Competitive Programming Tips

A cheat sheet filled with tips and tricks to help you succeed in competitive programming contests. Covers problem-solving strategies, common algorithms, and useful coding techniques.

Problem Solving Strategies

Understanding the Problem

Read Carefully: Ensure you fully understand the problem statement, input/output formats, and constraints.

Clarify Ambiguities: If anything is unclear, look for clarifications or examples. Don't make assumptions.

Identify Key Information: Pinpoint the core requirements and constraints that dictate the solution approach.

Test Cases: Create small, medium, and large test cases, including edge cases, to validate your understanding.

Designing an Algorithm

Choose the Right Algorithm: Select an appropriate algorithm based on the problem type and constraints (e.g., dynamic programming, graph algorithms, greedy algorithms).

Time Complexity: Analyze the time complexity of your algorithm to ensure it meets the problem's time limits. Use Big O notation.

Space Complexity: Consider the memory usage of your algorithm, especially for problems with memory constraints.

Pseudocode: Write pseudocode to outline your algorithm before implementing it in code. This helps in clarifying the logic and identifying potential issues.

Implementation Tips

Modular Code: Break down your code into smaller, reusable functions or classes to improve readability and maintainability.

Meaningful Variable Names: Use descriptive variable names to enhance code clarity.

Comments: Add comments to explain complex logic or algorithms. This aids debugging and understanding.

Debugging: Use debugging tools to step through your code and identify errors. Learn to use a debugger effectively.

Common Algorithms & Data Structures

Sorting Algorithms

Quicksort	Efficient sorting algorithm with average time complexity of O(n log n). Watch out for worst case O(n^2). Often implemented using recursion. Good for general-purpose sorting.
Merge Sort	Stable sorting algorithm with guaranteed O(n log n) time complexity. Uses a divide-and-conquer approach. Well-suited for sorting linked lists and external sorting.
Heapsort	Sorting algorithm with O(n log n) time complexity. An in-place algorithm. Useful when memory is limited.

Search Algorithms

Binary Search	Efficient search algorithm for sorted arrays or lists. Has a time complexity of O(log n). Requires data to be pre-sorted.
Breadth-First Search (BFS)	Graph traversal algorithm for finding the shortest path in unweighted graphs. Uses a queue data structure.
Depth-First Search (DFS)	Graph traversal algorithm that explores as far as possible along each branch before backtracking. Uses a stack data structure or recursion.

Dynamic Programming

Memoization: Store the results of expensive function calls and reuse them when the same inputs occur again.

Tabulation: Build a table of results bottom-up, iteratively filling in solutions to subproblems.

Optimal Substructure: An optimal solution can be constructed from optimal solutions of its subproblems.

Overlapping Subproblems: The same subproblems are solved repeatedly, allowing for memoization or tabulation.

Coding Techniques & Optimizations

Input/Output Optimization

Fast I/O: Use optimized I/O routines specific to the programming language to reduce overhead (e.g., `scanf/printf` in C/C++, `BufferedReader/PrintWriter` in Java).

Buffering: Read input in larger chunks to minimize the number of system calls.

Data Structure Selection

Arrays vs. Linked Lists: Choose arrays for fast random access and linked lists for efficient insertion/deletion.

Hash Tables: Use hash tables for fast lookups and insertions. Be mindful of hash collisions.

Trees: Use trees (e.g., binary search trees, AVL trees) for ordered data and efficient searching/insertion/deletion.

Heaps: Use heaps for priority queues and finding minimum/maximum elements.

Bit Manipulation

Bitwise Operators: Use bitwise operators (`&`, `|`, `^`, `~`, `<<`, `>>`) for efficient operations on integers (e.g., checking if a number is a power of 2, setting/clearing bits).

Bitmasks: Use bitmasks to represent sets or subsets of elements.

Loop Optimization

Loop Unrolling: Reduce loop overhead by processing multiple elements in each iteration.

Strength Reduction: Replace expensive operations (e.g., multiplication) with cheaper ones (e.g., addition).

Contest Strategies

Before the Contest

Practice: Solve a variety of problems from different platforms (e.g., LeetCode, Codeforces, HackerRank) to improve your skills and speed.

Familiarize: Get familiar with the contest platform, rules, and allowed resources.

Templates: Prepare code templates for common algorithms and data structures to save time during the contest.

During the Contest

Prioritize Problems: Quickly scan all problems and prioritize them based on difficulty and your strengths.

Time Management: Allocate time for each problem and track your progress. Don't spend too much time on a single problem initially.

Test Thoroughly: Test your code with a variety of test cases, including edge cases, before submitting.

Debug Strategically: If your code fails, use debugging techniques to identify the issue quickly.

After the Contest

Review Solutions: Analyze the official solutions and other participants' code to learn new techniques and improve your understanding.

Practice More: Continue practicing to reinforce your skills and address your weaknesses.