Real-world sorting

(not on exam)

Sorting algorithms so far

	Worst case	Averag case	Best case
Bubblesort	O(n		O(n) or O(n^2)
Selection sort			O(n^2)
Insertion sort			O(n)
Quicksort		lo	O(n log n)
Mergesort	O(n n)	O(n log n)	O(n log n)

No clear winner…
the best algorithms
combine ideas
from several

Introsort

Quicksort: fast in practice, but $O(n^2)$ worst case

Introsort:

- Start with Quicksort
- If the recursion depth gets too big, switch to heapsort, which is O(n log n)

Plus standard Quicksort optimisations:

- Choose pivot via median-of-three
- Switch to insertion sort for small arrays

Used by e.g. C++ STL, .NET, …

Dual-pivot quicksort

Instead of one pivot, pick two

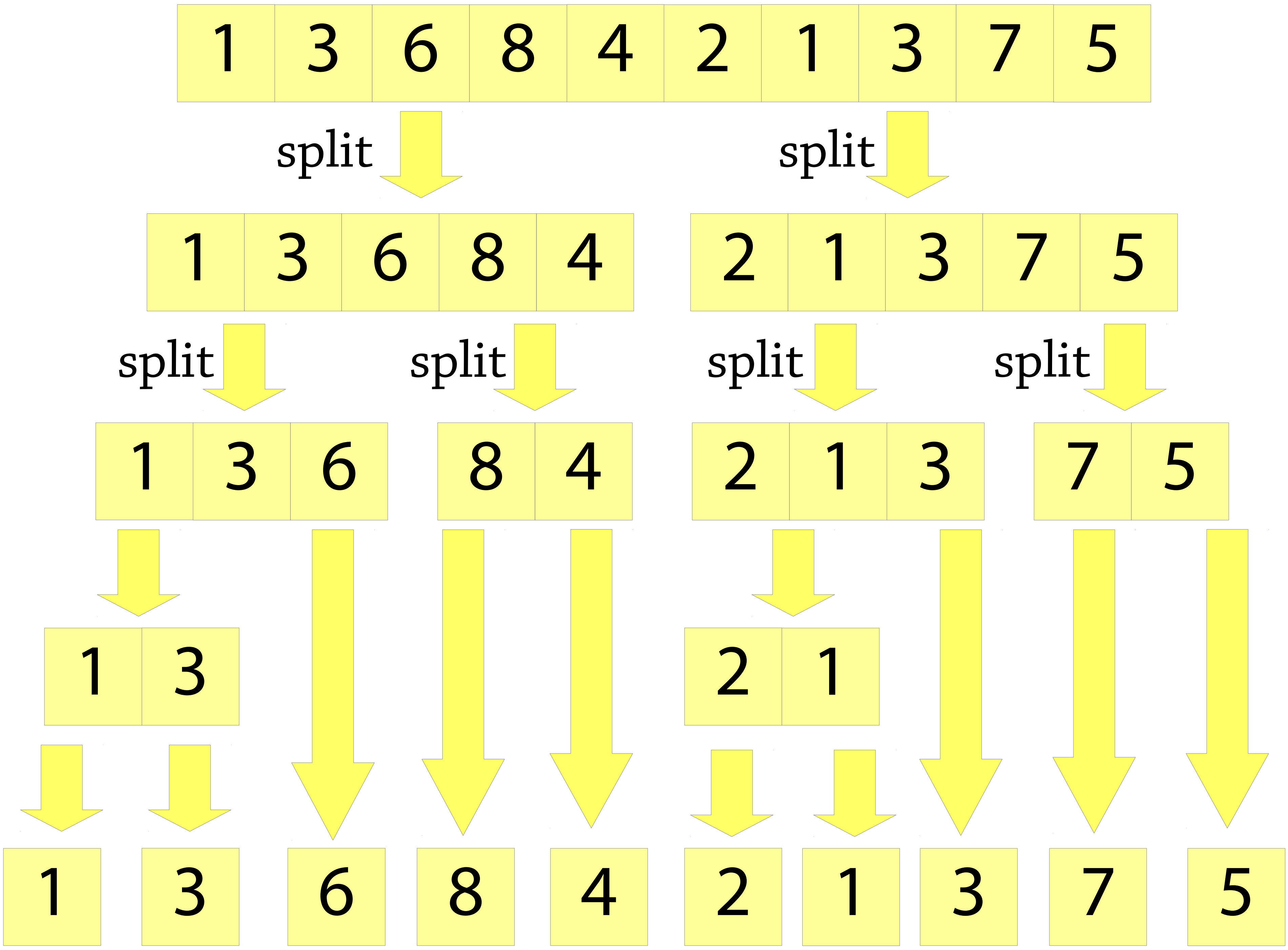
Instead of partitioning the array into two pieces, partition it into three pieces

- If pivots are x and y, then:
- elements $< x$
- elements $> x$ and $< y$
- elements > y

Same complexity as Quicksort, but fewer recursive calls because the array gets split up quicker

Used by Java for primitive types (int, ...)

Traditional merge sort
(a recap)



1 3 6 8 4 2 1 3 7 5

split split

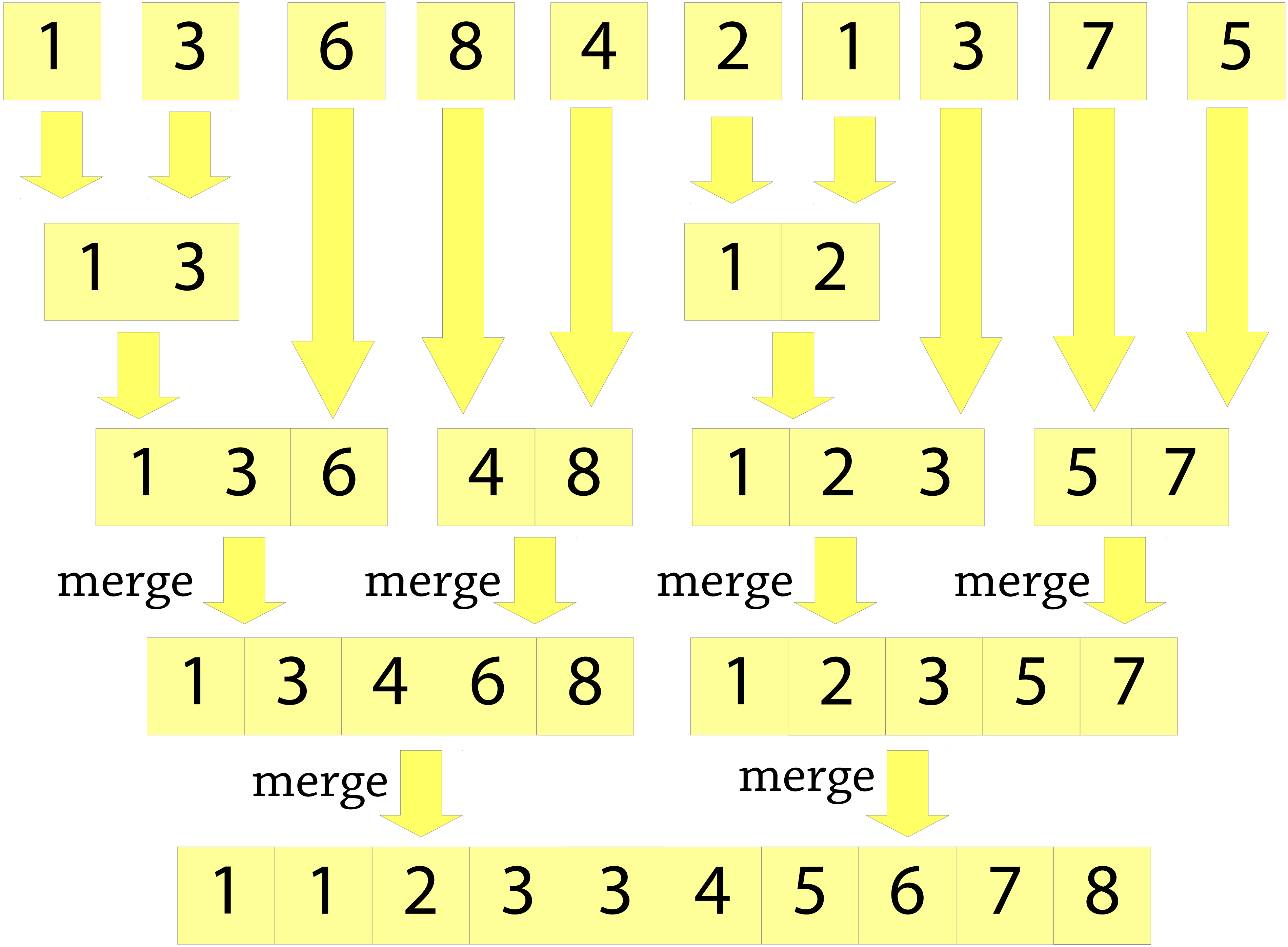
1 3 6 8 4   2 1 3 7 5

split split split split

1 3 6   8 4   2 1 3   7 5

1 3   2 1

1   3   6   8   4   2   1   3   7   5



1 3 6 8 4 2 1 3 7 5

1 3 1 2

1 3 6 4 8 1 2 3 5 7

merge merge merge merge

1 3 4 6 8 1 2 3 5 7

merge merge

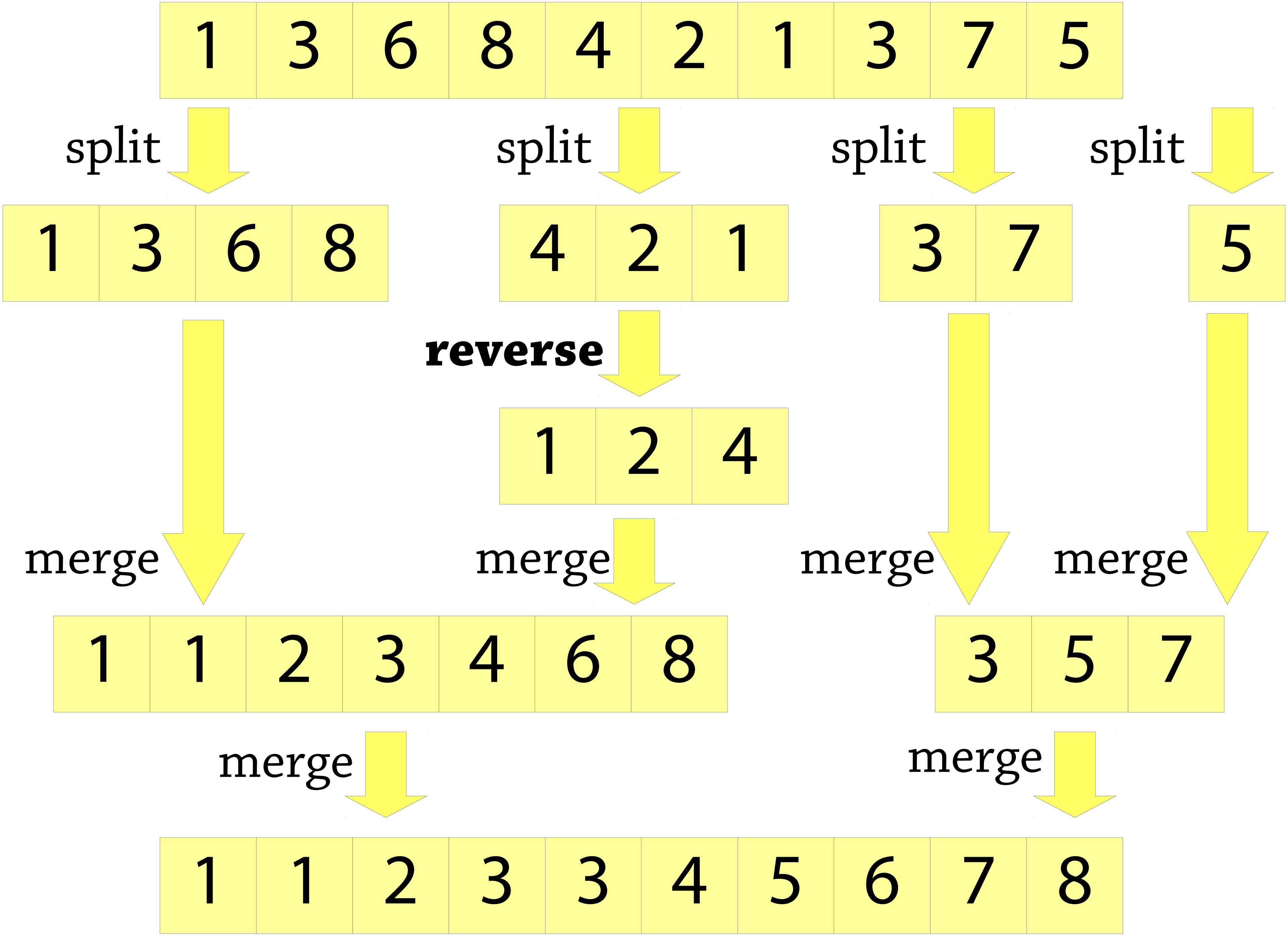
1 1 2 3 3 4 5 6 7 8

Natural merge sort

Traditional merge sort splits the input list into *single elements* before merging everything back together again

Better idea: split the input into *runs*

- A run is a sequence of increasing elements
- …or a sequence of decreasing elements
- First reverse all the decreasing runs, so they become increasing
- Then merge all the runs together



1 3 6 8 4 2 1 3 7 5

split split split split

1 3 6 8

4 2 1

3 7

5

reverse

1 2 4

merge merge merge merge

1 1 2 3 4 6 8

3 5 7

merge merge

1 1 2 3 3 4 5 6 7 8

Natural merge sort

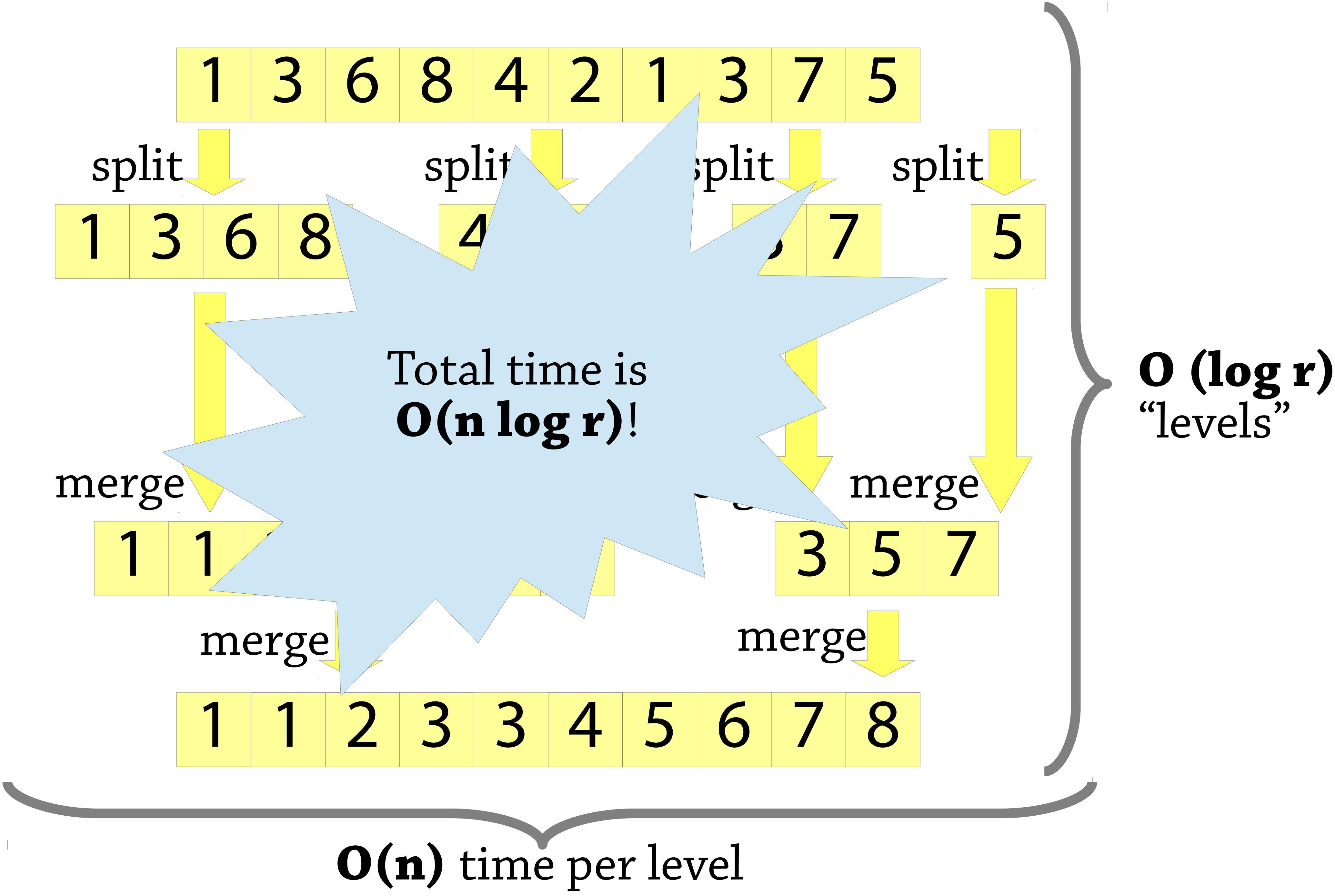
Big advantage: O(n) time for sorted data

- ...and reverse-sorted data
- ...and "almost"-sorted data

Complexity: O(n log r), where r is the number of runs in the input

- ...worst case, each run has two elements, so r = n/2, so O(n log n)

Used by GHC



Total time is **O(n log r)**!

O (log r) "levels"

O(n) time per level

Timsort

Natural mergesort is really good on sorted/nearly-sorted data

- You get long runs so not many merges to do

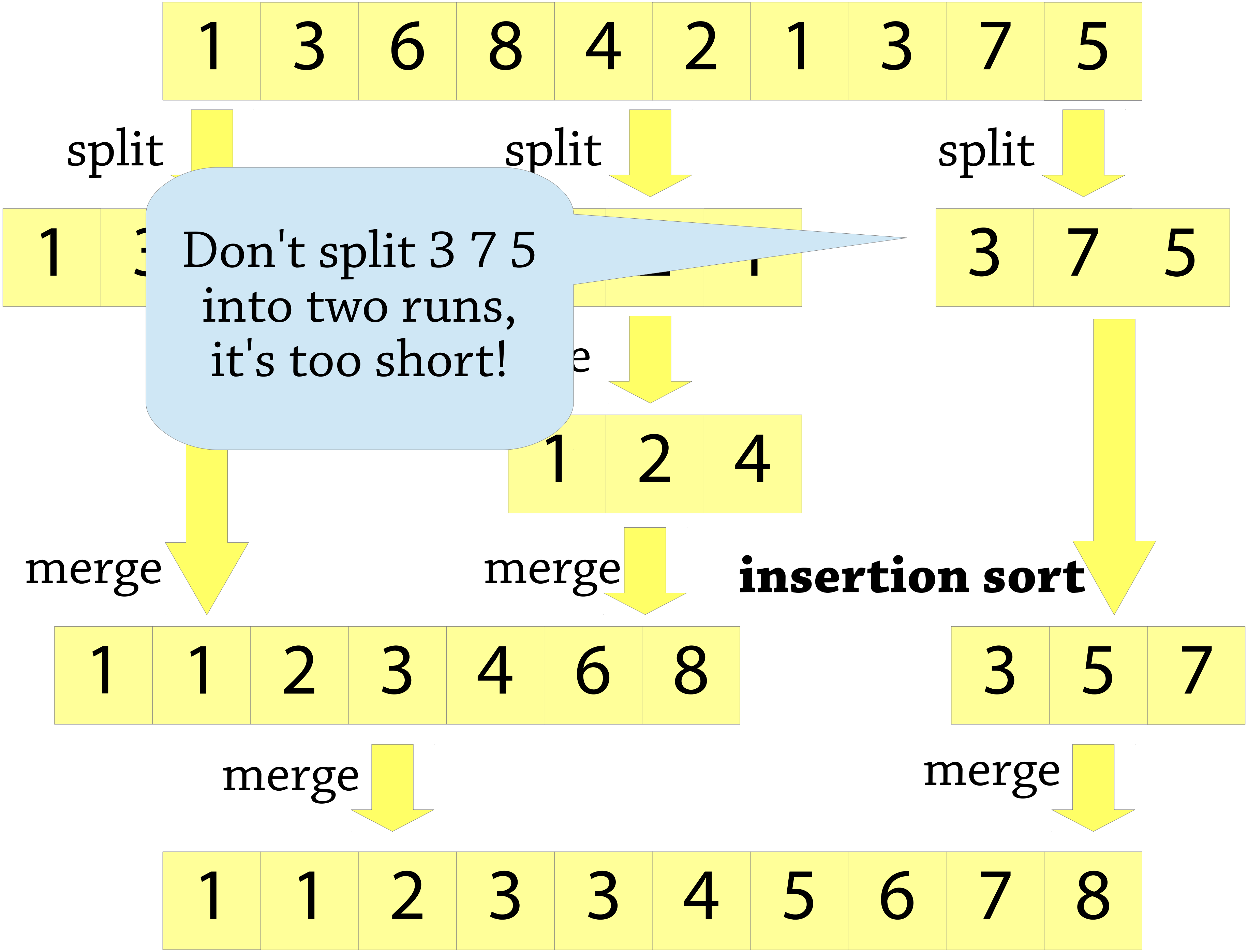
But not so good on random data

- Short runs so many merges

Idea of Timsort: on short random parts of the list, *switch to insertion sort*

How to detect randomness?

- Several short runs next to one another



1 3 6 8 4 2 1 3 7 5

split split split

1 3 2 1 3 7 5

Don't split 3 7 5
into two runs,
it's too short!

e

1 2 4

merge merge **insertion sort**

1 1 2 3 4 6 8 3 5 7

merge merge

1 1 2 3 3 4 5 6 7 8

Timsort

Specifically:

- If we come across a short run, join it together with all following short runs until we reach a threshold
- Then use insertion sort on that part

Also some optimisations for merge:

- Merge smaller runs together first
- If the merge begins with several elements from the same array, use binary search to find out how many and then copy them all in one go

Used in Java for arrays of objects, Python

http://en.wikipedia.org/wiki/Timsort

Summary

No one-size fits all answer

- Best overall complexity: natural mergesort
- But quicksort has smaller constant factors

Different algorithms are good in different situations

- ...you should find out which in the lab :)

Best sorting functions combine ideas from several algorithms

- Introsort: quicksort+heapsort+insertion sort
- Timsort: natural mergesort+insertion sort