SoS – Advanced Algorithms in Competitive Programming

Summer of Science | IIT Bombay

Hey folks!
So glad you're here — reading this means you've taken that first step toward diving deeper into the wonderful, painful, thrilling, and extremely satisfying world of competitive programming. This SoS is all about *advanced algorithms* — the ones you probably haven't seen in CS213 or CS218, but are crucial if you want to break into the 1900+ zone on Codeforces or simply want to level up your algorithmic thinking.

Now, I know everyone is coming from different levels. Some of you are deep into CP already, while others are just starting to stretch beyond basic C++, DSA. **So instead of giving you a rigid, one-size-fits-all structure, I've built a modular 8-week journey. Feel free to adapt, skip, or combine — this SoS is meant to help you grow, not to chain you down with tasks.**

But yes — if you're doing this for just a certificate or submission, then a midterm report after Week 4 and an end-of-term report after Week 8 will be expected.

🧭 The General Flow (TL;DR)

- **Weeks 1–4: Basics & Intermediate stuff — polish what you know, fill gaps, and explore lighter algorithms.**
- **Weeks 5–8: Hard stuff — advanced data structures, trees, transforms, suffix structures, and one *self-chosen beast* of an algorithm.**

🗓️ Week-wise Breakdown

🔹 Week 1 – Brushing the Basics

Let's start from scratch, or just dust things off.

If it's been a while since you solved problems consistently, this is your week to get back on track. Go through the **first 5–7 chapters of the CP Handbook**. **It's a goldmine**. Cover time complexity, prefix sums, greedy, basic recursion, sorting, binary search, and DP.

If you're super rusty or never really tried DP seriously before — I'd recommend moving DP to Week 2 instead of waiting. You can skip Week 4 in this case.

No pressure. Just solve a few problems each day. 2–3 problems/day → 15+ problems/week — trust me, that builds momentum.

🔹 Weeks 2 & 3 – Graphs & Trees

One of the biggest parts of CP. You can't skip this.

Graphs are everywhere — from classic BFS/DFS to shortest paths to disjoint sets. This is where the fun (and frustration) begins.

Spend these two weeks going through:

- BFS, DFS, Topo Sort
- Dijkstra, Bellman-Ford
- Kruskal's Algorithm + DSU (Disjoint Set Union)
- Directed Graphs, Strongly Connected Components
- LCA (Lowest Common Ancestor), Binary Lifting

⚠️ **Already done with DSA? Then feel free to crunch both weeks into one, or skip and move on faster. For each week you skip you can do one extra advance algorithm.**

🔹 Week 4 – Light Algorithms / Buffer

Some fun stuff. Some scary names. But super useful.

This is a bit of a chill week before the “real” advanced section begins. Pick any of the following and just explore:

- Mo’s Algorithm (sqrt decomposition queries)
- KMP and Z Algorithm (string matching!)
- 2 Pointers & Sliding Window
- Manacher’s Algorithm (palindromic centers – wild)
- Sweep Line Algorithm

You don’t need to do all. Even 2–3 are good. Solve problems, write your own implementations. The idea is to have fun.

🌟 **MIDTERM SUBMISSION:**
Nothing big. Just submit:

- A list of topics you’ve covered. And please mention the need of the algorithm and some simple use-case problem.
- Links to problems you solved
- (Optional) Github repo for your templates
- Explain implementation of any 2-3 algorithms from graph week or week-4 which you find interesting. DON’T give code, explain the working through figures and in your own words. Best way is to start with some small examples.

You’ve come halfway through. Take a breath. You’ve done great.

🚀 Weeks 5–8 – The Advanced League

This is what this SoS was meant for. Some of these are hard. Some are elegant. Some are painful. But all of them are important if you want to grow beyond the 1800-2000 level or wanna target ICPC.

🔸 Week 5 – Fenwick Tree & Segment Tree

Your first real data structure flex.

- **Fenwick Tree** (Point Update + Prefix Query)
- **Segment Tree** (and maybe Lazy Propagation)
- Build them from scratch. Learn how to debug them.
- Pro tip: Create Templates for reuse.

  Solve a few problems to practice, there are advance versions of segment tree, you can feel free to go on them too.

🔸 Week 6 – FFT & NTT

Sounds intimidating. Is intimidating. But magical.

- Learn why polynomial multiplication matters in CP.
- **Fast Fourier Transform (FFT)**
- **Number Theoretic Transform (NTT)** – works on integers modulo prime

Solve a few convolution problems. Watch some YouTube tutorials.
At the end of the week, just knowing how it works puts you ahead of most people.

🔸 Week 7 – Suffix Tree & Ukkonen's Algorithm

Strings are love. Strings are pain.

- Understand Suffix Arrays and LCP first.
- Then dive into Ukkonen's Algorithm — online Suffix Tree in O(n).

It's a challenge, but you'll feel like a wizard after finishing this. Use visuals. Trace examples. Watch visualizations.

🔸 Week 8 – Pick Your Own Adventure

Choose a boss fight.

You've done the core. Now pick any one advanced algorithm that excites you. Learn it. Implement it. Teach it.

Here are some ideas:

- Heavy-Light Decomposition
- Centroid Decomposition
- Suffix Automaton
- Li Chao Tree / Dynamic Convex Hull
- Mo's on Tree
- Bitmask DP / SOS DP
- Fast Zeta / Mobius Transform
- 2D Segment Tree / Wavelet Tree

You can choose this later on. Talk to any of the mentor for algorithm you want to choose in the end. A detailed list of algorithms will be modified with time and added on in the end. The algorithm should be a bit complex and advanced though. Make your own template. Prepare a mini walkthrough. Try 1-2 problems too.

🎓 ENDTERM SUBMISSION:

- Full topic list of what you've covered
- Problems solved (with links)

- Your own explanation (in simple words) for:
    - All advance algos.
    - 2–3 basic ones
- Include your own GitHub repo if you've written templates
- Presentation video (5–10 mins) explaining your chosen advanced algorithm

  Note: practice matters more for basic algorithms as for most of the advance algorithms you can use them sometimes as a black-box, but you should just know how to use them.

  Also, if you are just targeting just intern season and things, you can mostly just go through learning advance algorithms as black-box, just learning how the basic versions of them work and practice more on the regular stuff.

🌐 Resources

(A full shared doc with all algorithms and curated resources will be added on in this doc with time)
For now:

- **CP Handbook (Laaksonen)**
- **CP Algorithms**
- **CSES Problemset**
- **Codeforces**
- **USACO guide**
- **WhatsApp groups (post doubts, share links!)**

❤️ Final Thoughts

I don’t care about who finishes first or who solves the hardest problems. If you’re consistent, honest with yourself, and push even just a bit every day, this SoS will be worth it. Don’t let your impostor syndrome tell you otherwise.

You don’t need to be grandmaster to love algorithms. You just need curiosity, grit, and some good friends to debug with.

Let's learn, suffer, and grow together.
All the best.

---

1. Sorting (Bubble, Insertion, Selection, Merge, Quick)

Bubble Sort: Repeatedly swap adjacent elements if they are in the wrong order. $O(n^2)$.

Insertion Sort: Insert elements into their correct position as in card sorting. $O(n^2)$.

Selection Sort: Repeatedly select the minimum element and place it at the beginning. $O(n^2)$.

Merge Sort: Divide array into halves, recursively sort and merge. O(n log n).

Quick Sort: Pick a pivot, partition the array, and recursively sort. Avg: O(n log n), Worst: $O(n^2)$.

---

2. Binary Search (on values, on answer)

Efficiently find an element or check a condition in a sorted array by repeatedly halving the search space. O(log n).

---

3. Two Pointers

Use two indices to scan through the array to solve problems like subarray sums, pairs with given conditions. Useful for sorted arrays.

---

4. Sliding Window

A technique to maintain a window of elements (subarray) and move it to find optimal solutions like maximum sum subarray.

---

5. Greedy Algorithms

Make the locally optimal choice at each step, aiming for a global optimum. Examples: Activity selection, Huffman coding.

---

6. Prefix Sums / Difference Arrays

Prefix Sums: Precompute sum of elements up to each index to answer range sum queries in O(1).

Difference Arrays: Useful for range updates in O(1).

---

7. Brute Force / Complete Search

Try all possible combinations or configurations. Works when input size is small. Good for baseline solutions.

---

8. Bit Manipulation Basics

Use binary representation to perform operations like AND, OR, XOR, shifts efficiently. Great for subsets, parity checks, etc.

---

9. Bitmasking (Enumeration)

Represent subsets or states using bits and iterate over them. Widely used in DP and combinatorial problems.

---

10. Backtracking

Try all options and backtrack when a condition fails. Useful in puzzles like Sudoku, N-Queens.

---

11. Recursion

Function calling itself with simpler inputs. Essential in divide-and-conquer, tree traversals, DP.

---

12. Modular Arithmetic

Operations under a modulo to avoid overflows. Key in combinatorics and number theory problems.

---

13. Basic Dynamic Programming (0/1 Knapsack, Fibonacci)

Store solutions of subproblems to avoid recomputation. Classic problems: Fibonacci, Knapsack, Coin Change.

---

14. Divide and Conquer

Divide problem into smaller subproblems, solve recursively, and combine results. Ex: Merge Sort, Binary Search.

---

15. Sweep Line Basics

Use a line sweeping over a plane to process events in sorted order. Useful in computational geometry and intervals.

---

16. Simulation

Mimic the exact process described in the problem step-by-step. Important when logic is hard to optimize.

---

17. Sorting with Custom Comparators

Sort using custom logic for pairs, structs, or according to a defined rule. Useful in greedy and scheduling problems.

---

18. Counting Inversions

Use modified merge sort to count pairs (i < j, a[i] > a[j]). Measures how far array is from being sorted.

---

19. Ternary Search (on unimodal functions)

Search for maximum/minimum in a unimodal function by splitting range into three parts. Used in optimization problems.

BFS (Breadth-First Search)

Traverses level by level starting from a node. Great for shortest path in unweighted graphs.

DFS (Depth-First Search)

Goes deep before backtracking. Used for cycle detection, connected components, and topological sort.

Shortest Path (Dijkstra, Bellman-Ford)

- **Dijkstra**: Greedy approach using a priority queue. Works with non-negative weights.
- **Bellman-Ford**: Handles negative weights. Can detect negative cycles.

Floyd-Warshall

All-pairs shortest paths using DP. Simple and elegant but O(n3)O(n^3) complexity.

Topological Sort

Linear ordering of vertices in a DAG. Useful in scheduling, build systems.

Kahn's Algorithm

BFS-based topological sort using in-degree.

Connected Components

- **Undirected**: Use DFS/BFS.
- **Directed**: Use Kosaraju or Tarjan’s algorithms.

Strongly Connected Components (SCC)

- **Kosaraju’s**: Two-pass DFS.
- **Tarjan’s**: DFS + low-link values.

Bridges and Articulation Points

Critical connections and cut vertices in a graph. Found using DFS and low-link values.

Bipartite Checking

DFS or BFS coloring method. If no adjacent nodes share color, graph is bipartite.

Union-Find / DSU

Disjoint Set Union. Helps in Kruskal’s MST and connectivity checks.

Kruskal’s Algorithm

Greedy MST algorithm using edges sorted by weight and DSU.

Prim’s Algorithm

Builds MST starting from a node using a priority queue.

MST with DSU

Kruskal’s algorithm relies on efficient DSU with path compression and union by rank.

0-1 BFS

Special BFS using deque. Optimized for edges with weights 0 or 1.

Dijkstra with PQ, Set

Different container optimizations. PQ is standard; set may avoid reprocessing.

Dijkstra with Early Exit and Cost Functions

Early termination when target is found. Customize cost calculation.

Euler Tour of Graph

DFS-based method for tour that starts and ends at same node (for trees or undirected graphs).

Euler Circuit/Path (Hierholzer's Algorithm)

Constructs Euler path/circuit for graphs with proper degree conditions.

Cycle Detection

- **Directed**: Use visited + recursion stack.
- **Undirected**: Use parent tracking in DFS.

Matching in Bipartite Graph (Hopcroft-Karp)

Fast method to find max matching in bipartite graphs. Uses BFS + DFS layers.

Hungarian Algorithm

Solves the Assignment Problem optimally. Runs in O(n3)O(n^3).

Edmonds-Karp (Ford-Fulkerson)

BFS-based Max Flow algorithm. Clean and simple but not fastest.

Dinic’s Algorithm

Faster max flow using level graphs and blocking flows. Powerful for dense graphs.

Push-Relabel Algorithm

Advanced max flow method. Works well on specific types of graphs.

Minimum Cut (Stoer-Wagner)

Finds global minimum cut in an undirected graph. Runs in O(n3)O(n^3).

Johnson's Algorithm

Finds all-pairs shortest paths even with negative weights (no neg cycles). Uses Bellman-Ford + Dijkstra.

Minimum Cost Maximum Flow

Solves flow problems with additional cost constraint. Uses SPFA/Dijkstra with potentials.

Tree Basics

- **Tree**: Acyclic connected graph with edges and nodes.
- **Rooted Tree**: A tree with a designated root node.
- **Binary Tree**: Each node has at most two children.

DFS on Trees

Used for subtree processing, size/count calculations, and ancestor queries.

Tree Traversals

- **Inorder**: Left → Node → Right
- **Preorder**: Node → Left → Right
- **Postorder**: Left → Right → Node

Binary Lifting

Efficient method to answer LCA and kth ancestor queries in using DP table.

LCA (Lowest Common Ancestor)

- **Binary Lifting**: Preprocess in , queries in .
- **Euler Tour + RMQ**: Transform to range minimum problem.

Diameter of Tree

Longest path in tree. Found using two DFS/BFS runs.

Tree DP

Dynamic programming over trees. Example: DP[node] = combine(DP[child1], DP[child2], ...).

Rerooting DP

Technique to solve problems like maximum path sum for all roots using DP in two passes.

Centroid Decomposition

Decomposes tree into centroid tree for efficient path or subtree queries.

Heavy-Light Decomposition (HLD)

Breaks tree into chains to support path queries and updates efficiently with segment trees.

Fenwick/Segment Trees on Tree (Euler Tour Technique)

Use Euler tour + segment/Fenwick tree for subtree queries like sum/count/min.

Subtree Size, Depth, Parent

Calculated during DFS. Useful for queries and rerooting.

Binary Search on Trees

Use properties like monotonicity across paths to apply binary search on answers.

Tree Hashing

Hashing trees (e.g., subtree structure) for isomorphism checks or map-based caching.

Tree Isomorphism

Check if two trees are structurally the same. Use hashing or canonical forms.

🧮 Number Theory & Combinatorics

GCD, LCM

- **GCD (Greatest Common Divisor)**: Euclidean algorithm, essential for simplifying fractions and number theory problems.
- **LCM (Least Common Multiple)**: Can be calculated using GCD: `lcm(a, b) = a*b / gcd(a, b)`.

Extended Euclidean Algorithm

- Solves `ax + by = gcd(a, b)`.
- Used to find modular inverse when `a` and `m` are coprime.

Modular Inverse

- Find `x` such that `(a * x) % m = 1`.
- Use Extended Euclidean or Fermat’s Little Theorem.

Modular Exponentiation

- Fast exponentiation under modulo. Uses binary exponentiation.
- Key for large powers: `(a^b) % m`.

Sieve of Eratosthenes

- Marks primes in `O(n log log n)`. Foundation for prime generation.

Segmented Sieve

- Efficient sieve for large ranges. Combines classic sieve and segment logic.

Prime Factorization

- **Trial Division**: Up to sqrt(n).
- **Pollard-Rho**: Probabilistic and fast for large numbers.

Euler’s Totient Function

- Counts numbers less than `n` that are coprime with `n`.
- `φ(n) = n * Π (1 - 1/p)` for all prime divisors `p`.

Fermat’s Little Theorem

- If `p` is prime, `a^(p-1) ≡ 1 (mod p)`.
- Basis for fast modular inverse and primality.

Chinese Remainder Theorem (CRT)

- Solves system of congruences when moduli are pairwise coprime.

Lucas Theorem

- Computes `nCr mod p` when `n` and `r` can be large.
- Applies recursively using digits in base `p`.

Modular Combinatorics (nCr mod p)

- Use factorial precomputation, modular inverse, Lucas Theorem.

Inclusion-Exclusion Principle

- Counts union of multiple sets correctly by compensating for overcounting.

Mobius Function / Mobius Transform

- Used in number-theoretic transforms and for inversion formulas.

Fast Zeta Transform

- Aggregates over supersets. Dual to Möbius transform.

Bitmask DP / SOS DP

- Used when DP states are represented by subsets (bitmasks).
- SOS (Sum Over Subsets) helps in aggregating subset information efficiently.

Burnside’s Lemma (Hard)

- Counts distinct arrangements under symmetries.
- Used in group theory and combinatorics.

Fast Walsh-Hadamard Transform (XOR Convolution)

- Performs convolutions under XOR operation efficiently.

Geometry Algorithms

Orientation (CW/CCW)

- Determines turn direction of three points using cross product.

Line Intersection

- Checks if two lines/segments intersect and finds the intersection point.

Point in Polygon (Ray Casting)

- Ray casting or angle summation to determine if a point is inside a polygon.

Convex Hull

- Algorithms: Graham Scan, Andrew’s Monotone Chain.
- Finds minimal convex polygon enclosing given points.

Rotating Calipers

- Used with convex hulls to find farthest pair, max area, etc.

Sweep Line for Closest Pair

- Efficiently finds closest pair of points in O(n log n).

Line Sweep for Intersection Count

- Counts the number of intersections between line segments using event processing.

Minkowski Sum

- Computes the shape resulting from adding two polygons vectorially.

Geometry Hashing

- Used to detect congruent shapes or similar transformations.

Circle-Circle / Line-Circle Intersections

- Analytical geometry to find intersection points.

3D Geometry Basics

- Rare but includes vector math, dot/cross products, planes, and lines in 3D.

KMP Algorithm (Knuth-Morris-Pratt)

Efficient pattern matching algorithm. Preprocesses the pattern to avoid redundant comparisons. Time complexity: O(n + m).

Z Algorithm

Computes Z-array where Z[i] is length of longest substring starting at i that's also a prefix. Used in pattern matching. Time complexity: O(n).

Trie

Prefix tree structure for storing strings. Allows fast lookup, insert, delete. Useful in autocomplete, dictionary matching.

- **With count**: Stores frequency of strings.
- **With map**: Uses hashmap instead of fixed alphabet size.

Suffix Array (O(n log n))

Stores starting indices of sorted suffixes. Powerful tool for substring search, LCP, pattern matching.

LCP Array (Kasai’s Algorithm)

Stores length of longest common prefix between consecutive suffixes in suffix array. Built in O(n).

Suffix Automaton

DFA representing all substrings of a string. Useful for counting distinct substrings, finding occurrences.

Rabin-Karp (Rolling Hashing)

Hashing-based pattern search. Can find multiple patterns. Collision-handling is needed for correctness.

Manacher’s Algorithm

Finds longest palindromic substring in linear time. Center-expansion based.

Aho-Corasick

Multi-pattern search in a text. Constructs trie + failure links for all patterns. Time: O(n + total pattern length).

Ukkonen’s Algorithm (Suffix Tree)

Online linear-time construction of suffix tree. Complex but powerful for advanced string problems.

Hashing

Used to compare strings in constant time after O(n) preprocessing.

- **Double Hashing**: Use two hash functions to reduce collisions.
- **Polynomial Hashing**: Common choice; uses base and mod.

String Compression Techniques

- **Run-Length Encoding (RLE)**: Compress repeated characters (e.g., aaabb → a3b2).

Lyndon Decomposition

Decomposes a string into non-increasing sequence of Lyndon words. Useful in combinatorics.

Duval’s Algorithm

Finds the lex smallest rotation of a string in linear time. Based on Lyndon factorization.

Lexicographically Minimal/Maximal Rotation

Circular rotation that is minimal or maximal in lex order. Useful in string normalization and comparison.

Binary Indexed Tree (Fenwick Tree)

A data structure that provides efficient methods for prefix sums and point updates in O(logn)O(\log n). Easy to implement and very space-efficient.

Point Update + Prefix Sum

Classic BIT usage. Add a value to an index and query prefix sum up to an index.

Range Update + Point Query

Trick: Use two BITs or maintain difference array to allow range updates and get the value at a point.

Segment Tree

More powerful than BIT, supports custom associative functions (min, max, sum, gcd, etc.) in O(logn)O(\log n).

With Lazy Propagation

Extends Segment Tree to handle range updates efficiently. Pushes updates only when needed.

Range Min/Max/Sum Queries

Most common segment tree queries. You can combine min/max trees with position tracking.

Range Updates (add, set, etc.)

Use lazy propagation to support operations like range addition or setting all values in a range.

Sparse Table (Immutable RMQ)

For static arrays (no updates), supports fast range min/max/gcd queries in O(1)O(1) time after O(nlogn)O(n \log n) preprocessing.

Mo’s Algorithm

Offline query algorithm for range queries (like frequency/count queries). Useful when updates are not allowed. Works in O((n+q)n)O((n+q)\sqrt{n}).

Sqrt Decomposition

Divide array into blocks of size n\sqrt{n}. Allows range queries and updates in around O(n)O(\sqrt{n}) time. Good when segment tree is overkill.

Merge Sort Tree

A segment tree where each node stores a sorted list of elements. Used for answering queries like count of elements < k in range.

Persistent Segment Tree

Allows access to previous versions of the array. Enables rollback and historical queries. Great for time-traveling arrays.

Segment Tree Beats

Handles difficult problems like range max + range add. Advanced version that breaks when lazy propagation alone isn't enough.

Li Chao Tree (Line Container)

Special structure for maintaining a set of lines and querying maximum/minimum at given points. Used in dynamic convex hull tricks.

Treap / Randomized BST

Balanced BST that maintains a heap property. Can be used for dynamic order statistics and range splitting/merging.

K-D Tree

Multi-dimensional BST. Used in geometric problems, like nearest neighbor or range counting in 2D/3D space.

Wavelet Tree

Supports advanced range queries over sequences, like finding kth smallest in range or frequency of elements.

BIT/SegTree on Tree (Euler Tour)

Convert tree to array using Euler Tour and apply BIT/SegTree for subtree/range queries.

Dynamic Segment Trees (Coordinate Compression)

For very large or sparse ranges. Build segment trees only for accessed nodes using maps or pointers.

Ordered Set (PBDS in C++)

Special set that supports finding k-th largest/smallest and order-of-key queries. Part of C++ Policy-Based DS.

Other Algorithms

Some rare, some powerful. Boss fights of CP.

FFT (Fast Fourier Transform)

Efficient method for polynomial multiplication in O(nlogn)O(n \log n). Often used in convolution problems.

NTT (Number Theoretic Transform)

Integer-only version of FFT. Works with modular arithmetic for competitive programming.

Gauss Elimination

Solves systems of linear equations. Works in O(n3)O(n^3). Foundation of matrix algebra.

Berlekamp-Massey Algorithm

Finds the shortest linear recurrence for a sequence. Works in O(n2)O(n^2).

Matrix Exponentiation

Solves linear recurrences in logarithmic time. Useful in Fibonacci-type problems.

Convex Hull Trick (CHT)

Dynamic programming optimization for monotonic lines and queries. Reduces DP complexity.

Online CHT

CHT variant that supports queries in arbitrary order. Typically implemented with balanced BSTs.

Implicit Treap / Rope

Self-balancing BSTs used for sequence manipulation. Supports split, merge, insert in log time.

Sqrt Tree

Advanced data structure for fast range queries. Combines ideas from block decomposition and recursion.

Suffix Tree (Ukkonen’s Algorithm)

Linear time construction of suffix trees. Used in substring queries, pattern matching.

Hash Map Optimizations

Use custom hash functions (e.g., `custom_hash`) to reduce collisions and avoid TLE in C++.

Heavy-Light with Segment Tree

Combine HLD with segtree for efficient path and subtree queries in trees.

Simulated Annealing

Heuristic optimization method. Used in approximate solutions for NP-hard problems.

Randomized Algorithms

E.g., Treap, Pollard-Rho. Use randomness to simplify logic or avoid worst-case scenarios.

Meet in the Middle

Divide the input in two halves. Useful in exponential problems like subset sum, knapsack.

Divide & Conquer Optimization

Optimizes DP with quadrangle inequality. E.g., Knuth Optimization, Aliens Trick.

Ternary Search

Used to find minimum/maximum of unimodal functions in log time.

Parallel Binary Search

Use binary search in parallel for multiple queries. Good when queries depend on values.

Dynamic Programming Optimization Techniques

Convex Hull Trick

For DP recurrence of form: `dp[i] = min(dp[j] + m[j] * x[i])`. Efficiently keeps best lines.

Divide and Conquer DP

Applies when `opt(i) <= opt(i+1)`. Cuts DP table recursively to reduce time.

Bitmask DP

Used for TSP, subset DP. Stores state using bits. Watch memory usage.

Tree DP

DP on trees. Subtree DP, rerooting, path queries, etc.

Matrix Chain Multiplication

Parenthesization optimization. Uses DP over intervals.

Integer Partition DP

Count number of ways to write integer as sum of others. Classic DP.

🤖 Interactive Problem Techniques

Binary Guessing

Search range by making queries and updating bounds. Common in interactive judge problems.

General Tips

- Minimize queries.
- Always flush output after each query.
- Follow the protocol strictly.
