15. 3Sum

<https://leetcode.com/problems/3sum/>

Tags: 2-Pointer, 3-Pointer

1. **Listen**

**Problem Statement:**

Given an **integer array nums**, **return all the triplets [nums[i], nums[j], nums[k]]** such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

**Input:**

integer array **nums**

**Goal:**

find three numbers in the array that **add** up to **0**

**Return:**

return **all triplets [nums[i], nums[j], nums[k]]**

i != j

i != k

j != i

j != k

k != i

k != j

nums[i] + nums[j] + nums[k] == 0

1. **Examples**

Example 1:

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

**Output:** [[-1,-1,2],[-1,0,1]]

**Explanation:**

nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.

nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.

nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.

The distinct triplets are [-1,0,1] and [-1,-1,2].

Notice that the order of the output and the order of the triplets does not matter.

Example 2:

**Input:** nums = [0,1,1]

**Output:** []

**Explanation:** The only possible triplet does not sum up to 0.

**Constraints:**

* The solution set **must not contain duplicate triplets (indices)**.

There may be indices repeated in the triplet, as long as there is not a matching triplet already in the solution set.

* Indices cannot be equal to each other.

[-1,0,1,2,-4]

We cannot do [-1, -1, 2], because that would be [index 0, index 0, index 3]

The only solution here would be to do [-1, 0, 1]

* Array Length: 3 <= nums.length <= 3000 (array must be at least size 3)
* Index Value: -105 <= nums[i] <= 105

**Test Cases:**

* Array must be at least size 3, but there may or may not be triplets that add up to 0.

Test a case where there are no possible triplets that add up to 0.

== Size 3

> Size 3

1. **Brute Force**

Solution 1:

We can sort the input array in O(nlogn) time to make the inherit problem easier.

[-1,0,1,2,-1,-4] -> [-4,-1,-1,1,0,2]

Now, we will iterate over the array and use a sort of **three-pointer** technique.

So, we essentially need to find three numbers x, y, and z such that they add up to the given value. If we fix one of the numbers say x, we are left with the two-sum problem at hand!

We iterate over each index in the **nums** array. We will have a fixed pointer called **x** which represents our current place in our iteration over **nums**.

Let's call x **p1**.

For each **p1** over the array, we check to see if we can find two other indices in the array whose values add up to zero when added with **p1**.

These other two indices are y and z. Let’s call y **p2** and z **p3**.

Formally, we use two pointers (**p2** and **p3**) to check the rest of the array to see if we can find two other indices whose values add up to zero if added with **p1**.

This is a standard bi-directional 2Sum sweep of the remaining part of the array is effectively identical to <https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/>.

This solution takes O(n^2) time and takes O(1) space not counting the solution list.

1. **Optimize**

In this case, we would try to think of any possible bottlenecks, repeated code, or useless code. However, sometimes (rarely) our brute force solution is the most optimal solution. If time allows, try to think of a more optimal solution. But don't get stuck digging yourself in a hole. A brute force solution that works is better than no solution at all.

1. **Walkthrough**

We iterate over the array, so we start off at index 0, whose value is -4. This is p1.

Next, we check the rest of the array using p2 and p3.

The starting position of p2 starts one index ahead of where p1 is.

Pointer 3 starts at the end of the array.

while p2 and p3 do not overlap

If p1 + p2 + p3 == 0 then we have found a triplet.

Remove all adjacent duplicate elements.

Add it to the solution list.

update p2 and p3

Otherwise, if p1 + p2 + p3 < 0, we update the left pointer (pointer 2) up one.

Otherwise, if p1 + p2 + p3 > 0, we update the right pointer (pointer 3) down one.

1. **Implement**

Here is a first attempt at an implementation.

public List<List<Integer>> threeSum(int[] nums) {

Arrays.sort(nums);

List<List<Integer>> list = new LinkedList<>();

for(int i = 0; i < nums.length; i++)

{

if(nums[i] > 0) continue; // can’t have 0 in sorted array if nums[i] > 0

int p1 = i, p2 = i+1, p3 = nums.length-1;

while(p2 < p3)

{

if(nums[p1] + nums[p2] + nums[p3] == 0)

{

list.add(Arrays.asList(nums[p1], nums[p2], nums[p3]));

p2++;

p3--;

}

else if(nums[p1] + nums[p2] + nums[p3] < 0)

{

p2++;

}

else

{

p3--;

}

}

return list;

}

While this may work, we haven’t completely matched the problem constraints.

This solution includes duplicate triplets. Let’s try to eliminate duplicate triplets.

public List<List<Integer>> threeSum(int[] nums) {

Arrays.sort(nums);

List<List<Integer>> list = new LinkedList<>();

for(int i = 0; i < nums.length; i++)

{

if(i > 0 && nums[i] == nums[i-1]; // avoid duplicates in p1 value, saves time

if(nums[i] > 0) continue; // can’t sum up to 0 in sorted array if nums[i] > 0

int p1 = i, p2 = i+1, p3 = nums.length-1;

while(p2 < p3)

{

if(nums[p1] + nums[p2] + nums[p3] == 0)

{

while(lo < hi && nums[p2] == nums[p2+1]) p2++;

while(lo < hi && nums[p3] == nums[p3-1]) p3--;

list.add(Arrays.asList(nums[p1], nums[p2], nums[p3]));

p2++;

p3--;

}

else if(nums[p1] + nums[p2] + nums[p3] < 0)

{

p2++;

}

else

{

p3--;

}

}

return list;

}

1. **Test**

* Multiple possible distinct triplets

[]

* Multiple possible duplicate triplets

[]

* Many duplicate values

[]

* All 0’s

[]

* No possible triplets sum up to 0

[]