Advanced Time and Space Complexity Cheat Sheet

Time Complexity Classes

O(1): Constant Accessing array index, pushing/popping from a stack
O(log n): Logarithmic Binary search
O(n): Linear Linear search, traversing array
O(n log n): Linearithmic Merge sort, quick sort average
O(n^2): Quadratic Bubble sort, selection sort, nested loops
O(2^n): Exponential Recursive Fibonacci
O(n!): Factorial Solving permutations, traveling salesman brute force

Space Complexity Examples

O(1): No extra space in-place sorting
O(n): Extra array, hash table, recursion stack with depth n
O(log n): Recursive binary search call stack

Common Data Structures Time & Space

Array: Access O(1), Insert/Delete O(n), Space O(n)
Stack/Queue: Push/Pop O(1), Space O(n)
Hash Map: Insert/Search/Delete O(1) average, O(n) worst, Space O(n)
Set: Same as hash map
Linked List: Access O(n), Insert/Delete O(1) if node given, Space O(n)
Binary Search Tree: Search/Insert/Delete O(log n) avg, O(n) worst, Space O(n)
Heap: Insert/Delete O(log n), Access max/min O(1), Space O(n)
Trie: Insert/Search O(L), Space O(ALPHABET_SIZE * L)
Graph (Adj. List): Space O(V + E), DFS/BFS O(V + E)

Tips for Analyzing Complexities

1. Loops: Linear (O(n)), Nested = Multiplicative (O(n^2), etc.)
2. Recursive Calls: Use recurrence relations (e.g. T(n) = 2T(n/2) + O(n))
3. Divide and Conquer: Often O(n log n)
4. Hashing: Watch for space complexity and collisions
5. Recursion Stack: Adds implicit space usage
6. Dynamic Programming: Usually O(n) or O(n^2) in time and space unless optimized

Common Pitfalls

- Ignoring recursive stack space
- Assuming hash operations are always O(1)

Advanced Time and Space Complexity Cheat Sheet

- Forgetting amortized cost (e.g., Python list append is O(1) amortized)
- Misjudging input size impact n vs. n^2 vs. 2^n