Algorithmic Problem Solving Cheatsheet

A quick reference guide to algorithmic problem-solving techniques, data structures, and common algorithms, useful for coding interviews and general problem-solving.

Core Concepts & Strategies

Problem Solving Framework

1. Understand the Problem:

- Clarify ambiguities: Ask clarifying questions to fully understand the requirements.
- Identify inputs, outputs, and constraints.
- Edge cases: Consider empty inputs, null values, and large datasets.

2. Design an Algorithm:

- Break down the problem into smaller, manageable subproblems.
- Choose appropriate data structures.
- Consider time and space complexity.
- Explore different approaches (e.g., brute force, divide and conquer, dynamic programming).

3. Implement the Algorithm:

- Write clean, well-documented code.
- Handle edge cases and potential errors.
- Follow coding style guidelines.

4. Test Your Solution:

- Test with a variety of inputs, including edge cases.
- Debug and correct errors.
- Verify that the solution meets the requirements.

5. Analyze and Optimize:

- Analyze time and space complexity.
- Identify bottlenecks.
- Optimize the algorithm for better performance (e.g., using more efficient data structures or algorithms).

Common Algorithmic Techniques

Brute Force	Try all possible solutions. Simple to implement but often inefficient.
Divide and Conquer	Break the problem into smaller subproblems, solve them recursively, and combine the results. (e.g., Merge Sort, Quick Sort)
Dynamic Programming	Solve overlapping subproblems by storing their solutions to avoid recomputation. (e.g., Fibonacci sequence, knapsack problem)
Greedy Algorithms	Make locally optimal choices at each step to find a global optimum. (e.g., Dijkstra's algorithm, Huffman coding)
Backtracking	Explore potential solutions incrementally, abandoning partial solutions when they don't lead to a valid solution. (e.g., N-Queens problem, Sudoku solver)

Essential Data Structures

Arrays

Description	Contiguous block of memory storing elements of the same data type.
Access	O(1) - Random access using index.
Insertion/Deletion	O(n) - Requires shifting elements.
Use Cases	Storing lists of elements, implementing other data structures (e.g., stacks, queues).

Linked Lists

Description	Sequence of nodes, each containing data and a pointer to the next node.
Access	O(n) - Sequential access.
Insertion/Deletion	O(1) - If the node to be deleted or inserted after is known.
Use Cases	Implementing stacks, queues, and representing sequences where frequent insertions/deletions are needed.

Hash Tables

Description	Stores key-value pairs, using a hash function to map keys to indices in an array.
Access	O(1) - Average case. O(n) - Worst case (collisions).
Insertion/Deletion	O(1) - Average case. O(n) - Worst case.
Use Cases	Implementing dictionaries, caching, and frequency counting.

Trees

Description	Hierarchical data structure consisting of nodes connected by edges. Binary trees, binary search trees, and heaps are common types.
Access	O(log n) - For balanced binary search trees.
Insertion/Deletion	O(log n) - For balanced binary search trees.
Use Cases	Representing hierarchical data, searching, sorting, and implementing priority queues.

Common Algorithms

Sorting Algorithms

Bubble Sort	O(n^2) - Compares adjacent elements and swaps them if they are in the wrong order.
Insertion Sort	O(n^2) - Inserts each element into its correct position in the sorted portion of the array.
Merge Sort	O(n log n) - Divides the array into smaller subproblems, sorts them recursively, and merges the results.
Quick Sort	O(n log n) average, O(n^2) worst - Chooses a pivot element and partitions the array around it.
Heap Sort	O(n log n) - Uses a heap data structure to sort the array.

Searching Algorithms

Linear Search	O(n) - Sequentially checks each element in the array until the target is found.
Binary Search	O(log n) - Repeatedly divides the search interval in half. Requires a sorted array.

Graph Algorithms

Breadth-First Search (BFS): Explores the graph level by level, starting from a source node. Uses a queue.
Depth-First Search (DFS): Explores the graph by going as deep as possible along each branch before backtracking. Uses a stack (implicitly through recursion).
Dijkstra's Algorithm: Finds the shortest paths from a source node to all other nodes in a weighted graph.
Bellman-Ford Algorithm: Finds the shortest paths from a source node to all other nodes in a weighted graph, even with negative edge weights.

Complexity Analysis & Optimization

Time Complexity

Describes how the execution time of an algorithm grows as the input size increases.

- **O(1):** Constant time.
- **O(log n):** Logarithmic time.
- **O(n):** Linear time.
- **O(n log n):** Linearithmic time.
- **O(n^2):** Quadratic time.
- **O(2^n):** Exponential time.
- **O(n!):** Factorial time.

Space Complexity

Describes how the memory usage of an algorithm grows as the input size increases.

- **O(1):** Constant space.
- **O(n):** Linear space.
- **O(n^2):** Quadratic space.

Optimization Techniques

Memoization	Storing the results of expensive function calls and reusing them when the same inputs occur again (Dynamic Programming).
Caching	Storing frequently accessed data in a cache for faster retrieval.
Using Appropriate Data Structures	Choosing the right data structure can significantly improve performance (e.g., using a hash table for fast lookups).
Algorithmic Optimization	Replacing a less efficient algorithm with a more efficient one.