CS11313 - Spring 2023

Design & Analysis of Algorithms

Quicksort

Ibrahim Albluwi

Quicksort

- Honored as one of top 10 algorithms of 20th century in science and engineering.

from *SIAM News*, Volume 33, Number 4

The Best of the 20th Century: Editors Name Top 10 Algorithms

1962: Tony Hoare of Elliott Brothers, Ltd., London, presents **Quicksort**.

Putting N things in numerical or alphabetical order is mind-numbingly mundane. The intellectual challenge lies in devising ways of doing so quickly. Hoare's algorithm uses the age-old recursive strategy of divide and conquer to solve the problem: Pick one element as a "pivot," separate the rest into piles of "big" and "small" elements (as compared with the pivot), and then repeat this procedure on each pile. Although it's possible to get stuck doing all $N(N-1)/2$ comparisons (especially if you use as your pivot the first item on a list that's already sorted!), Quicksort runs on average with $O(N \log N)$ efficiency. Its elegant simplicity has made Quicksort the pos-terchild of computational complexity.

Quicksort

- Honored as one of top 10 algorithms of 20th century in science and engineering.
- Used in many widely-used software libraries.

Quicksort

- Honored as one of top 10 algorithms of 20th century in science and engineering.
- Used in many widely-used software libraries.
- Invented by Tony Hoare in 1959-1961 while working on a machine translation project between Russian and English.
- Couldn't implement it until he learned later ALGOL 60 and recursion.

Quicksort

By C. A. R. Hoare

A description is given of a new method of sorting in the random-access store of a computer. The method compares very favourably with other known methods in speed, in economy of storage, and in ease of programming. Certain refinements of the method, which may be useful in the optimization of inner loops, are described in the second part of the paper.

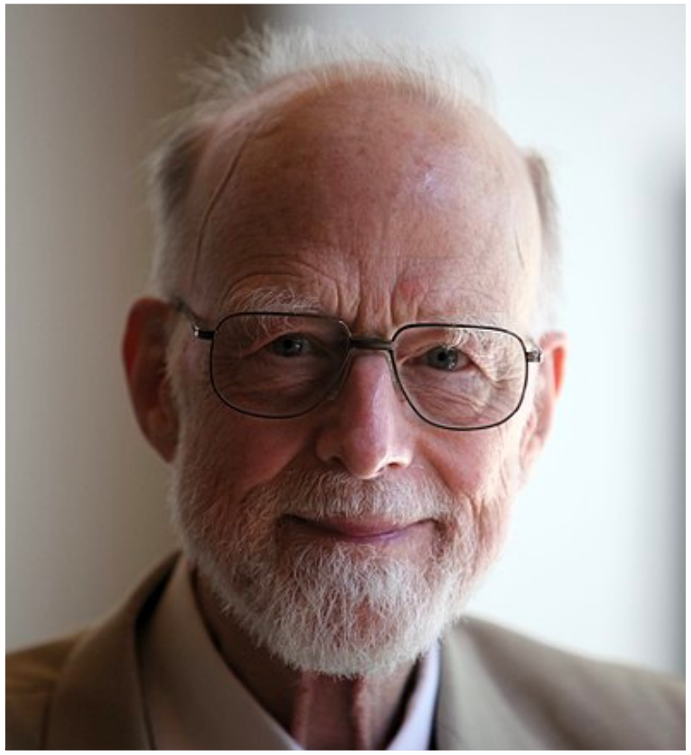
Part One: Theory

The sorting method described in this paper is based on the principle of resolving a problem into two simpler subproblems. Each of these subproblems may be resolved to produce yet simpler problems. The process is repeated until all the resulting problems are found to be trivial. These trivial problems may then be solved by known methods, thus obtaining a solution of the original more complex problem.

Partition

The problem of sorting a mass of items, occupying consecutive locations in the store of a computer, may be reduced to that of sorting two lesser segments of data, provided that it is known that the keys of each of the items held in locations lower than a certain dividing line

highest address and moves downward. The lower pointer starts first. If the item to which it refers has a key which is equal to or less than the bound, it moves up to point to the item in the next higher group of locations. It continues to move up until it finds an item with key value greater than the bound. In this case the lower pointer stops, and the upper pointer starts its scan. If the item to which it refers has a key which is equal to or greater than the bound, it moves down to point to the item in the next lower locations. It continues to move down until it finds an item with key value less than the bound. Now the two items to which the pointers refer are obviously in the wrong positions, and they must be exchanged. After the exchange, each pointer is stepped one item in its appropriate direction, and the lower pointer resumes its



Another Divide & Conquer Algorithm

```
MERGE-SORT(array)

  if array size is 0:
     return

  MERGE-SORT(left)
  MERGE-SORT(right)

  MERGE(left and right)
```

Recursively call `Merge-Sort` on the left and right halves.

Merge the two sorted halves.

```
QUICK-SORT(array)

  if array size is 0:
     return

  PARTITION(array)

  QUICK-SORT(left)
  QUICK-SORT(right)
```

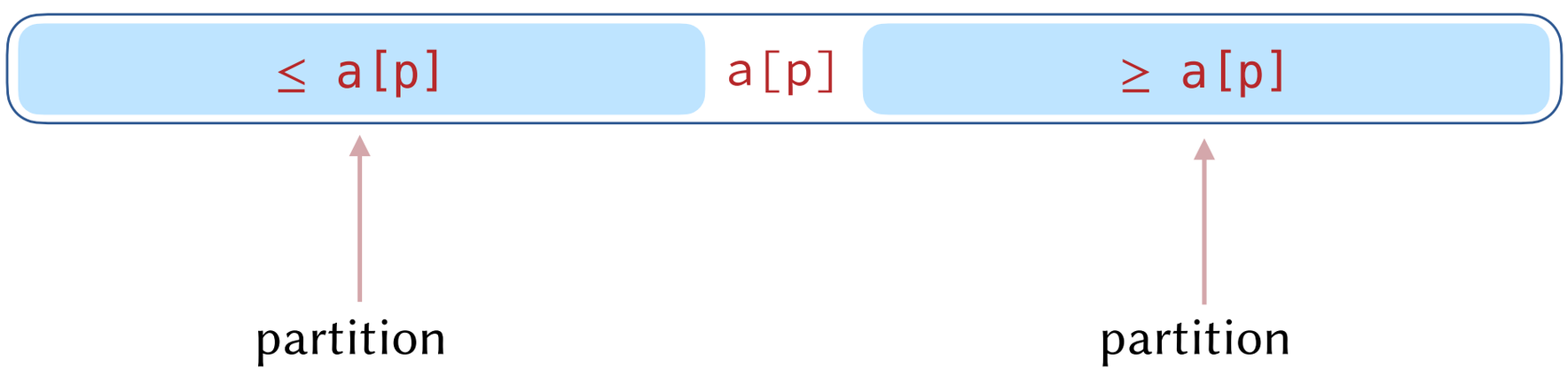
Cleverly *partition* the array into two halves.

Recursively call `Quicksort` on the left and right halves.

Quicksort Steps

The heart of the algorithm!

- Step 1. **Pick** an element at some index `p` (let's call it the **pivot**).
- Step 2. **Partition** the array, such that for some `p`:
  - All elements to the left of `a[p]` are less than or equal to `a[p]`.
  - All elements to the right of `a[p]` are greater than or equal to `a[p]`.
- Step 3. **Sort** each subarray (recursively repeat Steps 1 & 2 on each subarray).



Quicksort Steps

The heart of the algorithm!

- Step 1. **Pick** an element at some index `p` (let's call it the **pivot**).
- Step 2. **Partition** the array, such that for some `p`:
  - All elements to the left of `a[p]` are less than or equal to `a[p]`.
  - All elements to the right of `a[p]` are greater than or equal to `a[p]`.
- Step 3. **Sort** each subarray (recursively repeat Steps 1 & 2 on each subarray).

```
QUICK-SORT(a[], first, last)

  if first >= last:
     return

  p = PARTITION(a, first, last)

  QUICK-SORT (a, first, p - 1)
  QUICK-SORT (a, p + 1, last)
```

left half

right half

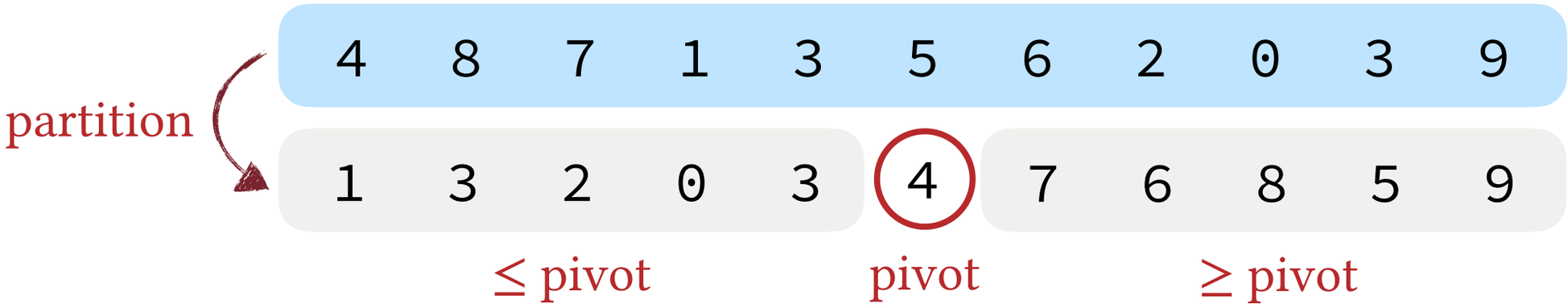
Demo

4	8	7	1	3	5	6	2	0	3	9

pivot

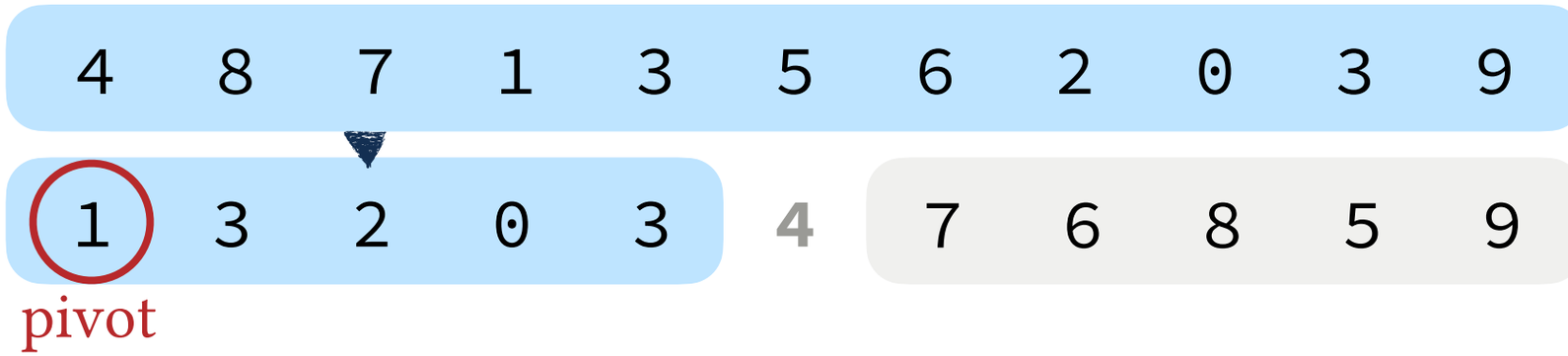
Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

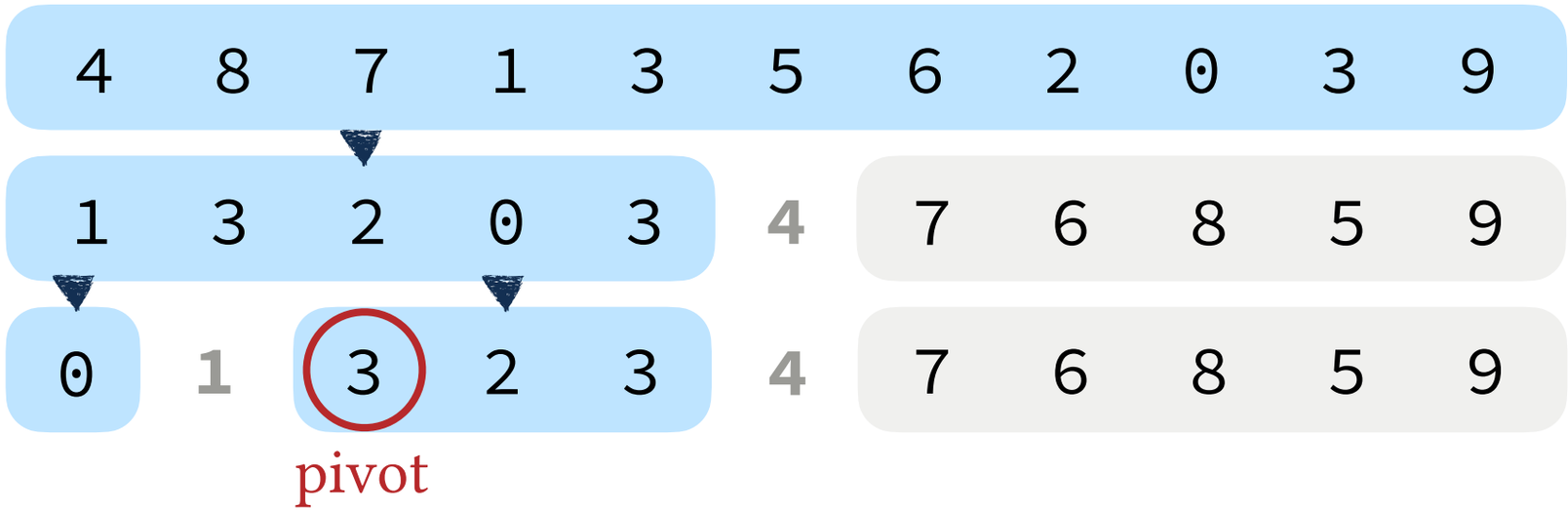
Demo

Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo

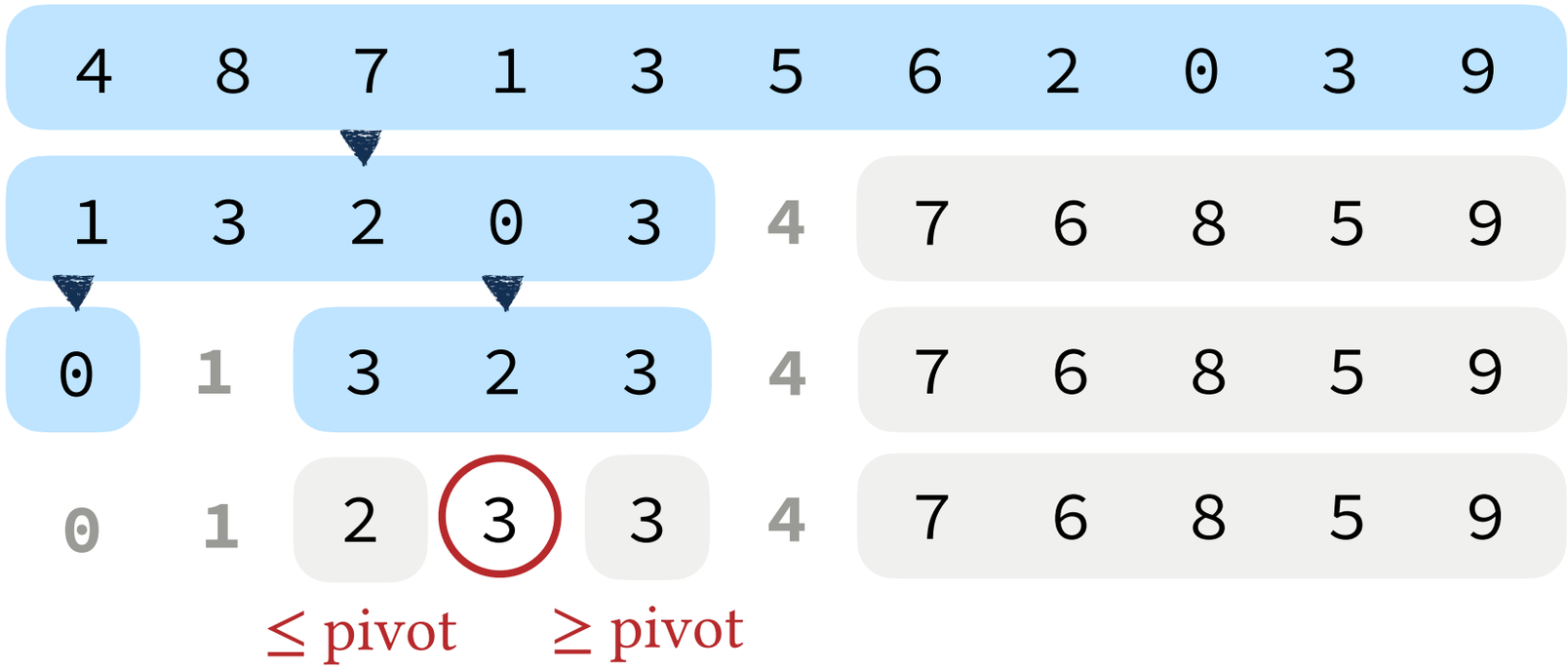
Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



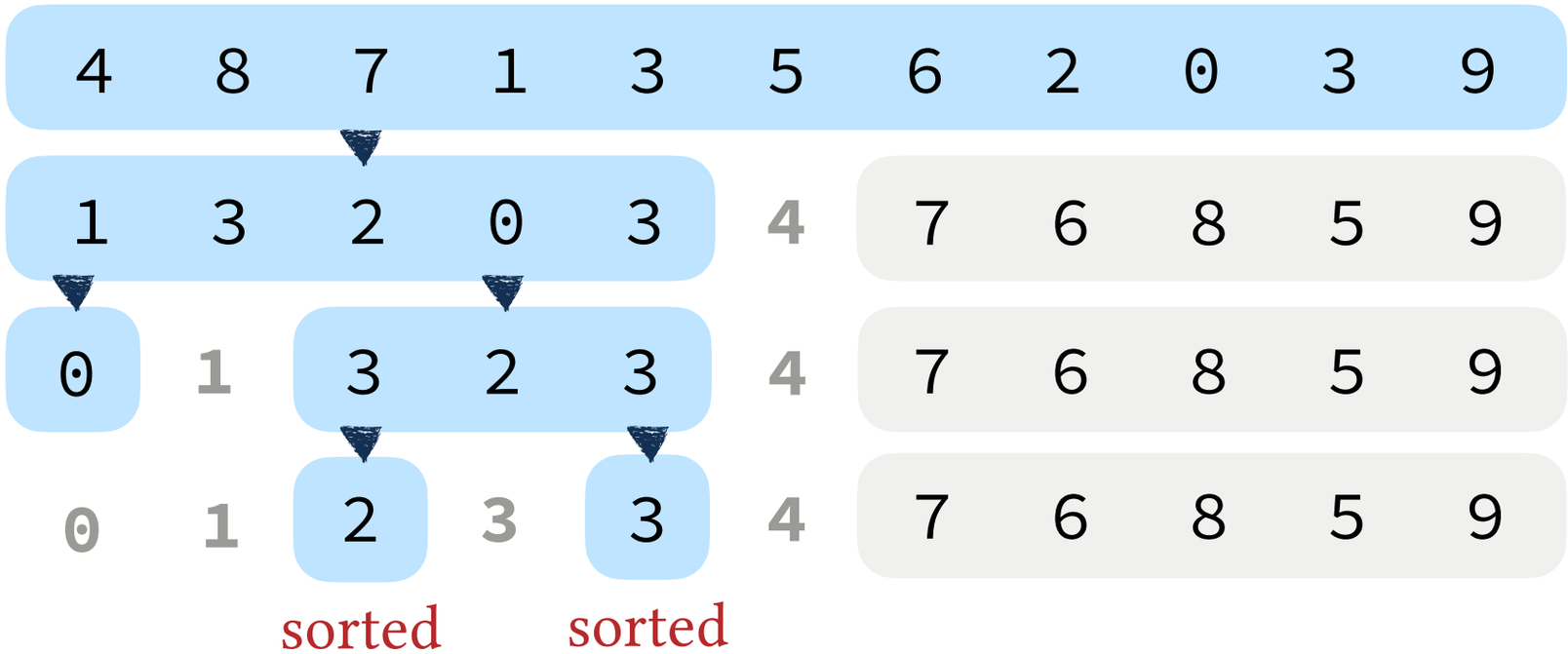
Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



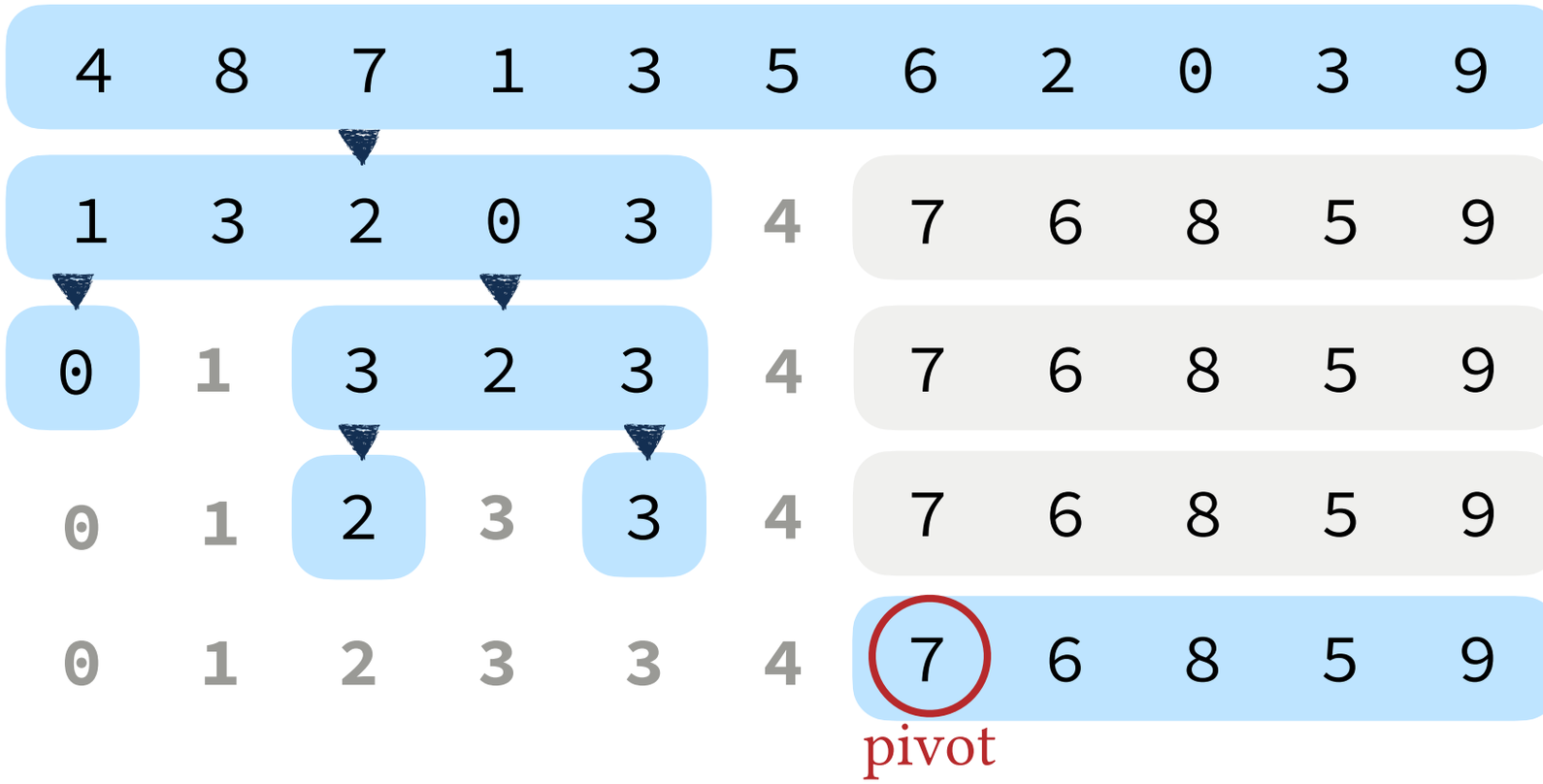
Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo

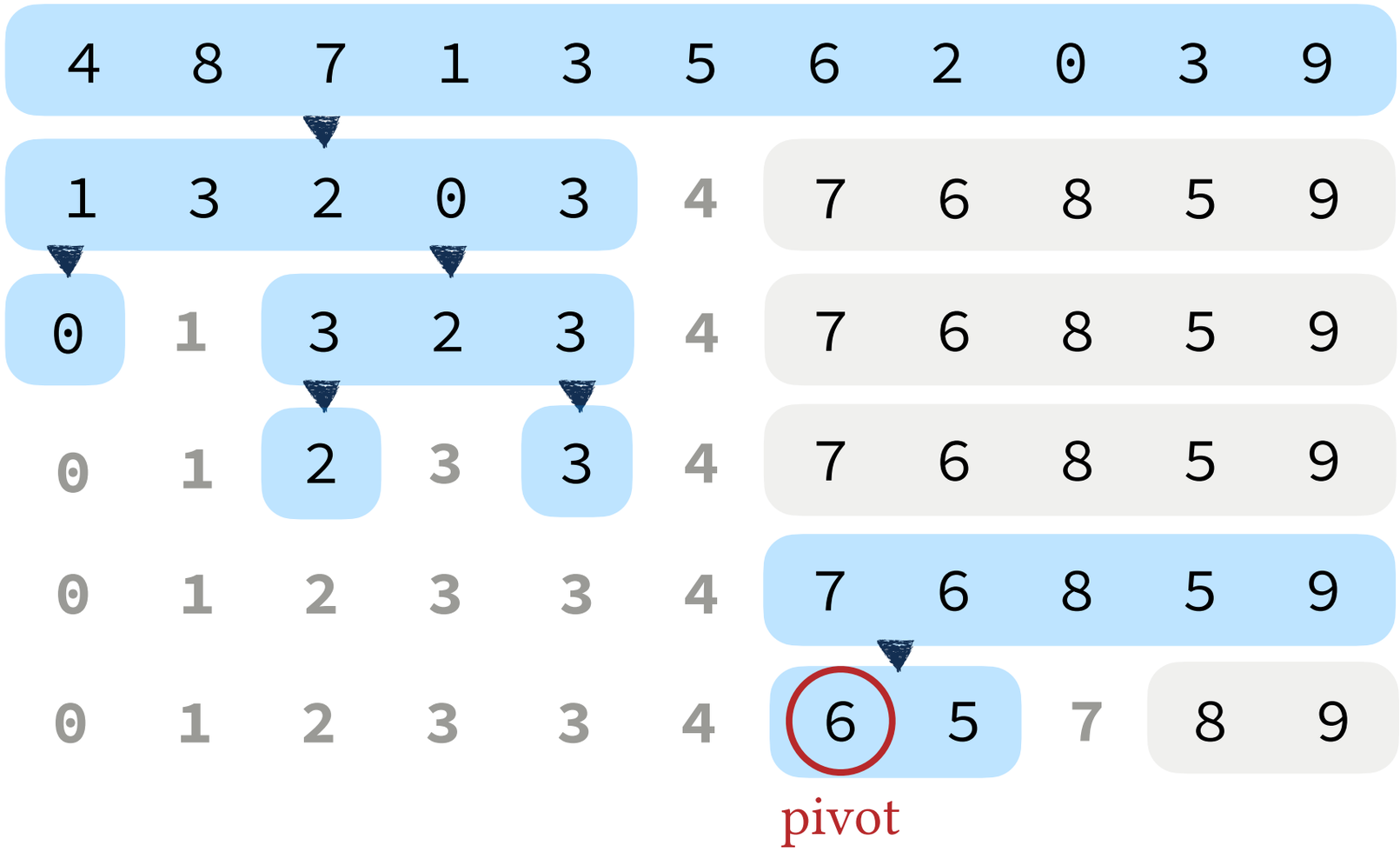


Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo

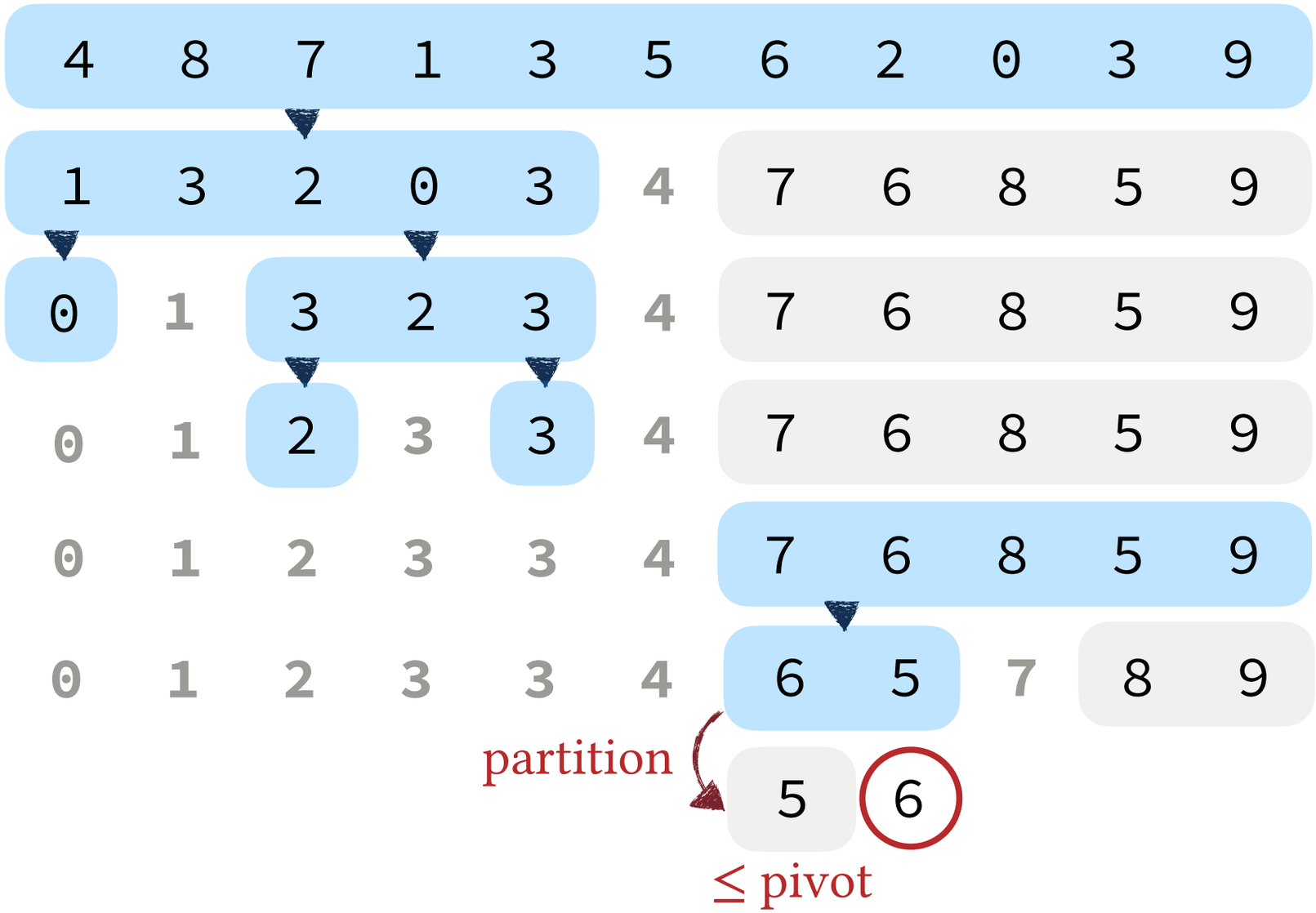
Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



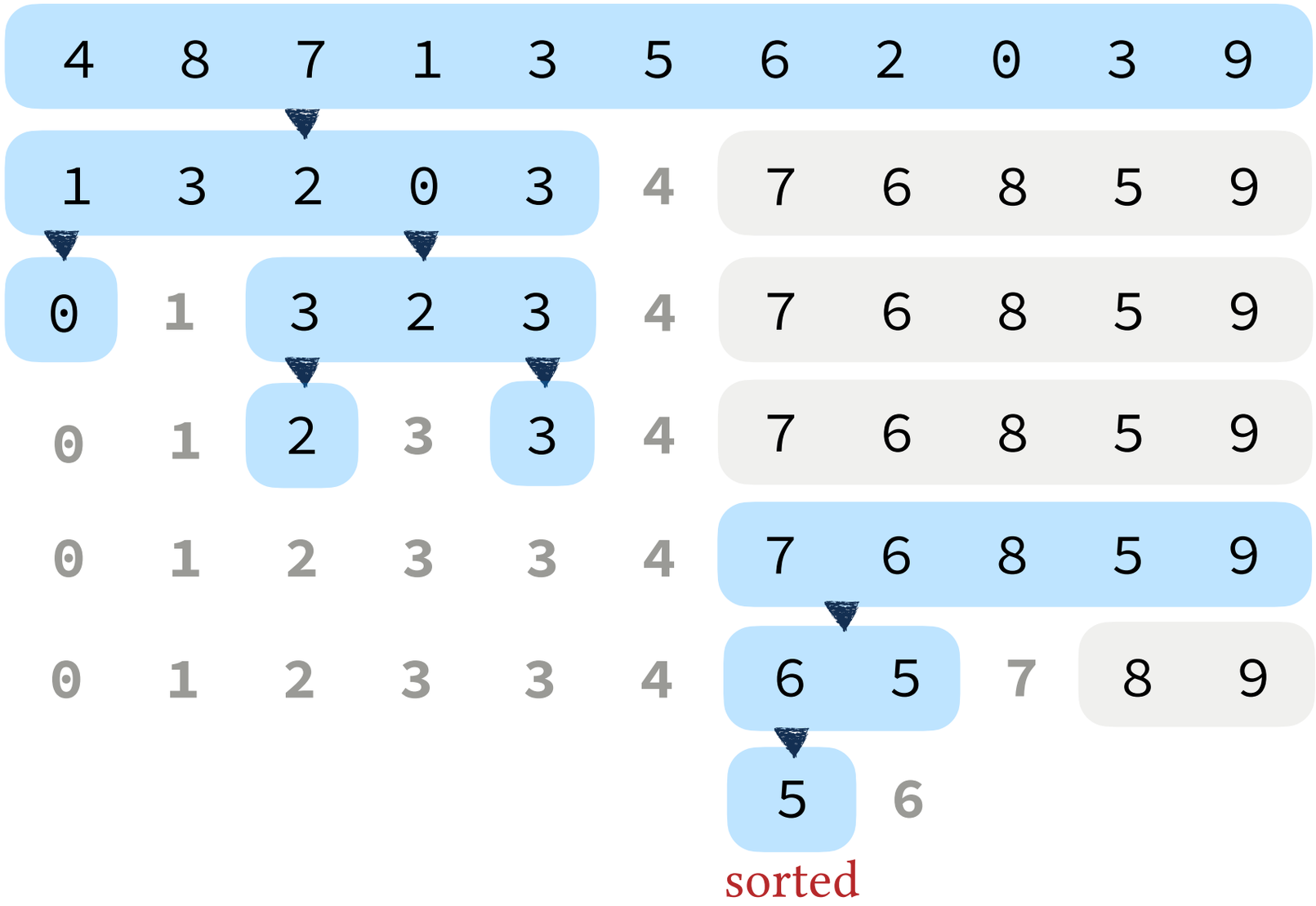
Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



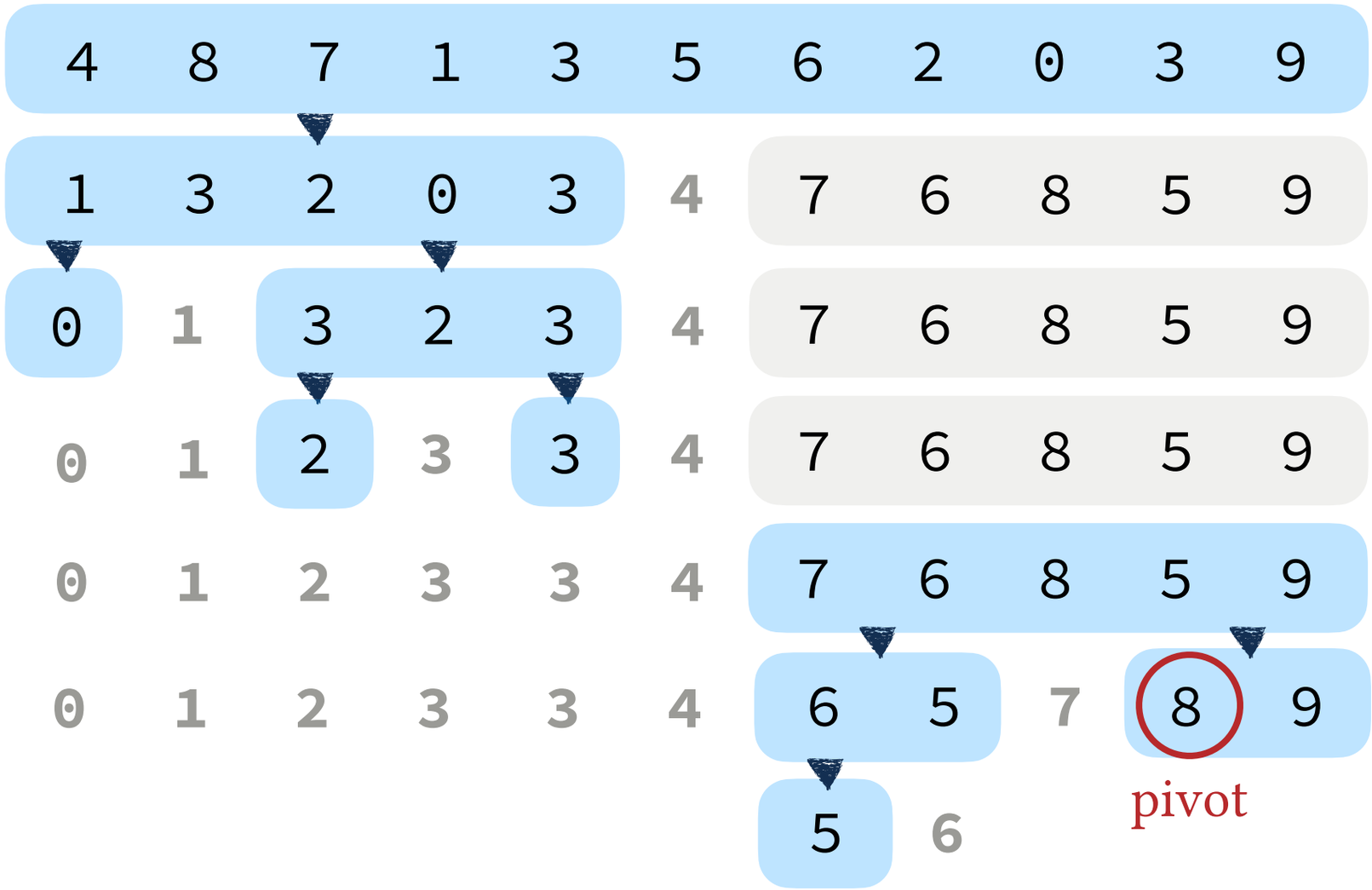
Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



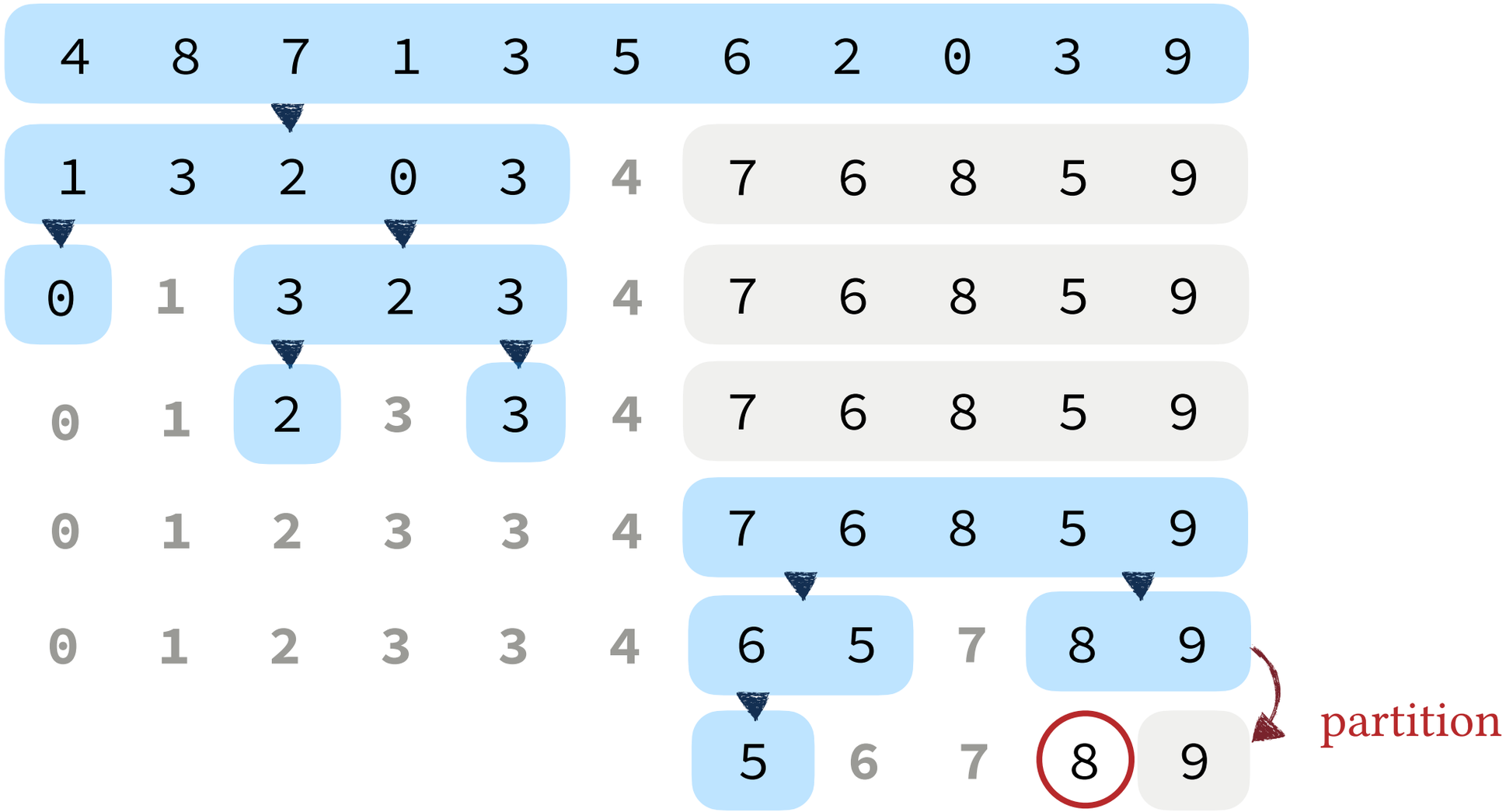
Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



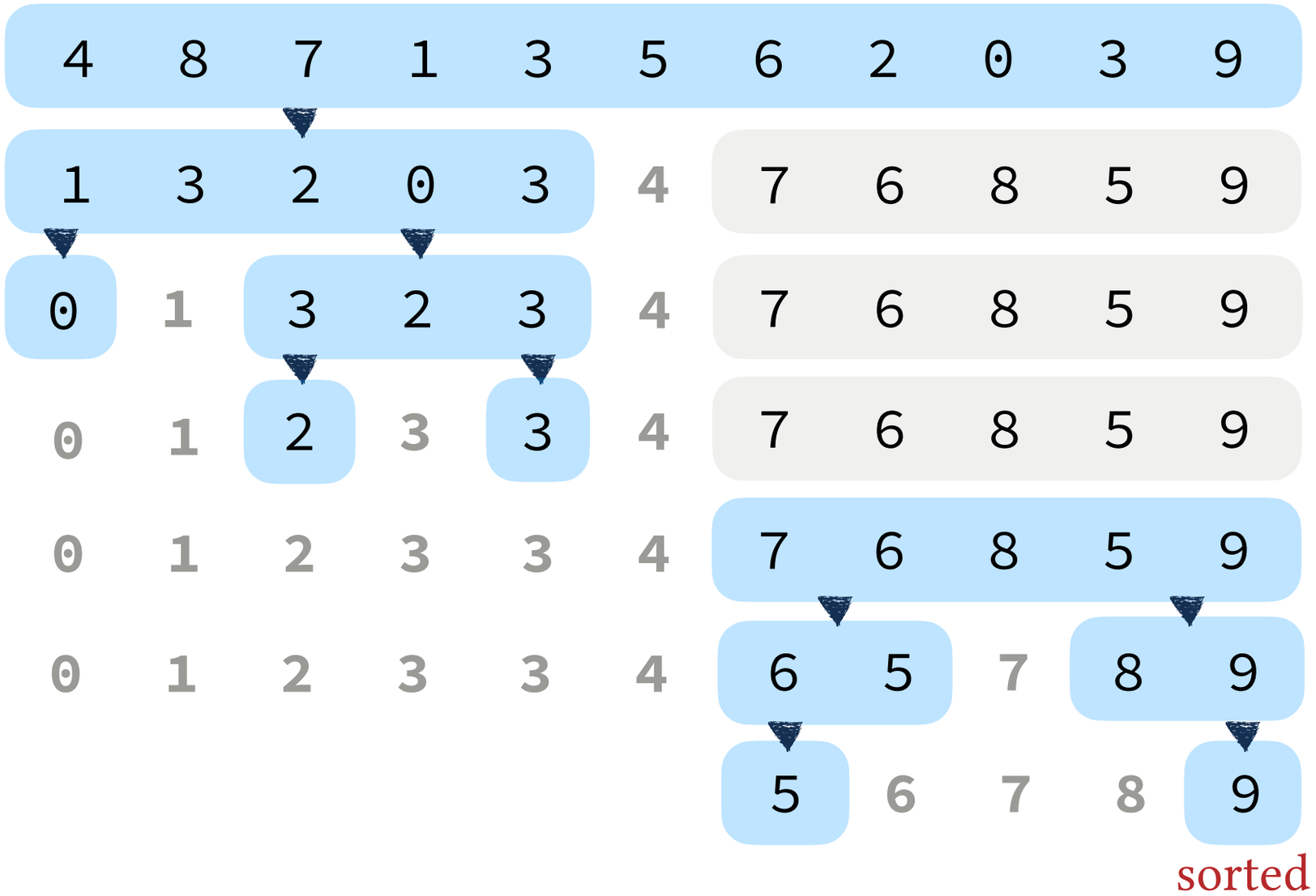
Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



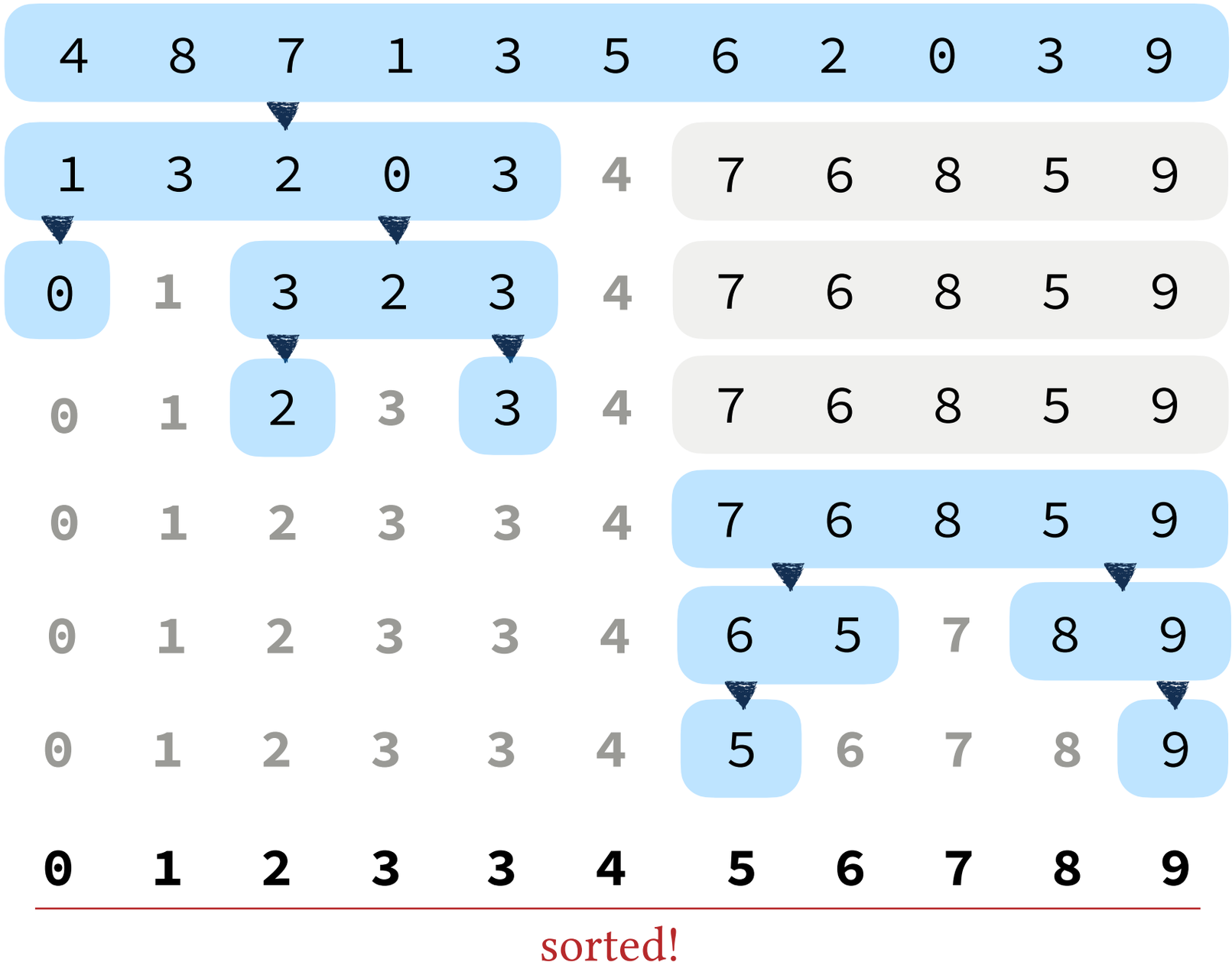
Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Demo



Note. The partitioning performed in these slides is not based on any particular partitioning algorithm

Partitioning an array

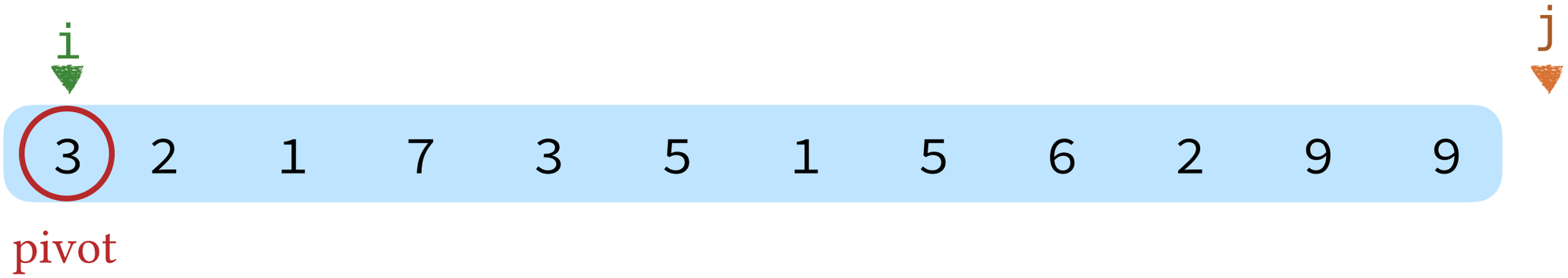
Hoare's Algorithm

Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.

move and then compare

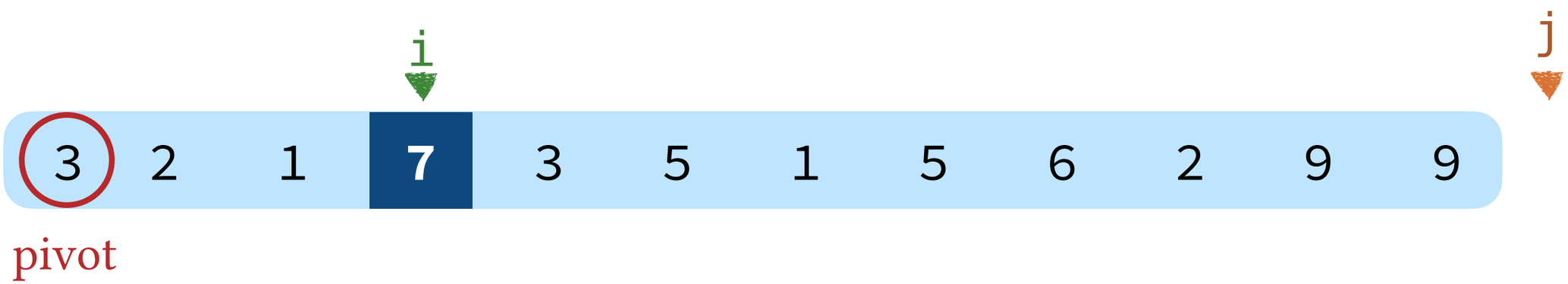


Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.

move and then compare

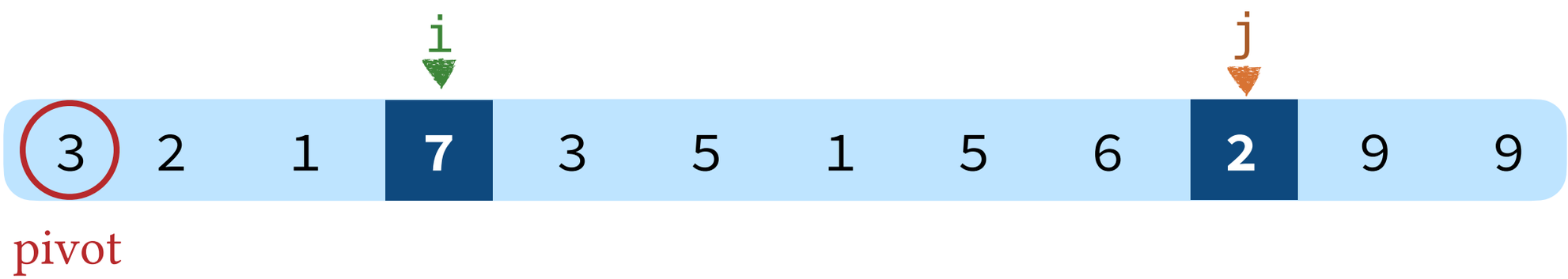


Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.

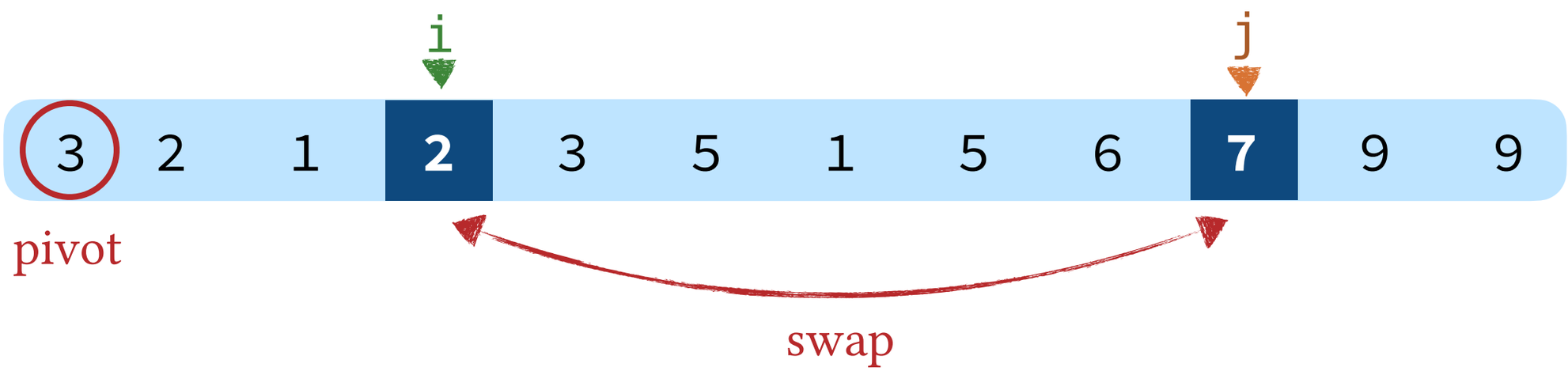
move and then compare



Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

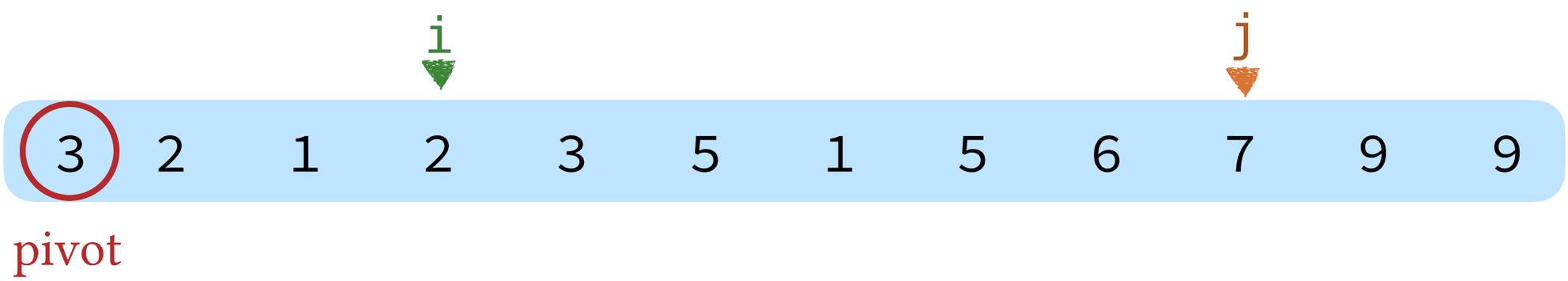
- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.



Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.

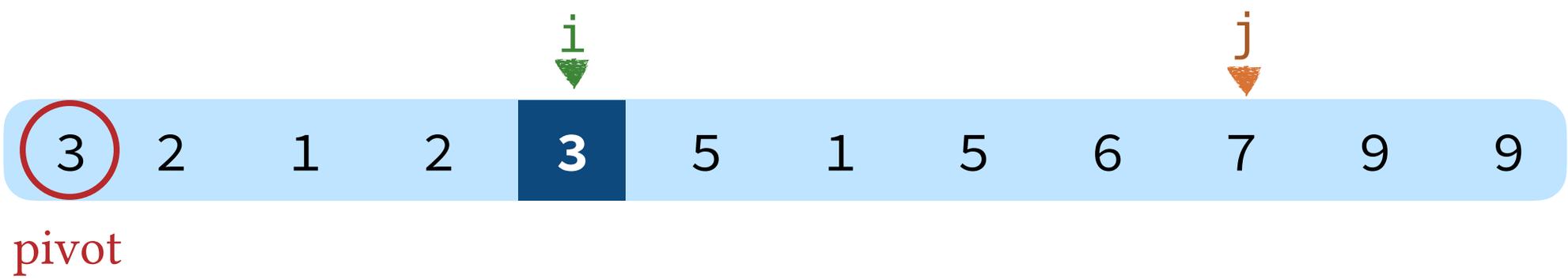


Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.

move and then compare

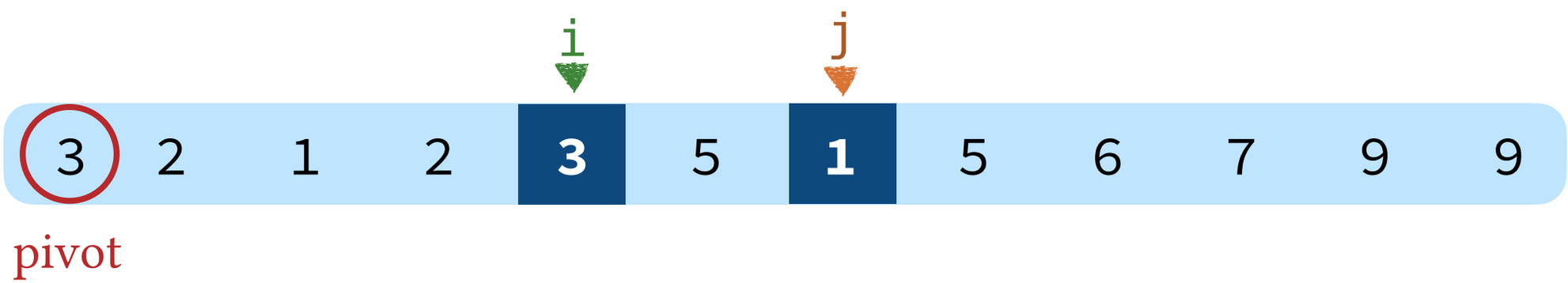


Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.

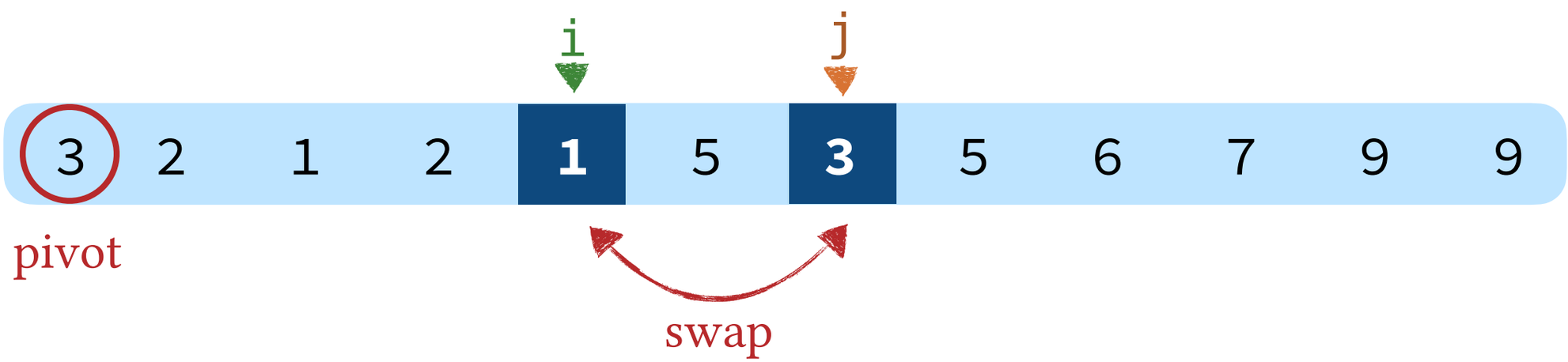
move and then compare



Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

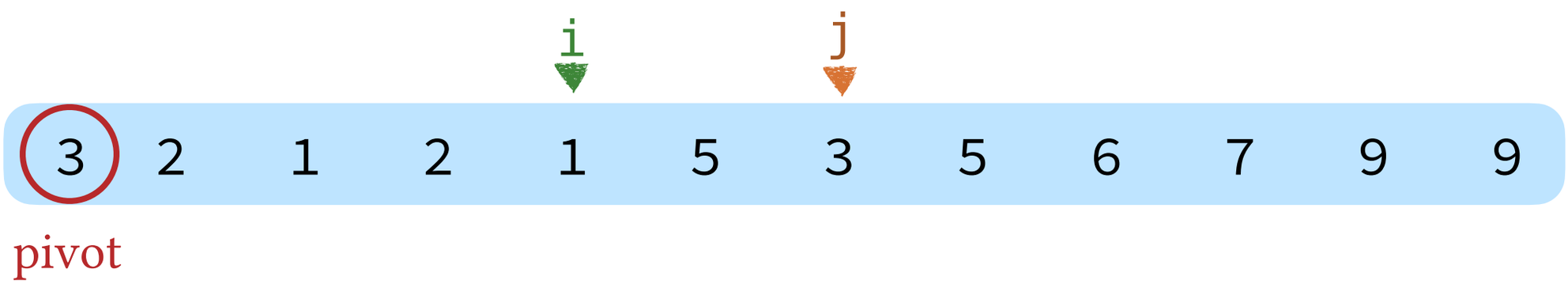
- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.



Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.

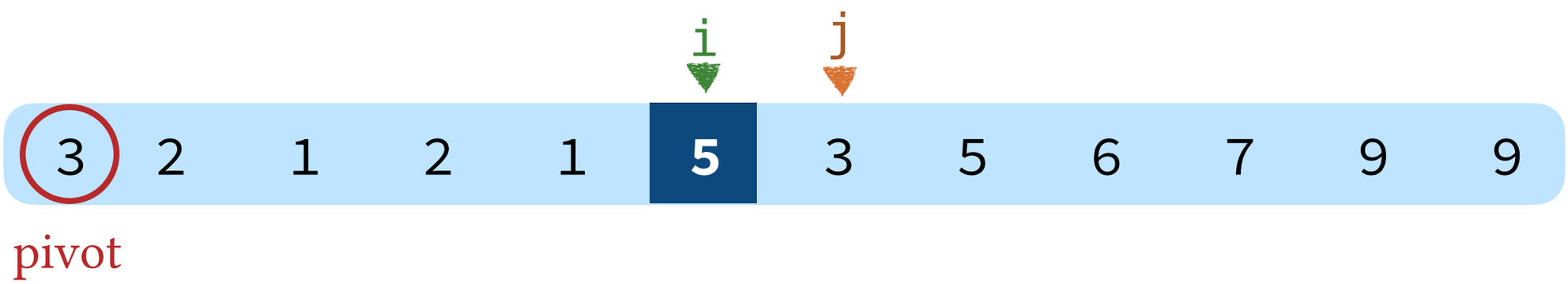


Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.

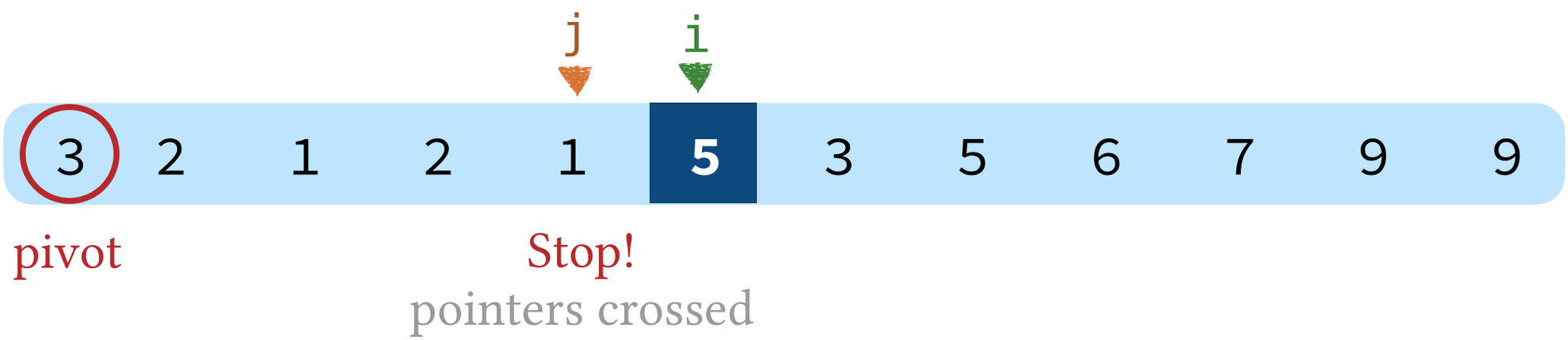
move and then compare



Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`. move and then compare
- Swap `a[i]` and `a[j]`.

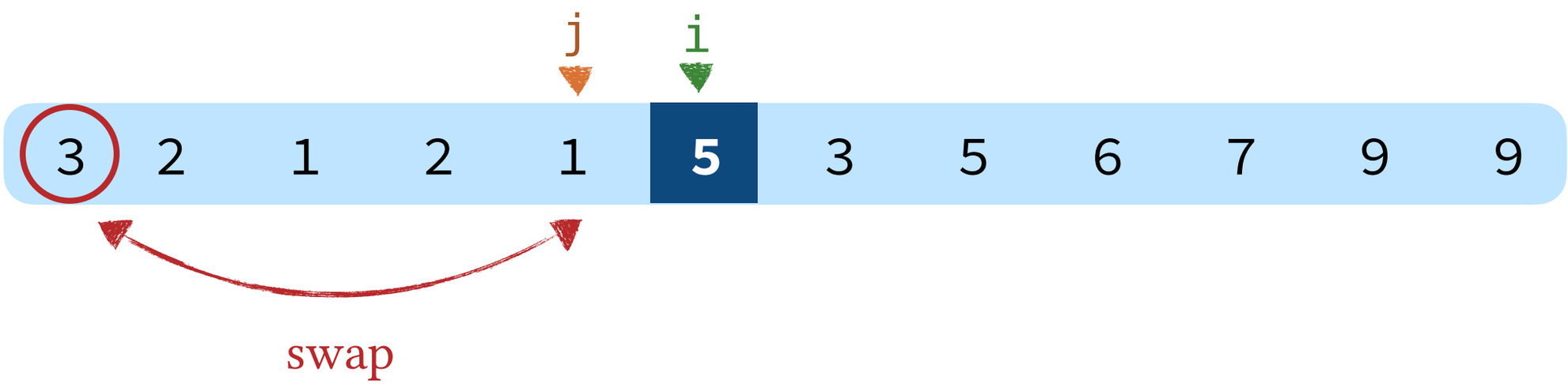


Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.

Swap `pivot` and `a[j]`.

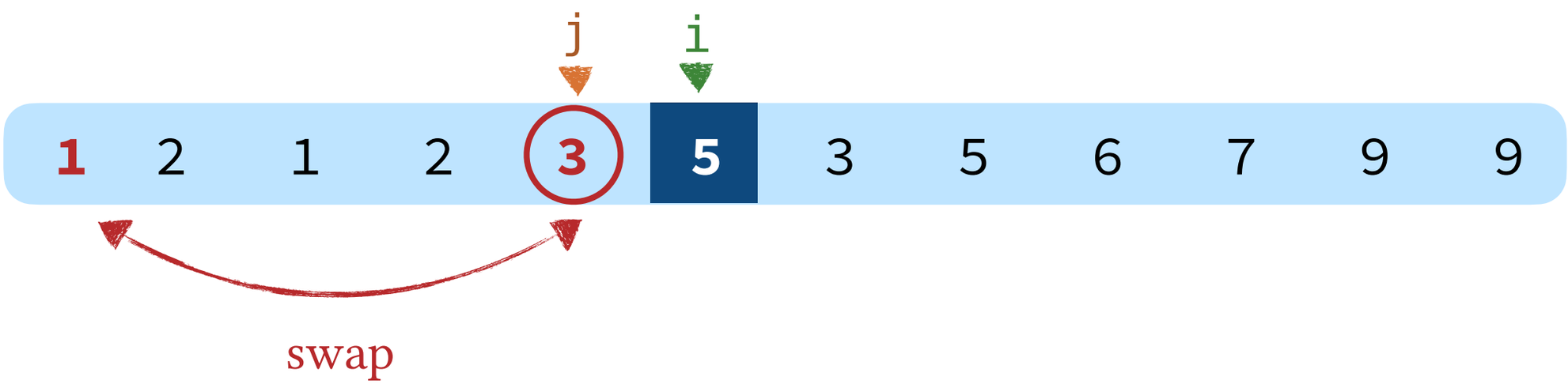


Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.

Swap `pivot` and `a[j]`.

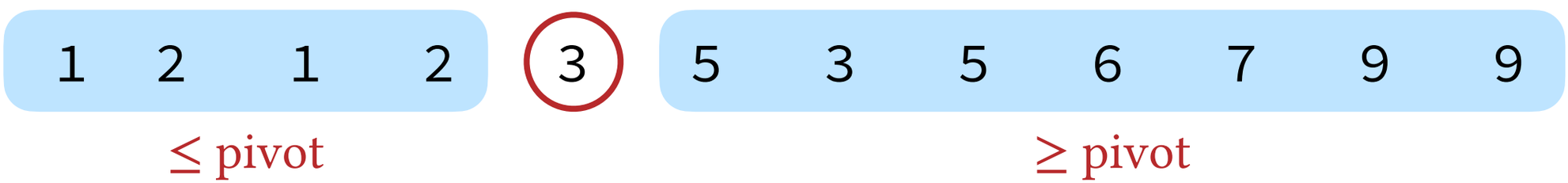


Partitioning Algorithm

Repeat until `i` and `j` pointers cross.

- Scan `i` **left-to-right** as long as `(a[i] < pivot)`.
- Scan `j` **right-to-left** so long as `(a[j] > pivot)`.
- Swap `a[i]` and `a[j]`.

Swap `pivot` and `a[j]`.



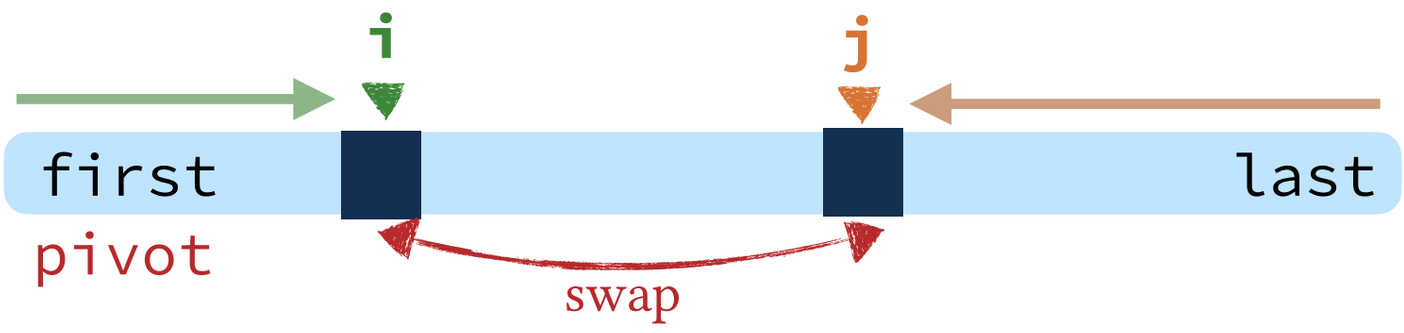
Partitioning Algorithm

```
PARTITION(a[], first, last)

pivot = a[first], i = first, j = last+1

repeat forever:
    do i = i+1
    while (i < last and a[i] < pivot)

    do j = j-1
    while (j > first and a[j] > pivot)

    if i >= j: break
    else: swap(a[i], a[j])
```

scan left-to-right

scan right-to-left

swap if pointers did not cross



Partitioning Algorithm

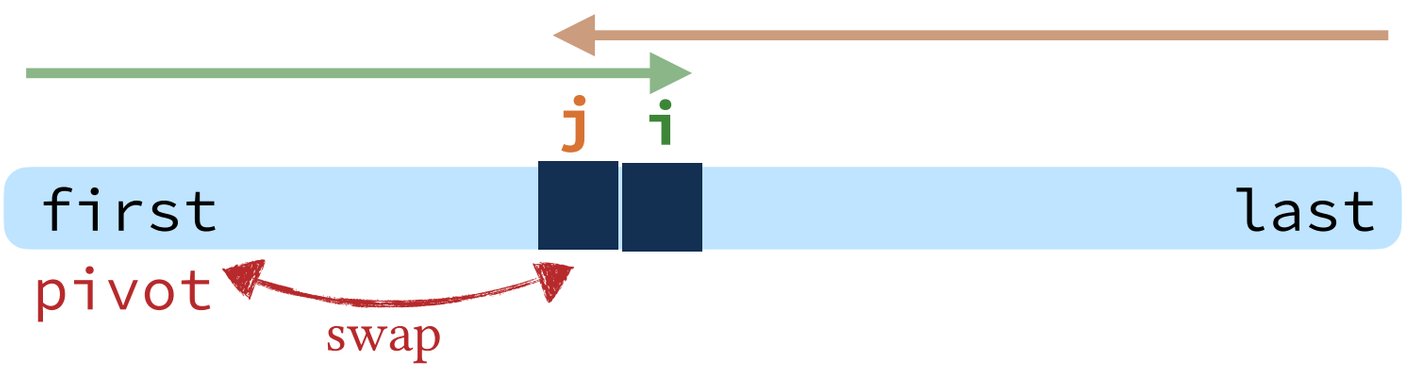
```
PARTITION(a[], first, last)

pivot = a[first], i = first, j = last+1

repeat forever:
    do i = i+1
    while (i < last and a[i] < pivot)

    do j = j-1
    while (j > first and a[j] > pivot)

    if i >= j: break
    else: swap(a[i], a[j])

swap(a[first], a[j])
return j
```

scan left-to-right

scan right-to-left

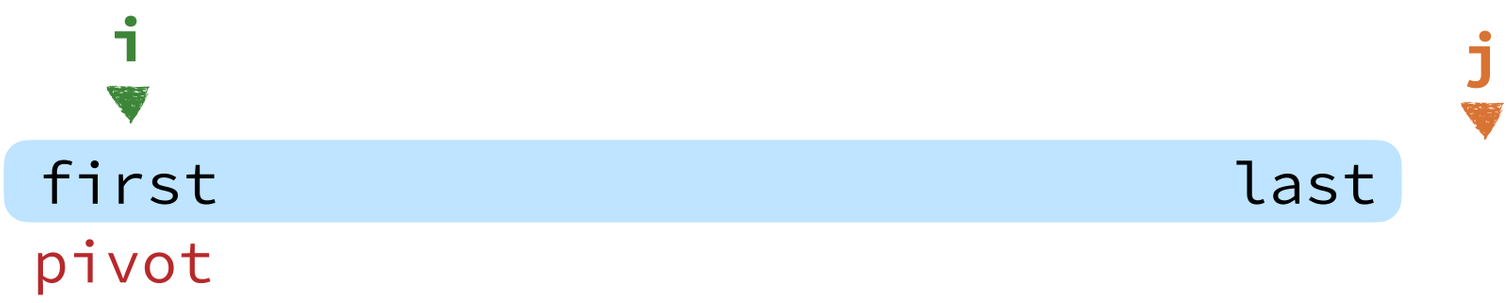
swap if pointers did not cross

place the pivot in its right position



Partitioning Algorithm

```
PARTITION(a[], first, last)

pivot = a[first], i = first, j = last+1

repeat forever:
    do i = i+1
    while (i < last and a[i] < pivot)

    do j = j-1
    while (j > first and a[j] > pivot)

    if i >= j: break
    else: swap(a[i], a[j])

swap(a[first], a[j])
return j
```



Running Time:

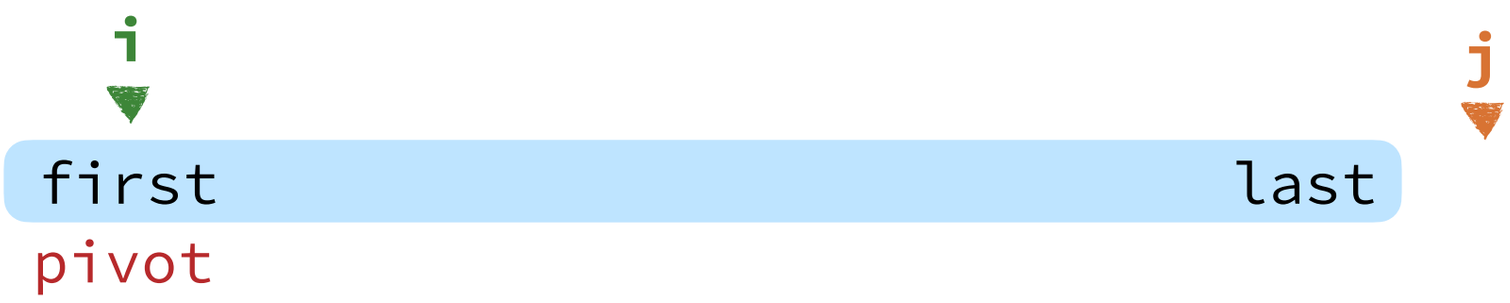
	best case	worst case
swaps	1	$\sim \frac{1}{2}n$
data compares		

e.g. sorted array

if every iteration of the outer loop causes a swap, *i* and *j* meet in the middle

Partitioning Algorithm

```
PARTITION(a[], first, last)

pivot = a[first], i = first, j = last+1

repeat forever:
    do i = i+1
    while (i < last and a[i] < pivot)

    do j = j-1
    while (j > first and a[j] > pivot)

    if i >= j: break
    else: swap(a[i], a[j])

swap(a[first], a[j])
return j
```



Running Time:

	best case	worst case
swaps	1	$\sim \frac{1}{2}n$
data compares	$\sim n$	

all the elements are always compared to the pivot (the worst case is exactly n+1 compares)

Quicksort Analysis

General Case: Pivot can be anywhere!

left and right partitions can be of different sizes.

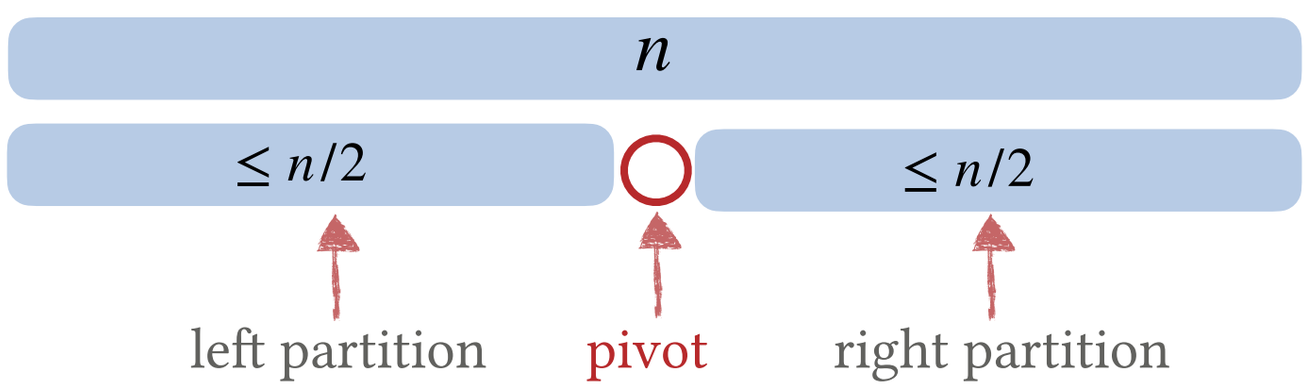
Best Case: Pivot is always the median.

left and right partitions are (almost) equal in size.

Number of Compares:

$$T(n) = T(\lfloor \tfrac{n-1}{2} \rfloor) + T(\lceil \tfrac{n-1}{2} \rceil) + \sim n$$

- $T(n)$: time to sort an array of size n
- $T(\lfloor \tfrac{n-1}{2} \rfloor)$: time to sort the *left* partition
- $T(\lceil \tfrac{n-1}{2} \rceil)$: time to sort the *right* partition
- $\sim n$: time to *partition* the array



Quicksort Analysis

General Case: Pivot can be anywhere!

left and right partitions can be of different sizes.

Best Case: Pivot is always the median.

left and right partitions are (almost) equal in size.

Number of Compares:

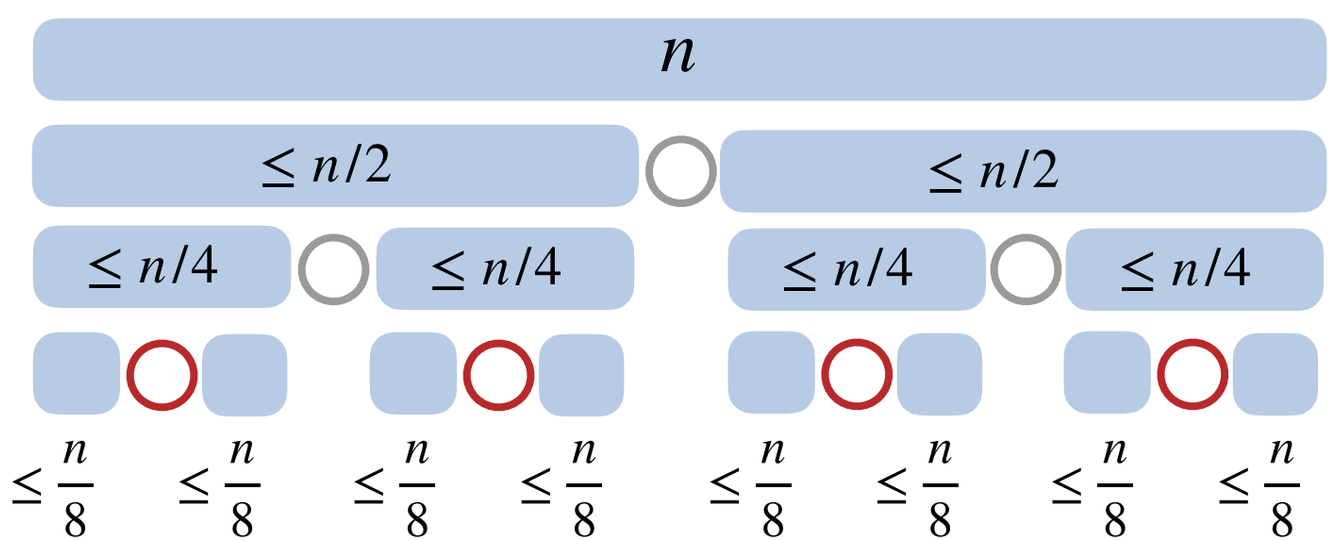
$$T(n) = T(\lfloor \tfrac{n-1}{2} \rfloor) + T(\lceil \tfrac{n-1}{2} \rceil) + \sim n$$

time to sort an array of size n

time to sort the *left* partition

time to sort the *right* partition

time to *partition* the array



Quicksort Analysis

General Case: Pivot can be anywhere!

left and right partitions can be of different sizes.

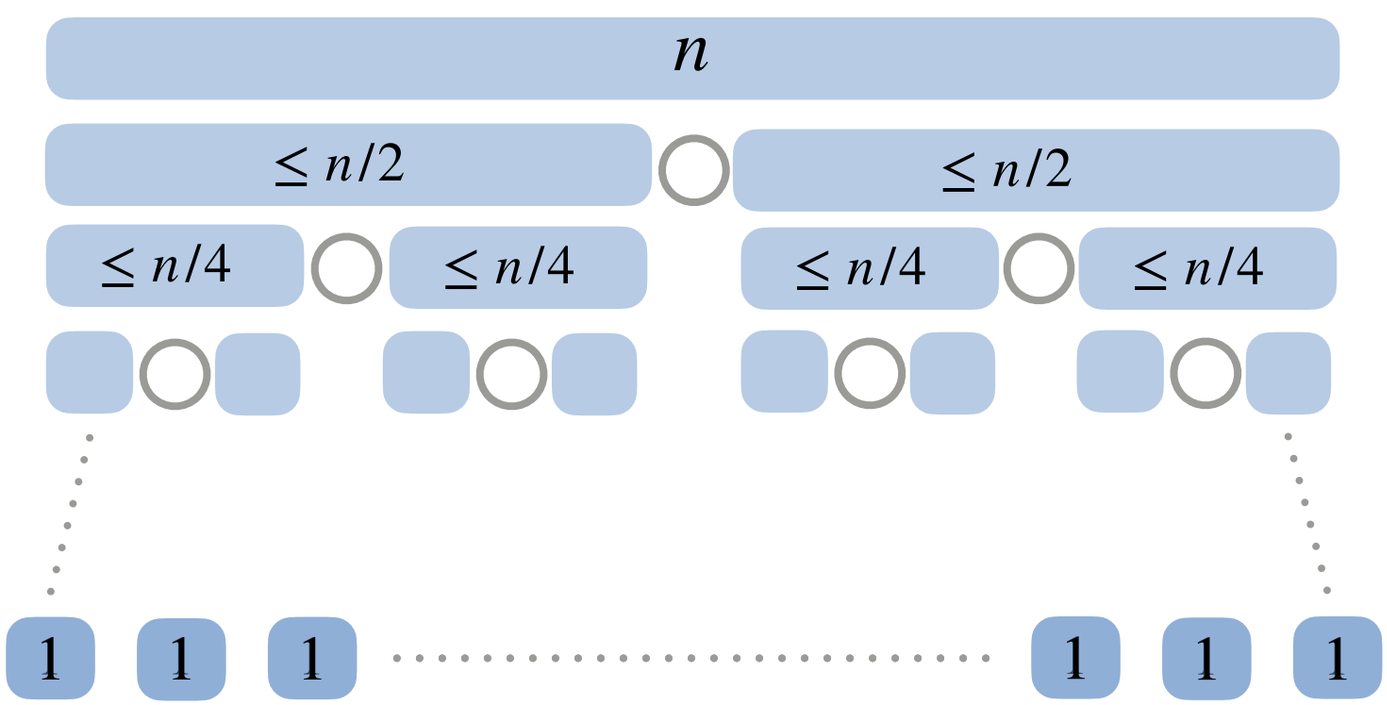
Best Case: Pivot is always the median.

left and right partitions are (almost) equal in size.

Number of Compares:

$$T(n) = T(\lfloor \tfrac{n-1}{2} \rfloor) + T(\lceil \tfrac{n-1}{2} \rceil) + \sim n$$

- $T(n)$: time to sort an array of size n
- $T(\lfloor \tfrac{n-1}{2} \rfloor)$: time to sort the *left* partition
- $T(\lceil \tfrac{n-1}{2} \rceil)$: time to sort the *right* partition
- $\sim n$: time to *partition* the array



Quicksort Analysis

General Case: Pivot can be anywhere!

left and right partitions can be of different sizes.

Best Case: Pivot is always the median.

left and right partitions are (almost) equal in size.

Number of Compares:

$$T(n) = T(\lfloor \tfrac{n-1}{2} \rfloor) + T(\lceil \tfrac{n-1}{2} \rceil) + \sim n$$

- $T(n)$: time to sort an array of size n
- $T(\lfloor \tfrac{n-1}{2} \rfloor)$: time to sort the *left* partition
- $T(\lceil \tfrac{n-1}{2} \rceil)$: time to sort the *right* partition
- $\sim n$: time to *partition* the array

Upper Bound. We can analyze the following recurrence

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + \sim n & \text{if } n > 1 \\ 0 & \text{if } n \leq 1 \end{cases}$$

This is similar to the Merge Sort recurrence equation!

$$T(n) = O(n \log n)$$

with a bit more work we can show that the number of compares is also $\Omega(n \log n)$

Quicksort Analysis

Worst Case: Pivot is always the minimum or always the maximum!

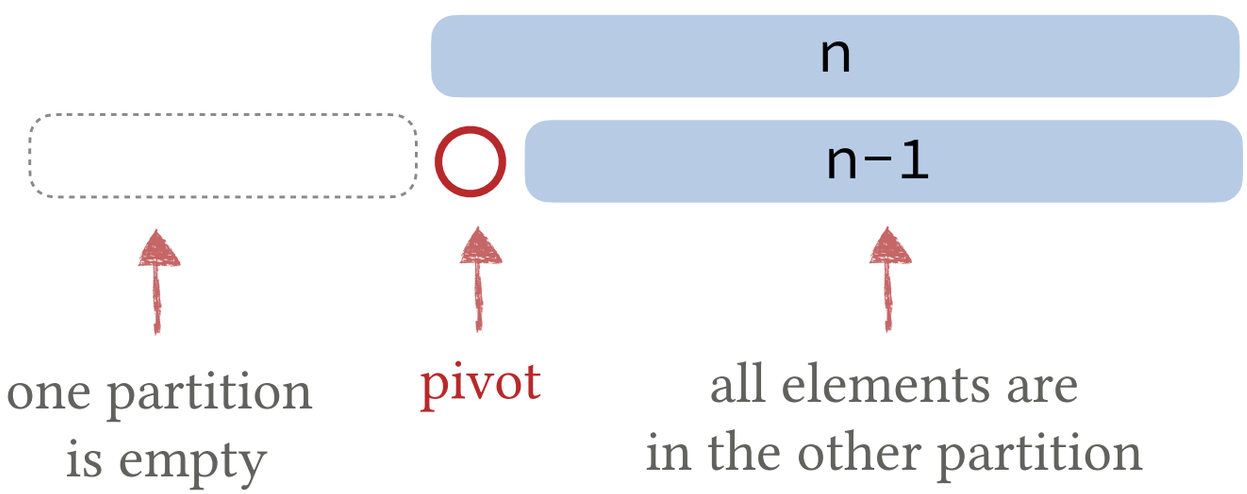
E.g. If the array is sorted in ascending or descending order

Number of Compares:

$$T(n) = T(n-1) + \sim n$$

time to *partition* the array

time to sort an array of size $n-1$



Quicksort Analysis

Worst Case: Pivot is always the minimum or always the maximum!

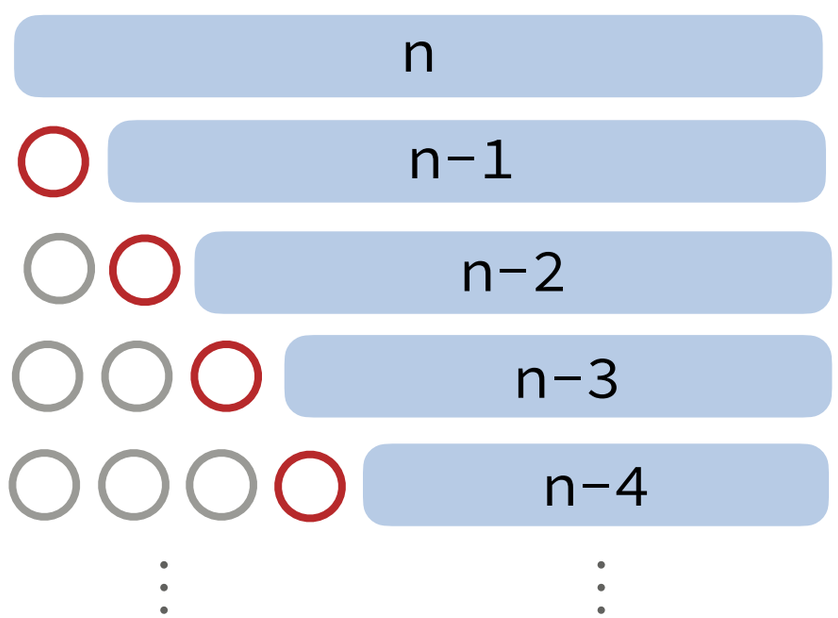
E.g. If the array is sorted in ascending or descending order

Number of Compares:

$$T(n) = T(n-1) + \sim n$$

time to *partition* the array

time to sort an array of size $n-1$



time to partition
n
$n-1$
$n-2$
$n-3$
$n-4$
⋮

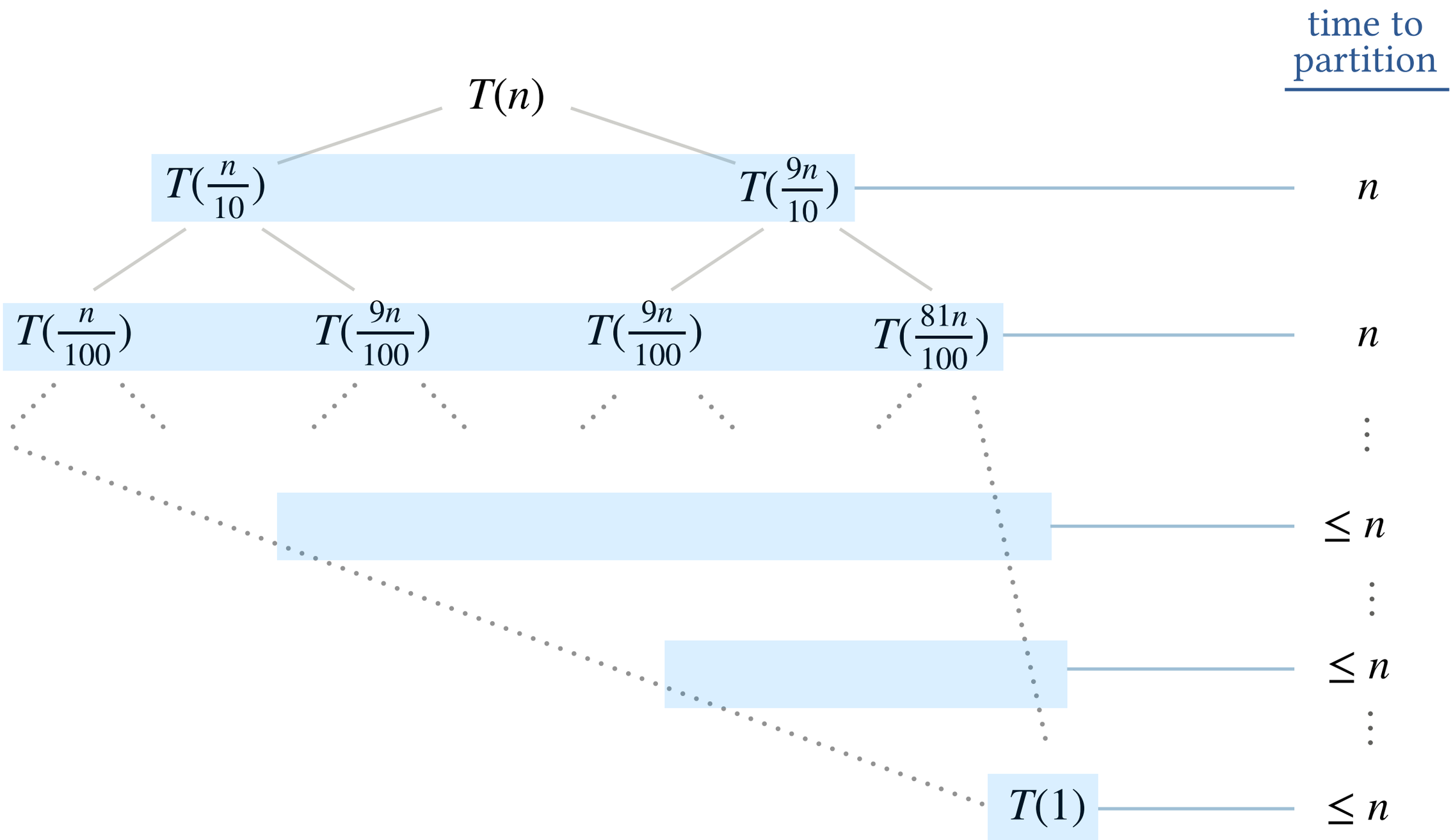
WORST CASE IS BAD!

$$1 + 2 + 3 + \ldots + n = \Theta(n^2)$$

Quicksort Analysis

A Bad (?) Case: Pivot always splits the array into partitions of sizes $\frac{n}{10}$ and $\frac{9n}{10}$.

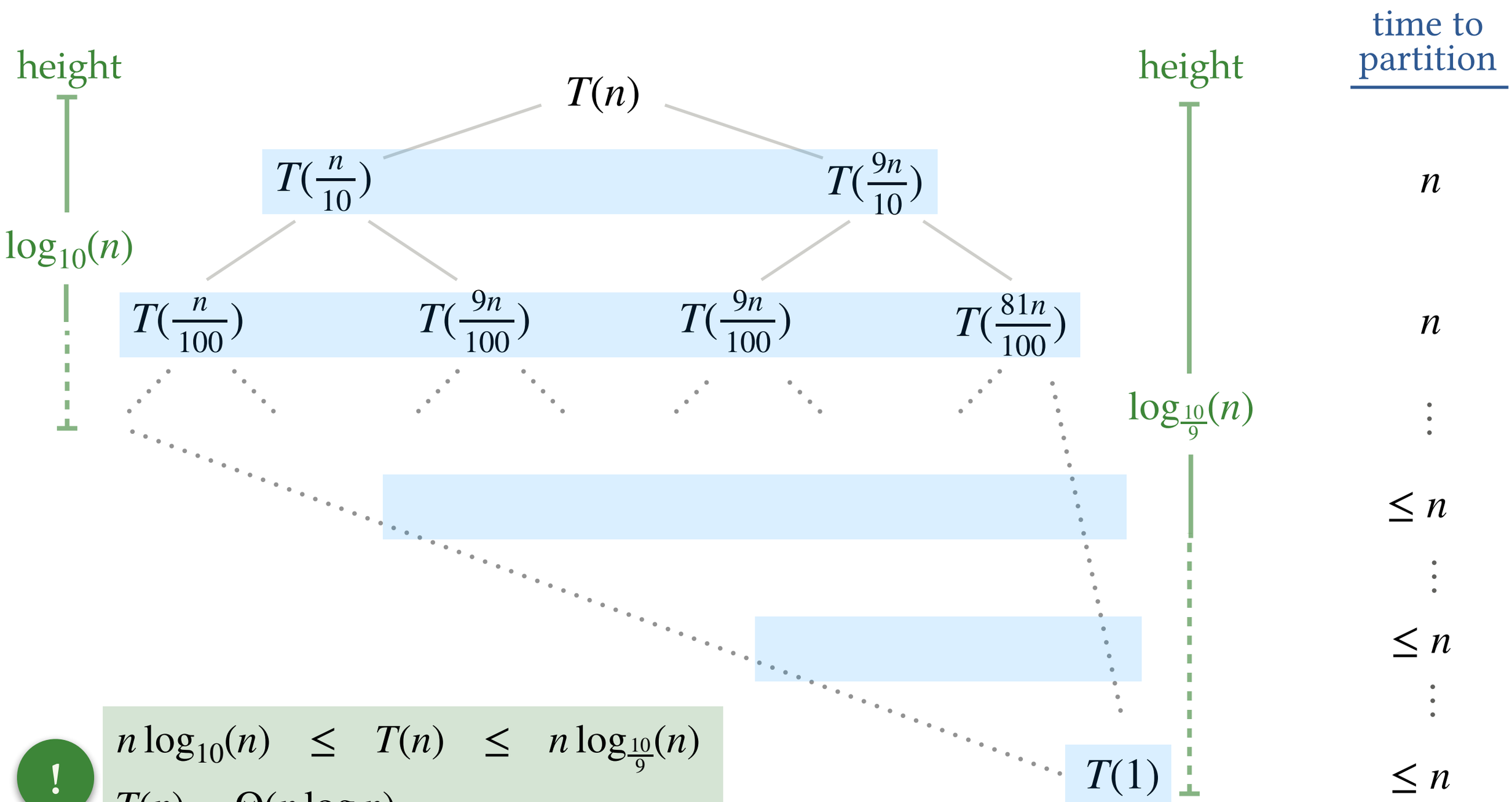
Number of Compares: $T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + \sim n$



Quicksort Analysis

A Good (?) Case: Pivot always splits the array into partitions of sizes $\frac{n}{10}$ and $\frac{9n}{10}$.

Number of Compares: $T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + \sim n$



!

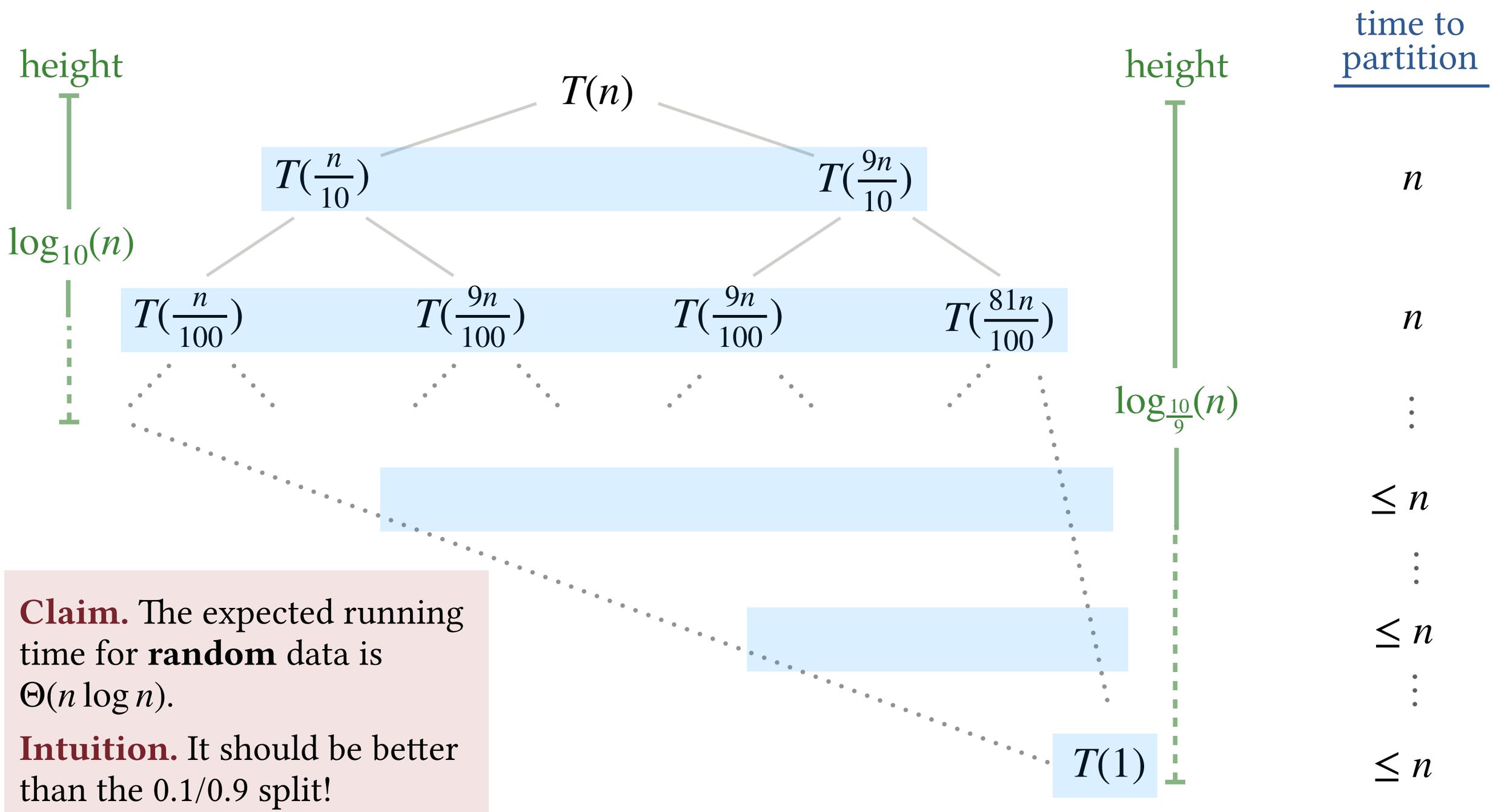
$$n \log_{10}(n) \leq T(n) \leq n \log_{\frac{10}{9}}(n)$$

$$T(n) = \Theta(n \log n)$$

Quicksort Analysis

A Good (?) Case: Pivot always splits the array into partitions of sizes $\frac{n}{10}$ and $\frac{9n}{10}$.

Number of Compares: $T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + \sim n$



Claim. The expected running time for **random** data is $\Theta(n \log n)$.

Intuition. It should be better than the 0.1/0.9 split!

Quicksort Analysis (summary)

Best Case: Pivot is always the median!

$$T(n) \quad = \quad 2T(\tfrac{n}{2}) \quad + \quad \Theta(n) \quad = \quad \Theta(n \log n)$$

Worst Case: Pivot is always the minimum or always the maximum! (e.g. sorted array)

$$T(n) \quad = \quad T(n-1) \quad + \quad \Theta(n) \quad = \quad \Theta(n^2)$$

Good news: Extremely unlikely to happen if the array is shuffled.
This worst case is irrelevant in practice (assuming that we pick the pivot randomly or that we shuffle the array before we quicksort)

Average Case: (math skipped!)

- Expected number of **compares** is $\sim 1.39 n \log_2 n$. → 39% more than Merge Sort
- Expected number of **swaps** is $\sim 0.23 n \log_2 n$.

↓

less than Merge Sort

Quicksort Analysis (summary)

Best Case: Pivot is always the median!

$$T(n) \quad = \quad 2T(\tfrac{n}{2}) \quad + \quad \Theta(n) \quad = \quad \Theta(n \log n)$$

Worst Case: Pivot is always the minimum or always the maximum! (e.g. sorted array)

$$T(n) \quad = \quad T(n-1) \quad + \quad \Theta(n) \quad = \quad \Theta(n^2)$$

Good news: Extremely unlikely to happen if the array is shuffled.
This worst case is irrelevant in practice (assuming that we pick the pivot randomly or that we shuffle the array before we quicksort)

Average Case: (math skipped!)

- Expected number of **compares** is $\sim 1.39 n \log_2 n$.
- Expected number of **swaps** is $\sim 0.23 n \log_2 n$.

? **Exercise**. Redo the worst-case and best-case analysis for the **data moves**!

Quicksort Analysis (summary)

Best Case: Pivot is always the median!

$$T(n) \quad = \quad 2T(\tfrac{n}{2}) \quad + \quad \Theta(n) \quad = \quad \Theta(n \log n)$$

Worst Case: Pivot is always the minimum or always the maximum! (e.g. sorted array)

$$T(n) \quad = \quad T(n-1) \quad + \quad \Theta(n) \quad = \quad \Theta(n^2)$$

Good news: Extremely unlikely to happen if the array is shuffled.
This worst case is irrelevant in practice (assuming that we pick the pivot randomly or that we shuffle the array before we quicksort)

Average Case: (math skipped!)

- Expected number of **compares** is $\sim 1.39 n \log_2 n$.
- Expected number of **swaps** is $\sim 0.23 n \log_2 n$.

If the array is not shuffled, the recursion stack can become of size $\Theta(n)$, which is most probably a stack-overflow!

Memory Use: Quicksort is an in-place sorting algorithm. It requires $\Theta(1)$ extra memory for partitioning and $O(\log n)$ extra space for the recursion stack.
(probabilistically guaranteed assuming the array is shuffled)

Optimization Quiz

How do we change the **PARTITION()** function so that the pivot is picked *randomly*?

```
PARTITION(a[], first, last)

pivot = a[first], i = first, j = last+1

repeat forever:
    do i = i+1
    while (i < last and a[i] < pivot)

    do j = j-1
    while (j > first and a[j] > pivot)

    if i >= j: break
    else: swap(a[i], a[j])

swap(a[first], a[j])
return j
```

Optimization Quiz

How do we change the **PARTITION()** function so that the pivot is picked *randomly*?

```
PARTITION(a[], first, last)

index = first + rand()%(last - first + 1)
swap(a[first], a[index])

pivot = a[first], i = first, j = last+1

repeat forever:
    do i = i+1
    while (i < last and a[i] < pivot)

    do j = j-1
    while (j > first and a[j] > pivot)

    if i >= j: break
    else: swap(a[i], a[j])

swap(a[first], a[j])
return j
```

Quicksort: empirical analysis

Running time estimates:

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (n^2)			mergesort (n log n)			quicksort (n log n)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

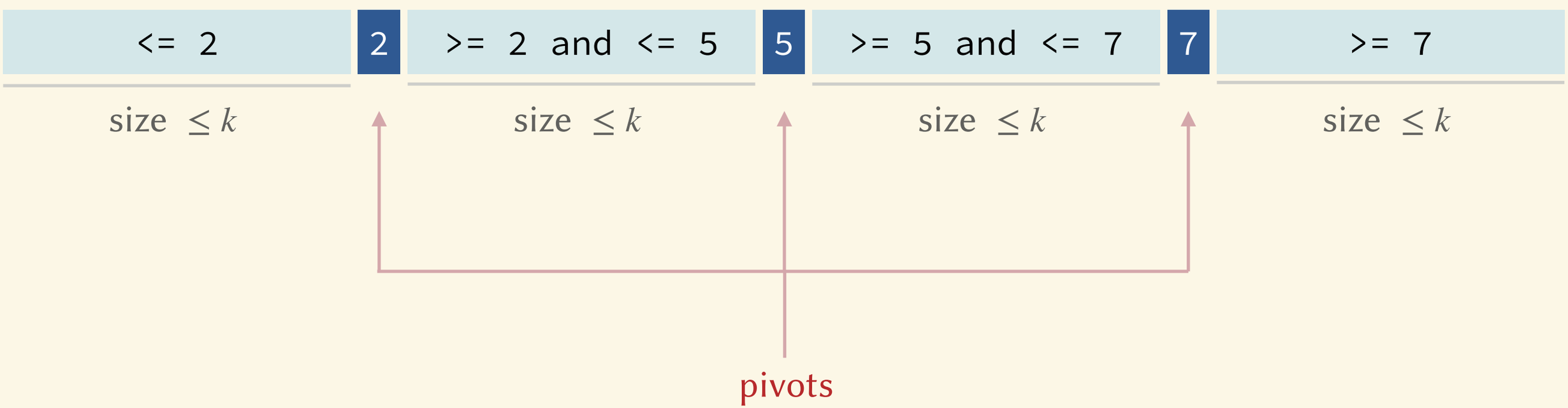
Lesson 2. Great algorithms are better than good ones.

Optimizations

Use *insertion* sort for small Arrays:

- Avoid spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.
- Alternatively: Stop Quicksort if the array size is $\leq k$. When Quicksort ends, run insertion sort. (insertion sort will run in $O(kn)$ time)

Example. Assume the array looks as follows after Quicksort terminates.



Optimizations

Use *insertion* sort for small Arrays:

- Avoid spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.
- Alternatively: Stop Quicksort if the array size is $\leq k$. When Quicksort ends, run insertion sort. (insertion sort will run in $O(kn)$ time)

Example. Assume the array looks as follows after Quicksort terminates.

<= 2	2	>= 2 and <= 5	5	>= 5 and <= 7	7	>= 7
size $\leq k$		size $\leq k$		size $\leq k$		size $\leq k$

- Insertion sort is guaranteed not to move any element more than k positions.
- Elements in a subarray of size k need $O(k^2)$ shifts.
- There are $\leq \frac{n}{k}$ subarrays.
- Total $\leq \frac{n}{k} \times k^2 = O(kn)$, which is good if k is small!

Optimizations

Use *insertion* sort for small Arrays:

- Avoid spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.
- Alternatively: Stop Quicksort if the array size is $\leq k$. When Quicksort ends, run insertion sort. (insertion sort will run in $O(kn)$ time)

Think: Can we use the same alternative strategy for optimizing Merge Sort?

Optimizations

Use *insertion* sort for small Arrays:

- Avoid spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.
- Alternatively: Stop Quicksort if the array size is $\leq k$. When Quicksort ends, run insertion sort. (insertion sort will run in $O(kn)$ time)

Group elements equal to the pivot together:

- **Example:** Dijkstra's 3-way partitioning algorithm.

1	2	1	2	**3**	**3**	**3**	5	6	7	9	9
< pivot				= pivot			> pivot				

! No need to call Quicksort on the middle partition!

Optimizations

Use *insertion* sort for small Arrays:

- Avoid spending a lot of time on many expensive recursive calls at the lower levels of the recursion tree.
- Alternatively: Stop Quicksort if the array size is $\leq k$. When Quicksort ends, run insertion sort. (insertion sort will run in $O(kn)$ time)

Group elements equal to the pivot together:

- **Example:** Dijkstra's 3-way partitioning algorithm.
- Performs well with arrays that have many duplicates.
- Best case becomes linear in n. (if number of distinct elements is small)

Use a pivot-picking strategy (instead of or in addition to shuffling):

- Median of three: Pick the median of the first, last and middle elements.
  Improves the expected number of comparisons to $1.188n \log n$ compares at the expense of a 3% increase in the number of swaps.
- Pseudo-median of nine: Sample 9 elements, divide them into 3 groups and pick the median of the medians.

Dual Pivot Quicksort (2009)

Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort

```
Hello All,

I'd like to share with you new Dual-Pivot Quicksort which is faster than the
known implementations (theoretically and experimental). I'd like to propose
to replace the JDK's Quicksort implementation by new one.

...

The new Dual-Pivot Quicksort uses *two* pivots elements in this manner:

1. Pick an elements P1, P2, called pivots from the array.
2. Assume that P1 <= P2, otherwise swap it.
3. Reorder the array into three parts: those less than the smaller pivot,
   those larger than the larger pivot, and in between are those elements
   between (or equal to) the two pivots.
4. Recursively sort the sub-arrays.

The invariant of the Dual-Pivot Quicksort is:
```

$< P_1$	$\geq P_1 \quad \leq P_2$	$> P_2$

...

https://mail.openjdk.java.net/pipermail/core-libs-dev/2009-September/002630.html

degenerates to 3-way partitioning if $P_1 = P_2$

Optimization Quiz

What is the effect of the following optimization?

```
PARTITION(a[], first, last)

pivot = a[first], i = first, j = last+1

repeat forever:
    do i = i+1
    while (i < last and a[i]  <= pivot)

    do j = j-1
    while (j > first and a[j] >= pivot)

    if i >= j: break
    else: swap(a[i], a[j])


swap(a[first], a[j])
return j
```

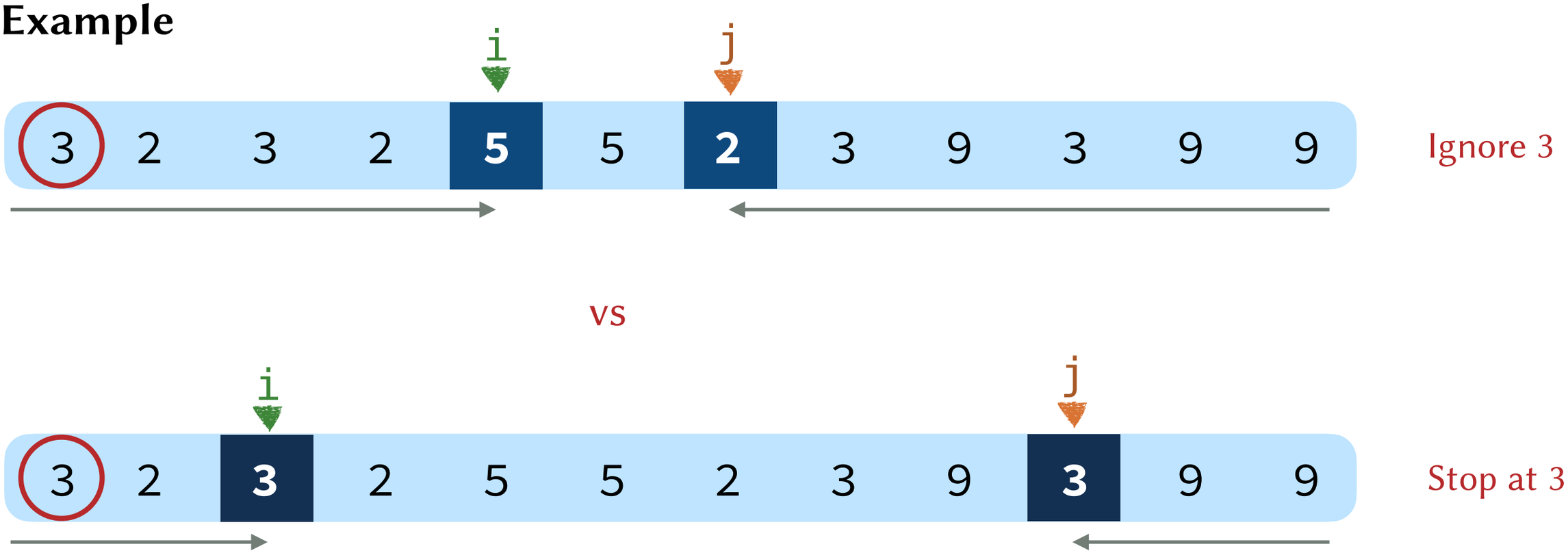
In other words: *Do not swap elements that are equal to the pivot!*

Optimization Quiz

What is the effect of the following optimization?

```
PARTITION(a[], first, last)
```

Example

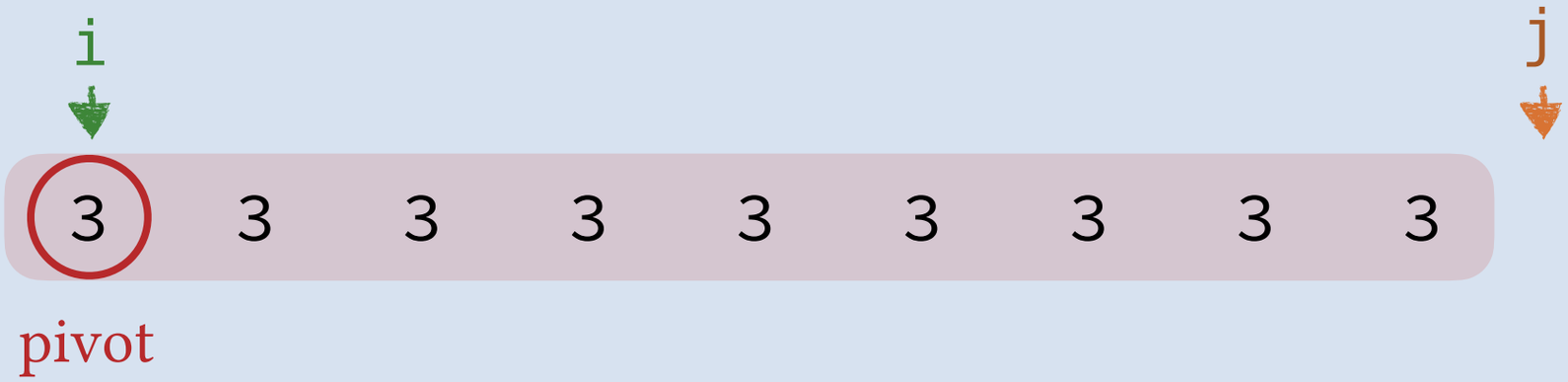



```
return j
```

In other words: *Do not swap elements that are equal to the pivot!*

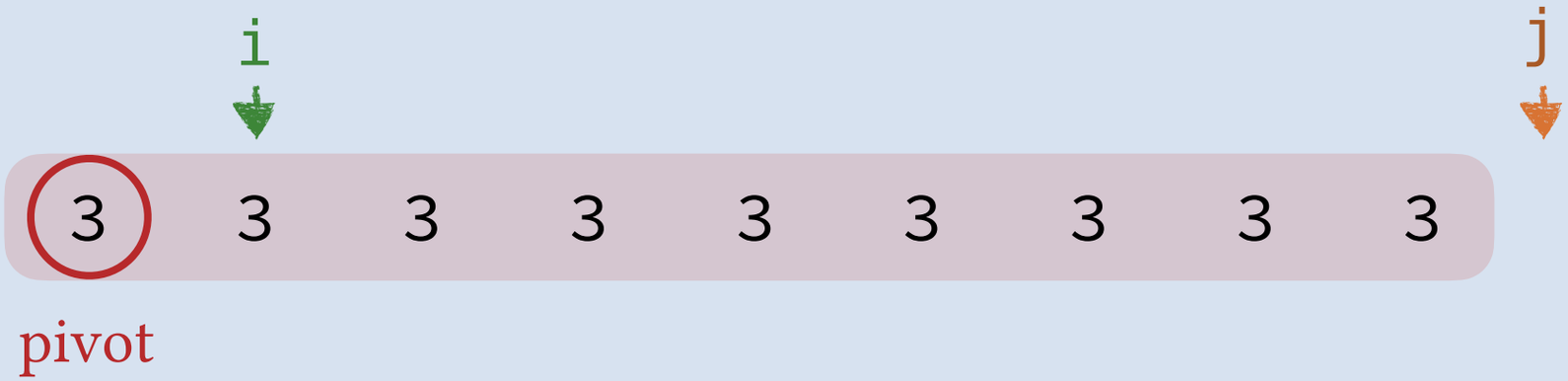
Optimization Quiz

Bad. Do not stop on elements equal to the pivot.



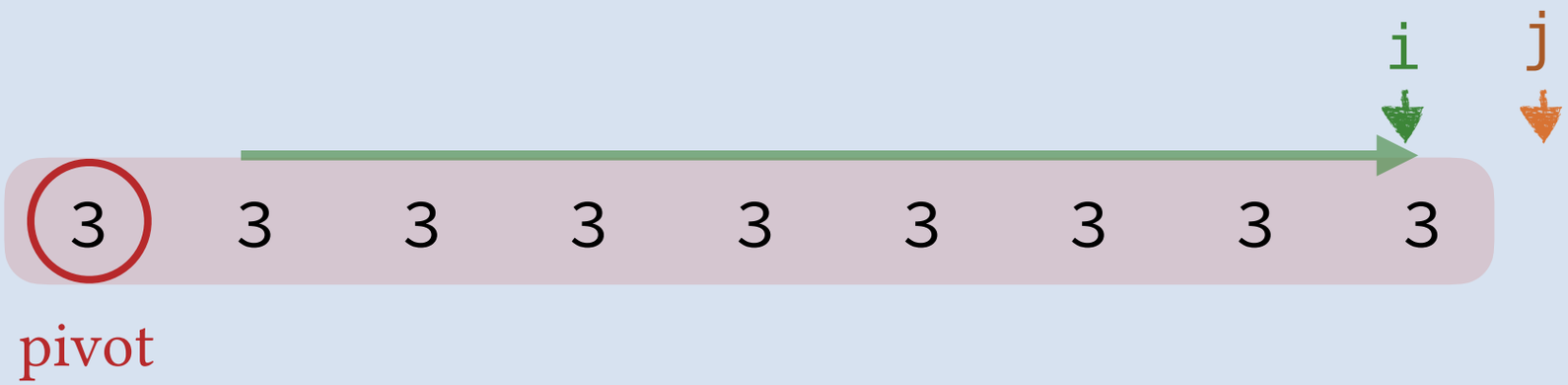
Optimization Quiz

Bad. Do not stop on elements equal to the pivot.



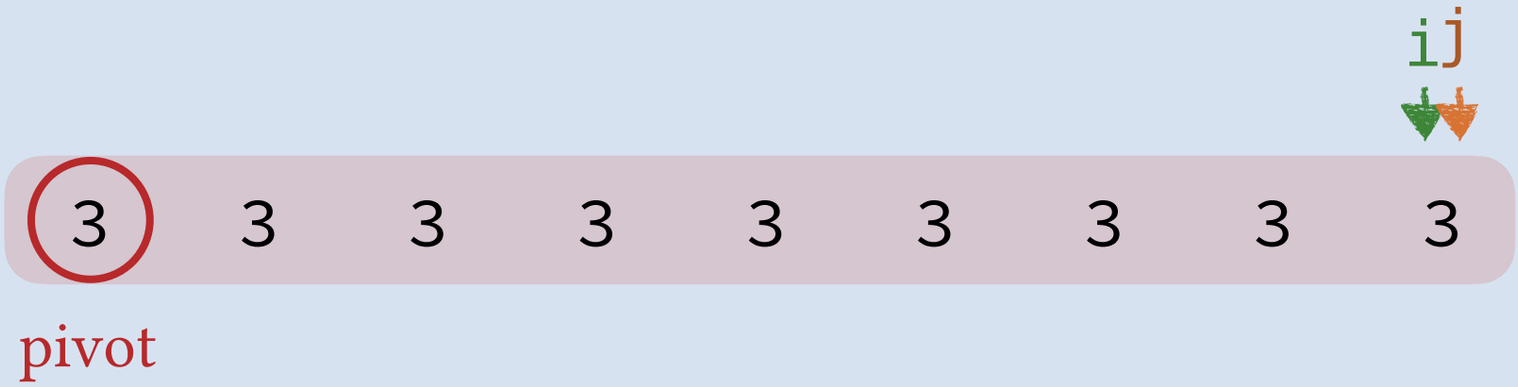
Optimization Quiz

Bad. Do not stop on elements equal to the pivot.



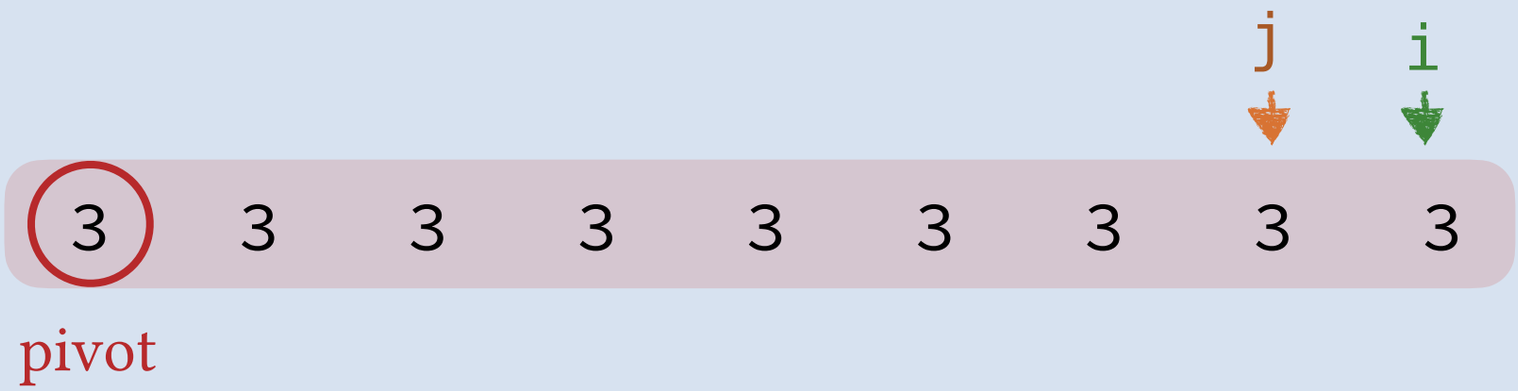
Optimization Quiz

Bad. Do not stop on elements equal to the pivot.



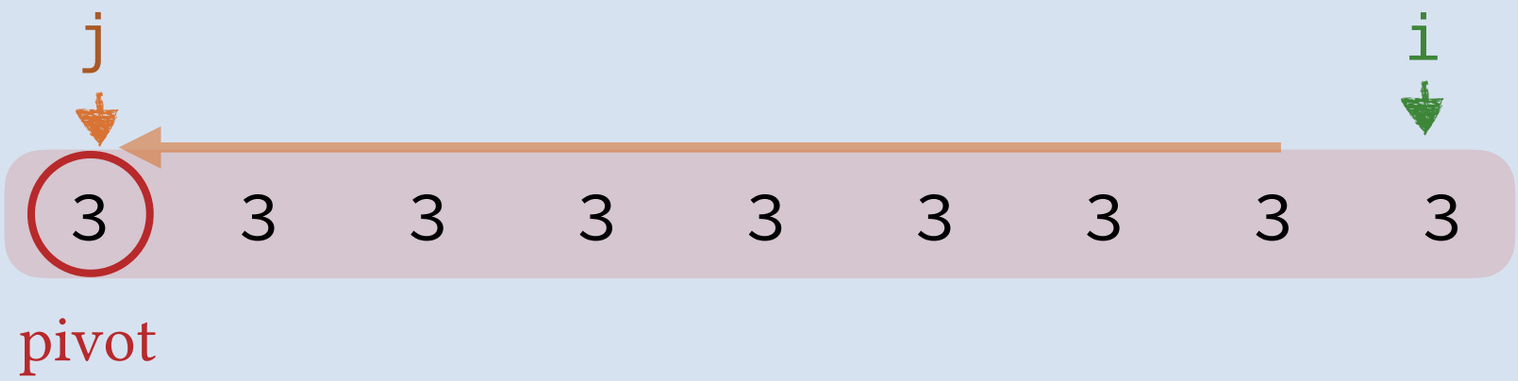
Optimization Quiz

Bad. Do not stop on elements equal to the pivot.



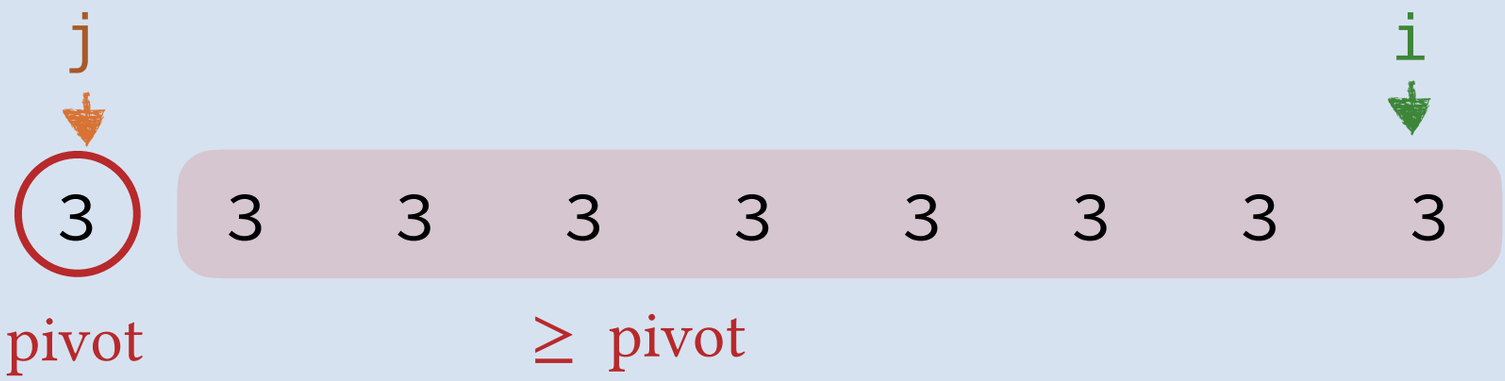
Optimization Quiz

Bad. Do not stop on elements equal to the pivot.



Optimization Quiz

Bad. Do not stop on elements equal to the pivot.



Optimization Quiz

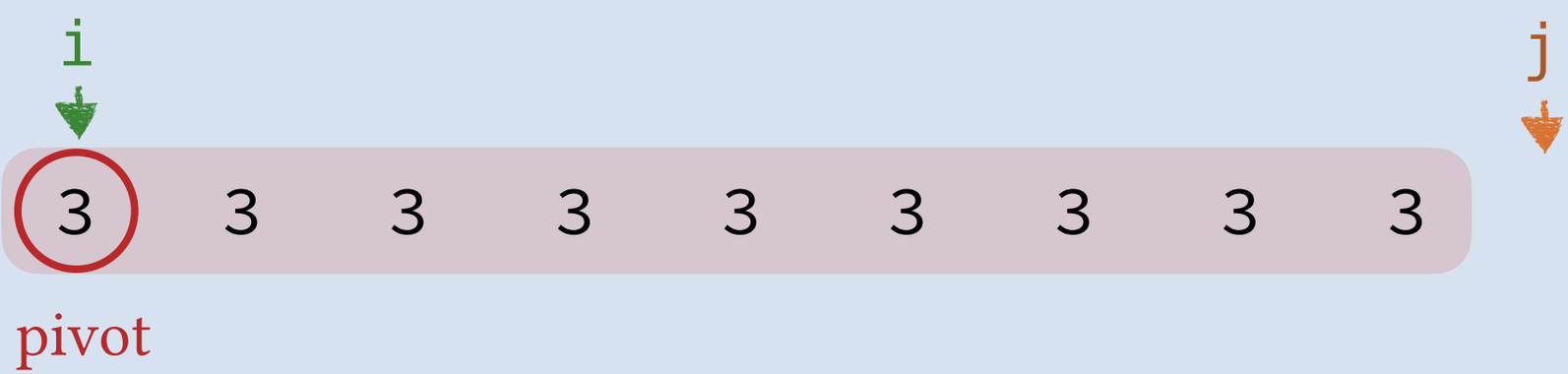
Bad. Do not stop on elements equal to the pivot.

Having many duplicate keys leads to worst case performance!

Optimization Quiz

Bad. Do not stop on elements equal to the pivot.
Having many duplicate keys leads to worst case performance!

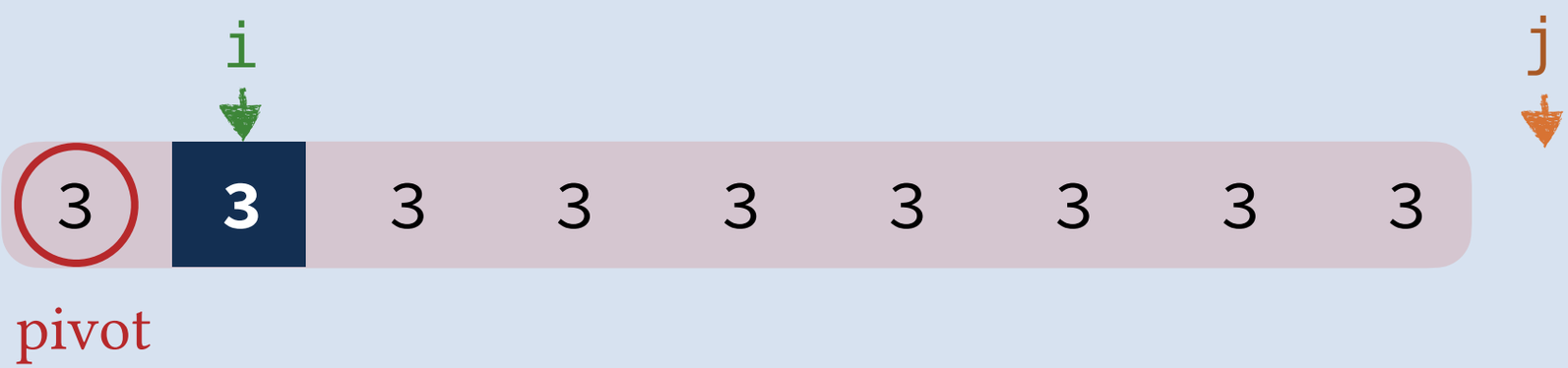
Better. Stop on elements equal to the pivot.
More swaps but the pivot becomes the middle element!



Optimization Quiz

Bad. Do not stop on elements equal to the pivot.
Having many duplicate keys leads to worst case performance!

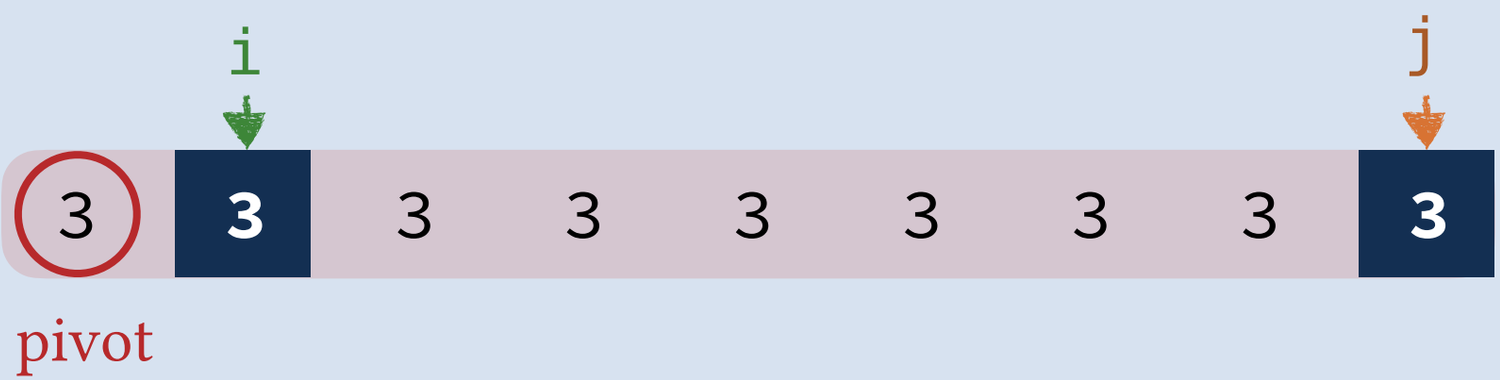
Better. Stop on elements equal to the pivot.
More swaps but the pivot becomes the middle element!



Optimization Quiz

Bad. Do not stop on elements equal to the pivot.
Having many duplicate keys leads to worst case performance!

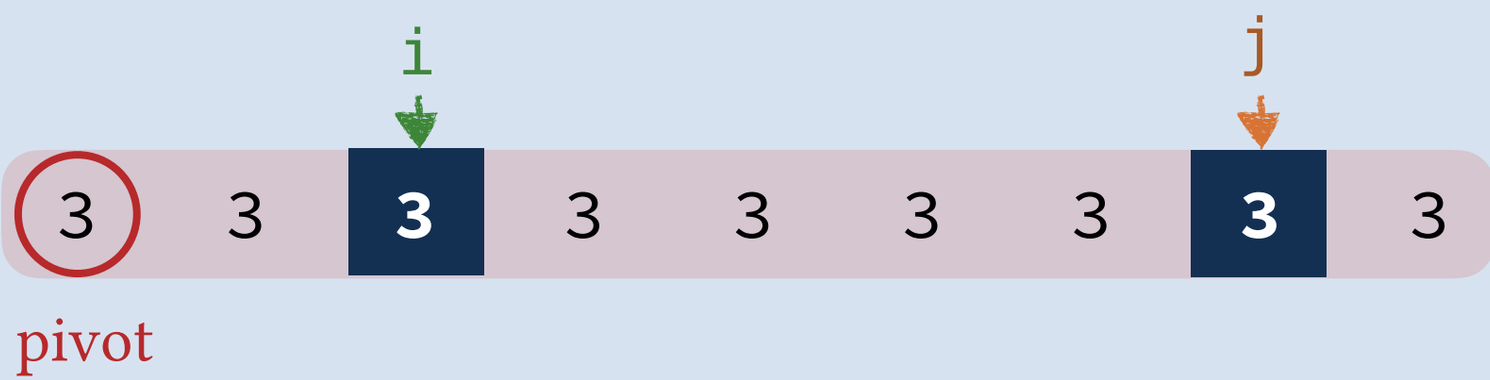
Better. Stop on elements equal to the pivot.
More swaps but the pivot becomes the middle element!



Optimization Quiz

Bad. Do not stop on elements equal to the pivot.
Having many duplicate keys leads to worst case performance!

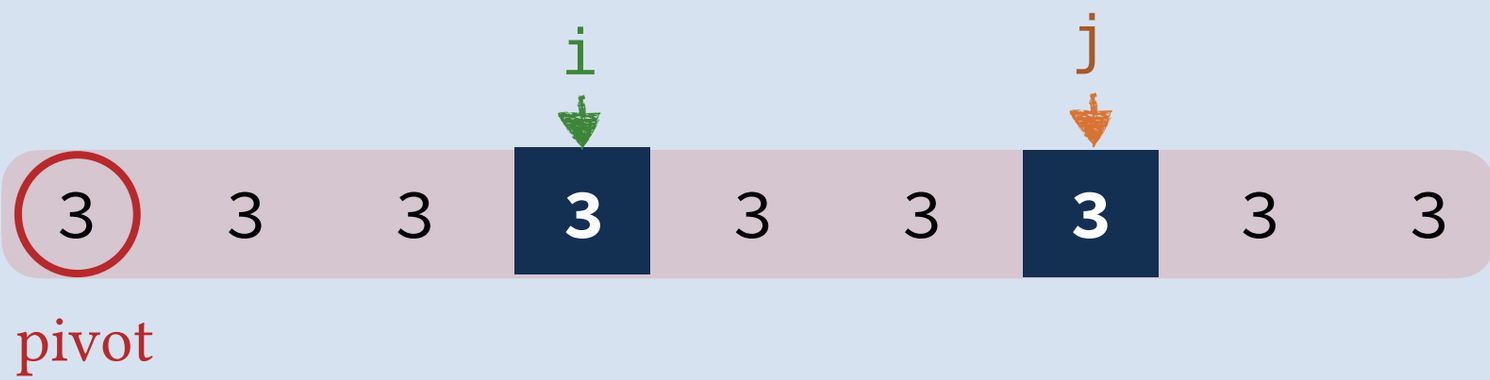
Better. Stop on elements equal to the pivot.
More swaps but the pivot becomes the middle element!



Optimization Quiz

Bad. Do not stop on elements equal to the pivot.
Having many duplicate keys leads to worst case performance!

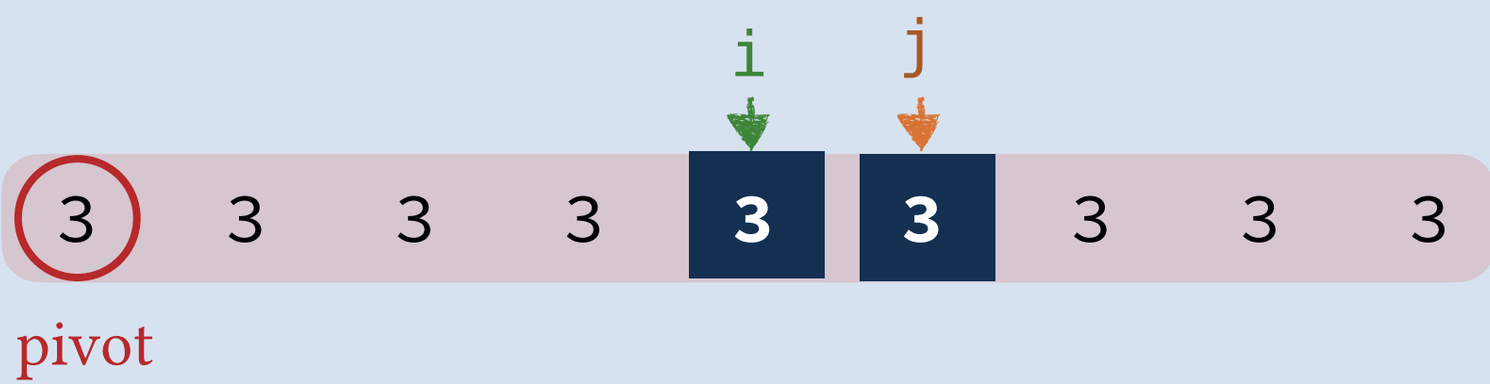
Better. Stop on elements equal to the pivot.
More swaps but the pivot becomes the middle element!



Optimization Quiz

Bad. Do not stop on elements equal to the pivot.
Having many duplicate keys leads to worst case performance!

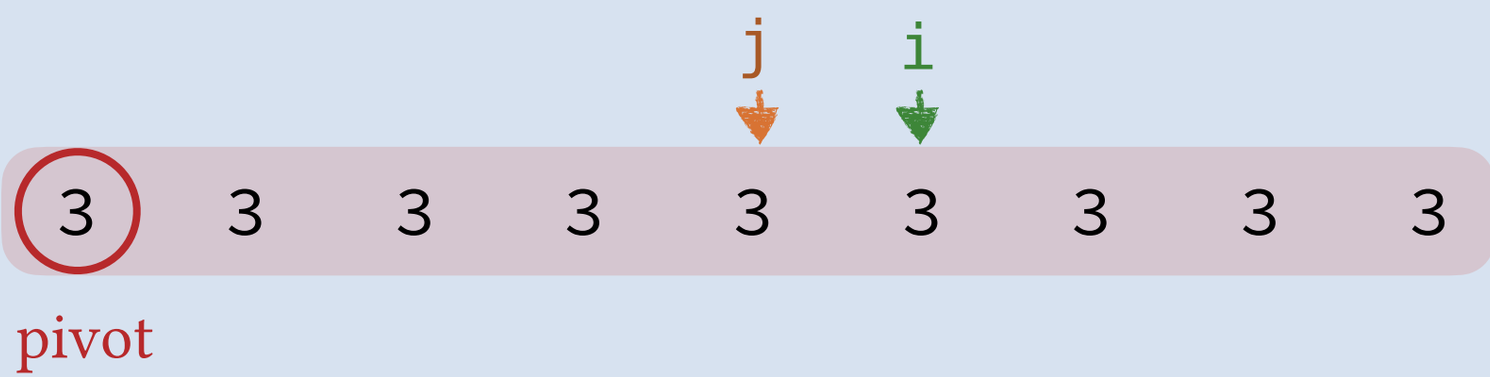
Better. Stop on elements equal to the pivot.
More swaps but the pivot becomes the middle element!



Optimization Quiz

Bad. Do not stop on elements equal to the pivot.
Having many duplicate keys leads to worst case performance!

Better. Stop on elements equal to the pivot.
More swaps but the pivot becomes the middle element!



Optimization Quiz

Bad. Do not stop on elements equal to the pivot.
Having many duplicate keys leads to worst case performance!

Better. Stop on elements equal to the pivot.
More swaps but the pivot becomes the middle element!

War story (system sort in C)

A beautiful bug report. [Allan Wilks and Rick Becker, 1991]

```
We found that qsort is unbearably slow on "organ-pipe" inputs like "01233210":

main (int argc, char**argv) {
   int n = atoi(argv[1]), i, x[100000];
   for (i = 0; i < n; i++)
      x[i] = i;
   for ( ; i < 2*n; i++)
      x[i] = 2*n-i-1;
   qsort(x, 2*n, sizeof(int), intcmp);
}

Here are the timings on our machine:
$ time a.out 2000
real    5.85s
$ time a.out 4000
real   21.64s
$time a.out 8000
real   85.11s
```

War story (system sort in C)

Bug. A `qsort()` call that should have taken seconds was taking minutes.

Using *median of three*
for inputs like:
`[0,1,2,3,4,5,4,3,2,1,0]`

At the time, almost all `qsort()` implementations based on those in:

- Version 7 Unix (1979): quadratic time to sort organ-pipe arrays.
- BSD Unix (1983): quadratic time to sort random arrays of 0s and 1s.

Not stopping
on duplicates!

By Kevin Wayne

Nuts & Bolts

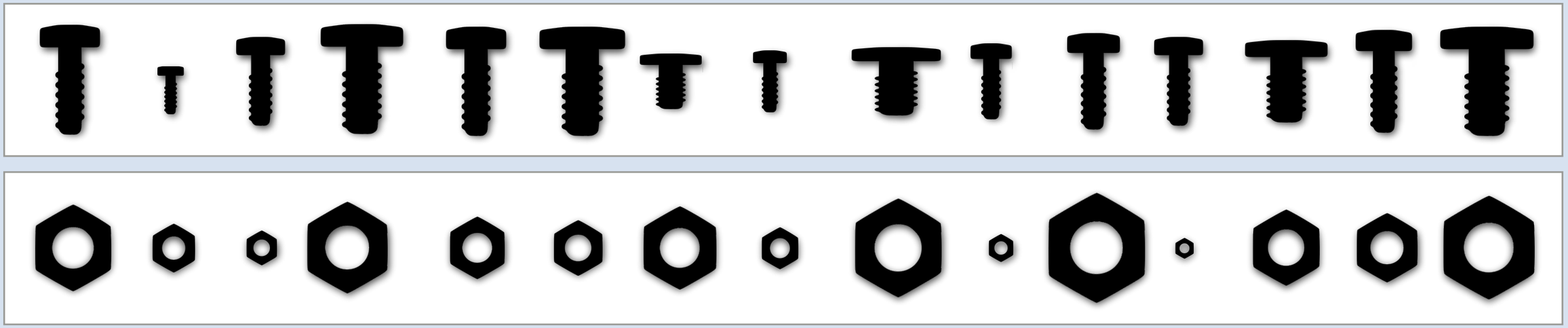
Given n nuts and n bolts of different sizes. Find the matching nut for every bolt.

Constraints:

- Each bolt matches only one nut.
- Comparisons between nuts and bolts are allowed.
- Comparisons between nuts and nuts (or bolts and bolts) are not allowed.

Nuts & Bolts

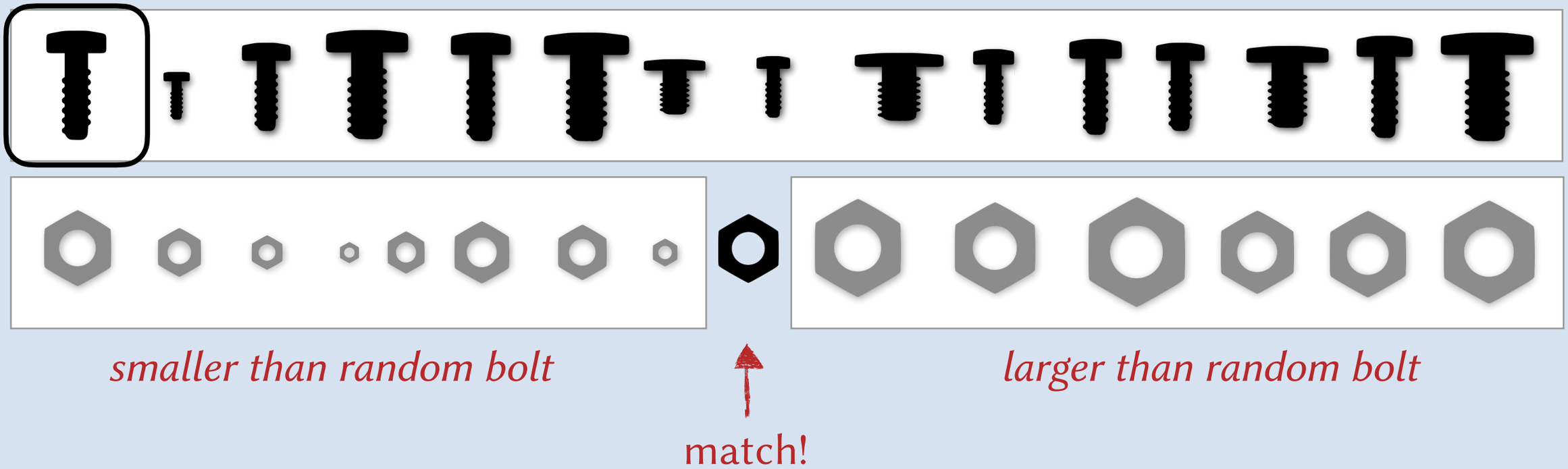
A Quicksort-like solution!



Nuts & Bolts

A Quicksort-like solution!

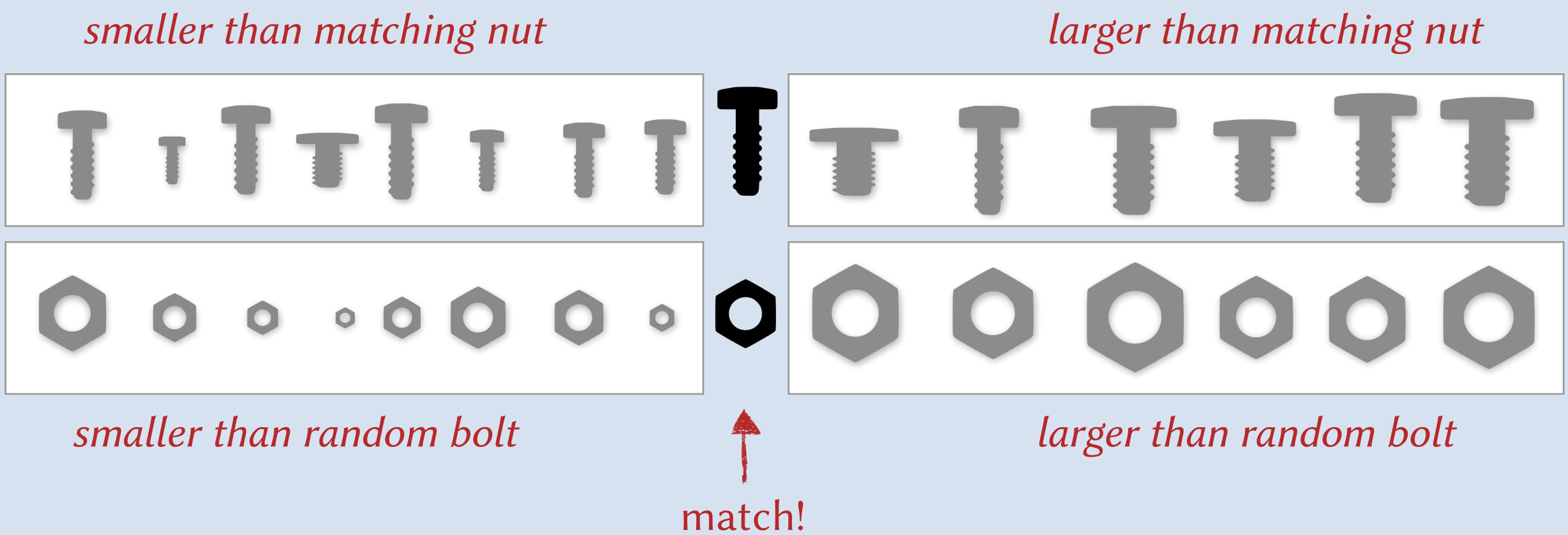
- Pick a *random* bolt.
- *Partition the nuts* based on this bolt. The partitioning step will find the matching nut.



Nuts & Bolts

A Quicksort-like solution!

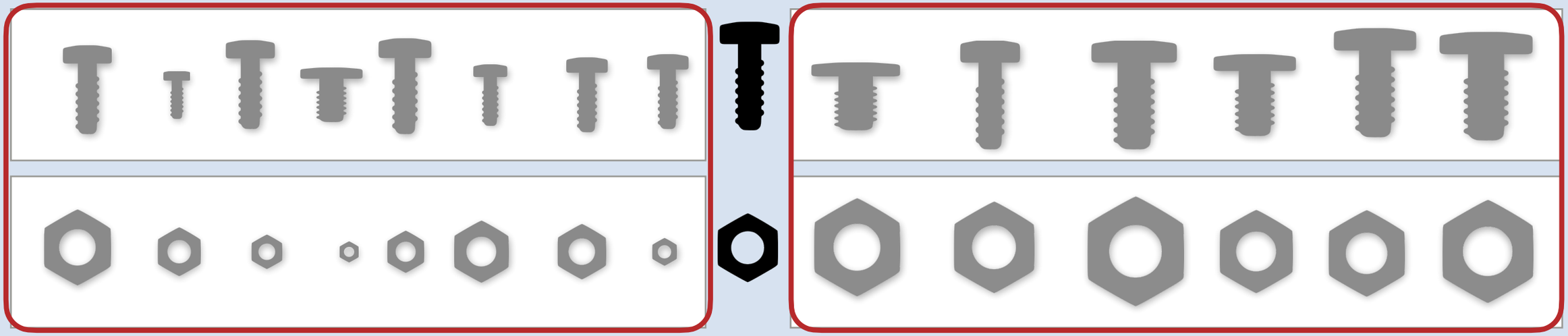
- Pick a *random* bolt.
- *Partition the nuts* based on this bolt. The partitioning step will find the matching nut.
- Use the matching nut to *partition the bolts*.



Nuts & Bolts

A Quicksort-like solution!

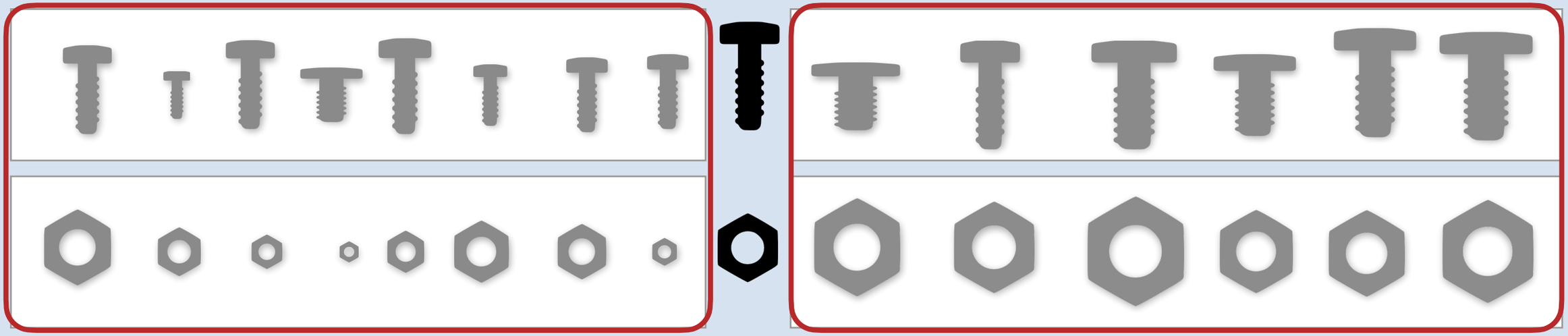
- Pick a *random* bolt.
- *Partition the nuts* based on this bolt. The partitioning step will find the matching nut.
- Use the matching nut to *partition the bolts*.
- *Recursively* do the same for the left and right partitions.



Nuts & Bolts

A Quicksort-like solution!

- Pick a *random* bolt.
- *Partition the nuts* based on this bolt. The partitioning step will find the matching nut.
- Use the matching nut to *partition the bolts*.
- *Recursively* do the same for the left and right partitions.



$\Theta(n \log n)$ Expected time!

These slides are partly based on: https://www.cs.princeton.edu/courses/archive/fall21/cos226/lectures/23Quicksort.pdf

https://upload.wikimedia.org/wikipedia/commons/2/2c/Sir_Tony_Hoare_IMG_5125.jpg