CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Data structures

Sorting

Dr. Alex Gerdes

DIT961 - VT 2018

Searching

- Suppose I give you an array, and ask you to find if a particular value is in it, say 4 (or 2)

5	3	9	2	8	7	3	2	1	4

- The only way is to look at each element in turn
- Quiz: what is the complexity?
  - $O(n)$
- This is called *linear search*
- You might have to look at every element before you find the right one

Searching

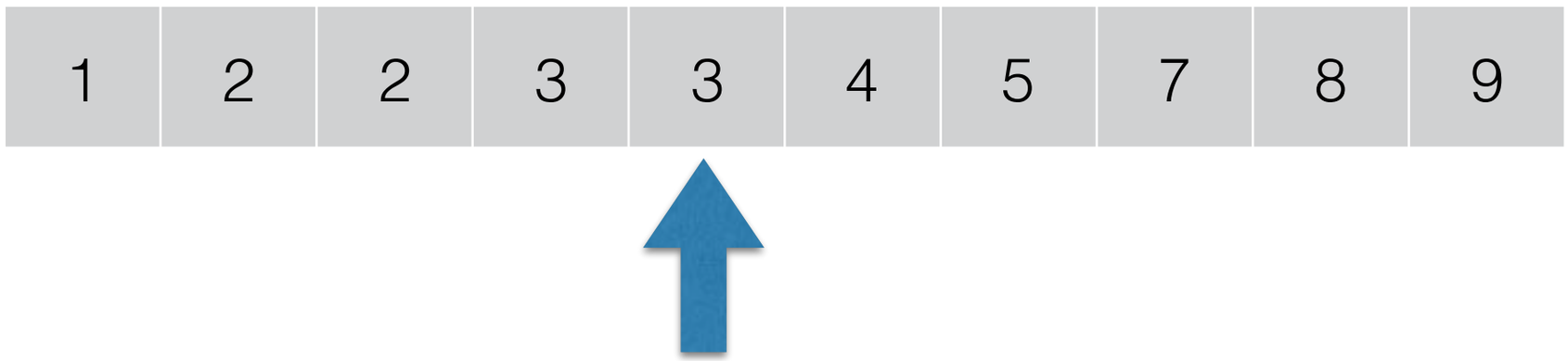
- But what if the array is *sorted*?

1	2	2	3	3	4	5	7	8	9

- Then we can use *binary search!*

Binary search

- Suppose we want to look for 4
- We start by looking at the element half way along the array, which happens to be 3

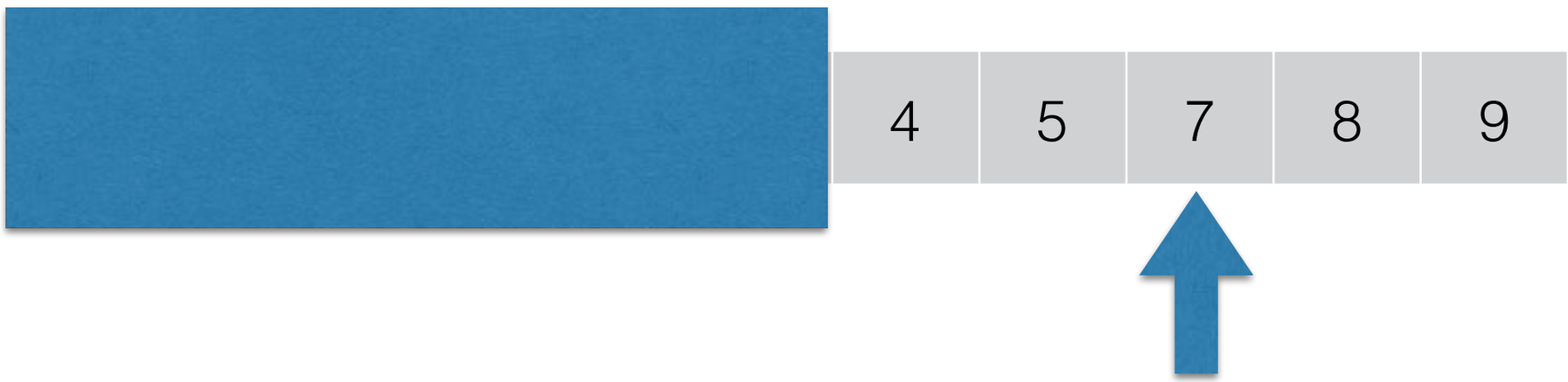


Binary search

- 3 is less than 4
- Since the array is sorted, we know that 4 must come *after* 3
- We can ignore everything before 3

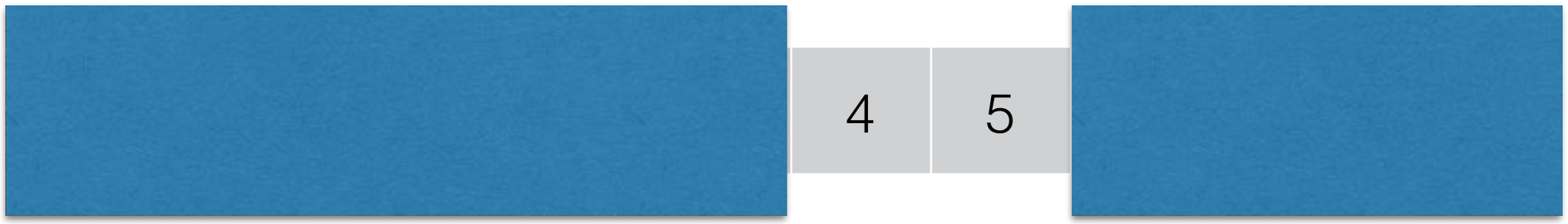
Binary search

- Now we repeat the process
- We look at the element half way along what's left of the array, this happens to be 7



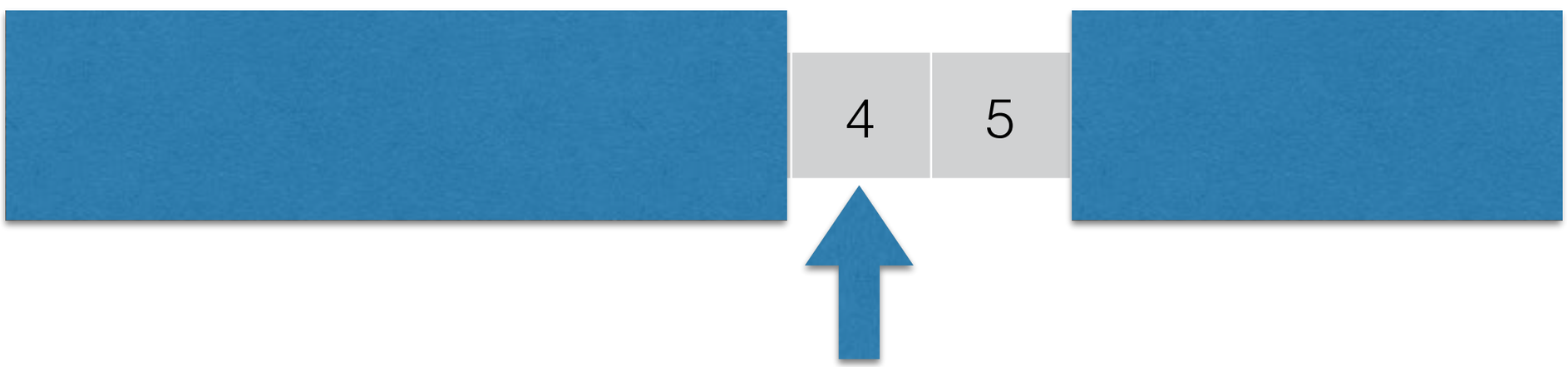
Binary search

- 7 is greater than 4
- Since the array is sorted, we know that 4 must come *before* 7
- We can safely ignore everything after 7



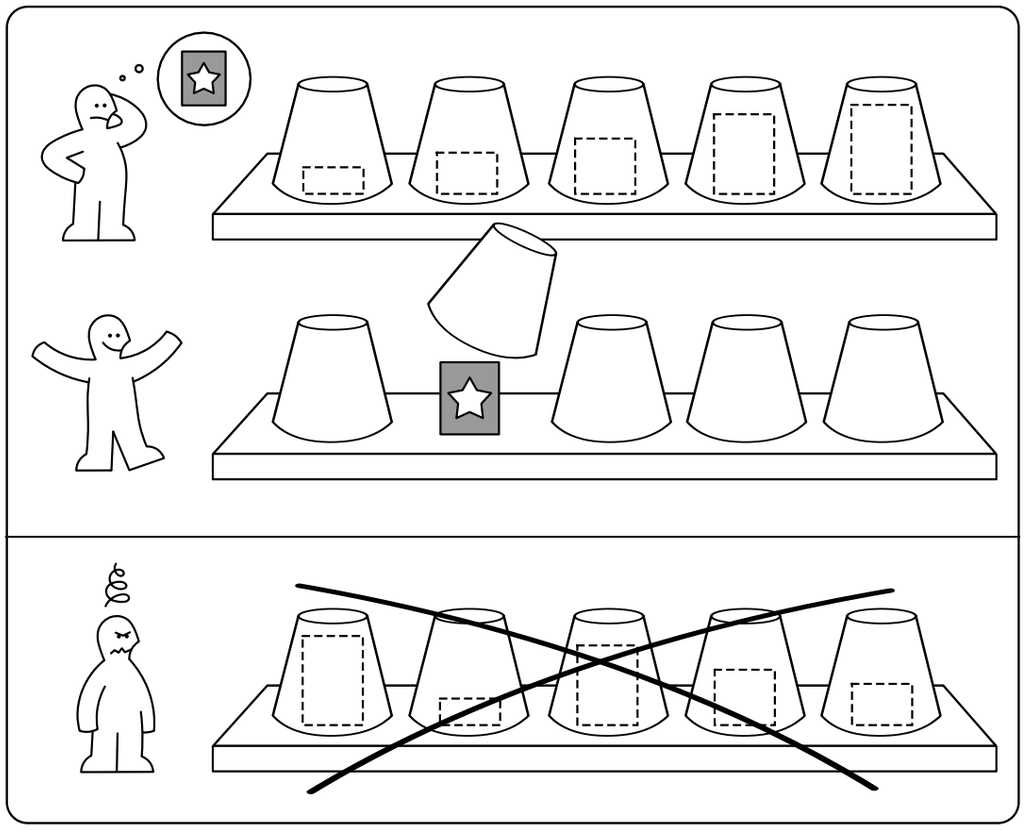
Binary search

- We repeat the process
- We look half way along the array again
- We find 4!

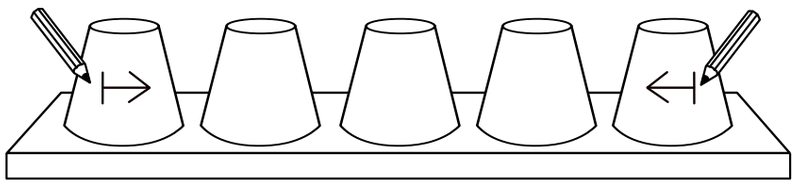


BINÄRY SEARCH

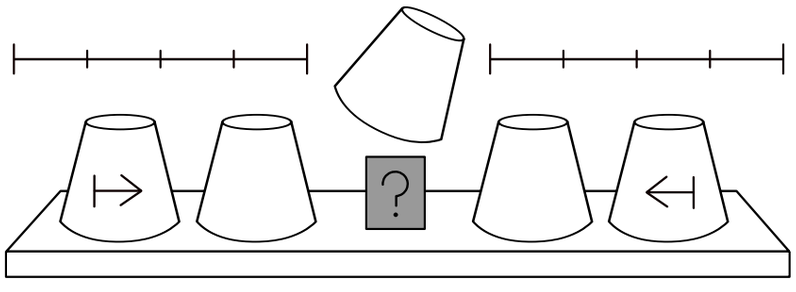
idea-instructions.com/binary-search/
v1.0, CC by-nc-sa 4.0



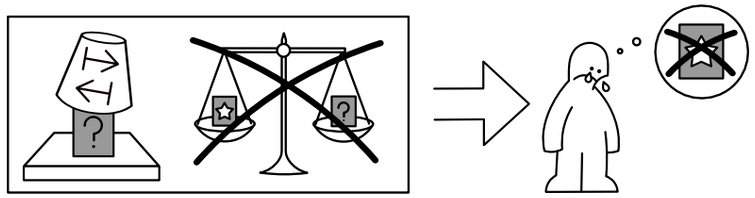
1



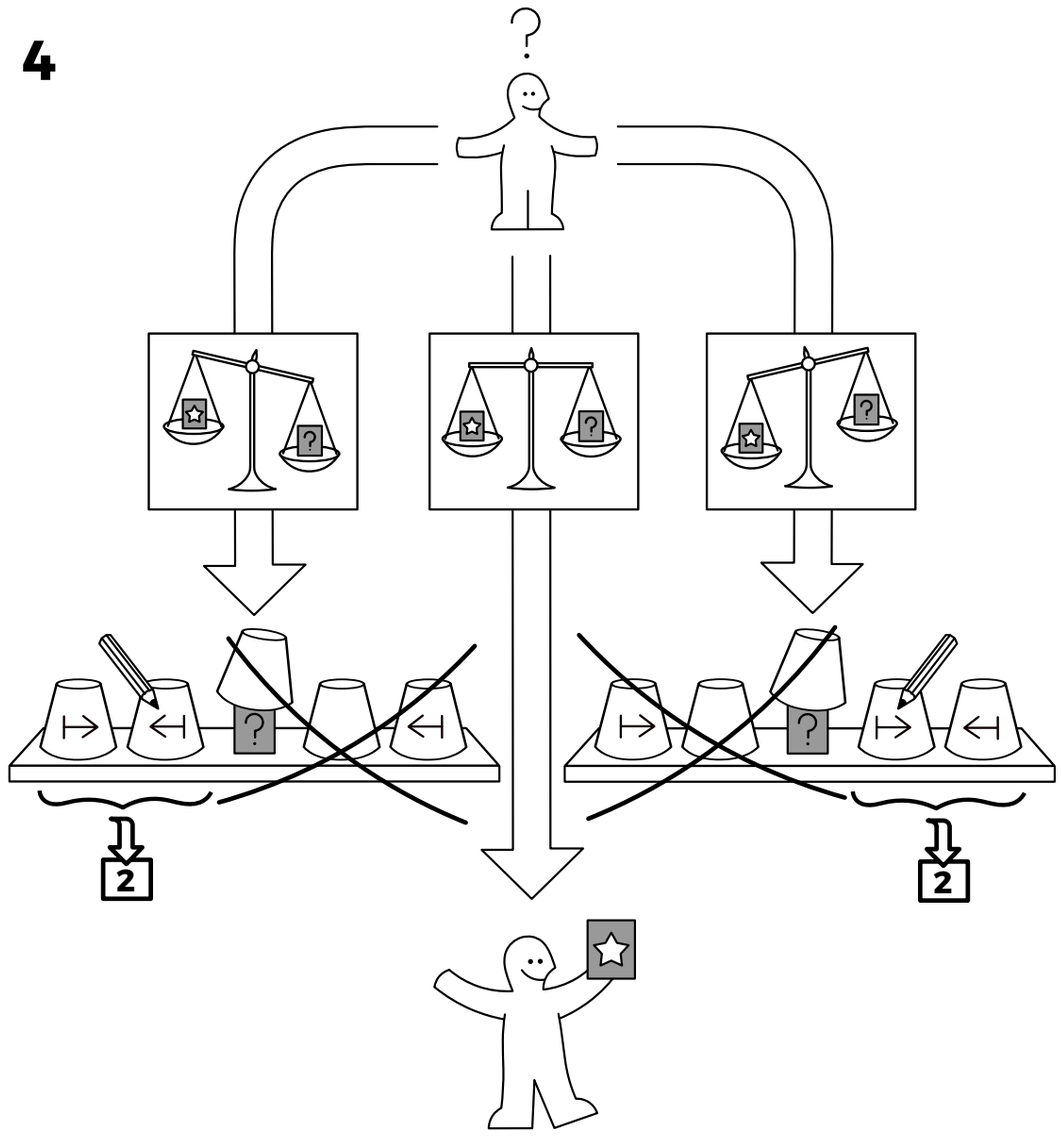
2



3



4



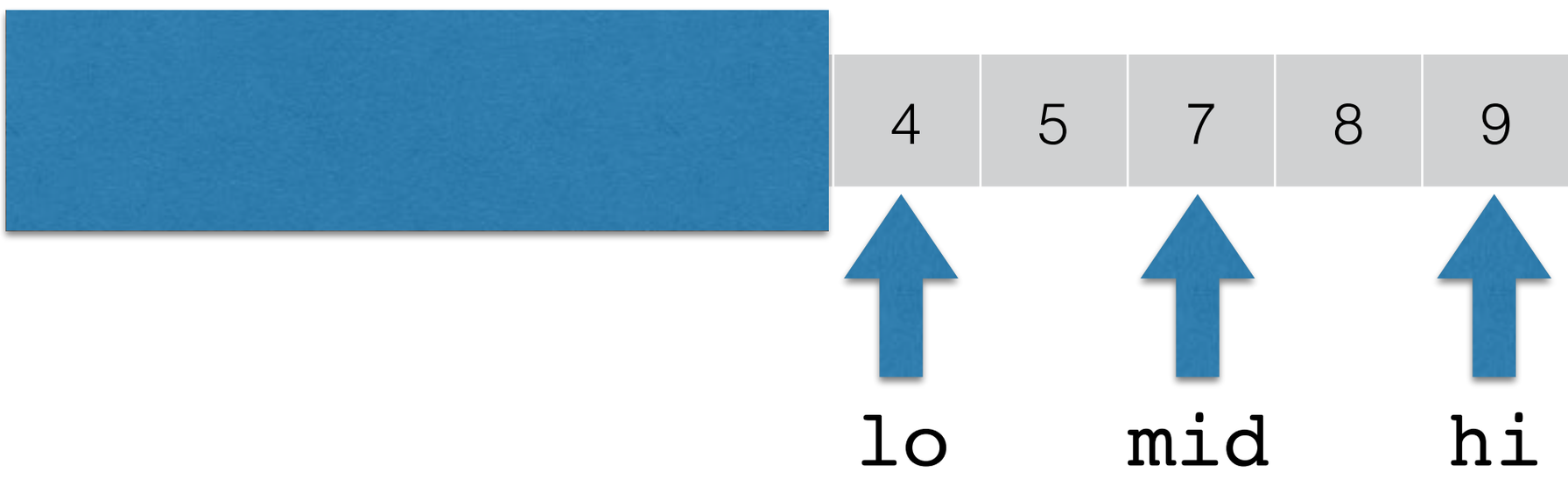
Performance of binary search

- Binary search repeatedly *chops the array in half*
  - If we double the size of the array, we need to look at one more array element
  - With an array of size 2^n, after n tries, we are down to 1 element
  - On an array of size n takes **$O(\log n)$** time!
- On an array of a billion elements, need to search 30 elements (compared to a billion tries for linear search!)

Implementing binary search

- Keep two indices `lo` and `hi`, they represent the part of the array to be searched

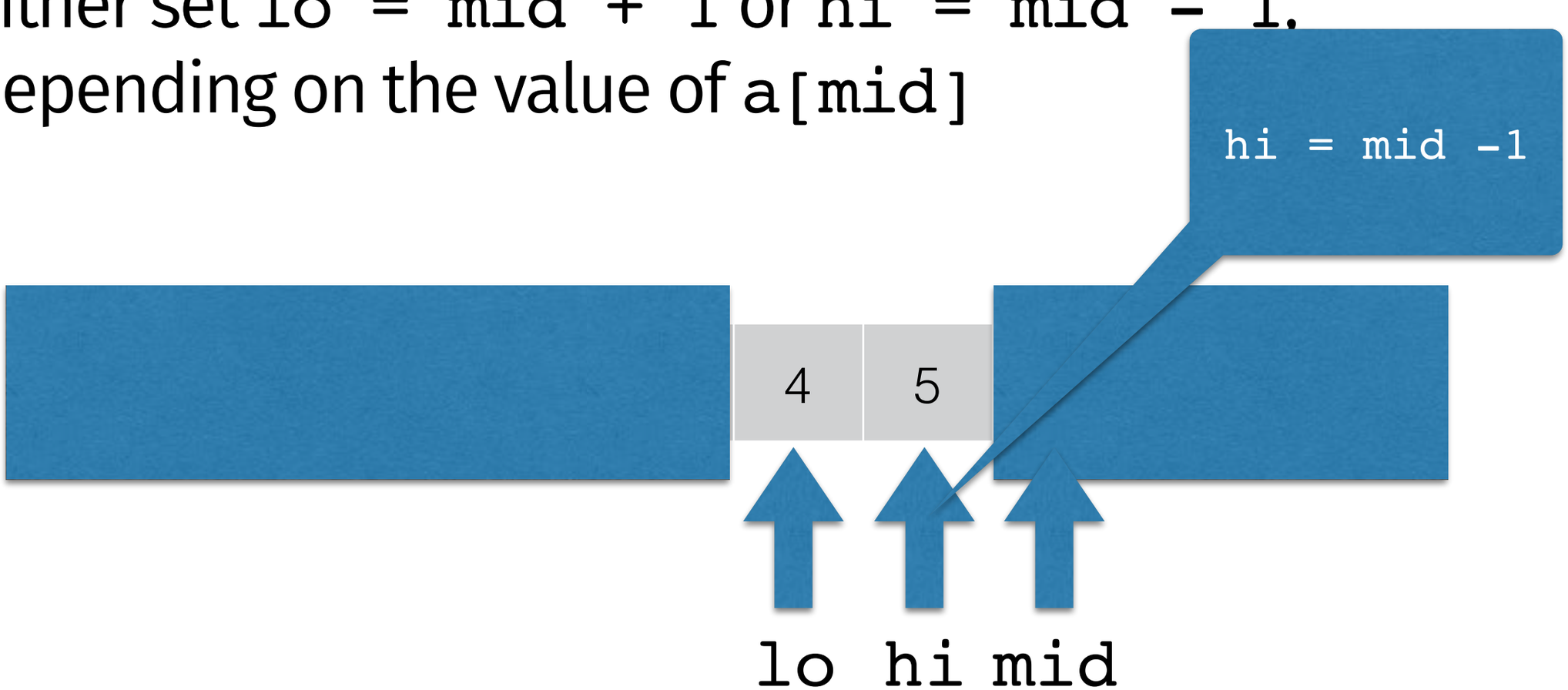
- Let `mid = (lo + hi) / 2` and look at `a[mid]`, then either set `lo = mid + 1` or `hi = mid - 1`, depending on the value of `a[mid]`



Implementing binary search

- Keep two indices `lo` and `hi`, they represent the part of the array to be searched

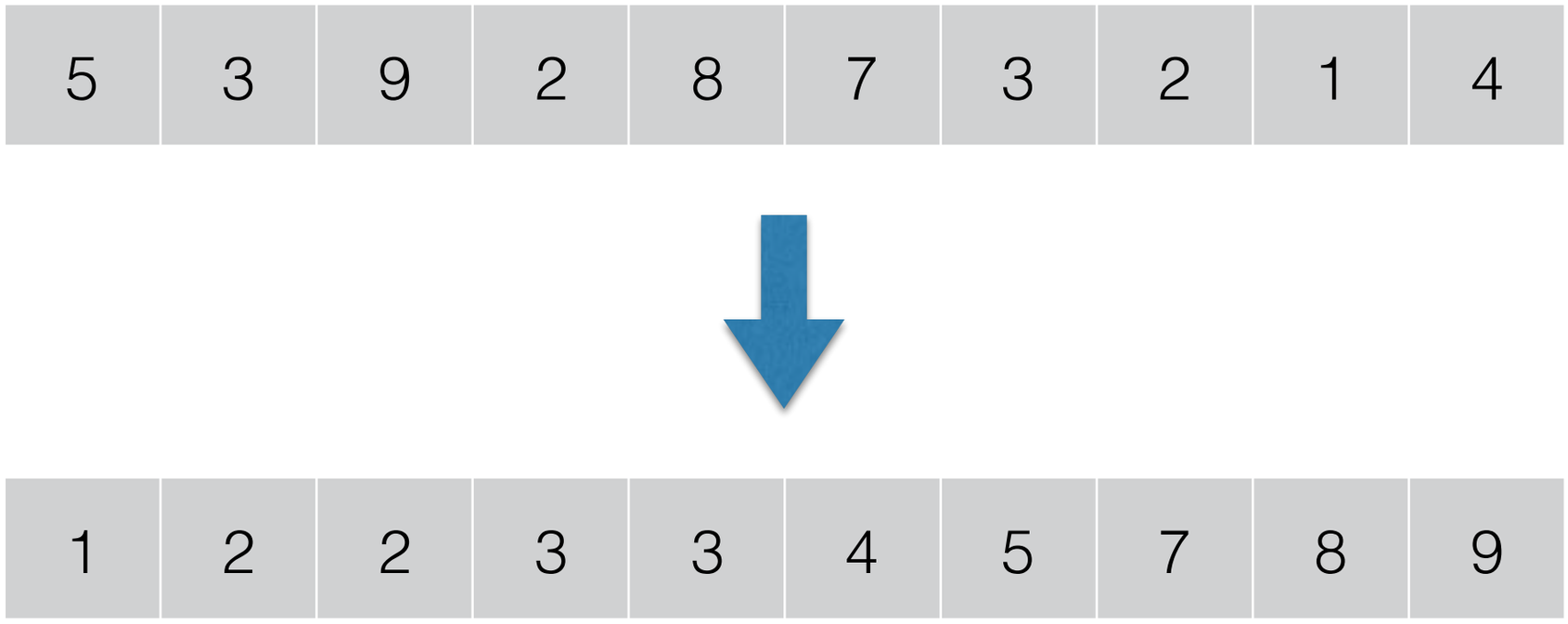
- Let `mid = (lo + hi) / 2` and look at `a[mid]`, then either set `lo = mid + 1` or `hi = mid - 1`, depending on the value of `a[mid]`



Sorting

Sorting

Zillions of sorting algorithms (bubblesort, insertion sort, selection sort, quicksort, heapsort, mergesort, shell sort, counting sort, ...)



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

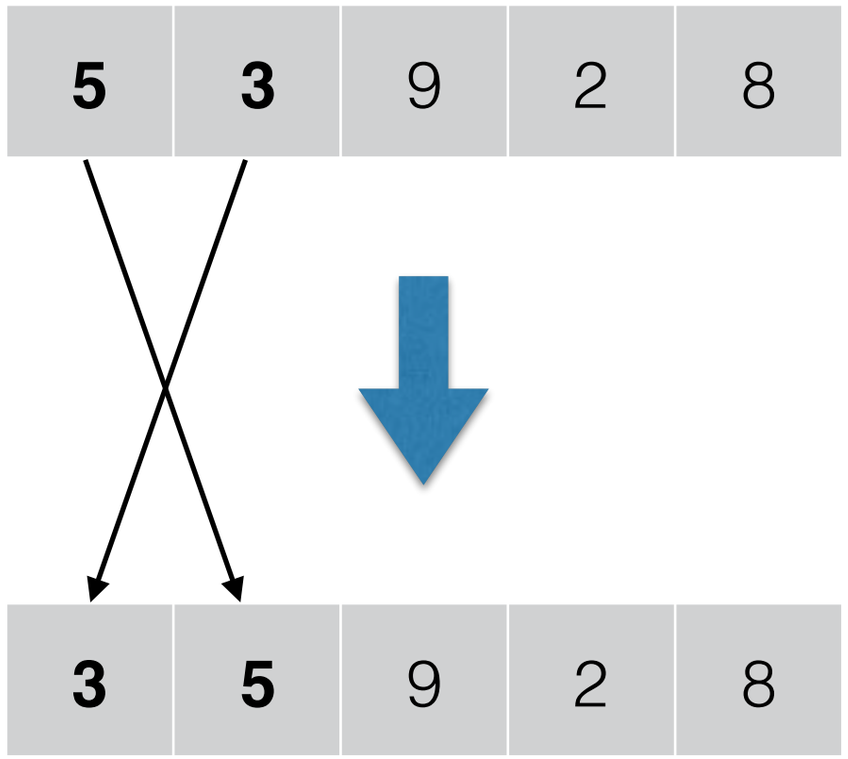
Bubblesort

Bubblesort

- Go through the array, comparing adjacent elements
  - If we find two that are in the wrong order, swap them
- Once we reach the end of the array, go back and start again!

Bubblesort

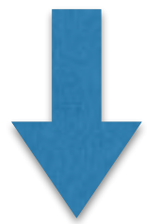
Compare `a[0]` and `a[1]`:



Bubblesort

Compare `a[1]` and `a[2]`:

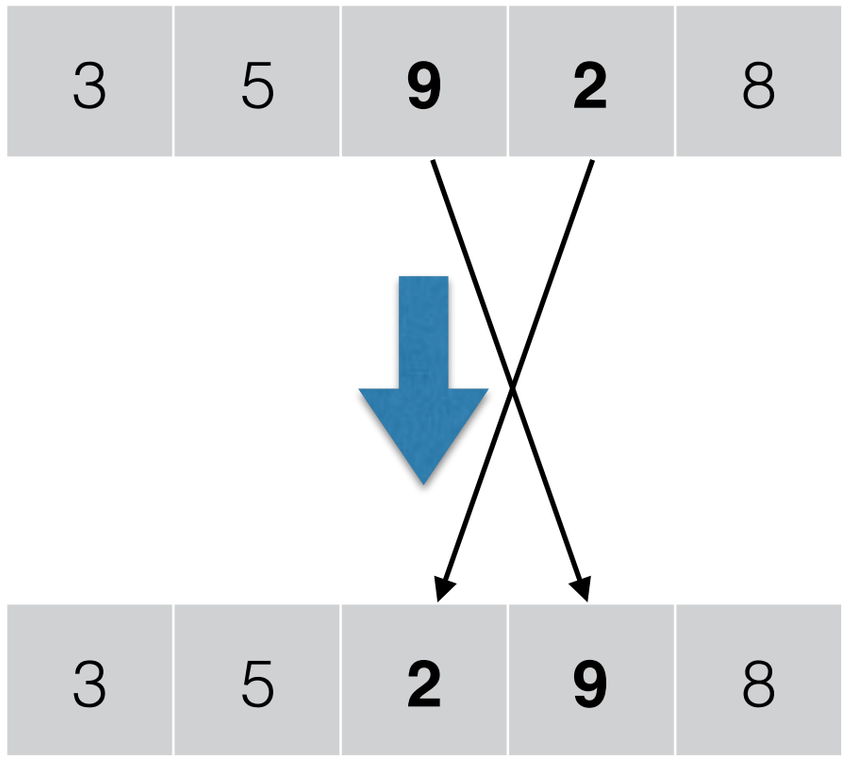
3	**5**	**9**	2	8



3	**5**	**9**	2	8

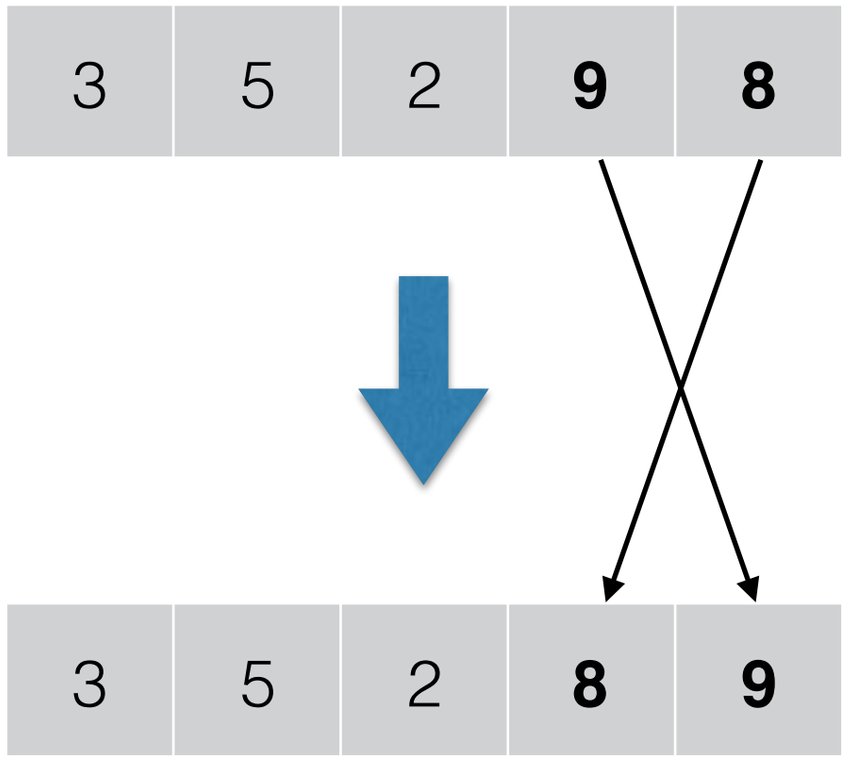
Bubblesort

Compare `a[2]` and `a[3]`:



Bubblesort

Compare `a[3]` and `a[4]`:



Bubblesort

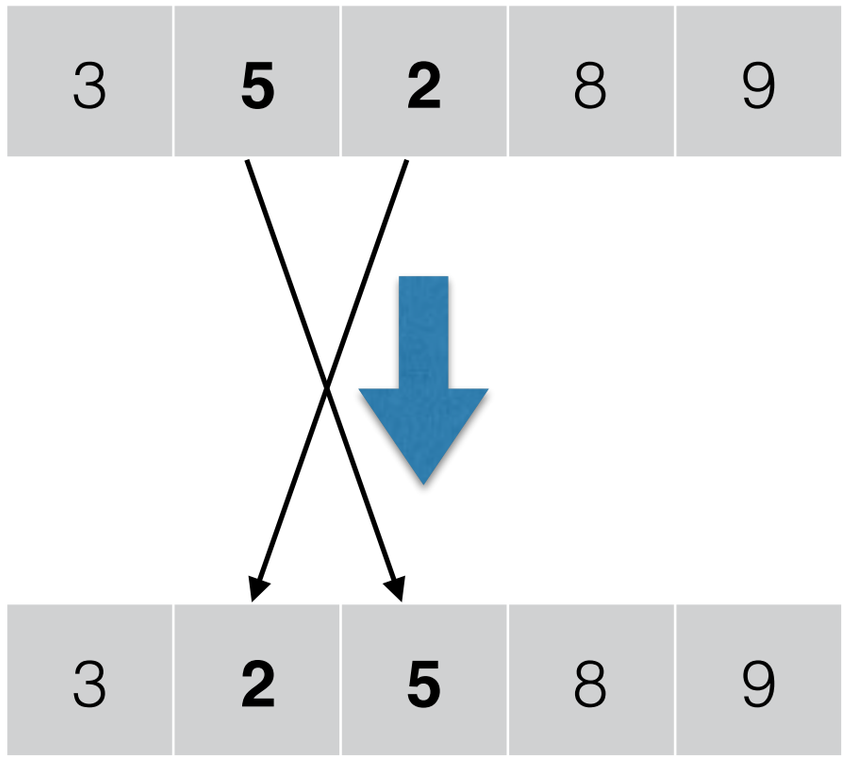
Back to the beginning!

3	**5**	2	8	9

3	**5**	2	8	9

Bubblesort

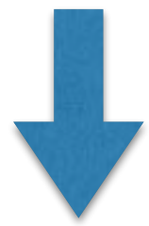
Compare `a[1]` and `a[2]`:



Bubblesort

Compare `a[2]` and `a[3]`:

3	2	**5**	**8**	9



3	2	**5**	**8**	9

Bubblesort

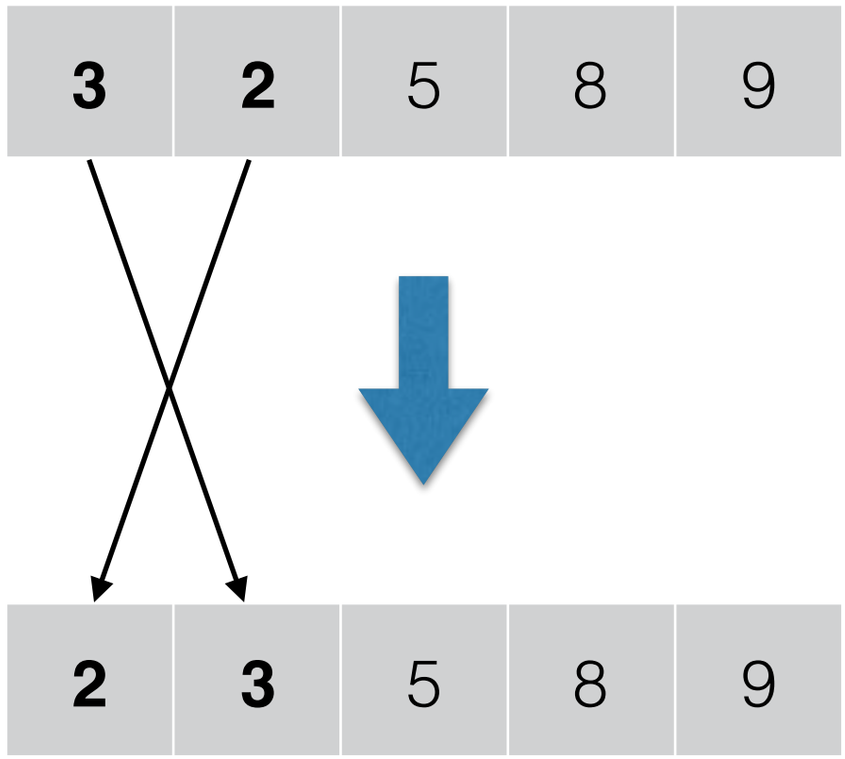
Compare `a[3]` and `a[4]`:

3	2	5	**8**	**9**

3	2	5	**8**	**9**

Bubblesort

Back to the beginning!



Bubblesort

- How do we know when to stop going back to the beginning?
  - When the array is sorted
- How many loops until that happens?
  - Each time we loop through the array, at least one more element ends up in the right place: the biggest element that was in the wrong place before
- So repeat as many times as there are elements in the input array

```
for k = 0 to array.length-1
  for i = 0 to array.length-2
    if array[i] < array[i+1]
      swap array[i] and array[i+1]
```

Insertion sort

Insertion sort

Imagine someone is dealing you cards. Whenever you get a new card you put it into the right place in your hand:

This is the idea of *insertion sort*.

Insertion sort

Sorting

5	3	9	2	8

Start by ”picking up” the 5:

5

Insertion sort

Sorting

5	3	9	2	8

Then insert 3 into the right place:

3	5

Insertion sort

Sorting

5	3	9	2	8

Then the 9:

3	5	9

Insertion sort

Sorting

5	3	9	2	8

Then the 2:

2	3	5	9

Insertion sort

Sorting

5	3	9	2	8

Finally the 8:

2	3	5	8	9

Complexity of insertion sort

- Insertion sort does n insertions for an array of size n
- Does this mean it is $O(n)$?
  - No! An insertion is not constant time.
- To insert into a sorted array, you must move all the elements up one, which is $O(n)$
- Thus total is $O(n^2)$

In-place insertion sort

- This version of insertion sort needs to make a new array to hold the result
- An *in-place* sorting algorithm is one that doesn't need to make temporary arrays
  - Has the potential to be more efficient
- Let's make an in-place insertion sort!
- Basic idea: loop through the array, and insert each element into the part which is already sorted

In-place insertion sort

5	3	9	2	8

- The first element of the array is sorted:

5	3	9	2	8

Dark gray part is sorted

In-place insertion sort

5	3	9	2	8

- Insert the 3 into the correct place:

3	5	9	2	8

In-place insertion sort

- Insert the 9 into the correct place:

In-place insertion sort

5	3	9	2	8

- Insert the 2 into the correct place:

2	3	5	9	8

In-place insertion sort

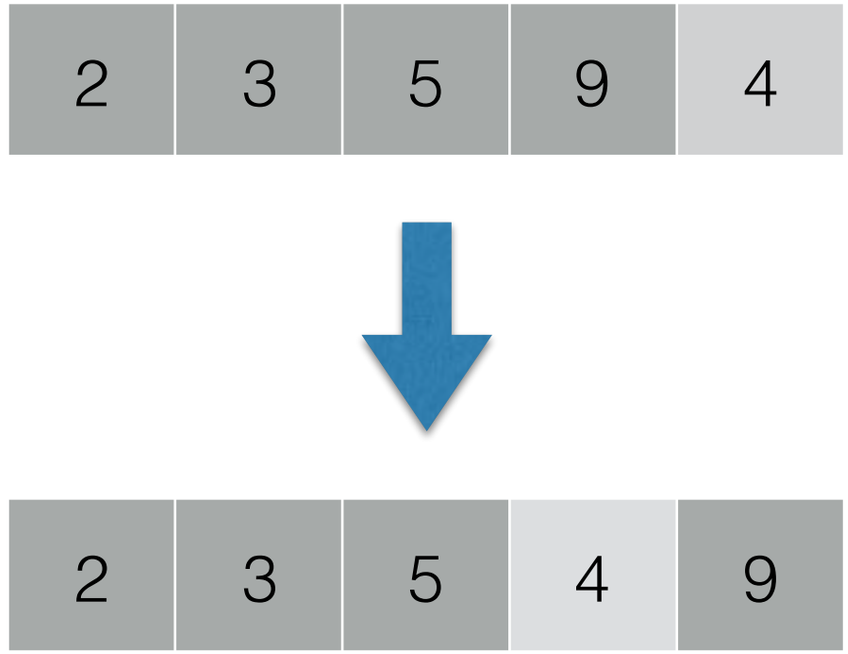
5	3	9	2	8

- Insert the 8 into the correct place:

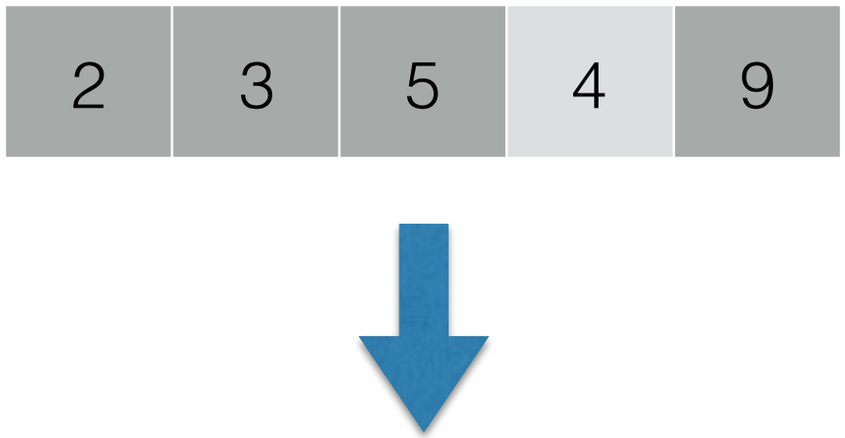
2	3	5	8	9

In-place insertion sort

- One way to do it: repeatedly *swap* the element with its neighbour on the left, until it's in the right position



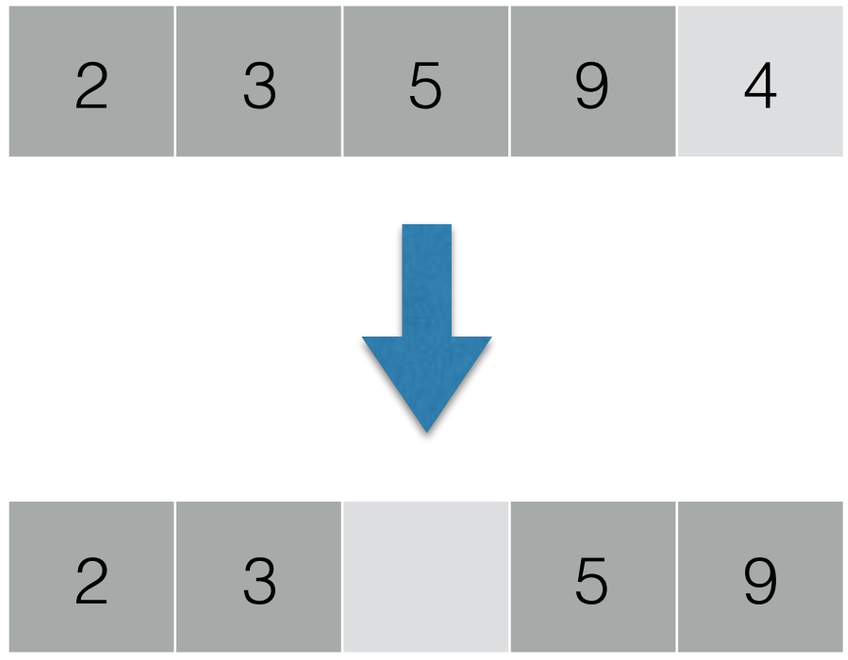
In-place insertion sort



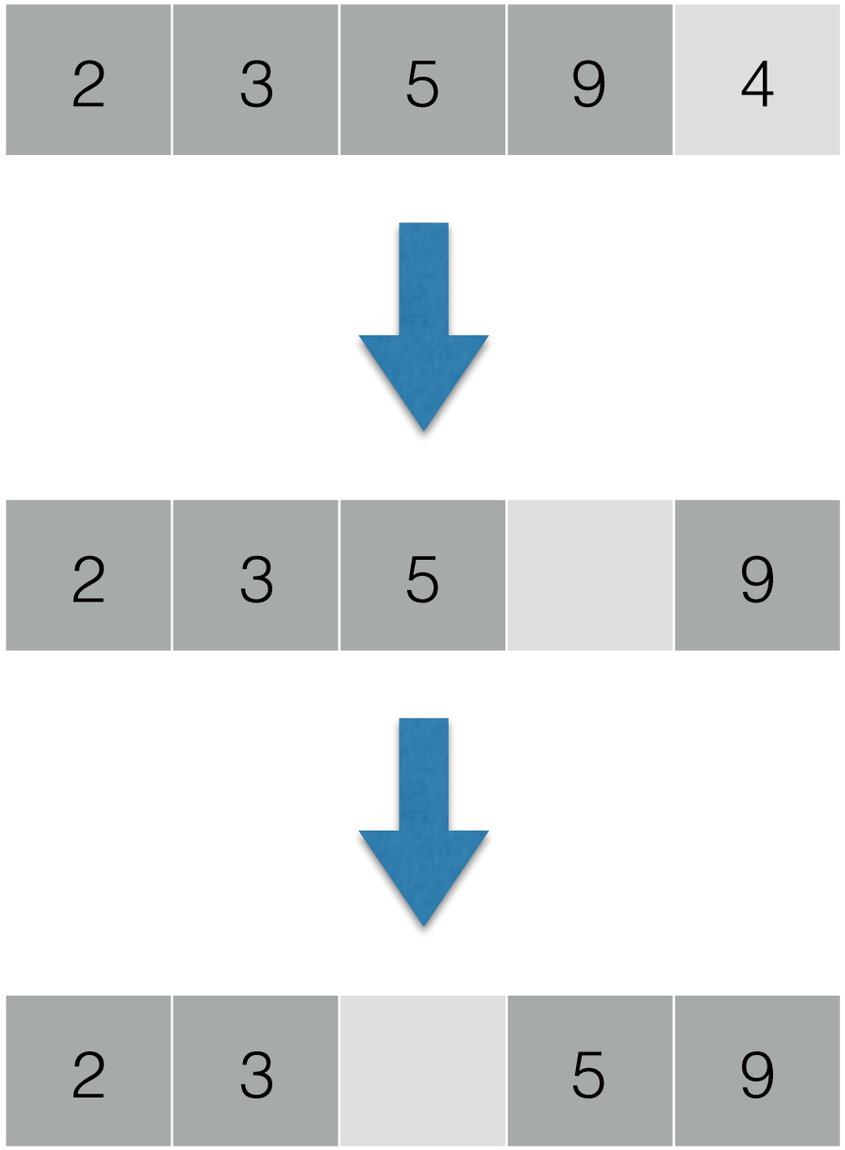

```
while n > 0 and array[n] > array[n-1]
   swap array[n] and array[n-1]
   n = n-1
```

In-place insertion sort

- An improvement: instead of swapping, move elements upwards to make a “hole” where we put the new value



In-place insertion sort



In-place insertion sort

```
for i = 1 to n-1
   insert array[i] into array[0..i)
```

This notation means `0, 1, …, i-1`

- An aside: we have the invariant that `array[0..i)` is sorted
  - An invariant is something that holds whenever the loop body starts to run
  - Initially, `i = 1` and `array[0..1)` is sorted
  - As the loop runs, more and more of the array becomes sorted
  - When the loop finishes, `i = n`, so `array[0..n)` is sorted – the whole array!

Selection sort

Selection sort

- Find the smallest element of the array, and delete it
- Find the smallest remaining element, and delete it
- And so on...

- Quiz: complexity?
- Finding the smallest element is $O(n)$, so total complexity is $O(n^2)$

Selection sort

Sorting

5	3	9	2	8

The smallest element is 2:

2

We also delete 2 from the input array

Selection sort

Sorting

5	3	9	8

Now smallest element is 3:

2	3

We delete 3 from the input array

Selection sort

Sorting

5	9	8

Now smallest element is 5:

2	3	5

We delete from the input array (... and so on)

In-place selection sort

- Instead of deleting the smallest element, *swap it* with the first element!

- The next time round, ignore the first element of the array: we know it's the smallest one

- Instead, find the smallest element of the *rest* of the array, and swap it with the second element

In-place insertion sort

5	3	9	2	8

The smallest element is 2:

2	3	9	**5**	8

In-place insertion sort

The smallest element in the rest of the array is 3:

In-place insertion sort

The smallest element in the rest of the array is 5:

In-place insertion sort

The smallest element in the rest of the array is 8:

In-place insertion sort

```
for i = 0 to a.length-1
  find the smallest element
    in a[i .. a.length)
  swap it with a[i]
```

Comparing the sorting algorithms

- All the algorithms so far are $O(n^2)$ in the worst case
- One of them is $O(n)$ in the best case (a sorted array) – which?
  - Answer: insertion sort
  - This makes insertion sort the best of our three algorithms – it's actually a fast sorting algorithm in general for small lists
  - The other two are bad, but selection sort is the basis for a better algorithm, heapsort