DESIGN AND ANALYSIS OF ALGORITHMS

AKSHAY H BANGERA

NNM23IS010

TASK-2

1. LEETCODE 75: SORT COLORS

Problem:

Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Complexity:

Complexity Analysis

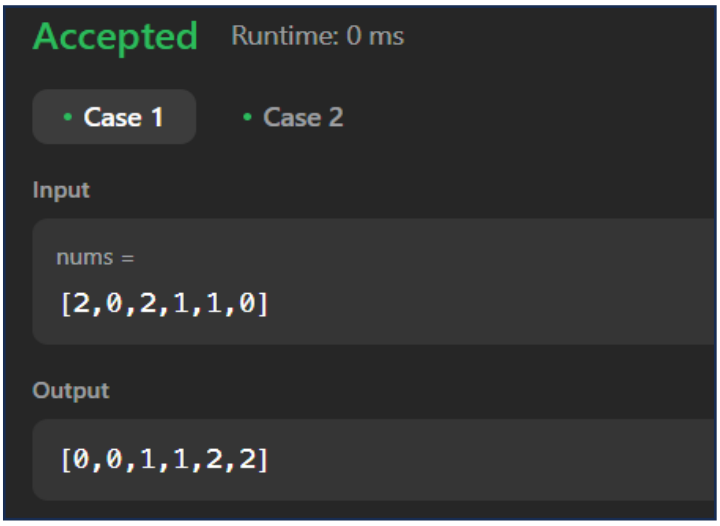
- Time Complexity: O(n) → Each element is processed at most once.
- Space Complexity: O(1) → Sorting is done in-place.

JAVA Code:

```
class Solution {
   public void sortColors(int[] nums) {
boolean swapped;
      for(int i=0; i<nums.length-1; i++) {
swapped = false;
         for(int j=0; j<nums.length-i-1; j++) {
if(nums[j] > nums[j+1]) {                int
temp = nums[j];
               nums[j] = nums[j+1];
   nums[j+1] = temp;
   swapped = true;
            }
```

```
        }
        if(!swapped) break;
      }
    }
}
```

OUTPUT:



Observations:

1. Bubble Sort works but is inefficient ($O(n^2)$) because it makes unnecessary swaps.
2. Early stopping optimization (checking swapped flag) helps if the array is already sorted.
3. The Dutch National Flag Algorithm (O(n)) is much faster for this problem.

2. LEETCODE 162: FIND PEAK ELEMENTS

Problem:

A peak element is an element that is strictly greater than its neighbors.

Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

Complexity:

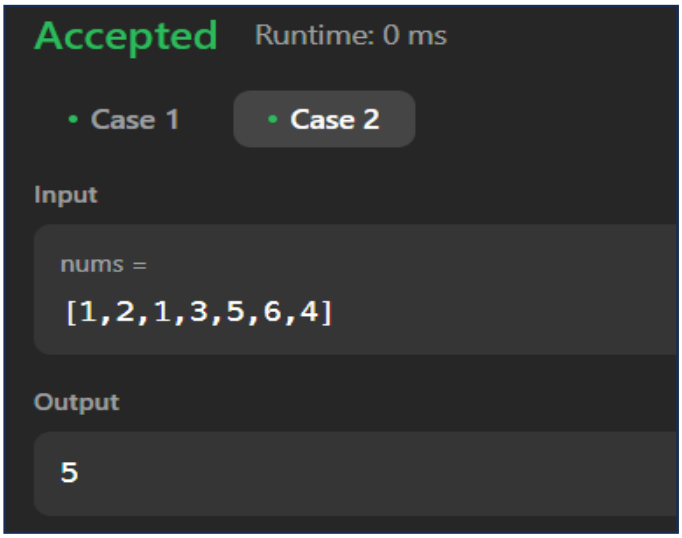
Complexity Analysis

- Time Complexity: O(log n) → The search space is halved in each step using Binary Search.
- Space Complexity: O(1) → Only a few pointers are used; no extra space is required.

C Code:

```
#include <stdio.h>

int findPeakElement(int nums[], int numsSize) {

int left = 0, right = numsSize - 1;     while (left <

right) {

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

if (nums[mid] > nums[mid + 1]) {

right = mid;

      } else {       left

   = mid + 1;

      }

   }

   return left;

}

void testFindPeak() {

   int nums[] = {1, 2, 1, 3, 5, 6, 4};     int n =

sizeof(nums) / sizeof(nums[0]);     int peakIndex =

findPeakElement(nums, n);     printf("Peak

element index: %d\n", peakIndex);

}
```

OUTPUT:



Observations:

1. Uses Binary Search (O(log n)) to efficiently find a peak element by halving the search space in each iteration.
2. Guaranteed to find a peak since an element is always considered greater than out-of-bound neighbours (-∞ assumption).
3. Space Complexity is O(1) as only a few integer variables (left, right, mid) are used, making it an in-place solution.

3.LEETCODE 189: ROTATE ARRAY

Problem:

Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.

Complexity:

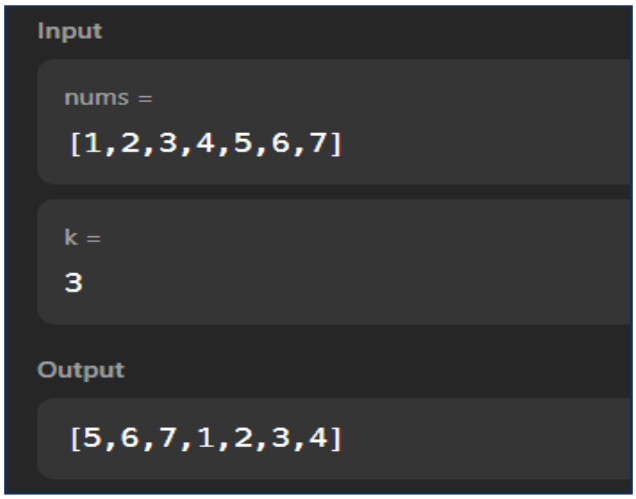
- Time Complexity: O(n) → Each element is processed at most once through three reversals.
- Space Complexity: O(1) → Rotations are done in-place without using extra storage.

C Code:

```
#include <stdio.h>

void reverse(int nums[], int start, int end) {

while (start < end) {        int temp =

nums[start];         nums[start] =

nums[end];         nums[end] = temp;

start++;         end--;

   }
```

```
}
void rotate(int nums[], int numsSize, int k) {     if
(numsSize == 0 || k % numsSize == 0) return;  k =
k % numsSize;  reverse(nums, 0, numsSize - 1);
reverse(nums, 0, k - 1); reverse(nums, k,
numsSize - 1);
}
void printArray(int nums[], int numsSize) {
for (int i = 0; i < numsSize; i++) {
printf("%d ", nums[i]);
   }
   printf("\n");
}
void testRotate() {
   int nums[] = {1, 2, 3, 4, 5, 6, 7};
int k = 3;
   int n = sizeof(nums) / sizeof(nums[0]);
printf("Original array: ");
printArray(nums, n);     rotate(nums, n,
k);     printf("Rotated array: ");
printArray(nums, n);
}
```

OUTPUT:



Observations:

1. The code efficiently rotates the array using the reverse method, achieving an O(n) time complexity and O(1) space complexity.
2. The three-step reversal approach correctly shifts elements by reversing the entire array, then the first k elements, and finally the remaining elements.

4.LEETCODE 442: FIND ALL DUPLICATES IN AN ARRAY

Problem:

Given an integer array nums of length n where all the integers of nums are in the range [1, n] and each integer appears at most twice, return an array of all the integers that appears twice.

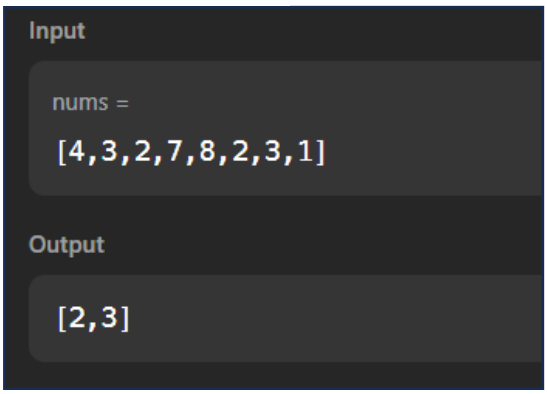
Complexity:

- Time Complexity: O(n) → Each element is processed at most once while marking and restoring values.
- Space Complexity: O(1) → The solution modifies the input array in-place without using extra storage (excluding the output list).

C Code:

```
#include <stdio.h> #include
<stdlib.h>
int* findDuplicates(int nums[], int numsSize, int* returnSize) {
   *returnSize = 0;
   int* result = (int*)malloc(numsSize * sizeof(int));
for (int i = 0; i < numsSize; i++) {        int index =
abs(nums[i]) - 1;          if (nums[index] < 0) {
          result[(*returnSize)++] = abs(nums[i]);
      } else {
          nums[index] = -nums[index];
      }
   }
   return result;
}
```

OUTPUT:



Observations:

1. The code uses index marking by negating elements to track visits, efficiently identifying duplicates in O(n) time.
2. It modifies the input array temporarily without extra space (excluding the output array), achieving O(1) space complexity.
3. The result array is dynamically allocated using malloc(), and returnSize keeps track of the number of duplicates found.