

259. 3Sum Smaller

March 5, 2016 | 23.8K views

Average Rating: 4.13 (32 votes)

Given an array of n integers $nums$ and a $target$, find the number of index triplets i, j, k with $0 \leq i < j < k < n$ that satisfy the condition $nums[i] + nums[j] + nums[k] < target$.

Example:

Input: $nums = [-2, 0, 1, 3]$, and $target = 2$
Output: 2
Explanation: Because there are two triplets which sums are less than 2:
[-2, 0, 1]
[-2, 0, 3]

Follow up: Could you solve it in $O(n^2)$ runtime?

Solution

Approach #1 (Brute Force) [Time Limit Exceeded]

The brute force approach is to find every possible triplets (i, j, k) subjected to $i < j < k$ and test for the condition.

Complexity analysis

- Time complexity: $O(n^3)$. The total number of such triplets is $\binom{n}{3}$, which is $\frac{n!}{(n-3)! \times 3!} = \frac{n \times (n-1) \times (n-2)}{6}$. Therefore, the time complexity of the brute force approach is $O(n^3)$.
- Space complexity: $O(1)$.

Approach #2 (Binary Search) [Accepted]

Before we solve this problem, it is helpful to first solve this simpler *twoSum* version.

Given a $nums$ array, find the number of index pairs i, j with $0 \leq i < j < n$ that satisfy the condition $nums[i] + nums[j] < target$

If we sort the array first, then we could apply binary search to find the largest index j such that $nums[i] + nums[j] < target$ for each i . Once we found that largest index j , we know there must be $j - i$ pairs that satisfy the above condition with i 's value fixed.

Finally, we can now apply the *twoSum* solution to *threeSum* directly by wrapping an outer for-loop around it.

```
public int threeSumSmaller(int[] nums, int target) {
    Arrays.sort(nums);
    int sum = 0;
    for (int i = 0; i < nums.length - 2; i++) {
        sum += twoSumSmaller(nums, i + 1, target - nums[i]);
    }
    return sum;
}

private int twoSumSmaller(int[] nums, int startIndex, int target) {
    int sum = 0;
    for (int i = startIndex; i < nums.length - 1; i++) {
        int j = binarySearch(nums, i, target - nums[i]);
        sum += j - i;
    }
    return sum;
}

private int binarySearch(int[] nums, int startIndex, int target) {
    int left = startIndex;
    int right = nums.length - 1;
    while (left < right) {
        int mid = (left + right + 1) / 2;
        if (nums[mid] < target) {
            left = mid;
        } else {
            right = mid - 1;
        }
    }
    return left;
}
```

Note that in the above binary search we choose the upper middle element $(\frac{left+right+1}{2})$ instead of the lower middle element $(\frac{left+right}{2})$. The reason is due to the terminating condition when there are two elements left. If we chose the lower middle element and the condition $nums[mid] < target$ evaluates to true, then the loop will never terminate. Choosing the upper middle element will guarantee termination.

Complexity analysis

- Time complexity: $O(n^2 \log n)$. The *binarySearch* function takes $O(\log n)$ time, therefore the *twoSumSmaller* takes $O(n \log n)$ time. The *threeSumSmaller* wraps with another for-loop, and therefore is $O(n^2 \log n)$ time.
- Space complexity: $O(1)$.

Approach #3 (Two Pointers) [Accepted]

Let us try sorting the array first. For example, $nums = [3, 5, 2, 8, 1]$ becomes $[1, 2, 3, 5, 8]$.

Let us look at an example $nums = [1, 2, 3, 5, 8]$, and $target = 7$.

[1, 2, 3, 5, 8]
↑ ↑
left right

Let us initialize two indices, *left* and *right* pointing to the first and last element respectively.

When we look at the sum of first and last element, it is $1 + 8 = 9$, which is $\geq target$. That tells us no index pair will ever contain the index *right*. So the next logical step is to move the right pointer one step to its left.

[1, 2, 3, 5, 8]
↑ ↑
left right

Now the pair sum is $1 + 5 = 6$, which is $< target$. How many pairs with one of the *index = left* that satisfy the condition? You can tell by the difference between *right* and *left* which is 3, namely $(1, 2)$, $(1, 3)$, and $(1, 5)$. Therefore, we move *left* one step to its right.

```
public int threeSumSmaller(int[] nums, int target) {
    Arrays.sort(nums);
    int sum = 0;
    for (int i = 0; i < nums.length - 2; i++) {
        sum += twoSumSmaller(nums, i + 1, target - nums[i]);
    }
    return sum;
}

private int twoSumSmaller(int[] nums, int startIndex, int target) {
    int sum = 0;
    int left = startIndex;
    int right = nums.length - 1;
    while (left < right) {
        if (nums[left] + nums[right] < target) {
            sum += right - left;
            left++;
        } else {
            right--;
        }
    }
    return sum;
}
```

Complexity analysis

- Time complexity: $O(n^2)$. The *twoSumSmaller* function takes $O(n)$ time because both *left* and *right* traverse at most n steps. Therefore, the overall time complexity is $O(n^2)$.
- Space complexity: $O(1)$.


Rate this article: ★★★★★

Previous

Next

Comments: 14


Sort By



Type comment here... (Markdown is supported)

Preview

Post



sha256pki

★ 552

August 9, 2017 6:42 AM

Sorting array, rearranges the array, so not sure how is it solving original question of finding i,j,k in unsorted array.

54


Up

Down

Share

Reply

SHOW 6 REPLIES



yehiahesham

★ 19

August 21, 2019 1:47 PM

is it me only, or the question was confusing about the i<j<k ? because most answers here are sorting the array which loses the positions to its values ! I mean if the problem wants any combination, it is poorly written !

13


Up

Down

Share

Reply

SHOW 4 REPLIES



sha256pki

★ 552

September 13, 2017 5:57 AM

Can I know why "right - left" is added to sum? I wonder how does it count distinct pairs of numbers between "left" and "right" that add upto less than target?

6


Up

Down

Share

Reply

SHOW 2 REPLIES



Chen_Xiang

★ 80

August 6, 2017 3:09 AM

I have a question for the twoSumSmaller:

```
private int twoSumSmaller(int[] nums, int startIndex, int target) {
    int sum = 0;
    for (int i = startIndex; i < nums.length - 1; i++) {
        int j = binarySearch(nums, i, target - nums[i]);
        sum += j - i;
    }
    return sum;
}
```

4


Up

Down

Share

Reply

Read More



RogerFederer

★ 855

December 11, 2017 2:19 AM

```
def threeSumSmaller(self, nums, target):
    """
    :type nums: List[int]
    :type target: int
    """
```

4


Up

Down

Share

Reply

SHOW 1 REPLY



Newsanev

★ 1138

April 9, 2019 7:32 PM

Hi, I was wondering do we need to take care of overflow problem? I think $target - nums[i]$ and $nums[left] + nums[right]$ have a chance to cause overflow. Correct me if I am wrong

2


Up

Down

Share

Reply

SHOW 1 REPLY



calvinchankf

★ 2977

March 31, 2019 12:36 PM

instead of customizing the binary search, we can also (re)use the lower bound binary search, and the target index is actually the result of the low bound binary search -1

```
func threeSumSmaller(nums []int, target int) int {
    // ...
}
```

0


Up

Down

Share

Reply

Read More



anku

★ 0

September 15, 2016 10:49 AM

How does the 2 pointer method take care of duplicates?

0


Up

Down

Share

Reply

SHOW 1 REPLY



piyush121

★ 184

August 28, 2016 4:43 AM

Couldn't have been better.


0

Up

Down

Share

Reply



tzookb

★ 0

2 days ago

how does "twoSumSmaller" sum the problem if I have this array:

```
[1,2,3,4,5]
```

0

Up

Down

Share

Reply

Read More

<

1

2

>