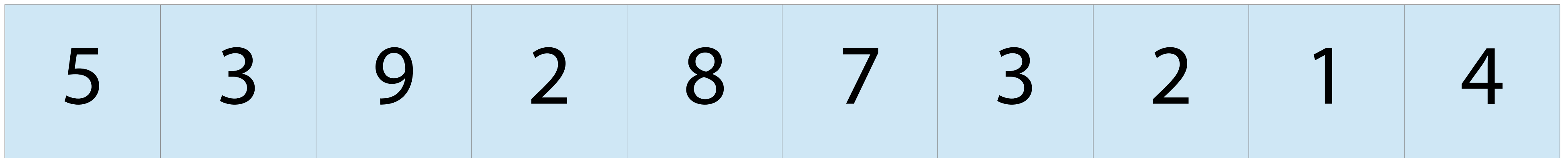
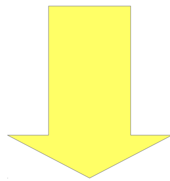
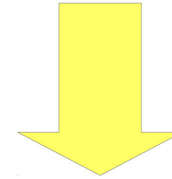
Quicksort

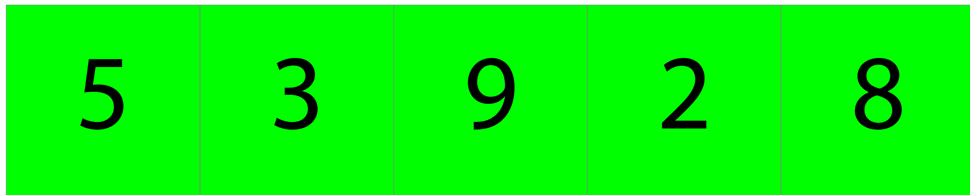
Mergesort again

1. *Split* the list into two equal parts



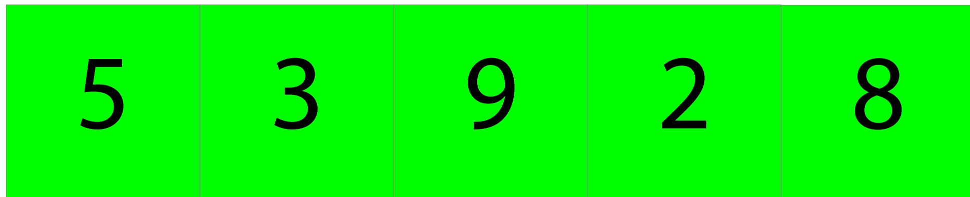


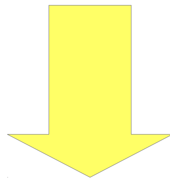


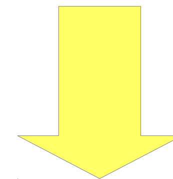


Mergesort again

2. *Recursively* mergesort the two parts

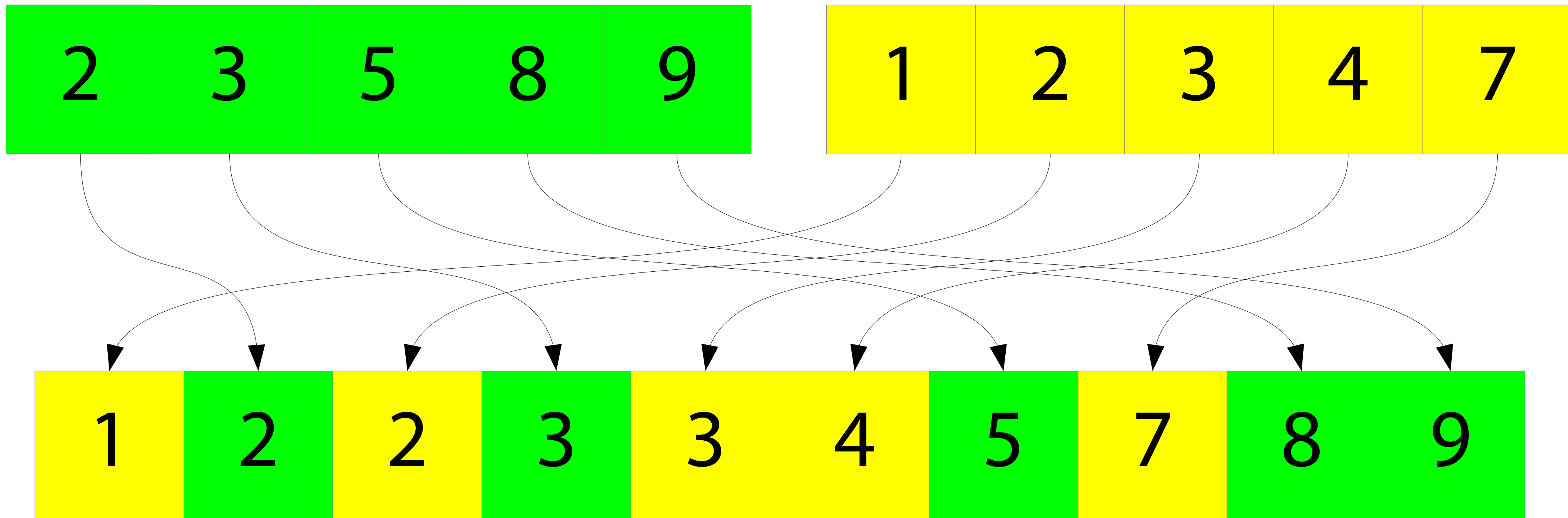






Mergesort again

3. *Merge* the two sorted lists together



Quicksort

Mergesort is great... except that it's not in-place

- So it needs to allocate memory
- And it has a high constant factor

Quicksort: let's do divide-and-conquer sorting, but do it in-place

Quicksort

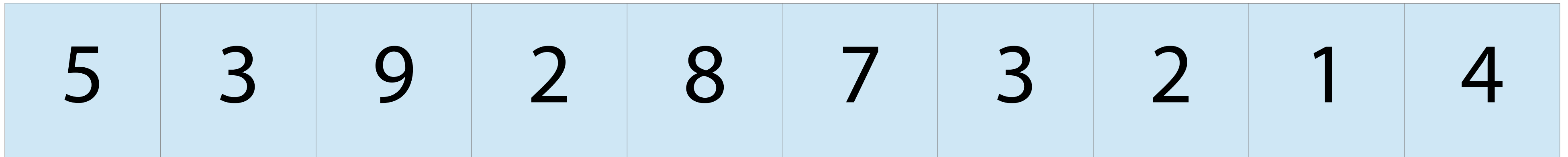
Pick an element from the array, called the *pivot*

Partition the array:

- First come all the elements smaller than the pivot, then the pivot, then all the elements greater than the pivot
- Partitioning has complexity O(n)

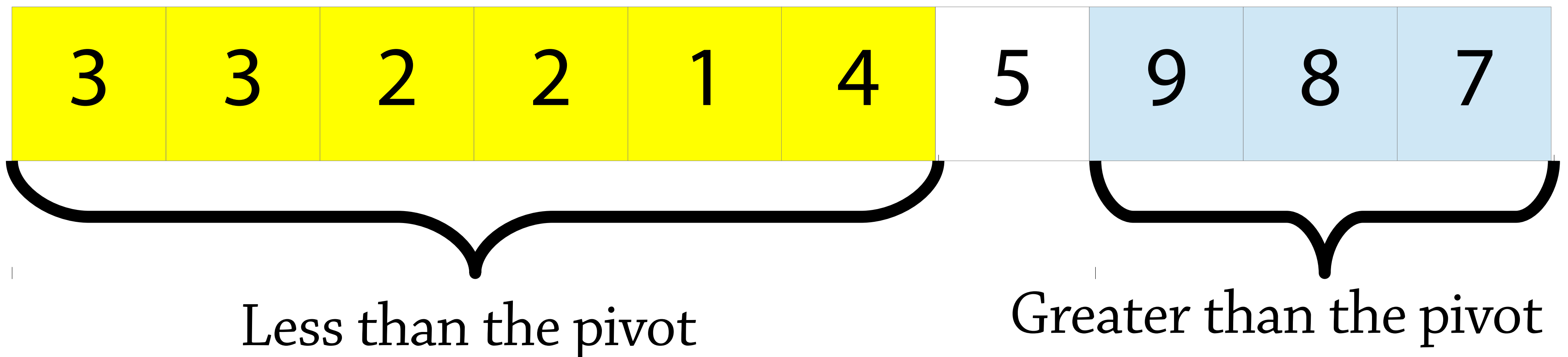
Recursively quicksort the two partitions

Quicksort



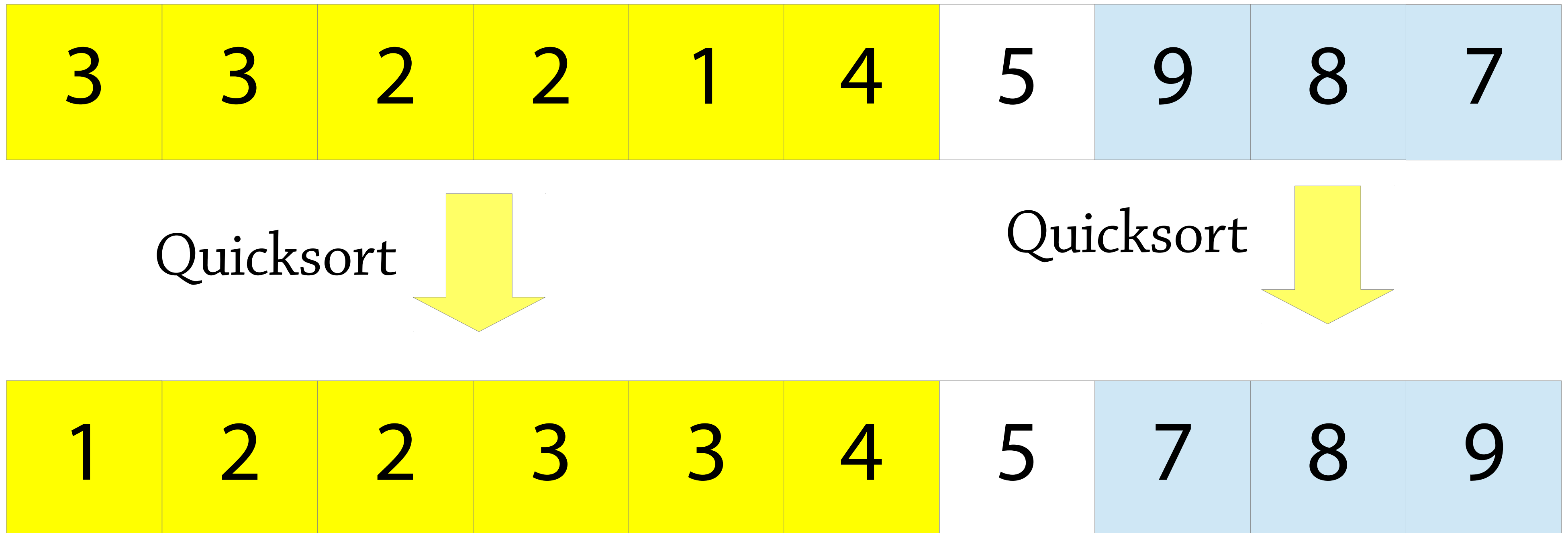
Say the pivot is 5.

Partition the array into: all elements less than 5, then 5, then all elements greater than 5



Quicksort

Now recursively quicksort 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

Quicksort's performance

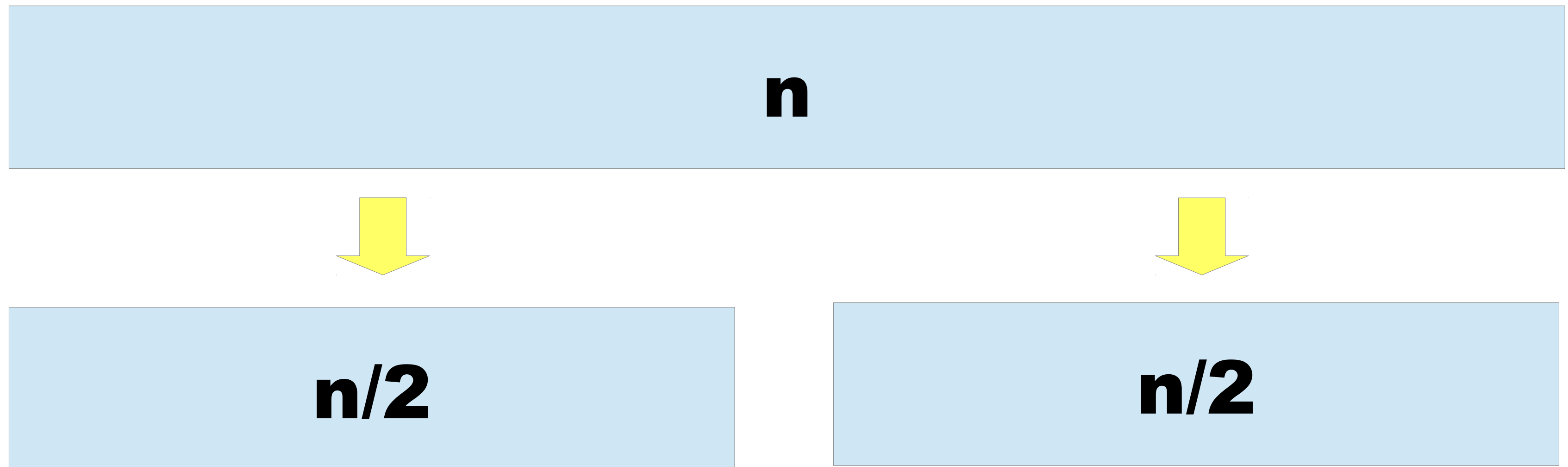
Mergesort is fast because it splits the array into two *equal* halves

Quicksort just gives you two halves of whatever size!

So does it still work fast?

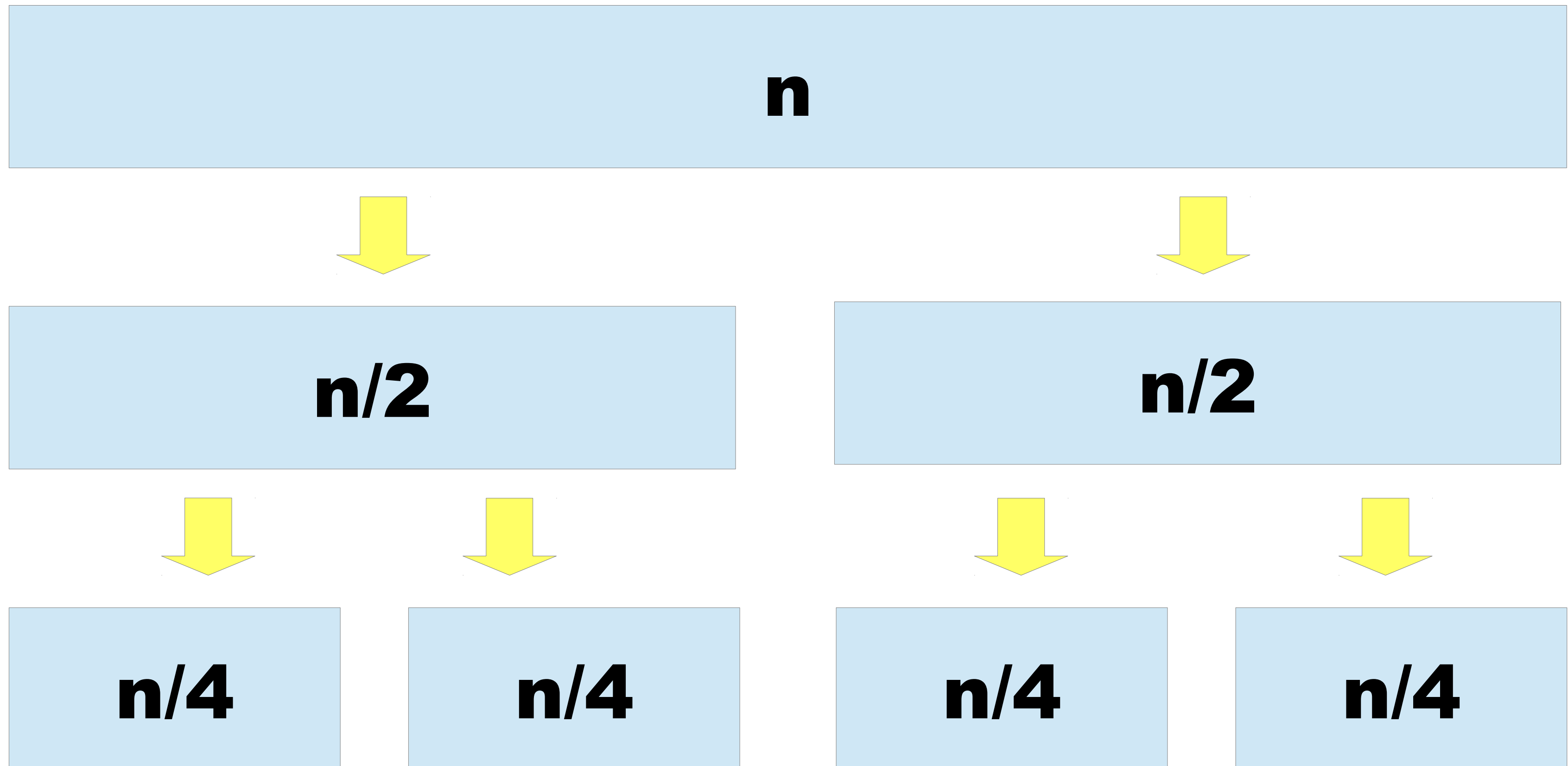
Complexity of quicksort

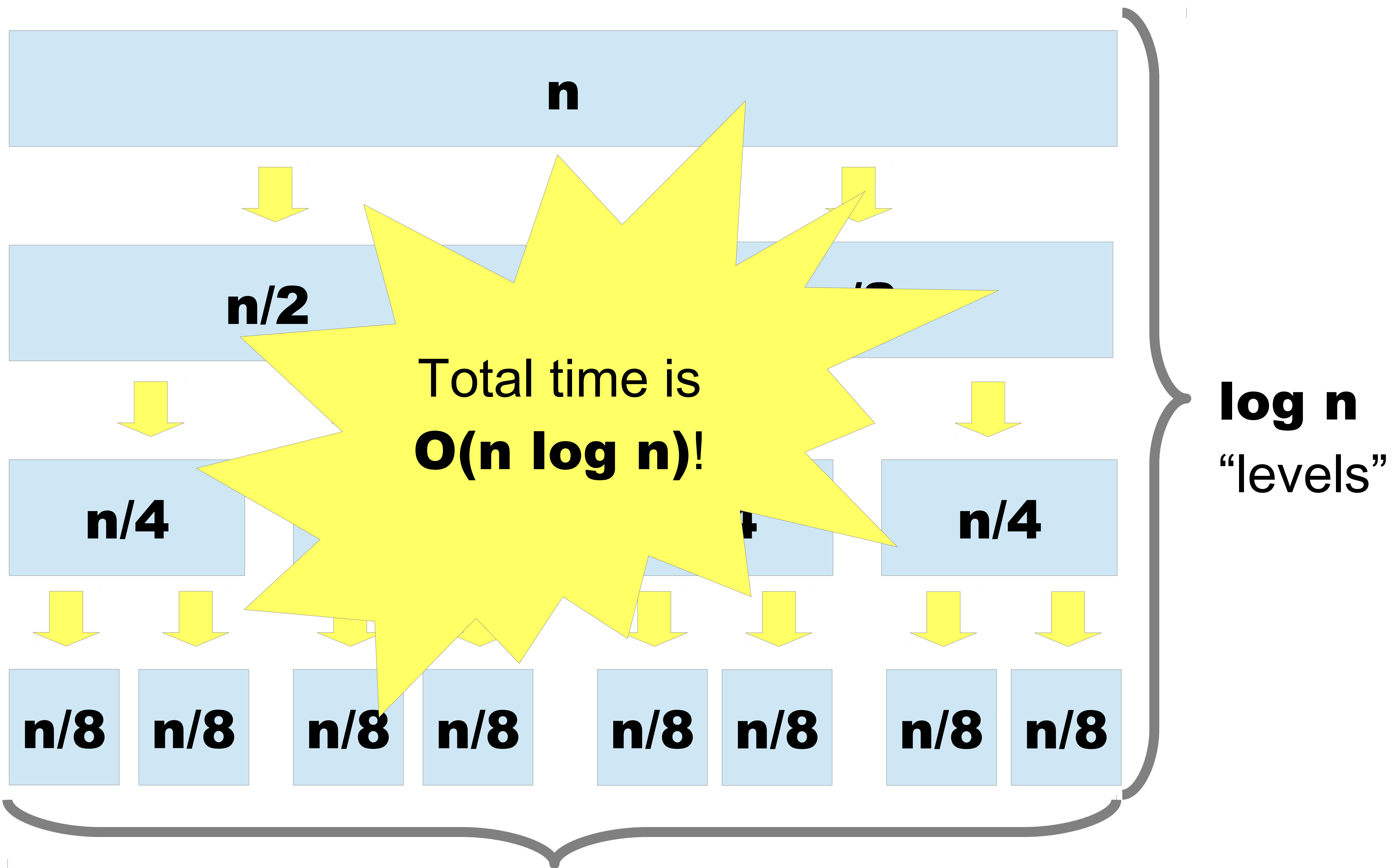
In the best case, partitioning splits an array of size n into two halves of size n/2:



Complexity of quicksort

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





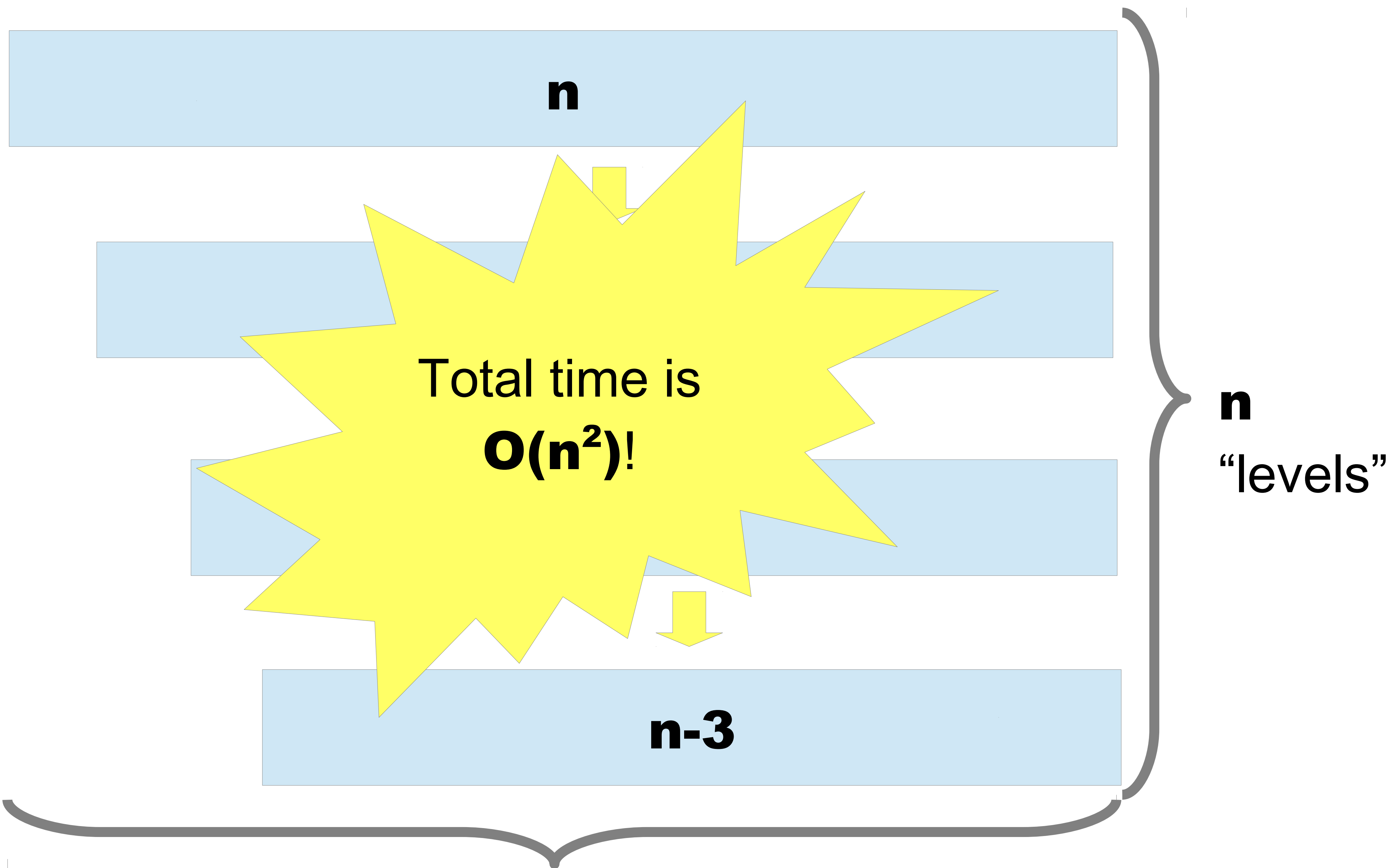
O(n) time per level

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:
  $n + (n-1) + (n-2) + \ldots + 1 = \mathbf{O(n^2)}$


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

Worst cases

When we simply use 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**

Complexity of quicksort

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

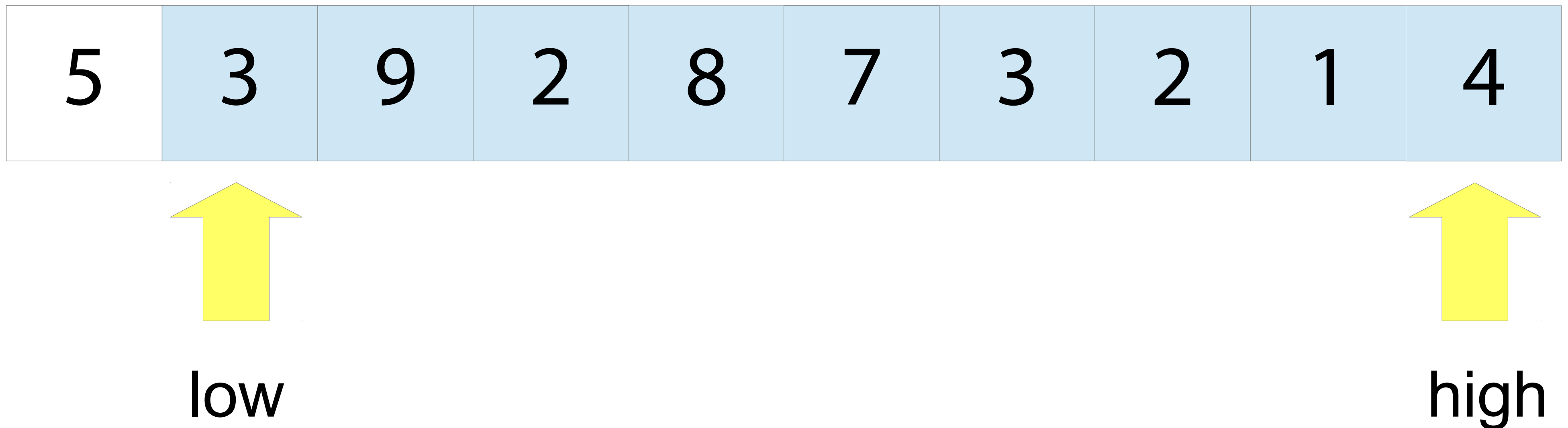
Partitioning algorithm

1. Pick a pivot (here 5)

5	3	9	2	8	7	3	2	1	4

Partitioning algorithm

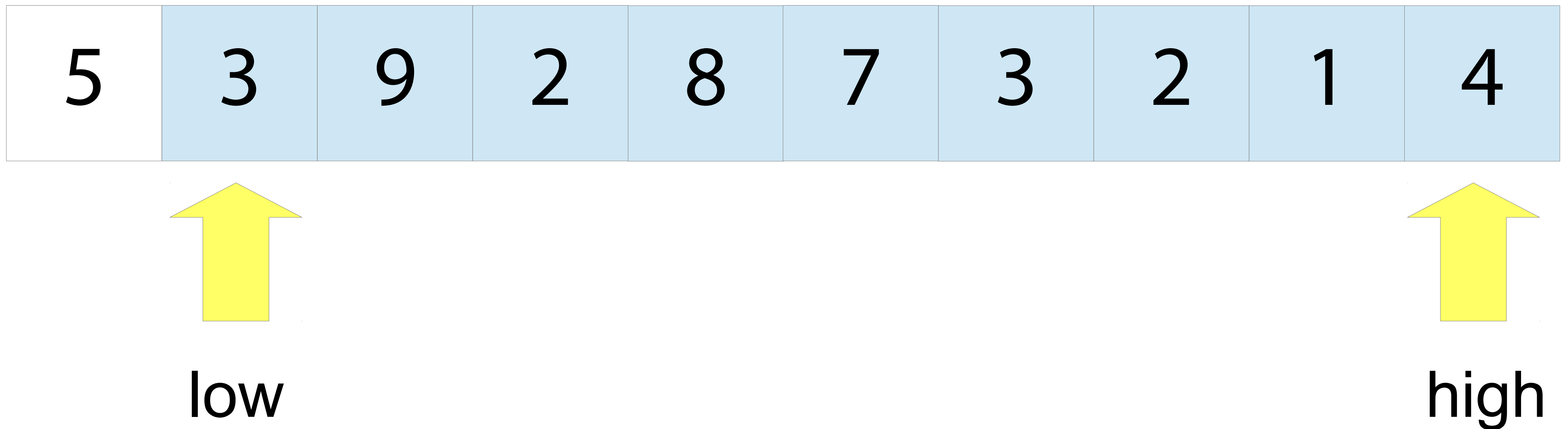
2. Set two indexes, low and high



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 (green)

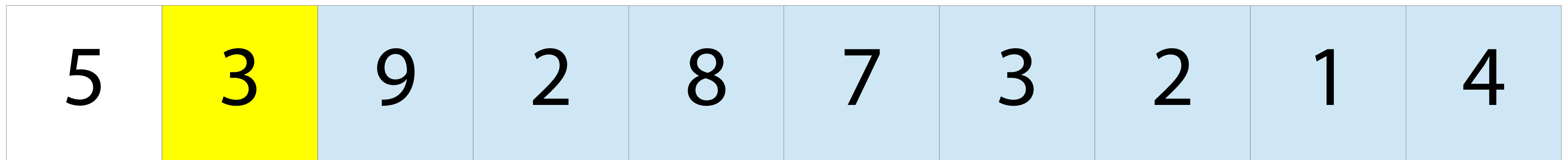
Partitioning algorithm

3. Move low right until you find something greater or equal to the pivot



Partitioning algorithm

3. Move low right until you find something greater or equal to the pivot



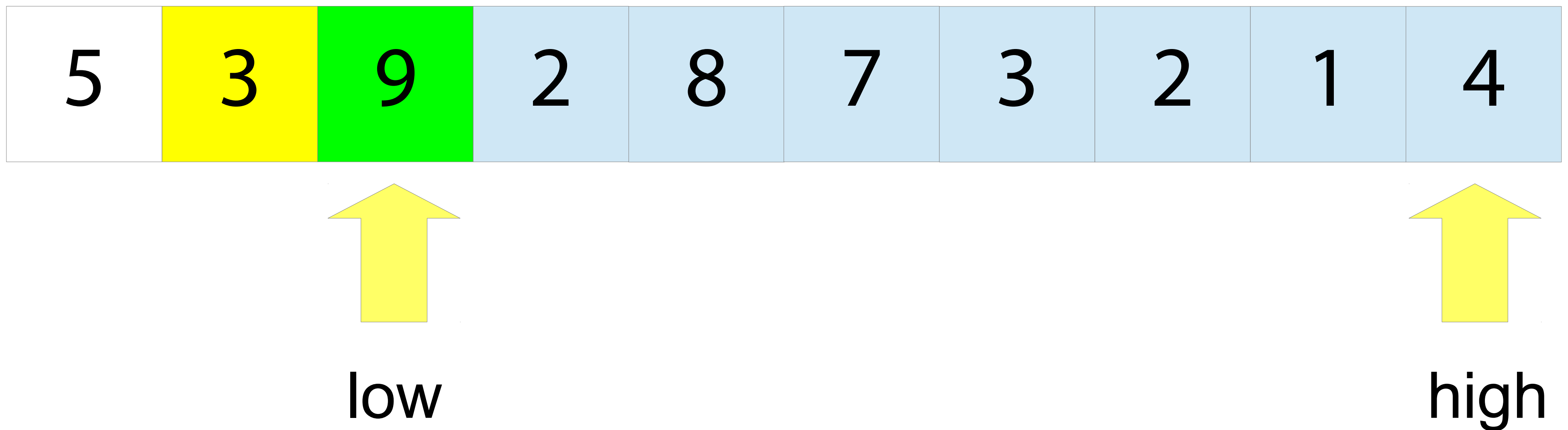
low

high

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

Partitioning algorithm

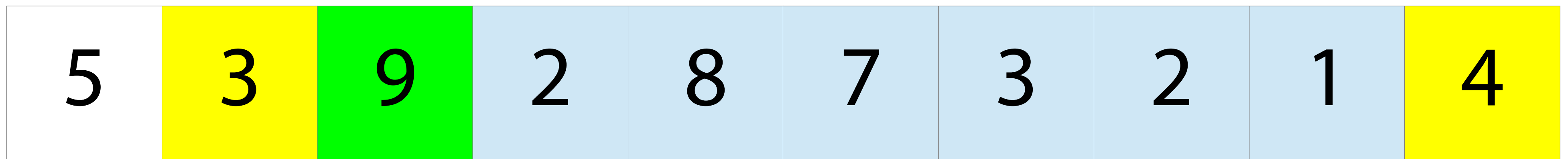
3. Move low right until you find something greater than the pivot

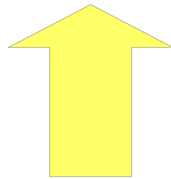


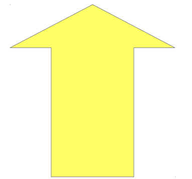

```
while (a[low] < pivot) low++;
```

Partitioning algorithm

3. Move high left until you find something less than the pivot





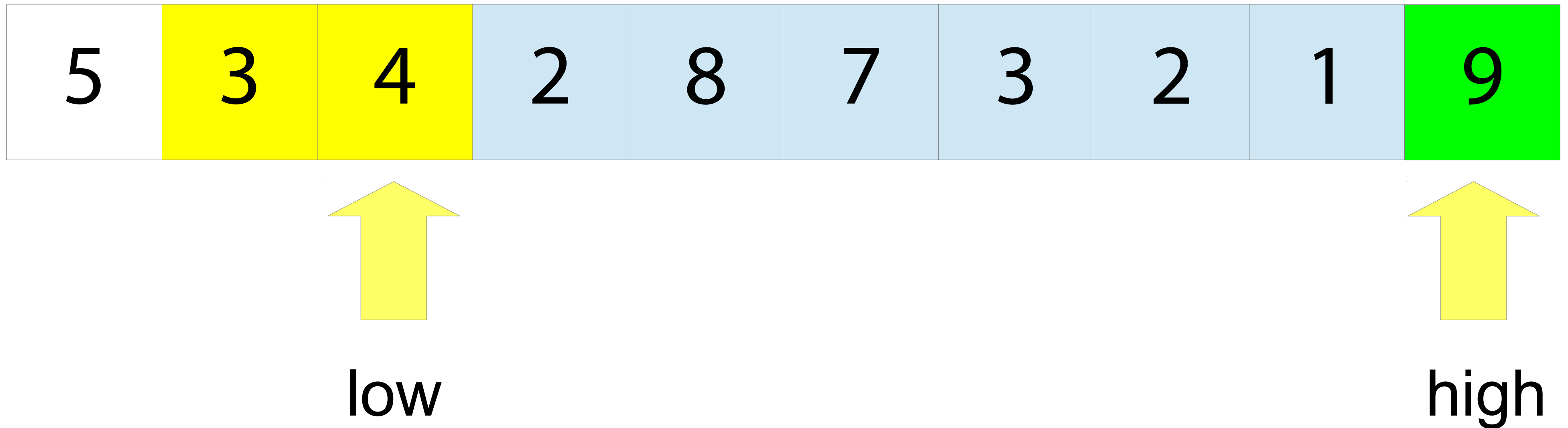


low high

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

Partitioning algorithm

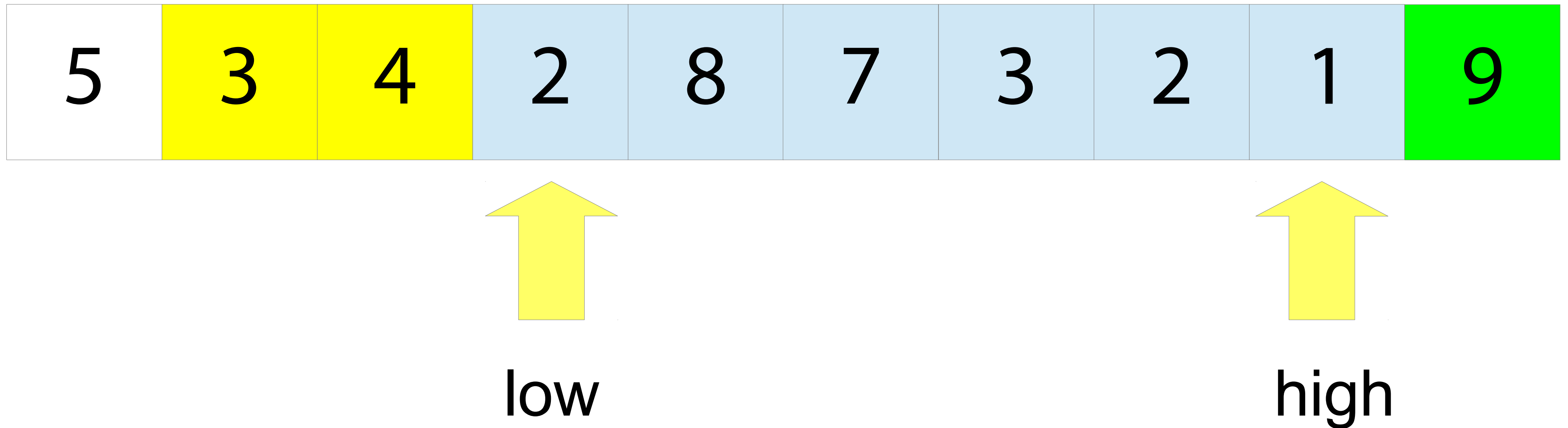
4. Swap them!




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

Partitioning algorithm

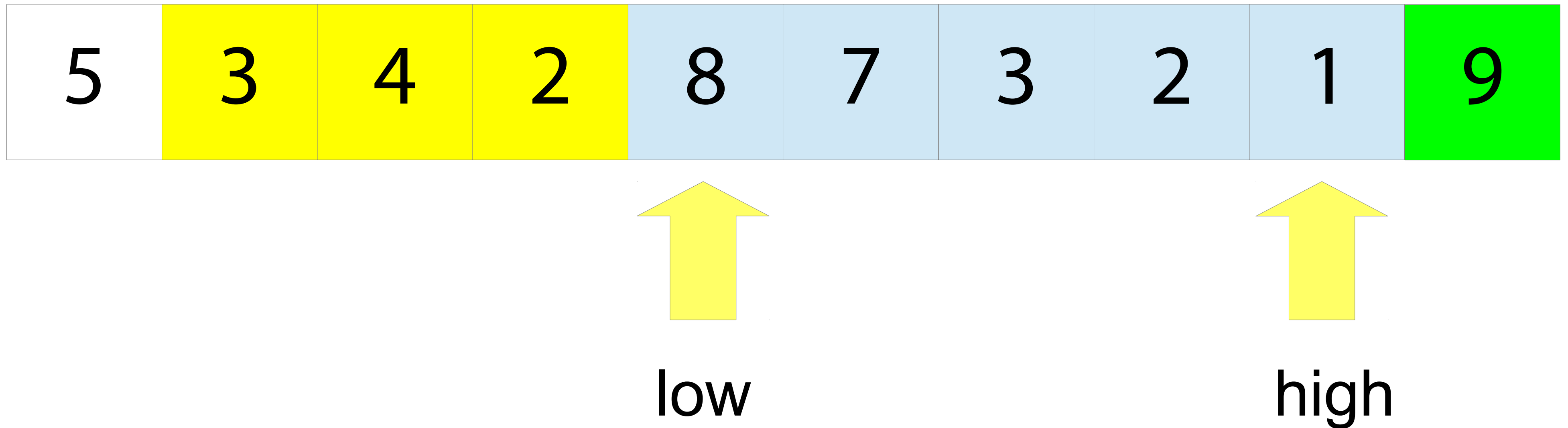
5. Advance low and high and repeat




```
low++; high--;
```

Partitioning algorithm

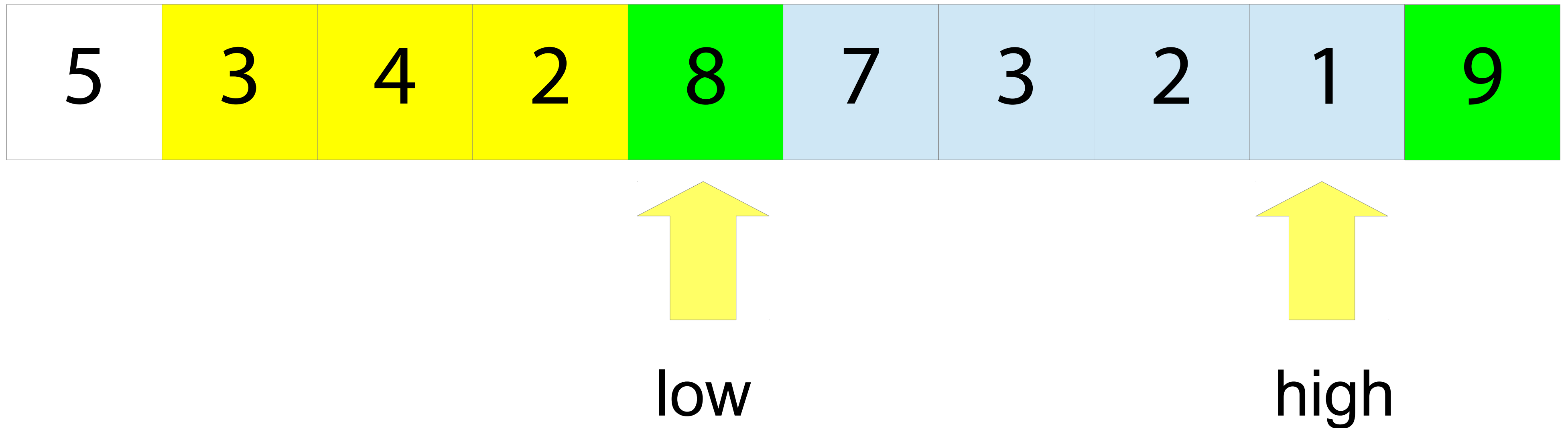
5. Advance low and high and repeat




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

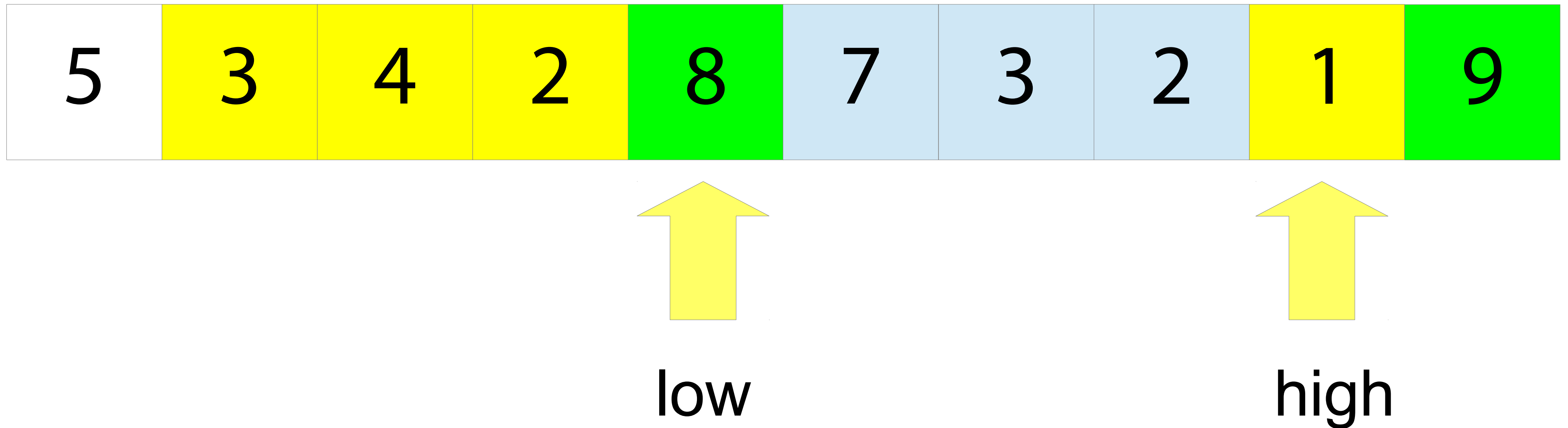
Partitioning algorithm

5. Advance low and high and repeat



Partitioning algorithm

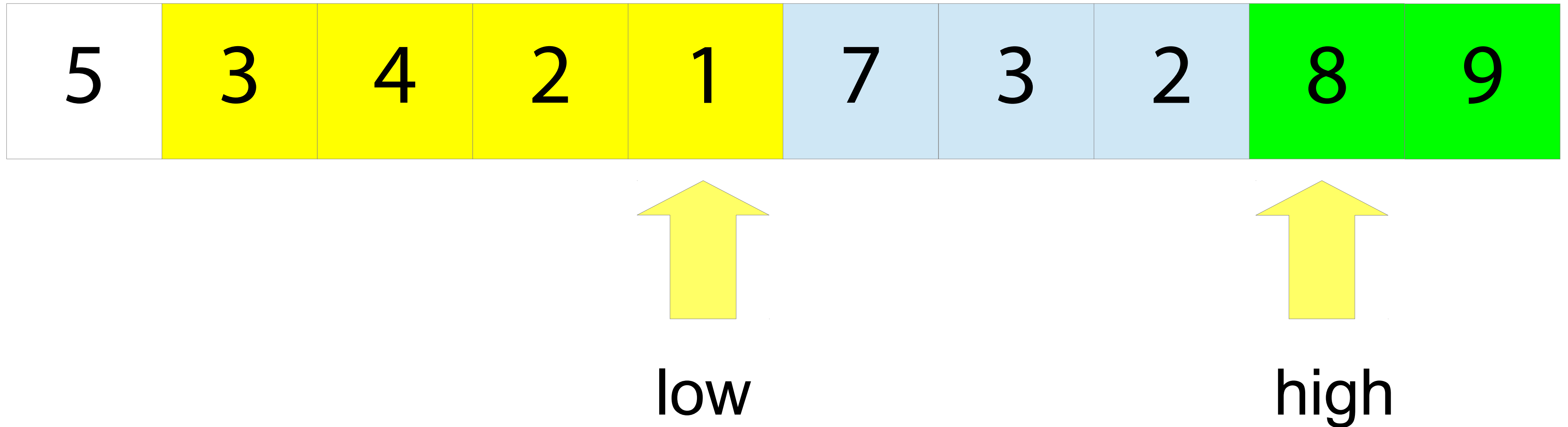
5. Advance low and high and repeat




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

Partitioning algorithm

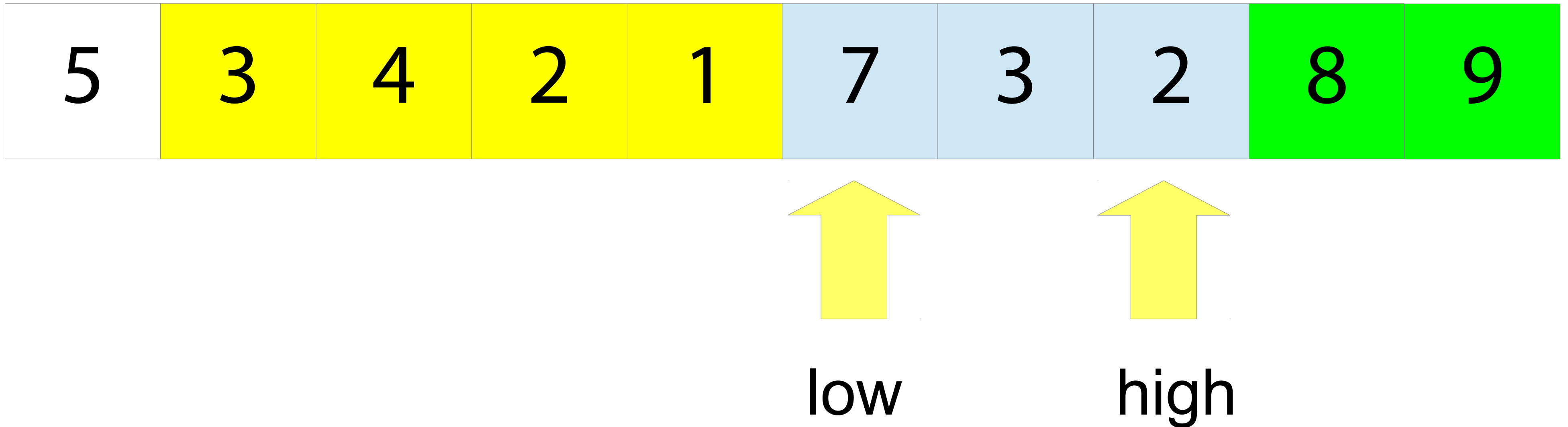
5. Advance low and high and repeat




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

Partitioning algorithm

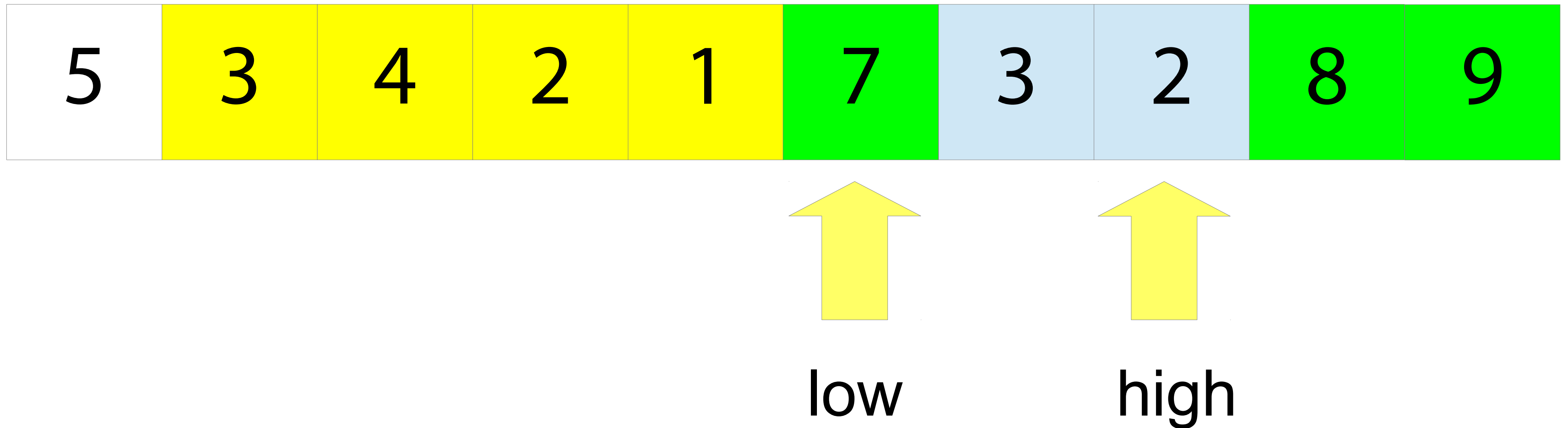
5. Advance low and high and repeat




```
low++; high--;
```

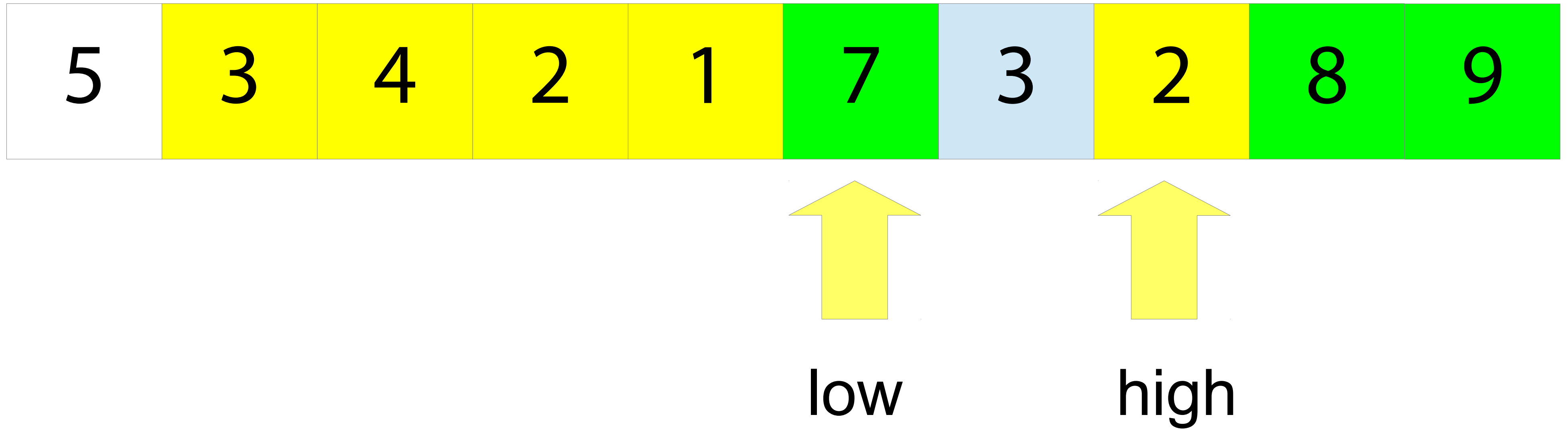
Partitioning algorithm

5. Advance low and high and repeat



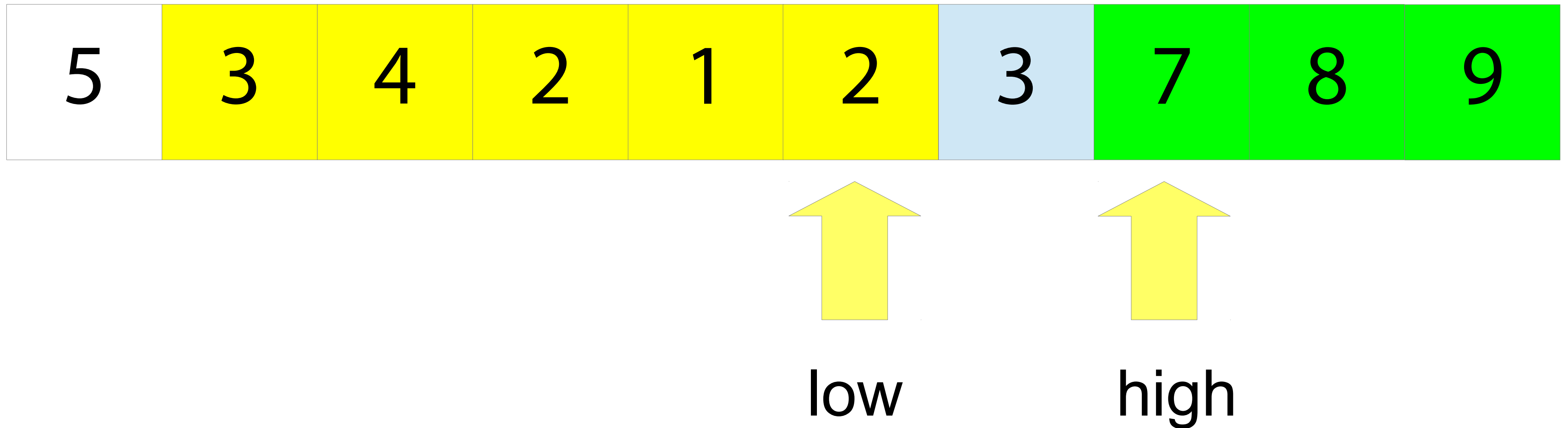
Partitioning algorithm

5. Advance low and high and repeat



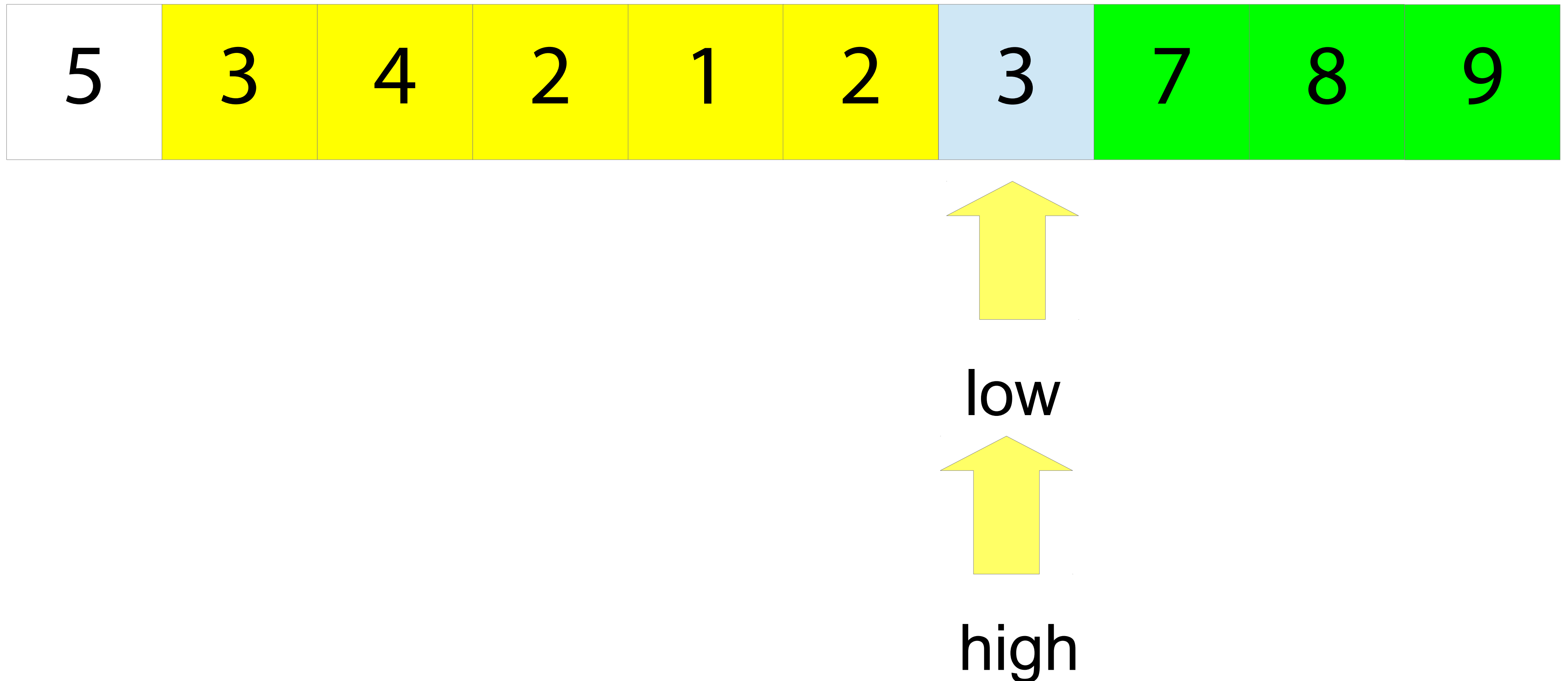
Partitioning algorithm

5. Advance low and high and repeat



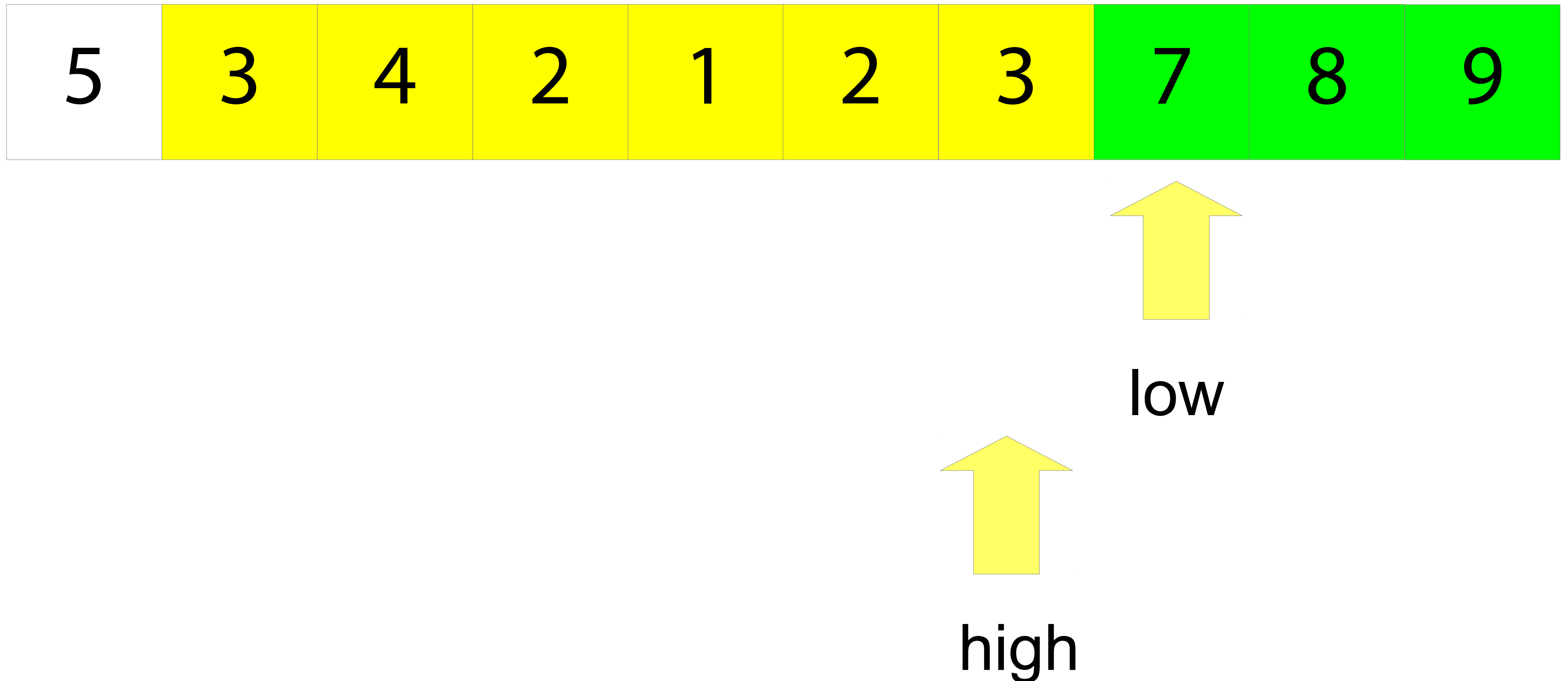
Partitioning algorithm

5. Advance low and high and repeat



Partitioning algorithm

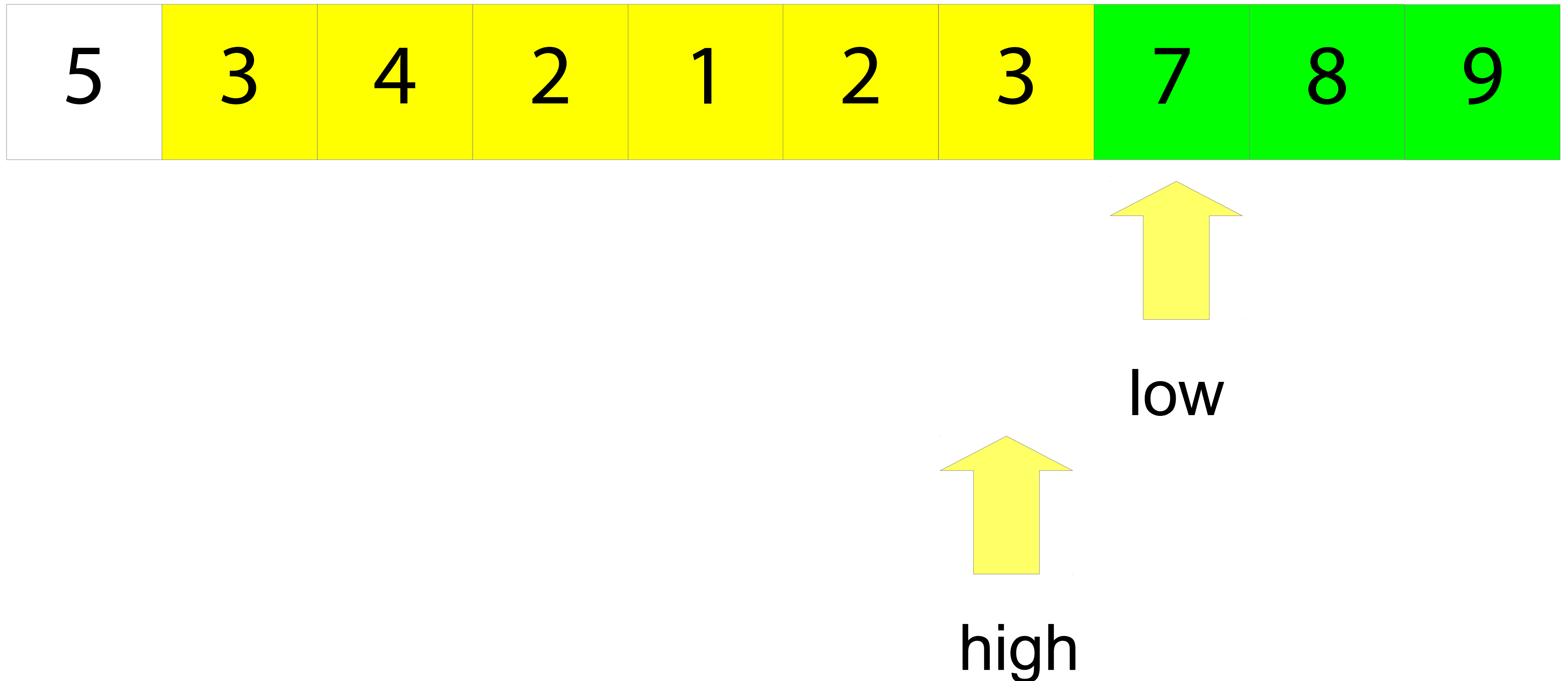
5. Advance low and high and repeat



Partitioning algorithm

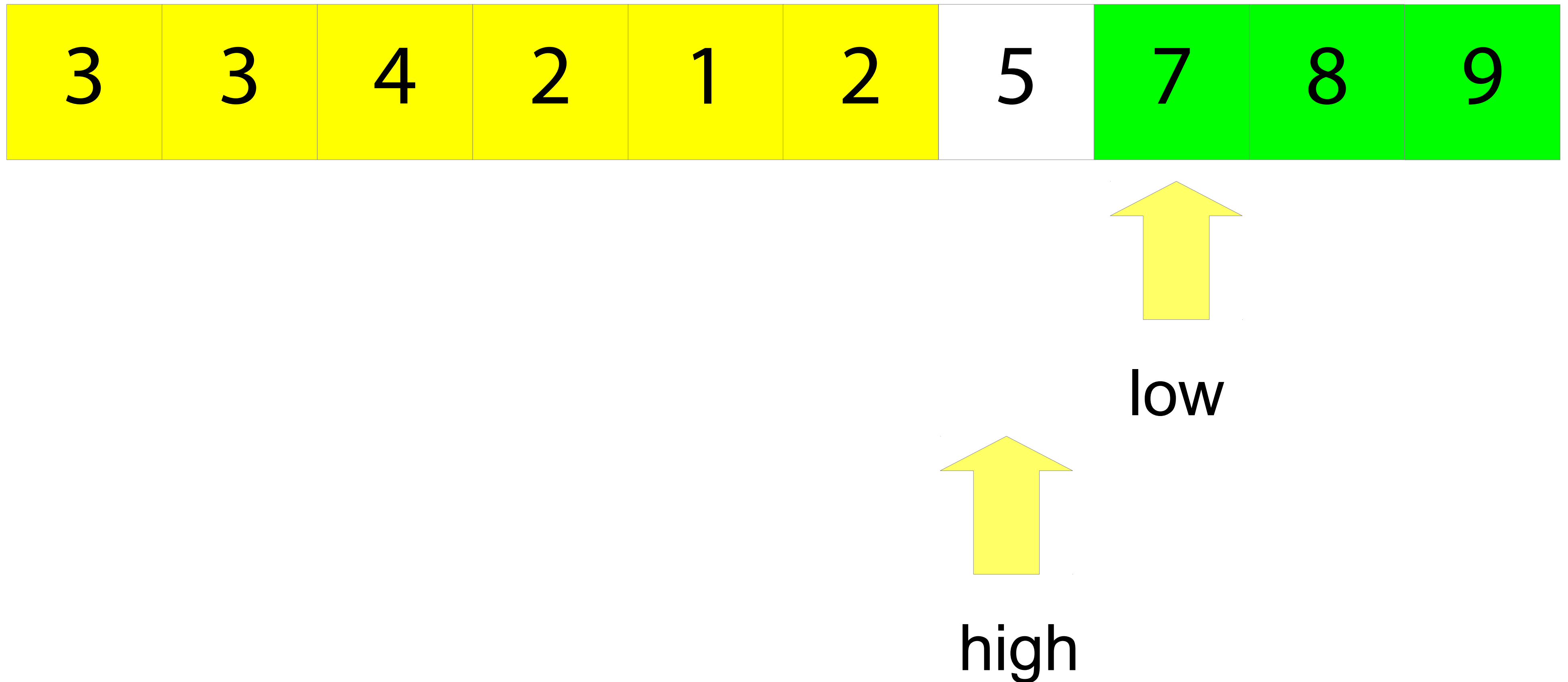
6. When low and high have crossed, we are finished!

But the pivot is in the wrong place.



Partitioning algorithm

7. Last step: swap pivot with high



Details

1. What to do if we want to use a different element (not the first) for the pivot?

- Swap the pivot with the first element before starting partitioning!

Details

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

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).

Merge sort vs quicksort

Quicksort:

- In-place
- O(n log n) but O(n^2) if you are not careful
- Works on arrays only (random access)
- Not stable

Compared to mergesort:

- Not in-place
- O(n log n)
- Only requires sequential access to the list – this makes it good in functional programming
- Stable

Sorting

Why is sorting important? Because sorted data is much easier to deal with!

- Searching – use binary instead of linear search
- Finding duplicates – takes linear instead of quadratic time
- etc.

Most sorting algorithms are based on *comparisons*

- Compare elements – is one bigger than the other? If not, do something about it!
- Advantage: they can work on all sorts of data
- Disadvantage: specialised algorithms for e.g. sorting lists of integers can be faster

Real-world sorting

Sorting algorithms so far

	Worst case	**Average case**	**Best case**
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$
Quicksort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Sorting algorithms so far

	Worst case	Averag ase	Best case
Insertion sort	$O(n^2$		$O(n)$
Quicksort			$(n \log n)$
Mergesort			$O(n \log n)$

No clear winner…
the best algorithms
combine ideas
from several

Introsort

Quicksort: fast in practice, but $O(n^2)$ worst case

Introsort:

- Start with Quicksort
- If the recursion depth gets too big, switch to heapsort, which is $O(n \log n)$

Plus standard Quicksort optimisations:

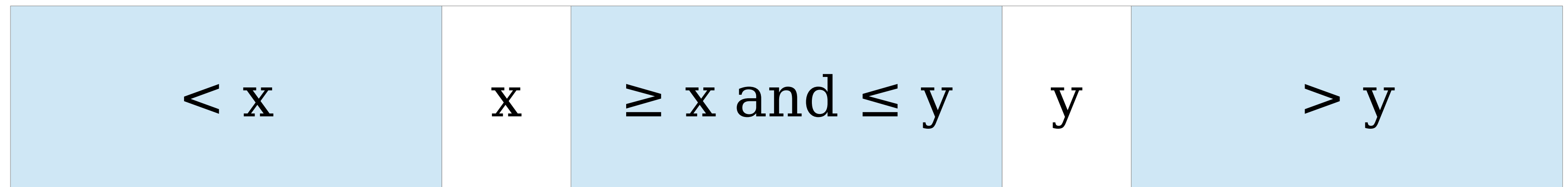
- Choose pivot via median-of-three
- Switch to insertion sort for small arrays

Used by e.g. C++ STL, .NET, ...

Dual-pivot quicksort

Instead of one pivot, pick two, *x* and *y*

Instead of partitioning the array into two halves, partition it into three thirds



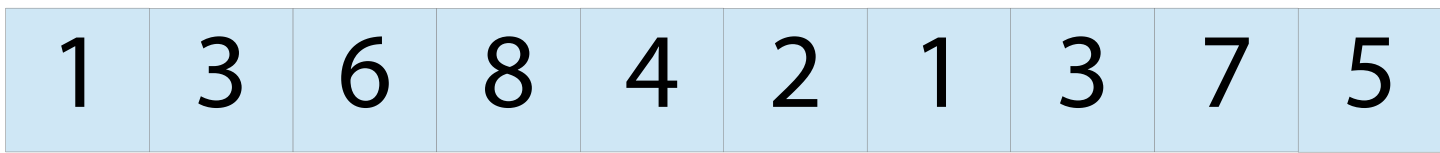
Look up the *Dutch national flag problem* if you want to know how

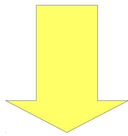
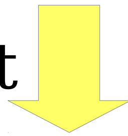
Recursively sort the three partitions!

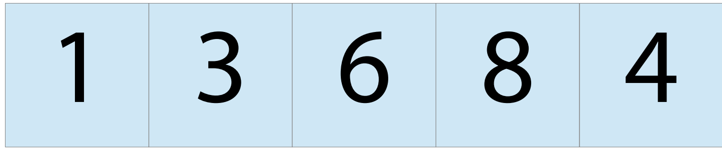
- If the two pivots are equal, don't sort the middle partition (neatly handles the case where the array has a lot of duplicates)

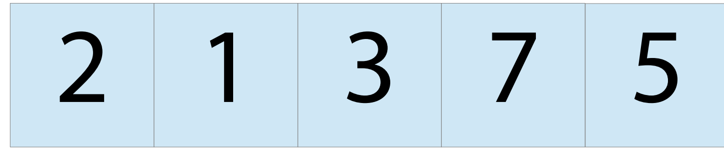
Used by Java for primitive types (int, ...)

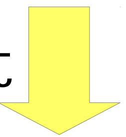
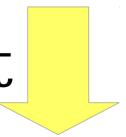
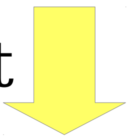
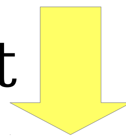
Traditional merge sort
(a recap)

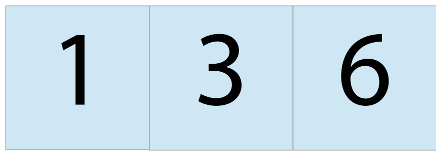


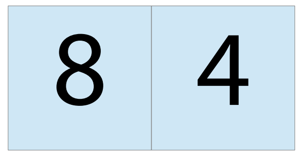
split

split


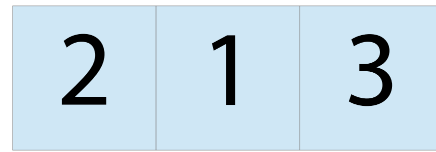


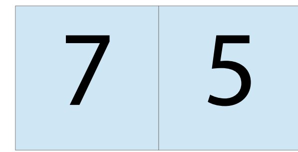


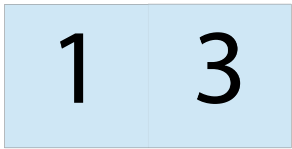
split

split

split

split


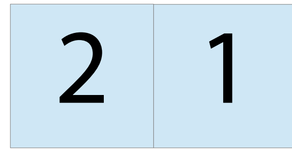


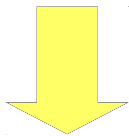
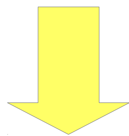
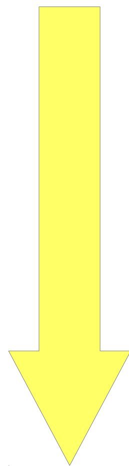
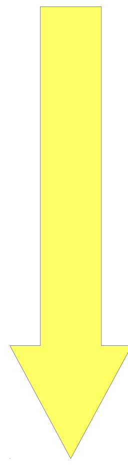
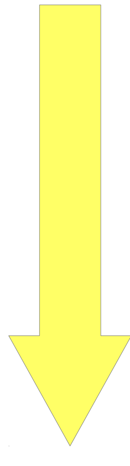
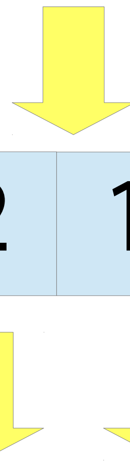


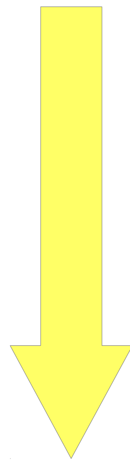
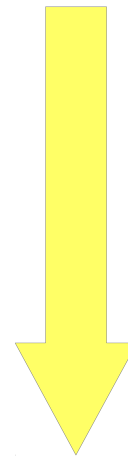
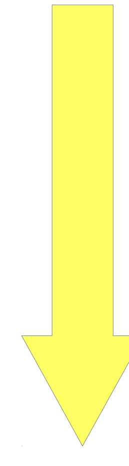


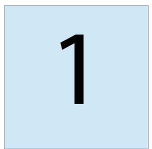


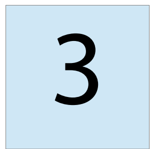


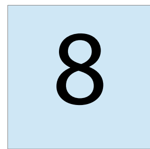


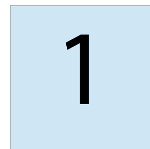







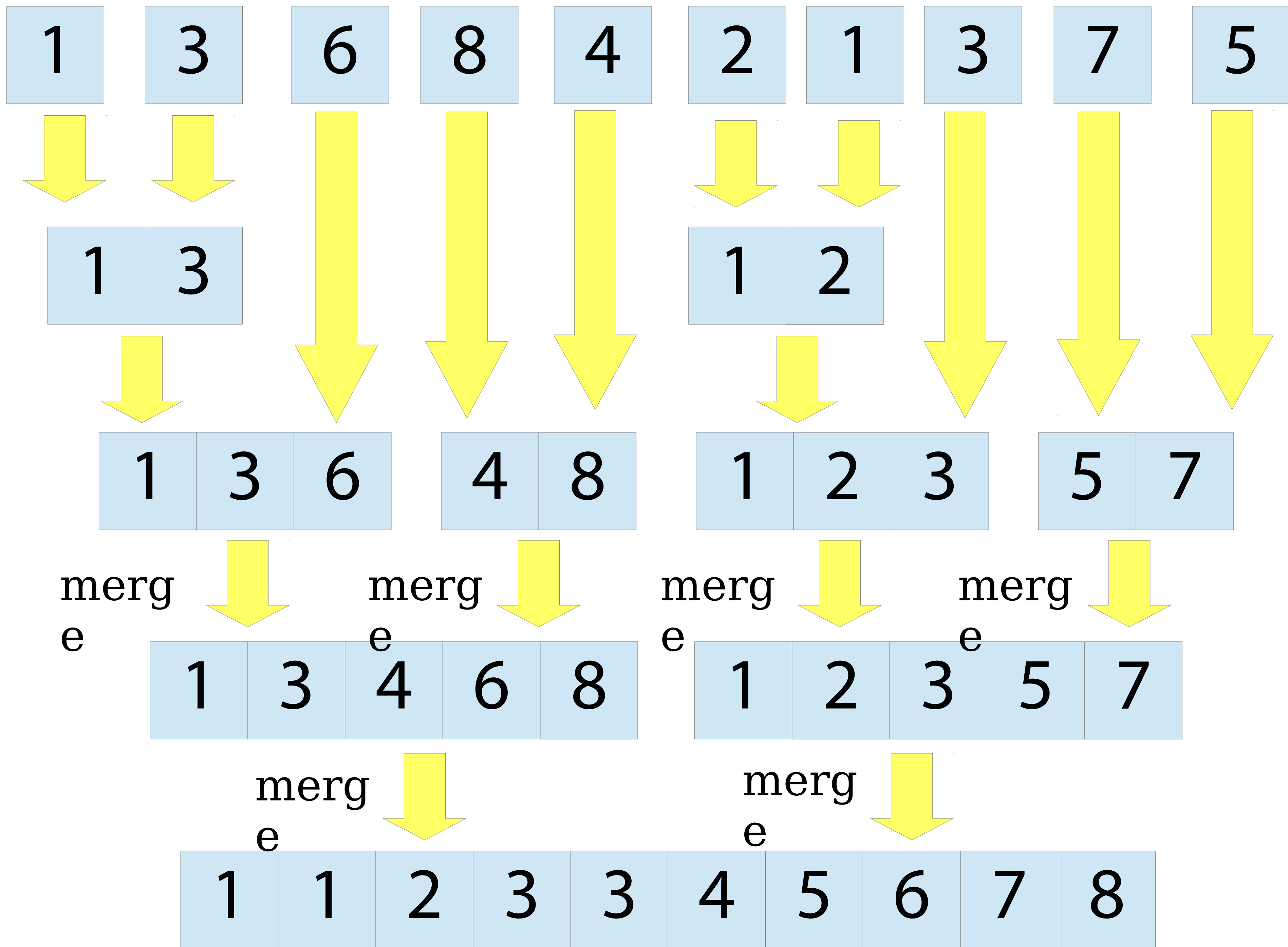












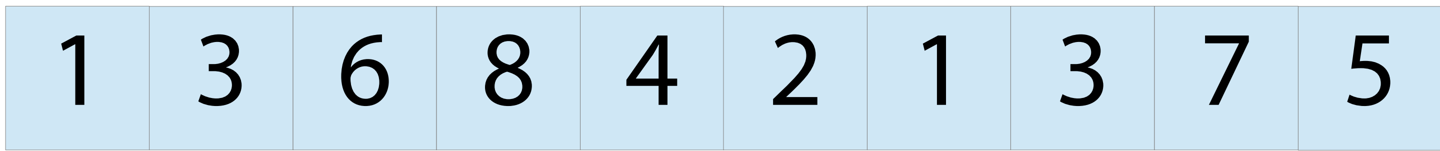

1 3 6 8 4 2 1 3 7 5
1 3
1 2
1 3 6
4 8
1 2 3
5 7
merg
e
merg
e
merg
e
merg
e
1 3 4 6 8
1 2 3 5 7
merg
e
merg
e
1 1 2 3 3 4 5 6 7 8

Natural merge sort

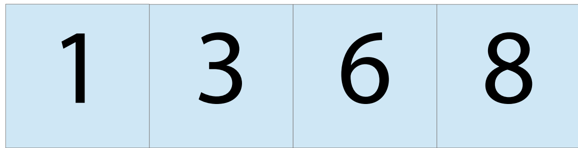
Traditional merge sort splits the input list into *single elements* before merging everything back together again

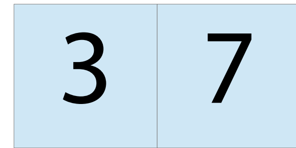
Better idea: split the input into *runs*

- A run is a sequence of increasing elements
- ...or a sequence of decreasing elements
- First reverse all the decreasing runs, so they become increasing
- Then merge all the runs together



split split split split

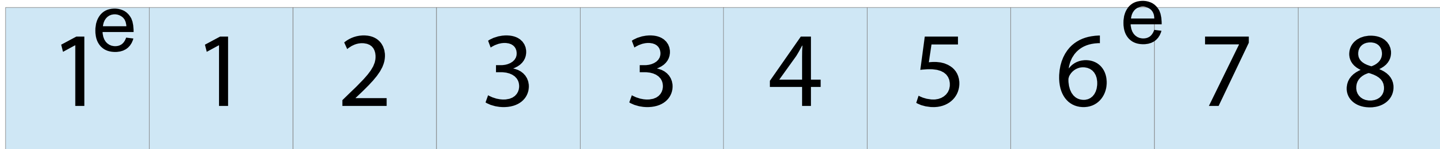




reverse

merge merge merge merge

merge merge



Natural merge sort

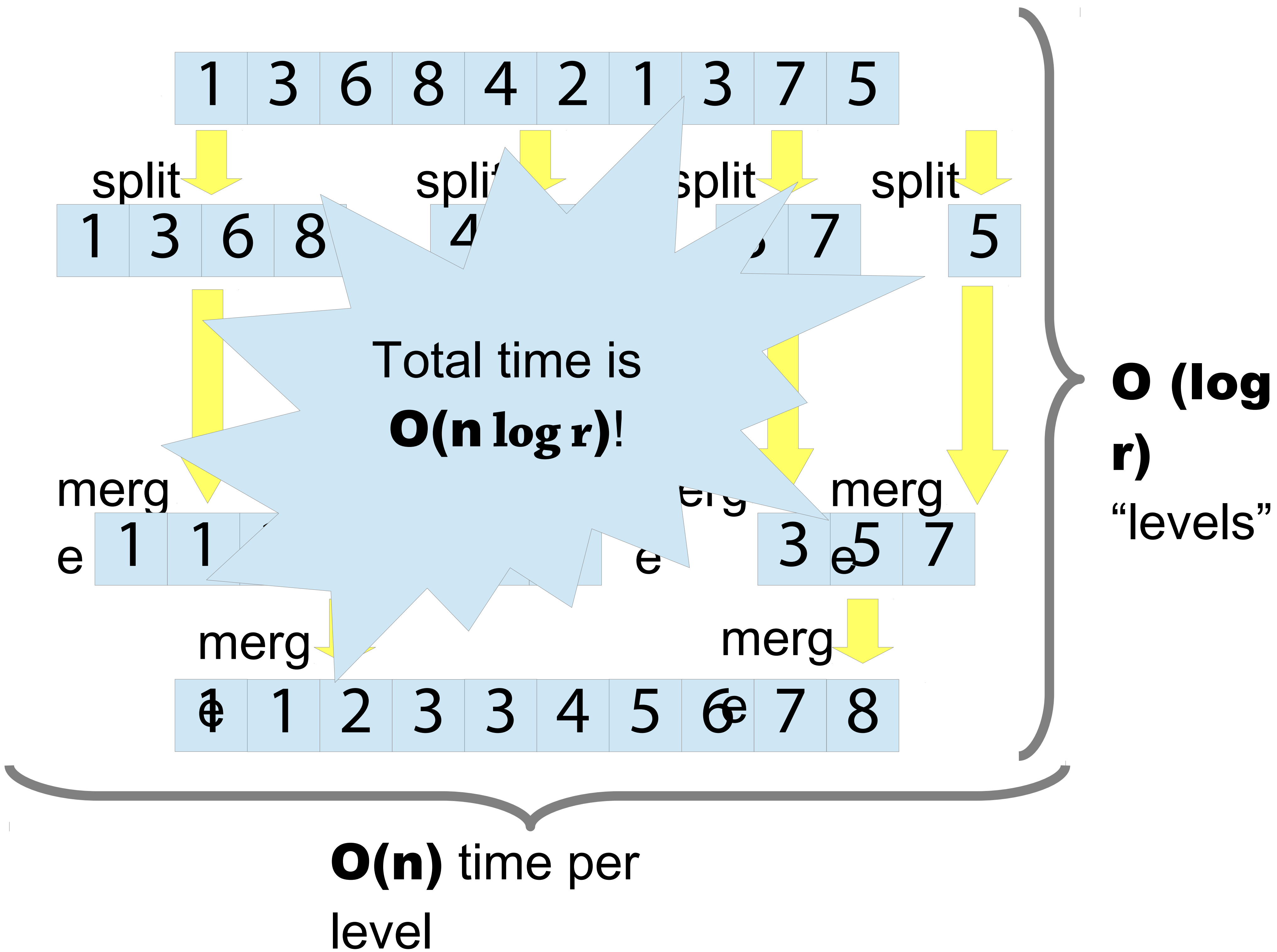
Big advantage: $O(n)$ time for sorted data

- ...and reverse-sorted data
- ...and “almost”-sorted data

Complexity: $O(n \log r)$, where r is the number of runs in the input

- ...worst case, each run has two elements, so $r = n/2$, so $O(n \log n)$

Used by GHC


1 3 6 8 4 2 1 3 7 5
split
split
split
split
1 3 6 8
5
7
Total time is
O(n log r)!
merg
e
merg
e
merg
e
3 5 7
merg
e
merg
e
1 1
1 1 2 3 3 4 5 6 7 8
O (log r)
"levels"
O(n) time per level

Timsort

Natural mergesort is really good on sorted/nearly-sorted data

- You get long runs so not many merges to do

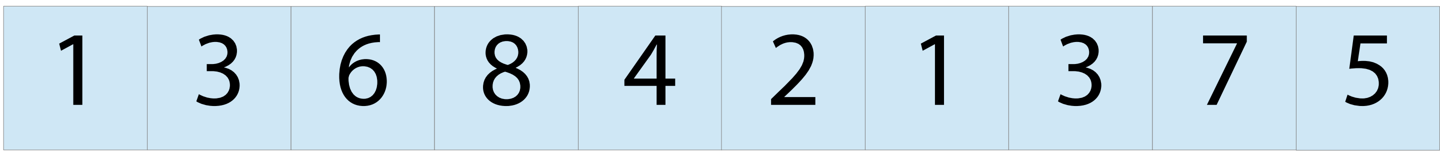
But not so good on highly unsorted data

- Short runs so many merges

Idea of Timsort: on short, very unsorted parts of the list, *switch to insertion sort*

How to detect unsortedness?

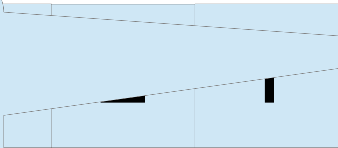
- Several short runs next to one another



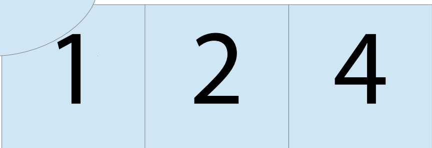
split

split

split



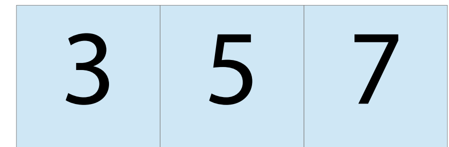
Don't split 3 7 5 into two runs, it's too short!



merge

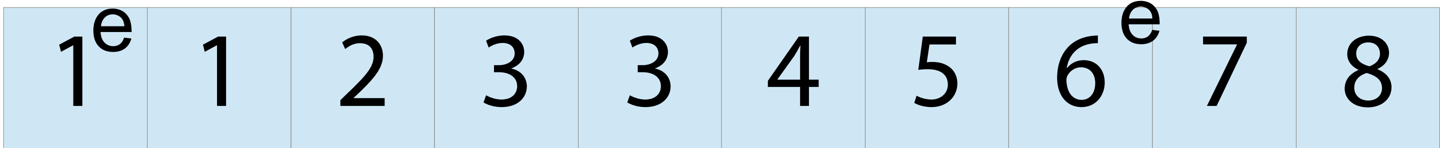
merge

insertion sort



merge

merge



Timsort

Specifically:

- If we come across a short run, join it together with all following short runs until we reach a threshold
- Then use insertion sort on that part

Also some optimisations for merge:

- Merge smaller runs together first
- If the merge begins with several elements from the same array, use binary search to find out how many and then copy them all in one go

Used in Java for arrays of objects, Python

http://en.wikipedia.org/wiki/Timsort

The best sorting algorithm?

A good sorting algorithm should:

- have O(n log n) complexity
- have O(n) complexity on nearly-sorted data
- be simple
- be in-place
- be cache-friendly

No algorithm seems to have all of these!

Summary

No one-size fits all answer

- Best overall complexity: natural mergesort
- But quicksort has smaller constant factors
- Timsort a good compromise?

Different algorithms are good in different situations

- ...something you should find in the lab :)

Best sorting functions combine ideas from several algorithms

- Introsort: quicksort+heapsort+insertion sort
- Timsort: natural mergesort+insertion sort