Previous Next

16. 3Sum Closest

May 13, 2020 | 15.8K views

Average Rating: 4.30 (23 votes)

Given an array `nums` of n integers and an integer `target`, find three integers in `nums` such that the sum is closest to `target`. Return the sum of the three integers. You may assume that each input would have exactly one solution.

Example 1:

```
Input: nums = [-1,2,1,-4], target = 1
Output: 2
Explanation: The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
```

Constraints:

- `3 <= nums.length <= 10^3`
- `-10^3 <= nums[i] <= 10^3`
- `-10^4 <= target <= 10^4`

Solution

This problem is a variation of 3Sum. The main difference is that the sum of a triplet is not necessarily equal to the target. Instead, the sum is in some *relation* with the target, which is *closest* to the target for this problem. In that sense, this problem shares similarities with 3Sum Smaller.

Before jumping in, let's check solutions for the similar problems:

1. 3Sum fixes one number and uses either the two pointers pattern or a hash set to find complementary pairs. Thus, the time complexity is $\mathcal{O}(n^2)$.

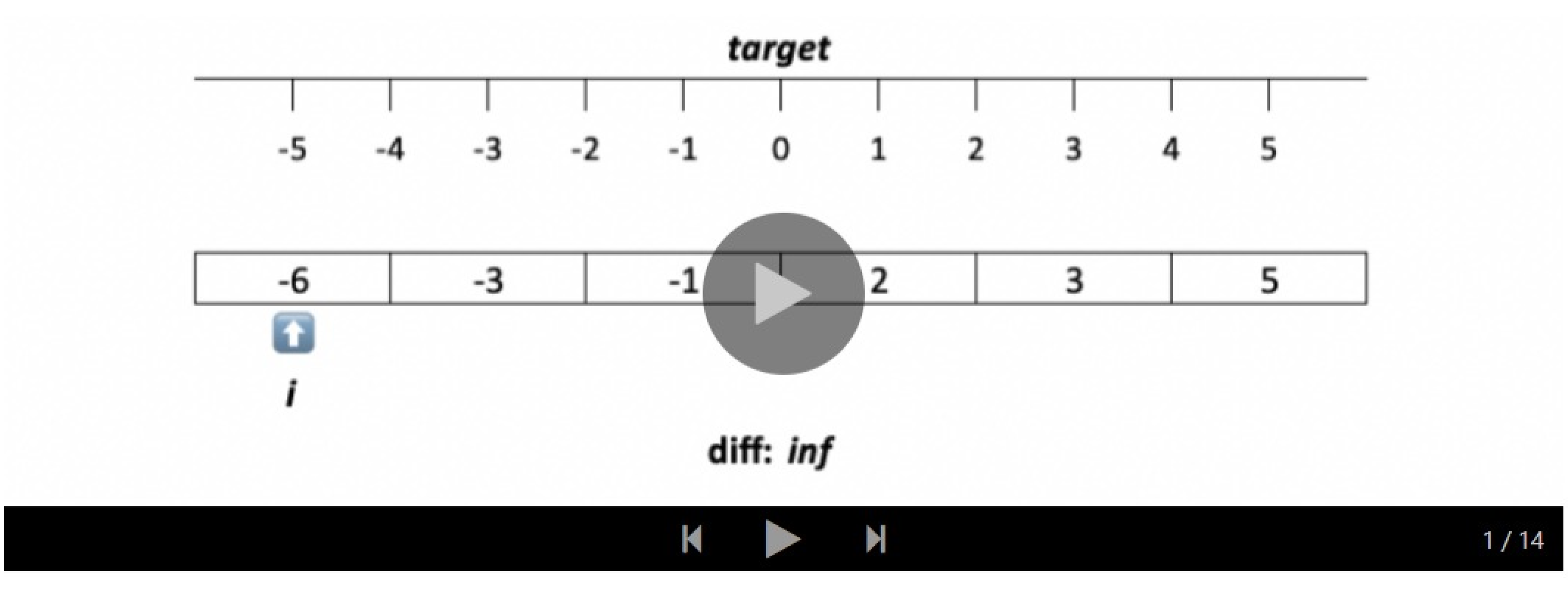
2. 3Sum Smaller, similarly to 3Sum, uses the two pointers pattern to enumerate smaller pairs. Note that we cannot use a hash set here because we do not have a specific value to look up.

For the same reason as for 3Sum Smaller, we cannot use a hash set approach here. So, we will focus on the two pointers pattern and shoot for $\mathcal{O}(n^2)$ time complexity as the best conceivable runtime (BCR).

Approach 1: Two Pointers

The two pointers pattern requires the array to be sorted, so we do that first. As our BCR is $\mathcal{O}(n^2)$, the sort operation would not change the overall time complexity.

In the sorted array, we process each value from left to right. For value `v`, we need to find a pair which sum, ideally, is equal to `target - v`. We will follow the same two pointers approach as for 3Sum, however, since this 'ideal' pair may not exist, we will track the smallest absolute difference between the sum and the target. The two pointers approach naturally enumerates pairs so that the sum moves toward the target.



Algorithm

1. Initialize the minimum difference `diff` with a large value.
2. Sort the input array `nums`.
3. Iterate through the array:
   - For the current position `i`, set `lo` to `i + 1`, and `hi` to the last index.
   - While the `lo` pointer is smaller than `hi`:
     - Set `sum` to `nums[i] + nums[lo] + nums[hi]`.
     - If the absolute difference between `sum` and `target` is smaller than the absolute value of `diff`:
       - Set `diff` to `target - sum`.
     - If `sum` is less than `target`, increment `lo`.
     - Else, decrement `hi`.
   - If `diff` is zero, break from the loop.
4. Return the value of the closest triplet, which is `target - diff`.

C++ | Java | Python3 — Copy

```python
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        diff = float('inf')
        nums.sort()
        for i in range(len(nums)):
            lo, hi = i + 1, len(nums) - 1
            while (lo < hi):
                sum = nums[i] + nums[lo] + nums[hi]
                if abs(target - sum) < abs(diff):
                    diff = target - sum
                if sum < target:
                    lo += 1
                else:
                    hi -= 1
            if diff == 0:
                break
        return target - diff
```

Complexity Analysis

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

  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.

Approach 2: Binary Search

We can adapt the 3Sum Smaller: Binary Search approach to this problem.

In the two pointers approach, we fix one number and use two pointers to enumerate pairs. Here, we fix two numbers, and use a binary search to find the third complement number. This is less efficient than the two pointers approach, however, it could be more intuitive to come up with.

Note that we may not find the exact complement number, so we check the difference between the complement and two numbers: the next higher and the previous lower. For example, if the complement is `42`, and our array is `[-10, -4, 15, 30, 60]`, the next higher is `60` (so the difference is `-18`), and the previous lower is `30` (and the difference is `12`).

Algorithm

1. Initialize the minimum difference `diff` with a large value.
2. Sort the input array `nums`.
3. Iterate through the array (outer loop):
   - For the current position `i`, iterate through the array starting from `j = i + 1` (inner loop):
     - Binary-search for `complement` (`target - nums[i] - nums[j]`) in the rest of the array.
     - For the next higher value, check its absolute difference with `complement` against `diff`.
     - For the previous lower value, check its absolute difference with `complement` against `diff`.
     - Update `diff` based on the smallest absolute difference.
   - If `diff` is zero, break from the loop.
4. Return the value of the closest triplet, which is `target - diff`.

C++ | Java | Python3 — Copy

```python
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        diff = float('inf')
        nums.sort()
        for i in range(len(nums)):
            for j in range(i + 1, len(nums)):
                complement = target - nums[i] - nums[j]
                hi = bisect_right(nums, complement, j + 1)
                lo = hi - 1
                if hi < len(nums) and abs(complement - nums[hi]) < abs(diff):
                    diff = complement - nums[hi]
                if lo > j and abs(complement - nums[lo]) < abs(diff):
                    diff = complement - nums[lo]
            if diff == 0:
                break
        return target - diff
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(n^2\log{n})$. Binary search takes $\mathcal{O}(\log{n})$, and we do it n times in the inner loop. Since we are going through n elements in the outer loop, the overall complexity is $\mathcal{O}(n^2\log{n})$.

- Space Complexity: from $\mathcal{O}(\log{n})$ to $\mathcal{O}(n)$, depending on the implementation of the sorting algorithm.

Further Thoughts

3Sum 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!

If an interviewer asks you whether you can achieve $\mathcal{O}(1)$ memory complexity, you can use the selection sort instead of a built-in sort in the Two Pointers approach. It will make it a bit slower, though the overall time complexity will be still $\mathcal{O}(n^2)$.