Better sorting algorithms
(Weiss chapter 8.5 – 8.6)

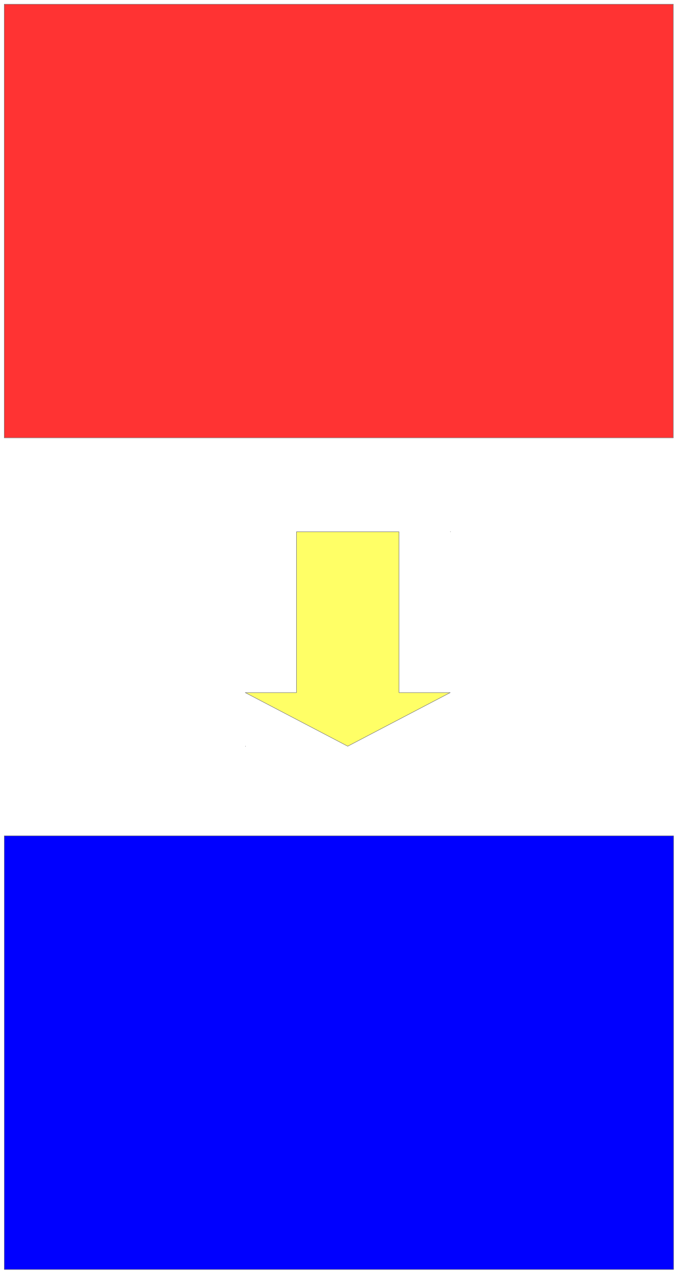
Divide and conquer

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

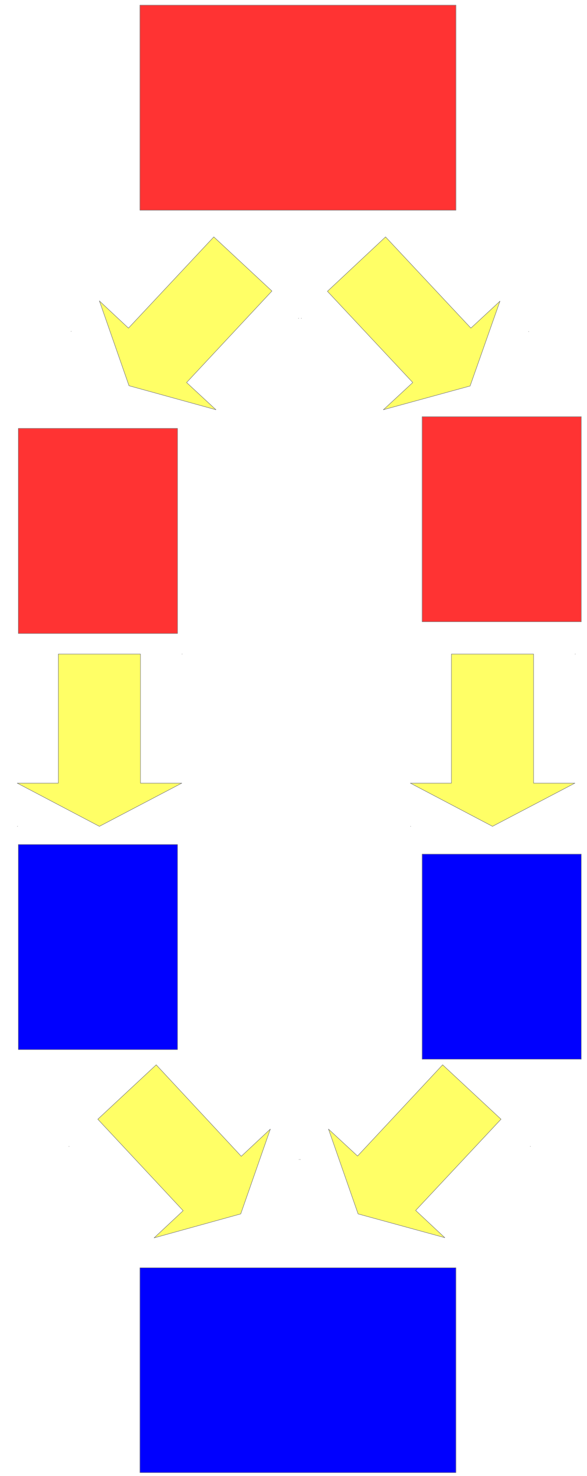
To solve this...



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

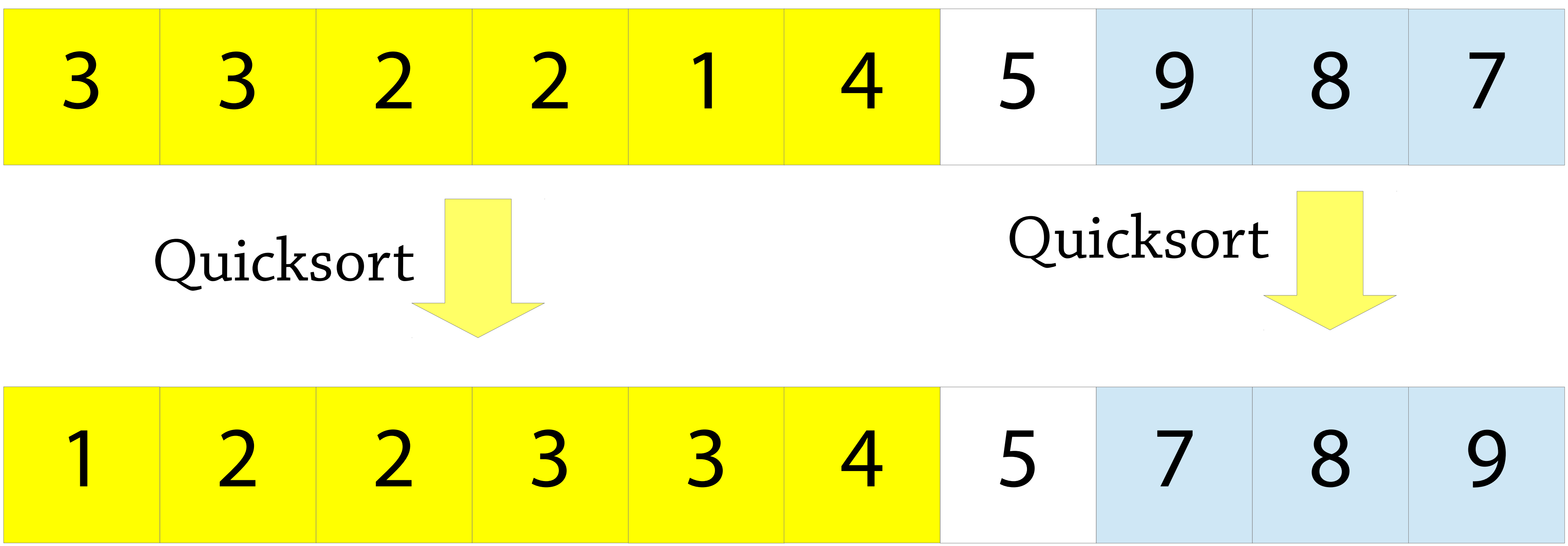
3	3	2	2	1	4	5	9	8	7

Less than the pivot

Greater than the pivot

Quicksort

Now recursively quicksort the two partitions!



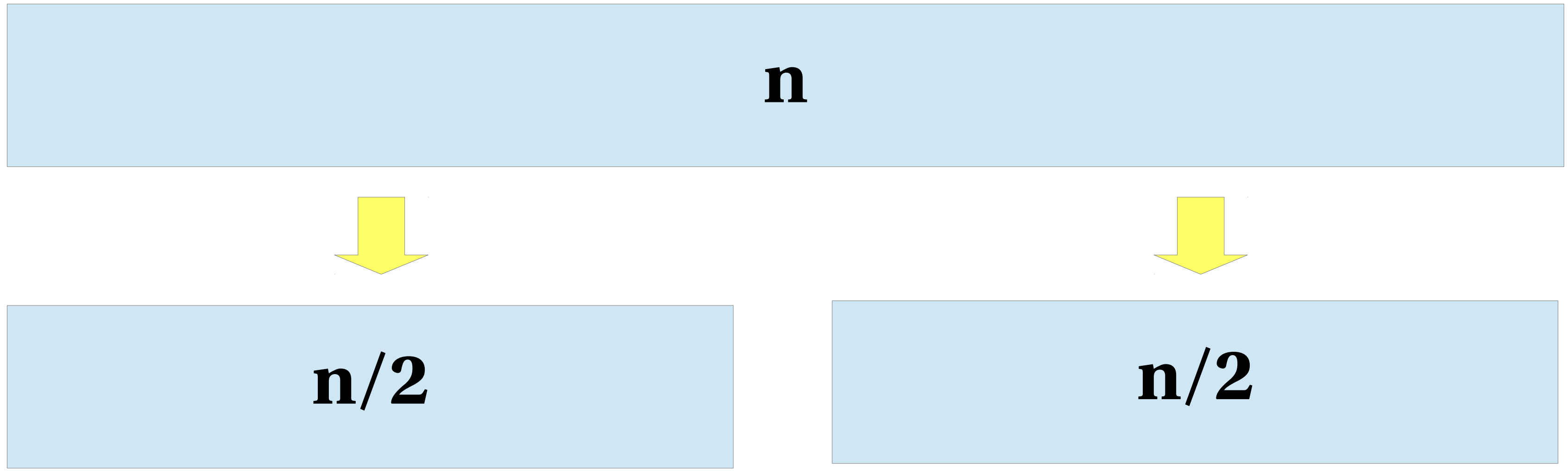
Pseudocode

```
// 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

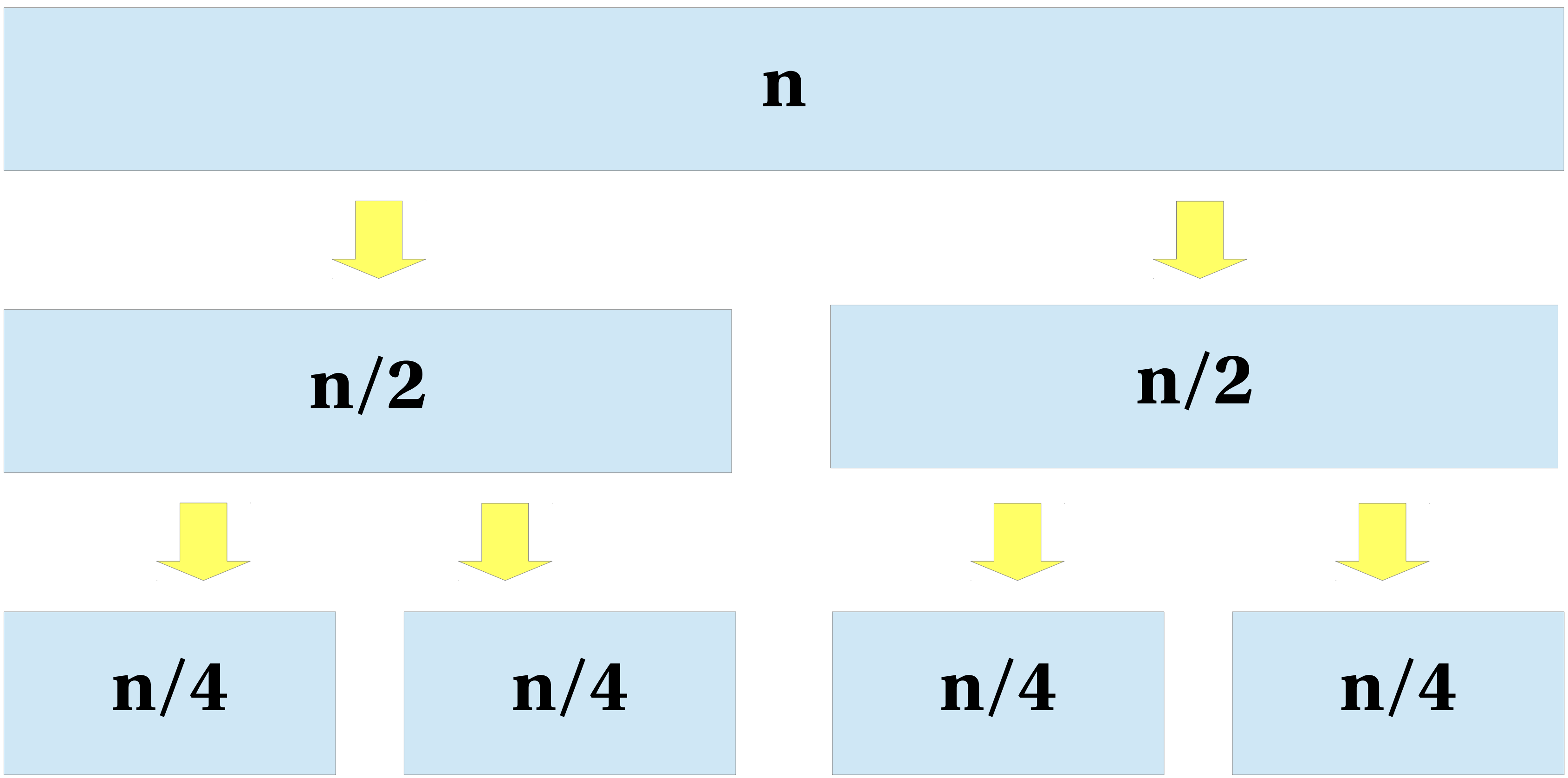
Complexity of quicksort

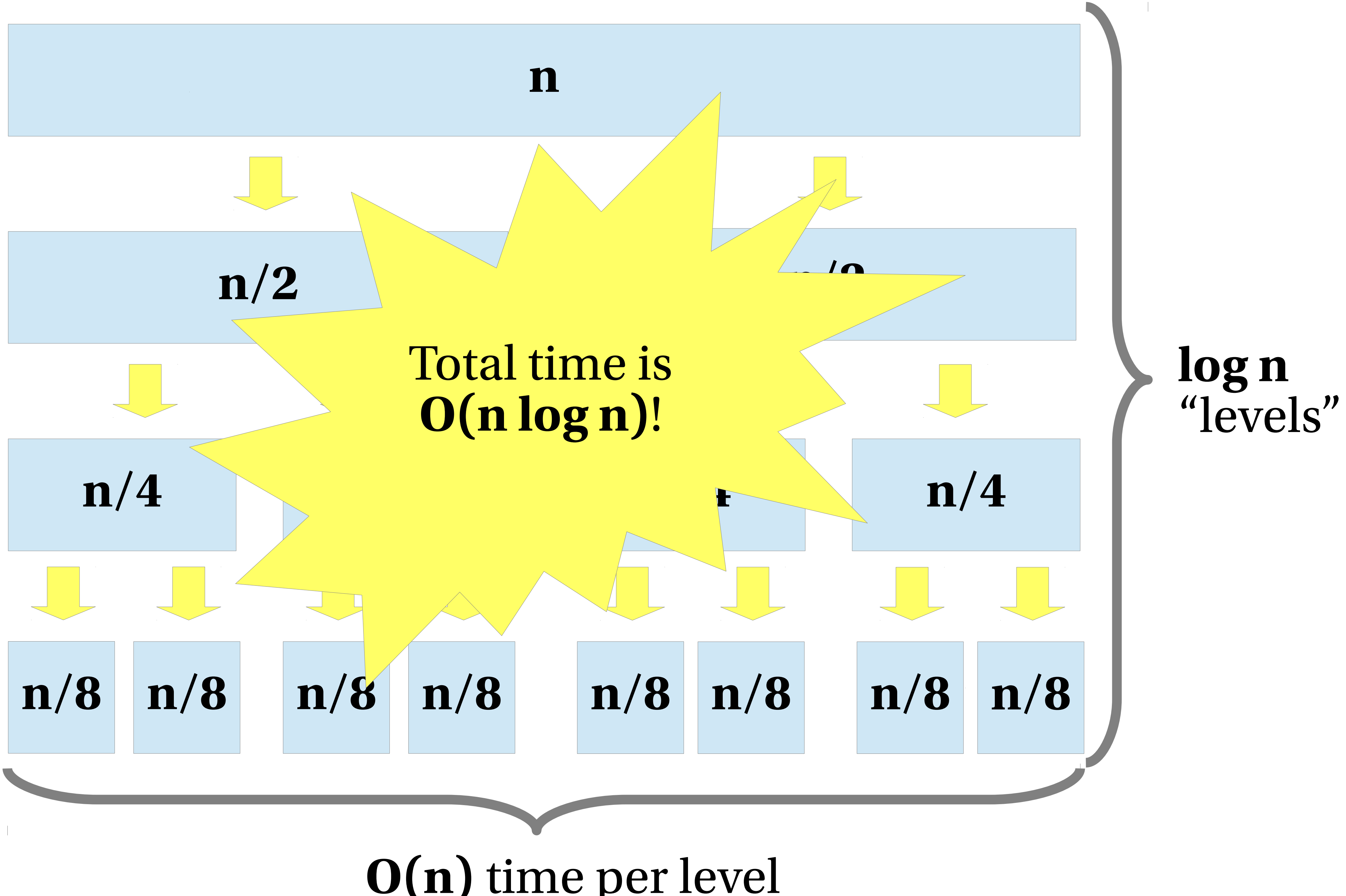
In the best case, partitioning splits an array of size n into two halves of size n/2:



Complexity of quicksort

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



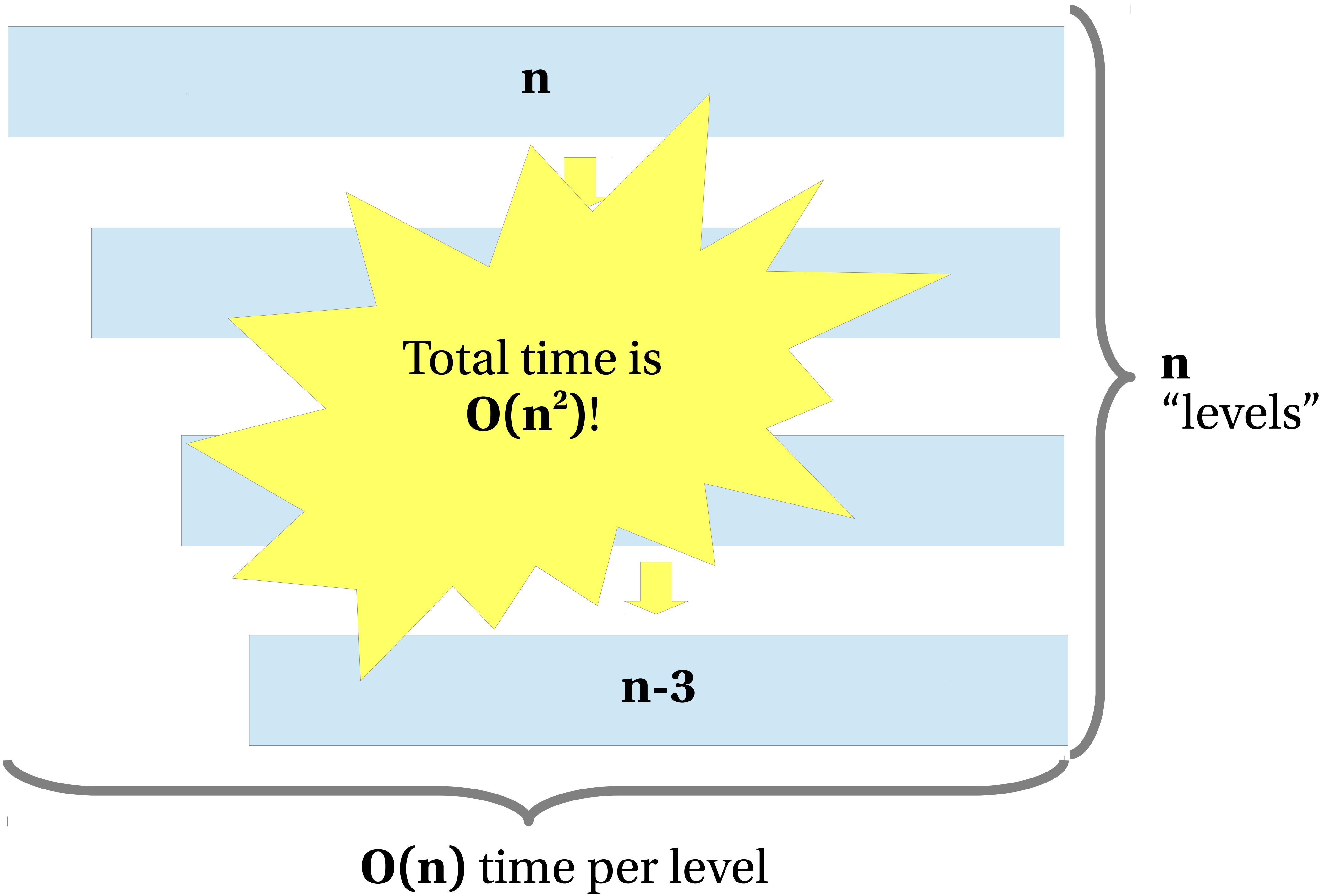

n
n/2
n/4
n/4
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
Total time is
O(n log n)!
log n
"levels"
O(n) time per level

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 pick the first element as the pivot, we get this worst case for:

- Sorted arrays
- Reverse-sorted arrays

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)

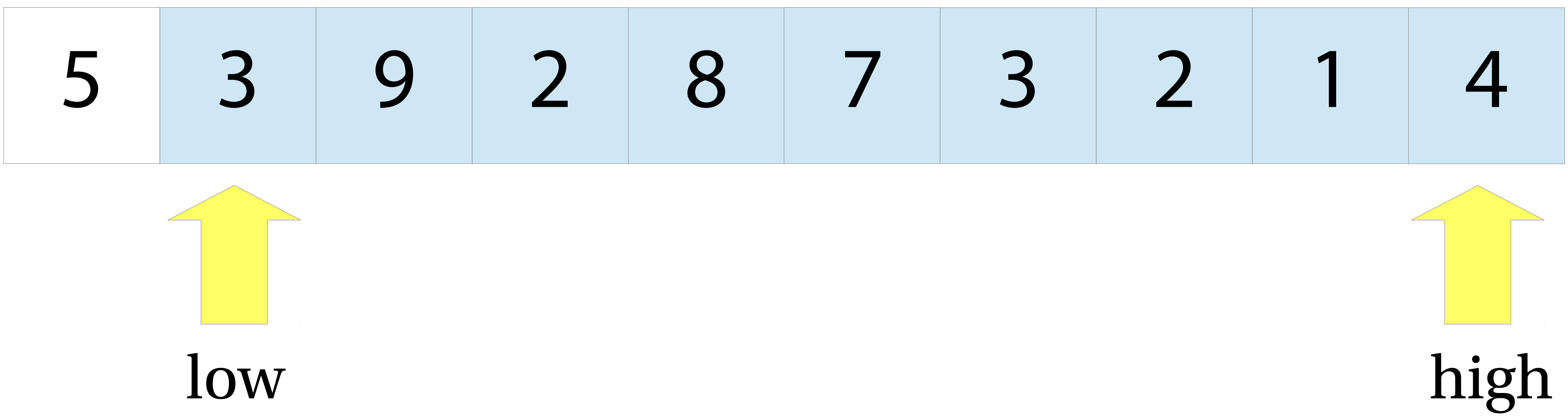
Partitioning algorithm

1. Pick a pivot (here 5)

5	3	9	2	8	7	3	2	1	4

Partitioning algorithm

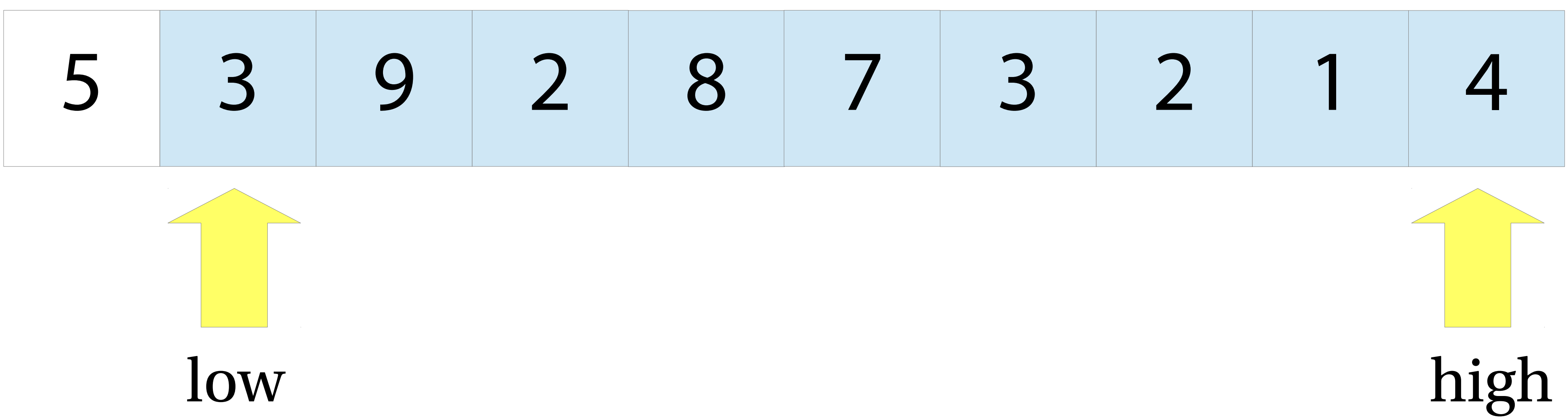
2. Set two indexes, low and high



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 (green)

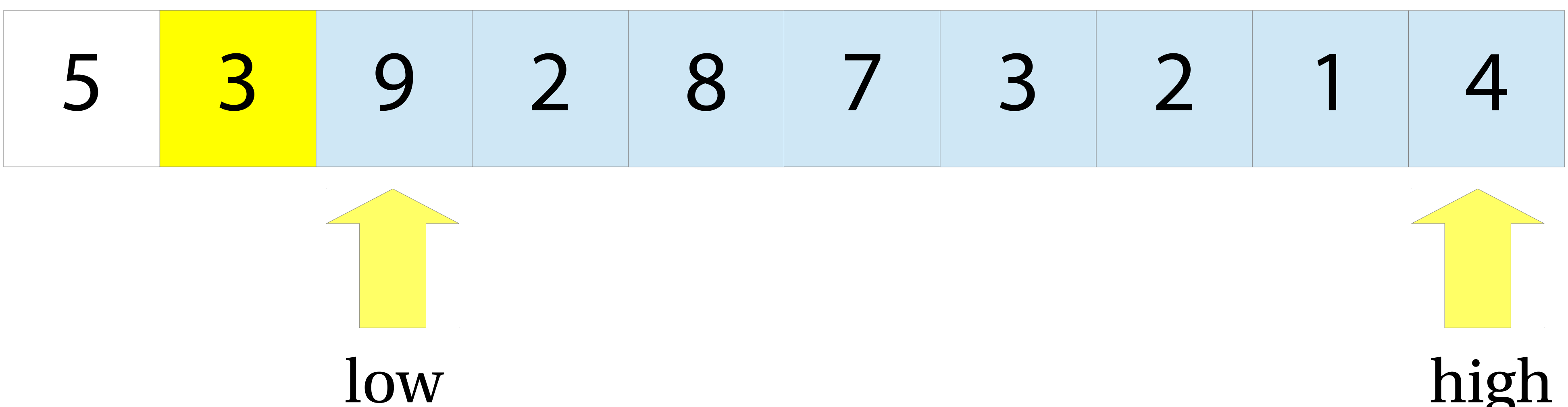
Partitioning algorithm

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



Partitioning algorithm

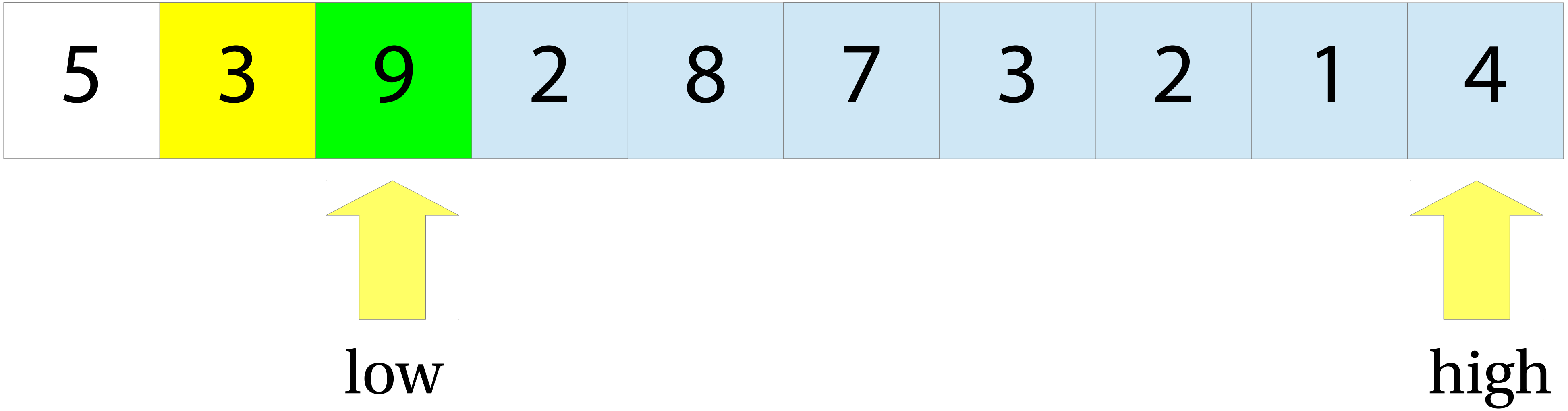
3. Move low right until you find something greater or equal to the pivot




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

Partitioning algorithm

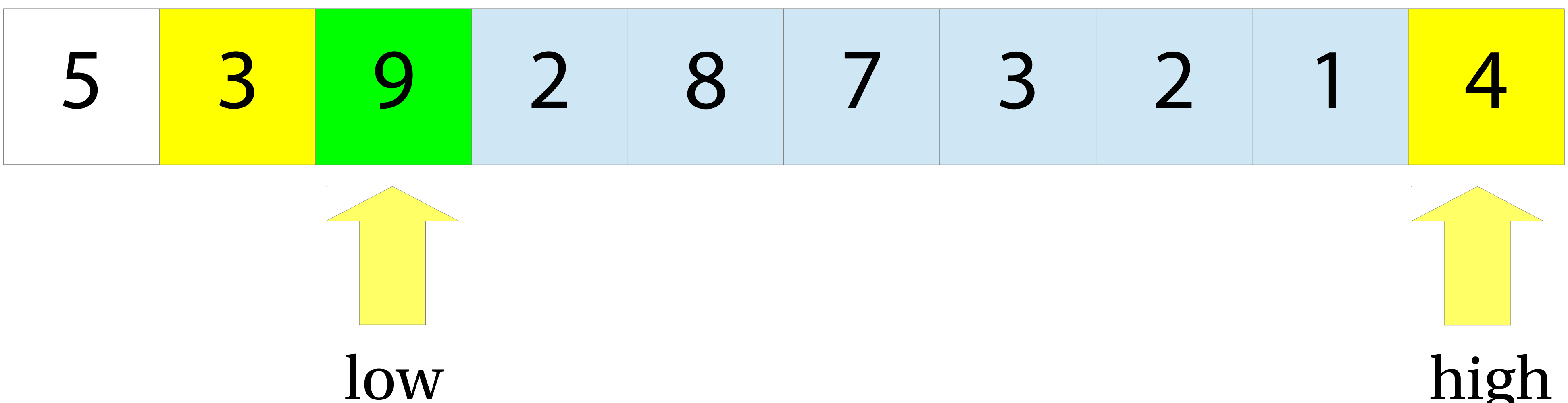
3. Move low right until you find something greater than the pivot




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

Partitioning algorithm

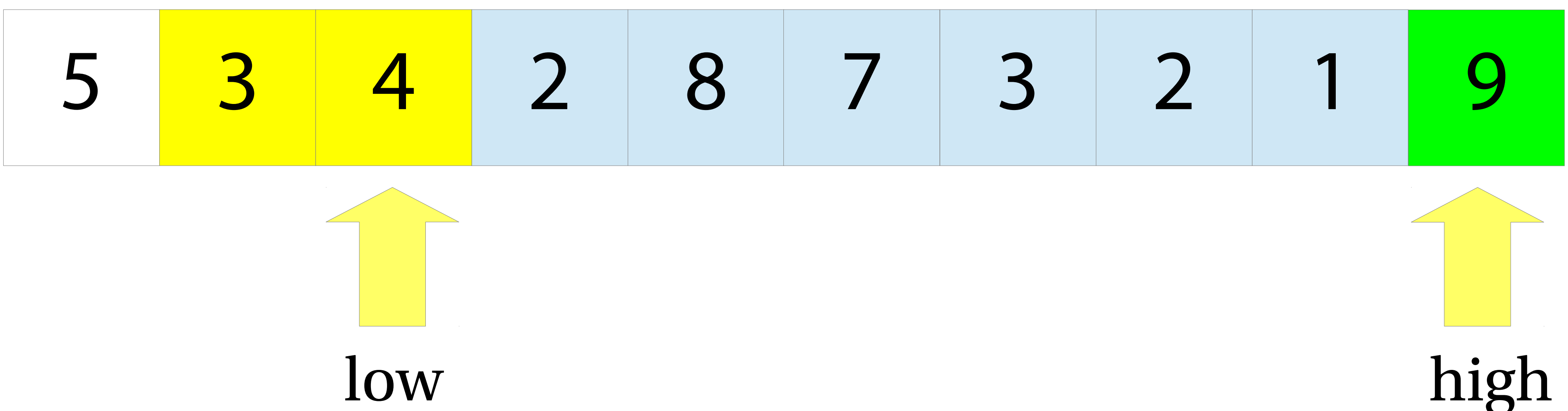
3. Move high left until you find something less than the pivot




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

Partitioning algorithm

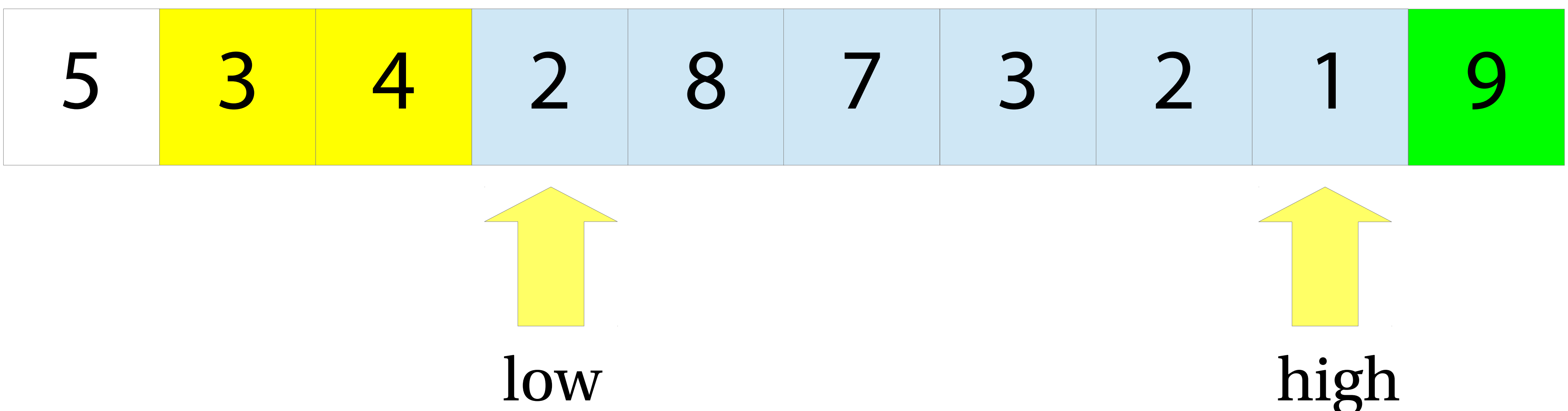
4. Swap them!




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

Partitioning algorithm

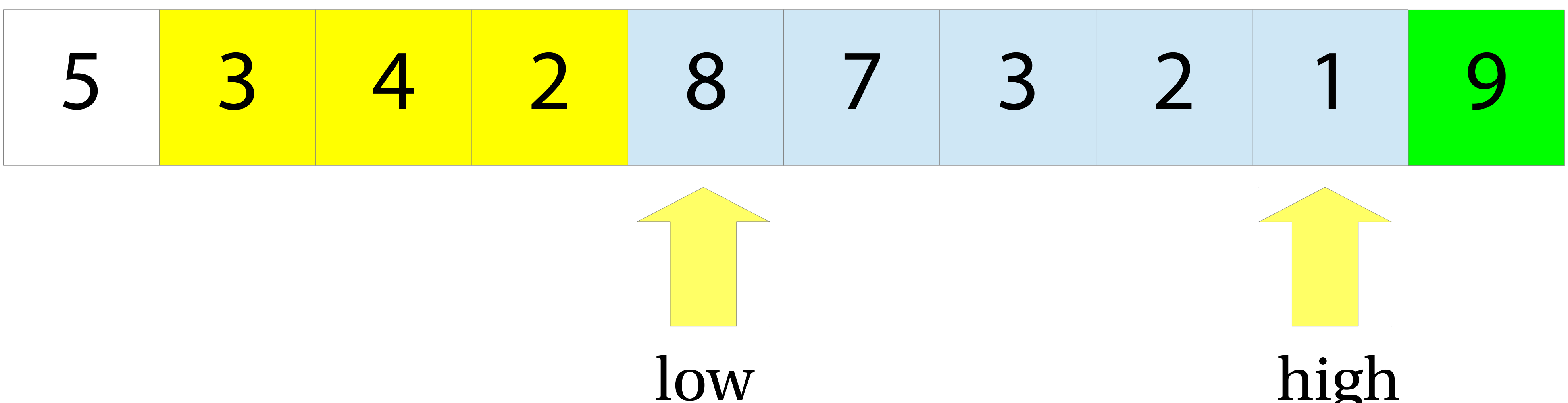
5. Advance low and high and repeat




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

Partitioning algorithm

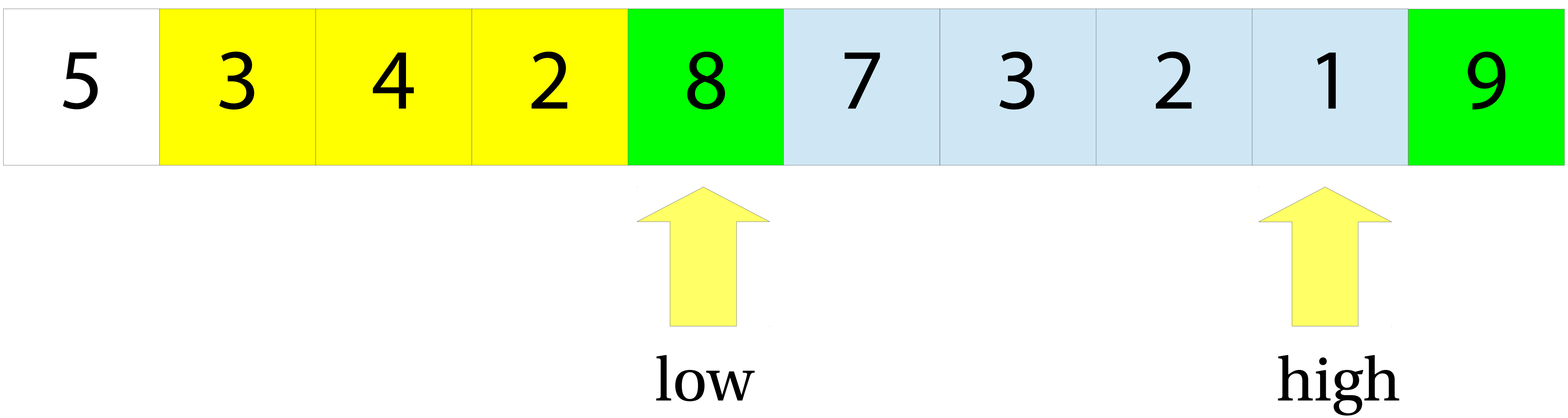
5. Advance low and high and repeat




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

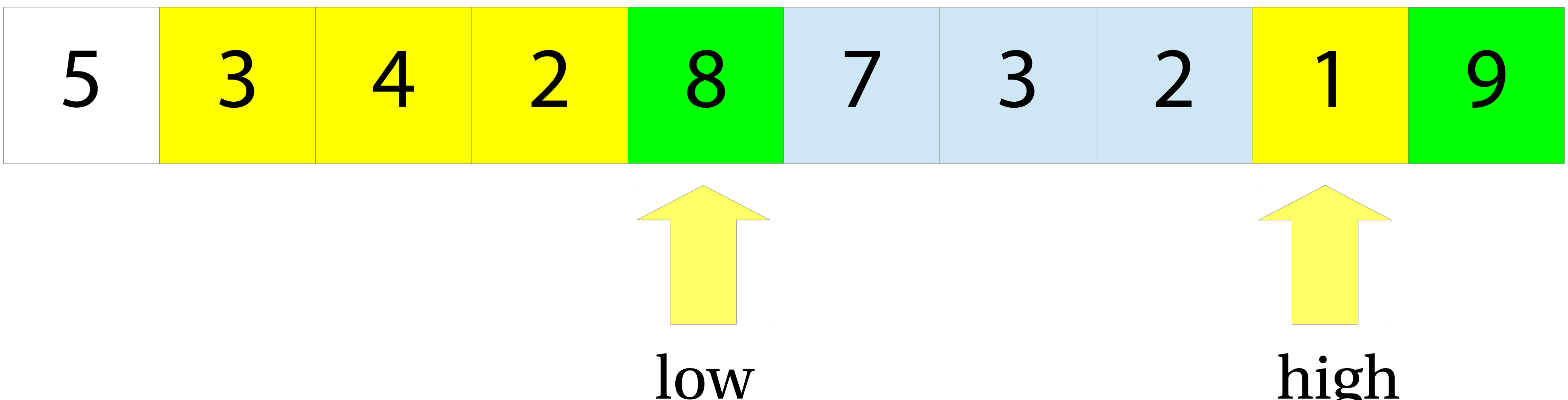
Partitioning algorithm

5. Advance low and high and repeat



Partitioning algorithm

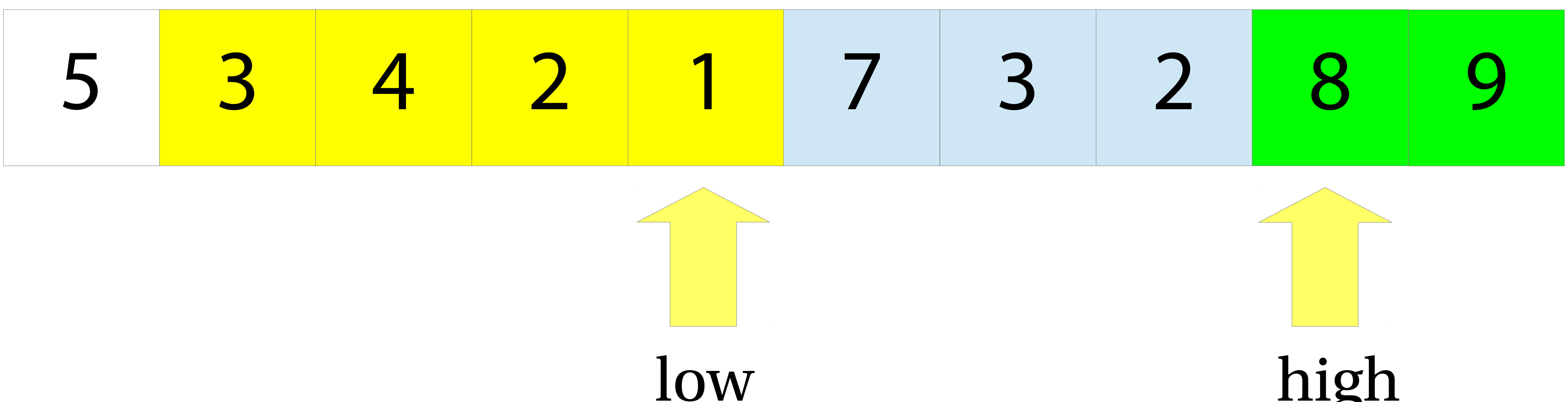
5. Advance low and high and repeat




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

Partitioning algorithm

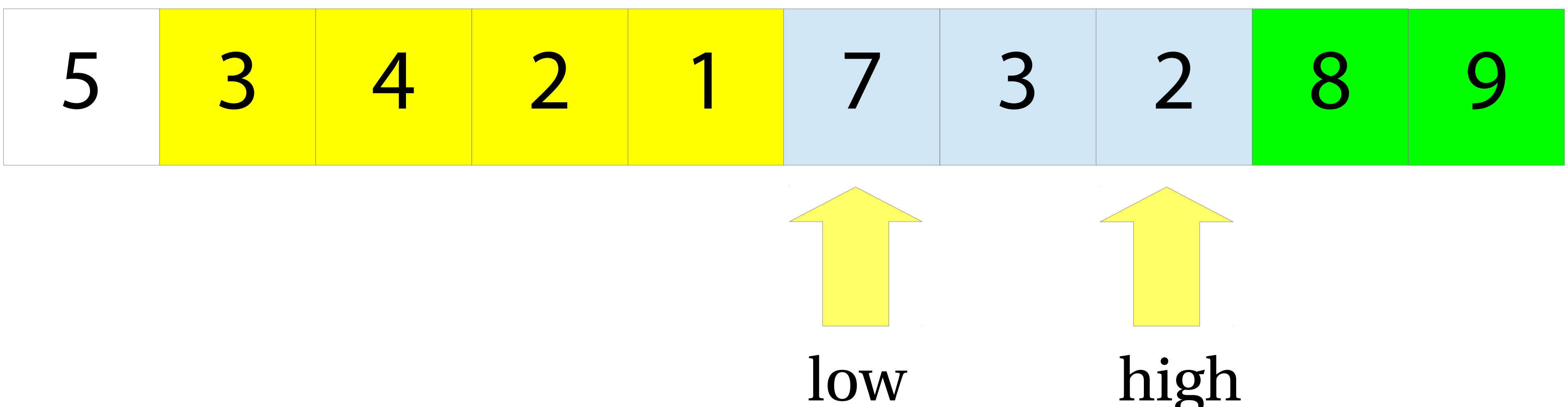
5. Advance low and high and repeat




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

Partitioning algorithm

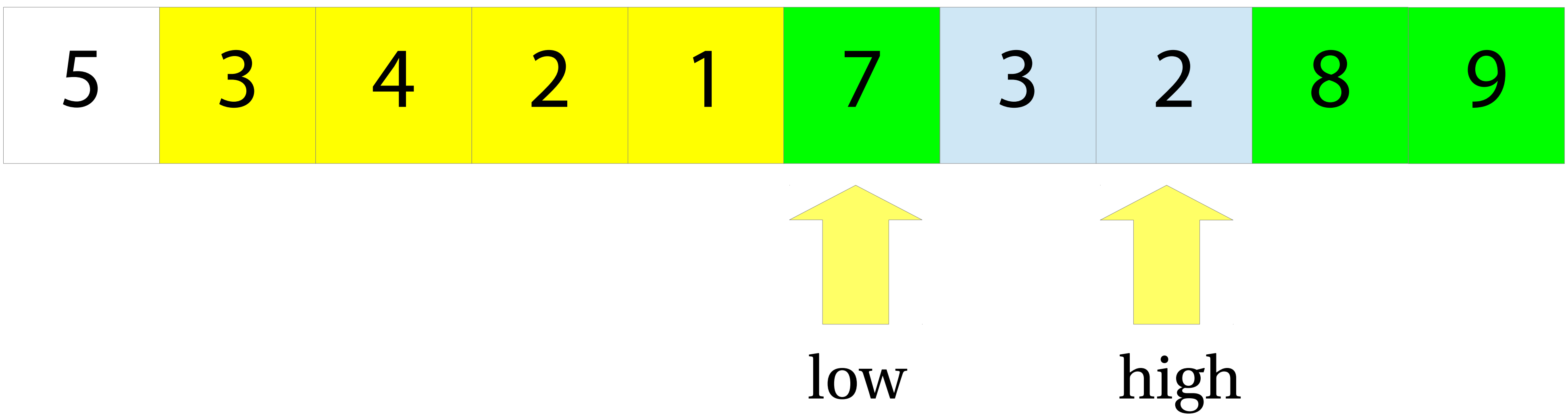
5. Advance low and high and repeat




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

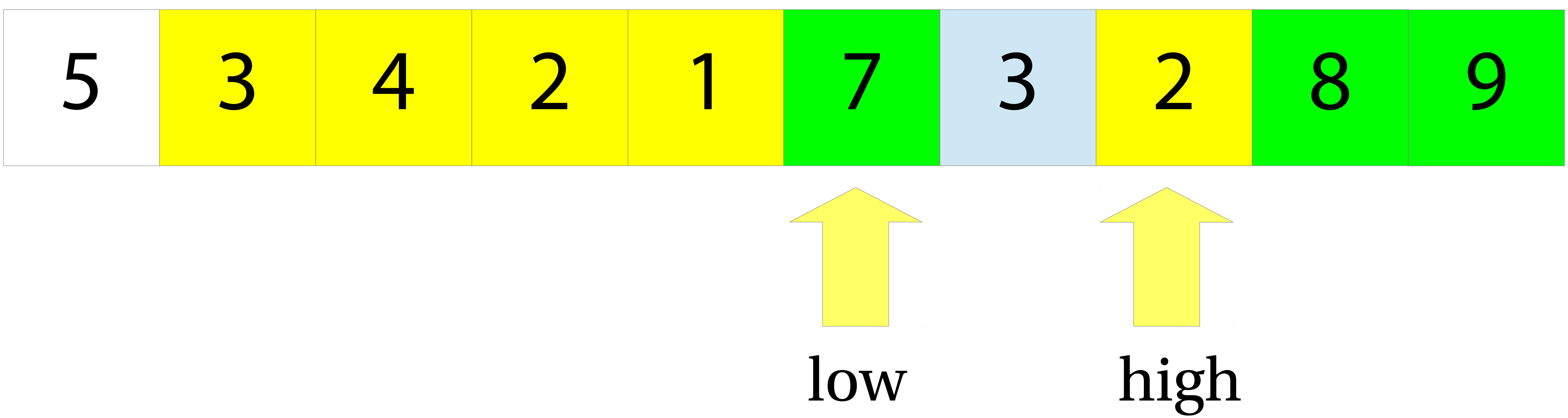
Partitioning algorithm

5. Advance low and high and repeat



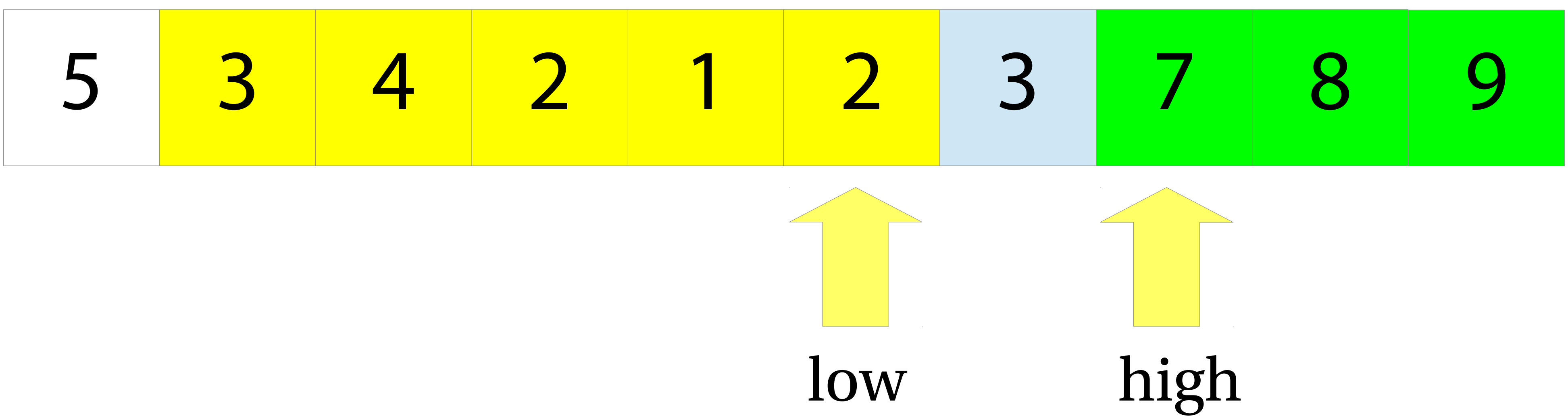
Partitioning algorithm

5. Advance low and high and repeat



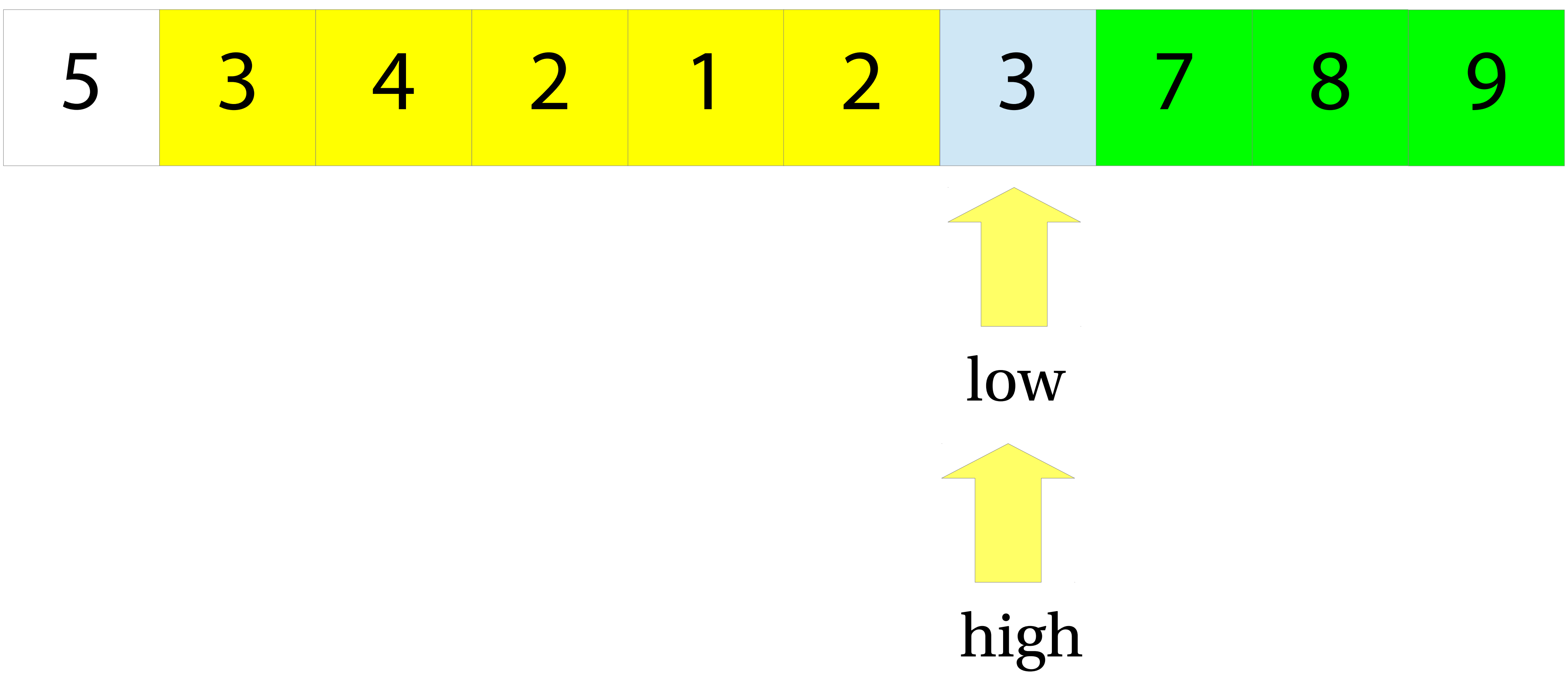
Partitioning algorithm

5. Advance low and high and repeat



Partitioning algorithm

5. Advance low and high and repeat



Partitioning algorithm

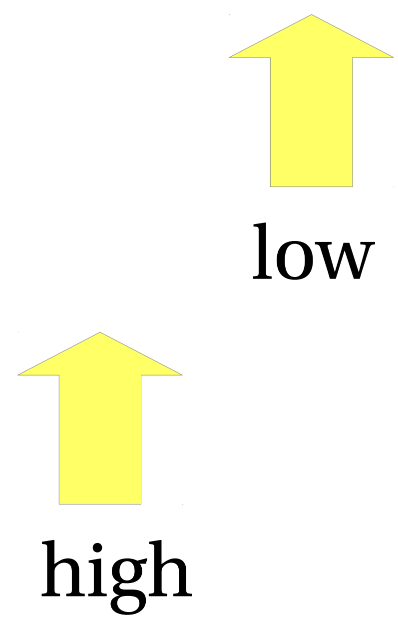
5. Advance low and high and repeat

Partitioning algorithm

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

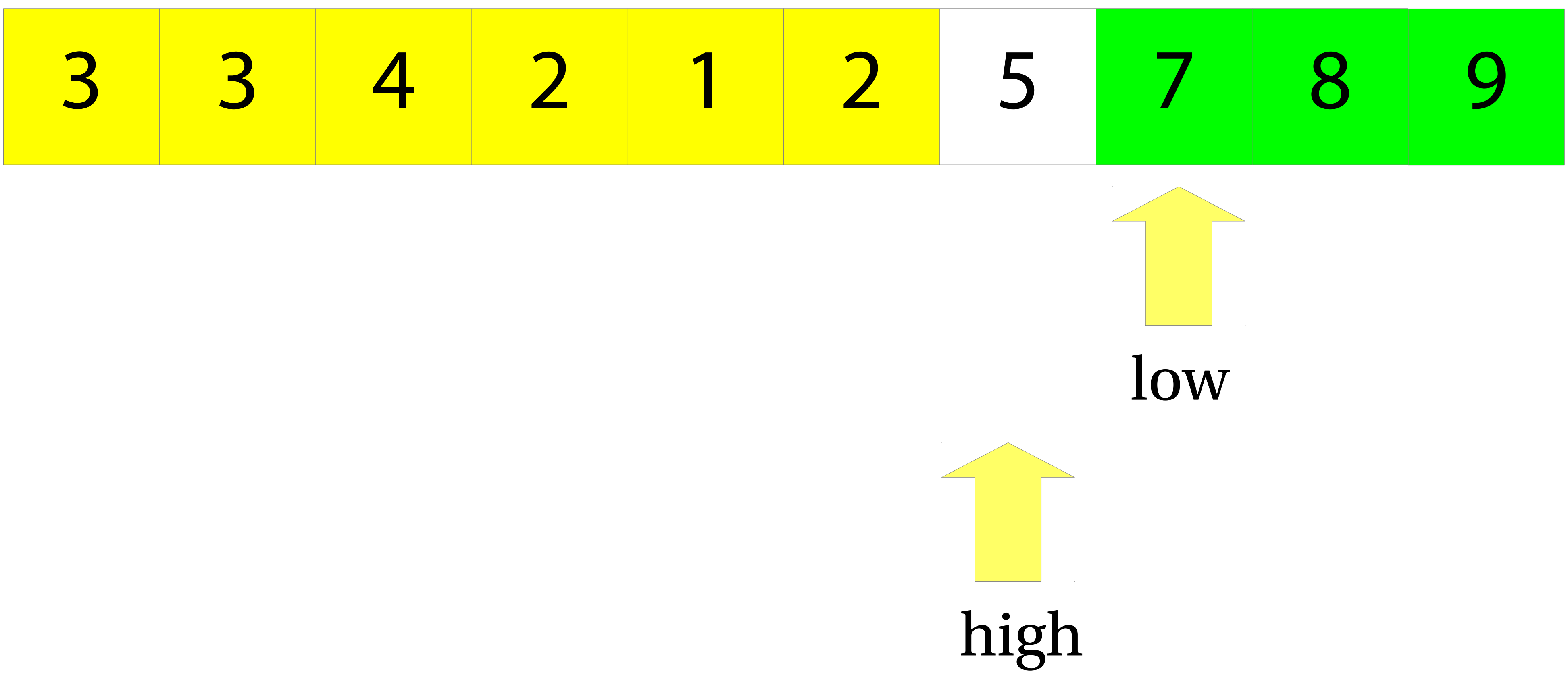
5	3	4	2	1	2	3	7	8	9

But the pivot is in the wrong place.



Partitioning algorithm

7. Last step: swap pivot with high



Details

1. What to do if the pivot is not the first element?

- 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

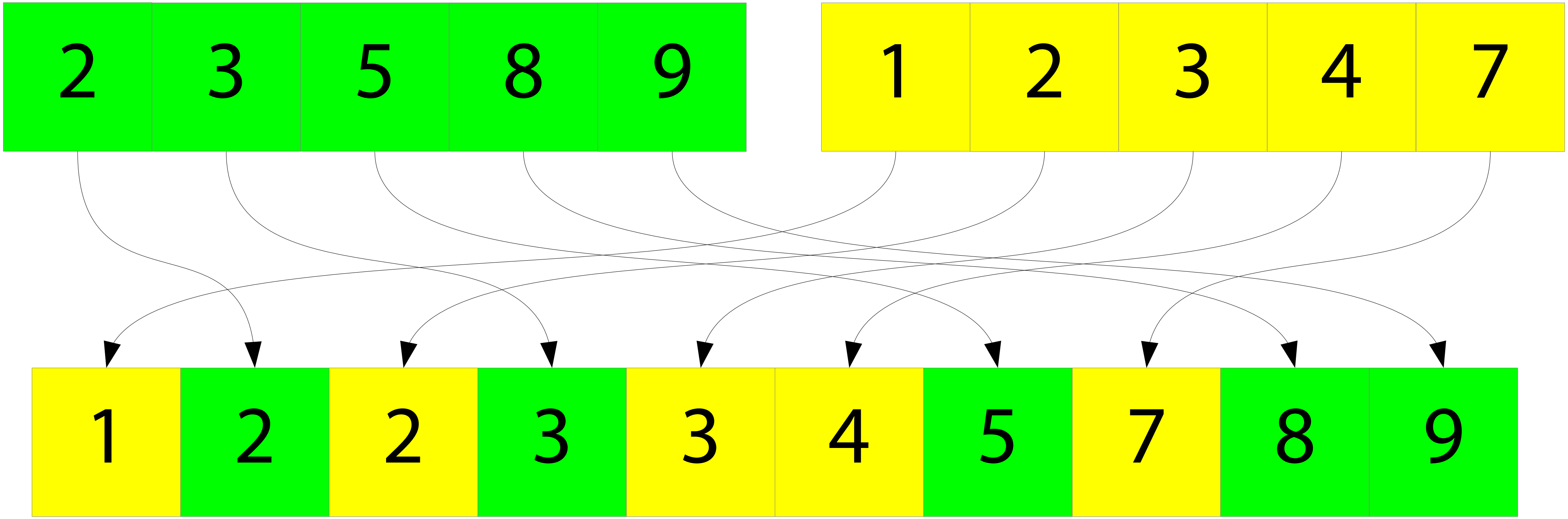
Quicksort

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

Mergesort

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



Mergesort

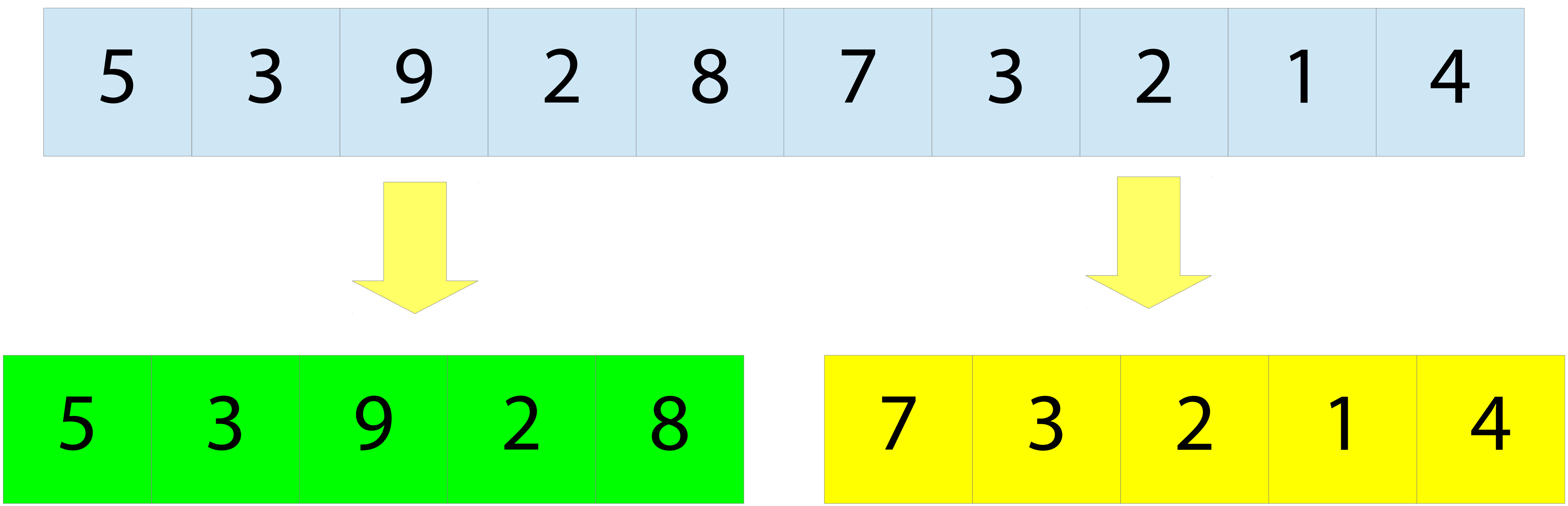
Another divide-and-conquer algorithm

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

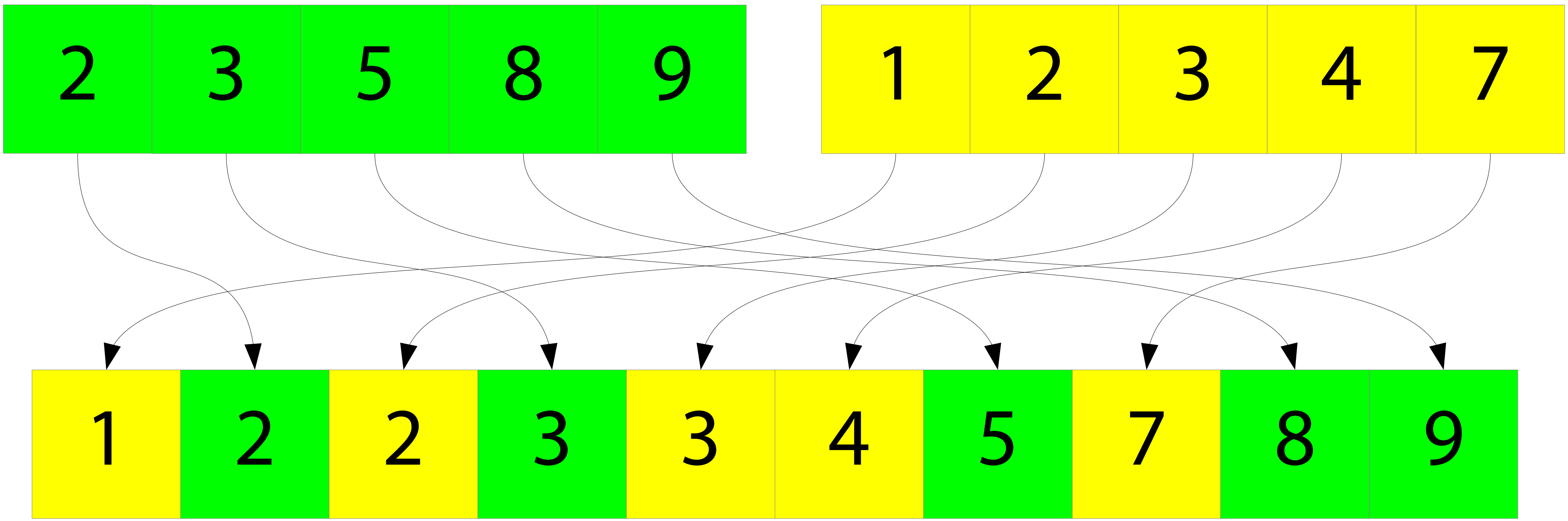


Mergesort

2. *Recursively* mergesort the two parts

Mergesort

3. *Merge* the two sorted lists together

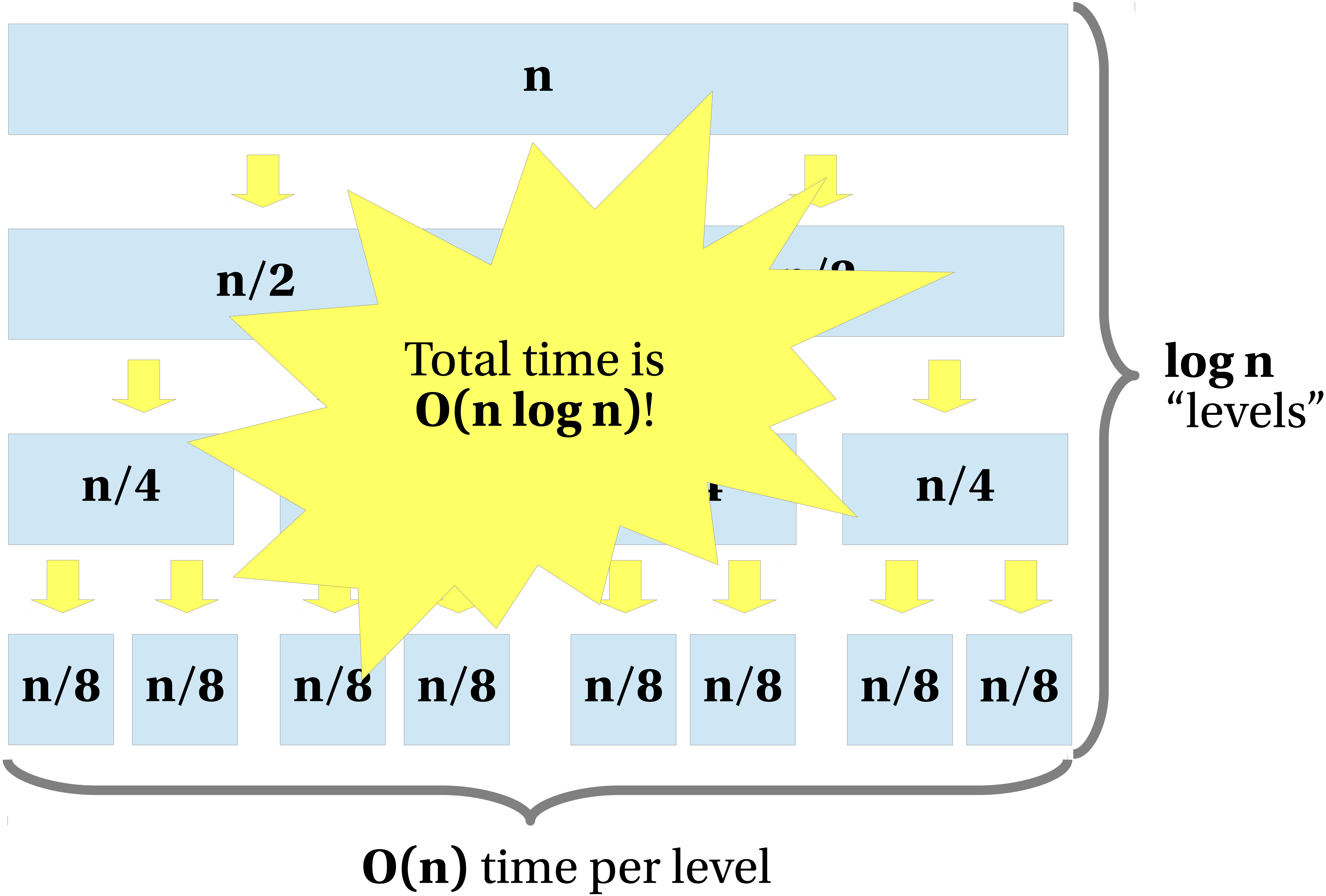


Complexity analysis

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


n
n/2
Total time is
O(n log n)!
log n
“levels”
n/4
n/4
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
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)

Both the best in their fields!

- Quicksort best imperative algorithm
- Mergesort best functional algorithm