LeetCode was HARD until I Learned these 15 Patterns

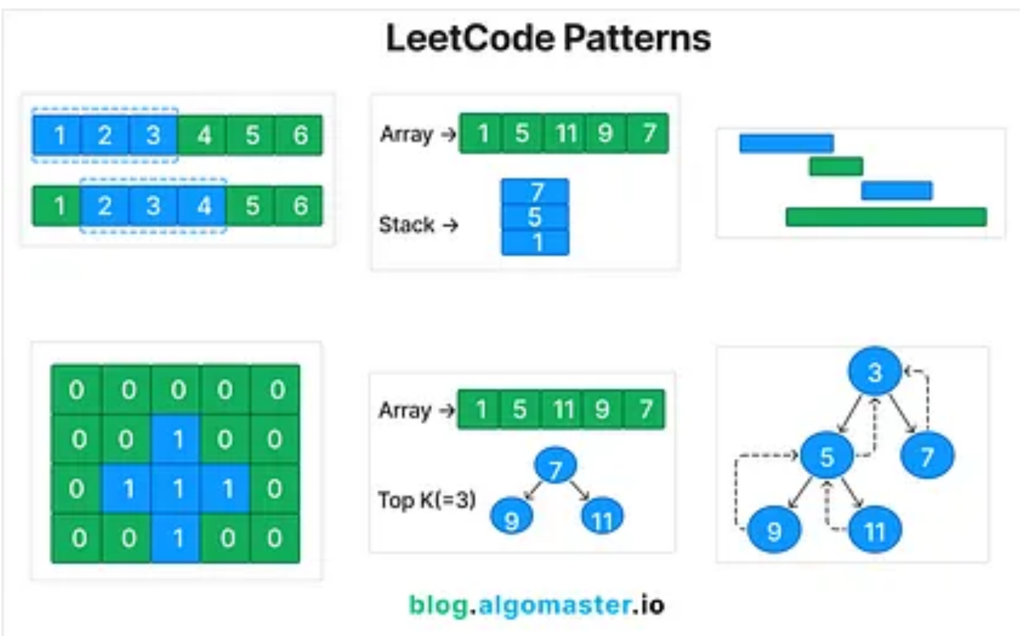
#21 - Patterns to master LeetCode

Ashish Pratap Singh

Having solved more than **1500 LeetCode problems,** if there is one thing I have learned, it's this:

> LeetCode is **less** about the number of problems you have solved and **more** about how many **patterns** you know.

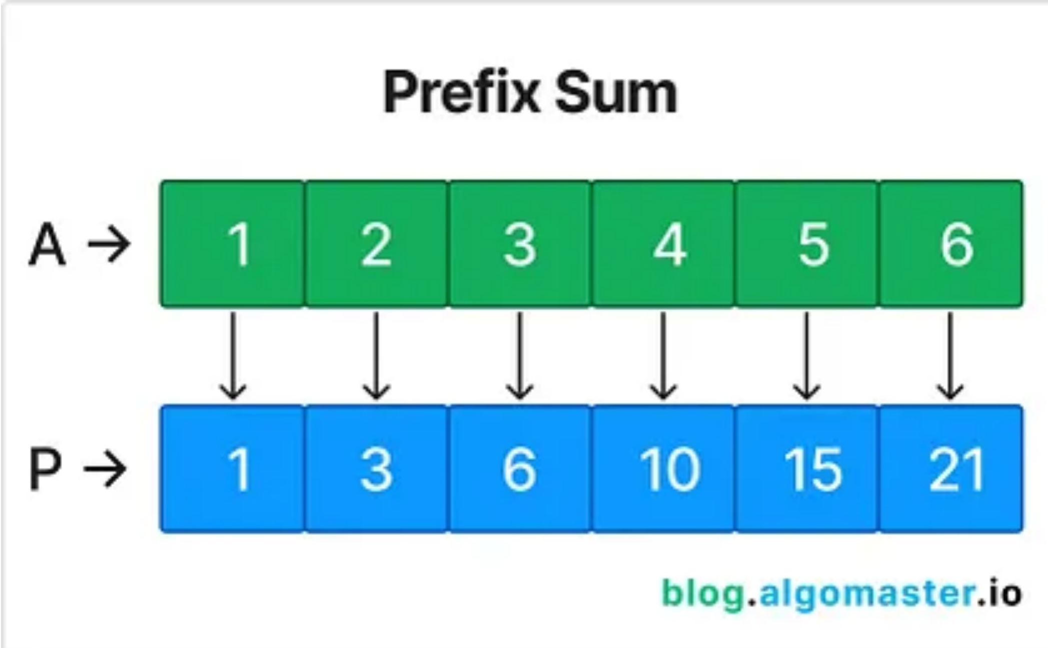
Learning patterns enables you to solve a wide variety of problems in lesser time and helps you quickly identify the right approach to a problem you have never seen before.



In this article, I'll walk you through the **15 most important** patterns I learned that made my LeetCode journey lot less painful.

I'll share when to use each pattern along with a sample problem and provide links to LeetCode problems you can practice to learn these patterns better.

1. Prefix Sum



Prefix Sum involves **preprocessing** an array to create a new array where each element at index `i` represents the sum of the array from the start up to `i`. This allows for efficient **sum queries on subarrays**.

Use this pattern when you need to perform multiple sum queries on a subarray or need to calculate cumulative sums.

Sample Problem:

Given an array `nums`, answer multiple queries about the sum of elements within a specific range `[i, j]`.

Example:

- **Input:** `nums = [1, 2, 3, 4, 5, 6]`, `i = 1`, `j = 3`
- **Output:** `9`

Explanation:

1. Preprocess the array `A` to create a prefix sum array: `P = [1, 3, 6, 10,`

`15, 21]`.

2. To find the sum between indices `i` and `j`, use the formula: `P[j] - P[i-1]`.

LeetCode Problems:

1. [Range Sum Query - Immutable (LeetCode #303)]
2. [Contiguous Array (LeetCode #525)]
3. [Subarray Sum Equals K (LeetCode #560)]

2. Two Pointers

The Two Pointers pattern involves using two pointers to iterate through an array or list, often used to find **pairs or elements** that meet specific criteria.

Use this pattern when dealing with sorted arrays or lists where you need to find pairs that satisfy a specific condition.

Sample Problem:

Find two numbers in a sorted array that add up to a target value.

Example:

- Input: `nums = [1, 2, 3, 4, 6]`, `target = 6`
- Output: `[1, 3]`

Explanation:

1. Initialize two pointers, one at the start (`left`) and one at the end (`right`) of the array.
2. Check the sum of the elements at the two pointers.
3. If the sum equals the target, return the indices.
4. If the sum is less than the target, move the left pointer to the right.
5. If the sum is greater than the target, move the right pointer to the left.

LeetCode Problems:

1. Two Sum II - Input Array is Sorted (LeetCode #167)
2. 3Sum (LeetCode #15)
3. Container With Most Water (LeetCode #11)

3. Sliding Window

The Sliding Window pattern is used to find a subarray or substring that satisfies a specific condition, optimizing the time complexity by maintaining a window of elements.

Use this pattern when dealing with problems involving contiguous subarrays or substrings.

Sample Problem:

Find the maximum sum of a subarray of size `k`.

Example:

- Input: `nums = [2, 1, 5, 1, 3, 2]`, `k = 3`
- Output: `9`

Explanation:

1. Start with the sum of the first `k` elements.

2. Slide the window one element at a time, subtracting the element that goes out of the window and adding the new element.
3. Keep track of the maximum sum encountered.

LeetCode Problems:

1. Maximum Average Subarray I (LeetCode #643)
2. Longest Substring Without Repeating Characters (LeetCode #3)
3. Minimum Window Substring (LeetCode #76)

4. Fast & Slow Pointers

The Fast & Slow Pointers (Tortoise and Hare) pattern is used to detect cycles in linked lists and other similar structures.

Sample Problem:

Detect if a linked list has a cycle.

Explanation:

1. Initialize two pointers, one moving one step at a time (**slow**) and the

other moving two steps at a time (**fast**).

2. If there is a cycle, the fast pointer will eventually meet the slow pointer.

3. If the fast pointer reaches the end of the list, there is no cycle.

LeetCode Problems:

1. Linked List Cycle (LeetCode #141)

2. Happy Number (LeetCode #202)

3. Find the Duplicate Number (LeetCode #287)

5. LinkedList In-place Reversal

The In-place Reversal of a LinkedList pattern reverses parts of a linked list without using extra space.

Use this pattern when you need to reverse sections of a linked list.

Sample Problem:

Reverse a sublist of a linked list from position `m` to `n`.

Example:

- **Input:** `head = [1, 2, 3, 4, 5]`, `m = 2`, `n = 4`
- **Output:** `[1, 4, 3, 2, 5]`

Explanation:

1. Identify the start and end of the sublist.
2. Reverse the nodes in place by adjusting the pointers.

LeetCode Problems:

1. Reverse Linked List (LeetCode #206)
2. Reverse Linked List II (LeetCode #92)
3. Swap Nodes in Pairs (LeetCode #24)

6. Monotonic Stack

The Monotonic Stack pattern uses a stack to maintain a sequence of elements in a specific order (increasing or decreasing).

Use this pattern for problems that require finding the **next greater** or **smaller** element.

Sample Problem:

Find the next greater element for each element in an array. Output -1 if the greater element doesn't exist.

Example:

- **Input:** `nums = [2, 1, 2, 4, 3]`
- **Output:** `[4, 2, 4, -1, -1]`

Explanation:

1. Use a stack to keep track of elements for which we haven't found the next greater element yet.
2. Iterate through the array, and for each element, pop elements from the

stack until you find a greater element.

3. If the stack is not empty, set the result for index at the top of the stack to current element.

4. Push the current element onto the stack.

LeetCode Problems:

1. Next Greater Element I (LeetCode #496)

2. Daily Temperatures (LeetCode #739)

3. Largest Rectangle in Histogram (LeetCode #84)

7. Top 'K' Elements

The Top 'K' Elements pattern finds the top k largest or smallest elements in an array or stream of data using **heaps** or **sorting**.

Sample Problem:

Find the k-th largest element in an unsorted array.

Example:

- **Input:** `nums = [3, 2, 1, 5, 6, 4]`, `k = 2`
- **Output:** `5`

Explanation:

1. Use a min-heap of size k to keep track of the k largest elements.
2. Iterate through the array, adding elements to the heap.
3. If the heap size exceeds k, remove the smallest element from the heap.
4. The root of the heap will be the k-th largest element.

LeetCode Problems:

1. Kth Largest Element in an Array (LeetCode #215)
2. Top K Frequent Elements (LeetCode #347)
3. Find K Pairs with Smallest Sums (LeetCode #373)

8. Overlapping Intervals

The Overlapping Intervals pattern is used to merge or handle overlapping intervals in an array.

In an interval array sorted by **start time**, two intervals `[a, b]` and `[c, d]` overlap if `b >= c` (i.e., the end time of the first interval is greater than or equal to the start time of the second interval).

Sample Problem:

Problem Statement: Merge all overlapping intervals.

Example:

- Input: `intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]`
- Output: `[[1, 6], [8, 10], [15, 18]]`

Explanation:

1. Sort the intervals by their start time.
2. Create an empty list called `merged` to store the merged intervals.
3. Iterate through the intervals and check if it overlaps with the last interval

in the `merged` list.

4. If it overlaps, merge the intervals by updating the end time of the last interval in `merged`.

5. If it does not overlap, simply add the current interval to the `merged` list.

LeetCode Problems:

1. [Merge Intervals (LeetCode #56)]()

2. [Insert Interval (LeetCode #57)]()

3. [Non-Overlapping Intervals (LeetCode #435)]()

9. Modified Binary Search

The Modified Binary Search pattern adapts binary search to solve a wider range of problems, such as finding elements in rotated sorted arrays.

Use this pattern for problems involving sorted or rotated arrays where you need to find a specific element.

Sample Problem:

Find an element in a rotated sorted array.

Example:

- Input: `nums = [4, 5, 6, 7, 0, 1, 2]`, `target = 0`
- Output: `4`

Explanation:

1. Perform binary search with an additional check to determine which half of the array is sorted.
2. We then check if the target is within the range of the sorted half.
3. If it is, we search that half; otherwise, we search the other half.

LeetCode Problems:

1. Search in Rotated Sorted Array (LeetCode #33)
2. Find Minimum in Rotated Sorted Array (LeetCode #153)
3. Search a 2D Matrix II (LeetCode #240)

10. Binary Tree Traversal

Binary Tree Traversal involves visiting all the nodes in a binary tree in a specific order.

- **PreOrder**: `root -> left -> right`
- **InOrder**: `left -> root -> right`
- **PostOrder**: `left -> right -> root`

Sample Problem:

Problem Statement: Perform inorder traversal of a binary tree.

Example:

- Input: `root = [1, null, 2, 3]`
- Output: `[1, 3, 2]`

Explanation:

1. Inorder traversal visits nodes in the order: left, root, right.
2. Use recursion or a stack to traverse the tree in this order.

LeetCode Problems:

1. PreOrder → Binary Tree Paths (LeetCode #257)
2. InOrder → Kth Smallest Element in a BST (LeetCode #230)
3. PostOrder → Binary Tree Maximum Path Sum (LeetCode #124)

11. Depth-First Search (DFS)

Depth-First Search (DFS) is a traversal technique that explores as far down a branch as possible before backtracking.

Use this pattern for exploring all paths or branches in graphs or trees.

Sample Problem:

Find all paths from the root to leaves in a binary tree.

Example:

- Input: `root = [1, 2, 3, null, 5]`
- Output: `["1->2->5", "1->3"]`

Explanation:

1. Use recursion or a stack to traverse each path from the root to the leaves.
2. Record each path as you traverse.

LeetCode Problems:

1. Clone Graph (LeetCode #133)

2. Path Sum II (LeetCode #113)

3. Course Schedule II (LeetCode #210)

12. Breadth-First Search (BFS)

Breadth-First Search (BFS) is a traversal technique that explores nodes level by level in a tree or graph.

Use this pattern for finding the shortest paths in unweighted graphs or level-order traversal in trees.

Sample Problem:

Perform level-order traversal of a binary tree.

Example:

- Input: `root = [3, 9, 20, null, null, 15, 7]`
- Output: `[[3], [9, 20], [15, 7]]`

Explanation:

1. Use a queue to keep track of nodes at each level.
2. Traverse each level and add the children of the current nodes to the queue.

LeetCode Problems:

1. Binary Tree Level Order Traversal (LeetCode #102)
2. Rotting Oranges (LeetCode #994)
3. Word Ladder (LeetCode #127)

13. Matrix Traversal

Matrix Traversal involves traversing elements in a matrix using different techniques (DFS, BFS, etc.).

Use this pattern for problems involving traversing 2D grids or matrices horizontally, vertically or diagonally.

Sample Problem:

Perform flood fill on a 2D grid. Change all the cells connected to the starting cell to new color.

Example:

- **Input:** `image = [[1,1,1],[1,1,0],[1,0,1]]`, `sr = 1`, `sc = 1`, `newColor = 2`
- **Output:** `[[2,2,2],[2,2,0],[2,0,1]]`

Explanation:

1. Use DFS or BFS to traverse the matrix starting from the given cell.
2. Change the color of the connected cells to the new color.

LeetCode Problems:

1. Flood Fill (LeetCode #733)
2. Number of Islands (LeetCode #200)
3. Surrounded Regions (LeetCode #130)

14. Backtracking

Backtracking explores all possible solutions and backtracks when a solution path fails.

Use this pattern when you need to find all (or some) solutions to a problem that satisfies given constraints. For example: combinatorial problems, such as generating permutations, combinations, or subsets.

Sample Problem:

Generate all permutations of a given list of numbers.

Example:

- **Input:** `nums = [1, 2, 3]`
- **Output:** `[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]`

Explanation:

1. Use recursion to generate permutations.

2. For each element, include it in the current permutation and recursively generate the remaining permutations.
3. Backtrack when all permutations for a given path are generated.

LeetCode Problems:

1. [Permutations (LeetCode #46)]
2. [Subsets (LeetCode #78)]
3. [N-Queens (LeetCode #51)]

15. Dynamic Programming Patterns

Dynamic Programming (DP) involves breaking down problems into smaller subproblems and solving them using a bottom-up or top-down approach.

Use this pattern for problems with overlapping subproblems and optimal substructure.

DP itself has multiple sub-patterns. Some of the most important ones are:

- Fibonacci Numbers
- 0/1 Knapsack
- Longest Common Subsequence (LCS)
- Longest Increasing Subsequence (LIS)
- Subset Sum
- Matrix Chain Multiplication

For more Dynamic Programming Patterns, checkout my other article:

Sample Problem:

Calculate the n-th Fibonacci number.

Example:

- Input: `n = 5`
- Output: `5` (The first five Fibonacci numbers are 0, 1, 1, 2, 3, 5)

Explanation:

1. Use a bottom-up approach to calculate the n-th Fibonacci number.
2. Start with the first two numbers (0 and 1) and iterate to calculate the next numbers like `(dp[i] = dp[i - 1] + dp[i - 2])`.

LeetCode Problems:

1. [Climbing Stairs (LeetCode #70)]
2. [House Robber (LeetCode #198)]
3. [Coin Change (LeetCode #322)]

4. Longest Common Subsequence (LCS) (LeetCode #1143)
5. Longest Increasing Subsequence (LIS) (LeetCode #322)
6. Partition Equal Subset Sum (LeetCode #416)

Thank you so much for reading.

If you found it valuable, hit a like ❤️ and consider subscribing for more such content every week.

If you have any questions or suggestions, leave a comment.

This post is public so feel free to share it.

Share

Checkout my **Youtube channel** for more in-depth content.

Follow me on **LinkedIn**, **X** and **Medium** to stay updated.

Checkout my **GitHub repositories** for free interview preparation resources.

I hope you have a lovely day!

See you soon,

Ashish