Big O

Big O: Introduction

- Big O is a way to measure the efficiency of algorithms.
- You can analyze **time complexity** and **space complexity**.
- **Time complexity** refers to the number of operations a program takes to complete a task.
  - **Time complexity is independent of actual time!**
  - This means a faster computer may run an algorithm faster, but Big O only measures the growth rate of operations, not the actual execution time.
- **Space complexity** refers to how much memory a program uses to complete its operations.

Big O: Worst Case

- **Ω (Omega)** → Best case scenario.
- **Θ (Theta)** → Average case scenario.
- **O (Big O)** → Worst case scenario.
- When we talk about Big O, we are usually referring to the **worst-case scenario**.

O(n)

- Runs in **n** times.
- Forms a **straight line** on a graph → **proportional growth**.
- For a dataset of size **n**, it takes **n** operations to complete.
- **Example**: Iterating through an entire list to find a specific number.

Drop Constants

- You can simplify **O(2n)** to **O(n)**.

O(n²)

- **A loop inside another loop**.
- It runs in **n²** because for each iteration of **n**, it iterates over **n** again.

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

Drop Non-Dominant Terms

- If you have an expression like **O(n² + n)**, the **n** becomes irrelevant for large datasets.
- Since **n²** is the **dominant** term, we drop **n**.
- So **O(n² + n)** → **O(n²)**.

O(1)

- The number of operations **does not** increase as **n** gets bigger.
- Also called **constant time**.
- Example: Accessing an element in an array by index.

O(log(n))

- **Binary Search!!** Example: log28=3\log_2{8} = 3
- Continuously **dividing in half** until the number is found.
- **The list must be sorted!**
- **O(n log n)** appears in some sorting algorithms (e.g., Merge Sort, Quick Sort).

Different Terms for Inputs

- If a function takes **two different arguments** and iterates through both, you **can't** say it's simply **O(2n)** → **O(n)**.

- Example: A function that receives **a** and **b**, and has two separate loops iterating through **a** and **b**.
  - **Time complexity: O(a + b)**.

Big O of Lists (Python)

- **append()** and **pop() (at the end)** → **O(1)**.
- **pop(index)** and **insert(index, value)** → **O(n)** (because items must be shifted).
- **Accessing an item by index** → **O(1)**.
- **Searching for an item by iterating through a list** → **O(n)**.

Wrap-Up

- **$O(n^2)$ → Loop within a loop**.
- **O(n) → Proportional**.
- **O(log n) → Divide and conquer**.
- **O(1) → Constant time**.

Sorting Algorithms

- **Quicksort is $O(n^2)$ worst case but has $\Omega(n \log n)$ in the best case.**
- **Bubble Sort & Selection Sort:**
  - **$\Omega(n)$ best case** (if nearly sorted).
  - **$O(n^2)$ worst case** (if completely unsorted).
- **Quicksort, Mergesort, and Timsort have higher space complexity than simpler sorts.**
- **If data is already sorted or nearly sorted, Bubble Sort and Selection Sort can be efficient.**
- **Otherwise, Quicksort, Mergesort, and Timsort are better choices.**
