General tips

DSU (Disjoint Set Union / Union-Find)

Use when

- Connectivity under **only edge additions**: “are u and v connected?”, “how many components?”, **cycle detection** in undirected graphs.
- **Kruskal’s** MST.
- Offline tricks: dynamic connectivity (add/remove) via **divide & conquer on time** or **rollback DSU**.

Don’t use when

- You need shortest paths or counts along paths; DSU has no notion of distance or order.

Segment tree (vs Fenwick/BIT)

Use SEG when

- Range queries **and** (possibly) range updates with custom ops: min/max/sum, **k-th** element, order statistics, “first index where prefix ≥ X”.
- You need **lazy propagation** (range add/assign + range query).
- Coordinate-compressed indices, dynamic point updates.

Use Fenwick when

- Only **prefix sums** / **point update** (or range update + point query via diff trick). It’s simpler, faster constant factors.

Avoid tree structures if

- All queries are offline and suit **Mo’s algorithm** (range add/remove, order-insensitive answers).

Bitset

Use when

- 0/1 **Knapsack** or subset-sum: `dp |= (dp << w)`; shifts are vectorized (≈64× speedup).
- Massive **set intersections** (e.g., adjacency as bitsets ➔ clique checks, triangle counting) in $O(n^3/64)$.
- **Reachability** on smallish `n` (≤ 4000) via bitset DP / transitive closure.
- Track letters/features in constant memory: `bitset<26>` etc.

Avoid when

- The universe is huge (needs `O(U)` bits). Then use hashing/sets.

Graph algorithms — quick picker

BFS

- Unweighted or all edges weight **1**. Also **multi-source BFS** (push all sources at dist 0).
- Shortest path in **grid/maze**; level structure tasks.
- **0-1 BFS** for edges with weights {0,1}.

DFS

- **Cycle detection**, **toposort** (DAG), **connected components**, **bridges/articulation points**, **SCC** (with Tarjan/Kosaraju).
- “Is there *any* path?” style reachability.

Dijkstra (with heap)

- Non-negative weights, single-source shortest path on **sparse** graphs.
- Don’t use if negative edge exists (even once).

Bellman–Ford

- There are **negative edges**; need neg-cycle detection reachable from source.
- Graph small/moderate (`n·m` fits). Avoid SPFA in contests unless input is friendly; worst-case is bad.

Floyd–Warshall

- **All-pairs shortest paths**, `n ≤ ~400`, or need to answer many (u,v) queries quickly.
- Also handy for transitive closure / **bitset-optimized** variants.

Prim vs Kruskal (MST)

- **Kruskal + DSU**: simple, great when you can sort edges once (sparse).
- **Prim (heap)**: good on very **dense** graphs (`m ~ n²`) or when graph is given as adjacency matrix.

Euler trail/circuit (Hierholzer)

- You must traverse **each edge exactly once**.
- **Undirected**: circuit if all degrees even and connected; trail if exactly two odd degrees.
- **Directed**: circuit if in-deg == out-deg for all and strongly connected in underlying sense; trail if exactly one node has out-deg = in-deg + 1 (start) and one has in-deg = out-deg + 1 (end).

Flow / Matching

- You see phrases like: "at most one", "capacity", "assign", "route", "disjoint paths", "cut".
- **Dinic** (or Push-Relabel) for max flow; great on unit capacities / bipartite graphs.
- **Hopcroft–Karp** for **bipartite matching** specifically (faster, simpler than full flow).
- Need costs? Use **Min-Cost Max-Flow** (small graphs) or assignment via **Hungarian** (dense, square cost matrix).
- Many problems = **build a network** (source/sink, split nodes for capacity constraints, add edges with capacities).

Greedy vs DP — spotting cues

Greedy (sort + local choice + proof by exchange)

- Intervals: **earliest finishing time**, **min arrows/points to cover**, **activity selection**.
- Scheduling with deadlines/penalties: process by **deadline**, keep best with a **priority queue**.
- "Make lexicographically/sum-wise best" with a natural **monotone** choice: e.g., **fractional knapsack** (but not 0/1!).
- **Matroid-like** problems: independence + greedy by weight works.
- If a local choice never hurts future options (you can argue an **exchange**), it's greedy.

DP (overlapping subproblems + compact state)

- "Pick a subset/sequence to maximize/minimize" with **hard constraints** (0/1 knapsack, partitions, edit distance).
- Optimal solution depends on earlier **state**, not only local choice (e.g., LIS, path counting with obstacles).
- You can define `f(i, ...)` over prefix/index/range, or `f[mask]` over subsets.
- DAG shortest path = DP on topological order.

When you're unsure

- Try greedy on small counterexamples; if you can cook one quickly, it's DP.
- If state space is **tiny** (≤ 1e6 states) and transitions are clear ➔ DP is safe.
- If there's a **monotone predicate** ("can we do X with budget B?"), consider **binary search on the answer** + feasibility check (greedy/flow/DSU).

Other high-yield patterns (very common in ICPC)

- **Two pointers / sliding window**: sorted or non-negative weights; "longest subarray with property".
- **Prefix sums / difference arrays / 2D prefix**: range sums, add-on-range.

- **Coordinate compression**: large values, few distinct coordinates.
- **Binary lifting / LCA**: tree queries (k-th ancestor, path queries with segment tree on Euler tour).
- **Meet-in-the-middle**: subset sums with `n ≈ 40`.
- **Hashing/maps**: count pairs/triples, de-dup, frequency tricks.
- **SCC / Toposort**: constraints like "A before B", "if X then Y" (2-SAT ➔ SCC).

Quick "tells" from statements

- "Every edge used exactly once" → **Euler**.
- "Each vertex exactly once" in general graph → NP-hard (don't brute unless `n` small). On DAG/tree it might be DP.
- "Minimum number of ... to connect / separate" → **cut/flow** or **MST**.
- "Add edges only, answer connectivity queries" → **DSU** (possibly offline).
- "Many range queries/point updates" → **Fenwick/Segment tree** (or **Mo's** offline).
- "Costs but also capacities/limits" → **Min-cost flow** / **assignment**.
- "Unweighted shortest path / few weights" → **BFS / 0-1 BFS**.
- "Negative edges / detect neg cycle" → **Bellman–Ford**.
- "All pairs, small n" → **Floyd–Warshall** (consider bitset speedups).

Tiny checklist during contest

1. Read all problems; tag each with: graph? ranges? strings? DP? geometry?
2. Glance at constraints; pick the **tightest** one and match to a complexity.
3. Try the simplest candidate (BFS/greedy) first; kill with a counterexample or move on.
4. For DP: define **state**, **transition**, **base**, **order**, **memory**.

5. For graph: check special cases (disconnected, multiple components, negative edges).
6. Always test on **toy cases** you invent to break your approach.

If you want, I can turn this into a one-page printable crib sheet or tailor examples for each category you struggle with most.