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 `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 `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[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

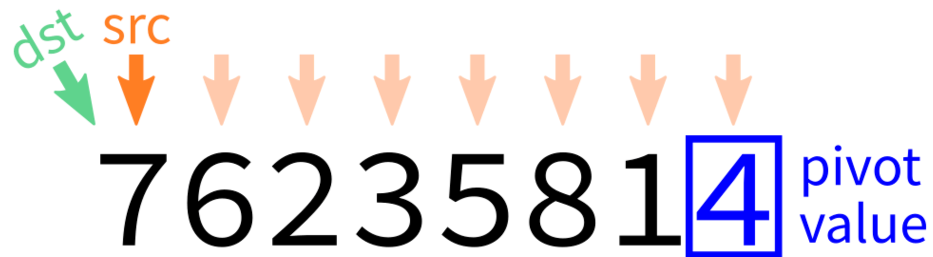
7623581 4 pivot value

PARTITION VISUALIZATION

dst src

76235814 pivot value

PARTITION VISUALIZATION



PARTITION VISUALIZATION

PARTITION VISUALIZATION

dst src

7623581 4 pivot value

L[src] ≥ pivot: do nothing

PARTITION VISUALIZATION

dst src

7623581 4 pivot value

PARTITION VISUALIZATION

dst src

7 6 2 3 5 8 1 4

pivot value

PARTITION VISUALIZATION

dst src

7623581**4** pivot value

L[src] ≥ pivot: do nothing

PARTITION VISUALIZATION

dst src

76235814 pivot value

PARTITION VISUALIZATION

dst

src

7 6 2 3 5 8 1 4

pivot value

PARTITION VISUALIZATION

dst

src

7623581**4** pivot value

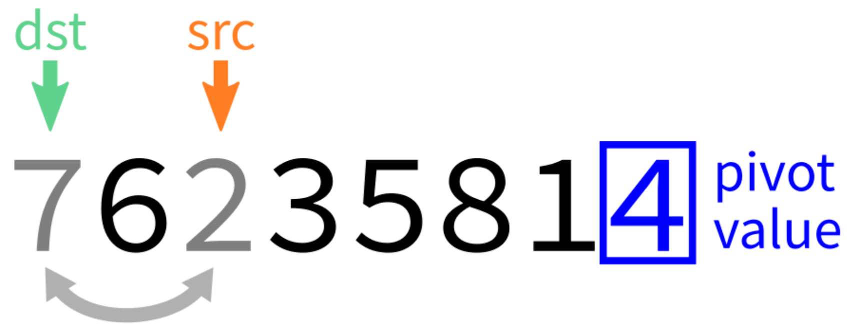
PARTITION VISUALIZATION

dst src

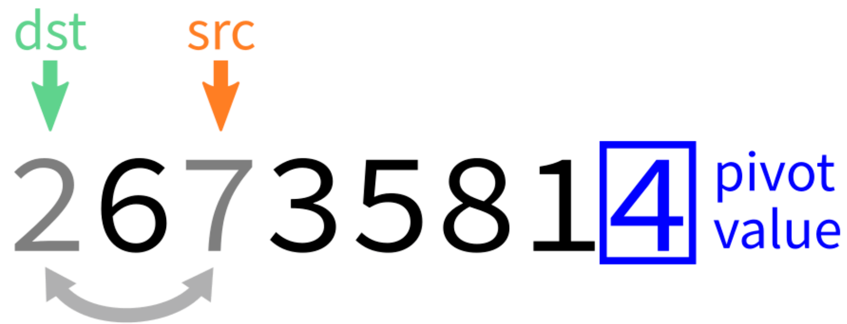
7 6 2 3 5 8 1 4 pivot value

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

PARTITION VISUALIZATION



PARTITION VISUALIZATION



PARTITION VISUALIZATION

dst

src

2673581**4** pivot value

PARTITION VISUALIZATION

dst

src

2673581 4

pivot value

PARTITION VISUALIZATION

dst

src

2 6 7 3 5 8 1 4

pivot value

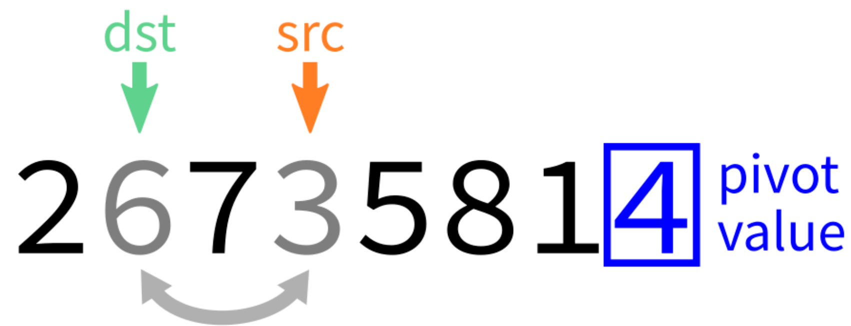
PARTITION VISUALIZATION

dst src

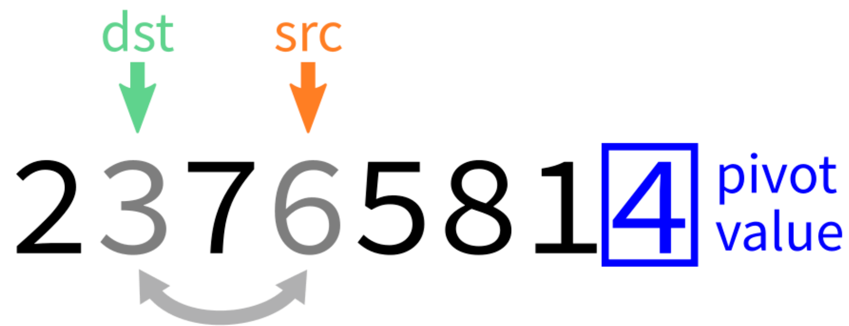
2 6 7 3 5 8 1 4 pivot value

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

PARTITION VISUALIZATION



PARTITION VISUALIZATION



PARTITION VISUALIZATION

dst

src

2 3 7 6 5 8 1 4

pivot value

PARTITION VISUALIZATION

dst

src

2376581**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

L[src] ≥ pivot: do nothing

PARTITION VISUALIZATION

dst
src

2376581 4 pivot value

PARTITION VISUALIZATION

dst
src

2376581 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

L[src] ≥ pivot: do nothing

PARTITION VISUALIZATION

dst
src

2376581 4 pivot value

PARTITION VISUALIZATION

dst

src

2376581 4

pivot value

PARTITION VISUALIZATION

dst

src

2 3 7 6 5 8 1 4

pivot value

PARTITION VISUALIZATION

dst

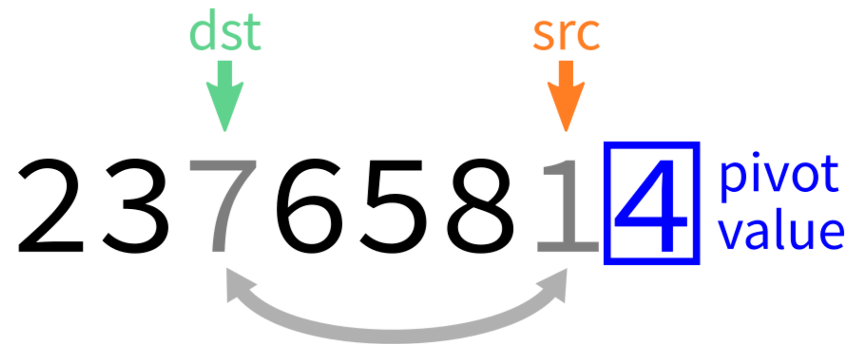
src

2376581 4

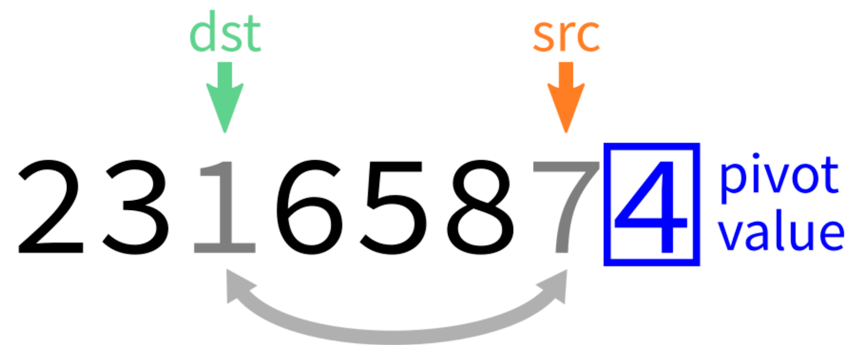
pivot value

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

PARTITION VISUALIZATION



PARTITION VISUALIZATION



PARTITION VISUALIZATION

dst

src

2316587 4

pivot value

PARTITION VISUALIZATION

dst

src

2316587 4 pivot value

PARTITION VISUALIZATION

dst

src

2316587 4

pivot value

PARTITION VISUALIZATION

dst

src

2316587**4** pivot value

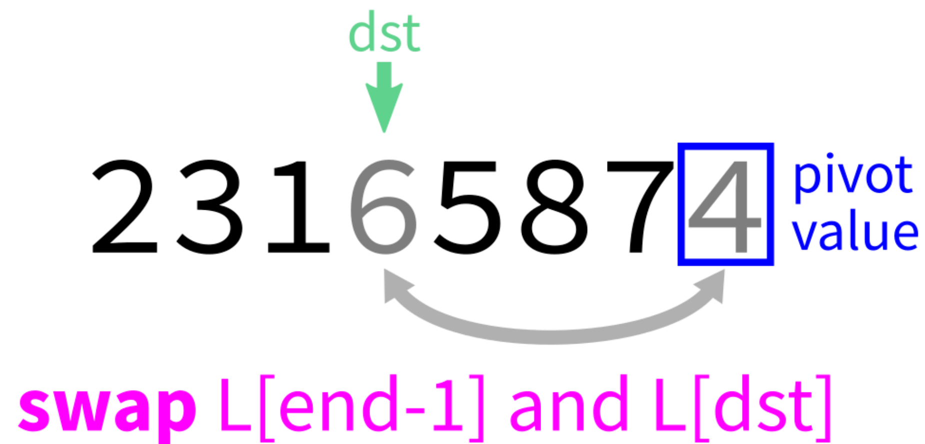
L[src] ≥ pivot: do nothing

PARTITION VISUALIZATION

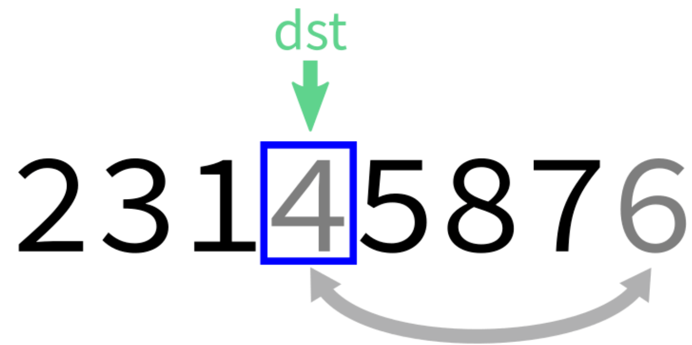
dst

2316587 4 pivot value

PARTITION VISUALIZATION



PARTITION VISUALIZATION



PARTITION VISUALIZATION

2 3 1 4 5 8 7 6

PARTITION VISUALIZATION

2 3 1 4 5 8 7 6

PARTITION VISUALIZATION

2 3 1 4 5 8 7 6

<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.

1. Let `pivot=L[end-1]`.
2. Initialize integer index `dst=start`.
3. For each integer `src` from `start` to `end-1`:
   - If `L[src] < pivot`, swap `L[src]` and `L[dst]`.
   - Increase `dst` by 1.
4. Swap `L[end-1]` and `L[dst]` to put the pivot in its proper place.
5. Return `dst`.

OTHER PARTITION STRATEGIES

Popular choices for the pivot:

- The last element, `L[end-1]` (used in lecture today)
- The first element, `L[start]`
- A random element of `L[start:end]`
- The element `L[(start+end)//2]`
- An element near the median of `L[start: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