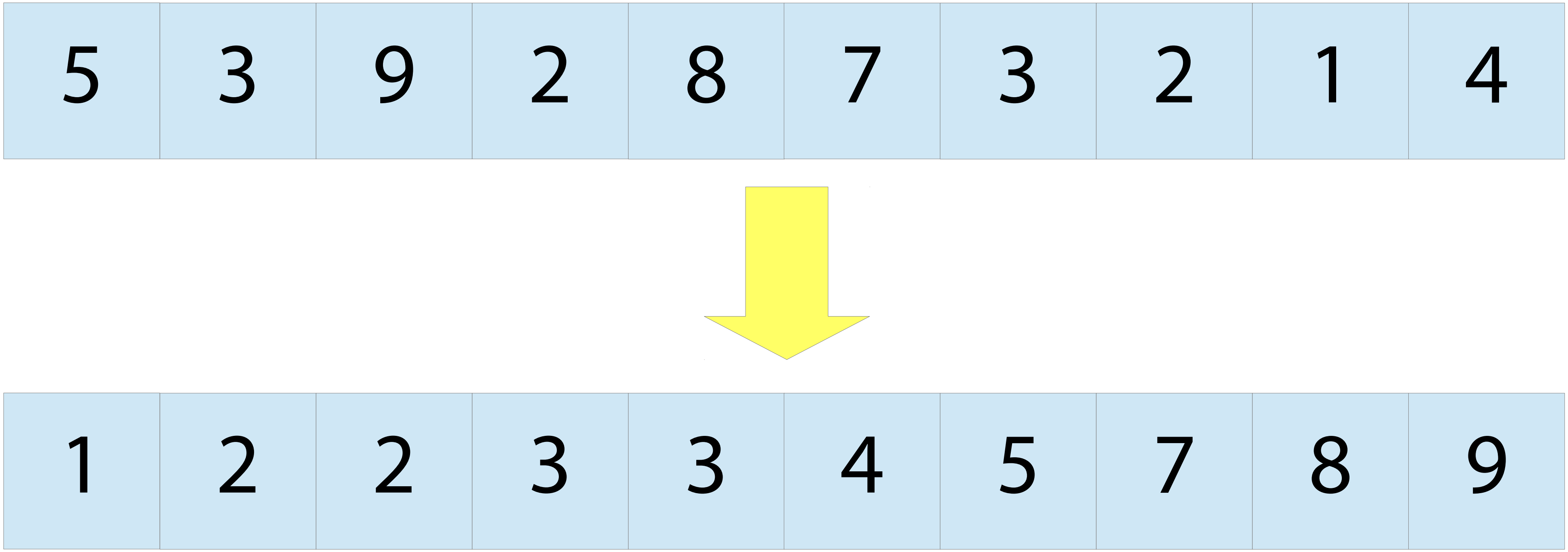
Sorting

(Weiss chapter 8.1 – 8.3)

Sorting



5	3	9	2	8	7	3	2	1	4

1	2	2	3	3	4	5	7	8	9

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

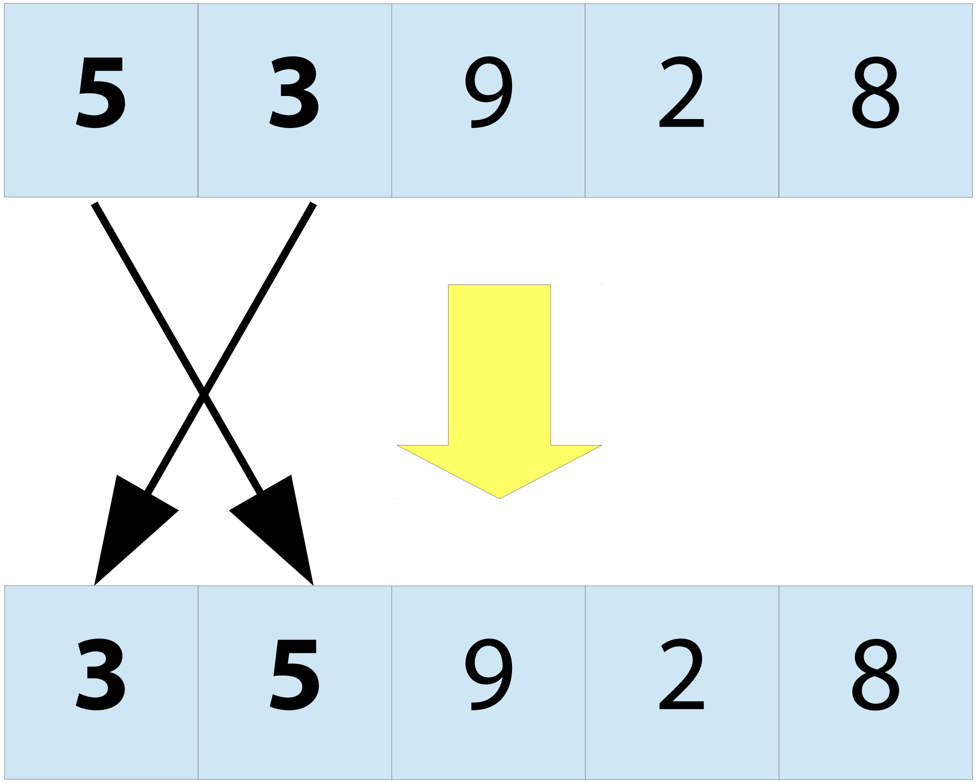
Go through the array, comparing adjacent elements

- 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]`:

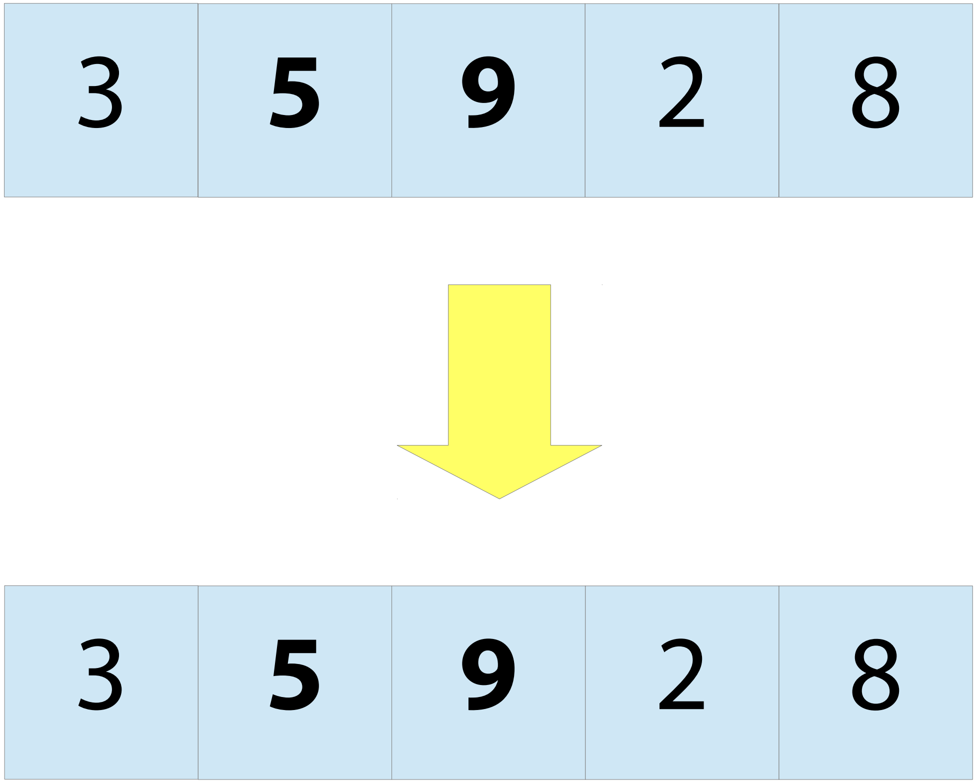


5	**3**	9	2	8

3	**5**	9	2	8

Bubblesort

Compare a[1] and a[2]:

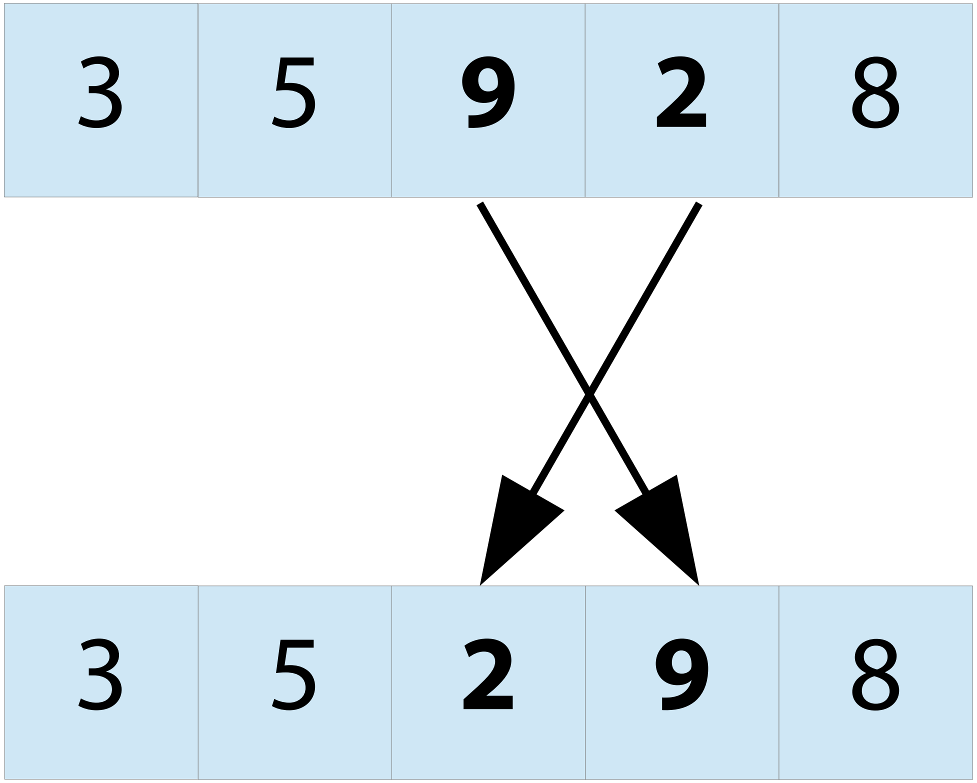


3	**5**	**9**	2	8

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

Bubblesort

Compare a[2] and a[3]:

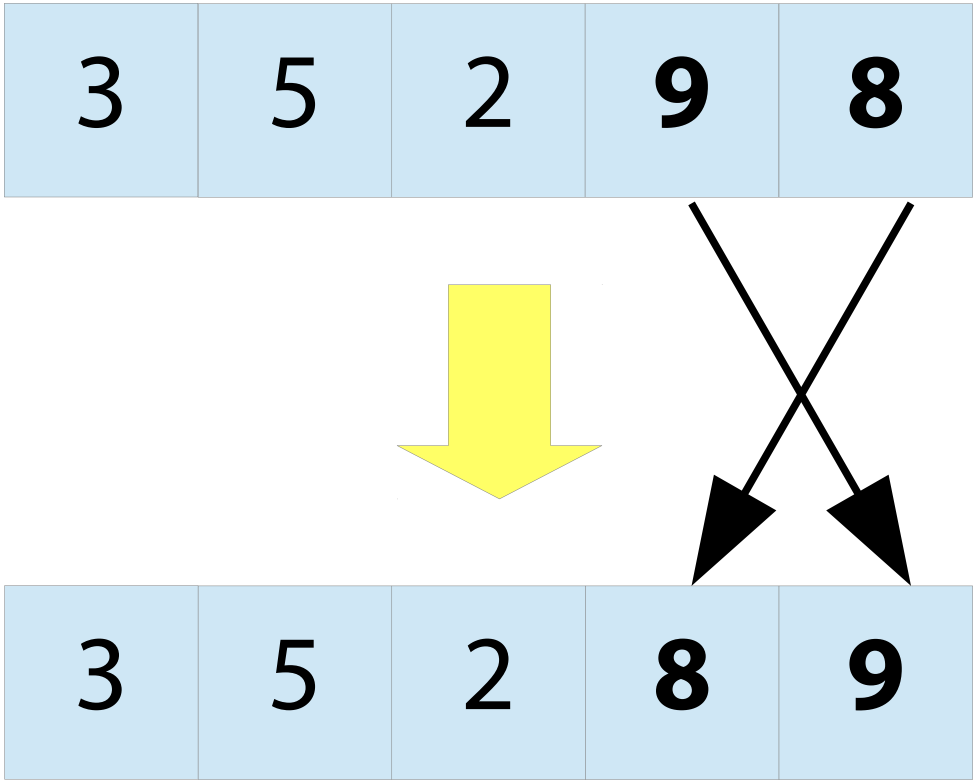


3	5	**9**	**2**	8

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

Bubblesort

Compare a[3] and a[4]:

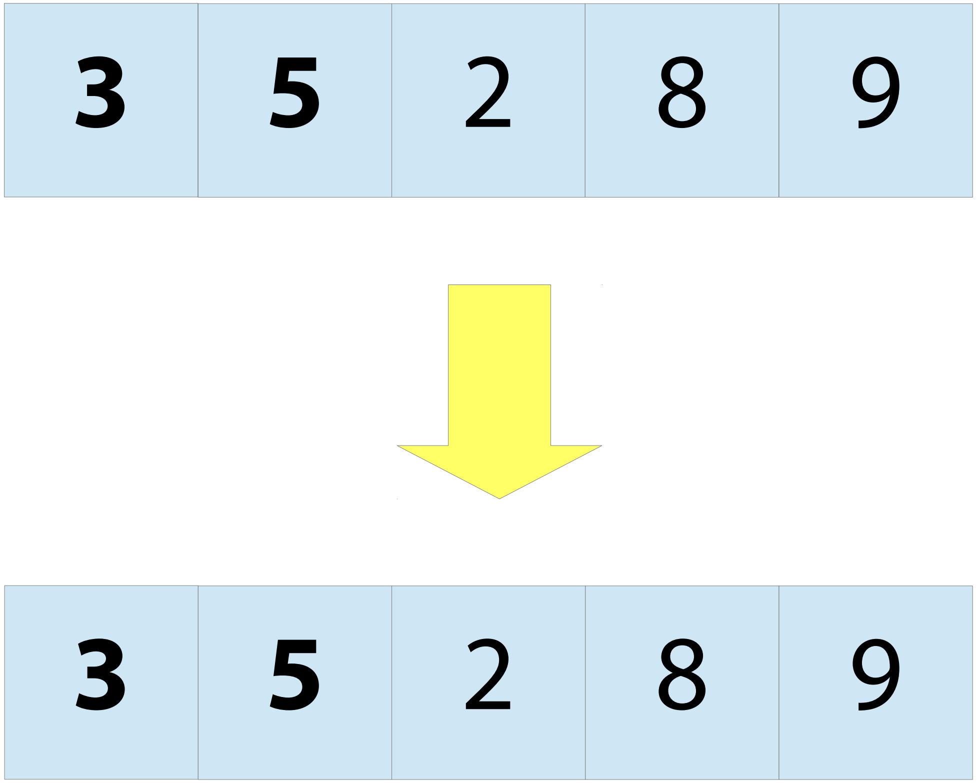


3	5	2	**9**	**8**

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

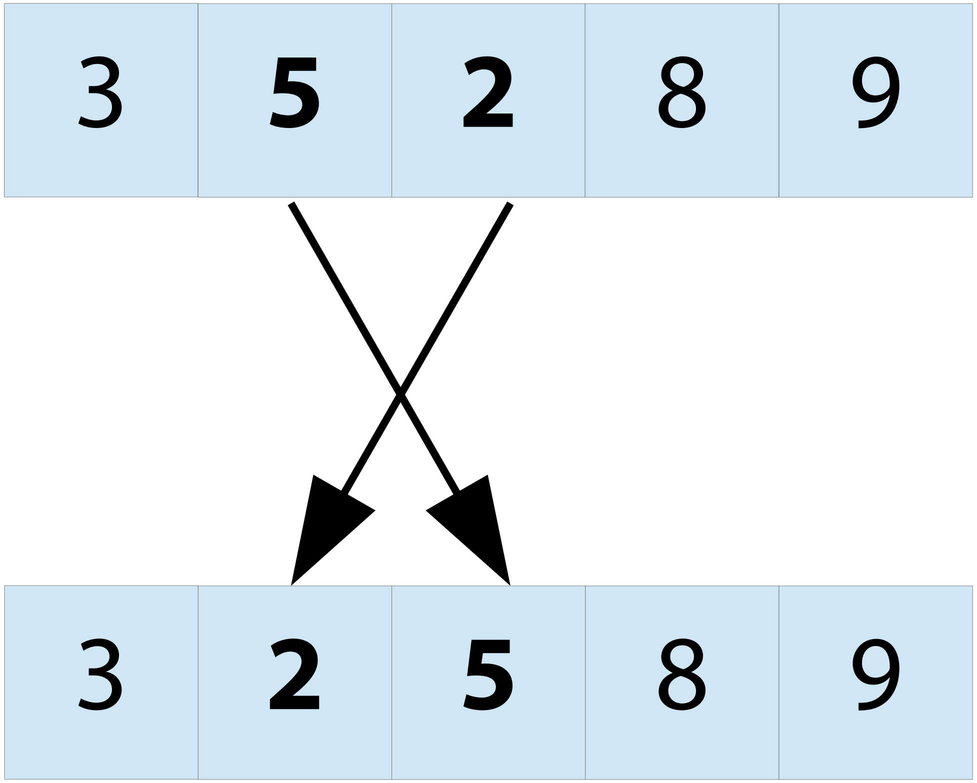
Bubblesort

Back to the beginning!



Bubblesort

Compare a[1] and a[2]:

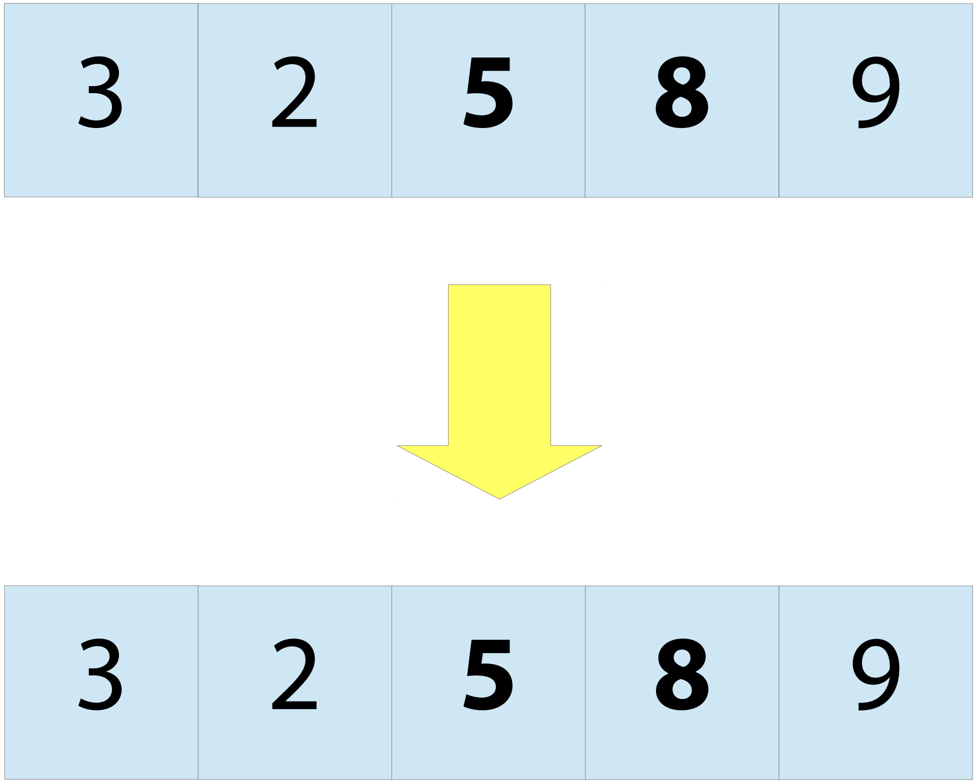


3	**5**	**2**	8	9

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

Bubblesort

Compare a[2] and a[3]:

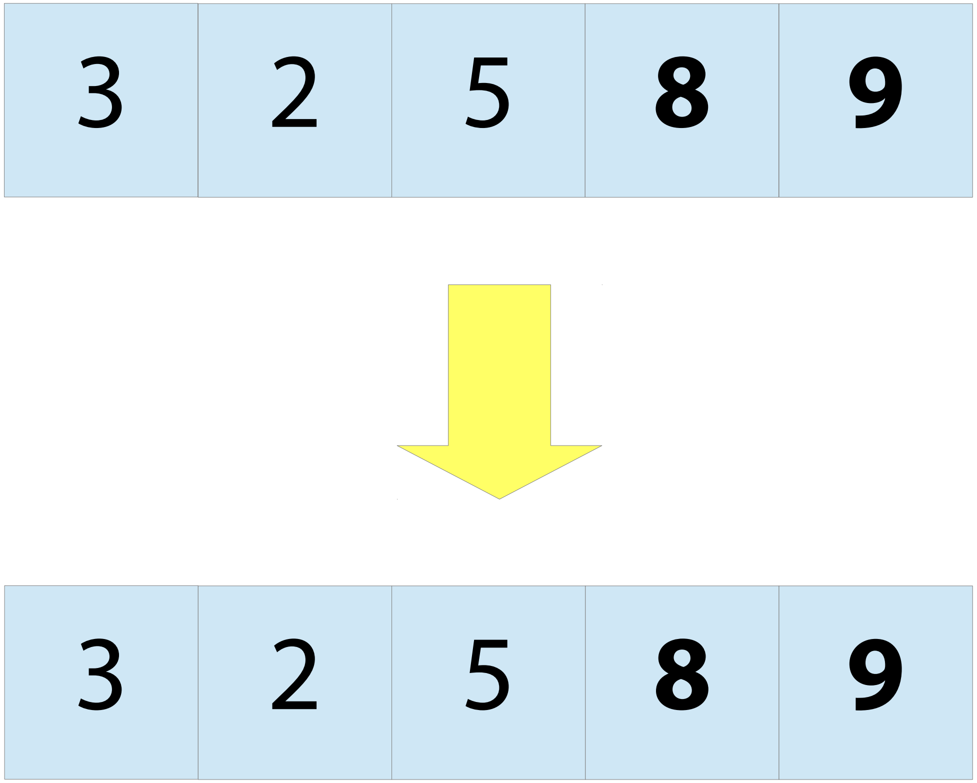


3	2	**5**	**8**	9

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

Bubblesort

Compare a[3] and a[4]:

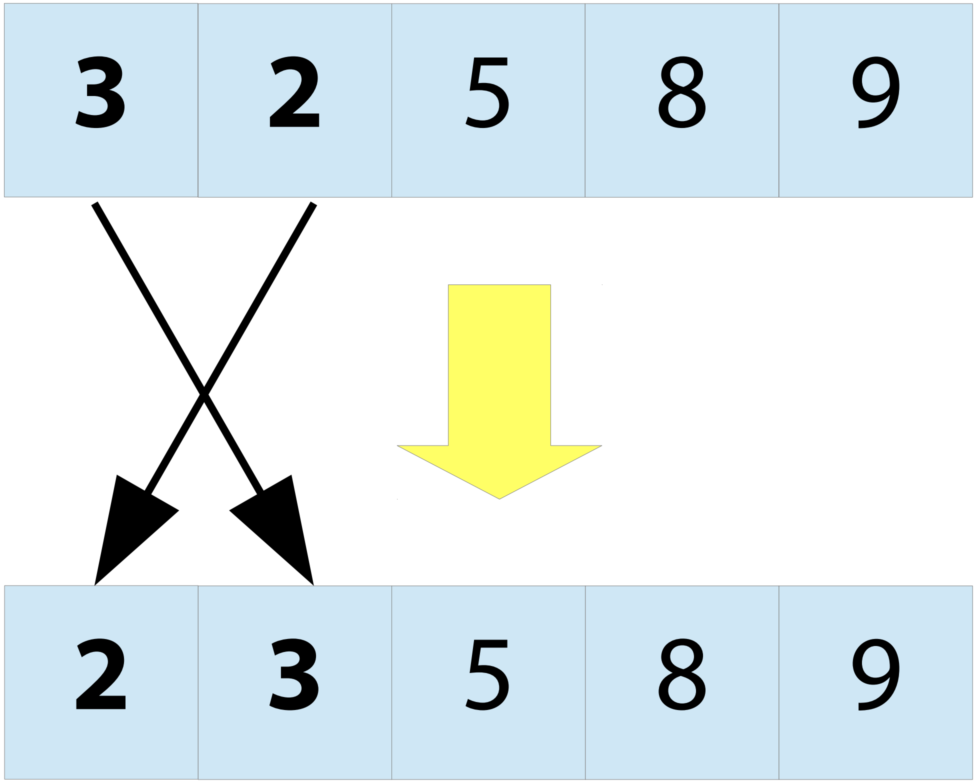


3	2	5	**8**	**9**

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

Bubblesort

Back to the beginning!



3	**2**	5	8	9

2	**3**	5	8	9

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

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 the 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

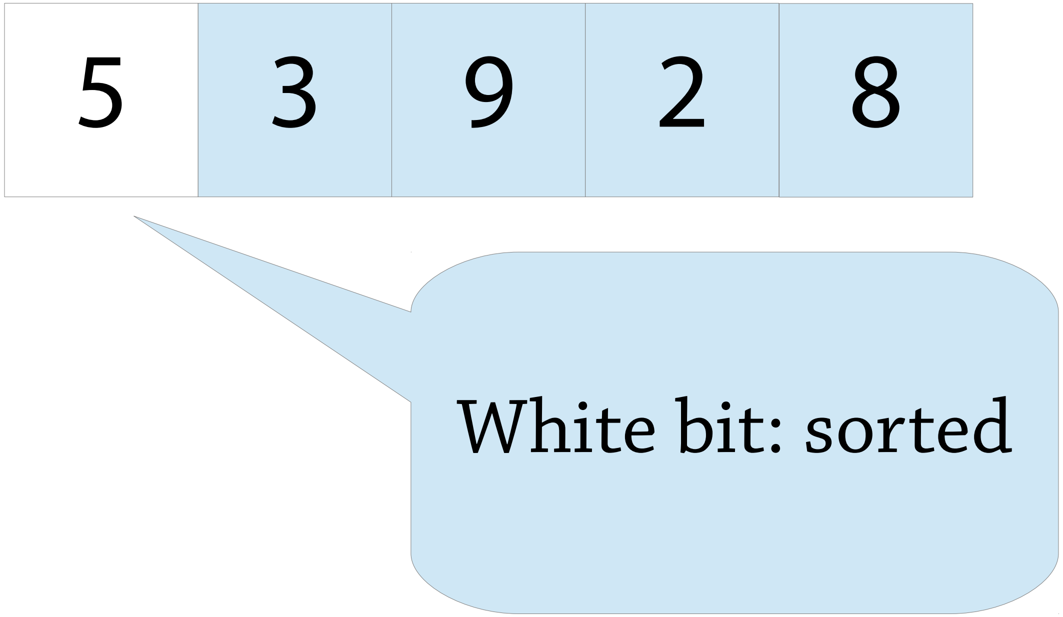
Let's make an in-place insertion sort!

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

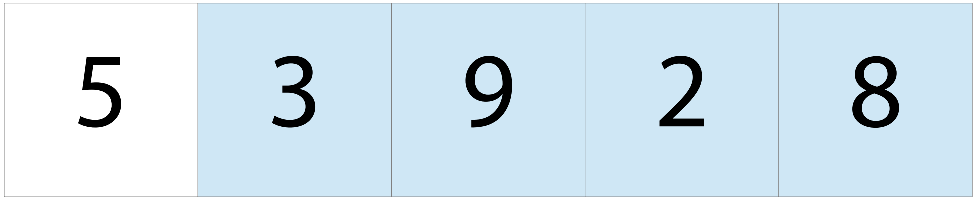
The first element of the array is sorted:



5	3	9	2	8

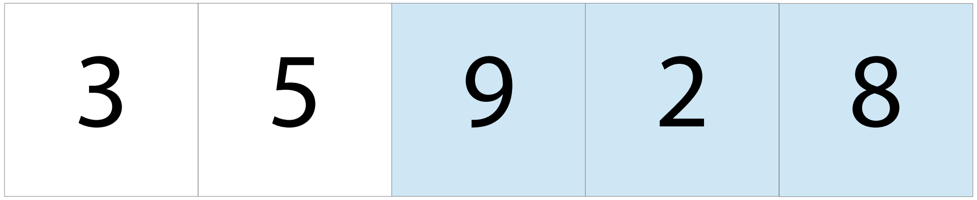
White bit: sorted

In-place insertion sort



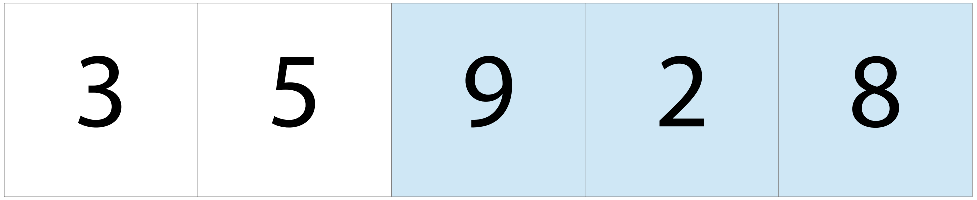
5	3	9	2	8

Insert the 3 into the correct place:



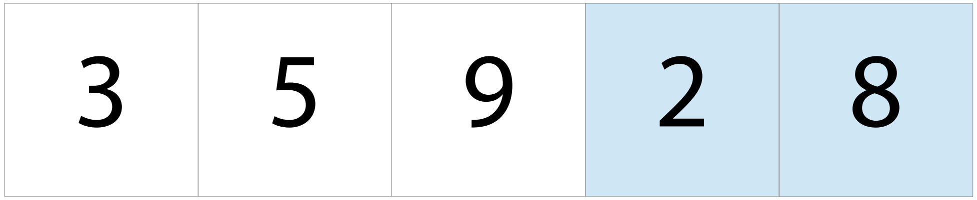
3	5	9	2	8

In-place insertion sort



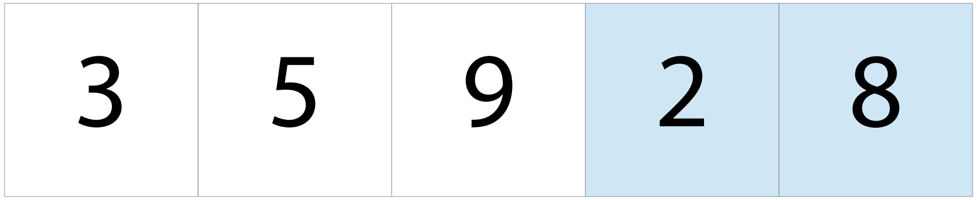
3	5	9	2	8

Insert the 9 into the correct place:



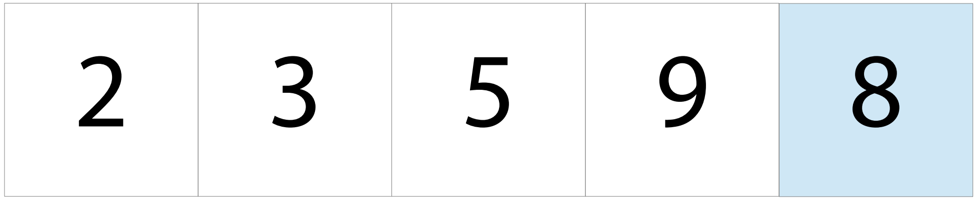
3	5	9	2	8

In-place insertion sort



3	5	9	2	8

Insert the 2 into the correct place:



2	3	5	9	8

In-place insertion sort

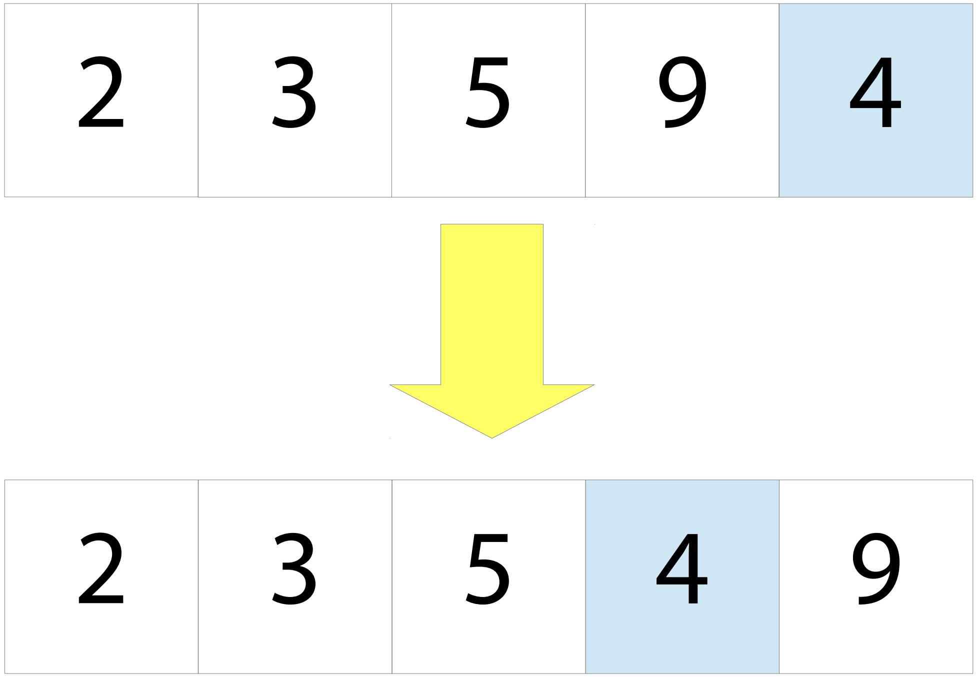
2	3	5	9	8

Insert the 8 into the correct place:

2	3	5	8	9

In-place insertion

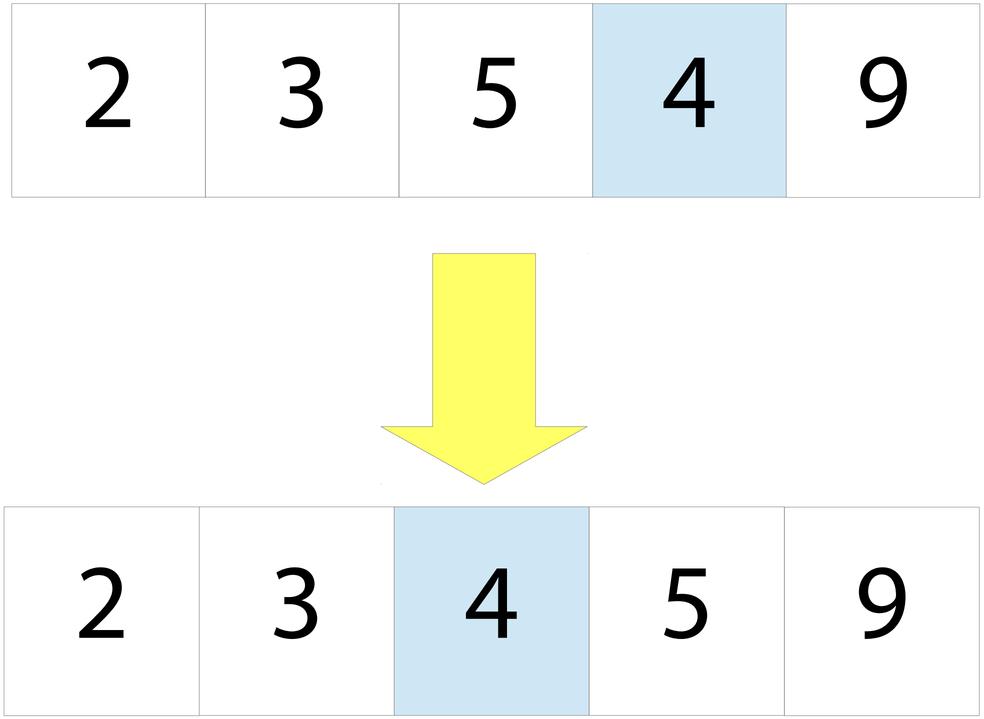
One way to do it: repeatedly swap the element with its neighbour on the left, until it's in the right position



2	3	5	9	4

2	3	5	4	9

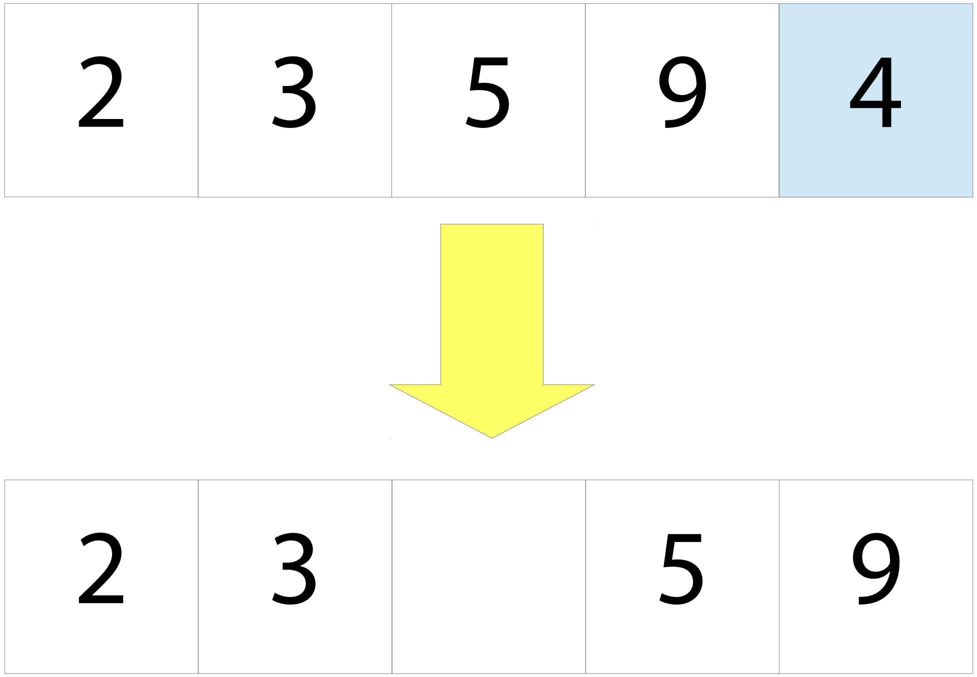
In-place insertion



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

In-place insertion

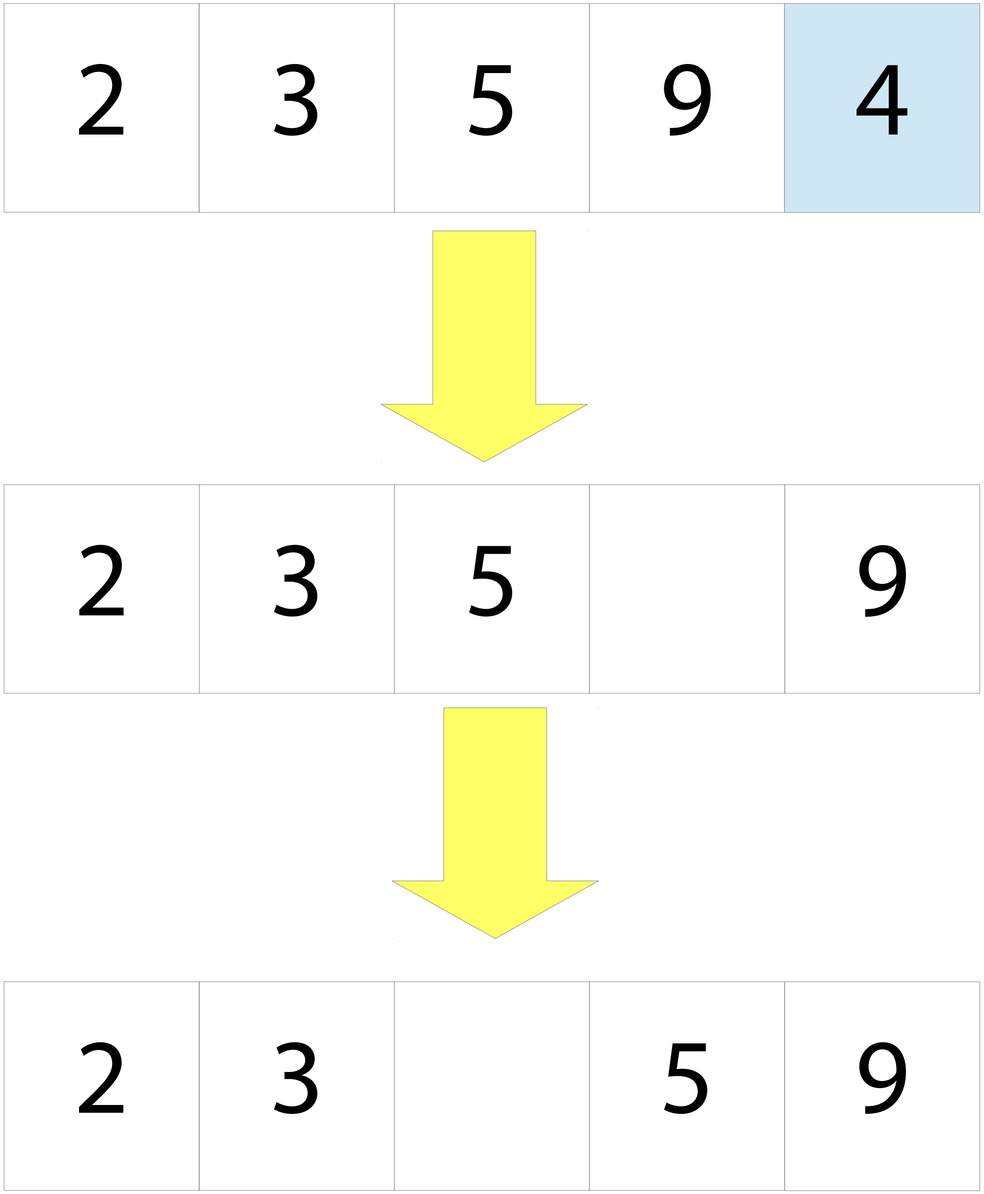
An improvement: instead of swapping, move elements upwards to make a “hole” where we put the new value



2	3	5	9	4

2	3		5	9

In-place insertion



2	3	5	9	4

2	3	5		9

2	3		5	9

In-place insertion so

This notation
means
0, 1, ..., i-1

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

Find the smallest element of the array, and delete it

Find the smallest remaining element, and delete it

And so on

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 the smallest element is 3:

2	3

We delete 3 from the input array.

Selection sort

Sorting

5	9	8

:

Now the smallest element is 5:

2	3	5

We delete 5 from the input array.

(...and so on)

In-place selection sort

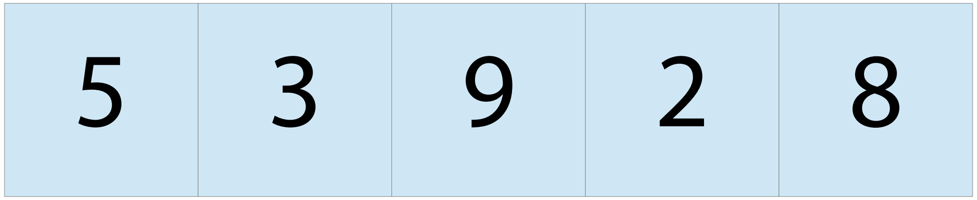
Instead of deleting the smallest element, *swap it* with the first element!

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 selection sort

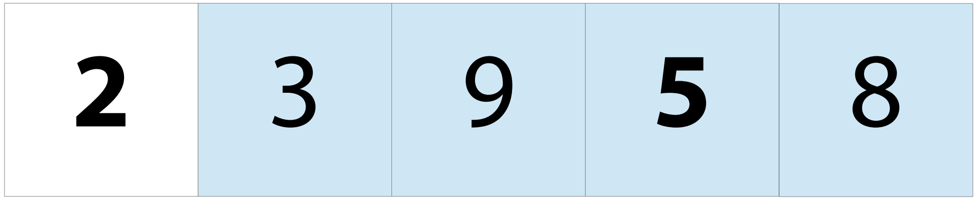
Sorting



5	3	9	2	8

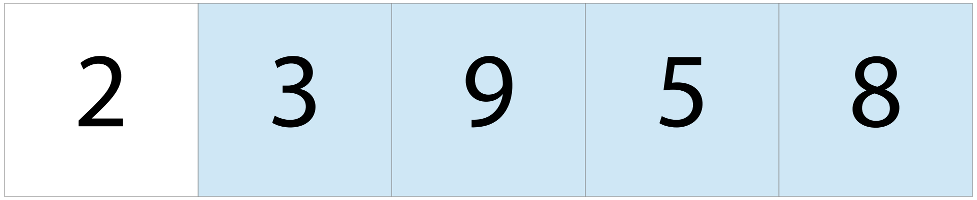
:

The smallest element is 2:



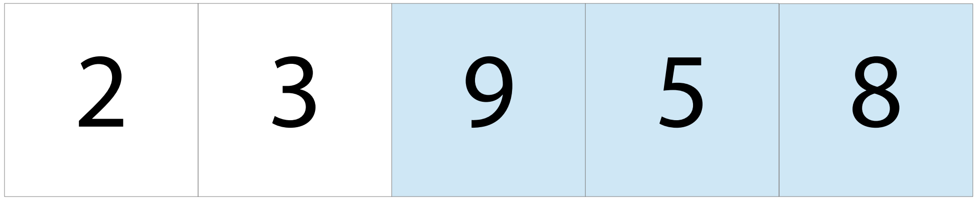
2	3	9	**5**	8

In-place selection sort



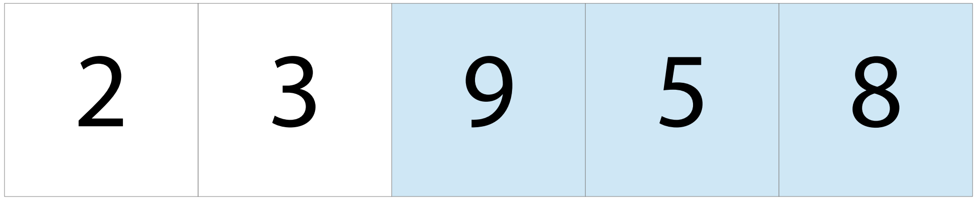
2	3	9	5	8

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



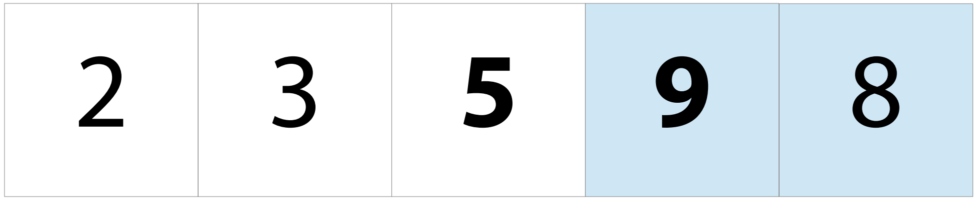
2	3	9	5	8

In-place selection sort



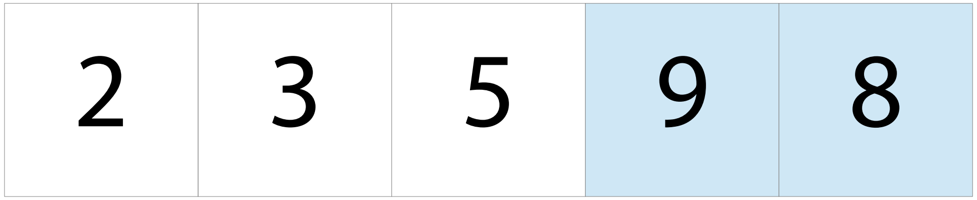
2	3	9	5	8

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



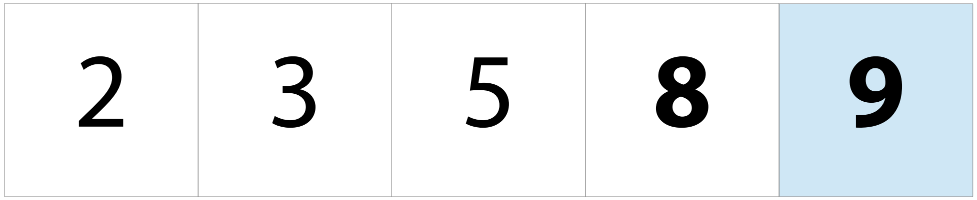
2	3	**5**	**9**	8

In-place selection sort



2	3	5	9	8

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



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

In-place selection 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?

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 the fastest sorting algorithm in general for small lists
- The other two are bad, but selection sort is the basis for a better algorithm, heapsort

A negative result

The algorithms so far as based on *swapping adjacent elements*

No sorting algorithm that works like this can be better than $O(n^2)$!

See section 8.3 for details.

(Not part of the course – an extra for those who are interested)