Advanced Programming
INFO135

Lecture 4: **Sorting algorithms** in Python (part2)

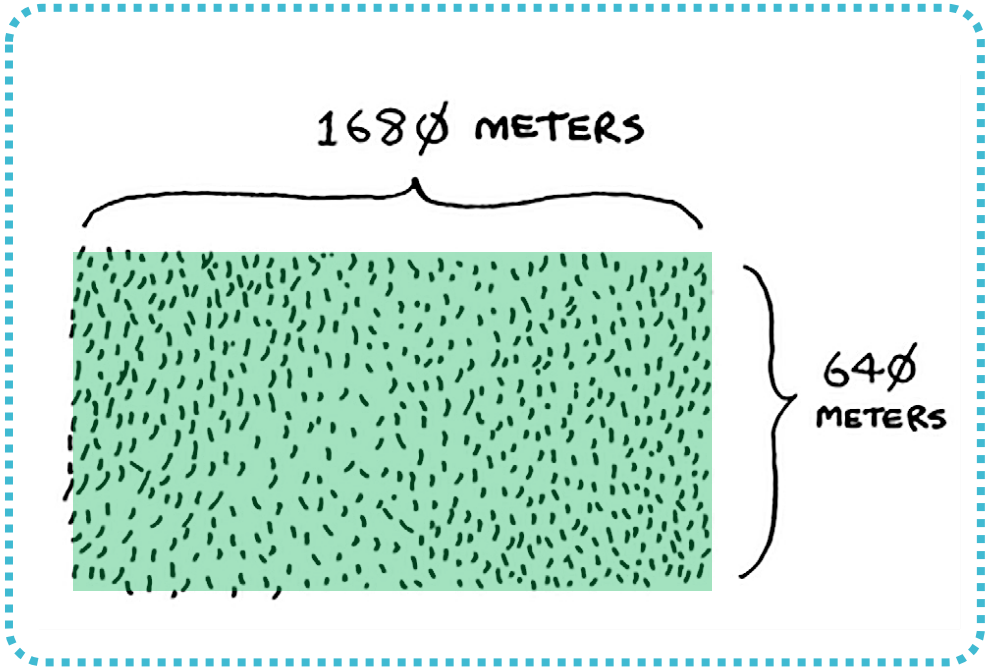
Mehdi Elahi

University of Bergen (UiB)

Divide & Conquer

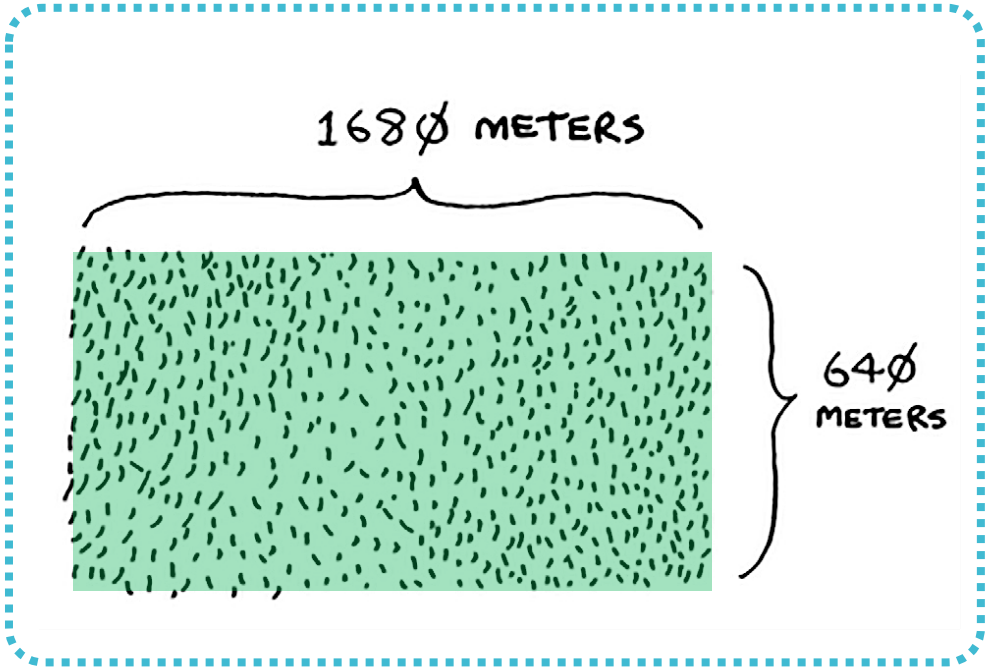
- **Divide & Conquer (D&C)** is a strategy to solve problems.

- **Example:** Suppose you have **a plot of land**.
- The size of your land is **1680 x 640 (m)**



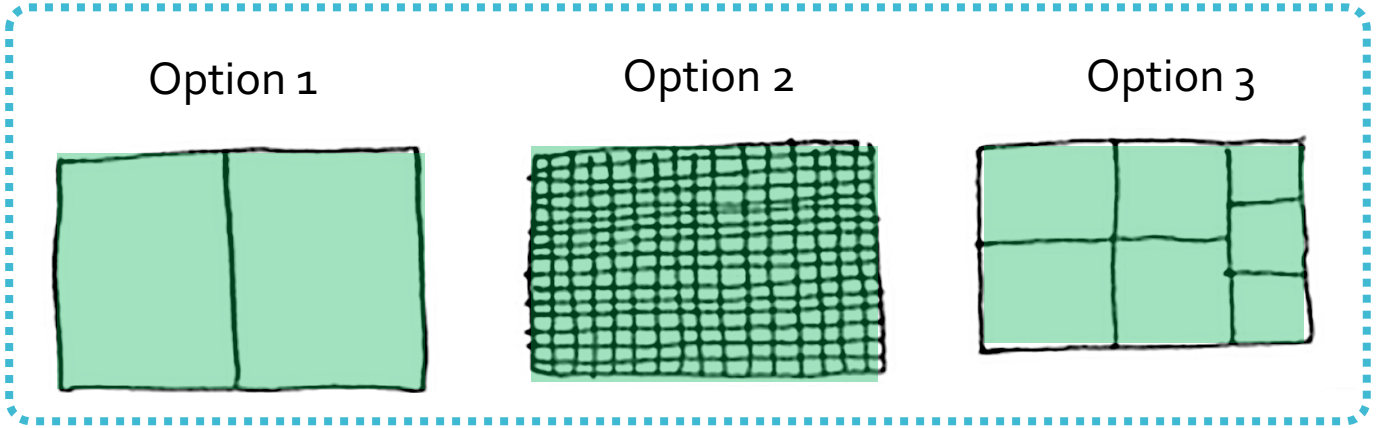
Divide & Conquer

- You want to divide this farm evenly into **square plots**.
- Plots should be **as big as possible**.



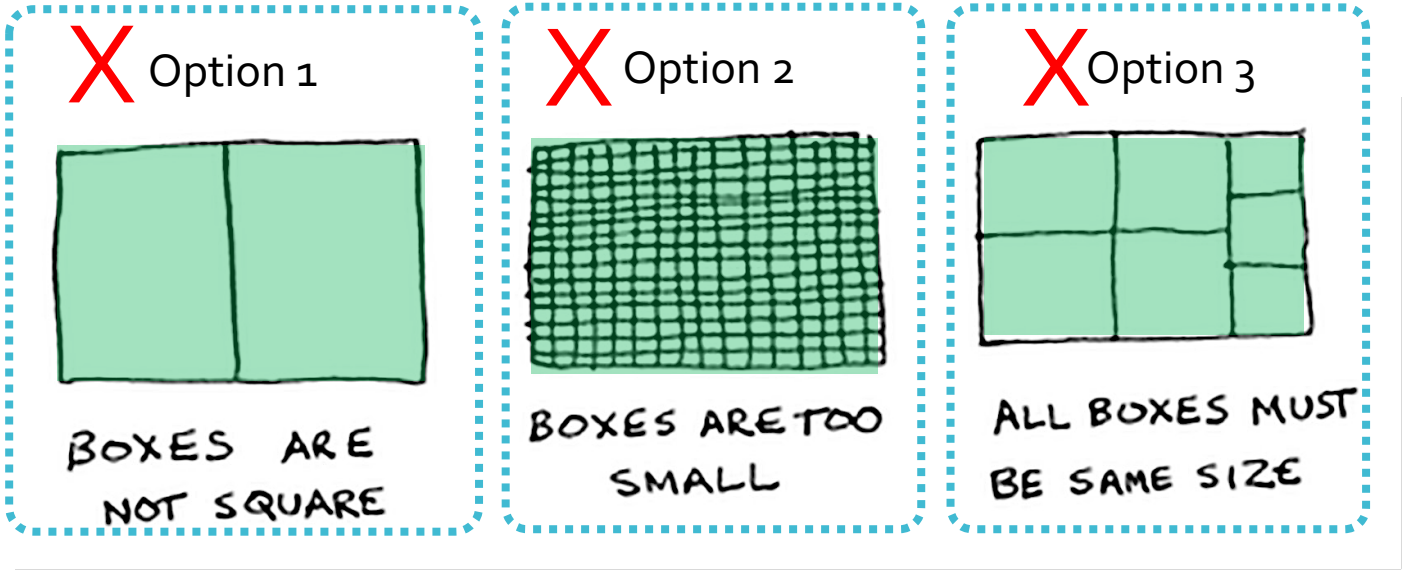
Divide & Conquer

- We can explore some of the **splitting** methods?
- These can be options:



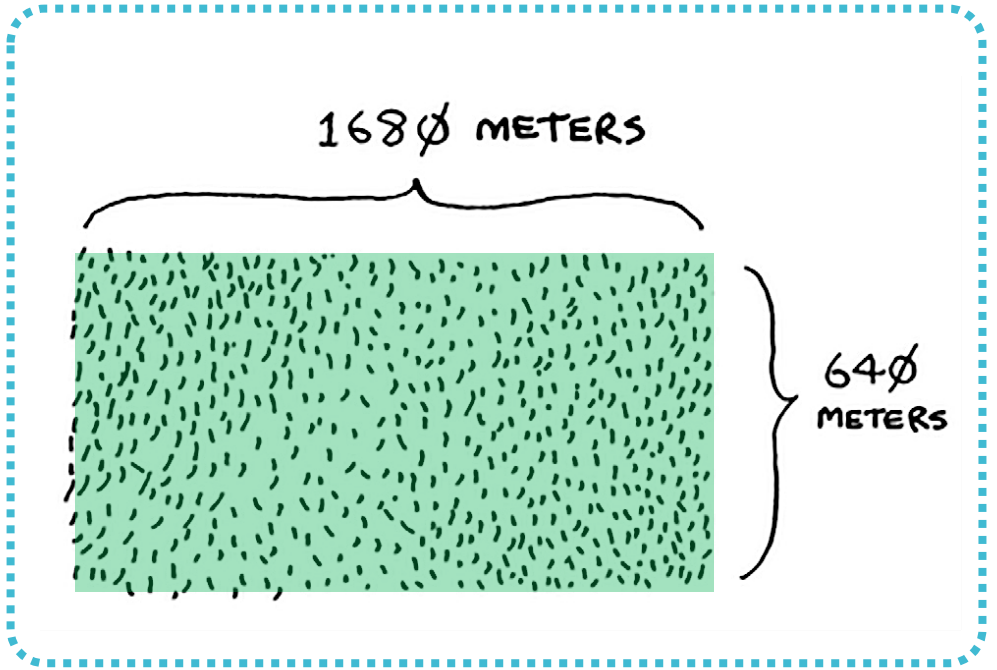
Divide & Conquer

- Well … there are issues with each of these options.
- So **none of them** will work.



Divide & Conquer

- **Question revisited:** How do you find the **largest square** size?



Divide & Conquer

- **Answer:**

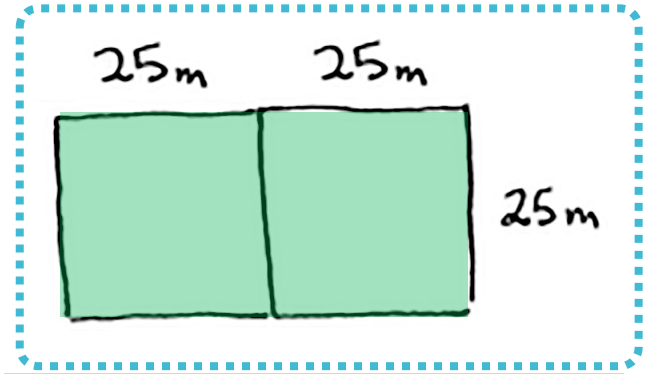
  You can use **Divide & Conquer** (D&C) strategy!

- **Divide & Conquer** strategy based algorithms are recursive and solve problem in two steps:
  - **step 1:** figure out the base case (the *simplest* case).
  - **step 2:** divide the problem until it becomes base case.

Divide & Conquer

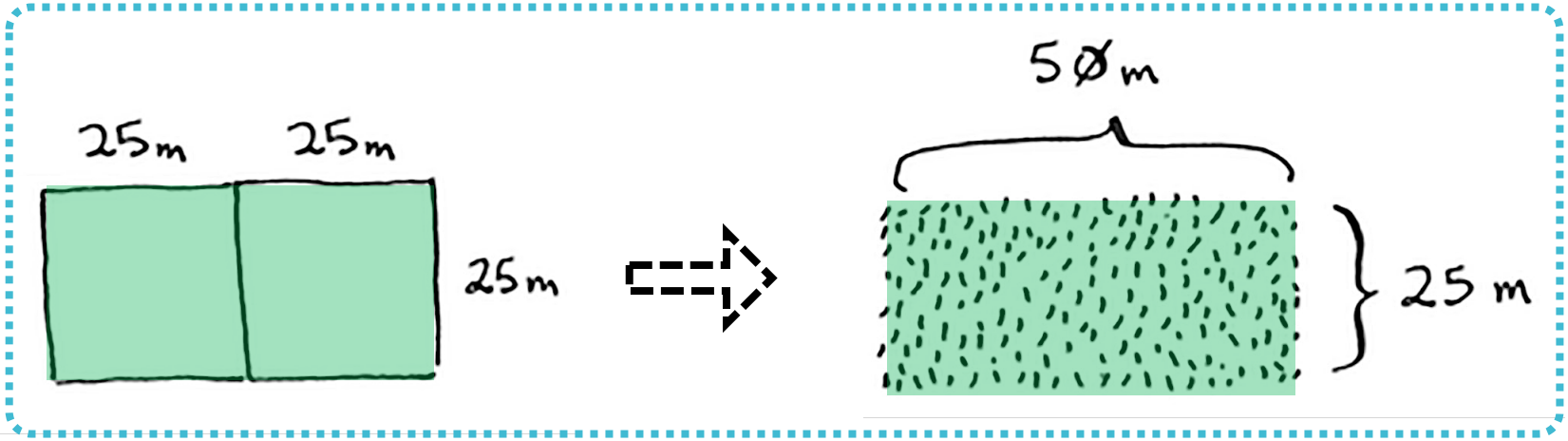
- Let's **Divide & Conquer** to find a solution to this problem:

- **step 1**: figure out the base case. Easiest case would be when one side was a multiple of another side, right?



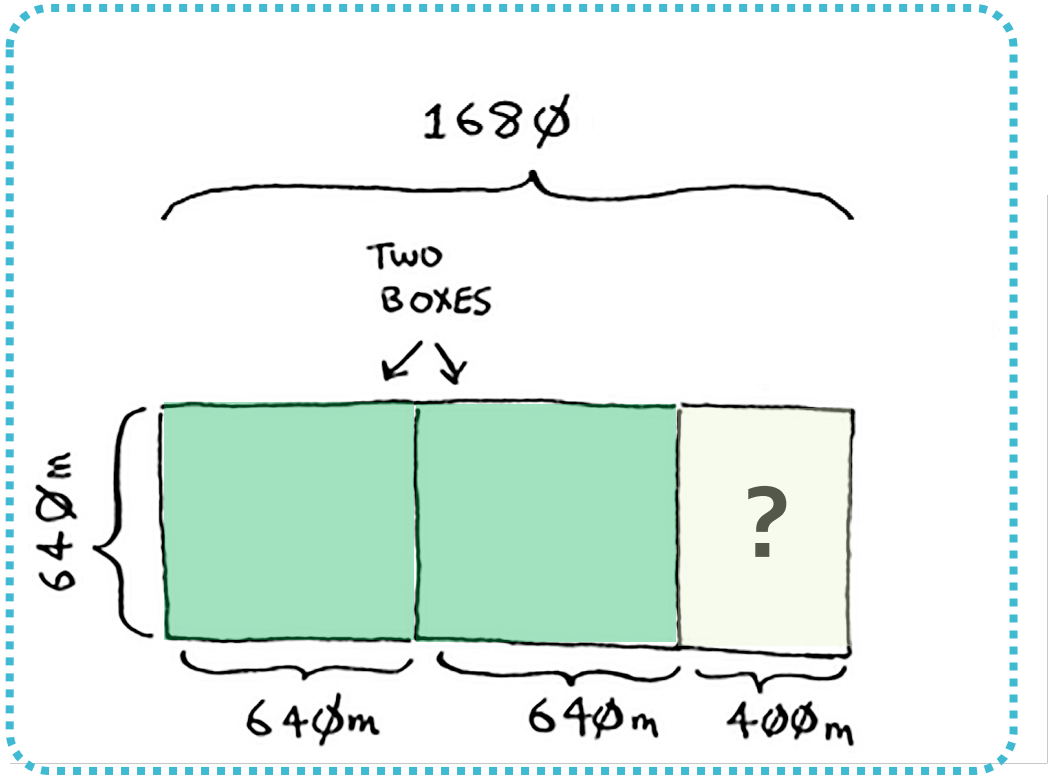
Divide & Conquer

- Suppose one side is **25 (m)** and the other side is **50 (m).**
- Then, the largest box you can use is **25 m × 25 m**.



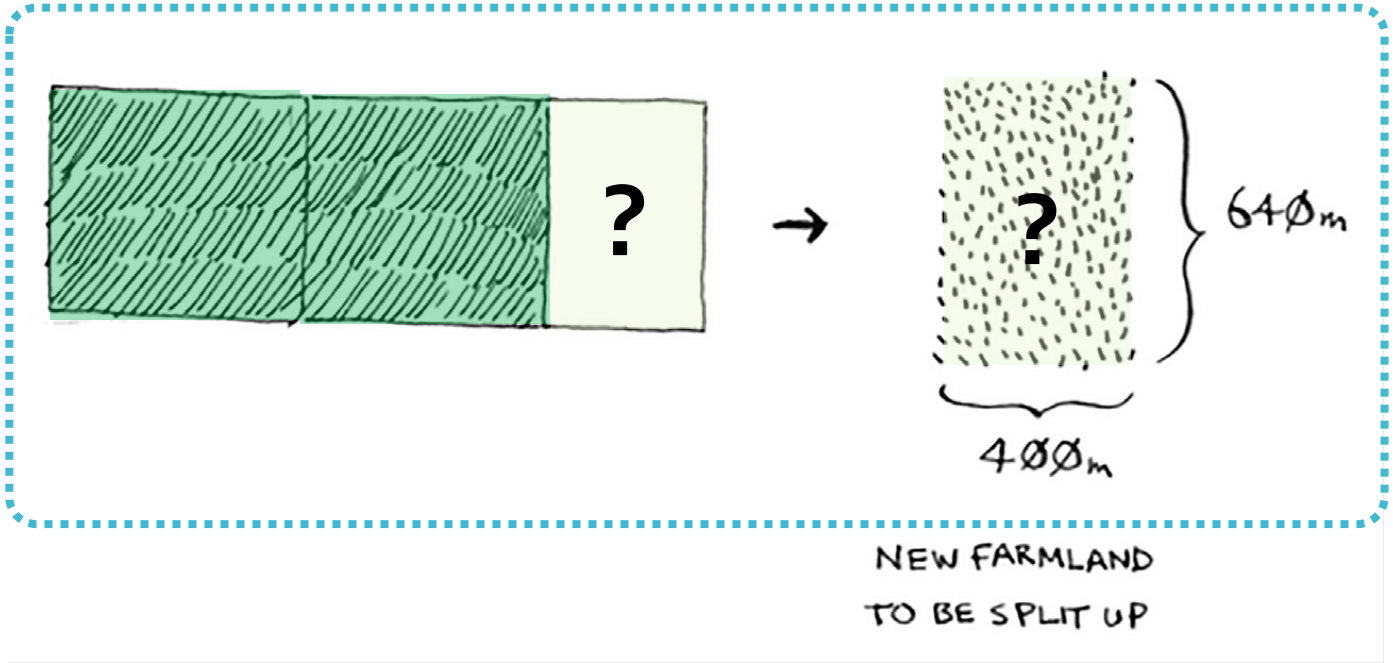
Divide & Conquer

- **second step:** with every recursive call, you have to reduce your problem to a smaller problem.
- But How?
- Let's mark out **two biggest boxes (640 × 640)** to use.



Divide & Conquer

- But there's some land left to be divided.
- Why not to **apply same algorithm** here?



Divide & Conquer

- What happened?
- **You started** out with a **1680 × 640 (m)** farm that needed to be split up.
- Now you need to split a smaller segment, **640 × 400 (m).**

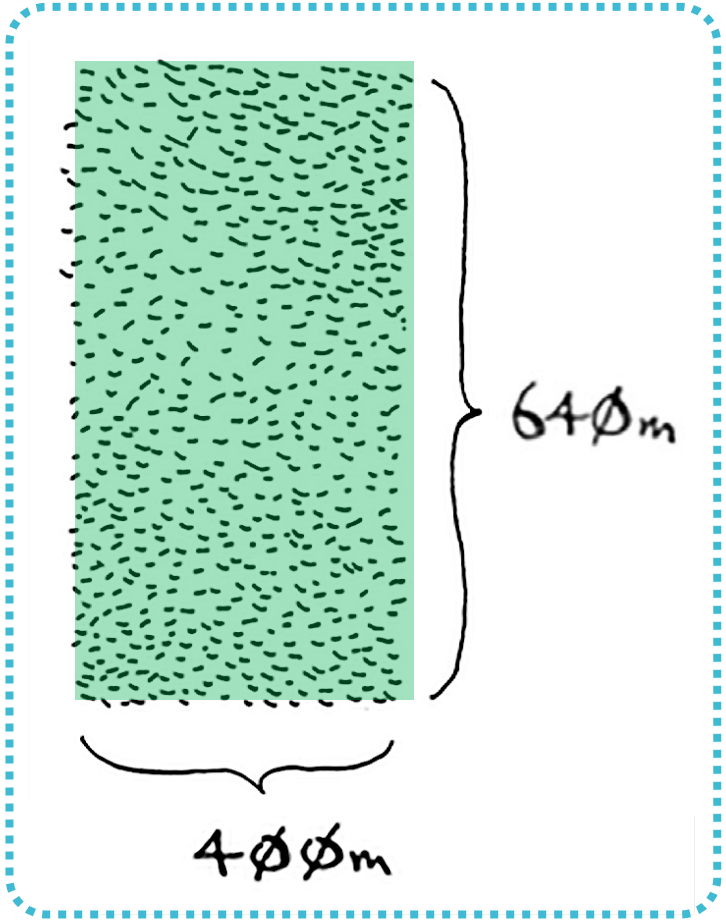
Divide & Conquer

- You just reduced the problem:
  - **from** a 1680 × 640 (m) farm
  - **to** a 640 × 400 (m) farm!

- You can continue and get the problem smaller and smaller.

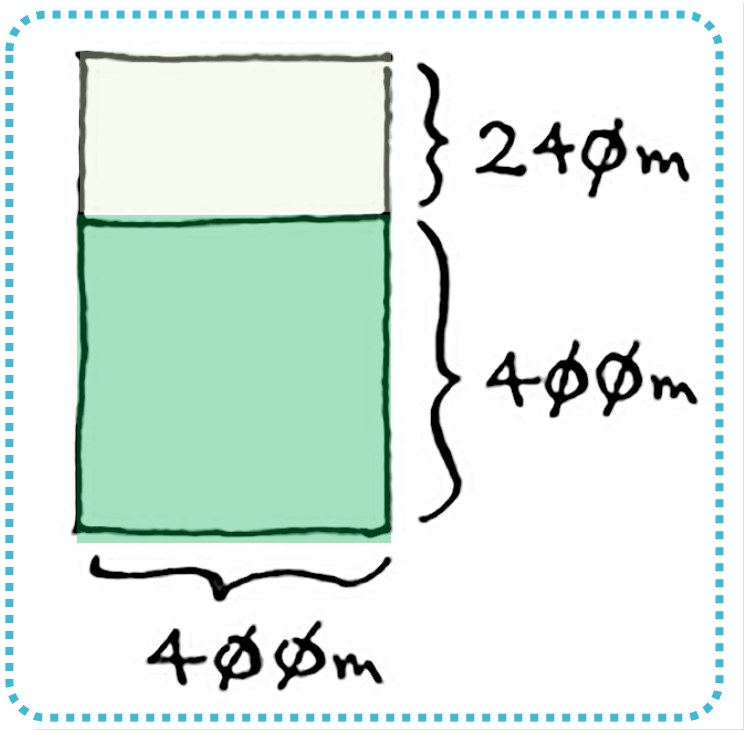
Quiz

- Smaller problem: a **land** with a **size 640 × 400 (m).**
- Need to be divides it evenly into **square plots**.
- What will be the **biggest plot**?



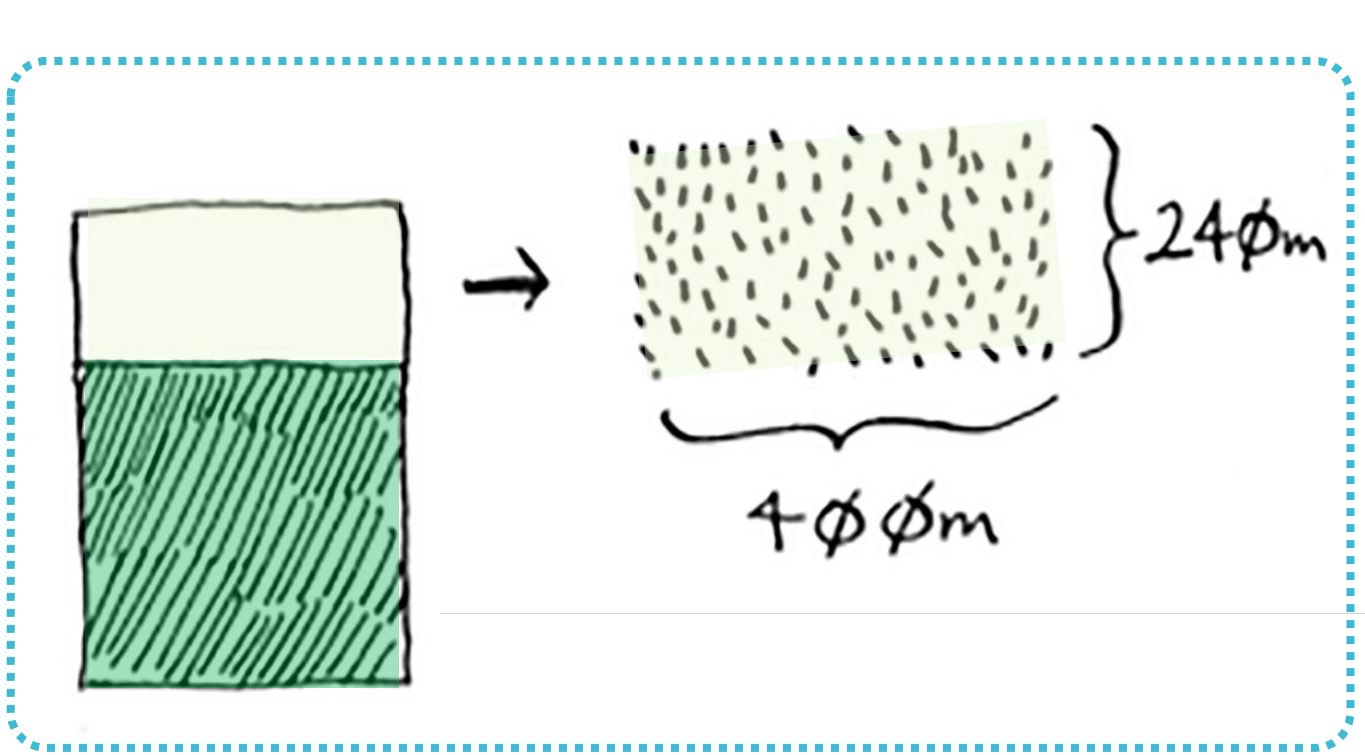
Answer

- Lets start with a plot of **400 × 400 (m).**

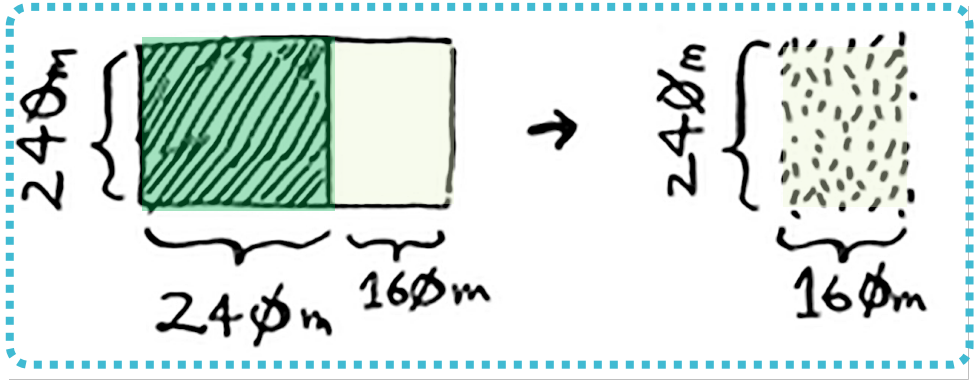


Answer

- And that leaves you with a smaller plot of 400 × 240 (m)



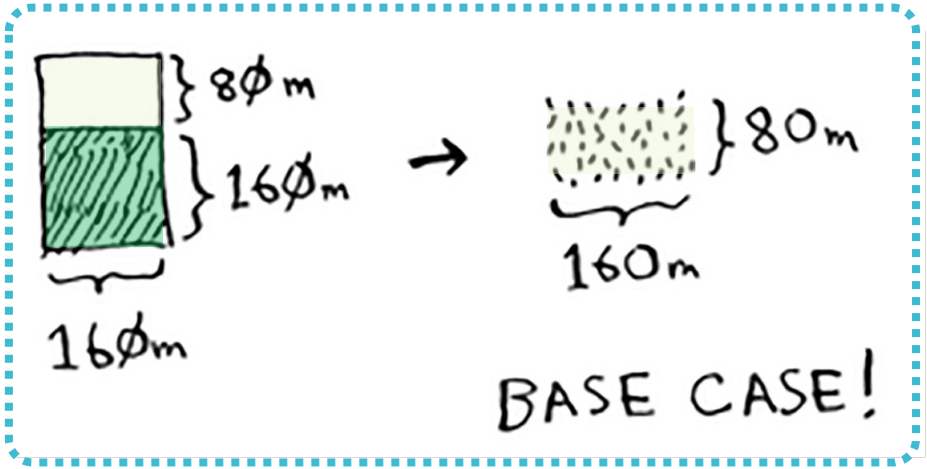
Answer



- You can draw a box on that to get a smaller segment, **240 × 160 (m)**.

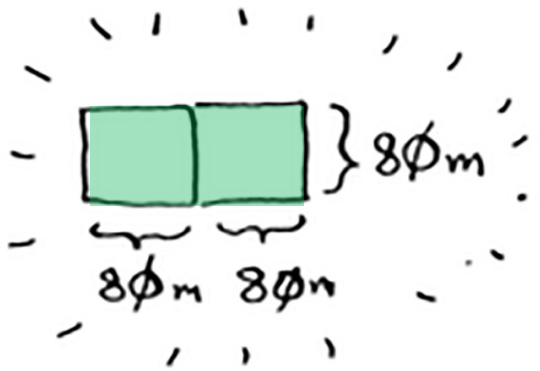
- And again, you can draw another box to get a smaller segment, 80 × 160 (m).

- This can be your **Base Case**.



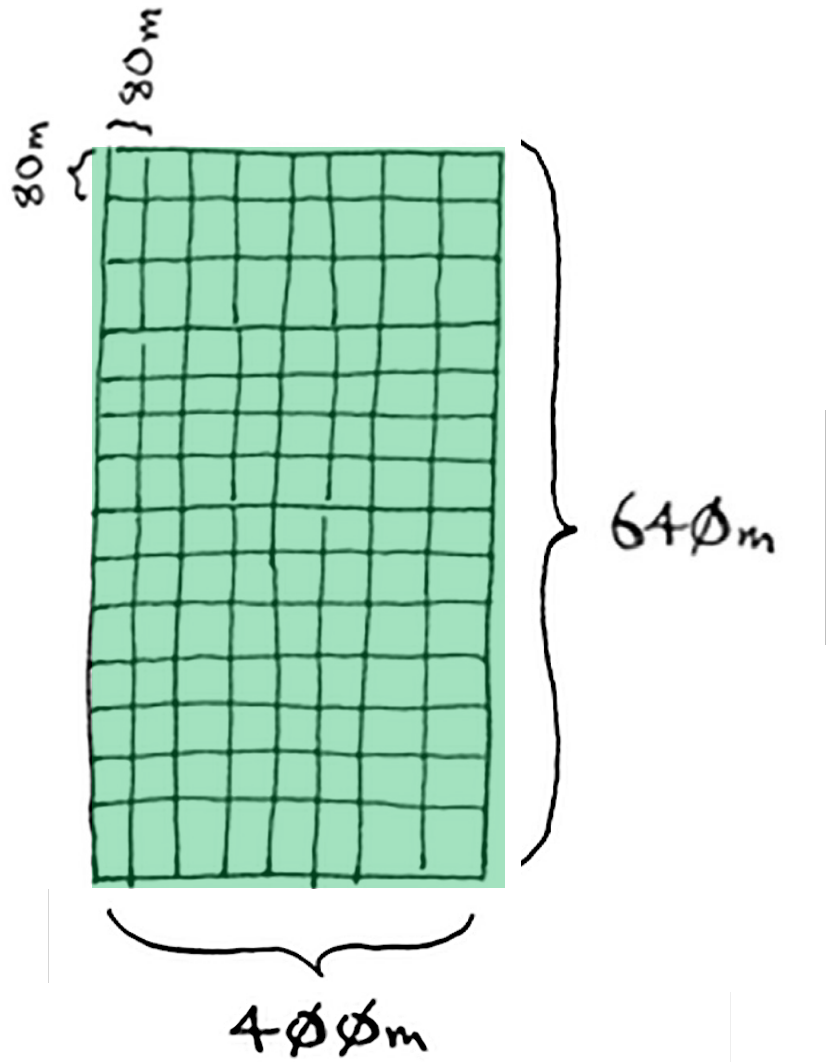
Answer

- Why Base Case?
- Because **80 is a factor of 160** and you can perfectly split up this segment using two boxes.
- And nothing is left over!



Answer

- At the end, for the original land, the biggest plot size you can use is **80 × 80 m.**



Application

- Now we will apply this approach in a sorting algorithm.

Quick Sort

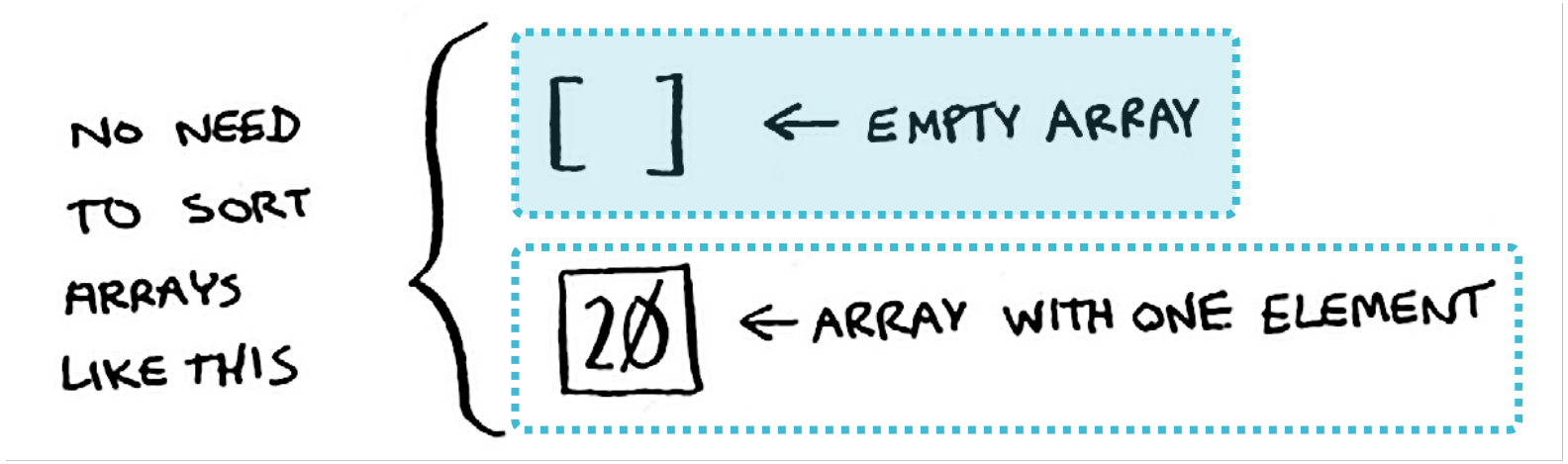
- **Quick Sort** is a sorting algorithm developed based on **Divide & Conquer.**
- It is faster than other sorts (e.g., selection sort).
- Developed by British computer scientist **Tony Hoare** in 1959.

Quick Sort

- Suppose you would like to apply **Quicksort** on an array.
- What might be the **simplest** array to sort?

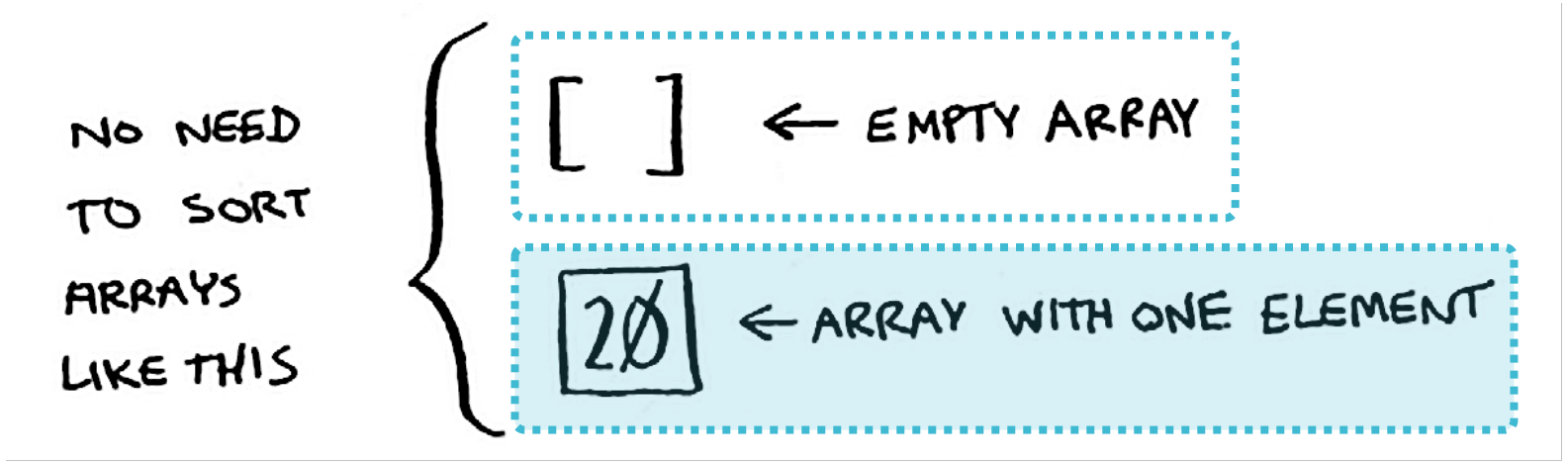
Quick Sort

- Suppose you would like to apply **Quicksort** on an array.
- What might be the **simplest** array to sort?



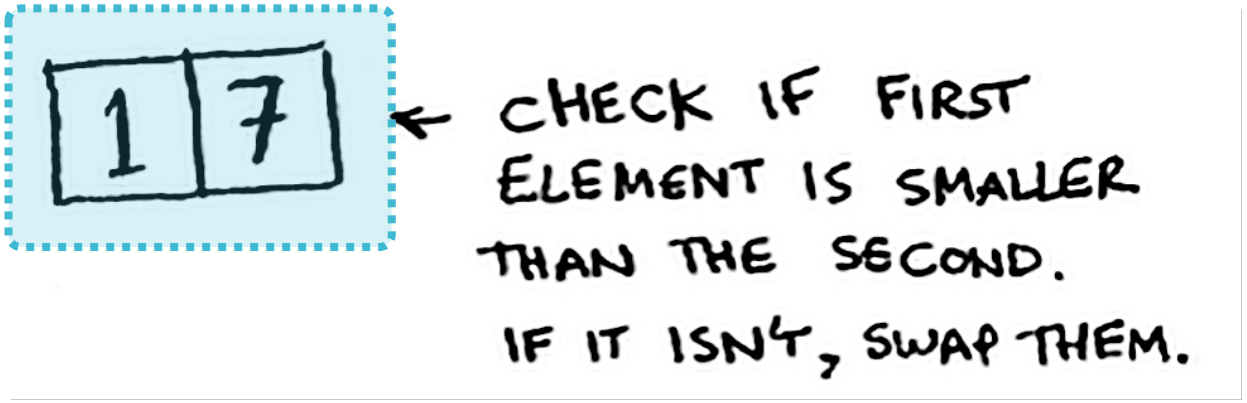
Quick Sort

- Suppose you would like to apply **Quicksort** on an array.
- What might be the **simplest** array to sort?



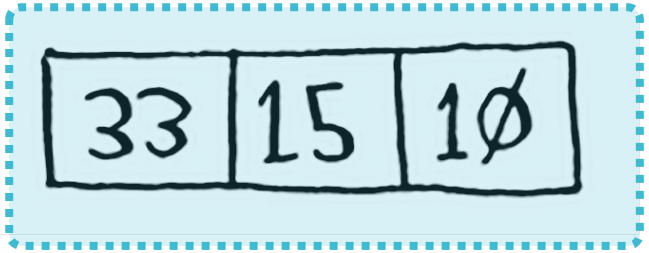
Quick Sort

- What about an array with **two elements**?



Quick Sort

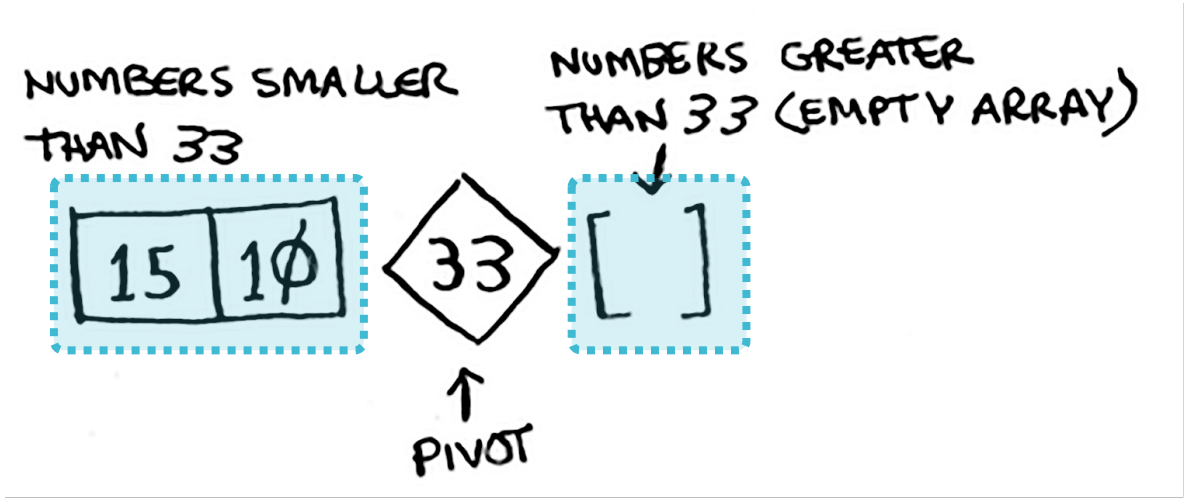
- And an array of **three elements**?



- First, **pick an element** in the array (called **Pivot**). Let's say the first element is the pivot.

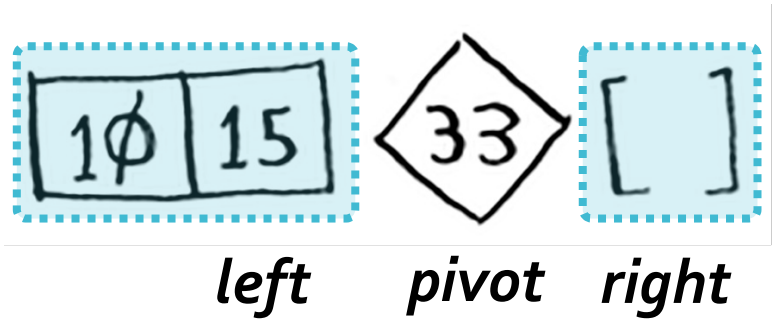
Quick Sort

- Now find the elements **smaller than the pivot** and the elements **larger than the pivot**.



Quick Sort

- This is called **partitioning**. The result is a:
  - **sub-array "*left*":** elements less than pivot
  - **pivot**
  - **sub-array "*right*":** elements greater than pivot



- Sub-arrays **may not** be **sorted**.

Quick Sort

- **Idea:** What **if** the **sub-arrays** are **sorted**?
- then you can just combine: **left** + **pivot** + **right**

- This will give you the sorted array:

```
[10, 15] + [33] + [] = [10, 15, 33]
```

Quick Sort

- **Quick sort** algorithm:

1) Pick a **pivot**.
2) **Partition** the array into two sub-arrays:
   a) ***left*: elements less than pivot**
   b) ***right*: elements greater than pivot.**
3) Call quicksort **recursively** on two sub-arrays.

Quick Sort

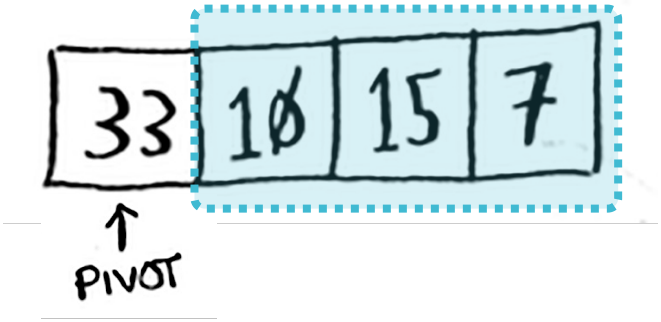
- **Quick sort** algorithm:
  1) Pick a **pivot**.
  2) **Partition** the array into two sub-arrays:
     a) ***left*: elements **less than pivot**
     b) ***right*: elements **greater than pivot.**
  3) Call quicksort **recursively** on two sub-arrays.

Quick Sort

- **Quick sort** algorithm:
  1) Pick a **pivot**.
  2) **Partition** the array into two sub-arrays:
     a) ***left*: elements **less than pivot**
     b) ***right*: elements **greater than pivot.**
  3) Call quicksort **recursively** on two sub-arrays.

Quick Sort

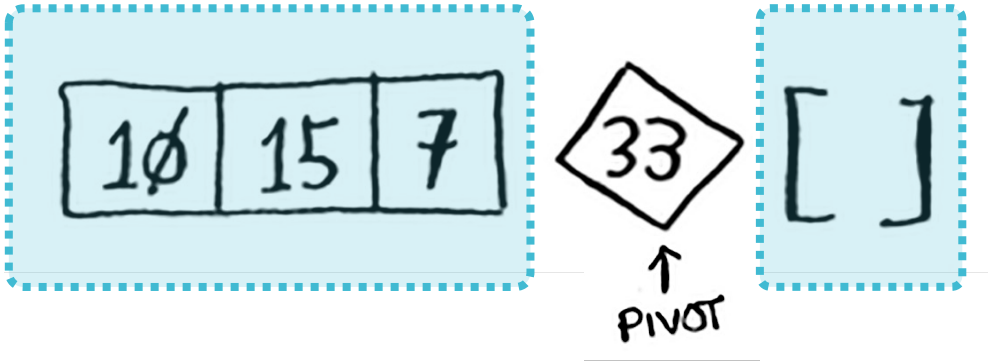
- We did for array of **three elements**.
- Suppose you have an array of **four elements**.



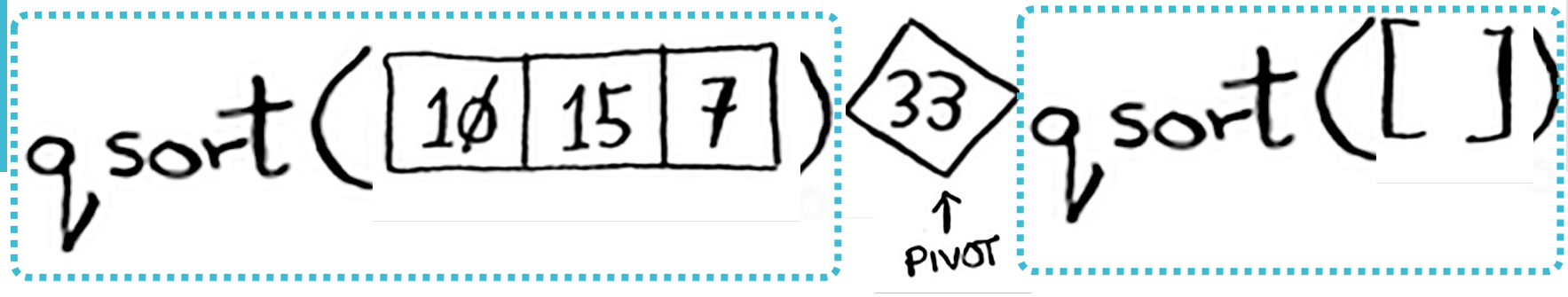
- First, pick a **pivot** (for example 33).

Quick Sort

- The array on the left has three elements.

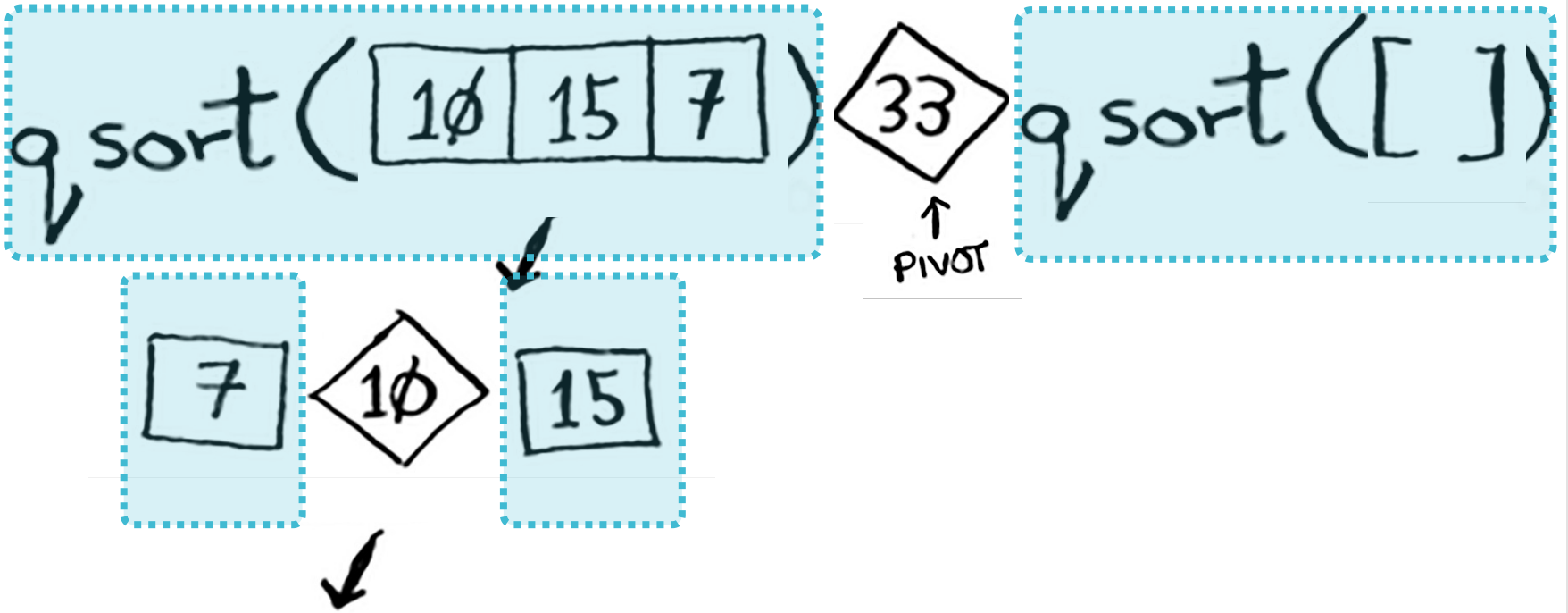


- So you can call quicksort on it **recursively**.



Quick Sort

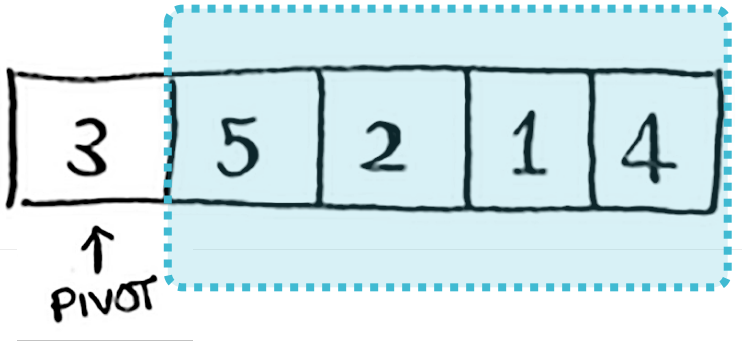
- You **already done** quicksort on **array of size three**.



Sorted array!

Quick Sort

- What about **larger arrays**?
- Suppose you have an array of **five elements**.

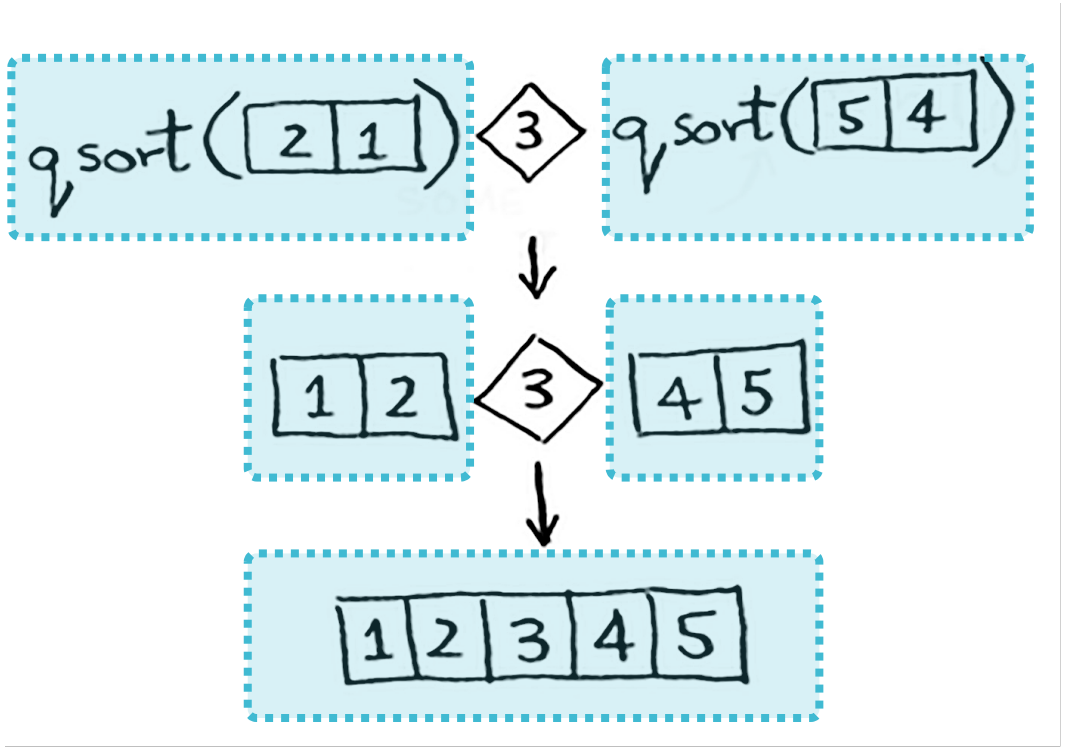


- First, pick a pivot (for example 3).

Quick Sort

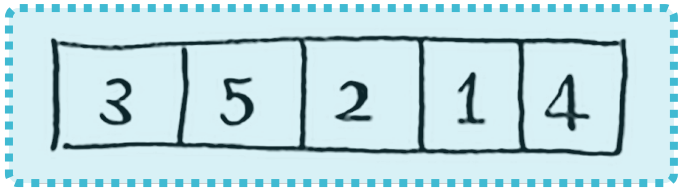
- After partitioning, you have the sub-arrays.
- Then, you call quicksort (**qsort**) on sub-arrays.

- Finally, you combine the sub-arrays and pivot.

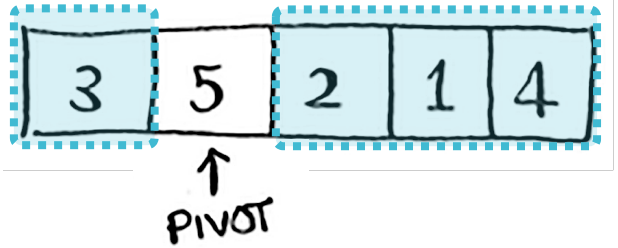


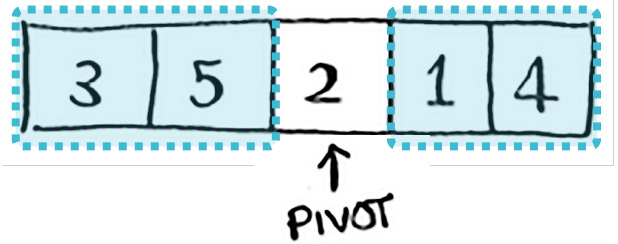
Quiz

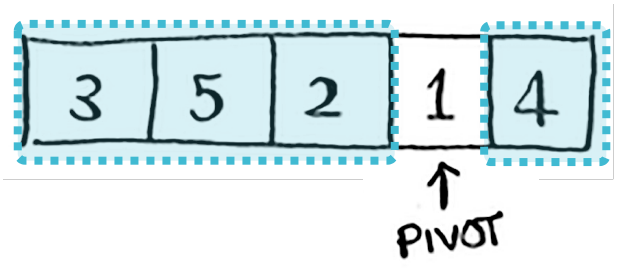
- Suppose you have **an array of five elements**.

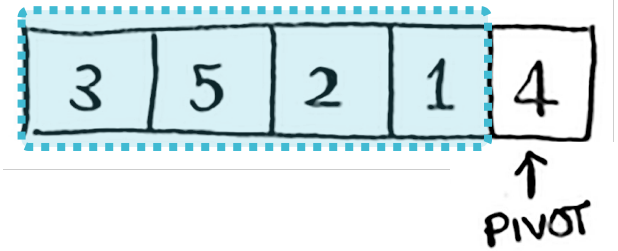


- Write down the ways you can partition this array, **depending on** different **pivots**?



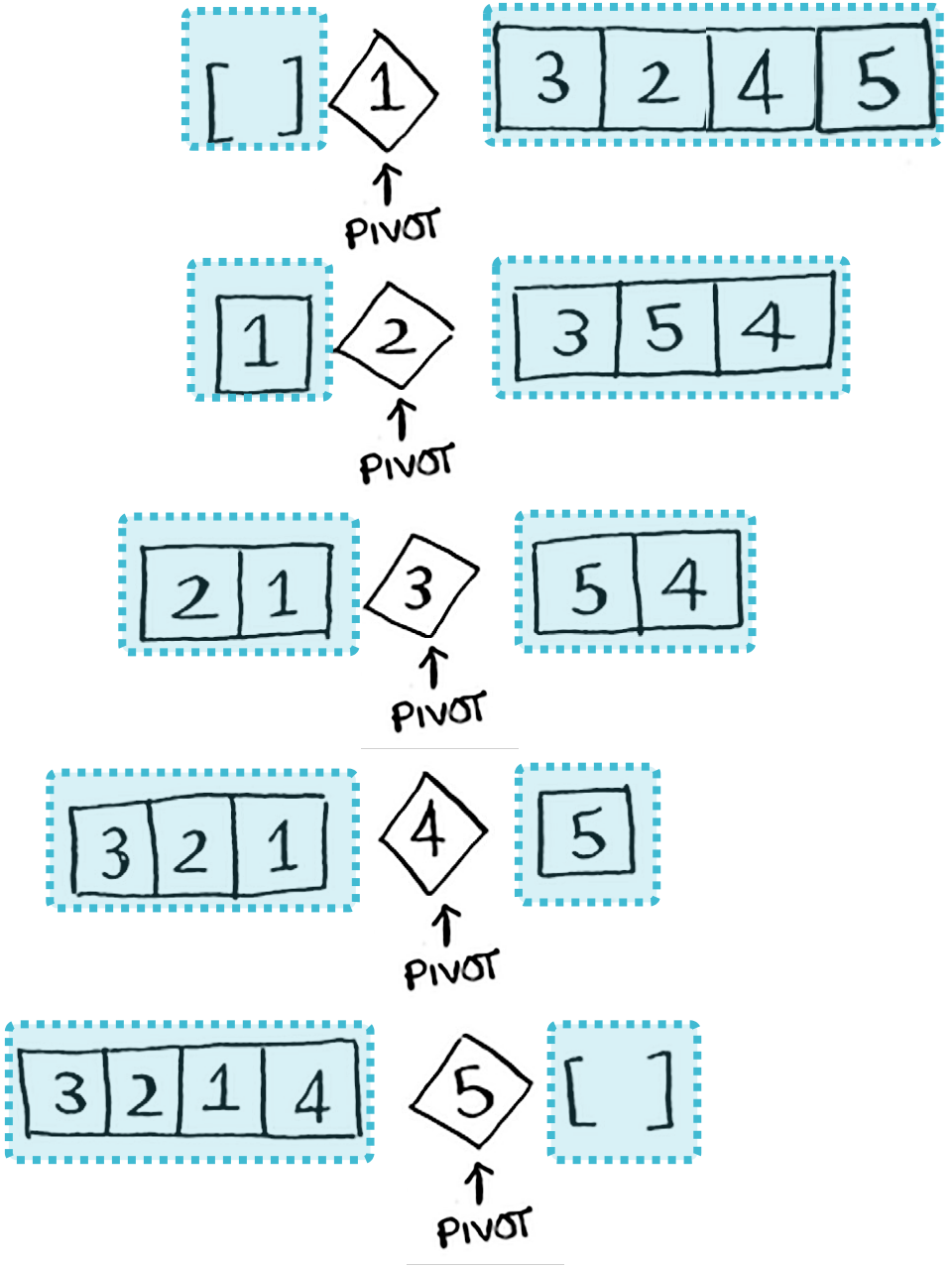






Answer

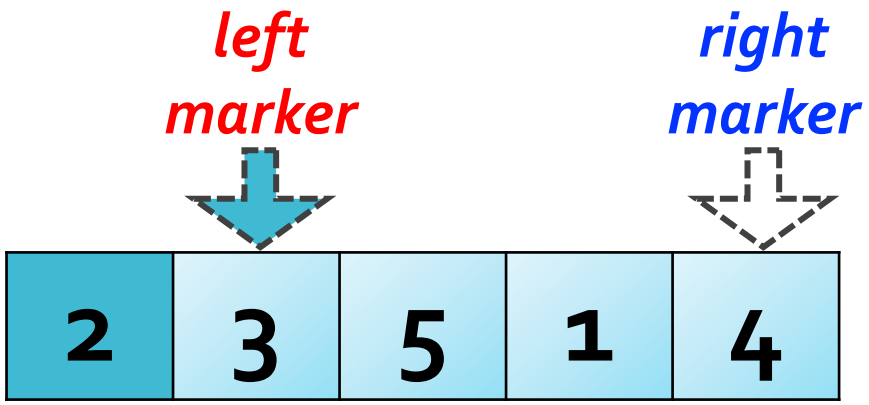
- You can partition the array, in different ways, **depending on** different **pivots**.



Quick Sort

***Lets Quick sort** this!*
*We choose 2 as **Pivot***

Quick Sort

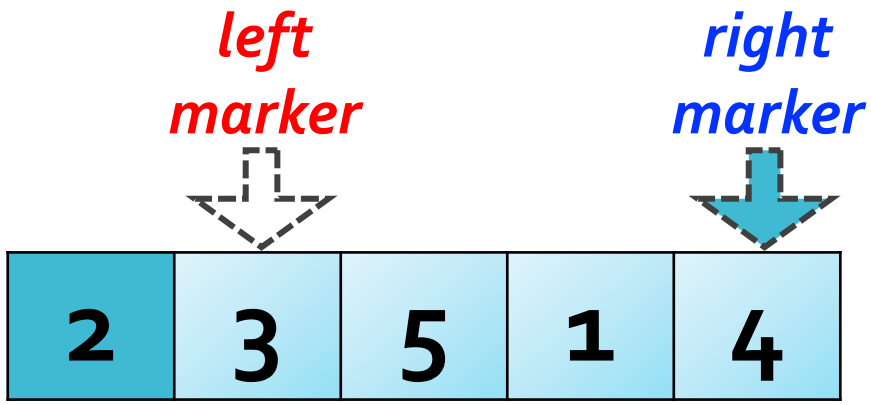


***left marker*:**

3 is larger than 2 (pivot):

stop!

Quick Sort

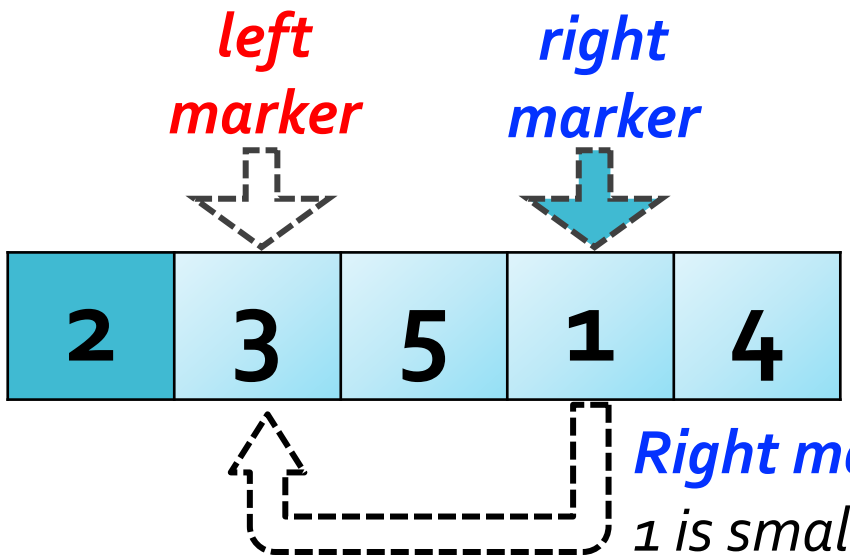


***Right marker*:**

4 is not smaller than 2 (pivot):

move!

Quick Sort

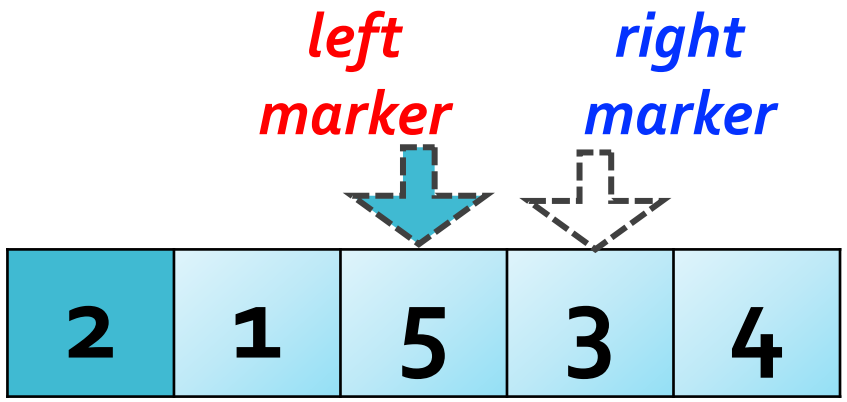


Right marker:

1 is smaller than 2 (pivot):

swap!

Quick Sort

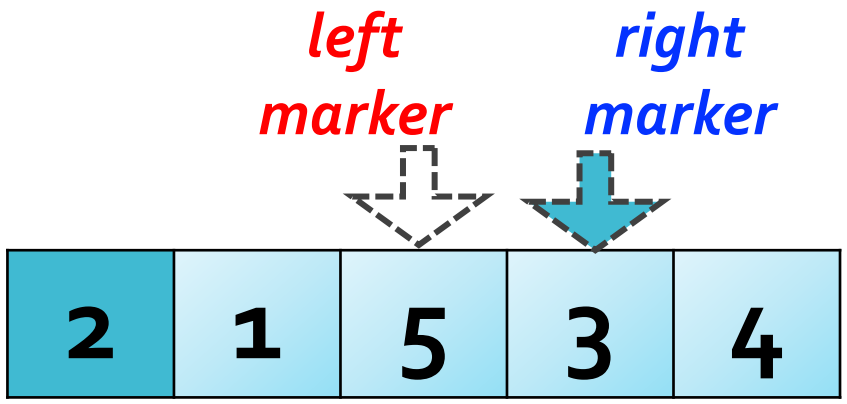


left marker:

5 is greater than 2 (pivot):

stop!

Quick Sort

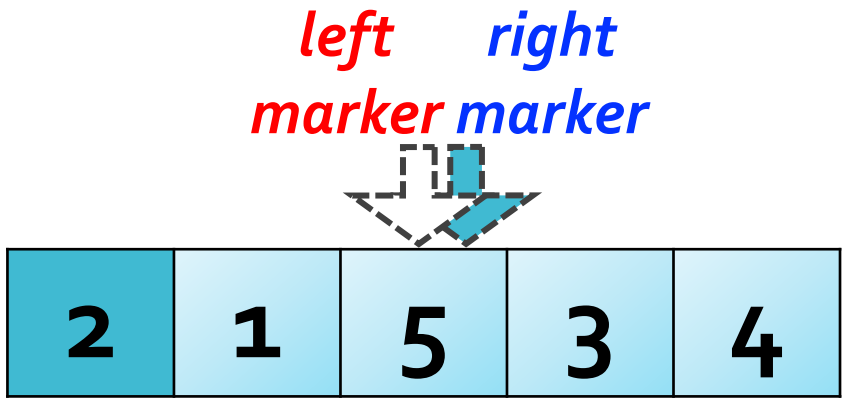


***right marker*:**

3 is not smaller than 2 (pivot):

move!

Quick Sort

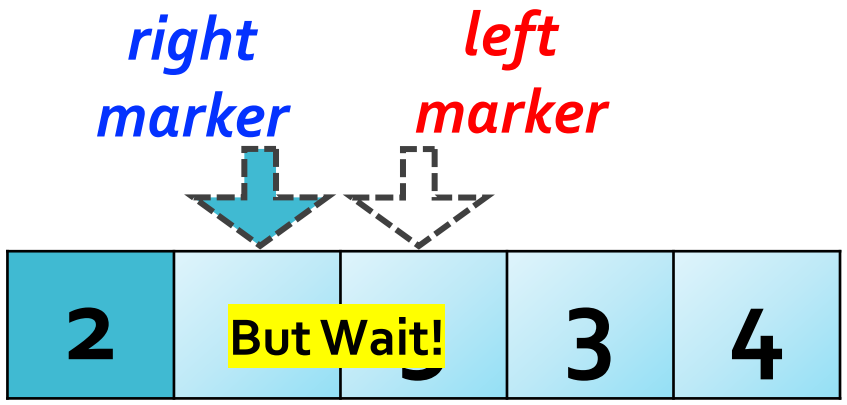


***right marker*:**

5 is not smaller than 2 (pivot):

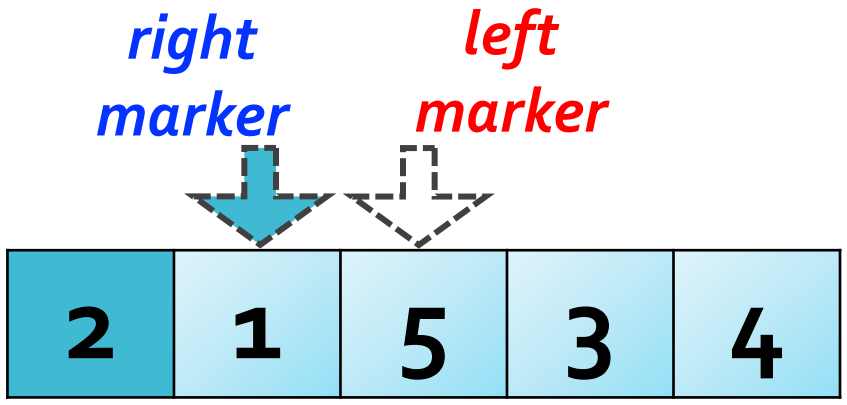
move!

Quick Sort



right: *1 is smaller than 2 (pivot):*
swap!

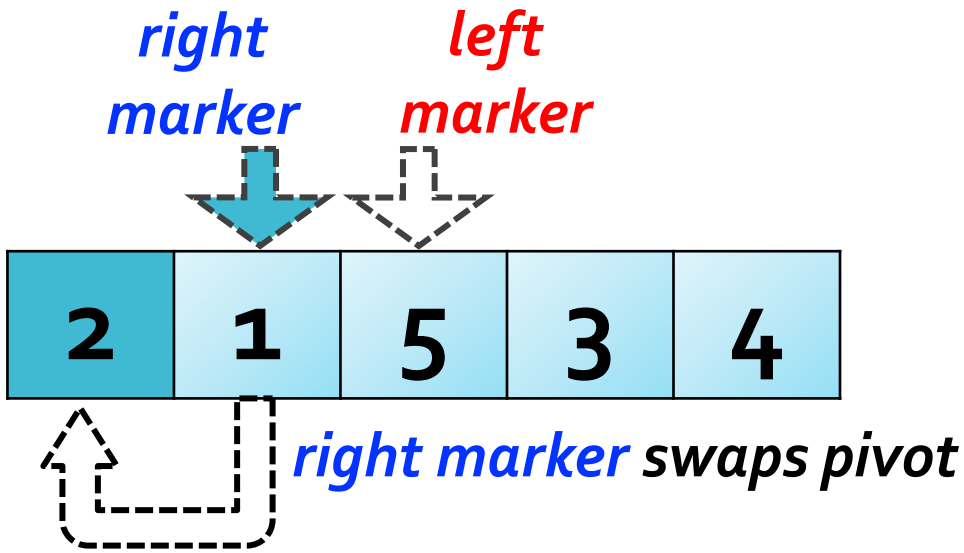
Quick Sort



right marker crossed *red marker*

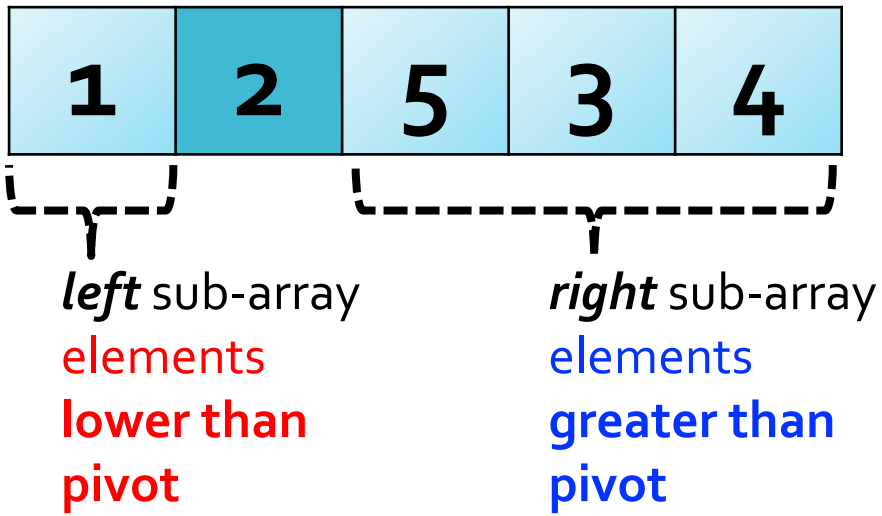
Then, *loop ends!*

Quick Sort

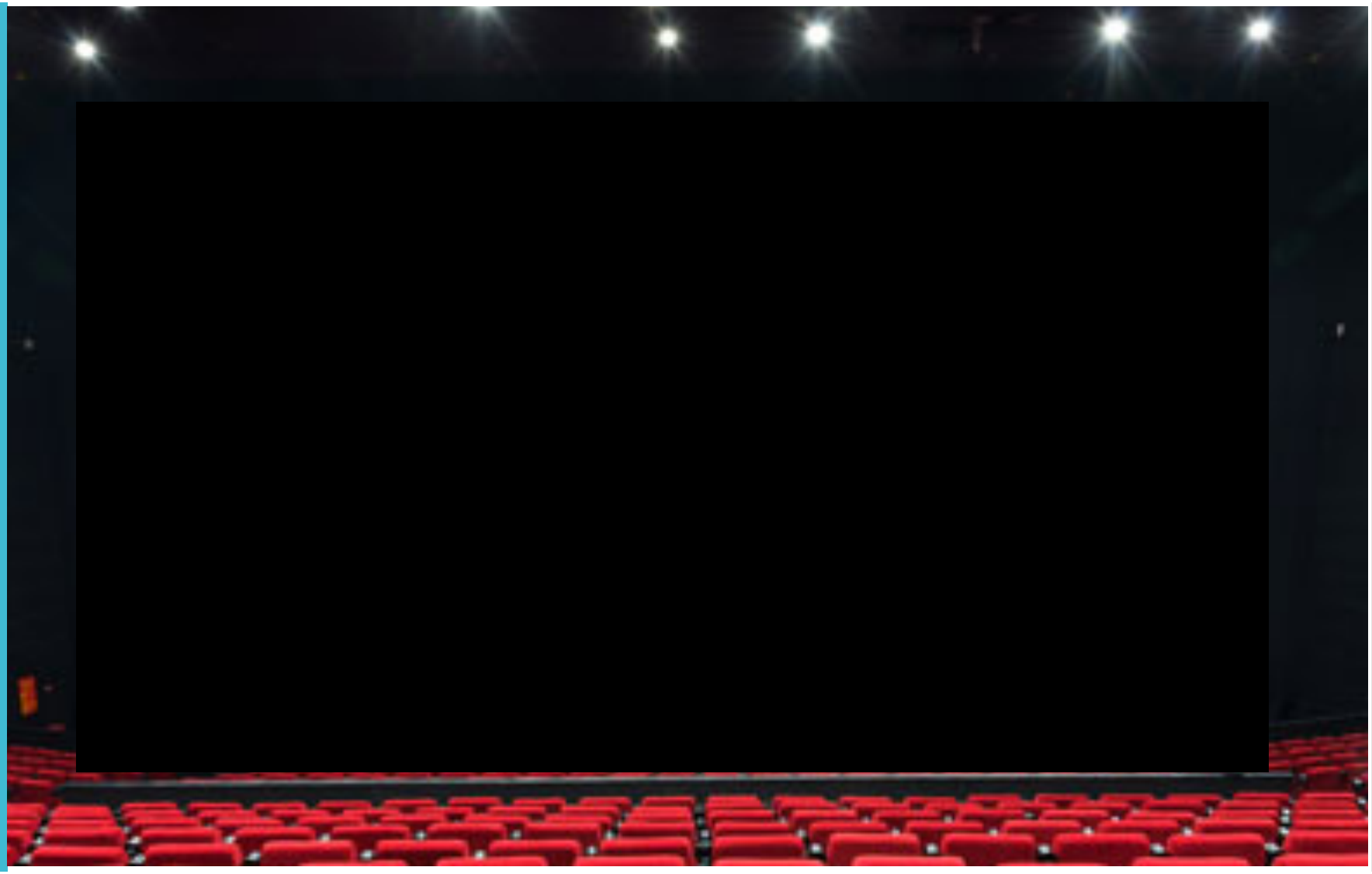


Quick Sort

Now you **call Quick sort** on the ***left*** and ***right*** **sub-arrays.**
At the end, **the array gets sorted.**



Bubble sort
vs
Quick Sort



ref : Youtube

Quick Sort

- Implementation of the **Quick sort**

```
def quick_sort(arr):

    if len(arr) < 2:
        return arr

    pivot = arr[0]
    left = [i for i in arr[1:] if i <= pivot]
    right = [i for i in arr[1:] if i > pivot]

    return quick_sort(left) + [pivot] + \
           quick_sort(right)
```

```
music_count = [156, 141, 35, 94, 88, 61, 111]
music_sorted = quick_sort(music_count)

print(music_sorted)
```

[Output]: `[35, 61, 88, 94, 111, 141, 156]`

Quick Sort

❖Implementation of the **Quick sort**

```
def quick_sort(arr):

    if len(arr) < 2:
        return arr

    pivot = arr[0]
    left = [i for i in arr[1:] if i <= pivot]
    right = [i for i in arr[1:] if i > pivot]

    return quick_sort(left) + [pivot] + \
           quick_sort(right)

music_count = [156, 141, 35, 94, 88, 61, 111]
music_sorted = quick_sort(music_count)

print(music_sorted)
```

[Output]: `[35, 61, 88, 94, 111, 141, 156]`

list comprehension

- always returns a **result list**.
- **concise way** to create lists.

- the following:

```
[expression for item in list if condition]
```

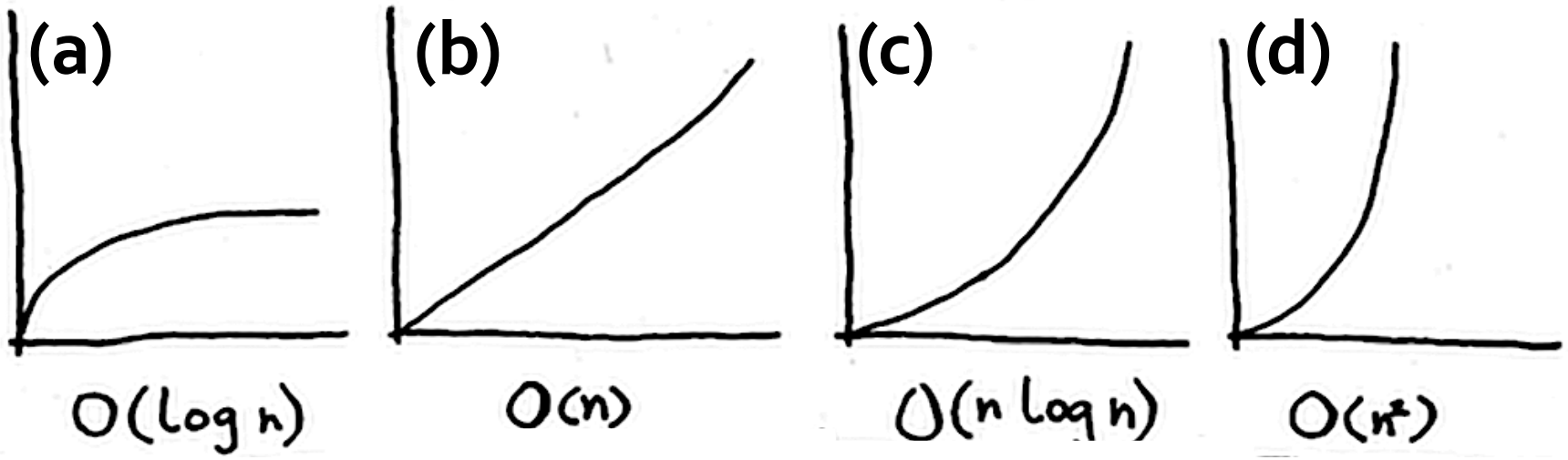
- is **equivalent** to

```
for item in list:
    if condition:
        expression
```

Quiz

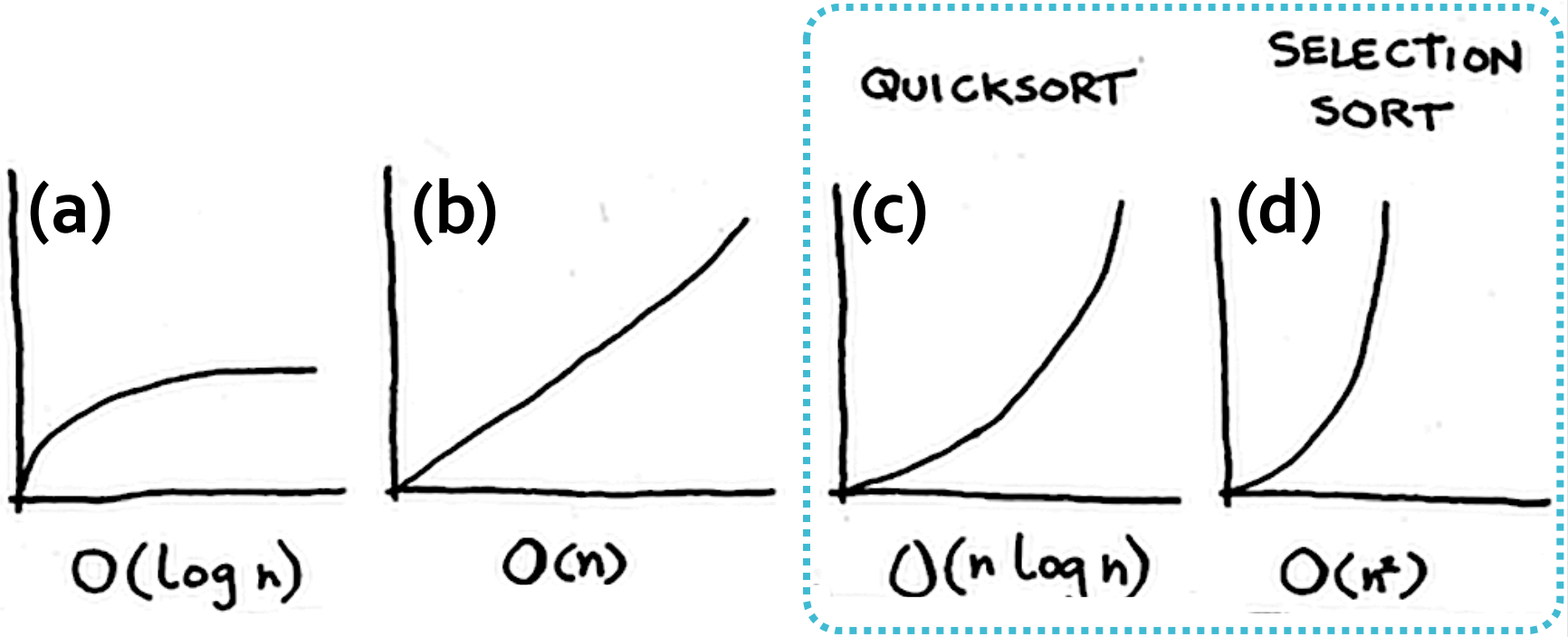
- Which of the following figures could represent the **Big O** notation of for:

1) **Quick sort**
2) **Selection sort**



Answer

- Which of the following figures could represent the **Big O** notation of for:

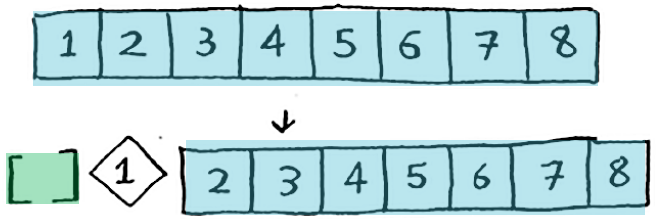


Quick Sort

- Quick sort is good, but its performance heavily **depends on** the chosen **pivot**.
- Lets assume a list **already sorted**:

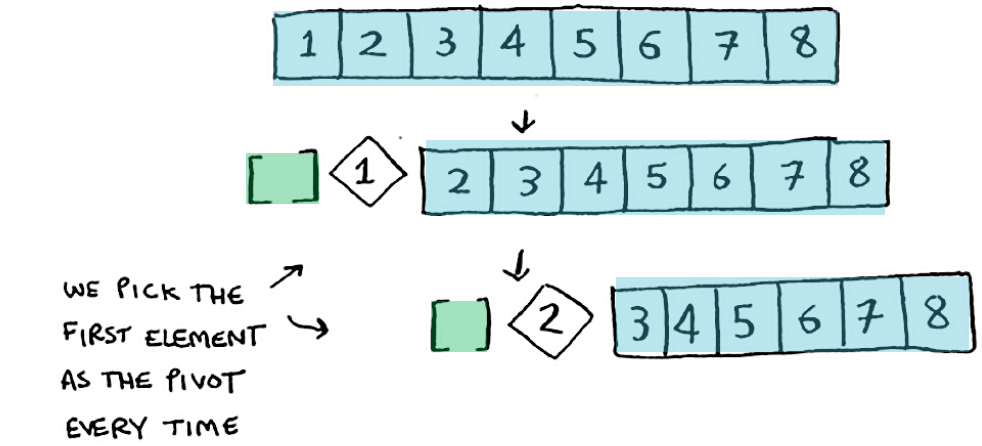
Quick Sort

- Performance of quicksort heavily **depends on** the chosen **pivot**.



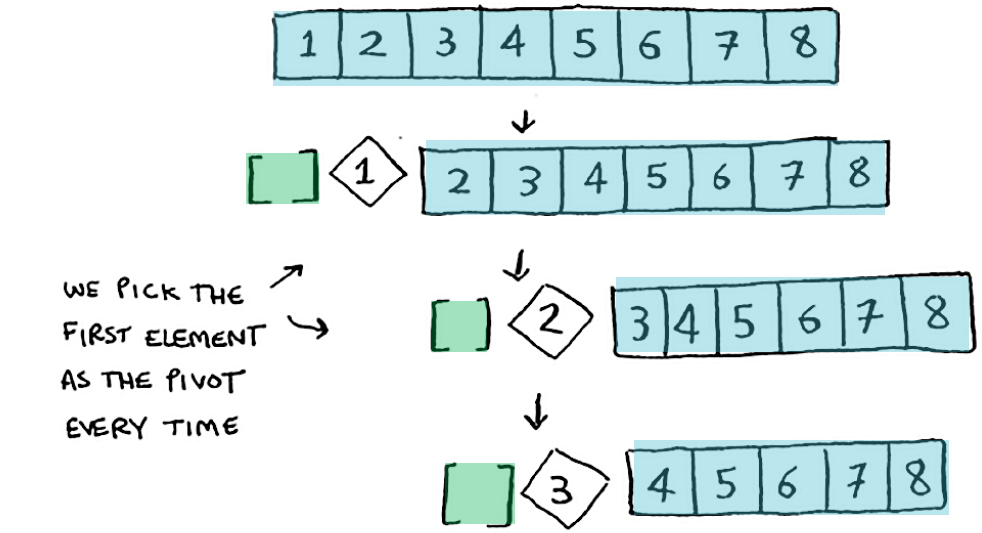
Quick Sort

- Performance of quicksort heavily **depends on** the chosen **pivot**.



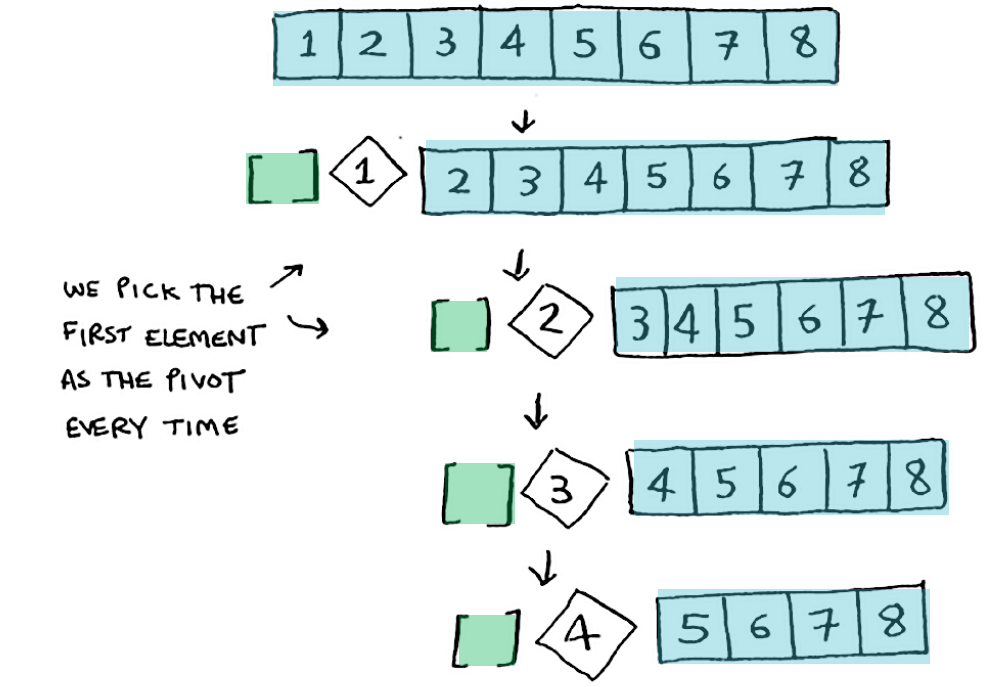
Quick Sort

- Performance of quicksort heavily **depends on** the chosen **pivot**.



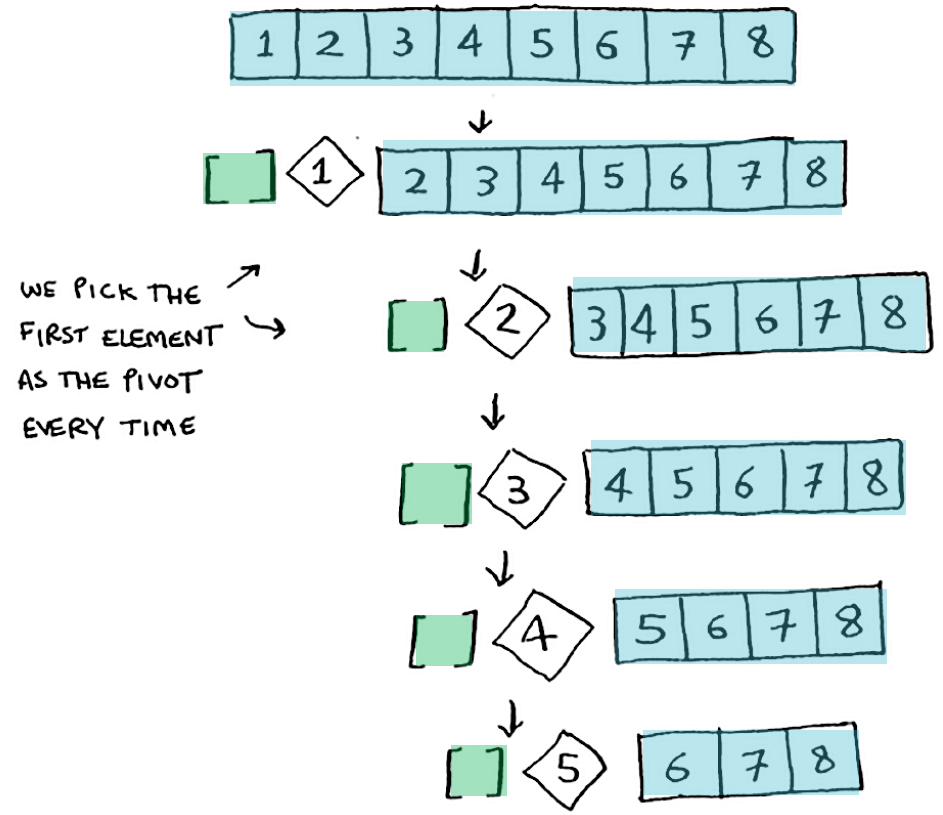
Quick Sort

- Performance of quicksort heavily **depends on** the chosen **pivot**.



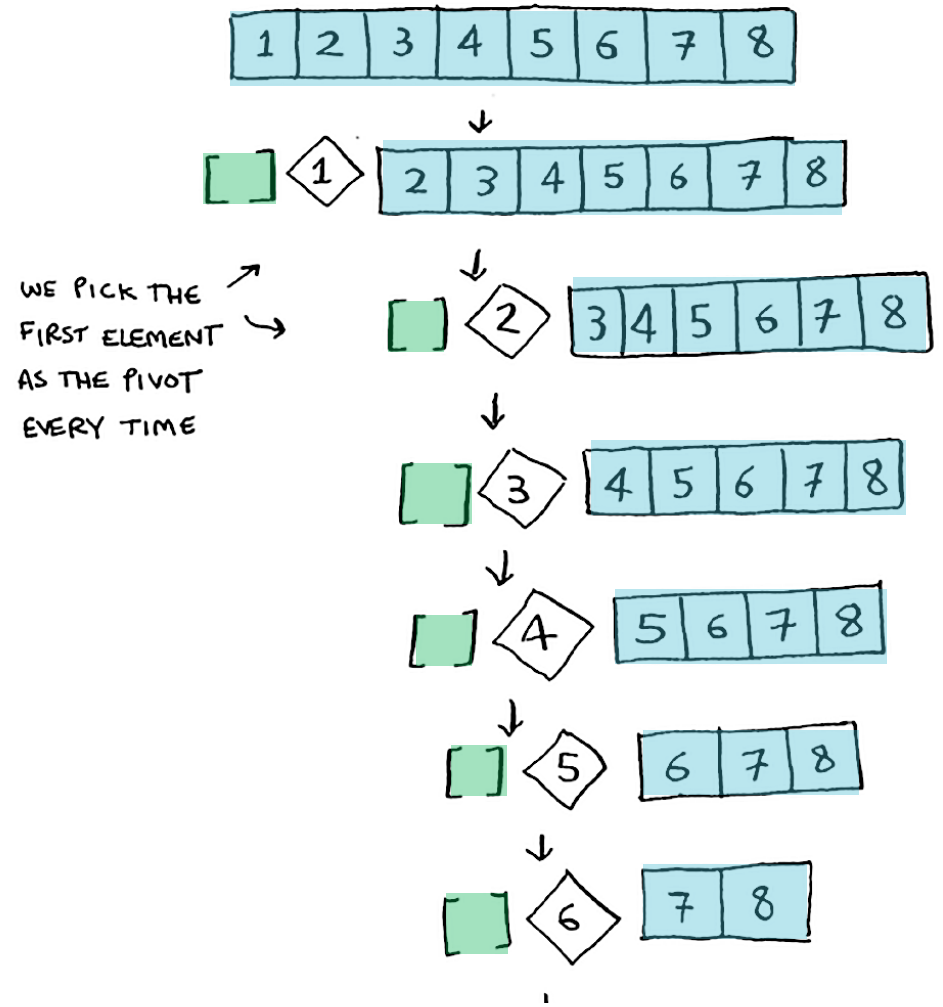
Quick Sort

- Performance of quicksort heavily **depends on** the chosen **pivot**.



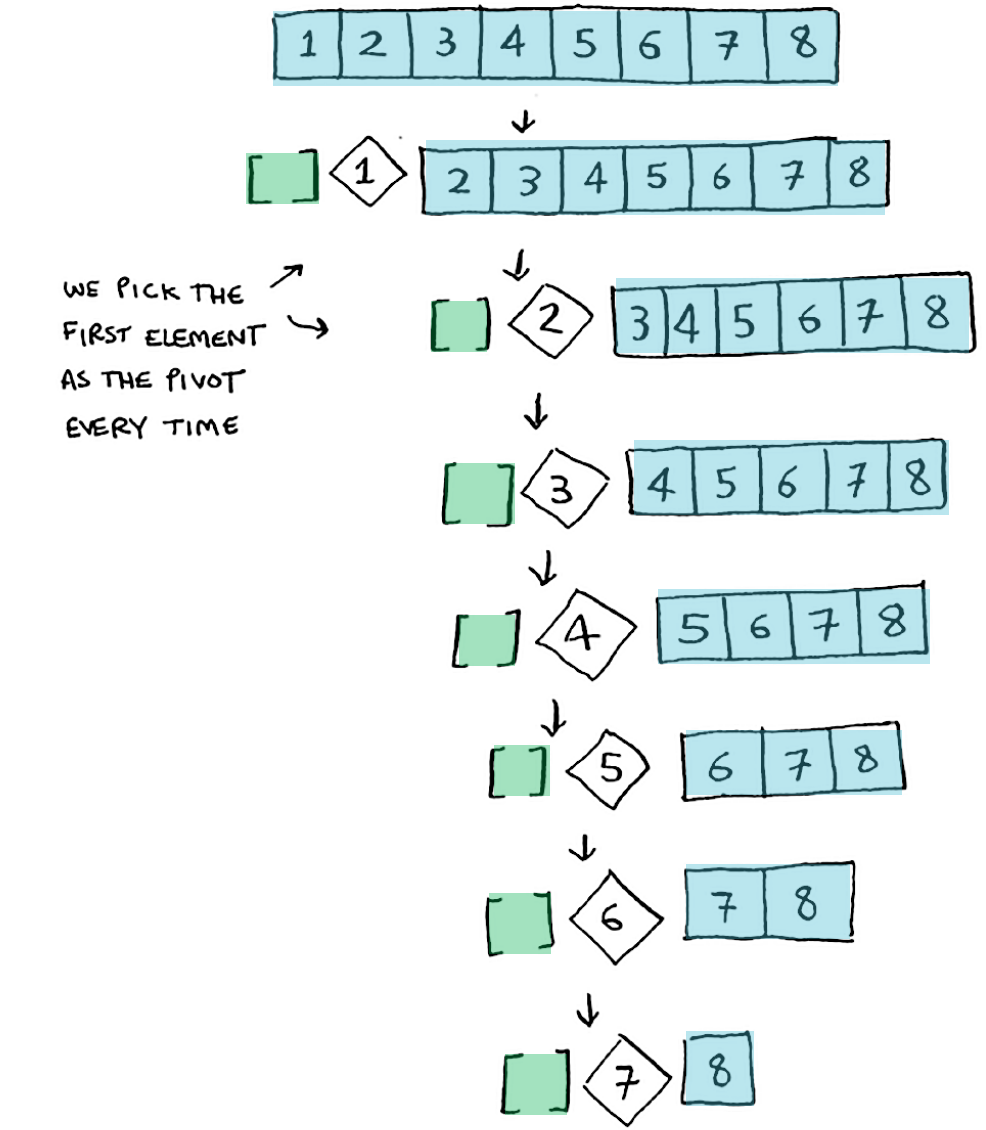
Quick Sort

- Performance of quicksort heavily **depends on** the chosen **pivot**.



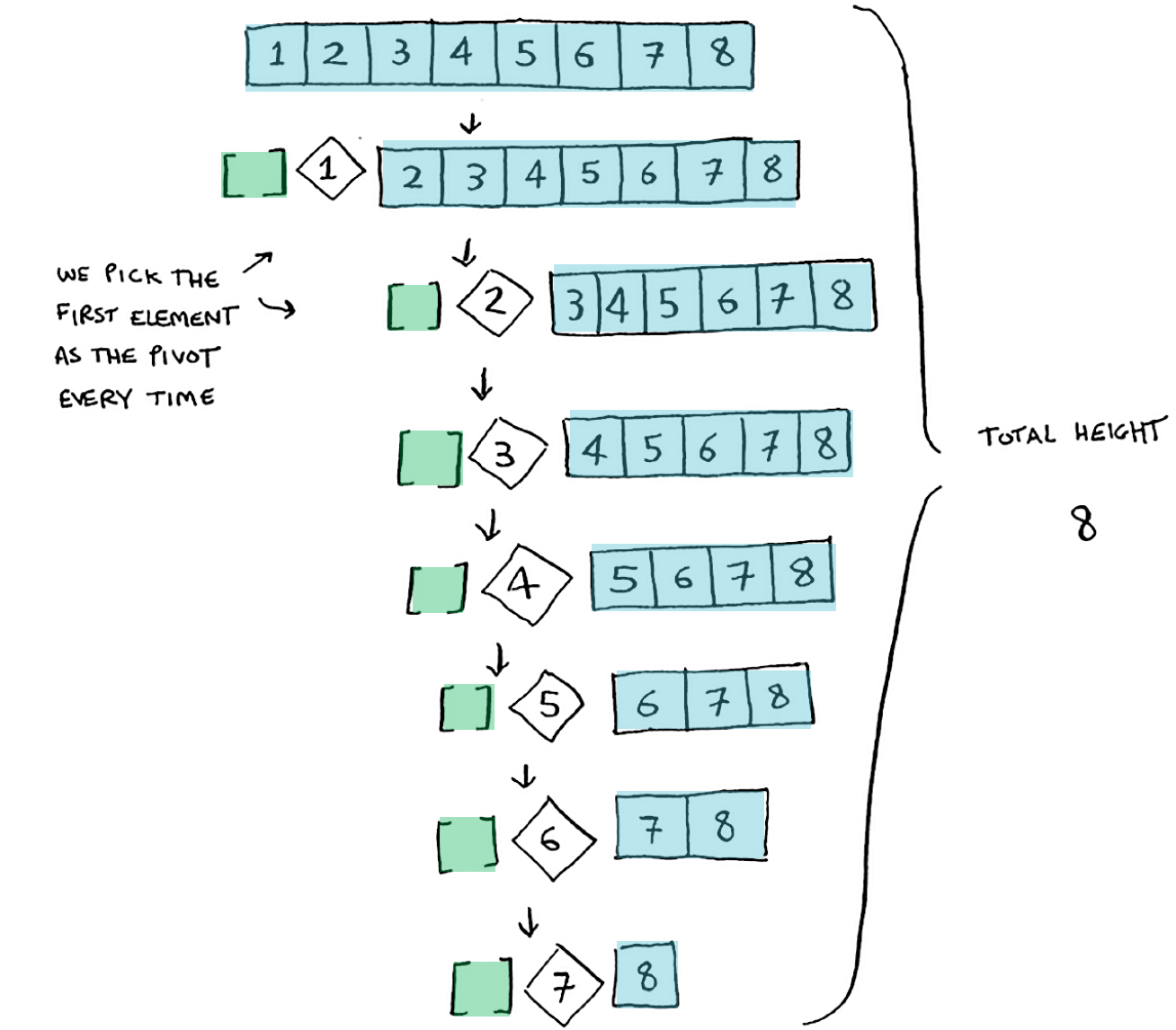
Quick Sort

- Performance of quicksort heavily **depends on** the chosen **pivot**.



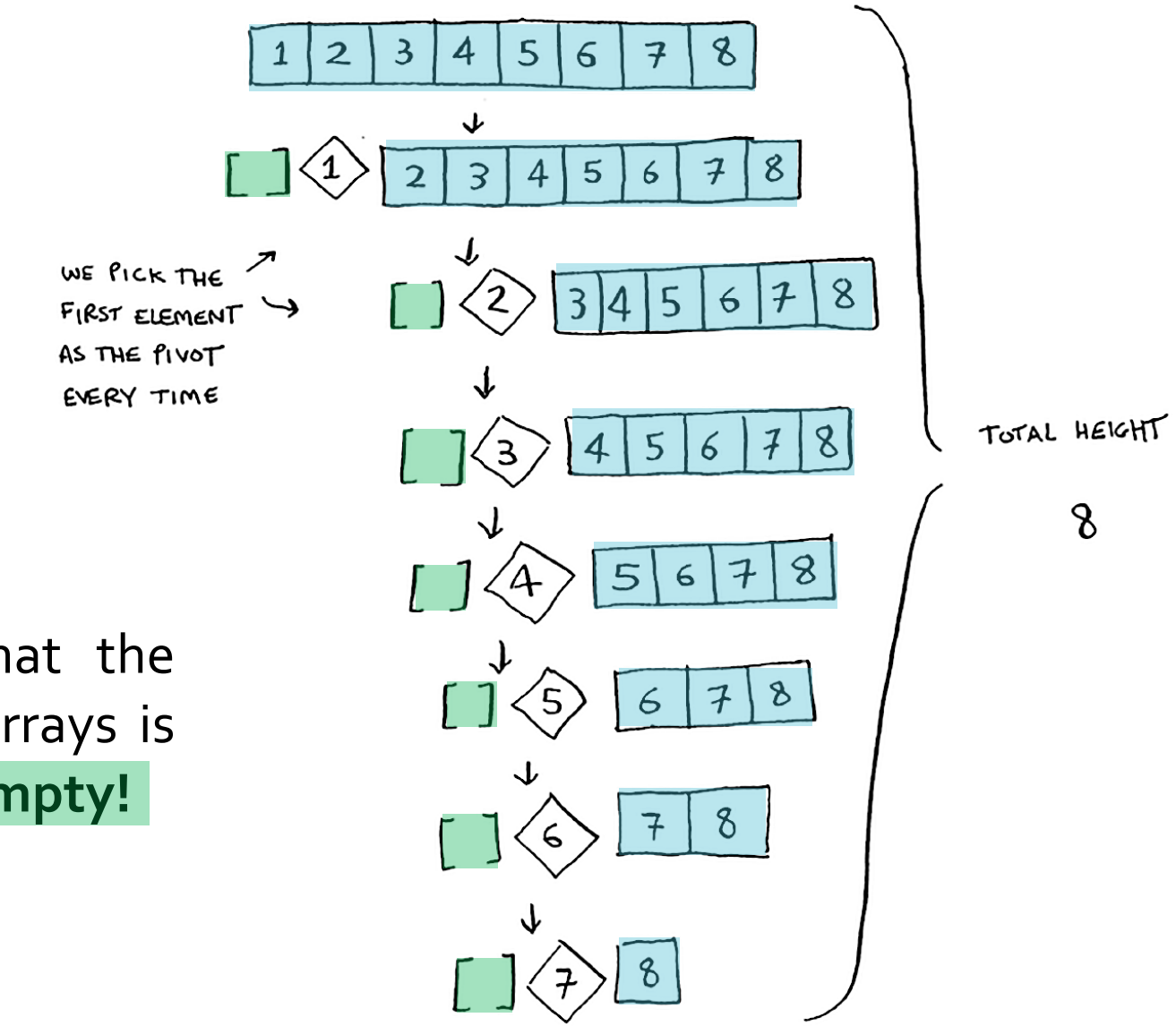
Quick Sort

- Performance of quicksort heavily **depends on** the chosen **pivot**.



Quick Sort

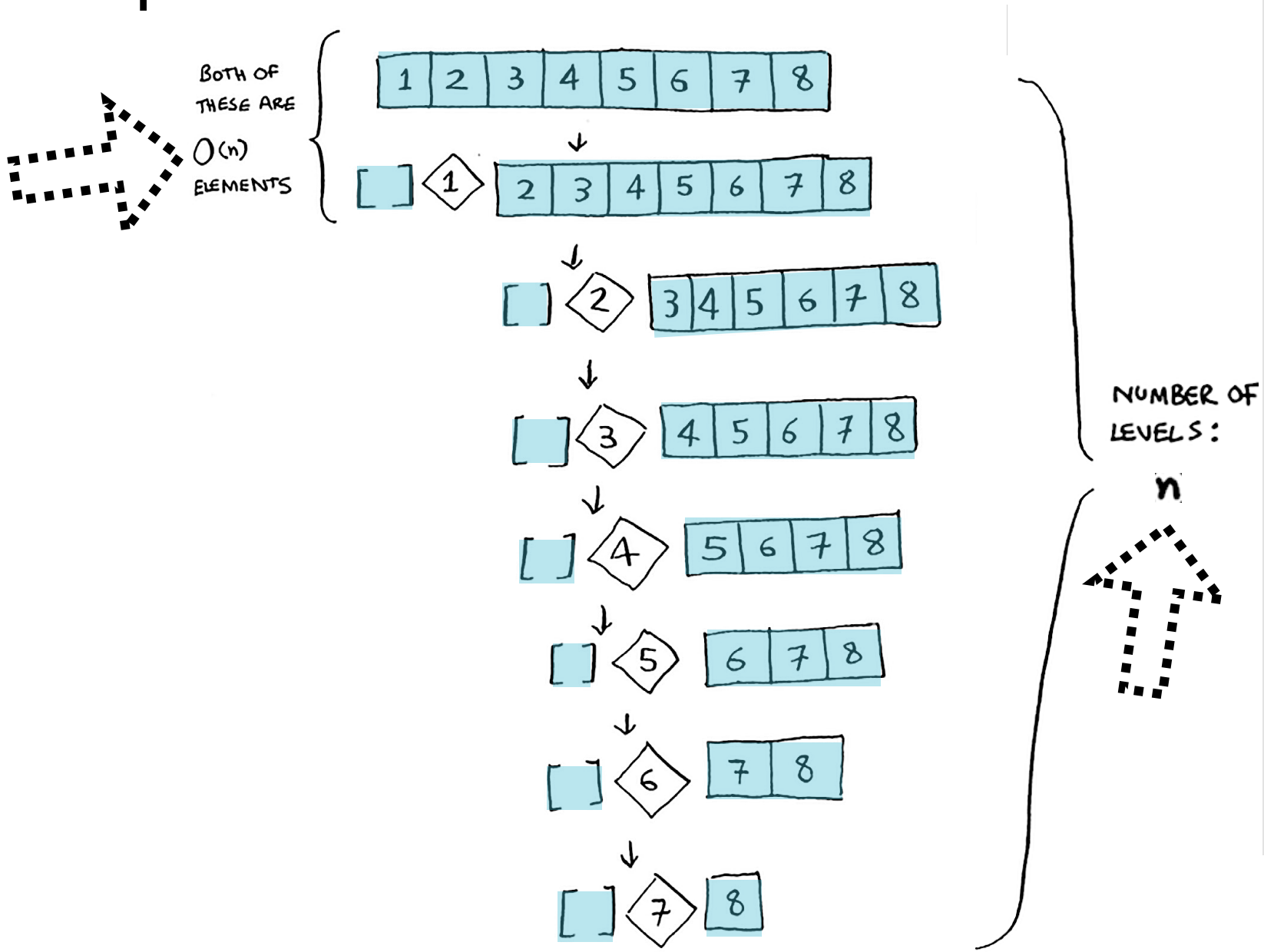
- Performance of quicksort heavily **depends on** the chosen **pivot**.



Notice that the *left* sub-arrays is **always empty!**

Quick Sort

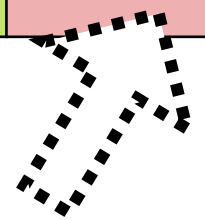
- Performance of quicksort heavily **depends on** the chosen **pivot**.



Quick Sort

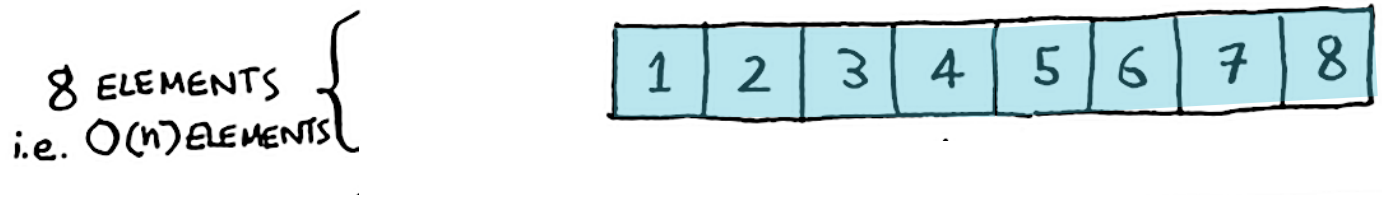
- Performance of quicksort heavily **depends on** the chosen **pivot**.

	Best case	Average case	Worse case
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$



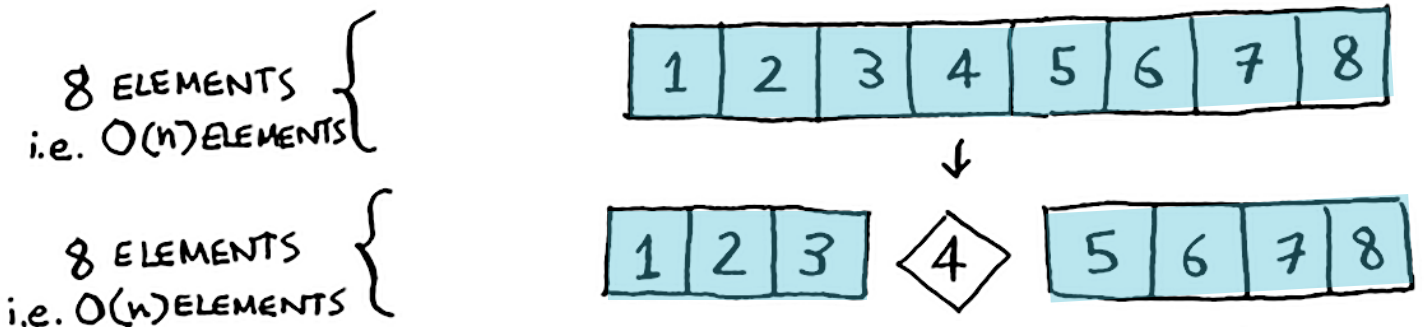
Quick Sort

- Even if you partition the array differently, you're still touching **O(n) elements every time**.



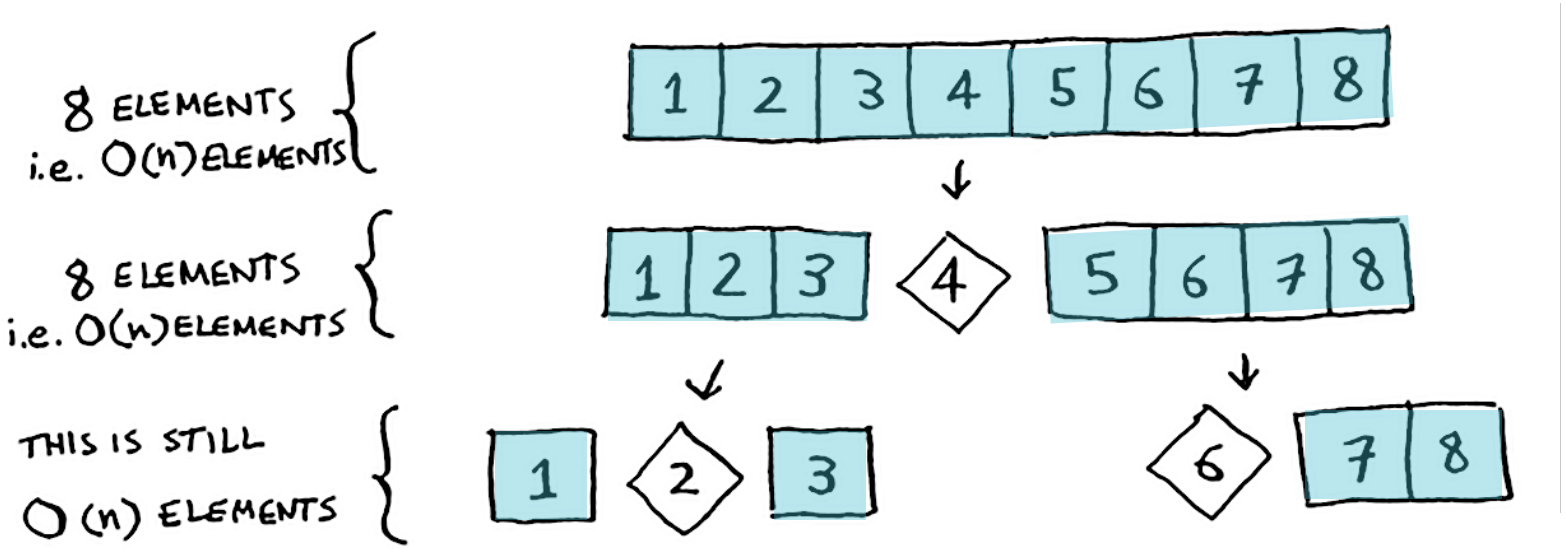
Quick Sort

- Even if you partition the array differently, you're still touching **O(n) elements every time.**



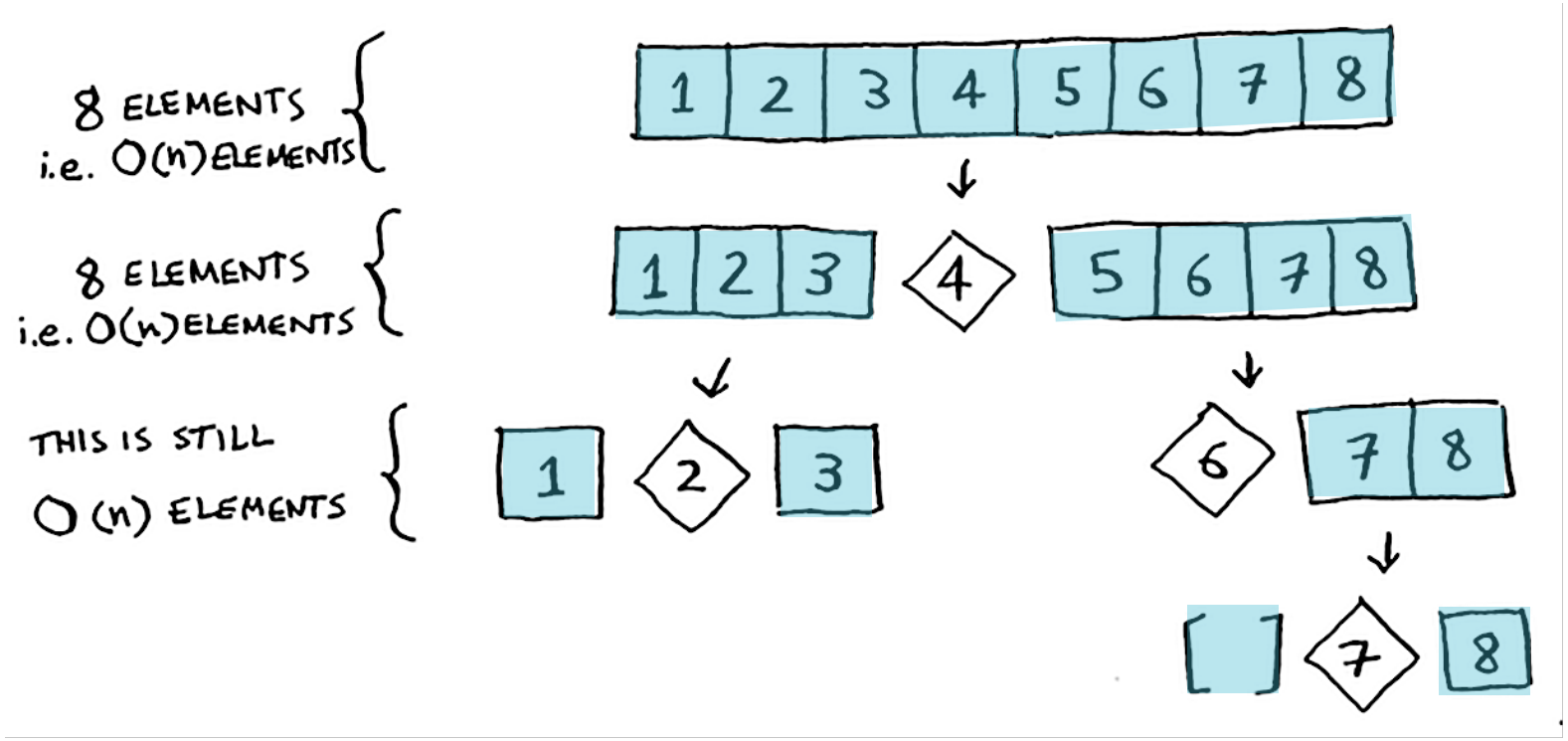
Quick Sort

- Even if you partition the array differently, you're still touching **O(n) elements every time.**



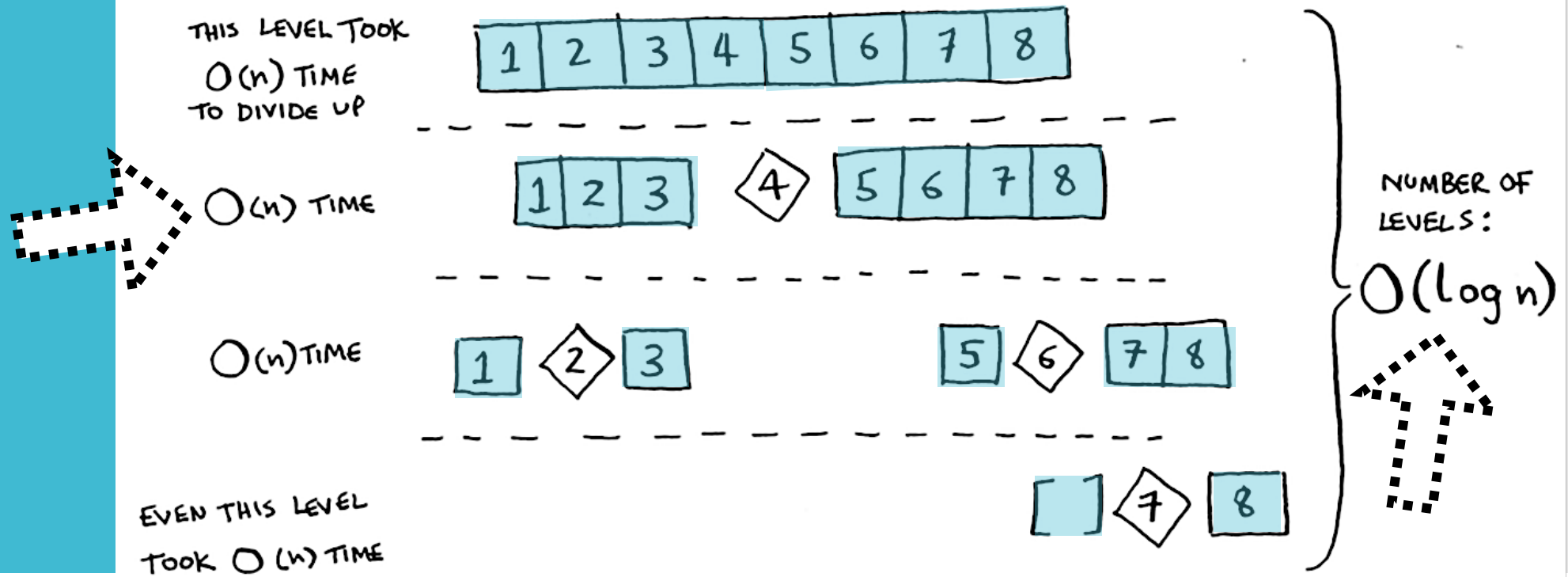
Quick Sort

- Even if you partition the array differently, you're still touching **O(n) elements every time.**



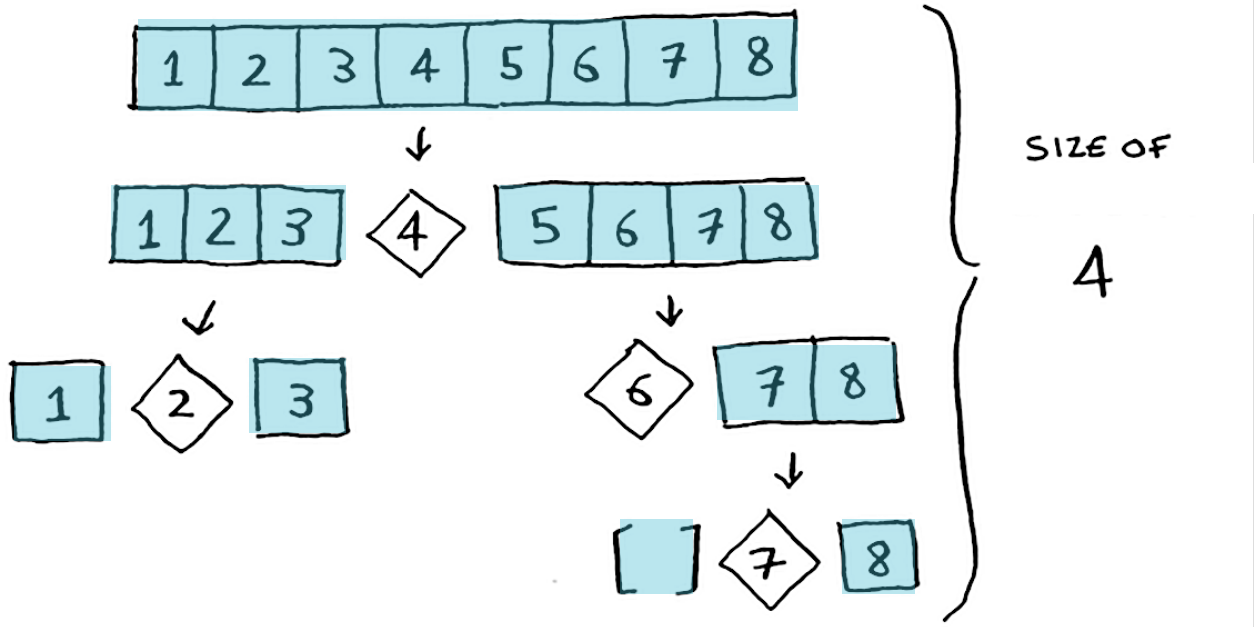
Quick Sort

- Even if you partition the array differently, you're still **touching O(n)** elements every time.



Quick Sort

- It's much **quicker** now!
- Because the **array is divided into half** every time and no need to make many calls.
- Base case is hit **sooner**.



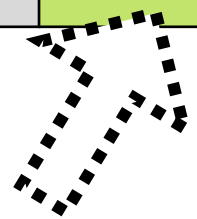
Quick Sort

- 1st example is the **worst-case scenario:**
  - **O(n*n) = O(n²)**
  - all elements are checked and this operation takes **O(n) time**, repeated "**n" times**.
- 2nd example is the **best-case scenario:**
  - **O(n log n)**
- All elements are checked and this operation takes **O(n) time**. And this is repeated "**log n" times**.

Quick Sort

- Performance of quicksort heavily **depends on** the chosen **pivot**.

	Best case	Average case	Worse case
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$



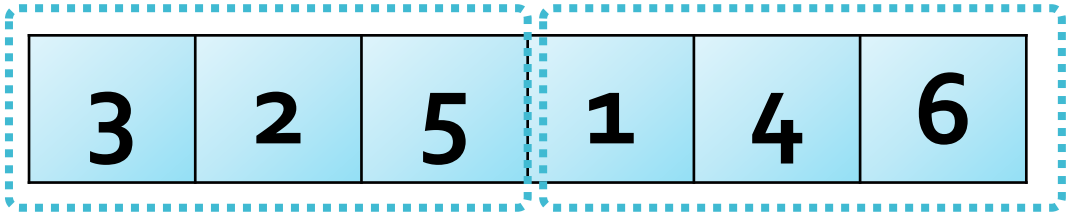
Quick Sort

	Best case	Average case	Worse case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$

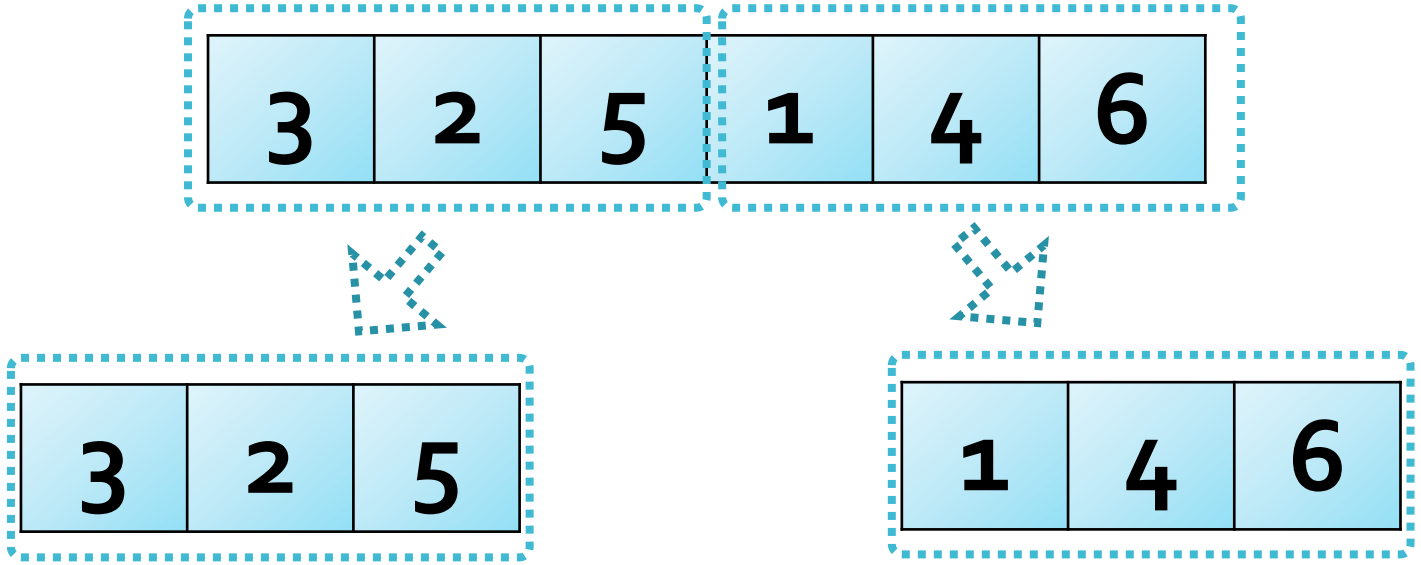
Merge Sort

- **Merge Sort** is a sorting algorithm developed based on uses **Divide & Conquer.**
- It is faster than other sorts (e.g., selection sort).
- Developed by Hungarian-American computer scientist **John von Neumann** in 1945.

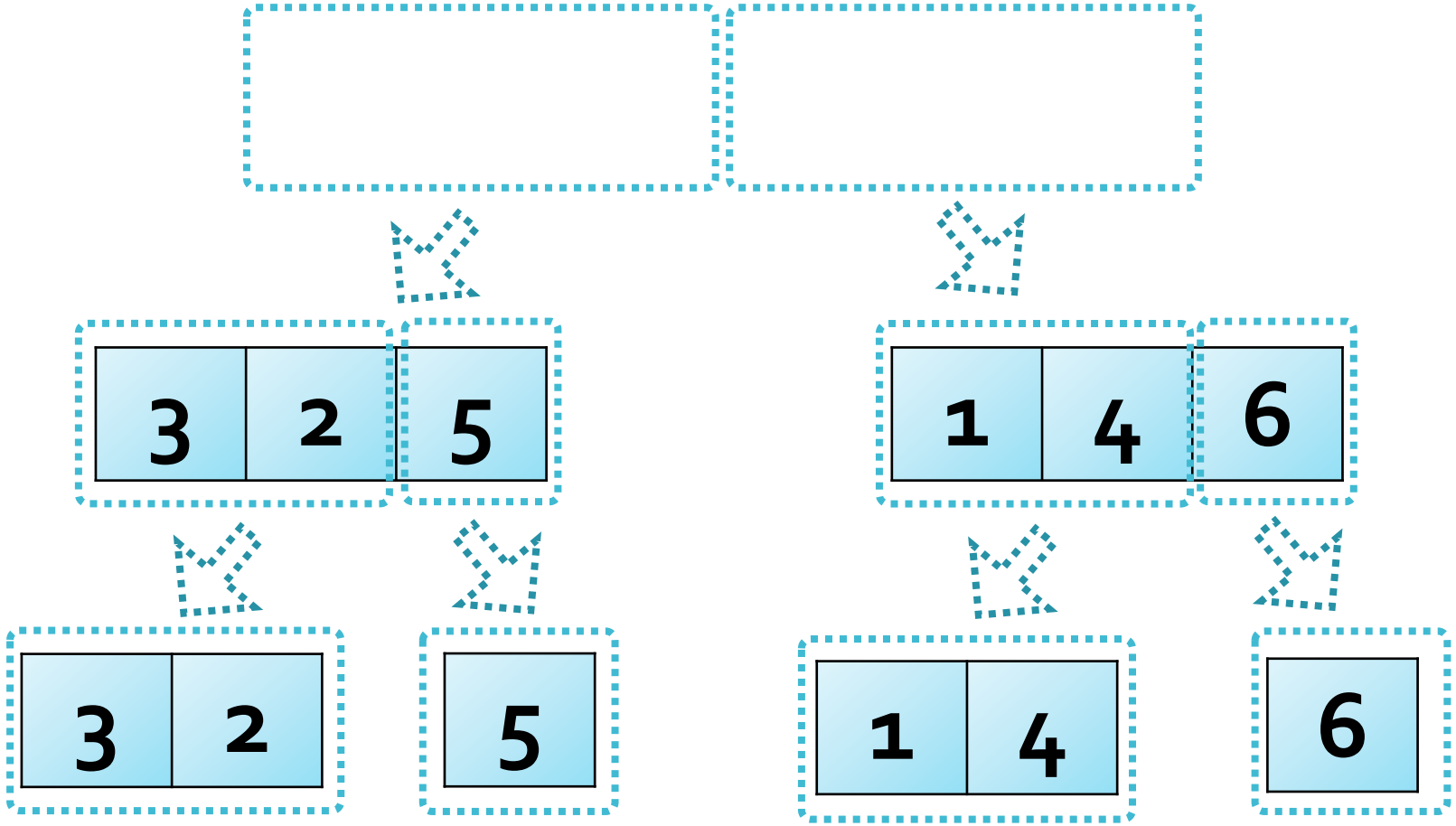
Merge sort



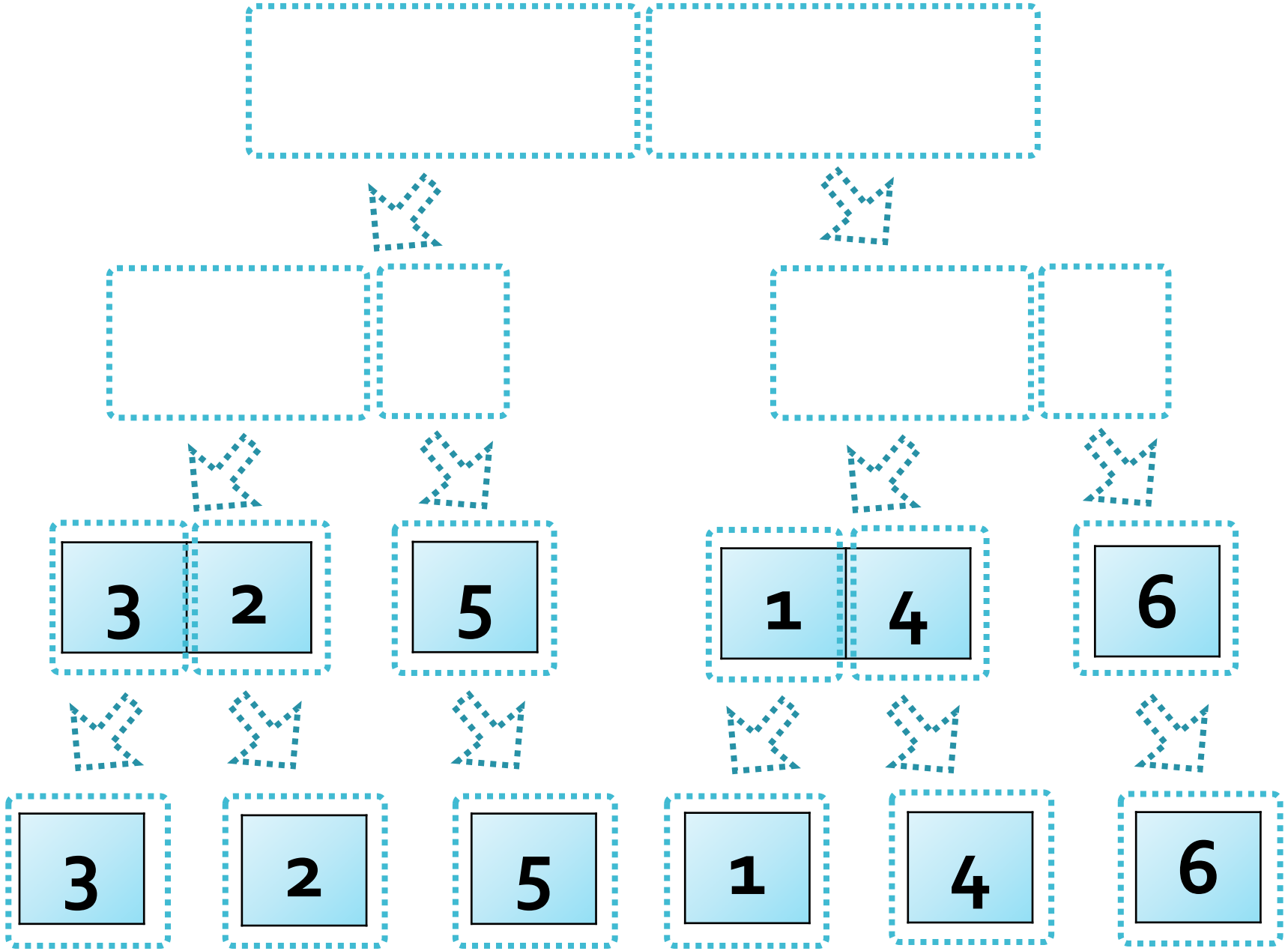
Merge sort



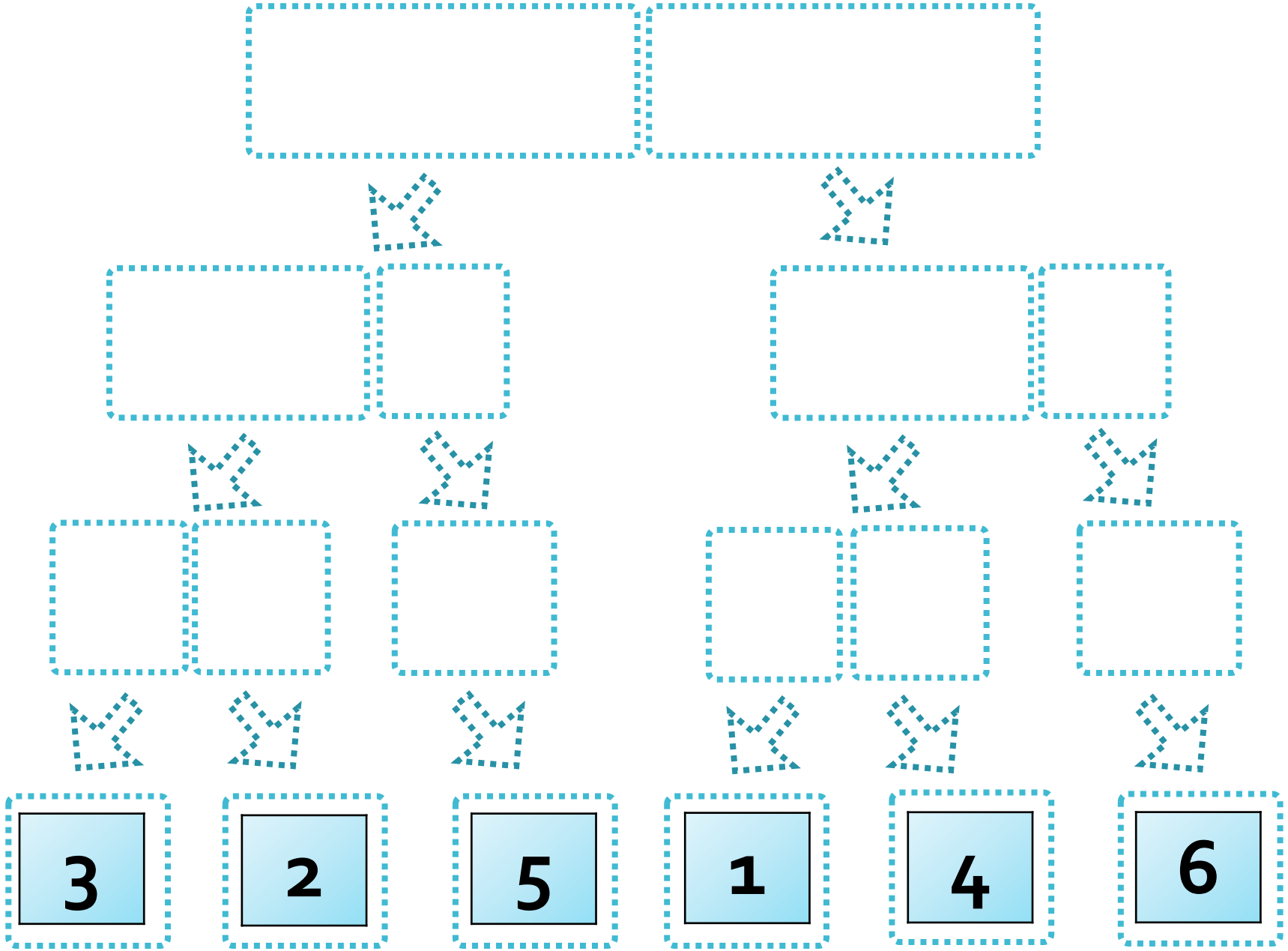
Merge sort



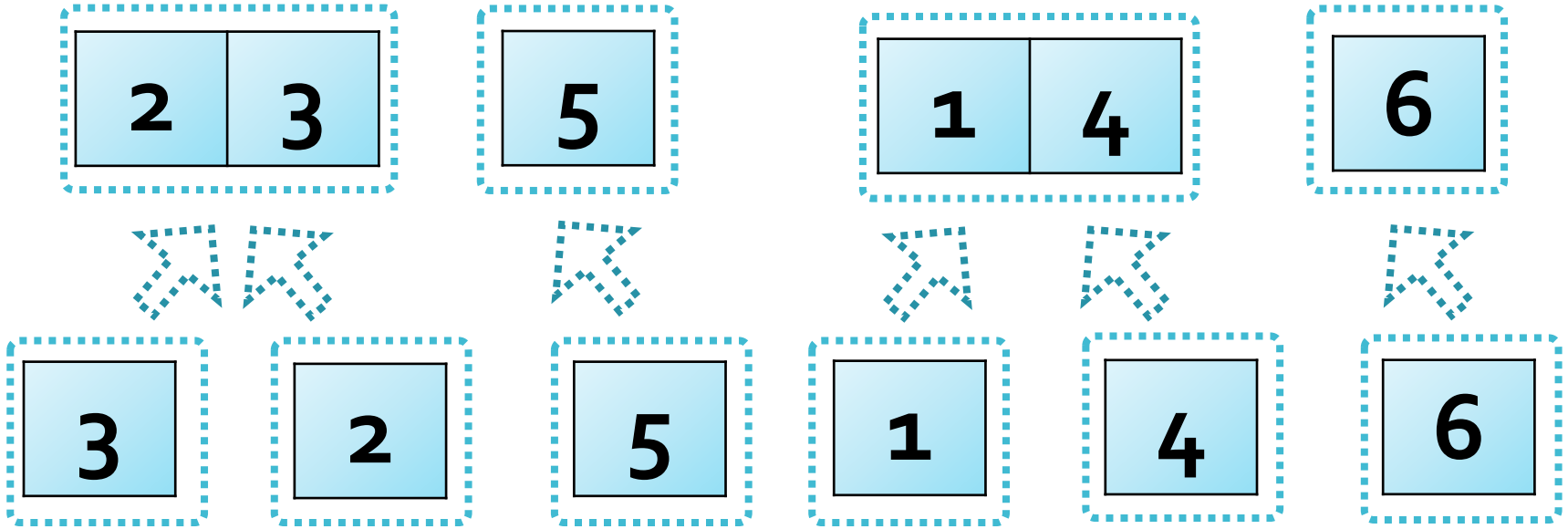
Merge sort



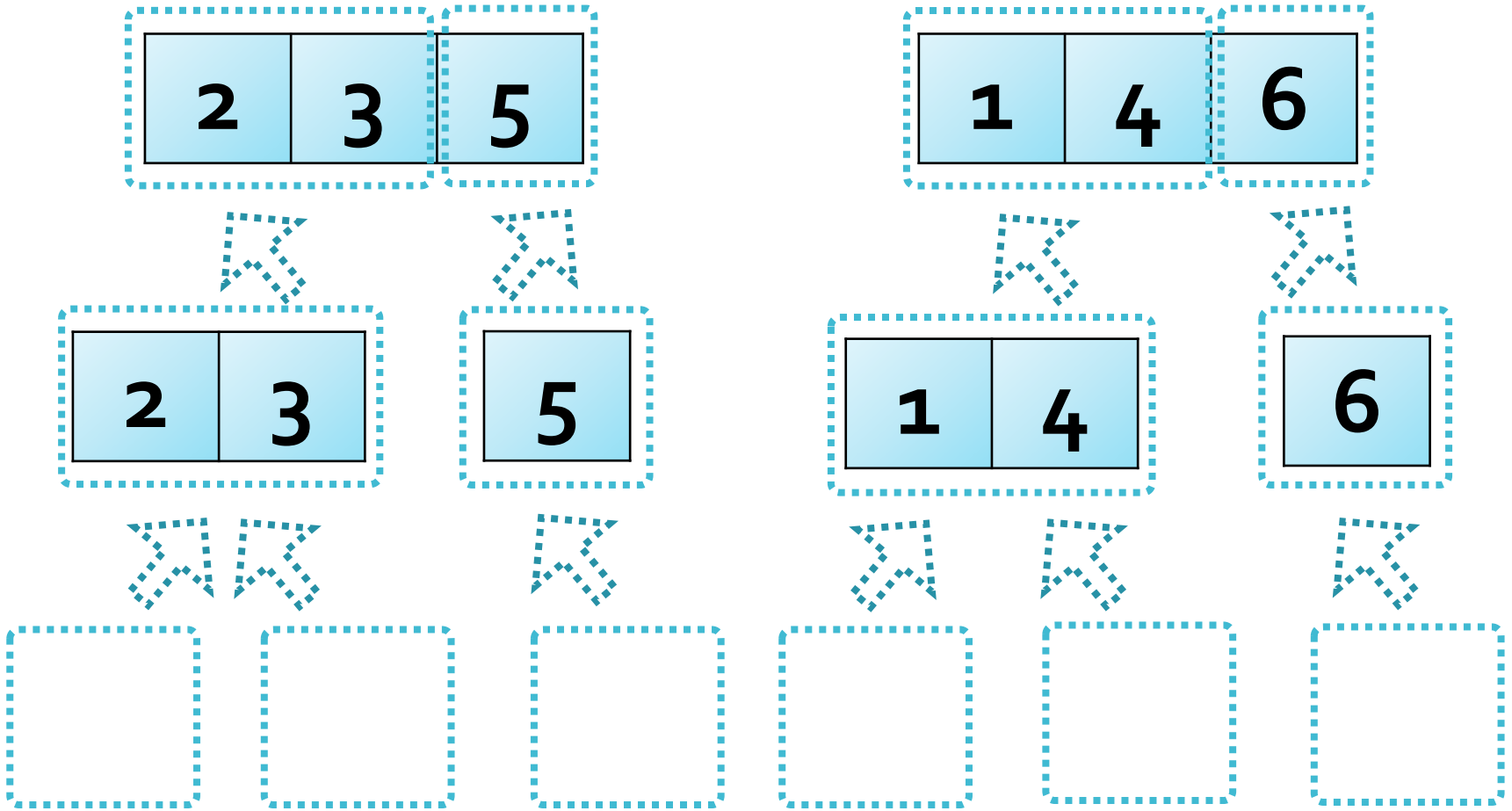
Merge sort



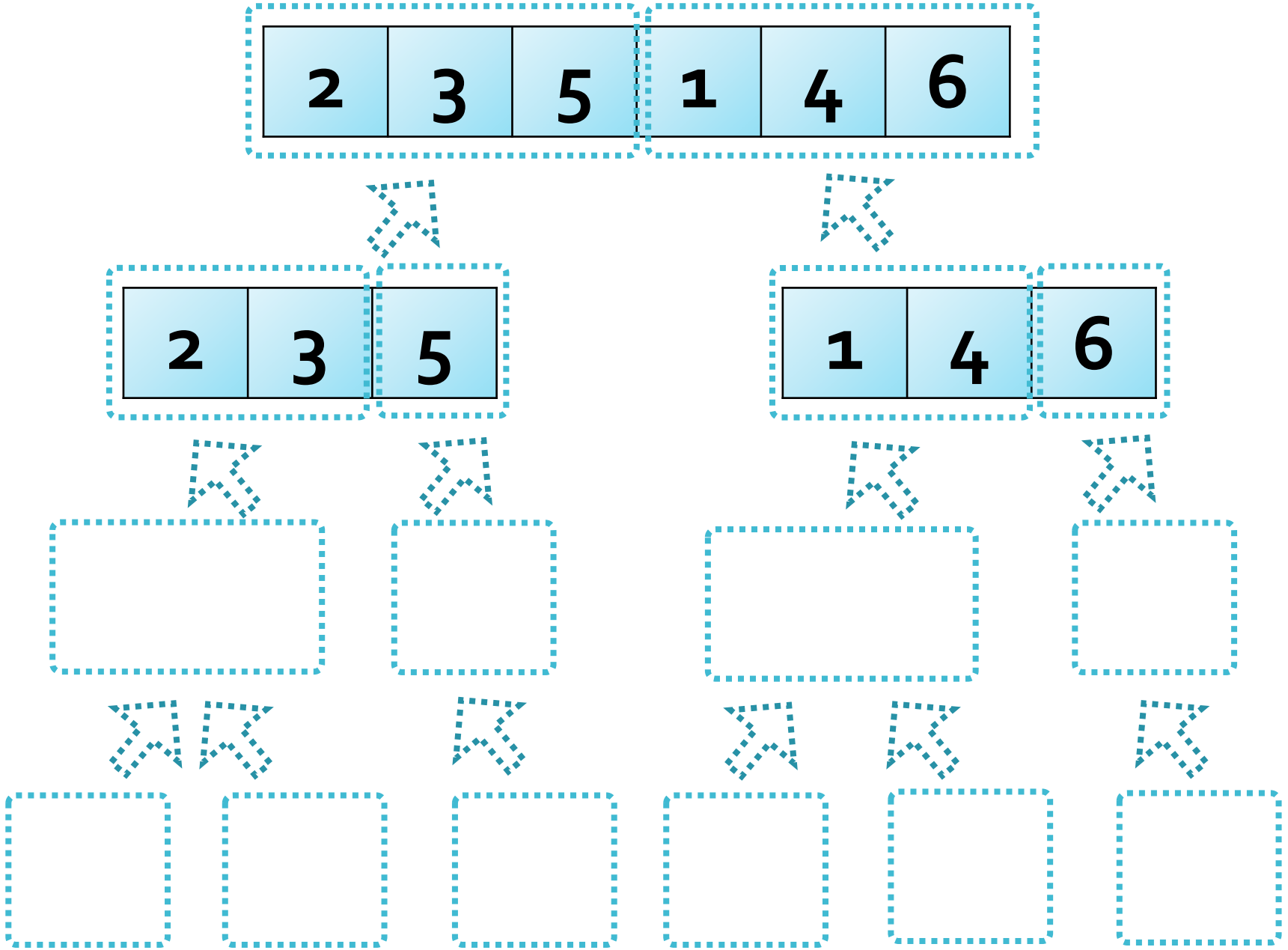
Merge sort



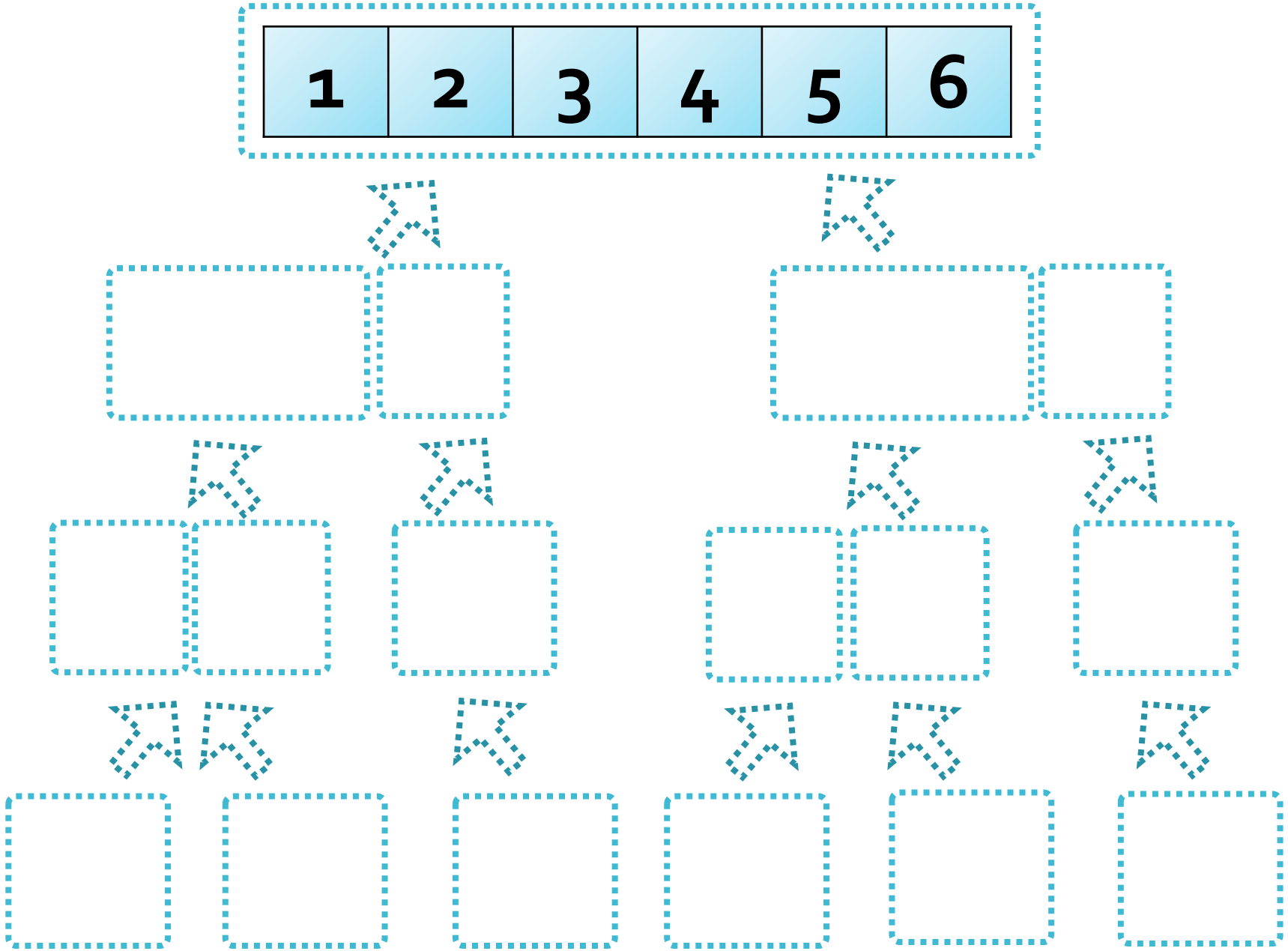
Merge sort



Merge sort



Merge sort



Merge Sort (part 1)

- Implementation of the **Merge sort:**

Part 1

```
def merge_sort(arr):

    if len(arr) > 1:
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]
        merge_sort(left)
        merge_sort(right)
```

Merge Sort (part 2)

Part 2

```
    i = j = k = 0
    while i < len(left) and j < len(right):

        if left[i] < right[j]:
            arr[k] = left[i]
            i += 1
        else:
            arr[k] = right[j]
            j += 1
        k += 1

    while i < len(left):
        arr[k] = left[i]
        i += 1
        k += 1

    while j < len(right):
        arr[k] = right[j]
        j += 1
        k += 1
```

Merge Sort (part 3)

```
i = j = k = 0
while i < len(left) and j < len(right):

    if left[i] < right[j]:
        arr[k] = left[i]
        i += 1
    else:
        arr[k] = right[j]
        j += 1
    k += 1
```

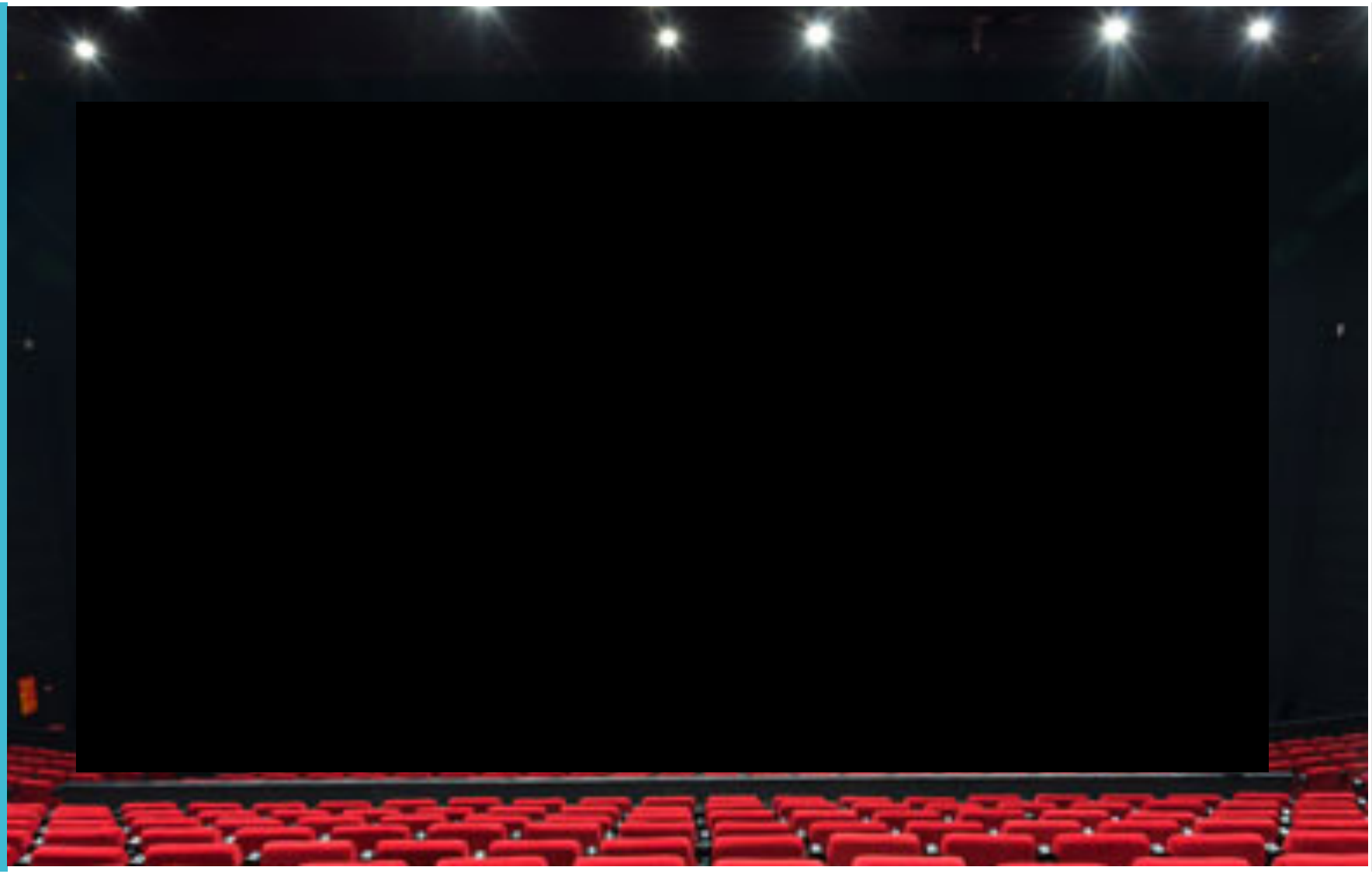
```
while i < len(left):
    arr[k] = left[i]
    i += 1
    k += 1

while j < len(right):
    arr[k] = right[j]
    j += 1
    k += 1
```

Merge sort

	Best case	Average case	Worse case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)

Merge sort
vs
Quick Sort



ref : Youtube

Quiz

- Given the following list:

[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]

- which of the following is the list after **three recursive calls** to Merge sort?

1) [16, 49, 39, 27, 43, 34, 46, 40]
2) [21, 1]
3) [21, 1, 26, 45]
4) [21]

Answer

- which of the following is the list after **three recursive calls** to Merge sort?

[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]

1) [16, 49, 39, 27, 43, 34, 46, 40]
2) [21, 1]
3) [21, 1, 26, 45]
4) [21]

Answer

- which of the following is the list after **three recursive calls** to Merge sort?

[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]

1) [16, 49, 39, 27, 43, 34, 46, 40]
2) [21, 1]
3) [21, 1, 26, 45]
4) [21]

Answer

- which of the following is the list after **three recursive calls** to Merge sort?

[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]

1) [16, 49, 39, 27, 43, 34, 46, 40]
2) **[21, 1]**
3) [21, 1, 26, 45]
4) [21]

Next Lesson

- **Recursion**