



# CoGrammar

## ORDER COMPLEXITY



**SKILLS  
FOR LIFE**

**SKILLS BOOTCAMPS**



Department  
for Education

## Foundational Sessions Housekeeping

---

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.  
**(FBV: Mutual Respect.)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Open Classes.  
You can submit these questions here:

[SE Open Class Questions](#) or [DS Open Class Questions](#)

## Foundational Sessions Housekeeping cont.

---

- For all **non-academic questions**, please submit a query: [www.hyperiondev.com/support](https://www.hyperiondev.com/support)
- Report a **safeguarding** incident: [www.hyperiondev.com/safeguardreporting](https://www.hyperiondev.com/safeguardreporting)
- We would love your **feedback** on lectures: [Feedback on Lectures](#)

# Reminders!

## Guided Learning Hours

*By now, ideally you should have 7 GLHs per week accrued. Remember to attend any and all sessions for support, and to ensure you reach 112 GLHs by the close of your Skills Bootcamp.*

# Progression Criteria

## ✓ **Criterion 1: Initial Requirements**

- Complete 15 hours of Guided Learning Hours and the first four tasks within two weeks.

## ✓ **Criterion 2: Mid-Course Progress**

- Software Engineering: Finish 14 tasks by week 8.
- Data Science: Finish 13 tasks by week 8.

## ✓ **Criterion 3: Post-Course Progress**


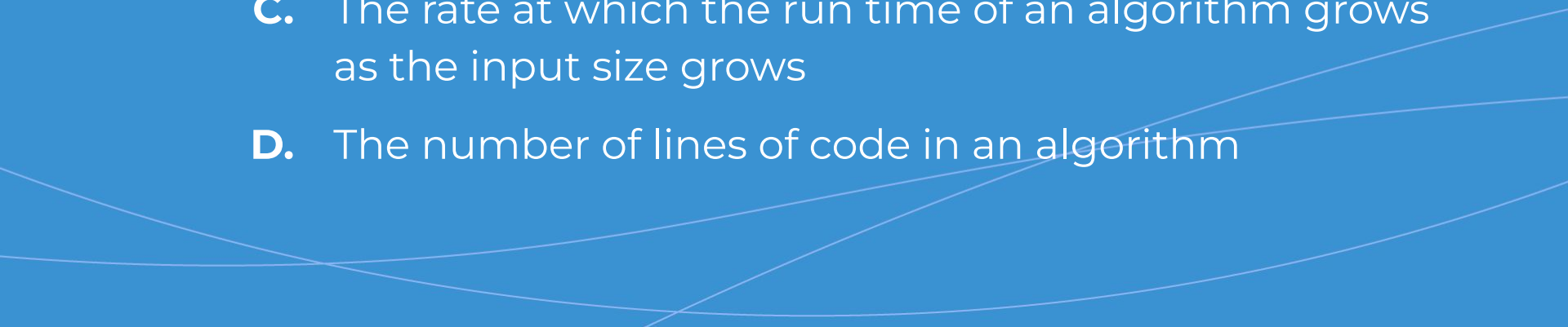
- Complete all mandatory tasks by 24th March 2024.
- Record an Invitation to Interview within 4 weeks of course completion, or by 30th March 2024.
- Achieve 112 GLH by 24th March 2024.

## ✓ **Criterion 4: Employability**

- Record a Final Job Outcome within 12 weeks of graduation, or by 23rd September 2024.



# What does the term 'Order Complexity' primarily refer to in algorithms?

- 
- A.** The order in which code is written in an algorithm
  - B.** The amount of time it takes to compile an algorithm
  - C.** The rate at which the run time of an algorithm grows as the input size grows
  - D.** The number of lines of code in an algorithm
- 

# What is an asymptote in the context of algorithm analysis?

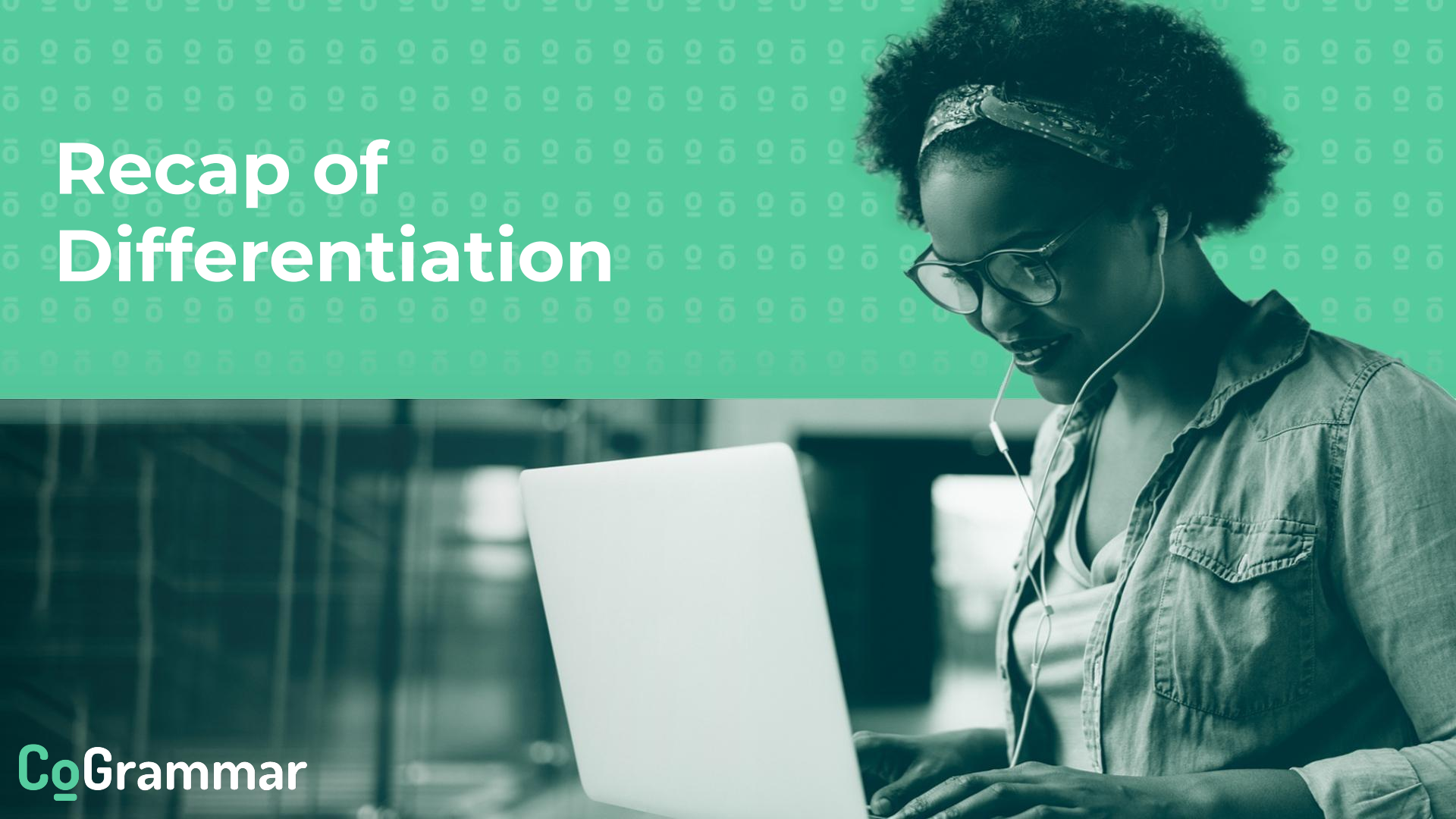
- A.** A specific type of algorithm used for data sorting
- B.** A value (line) to which the observed function constantly approaches but never touches or crosses
- C.** The maximum time an algorithm will take to run
- D.** A tool used to debug algorithms

# Which of the following best describes Big O notation?

- A.** A programming language used for developing algorithms
- B.** A notation that specifies the exact run time of an algorithm
- C.** A notation expressing the upper bound of the running time of an algorithm
- D.** A method to determine the memory usage of an algorithm



# Recap of Differentiation



# Gradients of Linear Functions

**A constant value that represents the rate of change of the function.**

- The gradient is calculated as the change in y over the change in x or “rise over run”:

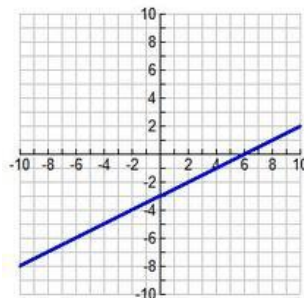
$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

using two points on the line  $(x_1, y_1)$  and  $(x_2, y_2)$ , the starting point and end point, respectively.

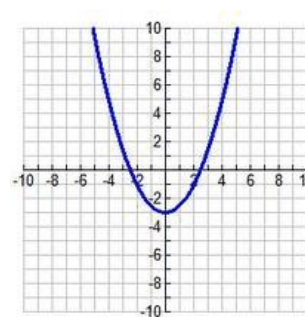
# Derivatives

The rate of change of the function with respect to an independent/input variable.

- Linear functions have a **constant rate** of change/gradient, where other types of functions do not e.g. quadratic functions



Linear



Quadratic

# Rules of Differentiation

- **Constant rule:** if  $C$  is a constant,

$$\frac{d}{dx}C = 0$$

- **Constant multiple rule:** if  $C$  is a constant,

$$\frac{d}{dx}Cf(x) = Cf'(x)$$

- **Power rule**

$$\frac{d}{dx}x^n = nx^{n-1}$$

- **Sum and Difference rule**

$$\frac{d}{dx}[f(x) \pm g(x)] = \frac{d}{dx}f(x) \pm \frac{d}{dx}g(x)$$

# Rules of Differentiation

- **Product Rule**

$$\frac{d}{dx} [f(x)g(x)] = f(x)g'(x) + g(x)f'(x)$$

- **Quotient Rule (derived from product rule)**

$$\frac{d}{dx} \left[ \frac{f(x)}{g(x)} \right] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$$

- **Chain Rule**

$$\frac{d}{dx} f[g(x)] = f'[g(x)]g'(x)$$

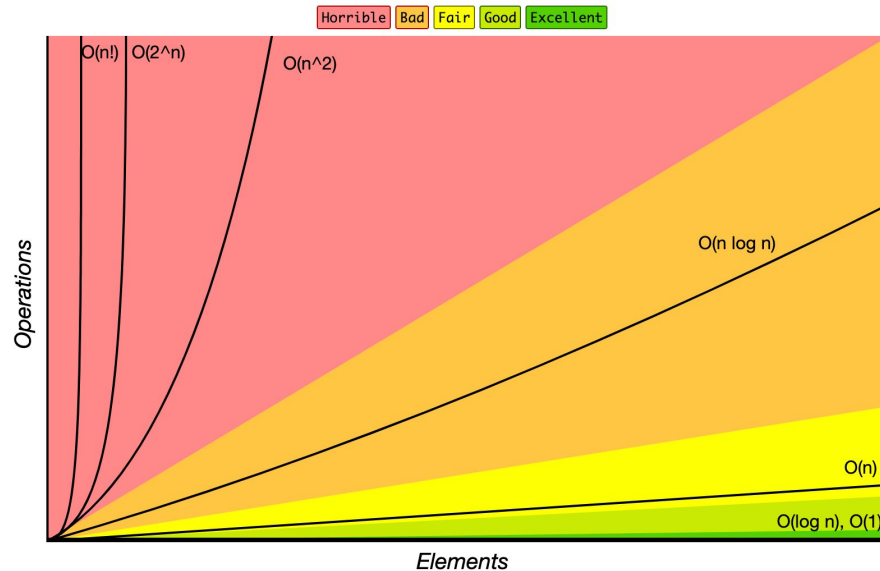


# Order Complexity Topics

1. Asymptotes
2. Algorithm Complexity Analysis
3. Comparing Time and Space Complexities
4. Common Sorting and Searching Algorithms

# Common Complexities

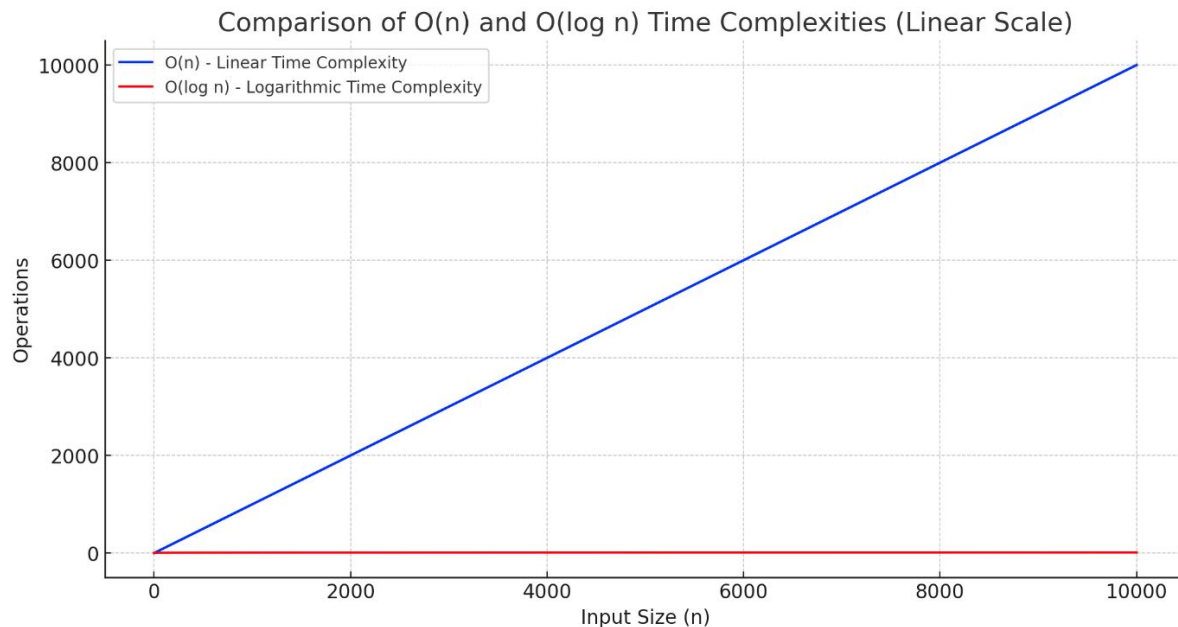
Keep these complexities in mind throughout the lecture as they are important and the most used complexities in industry.



# Why is Order Complexity Important?

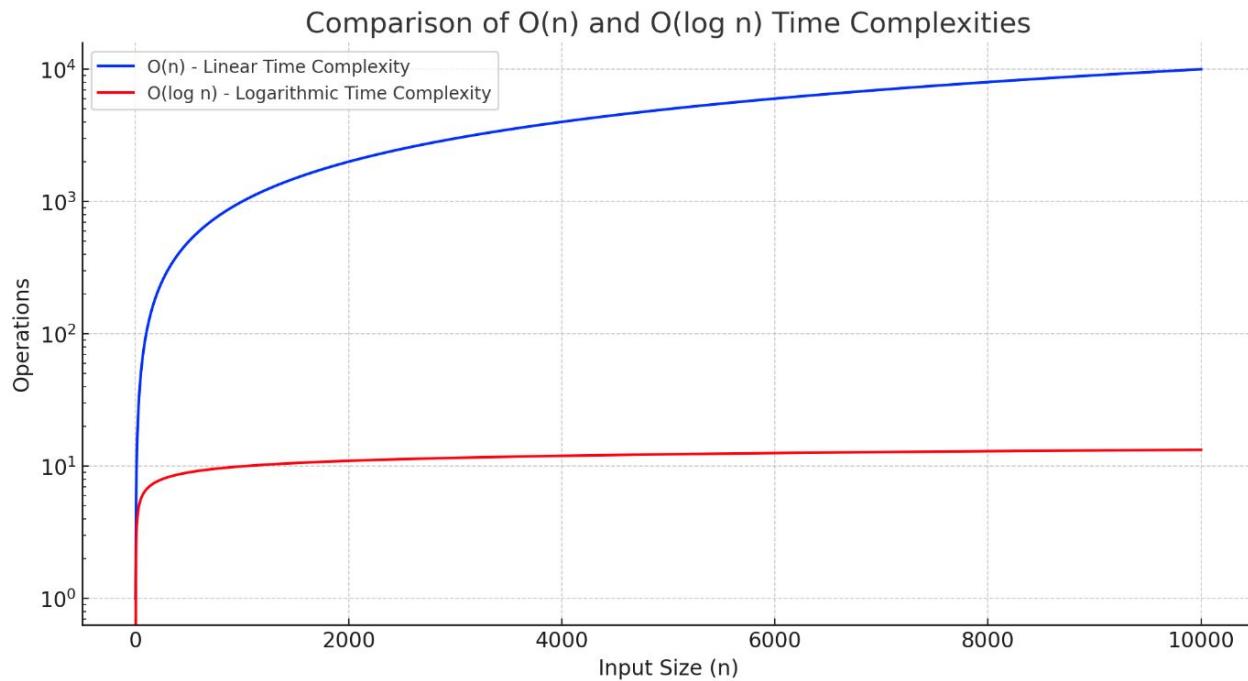
- Imagine a large e-commerce company that houses millions of products and allows customers to search for the products by keywords.
- If this search has a linear complexity it would take significantly more operations than if it has a logarithmic complexity, as shown on the next slide.

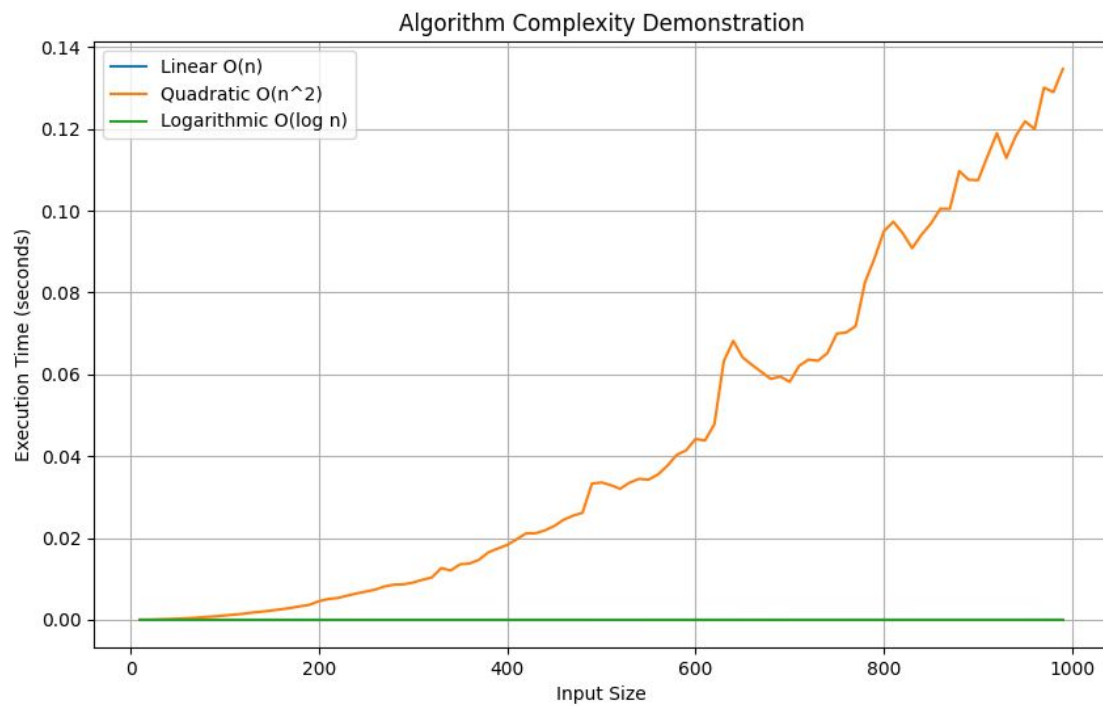




- The  **$O(\log n)$**  line appears relatively flat, especially when compared to the  **$O(n)$**  line, due to the nature of logarithmic growth. Logarithmic growth increases quickly at first but then slows down significantly as the input size grows.

- To see it a bit better we can log the y axis and we'll notice the logarithmic scale spikes from 0 to 10 operations quickly and levels out.





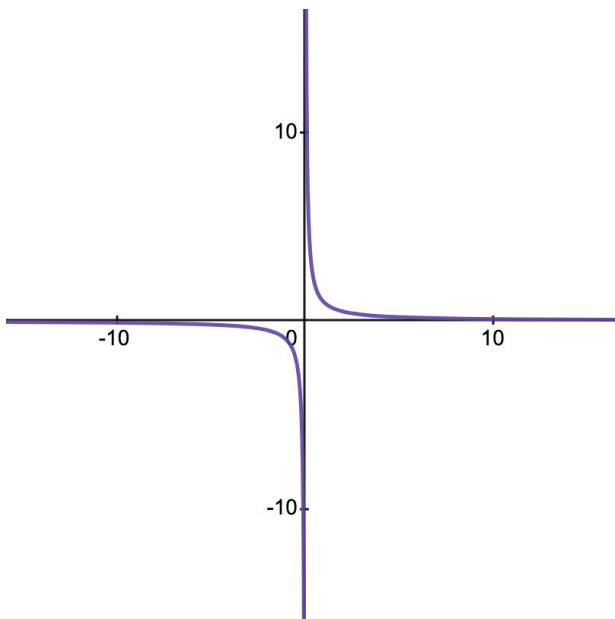
It is not hard to imagine the vast difference in customer retention and profits a company would realise if this one search function was more efficient. Thus it's important for you to understand as you are the engineers of these algorithms in the software.

# Asymptotes

**An asymptote is a line that a curve approaches as it heads towards infinity.**

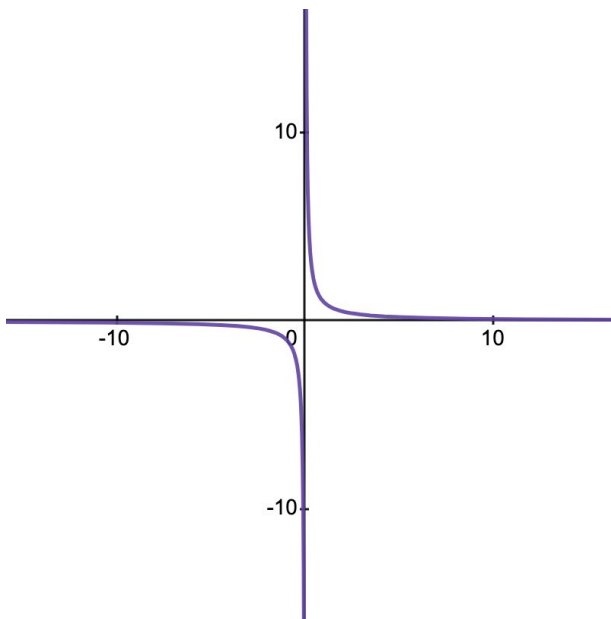
- We learn about asymptotes to **understand and predict the behavior of algorithms as the size of their input grows**, which is crucial for evaluating their efficiency and scalability.
- Types of Asymptotes:
  1. Vertical
  2. Horizontal
  3. Oblique

- **Vertical asymptotes** illustrate points where an algorithm fails or becomes highly inefficient.



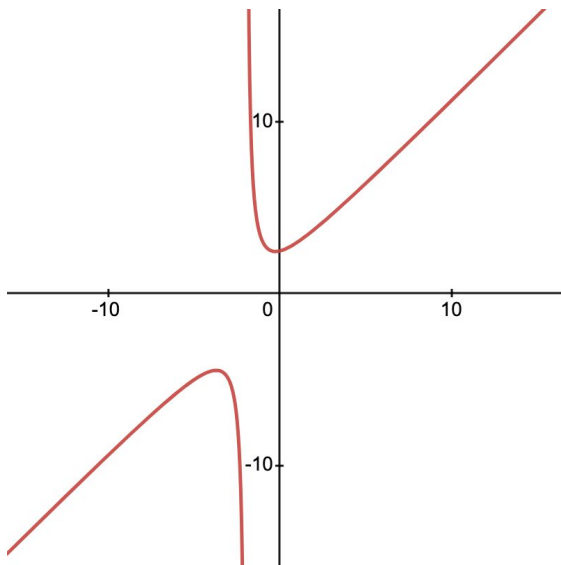
- For example  **$f(x)=1/x$** .
- Here  $x$  approaches 0 but never quite reaches it, thus  $x = 0$  is the vertical asymptote.
- Like in the graph, an algorithm might face a similar 'spike' in resource usage or fail at specific input values, akin to **infinite recursion without a base case**.

- **Horizontal asymptotes** represent algorithms whose performance stabilizes as input size grows.



- For example  **$f(x)=1/x$** .
- As  **$x$**  reaches infinity,  **$f(x)$**  becomes 0. Thus  **$f(x)=0$**  is the horizontal asymptote.
- This is similar to a **cache lookup operation** with  $O(1)$  complexity, where time taken remains constant irrespective of input size.

- **Oblique asymptotes** indicate algorithms with complexity increasing non-linearly with input size.



- For example  $f(x) = \frac{x^2 + 3x + 5}{x + 2}$  The oblique asymptote is the line described by  **$f(x) = x + 1$**  since this line is never touched.
- Comparable to  **$O(n \log n)$  complexity**, where time grows more than linearly (not as steep as quadratic) with larger input sizes.



# Algorithmic Complexity

The measure of the amount of resources, such as time and/or space, required by an algorithm to solve a problem as a function of the size of the input.

- Let's break down the definition:
  1. Measure of amount of resources: it can be **quantified**.
  2. Required to solve a problem: algorithms are **sets of instructions to solve specific problems**.
  3. A function of the size of the input: it is not fixed, it **changes as the input changes**.

- **Space complexity:** measures the amount of memory space required by a function.
- **Time complexity:** measures the amount of time a function takes to complete.

We keep talking about “measuring” resources, which leads us to one of the most important concepts in computer science and algorithmic construction: **Big O notation**.

- Like asymptotes show us what happens if  $x$  grows to infinity, **Big O notation** helps us understand the behaviour of an algorithm as the input size approaches infinity.

# Example: Linear Search

- Big O notation shows the worst-case complexity of an algorithm.

```
def linear_search(arr, target):  
    for i in range(len(arr)):  
        if arr[i] == target:  
            return i # Target found  
    return -1 # Target not found
```

```
# Example Usage  
arr = [3, 5, 2, 4, 9]  
target = 4
```

The worst case is if the target is the last element, meaning we iterate over all  $n$  elements of the input array, thus **the worst-case time complexity is  $O(n)$** .

No matter the input, only space to store  $i$  and  $target$  are needed, so **the worst-case space complexity is  $O(1)$** .

# Example: Fibonacci

```
def fibonacci_recursive(n):  
    if n <= 1:  
        return n  
    else:  
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
```

Each call generates two more calls, which results in an exponential growth in complexity, in fact the **worst-case time complexity is  $O(2^n)$** .

This is a recursive function, so it is stored as a “stack” in memory, which we won’t get into here. For each input it adds a layer to the stack, making the **worst-case space complexity  $O(n)$** .

# Comparing Time and Space Complexities

```
def linear_search(arr, target):  
    for i in range(len(arr)):  
        if arr[i] == target:  
            return i  
    return -1
```

Linear search has  **$O(n)$**  time complexity. Binary search has  **$O(\log(n))$**  time complexity.

Both have  **$O(1)$**  space complexities.

```
def binary_search(arr, target):  
    low, high = 0, len(arr) - 1  
    while low <= high:  
        mid = (low + high) // 2  
        if arr[mid] == target:  
            return mid  
        elif arr[mid] < target:  
            low = mid + 1  
        else:  
            high = mid - 1  
    return -1
```

- We already graphed the difference between  $O(n)$  and  $O(\log(n))$  time complexities at the start of the lecture, so to convince yourself why  **$O(\log(n))$  is far more efficient than  $O(n)$**  just replace  $n$  with a large number.
- For example:
  1.  $n = 100$ , then  $\log(n) = 2$
  2.  $n = 1\,000$ , then  $\log(n) = 3$
  3.  $n = 1\,000\,000\,000$ , then  $\log(n) = 9$

- Looking at the previous example we see that if the input is 1 billion  **$O(n)$  will take 1 billion time units where  $O(\log(n))$  would take 9 time units at most.**

# Common Sorting and Searching Algorithms

## Common sorting algorithms:

1. **Bubble Sort** - Time Complexity:  $O(n^2)$ , Space Complexity:  $O(1)$
2. **Selection Sort** - Time Complexity:  $O(n^2)$ , Space Complexity:  $O(1)$
3. **Insertion Sort** - Time Complexity:  $O(n^2)$ , Space Complexity:  $O(1)$
4. **Merge Sort** - Time Complexity:  $O(n \log(n))$ , Space Complexity:  $O(n)$
5. **Quick Sort** - Time Complexity:  $O(n \log(n))$  average,  $O(n^2)$  worst-case, Space Complexity:  $O(\log(n))$
6. **Heap Sort** - Time Complexity:  $O(n \log(n))$ , Space Complexity:  $O(1)$



## Common searching algorithms:

1. **Linear Search** - Time Complexity:  $O(n)$ , Space Complexity:  $O(1)$
2. **Binary Search** - Time Complexity:  $O(\log(n))$ , Space Complexity:  $O(1)$
3. **Jump Search** - Time Complexity:  $O(\sqrt{n})$ , Space Complexity:  $O(1)$
4. **Interpolation Search** - Time Complexity:  $O(\log(\log(n)))$  average,  $O(n)$  worst-case, Space Complexity:  $O(1)$

## Worked Example

You are optimizing a program that sorts large datasets. You have two sorting algorithms at your disposal: **Bubble Sort** and **Quicksort**. You need to decide which one to use based on their time and space complexities.

- Which algorithm would you choose for sorting very large datasets and why?
- How do the time complexities of Bubble Sort and Quicksort differ for large datasets?

## Worked Example

You are optimizing a program that sorts large datasets. You have two sorting algorithms at your disposal: **Bubble Sort** and **Quicksort**. You need to decide which one to use based on their time and space complexities.

- Which algorithm would you choose for sorting very large datasets and why?

**For sorting very large datasets, Quicksort would be the preferred algorithm over Bubble Sort. This is because Quicksort has a better average and worst-case time complexity than Bubble Sort. Quicksort generally operates in  $O(n \log n)$  time, whereas Bubble Sort operates in  $O(n^2)$  time. For large datasets, the  $n^2$  complexity of Bubble Sort makes it impractically slow compared to the more efficient  $n \log n$  complexity of Quicksort.**

## Worked Example

You are optimizing a program that sorts large datasets. You have two sorting algorithms at your disposal: **Bubble Sort** and **Quicksort**. You need to decide which one to use based on their time and space complexities.

- How do the time complexities of Bubble Sort and Quicksort differ for large datasets?

**As mentioned, Bubble Sort has a time complexity of  $O(n^2)$ , meaning the time it takes to complete the sorting operation grows quadratically as the size of the dataset increases. Quicksort, on the other hand, has an average time complexity of  $O(n \log n)$ , indicating that it scales more effectively with larger datasets. The time required by Quicksort grows at a rate proportional to the size of the dataset multiplied by its logarithm, which is significantly more efficient than the quadratic growth of Bubble Sort for large datasets.**

## Worked Example

Imagine you are developing a mobile application that needs to sort small arrays of data. You are considering **Insertion Sort** and **Merge Sort** for this task. Given the constraints of mobile devices, such as limited memory, your choice should factor in both time and space complexities.

- For small datasets, which sorting algorithm would be more efficient and why?
- How does the space complexity of Merge Sort impact its suitability for memory-constrained environments?

## Worked Example

Imagine you are developing a mobile application that needs to sort small arrays of data. You are considering **Insertion Sort** and **Merge Sort** for this task. Given the constraints of mobile devices, such as limited memory, your choice should factor in both time and space complexities.

- For small datasets, which sorting algorithm would be more efficient and why?

**For small datasets, Insertion Sort might be more efficient than Merge Sort. This is because Insertion Sort has a lower overhead and is generally faster on small datasets due to its simpler algorithm. Although Merge Sort has a better overall time complexity  $O(n \log n)$  compared to  $O(n^2)$  for Insertion Sort, the constant factors and simpler operations of Insertion Sort often make it faster for small arrays.**

## Worked Example

Imagine you are developing a mobile application that needs to sort small arrays of data. You are considering **Insertion Sort** and **Merge Sort** for this task. Given the constraints of mobile devices, such as limited memory, your choice should factor in both time and space complexities.

- How does the space complexity of Merge Sort impact its suitability for memory-constrained environments?

**Merge Sort has a space complexity of  $O(n)$ , meaning it requires additional memory proportional to the size of the dataset. In memory-constrained environments, such as mobile devices, this additional memory requirement can be a significant drawback. The need for extra space to hold the divided arrays during the merge process might not be ideal for environments with limited memory availability. This consideration makes Merge Sort less suitable compared to algorithms with a constant space complexity, like Insertion Sort, which has a space complexity of  $O(1)$  and does not require additional memory beyond the input array.**

# Summary

---



## Asymptotes

- ★ A line that a curve approaches as it heads towards infinity.


## Complexity Analysis

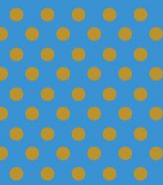

- ★ A measure of the amount of resources, such as time and/or space, required by an algorithm to solve a problem as a function of the size of the input.
- ★ Measured by Big O Notation, which measures the worst-case complexity and allows comparison of algorithms.



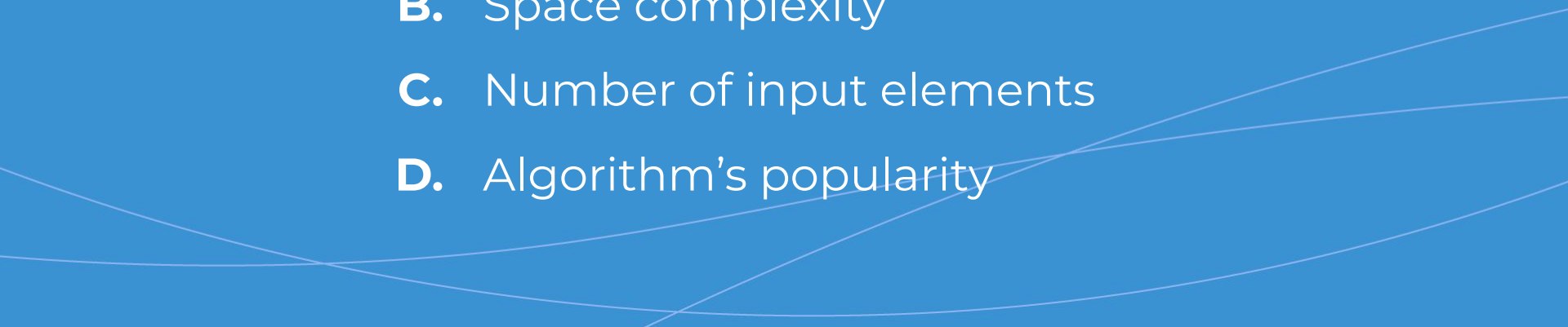


# How would you evaluate the worst-case complexity of a linear search algorithm in a list of $n$ elements?

- A.  $O(1)$
  - B.  $O(n)$
  - C.  $O(\log n)$
  - D.  $O(n^2)$
- 



# If you have to choose an algorithm for a memory-constrained environment, which aspect would you prioritize?

- A.** Time complexity
  - B.** Space complexity
  - C.** Number of input elements
  - D.** Algorithm's popularity
- 



# Questions and Answers

Questions around Sets, Functions and Variables

