3Sum

Created	@2025年4月13日 下午3:07
Question Type	Two Pointers
Difficulty	Medium
LeetCode Question Link	https://leetcode.com/problems/3sum/description/

1. Question Understanding

1.1 Description:

Based on my understanding, I need to find all combinations of three numbers in the array whose sum is 0.

1.2 Input:

The input type will be a list of integers.

1.3 Input Assumption:

N/A

1.4 Output:

The output should be a list of lists, where each inner list contains three numbers that sum to 0. We don’t need to return their indices, only the numbers themselves.

1.5 Example:

Regular case:

Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]

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

Edge Case 2:
Input: nums = [0,0,0]
Output: [[0,0,0]]
Explanation: The only possible triplet sums up to 0.

1.6 Other Q&A:

- **Question 1:** Is it possible that there is no solution?
  - Yes. In that case, please return an empty list (as shown in Edge Case 1).
- **Question 2:** Are `[1, 0, -1]` and `[-1, 0, 1]` considered the same result?
  - It's not entirely clear, but I don't think we need to treat them as separate results. We can test and decide.
  - Update: Yes!

2. Attempt 1

2.1 Thought

- To find three numbers whose sum is 0, it's the same as finding two other numbers whose sum is the negation of the fixed number.
- Since indices don't matter and we only care about the sum of values, we can first sort the array in ascending order.

- By sorting the array and then iterating from left to right:
  - **Why we don't need to check numbers on the left of the fixed number?** Once we fix a number, all possible combinations involving numbers to its left would have already been evaluated when those numbers were the "fixed" ones. Hence, we only need to look to the right side for new combinations.
  - **How we avoid duplicates:** We skip any repeated numbers. For example, if `nums[i] == nums[i-1]`, then we skip `nums[i]` to avoid generating the same triplet again. Basically, the fixed number on each iteration is the pivot of its own group of the solution.
- Steps:
  1. Sort the array in ascending order.
  2. Loop through the array with index `i`.
     - Fix `nums[i]`, then use a two-pointer approach on the sub-array to the right (`i+1` to end).
     - While moving the left or right pointers, skip over duplicates to avoid repeating triplets.
  3. Collect all distinct triplets that sum to 0.

2.2 Pseudo-Code: (Ignore this part. This is a draft of code for thinking purposes.)

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        # sorting nums
        # for num in nums:
            # if num == previous num:
                # continue
             # otherwise:
```

```
            # there could be some optimization here to early stop.
            # perform two sum algorithm
```

2.3 Implementation through python:

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

        # result array
        res = []
        # for loop through the array
        for i in range(len(nums) - 2):
            if i > 0 and nums[i] == nums[i - 1]:
                continue
            else:

                left = i + 1
                right = len(nums) - 1
                while left < right:
                    # calculate the sum of the three numbers
                    total = nums[i] + nums[left] + nums[right]
                    # if the sum is 0, add it to the result array
                    if total == 0:
                        res.append([nums[i], nums[left], nums[right]])
                        left += 1
                        right -= 1
                        # skip duplicates for left pointer
                        while left < right and nums[left] == nums[left - 1]:
                            left += 1
                        # skip duplicates for right pointer
                        while left < right and nums[right] == nums[right + 1]:
```

```
                right -= 1
            elif total < 0:
                left += 1
            else:
                right -= 1
        return res
```

2.4 Time Complexity and Space Complexity

2.4.1 Time Complexity

For each number in the sorted list, we perform a two-pointer (two-sum) approach, which takes $O(n)$ in the worst case. Since we do this for every element, the overall time complexity is $O(n^2)$.

2.4.2 Space Complexity

In the worst case, we could store all possible triplet combinations. The number of ways to choose 3 elements out of \(n\) is \(\binom{n}{3}\), so the space complexity can be a maximum of:

- $O\left(\binom{n}{3}\right)$ for storing triplet results.
- O(n) for the sorting algorithm, or O(1) if the sorting is done in-place.

3. Attempt 2

3.1 Thought

- A small change can significantly improve the algorithm. Since the array is sorted, if the current fixed number is positive, the remaining numbers must also be positive. At that point, we can break out of the loop because the remaining numbers are greater than the fixed number and cannot sum to 0.

3.2 Pseudo-Code: (Ignore this part. This is a draft of code for thinking purposes.)

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        # sorting nums
        # for num in nums:
            # !!! if num > 0:
                  # break
            # if num == previous num:
                # continue
              # otherwise:
                # there could be some optimization here to early stop.
                # perform two sum algorithm
```

3.3 Implementation through python:

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

        # result array
        res = []
        # for loop through the array
        for i in range(len(nums) - 2):
            if nums[i] > 0:
                break
            if i > 0 and nums[i] == nums[i - 1]:
                continue
            else:

                left = i + 1
                right = len(nums) - 1
                while left < right:
                    # calculate the sum of the three numbers
```

```
            total = nums[i] + nums[left] + nums[right]
            # if the sum is 0, add it to the result array
            if total == 0:
                res.append([nums[i], nums[left], nums[right]])
                left += 1
                right -= 1
                # skip duplicates for left pointer
                while left < right and nums[left] == nums[left - 1]:
                    left += 1
                # skip duplicates for right pointer
                while left < right and nums[right] == nums[right + 1]:
                    right -= 1
            elif total < 0:
                left += 1
            else:
                right -= 1
    return res
```

3.4 Time Complexity and Space Complexity

3.4.1 Time Complexity

For each number in the sorted list, we perform a two-pointer (two-sum) approach, which takes $O(n)$ in the worst case. Since we do this for every element, the overall time complexity is $O(n^2)$.

3.4.2 Space Complexity

In the worst case, we could store all possible triplet combinations. The number of ways to choose 3 elements out of \(n\) is \(\binom{n}{3}\), so the space complexity can be a maximum of:

- $O\left(\binom{n}{3}\right)$ for storing triplet results.
- O(n) for the sorting algorithm, or O(1) if the sorting is done in-place.