05-01-2023

CS051A

INTRO TO COMPUTER SCIENCE WITH TOPICS IN AI

25: Big O

Alexandra Papoutsaki

she/her/hers

Lectures

Zilong Ye

he/him/his

Labs

Lecture 25: Big O

- Administrative
- Big O

This week (May 1st-May 6th)

- No mentor hours this week
- Lab is optional (come with questions)
- Course feedback forms on Wednesday and recap (come with questions)
- Nim tournament winner!
  - Kevin Park from section 2 :)

Next week (May 8th-May 13th)

- I will hold office hours on Monday May 8th to answer any last-minute questions about the final.
  - 11am-noon and 1-4pm (Edmunds 222).
- Final exam:
  - Wednesday, May 10, 9am-12pm (Section 1)
  - Wednesday, May 10 2-5pm (Section 2)
  - Edmunds 114 (our classroom)
  - Can bring **four** pages of notes
  - Comprehensive, will cover the entire semester

Lecture 25: Big O

- Administrative
- Big O

For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing.

Francis Sullivan

Examples of algorithms

- What is an algorithm?
  - A sequence of steps to solve a problem or accomplish a task.

Common problems to solve with algorithms

- sort a list of numbers
- find a route from one place to another (cars, packet routing, phone routing, ...)
- find the longest common substring between two strings
- add two numbers
- microchip wiring/design (VLSI)
- solving sudoku
- cryptography
- compression (file, audio, video)
- spell checking
- pagerank
- classify a web page
- ...

Main parts to algorithm analysis

- developing algorithms that work (correctness)
- analyzing/understanding the run-time/space usage with the goal of making them faster or more space efficient (efficiency)

Algorithm analysis

- What questions might we want to ask/measure about an algorithm?
  - does it finish?
  - is it right?
  - how hard is it to code?
  - how long does it take to run? How efficient is it?
  - how much space does it use?

Asymptotic analysis

- Key idea: how does the run-time (or space) of our algorithm grow as we increase the input size of the problem we are trying to solve?
- For example, if we double the input, what will happen to the run-time? Will it…
  - stay unchanged?
  - double?
  - triple?
  - quadruple?
  - …
- Asymptotic analysis relies on math to see how the algorithm scales in respect to input size. It does NOT provide an exact running time (or space usage).

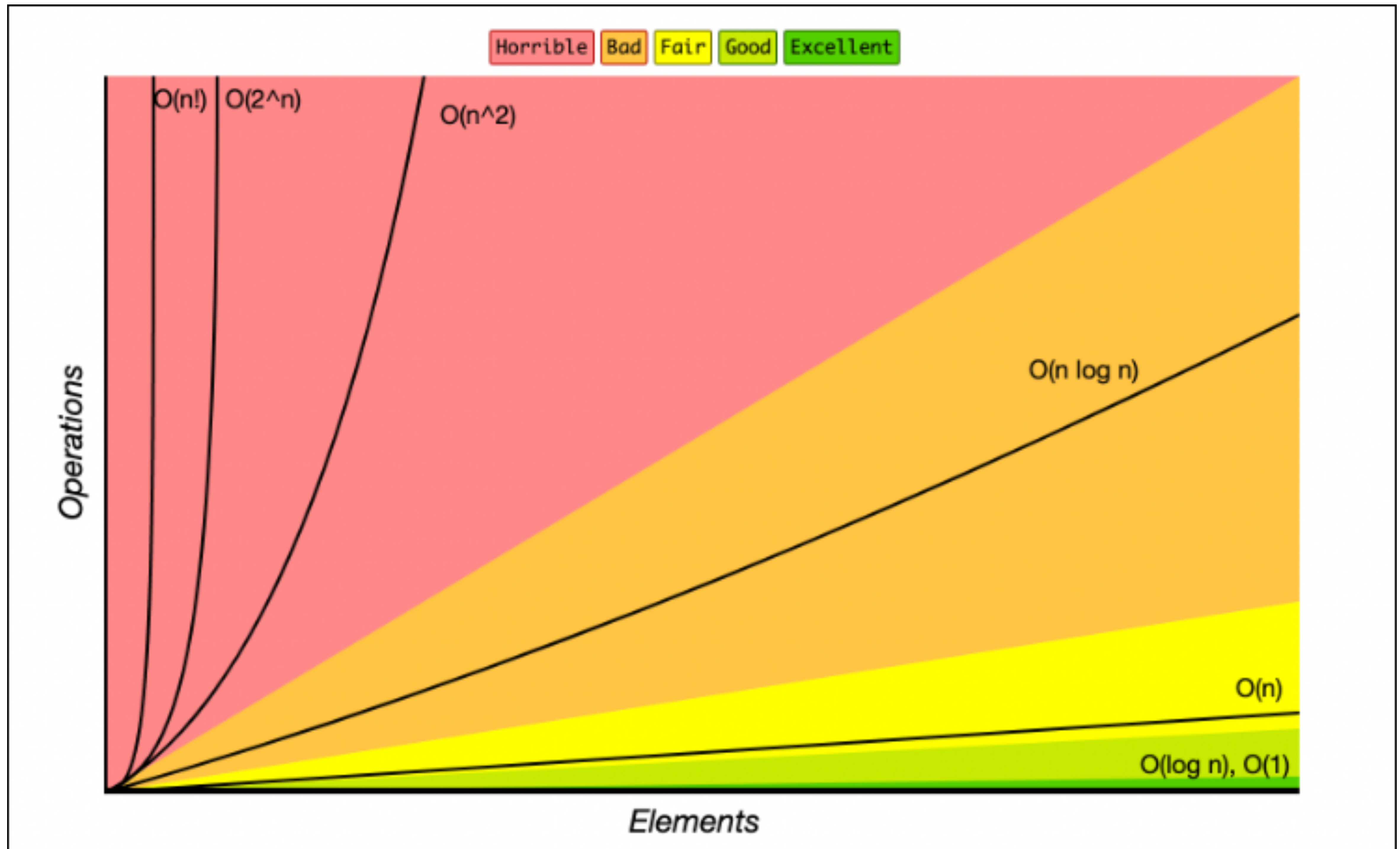
Big O Notation (O)

- Key idea: this notation provides an upper bound on the growth rate of an algorithm’s running time (or memory usage).
    - It represents the *worst case scenario,* that is the maximum amount of time (or space) an algorithm may need to solve a problem.
    - This gives us groups of methods/functions that behave similarly.
- $O(f(n))$: as n increases, the run-time increases with respect to $f(n)$.
    - n represents the problem size.

Common types of runtime complexities in Big O notation

- $O(1)$ = constant
  - Doubling the input size, won't affect the run-time.
  - Holy-grail but often pretty hard to accomplish.
- $O(\log_2 n)$ = logarithmic
  - Doubling the input size, will increase the runtime by a constant.
- $O(n)$ = linear
  - Doubling the input size, will result to double the run-time.
  - Rule of thumb: try to keep your programs running at or below this range.
- $O(n^2)$ = quadratic
  - Doubling the input size, will result to quadrupling the run-time
- Even faster growing (slower) families of algorithms such as exponential $O(2^n)$or factorial $O(n!)$.

Common types of runtime complexities in Big O notation



https://www.bigocheatsheet.com/

An example

- Say we have an $O(n^2)$ algorithm
  - For an input of size k it takes t time
  - Since it's quadratic, for an input of size k it does on the order of k^2 work to get the answer in time t.
  - If we double the size: $(2k)^2 = 4k^2$ work needs to be done to get the answer. How long will that take?
    - we know doing k^2 work takes time t
    - Thus, $4k^2$ work will take $4t$ time.

Revisit `contains` function on a list (i.e. `in`)

```
def contains(list, item):
    for thing in list:
        if thing == item:
            return True
    return False
```

- What is the Big-O for `contains`?
  - $O(n)$
  - The runtime increases linearly with the size of the list.

Sorting

- Input: A list of numbers `nums`.
- Output: The list of numbers in sorted order, i.e. `nums[i] <= nums[j]` for all `i < j`.
- many different ways to sort a list.

Selection sort

- High-level: starting from the beginning of the list and working to the end, find the smallest element in the remaining list.
  - In the first position: put the smallest item in the list
  - In the second position: put the next smallest item in the list
  - - …
  - To find the smallest item in the remaining list, simply traverse it, keeping track of the smallest value.

http://algs4.cs.princeton.edu

2.1 SELECTION SORT DEMO

selection_sort running time

- We'll use the variable n to describe the length of the list we want to sort.
- How many times do we go through the external for loop in selection_sort?
    - n times
    - External for loop runtime is $O(n)$
- Each time through the internal for loop in selection_sort, we find the smallest element. How much work is this?
    - first time: $n - 1$, second: $n - 2$, third: $n - 3$ …
    - Internal for loop runtime is $O(n)$
- What is the overall cost for selection_sort, if we focus on how many comparisons (some_list[j] < some_list[min_index]) we make?
    - $(n-1) + (n-2) + (n-3) + \ldots + 2 + 1 = \frac{n(n-1)}{2} \approx O(n^2)$
    - Another way of thinking about it is that we go through the external for loop n times and each time we go through the for loop we incur a cost of roughly n. Overall: $O(n^2)$.
    - Same complexity for average and best case.

Insertion sort

- High-level: starting from the beginning of the list and, working towards the end, keep the list items we've seen so far in sorted order. For each new item, traverse the list of things sorted already and insert it in the correct place.

http://algs4.cs.princeton.edu

2.1 INSERTION SORT DEMO

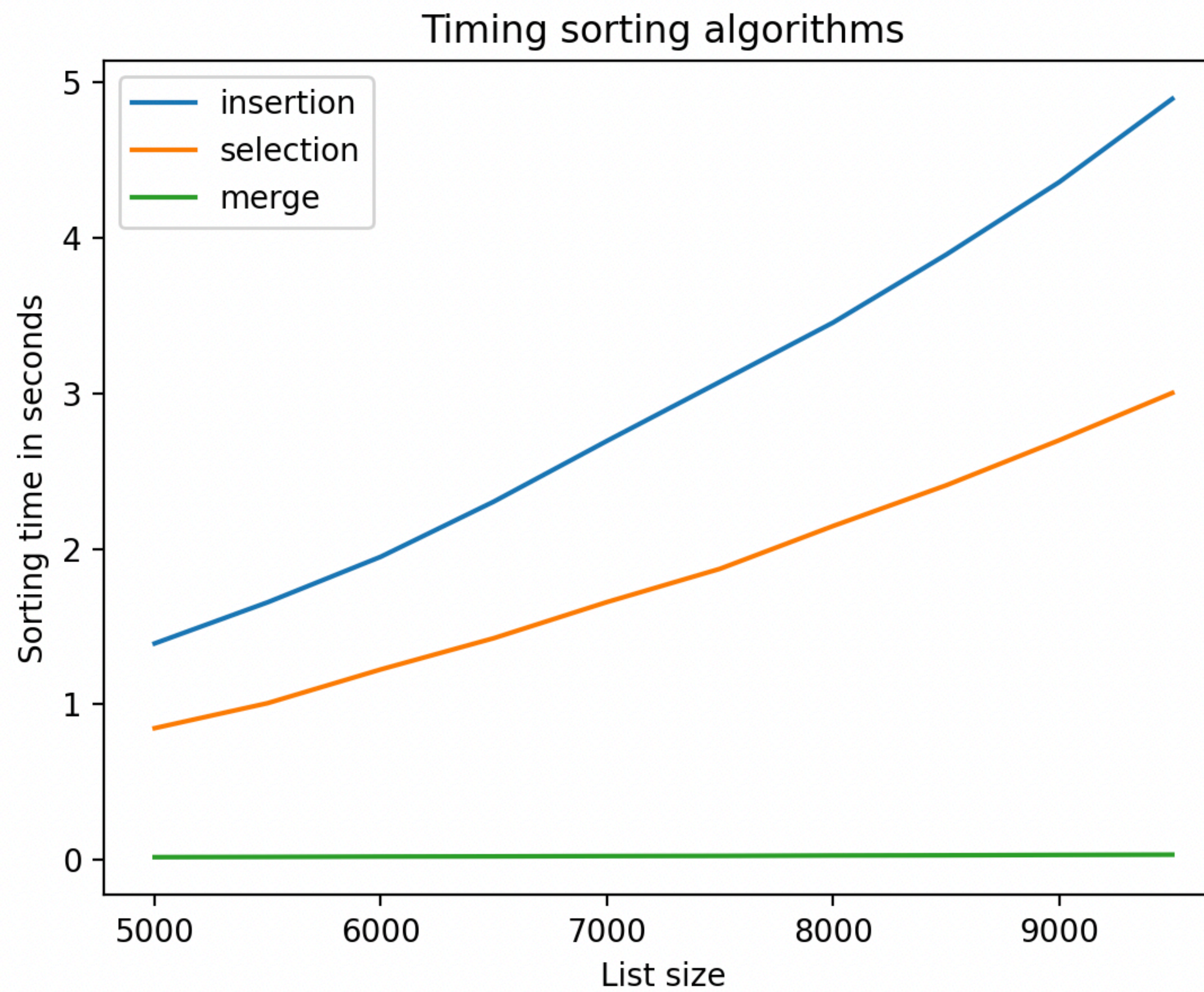
`insertion_sort` running time

- How many times do we compare elements (`if some_list[j] < some_list[j-1]`)?
  - In the best case: $n - 1$ times (we always break). ~$O(n)$
    - When does this happen?
      - When the list is already sorted.
- How many times do we compare/exchange elements in the worst case of a reversely sorted list:
  - $O(n^2)$
- How many times do we compare/exchange elements in the average case of sorting a randomly sorted list:
  - $O(n^2)$

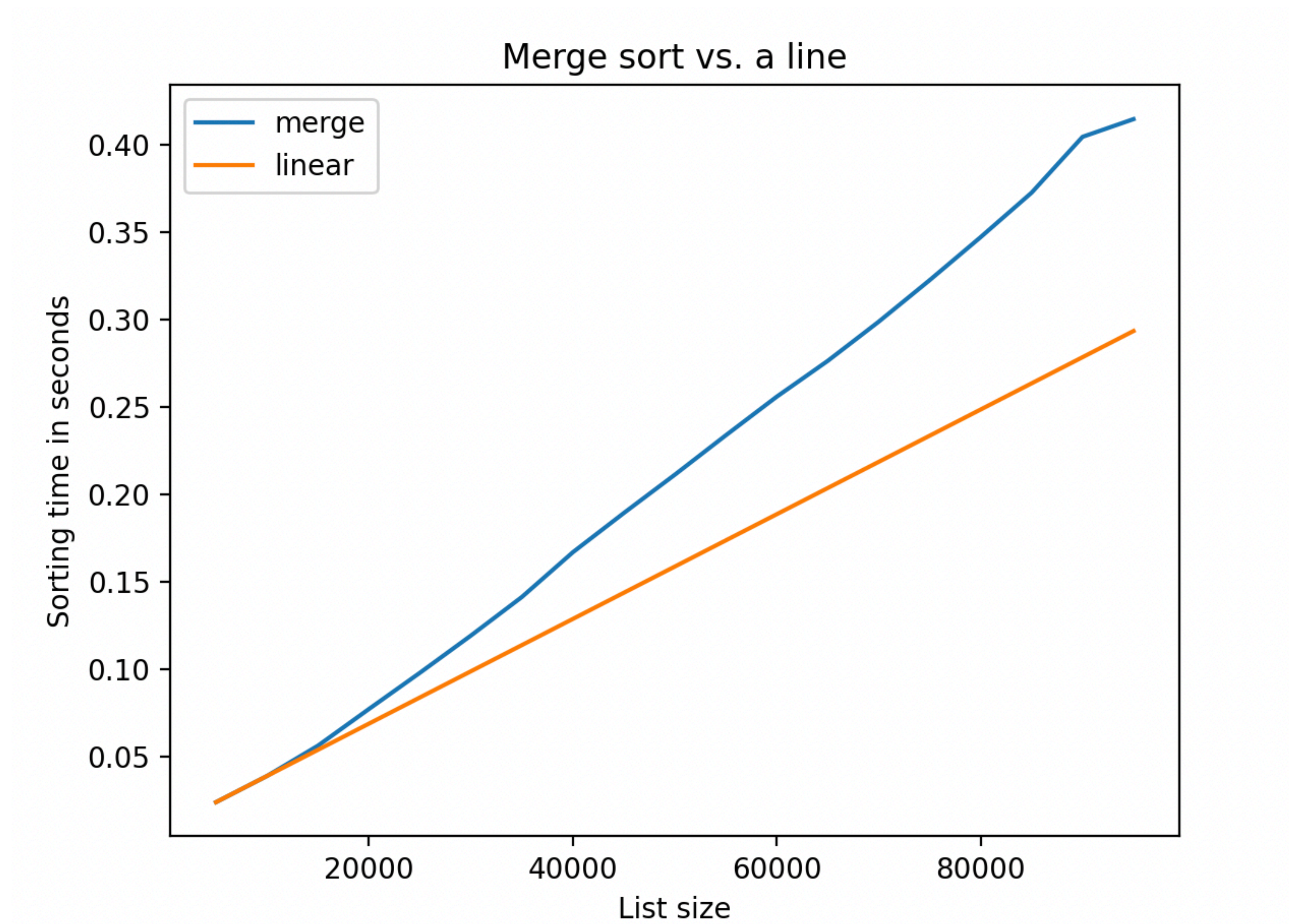
Merge sort

- You'll learn about it later :)
- $O(nlogn)$ running time!!
  - Linearithmic
  - Lower bound (best running time we can achieve) for comparison-based sorting algorithms!

Timing sorting algorithms



Timing merge sort



Resources

- [sorting.py]

Homework

- None!