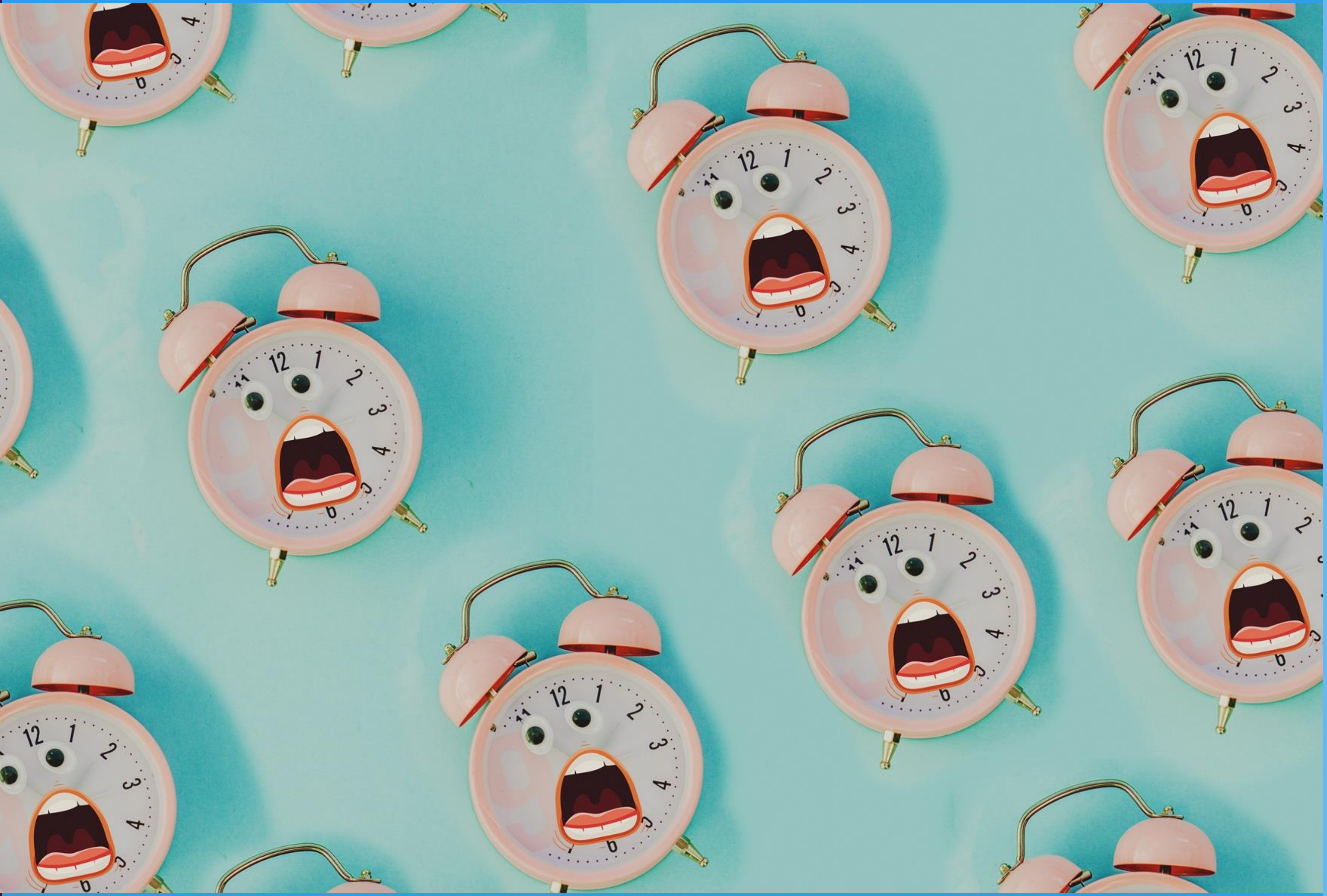


Data Structures and Algorithms

Asymptotic Runtime Analysis

CS51 – Spring 2026

Main parts to analyzing algorithms

- Does the algorithm work (**correctness**)?
- How fast does the algorithm run or how much memory does it need (**efficiency**)?
- We will focus on analyzing the runtime of algorithms.
- Our ultimate goal is to pick the best performing algorithm.

Asymptotic runtime analysis

- How does the runtime of our algorithm grow as we increase the size of the problem?
  - Size of the problem could be the number of elements in a list we want to sort or number of characters in a string we want to reverse, etc.
- For example, if we double the input, what will happen to the runtime? Will it...
  - stay unchanged?
  - double?
  - triple?
  - quadruple?
  - ...
- Asymptotic runtime analysis relies on math to see how the work that the algorithm scales in respect to input size.
- It does NOT provide an exact running time nor is an experimental approach where we measure how long our code takes to run using a stopwatch.

Big O notation

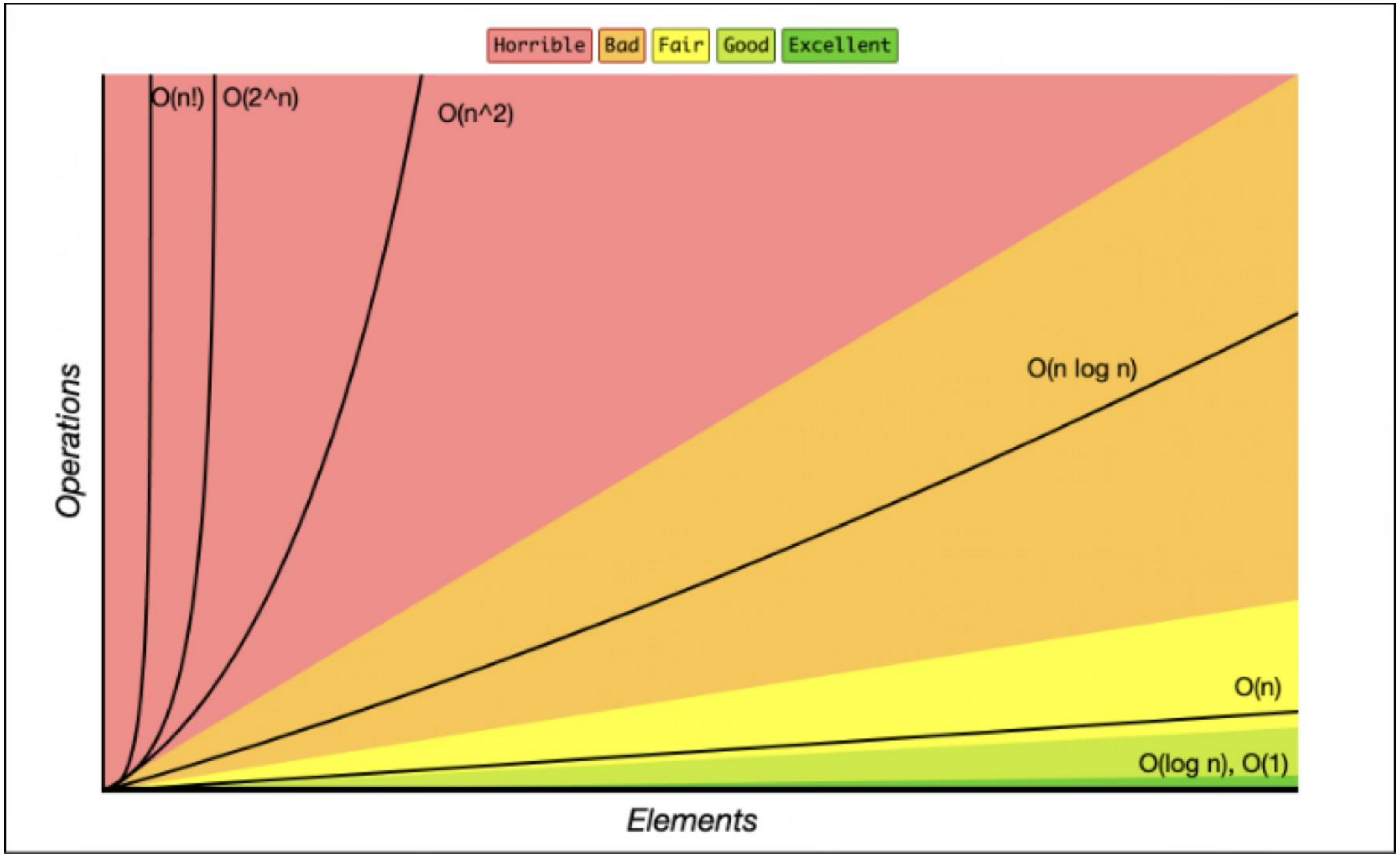
- Big O notation characterizes functions according to their growth rates.
- The **growth rate** of a function is also referred to as the **order of the function**.
- Definition: $f(n) = O\big(g(n)\big)$ if $|f(n)| \leq c \times |g(n)|$, for all $n > n_0$.
  - We read this as "$f(n)$ is big O of $g(n)$" or "$f(n)$ is of the order of $g(n)$"
- The analysis is asymptotical because we refer to very large sizes for n. In such settings, if a function consists of multiple terms only the one with the largest growth rate is kept, the others are omitted.
- For example, $f(n) = 6n^3 + 2n^2 + 1$. We can say that $f(n) = O(n^3)$ as the one with the largest exponent is the one that contributes the highest growth as the problem size increases.

https://en.wikipedia.org/wiki/Big_O_notation

Common classes of runtime complexities

- $O(1)$ = constant
  - Doubling the input size, won't affect the runtime. Holy-grail but often pretty hard to accomplish.
- $O(\log n)$ = logarithmic
  - Doubling the input size, will increase the runtime by a constant.
- $O(n)$ = linear
  - Doubling the input size, will result to double the runtime.
  - 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 runtime.
- $O(n^3)$ = cubic
  - Doubling the input size, will result to eight times longer runtime.
- Even faster growing (slower) algorithms such as exponential (c^n) or factorial ($n!$).

Common classes of runtime complexities



https://www.bigocheatsheet.com/

Practice time

Using big O notation, simplify the following functions:

- $3n^3 + 2n + 7$
- $2^n + n^2$
- 1000
- $n + 1000$
- $1 + \frac{1}{n}$

Answer

Using big O notation, simplify the following functions:

- $3n^3 + 2n + 7 = O(n^3)$
- $2^n + n^2 = O(2^n)$
- $1000 = O(1)$
- $n + 1000 = O(n)$
- $1 + \frac{1}{n} = O(1)$

Revisiting linear_search function

```
def linear_search(lst, element):
    n = len(lst)
    for i in range(n):
        if lst[i]==element:
            return i
    return -1
```

- What is the runtime for linear_search counting number of comparisons (==)?
  - In the worst case, $O(n)$. We have to go through the entire list to either not find the element at all or find it in the last position.
  - In the average case, the element will be somewhere in the middle and we will have to make about n/2 comparisons $O\left(\frac{n}{2}\right) = \mathrm{O(n)}$.
  - In the best case, the element will be in the first index and we will only need to make one comparison, thus $O(1)$.

Revisiting bubblesort

```
def bubblesort (lst):
    n = len(lst)
    for i in range(n):
        for j in range(n-i-1):
            if lst[j]>lst[j+1]:
                exchange(lst, j, j+1)
```

- What is the big O for bubblesort in the worst case?
  - In first outer loop, we make n-1 comparisons.
  - In second outer loop, we make n-2 comparisons.
  - In third outer loop, we make n-3 comparisons.
  - ...
  - In last outer loop, we make 1 comparison.
  - Hence, the number of comparisons is $(n-1)+(n-2)+(n-3)+\cdots+1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$
  - This pattern is common in nested if you have two nested for loops.
- What is the big O for the average and best case?
  - Still $O(n^2)$.

Revisiting optimized bubblesort

- Worst case and average case analysis doesn't change.
- But we can think of a best-case scenario where only n comparisons are needed because the list is already sorted.
- This small modification changed our best case from $O(n^2)$ to $O(n)$!

```
def bubblesort (lst):
    n = len(lst)
    for i in range(n):
        exchanged = False
        for j in range(n-i-1):
            if lst[j]>lst[j+1]:
                exchange(lst, j, j+1)
                exchanged = True
        if not exchanged:
                break
```

Revisiting selectionsort

```
def selectionsort (lst):
    n = len(lst)
    for i in range(n-1):
        min_index = i
        for j in range(i+1, n):
            if lst[j]<lst[min_index]:
                min_index = j
        exchange(lst, i, min_index)
```

- What is the big O for seletionsort in the worst case?
  - In first outer loop, we make n-1 comparisons.
  - In second outer loop, we make n-2 comparisons.
  - In third outer loop, we make n-3 comparisons.
  - ...
  - In last outer loop, we make 1 comparison.
  - Hence, the number of comparisons is $(n-1)+(n-2)+(n-3)+\cdots+1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$
  - What is the big O for the average and best case?
    - Still $O(n^2)$.

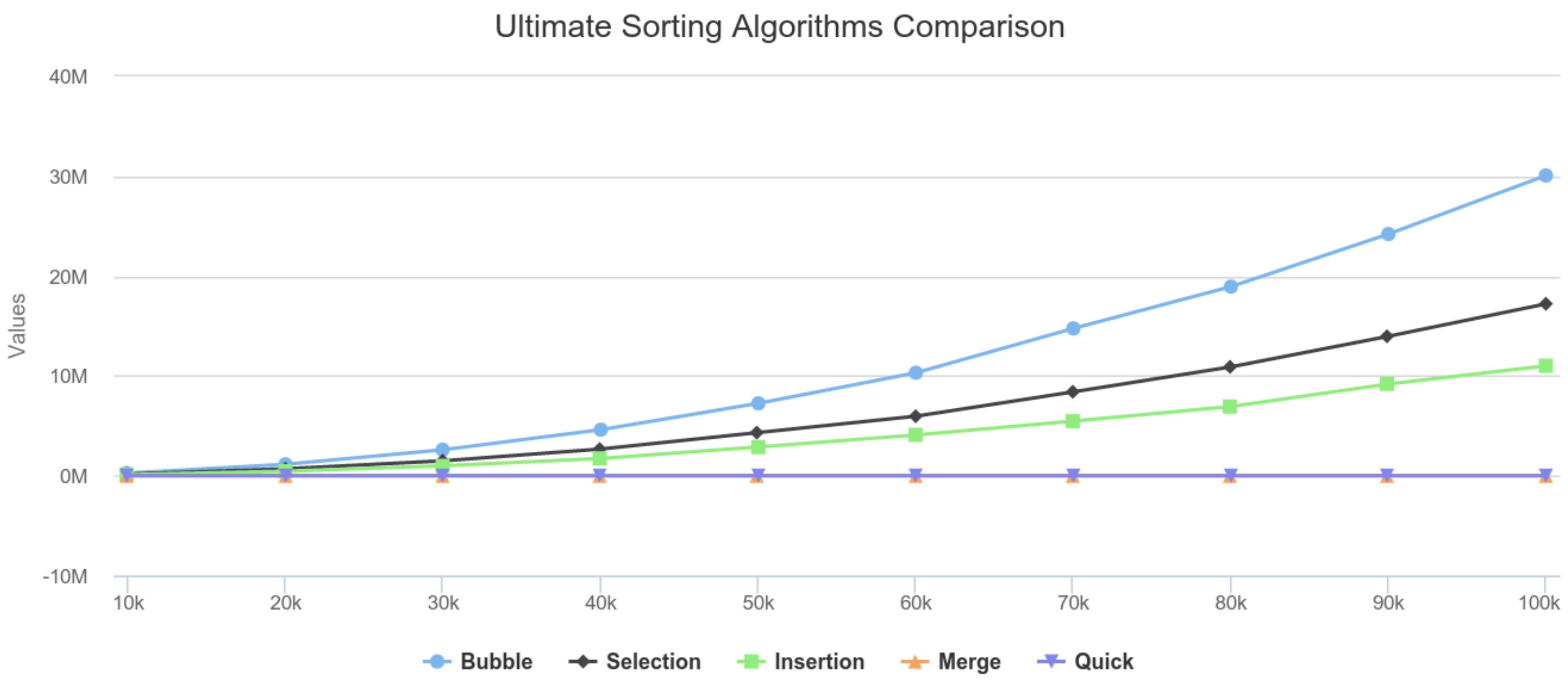
Revisiting insertionsort

```
def insertionsort (lst):
    n = len(lst)
    for i in range(1, n):
        for j in range(i, 0, -1):
            if lst[j]<lst[j-1]:
                exchange(lst, j, j-1)
            else:
                break
```

- What is the big O for insertionsort in the worst case?
  - $O(n^2)$.
  - The list is sorted in reverse order.
- What is the big O for the average case?
  - Still $O(n^2)$.
  - The list elements are randomly allocated.
- What is the big O for the best case?
  - $O(n)$.
  - The list is already sorted and we make one comparison only for each of the n runs of the outer loop.

Is this the best we can hope for?

- mergesort and quicksort are two comparison-based sorting algorithms with $O(n \log n)$ average performance.
- This is the lower bound for how well we can expect any comparison-based sorting algorithms.



https://titrias.com/ultimate-sorting-algorithms-comparison/

Fun times

https://youtu.be/k4RRi_ntQc8?si=Sj6UngM09eCja1FR

More fun

https://youtu.be/QdQmAdyfmDI?si=GdvEyQ2FO5lDoQks