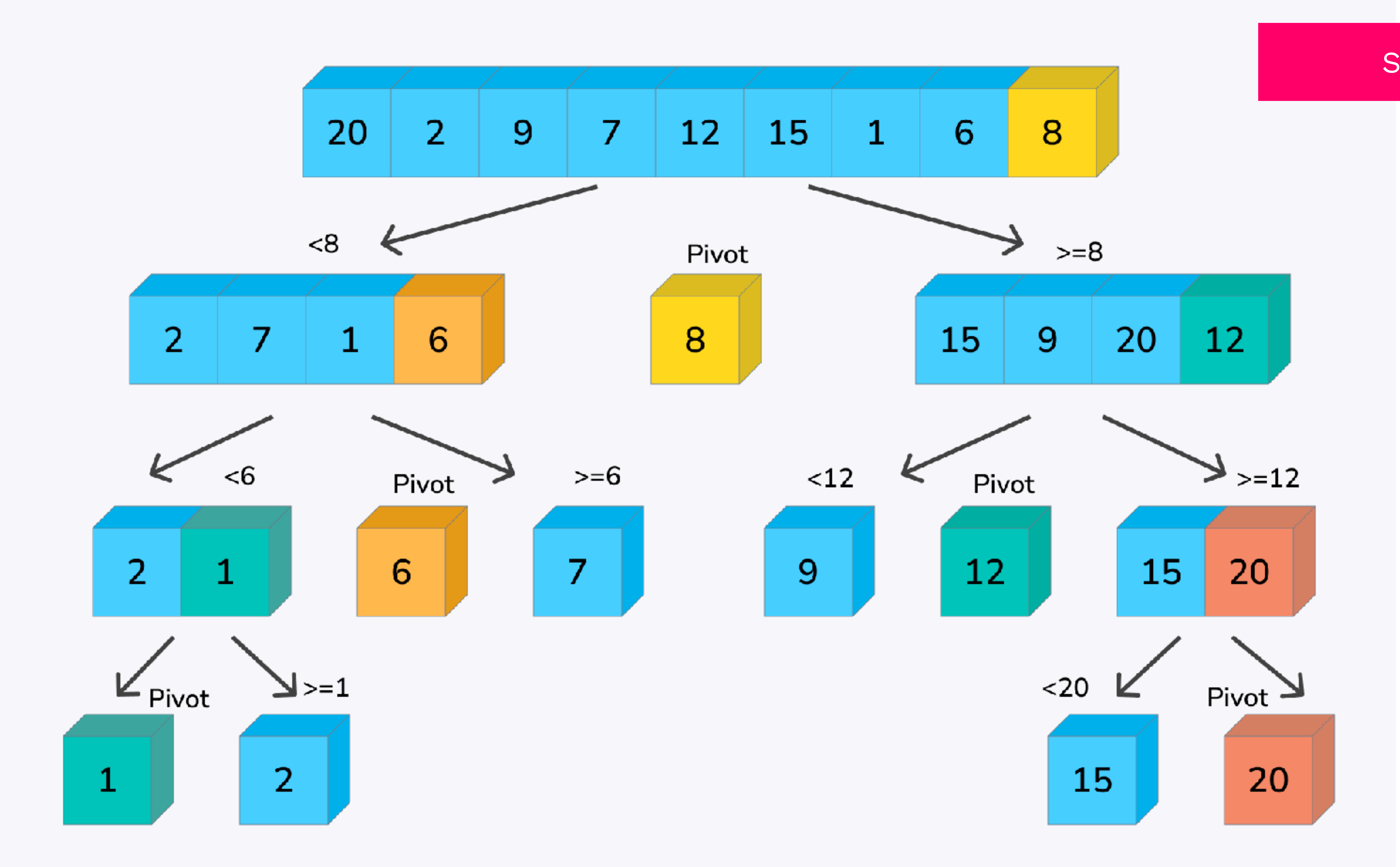
CS62 Class 12: Quicksort

Sorting



Last time review

	In place	Stable	Best	Average	Worst	Remarks
Selection	X		$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	n exchanges
Insertion	X	X	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	Use for small arrays or partially ordered
Merge sort		X	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	Guaranteed performance; stable

Perform the first 2 steps of selection, insertion, and the merging of mergesort for the following array:

3	1	5	4	2

Last week review

Selection sort: select smallest element and swap

3	1	5	4	2

1	3	5	4	2

1	2	5	4	3

Insertion sort: insert next element into sorted left side subarray

3	1	5	4	2

1	3	5	4	2

1	2	3	5	4

Merge sort: merge halves

[3] [1] [5] [4] [2]

Single elements

[1 3] [5] [2 4]

Groups of 2 merged

[1 3 5] [2 4]

Group of 3 merged

Agenda

- Quicksort basics & demo
- Quicksort code
- Quicksort analysis

Quicksort basics

Quicksort live demo!

- I need 5-10 volunteers who want to be sorted by height.

Quicksort = pivots & partitions

- The main idea behind Quicksort is we pick a **pivot, x,** to **partition** the array such that:
  - All entries to the left of x are <= x (smaller).
  - All entries to the right of x are >= x (bigger).
  - x is in the right place in the final, sorted array.
- Then we sort each subarray (to the left and to the right) recursively.

input (pivot = 6)

6	8	3	1	2	7	4	9

example of valid output

3	1	2	4	6	9	8	7

also example of valid output

3	4	1	2	6	9	7	8

Worksheet time!

- The main idea behind Quicksort is we pick a **pivot, x,** to **partition** the array such that:
  - All entries to the left of x are <= x (smaller).
  - All entries to the right of x are >= x (bigger).
  - x is in the right place in the final, sorted array.

Which are valid partitions of this array if 10 is the pivot?

5	550	10	4	10	9	330

A

4	5	9	10	10	550	330

C

4	5	9	10	10	330	550

B

5	9	10	4	10	330	550

D

5	9	10	4	10	550	330

Worksheet Answers

Which are valid partitions of this array if 10 is the pivot?

5	550	10	4	10	9	330

- The main idea behind Quicksort is we pick a **pivot, x,** to **partition** the array such that:
  - All entries to the left of x are <= x (smaller).
  - All entries to the right of x are >= x (bigger).
  - x is in the right place in the final, sorted array.

A ✅

4	5	9	10	10	550	330

C ✅

4	5	9	10	10	330	550

B ✅

5	9	10	4	10	330	550

D ❌

5	9	10	4	10	550	330

Context for Quicksort's Invention ([Source])

1960: Tony Hoare was working on a crude automated translation program for Russian and English.

"The cat wore a beautiful hat."

N words

...	...
beautiful	красивая
...	...
cat	кошка
...	...

Dictionary of D english words

"Кошка носил красивая шапка."

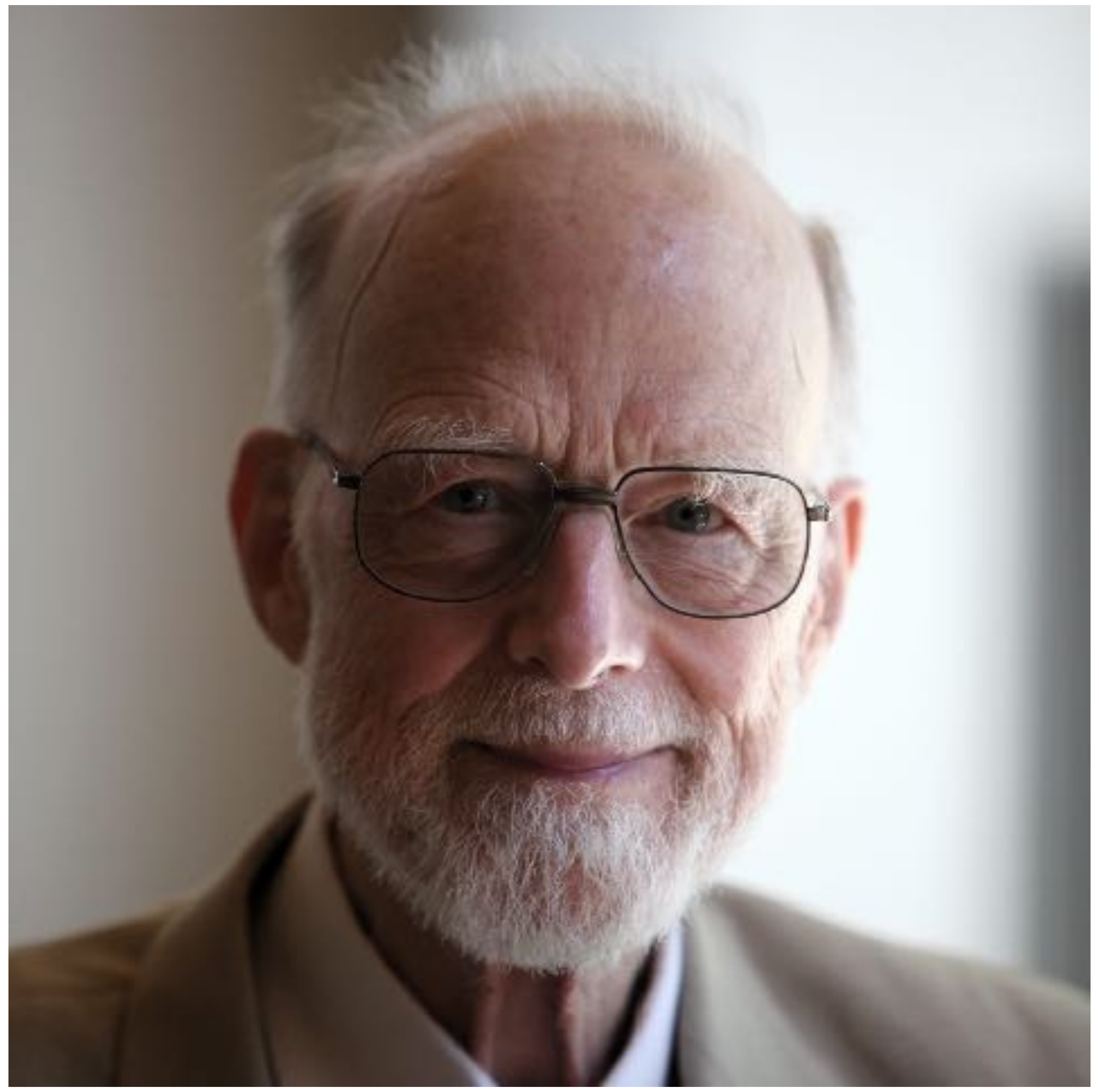
How would you do this?

- (Binary) Search for each word.
  - Find "the" in log D time.
  - Find "cat" in log D time...
- Total time: N log D

Context for Quicksort's invention

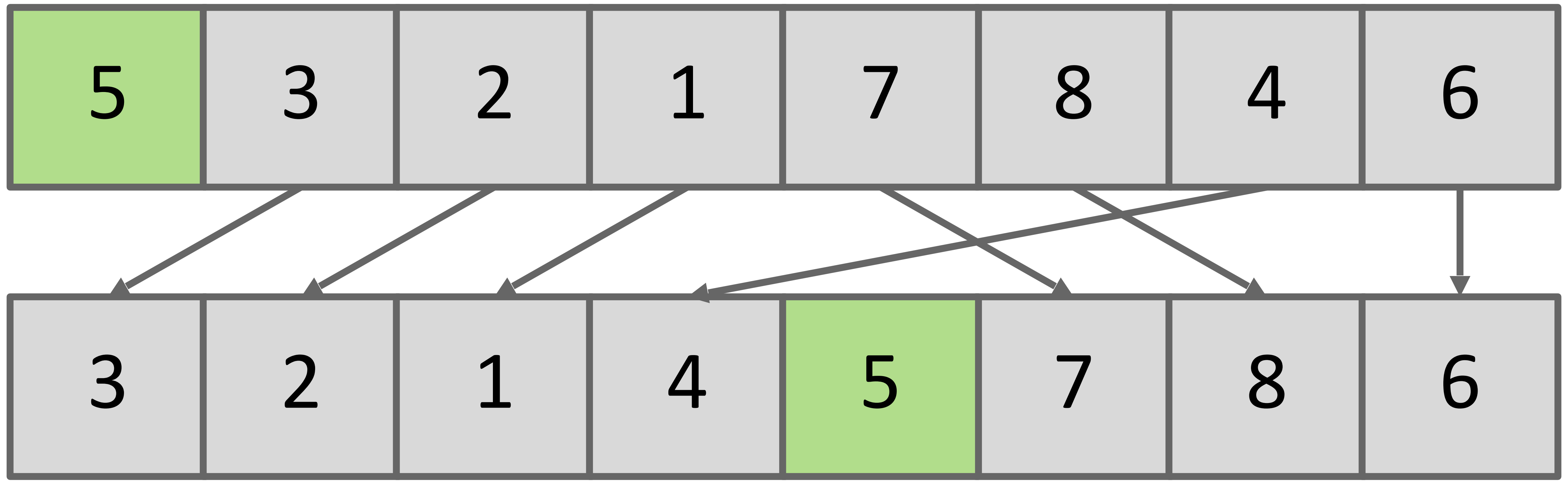
- However, we had hardware limitations at the time.
  - Dictionary stored on long piece of tape
  - Sentence is an array in RAM.
  - Search of tape takes very long (requires physical movement!).
  - D >> N.

- Better: **Sort the sentence** and scan dictionary tape once. Takes N log N + D time.
  - But Tony had to figure out how to sort an array...
  - Came up with Quicksort but did not know how to implement it.
  - Learned Algol 60 and recursion and implemented it.
  - Won the 1980 Turing Award (also invented the concept of null—and regretted it).



https://en.wikipedia.org/wiki/Tony_Hoare

Partition Sort, a.k.a. Quicksort

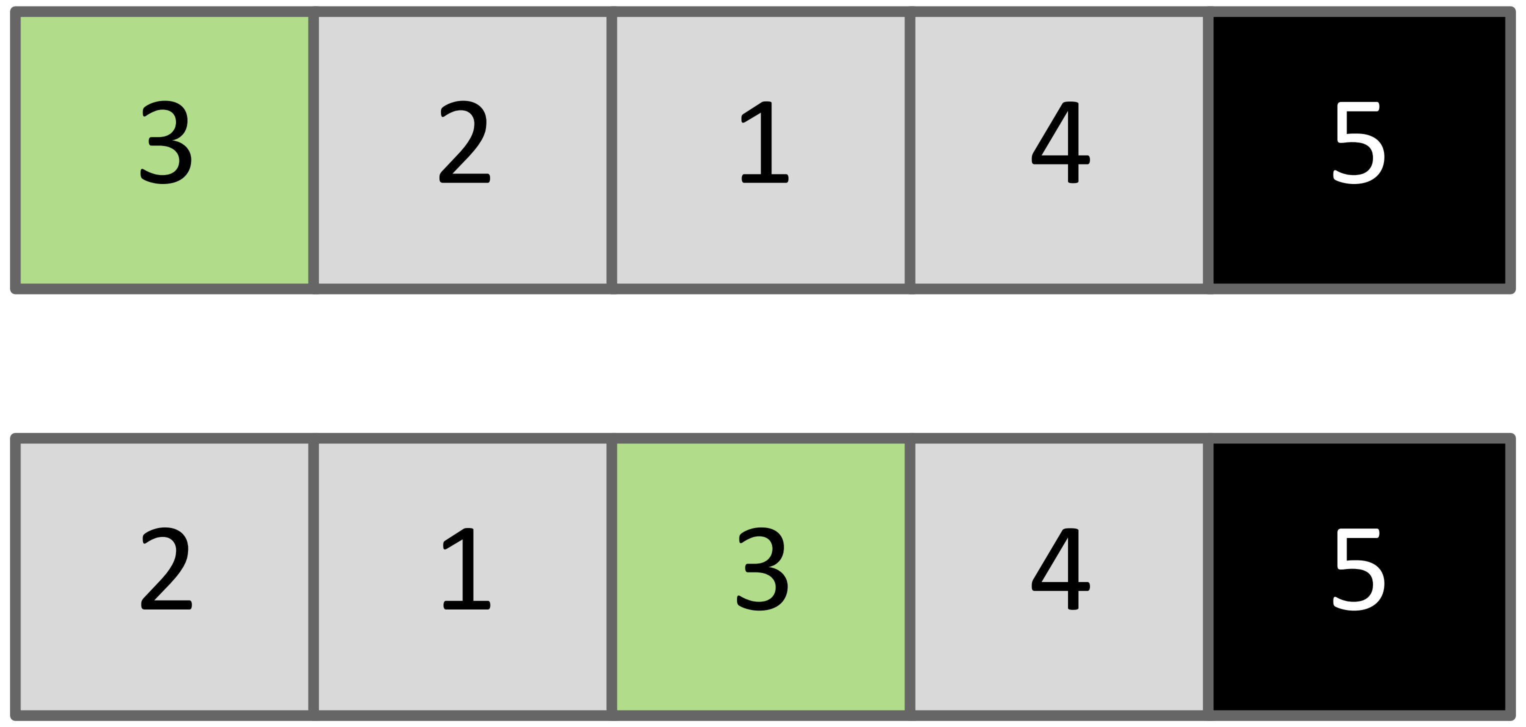


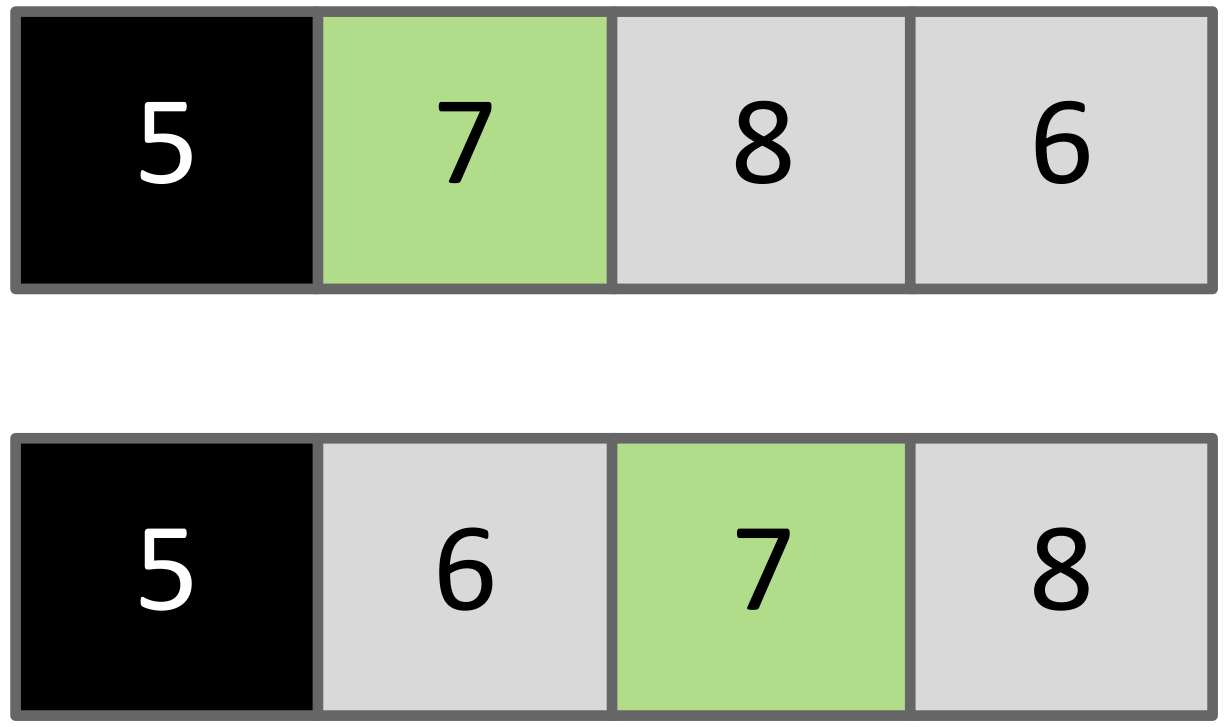
Q: How would we use this operation for sorting?

Note: this element order is slightly different than our implementation

Observations:

- 5 is “in its place.” Exactly where it’d be if the array were sorted.
- Can sort two halves separately, e.g. through recursive use of partitioning.



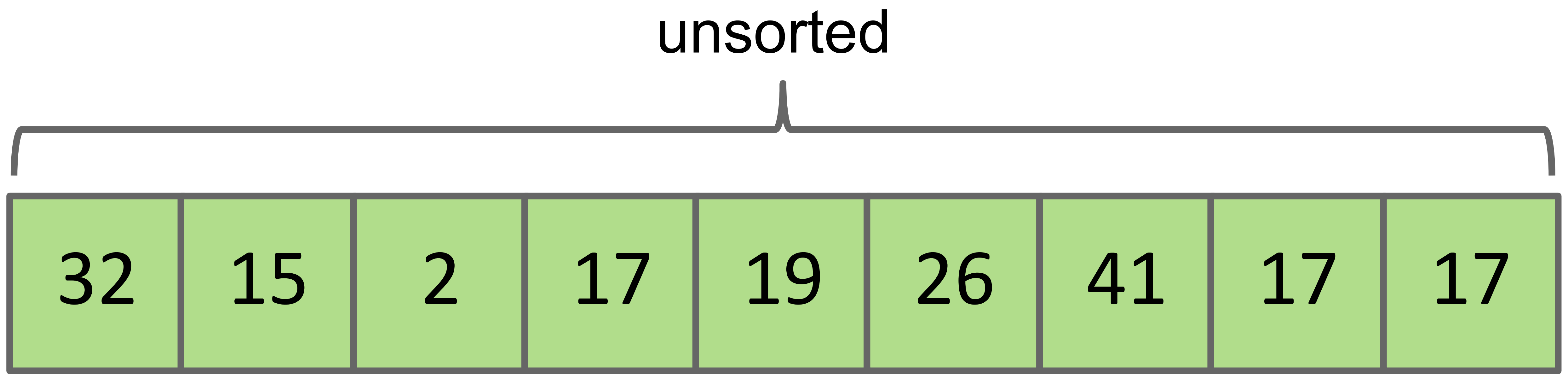


Quick Sort

Quick sorting N items:

- Partition on leftmost item.
- Quicksort left half.
- Quicksort right half.

Input:



Quick Sort

partition(32)

Quick sorting N items:

- **Partition on leftmost item (32).**
- Quicksort left half.
- Quicksort right half.

Input:

32	15	2	17	19	26	41	17	17

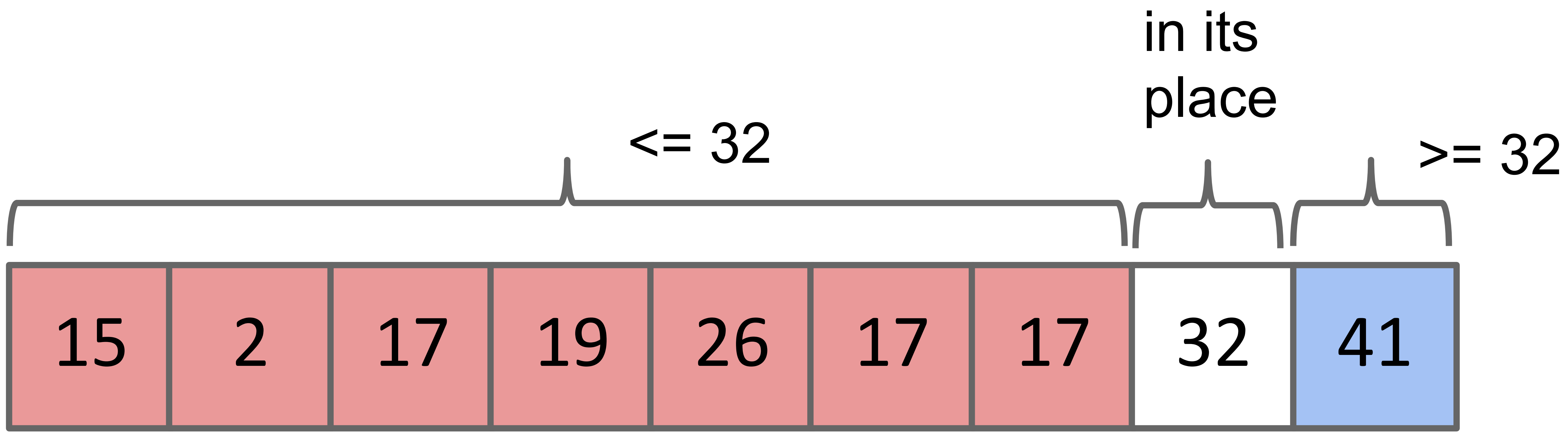
Quick Sort

partition(32)

Quick sorting N items:

- **Partition on leftmost item (32).**
- Quicksort left half.
- Quicksort right half.

Input:



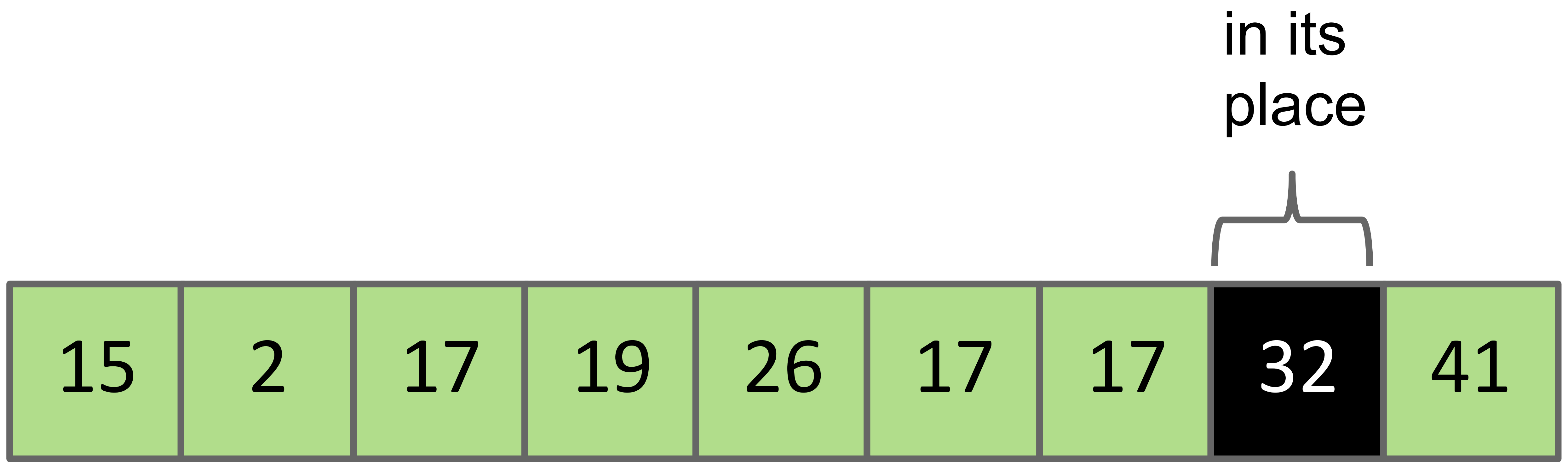
Quick Sort

partition(32)

Quick sorting N items:

- **Partition on leftmost item (32) (done).**
- Quicksort left half.
- Quicksort right half.

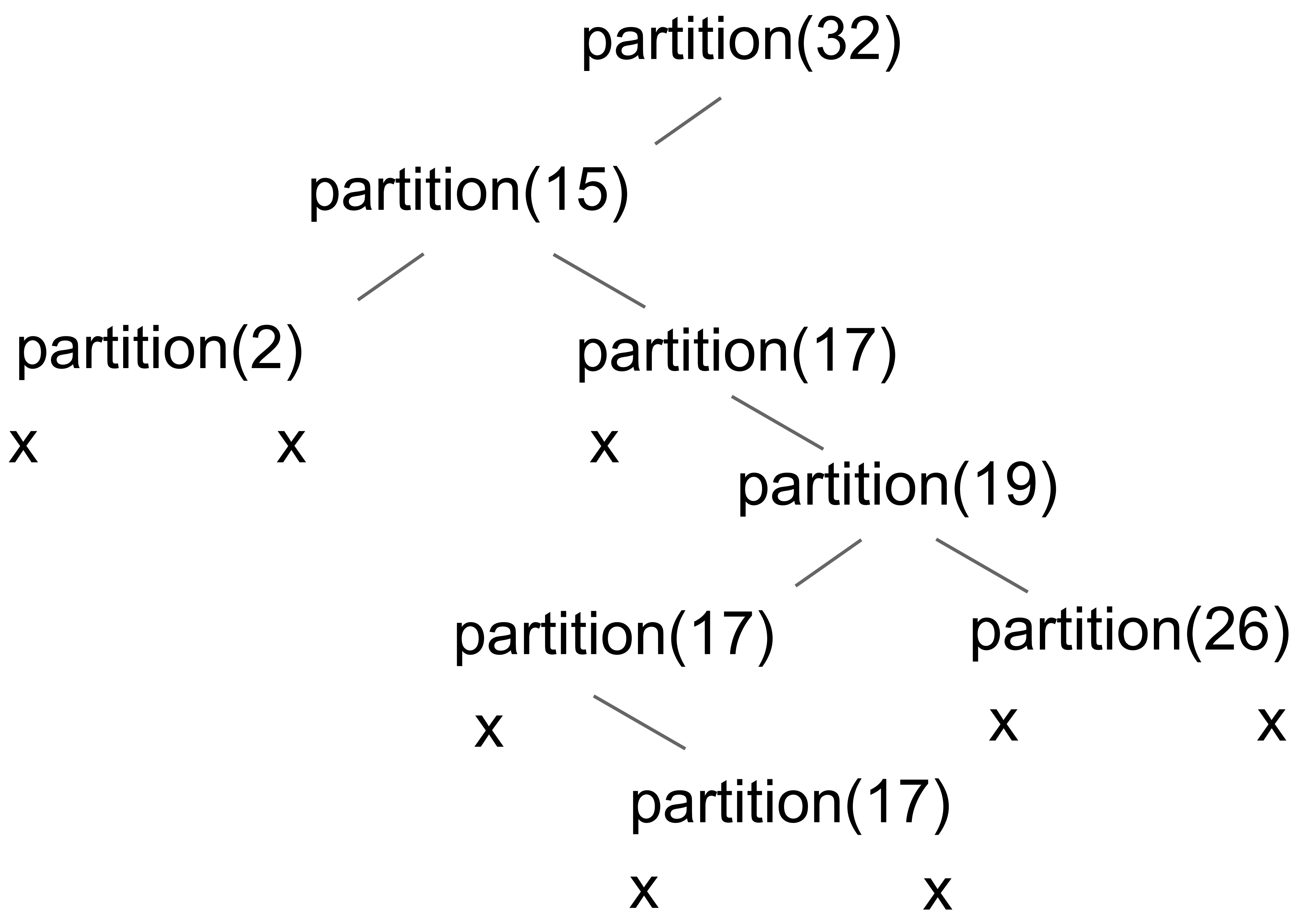
Input:



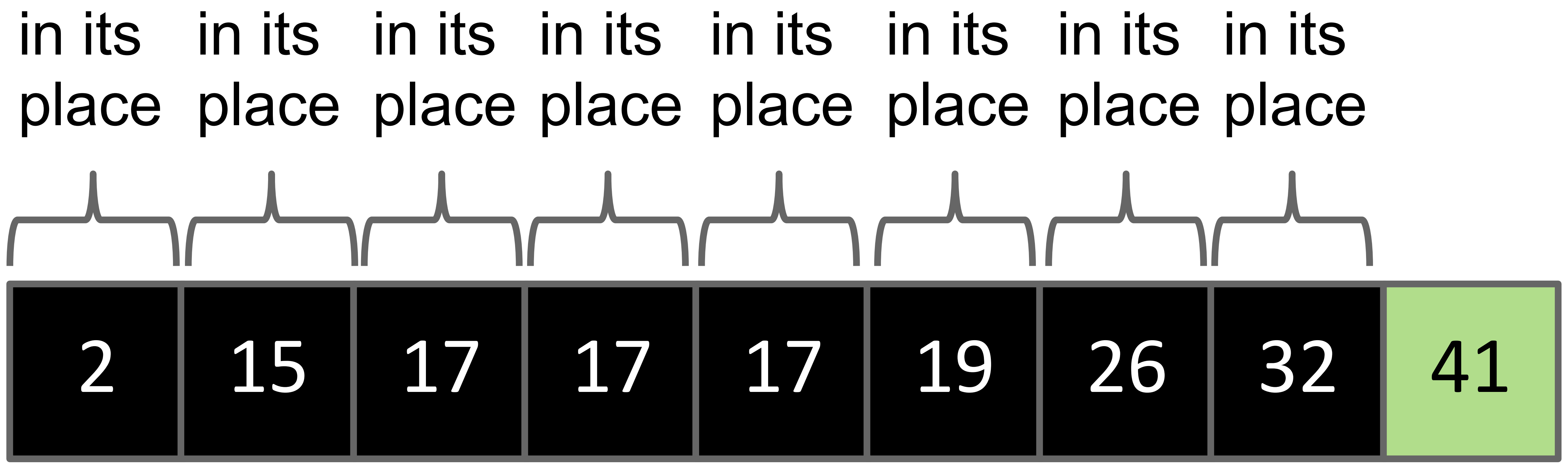
Quick Sort

Quick sorting N items:

- Partition on leftmost item (32) (done).
- **Quicksort left half (details not shown).**
- Quicksort right half.



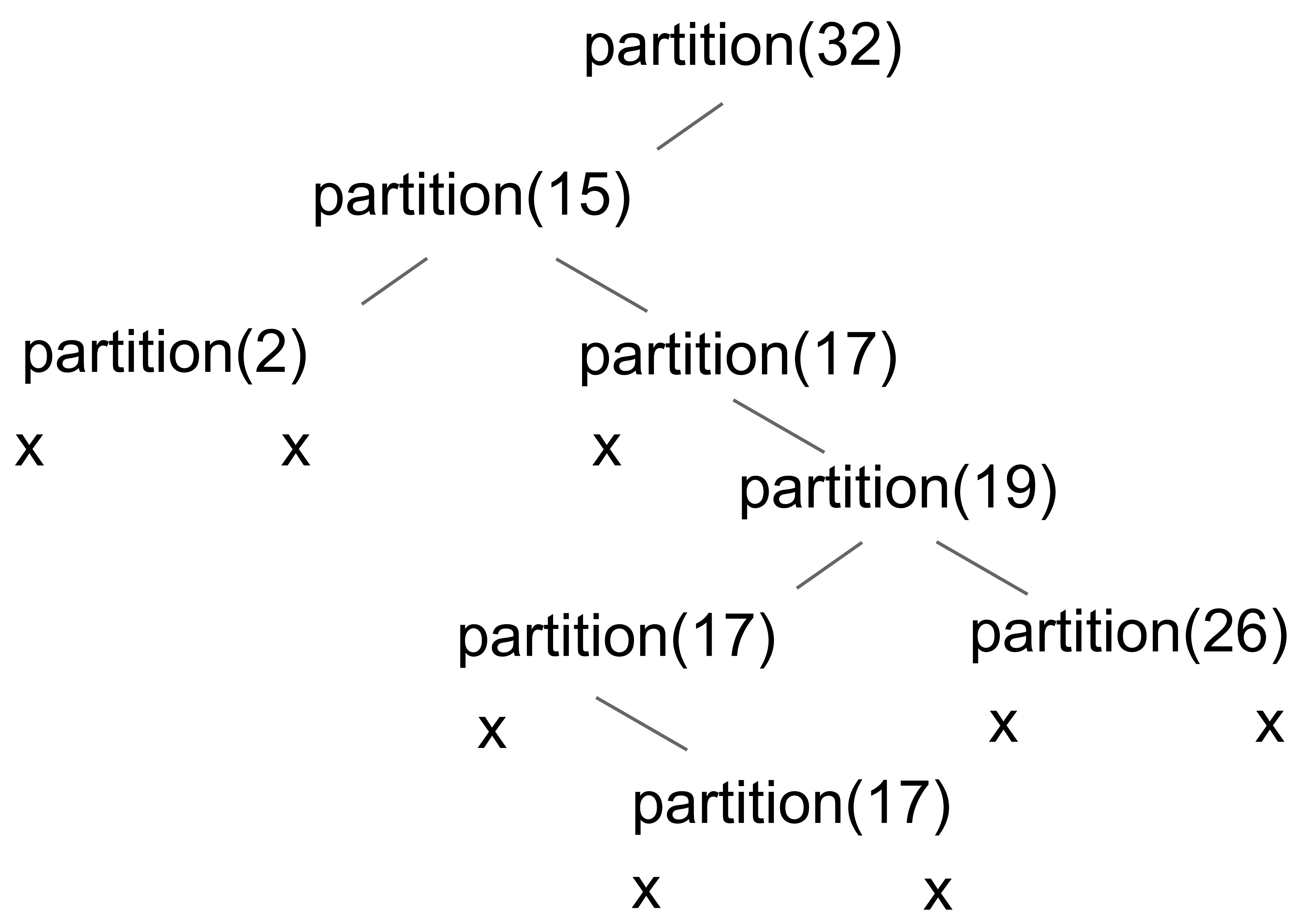
Input:



Quick Sort

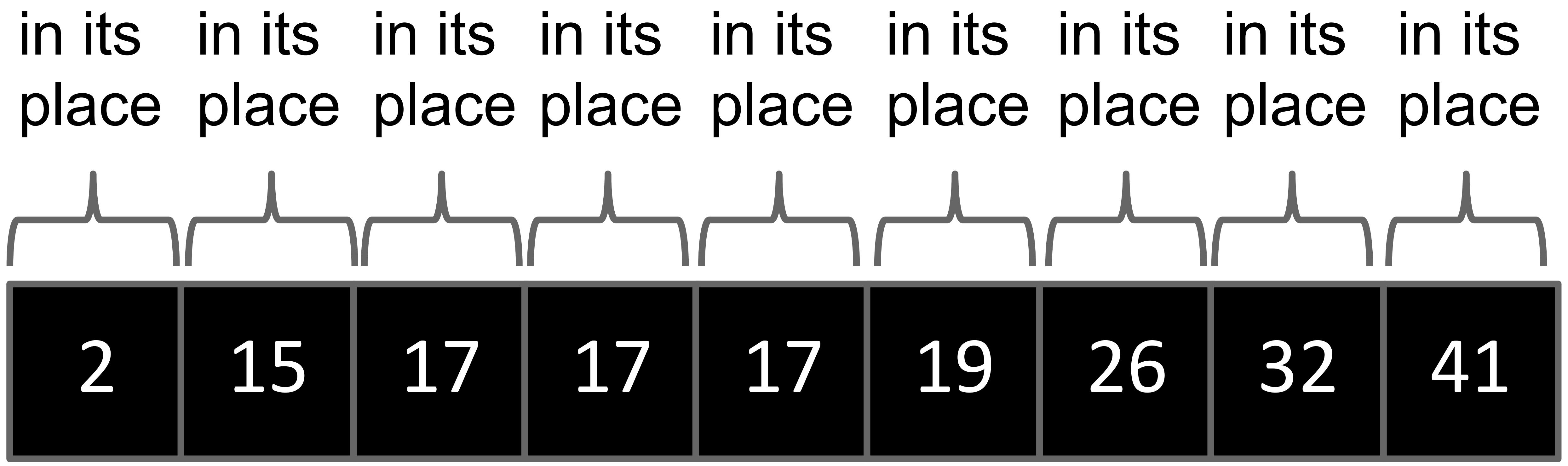
Quick sorting N items:

- Partition on leftmost item (32) (done).
- Quicksort left half (details not shown).
- **Quicksort right half (details not shown).**



If you don't fully trust the recursion, see [these extra slides] for a complete demo.

Input:



Quicksort code

Quicksort Code

```java
//helper method that sorts subarray from lo to hi
private static <E extends Comparable<E>> void quickSort(E[] a, int lo, int hi) {
    if (lo < hi){
        int pivot = partition(a, lo, hi);
        quickSort(a, lo, pivot - 1);
        quickSort(a, pivot + 1, hi);
    }

}

/*
 * Rearranges the array in ascending order, using the natural order.
 * @param a array to be sorted
 */
public static <E extends Comparable<E>> void quickSort(E[] a) {
    quickSort(a, 0, a.length - 1);
}
```

Partition

```java
private static <E extends Comparable<E>> int partition(E[] a, int lo, int hi) {
    E pivot = a[lo]; // Choose leftmost element as pivot
    int i = lo + 1; // Start from the next element
    int j = hi;

    while (true) {
        // Move right until we find an element >= pivot
        while (i <= j && a[i].compareTo(pivot) <= 0) {
            i++;
        }
        // Move left until we find an element < pivot
        while (j >= i && a[j].compareTo(pivot) > 0) {
            j--;
        }
        // If pointers cross, break
        if (i > j) {
            break;
        }

        // Swap elements to ensure correct partitioning
        E temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }

    // Swap pivot into its correct position
    E temp = a[lo];
    a[lo] = a[j];
    a[j] = temp;

    return j; // Return final pivot position
}
```

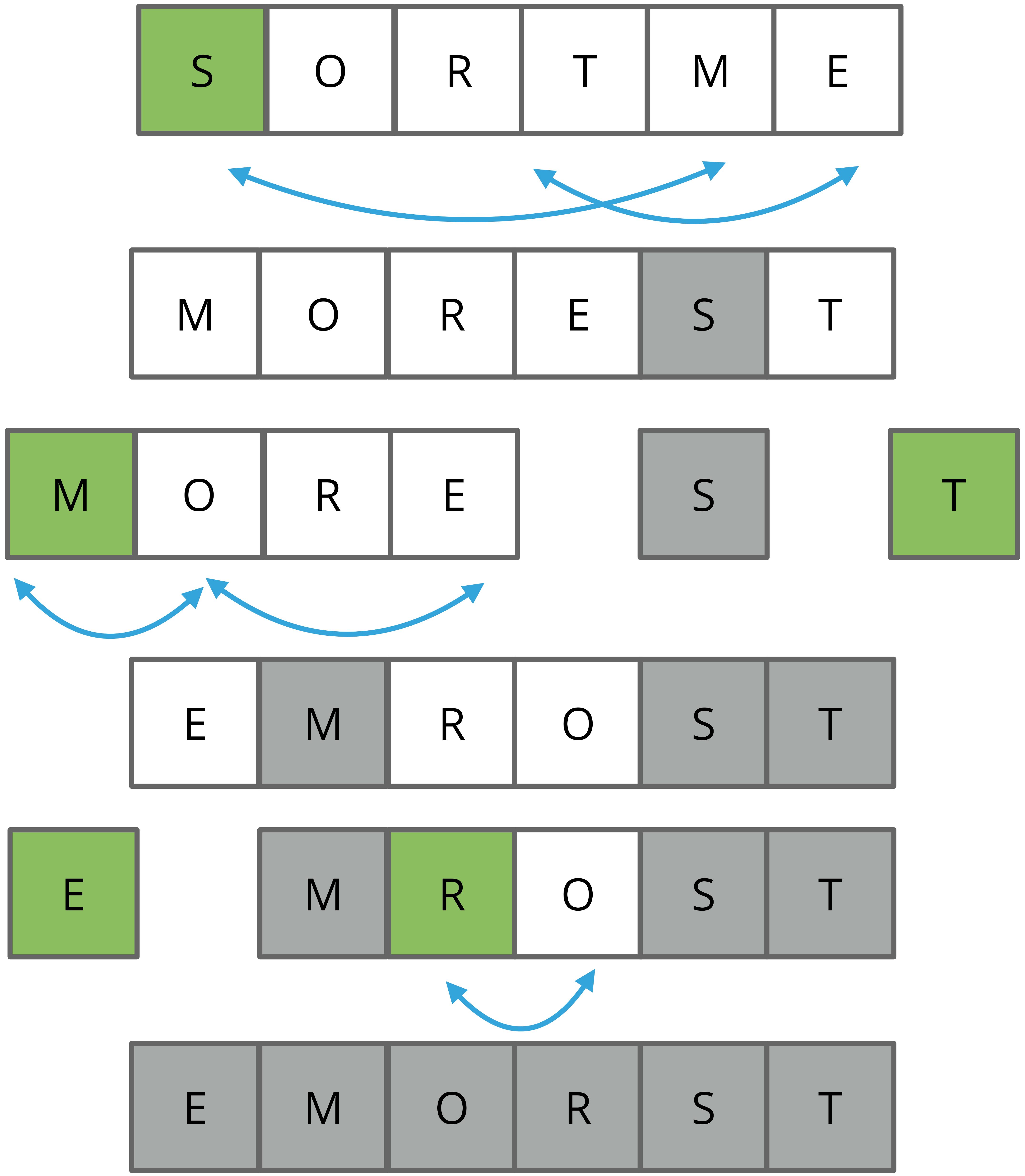
i starts on left side, j starts on right side

i = elems bigger than pivot, j = elems smaller than pivot

swap i and j since is bigger than the pivot (should be on the right side)
and j is smaller than the pivot (should be on left side)

finally, swap pivot with j

Code walkthrough with debugger



T is a single element, so no sorting needed!

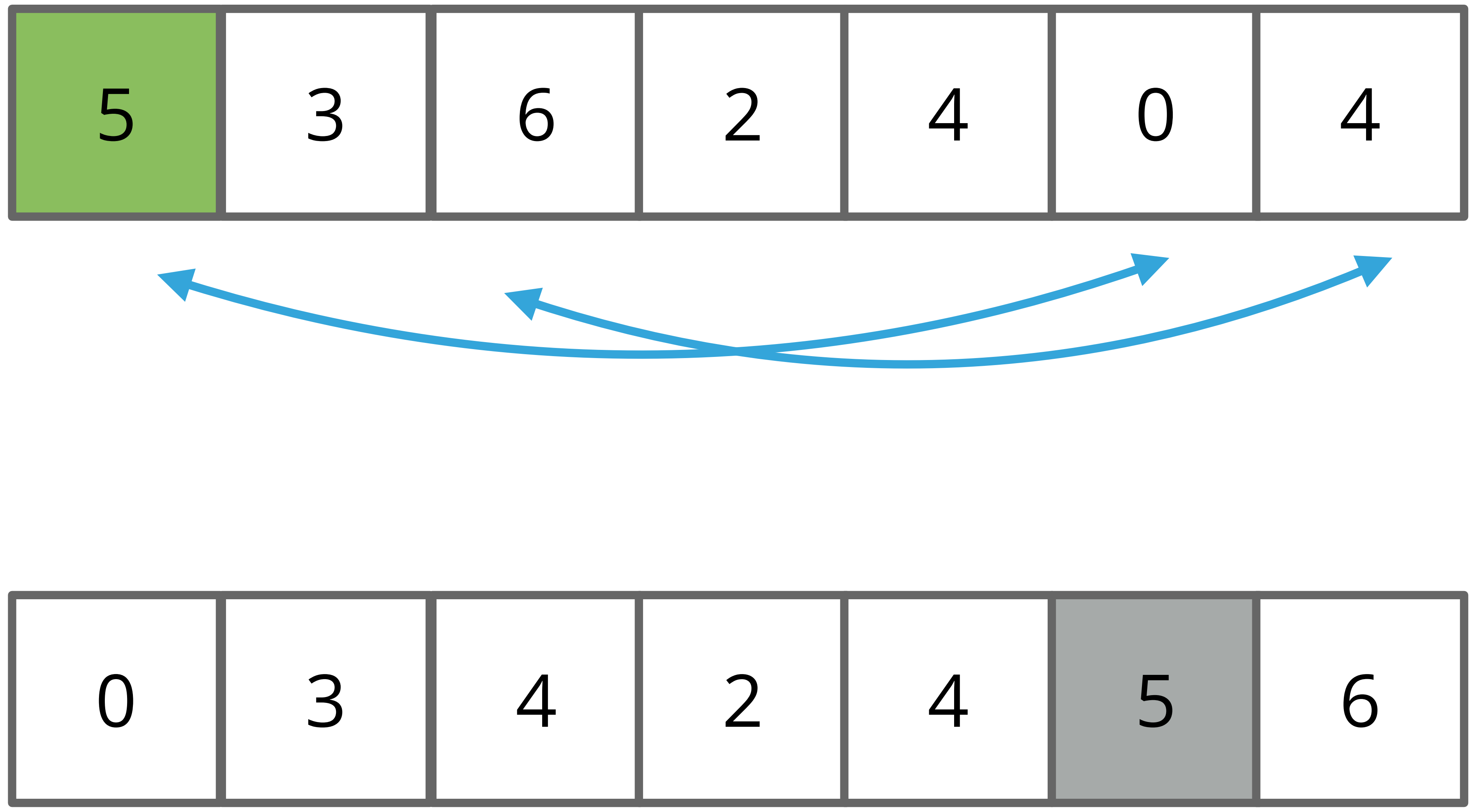
E is a single element, so no sorting needed!

Worksheet time!

Please draw what happens after the first partition of the following array

5	3	6	2	4	0	4

Worksheet answers



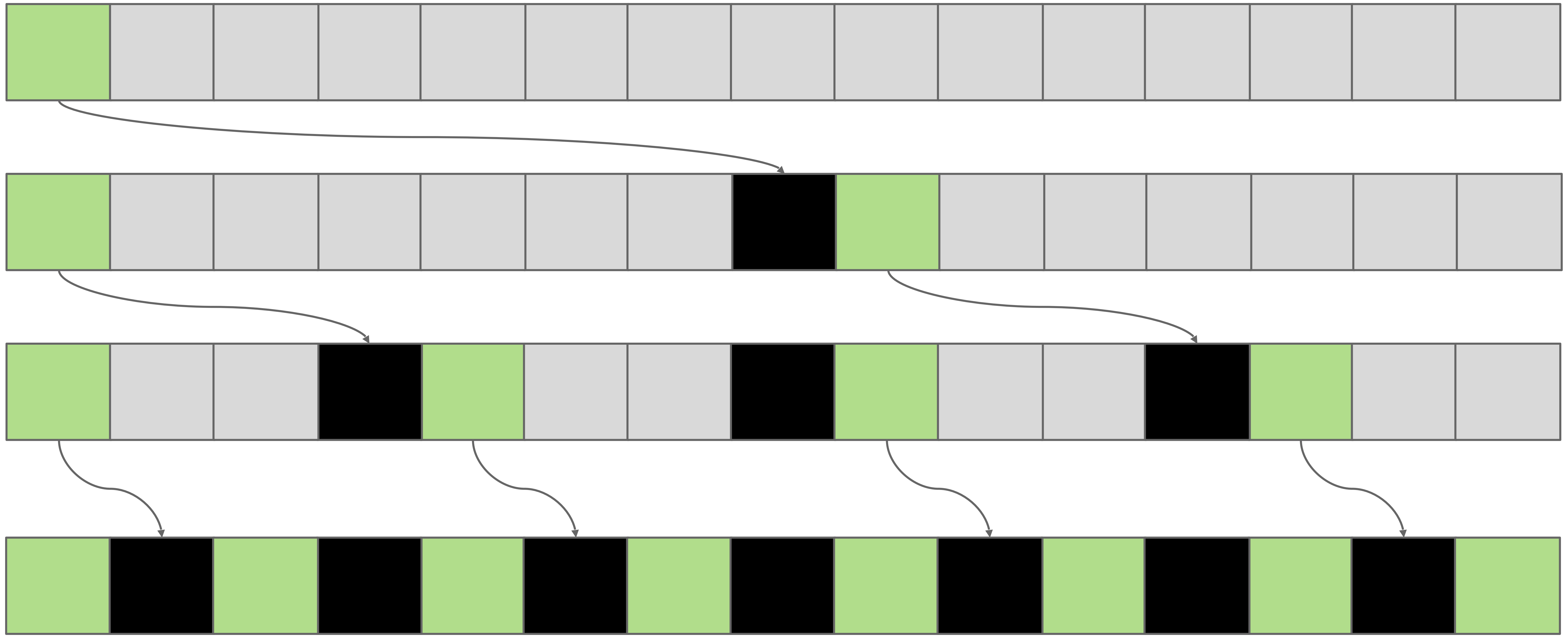
Quicksort analysis

Great algorithms are better than good ones

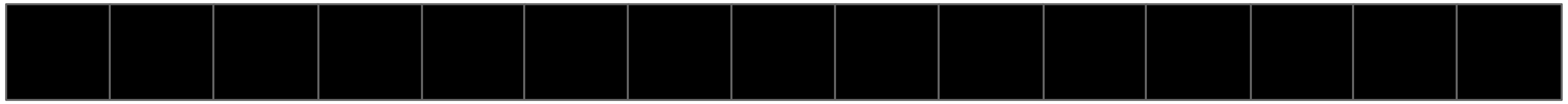
- Your laptop executes 10^8 comparisons per second
- A supercomputer executes 10^{12} comparisons per second

	Insertion sort			Mergesort			Quicksort		
Computer	**Thousand inputs**	**Million inputs**	**Billion inputs**	**Thousand inputs**	**Million inputs**	**Billion inputs**	**Thousand inputs**	**Million inputs**	**Billion inputs**
Home	Instant	2 hours	300 years	instant	1 sec	15 min	Instant	0.5 sec	10 min
Supercomputer	Instant	1 sec	1 week	instant	instant	instant	instant	instant	instant

Best case: pivot always lands in the middle



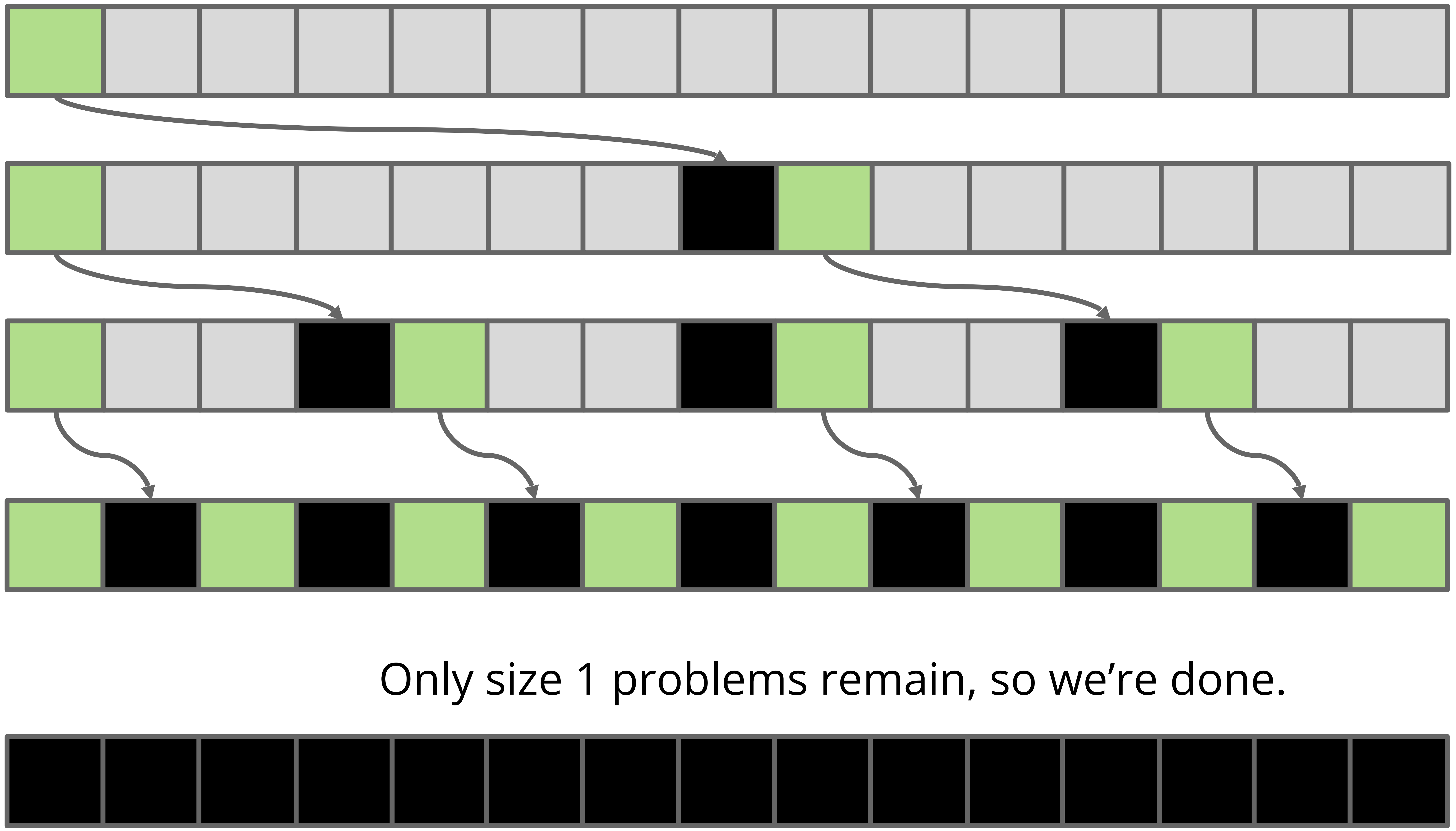
Only size 1 problems remain, so we're done.



Worksheet Q: what's the best case run time?

Ω(nlogn) best case

Total # of comparisons at each level:



$\approx N$

$\approx N/2 + \approx N/2 = \approx N$

$\approx N/4 * 4 = \approx N$

Only size 1 problems remain, so we're done.

Overall runtime:

$\Omega(NH)$ where H(eight) = $\Omega(\log N)$

so: $\Omega(N \log N)$

Just like Mergesort, we're dividing the work in half each level, so a log(n) relationship for height

Worst case: pivot always at the start

Worksheet Q: Give an example of an array input that would result in this behavior.

What is the run time?

Worst case: pivot always at the start

Worksheet Q: Give an example of an array input that would result in this behavior.

[1 2 3 4 5 6]

What is the run time?

O(n^2)

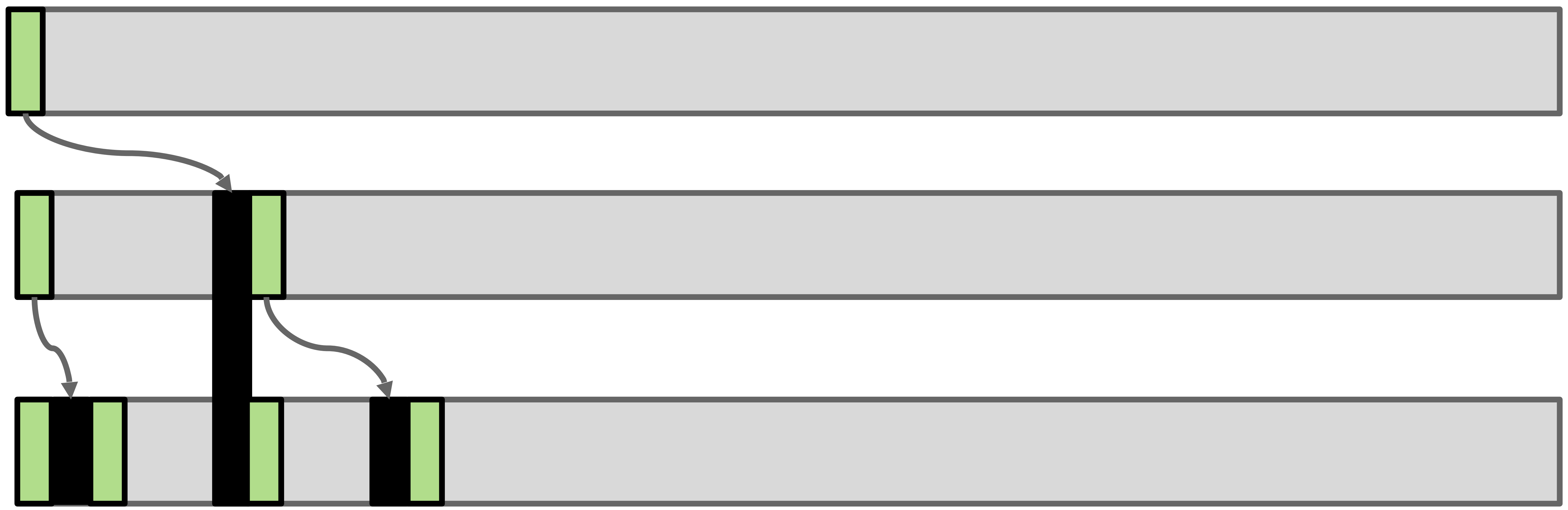
Now the height is N, instead of log(N)

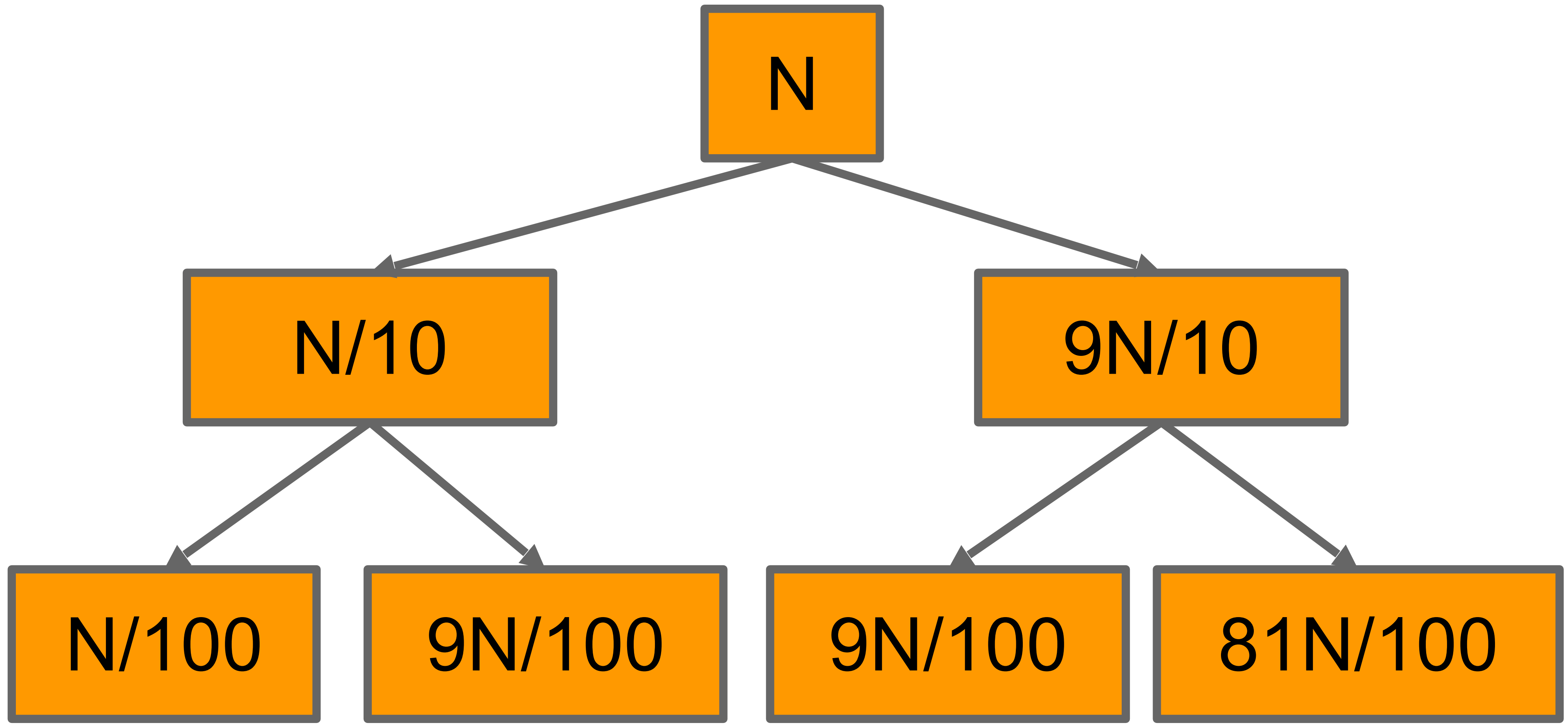
OK but

- How is Quicksort the fastest sorting algorithm in practice if the worst case is $O(n^2)$?
- We can just first *randomly shuffle* our data (takes N time, one operation) to avoid sorting on pre-sorted arrays. Then it's extremely unlikely to ever run into the worst case scenario (you're more likely to get struck by lightning).
- Average case is $\Theta(n\log n)$. We won't go into a detailed proof, but hopefully the next slide can convince you intuitively, and the following one empirically:

Argument #1: 10% Case

Suppose pivot always ends up at least 10% from either edge (not to scale).





Work at each level: O(N)

- Runtime is O(NH).
  - H is approximately $\log_{10/9} N = O(\log N)$
- Overall: O(N log N).

Punchline: Even if you are unlucky enough to have a pivot that never lands anywhere near the middle, but at least always 10% from the edge, runtime is still O(N log N).

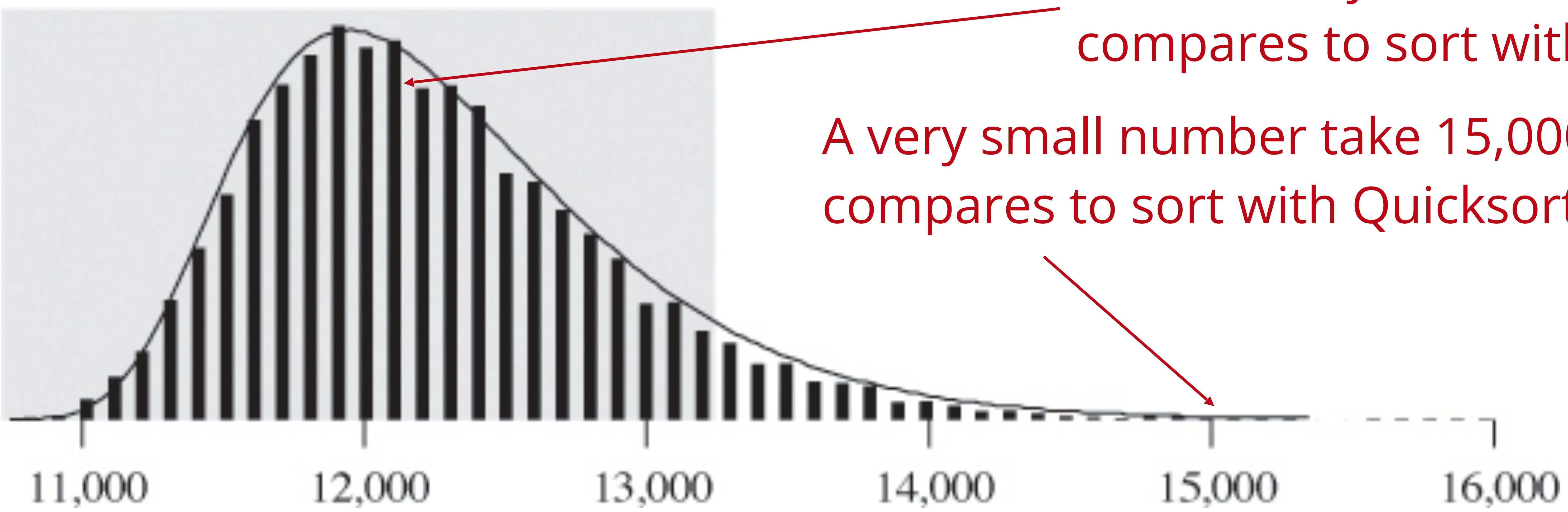
Empirical Quicksort Runtimes

For N items:

- Mean number of compares to complete Quicksort: ~2N ln N
- Standard deviation: $\sqrt{(21 - 2\pi^2)/3}N \approx 0.6482776N$

Lots of arrays take 12,000ish compares to sort with Quicksort.

A very small number take 15,000ish compares to sort with Quicksort.



Empirical histogram for quicksort compare counts (10,000 trials with N = 1000)

Chance of taking 1,000,000ish compares is effectively zero.

For more, see: http://www.informit.com/articles/article.aspx?p=2017754&seqNum=7

Things to remember about Quicksort

- ~39% more compares than mergesort but in practice it is faster because it does not move data much (no need to copy the array!).
- $O(n \log n)$ average, $O(n^2)$ worst, in practice faster than mergesort.
- In-place sorting.
- **Not** stable. (We swap!)
- It's mainly about choosing a smart pivot.
  - We just took the leftmost element
  - Tony Hoare's algorithm actually uses 2 pointers that walk towards each other
  - The modern Quicksort used in practice in Java to sort arrays of **primitives** uses 2 pivot points instead (Yaroslavskiy, Bentley, and Bloch, 2009)
  - Java uses Timsort (modified Mergesort) to sort arrays of **objects**, because of stability

Q: Why would stability be important for objects but not primitives?

Philosophies to avoid worst case Quicksorts

- 1) Randomness: pick a random pivot instead of the leftmost pivot, or shuffle your data before starting
- 2) Smarter pivot selection: calculate or approximate the median to serve as the pivot
- 3) Knowing when to stop: use insertion sort if the array size gets small/recursion gets too deep
- 4) Preprocessing the array: analyze array beforehand to see if Quicksort will be slow
    - This doesn’t really work in practice. You can’t just check if an array is sorted, because “almost” sorted arrays (e.g., [1, 2, 3, … 99, 98, 100]) are also basically O(n^2) time, and there’s no obvious way to see if an array is “almost” sorted

Sorting: the story so far

Which Sort	In place	Stable	Best	Average	Worst	Memory	Remarks
Selection	X		$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1)$	n exchanges
Insertion	X	X	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1)$	Fastest if almost sorted or small
Merge		X	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$\Theta(n)$	Guaranteed performance; stable
Quick	X		$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$\Theta(\log n)$	$n \log n$ probabilistic guarantee; fastest in practice

(call stack)

Lecture 12 wrap-up

- Lots of sorting! Quiz in lab will be about sorting. Lab is about timing sorting.
- Reminder: Wordle HW & checkpoint regrades due tomorrow 11:59pm
- HW6: On Disk sort released, due Weds 11:59pm due to fall break (w/ auto extension to Thu 6:59pm)

Resources

- Reading from textbook: Chapter 2.3 (pages 288–296)
- Quicksort video: https://www.youtube.com/watch?v=Hoixgm4-P4M
- Online textbook website - https://algs4.cs.princeton.edu/23quicksort/ (note we have a different implementation)
- Practice problem behind this slide

Practice Problem 1

- What would the resulting array for the first call to partition be for the following array if instead the pivot was the **rightmost** element: [E,A,S,Y,Q,U,E,S,T,I,O,N].

Answer 1

- What would the resulting array for the first call to partition be for the following array if instead the pivot was the rightmost element: [E,A,S,Y,Q,U,E,S,T,I,O,N].
- [E, A, E, I, N, U, S, S, T, Y, O, Q] and pivot: at index 4.