Day 38/180

Two Pointer in C++ 📑

1. Segregate 0s and 1s

```cpp
class Solution {
   public:

   void segregate0and1(int arr[], int n) {

      int start = 0, end = n - 1;
      // Use a while loop to segregate the array until start pointer is less than end pointer.
      while(start < end) {

         // If the element in the start index of the array is 0, increment the start pointer.
         if(arr[start] == 0) start++;

         // If the element in the start index of the array is 1, then:
         else {

            // If the element in the end index of the array is 0:
            if(arr[end] == 0){

               // Swap both the elements at start and end indices.
               swap(arr[start], arr[end]);

               // After the swap, increment the start pointer and decrement the end pointer.
               start++, end--;
            }
            // If the element in the end index of the array is 1,
            // decrement the end pointer only.
            else end--;
         }
      }
   }
};
```

Code Explanation and Complexity of Segregate 0s and 1s

1. Two-Pointer Approach:
    - Initialize two pointers, start at the beginning, and end at the end of the array.
    - While start is less than end:
        - If the element at start is already 0, move start to the next position.
        - If the element at start is 1:
            - Check the element at the end.
            - If arr[end] is 0, swap elements at start and end.
            - Move start forward and end backward.
            - If arr[end] is also 1, move end backward.
2. Termination Condition:
    - Continue this process until start is not less than end.
3. Result:
    - The array is modified in-place, satisfying the condition of having all 0s on the left side and all 1s on the right side.
4. Time Complexity:
    - The algorithm iterates through the array once, and in each iteration, it either increments start or both increments start and decrements end. Therefore, the time complexity is O(N), where N is the length of the array.
5. Space Complexity:
    - The algorithm uses a constant amount of extra space (two pointers), so the space complexity is O(1).

2. Two Sum II - Input Array Is Sorted

```
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        vector<int> ans;
        int start = 0, end = numbers.size() - 1;
        while (start < end) {
            // Check if the sum of elements at 'start' and 'end' equals
the target
            if (numbers[start] + numbers[end] == target) {
                // If yes, push the indices (adjusted by 1) into the
result vector
                ans.push_back(start + 1);
                ans.push_back(end + 1);
                return ans; // Return the result as there is exactly one
solution
            } else if (numbers[start] + numbers[end] < target) {
                // If the sum is less than the target, move 'start' to
the next position
                start++;
            } else {
                // If the sum is greater than the target, move 'end' to
the previous position
                end--;
            }
        }

        // If no solution is found, return an empty vector
        return ans;
    }
};
```

Code Explanation and Complexity of Two Sum II

1. Two-Pointer Approach:
    - Initialize two pointers, start at the beginning, and end at the end of the array.
    - While start is less than end:
        - If the sum of elements at start and end is equal to the target:
            - Push the indices (adjusted by 1) into the result vector.
            - Return the result as there is exactly one solution.
        - If the sum is less than the target, move start to the next position.
        - If the sum is greater than the target, move end to the previous position.
2. Termination Condition:
    - Continue this process until start is not less than end.
3. Result:
    - The result vector contains the indices (adjusted by 1) of the two numbers that add up to the target.
4. Time Complexity:
    - The algorithm iterates through the array once, and in each iteration, either start or end is moved. Therefore, the time complexity is O(N), where N is the length of the array.
5. Space Complexity:
    - The algorithm uses only a constant amount of extra space (for variables and the result vector), so the space complexity is O(1).

3. Pair With Given Difference

```cpp
int Solution::solve(vector < int > & A, int B) {
  // Sort the input array in ascending order
  sort(A.begin(), A.end());

  int start = 0, end = 1, n = A.size();

  // If B is negative, convert it to positive for ease of comparison
  if (B < 0)
    B = B * -1;

  while (end < n) {
    // Check if the difference between elements at 'end' and 'start' is
equal to B

    if (A[end] - A[start] == B)
      return 1; // If yes, return 1 (pair with given difference found)

    // If the difference is less than B, move 'end' pointer forward
    else if (A[end] - A[start] < B)
      end++;

    // If the difference is greater than B, move 'start' pointer forward
    else
      start++;

    // Avoid an infinite loop when start and end are at the same index
    if (start == end)
      end++;
  }

  // If no such pair is found, return 0
  return 0;
}
```

Code Explanation and Complexity of Pair With Given Difference

1. Sorting:
   - The array A is sorted in ascending order. Sorting simplifies the process of finding pairs with a specific difference.
2. Two-Pointer Approach:

   - Initialize two pointers, start and end, initially pointing to the first two elements of the sorted array.
   - The goal is to find a pair with a difference equal to the given integer B.
   - While the 'end' pointer is within the bounds of the array:
     - Check if the difference between the elements at 'start' and 'end' is equal to B.
       - If yes, return 1, indicating that a pair with the given difference is found.
     - If the difference is less than B, move the 'end' pointer forward.
     - If the difference is greater than B, move the 'start' pointer forward.
     - Avoid an infinite loop by ensuring that 'start' and 'end' are not at the same index.
3. Result:
   - If the loop completes without finding a pair, return 0, indicating that no such pair exists.
4. Time Complexity:
   - The dominant factor in the time complexity is the sorting step, which takes O(N log N), where N is the size of the array. The subsequent two-pointer approach takes linear time, so the overall time complexity is O(N log N).
5. Space Complexity:
   - The algorithm uses only a constant amount of extra space (for variables), so the space complexity is O(1).

4. Product Pair

```
class Solution {
public:
    bool isProduct(int arr[], int n, long long x) {
        // Check if the array size is less than 2, as we need at least
two elements for a pair
        if (n < 2)
            return false;

        // Use an unordered_set to store encountered elements
        unordered_set<int> s;

        // Iterate through the array
        for (int i = 0; i < n; i++) {
            // Handle the special case when the current element is 0
            // Since x % 0 is undefined in C++, we need to handle it
explicitly
            if (arr[i] == 0) {
                if (x == 0)
                    return true; // If x is also 0, return true
                else
                    continue; // Otherwise, move to the next element
            }
            // Check if x/arr[i] exists in the hash set (unordered_set)
            // If yes, then we found a pair with the required product
            if (x % arr[i] == 0) {
                if (s.find(x / arr[i]) != s.end())
                    return true; // Found a pair
                else
                    s.insert(arr[i]); // Insert arr[i] into the set
            }
        }

        // No pair with the required product found
        return false;
    }
};
```

Code Explanation and Complexity of Product Pair

1. Algorithm:
    - Initialize an unordered_set (s) to keep track of elements encountered during the iteration.
    - Iterate through the array using a for loop.
2. Special Handling for Zero:
    - If the current element is zero (arr[i] == 0):
        - Check if x is also zero.
        - If yes, return true, as there exists a pair (0, 0) with the required product.
        - If no, skip to the next iteration to avoid division by zero.
3. Checking for Pair:
    - For non-zero elements (arr[i] != 0):
        - Check if x is divisible by the current element (x % arr[i] == 0).
        - If true, check if the result of the division (x / arr[i]) is already in the set (s.find(...)).
            - If yes, return true, as a pair with the required product is found.
            - If no, insert the current element into the set.
4. Result:
    - If the loop completes without finding a pair, return false.
5. Time Complexity:
    - The function iterates through the array once, performing constant-time operations for each element.
    - Therefore, the time complexity is O(N), where N is the size of the array.
6. Space Complexity:
    - The function uses an unordered_set (s) to store elements.
    - In the worst case, the set may store all elements of the array.
    - Therefore, the space complexity is O(N), where N is the size of the array.

5. Remove Duplicates from Sorted Array

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        // Initialize a variable 'j' to keep track of the position for
the next unique element
        int j = 1;

        // Iterate through the array starting from the second element
(index 1)
        for(int i = 1; i < nums.size(); i++) {
            // Check if the current element is different from the
previous one
            if(nums[i] != nums[i - 1]) {
                // If yes, update the array by placing the unique element
at position 'j'
                nums[j] = nums[i];

                // Increment 'j' to the next position for the next unique
element
                j++;
            }
        }

        // 'j' represents the count of unique elements, and also the
position to truncate the array
        return j;
    }
};
```

<u>Code Explanation and Complexity of Remove Duplicates from Sorted Array</u>

The Intuition is to use two pointers, i and j, to iterate through the array. The variable j is used to keep track of the current index where a unique element should be placed. The initial value of j is 1 since the first element in the array is always unique and doesn't need to be changed.

Explanation:

The code starts iterating from i = 1 because we need to compare each element with its previous element to check for duplicates.

The main logic is inside the for loop:

1. If the current element nums[i] is not equal to the previous element nums[i - 1], it means we have encountered a new unique element.
2. In that case, we update nums[j] with the value of the unique element at nums[i], and then increment j by 1 to mark the next position for a new unique element.
3. By doing this, we effectively overwrite any duplicates in the array and only keep the unique elements.

Once the loop finishes, the value of j represents the length of the resulting array with duplicates removed.

Time Complexity: The function iterates through the array once, performing constant-time operations for each element. Therefore, the time complexity is O(N), where N is the size of the array.

Space Complexity: The function uses only a constant amount of extra space (for variables like j). Therefore, the space complexity is O(1).