# Recursive Sorting

CSC148, INTRODUCTION TO COMPUTER SCIENCE

DIANE HORTON, SADIA SHARMIN, & JONATHAN CALVER

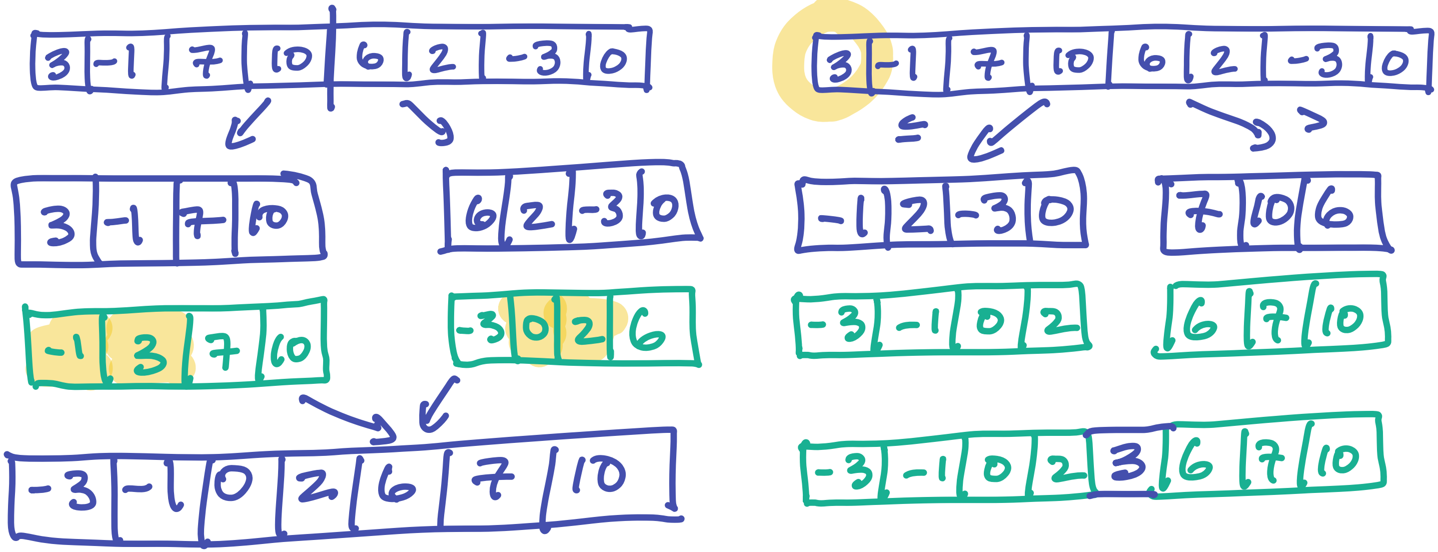
# Splitting lists, divide-and-conquer

Mergesort + quicksort both take this strategy:

1. **Divide** the input list into smaller lists.
2. **Recurse** on each smaller list.
3. **Combine** the results of each recursive call.

But how they divide + re-combine differs

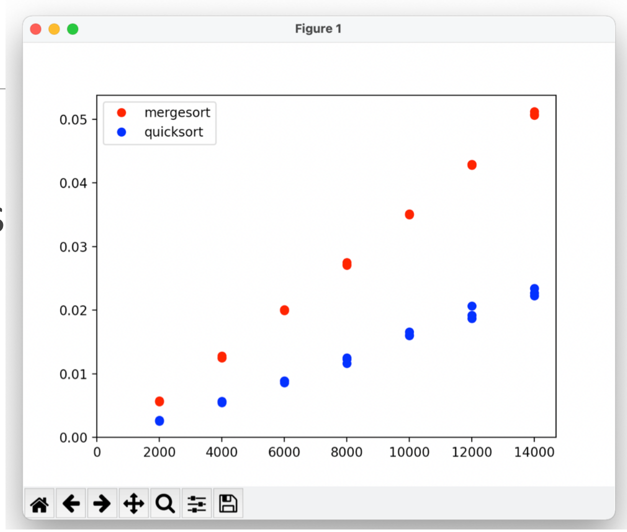
# mergesort and quicksort



# Running time demo

mergesort, quicksort, and insertion s

After doing the on-paper analysis,
We ran a demo of these
and generated this
graph:



# Worksheet!

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

Two key parts:

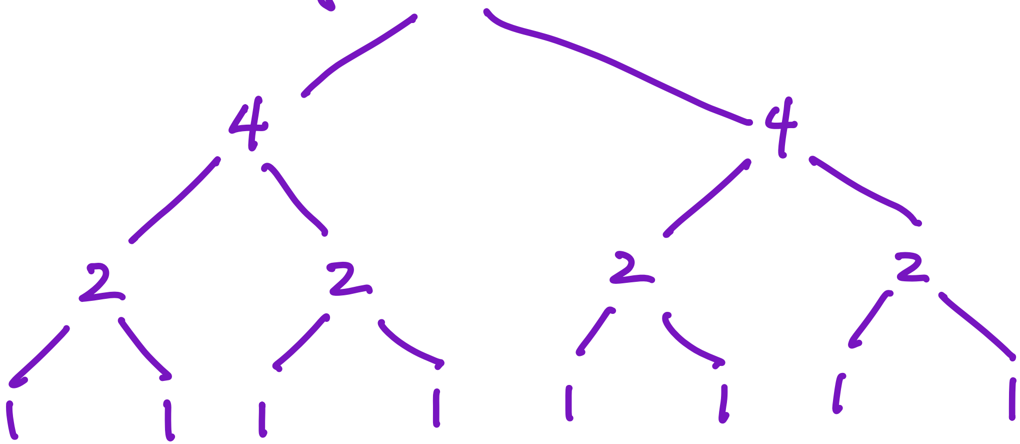
- how long do the *non-recursive parts* take?
- what is the structure of the recursive calls?

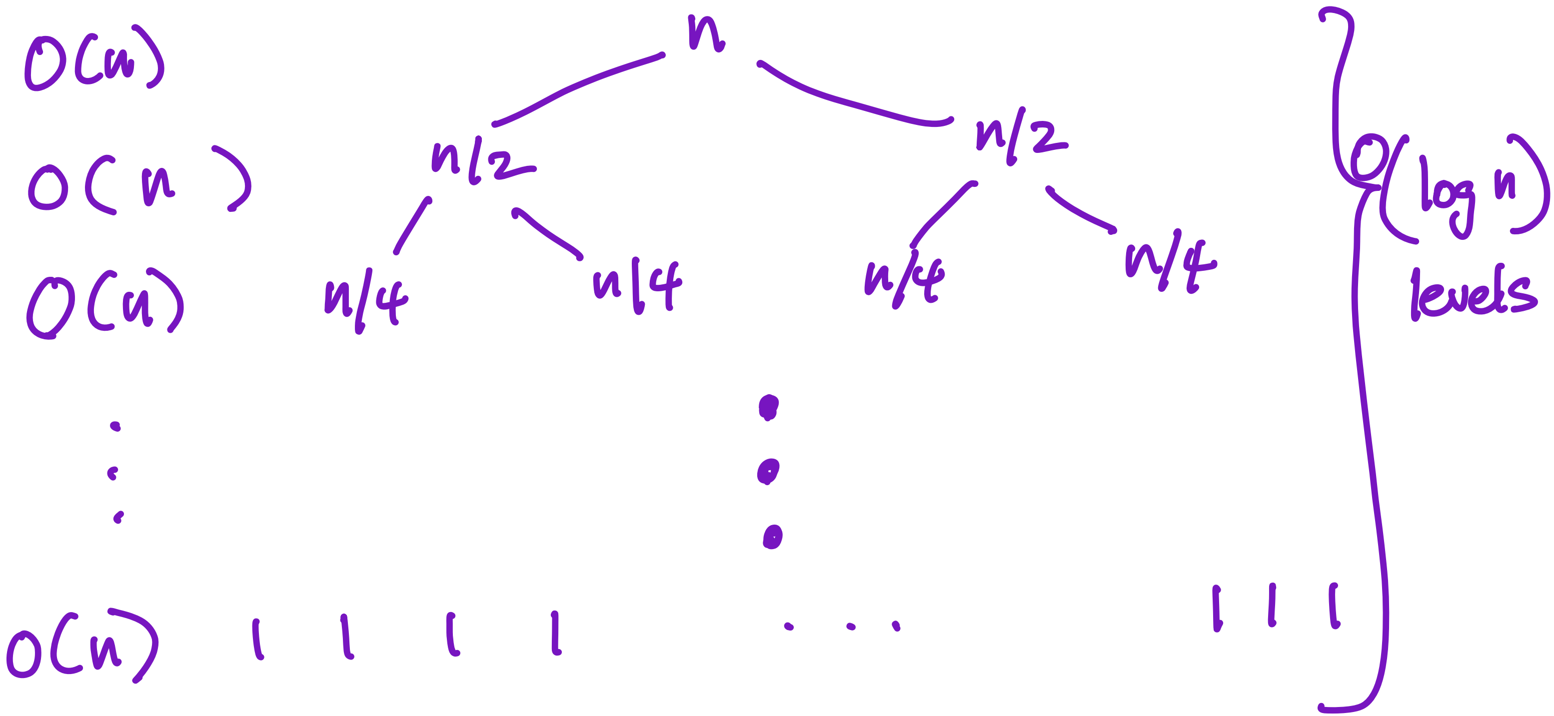
```python
def mergesort(lst):
    if len(lst) < 2:
        return lst[:]
    else:
        mid = len(lst) // 2
        left = lst[:mid]
        right = lst[mid:]

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

        return _merge(left_sorted, right_sorted)
```

See worksheet
Q3 + Q5

Thinking through the recursive calls for mergesort, with a specific n.

call ms on a list of length 8






Grand Total: mergesort is $O(n \log n)$

The picture is the same regardless of list contents

Mergesort doesn't look at the list to decide what to do.
It does the same thing for <u>any</u> list of length n.

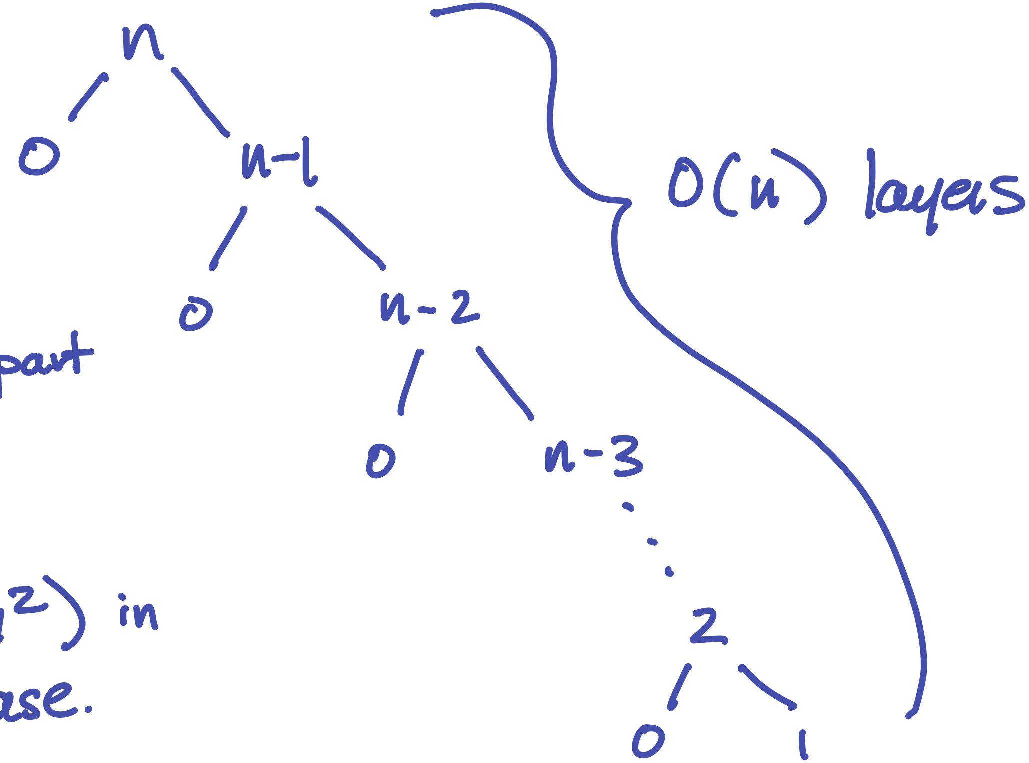
$\therefore$ worst case = best case = average case for mergesort

See worksheet

Q1, Q2 + Q6

```
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
```

worst case for Quicksort:



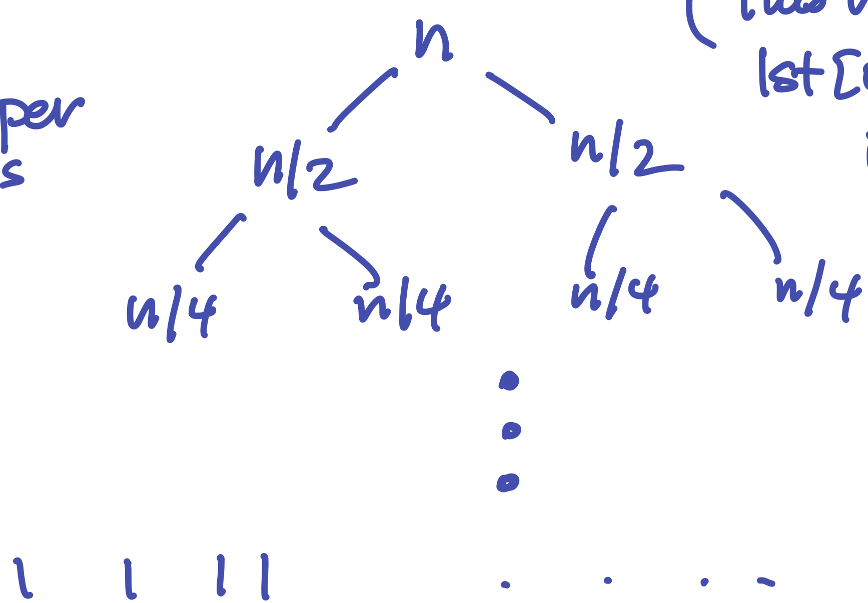

work per layer for the non-recursive part is $O(n)$.

$\therefore$ QS is $O(n^2)$ in the worst case.

Could it be better? Yes! We could get a complete + balanced tree of calls.

(This happens if, every time, lst[0], the pivot, is the median.)

Work per level is $O(n)$




Now we have only $O(\log n)$ levels.

∴ In this case, total work is $O(n \log n)$.

# 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))**. $\equiv$ mergesort

a lot?

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(n²)**. worst than mergesort

rare?

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

??? csc263

On average, QS is $O(n \log n)$

Furthermore, the constants for QS are much smaller than MS.

QS is actually better, on average, than MS.
(See the graph above.)

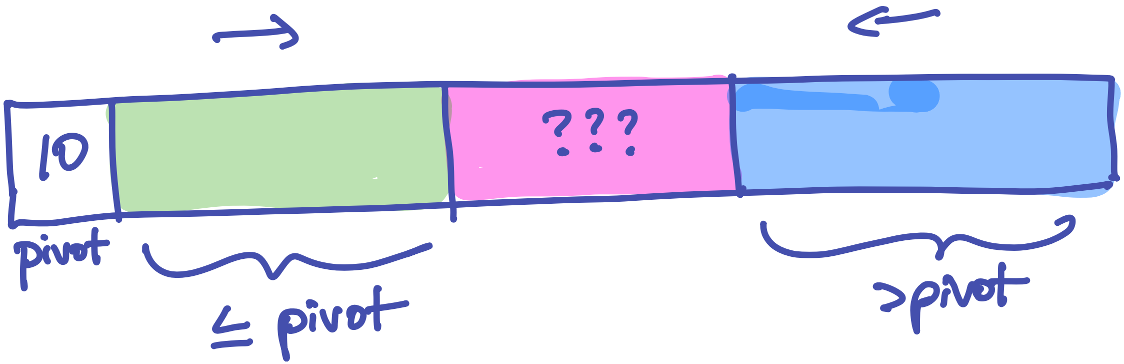
# In-place quicksort

MUTATING THE INPUT LIST IN A SPACE-EFFICIENT WAY.

Exaple: 10 7 20 30 3 6

10 7 20 30 3 6


# The key helper: in-place partition



```python
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
```

# 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)`

# Simulating slicing with indexes

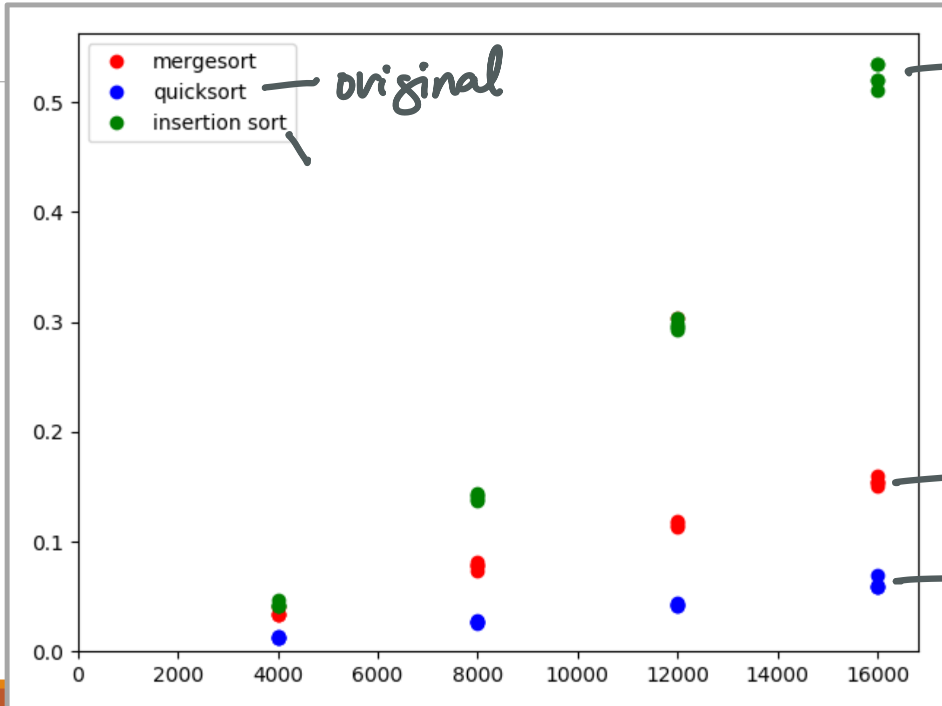
```
_in_place_partition(lst) →
_in_place_partition(lst, start, end)

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

# Lessons in efficiency

A CASE STUDY IN COMPARING SORTING ALGORITHMS

Shuffled input

1. Big-Oh describes behaviour as input size grows




original

insertion sort

mergesort

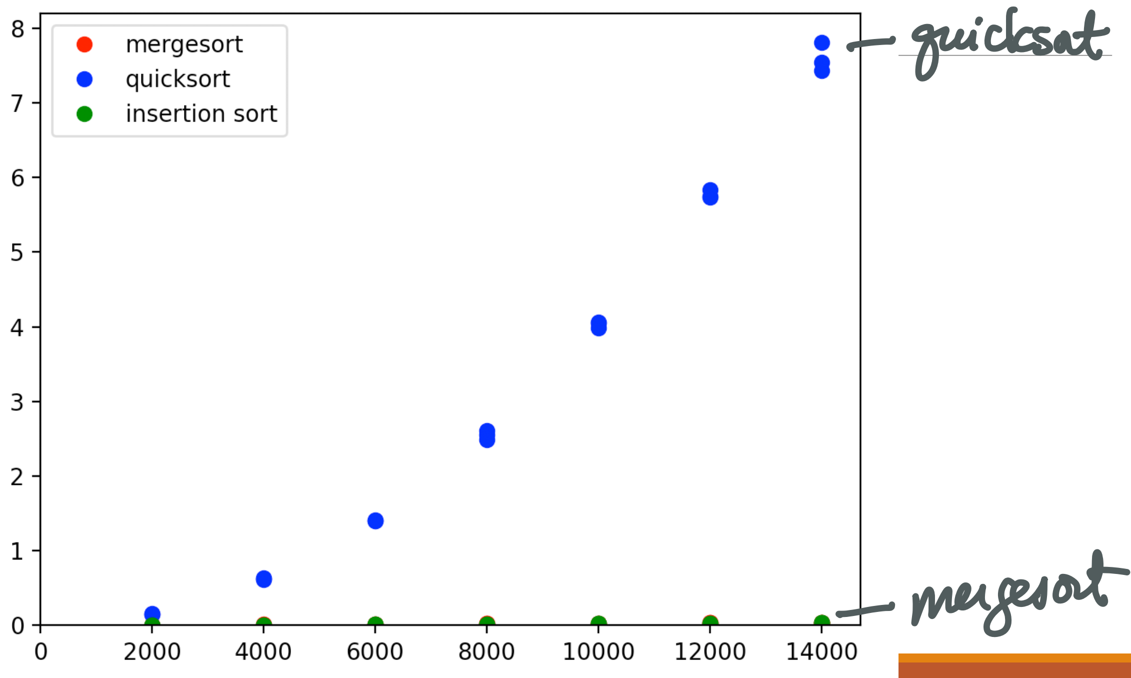
quicksort

time to sort

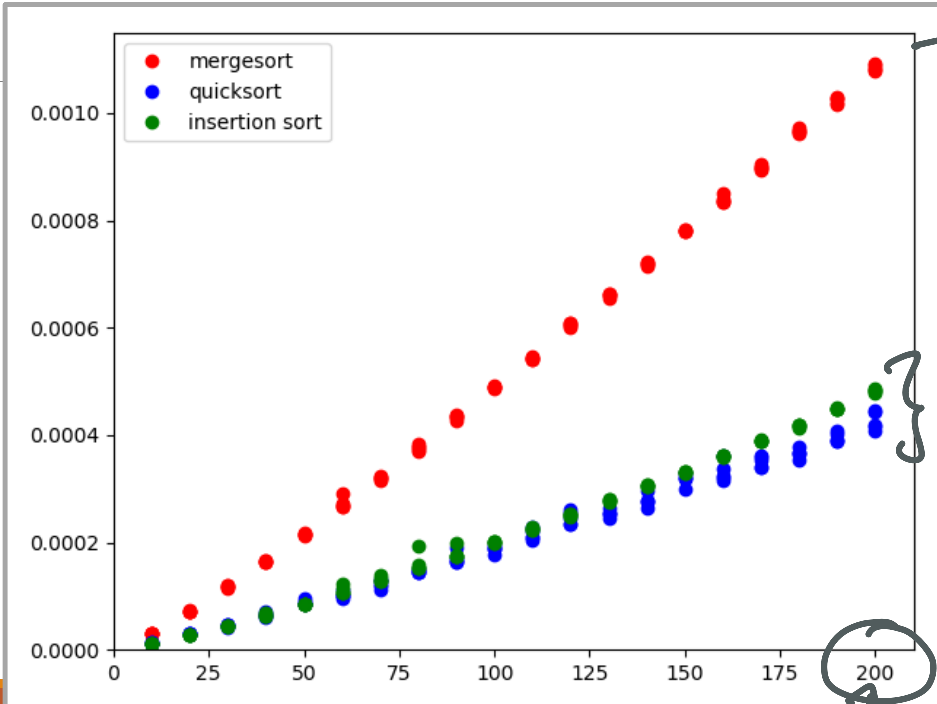
bigger lists

# Sorted Input

2. An algorithm can be "good on average" and "bad in the worst case"



3. Big-Oh is *not* good at predicting behaviour on small inputs.

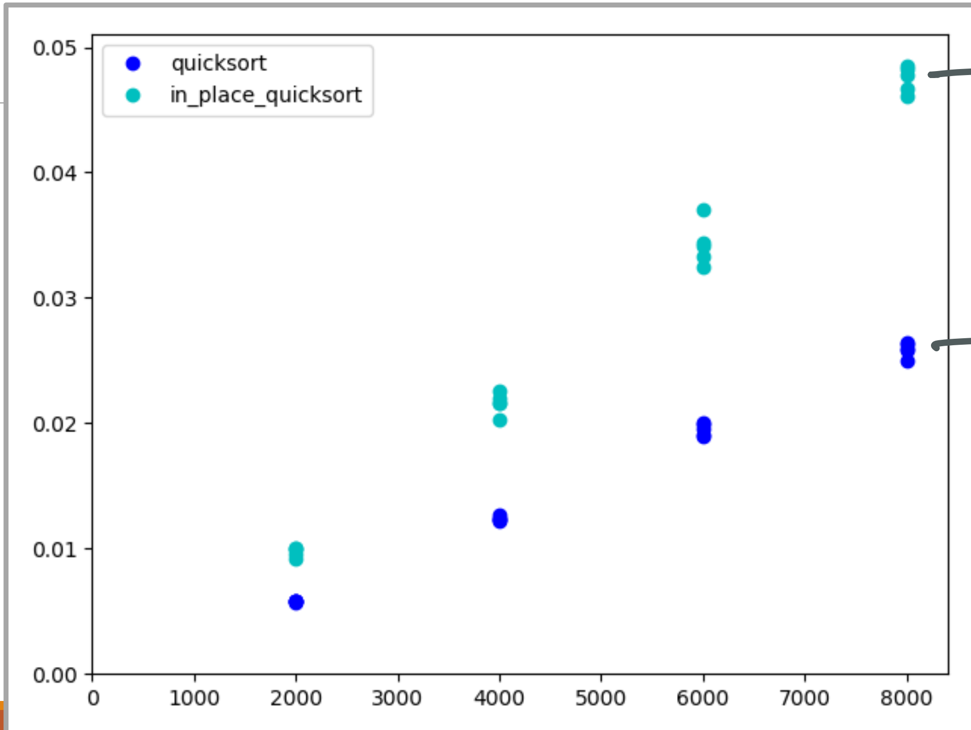



mergesort

ins + quicksort

How did we do – with all that effort to work inplace ?

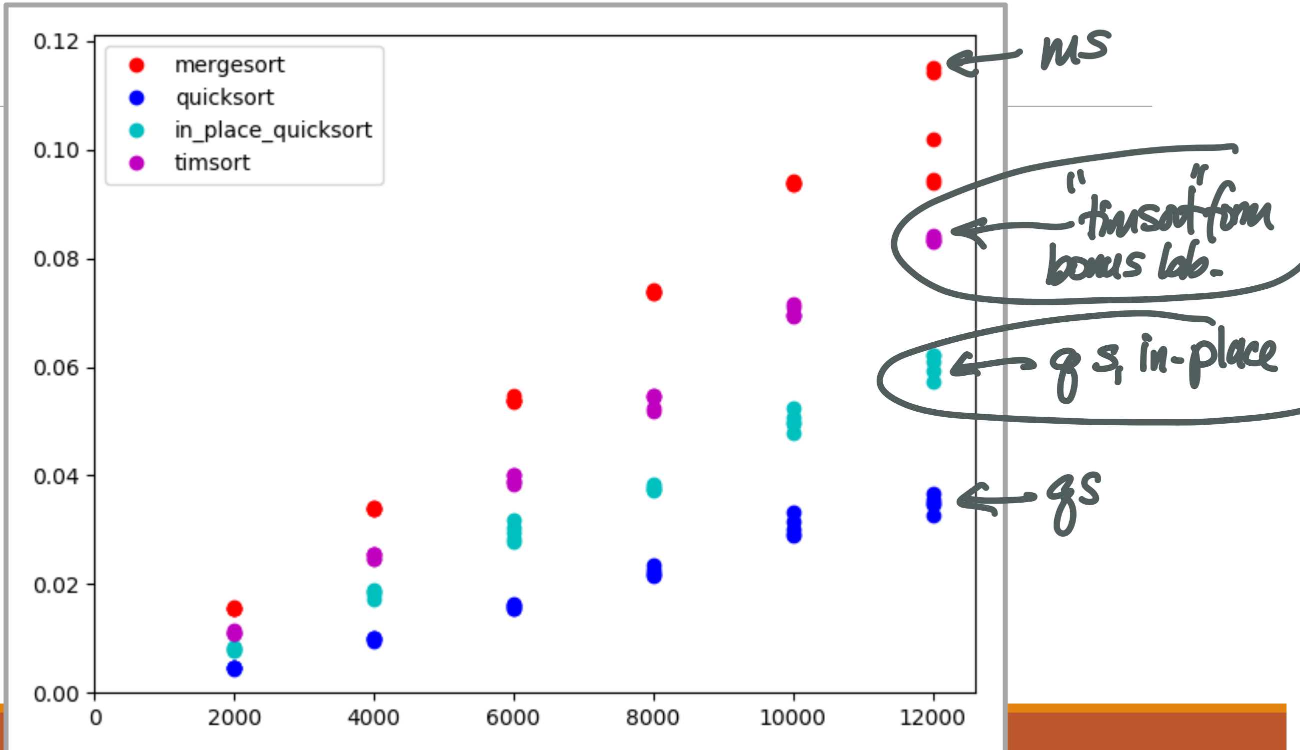
4. Saving space doesn't always mean saving time!



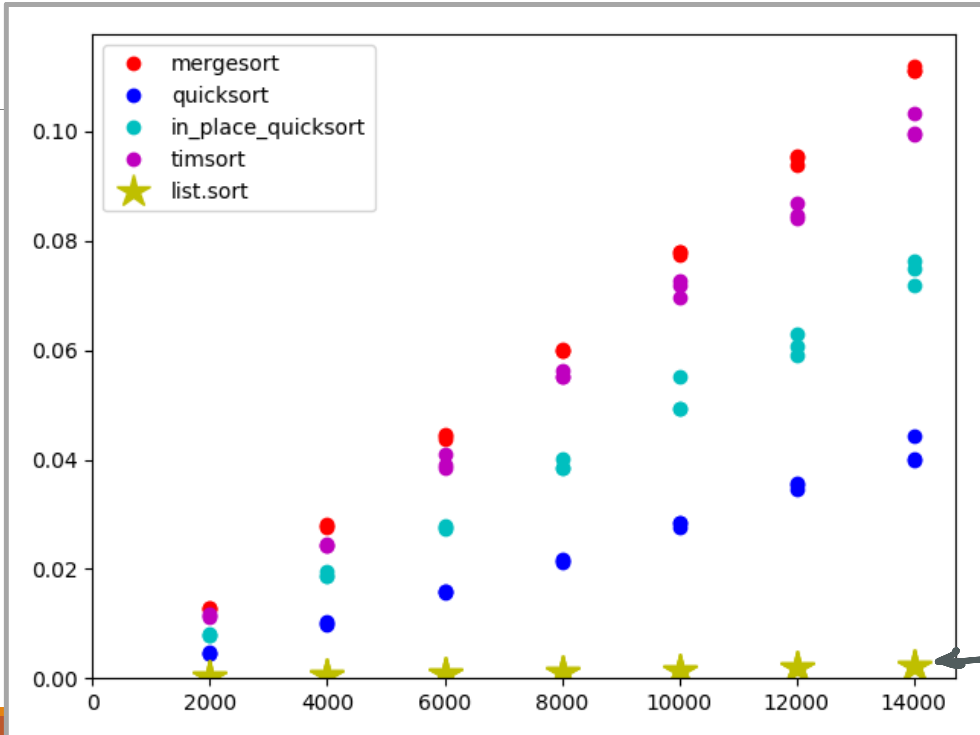

in-place qs

qs

# 5. Hard work doesn't always mean saving time, either!



# 6. But sometimes hard work pays off.*




lst.sort!