A Study Guide to Big O Notation

1.0 What is an Algorithm?

An algorithm is a set of step-by-step instructions designed to solve a specific problem. It takes an input, follows a defined process, and produces an output.For example, an algorithm to find the largest number in an array named nums would take nums as its input. The instructions could be:

1. Create a variable named maxVal and initialize it to 0.
2. Iterate through each number in the nums array.
3. If a number is greater than maxVal, update maxVal to be that number.After iterating through the entire array, the final value of maxVal is the output.

2.0 The Three Characteristics of a Good Algorithm

Effective algorithms share three essential characteristics that ensure they are reliable and predictable.

2.1 Must Be Deterministic

An algorithm is deterministic if it always produces the exact same output for a given input. No matter how many times you run the algorithm with the same starting data, the result will be identical.

2.2 Must Be Correct

An algorithm must be correct for *any arbitrary valid input* . It is not enough for it to work on just one test case. Consider the algorithm above, which finds the maximum value in an array. If we assume the problem specifies that nums contains only non-negative integers, the algorithm works.However, what if that assumption is violated and nums contains only negative numbers (e.g., -10, -5, -2)? The algorithm would incorrectly return 0 because maxVal was initialized to 0 and no negative number would be greater than it. This reveals a flaw. A truly correct algorithm must account for all valid scenarios, ensuring the maximum value is always selected from the array itself to handle cases like negative numbers.

2.3 Must Halt

A fundamental property of an algorithm is that its instructions are finite and it will eventually finish its task, or "halt." The step-by-step processes in a well-defined algorithm are designed to complete and produce an output, not run indefinitely.

3.0 Introduction to Big O (Computational Complexity)

Big O notation is used to describe the "computational complexity" of an algorithm. It provides a standardized way to measure how an algorithm's resource requirements—specifically time or memory—scale relative to the size of the input, which is commonly denoted as n.

3.1 Time Complexity vs. Space Complexity

- **Time Complexity:** As the input size grows, how much longer does the algorithm take to complete?
- **Space Complexity:** As the input size grows, how much more memory does the algorithm use?

3.2 Core Rules of Calculation

When calculating Big O, two fundamental rules simplify the analysis:

- **Ignore Constants:** Big O notation is not an exact count of operations but a representation of how performance scales. To understand why, imagine two algorithms. For an input of size n = 100, Algorithm A performs 100 operations and Algorithm B performs 500. Now, let's double the input to n = 200. Algorithm A now takes 200 operations, and B takes 1000. While B is slower in absolute terms, both algorithms required double the operations when the input size doubled. This proportional, linear growth is what Big O cares about. Thus, despite the constant factor of 5, both algorithms are simplified to O(n).
- **Focus on Infinity:** Big O analysis is concerned with the dominant term—the part of the complexity that grows the fastest—because as the input size (n) approaches infinity, other terms become insignificant. For example, consider an algorithm with a complexity of $O(n^2 + n)$. Why do we simplify this to $O(n^2)$? Because as n becomes incredibly large, the n^2 term grows so much faster that the value of the n term doesn't make a meaningful difference to the overall growth rate. We focus on the term that dictates the performance at a massive scale.

4.0 Common Time Complexities

This section covers some of the most common time complexities used to classify algorithms.

4.1 O(1) - Constant Time

Described as "constant time," this is the best possible complexity. The runtime of an O(1) algorithm is independent of the input size (n); it takes the same amount of time to complete regardless of how large the input is.

4.2 O(n) - Linear Time

Described as "linear time," the runtime of an O(n) algorithm grows in direct, linear proportion to the size of the input (n). If the input size doubles, the runtime also roughly doubles. A single for loop that iterates through an entire array is a classic example.

```
// Given an integer array "arr" with length n,
for (int num: arr) {
    print(num)
}
```

4.3 O(n²) - Quadratic Time

Described as "quadratic time," the runtime of an O(n²) algorithm grows with the square of the input size. This complexity often occurs when using nested loops that each iterate through the input. If the input size doubles, the runtime increases by a factor of four.

```
// Given an integer array "arr" with length n,
for (int num1: arr) {
    for (int num2: arr) {
        print(num1 + ", " + num2)
    }
}
```

4.4 O(log n) - Logarithmic Time

Described as "logarithmic time," this is an extremely fast and efficient complexity. It is the inverse of exponentiation. In O(log n) algorithms, the problem size is drastically reduced at each step. A hallmark of this complexity, seen in algorithms like binary search, is reducing the search space by 50% in every iteration, allowing it to handle massive inputs with very few operations.