# Two Techniques:

## The Potential Method

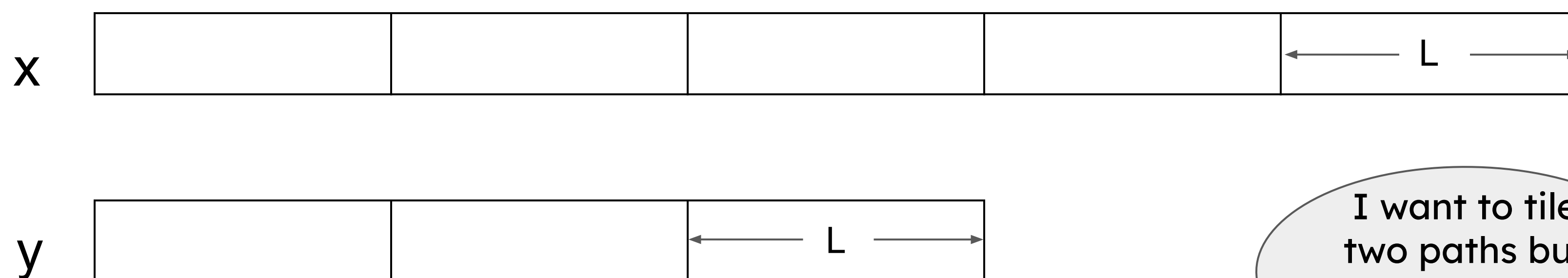

## Divide and Conquer

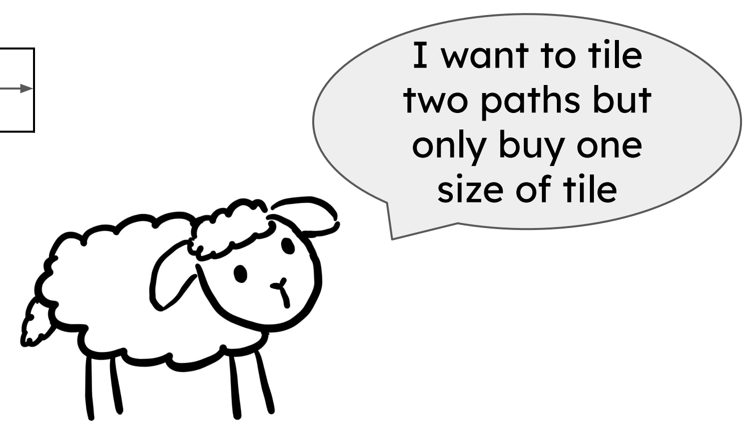
Last time:

# The Tiling problem (aka gcd)

**Input:** n-bit integers $x \geq y \geq 0$, but not both =0.

**Output:** largest integer L that divides both x and y (aka greatest common divisor)
***In other words:*** largest integer tile size that can exactly tile a path of length x and a path of length y



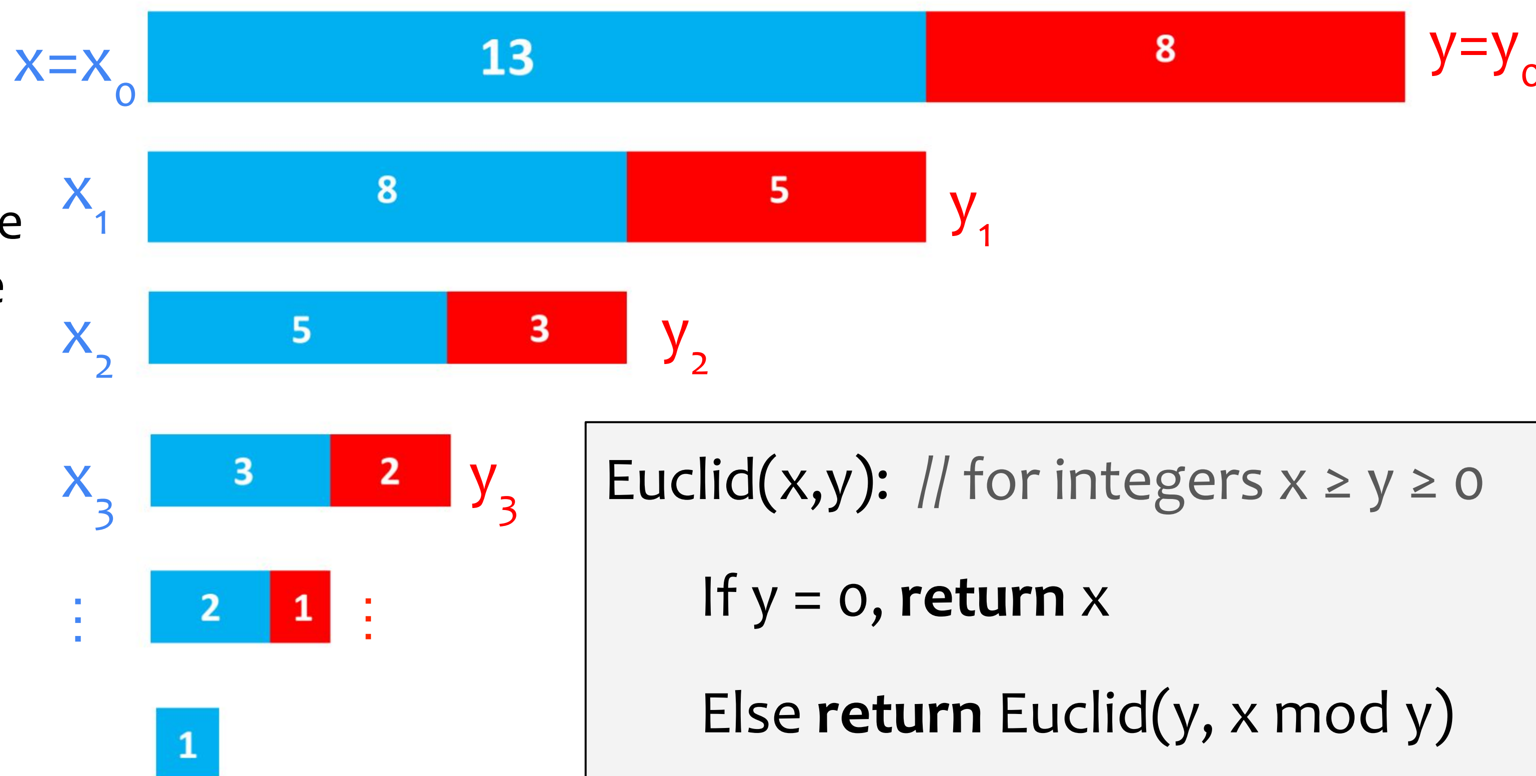
# Last time: Euclid's Algorithm (in pseudocode)

```
Euclid(x,y):  // for integers x ≥ y ≥ 0
    If y = 0, return x  // Base case
    Else return Euclid(y, x mod y)  // Recursive case
```

**Last time:** We discussed why Euclid's algorithm is **correct.**
**This time:** We will analyze the **running time** of Euclid's algorithm.

# An execution of Euclid's algorithm

Successive recursive calls ↓




Euclid(x,y): // for integers $x \geq y \geq 0$

If $y = 0$, **return** x

Else **return** Euclid(y, x mod y)

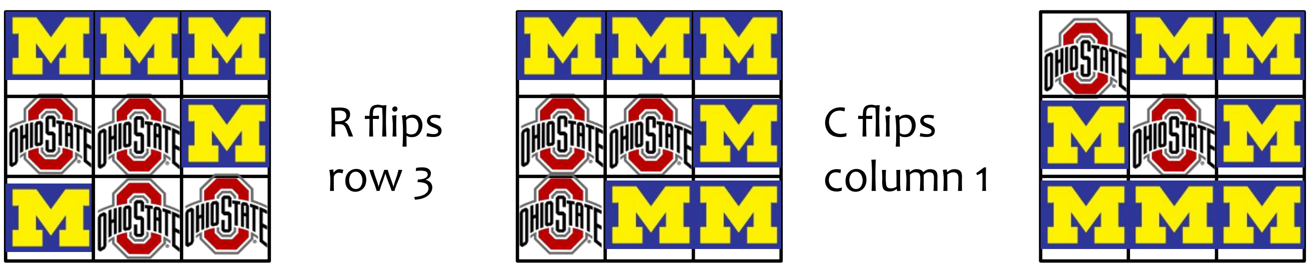
# The Potential Method

Today we will analyze the running time of Euclid's algorithm using the **potential method**.

… But first, a toy example to illustrate this method

# A Flipping Game

- 3 x 3 board covered with two-sided chips: /
- Two players, **R (row)** and **C (column)**, alternately perform "flips":
  - **R** flips every chip in a **row** with # > #
  - **C** flips every chip in a **column** with # > #
- If no flip is possible, then the game ends.
- **Question:** Must the game always end?



# Let’s formalize this reasoning into a general-purpose method

Intuitively, a **potential function argument** says:
If I start with a finite amount of water in a leaky bucket, then eventually water must stop leaking out.

**Ingredients of the argument:**

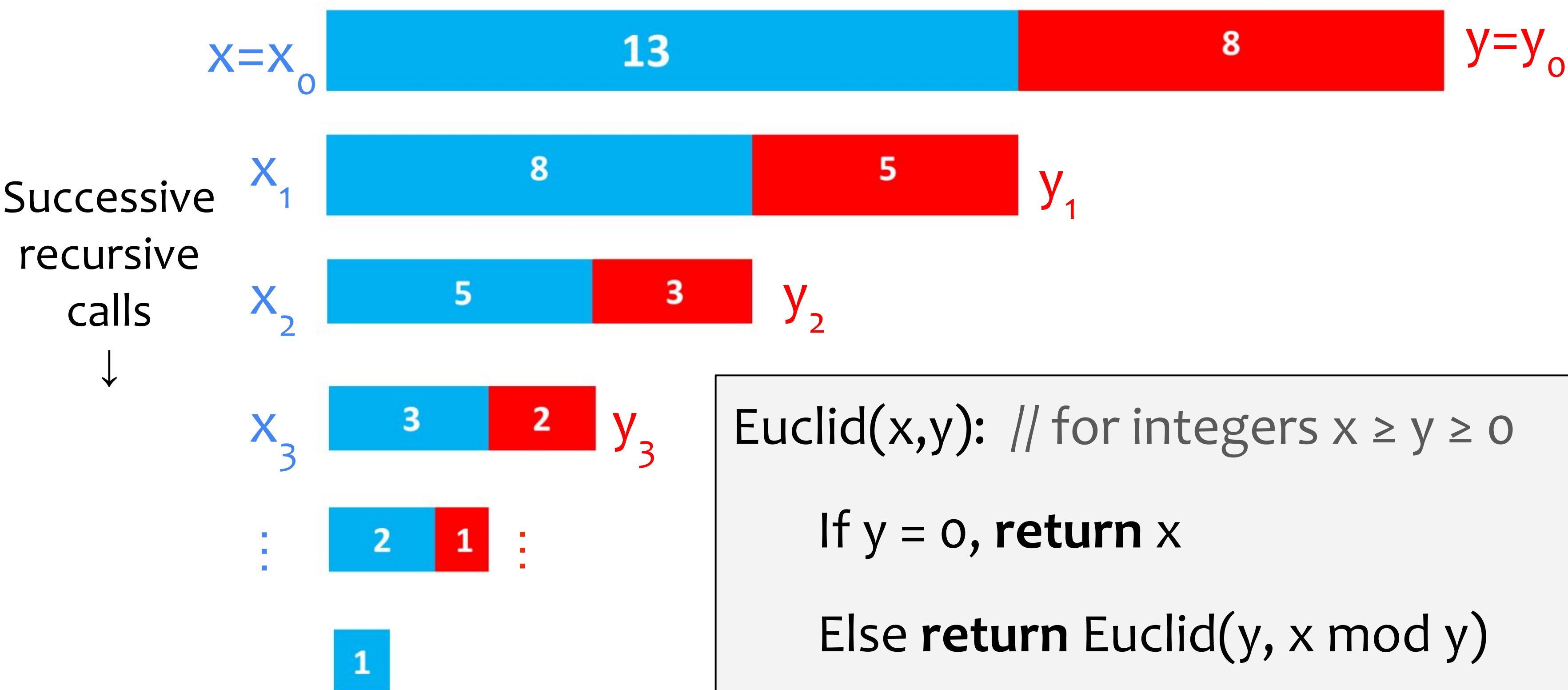
1. Define the unit of time e.g. one iteration of an algorithm
2. Define how we measure the amount of water in the bucket. This is the **potential function $S_i$** ⟵ amount of water in bucket at timestep i
3. Prove that the $S_i$ can never be negative
4. Prove that the bucket is leaking quickly. I.e. show that each timestep i, the value of **S** decreases by at least some amount.
5. Use this to upper bound on the total number of units of time.

# Analyzing the Flipping Game via a Potential Function

1. Unit of time = one player's turn.
2. Define the **potential function $S_i$** = # chips at turn i.
3. Note that **$S_i$** can never be negative.
4. At every turn the value of **$S_i$** decreases by at least **1**.
5. This implies that the total number of turns is at most **$S_0$**, which is at most 9.

Now let's apply the potential method to Euclid's Algorithm…

# An execution of Euclid's algorithm




Euclid(x,y): // for integers $x \geq y \geq 0$

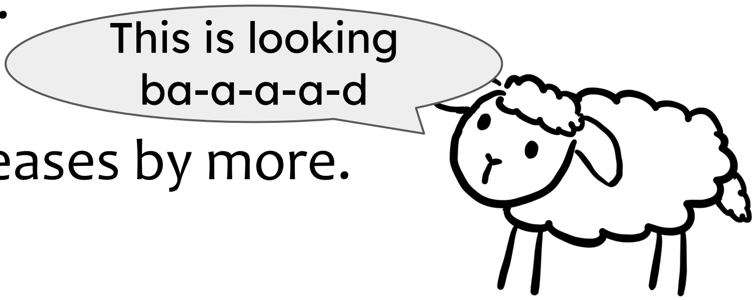
If $y = 0$, **return** x

Else **return** Euclid(y, x mod y)

# Analyzing Euclid's Algorithm via a Potential Function

1. Unit of time = one recursive call.
2. Define the **potential function $S_i = y_i$.**
3. Note that $S_i$ can never be negative.
4. At every recursive call the value of **S** decreases by at least **1**.
5. Thus, the total number of calls to Euclid is at most $S_0 = y$.

**But we already knew this!** Recall that the brute-force algorithm from last lecture already achieved y calls to Euclid.




**Conclusion:** We need a function **S** that decreases by more.

# Let's convince ourselves that the potential functions $S_i = y_i$ and $S_i = x_i$ are both doomed

**Why $S_i = y_i$ is doomed:** What is an example of x,y values such that **$S_i$ only decreases by 1**, i.e. $y - y_1 = 1$? (and $x,y \geq 4$)

**Why $S_i = x_i$ is doomed:** What is an example of x,y values such that **$S_i$ only decreases by 1**, i.e. $x - x_1 = 1$? (and $x,y \geq 4$)

# Analyzing Euclid's Algorithm via a Potential Function

Finding the right potential function can be a fine art.

It turns out that even though neither $S_i = y_i$ nor $S_i = x_i$ work, **$S_i = x_i+y_i$** does!

1. Unit of time = one recursive call.
2. Define the **potential function $S_i = x_i+y_i$.**
3. Note that **$S_i$** can never be negative.
4. **Claim 1.** At every recursive call the value of **S** decreases by at least a **multiplicative factor**, specifically **$S_{i+1} \le (2/3) \cdot S_i$** for all i (need to prove)
5. **Claim 2.** Claim 1 implies: total # recursive calls is O(log (x+y)) = O(n). (need to prove)

Consequence of Claim 2: **final running time is poly(n)**, since *x mod y* for n-bit numbers can be computed in poly(n) time (by grade-school algorithm)

# Analyzing Euclid's Algorithm via a Potential Function

**Claim 1.** $S_{i+1} \le (2/3) \cdot S_i$ (equivalently, $S_i \ge (3/2) \cdot S_{i+1}$) for all i.

Proof. Goal: Show $x_i + y_i \ge (3/2) \cdot (x_{i+1} + y_{i+1})$ i.e. $x_i + y_i \ge (3/2) \cdot (y_i + x_i \bmod y_i)$.

Express $x_i$ as: $x_i = q_i \cdot y_i + r_i$.

$$
\begin{aligned}
\text{So } x_i + y_i &= q_i \cdot y_i + r_i + y_i \\
&= (q_i + 1) \cdot y_i + r_i \\
&\ge 2y_i + r_i \\
&\ge 2y_i + r_i - (y_i - r_i)/2 \\
&= (3/2) \cdot (y_i + r_i) \\
&= (3/2) \cdot (y_i + x_i \bmod y_i).
\end{aligned}
$$

# Analyzing Euclid's Algorithm via a Potential Function

**Claim 2:** If $S_{i+1} \leq (2/3) \cdot S_i$ for all i, then total # recursive calls is O(log (x+y)).

Proof. Observe $S_i \geq 1$.

$$\text{So, } 1 \leq (2/3)^i \cdot (x+y)$$

$$(3/2)^i \leq (x+y)$$

$$i \leq \log_{3/2}(x+y).$$

$S_0 = x+y$,
$S_1 \leq (2/3) \cdot (x+y)$,
$S_2 \leq (2/3)^2 \cdot (x+y)$,
…
$S_i \leq (2/3)^i \cdot (x+y)$

# Analyzing Euclid's Algorithm via a Potential Function

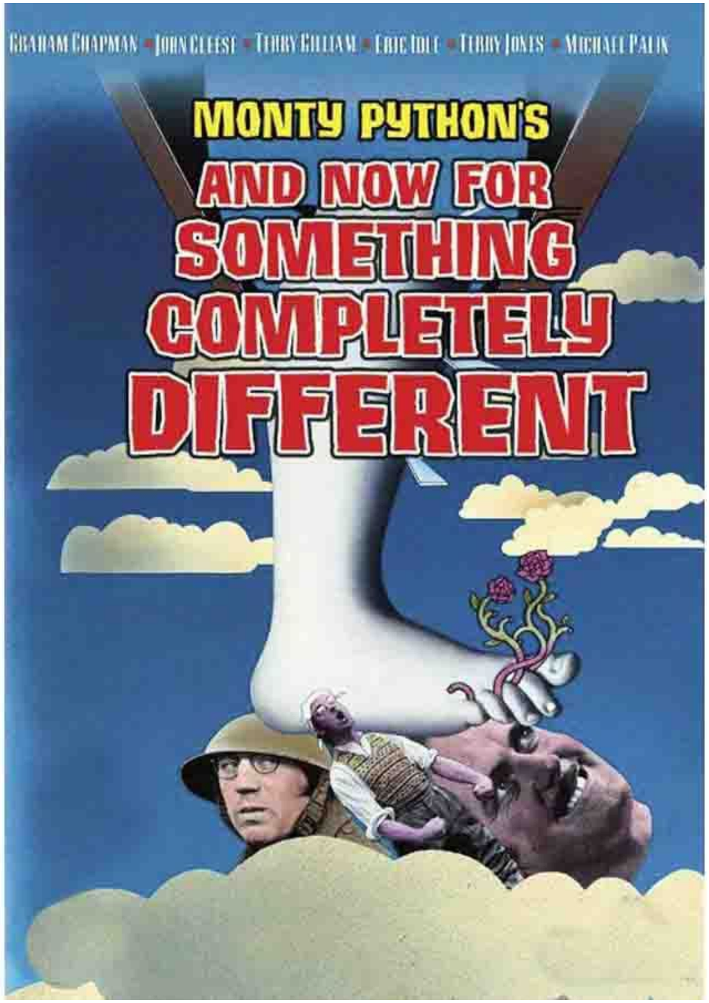
1. Unit of time = one recursive call.
2. Define the **potential function $S_i = x_i + y_i$.**
3. Note that $S_i$ can never be negative.
4. **Claim 1.** At every recursive call the value of **$S$** decreases by at least a **multiplicative factor**, specifically $S_i \geq (3/2) \cdot S_{i+1}$ for all i (need to prove)
5. **Claim 2.** Claim 1 implies: total # recursive calls is O(log (x+y)) = O(n). (need to prove)

Consequence of Claim 2: **final running time is poly(n)**, since *x mod y* for n-bit numbers can be computed in poly(n) time (by grade-school algorithm)

# When to use the potential method

Part of the challenge (and fun) of algorithm design is figuring out when to use which technique.

General intuition: The potential method could be useful when some quantity seems to be monotonically increasing or decreasing over the execution of the algorithm, getting you closer and closer to termination.




# A Design Technique: Divide and Conquer

# Overview: Divide-and-Conquer Algorithms

**Main Idea:**

1. **Divide** the input into smaller sub-problems
2. **Conquer** (solve) each sub-problem recursively
3. Combine the solutions to the subproblems

**Designing the Algorithm + Proving Correctness: an "art"**

- Depends on problem structure, ad-hoc, creative

**Running time Analysis: "mechanical"**

- Express runtime using a recurrence
- Can often solve using the "Master Theorem"

https://www.youtube.com/watch?v=ZRPoEKHXTJg

# Mergesort

**Input:** Array of numbers

| 1 | 2 | 3 | 4 | | | | | n |
|---|---|---|---|---|---|---|---|---|
| 6 | 0 | 4 | 6 | … | … | … | … | 3 |

**Output:** Sorted

| 1 | 2 | 3 | 4 | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 3 | 4 | 6 | 6 | … | … | … | … |

Discovered by John von Neumann in 1945

# Mergesort

# Mergesort

Unsorted array of length n

Unsorted array of length n/2

Unsorted array of length n/2

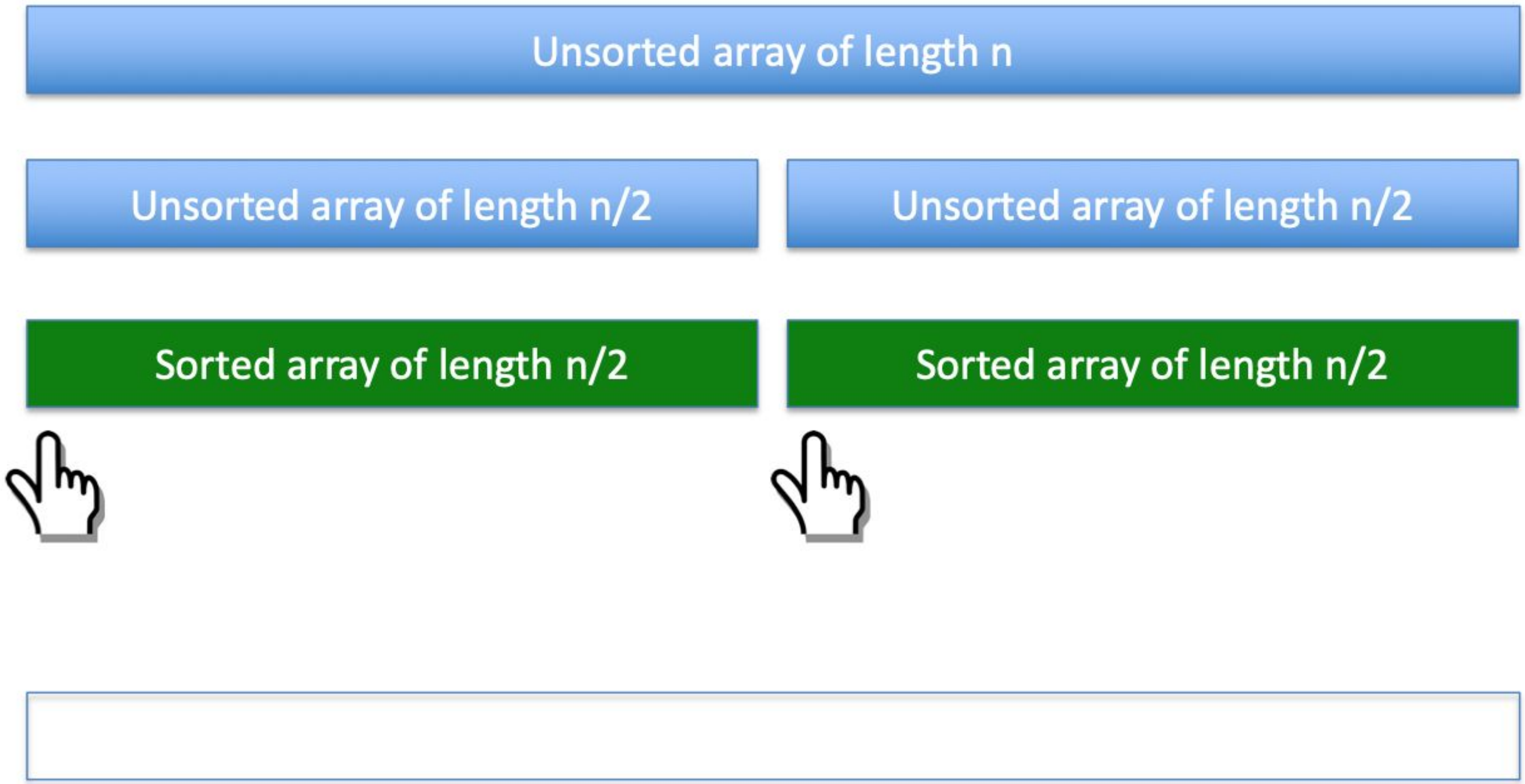
# Mergesort

Unsorted array of length n

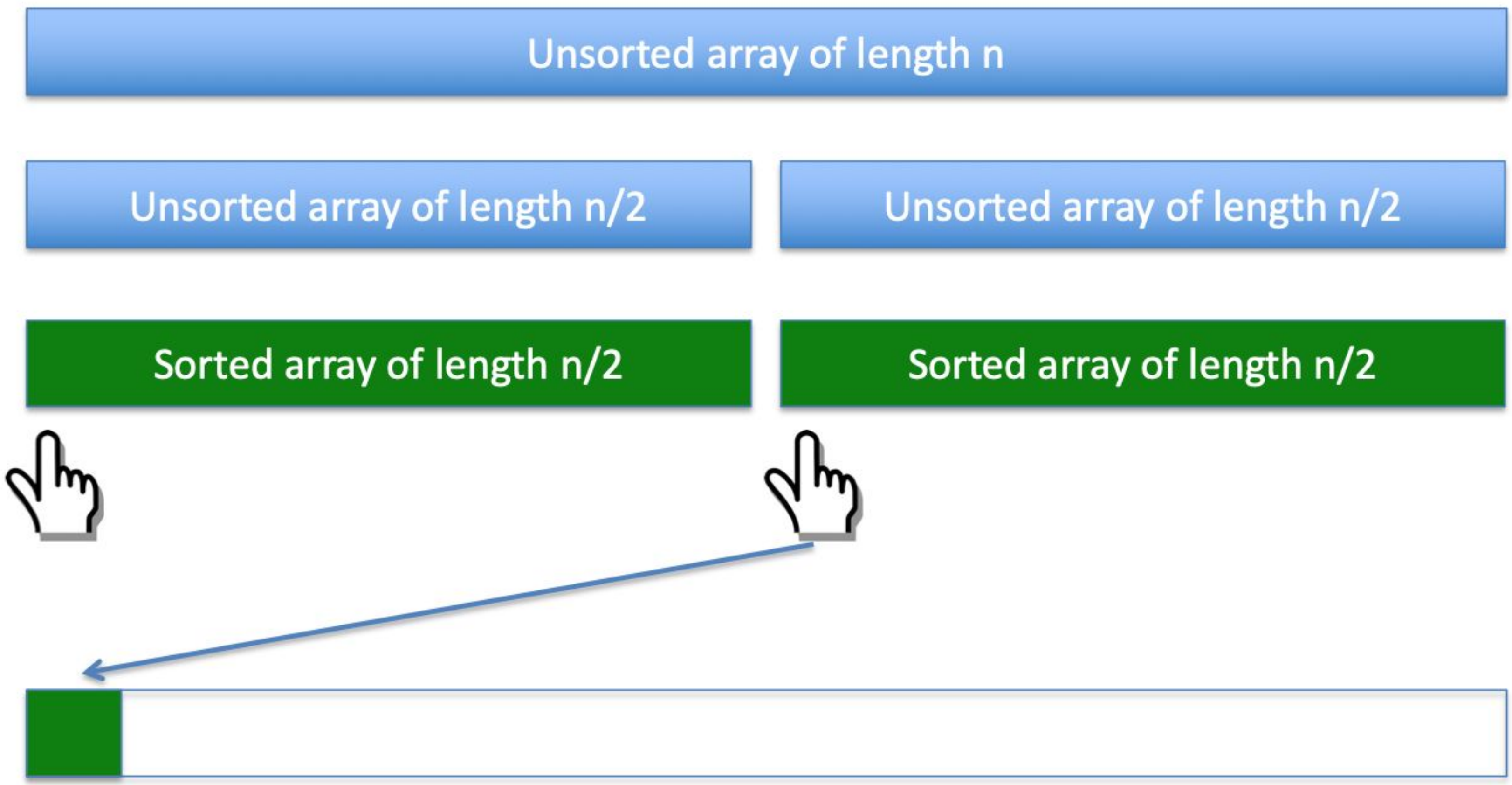
Unsorted array of length n/2

Unsorted array of length n/2

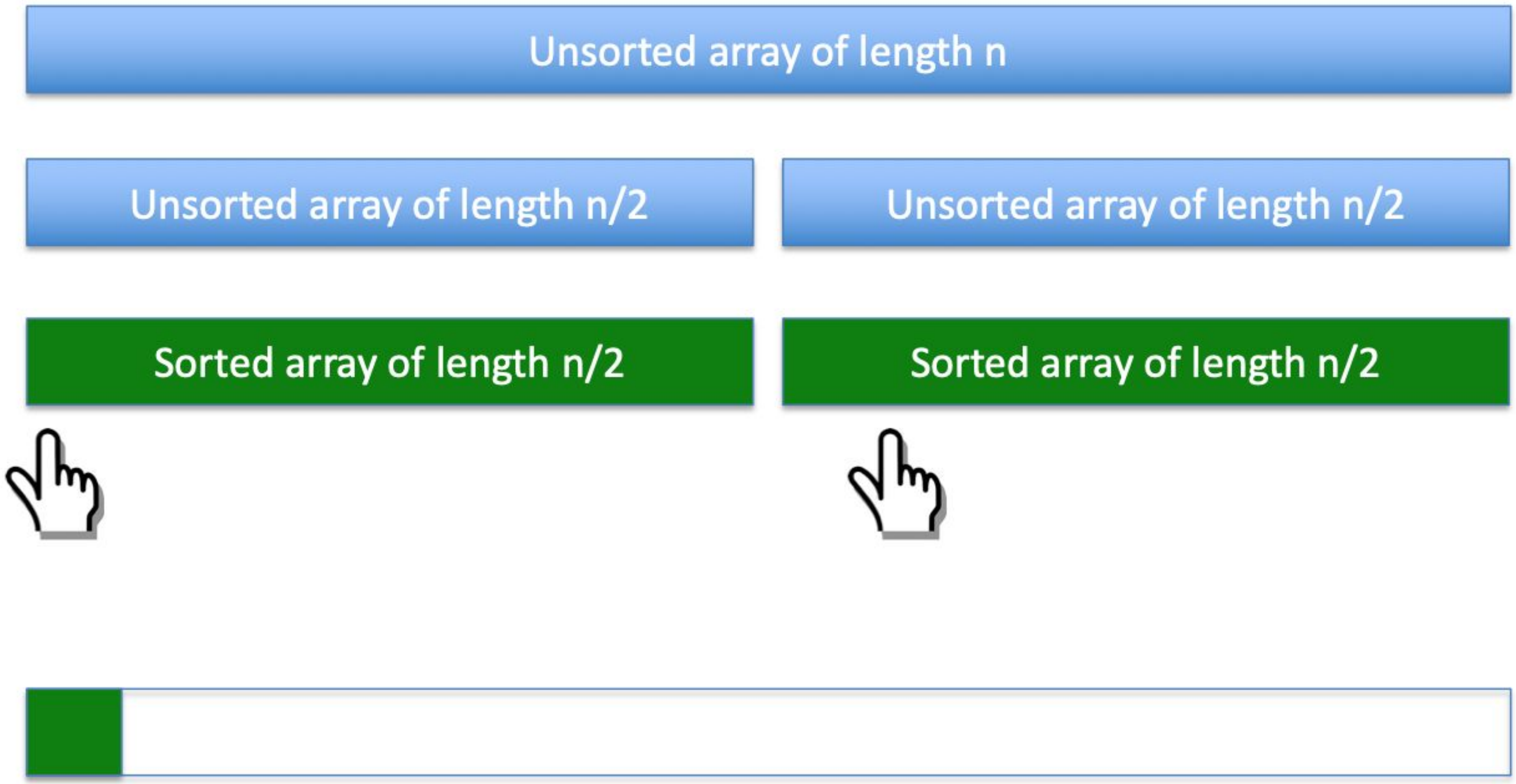
# Mergesort


Unsorted array of length n
Unsorted array of length n/2
Unsorted array of length n/2
Sorted array of length n/2
Sorted array of length n/2

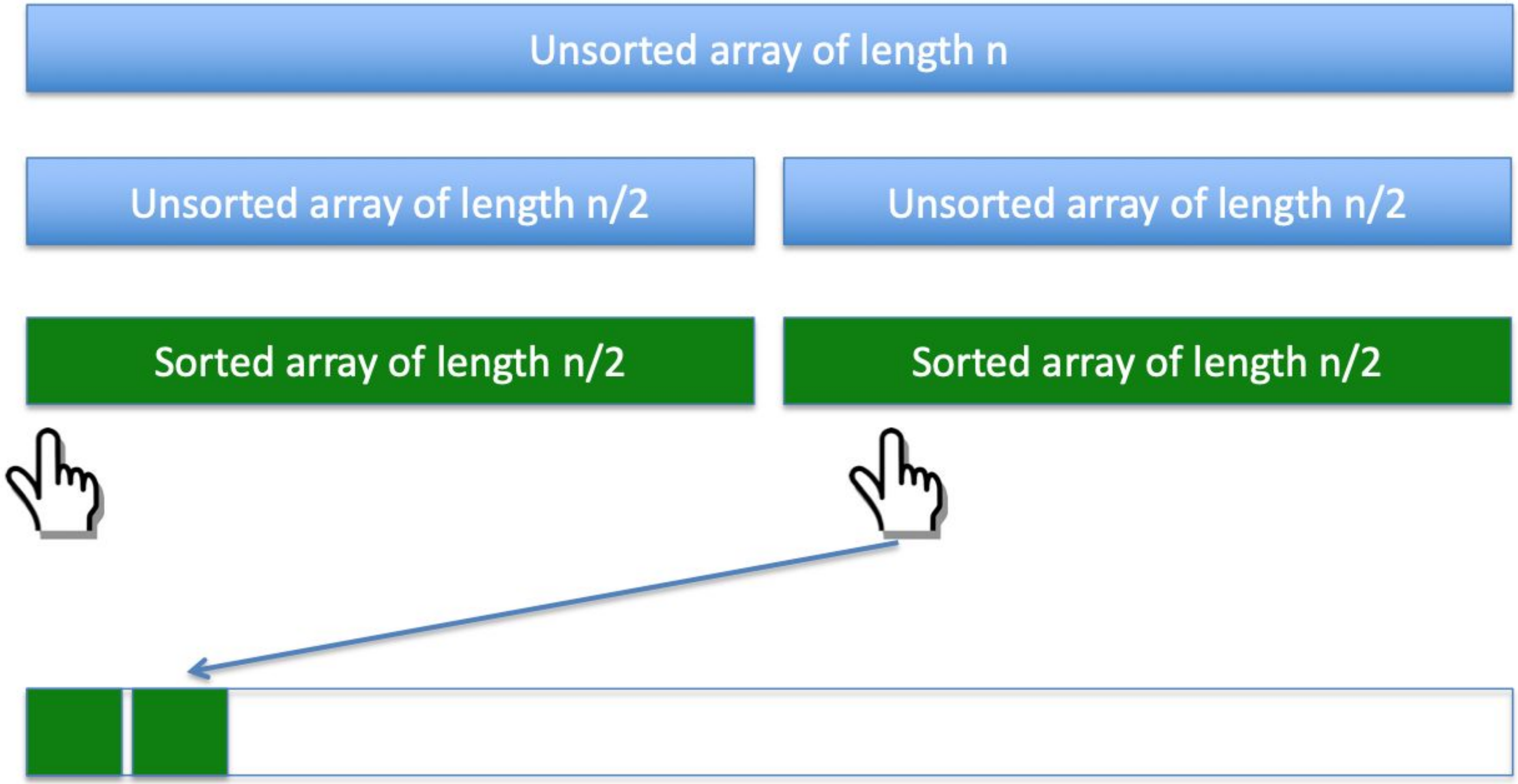
# Mergesort


Unsorted array of length n
Unsorted array of length n/2
Unsorted array of length n/2
Sorted array of length n/2
Sorted array of length n/2

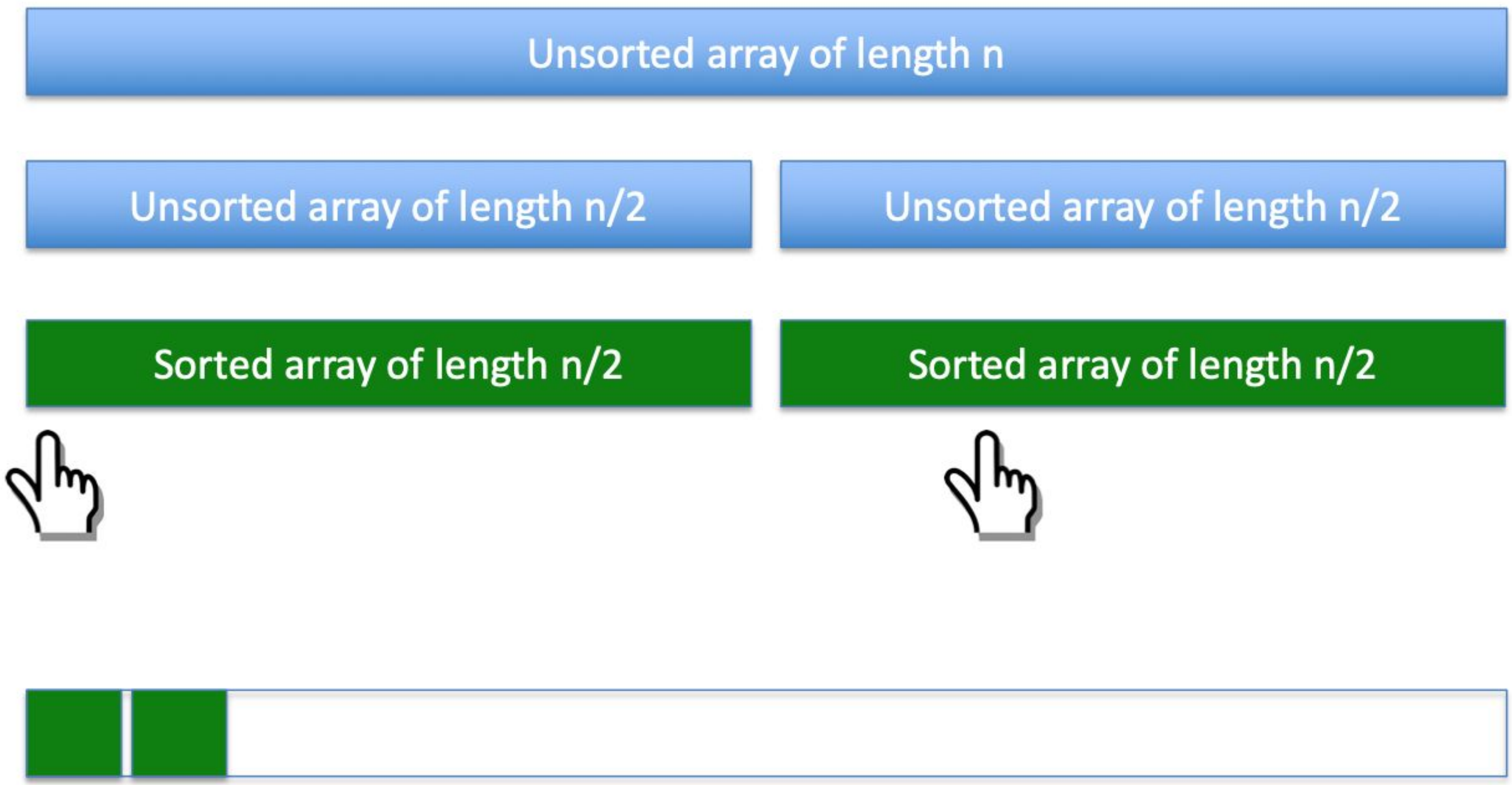
# Mergesort


Unsorted array of length n
Unsorted array of length n/2
Unsorted array of length n/2
Sorted array of length n/2
Sorted array of length n/2

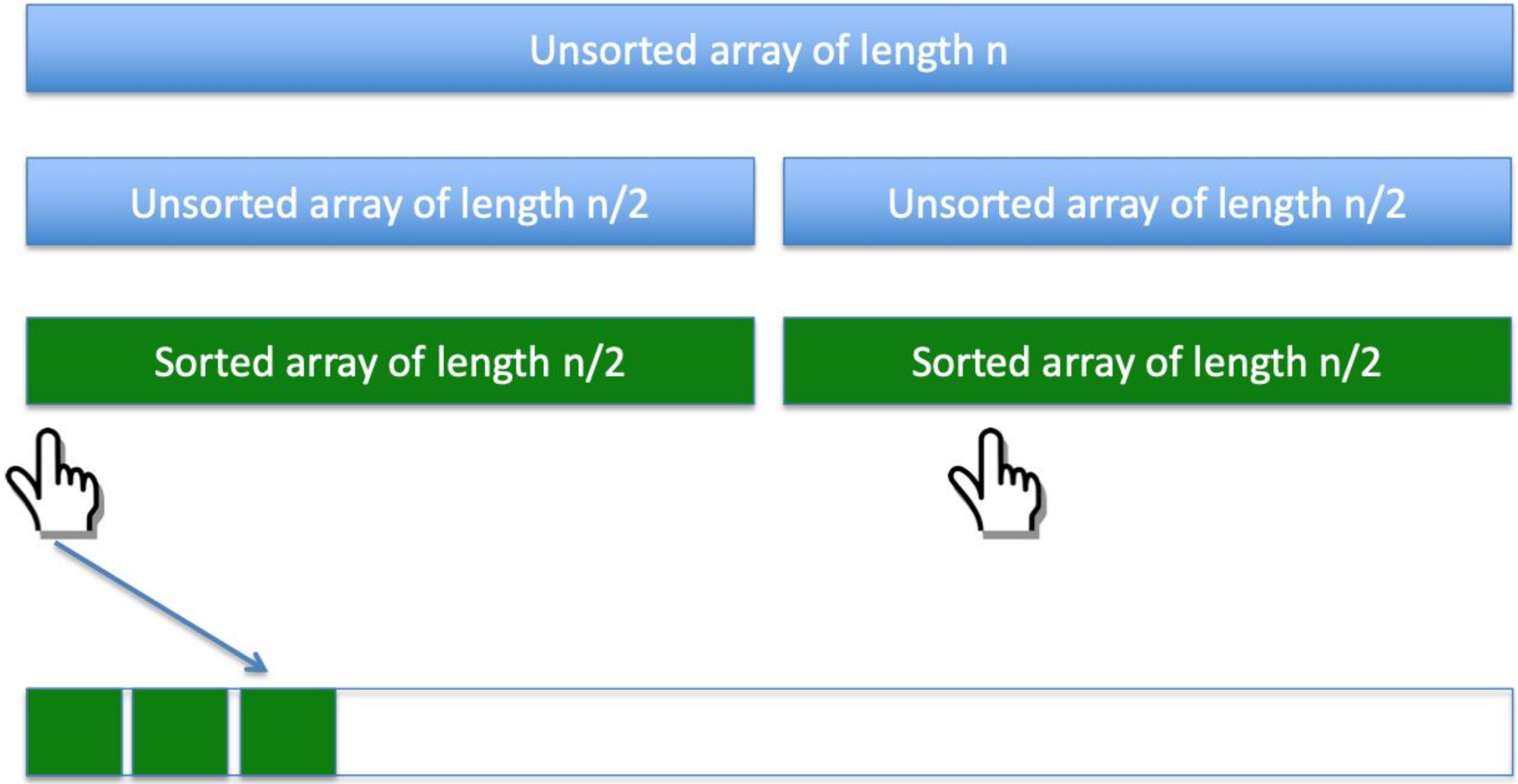
# Mergesort


Unsorted array of length n
Unsorted array of length n/2
Unsorted array of length n/2
Sorted array of length n/2
Sorted array of length n/2

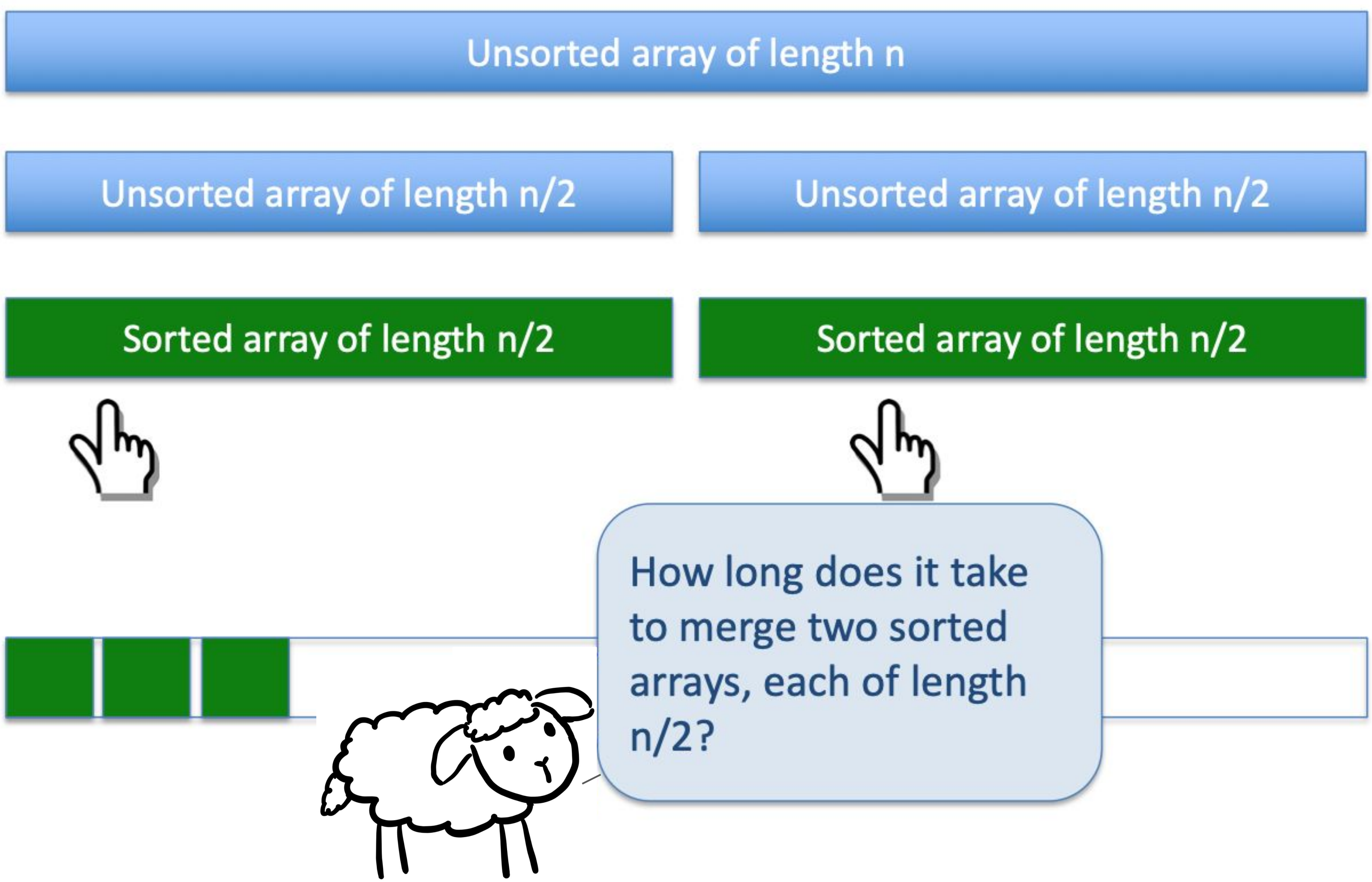
# Mergesort


Unsorted array of length n
Unsorted array of length n/2
Unsorted array of length n/2
Sorted array of length n/2
Sorted array of length n/2

# Mergesort


Unsorted array of length n
Unsorted array of length n/2
Unsorted array of length n/2
Sorted array of length n/2
Sorted array of length n/2

# Mergesort


Unsorted array of length n
Unsorted array of length n/2
Unsorted array of length n/2
Sorted array of length n/2
Sorted array of length n/2
How long does it take to merge two sorted arrays, each of length n/2?

# Recurrences and Running Times

T(n) = worst case running time of mergesort on input of length *n*

$$T(n) = \boxed{2\,T(n/2)} + \boxed{cn}$$

Merge two arrays of size *n*/2

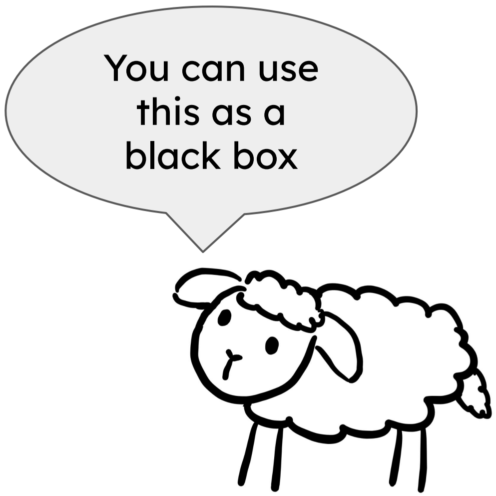
Two recursive calls on problems of size *n*/2

# Solving Recurrences

## The Master Theorem

**Formally:** Consider the recurrence relation $T(n) = kT(n/b) + O(n^d)$, when $k, b > 1$. Then:

$$T(n) = \begin{cases} O(n^d) & \text{if } \left(k/b^d\right) < 1 \\ O(n^d \log n) & \text{if } \left(k/b^d\right) = 1 \\ O(n^{\log_b k}) & \text{if } \left(k/b^d\right) > 1 \end{cases}$$




For Mergesort: k=2, b=2, d=1 ⇒ O(n log n).

# Hermit crabs sorting themselves

Another example of divide and conquer:

# Integer Multiplication

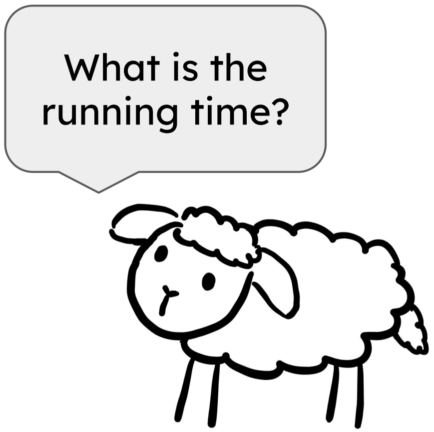
**Input:** Two n-digit positive integers x,y
**Output:** The product x·y

"Primitive operations" that can be done in constant time:

- add or multiply two single-digit numbers
- "shift" a number (i.e. add a 0 to the end)

**The Grade-School Algorithm**

| | | | |
|---|---|---|---|
| | | 3 | 4 |
| * | | 3 | 9 |
| | 3 | 0 | 6 |
| 1 | 0 | 2 | |
| 1 | 3 | 2 | 6 |



# An algorithm designer's mantra

"Perhaps the most important principle for the good algorithm designer is to refuse to be content."

- Aho, Hopcroft, Ullman, *The Design and Analysis of Computer Algorithms* (1974)

Another example of divide and conquer:

# Integer Multiplication

n digits

**Divide:** $x =$

| | |
|---|---|
| $x_L$ | $x_R$ |

$y =$

| | |
|---|---|
| $y_L$ | $y_R$ |

**Conquer:** $x \cdot y = (x_L \cdot 10^{n/2} + x_R)(y_L \cdot 10^{n/2} + y_R)$

$$= x_L y_L \cdot 10^n + (x_L y_R + x_R y_L) \cdot 10^{n/2} + x_R y_R$$

**Recurrence:**