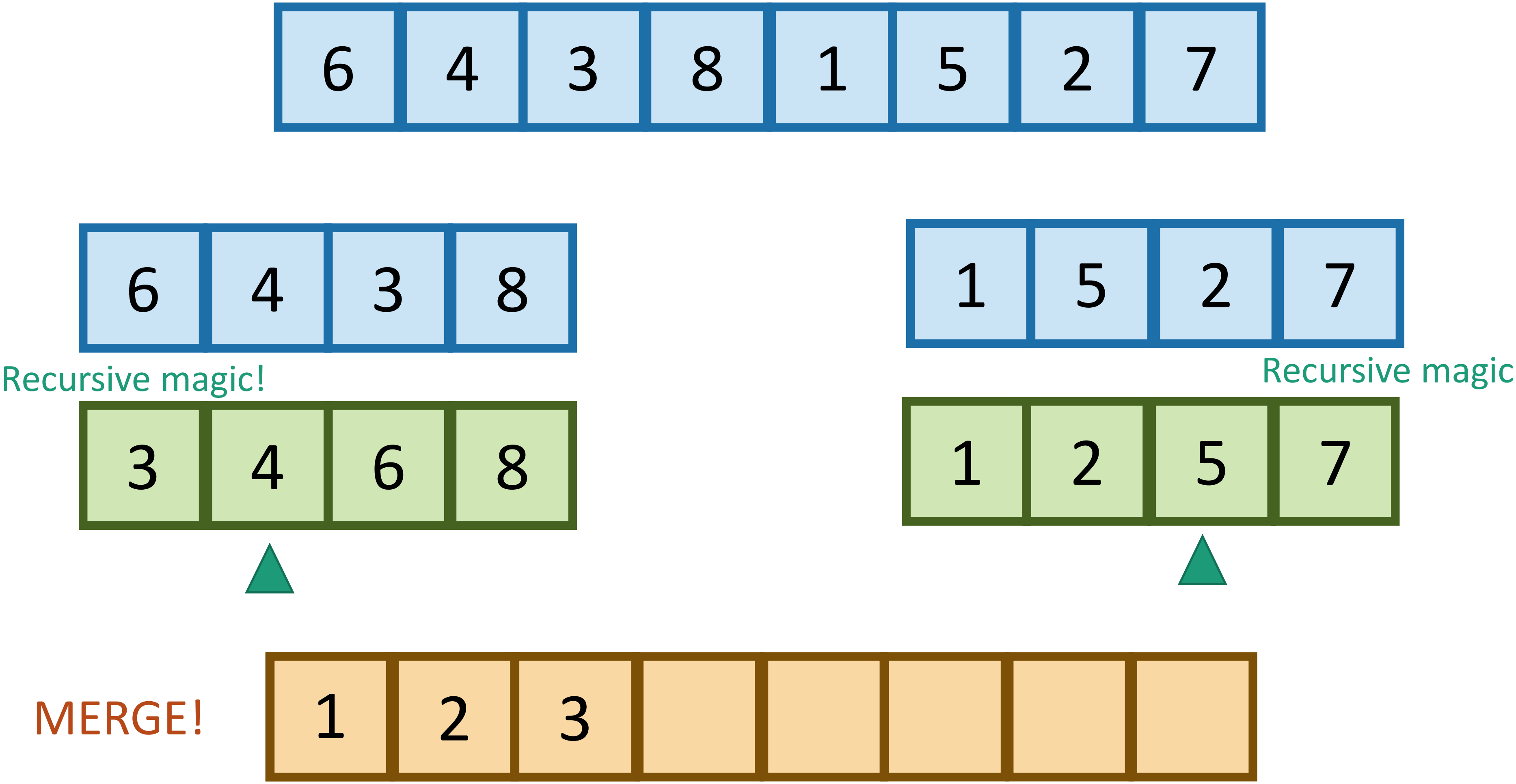
6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3

# MergeSort

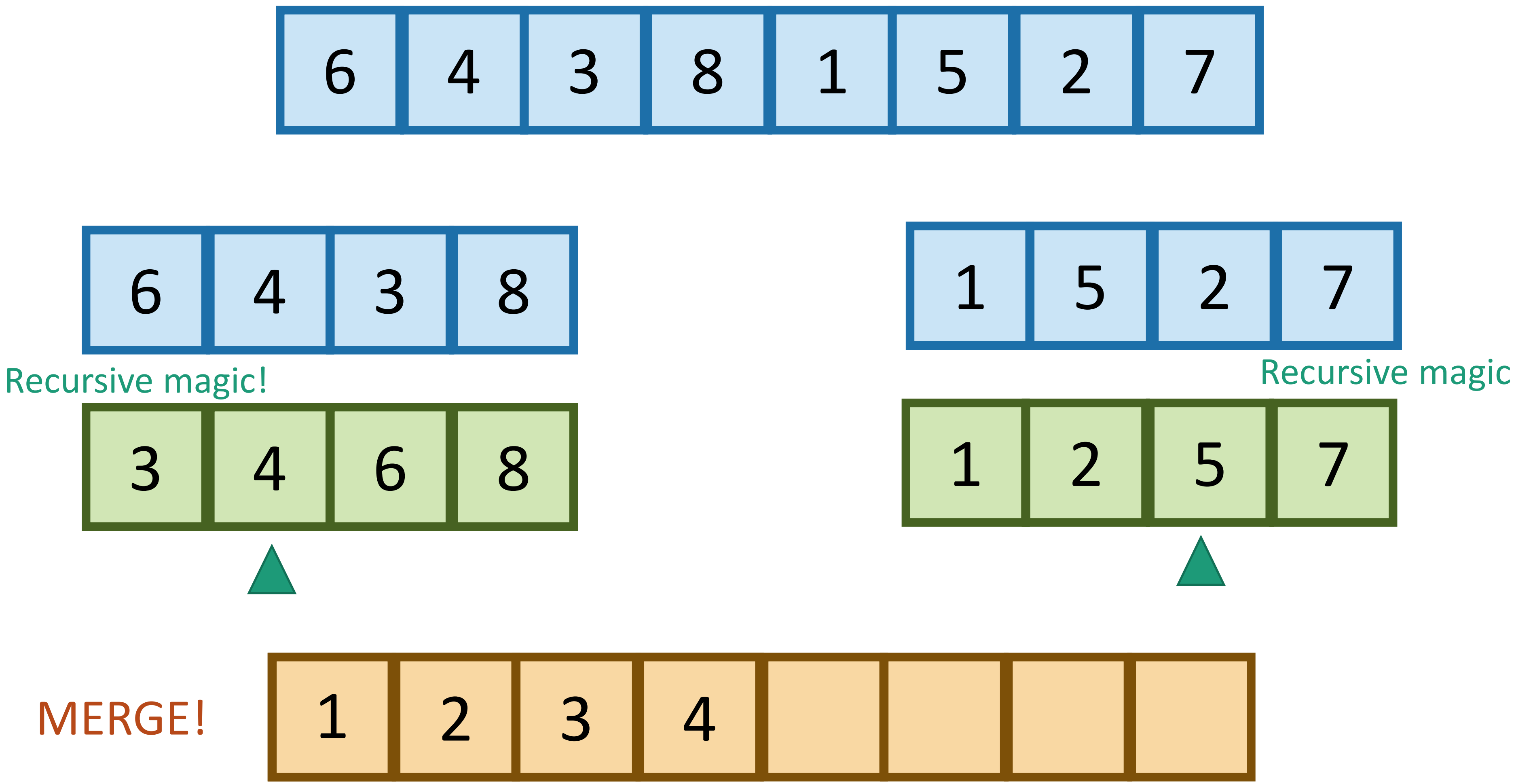


6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3

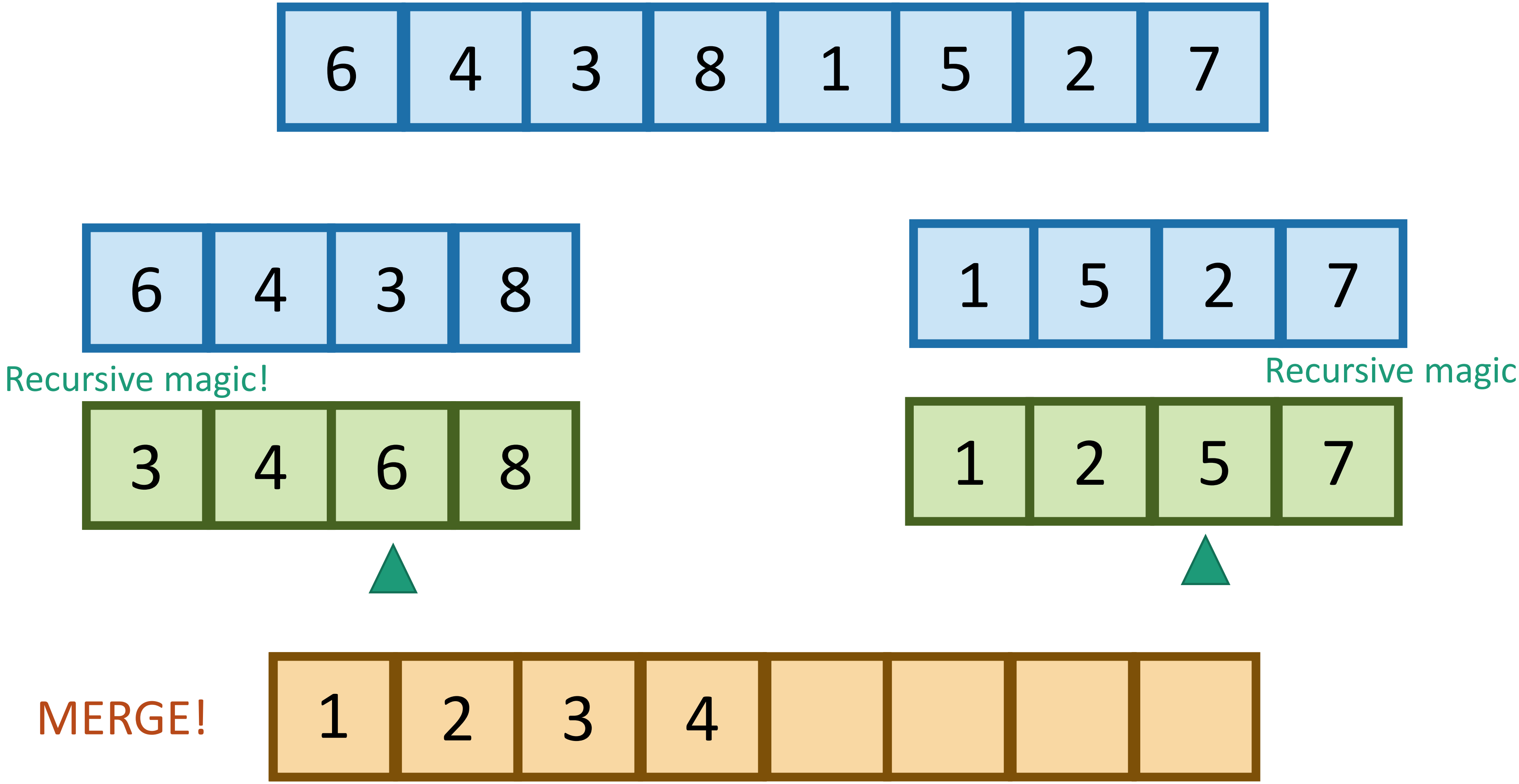
# MergeSort


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

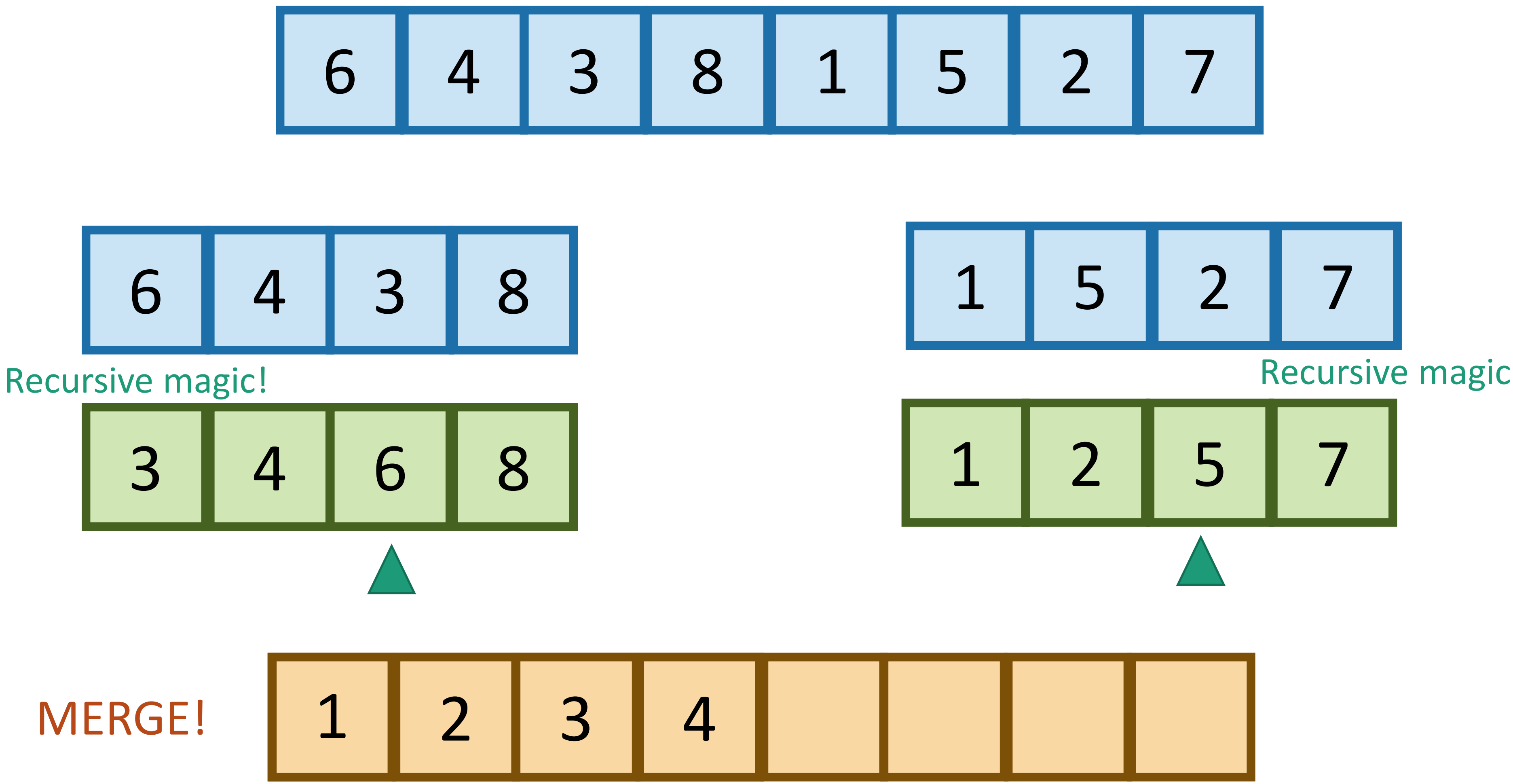
# MergeSort


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

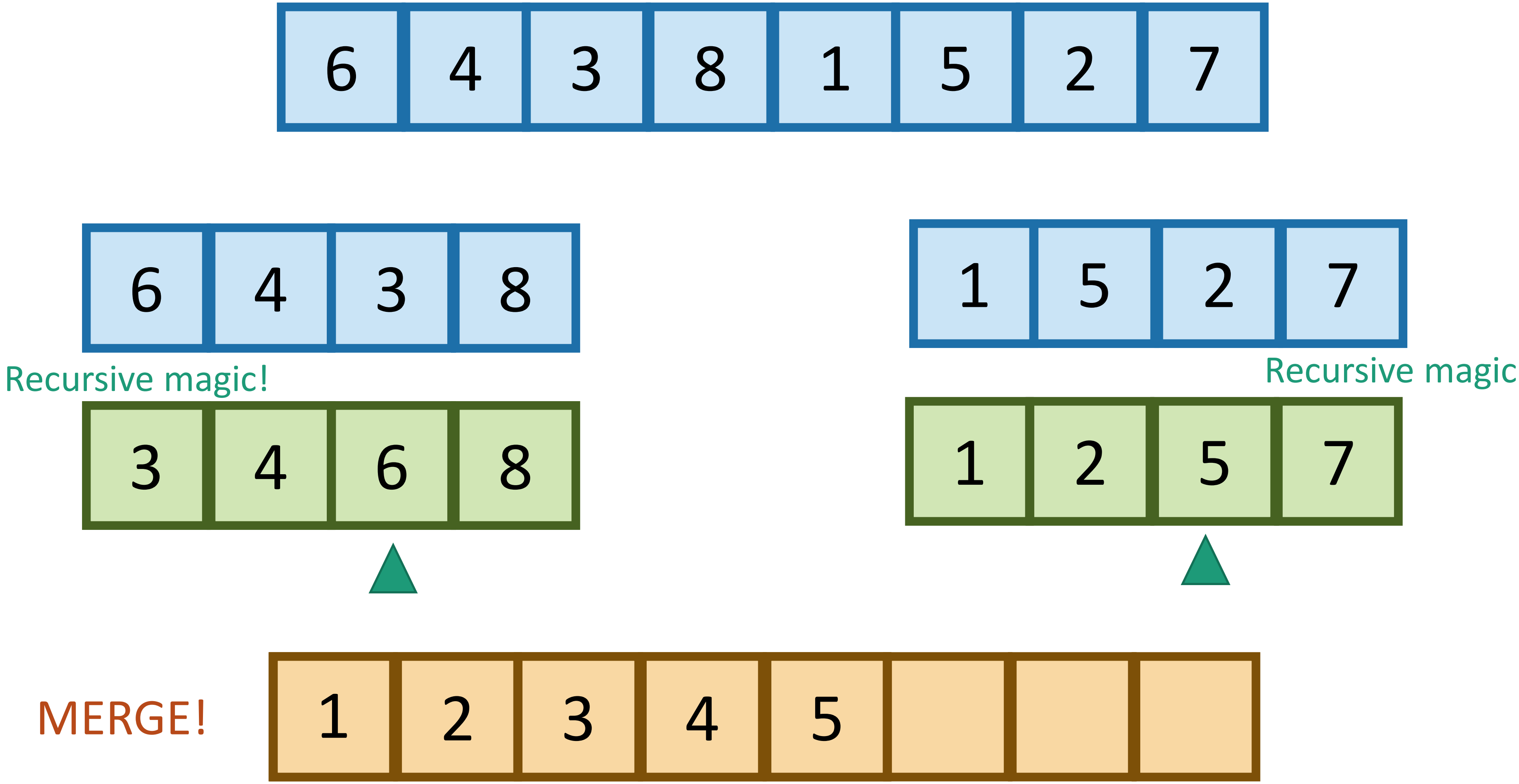
# MergeSort


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

# MergeSort

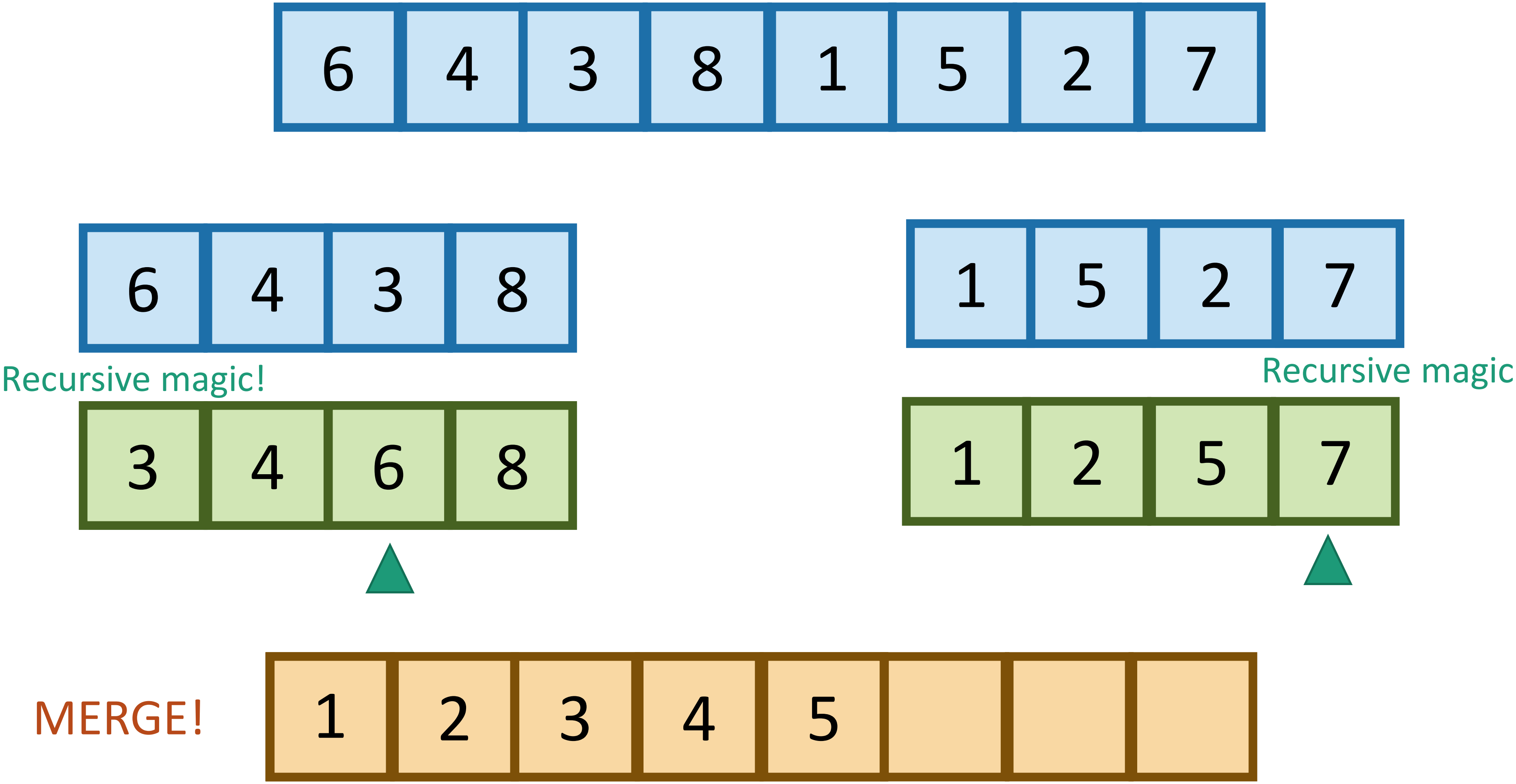

6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4

# MergeSort



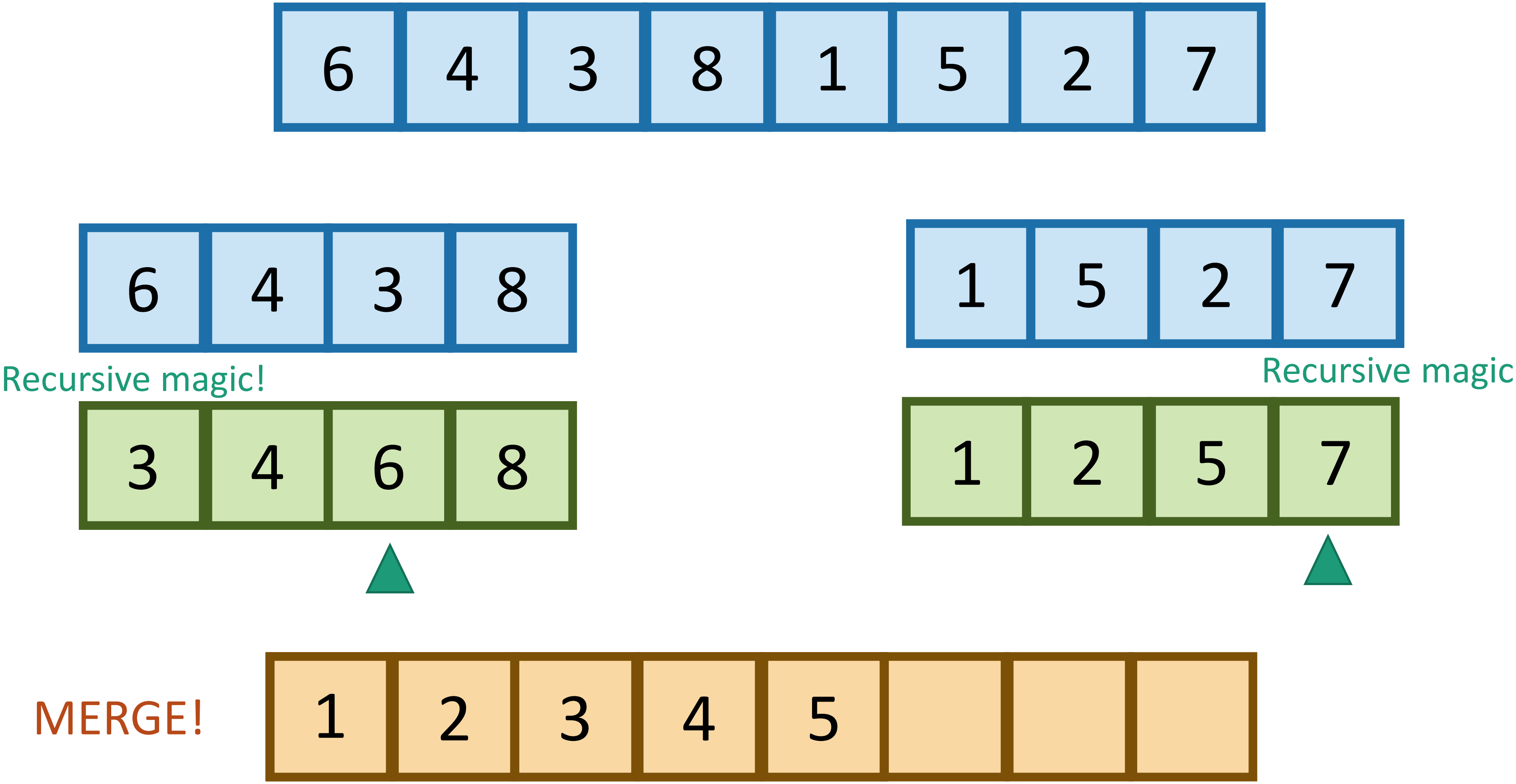
6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4 5

# MergeSort



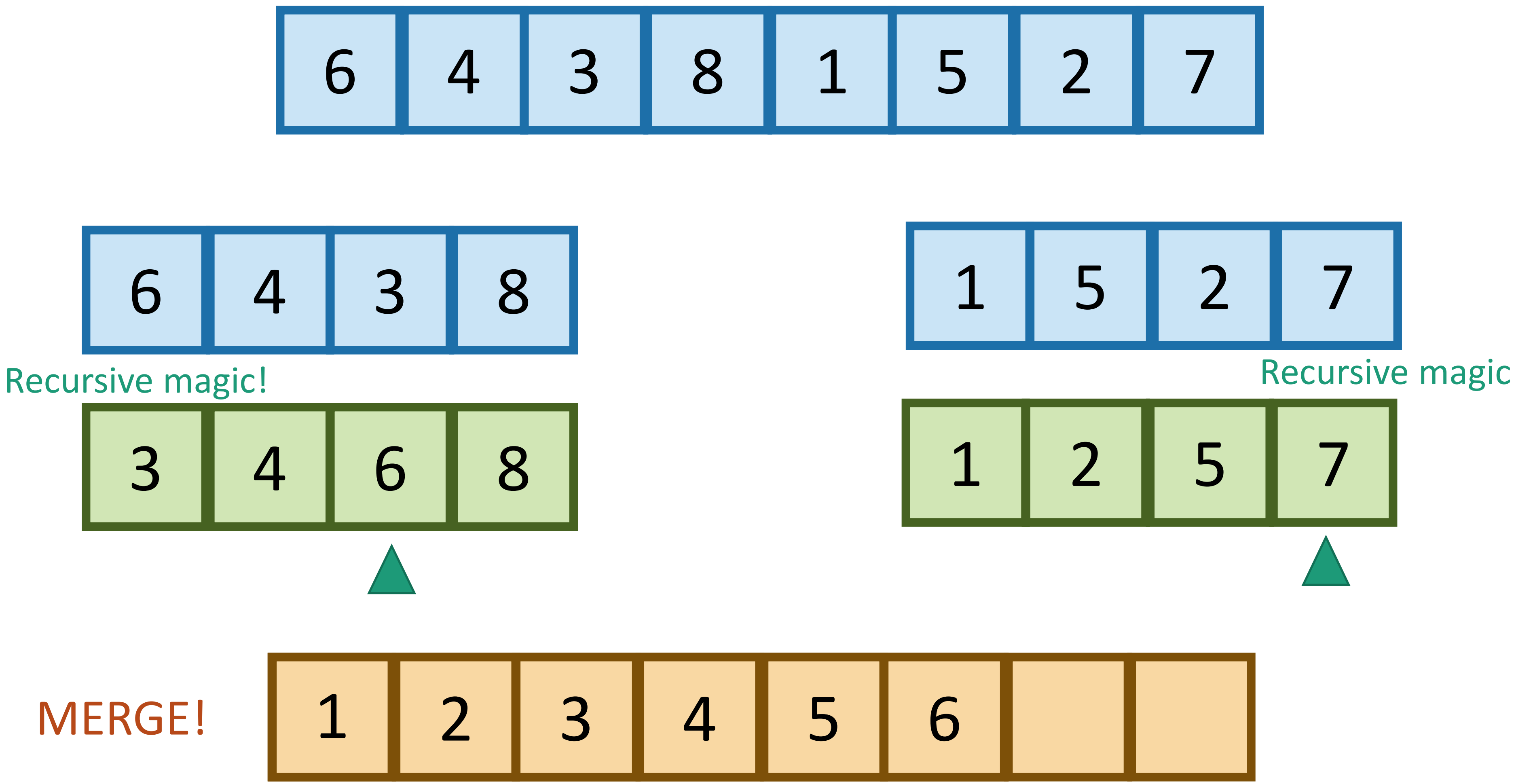
6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4 5

# MergeSort

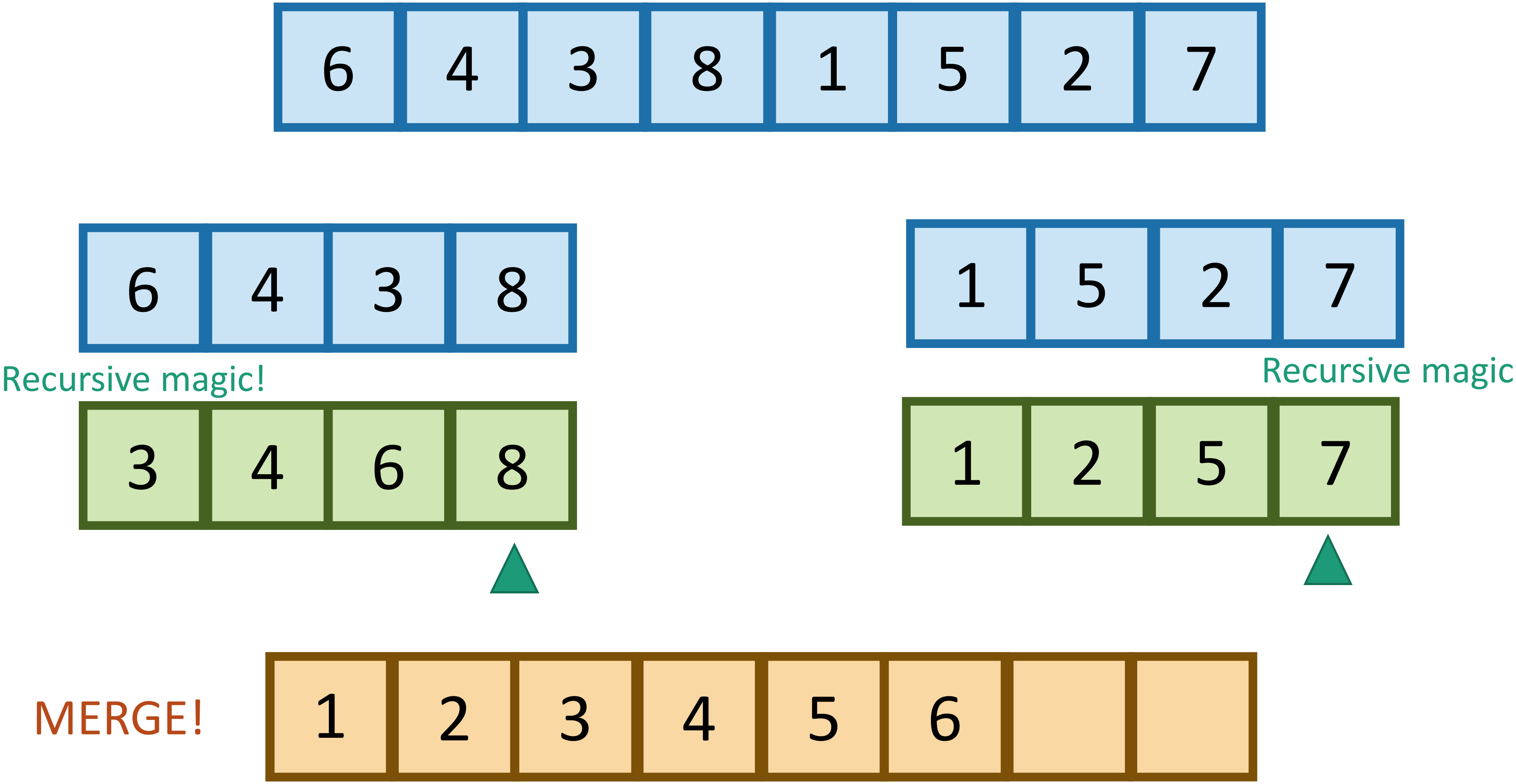


6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4 5

# MergeSort

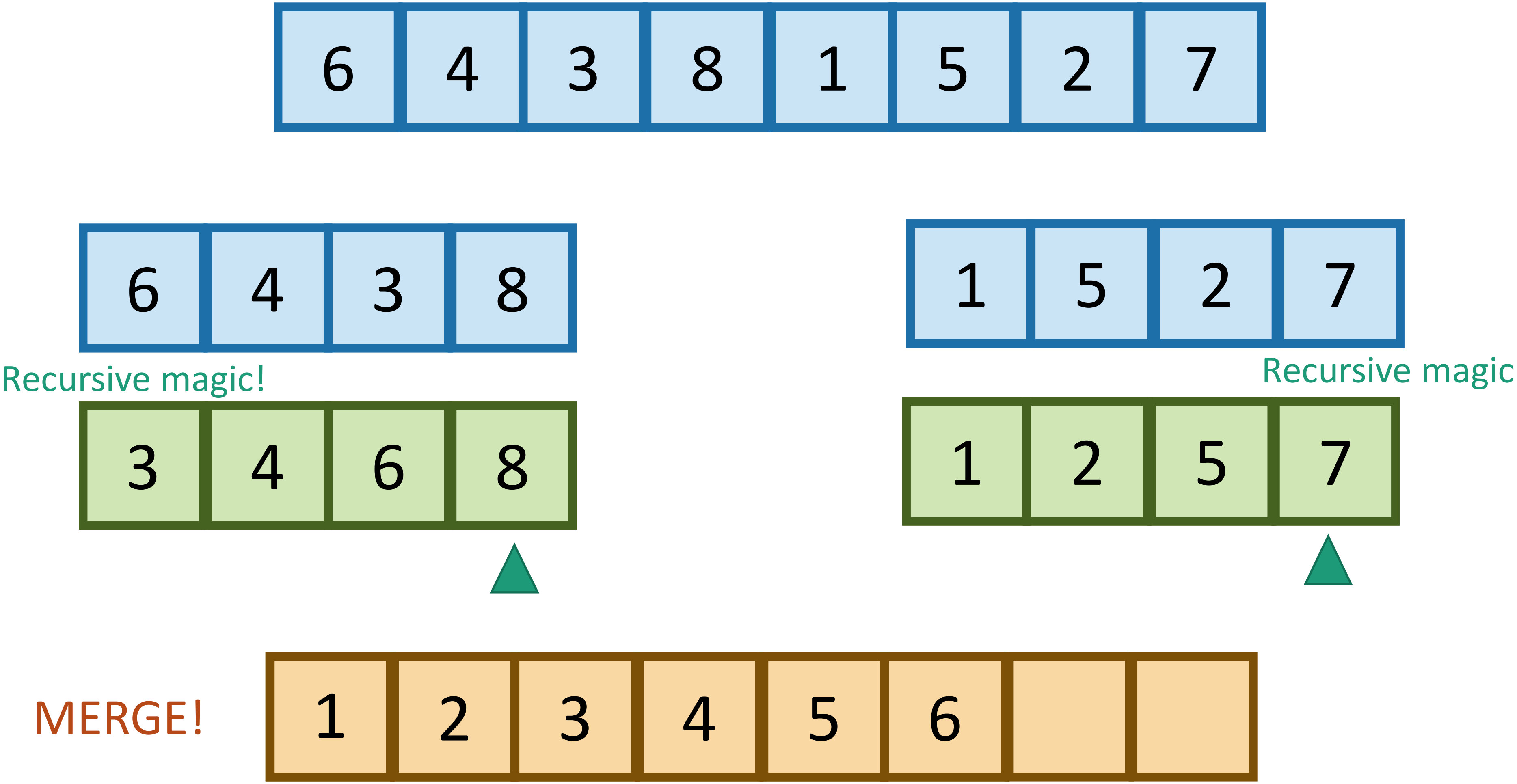

6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4 5 6

# MergeSort



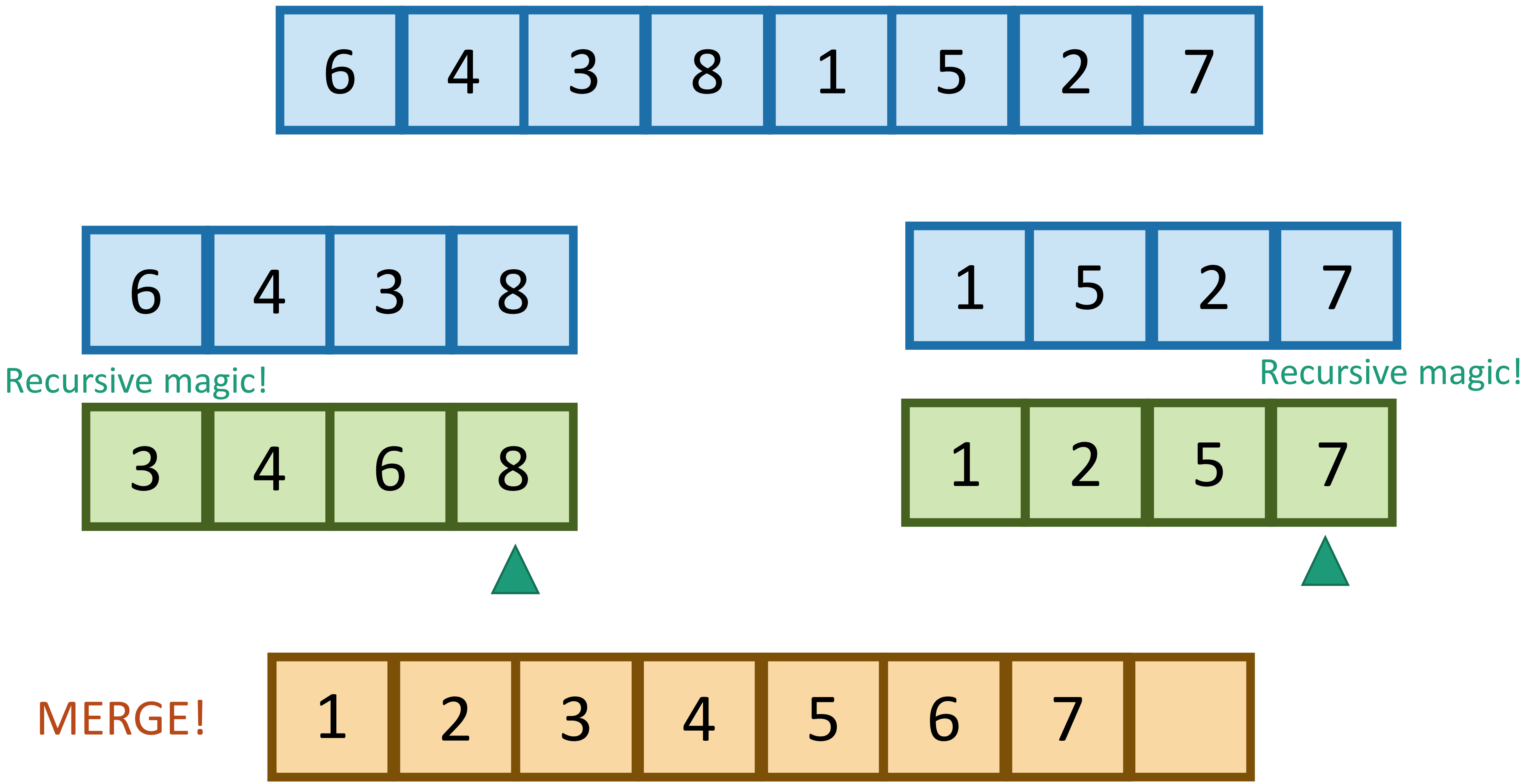
6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4 5 6

# MergeSort

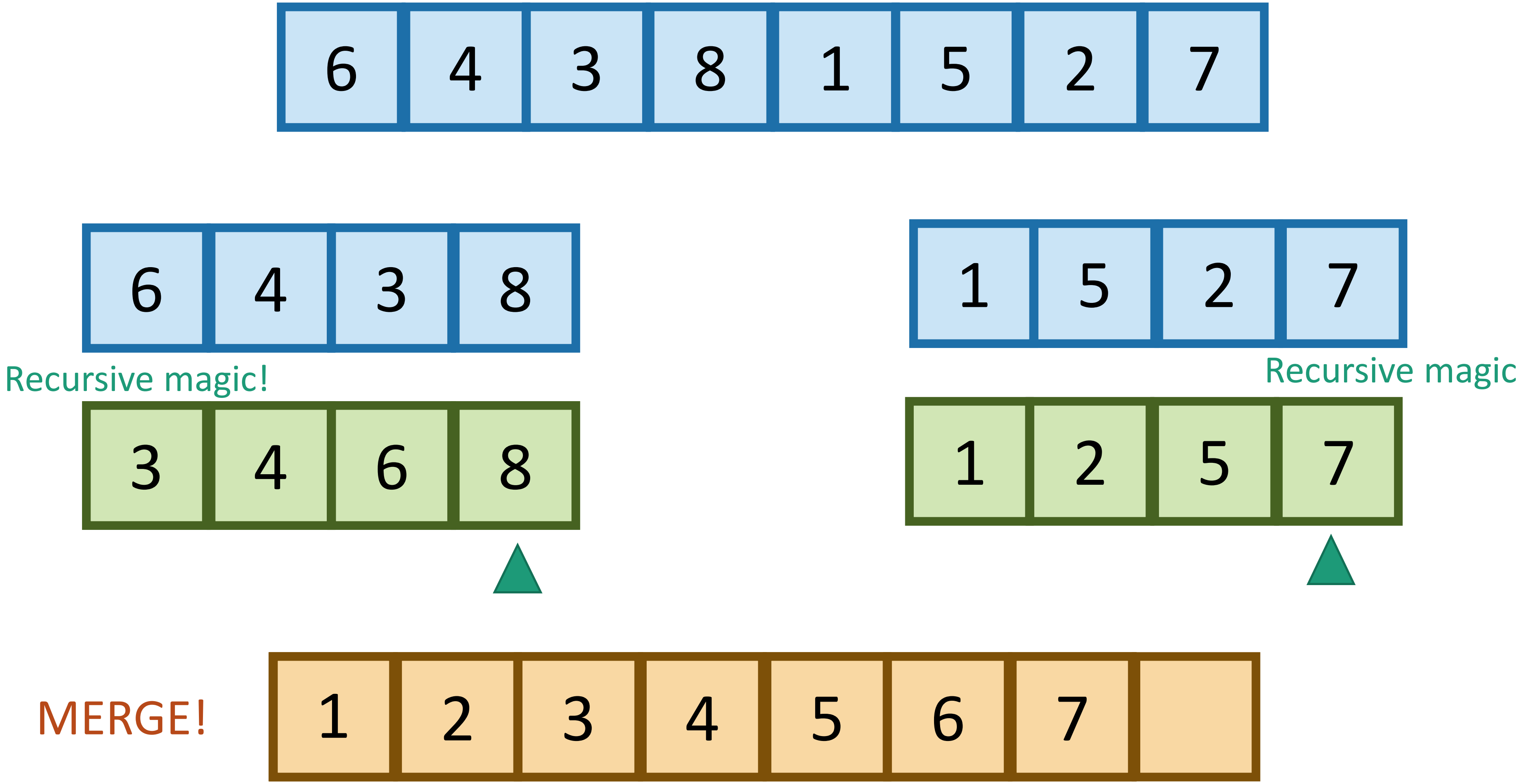


6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4 5 6

# MergeSort

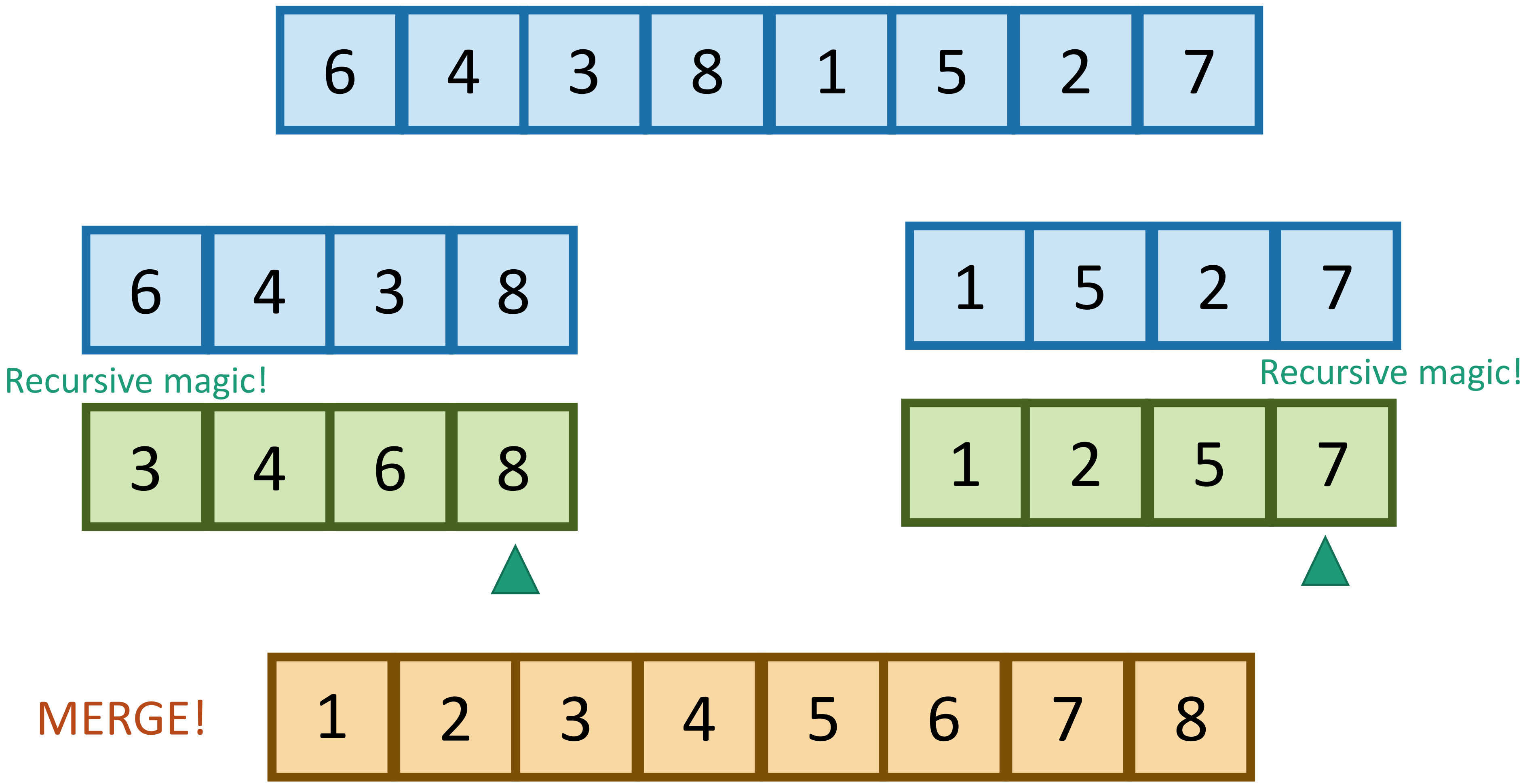

6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4 5 6 7

# MergeSort

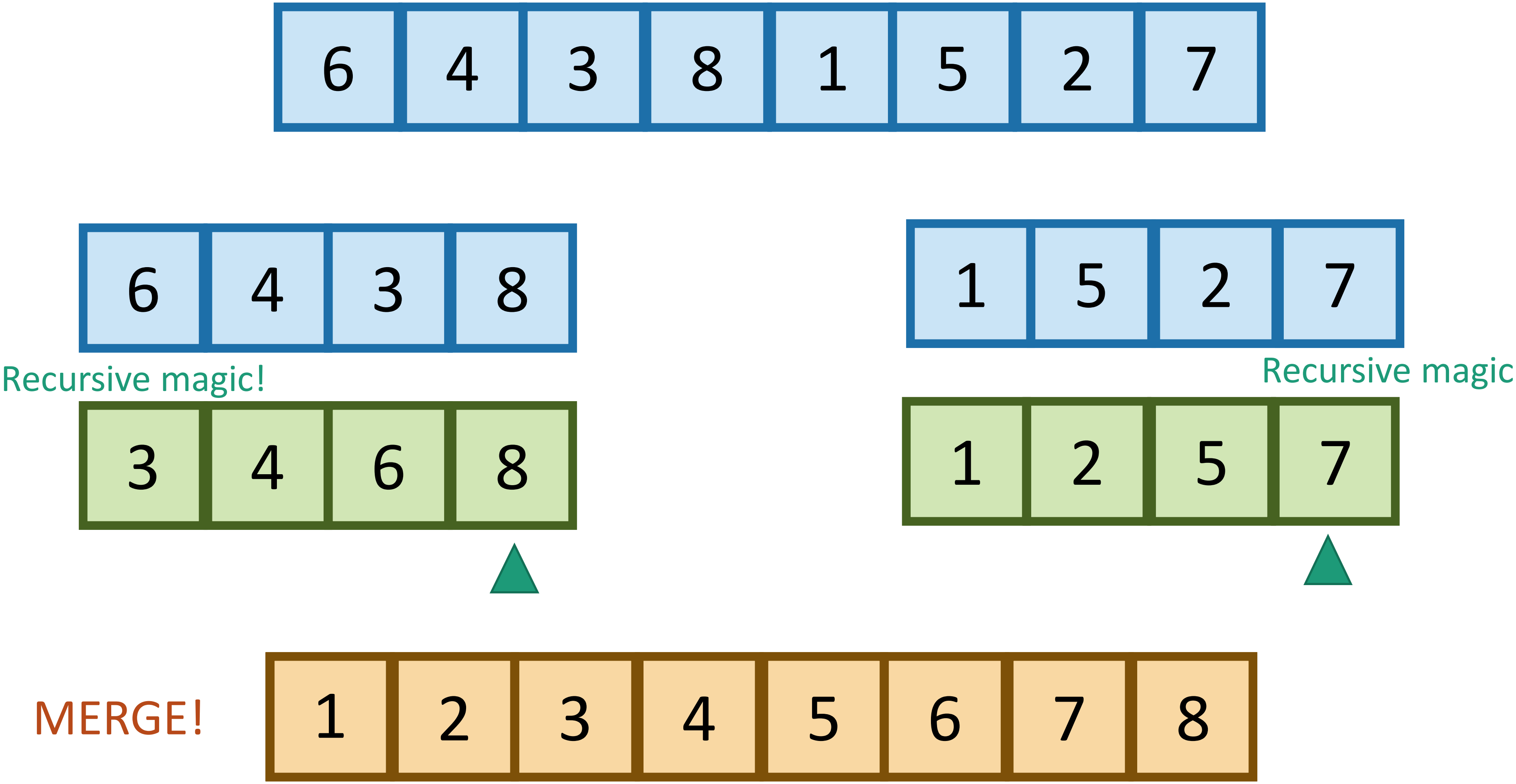


6 4 3 8 1 5 2 7
6 4 3 8
1 5 2 7
Recursive magic!
Recursive magic!
3 4 6 8
1 2 5 7
MERGE!
1 2 3 4 5 6 7

# MergeSort


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

# MergeSort




MERGE pseudocode + analysis on board, or CRLS

# MergeSort

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

| | | | |
|---|---|---|---|
| 6 | 4 | 3 | 8 |

Recursive magic!

| | | | |
|---|---|---|---|
| 3 | 4 | 6 | 8 |

| | | | |
|---|---|---|---|
| 1 | 5 | 2 | 7 |

Recursive magic!

| | | | |
|---|---|---|---|
| 1 | 2 | 5 | 7 |

MERGE!

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

How would you do this in-place?

Ollie the over-achieving Ostrich

MERGE pseudocode + analysis in CRLS (and notes)

# 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 ])

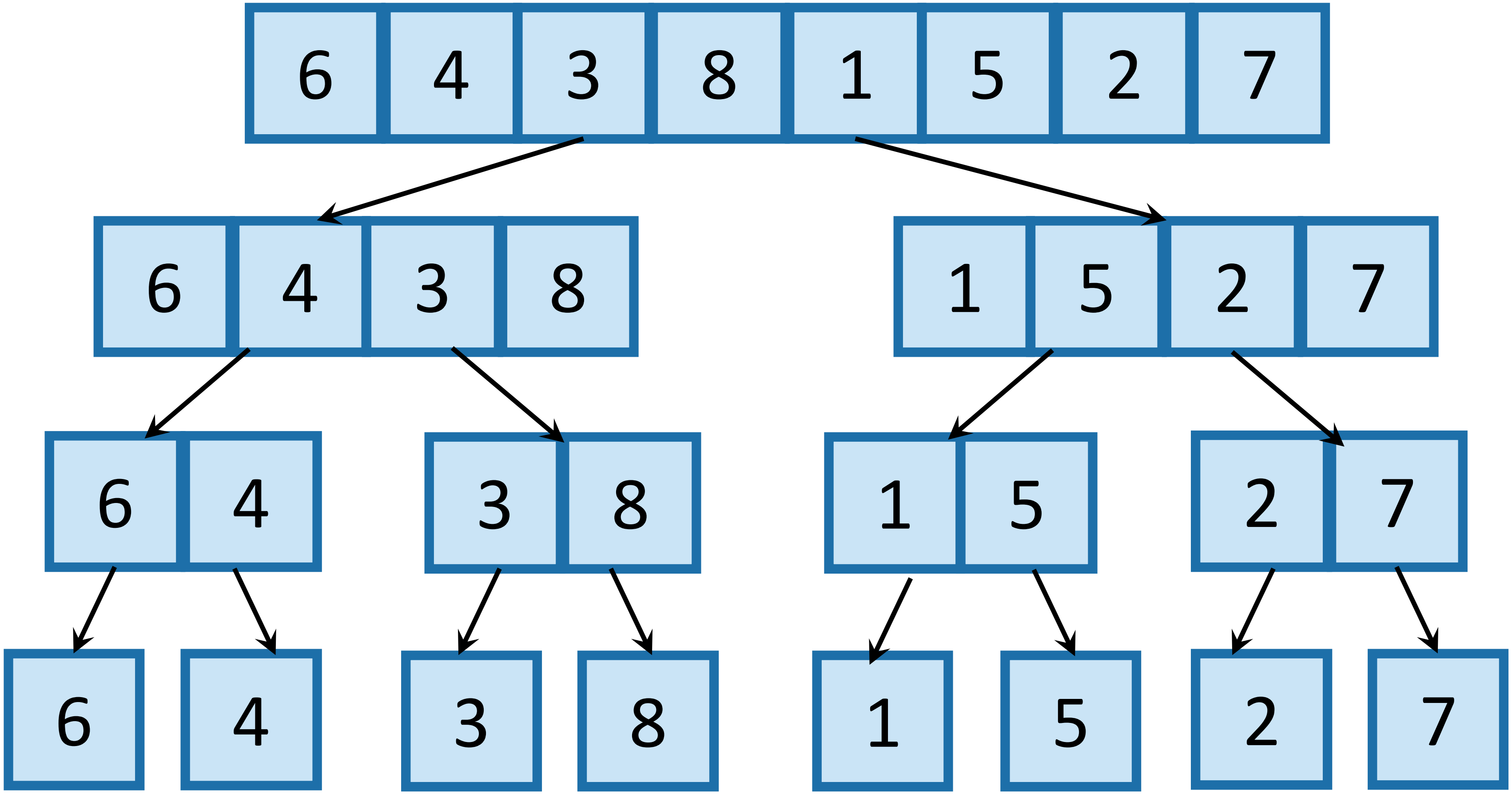
Sort the right half

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

Merge the two halves

# What actually happens?

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



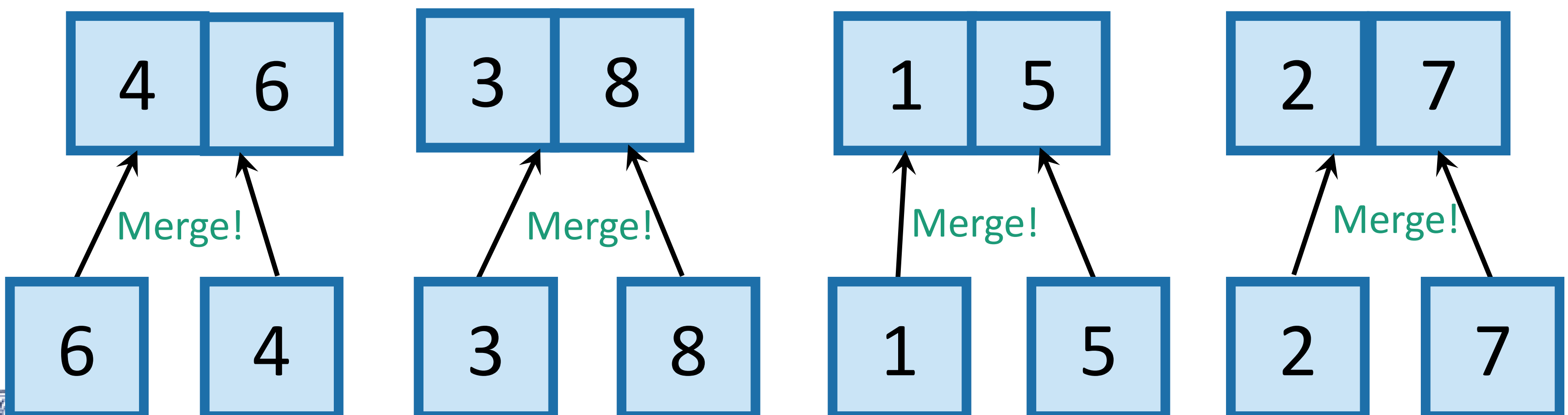

This array of length 1 is sorted!

# Then, merge them all back up!

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

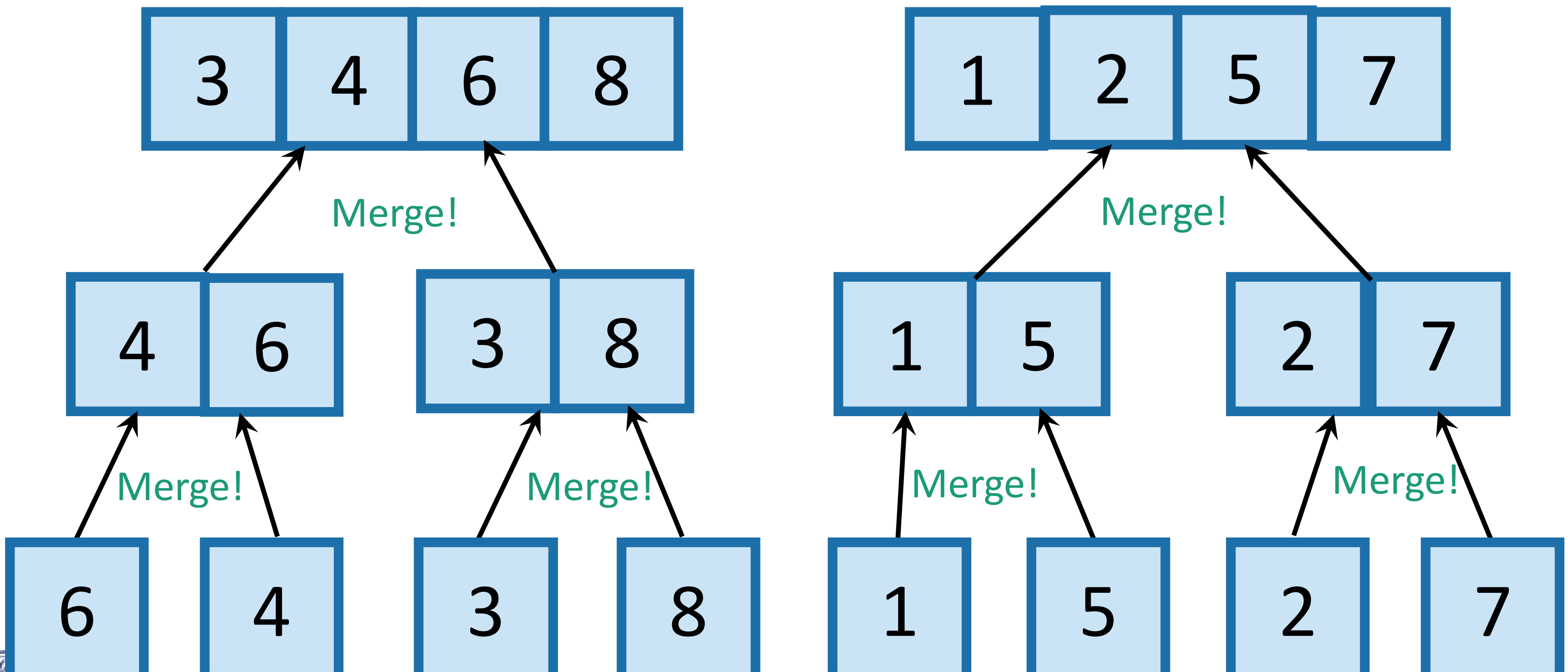
A bunch of sorted lists of length 1 (in the order of the original sequence).

# Then, merge them all back up!



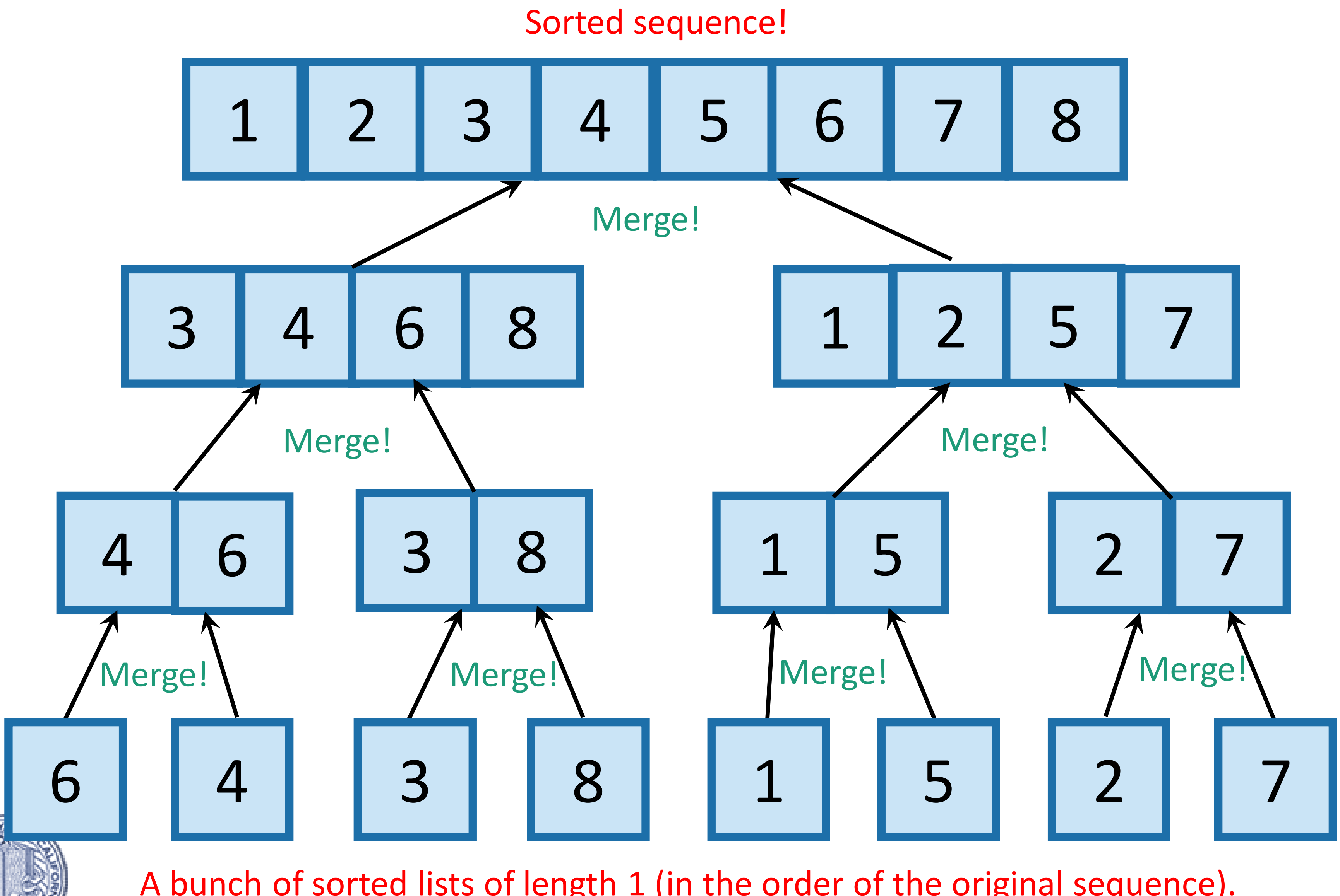

A bunch of sorted lists of length 1 (in the order of the original sequence).

# Then, merge them all back up!



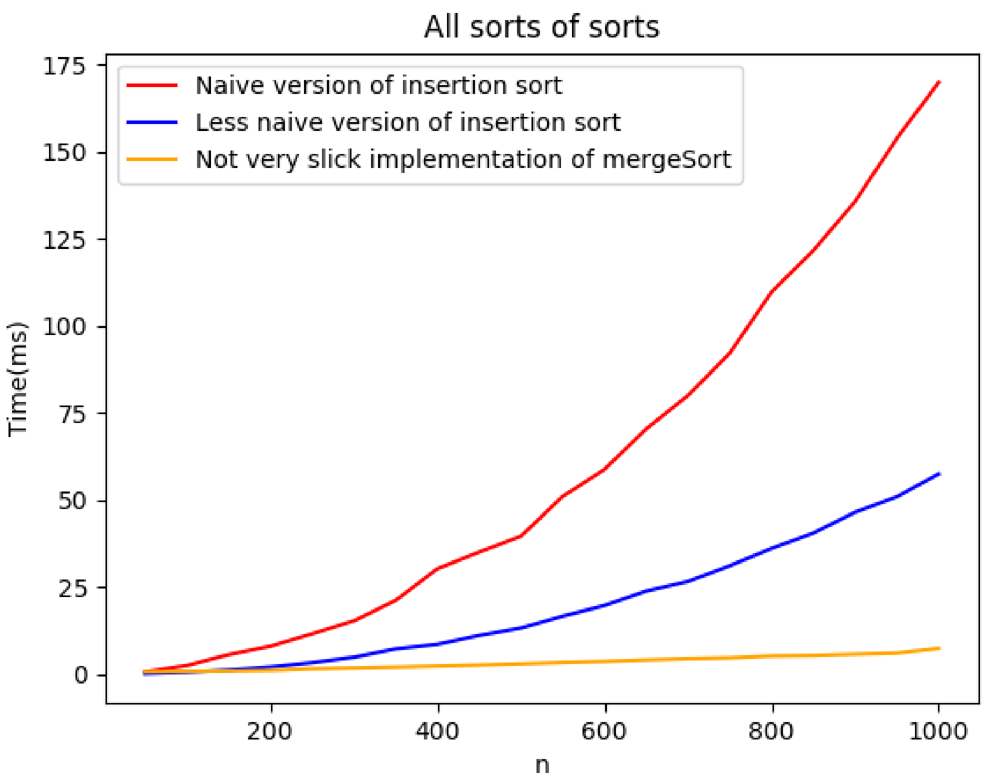

A bunch of sorted lists of length 1 (in the order of the original sequence).

# Then, merge them all back up!



# Two questions

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




Empirically:

1. Seems to work.
2. Seems fast.

# It works

Assume that n is a power of 2 for convenience.

- **Inductive hypothesis:**

"In every the recursive call on an array of length at most i, MERGESORT returns a sorted array."

Not technically a "loop invariant," but a "recursion invariant," that should hold at the beginning of every recursive call.

- **Base case (i=1):** a 1-element array is always sorted.
- **Inductive step:** Need to show: If L and R are sorted, then MERGE(L,R) is sorted.
- **Conclusion:** In the top recursive call, MERGESORT returns a sorted array.

Fill in the inductive step! (Either do it yourself or read it in CLRS Section 2.3.1!)

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

# Today

- Integer Multiplication (wrap up)
- Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?**... (to be cont.)**
- 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