General Assembly

STUDY GUIDE

BIG O NOTATION

Efficiency in Coding

Big O Notation is a technical way to describe the**efficiency** of an algorithm: how much time and space it needs to run.
- There are a lot of different ways to talk about an algorithm's efficiency, but software developers are mostly concerned with how an algorithm performs as its input gets bigger and bigger.
- For example, let's say you've written code that finds the largest value in an array. To gauge the efficiency of this code, you could ask:
- How does it perform when the array is 10 elements long?
- How does it perform when the array is 1,000 elements long?
- How does it perform when the array is 1,000,000 elements long?

Software engineers usually measure an algorithm's performance in the "**worst-case scenario**." In other words, they look at the maximum amount of time or space an algorithm could possibly require to run, given an input of a certain length.

What Is Big O Notation?

Big O Notation describes the behavior of an algorithm as its input increases. It can measure either time complexity or space complexity.
- **Time complexity**: the amount of time an algorithm takes to run
- Note: time complexity is not actually measured in increments of time (e.g., seconds); it's measured in terms of the number of basic steps an algorithm needs to take.

- **Space complexity**: the amount of memory or RAM an algorithm needs to run

Before we get into the technical classes of algorithmic complexity, let's get familiar with the informal categories using an analogy to cars:
- Highly efficient algorithms are like zippy little cars that get great gas mileage. They execute in the same amount of time or with the same amount of RAM, no matter the size of the input.
- Pretty good algorithms are like mid-size cars or station wagons. They execute in an amount of time that's directly proportional to the size of the input.
- Inefficient algorithms are like giant gas-guzzling trucks. They execute in an amount of time that increases exponentially (or worse) as the input increases.

Five Common Classes of Complexity

When using Big O Notation, it's conventional to drop coefficients, constants, and other less significant terms when describing an algorithm's complexity. We're less worried about the exact runtime of an algorithm and more about the broad class it falls into:
1. Constant complexity: **O(1)**
2. Linear complexity: **O(N)**
3. Quadratic complexity: **O(N^2)**
4. Logarithmic complexity: **O(log(N))**
5. Factorial complexity: **O(N!)**

Highly Efficient Algorithms: Constant and Logarithmic Complexity

Constant complexity is represented as **O(1)**.
- These algorithms have a constant space and time complexity, no matter the size of the input.
- Imagine you're throwing a bag of apples in the trash. This task will always take the same amount of time, no matter if you have 2 or 20 apples in that bag.

Logarithmic complexity is represented as **O(log(N))**.

- This type of algorithm is fast and efficient—it cuts the problem in half at each step.
- Imagine flipping through a phone book to find a someone's number.
- You could start at the beginning of the phone book and read every name on every page until it found the one you're looking for. That's a linear approach.
- It's much easier to read one random name and flip forward or backward depending on how close that name is to the name you're looking for. This approach is logarithmic.

Pretty Good Algorithms: Linear Complexity

Linear complexity is represented as **O(N)**.
- For this type of algorithm, as the input size increases, the processing time increases linearly.
- Imagine peeling apples to make apple sauce. This is a linear task: the amount of time it'll take you to peel all of the apples is directly proportional to the number of apples you have in the bag.

Inefficient Algorithms: Quadratic and Factorial Complexity

Quadratic complexity is represented as **O(N^2)**.
- For an input with the size **N**, algorithms with quadratic complexity execute **N^2** times.
- Quadratic algorithms may have a **nested loop**. Imagine sorting through a big deck of cards and removing all duplicates of each card you come across: you flip over the first card and sort through the whole deck, removing duplicates of that first card. Then you flip over the second card and sort through the whole deck again, looking for duplicates of the second card, etc. This is a quadratic task.

Factorial complexity is represented as **O(N!)**.
- The processing time of factorial algorithms grows exponentially (roughly speaking).
- These algorithms are highly inefficient and should be avoided at all costs.

Comparing Complexity Classes

Let's compare the runtimes of these five classes of complexity above for a variety of increasing inputs:

Input **(N)**	**O(1)**	**O(log(N))**	**O(N)**	**O(N^2)**	**O(N!)**
1	1	1	1	1	1
10	1	3	10	100	3,628,800
40	1	5	40	1,600	8.16e+47
80	1	6	80	6,400	80!
600	1	9	600	360,000	600!

Using Big O Notation

Big O Notation is important for a few reasons:
- It lets you compare and evaluate the efficiency of the algorithms at their disposal and make an informed choice about which to use in a given scenario.
- Example: If you need your algorithm to process information very quickly, you might trade off a higher space complexity for a lower time complexity.
- Big O can be used to categorize problems into different types of complexity.
- Example: Finding a given number in an unsorted array can only be solved in linear time. In a sorted array, though, a given number can be found in logarithmic time.

Job Interviews and Big O Notation

Big O is a popular technical interview subject because interviewers want to make sure that you understand the difference between efficiency and inefficiency so that you're not writing code that takes excessive memory or energy to run.

You may be asked to look at an algorithm and determine its Big O complexity. For these types of problems, ask yourself:
- Does the function have to go through an entire list? If so, there's an **N** in that Big O class somewhere.
- Are there nested loops? That might give you **O(N^2)** (or worse).
- Does the function break the list into smaller chunks? You could have **O(log(N))**.
- Is the amount of work the same, regardless of the size of the dataset? That means it's **O(1)**.