Cheat Sheet: Two Pointer Technique, 2-sum & 3-sum in Java (DSA)

1. Two Pointer Technique

Concept:
Use two indices (pointers) to traverse a structure from opposite ends (or with different speeds). Typically applied to sorted arrays or sequences.

When to Use:

- Sorted arrays
- Subarray problems
- Avoid nested loops
- Find combinations (pairs, triplets)

2. 2-Sum Problem

Problem:

Find two numbers in an array that sum up to a given `target`.

Approach 1: HashMap (for unsorted arrays)

```
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    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 new int[0];
}
```

- □ Time: O(n)
- □ Space: O(n)

Approach 2: Two Pointers (for sorted arrays)

```
public int[] twoSumSorted(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, right};
        else if (sum < target) left++;
        else right--;
    }
    return new int[0];
}
```

- Time: O(n)
- Space: O(1)

3. 3-Sum Problem

Problem:

Find all unique triplets in an array that sum up to 0.

Java Implementation:

```
public List<List<Integer>> threeSum(int[] nums) {
    Arrays.sort(nums);
    List<List<Integer>> res = new ArrayList<>();

    for (int i = 0; i < nums.length - 2; i++) {
        if (i > 0 && nums[i] == nums[i-1]) continue;
        int left = i + 1, right = nums.length - 1;

        while (left < right) {
            int sum = nums[i] + nums[left] + nums[right];

            if (sum == 0) {
                res.add(Arrays.asList(nums[i], nums[left], nums[right]));
                while (left < right && nums[left] == nums[left + 1]) left++;
                while (left < right && nums[right] == nums[right - 1]) right-
-;
                left++; right--;
            } else if (sum < 0) {
                left++;
            } else {
                right--;
            }
        }
    }
    return res;
}
```

- Time: O(n²)
- Space: O(1) (excluding output)

Comparison Table

Problem	Input Type	Time Complexity	Technique	Space
2-Sum	Unsorted	O(n)	HashMap	O(n)
2-Sum	Sorted	O(n)	Two Pointers	O(1)
3-Sum	Unsorted	O(n²)	Sorting + 2Ptr	O(1)

Use Cases of Two Pointers

- 2-Sum in sorted array
- 3-Sum/4-Sum problems
- Container With Most Water
- Trapping Rain Water
- Remove Duplicates from Sorted Array
- Move Zeroes to End
- Longest Substring Without Repeating Characters (with sliding variant)

More Info: Two Pointers, 4-Sum, K-Sum & Variations in Java (DSA)

4. 4-Sum Problem

Problem:

Find all unique quadruplets in an array that sum up to a target.

Java Implementation:

```
public List<List<Integer>> fourSum(int[] nums, int target) {
    Arrays.sort(nums);
    List<List<Integer>> res = new ArrayList<>();

    for (int i = 0; i < nums.length - 3; i++) {
        if (i > 0 && nums[i] == nums[i - 1]) continue;

        for (int j = i + 1; j < nums.length - 2; j++) {
```

```
            if (j > i + 1 && nums[j] == nums[j - 1]) continue;

            int left = j + 1, right = nums.length - 1;

            while (left < right) {
                long sum = (long) nums[i] + nums[j] + nums[left] +
nums[right];

                if (sum == target) {
                    res.add(Arrays.asList(nums[i], nums[j], nums[left],
nums[right]));
                    while (left < right && nums[left] == nums[left + 1])
left++;
                    while (left < right && nums[right] == nums[right - 1])
right--;
                    left++; right--;
                } else if (sum < target) {
                    left++;
                } else {
                    right--;
                }
            }
        }
    }
    return res;
}
```

- Time: $O(n^3)$
- Space: O(1) (excluding output)

5. K-Sum (Generalized)

Problem:

Find all unique k-element combinations that sum to a target.

✓Recursive Approach:

```
public List<List<Integer>> kSum(int[] nums, int k, int target) {
    Arrays.sort(nums);
    return kSumHelper(nums, 0, k, target);
}

private List<List<Integer>> kSumHelper(int[] nums, int start, int k, int
target) {
    List<List<Integer>> res = new ArrayList<>();
    if (k == 2) {
        int left = start, right = nums.length - 1;
        while (left < right) {
```

```
            int sum = nums[left] + nums[right];
            if (sum == target) {
                res.add(Arrays.asList(nums[left], nums[right]));
                while (left < right && nums[left] == nums[left + 1]) left++;
                while (left < right && nums[right] == nums[right - 1]) right-
-;
                left++; right--;
            } else if (sum < target) {
                left++;
            } else {
                right--;
            }
        }
    } else {
        for (int i = start; i <= nums.length - k; i++) {
            if (i > start && nums[i] == nums[i - 1]) continue;
            for (List<Integer> subset : kSumHelper(nums, i + 1, k - 1, target
- nums[i])) {
                List<Integer> list = new ArrayList<>();
                list.add(nums[i]);
                list.addAll(subset);
                res.add(list);
            }
        }
    }
    return res;
}
```

- Time: O(n^(k-1))
- Space: O(k) recursive stack (excluding output)

6. Variations of Two Pointers

Variation	Description	Use Case
Opposite Ends	One pointer at start, one at end	2-Sum, Container With Most Water
Fixed + Moving	One fixed pointer, one moves	3-Sum
Slow/Fast (Tortoise-Hare)	Slow and fast pointer for cycle detection	Linked List cycle detection
Sliding Window	Maintain a window with two pointers	Longest Substring Without Repeat
Shrinking Window	Shrink window conditionally	Minimum Window Substring

Variation	Description	Use Case
Same Direction	Both pointers move forward	Remove Duplicates in-place

Practice Questions (LeetCode Style)

Problem	Link	Tags
Two Sum	1. Two Sum	HashMap, Array
Two Sum II	167. Two Sum II - Input array is sorted	Two Pointers
3Sum	15. 3Sum	Sorting, Two Pointers
4Sum	18. 4Sum	Sorting, Two Pointers
Container With Most Water	11. Container With Most Water	Two Pointers
Trapping Rain Water	42. Trapping Rain Water	Two Pointers, Stack
Longest Substring Without Repeating Characters	3. Longest Substring	Sliding Window