EQUIPPD · FREE RESOURCE

The Technical Interview Study Plan

An honest map of what to study, for how long, and for which companies.

Includes a self-assessment, a twelve-week plan,
and a realistic target list framework.

Built for the Equippd community · equippd.io

Start by being honest about where you are

Most technical interview prep fails for one of two reasons: people study for companies far outside their current range and get demoralized, or they aim too low and never stretch. Both are fixable by assessing honestly first.

Rate yourself against this scale. Be accurate rather than generous — the plan only works if the starting point is real.

Tier	Where you are now	Weekly hours to move up	Realistic targets
1	Comfortable with syntax; easy problems are still slow	8–10	IT departments at non-tech companies, local firms, support engineering
2	Easy problems reliably; some mediums with hints	6–8	Large non-tech employers, banks, retail tech, government contractors
3	Mediums consistently; ~20+ problems solved	5–7	Mid-size tech, financial institutions, enterprise software
4	Mediums quickly, some hards; ~30+ solved	4–6	Well-known tech companies, strong startups
5	Hards regularly; ~50+ solved; can explain tradeoffs cold	3–5 to maintain	The most competitive tech employers

Apply across all three bands

Build your list as roughly one third stretch, one third likely, one third comfortable. People who only apply to stretch companies end up with zero offers and no leverage. People who only apply to comfortable ones leave money on the table. You want a real offer in hand before you negotiate with anyone.

What actually gets tested

The surface area is smaller than it feels. The overwhelming majority of questions draw from this list:

Area	What to know cold	Priority
Arrays & strings	Two pointers, sliding window, in-place manipulation	Critical
Hash maps	Frequency counting, lookup-to-O(1) conversions, sets	Critical
Big-O	Time and space for everything you write, said out loud	Critical
Trees	Traversals (DFS/BFS), binary search trees, recursion	High
Linked lists	Reversal, cycle detection, fast/slow pointers	High
Sorting & searching	Binary search and its variants; when sorting is the unlock	High
Stacks & queues	Matching problems, monotonic stack, BFS queues	High
Graphs	Adjacency representation, BFS/DFS, cycle detection	Medium
Heaps	Top-K problems, priority queues	Medium
Dynamic programming	Memoization first, then tabulation. Classic patterns	Medium
System design	Only for mid/senior roles — ignore it for internships	Role-dependent

Notice that the first three are marked Critical. A large share of screens never go past arrays, strings, and hash maps. Master those before touching dynamic programming.

The twelve-week plan

This assumes roughly six hours a week. Scale the problem counts, not the structure, if you have more or less time.

Weeks 1–3 — Foundations

- Pick **one** language and stop switching. Whichever you are fastest in wins.
- Re-learn your language's core collections cold: arrays, maps, sets, their built-in methods and their complexity.
- Solve 3–4 easy problems per week. Arrays, strings, hash maps only.
- After each problem, write one sentence on the pattern it used.

Weeks 4–7 — Patterns

- Move to mediums. Two per week, fully understood, beats six half-copied.
- Work by **pattern**, not randomly: a week of two-pointer, a week of sliding window, a week of tree traversal.
- Start a mistake log — every problem you could not solve, and why.
- Begin saying your reasoning out loud while you code. This is a separate skill and it is scored.

Weeks 8–10 — Simulation

- Do timed problems: 35 minutes, no hints, no lookups.
- Run at least four mock interviews with a real human.
- Practice on a whiteboard or plain text editor — no autocomplete, no syntax highlighting.
- Re-solve the problems in your mistake log from scratch.

Weeks 11–12 — Sharpen

- Review patterns rather than grinding new problems.
- Re-solve your ten hardest problems cold.
- Prepare your project walkthroughs — you will be asked to explain your own work.
- Sleep. A rested brain outperforms an extra twenty problems.

> **The single highest-leverage habit**
>
> Talk while you solve. Interviewers cannot read your mind, and a correct answer delivered in total silence scores worse than a partially complete one with clear reasoning. Narrate: what you notice, what you are ruling out, what tradeoff you are making. Practice this from week four, not the night before.

How to actually solve one in the room

1. **Restate the problem** in your own words and confirm you have it right.
2. **Ask about constraints and edge cases.** Empty input? Duplicates? Size limits? Negative numbers?
3. **Work one small example by hand** before writing any code.
4. **State a brute-force approach** and its complexity. This banks a working answer immediately.
5. **Propose an optimization** and say why it is better before you write it.
6. **Write the code**, narrating as you go.
7. **Trace your own code** against the example. Find your bugs before they do.
8. **State final time and space complexity** unprompted.

> **If you get stuck**
>
> Say what you have ruled out and why. “I considered sorting first, but that costs n log n and I think we can do this in one pass with a hash map — let me try that” is a strong signal even if the attempt fails. Silence is the only genuinely bad option.

Common ways people lose

- Grinding problem volume without ever reviewing mistakes.
- Reading solutions after ten minutes instead of sitting in the discomfort for thirty.
- Practicing only in an IDE with autocomplete, then freezing in a plain editor.
- Never practicing out loud, then being unable to speak and think simultaneously.
- Studying dynamic programming while still shaky on hash maps.
- Applying to twelve companies of identical difficulty and having no fallback.
- Neglecting the behavioral round, which is often what actually decides between two similar candidates.

*Pair this guide with **The Behavioral Interview Playbook**. Technical rounds get you considered; behavioral rounds frequently decide the offer.*