DSA Pattern Cheat Sheet (Java & Pseudo Templates)

1. Sliding Window

Problem Type: Find a subarray/window with given constraints (max/min sum, longest substring, etc.)

Template:

```
public int slidingWindow(int[] arr, int k) {
    int left = 0, sum = 0, maxSum = Integer.MIN_VALUE;

    for (int right = 0; right < arr.length; right++) {
        sum += arr[right];

        if (right - left + 1 >= k) { // Maintain window size k
            maxSum = Math.max(maxSum, sum);
            sum -= arr[left];
            left++;
        }
    }
    return maxSum;
}
```

2. Two Pointers

Problem Type: Sorted array, palindrome check, pairs with sum, etc.

Template:

```
public boolean twoPointers(int[] arr, int target) {
    int left = 0, right = arr.length - 1;

    while (left < right) {
        int sum = arr[left] + arr[right];
        if (sum == target) return true;
        else if (sum < target) left++;
        else right--;
    }
    return false;
}
```

3. Fast & Slow Pointers (Cycle Detection)

Problem Type: Detect cycle in linked list, middle of list, etc.

Template:

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

4. Binary Search

Problem Type: Find an element in a sorted array, lower/upper bounds

Template:

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

5. Prefix Sum

Problem Type: Find sum in a range efficiently

Template:

```
public int[] prefixSum(int[] arr) {
    int n = arr.length;
    int[] prefix = new int[n];
    prefix[0] = arr[0];
    for (int i = 1; i < n; i++) {
        prefix[i] = prefix[i - 1] + arr[i];
    }
    return prefix;
}
```

6. Kadane's Algorithm (Max Subarray Sum)

Problem Type: Maximum subarray sum in O(n)

Template:

```
public int maxSubArray(int[] arr) {
    int maxSum = arr[0], currentSum = arr[0];
    for (int i = 1; i < arr.length; i++) {
        currentSum = Math.max(arr[i], currentSum + arr[i]);
        maxSum = Math.max(maxSum, currentSum);
    }
    return maxSum;
}
```

7. Backtracking (Subset/Permutation Generation)

Problem Type: Generate all subsets, permutations, combinations

Template:

Template:

```
public void backtrack(List<List<Integer>> result, List<Integer> temp, int[] nums
    result.add(new ArrayList<>(temp));
    for (int i = start; i < nums.length; i++) {
        temp.add(nums[i]);
        backtrack(result, temp, nums, i + 1);
        temp.remove(temp.size() - 1);
    }
}
```

8. Breadth-First Search (BFS)

Problem Type: Shortest path, level order traversal

Template:

```
public void bfs(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();
        System.out.print(node.val + " ");
        if (node.left != null) queue.offer(node.left);
        if (node.right != null) queue.offer(node.right);
    }
}
```

9. Depth-First Search (DFS)

Problem Type: Graph traversal, tree traversal

Template:

```
public void dfs(TreeNode node) {
    if (node == null) return;
    System.out.print(node.val + " ");
    dfs(node.left);
    dfs(node.right);
}
```

10. Dijkstra's Algorithm (Shortest Path in Graphs)

Problem Type: Shortest path in weighted graphs

Template:

```
public int[] dijkstra(int n, List<int[]>[] graph, int src) {
    int[] dist = new int[n];
    Arrays.fill(dist, Integer.MAX_VALUE);
    PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a
    pq.offer(new int[]{src, 0});
    dist[src] = 0;

    while (!pq.isEmpty()) {
        int[] node = pq.poll();
        int u = node[0], d = node[1];
        if (d > dist[u]) continue;

        for (int[] neighbor : graph[u]) {
            int v = neighbor[0], weight = neighbor[1];
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.offer(new int[]{v, dist[v]});
            }
        }
    }
    return dist;
}
```

🔥 Usage:

- This cheat sheet provides common DSA patterns with Java solutions.

- Modify templates as per problem constraints.
- Optimize based on input size and edge cases.

Keep Practicing! 🚀