Triplets with Smaller Sum (medium)

We'll cover the following

Problem Statement

Given an array `arr` of unsorted numbers and a target sum, **count all triplets** in it such that `arr[i] + arr[j] + arr[k] < target` where `i`, `j`, and `k` are three different indices. Write a function to return the count of such triplets.

Example 1:

```
Input: [-1, 0, 2, 3], target=3
Output: 2
Explanation: There are two triplets whose sum is less than the target: [-1, 0, 3], [-1, 0, 2]
```

Example 2:

```
Input: [-1, 4, 2, 1, 3], target=5
Output: 4
Explanation: There are four triplets whose sum is less than the target:
   [-1, 1, 4], [-1, 1, 3], [-1, 1, 2], [-1, 2, 3]
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
1  import java.util.*;
2
3  class TripletWithSmallerSum {
4
5    public static int searchTriplets(int[] arr, int target) {
6      int count = -1;
7      // TODO: Write your code here
8      return count;
9    }
10 }
```

Test | Save | Reset

Solution

This problem follows the **Two Pointers** pattern and shares similarities with Triplet Sum to Zero. The only difference is that, in this problem, we need to find the triplets whose sum is less than the given target. To meet the condition `i != j != k` we need to make sure that each number is not used more than once.

Following a similar approach, first, we can 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 $X + Y + Z < target$. At this stage, our problem translates into finding a pair whose sum is less than "$target - X$" (as from the above equation $Y + Z == target - X$). We can use a similar approach as discussed in Triplet Sum to Zero.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```java
import java.util.*;

class TripletWithSmallerSum {

  public static int searchTriplets(int[] arr, int target) {
    Arrays.sort(arr);
    int count = 0;
    for (int i = 0; i < arr.length - 2; i++) {
      count += searchPair(arr, target - arr[i], i);
    }
    return count;
  }

  private static int searchPair(int[] arr, int targetSum, int first) {
    int count = 0;
    int left = first + 1, right = arr.length - 1;
    while (left < right) {
      if (arr[left] + arr[right] < targetSum) { // found the triplet
        // since arr[right] >= arr[left], therefore, we can replace arr[right] by any number between
        // left and right to get a sum less than the target sum
        count += right - left;
        left++;
      } else {
        right--; // we need a pair with a smaller sum
      }
    }
    return count;
  }
}
```

Run | Save | Reset

Time complexity

Sorting the array will take $O(N * logN)$. The `searchPair()` will take $O(N)$. So, overall `searchTriplets()` will take $O(N * logN + N^2)$, which is asymptotically equivalent to $O(N^2)$.

Space complexity

The space complexity of the above algorithm will be $O(N)$ which is required for sorting if we are not using an in-place sorting algorithm.

Similar Problems

Problem: Write a function to return the list of all such triplets instead of the count. How will the time complexity change in this case?

Solution: Following a similar approach we can create a list containing all the triplets. Here is the code - only the highlighted lines have changed:

Java | Python3 | C++ | JS

```java
import java.util.*;

class TripletWithSmallerSum {

  public static List<List<Integer>> searchTriplets(int[] arr, int target) {
    Arrays.sort(arr);
    List<List<Integer>> triplets = new ArrayList<>();
    for (int i = 0; i < arr.length - 2; i++) {
      searchPair(arr, target - arr[i], i, triplets);
    }
    return triplets;
  }

  private static void searchPair(int[] arr, int targetSum, int first, List<List<Integer>> triplets) {
    int left = first + 1, right = arr.length - 1;
    while (left < right) {
      if (arr[left] + arr[right] < targetSum) { // found the triplet
        // since arr[right] >= arr[left], therefore, we can replace arr[right] by any number between
        // left and right to get a sum less than the target sum
        for (int i = right; i > left; i--)
          triplets.add(Arrays.asList(arr[first], arr[left], arr[i]));
        left++;
      } else {
        right--; // we need a pair with a smaller sum
      }
    }
  }
}
```

Run | Save | Reset

Another simpler approach could be to check every triplet of the array with three nested loops and create a list

of triplets that meet the required condition.

Time complexity

Sorting the array will take $O(N * logN)$. The `searchPair()`, in this case, will take $O(N^2)$; the main `while` loop will run in $O(N)$ but the nested `for` loop can also take $O(N)$ - this will happen when the target sum is bigger than every triplet in the array.

So, overall `searchTriplets()` will take $O(N * logN + N^3)$, which is asymptotically equivalent to $O(N^3)$.

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

Triplet Sum Close to Target (medium)

Next

Subarrays with Product Less than a Ta...

Completed

Report an Issue Ask a Question