CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Data structures

More sorting

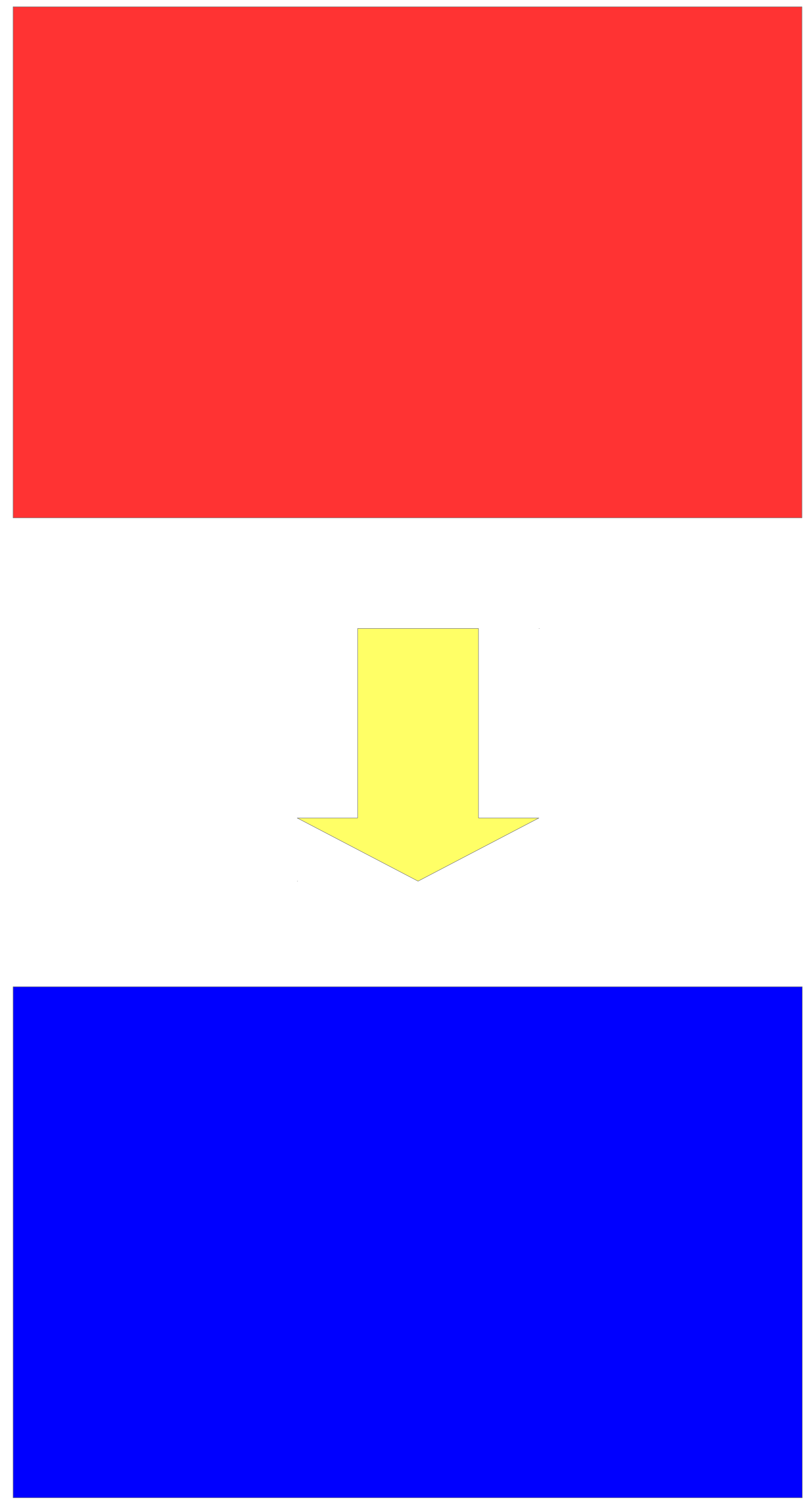
Dr. Alex Gerdes

DIT961 - VT 2018

Divide and conquer

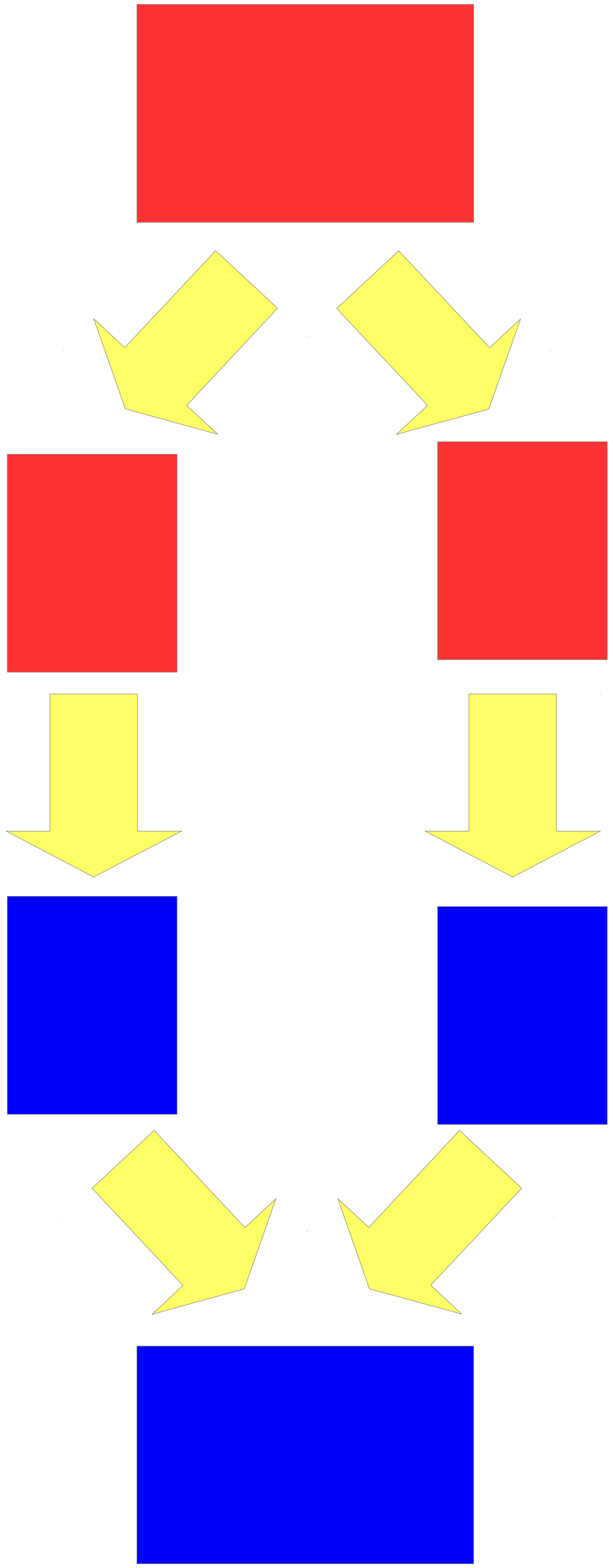
- Very general name for a type of recursive algorithm
- You have a problem to solve:
  - *Split* that problem into smaller subproblems
  - *Recursively* solve those subproblems
  - *Combine* the solutions for the subproblems to solve the whole problem

To solve this...



Divide and conquer

1. *Split* the problem into subproblems
2. *Recursively* solve the subproblems
3. *Combine* the solutions



Quicksort

- Pick an element from the array, called the *pivot*
- *Partition* the array:
  - First come all the elements smaller than the pivot, then the pivot, then all the elements greater than the pivot
- *Recursively* quicksort the two partitions

Quicksort

5	3	9	2	8	7	3	2	1	4

- Say the pivot is 5.
- Partition the array into: all elements less than 5, then 5, then all elements greater than 5

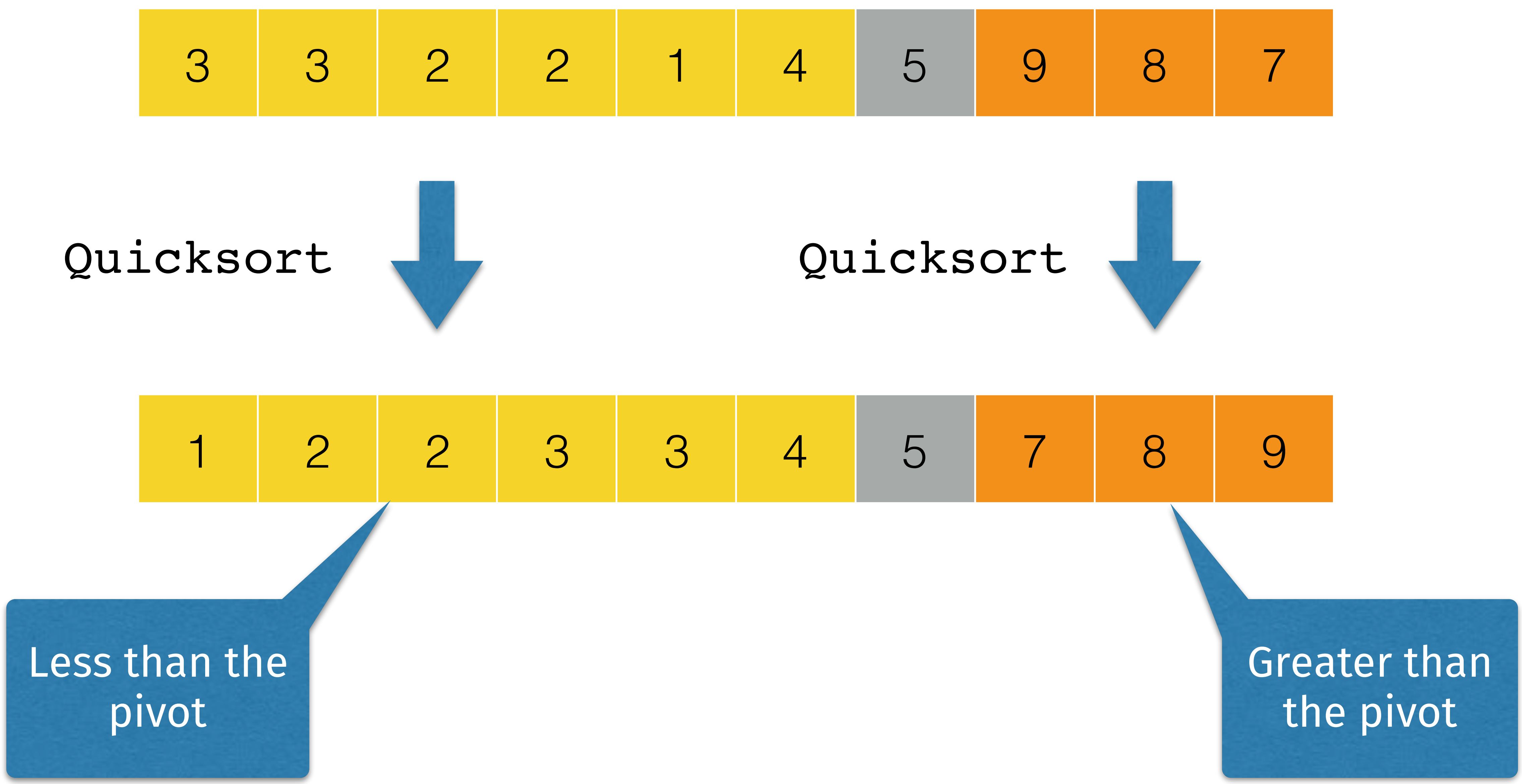
3	3	2	2	1	4	5	9	8	7

Less than the pivot

Greater than the pivot

Quicksort

- Now recursively quick sort the two partitions!



Pseudocode

```
// call as sort(a, 0, a.length-1);
void sort(int[] a, int low, int high) {
  if (low >= high) return;
  int pivot = partition(a, low, high);
    // assume that partition returns the
    // index where the pivot now is
  sort(a, low, pivot-1);
  sort(a, pivot+1, high);
}
```

- Common optimisation: switch to insertion sort when the input array is small

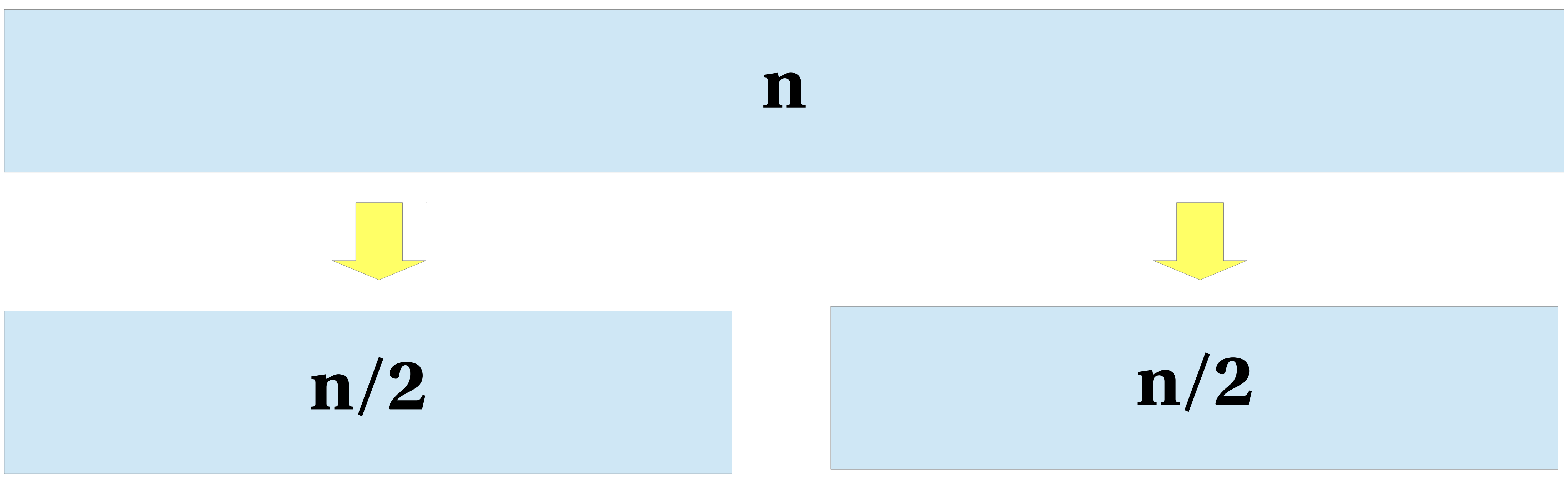
Quiz

What is the complexity of quicksort? *(assuming partition is $O(n)$)*

- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- Vet ej

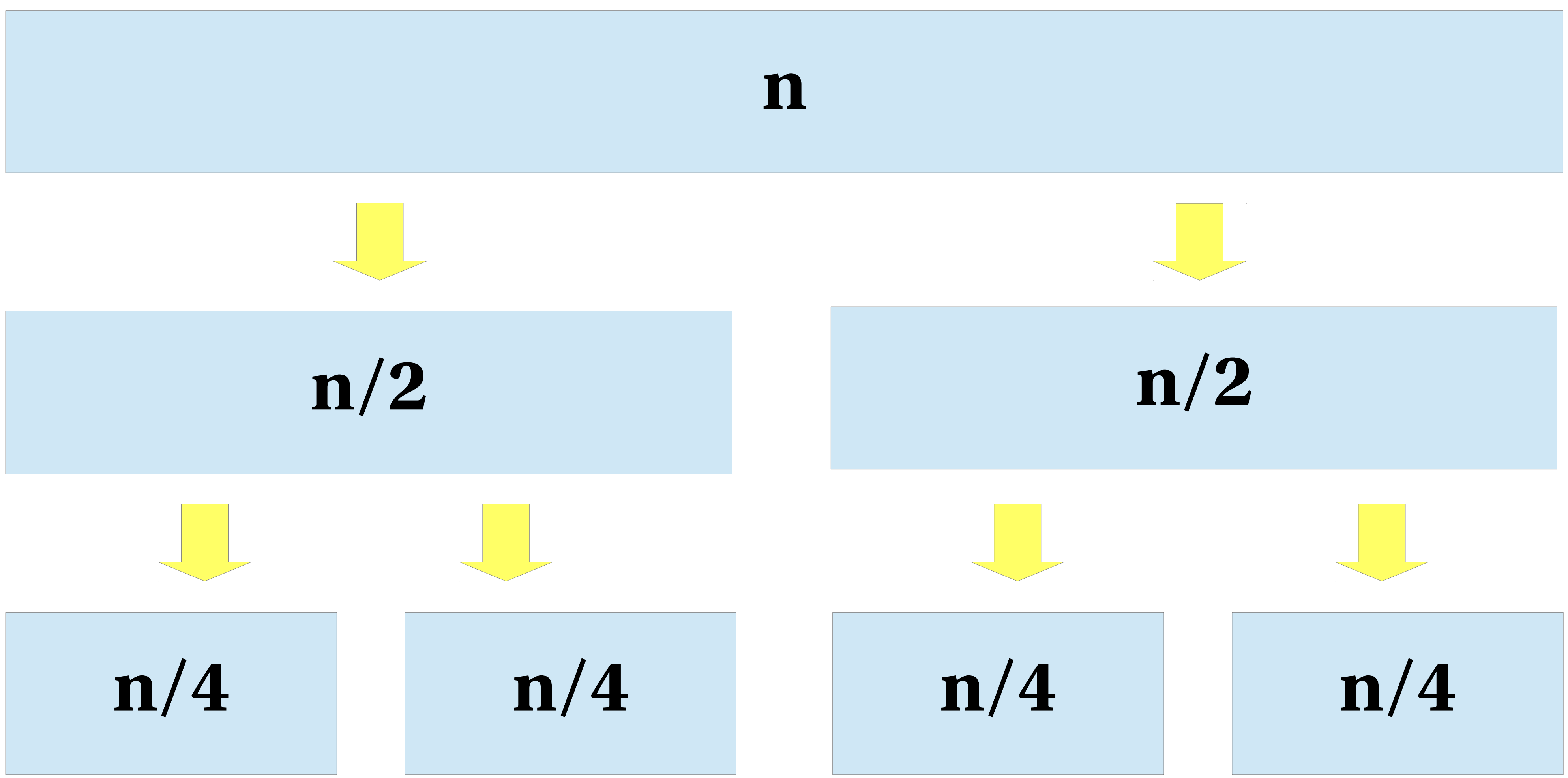
Complexity of quick sort

- In the best case, partitioning splits an array of size n into two halves of size $n/2$:

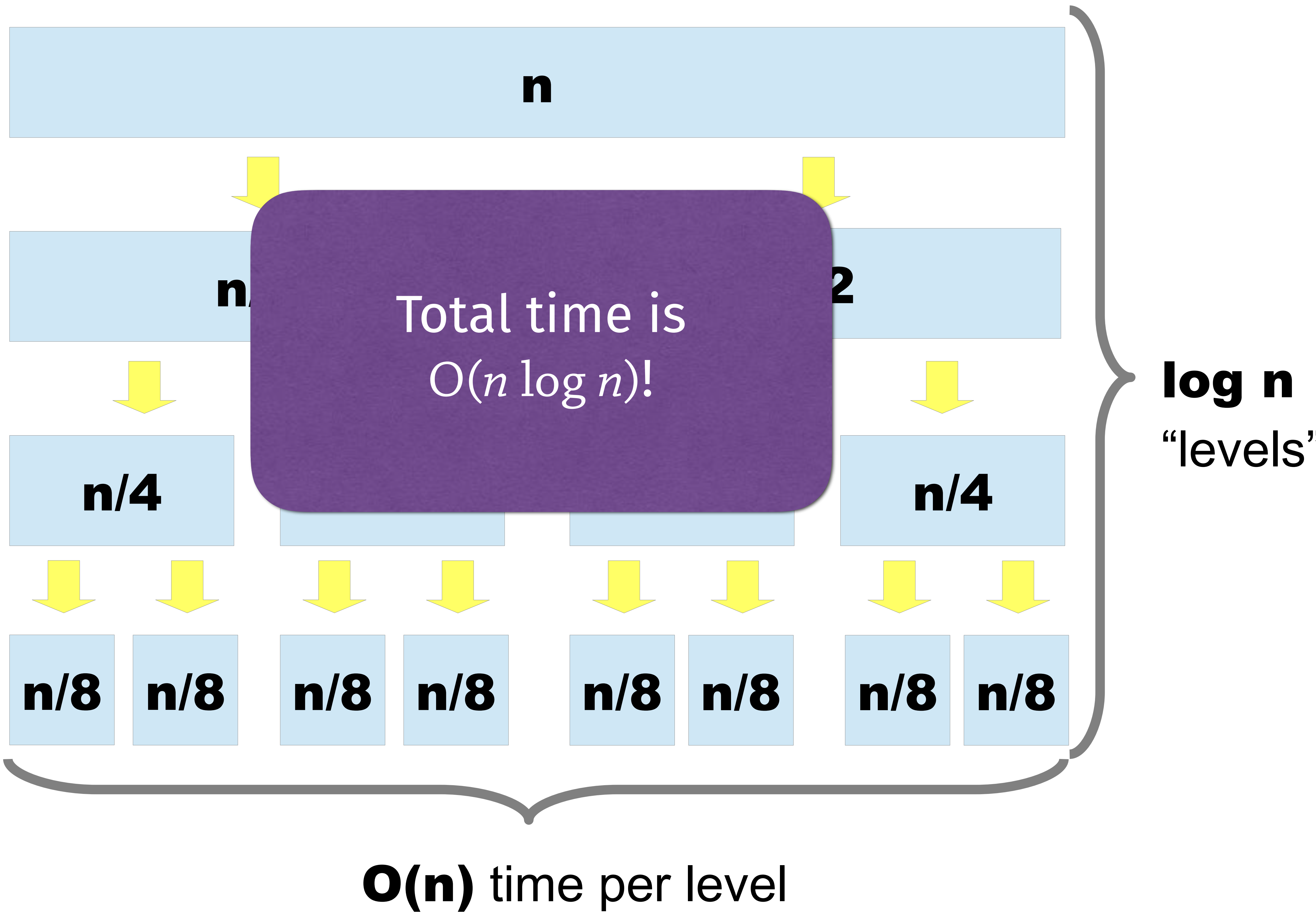


Complexity of quick sort

- The recursive calls will split these arrays into four arrays of size $n/4$:



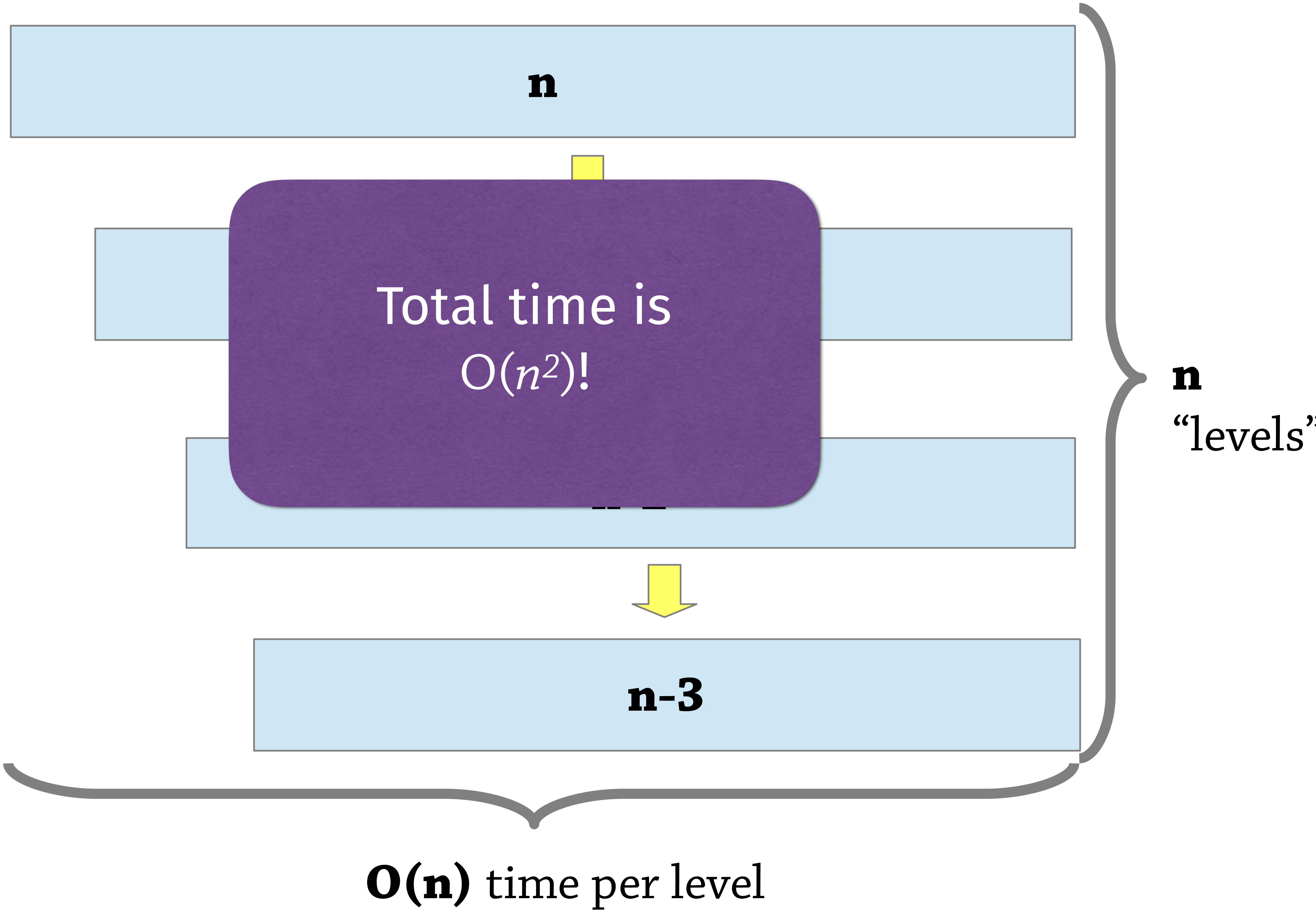
Complexity of quick sort



Total time is
$O(n \log n)$!

Complexity of quicksort

- But that's the best case!
- In the worst case, everything is greater than the pivot (say)
  - The recursive call has size n-1
  - Which in turn recurses with size n-2, etc.
  - Amount of time spent in partitioning:
    $n + (n-1) + (n-2) + \ldots + 1 = O(n^2)$


n
Total time is
O(n²)!
n-3
n
"levels"
O(n) time per level

Worst cases

- When we pick the first element as the pivot, we get this worst case for:
  - Sorted arrays
  - Reverse-sorted arrays
- The best pivot to use is the *median* value of the array, but in practice it's too expensive to compute...
- Most important decision in QuickSort: *what to use as the pivot*
- You don't need to split the array into exactly equal parts, it's enough to have some balance (e.g. 10%/90% split still gives $O(n \log n)$ runtime)

Complexity of quicksort

- Quicksort works well when the pivot splits the array into roughly equal parts
  - Median-of-three: pick first, middle and last element of the array and pick the median of those three
  - Pick pivot at random: gives $O(n \log n)$ expected (probabilistic) complexity
- Introsort: detect when we get into the $O(n^2)$ case and switch to a different algorithm (e.g. heapsort)

Partitioning algorithm

1. Pick a pivot (here 5)

5	3	9	2	8	7	3	2	1	4

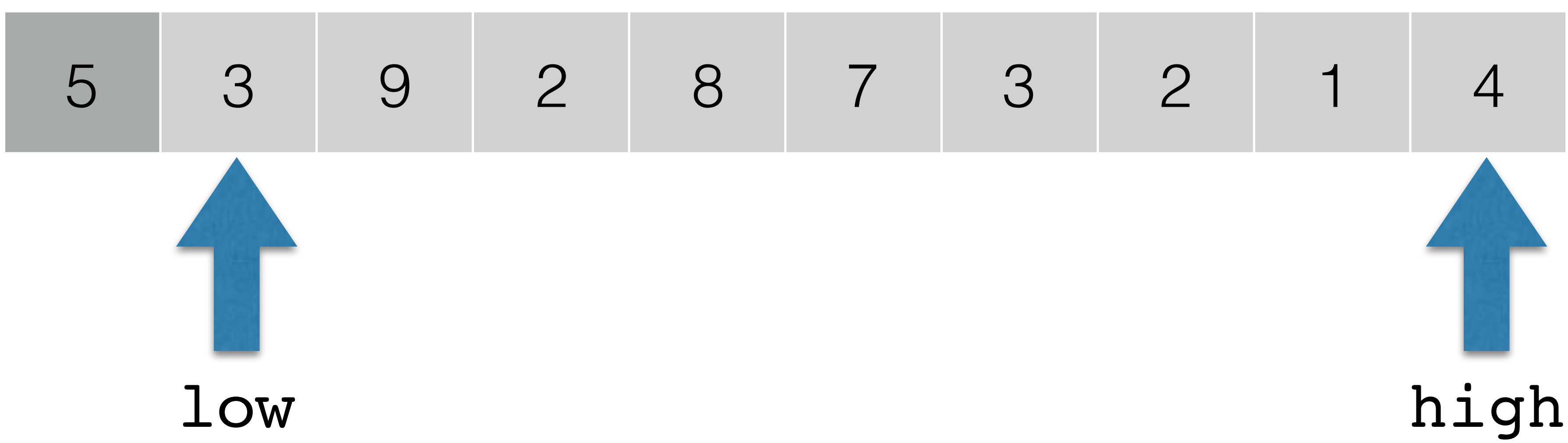
Partitioning algorithm

2. Set two indexes, `low` and `high`

Idea: everything to the left of `low` is *less* than the pivot (coloured yellow), everything to the right of `high` is *greater* than the pivot (orange)

Partitioning algorithm

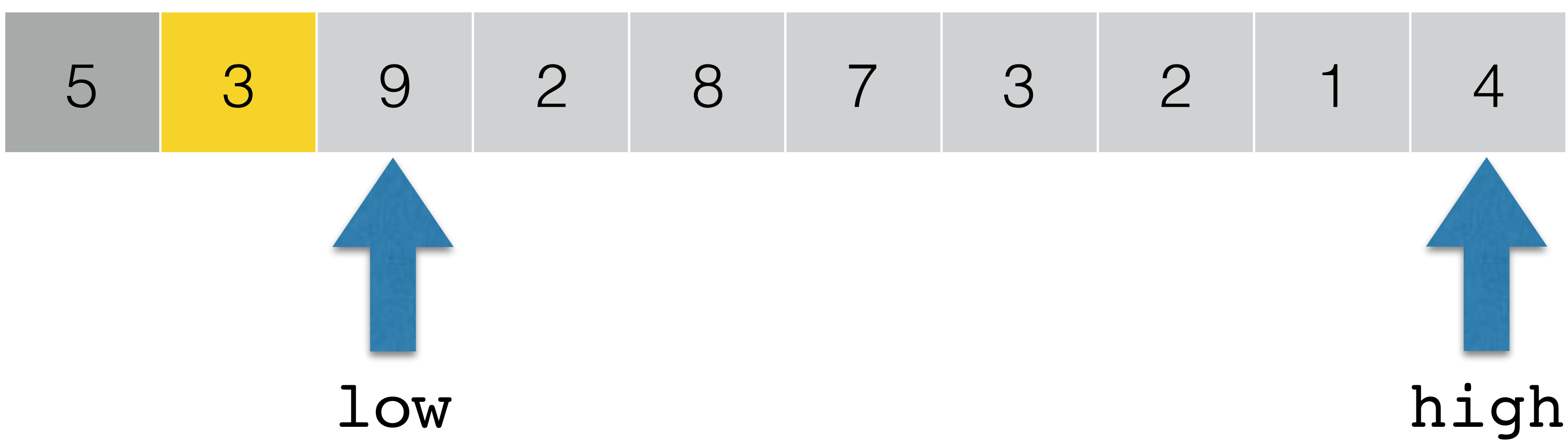
3. Move `low` right until you find something *greater* than the pivot




```
while (a[low] < pivot) low++;
```

Partitioning algorithm

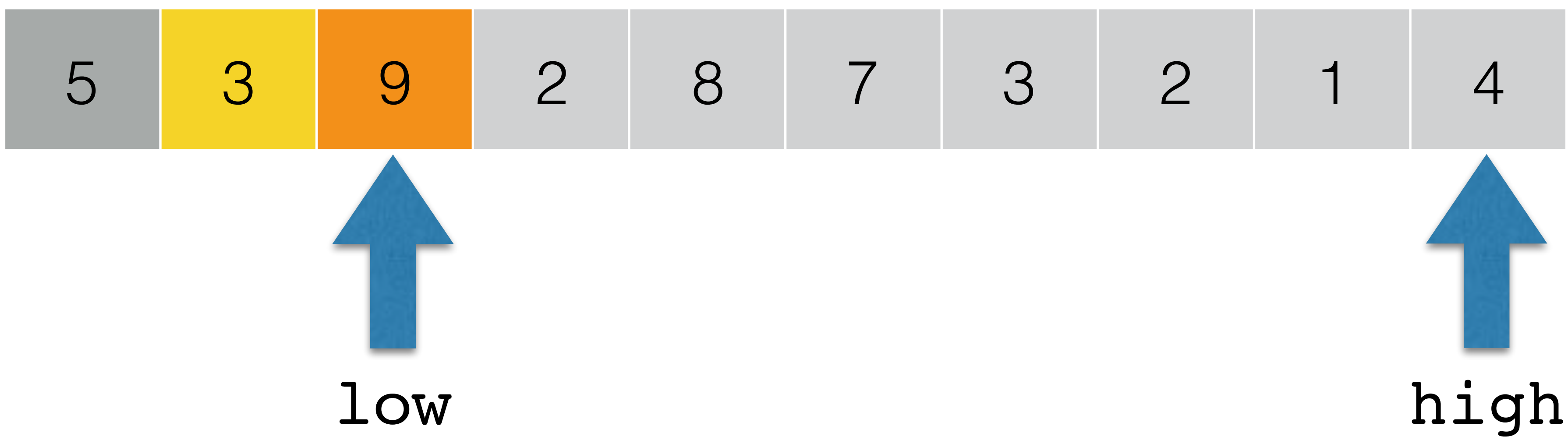
3. Move `low` right until you find something *greater* than the pivot




```
while (a[low] < pivot) low++;
```

Partitioning algorithm

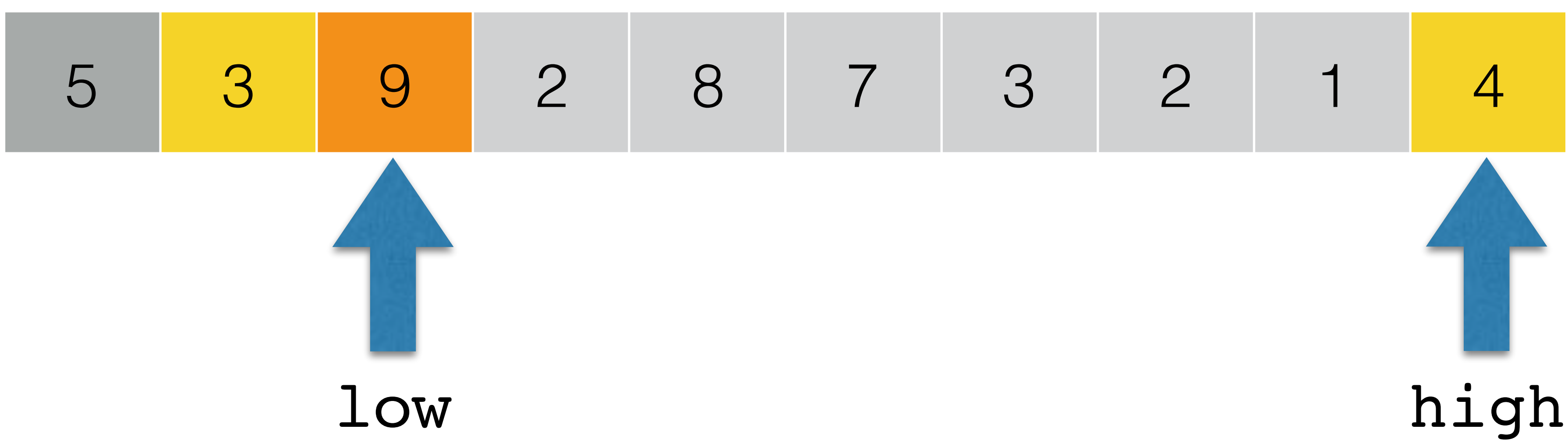
3. Move `low` right until you find something *greater* than the pivot




```
while (a[low] < pivot) low++;
```

Partitioning algorithm

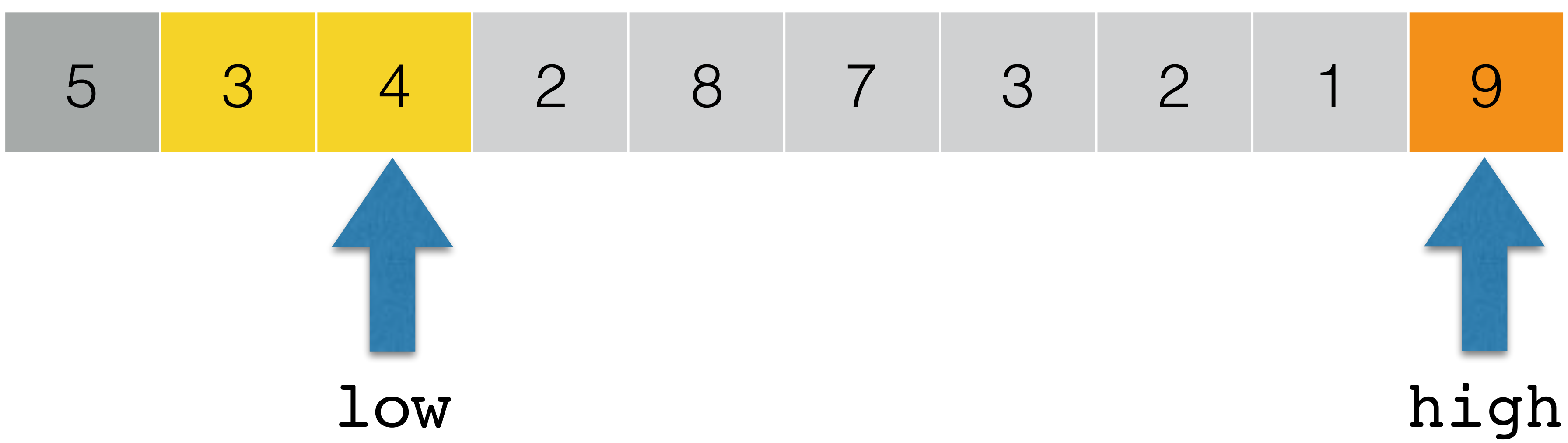
3. Move `low` right until you find something *greater* than the pivot




```
while (a[high] < pivot) high–;
```

Partitioning algorithm

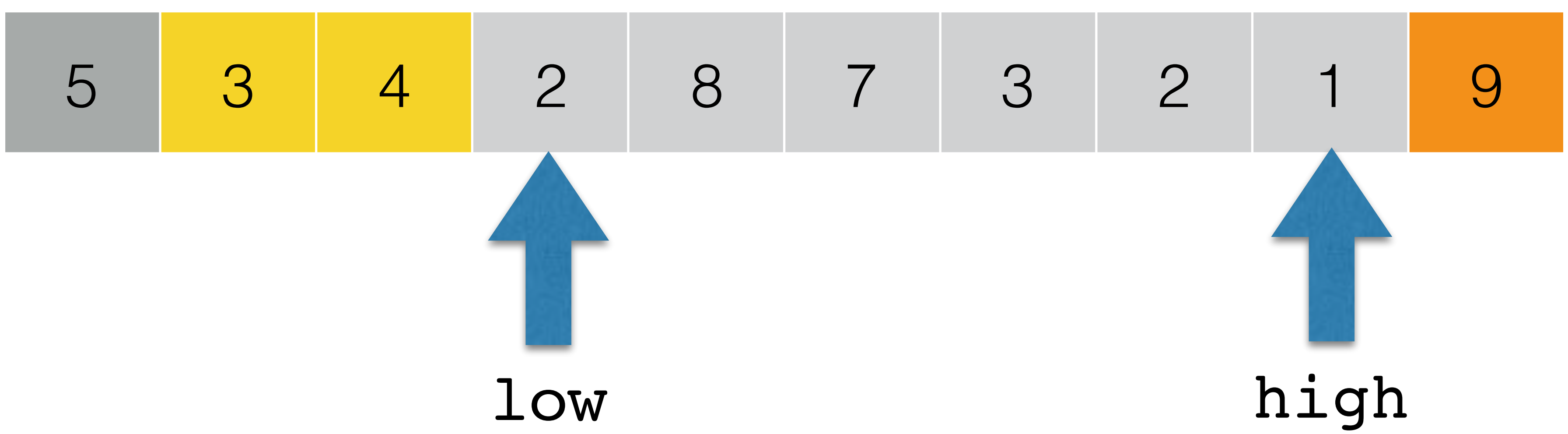
4. Swap them!




```
swap(a, low, high);
```

Partitioning algorithm

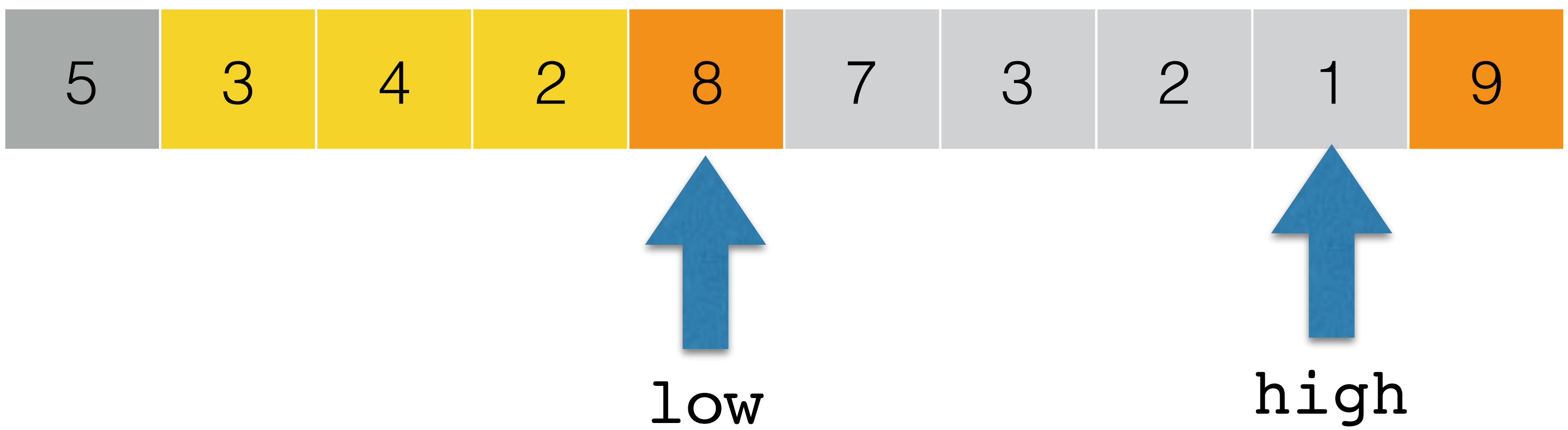
5. Advance `low` and `high` and repeat




```
low++; high–;
```

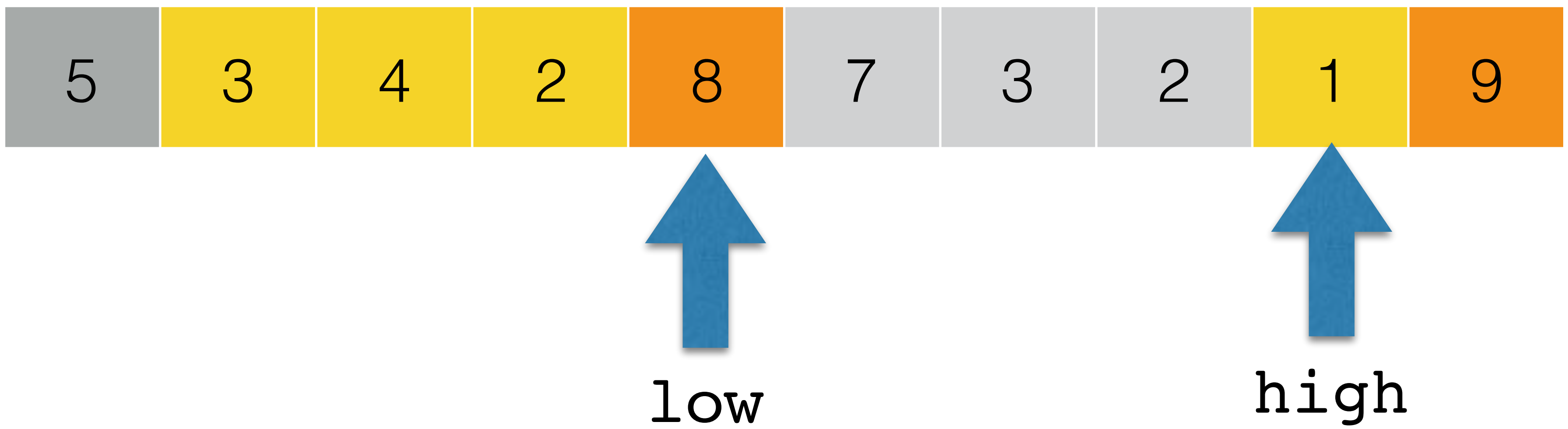
Partitioning algorithm

Move `low` until *higher* than pivot



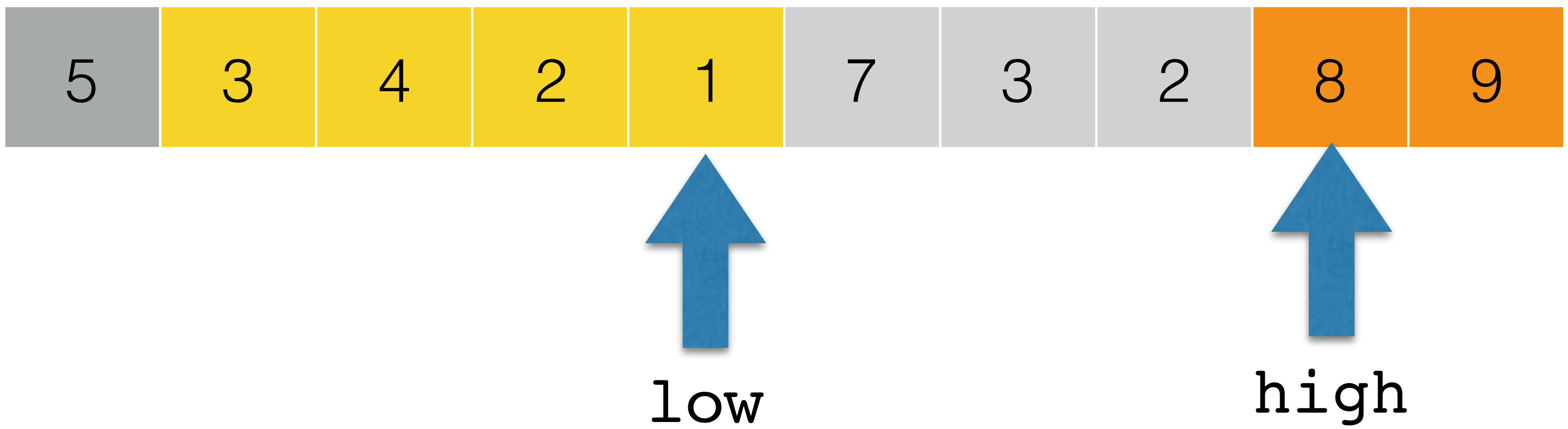
Partitioning algorithm

Move `high` until *lower* than pivot



Partitioning algorithm

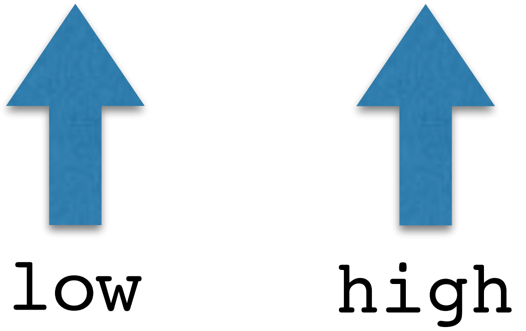
Swap `low` and `high`



Partitioning algorithm

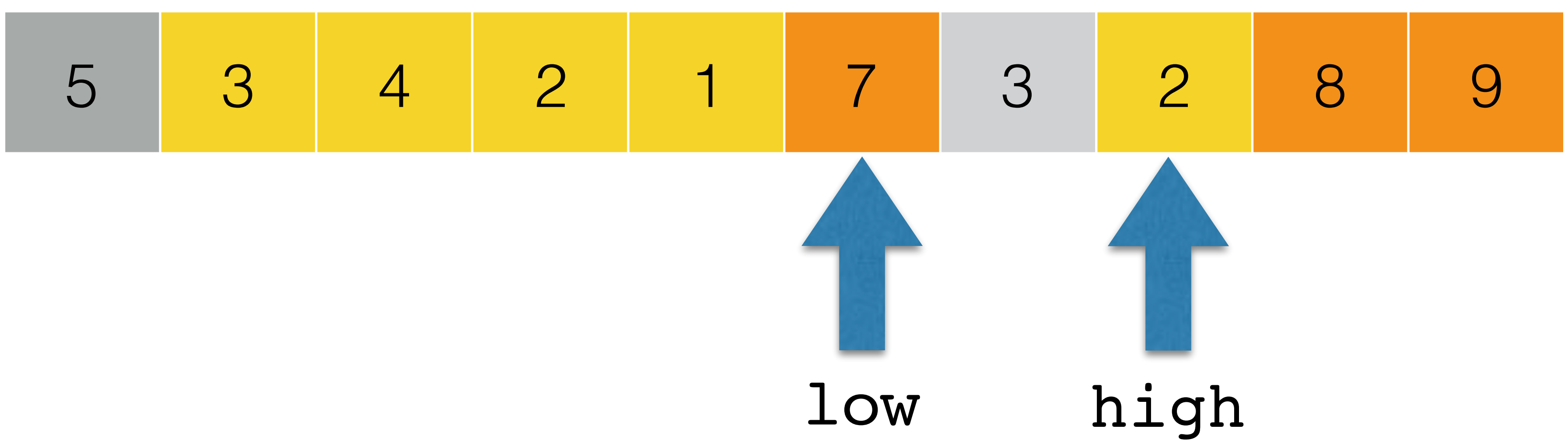
Advance and repeat

5	3	4	2	1	7	3	2	8	9



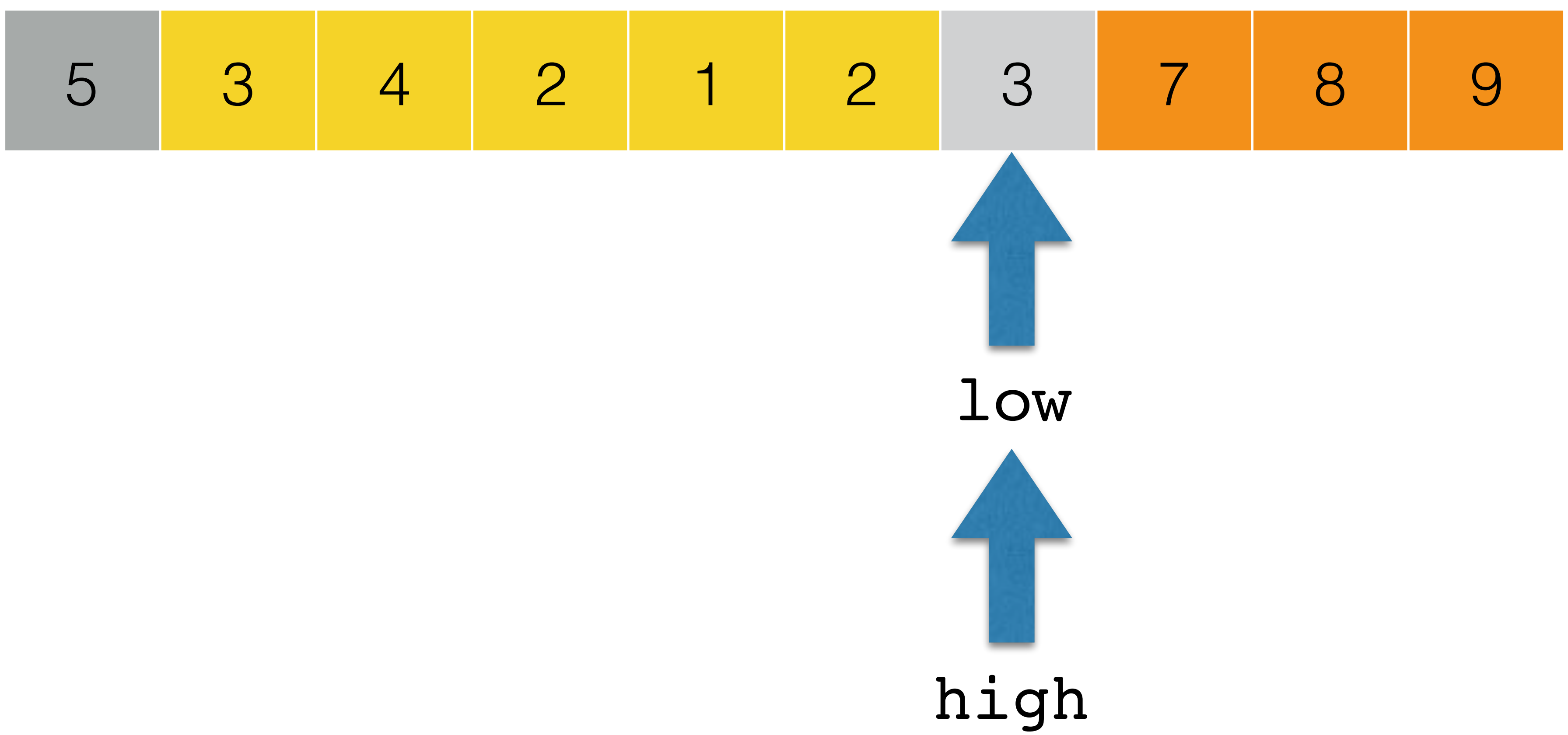
Partitioning algorithm

Move `low` and then `high`



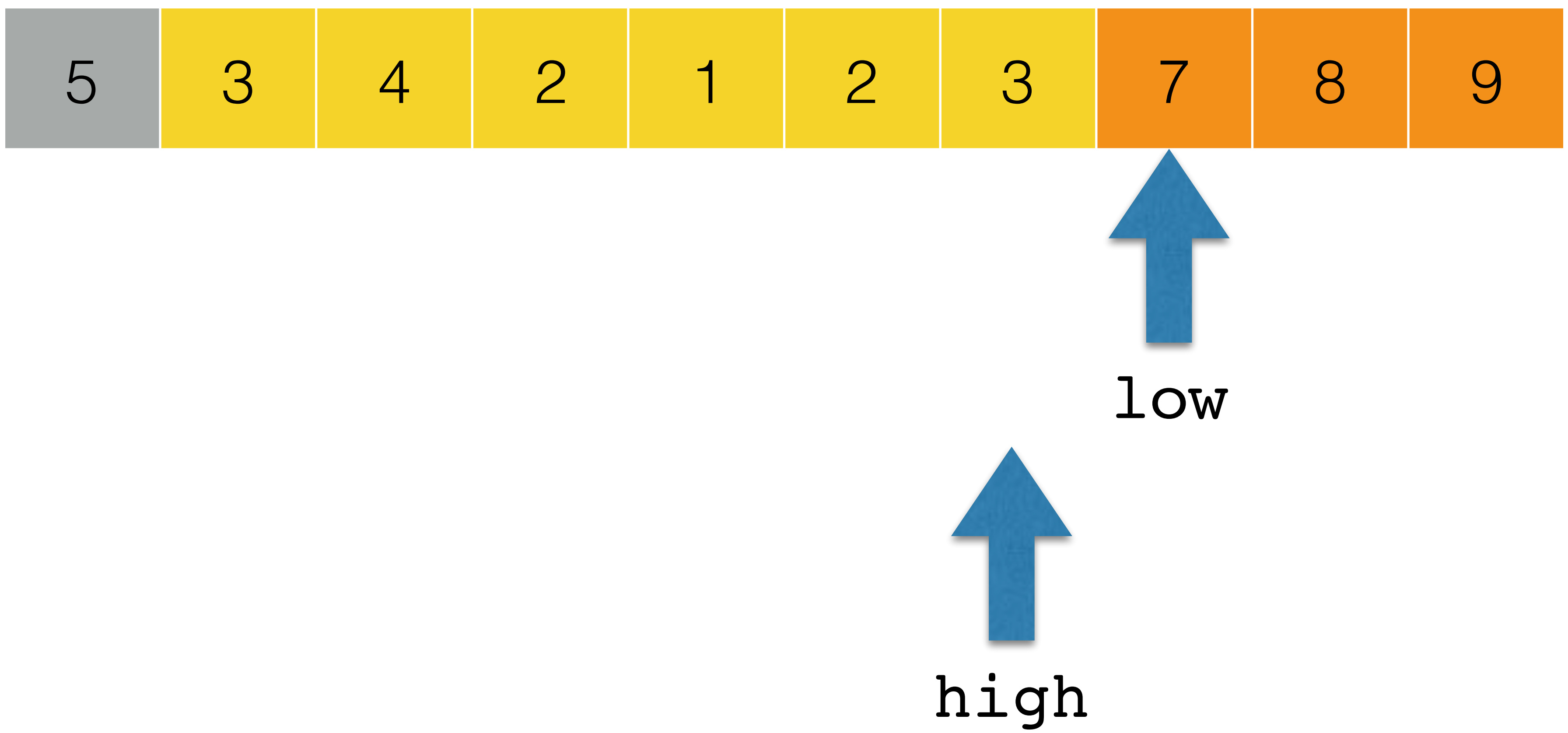
Partitioning algorithm

Swap and advance



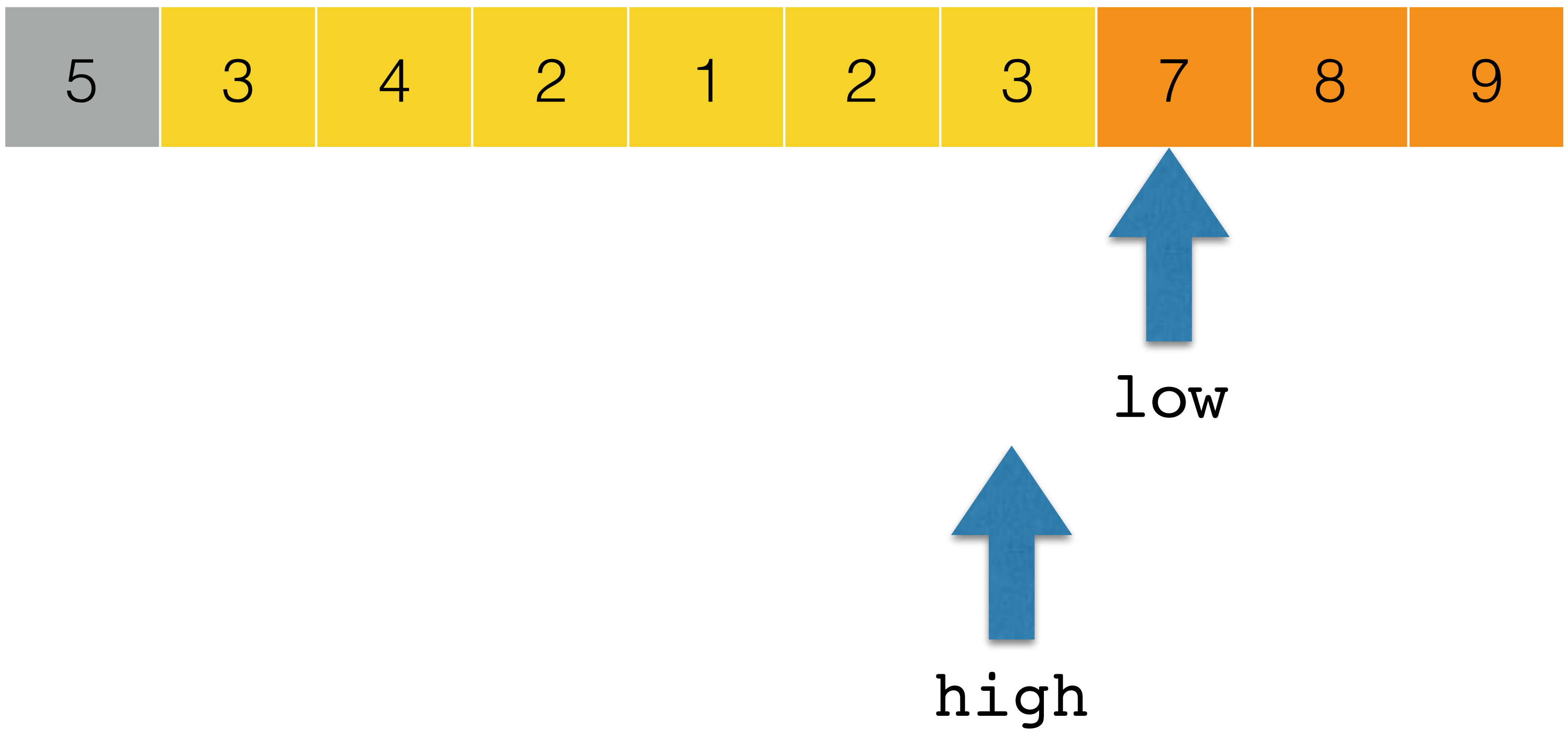
Partitioning algorithm

Move `high` and `low`



Partitioning algorithm

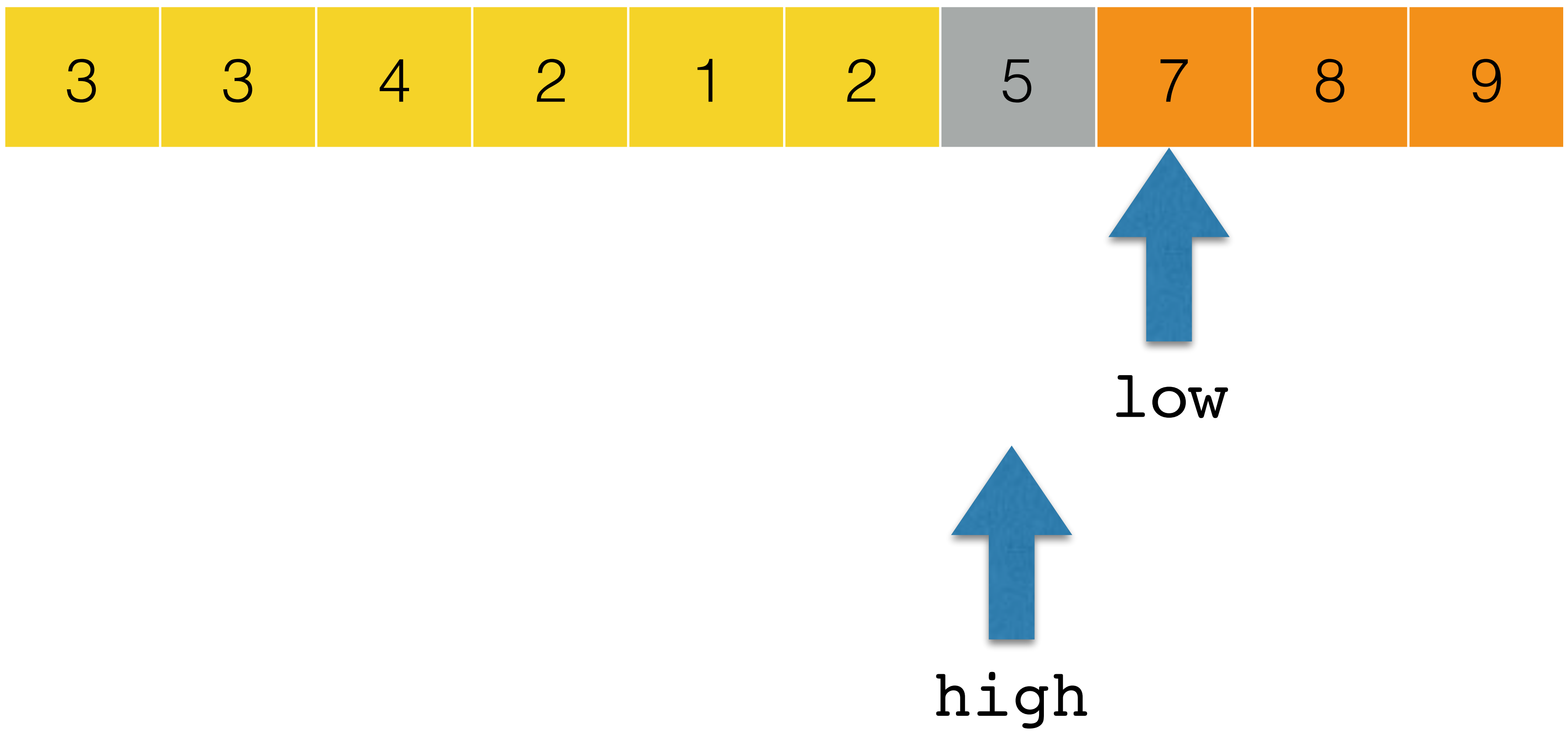
6. When `low` and `high` have crossed, we are finished!



But the pivot is in the wrong place…

Partitioning algorithm

7. Final step: swap `pivot` with `high`

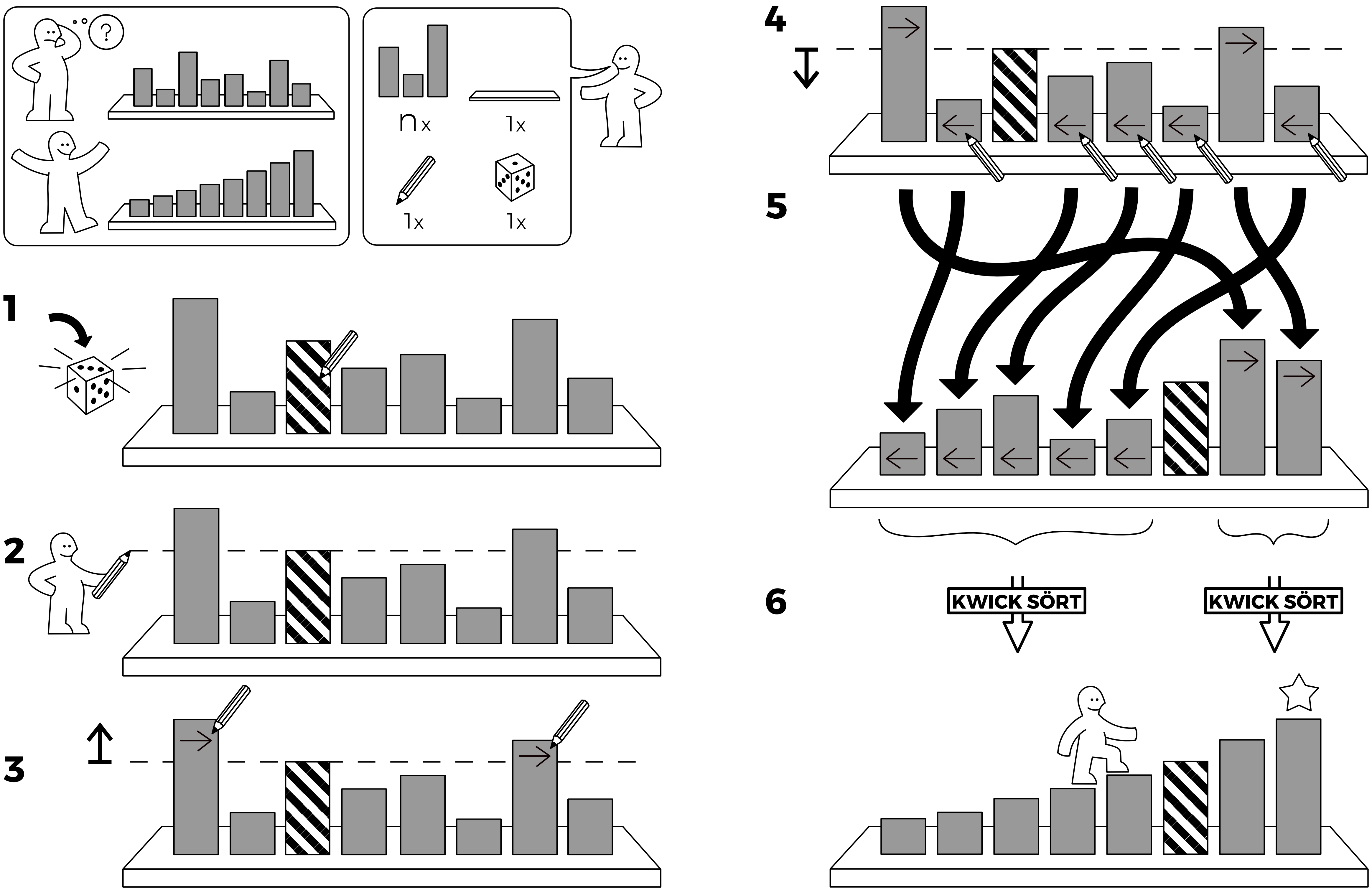


But the pivot is in the wrong place...

KWICK SÖRT

idea-instructions.com/quick-sort/

v1.0, CC by-nc-sa 4.0



Details

1. What to do if the pivot is not the first element?
   - Swap the pivot with the first element before starting partitioning!

2. What happens if the array contains many duplicates?
   - Notice that we only advance `a[low]` as long as `a[low] < pivot`
   - If `a[low] == pivot` we stop, same for `a[high]`
   - If the array contains just one element over and over again, `low` and `high` will advance at the same rate
   - Hence we get equal-sized partitions

Pivot

- Which pivot should we pick?
  - First element: gives $O(n^2)$ behaviour for already- sorted lists
  - Median-of-three: pick first, middle and last element of the array and pick the median of those three
  - Pick pivot at random: gives $O(n \log n)$ *expected* (probabilistic) complexity

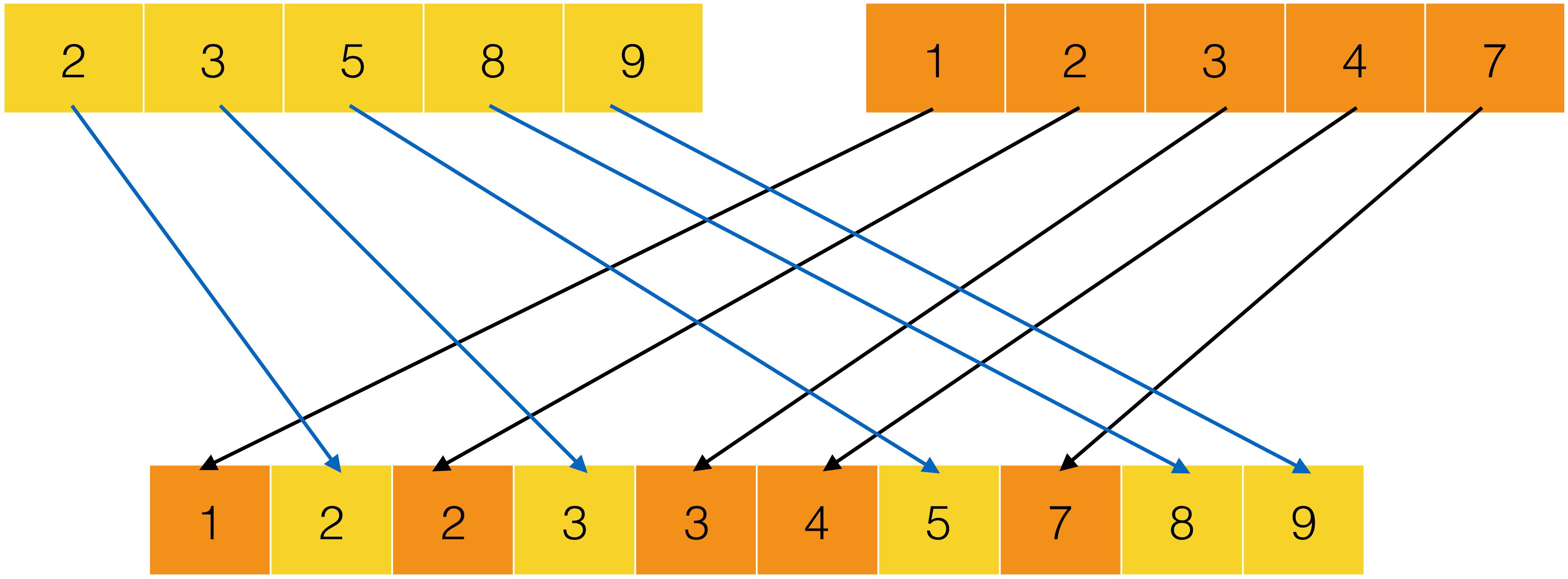
Quicksort

- Typically the fastest sorting algorithm...
  ...but very sensitive to details!
  - Must choose a good pivot to avoid $O(n^2)$ case
  - Must take care with duplicates
  - Switch to insertion sort for small arrays to get better constant factors
- If you do all that right, you get an in-place sorting algorithm, with low constant factors and $O(n \log n)$ complexity

Mergesort

Mergesort

- We can *merge* two sorted lists into one in linear time:

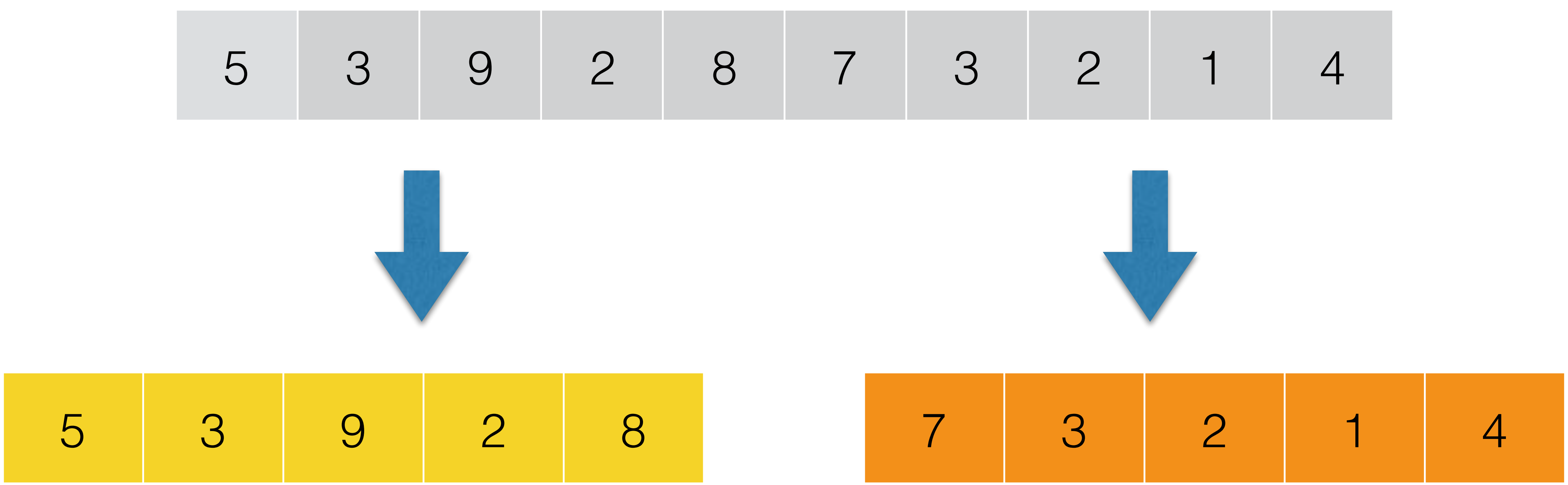


Mergesort

- Another divide-and-conquer algorithm
- To mergesort a list:
  - *Split* the list into two equal parts
  - *Recursively* mergesort the two parts
  - *Merge* the two sorted lists together

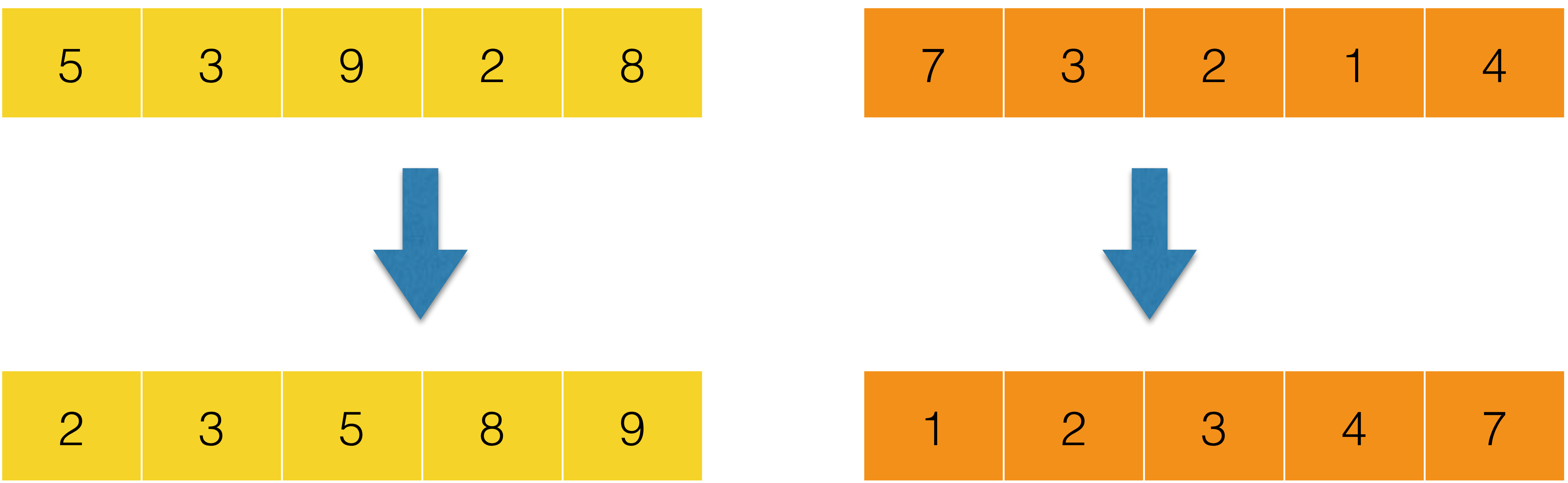
Mergesort

1. *Split* the list into two equal parts



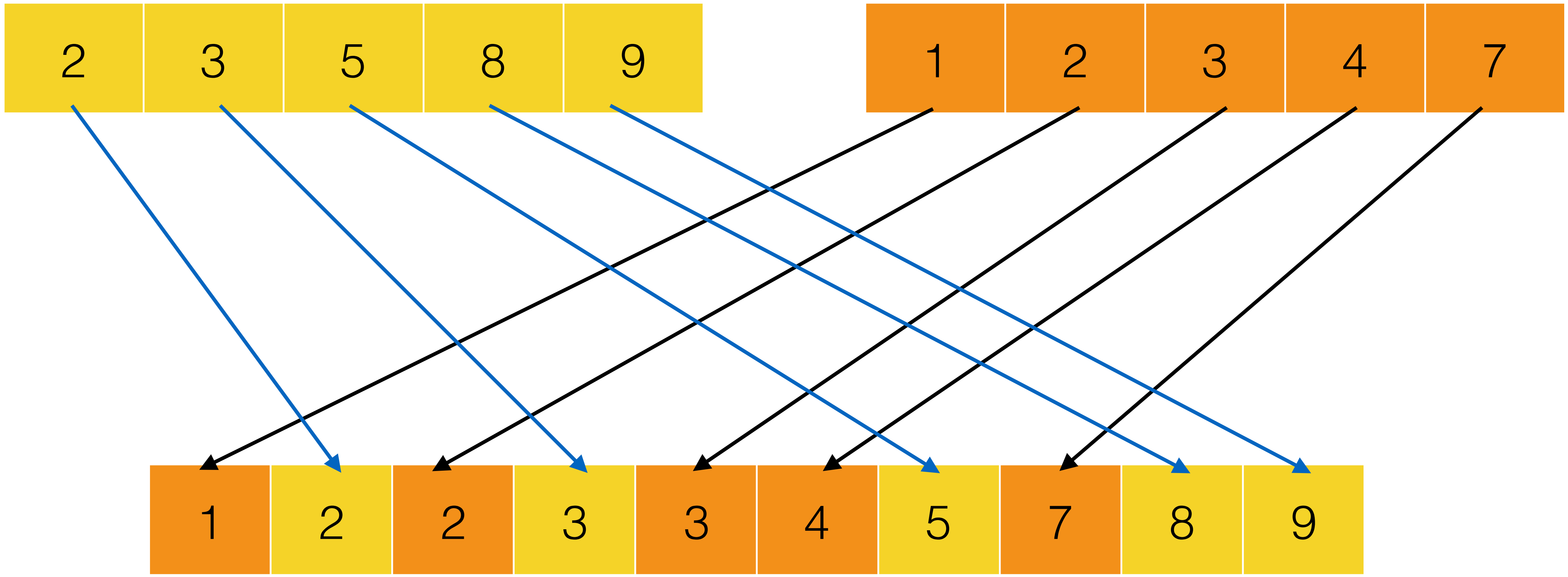
Mergesort

2. *Recursively* mergesort the two parts



Mergesort

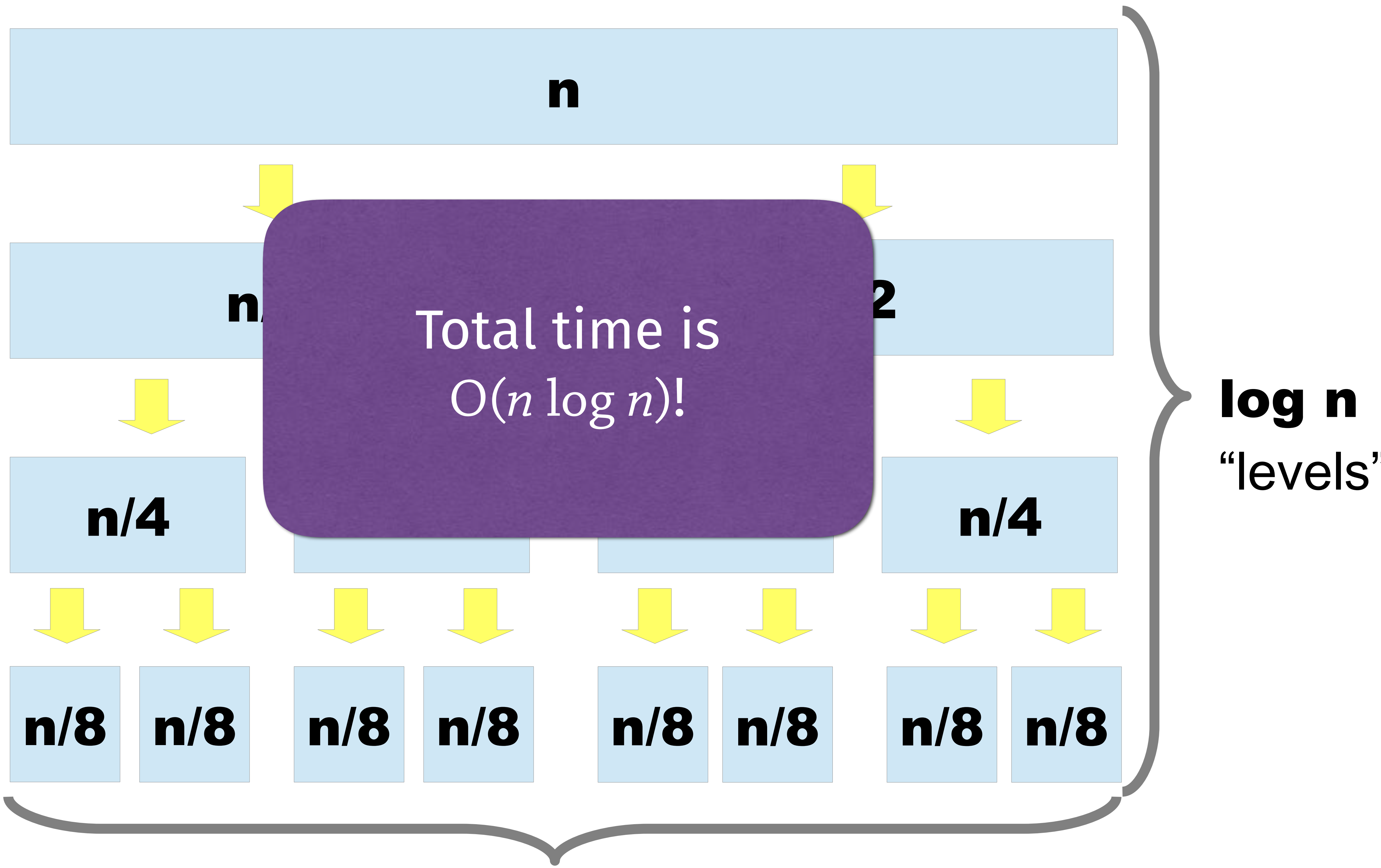
3. *Merge* the two sorted lists together



Complexity analysis

- Mergesort's divide-and-conquer approach is similar to quicksort
- But it *always splits the list into equally- sized pieces*!
- Hence $O(n \log n)$, just like the best case for quicksort – but this is the *worst case* for mergesort

Complexity of quick sort



Total time is $O(n \log n)$!

O(n) time per level

Mergesort vs quicksort

- Mergesort:
  - Not in-place
  - $O(n \log n)$
  - Only requires sequential access to the list – this makes it good in functional programming
- Quicksort:
  - In-place
  - $O(n \log n)$ but $O(n^2)$ if you are not careful
  - Works on arrays only (random access)
  - Unstable
- Both the best in their fields!
  - Quicksort best imperative algorithm
  - Mergesort best functional algorithm

Stable sorting

- When sorting complex objects, e.g. where each element contains various information about a person, the ordering may only take part of the data in account (via Comparable, Comparator, Ord)
- Then it's sometimes important that objects that are deemed equal by the ordering should appear in the same order as they did in the original list
- A sorting algorithm that does not change the order of equal elements is called *stable*

Stable sorting

- Let's say that we want to sort
  `[(5, "a"), (3, "d"), (2, "f"), (3, "b")]`
  and that the ordering of the pairs is defined to be the natural ordering of the first component

- Unstable sorting might result in
  `[(2, "f"),  (3, "b"), (3, "d"), (5, "a")]`

- Stable sorting always gives
  `[(2, "f"),  (3, "d"), (3, "b"), (5, "a")]`

- Insertion sort is stable (provided that the insert inequality check is the right one, so that equal elements are not swapped).