Cheat Sheet: Classic Binary Search & Condition-Based Problems (Java - DSA)

1. Introduction

Binary Search is a **Divide and Conquer** algorithm that reduces the problem size by half at each step. It works on **sorted arrays** (or conditions that allow simulating a sorted structure).

2. Problem Statement (Classic Binary Search)

Given a **sorted array** and a **target value**, return the **index** of the target if found. Otherwise, return -1.

3. Classic Binary Search Code (Iterative)

```
public int binarySearch(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;
}
```

Time and Space:

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

4. Binary Search on Condition-Based Problems

Examples:

- First/Last Occurrence of an element
- Search in Rotated Sorted Array
- Minimum in Rotated Sorted Array
- Allocate Minimum Pages (Books Allocation)
- Aggressive Cows (Minimize Max Distance)
- Koko Eating Bananas
- Capacity to Ship Packages

Template:

```
public int solveConditionProblem(int[] nums) {
```

```
    int low = 0, high = nums.length - 1;
    int answer = -1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (isConditionSatisfied(mid)) {
            answer = mid;
            high = mid - 1; // or low = mid + 1 depending on condition
        } else {
            low = mid + 1; // or high = mid - 1
        }
    }
    return answer;
}
```

Key: Define a monotonic condition that splits the search space into `True` and `False`.

5. Example: First Occurrence of Element

```
public int firstOccurrence(int[] nums, int target) {
    int low = 0, high = nums.length - 1, ans = -1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (nums[mid] == target) {
            ans = mid;
            high = mid - 1;
        } else if (nums[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return ans;
}
```

6. Applications

- **Efficient searching** in sorted arrays
- Solving **optimization problems** (min/max conditions)
- Finding **bounds** (lower bound, upper bound)
- Decision-based problems using **binary answer space**

7. Practice Links

- Leetcode 704: Binary Search
- TakeUForward: Binary Search

8. Tips

- Use `mid = low + (high - low) / 2` to avoid overflow
- Always check whether to update answer before moving pointers
- Understand when to move `low = mid + 1` or `high = mid - 1`
- Check for **sorted halves** in rotated arrays