CSC 148: Introduction to Computer Science

Week 12

Efficiency of algorithms (continued)

University of Toronto Mississauga,

Department of Mathematical and Computational Sciences

Recall

- Last week, before the long weekend, we looked at sorting algorithms.

- In particular, *in-place Quicksort*

- The goal was to compare the efficiency of various algorithms.

Your Task

- We've seen quite a few sorts this semester!
  - You're implementing one more – Timsort – in lab this week.

- First, review the timing code we're about to show you.
  - *How do you make it measure the* worst case?
  - *Any concerns with how we're timing these sorts?*

- Second, plot what you think we will see when we run this code.
  - *How will each algorithm grow?*
  - *Which will be fastest? Slowest?*

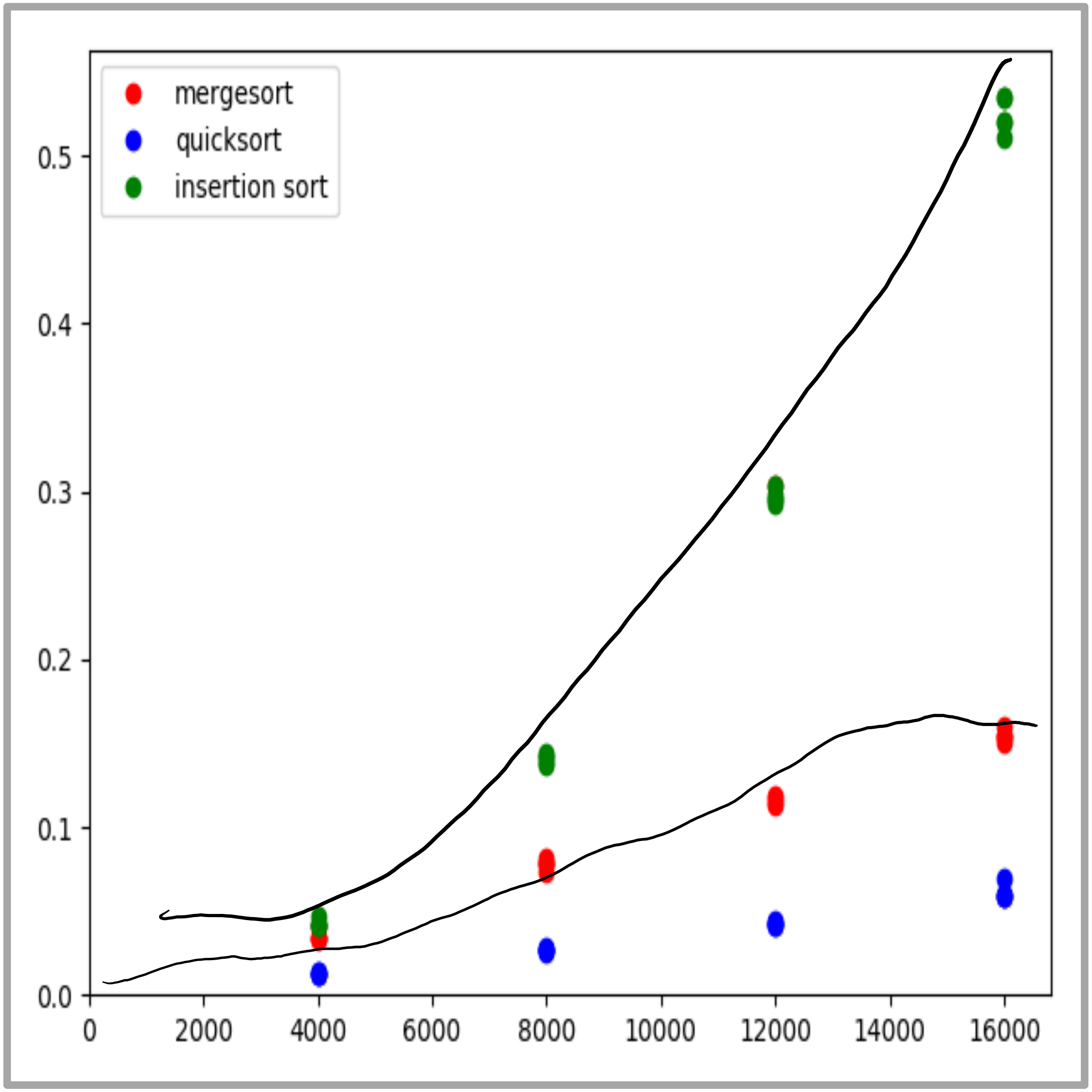
Lessons in Efficiency

A case study in comparing sorting algorithms

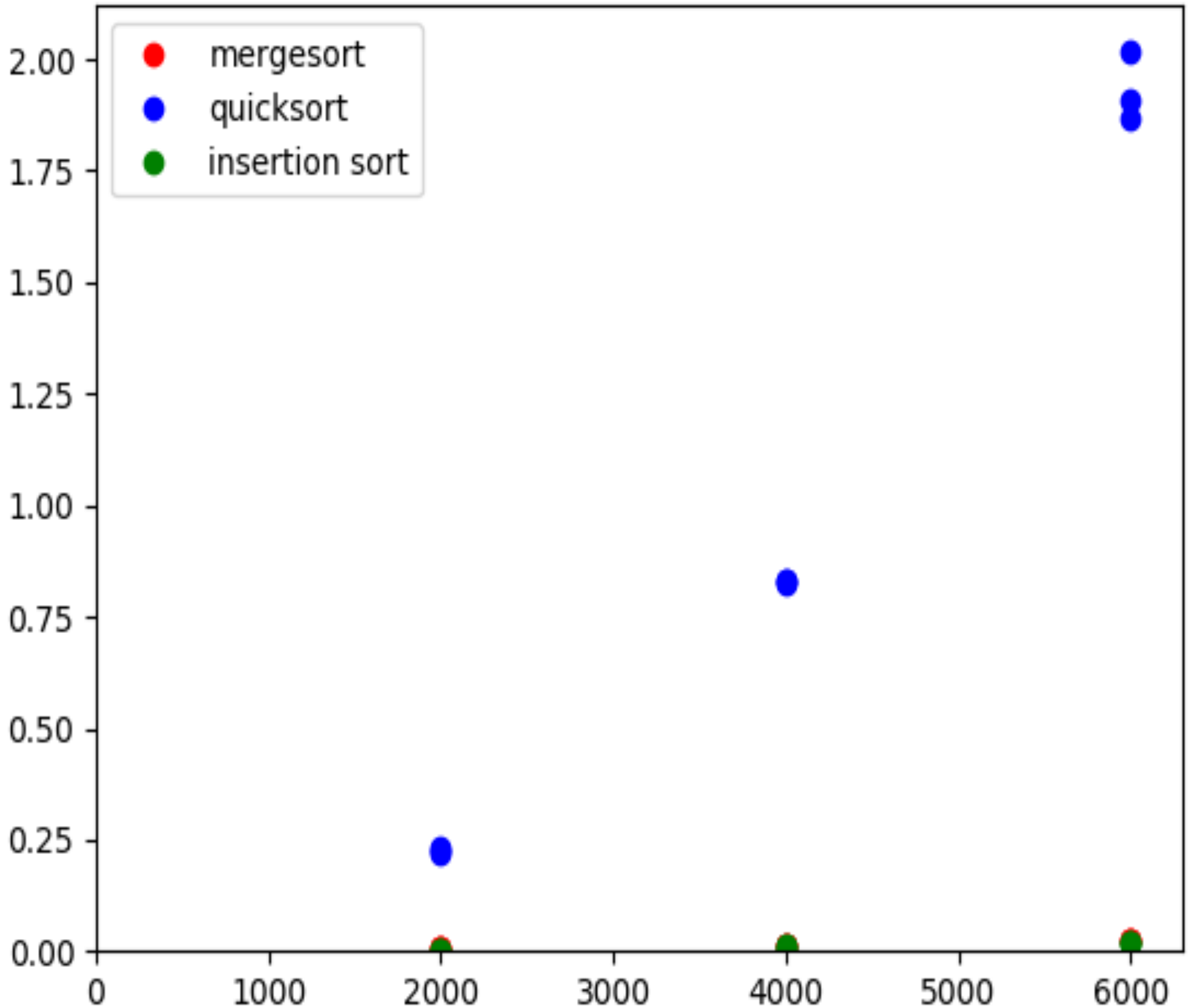
University of Toronto Mississauga,

Department of Mathematical and Computational Sciences

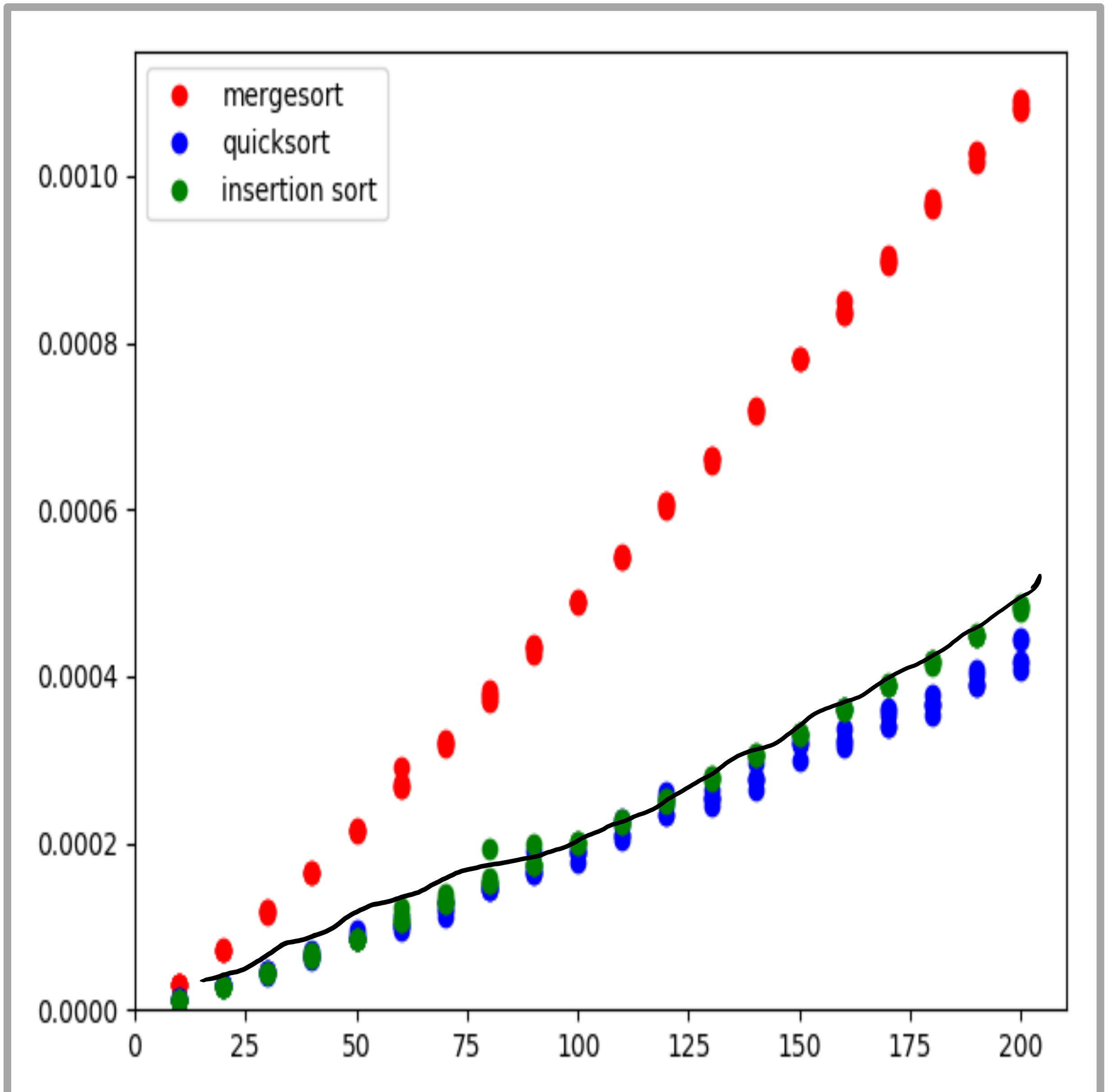
1. Big-Oh describes behaviour as input size grows



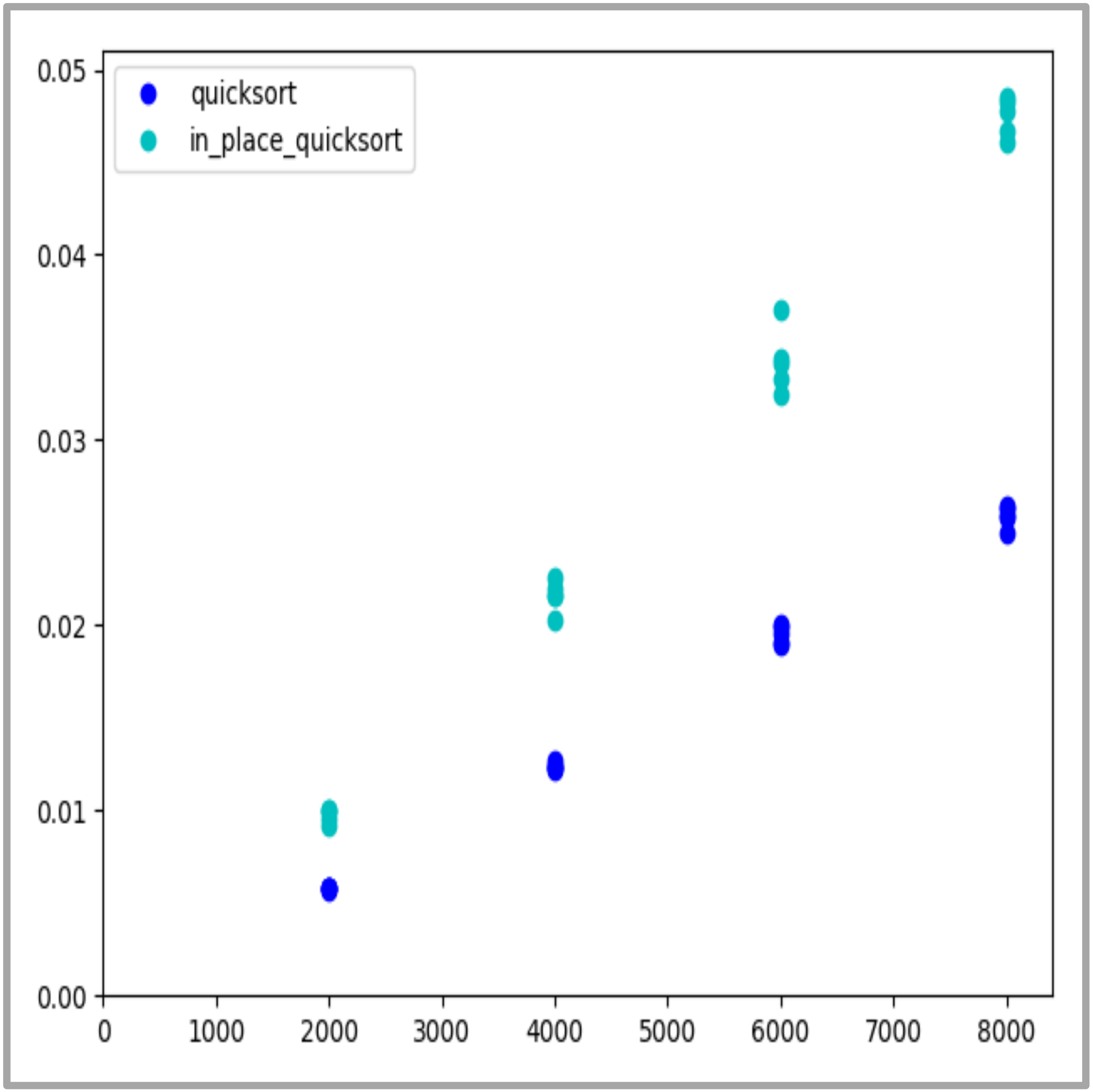
2. An algorithm can be "good on average" and "bad in the worst case"



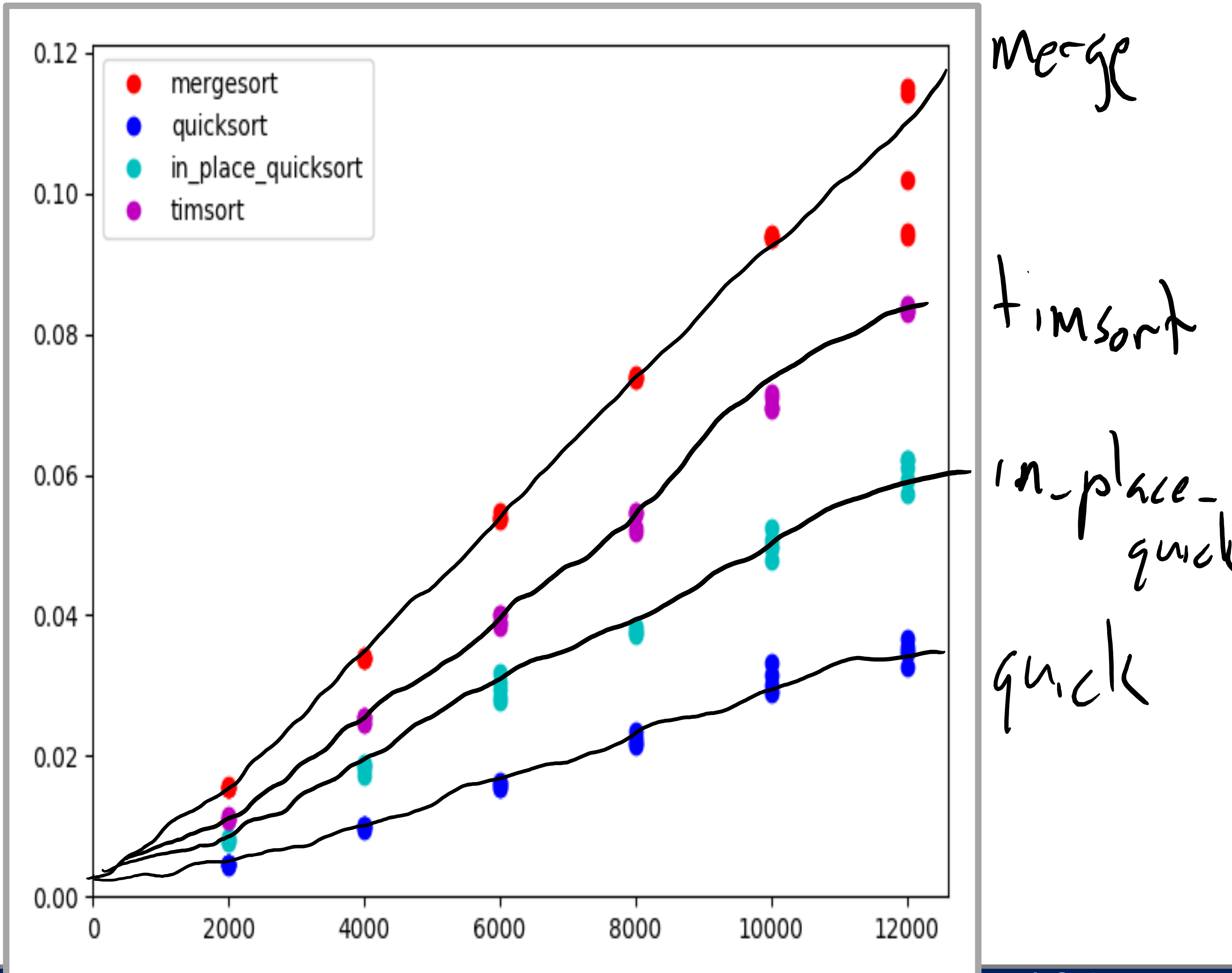
3. Big-Oh is **not** good at predicting behaviour on small inputs



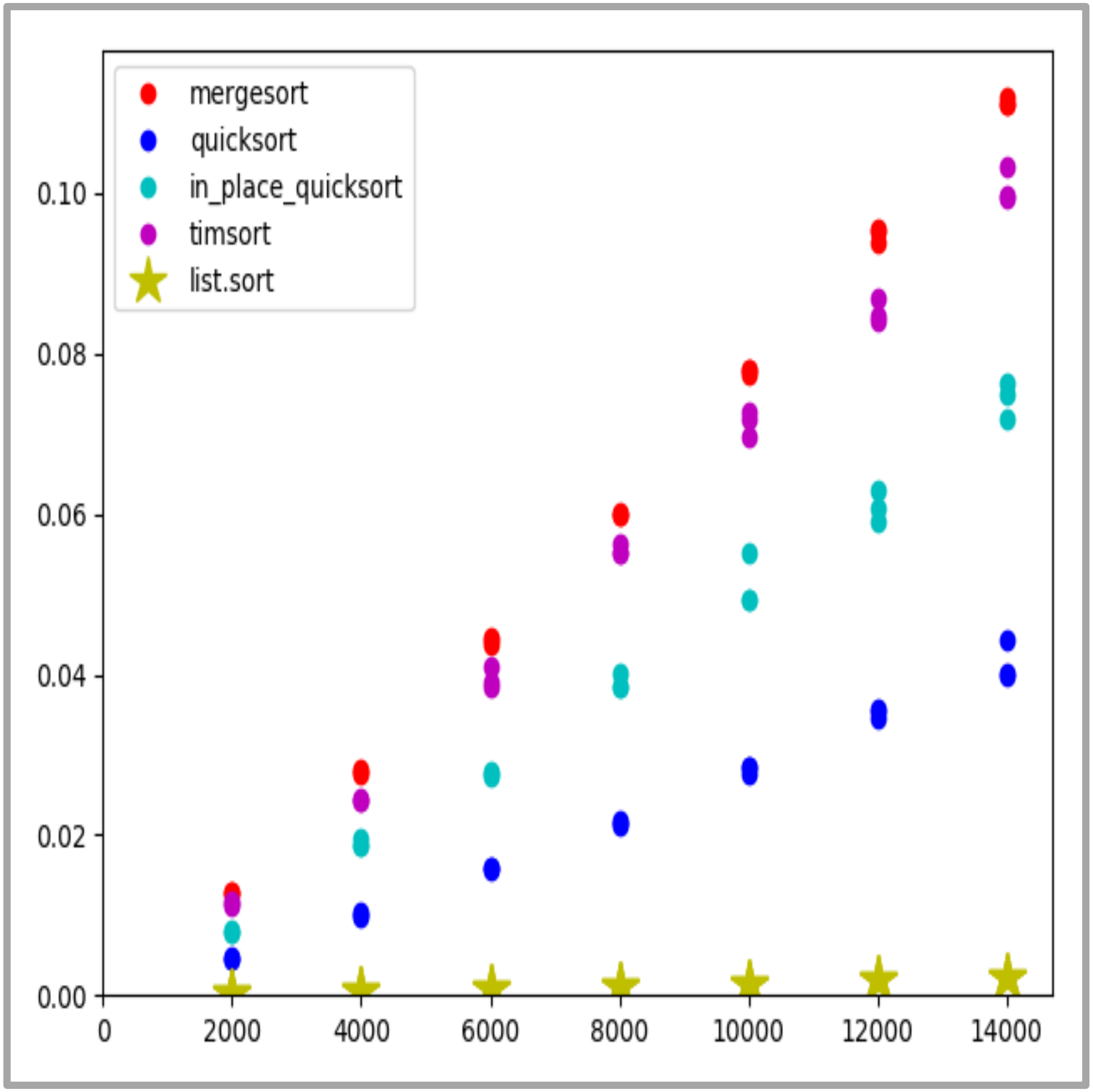
4. Saving space doesn't always mean saving time!



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



6. But sometimes hard work pays off.*



Side-note: Trusting recursion, expectation on base case

University of Toronto Mississauga,

Department of Mathematical and Computational Sciences

Recall Partial Tracing

- In partial tracing, we assume that each recursive call works correctly

  - `sum_list(sublist)    # nested lists`
  - `subtree.insert(item)`
  - `left.__contains__(item)`
  - `quicksort(smaller)`

Believing in Recursion

- How can we just *assume* the recursive call works? What it if doesn't?

- Let's examine the justification for our confidence.

Reasoning about Correctness

- Let P(n) =

  "*For any nested list obj of depth n,*

  sum_list(obj) returns, and

  returns the sum of the numbers in <obj>."

- We want to know that $\forall n \geq 0$, P(n).

Tracing from the Smallest Case Up

- For sum_list, we traced the function and concluded that:
  - P(0) is true.
  - P(1) is true as long as P(0) is true.
  - P(2) is true as long as P(0) and P(1) are true.

    . . . and we could have continued on to show that . . .
  - P(3) is true as long as P(0), P(1), and P(2) are true.
  - And P(4), P(5), ...
- Crucially, we did not trace the "as long as" parts.
- (We had already convinced ourselves of them.)

Reasoning More Formally

- If we show these two things:

  P(0) is true.

  $\forall k \geq 0$,

  P(k+1) is true as long as P(0), … P(k) are all true.

- … we can conclude that:

  $\forall n \geq 0$, P(n).

Reasoning More Formally

- If we show these two things:

  P(0) is true.

  $\forall k \geq 0$,

  P(k+1) is true as long as P(0), … P(k) are all true.

- … we can conclude that:

  $\forall n \geq 0$, P(n).

Yes, this is induction from MAT102.

Base Case(s)

- There must be at least one base case.
- There may be more than one.
- Any call to a recursive method must ultimately reach a base case.
  - Otherwise, we have “infinite” recursion.

Our Assumption has a Condition!

- In the recursive step, assume that each recursive call works correctly
- *… as long as the input(s) to the recursive call are smaller than the original(s)*

Danger: Infinite Recursion!

- Infinite recursion occurs when a series of recursive calls never reaches a base case.
- Every call makes a new recursive call, and this continues forever!

```
class Tree:
  def __len__(self) -> int:
    if self.is_empty():
      return 0
    else:
      return self.__len__()
```

Every call creates a new stack frame!

Danger: Infinite Recursion

- Because each function call requires a new stack frame, infinite recursion can lead to an "infinite" amount of computer memory consumption

- Python protects against this by limiting the total number of stack frames it allows
  - RecursionError:

```
RecursionError: maximum recursion depth exceeded
```

Setting the Recursion Limit

- Can be done via the sys module

```
import sys
sys.setrecursionlimit(5000)
```

- However, just because you can doesn't mean you should!
  - Chances are your code is the problem, not the default recursion limit ...

End of side-note

- On Wednesday: Back to efficiency conditions …
- Big-Oh notation and Big-Theta