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 `L`, let `p` be the last element of `L`.

We want to rearrange `L` so that it looks like:

`[ items < p, p, items ≥ p ]`

We say `L` has been **partitioned** at `p`, and we call `p` the **pivot**.

PARTITION ALGORITHM IDEA

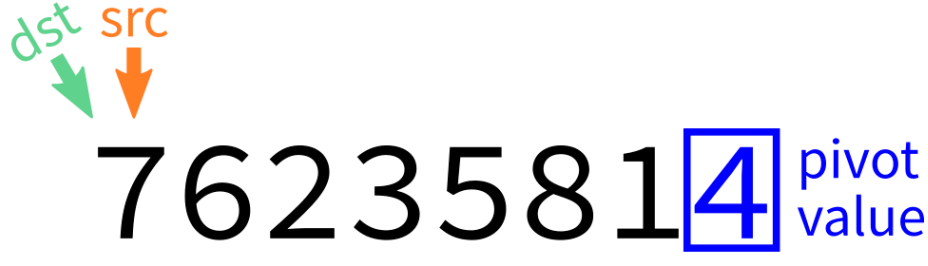
Scan through the list, moving things smaller than the pivot to the beginning.

PARTITION ALGORITHM VISUALIZATION

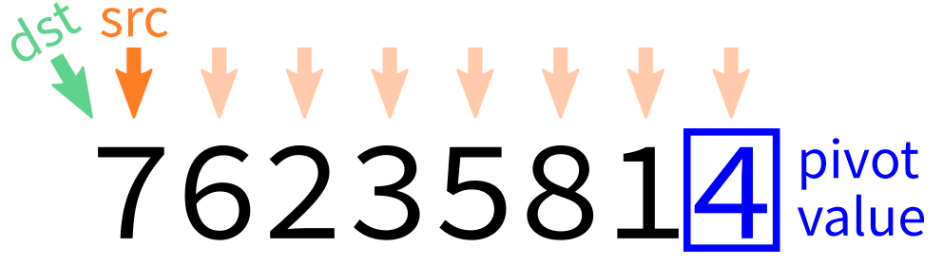
76235814

PARTITION ALGORITHM VISUALIZATION

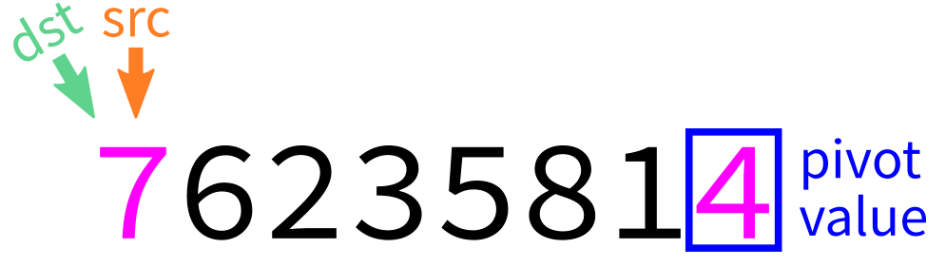
PARTITION ALGORITHM VISUALIZATION



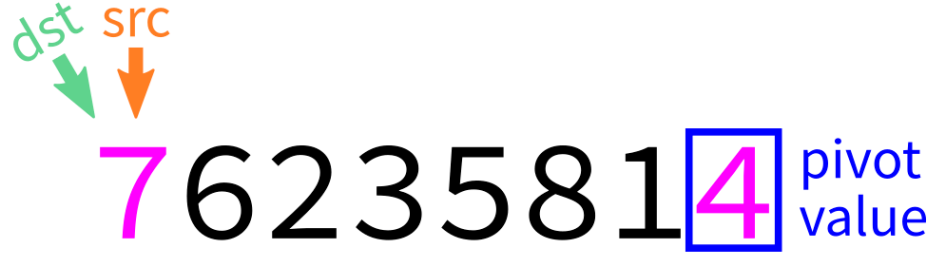
PARTITION ALGORITHM VISUALIZATION



PARTITION ALGORITHM VISUALIZATION

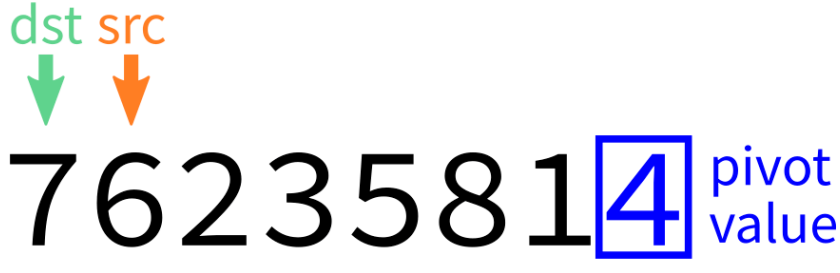


PARTITION ALGORITHM VISUALIZATION

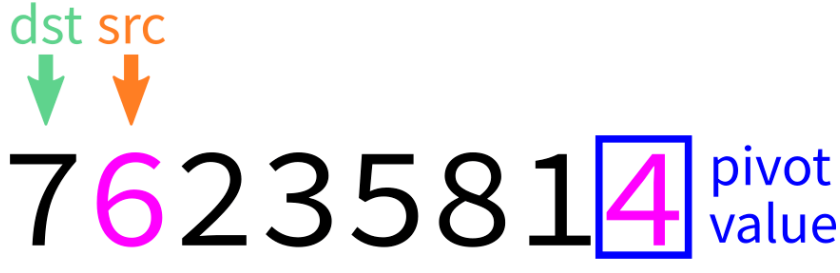


L[src] ≥ pivot: do nothing

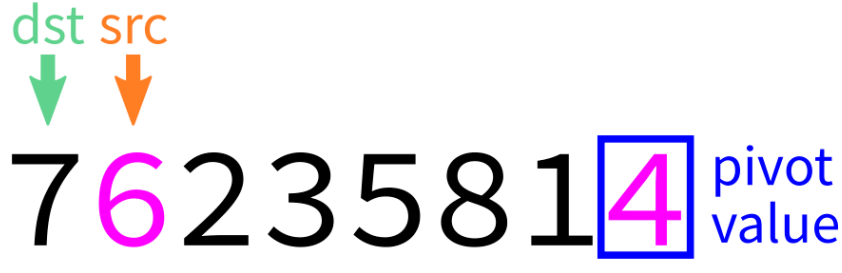
PARTITION ALGORITHM VISUALIZATION



PARTITION ALGORITHM VISUALIZATION

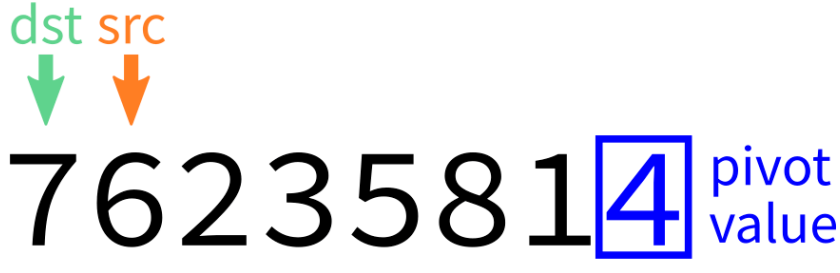


PARTITION ALGORITHM VISUALIZATION

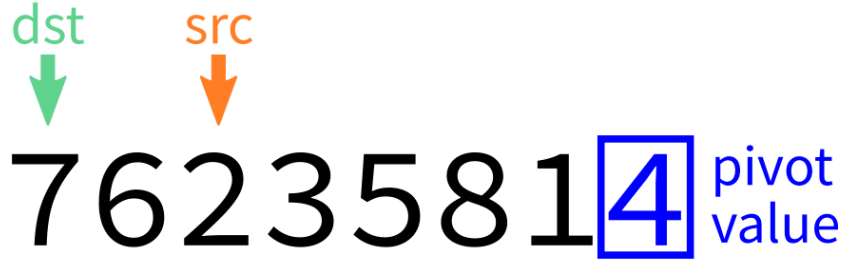


L[src] ≥ pivot: do nothing

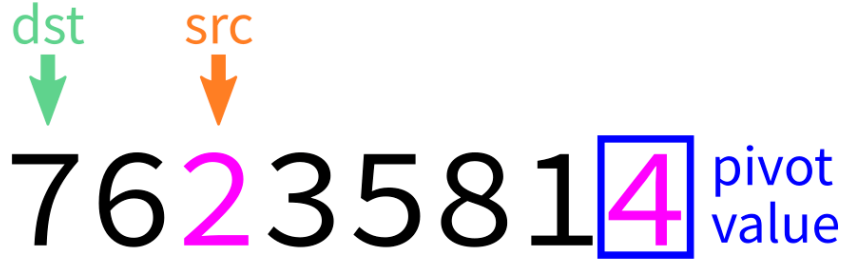
PARTITION ALGORITHM VISUALIZATION



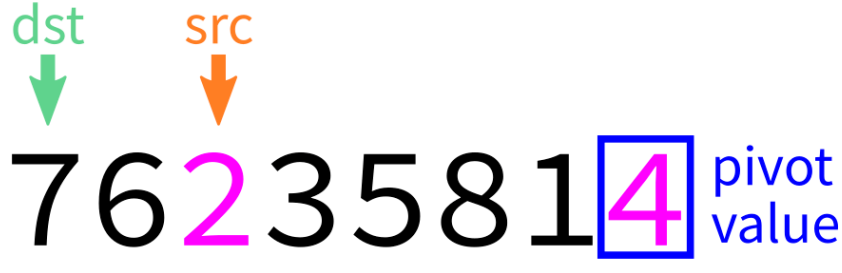
PARTITION ALGORITHM VISUALIZATION



PARTITION ALGORITHM VISUALIZATION

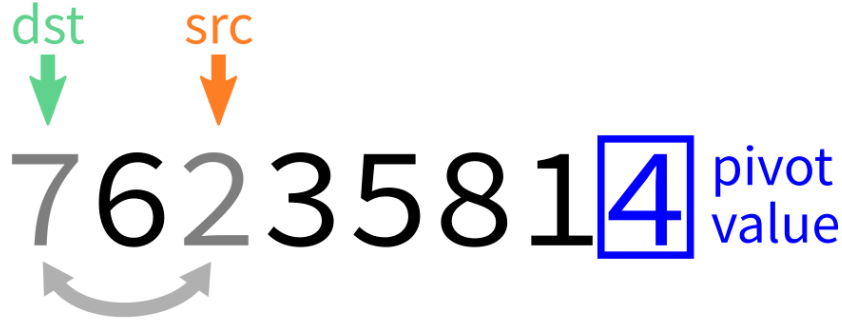


PARTITION ALGORITHM VISUALIZATION



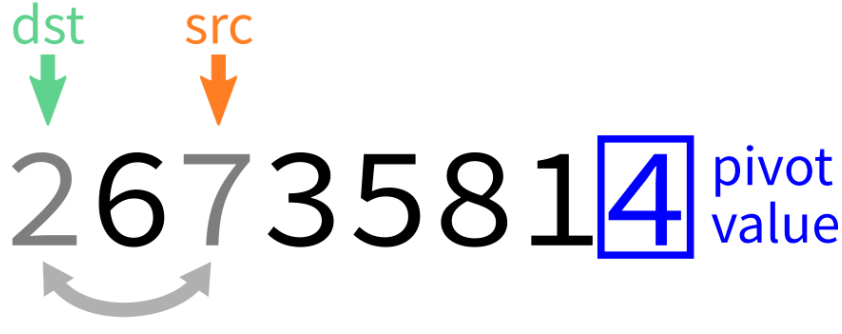
L[src] < pivot: **swap** L[src],L[dst]

PARTITION ALGORITHM VISUALIZATION



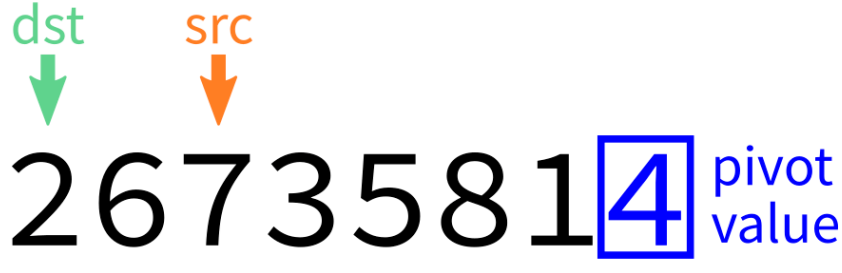
L[src] < pivot: **swap** L[src],L[dst]

PARTITION ALGORITHM VISUALIZATION



L[src] < pivot: **swap** L[src],L[dst]

PARTITION ALGORITHM VISUALIZATION



PARTITION ALGORITHM VISUALIZATION

PARTITION ALGORITHM VISUALIZATION

PARTITION ALGORITHM VISUALIZATION

L[src] < pivot: **swap** L[src],L[dst]

PARTITION ALGORITHM VISUALIZATION

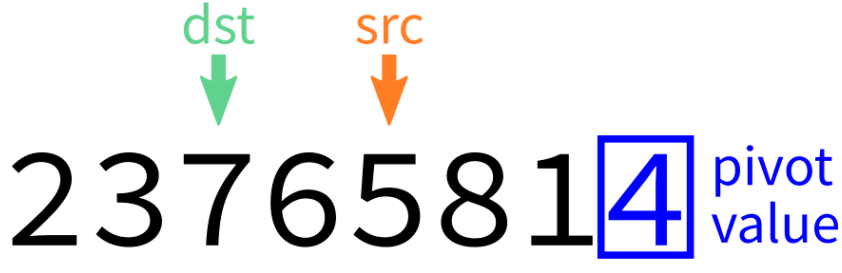
L[src] < pivot: **swap** L[src],L[dst]

PARTITION ALGORITHM VISUALIZATION

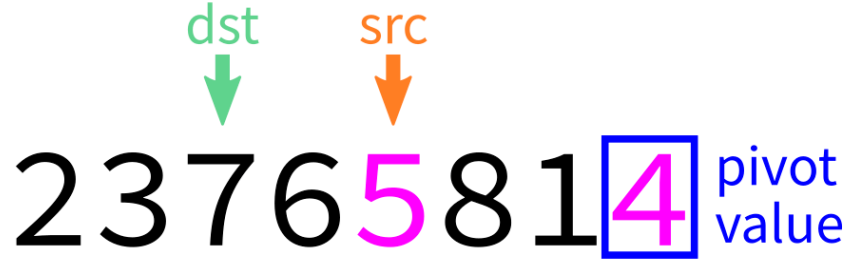
L[src] < pivot: **swap** L[src],L[dst]

PARTITION ALGORITHM VISUALIZATION

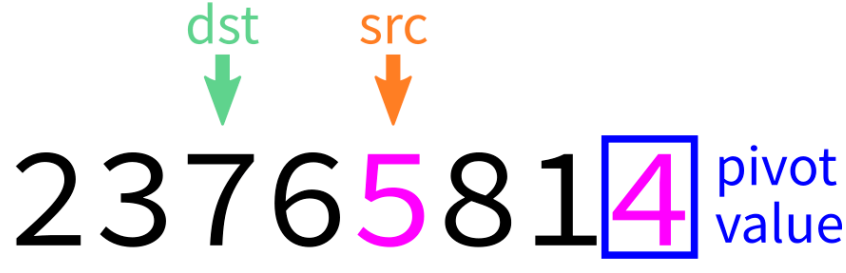
PARTITION ALGORITHM VISUALIZATION



PARTITION ALGORITHM VISUALIZATION

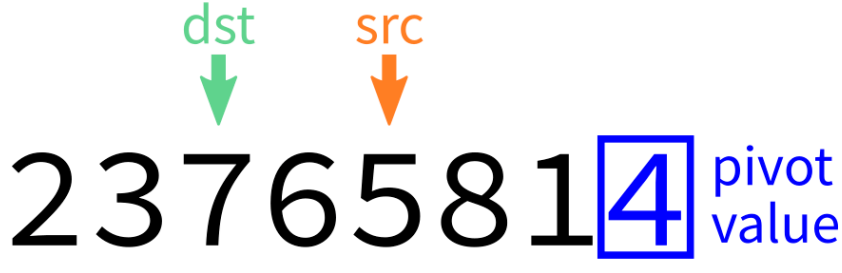


PARTITION ALGORITHM VISUALIZATION

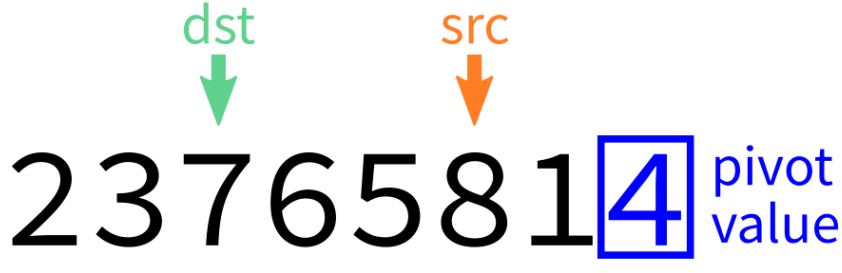


L[src] ≥ pivot: do nothing

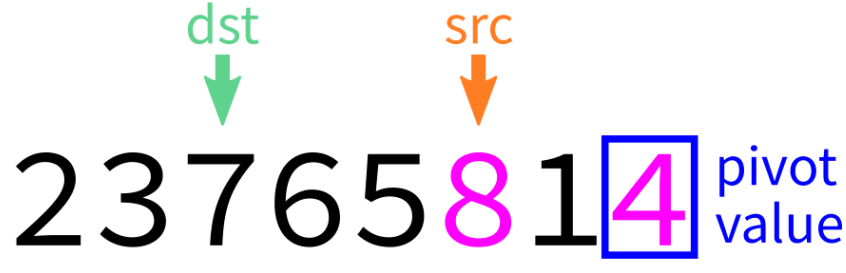
PARTITION ALGORITHM VISUALIZATION



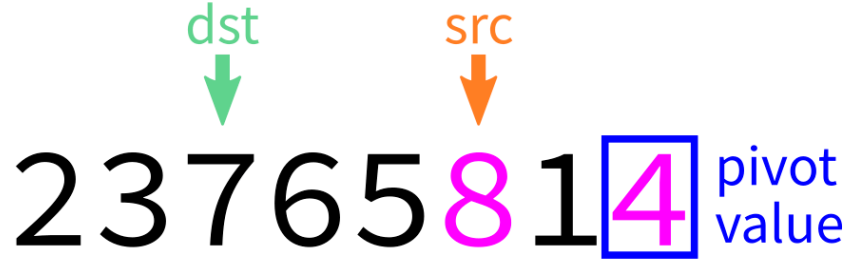
PARTITION ALGORITHM VISUALIZATION



PARTITION ALGORITHM VISUALIZATION

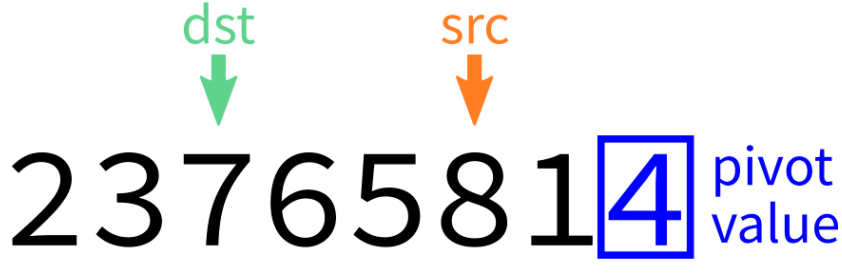


PARTITION ALGORITHM VISUALIZATION



L[src] ≥ pivot: do nothing

PARTITION ALGORITHM VISUALIZATION



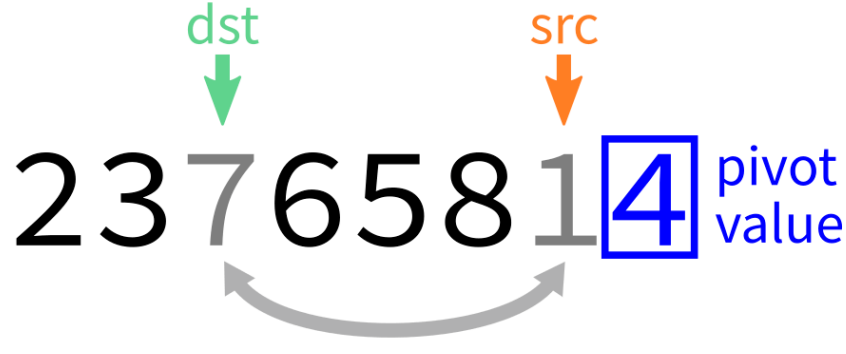
PARTITION ALGORITHM VISUALIZATION

PARTITION ALGORITHM VISUALIZATION

PARTITION ALGORITHM VISUALIZATION

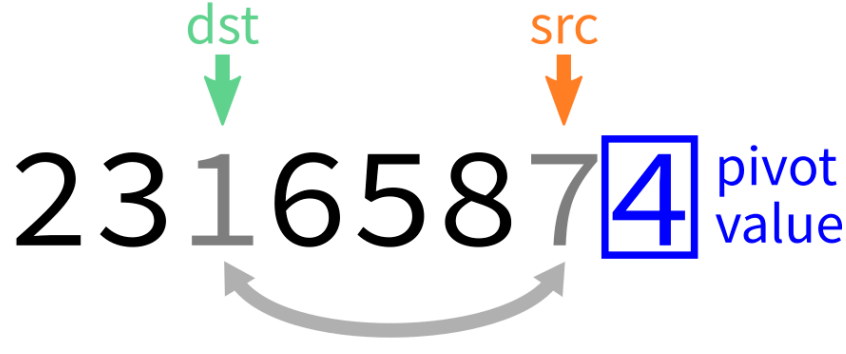
L[src] < pivot: **swap** L[src],L[dst]

PARTITION ALGORITHM VISUALIZATION



L[src] < pivot: **swap** L[src],L[dst]

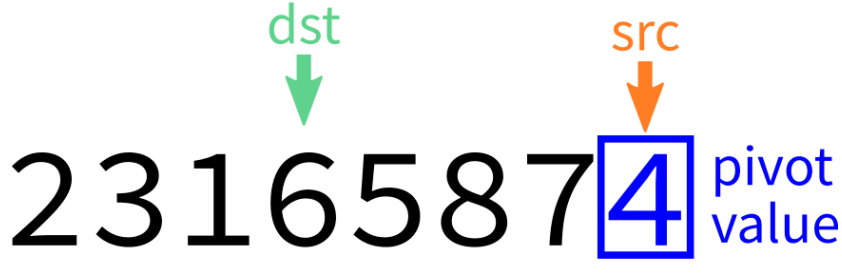
PARTITION ALGORITHM VISUALIZATION



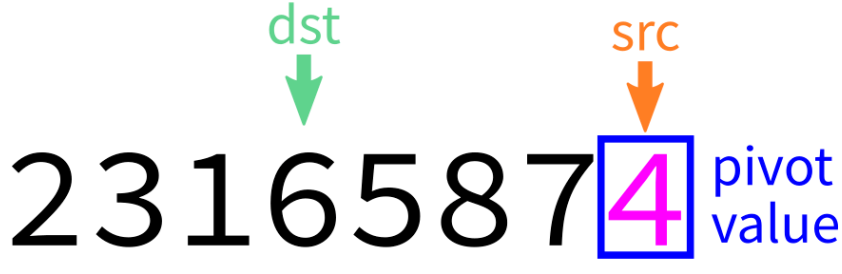
L[src] < pivot: **swap** L[src],L[dst]

PARTITION ALGORITHM VISUALIZATION

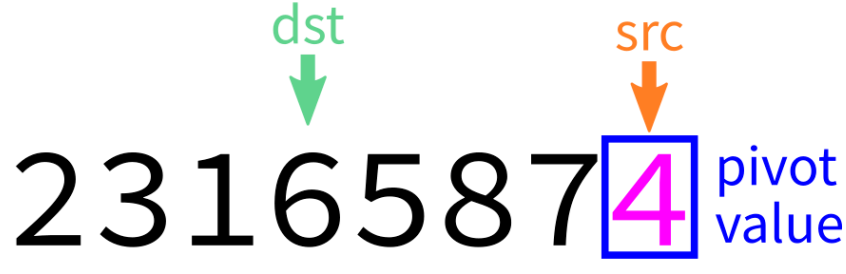
PARTITION ALGORITHM VISUALIZATION



PARTITION ALGORITHM VISUALIZATION

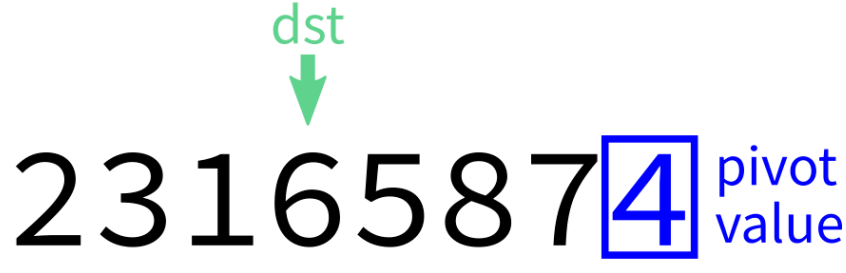


PARTITION ALGORITHM VISUALIZATION

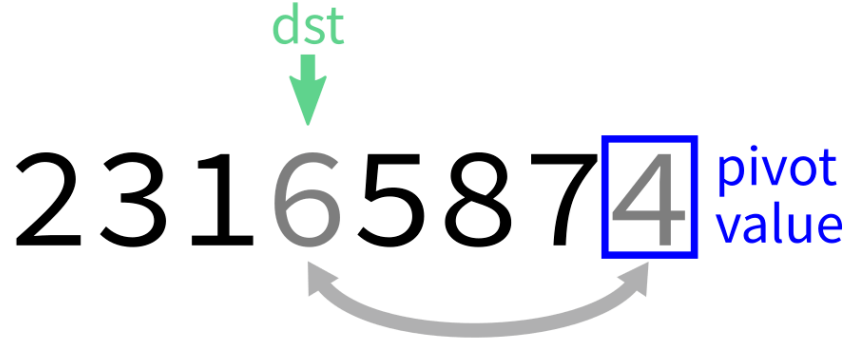


L[src] ≥ pivot: do nothing

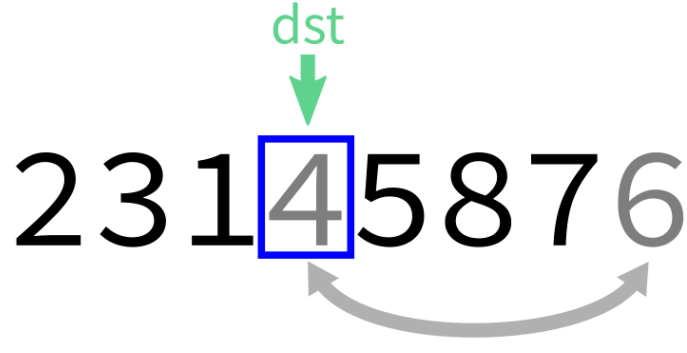
PARTITION ALGORITHM VISUALIZATION



PARTITION ALGORITHM VISUALIZATION

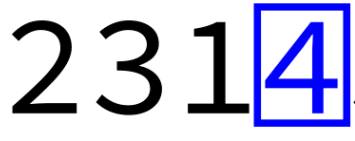


PARTITION ALGORITHM VISUALIZATION



swap L[end-1] and L[dst]

PARTITION ALGORITHM VISUALIZATION



PARTITION ALGORITHM VISUALIZATION

PARTITION ALGORITHM VISUALIZATION

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.

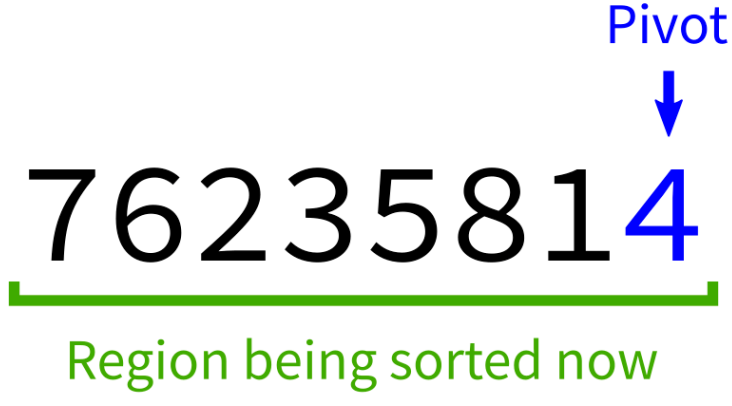
It's divide and conquer, but with no merge step. The hard work is instead in partitioning.

QUICKSORT VISUALIZATION

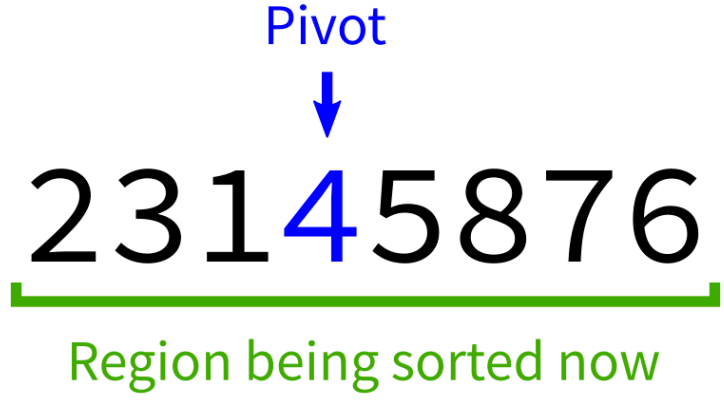
76235814

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION



QUICKSORT VISUALIZATION

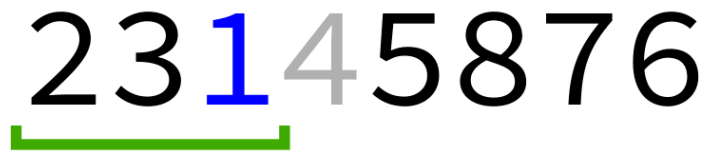


QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION



QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

QUICKSORT VISUALIZATION

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[end-1]` as a pivot, and reorder elements of `L` to partition `L[start: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