19 三数之和

更新时间：2019-08-29 09:39:58

> 生活永远不像我们想像的那样好，但也不会像我们想像的那样糟。
>
> ——莫泊桑

刷题内容

> 难度: Medium
>
> 原题链接：https://leetcode-cn.com/problems/3sum/

内容描述

给定一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

例如, 给定数组 nums = [-1, 0, 1, 2, -1, -4],

满足要求的三元组集合为：

```
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

题目详解

首先看题目描述，我们知道这道题是要在数组 `nums` 中找到三个元素 a，b，c。并且a，b，c三个元素相加等于0，可以有负整数元素，并将符合条件的三个元素放进一个数组中。每一种符合条件的方案组成数组放进另一个数组中，最后返回的结果是一个二维数组，在这道题中我们要注意两点：

- `nums` 中会出现重复的数字；
- 每一种组成方案只能出现一次。

了解这两点限制条件之后，我们来看下解题方案：

解题方案

思路 1：时间复杂度: O(N^3) 空间复杂度: O(N)

这道题和我们做过的 `两数相加` 这道题很相似，两数相加的时候可以用双重循环来暴力地找出符合条件的组合，这道题只是把两数相加换成了三数相加，所以可以使用三重暴力循环来解这道题。

每一重循环都从 `nums` 中取出一个元素，如果三个元素相加等于 0，则将这三个元素放入数组 `curRes` 中。最后判断 `curRes` 是否在最后结果 `res` 中。如果不再说明此组合是第一次出现，并将 `curRes` 添加到 `res` 中去，如果在 `res` 中则不执行任何操作。

因为题目限制每一种组成方案只能出现一次，所以在做循环之前首先要对 `nums` 进行排序，因为数组 `nums` 中可以有重复元素的出现，如果不排序的话可能会造成组合方案的重复。而在进行排序之后可以限制这种情况的发生。下面我们来看代码：

Python

```python
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        res = []
        nums.sort() # 将数组进行排序
        for i in range(n): # 第一重循环
            for j in range(i+1, n): # 第二重循环
                for k in range(j+1, n): # 第三重循环
                    if nums[i] + nums[j] + nums[k] == 0: # 如果三个数字加起来是0的话
                        curRes = [nums[i], nums[j], nums[k]] # 将这三个数放进数组
                        # 如果res当中不存在当前这种组合的话我们就把它放进去
                        if curRes not in res:
                            res.append(curRes)

        return res  # 最后将结果返回
```

Java

```java
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        int n = nums.length;
        // 将数组排序
        Arrays.sort(nums);
        List<List<Integer>> ret = new ArrayList<>();
        // 通过三层循环进行暴力搜索
        for (int i = 0; i < n; i++) {
            // 如果出现相同的值则跳过
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            for (int j = i + 1; j < n; j++) {
                // 如果出现相同的值则跳过
                if (j > i + 1 && nums[j] == nums[j - 1]) {
                    continue;
                }
                for (int k = j + 1; k < n; k++) {
                    // 如果出现相同的值则跳过
                    if (k > j + 1 && nums[k] == nums[k - 1]) {
                        continue;
                    }
                    // 如果三个数相加为 0，则是目标值集合
                    if (nums[i] + nums[j] + nums[k] == 0) {
                        List<Integer> temp = new ArrayList<>();
                        temp.add(nums[i]);
                        temp.add(nums[j]);
                        temp.add(nums[k]);
                        ret.add(temp);
                    }
                }
            }
        }
        return ret;
    }
}
```

Go

```go
func threeSum(nums []int) [][]int {
    n := len(nums)
    res := make([][]int, 0)
    sort.Ints(nums)
    for i := 0; i < n; i += 1 {
        for j := i+1; j < n; j += 1 {
            for k := j+1; k < n; k += 1 {
                if nums[i] + nums[j] + nums[k] == 0 { // 如果三个数字加起来是0的话
                    curRes := []int{nums[i], nums[j], nums[k]}
                    var flag bool
                    for _, r := range res {
                        if r[0] == curRes[0] && r[1] == curRes[1] && r[2] == curRes[2] {
                            flag = true
                            break
                        }
                    }
                    if !flag { // 如果res当中不存在当前这种组合的话我们就把它放进去
                        res = append(res, curRes)
                    }
                }
            }
        }
    }
    return res
}
```

c++

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();
        // 将数组排序
        sort(nums.begin(), nums.end());
        vector<vector<int>> ret;
        // 通过三层循环进行暴力搜索
        for (int i = 0;i < n;i++) {
            // 如果出现相同的值则跳过
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            for (int j = i + 1;j < n;j++) {
                // 如果出现相同的值则跳过
                if (j > i + 1 && nums[j] == nums[j - 1]) {
                    continue;
                }
                for (int k = j + 1;k < n;k++) {
                    // 如果出现相同的值则跳过
                    if (k > j + 1 && nums[k] == nums[k - 1]) {
                        continue;
                    }
                    // 如果三个数相加为 0，则是目标值集合
                    if (nums[i] + nums[j] + nums[k] == 0) {
                        vector<int> temp;
                        temp.push_back(nums[i]);
                        temp.push_back(nums[j]);
                        temp.push_back(nums[k]);
                        ret.push_back(temp);
                    }
                }
            }
        }
        return ret;
    }
};
```

上面的代码就是这道题最暴力也是最简单的解法，直接三重循环取出每一种组合来尝试结果是否等于0。但是这样的解法如果你去 LeetCode 上进行提交的话是不会通过的，因为时间复杂度太高，也就是超时了。你可能会问：“我在测试程序的时候执行速度很快啊？”。但如果 `nums` 的长度是100,1000或者10,000呢？这种方法的执行速度还会很快吗？为了解决这个问题我们采用了一种更加高效的方式，下面来看思路2。

思路 2：时间复杂度: O(N^2) 空间复杂度: O(N)

为了解决思路1中时间复杂度太高的问题，我们在思路2中采用了一种全新的方法，暂时称它为左右边界方法吧。先来看一下大致的步骤：

1. 获取 `nums` 的长度 n，将 `nums` 进行排序，排序的原因与思路1一致，并遍历 `nums`；
2. 获取左右边界索引，左边界 l 是 `nums[i]` 的下一个元素即索引是 `i+1` ，右边界 r 为最后一个元素即 `n-1`；
3. `nums[i]` 固定， `while` 循环使左右边界向中间移动；
4. `temp` = `nums[i]` + `nums[l]` + `nums[r]` 判断 temp 是否等于0；
5. 如果 `temp = 0` ，则将 `nums[i]`， `nums[l]`， `nums[r]` 三个元素组成数组放入 `res` 中，如果 temp 大于 0，我们需要将右边界 r 向左移动取一个小一些的元素，如果 temp 小于0，则将 l 向右移动取一个大一些的元素重新组合尝试。

上面这种方法大大缓解了思路1超大的时间复杂度，下面我们来看下具体的代码：

Python beats 40.29%

```python
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        n, res = len(nums), []
        nums.sort()
        for i in range(n):
            # 判断一下如果当前元素和上一个元素相等则跳出本次循环
            if i > 0 and nums[i] == nums[i-1]:   # i=0的时候我们需要直接往下执行
                continue
            l, r = i+1, n-1
            while l < r:
                tmp = nums[i] + nums[l] + nums[r]
                if tmp == 0: # 如果三个数字加起来是0的话
                    res.append([nums[i], nums[l], nums[r]])
                    l += 1
                    r -= 1
                    while l < r and nums[l] == nums[l-1]: # 重复数字我们不需要考虑
                        l += 1
                    while l < r and nums[r] == nums[r+1]: # 重复数字我们不需要考虑
                        r -= 1
                elif tmp > 0: # 说明我们需要一个更小的数字
                    r -= 1
                else: # 说明我们需要一个更大的数字
                    l += 1
        return res
```

Java beats 77.14%

```java
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        int n = nums.length;
        // 将数组进行排序
        Arrays.sort(nums);
        List<List<Integer>> ret = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            // 如果出现相同的值则跳过
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            // l 和 r 分别代表左右两个指针，实际上是双索引法的延申
            int l = i + 1;
            int r = n - 1;
            // 如果 l < r 说明还可以进行搜索满足条件的三个值
            while (l < r) {
                int temp = nums[i] + nums[l] + nums[r];
                if (temp == 0) {
                    // 如果 temp = 0 说明满足条件，可以添加到集合中
                    List<Integer> tempList = new ArrayList<>();
                    tempList.add(nums[i]);
                    tempList.add(nums[l]);
                    tempList.add(nums[r]);
                    ret.add(tempList);
                    // 满足条件就左右两个指针向中间靠拢
                    l++;
                    r--;
                    // l 右移的同时要排除重复的值
                    while (l < r && nums[l] == nums[l - 1]) {
                        l++;
                    }
                    // r 左移的同时要排除重复的值
                    while (l < r && nums[r] == nums[r + 1]) {
                        r--;
                    }
                } else if (temp > 0){
                    // temp > 0 说明需要更小的值，则右指针左移
                    r--;
                } else {
                    // temp < 0 说明需要更大的值，则左指针右移1
                    l++;
                }
            }
        }
        return ret;
    }
}
```

Go beats 91.91%

```go
func threeSum(nums []int) [][]int {
    n, res := len(nums), make([][]int, 0)
    sort.Ints(nums)
    for i := 0; i < n; i += 1 {
        if i > 0 && nums[i] == nums[i-1] { // i=0的时候我们需要直接往下执行
            continue
        }
        l, r := i+1, n-1
        for l < r {
            tmp := nums[i] + nums[l] + nums[r]
            if tmp == 0 { // 如果三个数字加起来是0的话
                res = append(res, []int{nums[i], nums[l], nums[r]})
                l += 1
                r -= 1
                for l < r && nums[l] == nums[l-1] { // 重复数字我们不需要考虑
                    l += 1
                }
                for l < r && nums[r] == nums[r+1] { // 重复数字我们不需要考虑
                    r -= 1
                }
            } else if tmp > 0 { // 说明我们需要一个更小的数字
                r -= 1
            } else { // 说明我们需要一个更大的数字
                l += 1
            }
        }
    }
    return res
}
```

c++ beats 51.93%

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();
        // 将数组进行排序
        sort(nums.begin(), nums.end());
        vector<vector<int>> ret;
        vector<int> temp(3, 0);
        for (int i = 0;i < n;i++) {
            // 如果出现相同的值则跳过
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            if (nums[i] * 3 > 0) {
                break;
            }
            // l 和 r 分别代表左右两个指针，实际上是双索引法的延申
            int left = i + 1;
            int right = n - 1;
            // 如果 l < r 说明还可以进行搜索满足条件的三个值
            while (left < right) {
                // 如果等于0说明满足条件，可以添加到集合中
                if (nums[i] + nums[left] + nums[right] == 0) {
                    temp[0] = nums[i];
                    temp[1] = nums[left];
                    temp[2] = nums[right];
                    ret.push_back(temp);
                }
                // 小于0 说明需要更大的值，则左指针右移1
                if (nums[i] + nums[left] + nums[right] < 0) {
                    left++;
                    while (left < right && nums[left] == nums[left - 1]) {
                        left++;
                    }
                } else { // 大于0 说明需要更小的值，则右指针左移
                    right--;
                    while (left < right && nums[right] == nums[right + 1]) {
                        right--;
                    }
                }
            }
        }
        return ret;
    }
};
```

上面的代码是经过我们优化后的，现在的时间复杂度是O(N^2)，你可以去 LeetCode 上提交一下试试，会获得一个不错的通过率。

小结

这个小节的题目和我们之前做过的第一题很像，所以最暴力的解法和第一题一样，都是多重循环嵌套。但是很明显超时了，为了解决这个问题我们采用了左右边界的方法来解决这个问题，在左右边界方法中要注意元素重复的问题。可能比较难理解，可以把代码多敲几遍，这样你会越来越熟练。

}