

# LECTURE 14

## QUICKSORT

MCS 275 Spring 2023

Emily Dumas

# LECTURE 14: QUICKSORT

Reminders and announcements:

- Project 2 due 6pm Fri 24 Feb.
- Project 2 autograder opens by Mon 20 Feb.
  - Having at least partial work ready to submit at that time is a good goal.

# TRANSFORMATION VS MUTATION

Last time we wrote a mergesort function that acts as a transformation: A list is given as input, a new sorted list is returned.

Another approach we could consider is sorting as a mutation: A list is provided, the function reorders its items and returns nothing.

# IN PLACE

A sorting transformation always uses an amount of extra memory proportional to the size of the list. (It needs a second list to store the output.)

A sort that operates as a mutation has the possibility of using only a fixed amount of memory to do its work. Doing so is called an **in place** sorting method.

# QUICKSORT

A recursive in place sorting method that, like mergesort, is reasonably efficient and widely used.

# PARTITION

Let's first study something weaker than sorting.

Given a list  $\mathbb{L}$ , let  $p$  be the last element of  $\mathbb{L}$ .

We want to rearrange  $\mathbb{L}$  so that it looks like:

$$[ \text{items} < p, \mathbf{p}, \text{items} \geq p ]$$

We say  $\mathbb{L}$  has been **partitioned** at  $p$ , and we call  $p$  the **pivot**.

# PARTITION ALGORITHM IDEA

Scan through the list, moving things smaller than the pivot to the beginning.

# PARTITION ALGORITHM VISUALIZATION

76235814

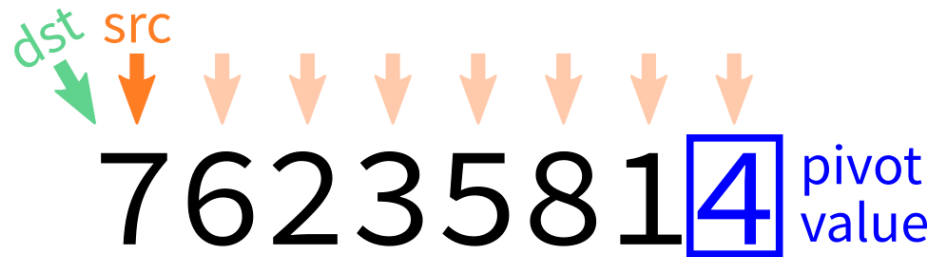
# PARTITION ALGORITHM VISUALIZATION

76235814 pivot  
value

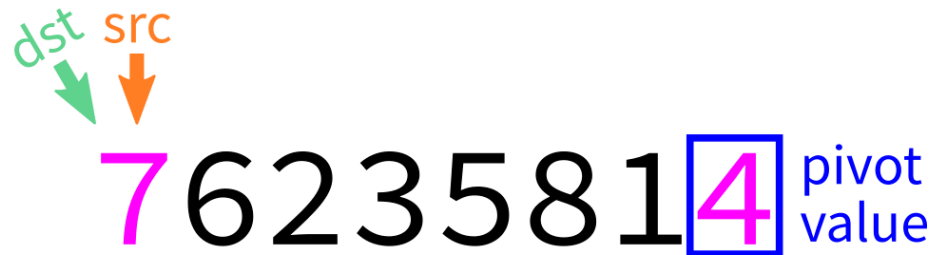
# PARTITION ALGORITHM VISUALIZATION

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

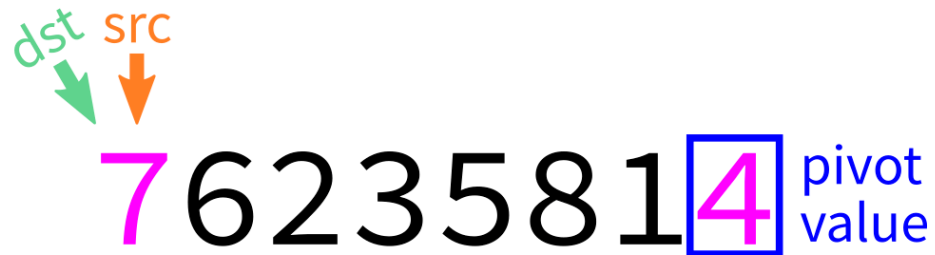
# PARTITION ALGORITHM VISUALIZATION



# PARTITION ALGORITHM VISUALIZATION



# PARTITION ALGORITHM VISUALIZATION



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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

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

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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

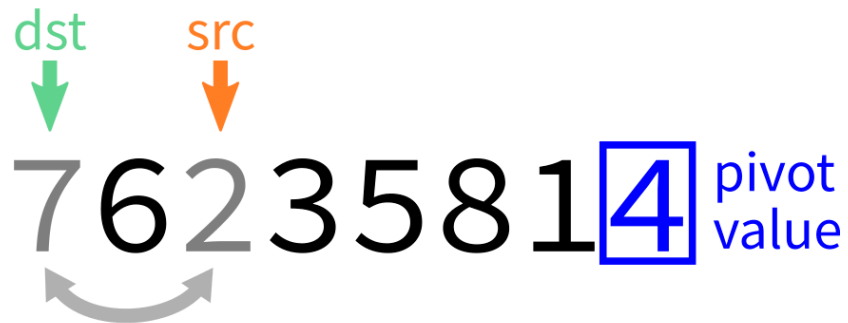
dst src  
↓ ↓  
7 6 2 3 5 8 1 4 pivot  
value

# PARTITION ALGORITHM VISUALIZATION

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

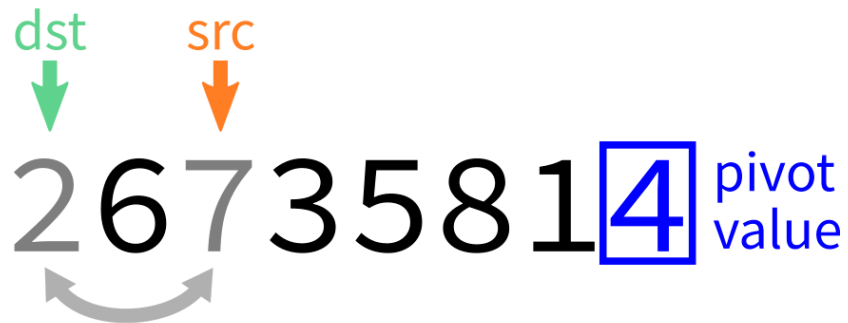
$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$

# PARTITION ALGORITHM VISUALIZATION



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

# PARTITION ALGORITHM VISUALIZATION



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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

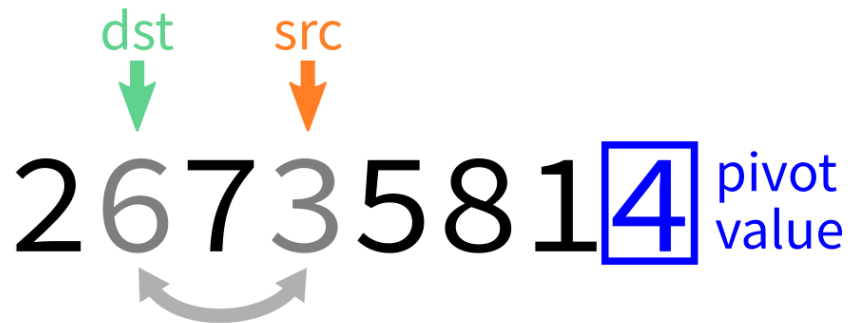
dst src  
↓ ↓  
2 6 7 3 5 8 1 4 pivot  
value

# PARTITION ALGORITHM VISUALIZATION

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

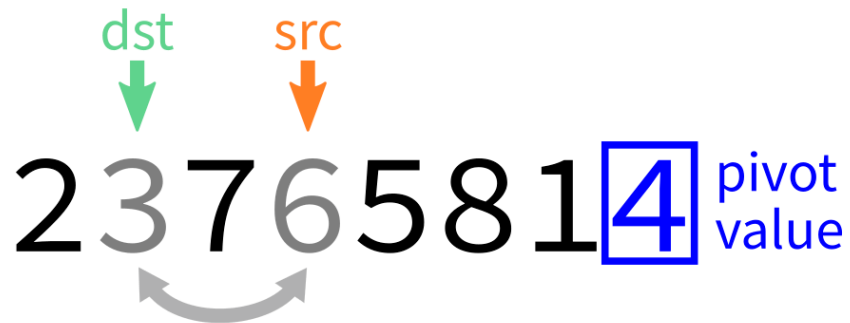
$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$

# PARTITION ALGORITHM VISUALIZATION



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

# PARTITION ALGORITHM VISUALIZATION



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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

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

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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

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

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

# PARTITION ALGORITHM VISUALIZATION

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

# PARTITION ALGORITHM VISUALIZATION

dst src  
↓ ↓  
23765814 pivot  
value

# PARTITION ALGORITHM VISUALIZATION

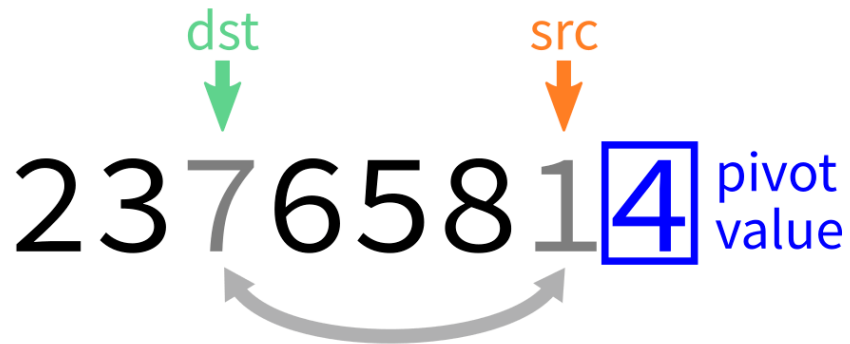
dst src  
↓ ↓  
23765814 pivot  
value

# PARTITION ALGORITHM VISUALIZATION

dst src  
↓ ↓  
23765814 pivot  
value

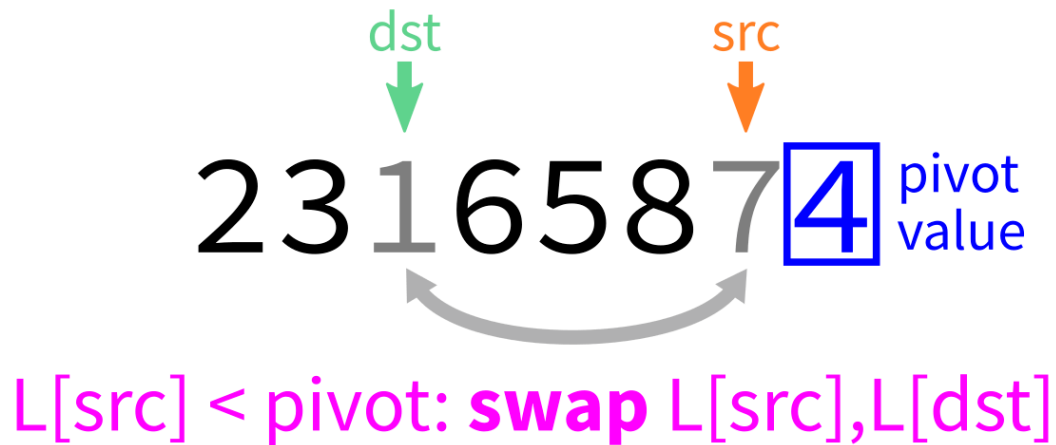
$L[src] < \text{pivot}$ : **swap**  $L[src], L[dst]$

# PARTITION ALGORITHM VISUALIZATION



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

# PARTITION ALGORITHM VISUALIZATION



# PARTITION ALGORITHM VISUALIZATION

dst src  
↓ ↓  
23165874 pivot  
value

# PARTITION ALGORITHM VISUALIZATION

23165874

dst

src

pivot value

# PARTITION ALGORITHM VISUALIZATION

dst src  
↓ ↓  
23165874 pivot  
value

The diagram illustrates a step in the partitioning process of a sorting algorithm. It shows an array of numbers: 2, 3, 1, 6, 5, 8, 7, and 4. The last element, 4, is highlighted with a blue square and labeled 'pivot value' in blue text. Above the array, a green arrow labeled 'dst' points down to the element 6, and an orange arrow labeled 'src' points down to the pivot element 4. The numbers 2, 3, 1, 6, 5, 8, and 7 are in black, while the pivot 4 is in pink.

# PARTITION ALGORITHM VISUALIZATION

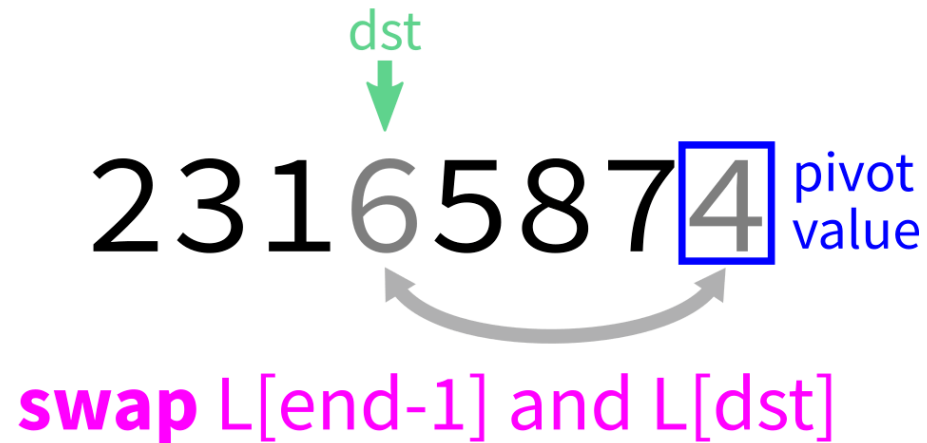
dst  
↓  
23165874 pivot  
src ↓ value

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

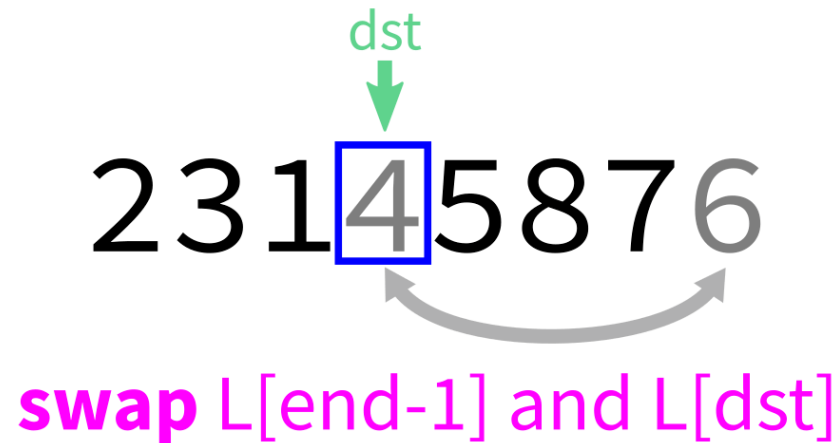
# PARTITION ALGORITHM VISUALIZATION

dst  
↓  
23165874 pivot  
value

# PARTITION ALGORITHM VISUALIZATION



# PARTITION ALGORITHM VISUALIZATION



# PARTITION ALGORITHM VISUALIZATION

23145876

# PARTITION ALGORITHM VISUALIZATION

23145876



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

# PARTITION ALGORITHM VISUALIZATION

23145876



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 under the elements [2, 3, 1] is labeled <4, indicating they are less than the pivot. A purple bracket under the elements [5, 8, 7, 6] is labeled ≥4, indicating they are greater than or equal to the pivot.

<4

≥4

# AFTER PARTITION

The two chunks of the list on either side of the pivot may not be sorted.

But we could bring each of them closer to being sorted by partitioning them...

# QUICKSORT SUMMARY

Starting with an unsorted list:

- If the list has 0 or 1 elements, return immediately.
- Otherwise, partition the list.
- Quicksort the part of the list before the pivot.
- Quicksort the part of the list after the pivot.

It's divide and conquer, but with no merge step. The hard work is instead in partitioning.

# QUICKSORT VISUALIZATION

76235814

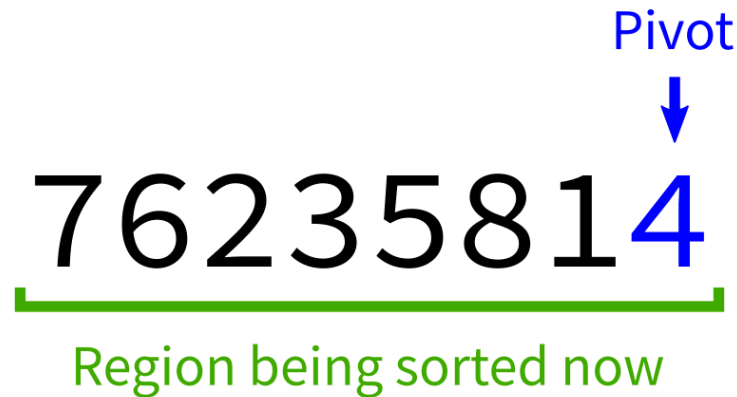
# QUICKSORT VISUALIZATION

76235814

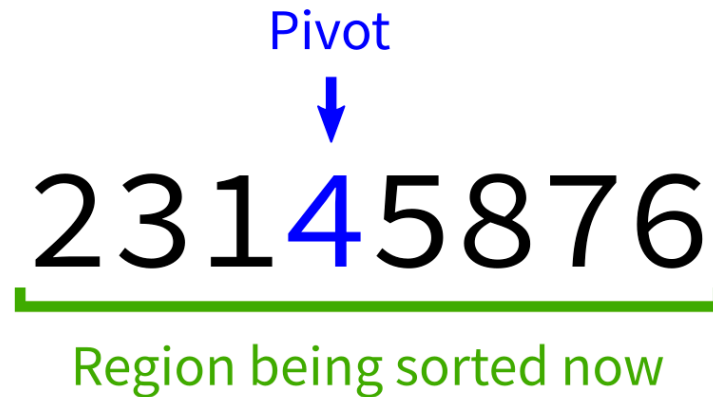


Region being sorted now

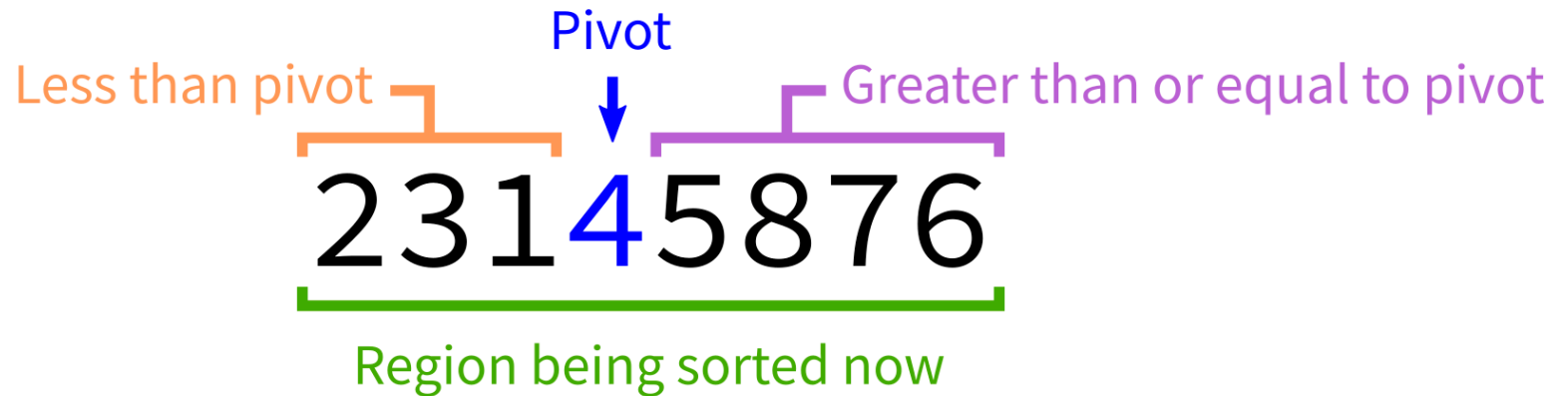
# QUICKSORT VISUALIZATION



# QUICKSORT VISUALIZATION



# QUICKSORT VISUALIZATION



# QUICKSORT VISUALIZATION

23145876

Region being sorted now

# QUICKSORT VISUALIZATION

23145876



# QUICKSORT VISUALIZATION

23145876

# QUICKSORT VISUALIZATION

13245876



# QUICKSORT VISUALIZATION

13245876



# QUICKSORT VISUALIZATION

13245876



# QUICKSORT VISUALIZATION

13245876



A visualization of the Quicksort algorithm. The array [1, 3, 2, 4, 5, 8, 7, 6] is shown. The pivot element is 2, highlighted in blue. A green bracket is under the pivot. Elements less than 2 (1) are to its left, and elements greater than 2 (3, 4, 5, 8, 7, 6) are to its right.

# QUICKSORT VISUALIZATION

12345876



A horizontal sequence of numbers: 1, 2, 3, 4, 5, 8, 7, 6. The number 2 is highlighted in blue. A green bracket is positioned below the numbers 2 and 3, indicating a subarray being processed.

# QUICKSORT VISUALIZATION

12345876



# QUICKSORT VISUALIZATION

12345876



# QUICKSORT VISUALIZATION

12345876



The image shows a sequence of numbers: 1, 2, 3, 4, 5, 8, 7, 6. The number 3 is highlighted in blue. A green bracket is positioned below the number 3, extending from the space between 2 and 3 to the space between 3 and 4, indicating the range of elements being partitioned around the pivot 3.

# QUICKSORT VISUALIZATION

12345876



# QUICKSORT VISUALIZATION

12345876



# QUICKSORT VISUALIZATION

12345876



# QUICKSORT VISUALIZATION

12345678

# QUICKSORT VISUALIZATION

12345678



# QUICKSORT VISUALIZATION

12345678



# QUICKSORT VISUALIZATION

12345678



# QUICKSORT VISUALIZATION

12345678

# QUICKSORT VISUALIZATION

12345678



The image shows a sequence of numbers 1 through 8. The numbers 1 through 6 are in a light gray color, while the number 7 is in black and the number 8 is in blue. A green horizontal line is positioned below the number 7, extending from its left edge to the right edge of the number 8. This visualizes the partitioning step of the Quicksort algorithm, where 7 is the pivot and the range [7, 8] is being processed.

# QUICKSORT VISUALIZATION

12345678

# QUICKSORT VISUALIZATION

12345678



# QUICKSORT VISUALIZATION

12345678



# QUICKSORT VISUALIZATION

12345678

# QUICKSORT VISUALIZATION

12345678

# CODING TIME

Let's implement `quicksort` in Python.

# 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)` 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.

# Algorithm `partition`:

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

**Goal:** Take  $L[\text{end}-1]$  as a pivot, and reorder elements of  $L$  to partition  $L[\text{start}:\text{end}]$  accordingly.

# WHY DISCUSS ALGORITHMS?

Python lists have built-in `.sort()` method. Why talk about sorting?

1. Study cases of easy-to-explain problems solved in clever ways.
2. See patterns of thinking that work in other settings.

# EVALUATING SORTS

Last time we discussed and implemented mergesort, developed by von Neumann (1945) and Goldstine (1947).

Today we discussed quicksort, first described by Hoare (1959) and the simpler partitioning scheme introduced by Lomuto.

**But are these actually good ways to sort a list?**

# EFFICIENCY

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

- Number of comparisons made
- Number of assignments (e.g.  $L[i] = x$  counts as 1)
- Number of Python statements executed

then the cost to sort a list of length  $n$  is less than  $Cn \log(n)$ , for some constant  $C$  that only depends on which expense measure you chose.

# ASYMPTOTICALLY OPTIMAL

$Cn \log(n)$  is pretty efficient for an operation that needs to look at all  $n$  elements. It's not linear in  $n$ , but it only grows a little faster than linear functions.

Furthermore,  $Cn \log(n)$  is the best possible time for comparison sort of  $n$  elements (though different methods might have better  $C$ ).

# QUICKSORT

Is quicksort similarly efficient?

# REFERENCES

- Recursion references from [Lecture 10](#).
- Sorting visualizations:
  - [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

- 2022-02-18 Last year's lecture on this topic finalized
- 2023-02-15 Updated for 2023

