

# LECTURE 18

## COMPARISON SORTS

MCS 275 Spring 2021

Emily Dumas

# LECTURE 18: COMPARISON SORTS

Course bulletins:

- Quiz 6 due Noon CST Tuesday.
- Project 2 due 6pm CST Friday. Autograder open.
- Worksheet 7 coming soon.

# GROWTH RATES

Let's look at the functions  $n$ ,  $n \log(n)$ , and  $n^2$  as  $n$  grows.

# PARTITION

We say  $\mathbb{L}$  is partitioned if there is an element (the **pivot**) so that:

- The pivot is in its final sorted position
- Any element less than the pivot appears before it
- Any element greater than or equal to the pivot appears after it

`partition(L, start, end)` should move around elements of  $\mathbb{L}$  between indices `start` and `end` to achieve this, returning the pivot position.

# Algorithm quicksort:

**Input:** list  $L$  and indices  $start$  and  $end$ .

**Goal:** reorder elements of  $L$  so that  $L[start:end]$  is sorted.

1. If  $(end - start)$  is less than or equal to 1, return immediately.
2. Otherwise, call `partition( $L, start, end$ )` to partition the list, letting  $m$  be the final location of the pivot.
3. Call `quicksort( $L, start, m$ )` and `quicksort( $L, m+1, end$ )` to sort the parts of the list on either side of the pivot.

# PARTITION ALGORITHM

We will always use  $L[\text{end}-1]$  as the pivot (though there are other common choices).

There is an algorithm for `partition` based on the idea of using swaps to move all small elements to a contiguous block at the beginning.

It makes a single pass through the entire list.

# PARTITION VISUALIZATION

76235814

# PARTITION VISUALIZATION

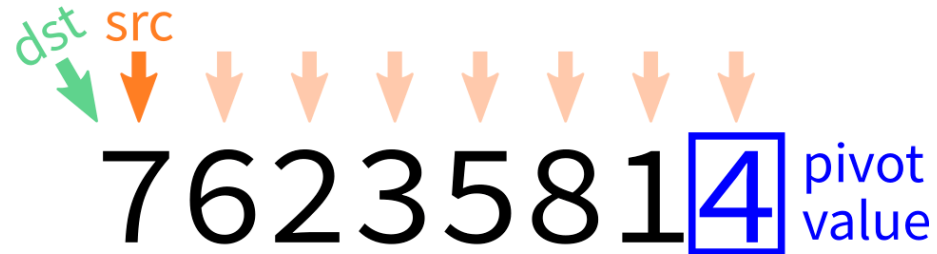
7 6 2 3 5 8 1 4 pivot  
value



# PARTITION VISUALIZATION

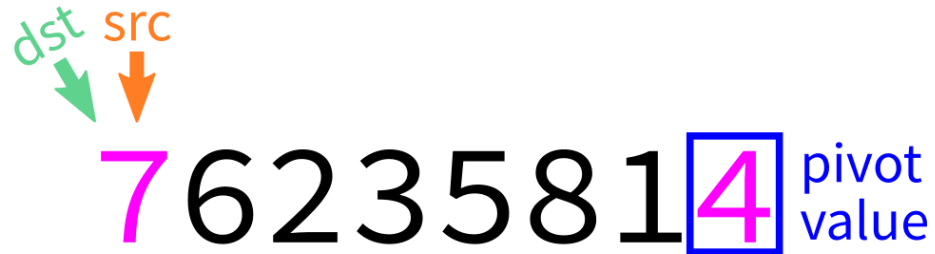
dst src  
7 6 2 3 5 8 1 **4** pivot  
value

# PARTITION VISUALIZATION



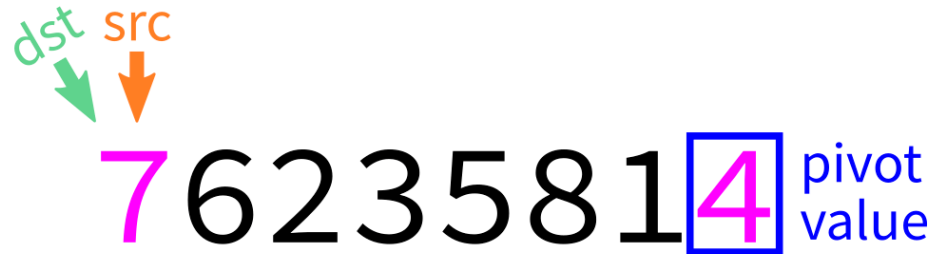
# PARTITION VISUALIZATION

dst src  
7 6 2 3 5 8 1 4 pivot  
value



The diagram illustrates a partitioning step in a sorting algorithm. It shows an array of numbers: 7, 6, 2, 3, 5, 8, 1, 4. The number 4 is highlighted with a blue box and labeled 'pivot value'. A green arrow labeled 'dst' points to the first element (7), and an orange arrow labeled 'src' points to the pivot element (4).

# PARTITION VISUALIZATION



$L[\text{src}] \geq \text{pivot}$ : do nothing

# PARTITION VISUALIZATION

dst src  
↓ ↓  
7 6 2 3 5 8 1 **4** pivot  
value

# PARTITION VISUALIZATION

dst src  
↓ ↓  
7 6 2 3 5 8 1 4 pivot  
value

# PARTITION VISUALIZATION

dst src  
↓ ↓  
7 6 2 3 5 8 1 4 pivot  
value

$L[\text{src}] \geq \text{pivot}$ : do nothing

# PARTITION VISUALIZATION

dst src  
↓ ↓  
7 6 2 3 5 8 1 **4** pivot  
value



# PARTITION VISUALIZATION

dst src  
↓ ↓  
7 6 2 3 5 8 1 **4** pivot  
value

# PARTITION VISUALIZATION

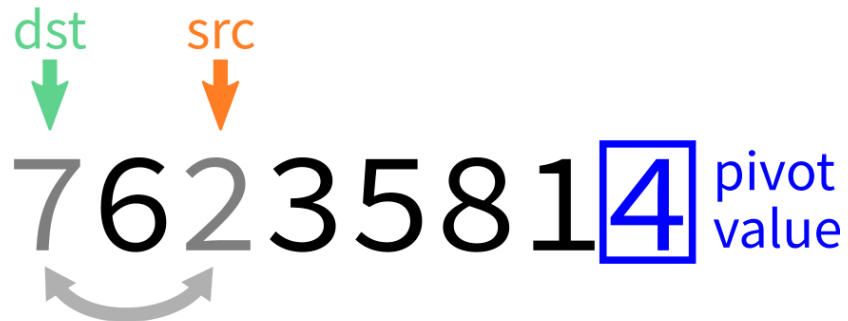
dst src  
↓ ↓  
7 6 2 3 5 8 1 4 pivot  
value

# PARTITION VISUALIZATION

dst src  
↓ ↓  
7 6 2 3 5 8 1 4 pivot  
value

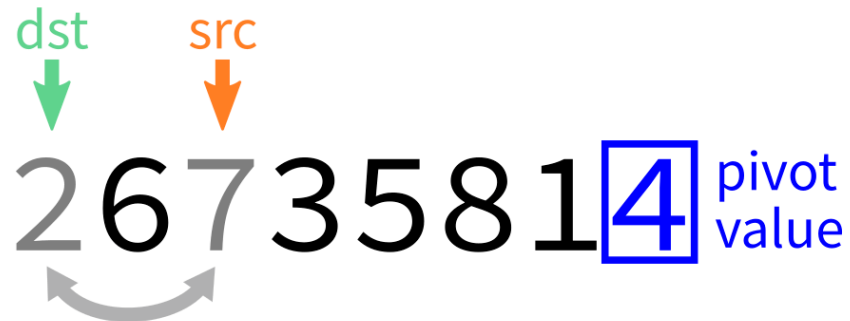
$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$

# PARTITION VISUALIZATION



$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$

# PARTITION VISUALIZATION



$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$

# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 6 7 3 5 8 1 **4** pivot  
value

# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 6 7 3 5 8 1 **4** pivot  
value

# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 6 7 3 5 8 1 4 pivot  
value

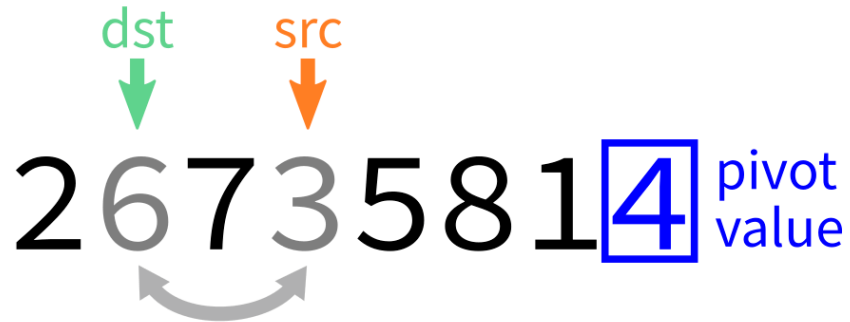


# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 6 7 3 5 8 1 4 pivot  
value

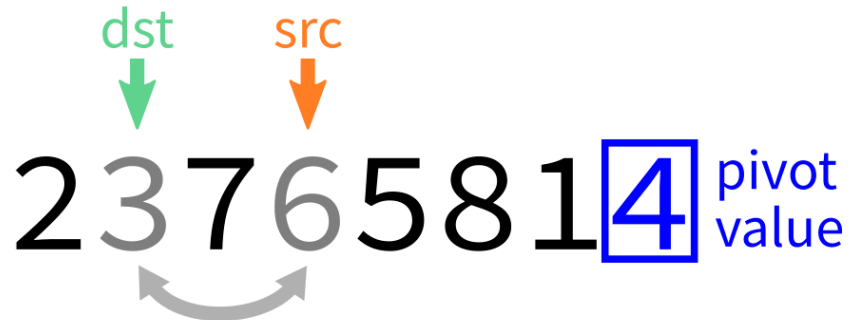
$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$

# PARTITION VISUALIZATION



$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$

# PARTITION VISUALIZATION



$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$

# PARTITION VISUALIZATION

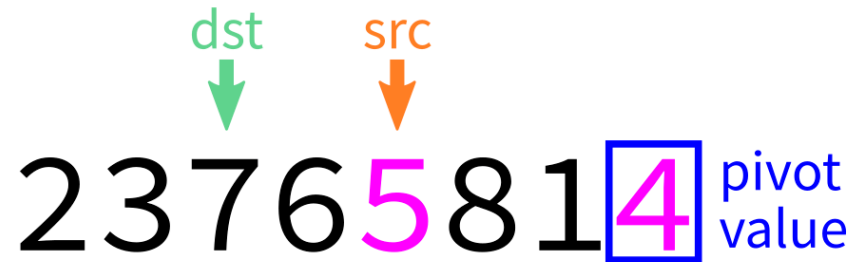
dst src  
↓ ↓  
2 3 7 6 5 8 1 **4** pivot  
value

# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 3 7 6 5 8 1 **4** pivot  
value

# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 3 7 6 5 8 1 4 pivot  
value



The diagram illustrates the partitioning process in a sorting algorithm. It shows an array of numbers: 2, 3, 7, 6, 5, 8, 1, and 4. The number 4 is designated as the 'pivot value' and is enclosed in a blue box. Above the array, two arrows indicate the movement of elements: a green arrow labeled 'dst' points to the number 7, and an orange arrow labeled 'src' points to the number 5. This suggests that elements greater than the pivot are being moved to the destination side, and elements less than the pivot are being moved to the source side.

# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 3 7 6 5 8 1 4 pivot  
value

$L[src] \geq \text{pivot}$ : do nothing

# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 3 7 6 5 8 1 **4** pivot  
value



# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 3 7 6 5 8 1 **4** pivot  
value

# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 3 7 6 5 8 1 4 pivot  
value

The diagram illustrates the partitioning process in a sorting algorithm. It shows an array of numbers: 2, 3, 7, 6, 5, 8, 1, and 4. The number 4 is designated as the pivot value, indicated by a blue box and the text 'pivot value' to its right. A green arrow labeled 'dst' points to the number 7, representing the destination index for elements less than the pivot. An orange arrow labeled 'src' points to the number 8, representing the source index for elements greater than the pivot.

# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 3 7 6 5 8 1 4 pivot  
value

$L[src] \geq \text{pivot}$ : do nothing

# PARTITION VISUALIZATION

dst src  
↓ ↓  
2 3 7 6 5 8 1 **4** pivot  
value

# PARTITION VISUALIZATION

dst  
↓  
23765814 pivot  
src  
↓  
value

# PARTITION VISUALIZATION

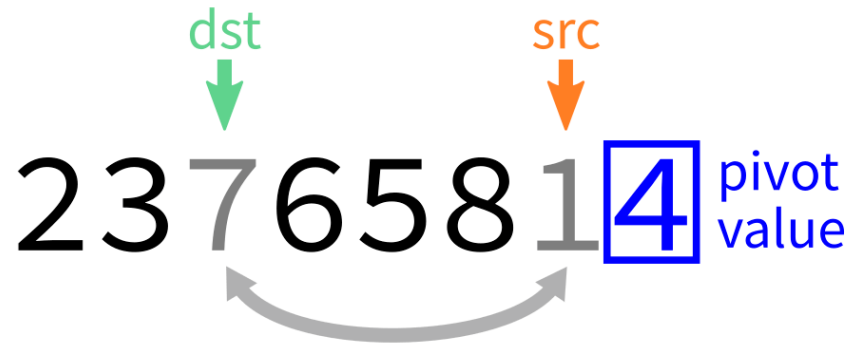
dst  
↓  
23765814 pivot  
src  
↓  
value

# PARTITION VISUALIZATION

dst  
↓  
23765814 pivot  
src  
↓  
value

$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$

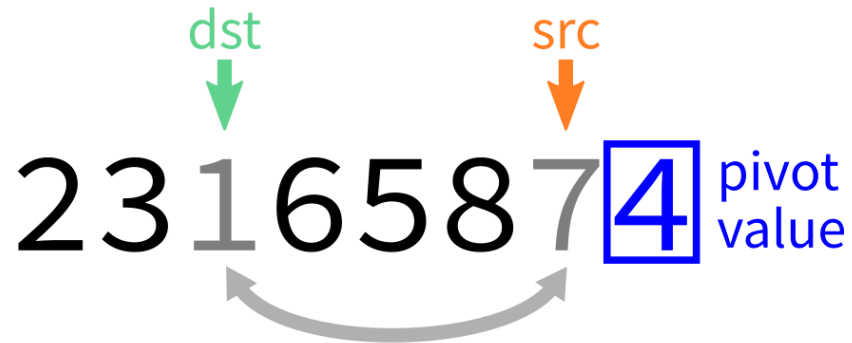
# PARTITION VISUALIZATION



$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$



# PARTITION VISUALIZATION



$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$

# PARTITION VISUALIZATION

dst src  
↓ ↓  
23165874 pivot  
value



The diagram illustrates the partitioning process in a sorting algorithm. It shows an array of numbers: 2, 3, 1, 6, 5, 8, 7, and 4. The last element, 4, is designated as the pivot value, indicated by a blue box and the text 'pivot value'. A green arrow labeled 'dst' points to the first element (2), representing the destination for elements less than the pivot. An orange arrow labeled 'src' points to the pivot element (4), representing the source for elements greater than the pivot.

# PARTITION VISUALIZATION

23165874

dst

src

pivot value

The diagram illustrates the partitioning step of a sorting algorithm. It shows an array of numbers: 2, 3, 1, 6, 5, 8, 7, 4. The last element, 4, is the pivot value, highlighted with a blue box and labeled 'pivot value'. A green arrow labeled 'dst' points to the element 6, indicating its target position. An orange arrow labeled 'src' points to the pivot element 4, indicating its source position.

# PARTITION VISUALIZATION

23165874

dst

src

pivot value



The diagram illustrates the partitioning step of a sorting algorithm. The array is [2, 3, 1, 6, 5, 8, 7, 4]. The pivot value is 4, which is highlighted in a blue box. A green arrow labeled 'dst' points to the element 6, indicating its target position. An orange arrow labeled 'src' points to the pivot element 4, indicating its source position. The text 'pivot value' is written in blue next to the pivot element.

# PARTITION VISUALIZATION

dst  
↓  
23165874 pivot  
value

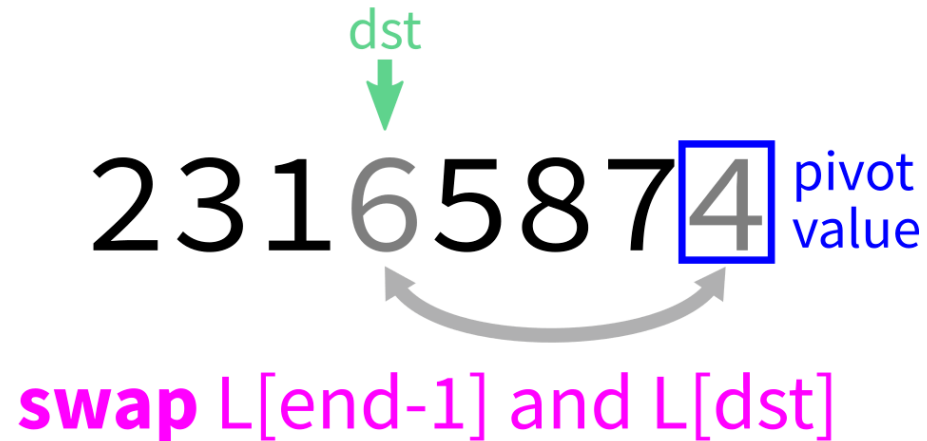


$L[src] \geq \text{pivot}$ : do nothing

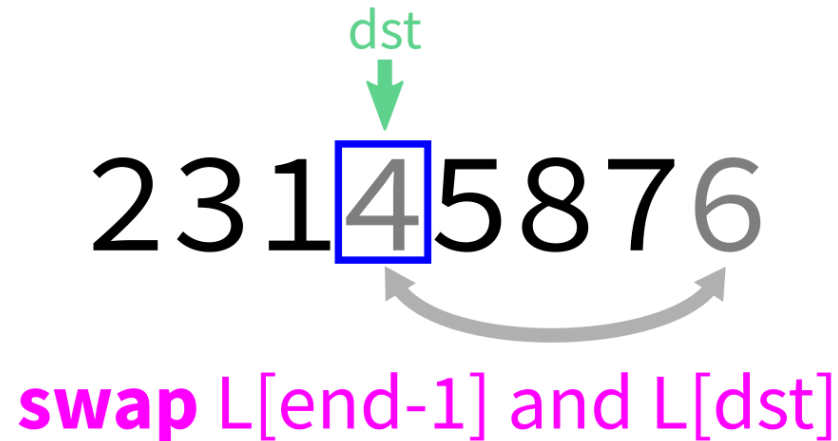
# PARTITION VISUALIZATION

dst  
↓  
23165874 pivot  
value

# PARTITION VISUALIZATION



# PARTITION VISUALIZATION





# PARTITION VISUALIZATION

231**4**5876

# PARTITION VISUALIZATION

23145876



The image shows the array [2, 3, 1, 4, 5, 8, 7, 6] during a partitioning step. The pivot element is 4, which is highlighted with a blue square. An orange bracket is positioned under the elements 2, 3, and 1, representing the left partition. A purple bracket is positioned under the elements 5, 8, 7, and 6, representing the right partition.

# PARTITION VISUALIZATION

231**4**5876



The diagram illustrates the partitioning of an array [2, 3, 1, 4, 5, 8, 7, 6] around a pivot element 4. The pivot is highlighted with a blue square. An orange bracket below the elements 2, 3, and 1 is labeled <4, indicating they are less than the pivot. A purple bracket below the elements 5, 8, 7, and 6 is labeled ≥4, indicating they are greater than or equal to the pivot.

<4      ≥4

# CODING TIME

Let's implement `partition` (with last element pivot) in Python.

# Algorithm `partition`:

**Input:** list `L` and indices `start` and `end`.

**Goal:** Take `L[end-1]` as a pivot, and reorder elements of `L` to partition `L[start:end]` accordingly.

# OTHER PARTITION STRATEGIES

Popular choices for the pivot:

- The last element,  $L[\text{end}-1]$  (used in lecture today)
- The first element,  $L[\text{start}]$
- A random element of  $L[\text{start}:\text{end}]$
- The element  $L[(\text{start}+\text{end}) // 2]$
- An element near the median of  $L[\text{start}:\text{end}]$   
(more complicated to find!)

# HOW TO CHOOSE?

Knowing something about your starting data may guide choice of partition strategy (or even the choice to use something other than quicksort).

Almost-sorted data is a common special case where first or last pivots are bad.

# EFFICIENCY

**Theorem:** If you measure the time cost of quicksort in any of these terms

- Number of comparisons made
- Number of swaps or assignments
- Number of Python statements executed

then the cost to sort a list of length  $n$  is less than  $Cn^2$ , for some constant  $C$ .

But if you average over all possible orders of the input data, the result is less than  $Cn \log(n)$ .



# BAD CASE

What if we ask our version of `quicksort` to sort a list that is already sorted?

Recursion depth is  $n$  (whereas if the pivot is always the median it would be  $\approx \log_2 n$ ).

Number of comparisons  $\approx Cn^2$ . Very slow!

# STABILITY

A sort is called **stable** if items that compare as equal stay in the same relative order after sorting.

This could be important if the items are more complex objects we want to sort by one attribute (e.g. sort alphabetized employee records by hiring year).

As we implemented them:

- Mergesort is stable
- Quicksort is not stable

# EFFICIENCY SUMMARY

Algorithm	Time (worst)	Time (average)	Stable?	Space
Mergesort	$Cn \log(n)$	$Cn \log(n)$	Yes	$Cn$
Quicksort	$Cn^2$	$Cn \log(n)$	No	$C$

(Every time  $C$  is used, it represents a different constant.)

# OTHER COMPARISON SORTS

- **Insertion sort** -- Convert the beginning of the list to a sorted list, starting with one element and growing by one element at a time.
- **Bubble sort** -- Process the list from left to right. Any time two adjacent elements are in the wrong order, switch them. Repeat  $n$  times.

# EFFICIENCY SUMMARY

Algorithm	Time (worst)	Time (average)	Stable?	Space
Mergesort	$Cn \log(n)$	$Cn \log(n)$	Yes	$Cn$
Quicksort	$Cn^2$	$Cn \log(n)$	No	$C$
Insertion	$Cn^2$	$Cn^2$	Yes	$C$
Bubble	$Cn^2$	$Cn^2$	Yes	$C$

(Every time  $C$  is used, it represents a different constant.)

# CLOSING THOUGHTS ON SORTING

Mergesort is rarely a bad choice. It is stable and sorts in  $Cn \log(n)$  time. Nearly sorted input is not a pathological case. Its main weakness is its use of memory proportional to the input size.

**Heapsort**, which we'll discuss later, has  $Cn \log(n)$  running time and uses constant space, but it is not stable.

There are stable comparison sorts with  $Cn \log(n)$  running time and constant space (best in every category!) though they are more complex.

If swaps and comparisons have very different cost, it may be important to select an algorithm that minimizes one of them. Python's `list.sort` assumes that comparisons are expensive, and uses **Timsort**.

# QUADRATIC DANGER

Algorithms that take time proportional to  $n^2$  are a big source of real-world trouble. They are often fast enough in small-scale tests to not be noticed as a problem, yet are slow enough for large inputs to disable the fastest computers.



# REFERENCES

Unchanged from Lecture 17

- You can refer to the [general references about recursion](#) that have appeared in several recent lectures. The rest of this list is specific to mergesort and quicksort.
- Making nice visualizations of sorting algorithms is a cottage industry in CS education. Some you might like to check out:
  - [2D visualization through color sorting](#) by Linus Lee
  - [Animated bar graph visualization of many sorting algorithms](#) by Alex Macy
  - Slanted line animated visualizations of [mergesort](#) and [quicksort](#) by Mike Bostock

# REVISION HISTORY

- 2021-02-22 Fixed partition visualization
- 2021-02-22 Initial publication

