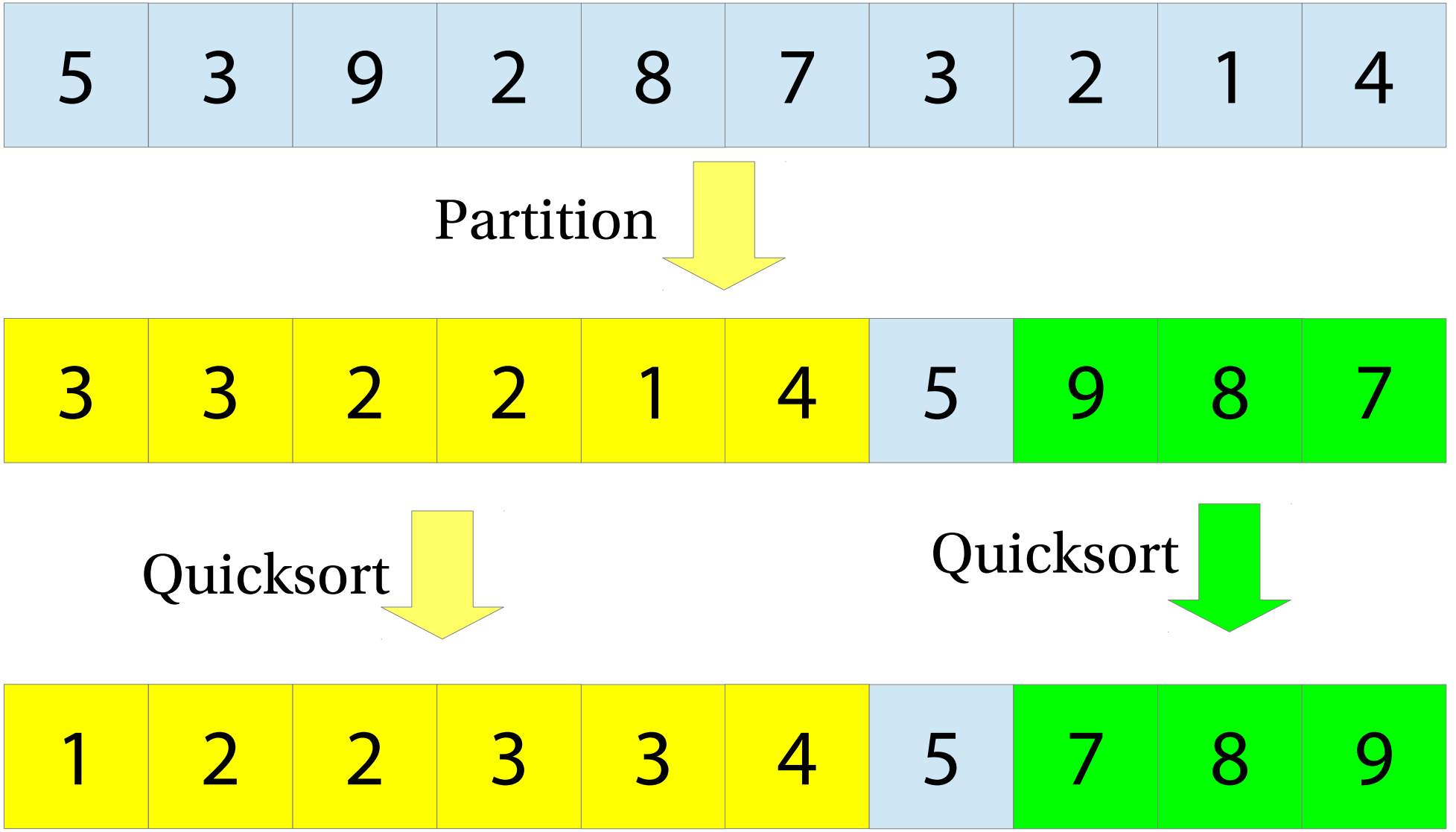
Sorting

Weiss chapter 7.5, 8.5, 8.6, 8.8

Quicksort – a reminder



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 than the pivot


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

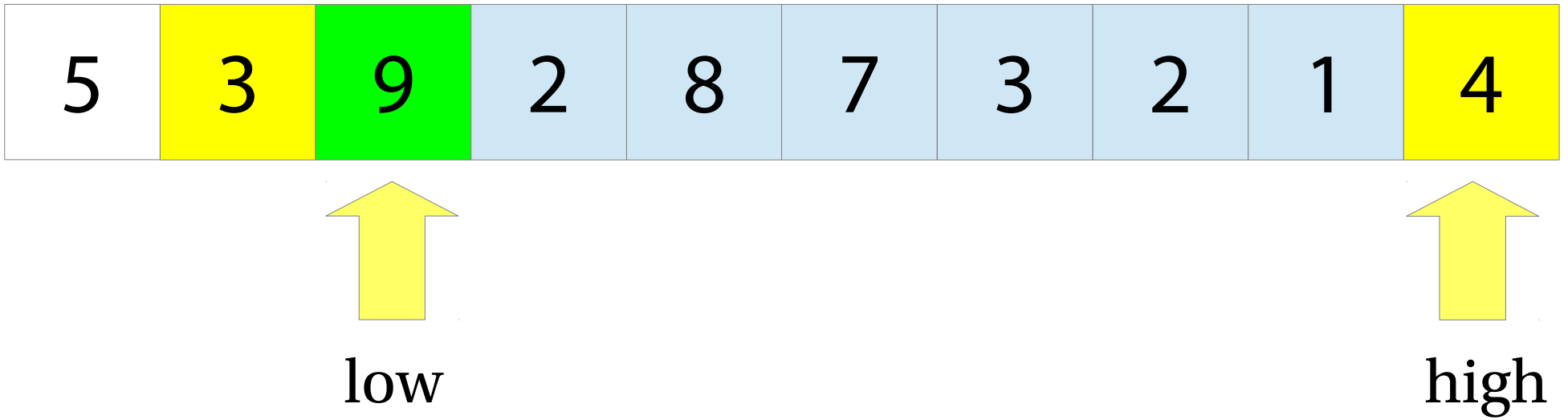
Partitioning algorithm

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


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

Partitioning algorithm

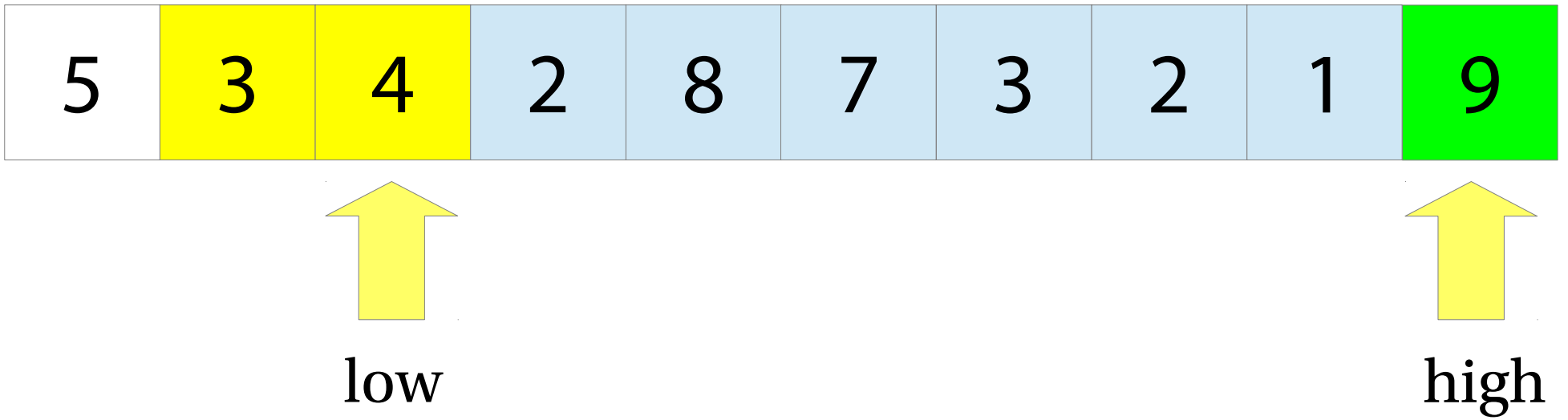
3. Move high left until you find something less than the pivot




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

Partitioning algorithm

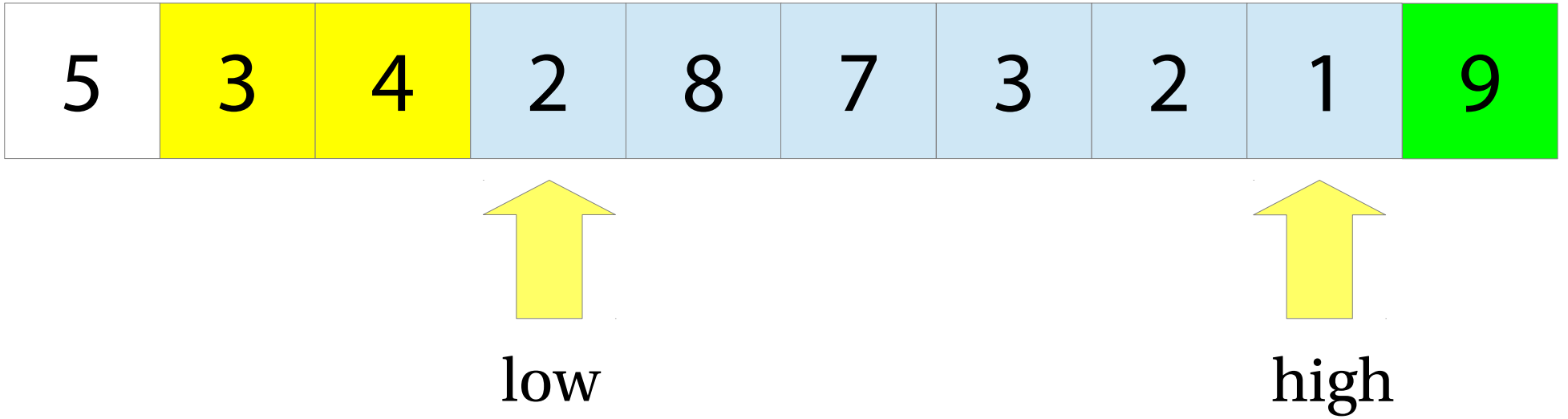
4. Swap them!




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

Partitioning algorithm

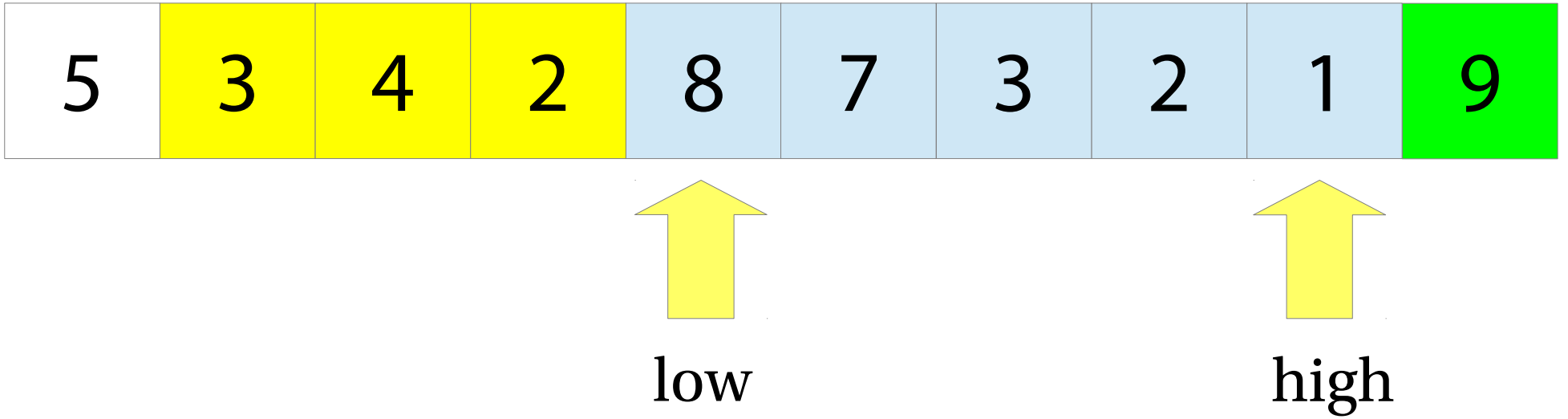
5. Advance low and high and repeat




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

Partitioning algorithm

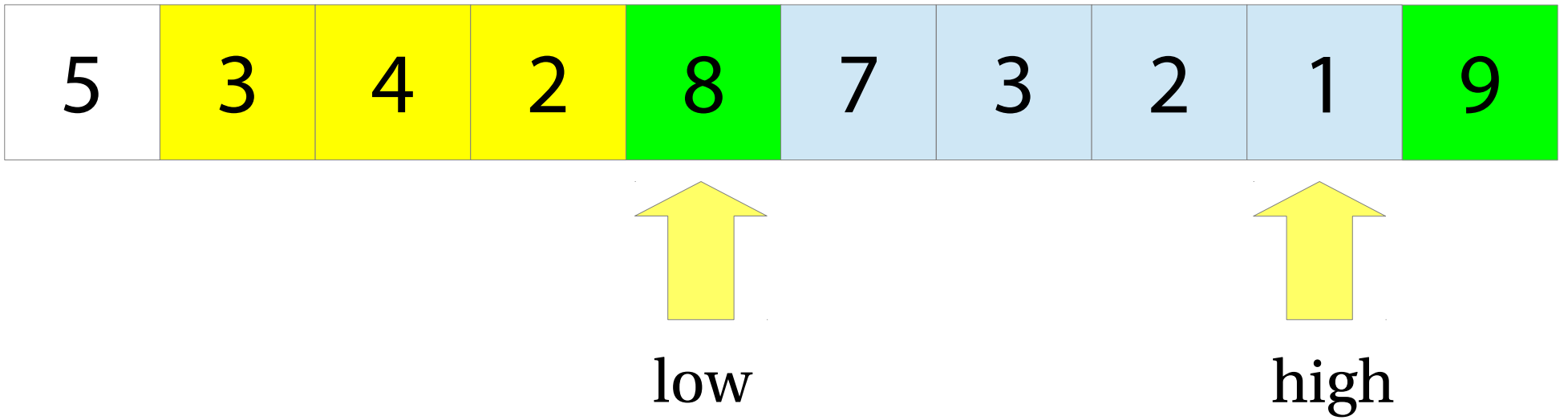
5. Advance low and high and repeat




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

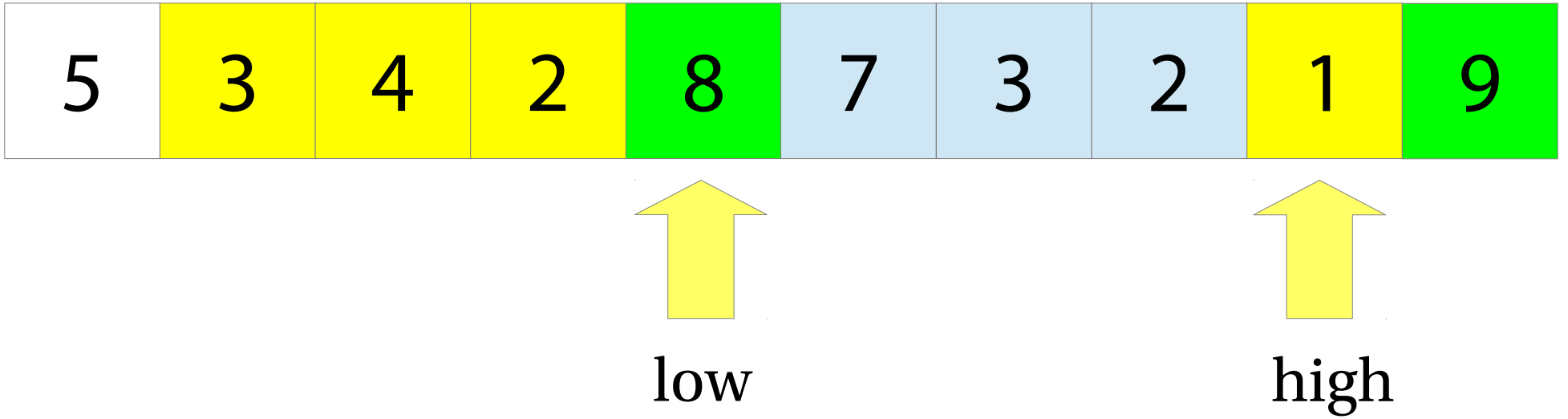
Partitioning algorithm

5. Advance low and high and repeat



Partitioning algorithm

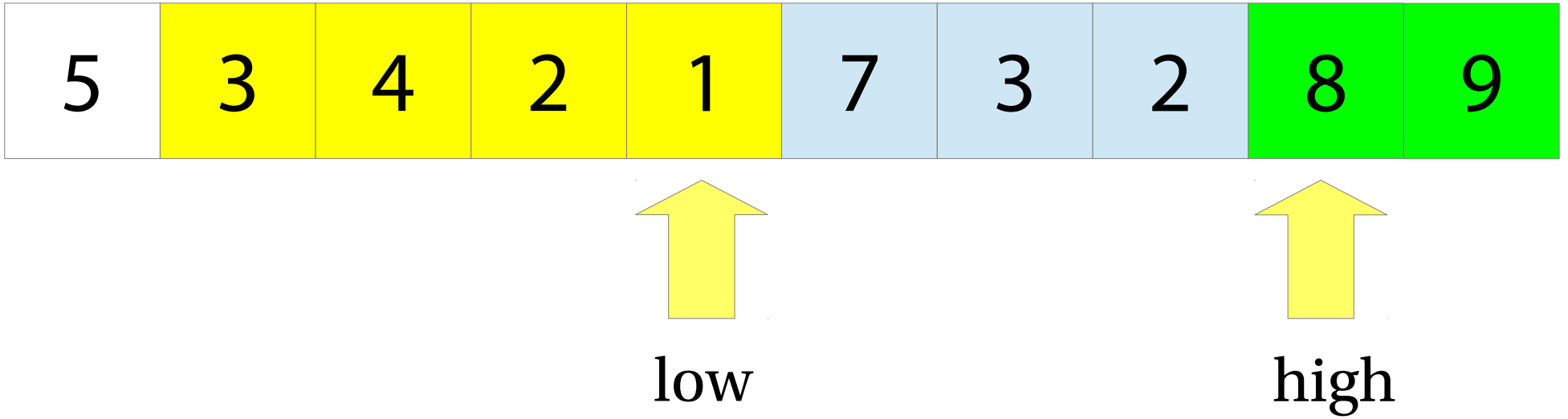
5. Advance low and high and repeat




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

Partitioning algorithm

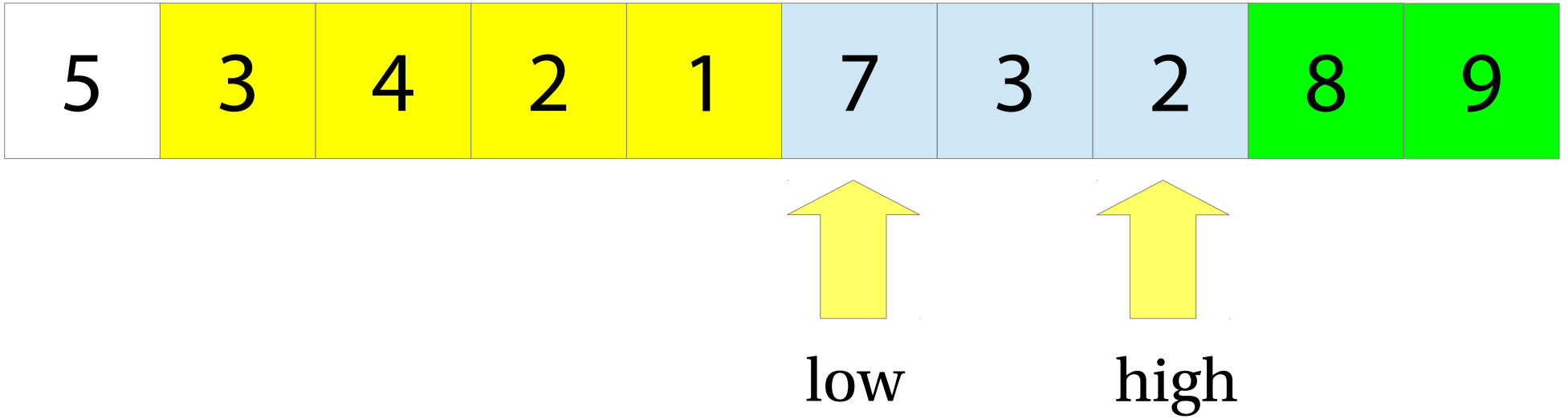
5. Advance low and high and repeat




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

Partitioning algorithm

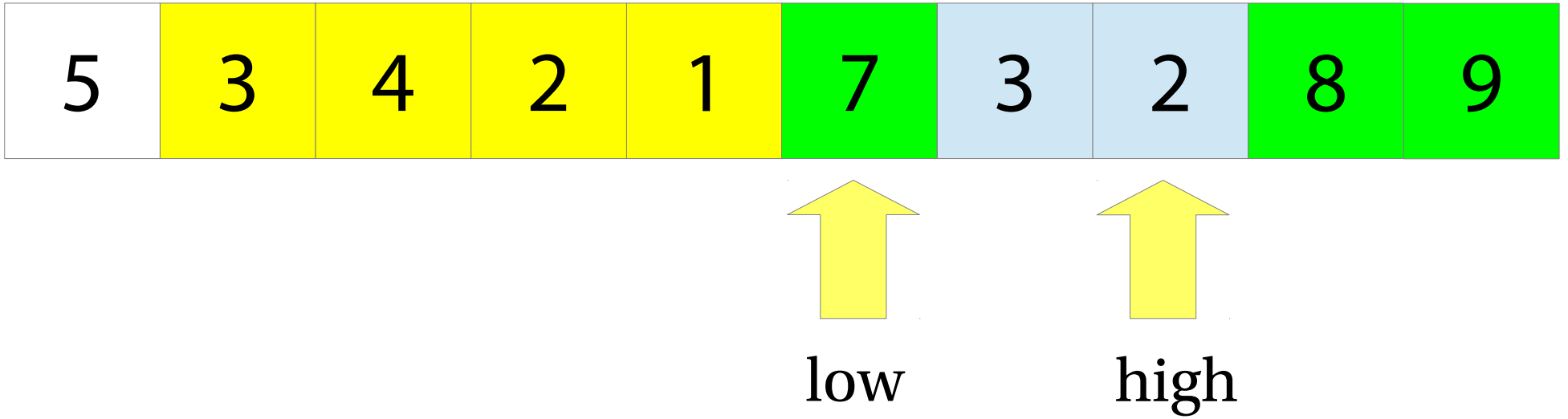
5. Advance low and high and repeat




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

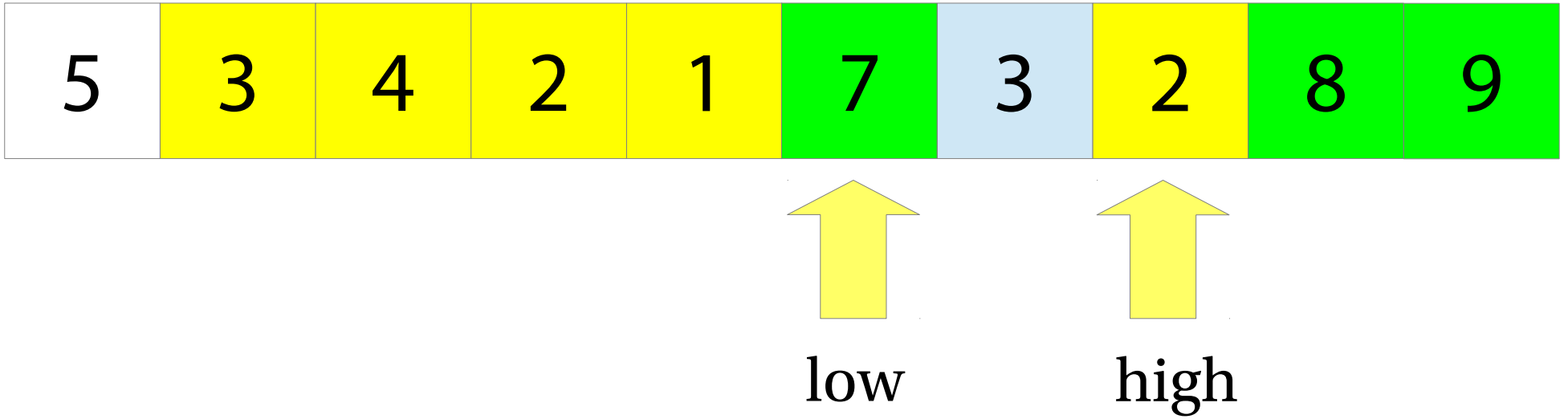
Partitioning algorithm

5. Advance low and high and repeat



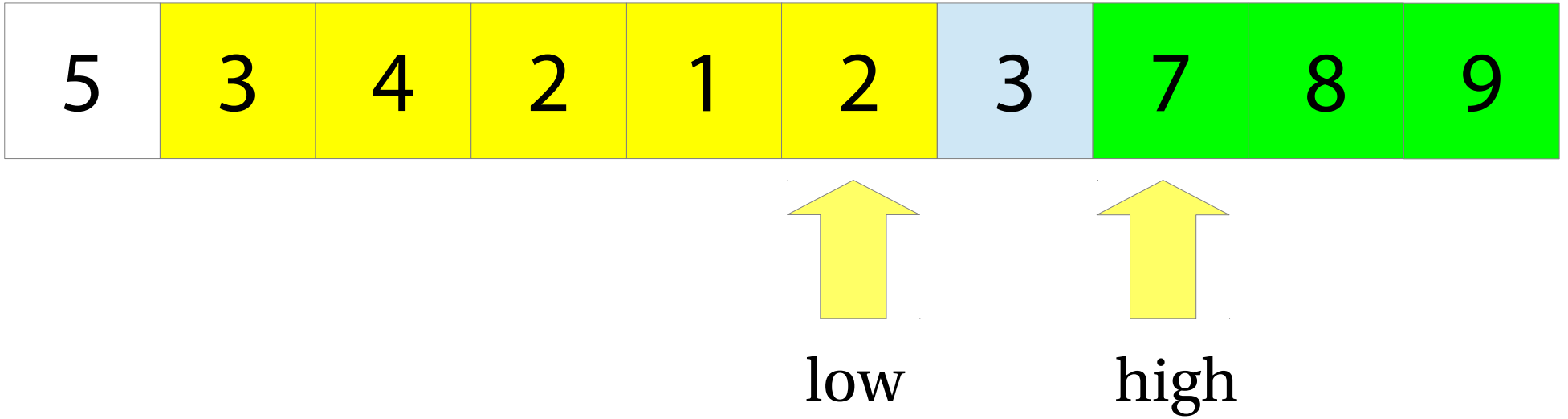
Partitioning algorithm

5. Advance low and high and repeat



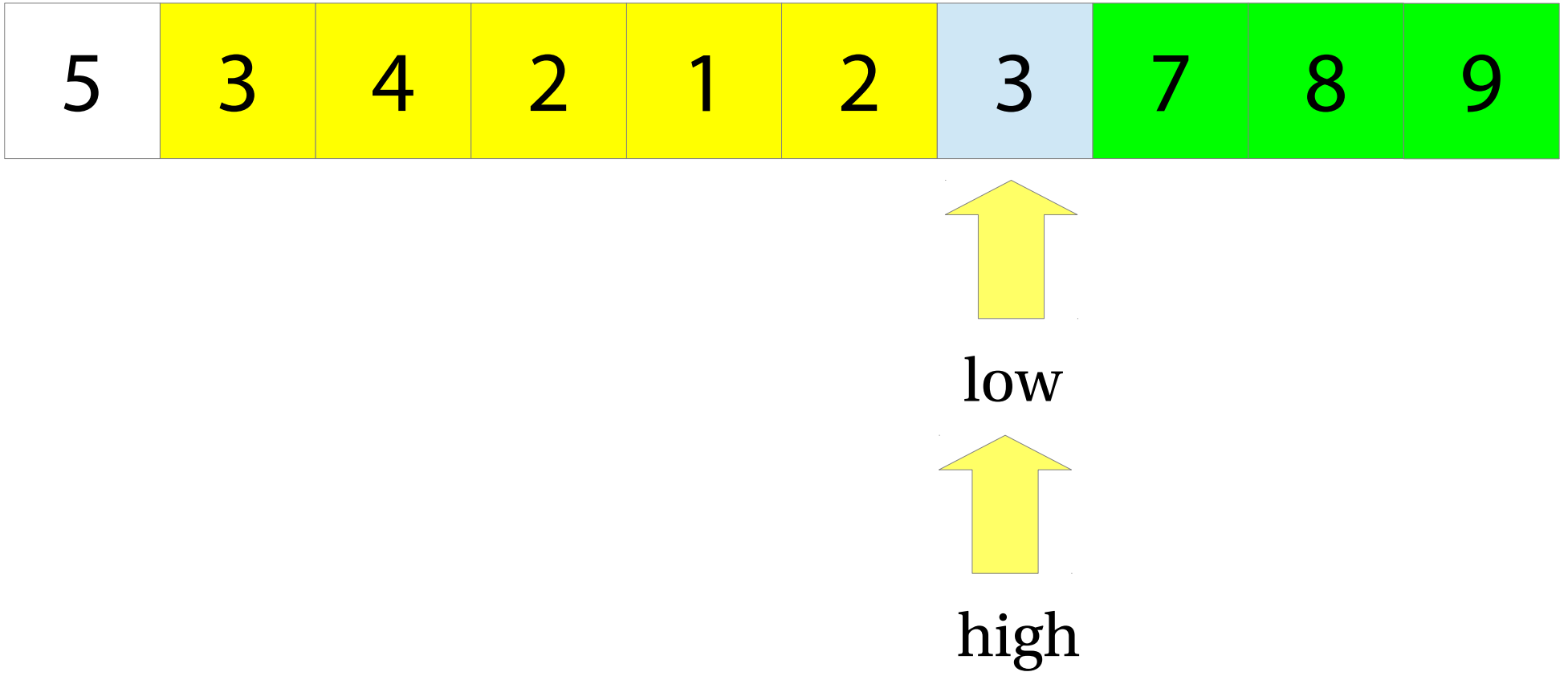
Partitioning algorithm

5. Advance low and high and repeat



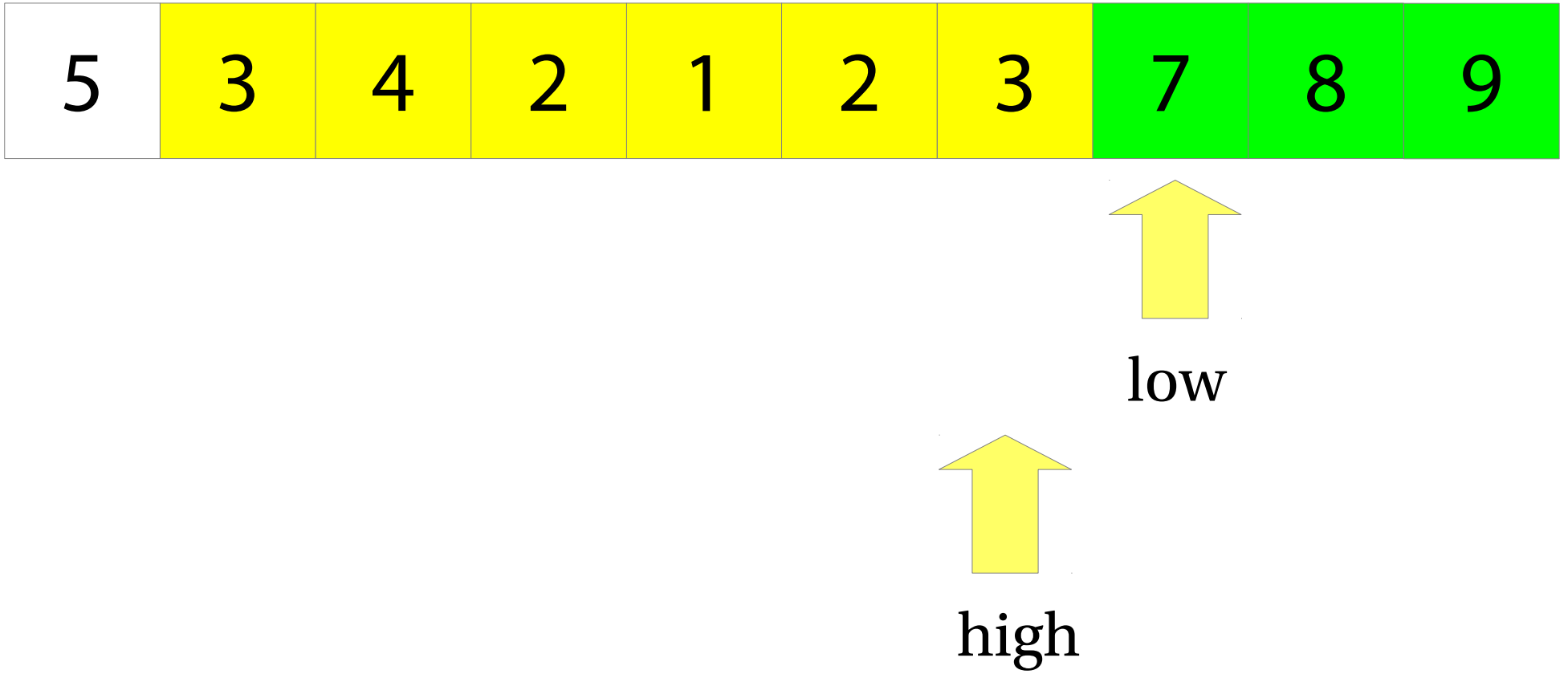
Partitioning algorithm

5. Advance low and high and repeat



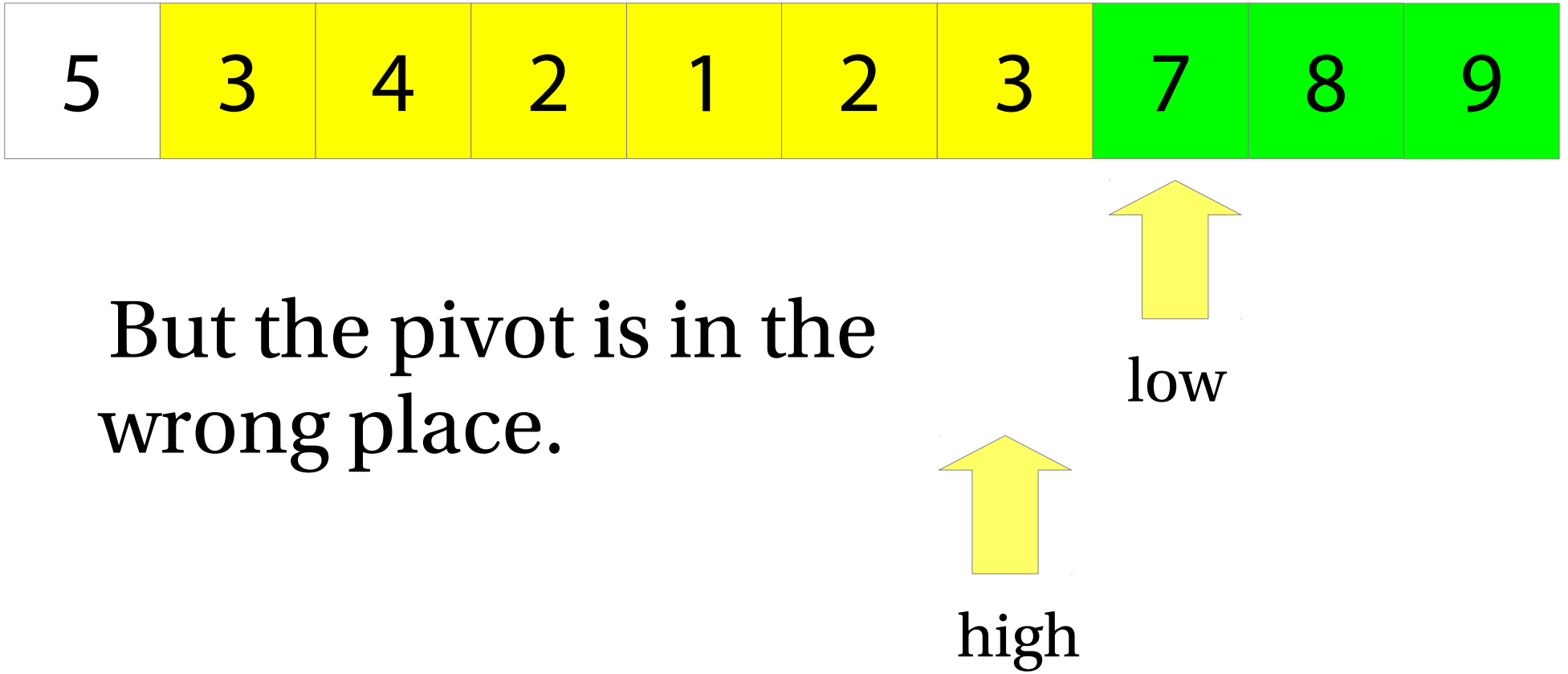
Partitioning algorithm

5. Advance low and high and repeat



Partitioning algorithm

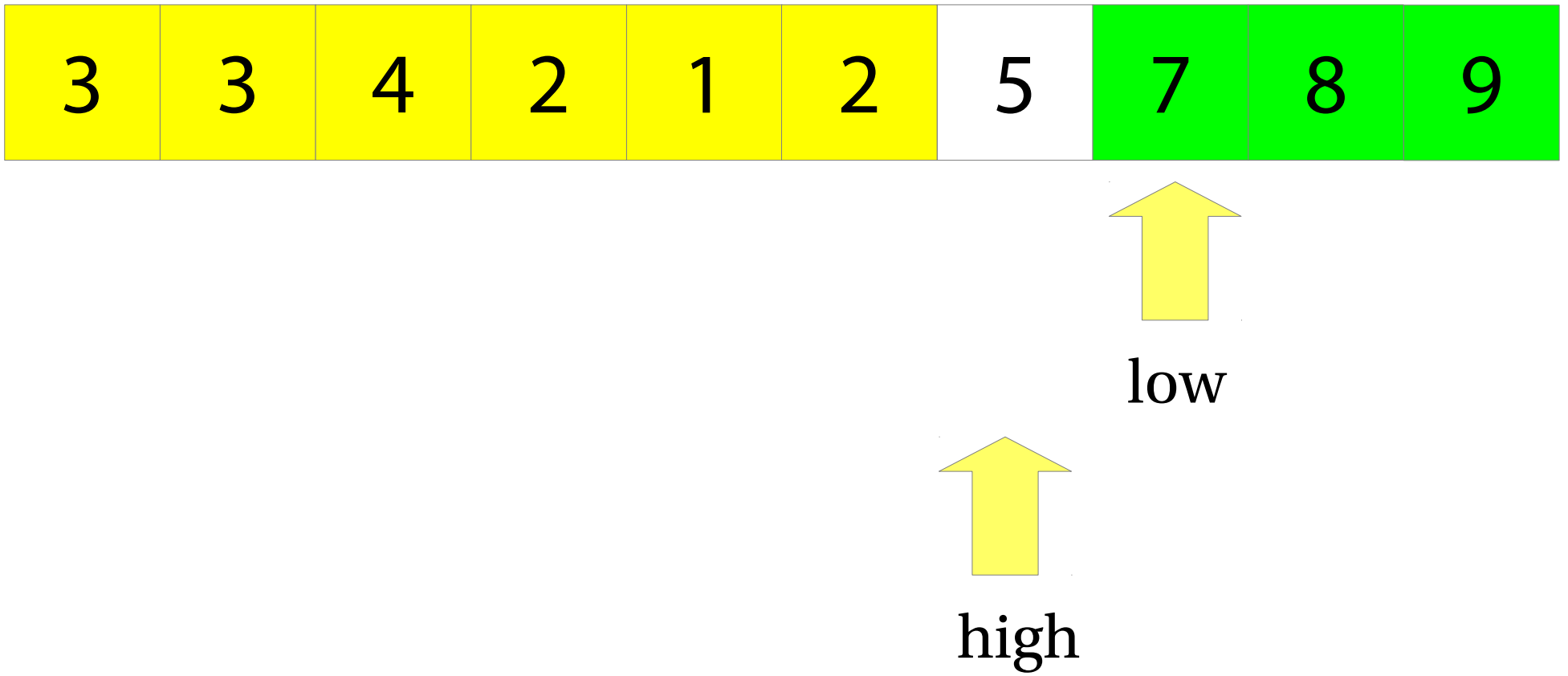
6. When low and high have crossed, we are finished!



But the pivot is in the wrong place.

Partitioning algorithm

7. Last step: swap pivot with high



Partitioning algorithm

```
int pivot = a[0];
int low = 1;
int high = a.length – 1;
while(low >= high) {
  while(low <= high && a[low] < pivot)
    low++;
  while(low <= high && a[high] > pivot)
    high--;
  if (low <= high) {
    swap(a[low], a[high]);
    low++; high--;
  }
}
swap(a[0], a[high]);
// Pivot is now at index 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

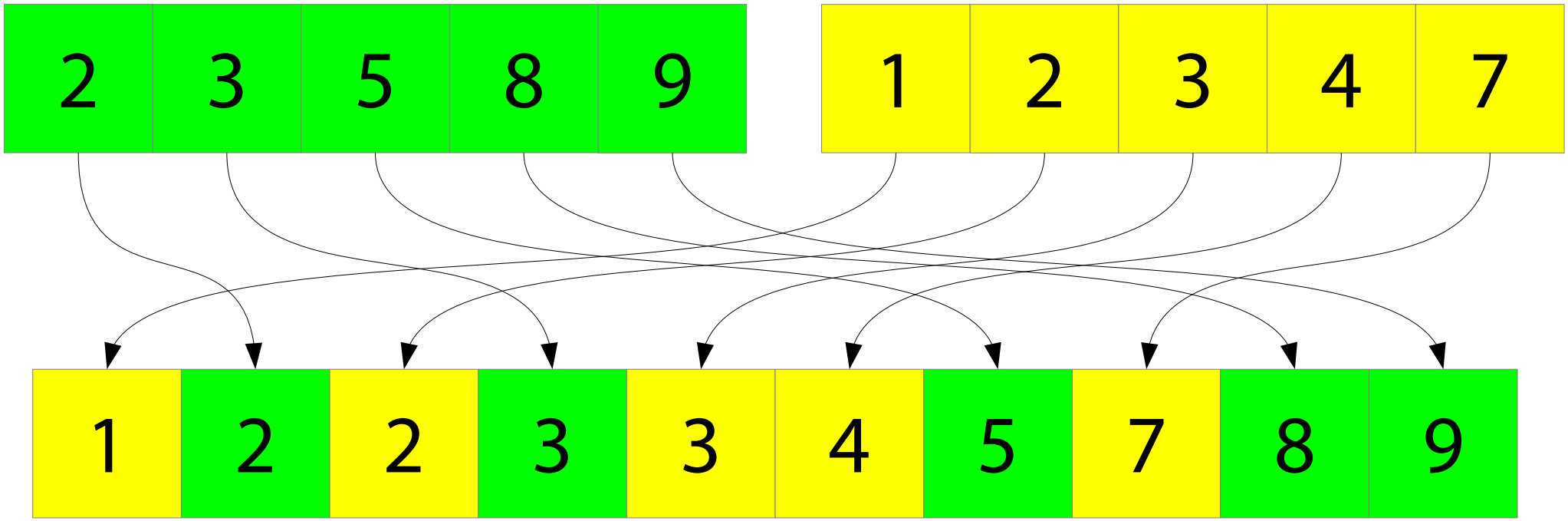
Quicksort

Can be very fast, but many implementation details to get right!

- 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:



Merging

```
merge :: Ord a => [a] → [a] → [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
   | x < y = x:merge xs (y:ys)
   | otherwise = y:merge (x:xs) ys
```

Mergesort

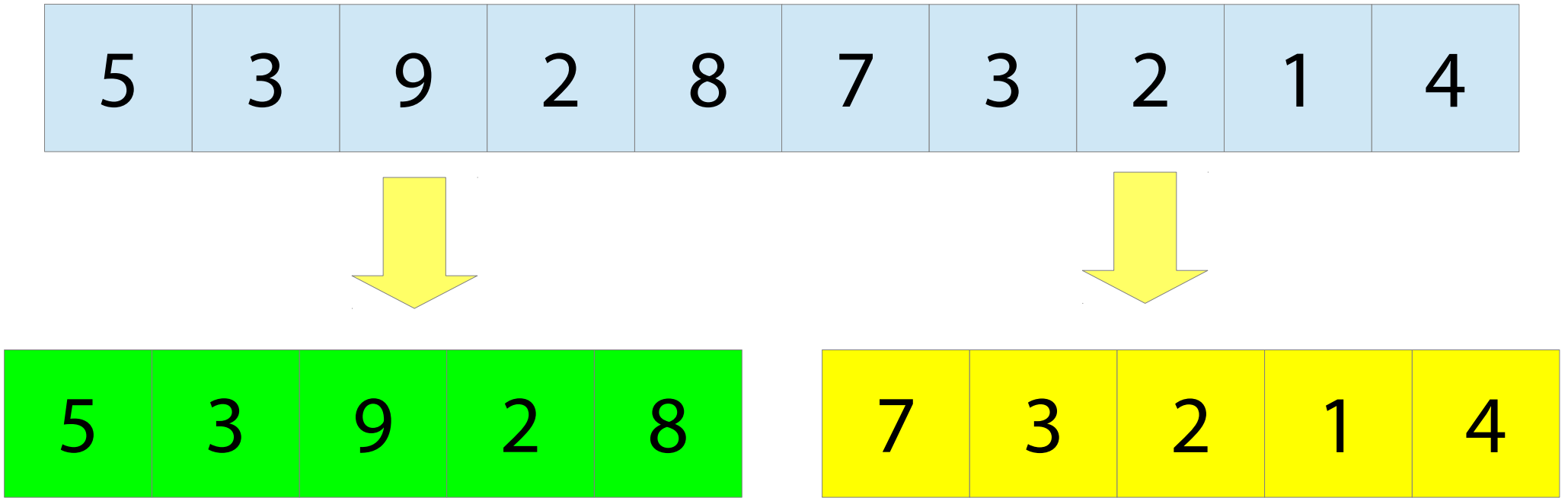
Another divide-and-conquer algorithm

To mergesort a list:

- *Split* the list into two equal parts
- *Recursively* mergesort the two parts
- *Merge* the two sorted lists together

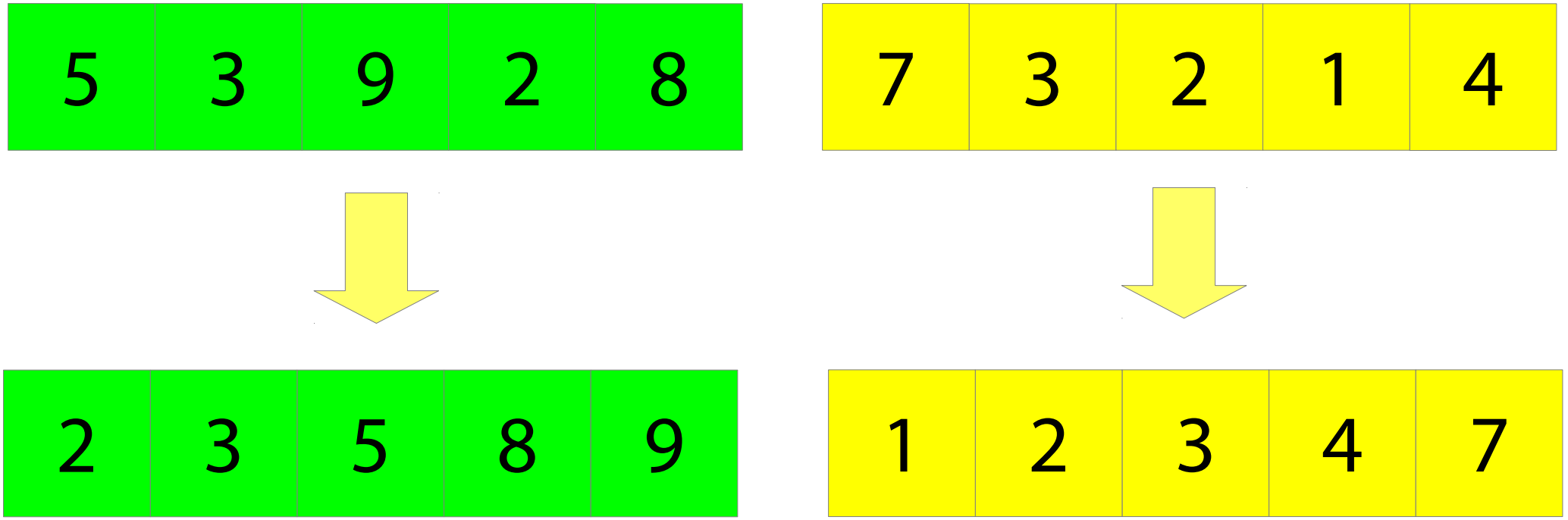
Mergesort

1. *Split* the list into two equal parts



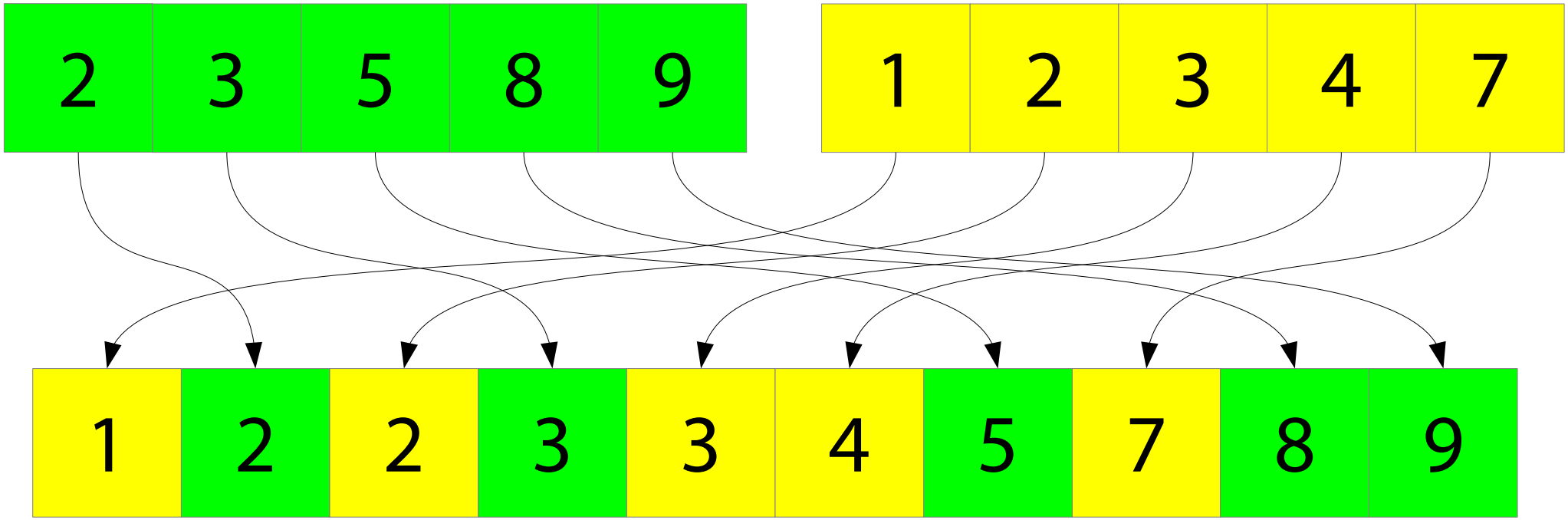
Mergesort

2. *Recursively* mergesort the two parts



Mergesort

3. *Merge* the two sorted lists together



Mergesort in Haskell

```
sort [] = []
sort [x] = [x]
sort xs = merge (sort ys) (sort zs)
  where
    (ys, zs) = splitInHalf xs
```

Split in Haskell

```
splitInHalf :: [a] → ([a], [a])
splitInHalf xs =
  (take n xs, drop n xs)
  where
    n = length xs `div` 2
```

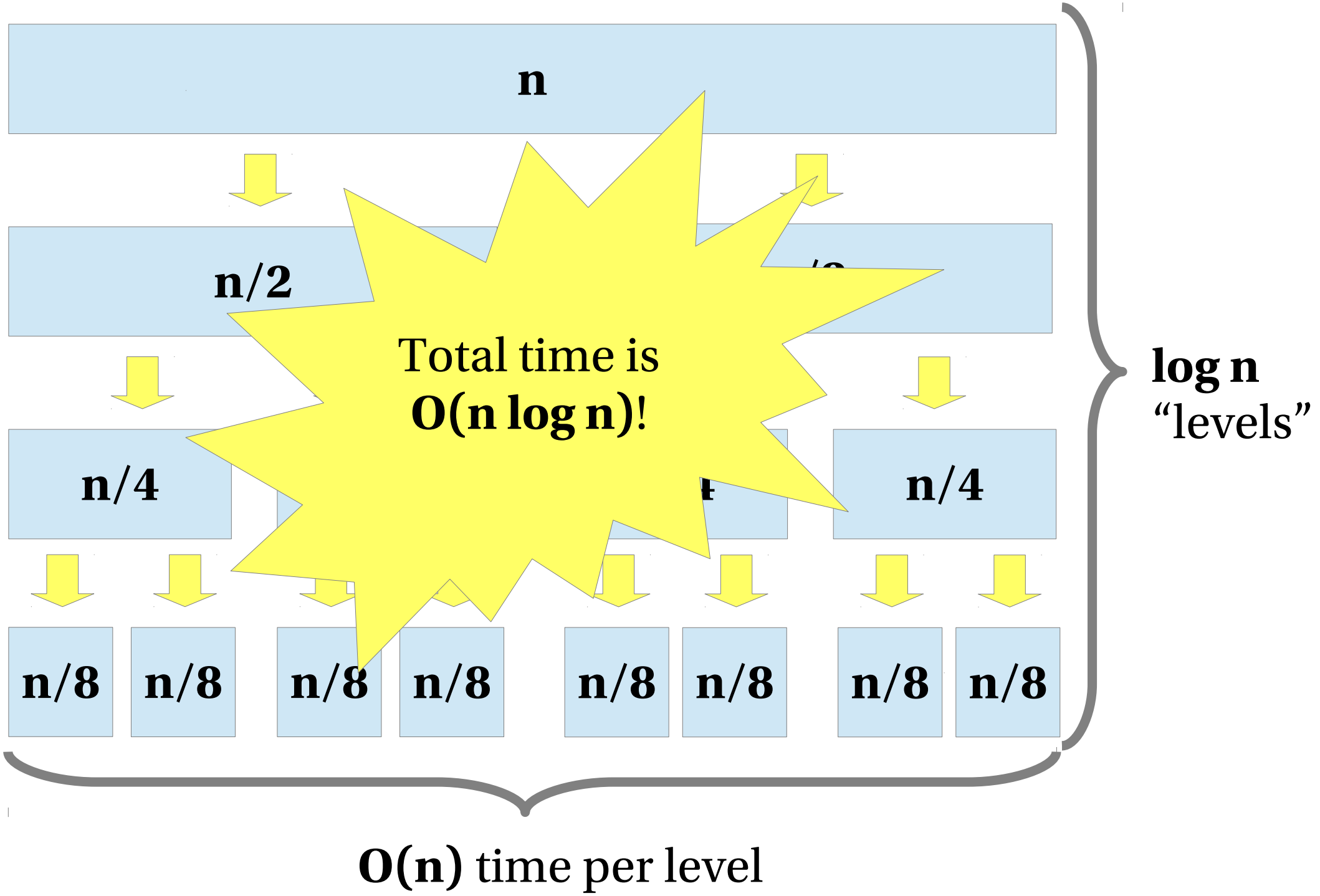
(traverses the list three times – once for take, once for drop, once for length. Better implementation: exercise!)

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
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

Mergesort vs quicksort

Mergesort:

- Not in-place
- O(n log n)
- Works on Haskell lists (sequential access)

Quicksort:

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

Can we do better than O(n log n)?

It turns out that O(n log n) is the best we can do if our sorting algorithm is based on comparing elements.

But how do we prove a negative (there is *no algorithm* better than O(n log n))?

Let's look at a simpler example first.

Can we do better than O(n log n)?

I am thinking of a number between 1 and 3. Can you guess what it is with one yes/no question?

No, of course not.

Why?

Suppose if I answered yes you would guess 1, and if I answered no you would guess 2. You would never guess 3!

There are 3 numbers, but only 2 answers.

Can we do better than O(n log n)?

I am thinking of a number between 1 and 5. Can you guess what it is with two yes/no questions?

No.

Why?

There are 5 possible numbers, but only 4 possible answers to your two questions (yes/yes, yes/no, no/yes, no/no).

Can we do better than O(n log n)?

General principle:

If I am thinking of one of n objects, and you have to guess which one, there must be at least n possible outcomes of your questioning.

If the questions are yes/no, you have to ask at least log n questions.

Can we do better than O(n log n)?

I have an array of length n, and you have to guess what order it is in. You may ask yes/no questions: “is this element less than that element?”

There are n! permutations of n elements

So you need to perform **log (n!)** comparisons.

But how much is log (n!), anyway?

Can we do better than O(n log n)?

$$\log(n!)$$
$$= \log(n \times (n\text{-}1) \times \ldots \times 1)$$
$$= \log n + \log(n\text{-}1) \times \ldots \times \log 1$$
$$\leq \log n + \log n + \ldots + \log n \text{ (n times)}$$
$$= n \log n$$

So $\log(n!) \leq n \log n$

Can we do better than O(n log n)?

$\log (n!)$
$= \log (n \times (n\text{-}1) \times \ldots (n/2+1) \times \ldots \times 1)$
$= \log n + \log (n\text{-}1) + \ldots + \log (n/2+1) + \ldots + \log 1$
$\geq \log n + \log (n\text{-}1) + \ldots + \log (n/2+1)$
$\geq \log (n/2) + \log (n/2) + \ldots + \log (n/2)$ $(n/2)$ times)
$= (n/2) \log (n/2)$
$= (n/2) (\log n - \log 2)$
$= (n/2) (\log n - 1)$

So $\log (n!) \leq n \log n$ and $\log (n!) \geq (n/2) (\log n - 1)$
So $\log (n!) = O (n \log n)$
So comparison-based sorting must take at least $O(n \log n)$ comparisons!

O(n) sorting algorithms

O(n log n) bound only applies for *comparison-based* sorting!

Suppose I want to sort an array of integers, all between 0 and k-1.

Make an array of k integers, and *count* how many times each value appears in the input array. Then produce the right number of each value. This is called *counting sort*.

Only practical if k is small. If it is big, use *bucket sort*. (See Wikipedia)

Counting sort

```
// Assumes all integers are between 0 and k-1
void sort(int[] a, int k) {
   int counts[k];
   // Count how many times each value occurs
   for(int x: a) counts[x]++;
   // Write the results out
   int idx = 0;
   for(int i = 0; i < k; i++) {
      for (int j = 0; j < counts[i]; j++)
         a[idx++] = i;
}
```

The best sorting algorithm?

The ideal sorting algorithm should be:

- O(n log n) worst case
- O(n) when input is almost sorted
- Simple
- In-place

No sorting algorithm has all four!

Simple algorithms

Bubblesort is just bad

Selection sort: simple but not too fast

Insertion sort: $O(n^2)$ worst case, $O(n)$ on sorted lists

- Because insertion is $O(1)$ best case
- Insertion sort is the fastest algorithm of all on small arrays and almost-sorted arrays!

Fancier algorithms

Quicksort – normally O(n log n), but always danger of $O(n^2)$. Very fast normally.

Introsort – quicksort, but switch to heapsort if the recursion depth gets too big (used in some C++ compilers)

Heapsort (later in the course) – always O(n log n)

Mergesort

Always O(n log n), but not in-place

"Smooth mergesort": split list into already-sorted *runs*, merge those. Gives O(n) on sorted lists (used in GHC)

Timsort: super-optimised smooth mergesort, used in Python and Java

O(n) sorting algorithms

Special cases: counting sort, bucket sort, radix sort, ...

Complexity of recursive algorithms

Calculating complexity

Let T(n) be the time mergesort takes on a list of size n

Then write down a *recurrence relation*:

$$T(n) = O(n) + 2T(n/2)$$

Time to sort a list of size n

Linear amount of time spent in splitting + merging

Plus two recursive calls of size n/2

Calculating complexity

Procedure for calculating complexity of a recursive algorithm:

- Write down a recurrence relation
- *Solve* the recurrence relation to get a formula for T(n) (difficult!)

We will see solutions for a few recurrence relations only

Calculating complexity

$T(n) = O(1) + T(n\text{-}1)$: $T(n) = O(n)$

$T(n) = O(n) + T(n\text{-}1)$: $T(n) = O(n^2)$

$T(n) = O(1) + T(n/2)$: $T(n) = O(\log n)$

$T(n) = O(n) + T(n/2)$: $T(n) = O(n)$

How do we calculate these?

Expanding out recurrence relations

$T(n) = 1 + T(n-1)$:

$T(n) = 1 + T(n-1)$

$= 2 + T(n-2)$

$= 3 + T(n-3)$

$= \ldots$

$= n + T(n-n)$

$= n + T(0)$

$= O(n)$

Expanding out recurrence relations

$T(n) = n + T(n-1)$:

$T(n) = n + T(n-1)$

$= n + (n-1) + T(n-2)$

$= n + (n-1) + (n-2) + T(n-3)$

$= \ldots$

$= n + (n-1) + (n-2) + \ldots + 1$

$= n(n+1) / 2$

$= O(n^2)$

Expanding out recurrence relations

$T(n) = 1 + T(n/2)$:

$T(n) = 1 + T(n/2)$

$= 2 + T(n/4)$

$= 3 + T(n/8)$

$= \ldots$

$= \log n + T(n/n)$

$= \log n + T(1)$

$= O(\log n)$

Expanding out recurrence relations

$T(n) = n + T(n/2)$:

$T(n) = n + T(n/2)$

$= n + n/2 + T(n/4)$

$= n + n/2 + n/4 + T(n/8)$

$= \ldots$

$= n + n/2 + n/4 + \ldots$

$< 2n$

$= O(n)$

More rules of thumb

Rule of thumb for functions that recurse once:

- Count how deep the recursion can get, then multiply by the work done per level

$T(n) = f(n) + T(n-1)$: $T(n) = O(n \times f(n))$

$T(n) = f(n) + T(n/2)$: $T(n) = O(f(n) \log n)$

This occasionally gives overestimates!

Divide-and-conquer algorithms

$T(n) = O(n) + 2T(n/2)$: $T(n) = O(n \log n)$

- This is mergesort! There is a nice proof in the book (theorem 7.4).

$T(n) = 2T(n-1)$: $T(n) = O(2^n)$

- Because 2^n recursive calls of depth n

Other cases: *master theorem* (Wikipedia) or theorem 7.5 from book

- Kind of fiddly – best to just look it up if you need it

Complexity of recursive functions

Basic idea – recurrence relations

Easy enough to write down, hard to solve

- One technique: expand out the recurrence and see what happens
- Multiply work per level with number of levels
- Drawing a diagram (like for quicksort) can help!

Master theorem for divide and conquer

A few special cases work for most algorithms – remember those