CSC 148: Introduction to Computer Science

Week 11

Efficiency considerations

Sorting efficiency

University of Toronto Mississauga,

Department of Mathematical and Computational Sciences

sorting

- How does the time to sort a list with n elements vary with n?

- 108 sorts:
  - bubble sort -> n^2
  - selection sort -> n^2
  - insertion sort -> n^2

sorting

- How does the time to sort a list with n elements vary with n?

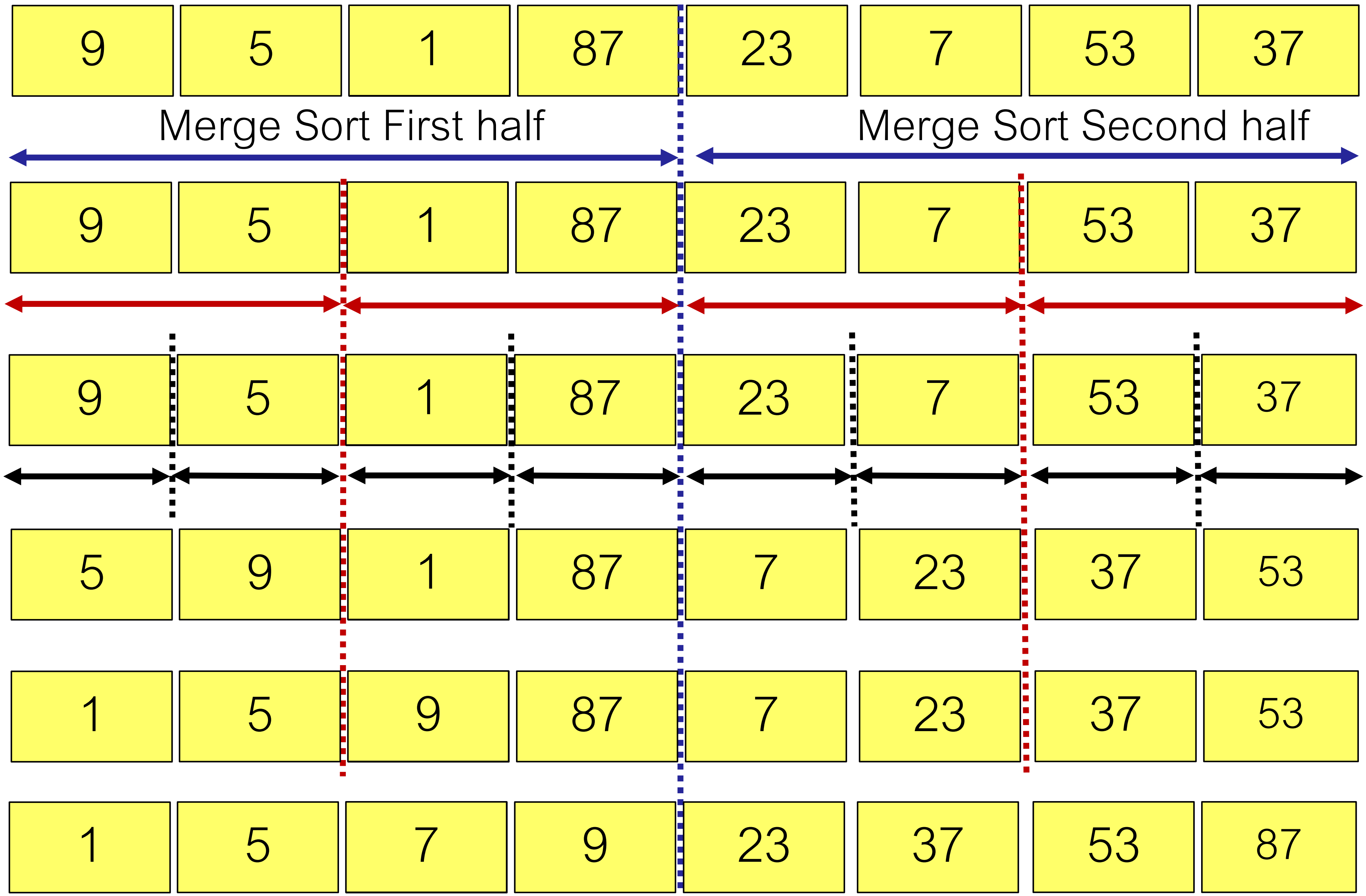
- 108 sorts:
  - bubble sort -> n^2
  - selection sort -> n^2
  - insertion sort -> n^2
- Some other sort?
  - Merge sort?
  - Quicksort?
  - Radix sorts?
    - Apropos of nothing: Andrew's favorite for sorting exams

Merge Sort

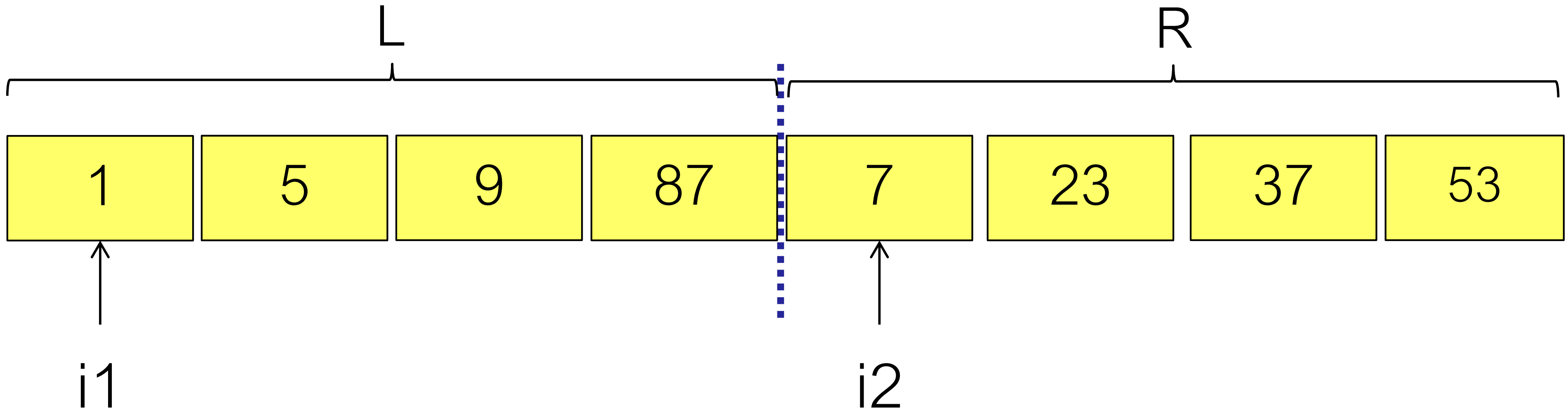
Recursive algorithm based on the idea of “divide and conquer”

- Split a list in two halves repeatedly
- Halves with 0 or 1 elements are guaranteed sorted
- Merge the two halves "on the way back”
    - All the work is in merging!

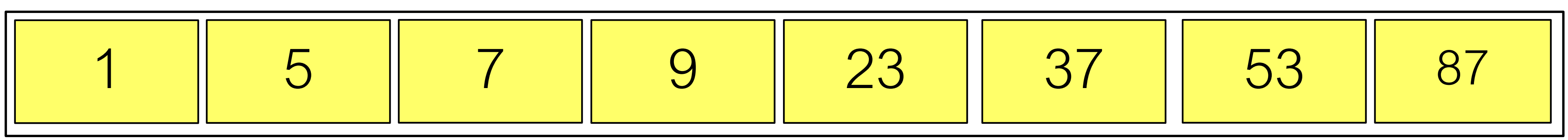
Merge Sort: split all the way, then merge



Merge step: merge(L, R)



sorted list (different than L or R)



The halves might not be perfectly equal though...

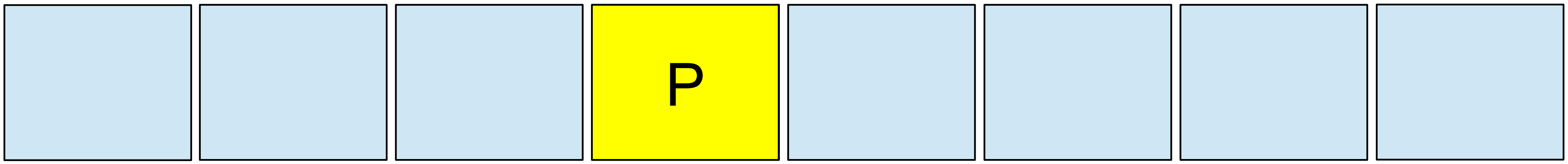
Quicksort

Another "divide and conquer" algorithm – but doesn't necessarily split into *half*.

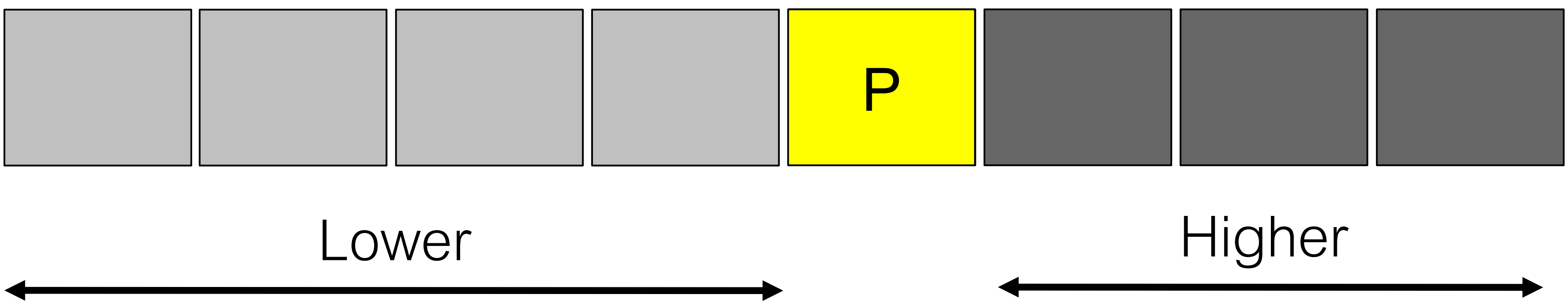
- Split a list ("partitioning") into the part smaller than some value (called pivot) and the part not smaller than that value
- Sort these two parts
- Recombine the list

Partitioning

- Begin with the unsorted list and select a pivot P at random



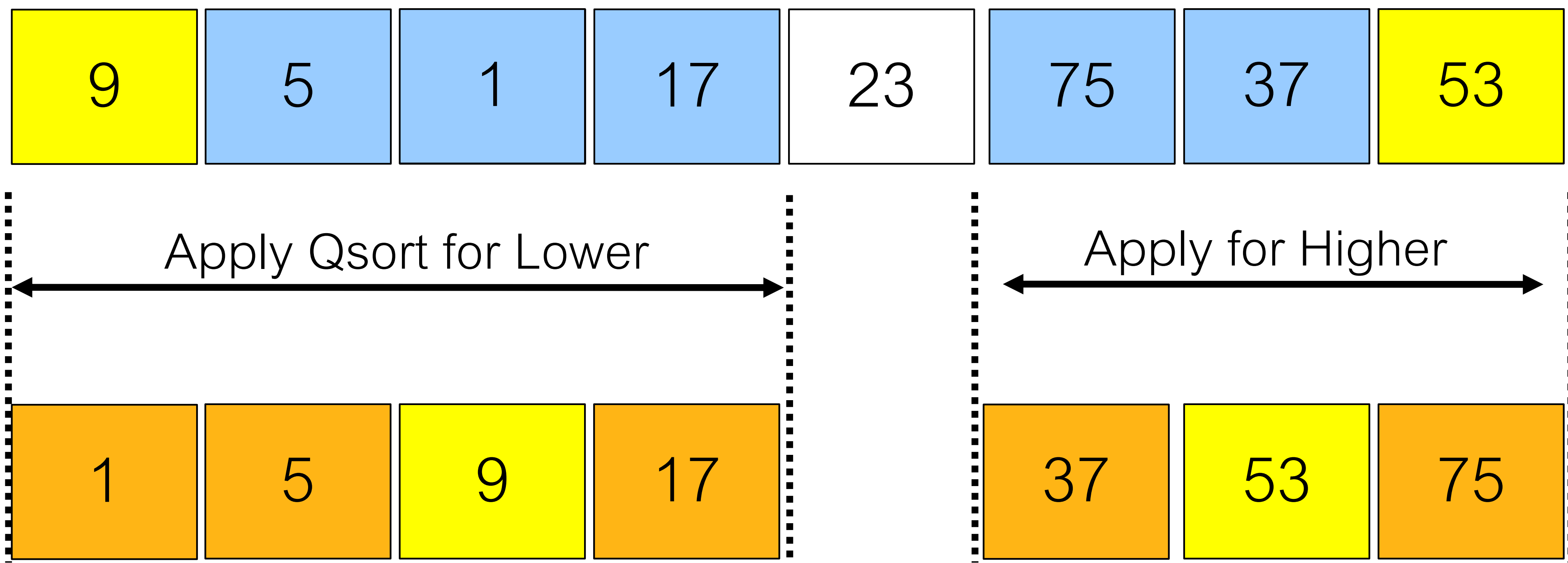
- Split list such that all elements to the left are lower than P and all to the right are higher



- Several ways to do the partition step ...

Quicksort Recursion

- Recurse: repeat the same idea for the two partitions
- Pick pivot, process such that all lower than it are on the left, all higher on the right



Worksheet!

- How do we analyse the running time of recursive algorithms in general? (Not just for trees.)

- Two key parts:
  - how long do the non-recursive parts take?
  - what is the structure of the recursive calls?

Merge sort

- idea: break a list up (partition) into two halves, merge sort each half, then recombine (merge) the halves

```
def mergesort(lst: list) -> list:
    """Return a sorted list with the same elements as <lst>.

    This is a *non-mutating* version of mergesort; it does not mutate the
    input list.
    """
    if len(lst) < 2:
        return lst[:]
    else:
        mid = len(lst) // 2
        left_sorted = mergesort(lst[:mid])
        right_sorted = mergesort(lst[mid:])

        return _merge(left_sorted, right_sorted)
```

Lists of length < 2 are already sorted

First half

Second half

Merge the two sorted halves, "on the way back". How?

Counting Merge Sort

- Assume a list of size n
- Merge operation takes linear time ... why?
- The "divide" step also takes linear time (approx n steps) ... why?
- What about the cost of the two recursive calls?

Counting Merge Sort: $n = 8$

prop. to log n

splits

ms([4, 2, 6, 8, 1, 3, 5, 7])

merge(ms([4, 2, 6, 8]) , ms([1, 3, 5, 7]))

merge(merge(ms([4,2]), ms([6,8])) , merge(ms([1,3]), ms ([5,7]))

merge(merge(merge(ms([4]),ms([2])),merge(ms([6]),ms([8])), merge(merge(ms([1]),ms([3])),merge(ms([5]),ms([7])

merge(merge(merge([4],[2]),merge([6],[8])), merge(merge([1],[3]),merge([5],[7])))),

merge(merge([2,4],[6,8]),merge([1,3],[5,7]))

merge([2,4,6,8], [1,3,5,7])

[1, 2, 3, 4, 5, 6, 7, 8]

log n merge with

prop. to n copies

=> n x log n

Merge Sort

```
def mergesort(lst):
    if len(lst) < 2:
        return lst[:]
    else:
        mid = len(lst) // 2
        left = lst[:mid]
        right = lst[mid:]

        left_sorted = mergesort(left)
        right_sorted = mergesort(right)

        return _merge(left_sorted, right_sorted)
```

Quicksort

- idea: break a list up (partition) into the part smaller than some value (pivot) and not smaller than that value, sort these parts, then recombine the list:

```
def quicksort(lst: list) -> list:
    """
    Return a sorted list with the same elements as <lst>.
    This is a *non-mutating* version of quicksort; it does not mutate the
    input list.
    """
    if len(lst) < 2:
        return lst[:]
    else:
        # smaller, bigger = _partition(lst[1:], lst[0])
        smaller = [i  for  i  in  lst[1:]  if  i < lst[0]]
        bigger = [i  for  i  in  lst[1:]  if  i >= lst[0]]
        return (quicksort(smaller) +
                [lst[0]] +
                quicksort(bigger) )
```

Lists of length < 2 are already sorted

Simple partition step

Sort smaller elements

in its correct position

Sort larger elements

Counting Quicksort: $n = 7$

n comparisons at each split

log n splits

qs([4, 2, 6, 1, 3, 5, 7])

qs([2, 1, 3]) + [4] + qs([6, 5, 7])

qs([1]) + [2] + qs([3]) + [4] + qs([5]) + [6] + qs([7])

[1] + [2] + [3] + [4] + [5] + [6] + [7]

[1, 2, 3] + [4] + [5, 6, 7]

[1, 2, 3, 4, 5, 6, 7]

log n concatenation, with n work to concatenate

=> n x log n

Quicksort

```
def quicksort(lst):
    if len(lst) < 2:
        return lst[:]
    else:
        pivot = lst[0]

        smaller, bigger = _partition(lst[1:], pivot)

        smaller_sorted = quicksort(smaller)
        bigger_sorted = quicksort(bigger)

        return smaller_sorted + [pivot] +
               bigger_sorted
```

Quicksort

Do we always have *n log n*?

Mergesort: we know we always split in halves, no matter what

Quicksort: no guarantees, depends on how we pick the pivot

- What's the average case? What's the worst case?

Quicksort: Good *on Average*

- If we always choose a pivot that's an approximate median, then the two partitions are roughly equal, and the running time is **O(n log(n))**

- If we always choose a pivot that's an approximate min/max, then they two partitions are very unequal, and the running time is **$O(n^2)$**

- In practice, with good pivot selection, it's rare to come across worst case behaviour.

The limitations of Big-Oh

- Big-Oh notation is a simplification of running time analysis and allows us to ignore constants when analysing efficiency.

- But constants can make a difference, too!

- $O(n \log n)$ Merge Sort vs. $O(n \log n)$ Quicksort vs. $O(n^2)$ Bubble