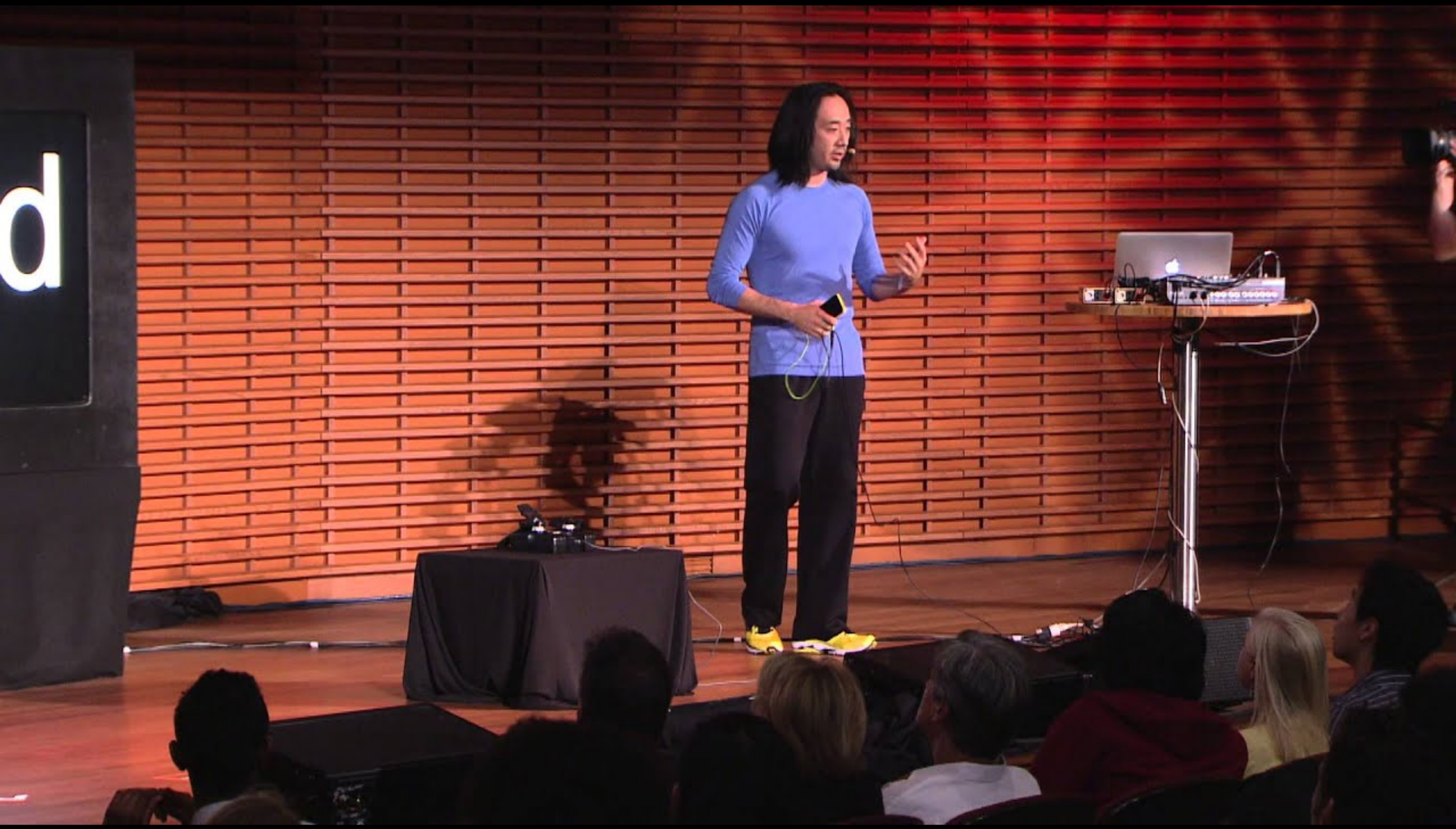
CSC 148

Week 11

Sadia ‘Rain’ Sharmin

Classes begin 10 minutes after the hour



This Week's Feature: *Ge Wang*

Recursive Sorting Algorithms

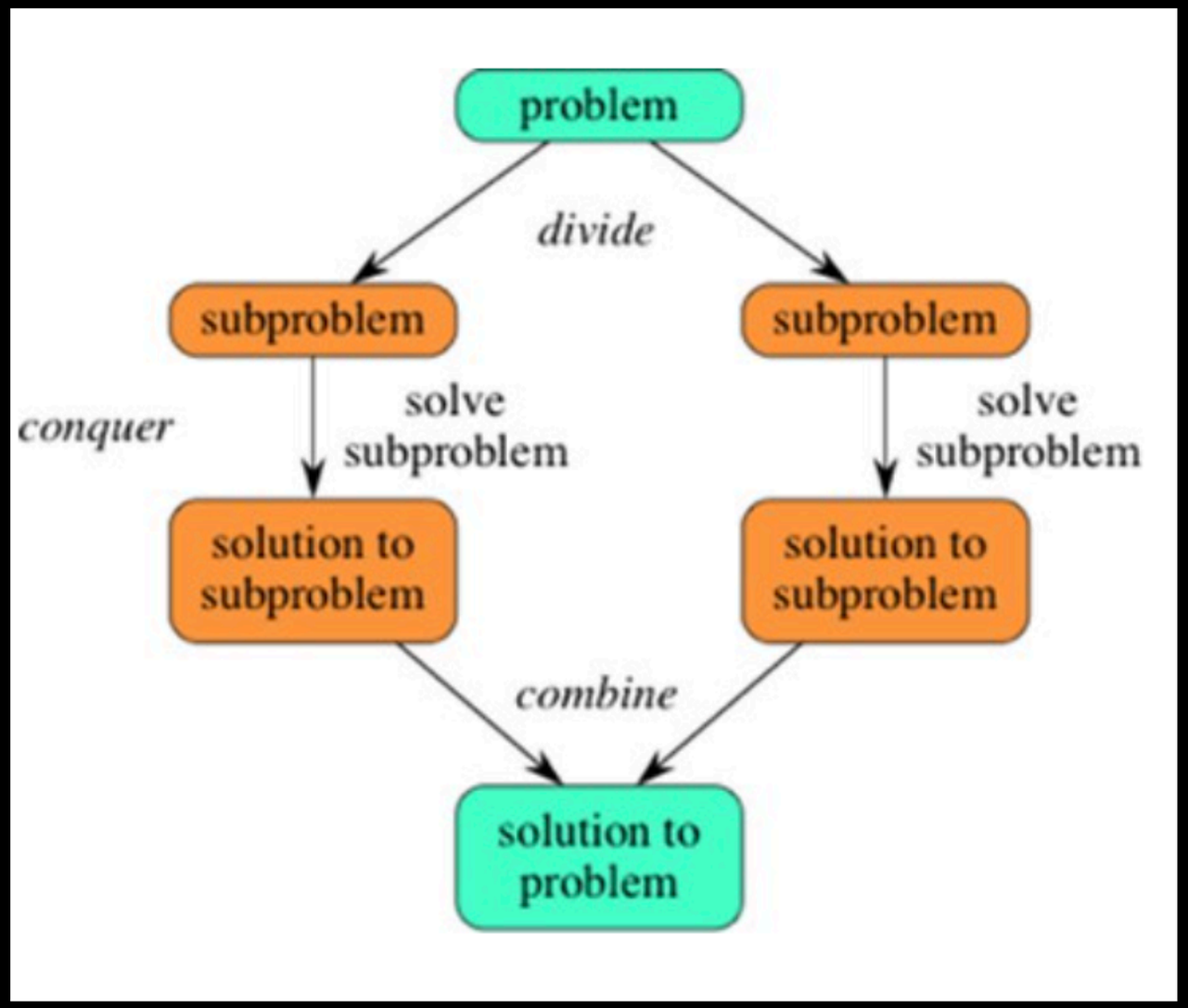
Divide & Conquer

Splitting Lists

1. Divide the input list into smaller lists.

2. Recurse on each smaller list.

3. Combine the results of each recursive call.



Divide & Conquer

Sorting Algorithms

Mergesort and Quicksort are two examples of divide and conquer algorithms

Recursive Sorting

Mergesort Algorithm

- Split a list in two halves repeatedly
- Halves with 0 or 1 elements are guaranteed sorted
- Merge the two halves "on the way back"

Mergesort

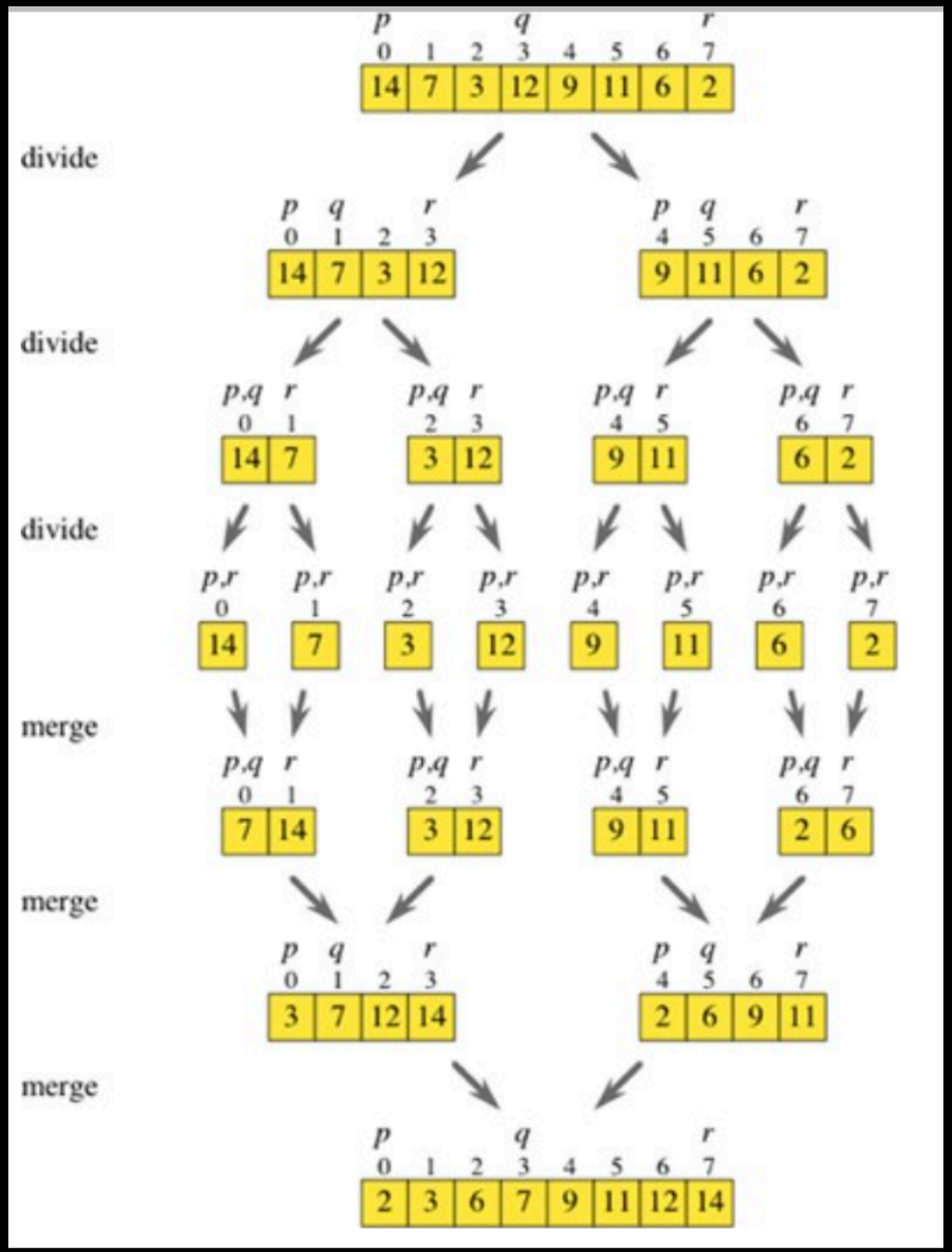
Split all the way, then merge

In the diagram:

q = middle position

left side = lst[p:q]

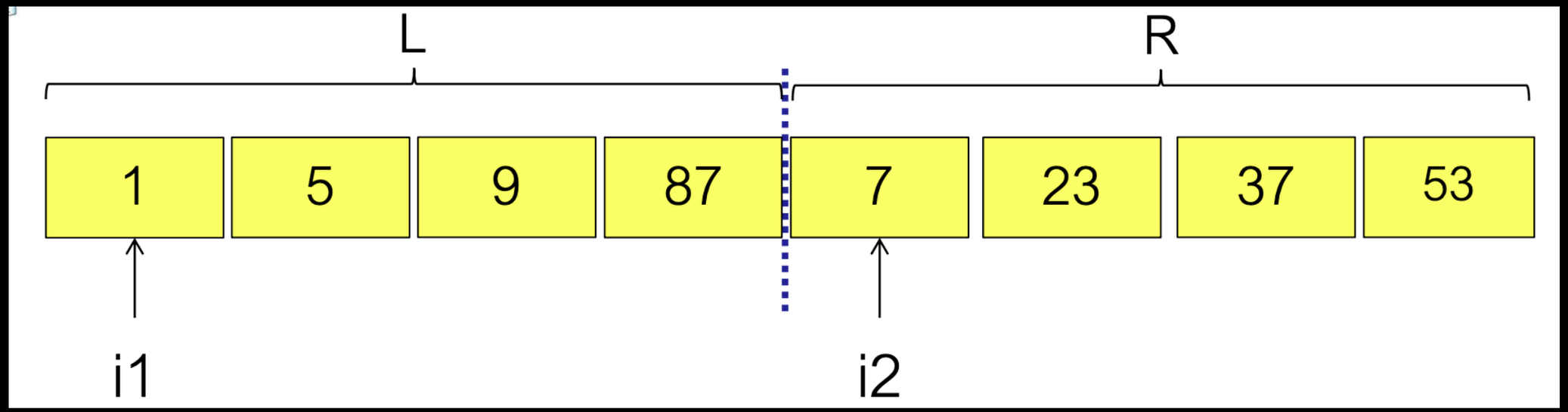
right subproblem = lst[q:r]



Recursive Sorting

Mergesort Algorithm

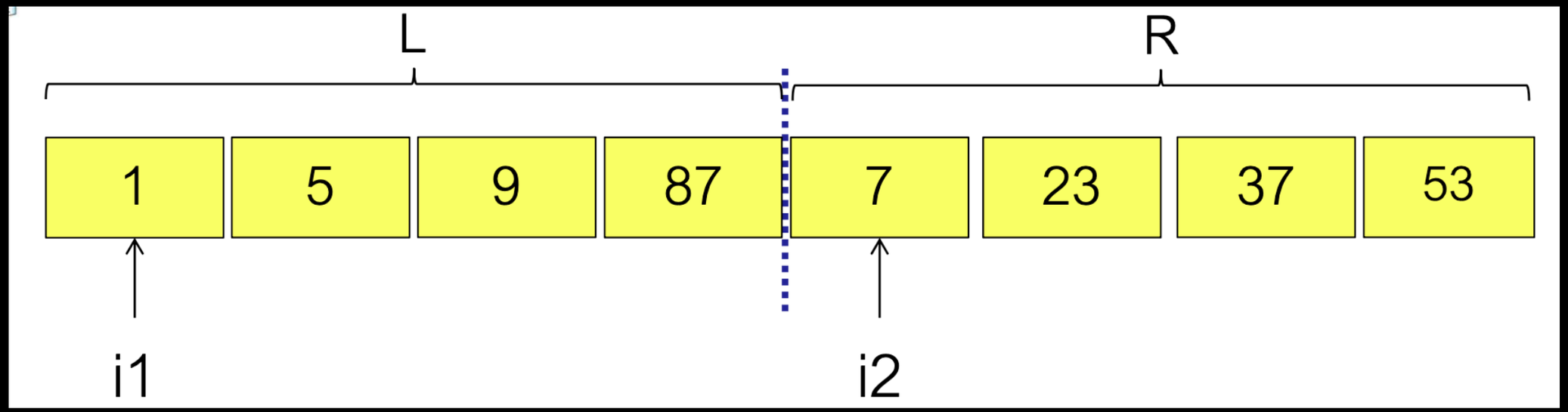
Merge step: merge(L, R)



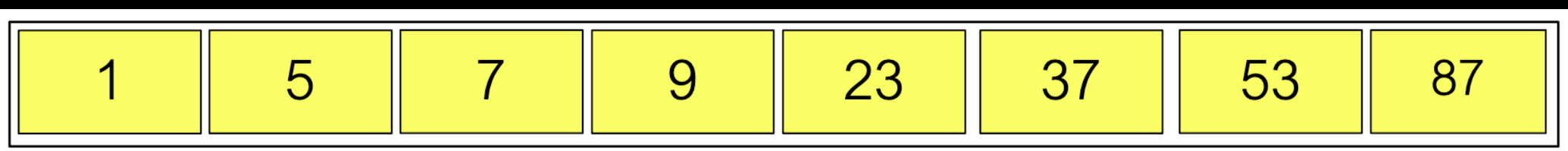
Recursive Sorting

Mergesort Algorithm

Merge step: merge(L, R)



The sorted list:



Efficiency

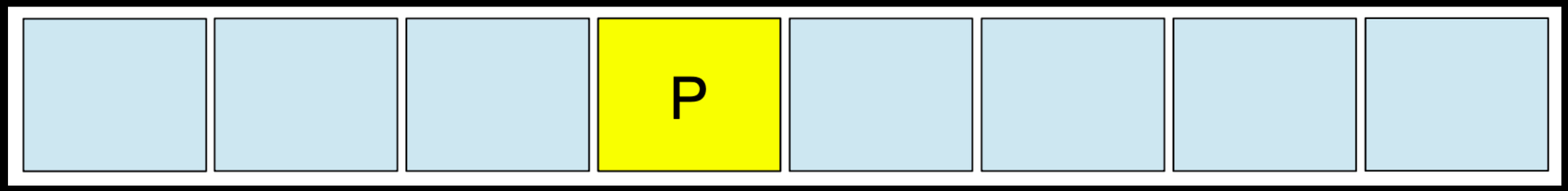
Quicksort

Efficient sorting algorithm that works by:

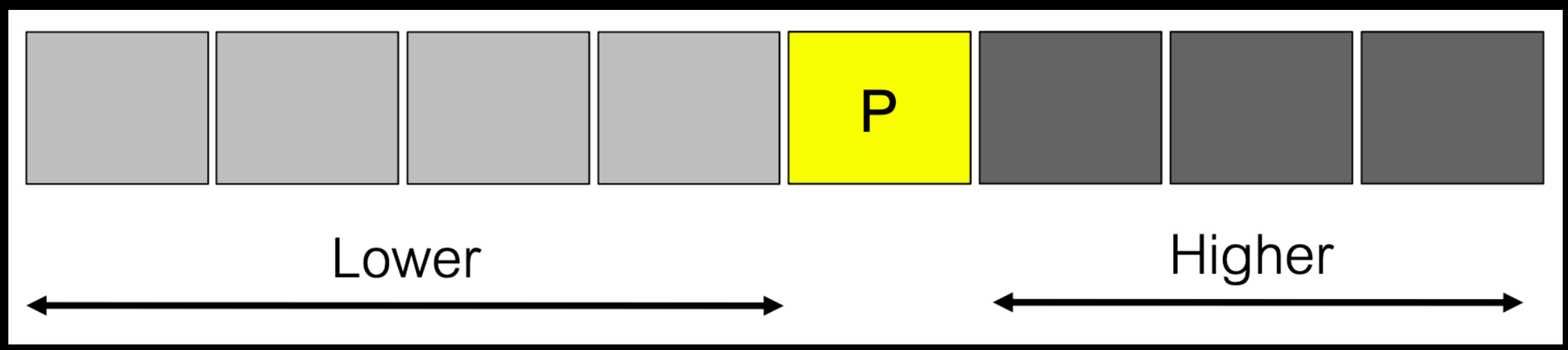
- Splitting a list (partitioning it) into the part smaller than some value (called pivot) and the part not smaller than that value
- Sorting these two parts
- Recombining the list

Partition step

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



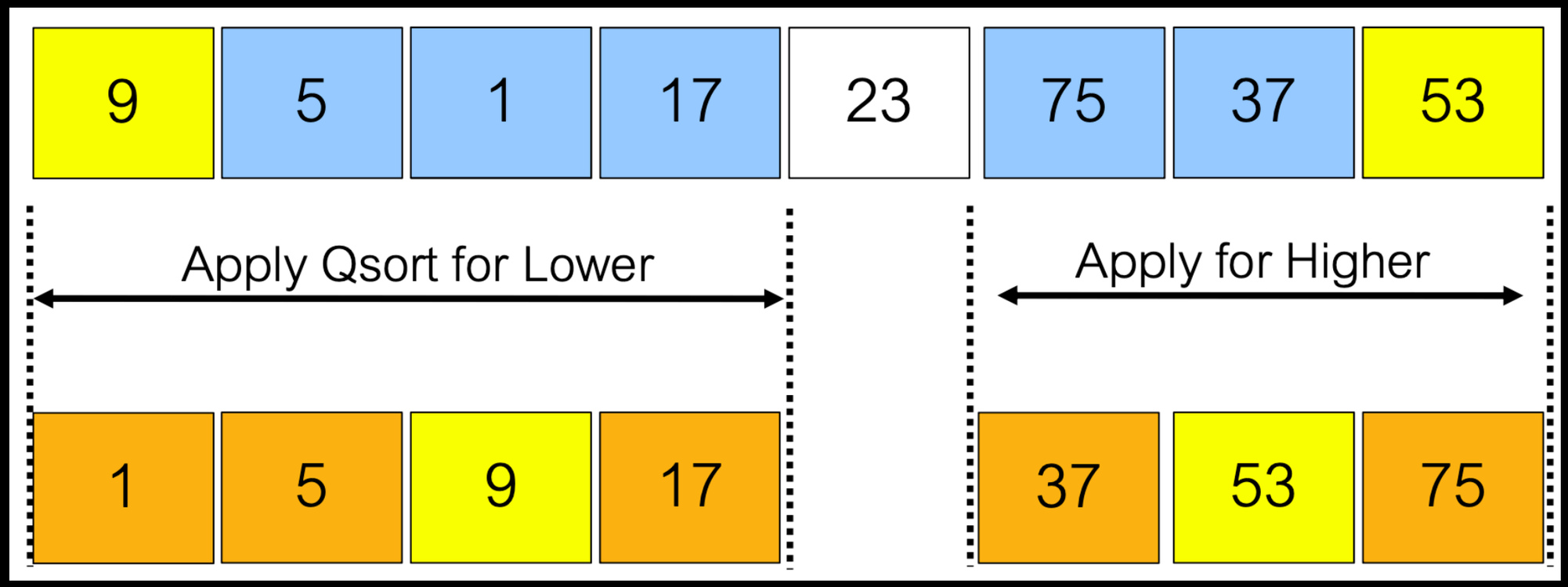
Several ways to implement the partitioning step

Quicksort

Recursive step

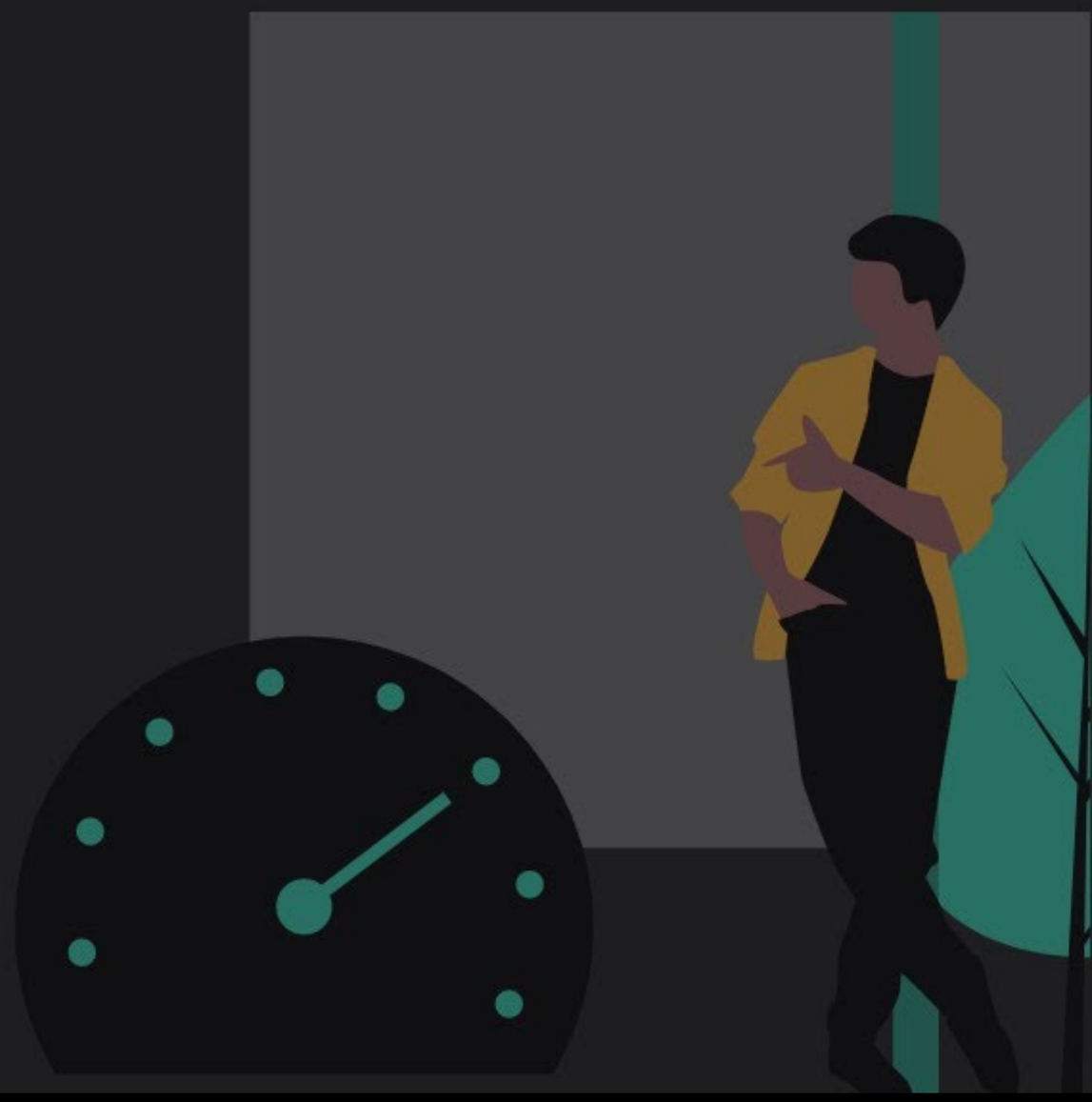
Recurse: repeat the same idea for the two partitions

Pick pivot, process such that all lower than it are on the left, all higher on the right



AlgoRythmics

Quick Sort



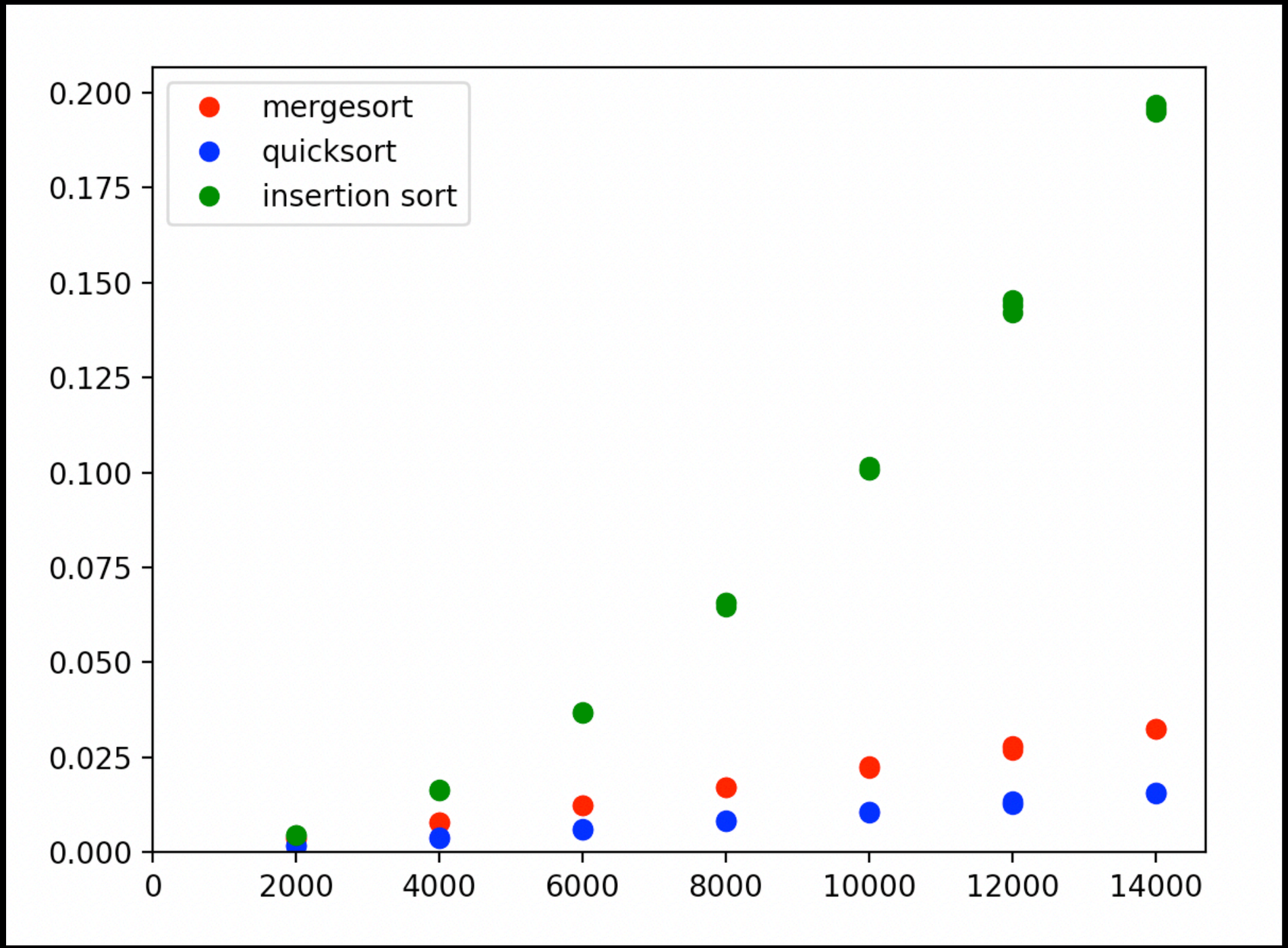
DEMO

Running time of merge sort, quick sort, insertion sort

See profile_sorting.py

Recursive Sorting

Efficiency

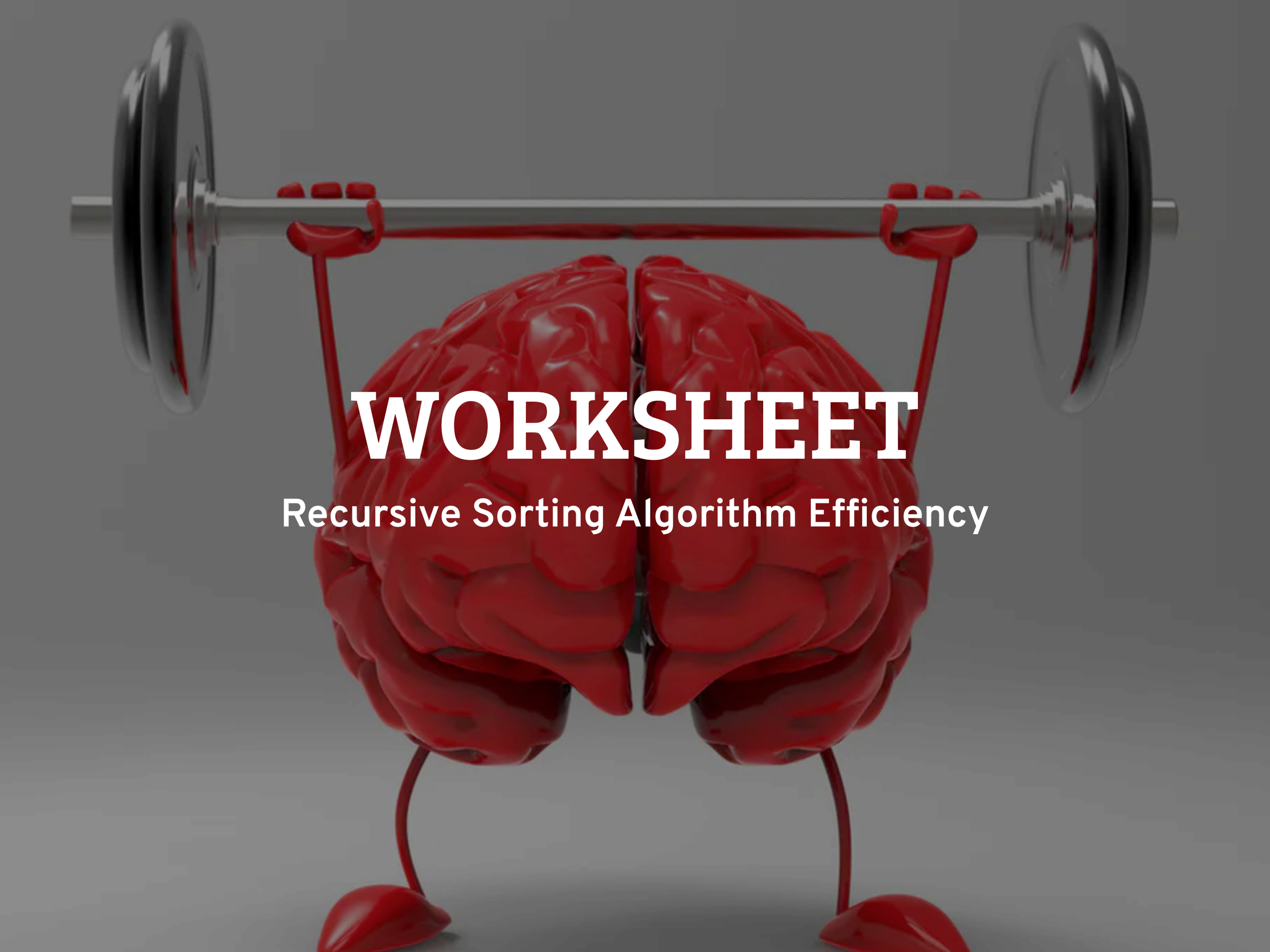


Efficiency

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?



WORKSHEET

Recursive Sorting Algorithm Efficiency

Efficiency

Mergesort

The idea: break a list up (partition) into two halves, mergesort 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?

Efficiency

Mergesort

```
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)
```

Efficiency

Mergesort

Let's do an example with list of size 16 to figure out the pattern for: How many times do we do a recursive call for *n* elements?

Efficiency

Mergesort

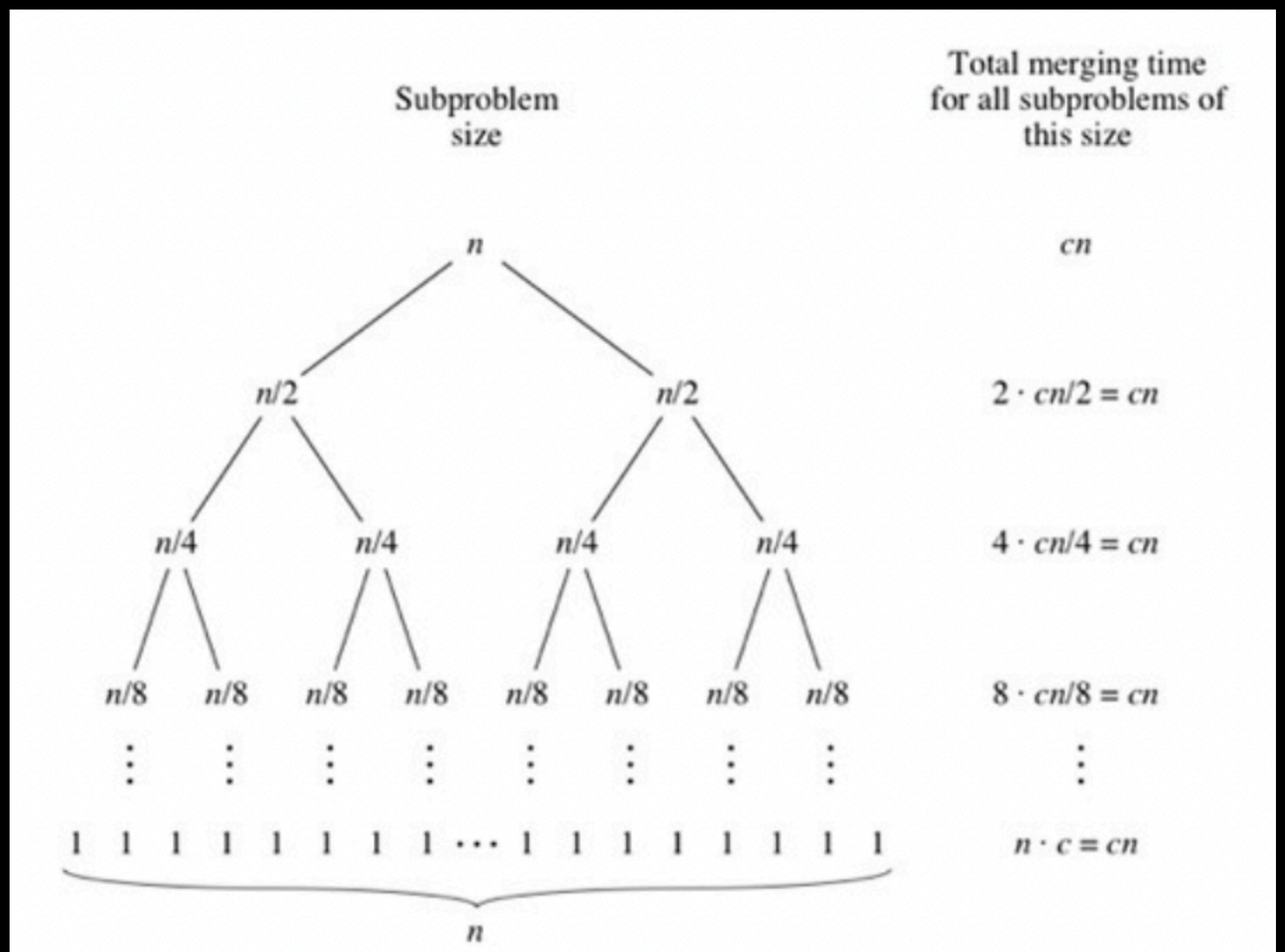
How many times do we do a recursive call for *n* elements?

Efficiency

Mergesort

Time complexity of entire recursion tree:

# of levels * time taken for one level



log n levels

n work each level

= O(n * logn)

Efficiency

Quicksort

The 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

Efficiency

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
```

Efficiency

Quicksort

Let's do an example with list of size 7

Efficiency

Quicksort Analysis

Let's dig a little deeper.

Do we always have n log n though?

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?

Efficiency

Quicksort

Quicksort: in theory, a mixed bag

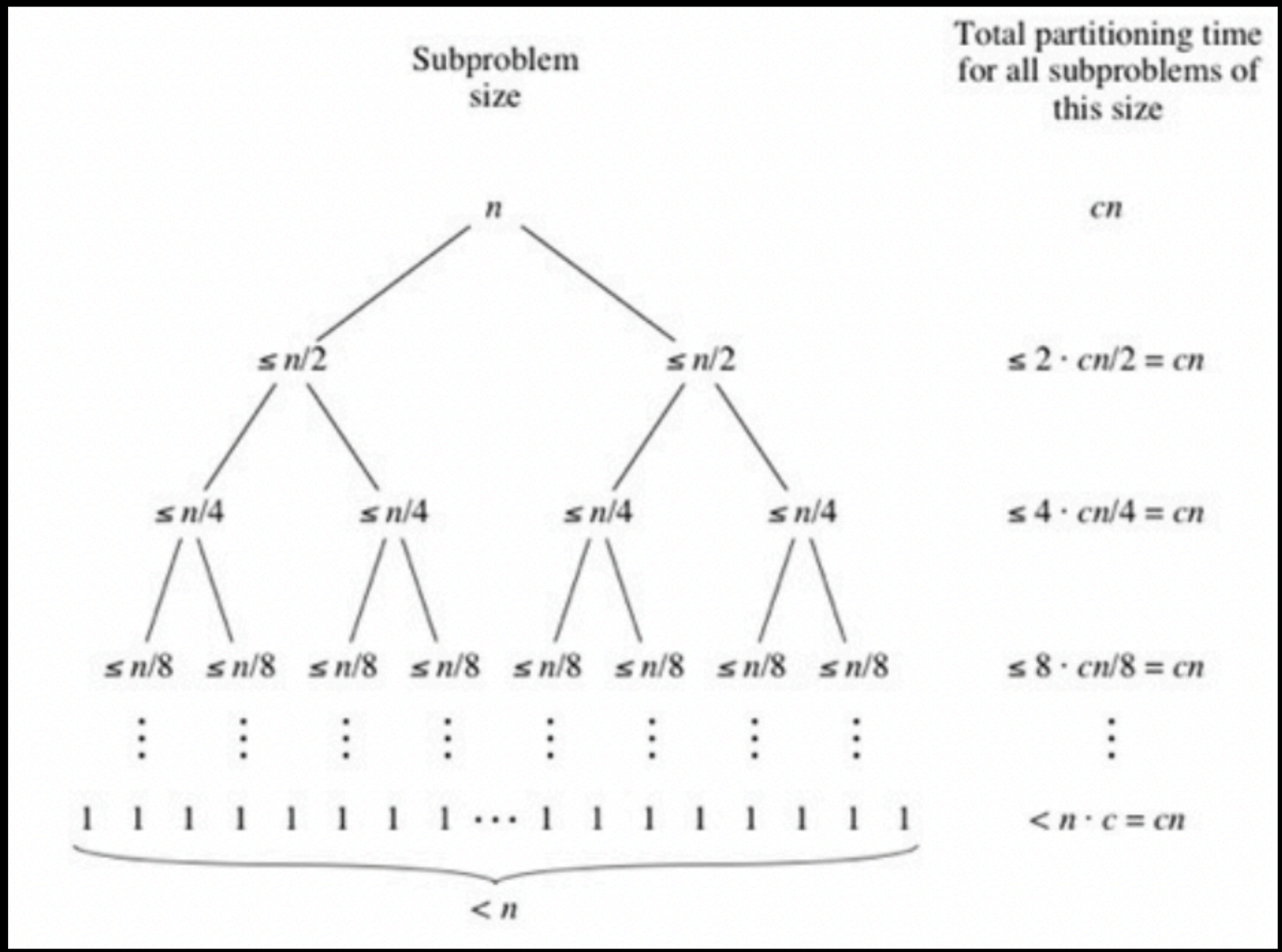
If we always choose a pivot that's an approximate median, then the two partitions are roughly equal, and the running time is O(n log(n)).

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

Efficiency

Quicksort

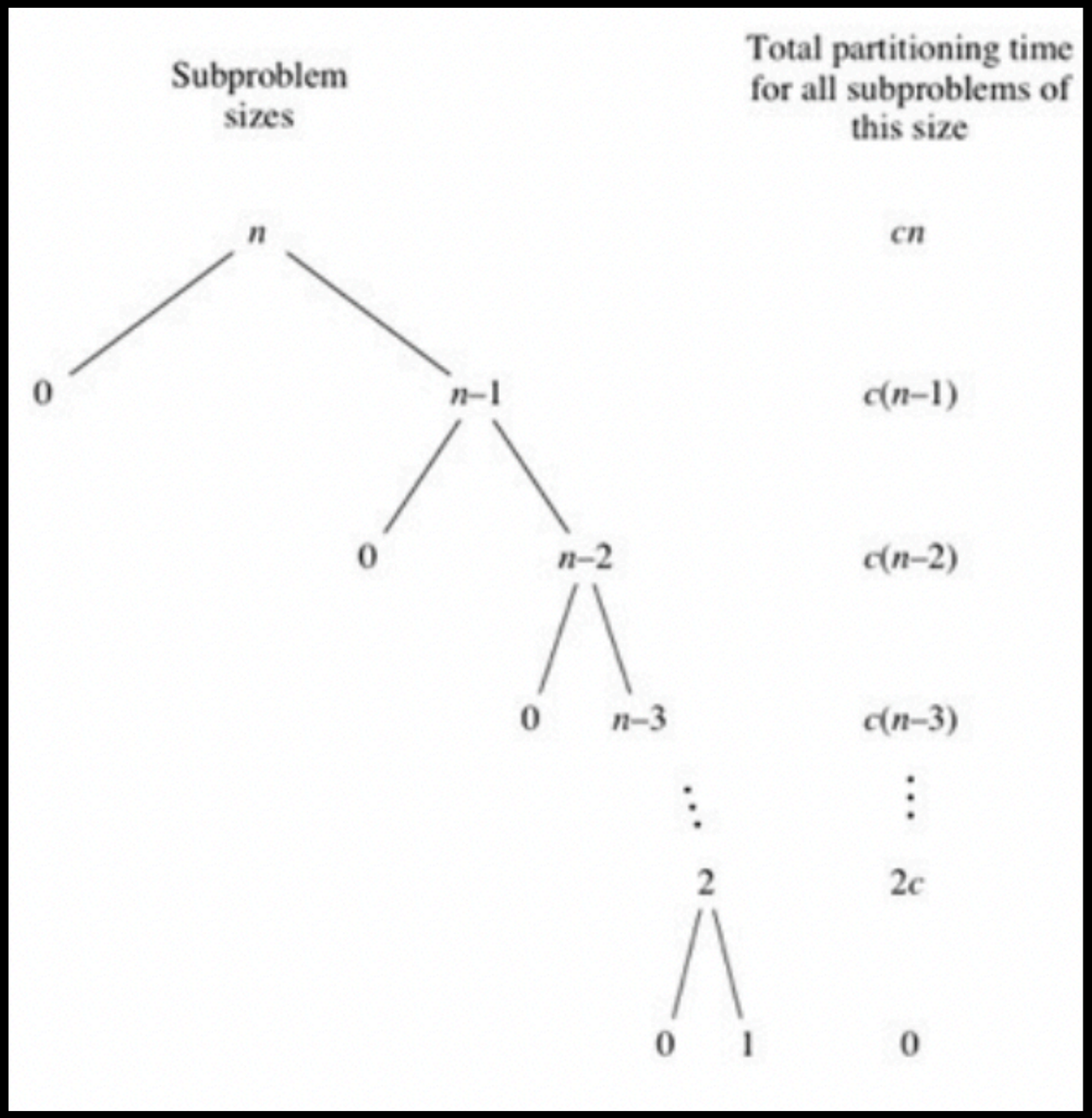
Best case for quicksort



Efficiency

Quicksort

Worst case for quicksort



Efficiency

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!

$O(n \log n)$ vs. $O(n \log n)$ vs. $O(n^2)$

In-place Quicksort

Quicksort

So far...

```
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
```

Our quicksort algorithm returns the new sorted list, does not mutate the original one

How about sorting the original list in place?

Quicksort

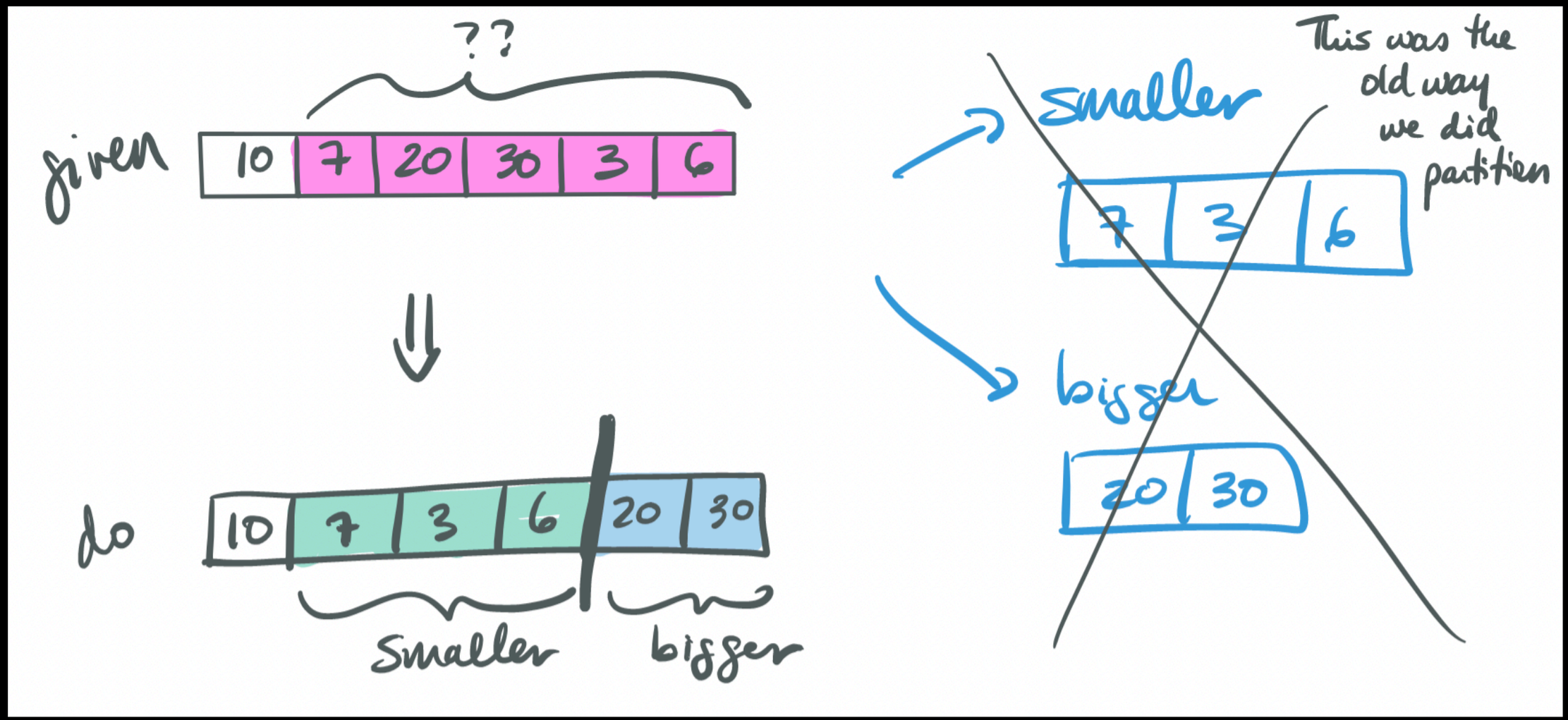
Key idea

For mutating the input list in a space-efficient way, the key helper: in-place partition!

That is, we will write a version of _partition that mutates lst directly

Key idea

Our in-place partition will mutate the list rather than returning two new ones. The mutated list will be organized so that all elements less than the pivot occur before some index.



Quicksort

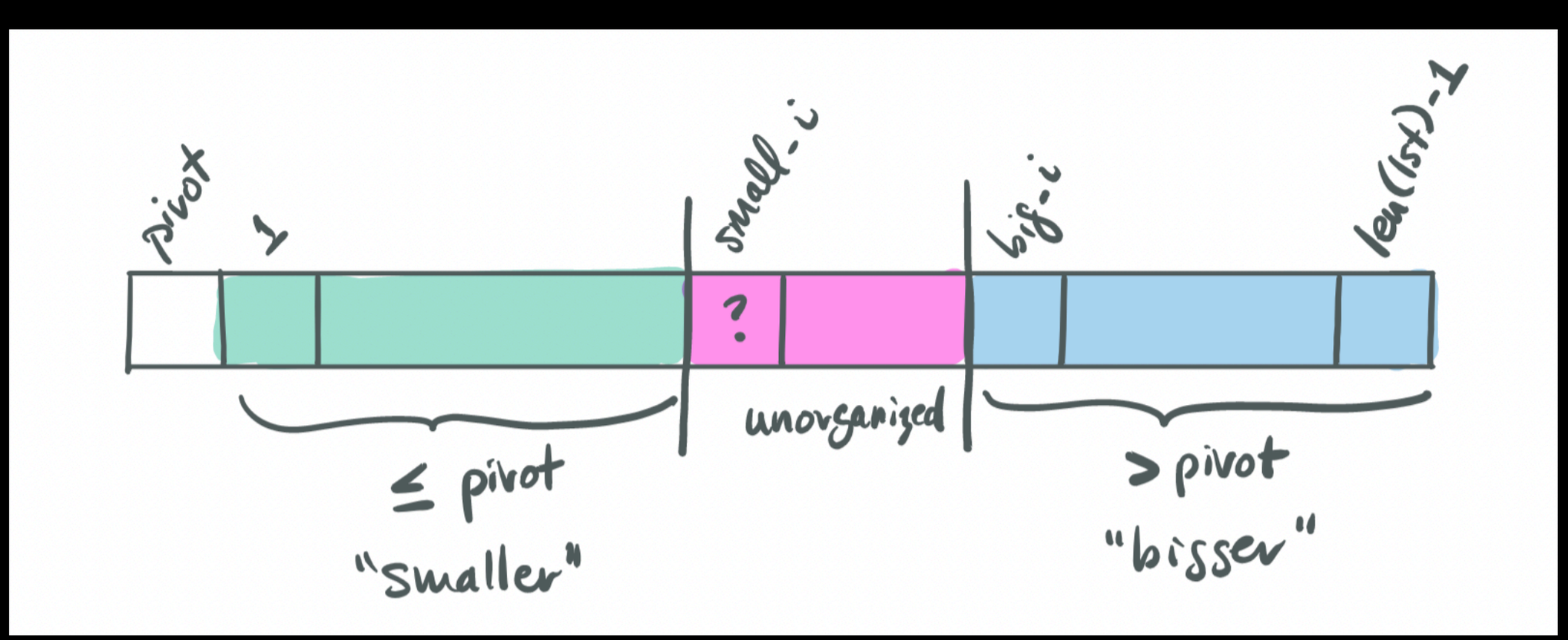
Loop invariants

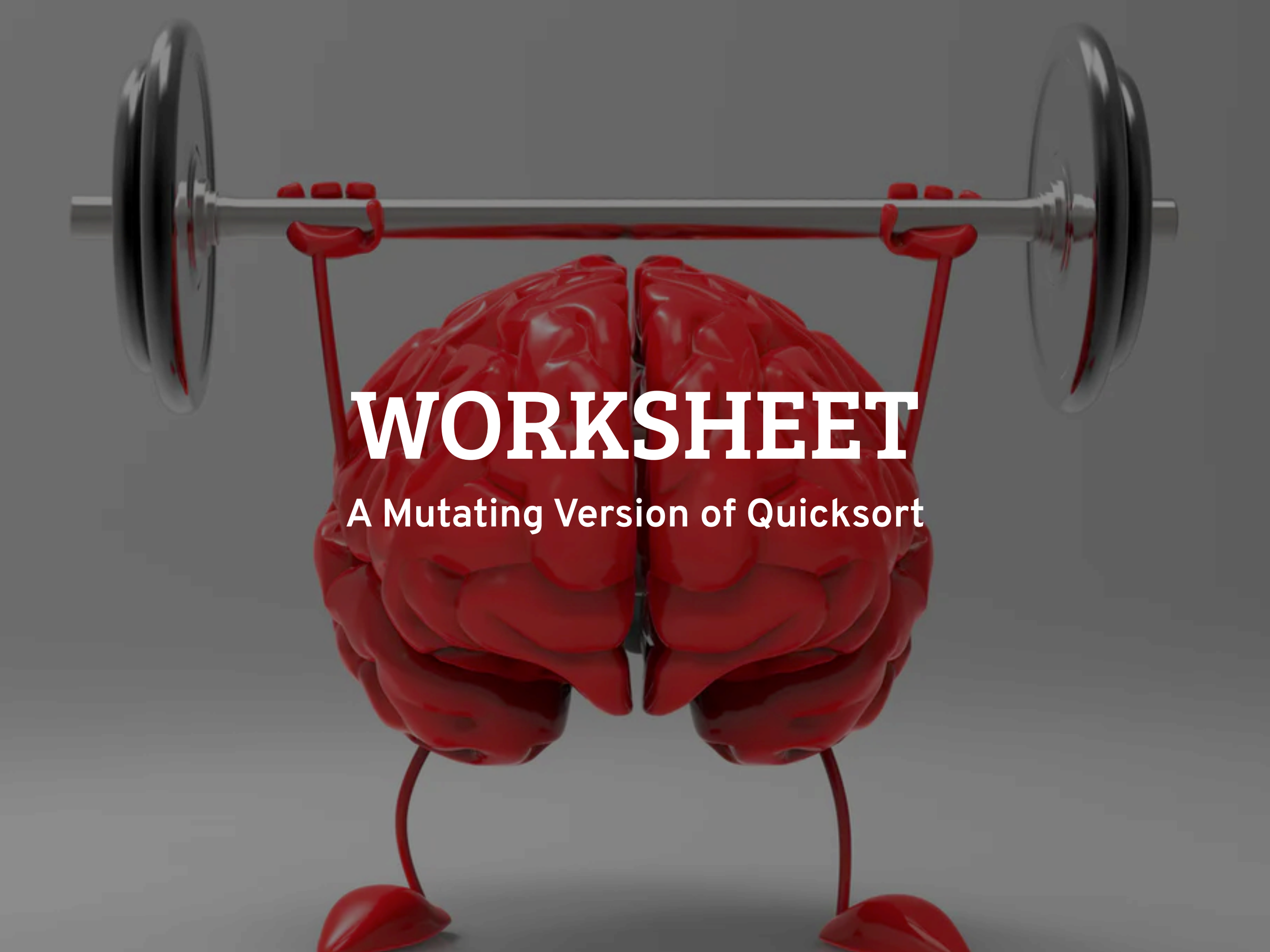
We will keep these true at the beginning and end of every loop iteration:

`lst[1:small_i]` contains the elements that are known to be `<= lst[0]` (the "smaller partition")

`lst[big_i:]` contains the elements that are known to be `> lst[0]` (the "bigger partition")

`lst[small_i:big_i]` contains the elements that have not yet been compared to the pivot (the "unsorted section")





WORKSHEET

A Mutating Version of Quicksort

Quicksort

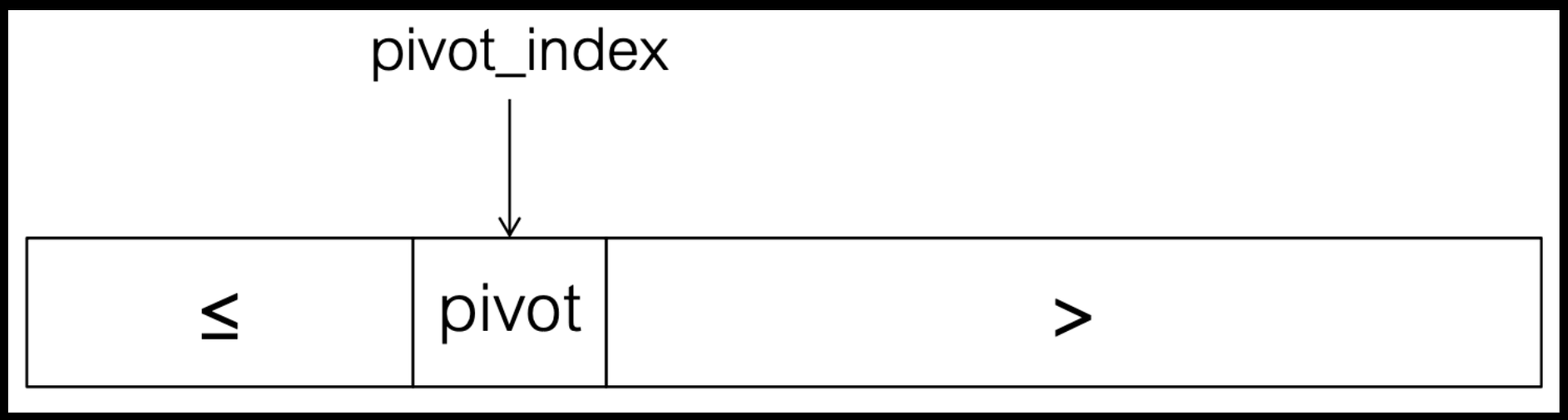
Simulating slicing with indexes

We often want to operate on just part of a list:

f(lst[start:end])

Rather than create a new list object, we pass in the indexes:

f(lst, start, end)



Quicksort

Simulating slicing with indexes

We often want to operate on just part of a list:

f(lst[start:end])

Rather than create a new list object, we pass in the indexes:

f(lst, start, end)

```
_in_place_partition(lst) →
_in_place_partition(lst, start, end)


quicksort(lst) →
_in_place_quicksort(lst, start, end)
```