DSA Patterns & Solutions Cheat Sheet

Quick Pattern Recognition Guide

1. Array & Hashing

When to use: Counting, frequency analysis, lookup operations
Key techniques: Hash maps, sets, frequency counters
Time: O(n), Space: O(n)

```
// Template: Two Sum
function twoSum(nums, target) {
    const map = new Map();
    for (let i = 0; i < nums.length; i++) {
        const complement = target - nums[i];
        if (map.has(complement)) {
            return [map.get(complement), i];
        }
        map.set(nums[i], i);
    }
    return [];
}
```

2. Two Pointers

When to use: Sorted arrays, palindromes, pair finding
Key techniques: Left/right pointers, fast/slow pointers
Time: O(n), Space: O(1)

```
// Template: Valid Palindrome
function isPalindrome(s) {
    let left = 0, right = s.length - 1;
    while (left < right) {
        while (left < right && !isAlphaNum(s[left])) left++;
        while (left < right && !isAlphaNum(s[right])) right--;
        if (s[left].toLowerCase() !== s[right].toLowerCase()) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

3. Sliding Window

When to use: Subarray/substring problems, optimization
Key techniques: Expand window, contract window, maintain window properties
Time: O(n), Space: O(1) or O(k)

```javascript
// Template: Longest Substring Without Repeating Characters
function lengthOfLongestSubstring(s) {
    const charSet = new Set();
    let left = 0, maxLength = 0;

    for (let right = 0; right < s.length; right++) {
        while (charSet.has(s[right])) {
            charSet.delete(s[left]);
            left++;
        }
        charSet.add(s[right]);
        maxLength = Math.max(maxLength, right - left + 1);
    }
    return maxLength;
}
```

4. Stack

When to use: Parsing, balanced parentheses, monotonic problems
Key techniques: LIFO operations, maintain order
Time: O(n), Space: O(n)

```javascript
// Template: Valid Parentheses
function isValid(s) {
    const stack = [];
    const map = { ')': '(', '}': '{', ']': '[' };

    for (let char of s) {
        if (char in map) {
            if (stack.length === 0 || stack.pop() !== map[char]) {
                return false;
            }
        } else {
            stack.push(char);
        }
    }
    return stack.length === 0;
}
```

5. Binary Search

When to use: Sorted arrays, search space reduction
Key techniques: Mid calculation, boundary updates
Time: O(log n), Space: O(1)

```javascript
// Template: Binary Search
function binarySearch(nums, target) {
    let left = 0, right = nums.length - 1;

    while (left <= right) {
        const mid = Math.floor(left + (right - left) / 2);
        if (nums[mid] === target) return mid;
        else if (nums[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

6. Linked List

When to use: Sequential access, pointer manipulation
Key techniques: Dummy nodes, two pointers, reversal
Time: O(n), Space: O(1)

```
// Template: Reverse Linked List
function reverseList(head) {
    let prev = null;
    let current = head;

    while (current !== null) {
        const nextTemp = current.next;
        current.next = prev;
        prev = current;
        current = nextTemp;
    }
    return prev;
}
```

7. Trees

When to use: Hierarchical data, recursive problems
Key techniques: DFS, BFS, recursion, tree properties
Time: O(n), Space: O(h) where h is height

```javascript
// Template: Tree DFS
function dfs(root) {
    if (!root) return null;

    // Process current node
    const result = root.val;

    // Recursive calls
    const left = dfs(root.left);
    const right = dfs(root.right);

    return result;
}

// Template: Tree BFS
function bfs(root) {
    if (!root) return [];

    const queue = [root];
    const result = [];

    while (queue.length > 0) {
        const levelSize = queue.length;
        const currentLevel = [];

        for (let i = 0; i < levelSize; i++) {
            const node = queue.shift();
            currentLevel.push(node.val);

            if (node.left) queue.push(node.left);
            if (node.right) queue.push(node.right);
        }
        result.push(currentLevel);
    }
    return result;
}
```

8. Tries

When to use: Prefix matching, word searches, autocomplete
Key techniques: Character-based tree traversal
Time: O(m) where m is word length, Space: O(ALPHABET_SIZE * N)

```
// Template: Trie Implementation
class TrieNode {
    constructor() {
        this.children = {};
        this.isEndOfWord = false;
    }
}

class Trie {
    constructor() {
        this.root = new TrieNode();
    }

    insert(word) {
        let current = this.root;
        for (let char of word) {
            if (!current.children[char]) {
                current.children[char] = new TrieNode();
            }
            current = current.children[char];
        }
        current.isEndOfWord = true;
    }

    search(word) {
        let current = this.root;
        for (let char of word) {
            if (!current.children[char]) return false;
            current = current.children[char];
        }
        return current.isEndOfWord;
    }
}
```

9. Heap / Priority Queue

When to use: Finding K elements, scheduling, optimization

Key techniques: Min heap, max heap, heap operations

Time: O(log n) for insert/delete, Space: O(n)

```
// Template: Find Kth Largest Element
function findKthLargest(nums, k) {
    const minHeap = new MinHeap();

    for (let num of nums) {
        minHeap.add(num);
        if (minHeap.size() > k) {
            minHeap.poll();
        }
    }
    return minHeap.peek();
}
```

10. Backtracking

When to use: Generating combinations, permutations, solving puzzles

Key techniques: Choose, explore, unchoose pattern

Time: O(N!), Space: O(N)

```
// Template: Combinations
function backtrack(nums, path, result, start) {
    // Base case - add current path to result
    result.push([...path]);

    for (let i = start; i < nums.length; i++) {
        // Choose
        path.push(nums[i]);
        // Explore
        backtrack(nums, path, result, i + 1);
        // Unchoose
        path.pop();
    }
}
```

11. Graphs

When to use: Connected components, shortest paths, dependencies
Key techniques: DFS, BFS, adjacency lists/matrix
Time: O(V + E), Space: O(V + E)

```
// Template: Graph DFS
function dfs(graph, node, visited) {
    if (visited.has(node)) return;

    visited.add(node);
    // Process node

    for (let neighbor of graph[node]) {
        dfs(graph, neighbor, visited);
    }
}

// Template: Graph BFS
function bfs(graph, start) {
    const queue = [start];
    const visited = new Set([start]);

    while (queue.length > 0) {
        const node = queue.shift();
        // Process node

        for (let neighbor of graph[node]) {
            if (!visited.has(neighbor)) {
                visited.add(neighbor);
                queue.push(neighbor);
            }
        }
    }
}
```

12. Dynamic Programming

When to use: Optimization problems, overlapping subproblems
Key techniques: Memoization, tabulation, optimal substructure
Time: O(nm), Space: O(nm)

```
// Template: 1D DP (Fibonacci-style)
function dp1D(n) {
    if (n <= 1) return n;

    const dp = new Array(n + 1);
    dp[0] = 0;
    dp[1] = 1;

    for (let i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}

// Template: 2D DP
function dp2D(m, n) {
    const dp = Array(m).fill().map(() => Array(n).fill(0));

    // Base cases
    dp[0][0] = 1;

    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            if (i > 0) dp[i][j] += dp[i-1][j];
            if (j > 0) dp[i][j] += dp[i][j-1];
        }
    }
    return dp[m-1][n-1];
}
```

13. Greedy

When to use: Local optimization leads to global optimum

Key techniques: Sort, choose locally optimal solution

Time: O(n log n), Space: O(1)

```
// Template: Activity Selection
function activitySelection(intervals) {
    intervals.sort((a, b) => a[1] - b[1]); // Sort by end time

    let count = 0;
    let lastEnd = 0;

    for (let interval of intervals) {
        if (interval[0] >= lastEnd) {
            count++;
            lastEnd = interval[1];
        }
    }
    return count;
}
```

14. Union Find (Disjoint Set)

When to use: Connected components, cycle detection

Key techniques: Path compression, union by rank

Time: O(α(n)) amortized, Space: O(n)

```
// Template: Union Find
class UnionFind {
    constructor(n) {
        this.parent = Array.from({length: n}, (_, i) => i);
        this.rank = new Array(n).fill(0);
    }

    find(x) {
        if (this.parent[x] !== x) {
            this.parent[x] = this.find(this.parent[x]); // Path compression
        }
        return this.parent[x];
    }

    union(x, y) {
        const rootX = this.find(x);
        const rootY = this.find(y);

        if (rootX !== rootY) {
            if (this.rank[rootX] < this.rank[rootY]) {
                this.parent[rootX] = rootY;
            } else if (this.rank[rootX] > this.rank[rootY]) {
                this.parent[rootY] = rootX;
            } else {
                this.parent[rootY] = rootX;
                this.rank[rootX]++;
            }
        }
    }
}
```

15. Intervals

When to use: Scheduling, merging ranges, overlap detection

Key techniques: Sort by start/end time, merge conditions

Time: O(n log n), Space: O(n)

```
// Template: Merge Intervals
function merge(intervals) {
    if (!intervals.length) return [];

    intervals.sort((a, b) => a[0] - b[0]);
    const result = [intervals[0]];

    for (let i = 1; i < intervals.length; i++) {
        const current = intervals[i];
        const last = result[result.length - 1];

        if (current[0] <= last[1]) {
            last[1] = Math.max(last[1], current[1]);
        } else {
            result.push(current);
        }
    }
    return result;
}
```

Quick Reference: Time Complexity Cheat Sheet

Opera-tion	Array	Linked List	Stack	Queue	Hash Table	Binary Tree	Heap
Access	O(1)	O(n)	O(n)	O(n)	O(1)	O(log n)	O(n)
Search	O(n)	O(n)	O(n)	O(n)	O(1)	O(log n)	O(n)
Insert	O(n)	O(1)	O(1)	O(1)	O(1)	O(log n)	O(log n)
Delete	O(n)	O(1)	O(1)	O(1)	O(1)	O(log n)	O(log n)

Space Complexity Guidelines

- **O(1)**: Few variables, no additional data structures
- **O(log n)**: Recursion depth in balanced trees
- **O(n)**: Single array/hash table of input size
- **O(n^2)**: 2D matrix or nested loops with storage

Common Edge Cases to Test

1. **Empty input**: [], “”, null
2. **Single element**: [1], “a”
3. **Two elements**: [1,2], “ab”
4. **Duplicates**: [1,1,2], “aab”
5. **Sorted input**: [1,2,3], Already optimal
6. **Reverse sorted**: [3,2,1], Worst case
7. **All same elements**: [1,1,1], Edge patterns
8. **Large numbers**: Integer overflow cases
9. **Negative numbers**: Mixed positive/negative
10. **Boundary values**: 0, -1, MAX_INT

Remember: Practice these patterns until they become second nature. Focus on understanding the WHY behind each approach, not just memorizing code!