Binary search

Searching

Suppose I give you an array, and ask you to find if a particular value is in it, say 4.

5	3	9	2	8	7	3	2	1	4

The only way is to look at each element in turn.

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.

1	2	2	3	3	4	5	7	8	9

Binary search

3 is less than 4.

Since the array is sorted, we know that 4 must come after 3.

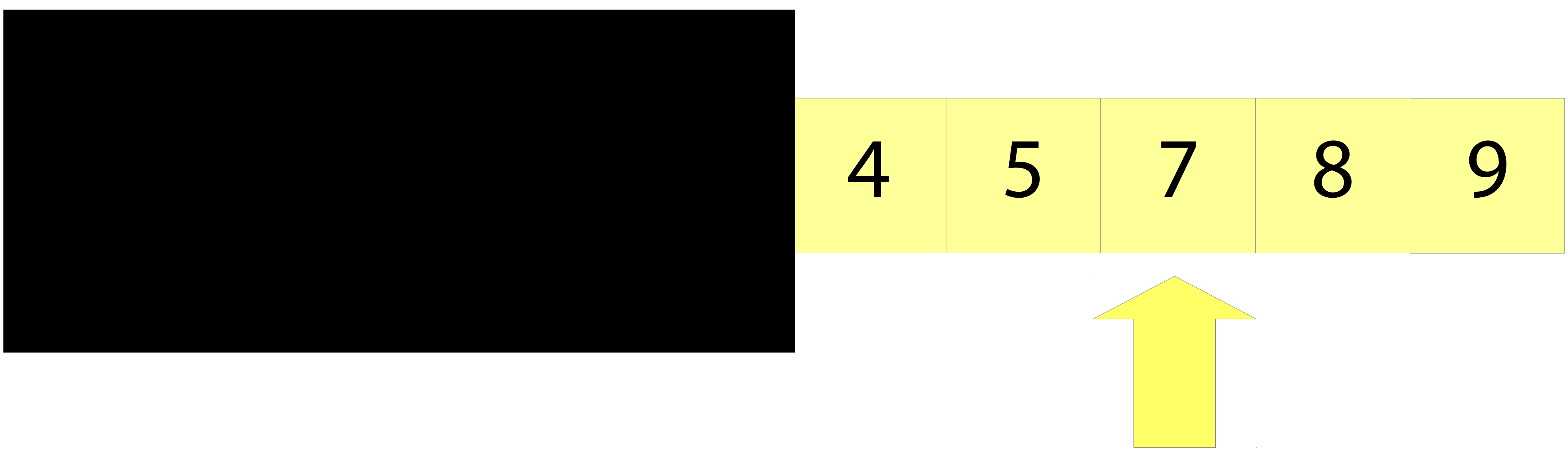
We can ignore everything before 3.

[redacted]	4	5	7	8	9

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 ignore everything after 7.

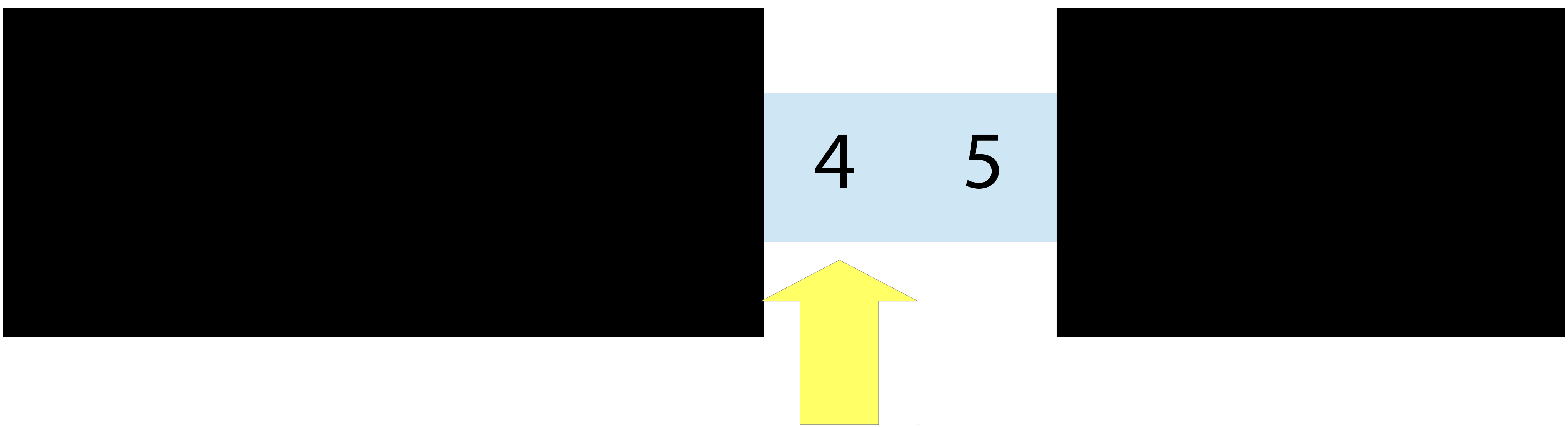
[redacted]	4	5	[redacted]

Binary search

We repeat the process.

We look half way along the array again.

We find 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!

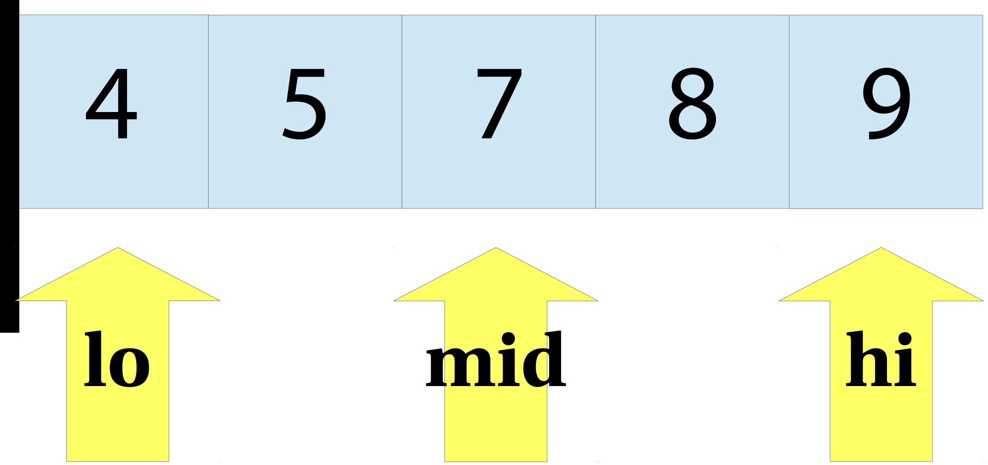
On an array of a billion elements, need to search **30** elements

(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 search.

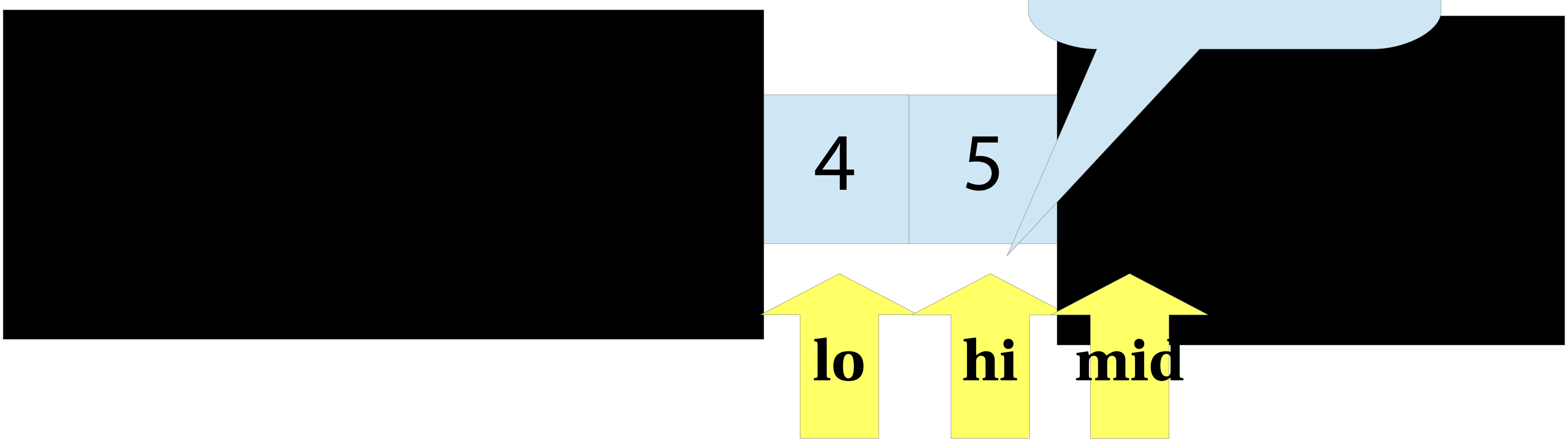
[redacted]



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`. T hi = mid - 1 t
the part of the array to search



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

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, …)

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

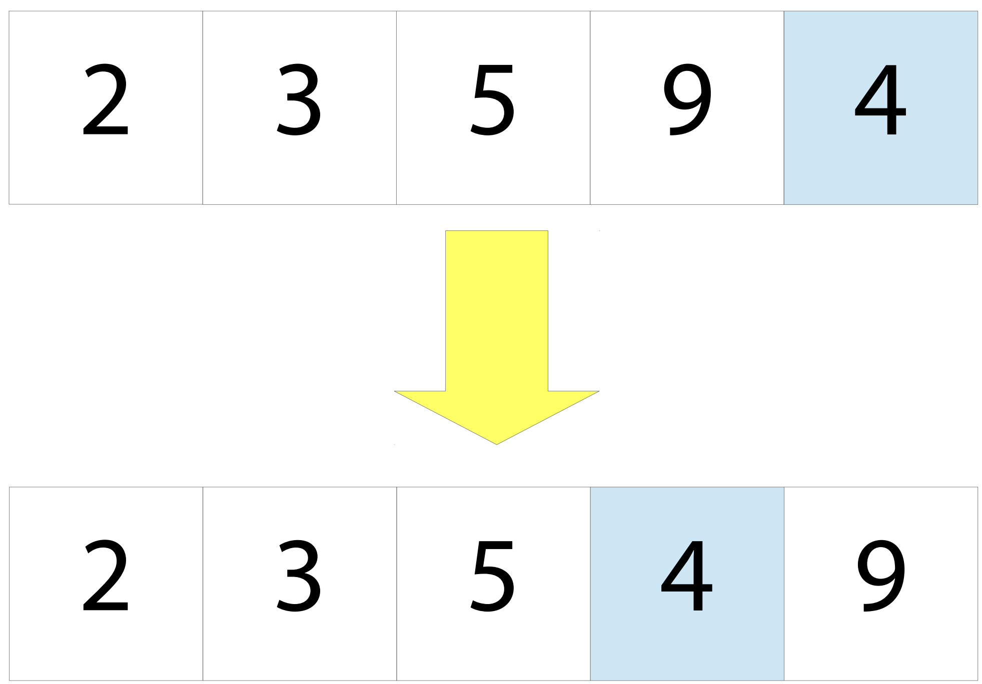
2	3	5	9	8

Insert the 8 into the correct place:

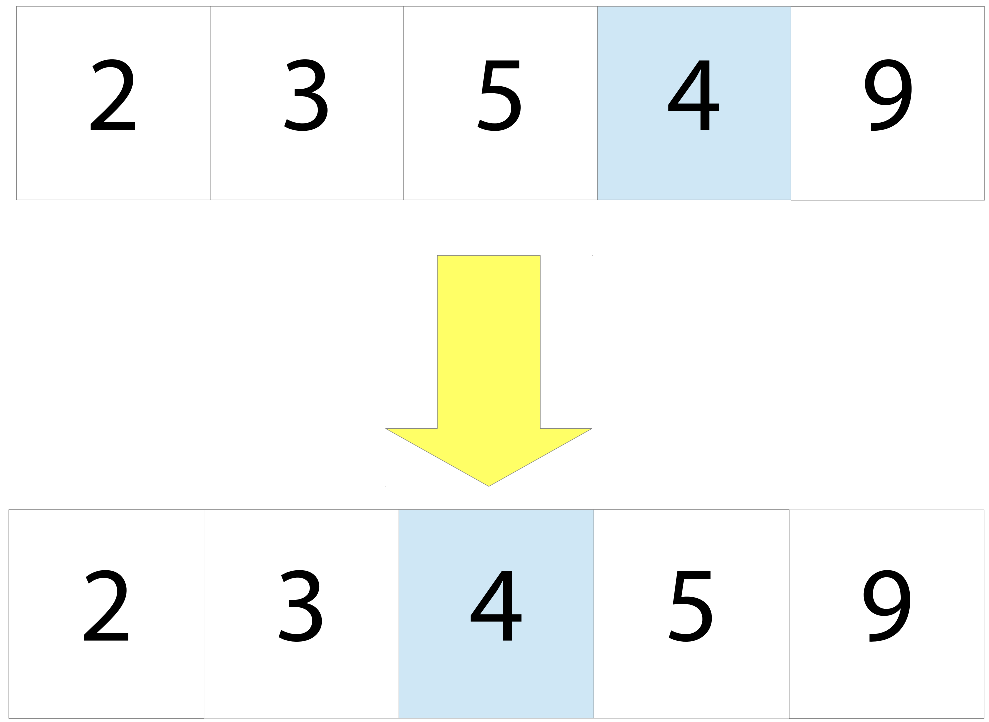
2	3	5	8	9

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



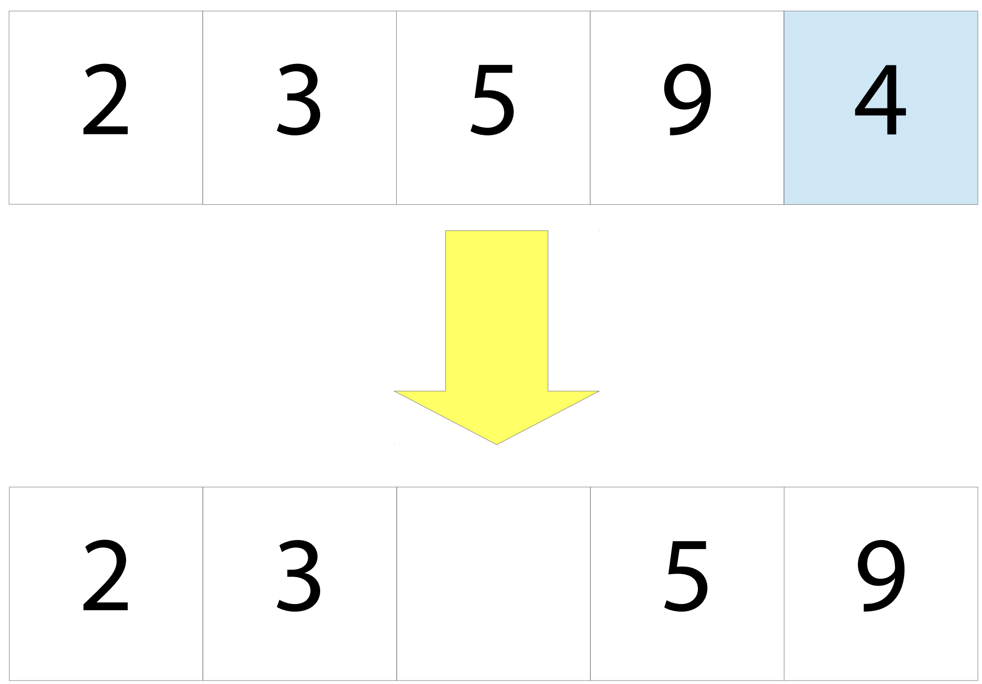
In-place insertion



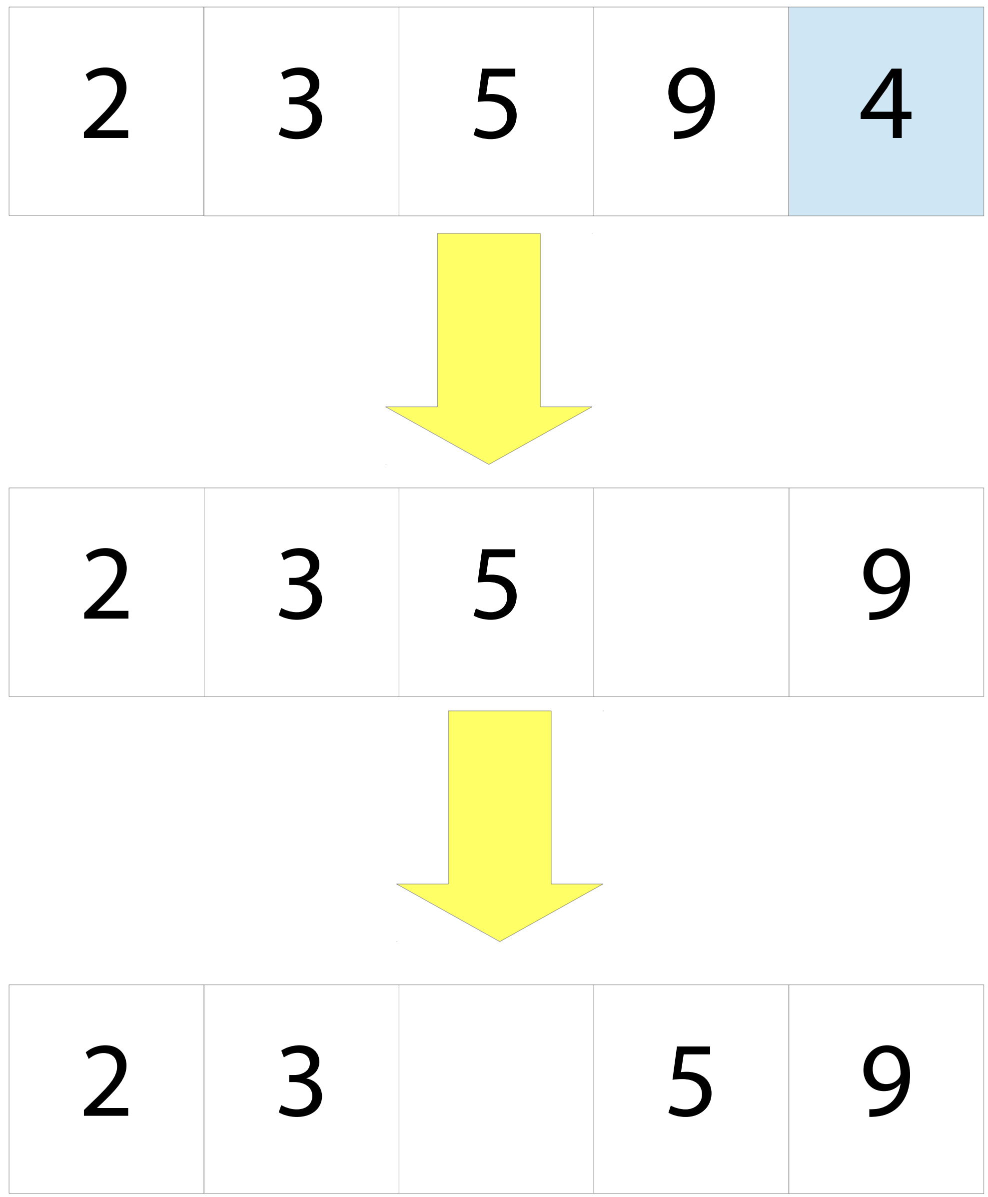

```
while n > 0 and array[n] > array[n-1]
  swap array[n] and array[n-1]
  n = n-1
```

In-place insertion

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



In-place insertion



In-place insertion so

This notation means
0, 1, …, i-1

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

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!

Insertion sort

$O(n^2)$ in the worst case

$O(n)$ in the best case (a sorted array)

Actually the fastest sorting algorithm in general for small lists – it has low constant factors

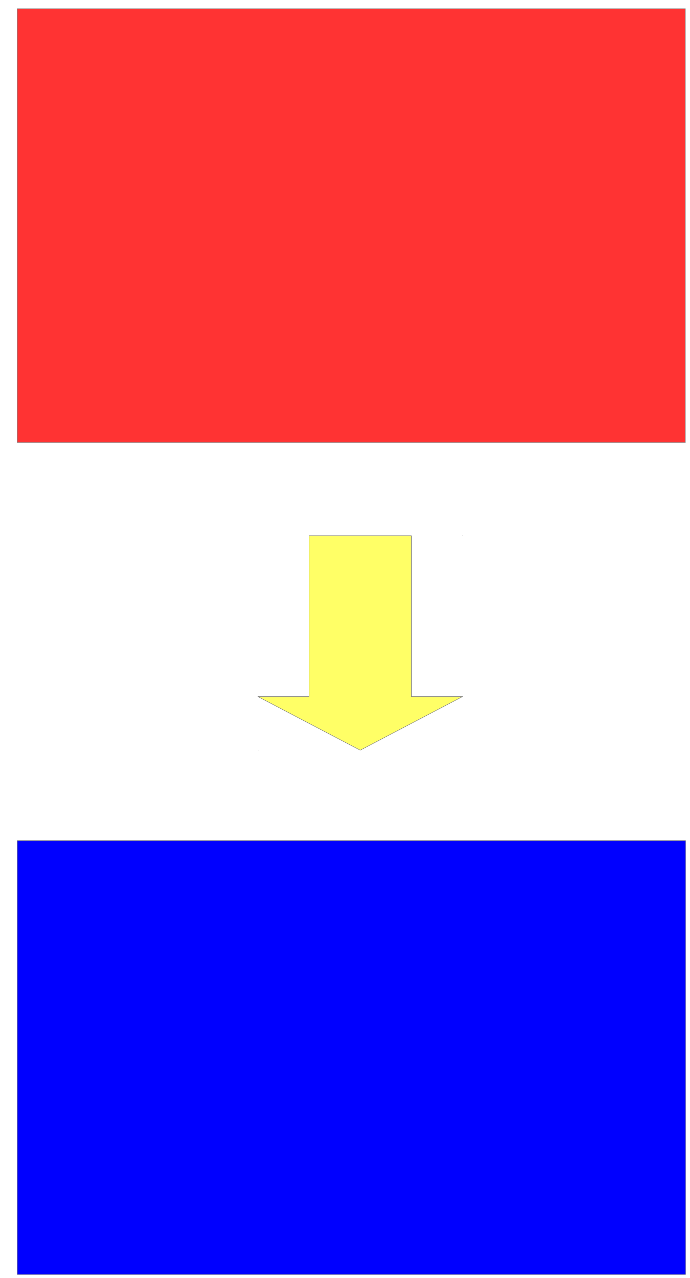
Divide and conquer

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

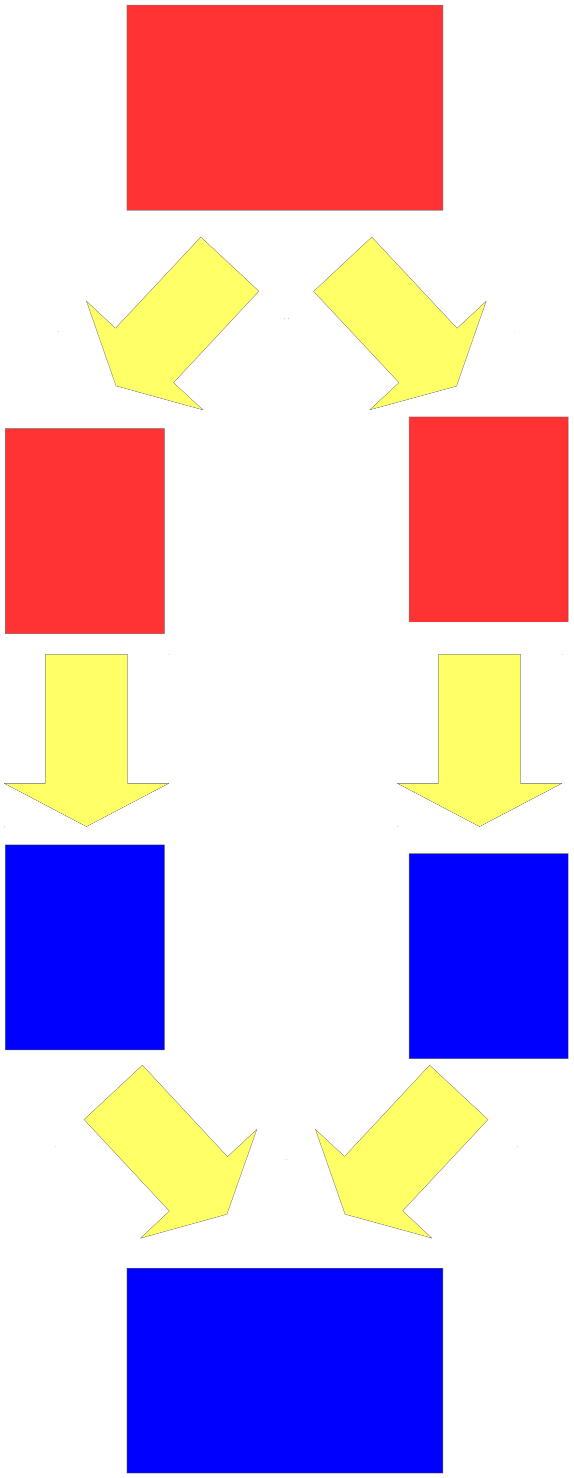
To solve this...



1. *Split* the problem
into subproblems

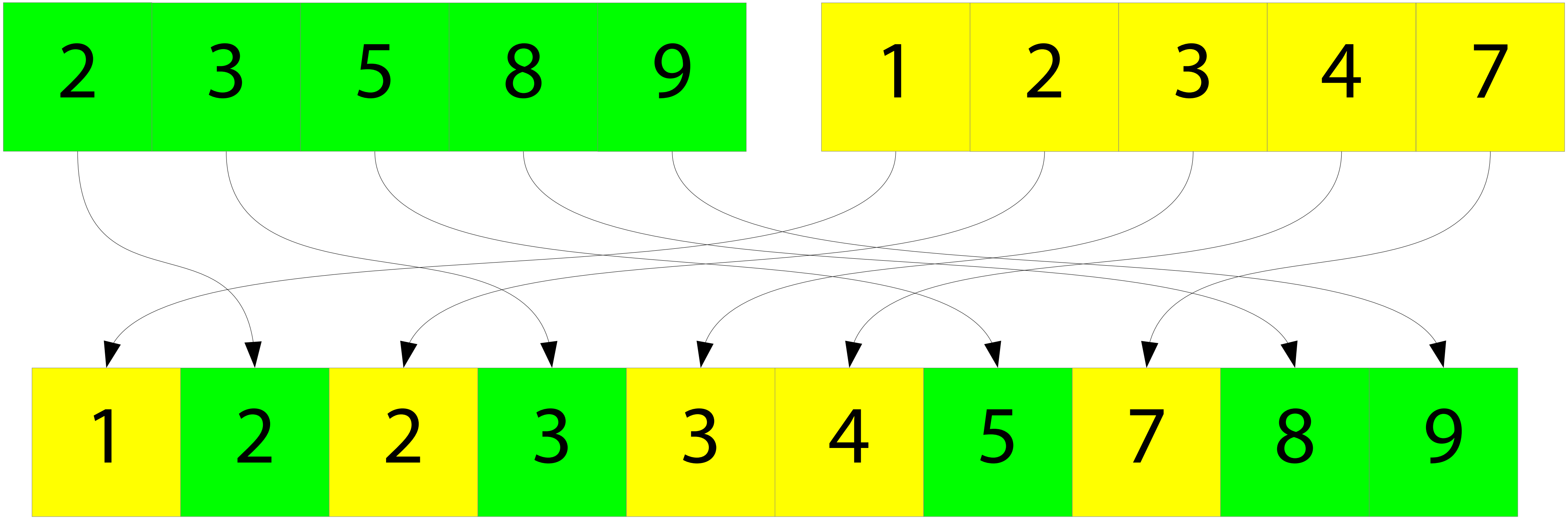
2. *Recursively* solve
the subproblems

3. *Combine*
the solutions



Mergesort

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



Mergesort

A 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

5	3	9	2	8

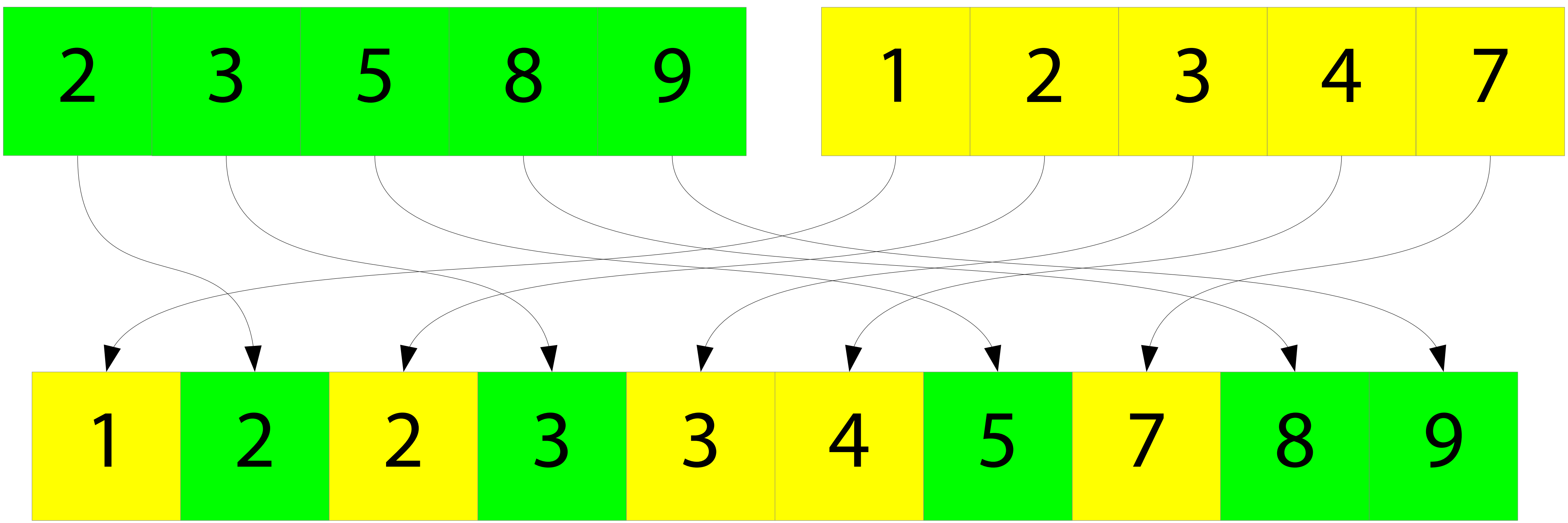
7	3	2	1	4

2	3	5	8	9

1	2	3	4	7

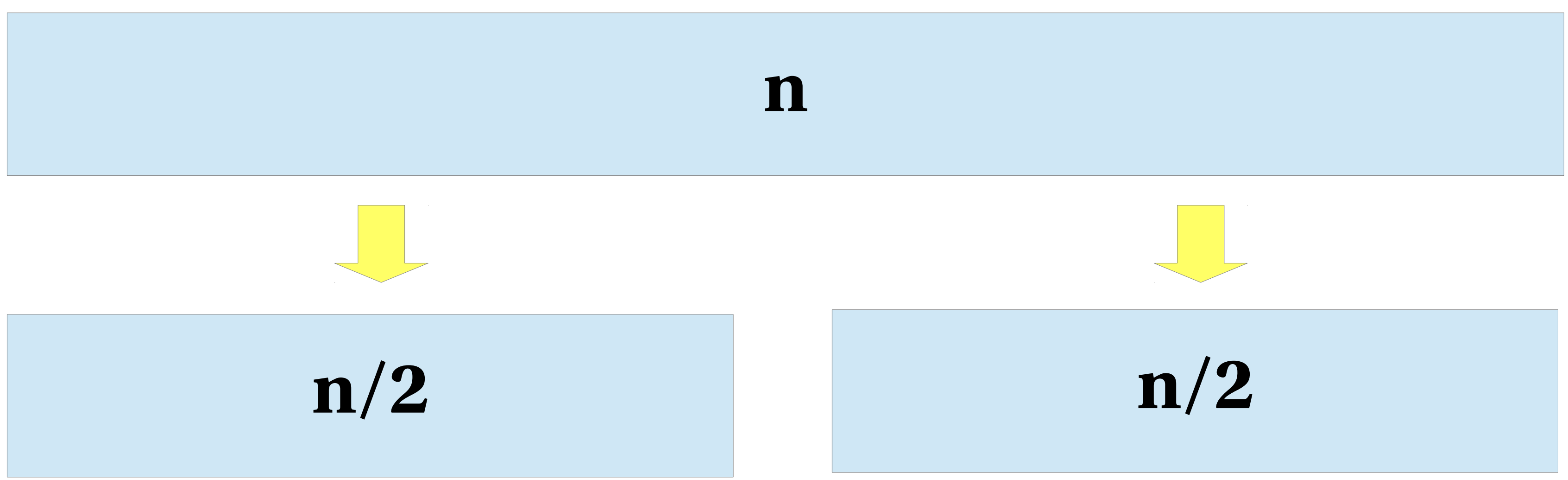
Mergesort

3. *Merge* the two sorted lists together



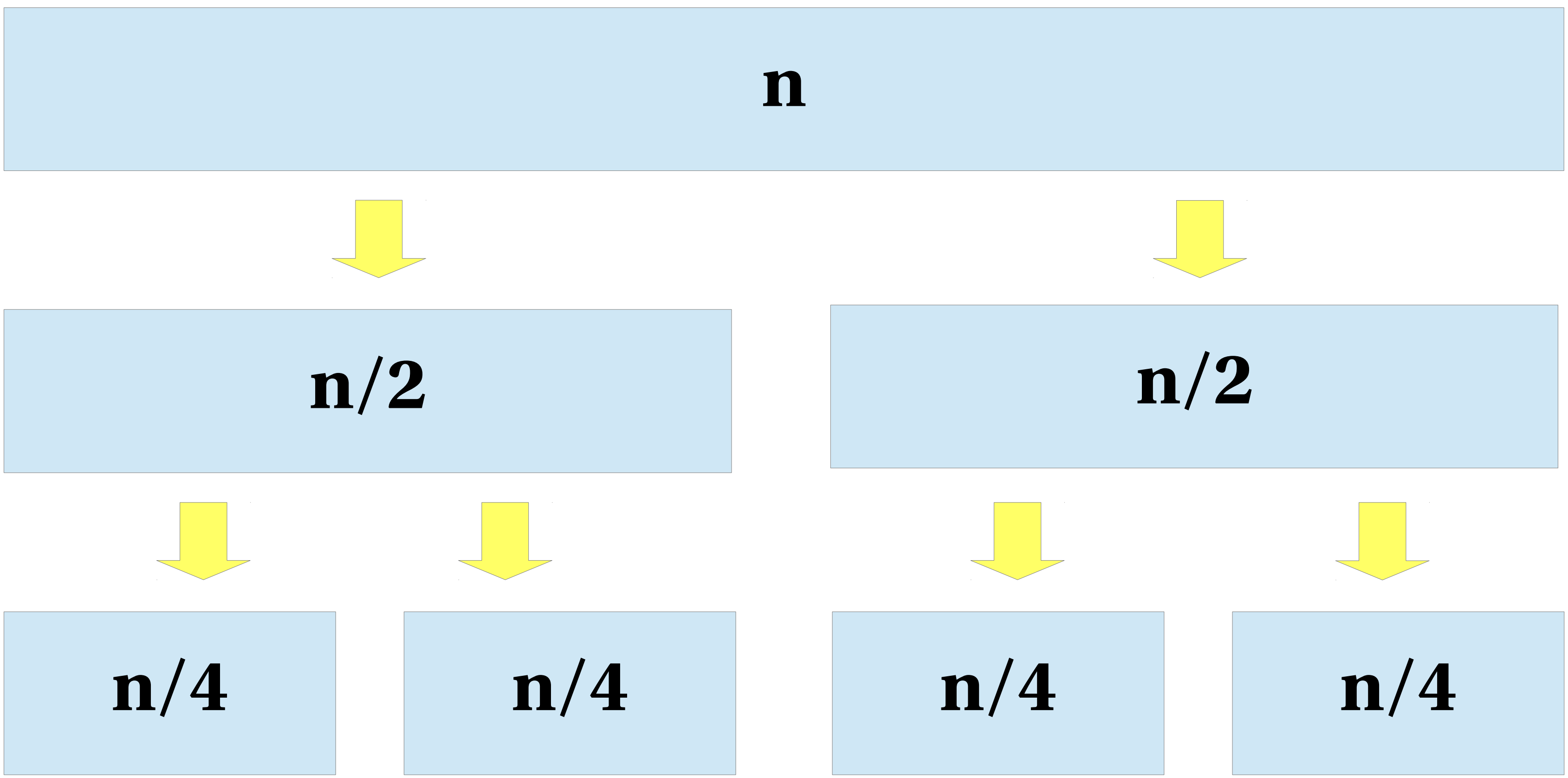
Complexity of mergesort

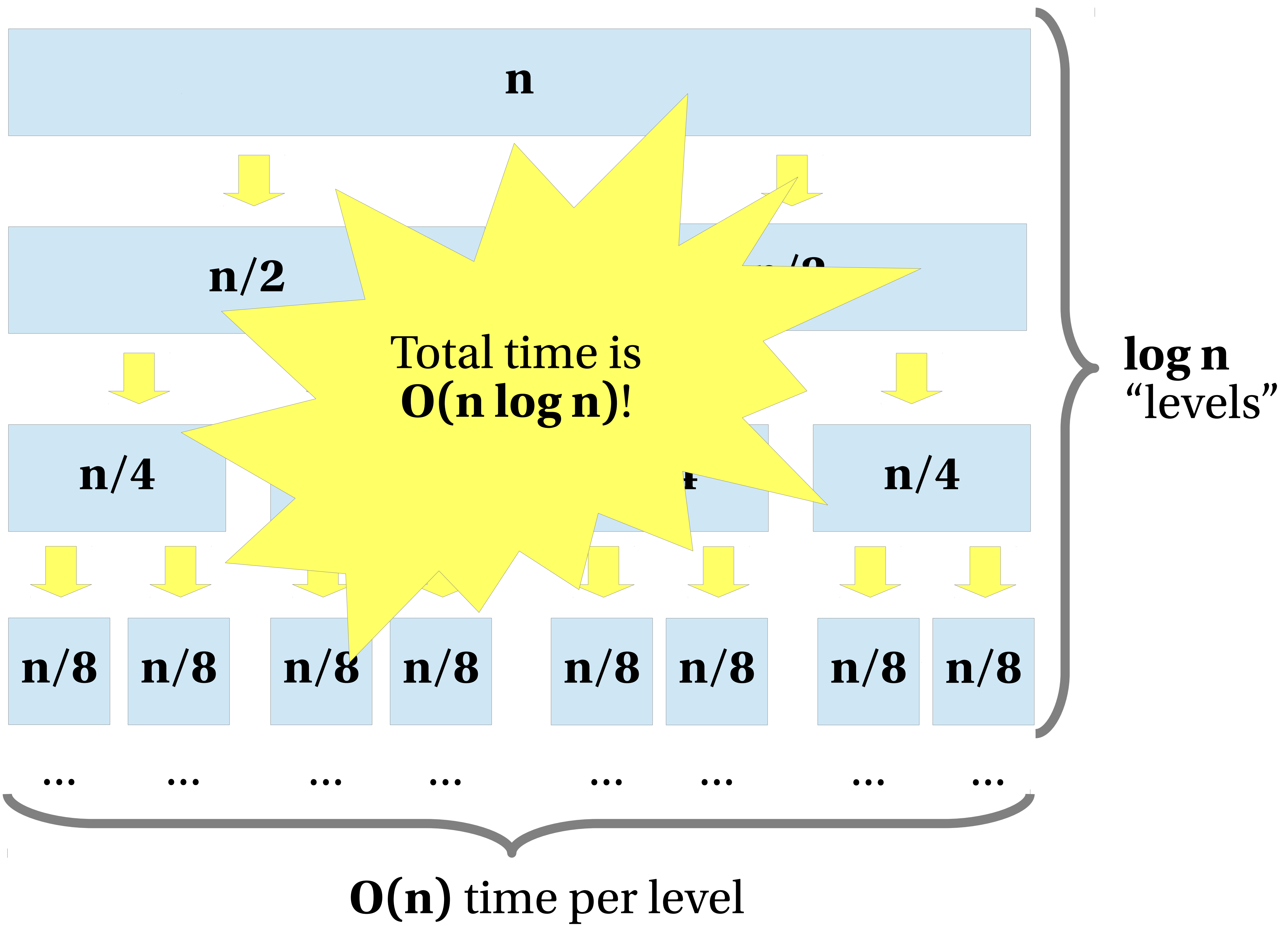
An array of size n gets split into two arrays of size n/2:



Complexity of mergesort

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 analysis

Mergesort's complexity is O(n log n)

- Recursion goes log n “levels” deep
- At each level there is a total of O(n) work

General “divide and conquer” theorem:

- If an algorithm does O(n) work to split the input into two pieces of size n/2 (or k pieces of size n/k)...
- ...then recursively processes those pieces...
- ...then does O(n) work to recombine the results...
- ...then the complexity is O(n log n)

Sorting so far

There are a *huge* number of sorting algorithms

- No single best one, each has advantages (hopefully) and disadvantages

Insertion sort:

- $O(n^2)$ so not good overall
- Good on small arrays though – low *constant factors*

Merge sort:

- $O(n \log n)$, hooray!
- But not in-place and high constant factors