Experiment - 4(A)

Student Name: DEVENSH YADAV   **UID:** 22BCS16082
Branch: CSE   **Section/Group:** NTPP_602-A
Semester: 6   **Date of Performance:** 13-02-25
Subject Name: Advanced Programming Lab-2   **Subject Code:** 22CSH-359

1. **Title:** Sorting and Searching (Merge Sorted Array)
   https://leetcode.com/problems/merge-sorted-array/

   2. **Objective:** To merge two sorted arrays nums1 and nums2 into one sorted array, while ensuring that the result is stored in nums1. The algorithm should be efficient and merge the arrays in-place.

3. **Algorithm:**
   - Start with three pointers: i for the last element of nums1 (index m-1), j for the last element of nums2 (index n-1), and k for the last position in nums1 (index m+n-1).
   - Compare the elements at nums1[i] and nums2[j]. Place the larger of the two at nums1[k], and decrement k.
   - If nums1[i] is greater, move i left. If nums2[j] is greater, move j left.
   - If any elements are left in nums2, copy them to nums1 (since nums1 already contains the first m elements in sorted order).

   - Continue the process until all elements from both arrays are merged.

4. **Implementation/Code:**

```python
class Solution:
def merge(self, nums1, m, nums2, n):
    # Start from the end of both arrays
    i, j, k = m - 1, n - 1, m + n - 1

    # Merge in reverse order
    while i >= 0 and j >= 0:
        if nums1[i] > nums2[j]:
            nums1[k] = nums1[i]
            i -= 1
        else:
            nums1[k] = nums2[j]
            j -= 1
        k -= 1
```
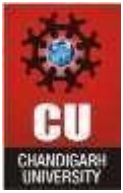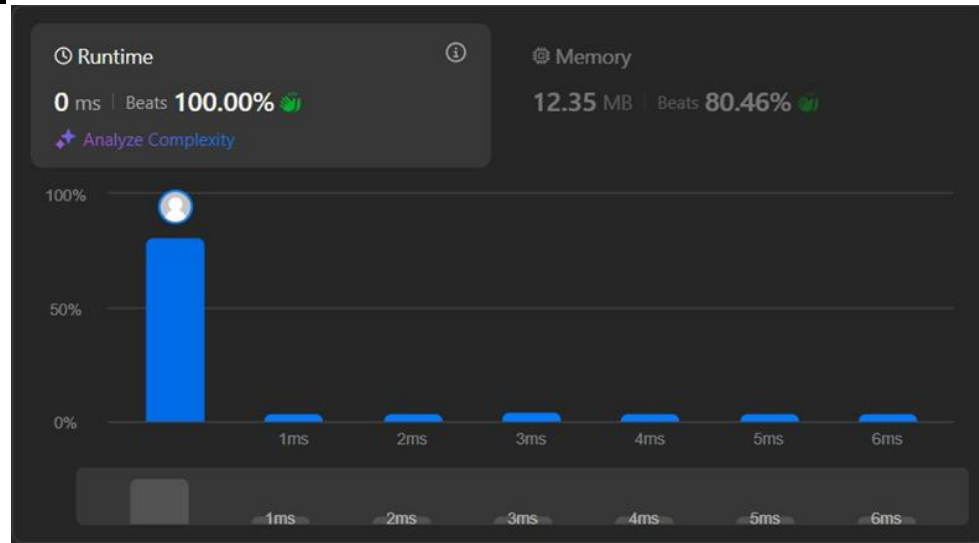
```
# If nums2 is not fully traversed, copy the remaining elements
while j >= 0:
    nums1[k] = nums2[j]
    j -= 1
    k -= 1
```

5. Output:

6. Time Complexity: O(m+n)

7. Space Complexity: O(1)

8. Learning Outcomes:

- **Array Manipulation**: Understanding how to manipulate arrays in-place.
- **Two Pointer Technique**: Using multiple pointers to traverse arrays efficiently and merge them.
- **In-place Sorting**: Learning how to merge sorted arrays without extra space.
- **Optimization**: Achieving linear time complexity by processing the arrays from the end, avoiding unnecessary moves.
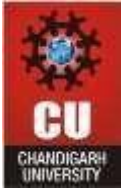
Experiment 4(B)

1. **Title:** First Bad Colors (https://leetcode.com/problems/first-bad-version/)

2. **Objective:** The task is to find the first bad version in a series of product versions, where all versions after the first bad version are also bad. You should minimize the number of calls to the isBadVersion(version) API.

3. **Algorithm:**
   - Initialize two pointers `left` as 1 and `right` as `n`.
   - Perform a binary search:

      - While `left` is less than `right`, calculate the mid-point `mid = (left + right) // 2`.
      - If `isBadVersion(mid)` is `True`, then the first bad version is at `mid` or to the left of it, so update `right = mid`.
      - If `isBadVersion(mid)` is `False`, then the first bad version must be to the right, so update `left = mid + 1`.

   - When the loop terminates, `left` will point to the first bad version.

4. **Implementation/Code:**

```
class Solution:

    def firstBadVersion(self, n):

        left, right = 1, n


        while left < right:

            mid = left + (right - left) // 2

            if isBadVersion(mid):

                right = mid  # Potential first bad version

            else:

                left = mid + 1  # Bad version must be on the right


        return left
```
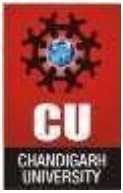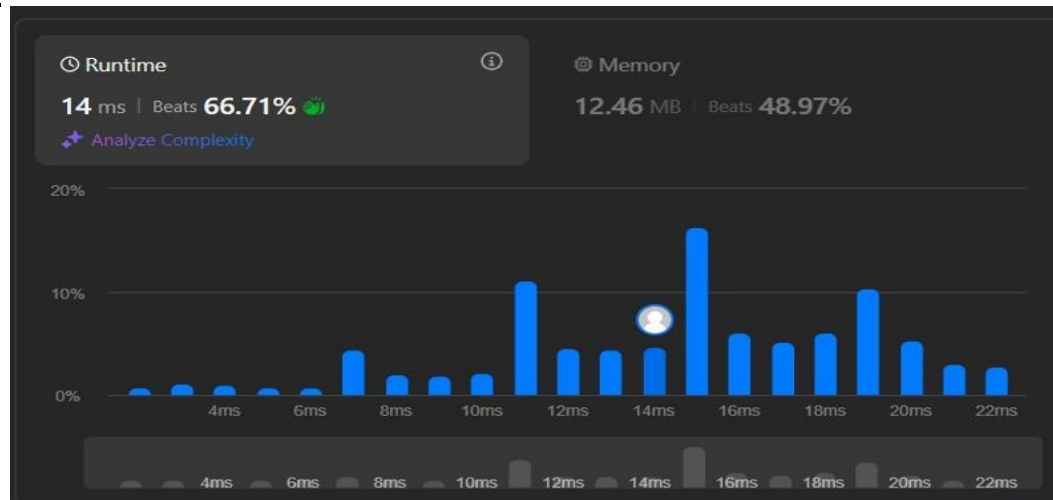
6. **Output:**

8. **Time Complexity:** O(log n)   **9.** **Space Complexity:** O(1)

9. Learning Outcomes:

1. **Binary Search**: Understanding and applying the binary search algorithm to reduce the number of checks (calls to isBadVersion).

2. **Optimization**: Minimizing the number of calls to an external API, which is a key performance concern in real-world applications.

3. **Problem Solving**: Effectively narrowing down the range for the first bad version using a divide-and-conquer strategy.
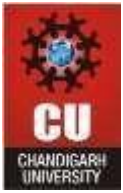
Experiment 4(C)

1. **Title:** Sort Colors (https://leetcode.com/problems/sort-colors/)

2. **Objective:** To sort an array containing three distinct values (0, 1, 2) in place, where 0 represents red, 1 represents white, and 2 represents blue. The array should be sorted in such a way that all 0s come first, followed by all 1s, and then all 2s.
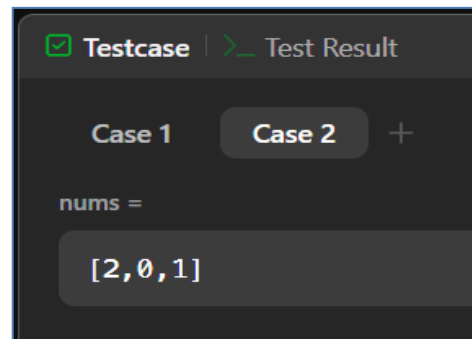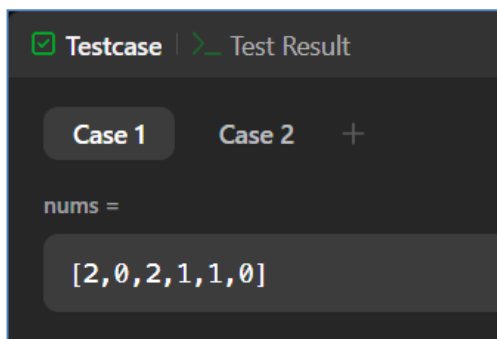
4. Algorithm:

- Initialize three pointers: `low` (starting at the beginning of the array), `mid` (also starting at the beginning), and `high` (starting at the end of the array).
- Traverse the array using the `mid` pointer:

    - If `nums[mid]` is 0, swap it with `nums[low]` and increment both `low` and `mid`.
    - If `nums[mid]` is 1, just increment `mid`.
    - If `nums[mid]` is 2, swap it with `nums[high]` and decrement `high`.

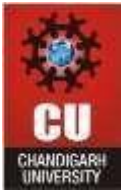- Continue this process until `mid` exceeds `high`.

5. Implementation/Code:

```python
class Solution:
    def sortColors(self, nums):
        low, mid, high = 0, 0, len(nums) - 1

        while mid <= high:
            if nums[mid] == 0:
                nums[low], nums[mid] = nums[mid], nums[low]
                low += 1
                mid += 1
            elif nums[mid] == 1:
                mid += 1
            else:
                nums[mid], nums[high] = nums[high], nums[mid]
                high -= 1
```
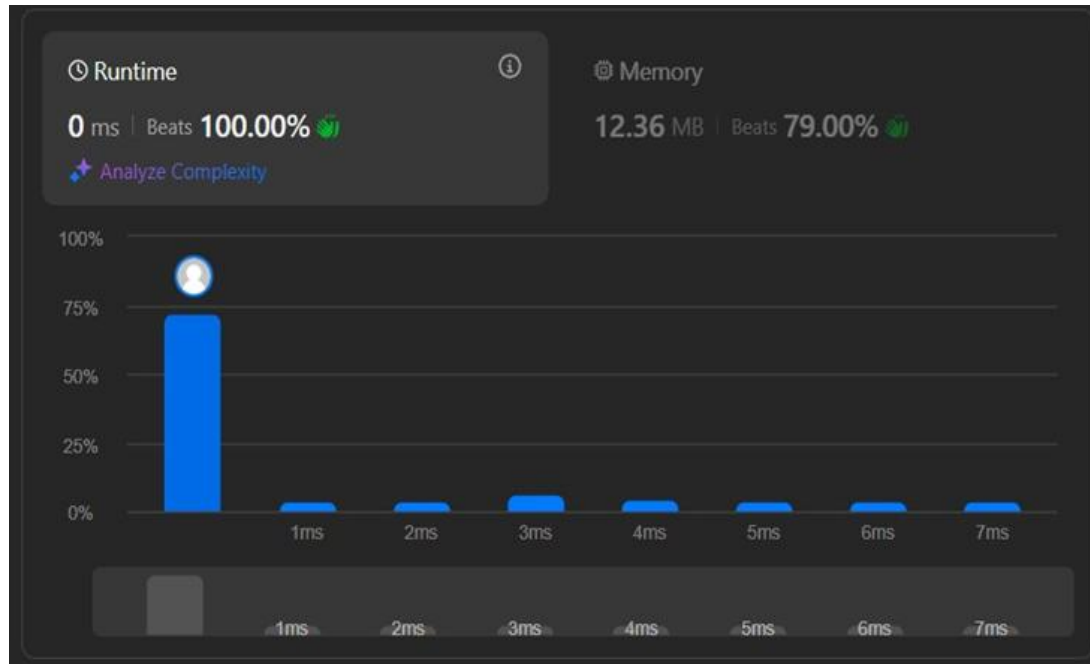
6. Output:

✓ Testcase >_ Test Result

Case 1   Case 2   +

nums =

[2,0,2,1,1,0]

✓ Testcase >_ Test Result

Case 1   Case 2   +

nums =

[2,0,1]

8. Time Complexity: O(N)　　　　**9. Space Complexity:** O(1)

10.　Learning Outcomes:

1. **Three-way Partitioning**: Understanding how to partition an array into three groups based on a specific condition (here, 0, 1, 2).
2. **In-place Sorting**: Sorting without using extra space for another array.
3. **Two Pointer Technique**: Efficient use of multiple pointers (low, mid, high) to traverse the array in a single pass.
4. **Optimization**: Solving the problem in linear time (O(n)) and constant space (O(1)).