Triplet Sum to Zero (medium)

We'll cover the following

Problem Statement

Given an array of unsorted numbers, find all **unique triplets in it that add up to zero**.

Example 1:

```
Input: [-3, 0, 1, 2, -1, 1, -2]
Output: [-3, 1, 2], [-2, 0, 2], [-2, 1, 1], [-1, 0, 1]
Explanation: There are four unique triplets whose sum is equal to zero.
```

Example 2:

```
Input: [-5, 2, -1, -2, 3]
Output: [[-5, 2, 3], [-2, -1, 3]]
Explanation: There are two unique triplets whose sum is equal to zero.
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```java
import java.util.*;

class TripletSumToZero {

  public static List<List<Integer>> searchTriplets(int[] arr) {
    List<List<Integer>> triplets = new ArrayList<>();
    // TODO: Write your code here
    return triplets;
  }
}
```

Test | Save | Reset

Solution

This problem follows the **Two Pointers** pattern and shares similarities with Pair with Target Sum. A couple of differences are that the input array is not sorted and instead of a pair we need to find triplets with a target sum of zero.

To follow a similar approach, first, we will sort the array and then iterate through it taking one number at a time. Let's say during our iteration we are at number 'X', so we need to find 'Y' and 'Z' such that $X + Y + Z == 0$. At this stage, our problem translates into finding a pair whose sum is equal to "$-X$" (as from the above equation $Y + Z == -X$).

Another difference from Pair with Target Sum is that we need to find all the unique triplets. To handle this, we have to skip any duplicate number. Since we will be sorting the array, so all the duplicate numbers will be next to each other and are easier to skip.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```java
import java.util.*;
```

```java
class TripletSumToZero {

  public static List<List<Integer>> searchTriplets(int[] arr) {
    Arrays.sort(arr);
    List<List<Integer>> triplets = new ArrayList<>();
    for (int i = 0; i < arr.length - 2; i++) {
      if (i > 0 && arr[i] == arr[i - 1]) // skip same element to avoid duplicate triplets
        continue;
      searchPair(arr, -arr[i], i + 1, triplets);
    }

    return triplets;
  }

  private static void searchPair(int[] arr, int targetSum, int left, List<List<Integer>> triplets) {
    int right = arr.length - 1;
    while (left < right) {
      int currentSum = arr[left] + arr[right];
      if (currentSum == targetSum) { // found the triplet
        triplets.add(Arrays.asList(-targetSum, arr[left], arr[right]));
        left++;
        right--;
        while (left < right && arr[left] == arr[left - 1])
          left++; // skip same element to avoid duplicate triplets
        while (left < right && arr[right] == arr[right + 1])
          right--; // skip same element to avoid duplicate triplets
```

Run Save Reset

Time complexity

Sorting the array will take $O(N * logN)$. The `searchPair()` function will take $O(N)$. As we are calling `searchPair()` for every number in the input array, this means that overall `searchTriplets()` will take $O(N * logN + N^2)$, which is asymptotically equivalent to $O(N^2)$.

Space complexity

Ignoring the space required for the output array, the space complexity of the above algorithm will be $O(N)$ which is required for sorting.

Back

Squaring a Sorted Array (easy)

Next

Triplet Sum Close to Target (medium)

Completed