Remembering all the solutions to **LeetCode 150** problems—or even a larger set—might seem overwhelming, but **it is definitely possible** with the right strategies! Instead of focusing on memorizing each solution verbatim, the goal should be to **understand the core concepts** and **patterns** behind the problems. Once you understand these patterns, you'll be able to **apply them** to various problems without needing to recall every detail.

Here are some strategies to help you remember and internalize the key concepts of LeetCode problems:

**1. Focus on Patterns, Not Solutions**

* Many problems on LeetCode are based on a few **core algorithms** and **data structures**. If you can master the most common ones, you'll be able to solve many problems without memorizing specific solutions.

**Key patterns to focus on**:

* **Two-pointer technique**: Common in problems involving arrays, linked lists, etc. (e.g., Container With Most Water, 3Sum).
* **Sliding window**: Used for subarray problems (e.g., Longest Substring Without Repeating Characters).
* **Dynamic programming (DP)**: Used for optimization problems (e.g., Longest Common Subsequence, Knapsack Problem).
* **Backtracking**: Used for problems where you try all possibilities (e.g., N-Queens, Sudoku Solver).
* **Greedy algorithms**: Where you make the locally optimal choice at each step (e.g., Jump Game, Interval Scheduling).
* **Depth-First Search (DFS) & Breadth-First Search (BFS)**: For tree and graph traversal (e.g., Binary Tree Level Order Traversal, Word Ladder).
* **Binary Search**: Used for sorted arrays or search space problems (e.g., Find Minimum in Rotated Sorted Array).

**Remember the Big Picture**:

* **If a problem involves sorting**, think about using **binary search**.
* **If it's about partitioning or grouping data**, think about using **two pointers** or **sliding windows**.
* **If you need to optimize decisions**, think about using a **greedy algorithm** or **dynamic programming**.

Understanding these patterns will help you **apply the right technique** to solve many problems without needing to memorize the specific solution each time.

**2. Learn the Common Data Structures**

Many problems on LeetCode rely on the use of a small set of **core data structures**:

* **Arrays & Strings**
* **Hash Tables** (for efficient lookup, e.g., Two Sum, Group Anagrams)
* **Linked Lists** (e.g., Reverse Linked List, Detect Cycle in Linked List)
* **Stacks & Queues** (for problems like **valid parentheses**, **next greater element**)
* **Heaps/Priority Queues** (e.g., Top K Frequent Elements)
* **Graphs & Trees** (DFS, BFS for traversals, shortest paths, etc.)
* **Tries** (e.g., for **word search** problems or prefix matching)

If you know how to implement and use these structures efficiently, you’ll be able to apply them in most problems. Focus on understanding how these data structures work and their time complexity (e.g., O(1) for hash lookups, O(log n) for heaps, O(n) for BFS/DFS).

**3. Master the Classic Problem Categories**

LeetCode problems can generally be categorized into a few broad types. Once you know how to solve one problem from each category, you’ll be able to recognize the pattern and approach other similar problems with ease.

**Categories and Key Problem Types:**

* **Array Problems**: Subarrays, sliding window, two-pointer technique.
  + Example: Max Subarray Sum, Two Sum.
* **Linked List Problems**: Reversal, cycle detection, merging.
  + Example: Reverse Linked List, Detect Cycle in Linked List.
* **Tree Problems**: Binary tree traversal, path finding, lowest common ancestor.
  + Example: Binary Tree Level Order Traversal, Maximum Depth of Binary Tree.
* **Dynamic Programming**: Optimal substructure, overlapping subproblems.
  + Example: Fibonacci Sequence, Longest Common Subsequence.
* **Backtracking**: Recursive exploration of all possibilities.
  + Example: N-Queens, Permutations.
* **Greedy Algorithms**: Local optimization for global optimization.
  + Example: Jump Game, Interval Scheduling Maximization.
* **Graph Problems**: BFS/DFS for traversal, shortest path, topological sorting.
  + Example: Word Ladder, Clone Graph.
* **String Matching**: Substrings, pattern matching, KMP.
  + Example: Longest Substring Without Repeating Characters, Valid Palindrome.
* **Sorting and Searching**: Binary search, sorting algorithms.
  + Example: Find the Peak Element, Search in Rotated Sorted Array.

When you recognize the problem category, you'll know which algorithm or data structure to apply. Over time, this knowledge becomes internalized.

**4. Practice with a Purpose**

**The best way to memorize solutions** is through **repeated practice** and **active problem-solving**. Don’t just read the solution—**solve the problem yourself**. After solving a problem, make sure to:

* **Write down the key takeaways** from the solution (e.g., "I need to use a sliding window for substring problems").
* **Create a personal cheat sheet** of common patterns and solutions that you can quickly reference when you encounter a similar problem.
* **Revisit the problem after a few days** to see if you can solve it without looking at the solution. This helps reinforce the concepts.

**Focus on the concepts**, not on memorizing the specific solution code. For example:

* If you know **dynamic programming**, you’ll understand how to solve problems like **knapsack**, **coin change**, **house robber**, etc., even if they vary slightly in details.
* If you’ve solved a problem using **DFS**, you’ll easily recognize how to solve other DFS problems (like **pathfinding** in grids or trees).

**5. Build Mental Models for Common Problem Types**

Developing **mental models** for problem-solving is a great way to retain solutions.

**Example Mental Models:**

* **For "Two Sum" type problems**: Use a **hashmap** to store seen elements as you iterate through the array.
* **For "Pathfinding" problems in graphs**: Use **BFS** for the shortest path or **DFS** for exhaustive exploration.
* **For optimization**: Think of **greedy algorithms** when you can make local optimal choices (e.g., minimizing a cost, maximizing a profit).

Write down these mental models and refer to them often. Over time, they will help you decide which approach to use for various types of problems.

**6. Understand the Why Behind the Solution**

For each problem you solve, **ask yourself why the solution works** and what is the core idea. If you understand the **why**, you’re less likely to forget the solution.

* For example, **why does the sliding window technique** work for subarray problems? It reduces the need to recheck the entire subarray by maintaining a dynamic range that adjusts as we slide.
* **Why does dynamic programming work**? It breaks problems into overlapping subproblems and stores intermediate results to avoid redundant work.

Understanding **why** each algorithm works will help you internalize the logic and apply it to different problems.

**7. Use Spaced Repetition and Revision**

Use a **spaced repetition system (SRS)** or a **flashcard system** (like Anki) to periodically review problems you’ve solved. This method helps you **retain** solutions over time. You can create flashcards for:

* Problem names and categories.
* Key algorithmic patterns (e.g., "Sliding Window," "Greedy").
* Time and space complexity.

Spaced repetition optimizes your learning by reviewing concepts at intervals, reinforcing your memory.

**8. Participate in LeetCode Contests**

Participating in LeetCode contests will give you exposure to a wide range of problems under time constraints. Even if you don’t solve all the problems, the exposure to **new problem types** and **techniques** will reinforce your learning and help you remember solutions.

**In Conclusion:**

It’s not necessary to **memorize** each solution to every problem on LeetCode. Instead, focus on understanding the **core principles**, **patterns**, and **data structures** that are frequently used. Once you’ve mastered these, you can **recognize the approach** needed for a wide variety of problems.

By practicing **repeatedly**, focusing on **patterns**, and using **spaced repetition**, you’ll be able to recall solutions quickly and apply the right techniques efficiently!