TIP102 - Mock Technical Interview Questions

3Sum (Medium)
Longest Substring Without Repeating Characters (Medium)

3Sum

Problem:

Given an integer array nums, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

Example 1:

```
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.

The distinct triplets are [-1,0,1] and [-1,-1,2].
Notice that the order of the output and the order of the triplets does not matter.
```

Example 2:

```
Input: nums = [0,1,1]
Output: []
Explanation: The only possible triplet does not sum up to 0.
```

Example 3:

```
Input: nums = [0,0,0]
Output: [[0,0,0]]
Explanation: The only possible triplet sums up to 0.
```

Clarifying Questions

- What is the expected output?
  - The expected output is a list of all unique triplets that sum to 0
- Can the input array contain duplicate elements?
  - Yes
- Does the solution have to be in a specific order?
  - No

Hints

- Consider **sorting** the input array first to make it easier to find solutions.
- Use a **two-pointer** approach to check for solutions.
- Be sure to **skip duplicates** to avoid counting the same solution multiple times.

General Approach

1. Sort the input array in ascending order.
2. Loop over the array and check for solutions.
3. For each element in the array, set the target sum to 0 minus the current value.
4. Set the left and right pointers and loop until the pointers cross.
5. In each iteration, check if the sum of the values pointed to by the left and right pointers is equal to the target sum.
6. If it is, add the solution to the list and move the pointers. If the sum is too small, move the left pointer. If the sum is too large, move the right pointer.
7. Return the list of solutions.

Space and Time Complexity

The *space complexity* of this approach is **O(1)**, since we only use a constant amount of additional space to store the pointers and the solutions. The *time complexity* is **O(n^2)**, since we loop over the array once and then use a two-pointer approach to check for solutions in each iteration.

SOLUTION

Python

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        # Sort the input array
        nums.sort()

        # Initialize the set of solutions
        solutions = set()

        # Loop over the array and check for solutions
        for i in range(len(nums)):
            # Set the target sum to 0 minus the current value
            target = 0 - nums[i]

            # Set the left and right pointers
            left = i + 1
            right = len(nums) - 1

            # Loop until the pointers cross
            while left < right:
                # Check if the sum of the values pointed to by the left and right
                if nums[left] + nums[right] == target:
                    # Add the solution to the set and move the pointers
                    solutions.add((nums[i], nums[left], nums[right]))
                    left += 1
                    right -= 1

                    # Skip duplicates
                    while left < right and nums[left] == nums[left - 1]:
                        left += 1
                    while left < right and nums[right] == nums[right + 1]:
                        right -= 1
                elif nums[left] + nums[right] < target:
                    # Move the left pointer
                    left += 1
                else:
                    # Move the right pointer
                    right -= 1

        # Return the list of solutions
        return list(solutions)
```

Java

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        // Sort the array in ascending order
        Arrays.sort(nums);

        // Initialize a list to store the solutions
        List<List<Integer>> solutions = new ArrayList<>();

        // Loop over the array and check for solutions
        for (int i = 0; i < nums.length - 2; i++) {
            // Skip duplicates
            if (i > 0 && nums[i] == nums[i - 1]) continue;

            // Set the target sum to 0 minus the current value
            int target = 0 - nums[i];

            // Set the left and right pointers
            int left = i + 1;
            int right = nums.length - 1;

            // Loop until the pointers cross
            while (left < right) {
            // Check if the sum of the left and right values is equal to the targe
            if (nums[left] + nums[right] == target) {
                // Add the solution to the list
                solutions.add(Arrays.asList(nums[i], nums[left], nums[right]));

                // Skip duplicates
                while (left < right && nums[left] == nums[left + 1]) left++;
                while (left < right && nums[right] == nums[right - 1]) right--;

                // Move the pointers
                left++;
                right--;
            } else if (nums[left] + nums[right] < target) {
                // Move the left pointer if the sum is too small
                left++;
            } else {
                // Move the right pointer if the sum is too large
                right--;
            }
            }
        }

        // Return the list of solutions
        return solutions;
    }
```

```
}
```

Follow-up Questions

- How would you extend your solution to handle 4Sum, 5Sum, and kSum problems?
- Can you provide an example input where your solution would not work and explain why?
- Can you think of any alternative approaches to solving the problem?
- Can you think of any real-world scenarios where the problem could be applied?

Leetcode Reference

3Sum (https://leetcode.com/problems/3sum/)

Longest Substring Without Repeating Characters

Problem:

Given a string `s`, find the length of the longest substring without repeating characters.

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a subs
```

Clarifying Questions

- What is the expected output?
  - The expected output is the length of the longest substring in the input string that does not contain any repeating characters
- Is the input string guaranteed to be non-empty?
  - Yes
- Are the characters in the input string case-sensitive?
  - Yes, meaning that “a” and “A” are considered to be different characters

Hints

- Use a sliding window approach to find the longest substring.
- Keep track of the characters that have been seen in the current window using a set or a map. (A set would automatically avoid duplicates, which is a requirement)
- If a character is already in the set or map, shrink the window from the left until the character is no longer in the set or map.

General Approach

1. Initialize the set or map to an empty collection and set the maximum length to 0.
2. Set the left and right pointers to 0 and loop until the right pointer reaches the end of the string.
3. In each iteration, add the character pointed to by the right pointer to the set or map.
4. If the set or map already contains the character, shrink the window from the left until the character is no longer in the set or map.
5. Update the maximum length if the current length is greater than the maximum length.
6. Return the maximum length.

Space and Time Complexity

The *space complexity* of this approach is **O(n)**, since we use a set or map to store the characters in the current window. The *time complexity* is **O(n)**, since we loop over the string once and perform a constant amount of work in each iteration.

SOLUTION

Python

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        # Initialize the longest length and the current length
        longest = 0
        current = 0

        # Initialize the dictionary of characters in the current substring
        seen = {}

        # Initialize the start index of the current substring
        start = 0

        # Loop over the string and update the current length
        for i, ch in enumerate(s):
            # Check if the character is in the dictionary of seen characters
            if ch in seen:
                # Update the longest length if needed
                longest = max(longest, current)

                # Update the start index of the current substring
                start = max(start, seen[ch] + 1)

            # Add the character to the dictionary and increment the current length
            seen[ch] = i
            current = i - start + 1

        # Return the longest length
        return max(longest, current)
```

Java

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        // Initialize the longest length and the current length
        int longest = 0;
        int current = 0;

        // Initialize the map of characters in the current substring
        Map<Character, Integer> seen = new HashMap<>();

        // Initialize the result
        int res = 0;

        // Loop over the string and update the current length
        while (current < s.length()) {
            // Get the current character
            char r = s.charAt(current);

            // Increment the count of the current character in the map
            seen.put(r, seen.getOrDefault(r,0) + 1);

            // While the current character has been seen more than once, remove ch
            while (seen.get(r) > 1) {
                // Get the character at the start of the current substring
                char l = s.charAt(longest);

                // Decrement the count of the character in the map
                seen.put(l, seen.get(l) - 1);

                // Move the start of the current substring to the next character
                longest++;
            }

            // Update the result if needed
            res = Math.max(res, current - longest + 1);

            // Move to the next character
            current++;
        }

        // Return the result
        return res;
    }
}
```

Follow-up Questions

- How would you improve the performance of your solution?

- Can you provide an example input where your solution would not work and explain why?
- Can you think of any alternative approaches to solving the problem?
- Can you think of any real-world scenarios where the problem could be applied?

Leetcode Reference

Longest Substring Without Repeating Characters (https://leetcode.com/problems/longest-substring-without-repeating-characters/)