Senior Java Developer's Guide to LeetCode Patterns

This guide provides a deep dive into the algorithmic patterns required to crack high-level engineering interviews. For a senior developer, the goal is not just a "working" solution, but one that demonstrates an understanding of Java internals, memory management, and computational trade-offs.

1. Two Pointers (The "Opposite Ends" Approach)

Pattern Identification & Logic

- **How to Identify:** The input is **Sorted** (or sorting is feasible). You need to find a pair or triplet that meets a criteria, or you need to reverse/manipulate elements in-place.
- **The Algorithm:** Initialize `left = 0` and `right = length - 1`. Shrink the search space by moving pointers toward each other based on the current state relative to the target.
- **The 'Trick' to Know:** In a sorted array, moving the left pointer *always* increases (or keeps equal) the sum, and moving the right pointer *always* decreases it. This monotonic property allows us to discard an entire row of possible pairs in $O(1)$.

Java Implementation (Brute Force vs. Optimal)

Brute Force (Nested Loops)

```
public int[] twoSumBrute(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[i] + nums[j] == target) return new int[]{i + 1, j + 1};
        }
    }
    return new int[]{-1, -1};
}
```

Optimal Pattern (Two Pointers)

```
public int[] twoSumOptimal(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int sum = nums[left] + nums[right];
        if (sum == target) return new int[]{left + 1, right + 1};
        if (sum < target) left++; // Need a larger sum
        else right--; // Need a smaller sum
    }
    return new int[]{-1, -1};
}
```

Java Architecture Insights: We prefer primitive arrays `int[]` over `ArrayList<Integer>` for these patterns to avoid the overhead of **Auto-boxing/Unboxing** and to ensure the data is stored in a contiguous block of memory, which is cache-friendly.

Mental Model & Visualization

```
Target: 9 | Array: [2, 3, 4, 6, 7, 11]
Step 1: [2, 3, 4, 6, 7, 11] | L=2, R=11 | Sum=13 > 9 | Action: right--
Step 2: [2, 3, 4, 6, 7, 11] | L=2, R=7  | Sum=9  == 9 | Action: FOUND!
```

Senior Mental Trigger: "Exploit the Sorting: Move the bounds to converge on the target."

Curated Problem Lists

- **Commonly Asked:** LC 125 (Valid Palindrome), LC 167 (Two Sum II), LC 26 (Remove Duplicates), LC 88 (Merge Sorted Array), LC 977 (Squares of Sorted Array).
- **FAANG 'Aha!' Level:** LC 11 (Container With Most Water), LC 15 (3Sum), LC 42 (Trapping Rain Water - Hard), LC 75 (Sort Colors), LC 16 (3Sum Closest).

Complexity Table

Approach	Time Complexity	Space Complexity
Brute Force	$O(n^2)$	$O(1)$
Two Pointers	$O(n)$	$O(1)$

2. Sliding Window (The "Moving Frame" Approach)

Pattern Identification & Logic

- **How to Identify:** The problem asks for a **Sub-array** or **Sub-string** that is contiguous. Keywords include "Longest," "Shortest," or "Subarray sum equals K."
- **The Algorithm:** Use two pointers to represent a window. Expand the `right` pointer to include new elements. When a condition is violated, contract the `left` pointer to shrink the window until the condition is met again.
- **The 'Trick' to Know:** Instead of re-summing the entire window every time it moves, we use **Incremental Updates**. `newSum = oldSum - leavingElement + enteringElement`.

Java Implementation

Brute Force

```
public double findMaxAverageBrute(int[] nums, int k) {
    double max = -Double.MAX_VALUE;
    for (int i = 0; i <= nums.length - k; i++) {
        double sum = 0;
        for (int j = i; j < i + k; j++) sum += nums[j];
        max = Math.max(max, sum / k);
    }
    return max;
}
```

Optimal Pattern (Sliding Window)

```
public double findMaxAverageOptimal(int[] nums, int k) {
    double sum = 0;
    for (int i = 0; i < k; i++) sum += nums[i];
    double max = sum;
    for (int i = k; i < nums.length; i++) {
        sum += nums[i] - nums[i - k]; // Slide: Add new, remove old
        max = Math.max(max, sum);
    }
    return max / k;
}
```

Java Architecture Insights: When dealing with Strings and sliding windows (e.g., LC 3), remember that `String.substring()` in modern Java creates a new String object ($O(k)$ time). For better performance, manipulate a `char[]` or use `indexOf` and pointers to avoid object allocation pressure.

Mental Model & Visualization

```
Array: [1, 3, -1, -3, 5, 3], K = 3
[1, 3, -1] -> Sum=3
   [3, -1, -3] -> Sum = 3 - (1) + (-3) = -1
      [-1, -3, 5] -> Sum = -1 - (3) + (5) = 1
```

Senior Mental Trigger: "Middle elements are reusable; only the edges change."

Curated Problem Lists

- **Commonly Asked:** LC 643 (Max Average Subarray I), LC 209 (Min Size Subarray Sum), LC 1456 (Max Vowels in Substring), LC 1004 (Max Consecutive Ones III).
- **FAANG 'Aha!' Level:** LC 3 (Longest Substring w/o Repeating Chars), LC 76 (Minimum Window Substring - Hard), LC 30 (Substring with Concatenation), LC 239 (Sliding Window Maximum - Hard).

Complexity Table

Approach	Time Complexity	Space Complexity
Brute Force	$O(n \cdot k)$	$O(1)$
Sliding Window	$O(n)$	$O(1)$ or $O(k)$ for Map

3. Hashing (The "Memory Trade-off" Approach)

Pattern Identification & Logic

- **How to Identify:** You need to find a value that was "seen before" or track the frequency of elements. $O(1)$ lookup is required to beat nested loops.
- **The Algorithm:** Iterate through the collection once. For each element, check the `HashMap` or `HashSet` for a target condition (e.g., `target - current`). If not found, store the current element

for future iterations.

- **The 'Trick' to Know:** The **Complement Strategy**. Instead of looking for `a + b = target`, iterate through `a` and look for the existence of `target - a` in the history.

Java Implementation

Brute Force

```
public int[] twoSumBrute(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[i] + nums[j] == target) return new int[]{i, j};
        }
    }
    return null;
}
```

Optimal Pattern (HashMap)

```
public int[] twoSumOptimal(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>(); // Value -> Index
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[]{map.get(complement), i};
        }
        map.put(nums[i], i);
    }
    return null;
}
```

Java Architecture Insights: For Seniors, understand that `HashMap` uses a **linked list (or red-black tree in Java 8+)** for collision handling. For fixed character sets (like lowercase English letters), an `int[26]` array is faster than a `HashMap<Character, Integer>` because it avoids the overhead of object creation and hashing functions.

Mental Model & Visualization

```
Target: 10 | Array: [2, 7, 11, 8]
1. Seen 2? No. Add {2:0} to Map.
2. Seen (10-7=3)? No. Add {7:1} to Map.
3. Seen (10-11=-1)? No. Add {11:2} to Map.
4. Seen (10-8=2)? YES! Found 2 at index 0. Return [0, 3].
```

Senior Mental Trigger: "Trade space for time: Use a Map to remember the past."

Curated Problem Lists

- **Commonly Asked:** LC 1 (Two Sum), LC 242 (Valid Anagram), LC 217 (Contains Duplicate), LC 387 (First Unique Char), LC 13 (Roman to Integer).

- **FAANG 'Aha!' Level:** LC 49 (Group Anagrams), LC 128 (Longest Consecutive Sequence), LC 560 (Subarray Sum Equals K), LC 146 (LRU Cache - Hard/System Design).

Complexity Table

Approach	Time Complexity	Space Complexity
Brute Force	$O(n^2)$	$O(1)$
Hashing	$O(n)$	$O(n)$

4. Fast & Slow Pointers (The "Hare and Tortoise")

Pattern Identification & Logic

- **How to Identify:** Primarily used in **Linked Lists** or cyclic arrays. Used to find the middle, detect a cycle, or find the start of a cycle.
- **The Algorithm:** Two pointers start at the head. `slow` moves 1 step, `fast` moves 2 steps.
- **The 'Trick' to Know:** If a cycle exists, the `fast` pointer will eventually catch up to the `slow` pointer inside the loop. The distance between them decreases by exactly 1 in each step, guaranteeing they meet.

Java Implementation (Cycle Detection)

Brute Force

```
public boolean hasCycleBrute(ListNode head) {
    Set<ListNode> visited = new HashSet<>();
    while (head != null) {
        if (visited.contains(head)) return true;
        visited.add(head);
        head = head.next;
    }
    return false;
}
```

Optimal Pattern (Floyd's Algorithm)

```
public boolean hasCycleOptimal(ListNode head) {
    if (head == null) return false;
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) return true; // Collision!
    }
    return false;
}
```

Java Architecture Insights: Linked Lists in Java are non-contiguous. Every node is a separate object on the heap. Fast and slow pointers are just **references** (4-8 bytes). This is highly memory-efficient compared to storing every node reference in a `HashSet`.

Mental Model & Visualization

```
Cycle: 1 -> 2 -> 3 -> 4 -> 2...
Step 1: S=1, F=1
Step 2: S=2, F=3
Step 3: S=3, F=2
Step 4: S=4, F=4 -> MET!
```

Senior Mental Trigger: "If you're running in circles, the fast one will eventually lap the slow one."

Curated Problem Lists

- **Commonly Asked:** LC 141 (Linked List Cycle), LC 876 (Middle of Linked List), LC 234 (Palindrome Linked List), LC 143 (Reorder List).
- **FAANG 'Aha!' Level:** LC 142 (Linked List Cycle II), LC 202 (Happy Number), LC 287 (Find the Duplicate Number), LC 457 (Circular Array Loop).

Complexity Table

Approach	Time Complexity	Space Complexity
HashSet	$O(n)$	$O(n)$
Fast & Slow	$O(n)$	$O(1)$

5. Trees & Graphs (BFS / Level Order)

Pattern Identification & Logic

- **How to Identify:** You need to traverse a tree/graph **level-by-level** or find the **shortest path** in an unweighted graph.
- **The Algorithm:** Use a **Queue**. Add the root. While the queue is not empty: 1. Get the current size (number of nodes in this level). 2. Process all nodes in that level. 3. Add their children to the queue.
- **The 'Trick' to Know:** The `size()` snapshot. By capturing the queue size at the start of each while-loop iteration, you can isolate levels without needing to store level info in the nodes.

Java Implementation

Optimal Pattern (BFS)

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) return result;

    Queue<TreeNode> queue = new ArrayDeque<>(); // Better than LinkedList
```

```
        queue.add(root);

        while (!queue.isEmpty()) {
            int levelSize = queue.size(); // SNAPSHOT: Current level only
            List<Integer> currentLevel = new ArrayList<>(levelSize);

            for (int i = 0; i < levelSize; i++) {
                TreeNode node = queue.poll();
                currentLevel.add(node.val);
                if (node.left != null) queue.add(node.left);
                if (node.right != null) queue.add(node.right);
            }
            result.add(currentLevel);
        }
        return result;
    }
```

Java Architecture Insights: Use `ArrayDeque` instead of `LinkedList` for your Queue in Java. `ArrayDeque` is backed by a circular array and has better cache locality and lower memory overhead than `LinkedList`, which creates a `Node` object for every single insertion.

Mental Model & Visualization

```
Level 0: [1] -> Queue: [1]
Level 1: Pop 1, Add children -> Queue: [2, 3]
Level 2: Pop 2, Pop 3, Add children -> Queue: [4, 5, 6, 7]
```

Senior Mental Trigger: "Queue up for horizontal exploration."

Curated Problem Lists

- **Commonly Asked:** LC 102 (Binary Tree Level Order), LC 104 (Max Depth), LC 107 (Level Order II), LC 111 (Min Depth).
- **FAANG 'Aha!' Level:** LC 103 (Zigzag Level Order), LC 199 (Right Side View), LC 994 (Rotting Oranges), LC 116 (Populating Next Pointers).

Complexity Table

Pattern	Time Complexity	Space Complexity
BFS	$O(V + E)$	$O(V)$ (Width of tree)

6. Binary Search (Search Space Reduction)

Pattern Identification & Logic

- **How to Identify:** The input is sorted, OR you are looking for a value in a range where you can easily eliminate half the possibilities (e.g., "Find the smallest number that...").
- **The Algorithm:** `low = 0, high = max`. Find `mid`. If `mid` is too small, `low = mid + 1`. If `mid` is too big, `high = mid - 1`.

- **The 'Trick' to Know: Binary Search on Answer Space**. You don't always search an array; sometimes you search the possible *results* (e.g., searching for the minimum capacity of a ship).

Java Implementation (Rotated Sorted Array)

Optimal Pattern

```
public int search(int[] nums, int target) {
    int low = 0, high = nums.length - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2; // TRICK: Avoid overflow
        if (nums[mid] == target) return mid;

        // Identify which half is sorted
        if (nums[low] <= nums[mid]) { // Left is sorted
            if (target >= nums[low] && target < nums[mid]) high = mid - 1;
            else low = mid + 1;
        } else { // Right is sorted
            if (target > nums[mid] && target <= nums[high]) low = mid + 1;
            else high = mid - 1;
        }
    }
    return -1;
}
```

Java Architecture Insights: Senior devs never use `(low + high) / 2`. If `low` and `high` are large, their sum can exceed `Integer.MAX_VALUE`, causing a stack overflow or negative index error. Use `low + (high - low) / 2`.

Mental Model & Visualization

```
Sorted Range: [10, 20, 30, 40, 50] | Target: 40
1. Mid is 30. 30 < 40? Yes. Search right half [40, 50].
2. Mid is 45. 45 > 40? Yes. Search left half [40, 40].
3. Mid is 40. Found.
```

Senior Mental Trigger: "Discard half the world with every question."

Curated Problem Lists

- **Commonly Asked:** LC 704 (Binary Search), LC 35 (Search Insert Position), LC 74 (Search 2D Matrix), LC 278 (First Bad Version).
- **FAANG 'Aha!' Level:** LC 33 (Rotated Sorted Array), LC 153 (Min in Rotated Array), LC 4 (Median of Two Sorted Arrays - Hard), LC 1011 (Capacity to Ship Packages).

7. Backtracking (Decision Trees)

Pattern Identification & Logic

- **How to Identify:** You need to find **all** possible solutions, combinations, or permutations.

- **The Algorithm:** Choose an option -> Recurse -> **Undo the choice** (backtrack) -> Try the next option.
- **The 'Trick' to Know:** State Management. You must restore the state of your `path` or `visited` structure so the previous recursive call doesn't see your modified data.

Java Implementation (Permutations)

Optimal Pattern

```
public List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> res = new ArrayList<>();
    backtrack(res, new ArrayList<>(), nums);
    return res;
}

private void backtrack(List<List<Integer>> res, List<Integer> temp, int[] nums) {
    if (temp.size() == nums.length) {
        res.add(new ArrayList<>(temp)); // TRICK: Must create a NEW list!
        return;
    }
    for (int i = 0; i < nums.length; i++) {
        if (temp.contains(nums[i])) continue; // Skip used elements
        temp.add(nums[i]); // Choose
        backtrack(res, temp, nums); // Explore
        temp.remove(temp.size() - 1); // Backtrack (Undo)
    }
}
```

Java Architecture Insights: In `res.add(new ArrayList<>(temp))`, we create a deep copy. If you just do `res.add(temp)`, the final result will be a list full of empty sub-lists because the `temp` object is modified and eventually emptied during backtracking.

Mental Model & Visualization

```
Choices: [1, 2]
1. Pick 1 -> List: [1] -> Pick 2 -> List: [1, 2] (Save)
2. Backtrack -> List: [1] -> Backtrack -> List: []
3. Pick 2 -> List: [2] -> Pick 1 -> List: [2, 1] (Save)
```

Senior Mental Trigger: "Try every path, but always clean up your tracks."

Curated Problem Lists

- **Commonly Asked:** LC 46 (Permutations), LC 77 (Combinations), LC 78 (Subsets), LC 39 (Combination Sum).
- **FAANG 'Aha!' Level:** LC 51 (N-Queens - Hard), LC 79 (Word Search), LC 17 (Letter Combinations of Phone), LC 22 (Generate Parentheses).

8. Dynamic Programming (Optimization through Memory)

Pattern Identification & Logic

- **How to Identify:** The problem has "Overlapping Subproblems" (you solve the same thing multiple times) and "Optimal Substructure" (the answer for N depends on the answer for N-1).
- **The Algorithm:**
  - **Top-Down (Memoization):** Use recursion + a `HashMap` or array to cache results.
  - **Bottom-Up (Tabulation):** Build an array from the base case up.
- **The 'Trick' to Know: Space Optimization**. Often, to calculate `dp[i]`, you only need `dp[i-1]` and `dp[i-2]`. You don't need a whole array; just two variables.

Java Implementation (Coin Change)

Optimal Pattern (Tabulation)

```
public int coinChange(int[] coins, int amount) {
    int[] dp = new int[amount + 1];
    Arrays.fill(dp, amount + 1); // Max value placeholder
    dp[0] = 0;

    for (int i = 1; i <= amount; i++) {
        for (int coin : coins) {
            if (i - coin >= 0) {
                dp[i] = Math.min(dp[i], 1 + dp[i - coin]);
            }
        }
    }
    return dp[amount] > amount ? -1 : dp[amount];
}
```

Java Architecture Insights: `Arrays.fill()` is highly optimized using `System.arraycopy` or low-level intrinsics. When choosing between Memoization and Tabulation, remember that Top-Down (Recursion) can hit a `StackOverflowError` in Java if the tree is too deep, whereas Bottom-Up is purely iterative and safer.

Mental Model & Visualization

```
Climbing Stairs (n=3):
dp[1] = 1 way
dp[2] = 2 ways
dp[3] = dp[1] + dp[2] = 3 ways.
```

Senior Mental Trigger: "Don't calculate what you've already solved."

Curated Problem Lists

- **Commonly Asked:** LC 70 (Climbing Stairs), LC 198 (House Robber), LC 322 (Coin Change), LC 53 (Maximum Subarray).
- **FAANG 'Aha!' Level:** LC 300 (Longest Increasing Subsequence), LC 1143 (Longest Common Subsequence), LC 416 (Partition Equal Subset Sum), LC 72 (Edit Distance - Hard).

Complexity Table

Type	Time Complexity	Space Complexity