Note 2.

Table of Content:

Topic:

2: Two Pointers or Iterators

Two pointers is a pattern where two iterate through the data structure in tandem (song song) until one or both of the pointers hit a certain condition. Two pointers is often useful when searching pairs in a ***sorted array*** or ***linked list****;*

for example, when you have to compare each element of an array to its other elements.

Two pointers are needed because with just pointer, you would have to continually loop back through the array to find the answer. This back and forth with a single iterator is inefficient for time and space complexity – a concept referred to as asymptotic analysis. While the brute force or naïve solution with 1 pointer would work, it will produce something along the lines of O(n^2). In many cases, 2 pointers can help you find a solution with better space or runtime complexity.

Diagram

Description automatically generated

A picture containing text, clock

Description automatically generated

Diagram

Description automatically generated

--- Squares of a sorted array

I: an integer array

* by nums sorted in non-decreasing order. (aka increasing order)
* there are negative numbers

O: an array of the squares of each number sorted in non-decreasing order

C:

* 1 <= nums.length <= 10^4
* -10^4 <= nums[i] <= 10^4
* Nums is sorted is non-decreasing order

E: empty array.

Intuition

Since the Array A is sorted, loosely speaking it has some negative elements with squares in decreasing order, then some non-negative elements with squares in increasing order.

For example with [-3, -2, -1, 4, 5, 6], we have the negative part [-3, -2, -1] with squares [9, 4, 1], and the positive part [4, 5, 6] with squares [16, 25, 36]. Our strategy is to iterate over the negative part in reverse, and the positive part in the forward direction.

2 pointer technique is different than the sliding window technique are that:

1: 2 pointer technique: 1 pointer starts at the beginning, the other pointer starts at the end

2: sliding window technique is that: 2 pointers start at the same point.

Diagram, rectangle

Description automatically generated

Text, letter

Description automatically generated

Algorithm

We can use 2 pointers to read the positive and negative parts of the array – one pointer j in the positive direction, and another I in the negative direction.

Need to fill the result array.

Now that we are reading 2 increasing arrays (the squares of the elements), we can merge these arrays together using a 2 pointer technique.

**threeSum problem:**

Two sum, two sum II and 3 sum share similariy that the sum of elements must match the target exactly. A difference is that, instead of exactly one answer we need to find all unique triplets that sum to zero.

Before jumping in, let’s check the existing solutions and determin the best conceiable runtime (BCR) for 3 sum:

1: 2 sum uses a hashmap to find complement values, and therefore achieves O(N) time complexity

Approach 1: 2 pointers

We will follow the same 2 pointers patterns as in 2 sum II. It requires the array to be sorted. As our BCR is O(n^2), sorting the array would not change the overall time complexity.

To make the result contain 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.

After sorting the array, we move our pivot element nums[i[ and analyze elements to its right. We find all pairs whose sum is equal -nums[i] using the 2 pointer pattern, so that the sum of the pivot element (nums[i]) and the pair (-nums[i]) is equal to zero.

As a quick refresher, the pointers are initially set to the first and the last element respectively. We compare the sum of these two elements to the target. If it is smaller, we increment the lower pointer lo. Otherwise, we decrement the higher pointer hi. Thus, the sum always moves toward the target, and we “prune” pairs that would move it further away. Again, this works only if the array is sorted.

Algorithms:

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 twoSum II function:

* Set the left pointer to i + 1, and right pointer hi to the last index.
* While left pointer is smaller than right:
  + If sum of nums[i] + nums[left] + nums[right] is less than zero, increment left
* If sum is greater than zero, decrement right.
* Otherwise, we found a triplet:
  + Add it to the result res
  + Decrement right and increasement left
  + Increment left while the next value is the same as before to avoid duplicates in the result.

3: return the result res.