Introduction to LeetCode

1. Two Sum

Approach 1: Brute Force
Algorithm

The brute force approach is simple. Loop through each element x and find if there is another value that equals to target−x .

Implementation
```
class Solution {
public:
    vector<int> twoSum(vector<int> &nums, int target) {
        for (int i = 0; i < nums.size(); i++) {
            for (int j = i + 1; j < nums.size(); j++) {
                if (nums[j] == target - nums[i]) {
                    return {i, j};
                }
            }
        }
        // Return an empty vector if no solution is found
        return {};
    }
};
```

Complexity Analysis

Time complexity: O(n^2)

For each element, we try to find its complement by looping through the rest of the array which takes O(n) time. Therefore, the time complexity is O(n^2) .

Space complexity: O(1).
The space required does not depend on the size of the input array, so only constant space is used.

Approach 2: Two-pass Hash Table

Intuition
To improve our runtime complexity, we need a more efficient way to check if the complement exists in the array. If the complement exists, we need to get its index. What is the best way to maintain a mapping of each element in the array to its index? A hash table.

We can reduce the lookup time from O(n) to O(1) by trading space for speed. A hash table is well suited for this purpose because it supports fast lookup in near constant time. I say "near" because if a collision occurred, a lookup could degenerate to O(n) time.

However, lookup in a hash table should be amortized O(1) time as long as the hash function was chosen carefully.

Algorithm
A simple implementation uses two iterations. In the first iteration, we add each element's value as a key and its index as a value to the hash table. Then, in the second iteration, we check if each element's complement (target−nums[i]target - nums[i]target−nums[i]) exists in the hash table. If it does exist, we return current element's index and its complement's index. Beware that the complement must not be nums[i]nums[i]nums[i] itself!

Implementation
```
class Solution {
public:
    vector<int> twoSum(vector<int> &nums, int target) {
        unordered_map<int, int> hash;
        for (int i = 0; i < nums.size(); i++) {
            hash[nums[i]] = i;
        }
        for (int i = 0; i < nums.size(); i++) {
            int complement = target - nums[i];
            if (hash.find(complement) != hash.end() &&
hash[complement] != i) {
                return {i, hash[complement]};
            }
        }
        return {};
    }
};
```

Complexity Analysis
Time complexity: O(n)
We traverse the list containing n elements exactly twice. Since the hash table reduces the lookup time to O(1), the overall time complexity is O(n).

Space complexity: O(n).
The extra space required depends on the number of items stored in the hash table, which stores exactly n elements.

Approach 3: One-pass Hash Table

Algorithm
It turns out we can do it in one-pass. While we are iterating and inserting elements into the hash table, we also look back to check if the current element's complement already exists in the hash table. If it exists, we have found a solution and return the indices immediately.

Implementation
```
class Solution {
```

```
public:
    vector<int> twoSum(vector<int> &nums, int target) {
        unordered_map<int, int> hash;
        for (int i = 0; i < nums.size(); ++i) {
            int complement = target - nums[i];
            if (hash.find(complement) != hash.end()) {
                return {hash[complement], i};
            }
            hash[nums[i]] = i;
        }
        return {};
    }
};
```

Complexity Analysis
Time complexity: O(n).
We traverse the list containing n elements only once.
Each lookup in the table costs only O(1) time.

Space complexity: O(n).
The extra space required depends on the number of items stored in the hash table, which stores at most n elements.

2. Subarray Sums Divisible by K

Overview
The problem presents an integer array nums and an integer k. Our task is to find the number of non-empty subarrays that have a sum divisible by k.

Approach: Prefix Sums and Counting
Intuition
The problem is based on the concept of using prefix sums to compute the total number of subarrays that are divisible by k. A prefix sum array for nums is another array prefixSum of the same size as nums, such that the value of prefixSum[i] is the sum of all elements of the nums array from index 0 to index i, i.e., nums[0] + nums[1] + nums[2] + . . . + nums[i].

The sum of the subarray i + 1 to j (inclusive) is computed by prefixSum[j] - prefixSum[i]. Using this, we can count the number of pairs that exist for every pair (i, j) where i < j and (prefixSum[j] - prefix[i]) % k = 0. There are n * (n - 1) / 2 pairs for an array of length n (pick any two from n). As a result, while this will provide the correct answer for every test case, it will take O(n^2) time, indicating that the time limit has been exceeded (TLE).

The character % is the modulo operator.

Let's try to use the information with respect to the remainders of every prefix sum and try to optimize the above approach.

As stated previously, our task is to determine the number of pairs (i, j) where i < j and (prefixSum[j] - prefix[i]) % k = 0. This equality can only be true if prefixSum[i] % k = prefixSum[j] % k. We will demonstrate this property.

We can express any number as number = divisor × quotient + remainder. For example, 13 when divided by 3 can be written as 13 = 3 * 4 + 1. So we can express:
a) prefixSum[i] as prefixSum[i] = A * k + R0 where A is the quotient and R0 is the remainder when divided by k.
b) Similarly, prefixSum[j] = B * k + R1 where B is the quotient and R1 is the remainder when divided by k.

We can write, prefixSum[j] - prefixSum[i] = k * (B - A) + (R1 - R0). The first term (k * (B - A)) is divisible by k, so for the entire expression to be divisible by k, R1 - R0 must also be divisible by k. This gives us an equation R1 - R0 = C * k, where C is some integer. Rearranging it yields R1 = C * k + R0. Because the values of R0 and R1 will be between 0 and k - 1, R1 cannot be greater than k. So the only possible value for C is 0, leading to R0 = R1, which proves the above property. If C > 0, then the RHS would be at least k, but as stated the LHS (R1) is between 0 and k - 1.

Let's say a subarray ranging from index 0 to index j has a remainder R when the sum of its elements (prefix sum) is divided by k. Our task now becomes to figure out how many subarrays 0..i exist with i < j having the same remainder R when their prefix sum is divided by k. So, we need to maintain the count of remainders while moving in the array.

We start with an integer prefixMod = 0 to store the remainder when the sum of the elements of a subarray that start from index 0 is divided by k. We do not need the prefix sum array, since we only need to maintain the count of each remainder (0 to k - 1) so far. To maintain the count of the remainders, we initialize an array modGroups[k], where modGroups[R] stores the number of times R was the remainder so far.

We iterate over all the elements starting from index 0. We set prefixMod = (prefixMod + num[i] % k + k) % k for each element at index i to find the remainder of the sum of the subarray ranging from index 0 to index i when divided by k. The + k is needed to handle negative numbers. We can then add the number of subarrays previously seen having the same remainder prefixMod to cancel out the remainder. The total number of these arrays is in modGroups[prefixMod]. In the end, we increment the count of modGroups[R] by one to include the current subarray with the remainder R for future matches.

Till now, we chose some previous subarrays (if they exist) to delete the remainder from the existing array formed till index i when the sum of its elements is divided by k. What if the sum of the elements of the array till index i is divisible by k and we don't need another subarray to delete the remainder?

To count the complete subarray from index 0 to index i, we also initialize modGroups[0] = 1 at the start so that if a complete subarray from index 0 to the current index is divisible by k, we include the complete array in the count of modGroups[0]. It is set to start with 1 to cover the complete subarray case. For example, let's assume we are index i. Say, we have previously encountered three subarrays from index 0 to some index j where j < i that were divisible by 'k'. Now, assume the sum of elements in the array up to index i is also divisible by k. So, we will have 4 options to form a subarray ending at index i that is divisible by k. Three of these come from choosing the subarrays (resulting in subarray j + 1, .., i that is divisble by k) that were divisble by k and one comes from choosing the complete subarray starting from index 0 till index i.

Algorithm

1. Initialize an integer prefixMod = 0 to store the remainder when the sum of the elements of a array till the current index when divided by k, and the answer variable result = 0 to store the number of subarrays divisible by k.
2. Initialize an array, modGroups[k] where modGroup[R] stores the number of subarrays encountered with the sum of elements having a remainder R when divided by k. Set modGroups[0] = 1.
3. Iterate over all the elements of num.

- For each index i, compute the prefix modulo as prefixMod = (prefixMod + num[i] % k + k) % k. We take modulo twice in (prefixMod + num[i] % k + k) % k to remove negative numbers since num[i] can be a negative number and the sum prefixMod + nums[i] % k can turn out to be negative. To remove the negative number we add k to make it positive and then takes its modulo again with k.
- Add the number of subarrays encountered till now that have the same remainder to the result: result = result + modGroups[prefixMod].
- In the end, we include the remainder of the subarray in the modGroups, i.e., modGroups[prefixMod] = modGroups[prefixMod] + 1 for future matches.

1. Return result.

Implementation

```
class Solution {
public:
    int subarraysDivByK(vector<int>& nums, int k) {
        int n = nums.size();
        int prefixMod = 0, result = 0;

        // There are k mod groups 0...k-1.
        vector<int> modGroups(k);
        modGroups[0] = 1;

        for (int num : nums) {
            // Take modulo twice to avoid negative
remainders.
            prefixMod = (prefixMod + num % k + k) % k;
            // Add the count of subarrays that have the same
remainder as the current
            // one to cancel out the remainders.
            result += modGroups[prefixMod];
            modGroups[prefixMod]++;
        }

        return result;
    }
};
```

Complexity Analysis

Here, n is the length of nums and k is the given integer.

- Time complexity: O(n + k)

We require O(k) time to initialize the modGroups array. We also require O(n) time to iterate over all the elements of the nums array. The computation of the prefixSum and the calculation of the subarrays divisible by k take O(1) time for each index of the array.

- Space complexity: O(k)

We require O(k) space for the modGroups array.

3. Remove Element

Intuition
The intuition behind this solution is to iterate through the array and keep track of two pointers: index and i. The index pointer represents the position where the next non-target element should be placed, while the i pointer iterates through the array elements. By overwriting the target elements with non-target elements, the solution effectively removes all occurrences of the target value from the array.

Approach
- Initialize index to 0, which represents the current position for the next non-target element.

- Iterate through each element of the input array using the i pointer.
- For each element nums[i], check if it is equal to the target value.
    - If nums[i] is not equal to val, it means it is a non-target element.
    - Set nums[index] to nums[i] to store the non-target element at the current index position.
    - Increment index by 1 to move to the next position for the next non-target element.
- Continue this process until all elements in the array have been processed.
- Finally, return the value of index, which represents the length of the modified array.

Complexity
Time complexity: O(n)

Space complexity: O(1)

Code
```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int index = 0;
        for(int i = 0; i< nums.size(); i++){
            if(nums[i] != val){
                nums[index] = nums[i];
                index++;
            }
        }
        return index;
    }
};
```