Previous Next

15. 3Sum

April 23, 2020 | 58.6K views

Given an array `nums` of n integers, are there elements a, b, c in `nums` such that $a + b + c = 0$? Find all unique triplets in the array which gives the sum of zero.

Note:

The solution set must not contain duplicate triplets.

Example:

```
Given array nums = [-1, 0, 1, 2, -1, -4],

A solution set is:
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

Solution

This problem is a follow-up of Two Sum, and it is a good idea to first take a look at Two Sum and Two Sum II - Input Array is Sorted. An interviewer may ask to solve Two Sum first, and then throw 3Sum at you. Pay attention to subtle differences in problem description and try to re-use existing solutions!

Two Sum, Two Sum II and 3Sum share a similarity that the sum of elements must match the target exactly. A difference is that, instead of giving just one combination of elements, we need to find all unique combinations that sum to the target.

Before jumping in, let's check existing solutions and determine the best conceivable runtime (BCR) for 3Sum:

1. Two Sum uses a hashmap to find complement values, and therefore achieves $\mathcal{O}(N)$ time complexity.
2. Two Sum II uses the two pointers pattern and also has $\mathcal{O}(N)$ time complexity for a sorted array. We can use this approach for any array if we sort it first, which bumps the time complexity to $\mathcal{O}(n \log n)$.

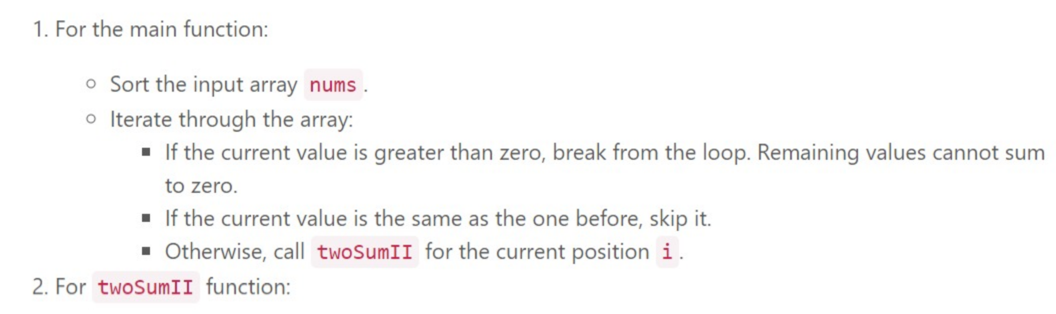
Considering that there is one more dimension in 3Sum, it sounds reasonable to shoot for $\mathcal{O}(n^2)$ time complexity as our BCR.

Approach 1: Two Pointers

It's easier to deal with duplicates if the array is sorted. As our BCR is $\mathcal{O}(n^2)$, sorting the array would not change the overall time complexity.

After sorting the array, we process each value from left to right. For value `v`, we need to find all pairs whose sum is equal to `-v`. To do that, we use the Two Sum II: Two Pointers approach for the rest of the array.

To make sure the result contains unique triplets, we need to skip duplicate values. It is easy to do because repeating values are next to each other in a sorted array.



Algorithm

The implementation is straightforward - we just need to modify `twoSumII` to produce triplets and skip repeating values.

1. For the main function:
   - Sort the input array `nums`.
   - Iterate through the array:
     - If the current value is greater than zero, break from the loop. Remaining values cannot sum to zero.
     - If the current value is the same as the one before, skip it.
     - Otherwise, call `twoSumII` for the current position `i`.
2. For `twoSumII` function:
   - Set the low pointer `lo` to `i + 1`, and high pointer `hi` to the last index.
   - While low pointer is smaller than high:
     - If the sum of `nums[i]`, `nums[lo]` and `nums[hi]` is less than zero, increment `lo`.
       - Also increment `lo` if the value is the same as for `lo - 1`.
     - If the sum is greater than zero, decrement `hi`.
       - Also decrement `hi` if the value is the same as for `hi + 1`.
     - Otherwise, we found a triplet:
       - Add it to the result `res`.
       - Decrement `hi` and increment `lo`.
3. Return the result `res`.

C++ Java Python3 Copy

```python
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        res = []
        nums.sort()
        for i in range(len(nums)):
            if nums[i] > 0:
                break
            if i == 0 or nums[i - 1] != nums[i]:
                self.twoSumII(nums, i, res)
        return res

    def twoSumII(self, nums: List[int], i: int, res: List[List[int]]):
        lo, hi = i + 1, len(nums) - 1
        while (lo < hi):
            sum = nums[i] + nums[lo] + nums[hi]
            if sum < 0 or (lo > i + 1 and nums[lo] == nums[lo - 1]):
                lo += 1
            elif sum > 0 or (hi < len(nums) - 1 and nums[hi] == nums[hi + 1]):
                hi -= 1
            else:
                res.append([nums[i], nums[lo], nums[hi]])
                lo += 1
                hi -= 1
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(n^2)$. `twoSumII` is $\mathcal{O}(n)$, and we call it n times.

  Sorting the array takes $\mathcal{O}(n \log n)$, so overall complexity is $\mathcal{O}(n \log n + n^2)$. This is asymptotically equivalent to $\mathcal{O}(n^2)$.

- Space Complexity: from $\mathcal{O}(\log n)$ to $\mathcal{O}(n)$, depending on the implementation of the sorting algorithm. For the purpose of complexity analysis, we ignore the memory required for the output.

Approach 2: Hash Set

Since triplets must sum up to the target value, we can try the hash table approach from the Two Sum solution. This approach won't work, however, if the sum needs to be smaller than a target, like in 3Sum Smaller.

Handling duplicates here is trickier compared to the two pointers approach. We can put a combination of three values into a hash set to efficiently check whether we've found that triplet already. Values in a combination should be ordered (e.g. ascending). Otherwise, we can have results with the same values in the different positions.

Fortunately, we do not need to store all three values - storing the smallest and the largest ones is sufficient to identify any triplet. Because three values sum to the target, the third value will always be the same.

Algorithm

We process each value from left to right. For value `v`, we need to find all pairs whose sum is equal `-v`. To find such pairs, we apply the Two Sum: One-pass Hash Table approach to the rest of the array. To ensure unique triplets, we use a hash set `found` as described above.

Because hashmap operations could be expensive, the solution below may be too slow. We'll add some optimizations in the next section.

C++ Java Python3 Copy

```python
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        res = []
        found = set()
        for i, val1 in enumerate(nums):
            seen = set()
            for j, val2 in enumerate(nums[i+1:]):
                complement = -val1 - val2
                if complement in seen:
                    min_val = min((val1, val2, complement))
                    max_val = max((val1, val2, complement))
                    if (min_val, max_val) not in found:
                        found.add((min_val, max_val))
                        res.append([val1, val2, complement])
                seen.add(val2)
        return res
```

Optimized Algorithm

These optimizations don't change the big-O complexity, but help speed things up: 1. Use another hash set `dups` to skip duplicates in the outer loop. 2. Instead of re-populating a hash set every time in the inner loop, we can populate a hashmap once and then only modify values. After we process `nums[j]` in the inner loop, we set the hashmap value to `i`. This indicates that we can now use `nums[j]` to find pairs for `nums[i]`.

C++ Java Python3 Copy

```python
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        res = []
        found, dups = set(), set()
        seen = {}
        for i, val1 in enumerate(nums):
            if val1 not in dups:
                dups.add(val1)
                for j, val2 in enumerate(nums[i+1:]):
                    complement = -val1 - val2
                    if complement in seen and seen[complement] == i:
                        min_val = min((val1, val2, complement))
                        max_val = max((val1, val2, complement))
                        if (min_val, max_val) not in found:
                            found.add((min_val, max_val))
                            res.append([val1, val2, complement])
                    seen[val2] = i
        return res
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(n^2)$. We have outer and inner loops, each going through n elements.

  While the asymptotic complexity is the same, this algorithm is noticeably slower than the previous approach. Lookups in a hash set, though requiring a constant time, are expensive compared to the direct memory access.

- Space Complexity: $\mathcal{O}(n^2)$. We may need to store up to n^2 elements in a hash set for deduplication.

  We need the same amount of memory here as to store the output. In the worst case, there could be $\mathcal{O}(n^2)$ triplets in the output, like for this example: `[-k, -k + 1, ..., -1, 0, 1, ... k - 1, k]`. Adding a new number to this sequence will produce `n / 3` new triplets.

Further Thoughts

This is a well-known problem with many variations and its own Wikipedia page.

For an interview, we recommend focusing on the Two Pointers approach above. It's easier to get it right and adapt for other variations of 3Sum. Interviewers love asking follow-up problems like 3Sum Smaller and 3Sum Closest!