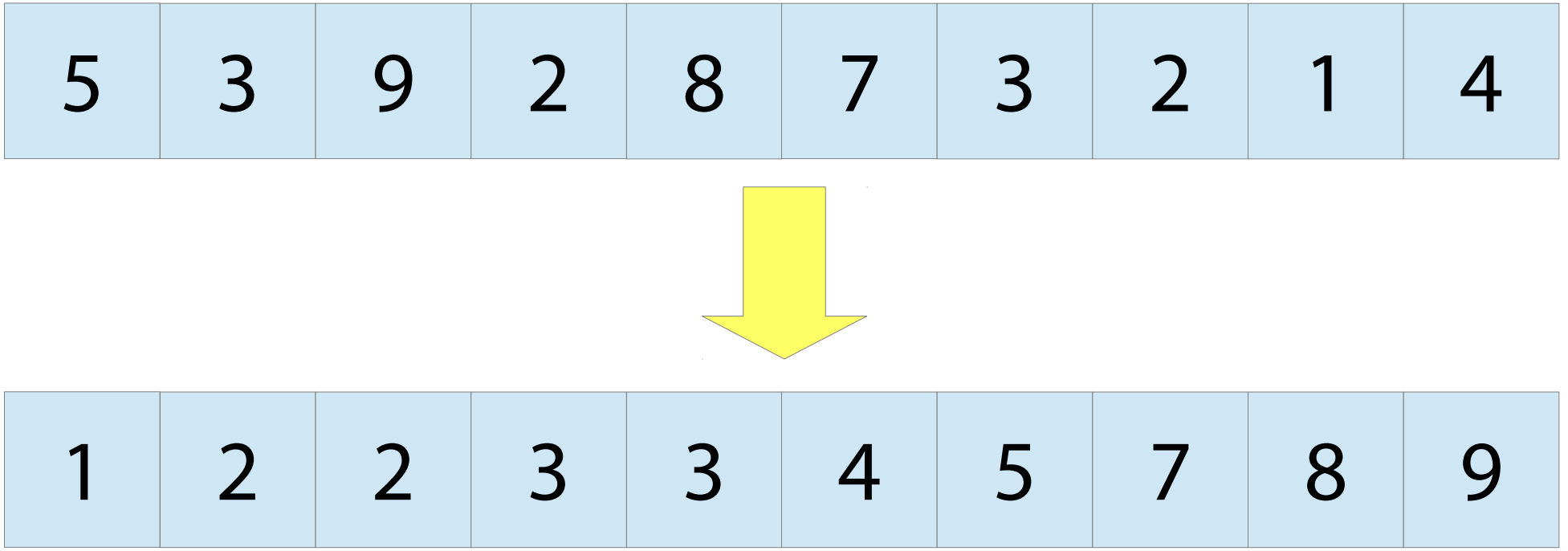
Sorting

Weiss chapter 8.1 – 8.3, 8.6

Sorting

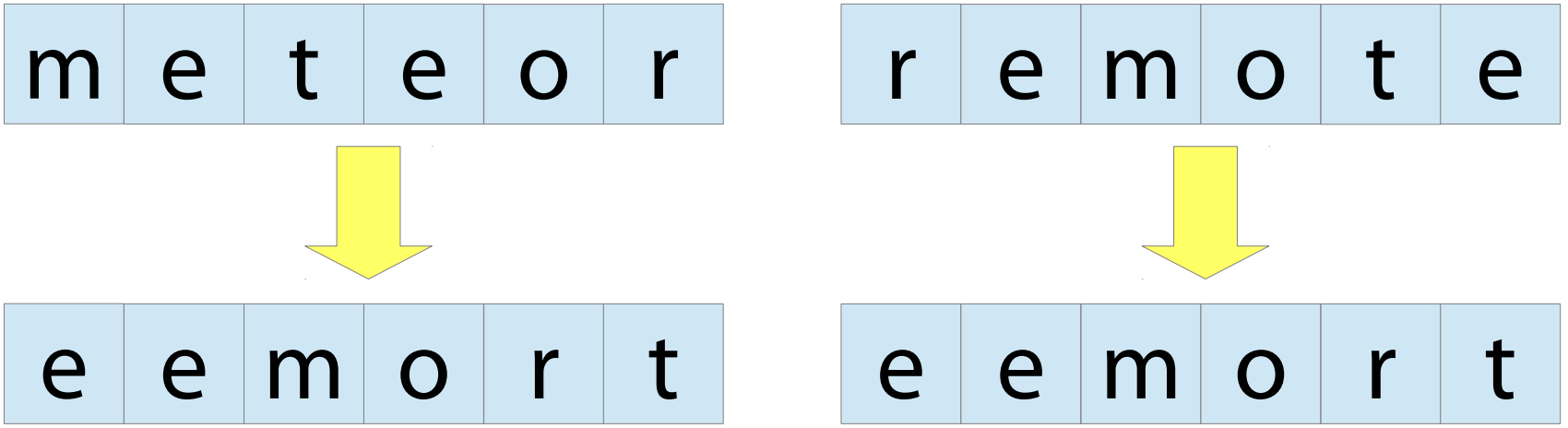


Very many different sorting algorithms (bubblesort, insertion sort, selection sort, quicksort, heapsort, mergesort, shell sort, counting sort, radix sort, ...)

Application of sorting

Are two words anagrams of each other?

Sort both words, check if they come out the same!



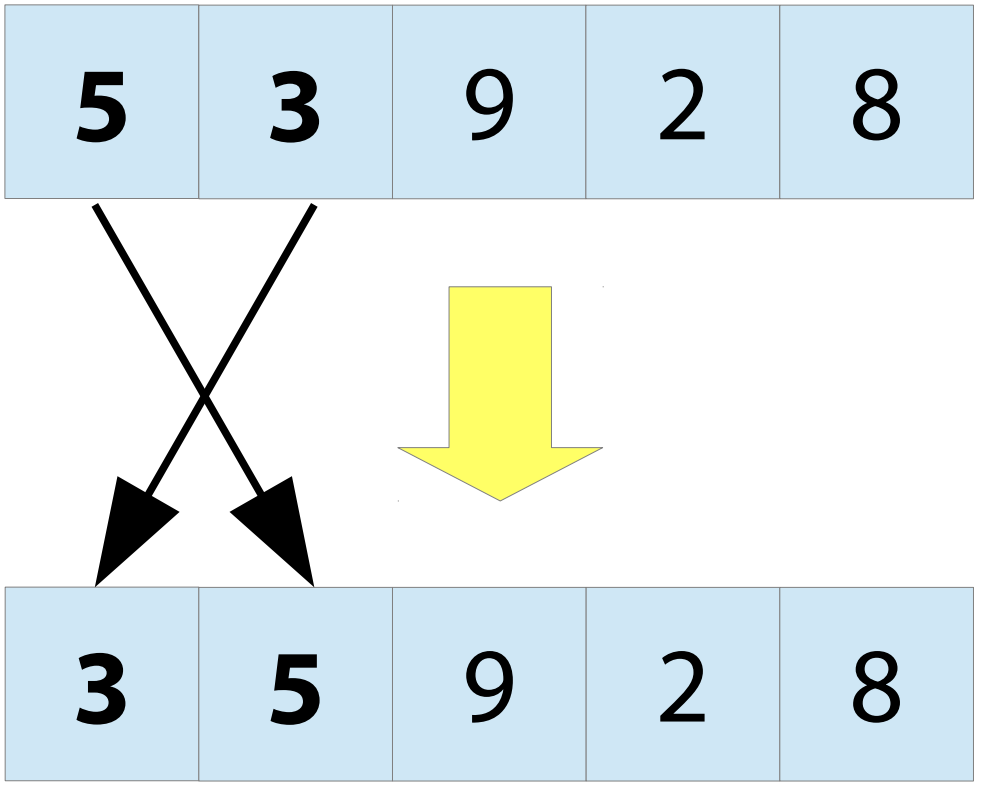
Bubblesort

Check for *adjacent elements* that are in the wrong order, and swap them

Starting from one end of the array and working towards the other

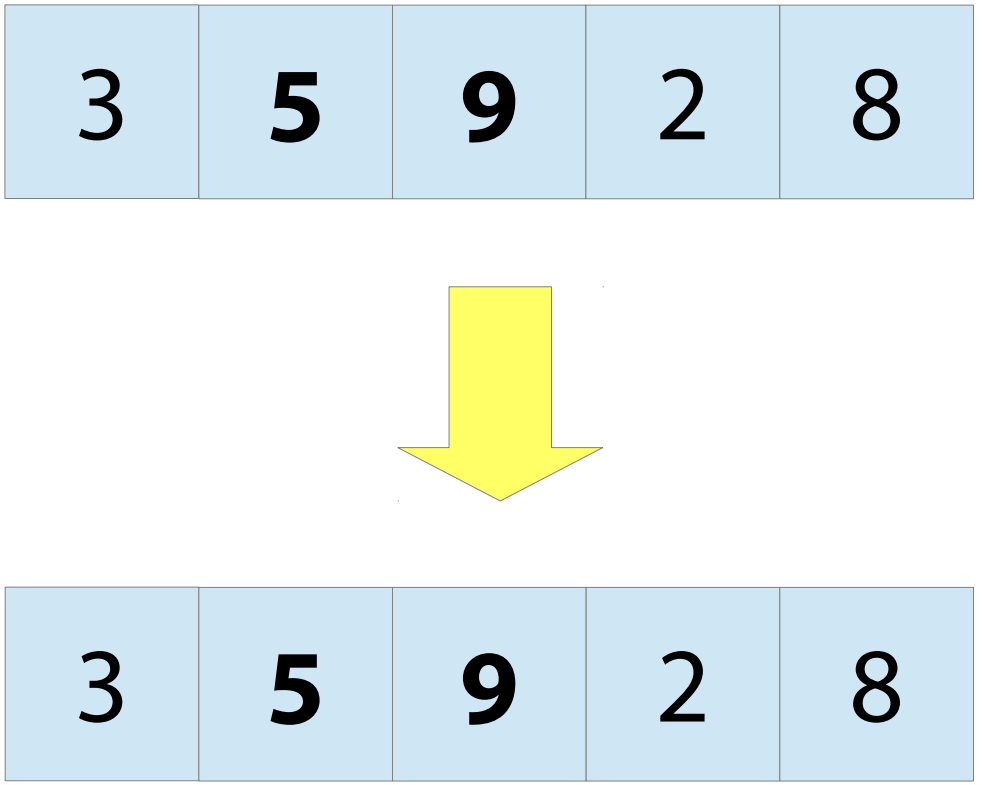
Bubblesort

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



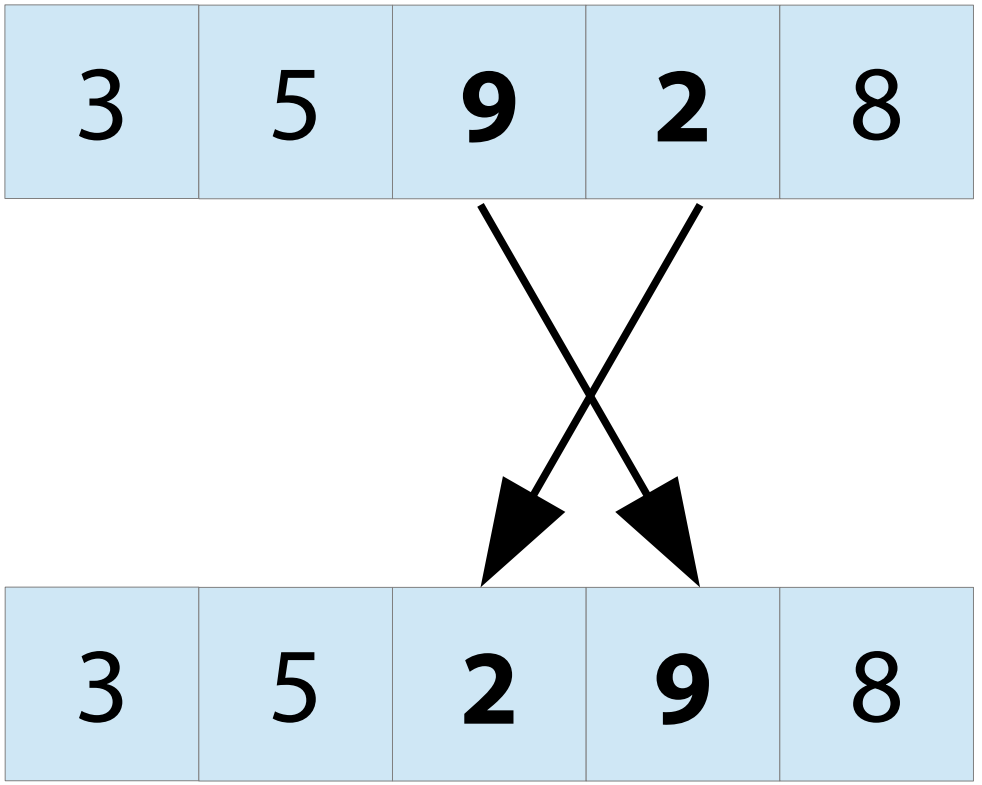
Bubblesort

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



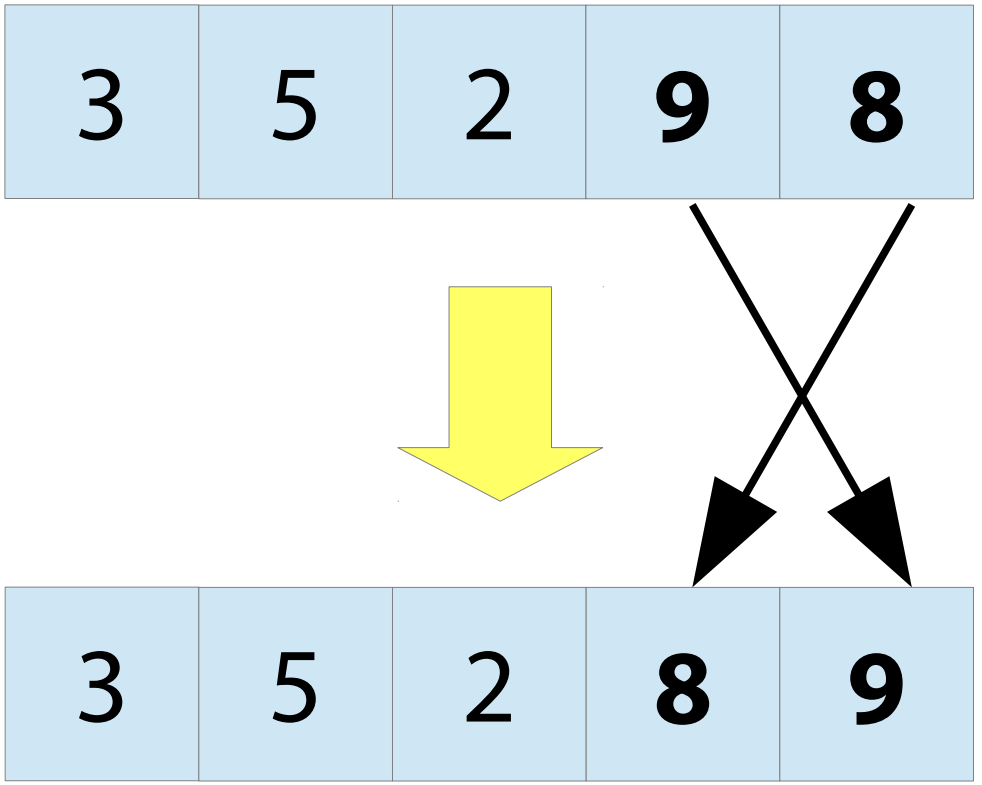
Bubblesort

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



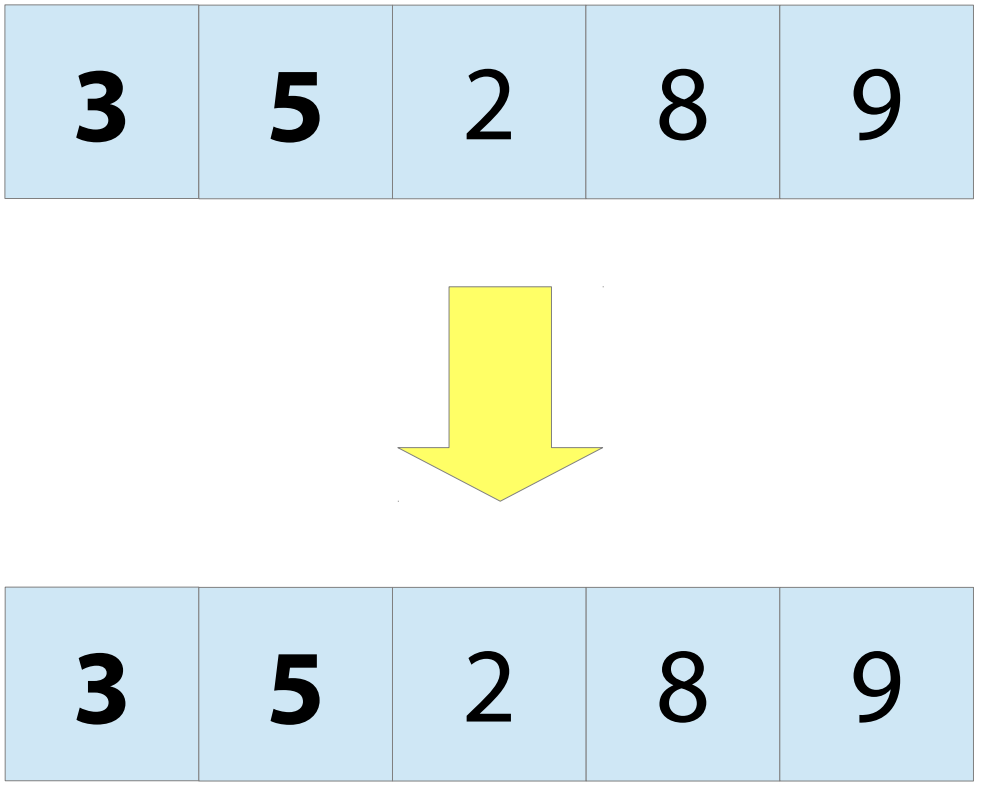
Bubblesort

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



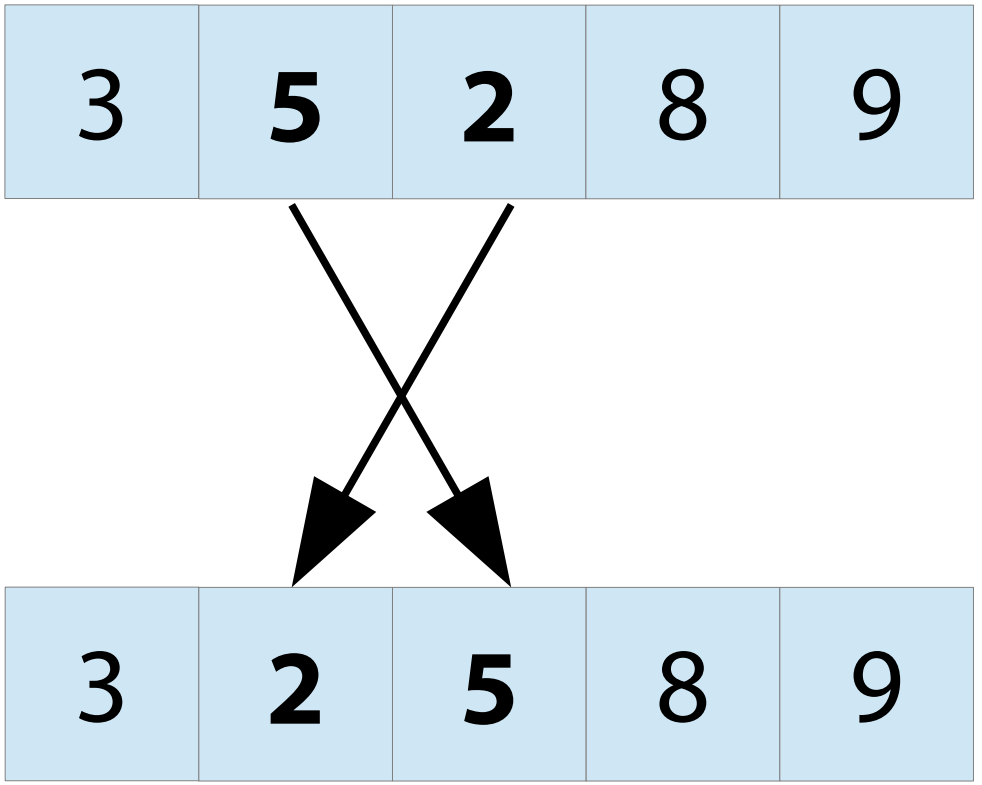
Bubblesort

Back to the beginning!



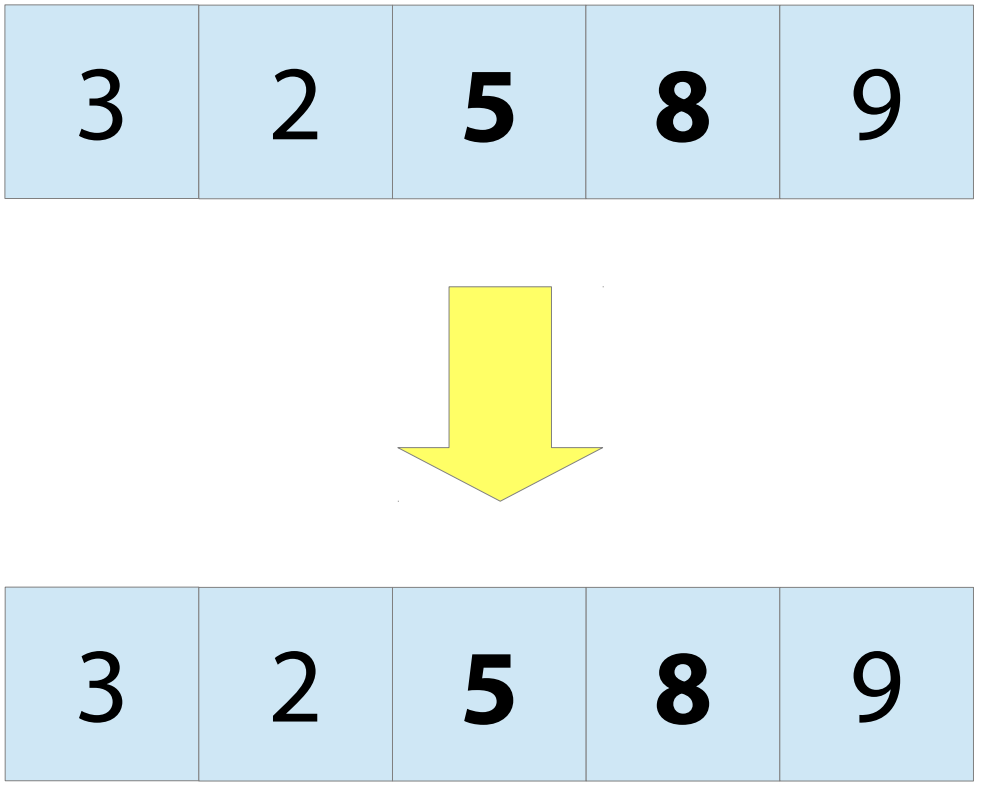
Bubblesort

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



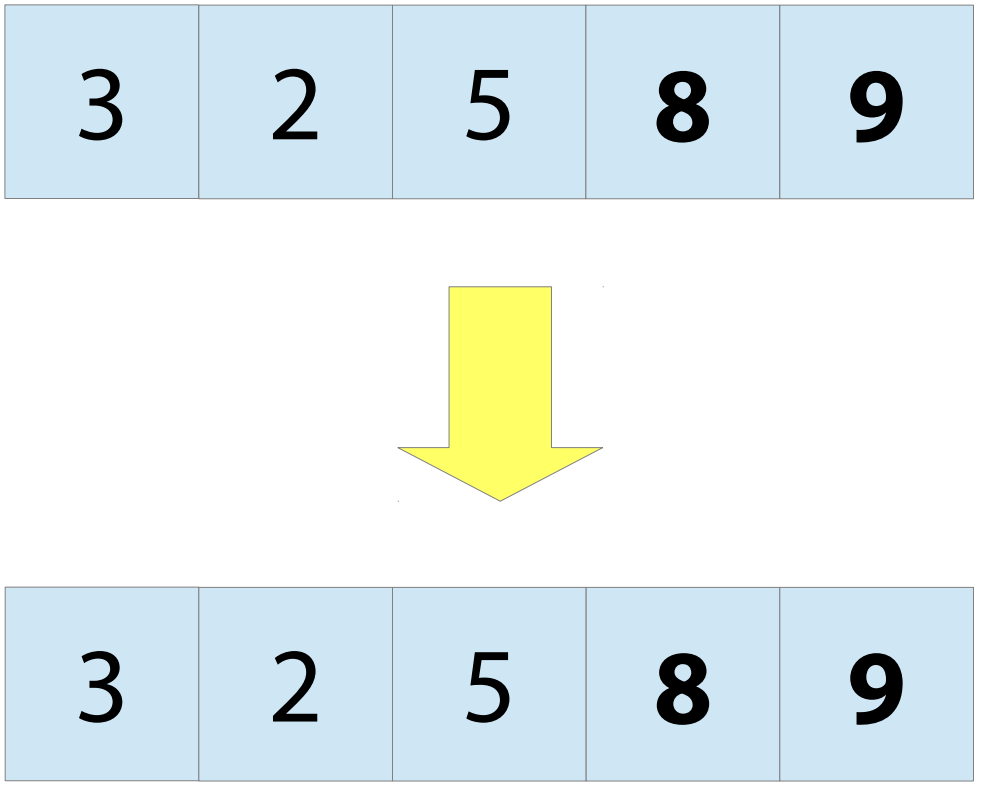
Bubblesort

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



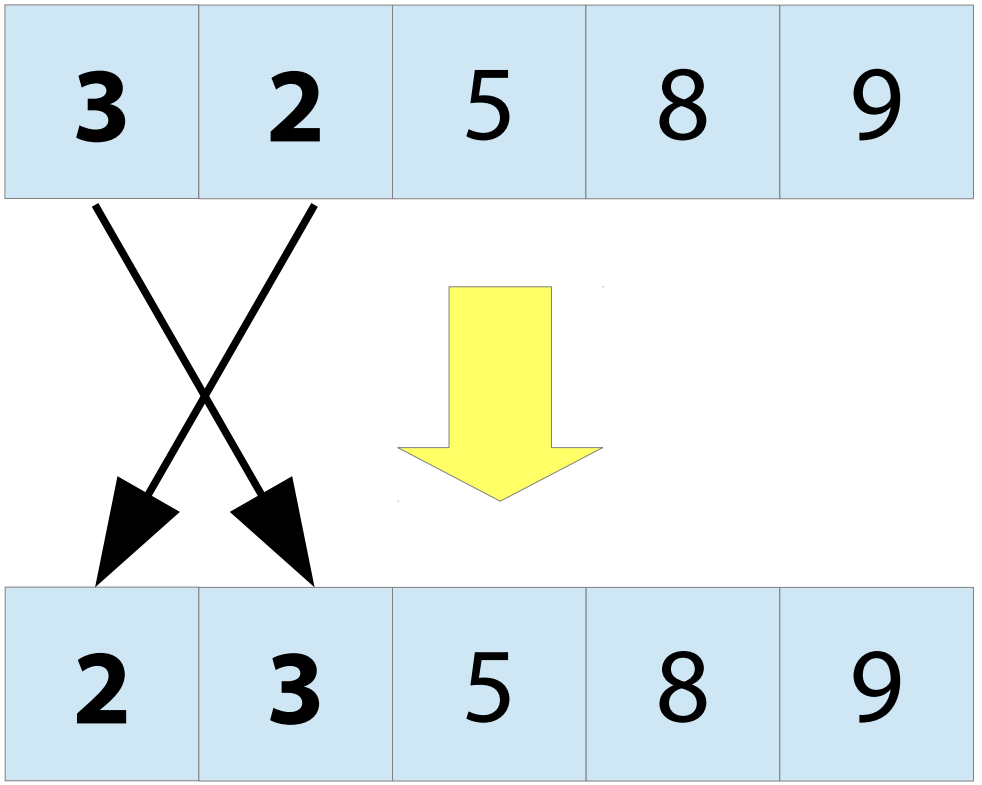
Bubblesort

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



Bubblesort

Back to the beginning!



Bubblesort

What order to do the swaps in?

- Start at the beginning of the array, loop upwards until you reach the top
- Then go round again

How do we know when to stop?

- When the array is sorted
- When the last loop didn't swap any elements

Bubblesort

```
void sort(int[] array) {
  boolean swapped;
  do {
    swapped = false;
    for (int i = 0; i < array.length-1; i++)
      if (a[i] > a[i+1]) {
        // Swap a[i] and a[i+1]
        int x = a[i];
        a[i] = a[i+1];
        a[i] = x;
        swapped = true;
      }
  } while(swapped);
}
```

This for-loop is O(n)

But how many times does the do-loop execute?

Performance of bubblesort

After one loop, the biggest element in the array has “bubbled up” to the top (hence the name bubblesort)

- Look at what happens to 9 in our example

So the do-loop executes n times

Total complexity $O(n^2)$

(we assume that comparisons and swaps take O(1) time)

Bubble sort is *bad*!

What if the array is in reverse order?

9	8	5	3	2

After one loop, only the 9 is in the right place:

8	5	3	2	9

It is very inefficient.

Insertion sort

Imagine someone deals you cards. You pick up each one in turn and 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

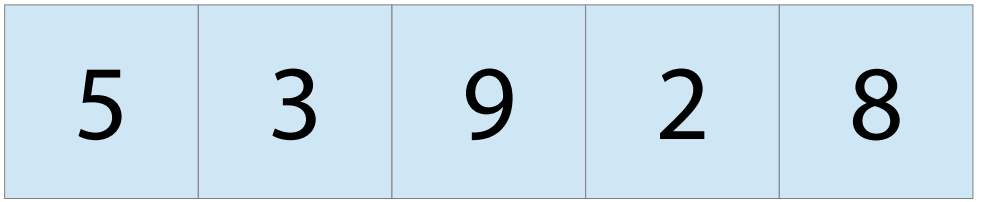
5	3	9	2	8

:

Then insert the 3 into the right place:

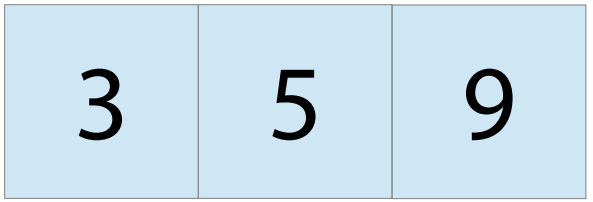
3	5

Insertion sort

Sorting 

:

Then the 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 $O(n^2)$.

Insertion sort in Haskell

```
insert x [] = [x]
insert x (y:ys)
  | x < y = x:y:ys
  | otherwise = y:insert x ys

sort [] = []
sort (x:xs) = insert x (sort xs)
```

In-place insertion sort

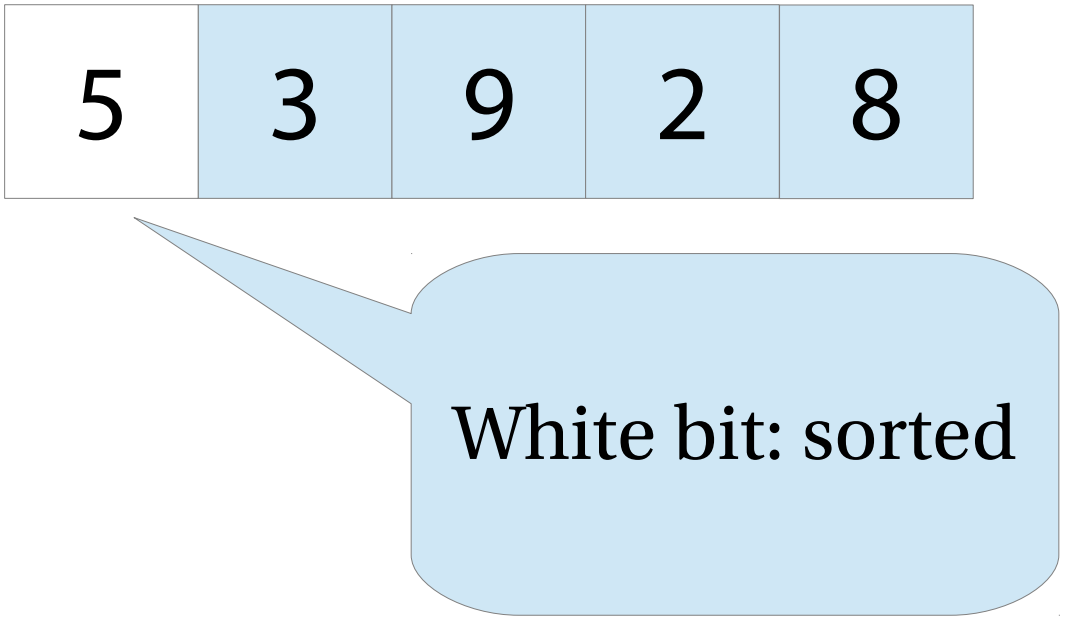
A sorting algorithm is *in-place* if it does not need to create any temporary arrays

Let's make an in-place insertion sort!

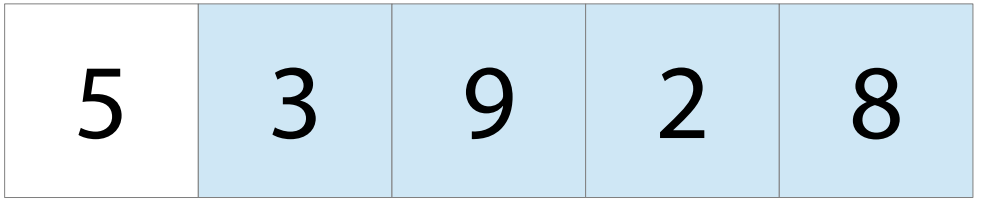
Basic idea: loop through the array, and insert each element into the part which is already sorted

In-place insertion sort

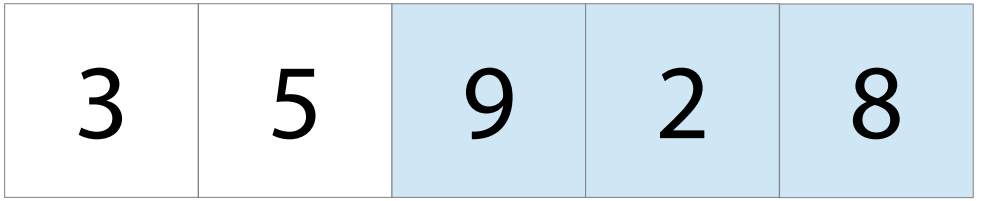
The first element of the array is sorted:



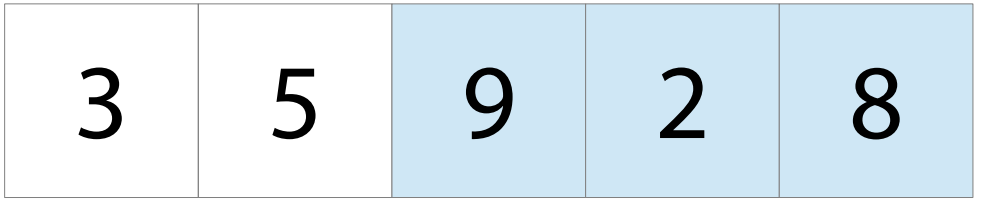
In-place insertion sort



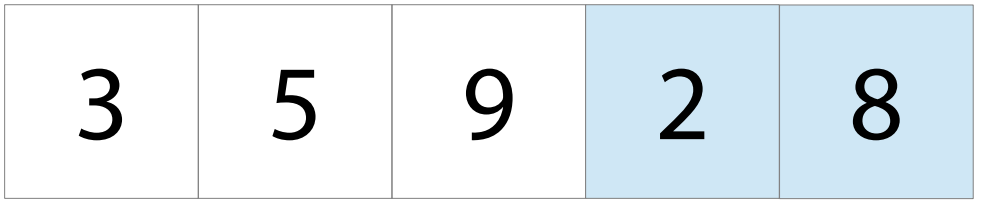
Insert the 3 into the correct place:



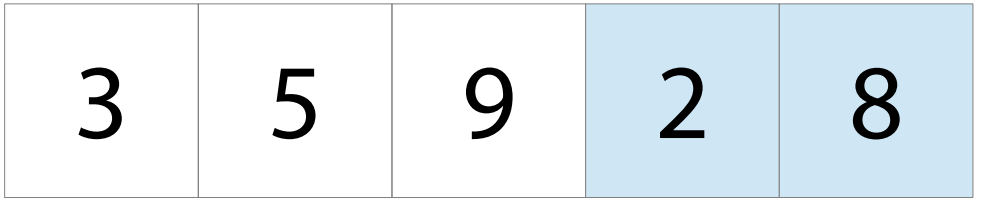
In-place insertion sort



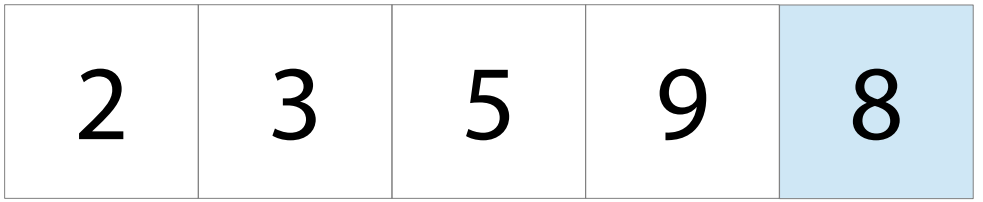
Insert the 9 into the correct place:



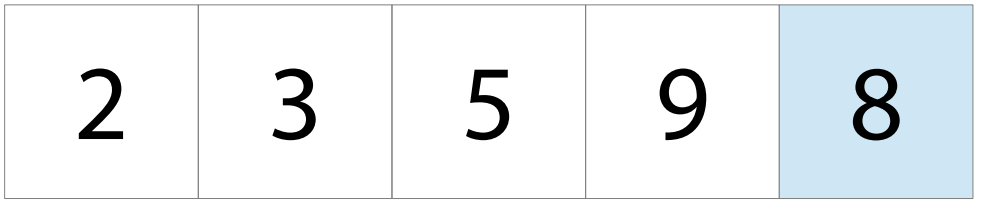
In-place insertion sort



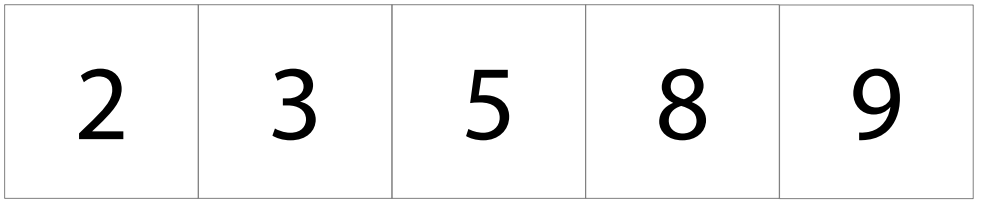
Insert the 2 into the correct place:



In-place insertion sort

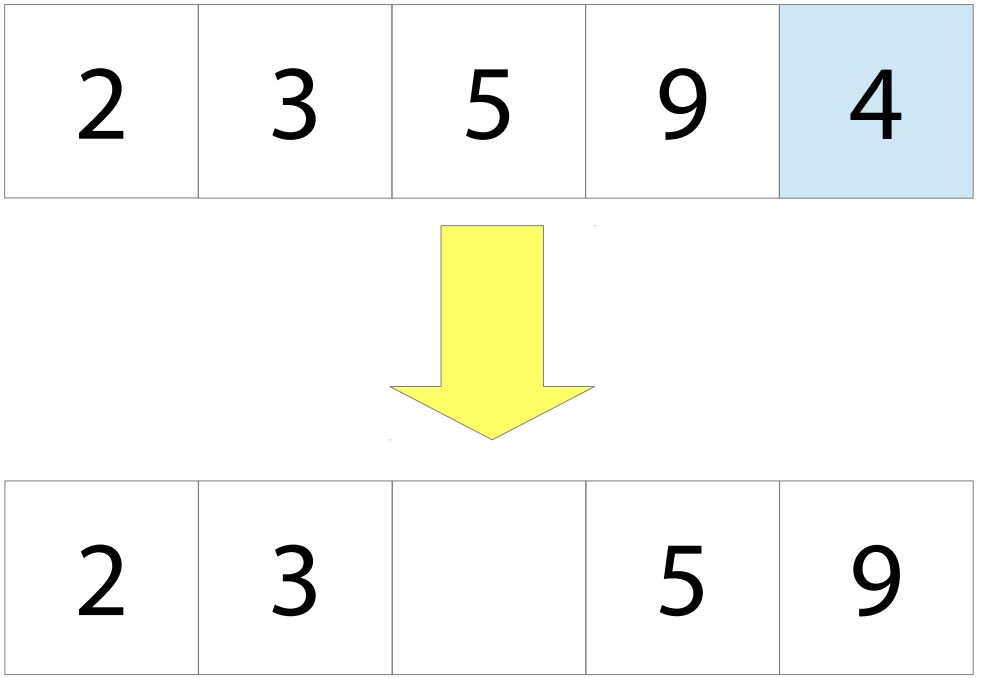


Insert the 8 into the correct place:

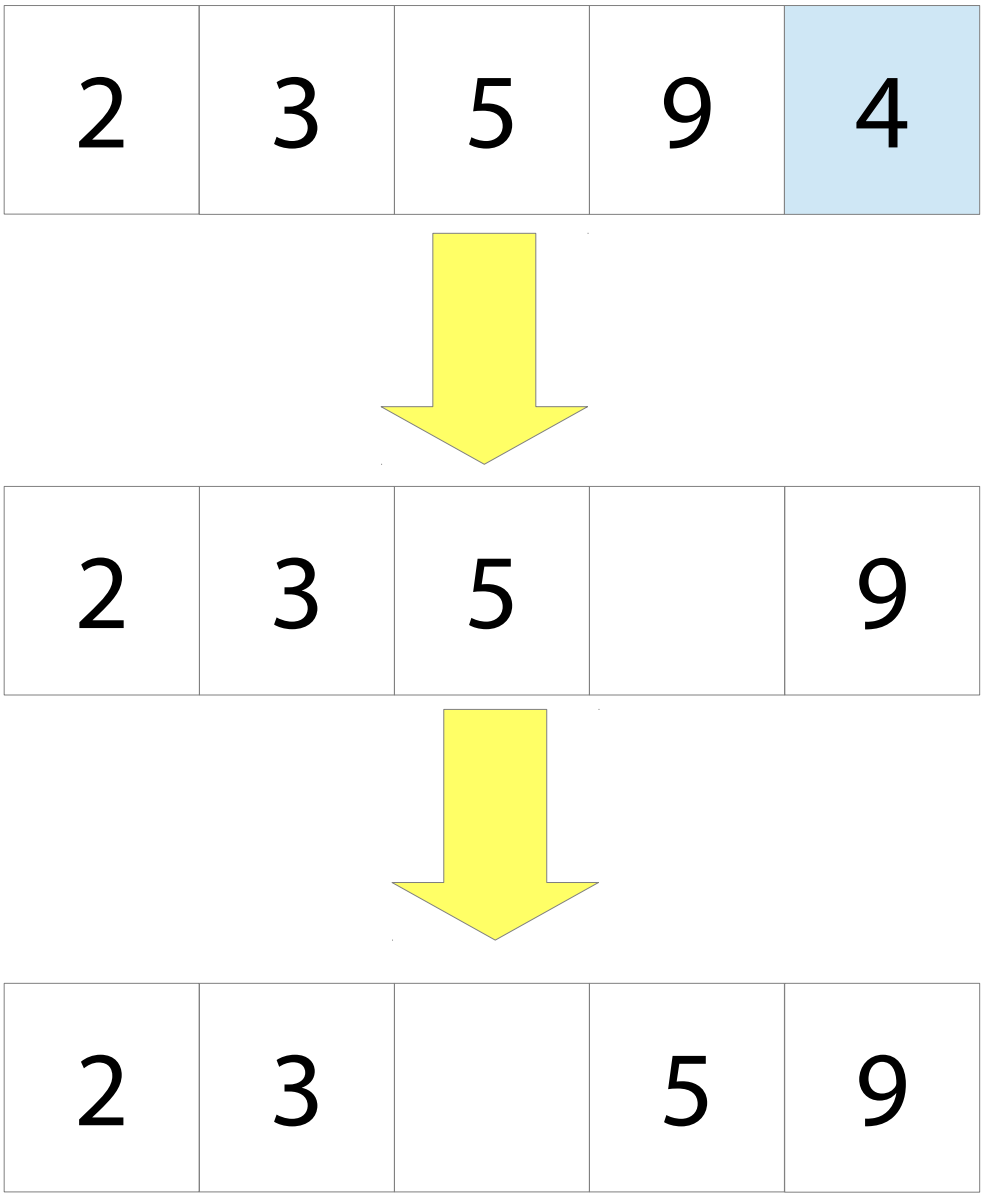


In-place insertion

To insert an item, make space by moving everything greater than it upwards



In-place insertion



In-place insertion

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

```
// Assuming that a[0..n) is sorted,
// inserting a[n] into the right place
// so that a[0..n] is sorted
void insert(int[] a, int n) {
  int x = a[n];
  int i = n;
  while(i > 0 && a[i-1] > x) {
    a[i] = a[i-1];
    i--;
  }
  a[i] = x;
}
```

In-place insertion sort

```
void sort(int[] array) {
   for (int i = 1; i < n; i++)
      insert(array, i);
}
```

An aside: we have the *invariant* that `array[0..i)` is sorted

- An invariant is something that holds whenever the loop starts
- Initially, i = 1 and `array[0..1)` is sorted
- When `array[0..i)` is sorted, the loop body makes `array[0..i+1)` sorted, establishing the invariant for the next iteration
- When the loop finishes, `i = n`, so `array[0..n)` is sorted – the whole array!

A negative result

Bubblesort and insertion sort are both 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.

Selection sort

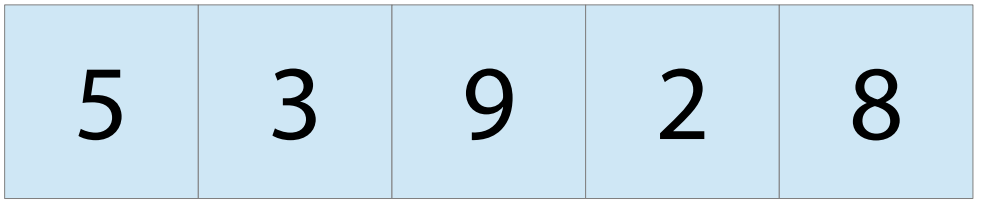
Find the smallest element of the array, and delete it

Find the smallest remaining element, and delete it

And so on

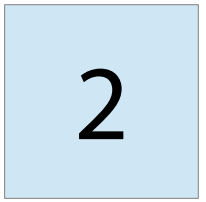
Finding the smallest element is O(n), so total complexity is $O(n^2)$

Selection sort

Sorting 

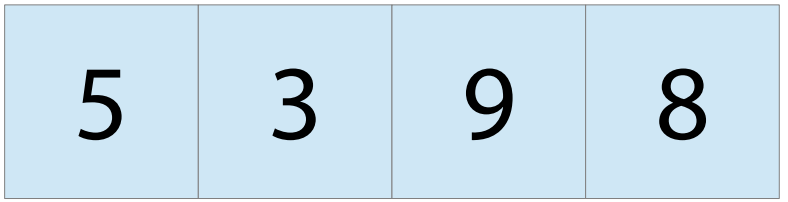
:

The smallest element is 2:



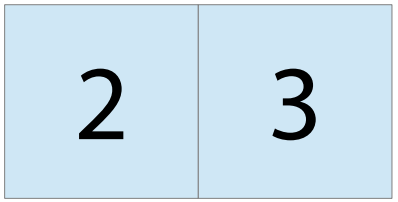
We also delete 2 from the input array.

Selection sort

Sorting 

:

Now the smallest element is 3:



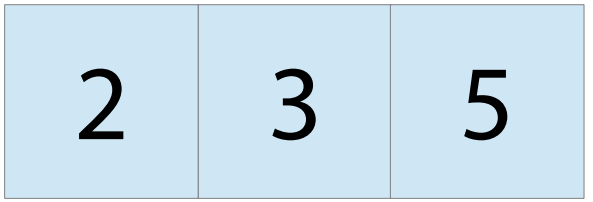
We delete 3 from the input array.

Selection sort

Sorting 

:

Now the smallest element is 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

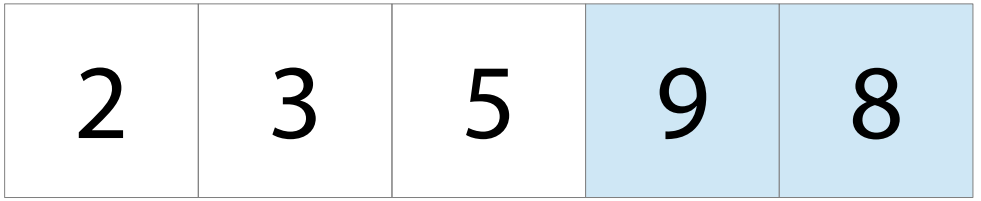
In-place selection sort

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

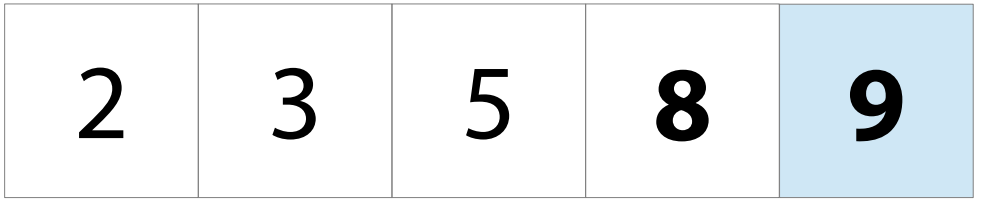
In-place selection sort

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

In-place selection sort



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



In-place selection sort

```
int sort(int[] a) {
  for (int i = 0; i < a.length - 1; i++) {
    int min =
      /* the smallest element in a[i..n) */;
    int x = a[min];
    a[min] = a[i];
    a[i] = x;
  }
}
```

Divide and conquer algorithms and quicksort

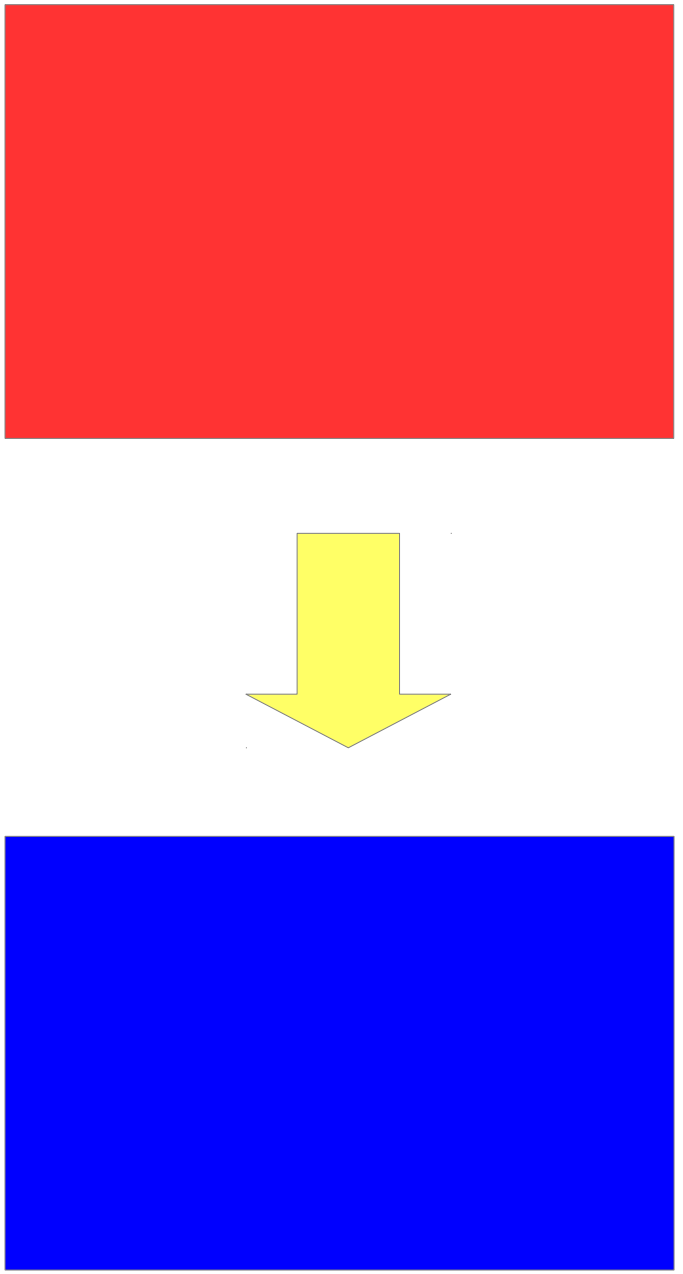
Divide and conquer

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

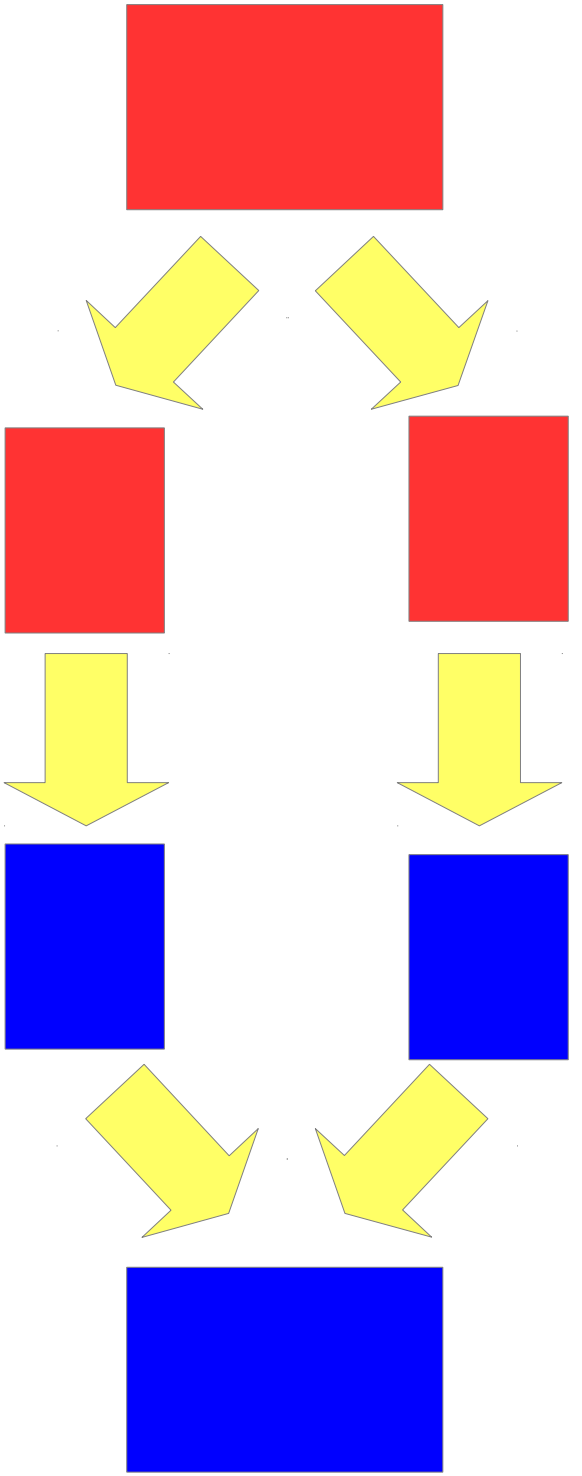
To solve this...



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

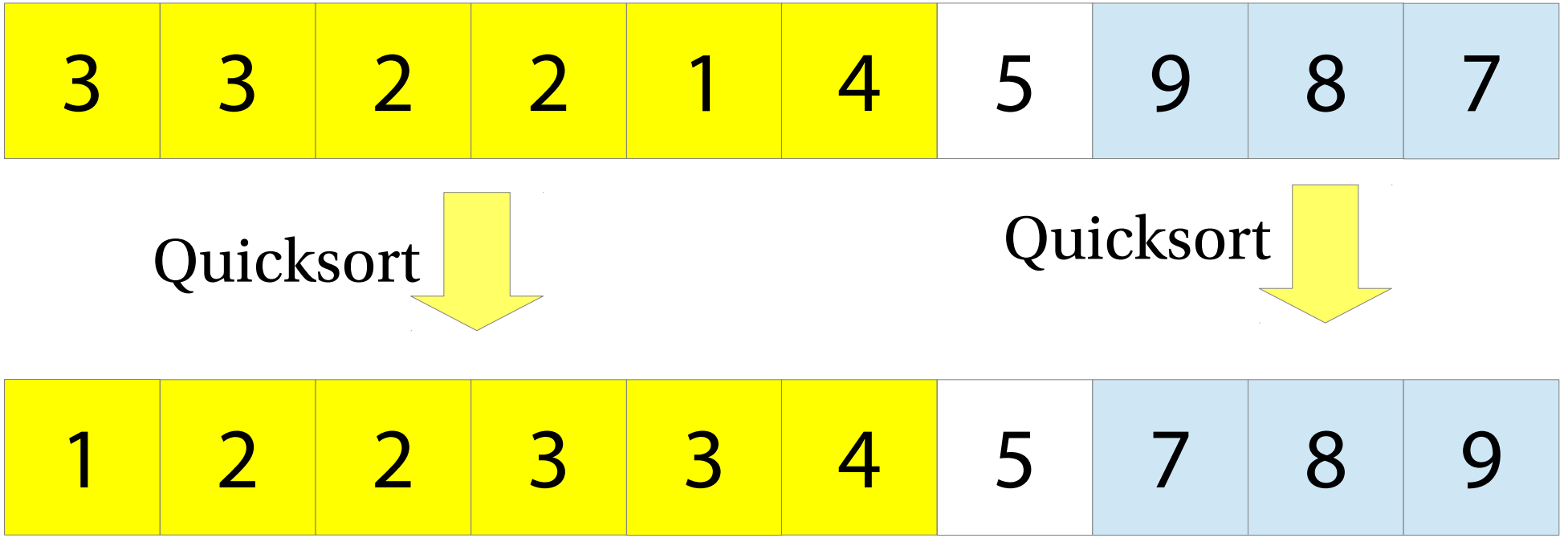
3	3	2	2	1	4	5	9	8	7

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);
}
```

Usual optimisation: switch to insertion sort when the input array is small

Haskell code

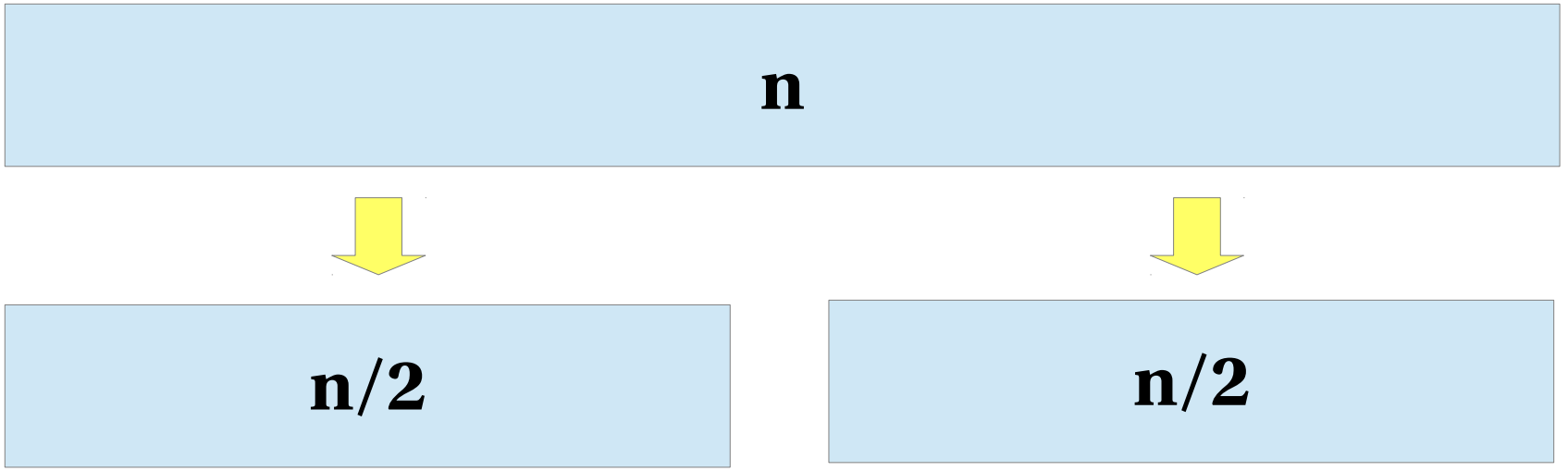
```
sort [] = []
sort (x:xs) =
   sort (filter (< x) xs) ++
   [x] ++
   sort (filter (>= x) xs)
```

Split: filter

Combine: ++

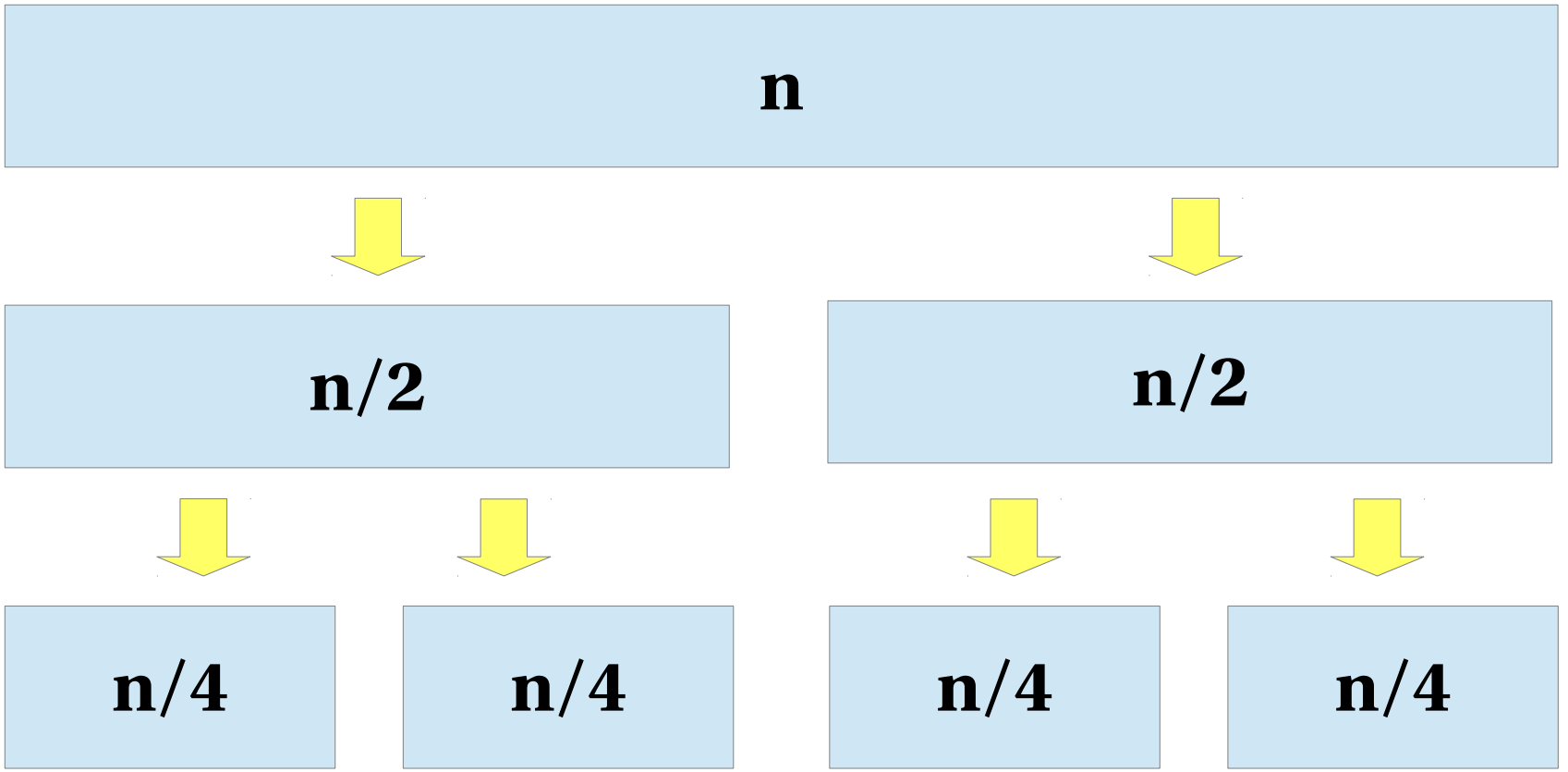
Complexity of quicksort

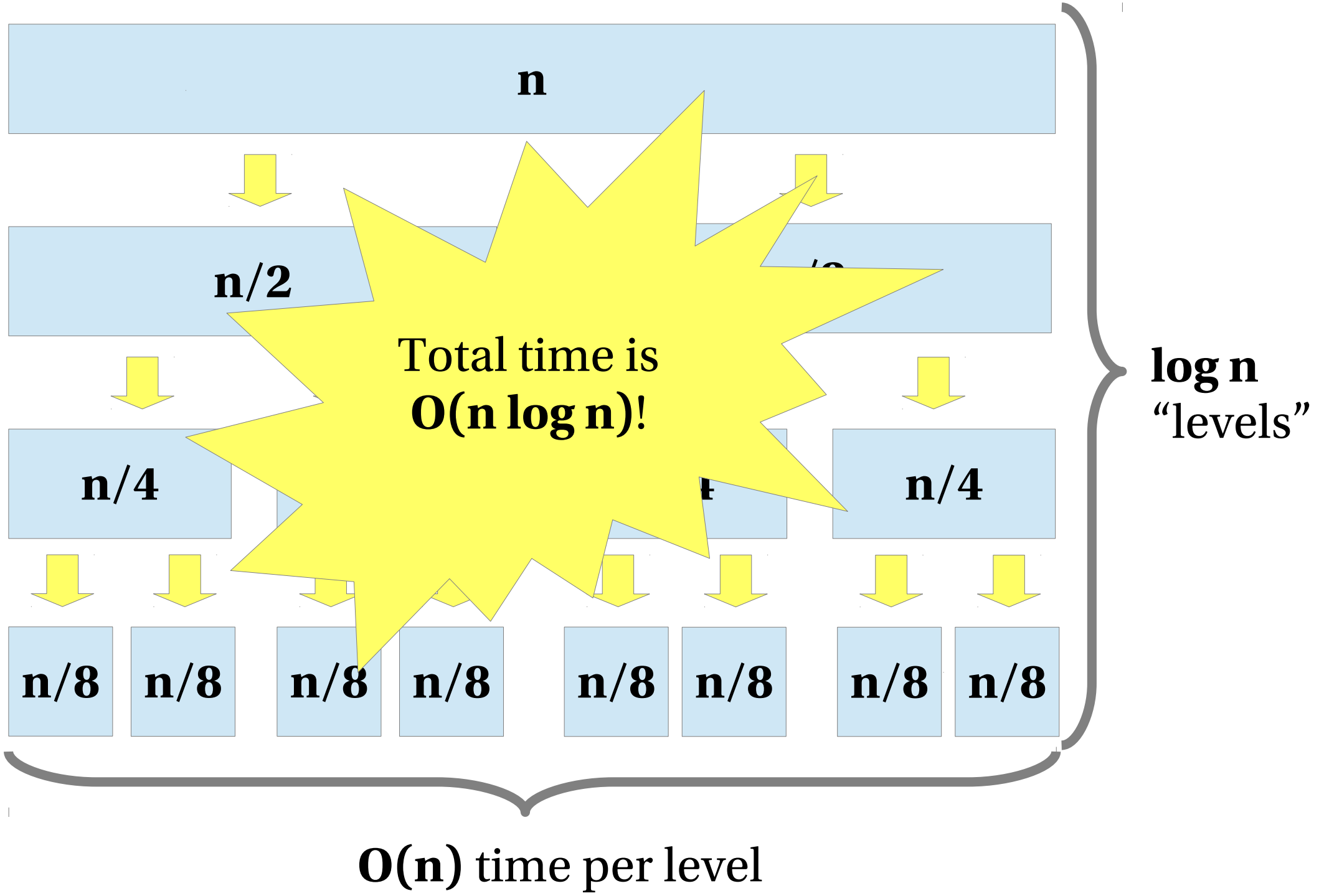
In the best case, partitioning splits an array of size n into two halves of size n/2:



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

Worst cases

Sorted array

Reverse-sorted array

Try these out!

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)

Summary of quicksort

- Divide-and-conquer algorithm: choose pivot, partition array into two, recursively sort both partitions
- O(n log n) if both partitions have about equal size, $O(n^2)$ if one is much bigger than the other
  - One solution: choose pivot at random (others in book)
- Very fast in practice

Next lecture

How to perform partitioning

More sorting algorithms

Is O(n log n) the limit of sorting?

How to find the complexity of recursive programs