1. **find target on sorted array (Leet Code 704)**

```
class Solution {
  public int search(int[] nums, int target) {

    return binarySearch(0, nums.length - 1, nums, target);

  }

  public int binarySearch(int left, int right, int[] nums, int target) {
    if (left > right) {
      return -1;
    }
    int middle = (right + left) / 2;
    if(nums[middle] == target) {
      return middle;
    }
    if(target < nums[middle]) {
      return binarySearch(left, middle - 1, nums, target);
    }
     return binarySearch(middle + 1, right, nums, target);

  }
}
```

Approach:

- **Divide the search space:** Use binary search on the sorted array by repeatedly halving the search range (left to right).
- **Find the middle element:** Calculate middle = (left + right) / 2.
- **Compare with target:**
    - If nums[middle] == target, return the index.
    - If target < nums[middle], search the **left half** (left to middle - 1).
    - Otherwise, search the **right half** (middle + 1 to right).
- **Stop condition:** If left > right, the element doesn't exist in the array, return -1.

2. **find target on sorted rotated array (Leet Code 33)**

```
class Solution {
  public int search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;

    while (left <= right) {
      int mid = (left + right) / 2;

      if (nums[mid] == target) {
        return mid;
      }

      if (nums[left] <= nums[mid]) {
        if (target >= nums[left] && target < nums[mid]) {
          right = mid - 1;
        } else {
          left = mid + 1;
        }
      }

      else {
        if (target > nums[mid] && target <= nums[right]) {
          left = mid + 1;
        } else {
          right = mid - 1;
        }
      }
    }

    return -1;
  }
}
```

Approach:

- **Binary search loop:** Start with left = 0, right = n-1 and calculate mid. If nums[mid] == target, return mid.
- **Check which half is sorted:**

If nums[left] <= nums[mid], then **left half is sorted**.
Else, the **right half is sorted**.

- **Decide where target lies:**
  If target is within the sorted half's range, shrink search space to that half.
  Otherwise, search in the other half.
- **Continue until found or exhausted:**
  Keep adjusting left and right until the target is found, otherwise return -1.

3. **find minimum in rotated sorted array (Leet Code 153)**

```
class Solution {
  public int findMin(int[] nums) {
    int low = 0, high = nums.length - 1;

    while (low < high) {
      int mid = low + (high - low) / 2;

      if (nums[mid] > nums[high]) {

        low = mid + 1;
      } else {

        high = mid;
      }
    }

    return nums[low];
  }
}
```

Approach:

- **Check rotation:** If nums[0] < nums[n-1], the array is not rotated → return nums[0].
- **Binary search idea:** Compare nums[mid] with nums[high] to decide which half has the minimum.
- **Move pointers:**
    - If nums[mid] > nums[high], minimum lies in the **right half** → low = mid + 1.
    - Else, minimum lies in the **left half (including mid)** → high = mid.
- **Stop condition:** When low == high, that index is the minimum element.

4. **median of two sorted array(Leet Code 4)**

```
class Solution {
  public double findMedianSortedArrays(int[] nums1, int[] nums2) {
    if (nums1.length > nums2.length) {
      return findMedianSortedArrays(nums2, nums1);      }

    int m = nums1.length, n = nums2.length;
    int low = 0, high = m;

    while (low <= high) {
      int i = (low + high) / 2;
      int j = (m + n + 1) / 2 - i;

      int left1 = (i == 0) ? Integer.MIN_VALUE : nums1[i - 1];
      int right1 = (i == m) ? Integer.MAX_VALUE : nums1[i];
      int left2 = (j == 0) ? Integer.MIN_VALUE : nums2[j - 1];
      int right2 = (j == n) ? Integer.MAX_VALUE : nums2[j];

      if (left1 <= right2 && left2 <= right1) {
        if ((m + n) % 2 == 0) {
          return (Math.max(left1, left2) + Math.min(right1, right2)) / 2.0;
        } else {
```

```
            return Math.max(left1, left2);
          }
        } else if (left1 > right2) {
          high = i - 1;
        } else {
          low = i + 1;
        }
      }
      return 0.0;
  }
}
```

Approach**:**

- **Ensure nums1 is smaller** → Binary search on the smaller array to reduce complexity.
- **Partition both arrays:**
  Pick an index i in nums1, and compute j = (m + n + 1)/2 - i in nums2 so that left and right partitions together split the arrays in half.
- **Check validity of partition:**
  - If nums1[i-1] <= nums2[j] **and** nums2[j-1] <= nums1[i], partition is correct.
  - Otherwise, adjust binary search:
    - If nums1[i-1] > nums2[j] → move left.
    - Else → move right.
- **Median calculation:**
  - If (m+n) is odd → max(leftPart) is the median.
  - If (m+n) is even → median = (max(leftPart) + min(rightPart)) / 2.