[Click on the logo to find **Problem Statement**]

Intuition

The code aims to find all unique triplets in a given array `nums` whose sum equals zero. It employs a two-pointer approach after sorting the array. The primary intuition is to iterate through the array, considering each element as the potential first element of a triplet. To avoid duplicate triplets, we skip consecutive identical elements.

For each chosen first element, the code uses two pointers (`l` and `r`) to scan the remaining elements. The pointers move towards each other, adjusting based on the sum of the current triplet compared to zero. If the sum is greater than zero, the right pointer (`r`) is moved to the left; if the sum is less than zero, the left pointer (`l`) is moved to the right. If the sum is zero, a valid triplet is found, and it is added to the result (`res`). Additional steps ensure uniqueness by skipping over duplicate elements during the process.

The sorting step is crucial for this algorithm, enabling efficient traversal and allowing us to handle duplicates gracefully. The overall strategy optimally identifies triplets with a time complexity of O(n^2) and utilizes only a constant amount of extra space, resulting in a space complexity of O(1).

Approach

1. **Sort the Array:**
    - Sort the given array `nums` in ascending order. Sorting is crucial for the two-pointer approach and helps in efficiently handling duplicates.
2. **Iterate Through the Array:**
    - Use a for loop to iterate through each element of the array, considering it as a potential first element (`a`) of a triplet.
    - Skip duplicate elements by checking if the current element is the same as the previous one (`if a == nums[i - 1]`), and continue to the next iteration if true.
3. **Initialize Two Pointers:**
    - Initialize two pointers (`l` and `r`) to point to the elements immediately following the current chosen element. `l` starts at the element to the right of `a` , and `r` starts at the last

element in the array.

4. **Two-Pointer Traversal:**
    - Use a while loop to traverse the array with the two pointers.
    - Calculate the sum of the current triplet: `threeSum = a + nums[l] + nums[r]`.
    - Adjust the pointers based on the comparison of `threeSum` with zero:
        - If `threeSum > 0`, decrement `r` to reduce the sum.
        - If `threeSum < 0`, increment `l` to increase the sum.
        - If `threeSum == 0`, a valid triplet is found, add `[a, nums[l], nums[r]]` to the result (`res`), and move `l` to the right to avoid duplicates.
5. **Handle Duplicates:**
    - Inside the while loop, skip over consecutive identical elements to ensure unique triplets.
        - Use `while l < r and nums[l] == nums[l - 1]` to skip over duplicates of the second element.
6. **Return the Result:**
    - Once the entire array is traversed, return the list of unique triplets found (`res`).

Complexity

- Time complexity: O(n^2)
- Space complexity: O(1)

Code

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        res = []
        nums.sort()

        for i, a in enumerate(nums):
            if i > 0 and a == nums[i - 1]:
                continue

            l, r = i + 1, len(nums) - 1

            while l < r:
                threeSum = a + nums[l] + nums[r]

                if threeSum > 0:
```

```
            r -= 1
        elif threeSum < 0:
            l += 1
        else:
            res.append([a, nums[l], nums[r]])
            l += 1

            while l < r and nums[l] == nums[l - 1]:
                l += 1

return res
```

If you want to see more solutions to coding problems, you can visit:

GitHub