

Recursive Sorting

CSC148, INTRODUCTION TO COMPUTER SCIENCE

DIANE HORTON, SADIA SHARMIN, & JONATHAN CALVER



Splitting lists, divide-and-conquer

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

mergesort and quicksort



Running time demo

mergesort, quicksort, and insertion sort

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?


```
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)
```







```
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: 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 the two partitions are very unequal, and the running time is **$O(n^2)$** .

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. $O(n^2)$

In-place quicksort


MUTATING THE INPUT LIST IN A SPACE-EFFICIENT WAY.



The key helper: in-place partition



```
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)`

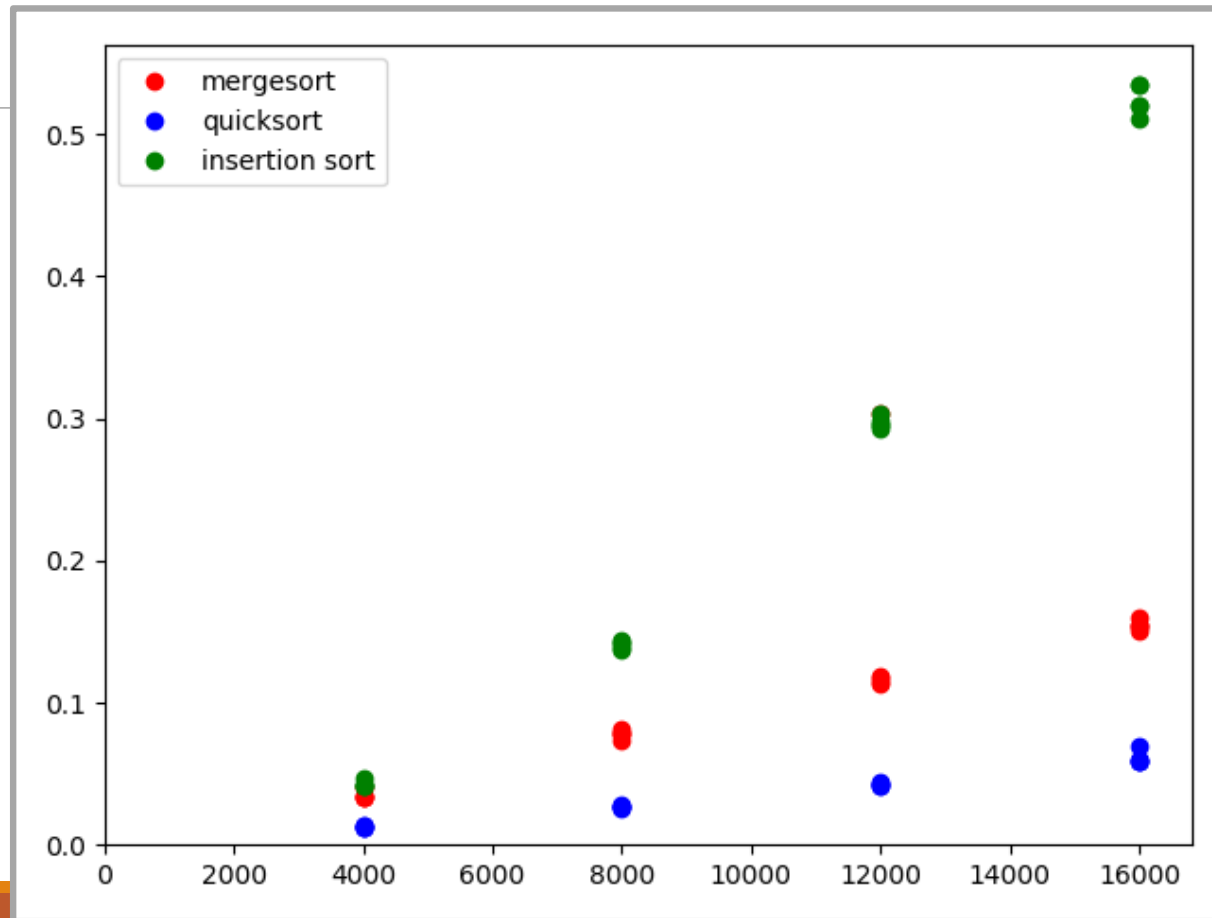


Lessons in efficiency

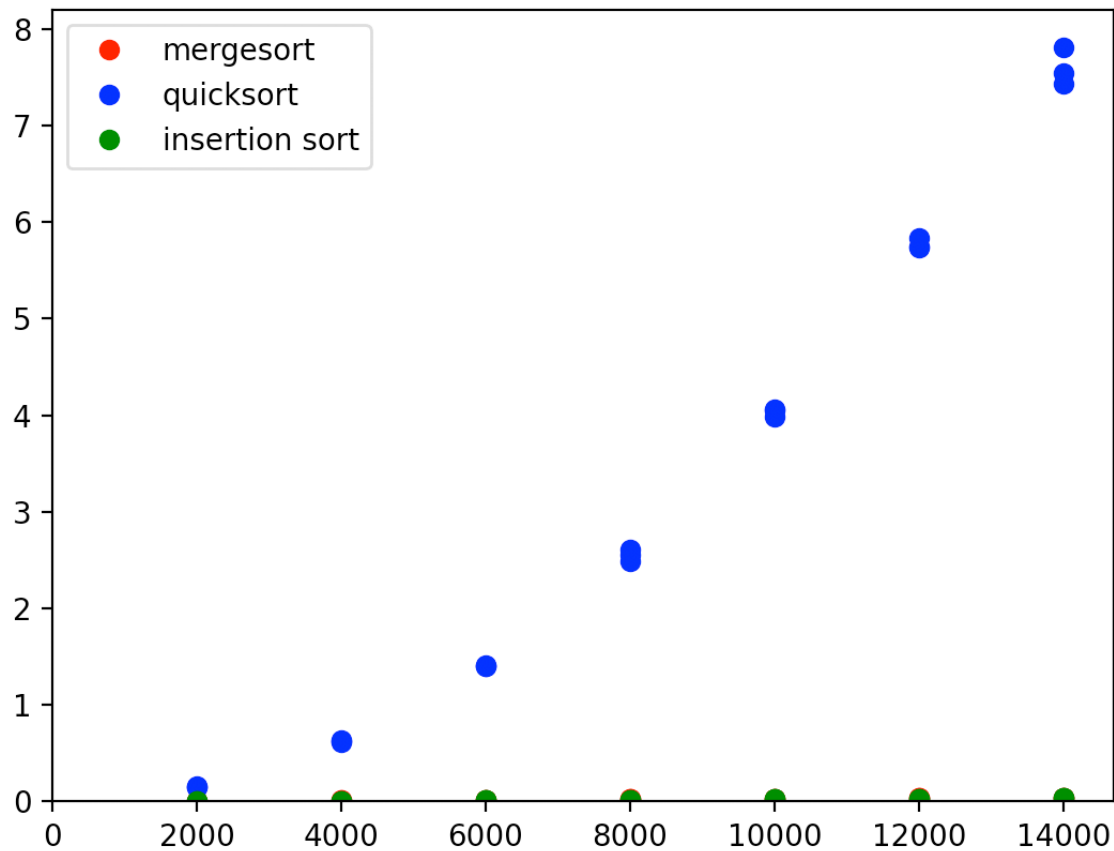
A CASE STUDY IN COMPARING SORTING ALGORITHMS



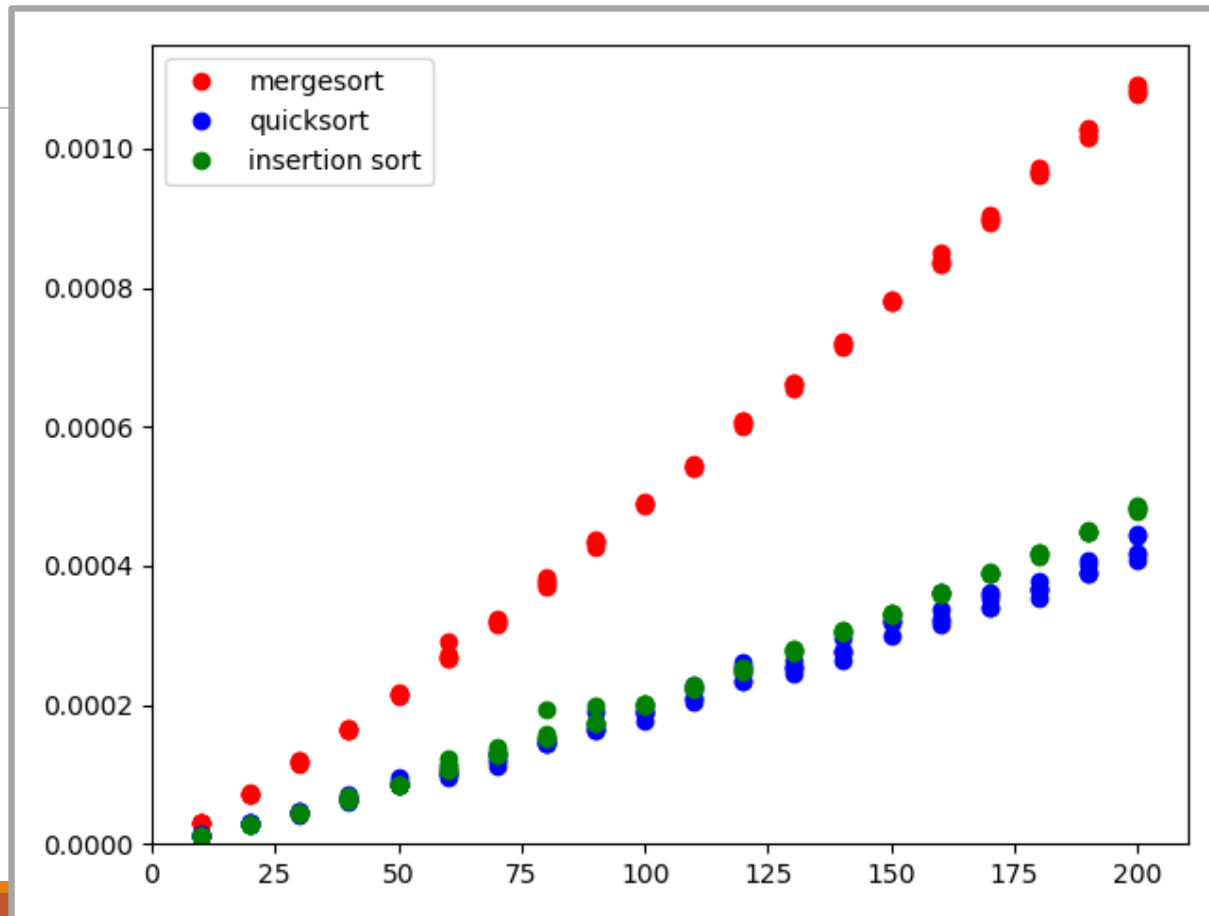
1. Big-Oh describes behaviour as input size grows



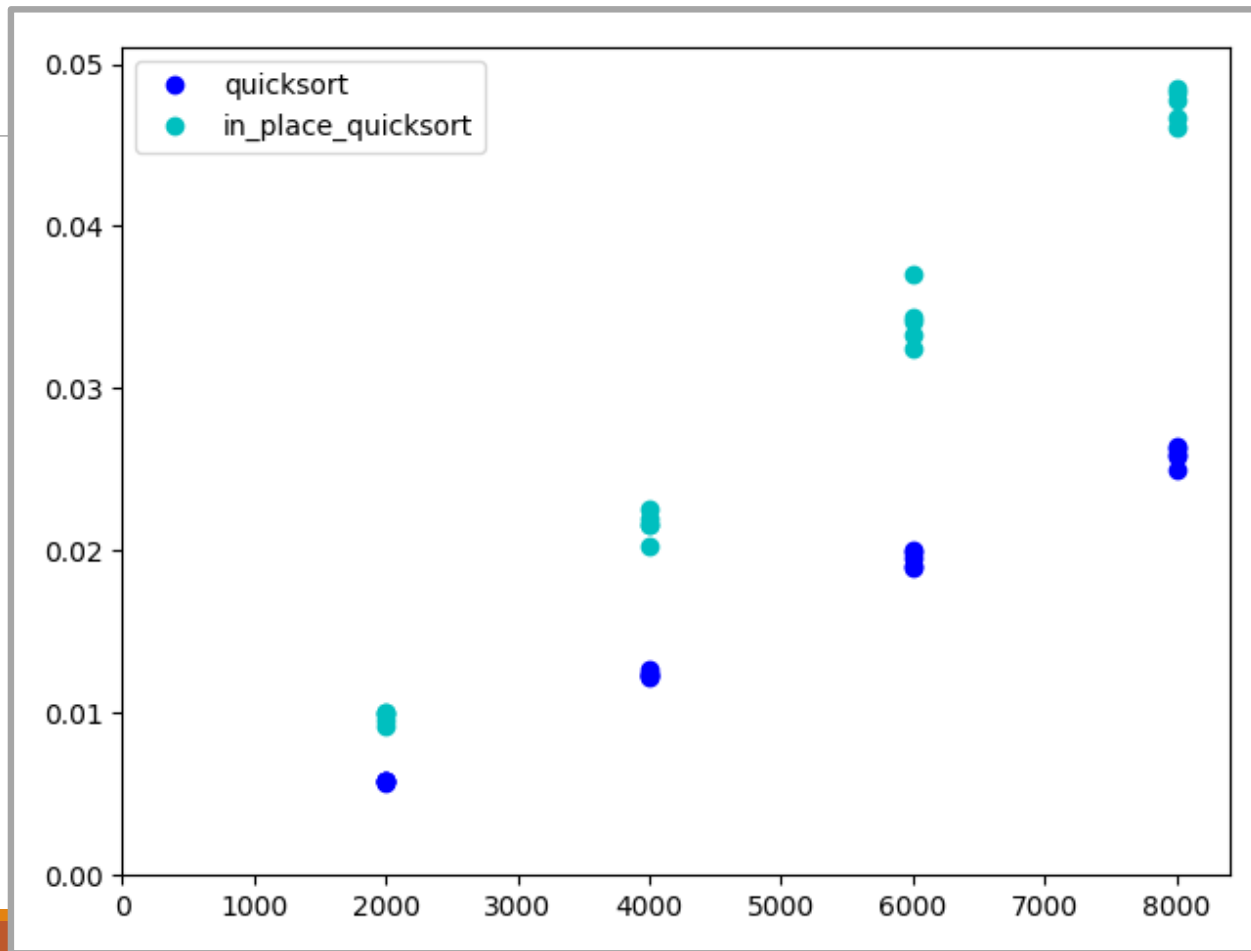
2. An algorithm can be “good on average” and “bad in the worst case”



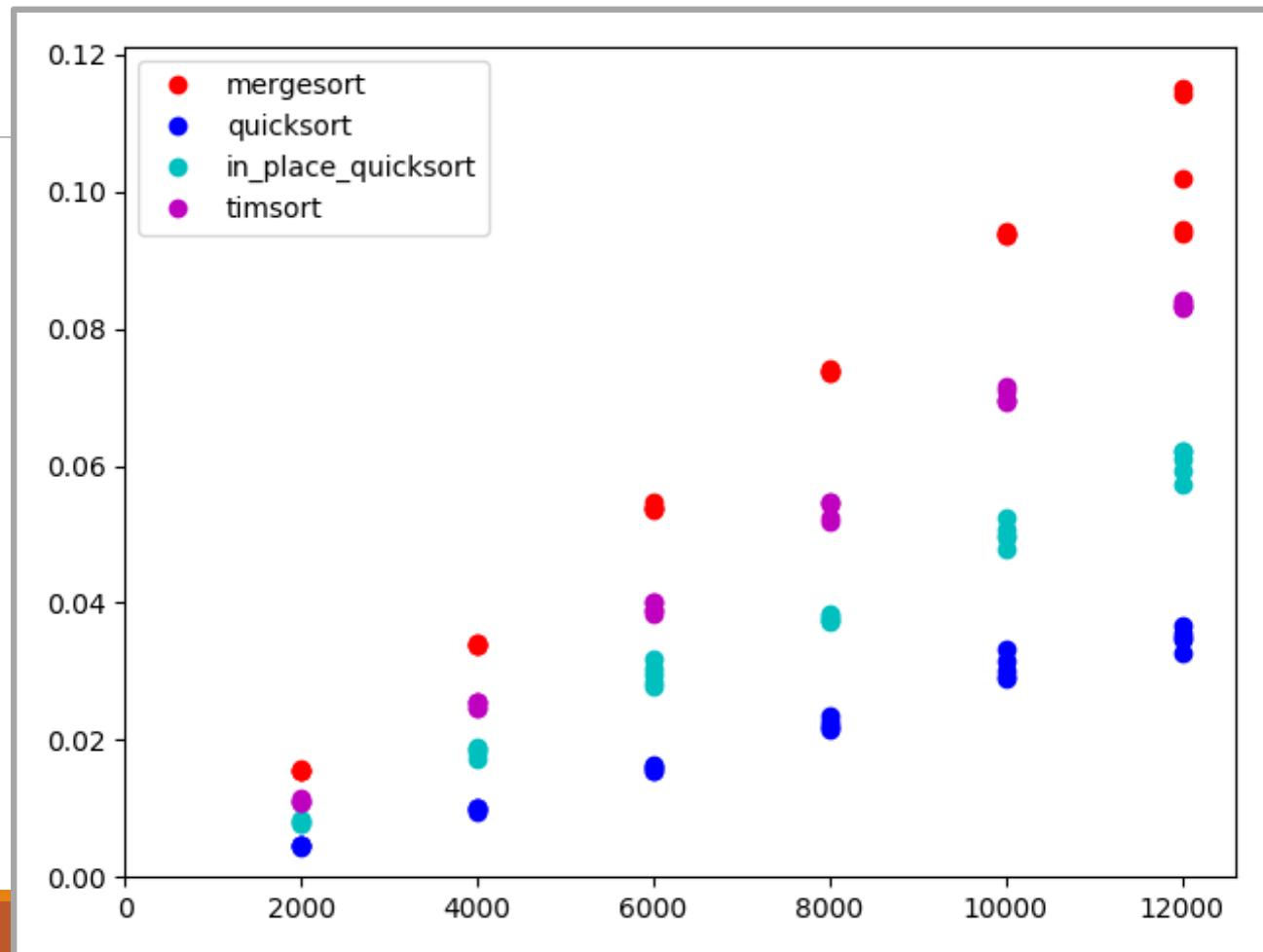
3. Big-Oh is *not* good at predicting behaviour on small inputs.



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.*

