Time Complexity Analysis Cheatsheet

A concise guide to understanding and analyzing the time complexity of algorithms, essential for technical interviews and efficient programming.

Big O Notation Basics

Common Time Complexities

`O(1)` - Constant	Execution time is independent of input size. **Example:** Accessing an element in an array by index.
`O(log n)` - Logarithmic	Execution time increases logarithmically with input size. **Example:** Binary search.
`O(n)` - Linear	Execution time increases linearly with input size. **Example:** Looping through an array.
`O(n log n)` - Loglinear	Execution time is a combination of linear and logarithmic. **Example:** Merge sort, quicksort (average case).
`O(n^2)` - Quadratic	Execution time increases quadratically with input size. **Example:** Nested loops.
`O(2^n)` - Exponential	Execution time doubles with each addition to the input data set. **Example:** Recursive Fibonacci calculation.
`O(n!)` - Factorial	Execution time grows factorially with input size. **Example:** Traveling Salesman Problem (brute force).

Understanding Big O

Big O notation describes the **upper bound** of an algorithm's time complexity. It focuses on the worst-case scenario and ignores constant factors and lower-order terms.

When analyzing algorithms, we care about how the execution time grows as the input size increases. Big O helps us compare the scalability of different algorithms.

Analyzing Code for Time Complexity

Basic Operations

Arithmetic Operations (+, -, *, /)	`O(1)`
Variable Assignment	`O(1)`
Array Indexing	`O(1)`

Control Structures

Simple `for` loop (iterating `n` times)	`O(n)`
Nested `for` loops (iterating `n` times each)	`O(n^2)`
`while` loop (dependent on input size)	Determined by the condition. Could be `O(n)`, `O(log n)`, etc.
`if-else` statements	The complexity is determined by the most complex branch.

Function Calls

The time complexity of a function call is the time complexity of the function being called. Be mindful of recursive calls!

Example: If `foo()` has a time complexity of `O(n)`, then calling `foo()` in your code adds `O(n)` to the overall complexity.

Data Structures and Time Complexity

Common Data Structure Operations

Array Access (by index)	`O(1)`
Array Search (unsorted)	`O(n)`
Sorted Array Search (Binary Search)	`O(log n)`
Linked List Access (by index)	`O(n)`
Hash Table Insertion/Deletion/Access (average case)	`O(1)`
Binary Search Tree Insertion/Deletion/Search (average case)	`O(log n)`
Heap (min/max) Insertion/Deletion/Access	`O(log n)`

Tips and Best Practices

General Advice

Always consider the **worst-case scenario** when determining time complexity.
Ignore constant factors and lower-order terms. `O(2n)` is simplified to `O(n)` and `O(n^2 + n)` is simplified to `O(n^2)`.
Understand the underlying data structures and algorithms being used. This is crucial for accurate analysis.
Practice analyzing code snippets to improve your ability to quickly determine time complexity.
When asked about time complexity in an interview, explain your reasoning clearly and concisely.

Amortized Analysis

Amortized analysis is a method for analyzing the time complexity of an algorithm that performs a sequence of operations. It averages the time taken over a sequence of operations, even if some operations are very expensive.
Example: Dynamic arrays (like ArrayList in Java or vector in C++) have `O(1)` amortized time complexity for adding elements, even though resizing the array takes `O(n)` time.