Common DSA Patterns for Interview Preparation

Your Name

July 15, 2025

Contents

1 Introduction

This document outlines key algorithmic patterns for solving Data Structures and Algorithms (DSA) problems, essential for technical interviews. Each pattern is explained in simple terms, with Java code examples and relevant LeetCode problems to aid revision. The focus is on practical understanding for a student preparing for placements within a year, covering patterns like Dynamic Programming, Greedy, Heap, Backtracking, and previously discussed patterns (Sliding Window, Two Pointers, etc.).

2 Sliding Window

The Sliding Window pattern is used for problems involving arrays or strings, where you process a subarray (or substring) that moves across the data. The "window" is a range of elements, and you adjust its size or position to find an optimal solution, like a maximum sum or longest substring.

2.1 How It Works

- **Fixed Window**: Keep a window of fixed size, slide it across the array, and compute results (e.g., maximum sum of k elements).
- **Variable Window**: Expand or shrink the window based on a condition (e.g., find the longest substring with distinct characters).

2.2 Example: Maximum Sum Subarray of Size k

Given an array, find the maximum sum of any subarray of size k.

Input: arr = [1, 4, 2, 10, 2, 3, 1, 0, 20], $k = 4$

Output: 24 (subarray [4, 2, 10, 2])

Solution:

- Compute the sum of the first k elements.
- Slide the window by adding the next element and removing the first element of the previous window.
- Track the maximum sum.

```java
class Solution {
    public int maxSumSubarray(int[] arr, int k) {
        int n = arr.length;
        if (n < k) return 0;
        int windowSum = 0;
        for (int i = 0; i < k; i++) windowSum += arr[i];
        int maxSum = windowSum;
        for (int i = 0; i < n - k; i++) {
            windowSum = windowSum - arr[i] + arr[i + k];
            maxSum = Math.max(maxSum, windowSum);
        }
```

```
        return maxSum;
    }
}
```

2.3 When to Use

Use Sliding Window for problems involving:

- Subarrays or substrings with constraints (e.g., size k, distinct characters).
- Optimizing sums, counts, or lengths.

LeetCode Examples: "Maximum Average Subarray I" (Easy), "Longest Substring Without Repeating Characters" (Medium).

3 Two Pointers

The Two Pointers pattern uses two indices to traverse an array or string, often to find pairs or satisfy conditions. Its efficient for sorted arrays or when searching for specific sums or differences.

3.1 How It Works

- **Same Direction**: Both pointers move in the same direction, adjusting based on conditions.
- **Opposite Direction**: Pointers start at opposite ends (e.g., start and end of array) and move toward each other.

3.2 Example: Two Sum in Sorted Array

Given a sorted array and a target sum, find two numbers that add up to the target.

Input: numbers = [2, 7, 11, 15], target = 9

Output: [1, 2] (indices of 2 and 7)

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int left = 0, right = numbers.length - 1;
        while (left < right) {
            int sum = numbers[left] + numbers[right];
            if (sum == target) return new int[]{left + 1, right +
                1};
            else if (sum < target) left++;
            else right--;
        }
        return new int[]{};
    }
}
```

3.3 When to Use

Use Two Pointers for:

- Finding pairs (e.g., sum, difference).
- Problems with sorted arrays or strings.
- Merging or partitioning arrays.

LeetCode Examples: "Two Sum II - Input Array Is Sorted" (Medium), "Container With Most Water" (Medium).

4 Fast and Slow Pointers

The Fast and Slow Pointers pattern uses two pointers moving at different speeds, typically in linked lists, to detect cycles or find midpoints.

4.1 How It Works

- **Fast Pointer**: Moves twice as fast (e.g., two nodes at a time).
- **Slow Pointer**: Moves one step at a time.
- If they meet, a cycle exists; otherwise, they reach the end.

4.2 Example: Detect Cycle in Linked List

Given a linked list, determine if it has a cycle.

Input: head = [3, 2, 0, -4], cycle at position 1

Output: true

```
class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == 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;
        }
        return false;
    }
}
```

4.3 When to Use

Use Fast and Slow Pointers for:

- Detecting cycles in linked lists.
- Finding midpoints or specific nodes.

LeetCode Examples: "Linked List Cycle" (Easy), "Find the Duplicate Number" (Medium).

5 Binary Search

Binary Search finds an element in a sorted array by repeatedly dividing the search space in half, reducing time complexity to $O(\log n)$.

5.1 How It Works

- Start with left and right pointers at the arrays ends.
- Compute the middle index.
- If the middle element is the target, return it.
- If the target is smaller, search the left half; if larger, search the right half.

5.2 Example: Search in Sorted Array

Given a sorted array and a target, find the targets index.

Input: nums = [1, 3, 5, 6], target = 5

Output: 2

```
class Solution {
    public int search(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) return mid;
            else if (nums[mid] < target) left = mid + 1;
            else right = mid - 1;
        }
        return -1;
    }
}
```

5.3 When to Use

Use Binary Search for:

- Searching in sorted arrays or matrices.
- Problems requiring optimization (e.g., finding minimum/maximum).

LeetCode Examples: "Binary Search" (Easy), "Search in Rotated Sorted Array" (Medium).

6 Prefix Sum

The Prefix Sum pattern precomputes cumulative sums of an array to answer range sum queries efficiently.

6.1 How It Works

- Create an array where `prefix[i]` = sum of elements from index 0 to i.
- For a range sum from index `l` to `r`, compute `prefix[r] - prefix[l-1]`.

6.2 Example: Range Sum Query

Given an array, find the sum of elements between indices `l` and `r`.

Input: nums = [1, 2, 3, 4], queries = [[1, 3], [0, 2]]

Output: [9, 6]

```
class Solution {
    public int[] sumQuery(int[] nums, int[][] queries) {
        int[] prefix = new int[nums.length + 1];
        for (int i = 0; i < nums.length; i++) {
            prefix[i + 1] = prefix[i] + nums[i];
        }
        int[] result = new int[queries.length];
        for (int i = 0; i < queries.length; i++) {
            int l = queries[i][0], r = queries[i][1];
            result[i] = prefix[r + 1] - prefix[l];
        }
        return result;
    }
}
```

6.3 When to Use

Use Prefix Sum for:

- Range sum or count queries.
- Problems involving cumulative calculations.

LeetCode Examples: "Range Sum Query - Immutable" (Easy), "Subarray Sum Equals K" (Medium).

7 Depth-First Search (DFS)

DFS explores a graph or tree by going as deep as possible along a branch before backtracking, often using recursion or a stack.

7.1 How It Works

- Start at a node, mark it as visited.
- Recursively explore each unvisited neighbor.
- Backtrack when no unvisited neighbors remain.

7.2 Example: Number of Islands

Given a grid of '1's (land) and '0's (water), count the number of islands.

Input: grid = [["1","1","0"], ["1","1","0"], ["0","0","1"]]

Output: 2

```
class Solution {
    public int numIslands(char[][] grid) {
        int rows = grid.length, cols = grid[0].length;
        int islands = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (grid[i][j] == '1') {
                    dfs(grid, i, j);
                    islands++;
                }
            }
        }
        return islands;
    }

    private void dfs(char[][] grid, int i, int j) {
        if (i < 0 || i >= grid.length || j < 0 || j >=
           grid[0].length || grid[i][j] != '1') return;
        grid[i][j] = '0';
        dfs(grid, i + 1, j);
        dfs(grid, i - 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i, j - 1);
    }
}
```

]

When to Use Use DFS for:

- Graph or tree traversal.
- Problems involving connectivity or paths.

LeetCode Examples: "Number of Islands" (Medium), "Flood Fill" (Easy).

8 Breadth-First Search (BFS)

BFS explores a graph level by level, using a queue to process nodes in order of their distance from the starting node.

8.1 How It Works

- Start at a node, add it to a queue.

- Process each node in the queue, adding its unvisited neighbors.
- Continue until the queue is empty.

8.2 Example: Shortest Path in Binary Matrix

Given a binary matrix, find the shortest path from top-left to bottom-right.

Input: grid = [[0,1],[1,0]]

Output: 2

```
class Solution {
    public int shortestPathBinaryMatrix(int[][] grid) {
        if (grid[0][0] == 1) return -1;
        int n = grid.length;
        Queue<int[]> queue = new LinkedList<>();
        queue.offer(new int[]{0, 0, 1}); // [row, col, distance]
        grid[0][0] = 1;
        int[][] directions = {{0,1}, {1,0}, {0,-1}, {-1,0}, {1,1},
           {-1,-1}, {1,-1}, {-1,1}};
        while (!queue.isEmpty()) {
            int[] cell = queue.poll();
            int r = cell[0], c = cell[1], dist = cell[2];
            if (r == n - 1 && c == n - 1) return dist;
            for (int[] d : directions) {
                int nr = r + d[0], nc = c + d[1];
                if (nr >= 0 && nr < n && nc >= 0 && nc < n &&
                   grid[nr][nc] == 0) {
                    grid[nr][nc] = 1;
                    queue.offer(new int[]{nr, nc, dist + 1});
                }
            }
        }
        return -1;
    }
}
```

8.3 When to Use

Use BFS for:

- Shortest path problems.
- Level-order traversal.

LeetCode Examples: "Shortest Path in Binary Matrix" (Medium), "Word Ladder" (Hard).

9 Dynamic Programming

Dynamic Programming (DP) solves problems by breaking them into smaller subproblems, solving each subproblem once, and storing results (memoization or tabulation) to avoid re-

dundant computations. Its used for optimization problems (e.g., minimum cost, maximum profit).

9.1 How It Works

- **Identify Subproblems**: Break the problem into smaller, overlapping subproblems.
- **Store Results**: Use a table (array) or memoization to store solutions to subproblems.
- **Build Solution**: Combine subproblem solutions to solve the original problem.

9.2 Example: Climbing Stairs

You can climb 1 or 2 steps at a time to reach the top of a staircase with n steps. How many distinct ways can you climb to the top?

Input: n = 3

Output: 3 (ways: [1,1,1], [1,2], [2,1])

Solution:

- For step i, you can reach it from step $i-1$ (1 step) or $i-2$ (2 steps).
- Use a DP array where `dp[i]` = number of ways to reach step i.
- `dp[i] = dp[i-1] + dp[i-2]` (like Fibonacci).

```
class Solution {
    public int climbStairs(int n) {
        if (n <= 2) return n;
        int[] dp = new int[n + 1];
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}
```

9.3 When to Use

Use DP for:

- Optimization problems (min/max, count ways).
- Problems with overlapping subproblems (e.g., Fibonacci-like).

: **LeetCode Examples**: "Climbing Stairs" (Easy), "House Robber" (Easy), "Longest Palindromic Substring" (Medium), "Coin Change" (Medium).

10 Greedy

The Greedy pattern makes locally optimal choices at each step, hoping to find a global optimum. Its simpler than DP but doesnt always guarantee the best solution.

10.1 How It Works

- At each step, choose the best option available (e.g., maximum profit, minimum cost).
- Continue until the problem is solved.

10.2 Example: Best Time to Buy and Sell Stock

Given an array of stock prices, find the maximum profit by buying and selling once.

Input: prices = [7,1,5,3,6,4]

Output: 5 (buy at 1, sell at 6)

Solution:

- Track the minimum price seen so far.
- At each price, calculate the potential profit and update the maximum.

```java
class Solution {
    public int maxProfit(int[] prices) {
        int minPrice = Integer.MAX_VALUE;
        int maxProfit = 0;
        for (int price : prices) {
            minPrice = Math.min(minPrice, price);
            maxProfit = Math.max(maxProfit, price - minPrice);
        }
        return maxProfit;
    }
}
```

10.3 When to Use

Use Greedy for:

- Problems where local optimal choices lead to a global optimum.
- Scheduling or allocation problems.

LeetCode Examples: "Best Time to Buy and Sell Stock" (Easy), "Jump Game" (Medium).

11 Heap (Priority Queue)

A Heap is a tree-based data structure that maintains the min or max property, used for problems requiring priority-based processing (e.g., smallest/largest elements).

11.1 How It Works

- **Min-Heap**: Parent is smaller than children.
- **Max-Heap**: Parent is larger than children.
- Use a Priority Queue to efficiently extract the min/max element.

11.2 Example: Top K Frequent Elements

Given an array, return the k most frequent elements.

Input: nums = [1,1,1,2,2,3], k = 2

Output: [1,2]

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> freq = new HashMap<>();
        for (int num : nums) {
            freq.put(num, freq.getOrDefault(num, 0) + 1);
        }
        PriorityQueue<Map.Entry<Integer, Integer>> pq = new
           PriorityQueue<>(
            (a, b) -> b.getValue() - a.getValue()
        );
        pq.addAll(freq.entrySet());
        int[] result = new int[k];
        for (int i = 0; i < k; i++) {
            result[i] = pq.poll().getKey();
        }
        return result;
    }
}
```

11.3 When to Use

Use Heap for:

- Problems requiring the smallest/largest k elements.
- Priority-based scheduling.

LeetCode Examples: "Top K Frequent Elements" (Medium), "Merge K Sorted Lists" (Hard).

12 Backtracking

Backtracking systematically tries all possible solutions, undoing choices that dont work, often used for combinatorial problems like permutations.

12.1 How It Works

- Try a partial solution.

- If valid, continue building; if invalid, backtrack and try another option.
- Use recursion to explore all possibilities.

12.2 Example: Generate Parentheses

Generate all valid combinations of n pairs of parentheses.

Input: n = 3

Output: ["((()))","(()())","(())()","()(()","()()()"]

```java
class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> result = new ArrayList<>();
        backtrack(result, "", 0, 0, n);
        return result;
    }

    private void backtrack(List<String> result, String curr, int
       open, int close, int n) {
        if (curr.length() == n * 2) {
            result.add(curr);
            return;
        }
        if (open < n) {
            backtrack(result, curr + "(", open + 1, close, n);
        }
        if (close < open) {
            backtrack(result, curr + ")", open, close + 1, n);
        }
    }
}
```

12.3 When to Use

Use Backtracking for:

- Combinatorial problems (permutations, combinations).
- Constraint satisfaction problems.

LeetCode Examples: "Generate Parentheses" (Medium), "N-Queens" (Hard).

13 Trie

A Trie (prefix tree) is a tree-like data structure for storing strings, used for efficient prefix-based searches.

13.1 How It Works

- Each node represents a character.

- Paths from the root to leaves form strings.
- Used for dictionary or autocomplete problems.

13.2 Example: Implement Trie

Implement a Trie with insert, search, and startsWith methods.

Input: insert("apple"), search("apple"), startsWith("app")

Output: true, true

```
class Trie {
    class TrieNode {
        TrieNode[] children = new TrieNode[26];
        boolean isEnd;
    }

    TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
        }
        node.isEnd = true;
    }

    public boolean search(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) return false;
            node = node.children[index];
        }
        return node.isEnd;
    }

    public boolean startsWith(String prefix) {
        TrieNode node = root;
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) return false;
            node = node.children[index];
        }
```

```
42              return true;
43          }
44      }
```

13.3 When to Use

Use Trie for:

- Prefix-based string searches.
- Dictionary or autocomplete problems.

LeetCode Examples: "Implement Trie (Prefix Tree)" (Medium), "Word Search II" (Hard).

14 Conclusion

Mastering these patternsSliding Window, Two Pointers, Fast and Slow Pointers, Binary Search, Prefix Sum, DFS, BFS, Dynamic Programming, Greedy, Heap, Backtracking, and Triecovers a wide range of DSA problems for interviews. Practice 3-5 problems per pattern weekly on LeetCode, focusing on Easy and Medium problems initially, then progressing to Hard. Use these patterns in your Go-based projects (e.g., implementing a Trie-based search API) to align with your AWS/DevOps goals. Regular revision and mock interviews will ensure readiness for placements.