Verbetcetera

3Sum Leetcode problem

Task

Let's solve a popular interview question today: 3Sum
Given an array of *n* integers, are there elements *a*, *b*, *c* in *nums* such
0? Find all unique triplets in the array which give the sum of zero.

Note:
The solution set must not contain duplicate triplets.

> Given array nums = [-1, 0, 1, 2, -1, -4],
>
> A solution set is:
> [
> [-1, 0, 1],
> [-1, -1, 2]
>]

Let's keep in mind that the solution should be fast, otherwise we would get a TLE (Time Limit Exceeded) error.

Solution

Brute force

The first idea that comes to mind is to just iterate through all combinations of 3 numbers from the input array and check if their sum is equal to 0.
The complexity of this solution will be O(n3) because we need to consider every number for every of the three positions using 3 nested loops.

Now we should think of how we can enforce the *no duplicates* require
What does this mean?
It means that we should not consider the same value multiple times for the same

One solution to this problem is to just add all triplets to the hashset, thus eliminating all the duplicates.
But there is a simpler solution: sort the array and on every loop skip all values equal to value that has already been processed in the previous iteration.

If you are interested, you can view the listings for C++, Python or Java

But this solution will give us TLE error, meaning that it is too slow.

We need to understand how we can speed up our solution and make it O(n2) complexity.

Two pointer solution

We are looking for triplets such that $a + b + c = 0$.
Let's fix single number x as a and rearrange the equation:

> $x + b + c = 0$
> $b + c = -x$

We reduced the original problem to finding such b and c that add up t

To solve this problem effectively we will use a *two pointer* technique.
For this solution to work we need to sort the array at first.
After this we create 2 pointers, *left* and *right,* to the first and last elem

While the *left* pointer is less then *right* we compute the *sum* of values kept at pointers.

There are three possibilities how *sum* compares to -*x*:

- *sum* < -*x*, we shift the *left* pointer to the right, because we want to increase the *sum*
- *sum* > -*x*, we shift the *right* pointer to the left, because we want to decrease the *sum*
- *sum* = -*x*, we found one of the solutions! We store triplet *a*, *b*, *c* it in the *result* variable, and shift the *left* pointer to the right

But there is a question we need to answer: couldn't we miss some pairs?
We will prove it by contradiction for the left pointer (the proof for the right pointer is the same).
Let's assume that we moved the left pointer multiple times (from position *left*0 *to left*n) and then moved *right* pointer once (from position *right*0 to *right*1) and there is a pair that we skipped.
For each of the steps *i* (0 < *i* < *n*) condition *left*i + *right*0 < -*x* was true, because *left* pointer was shifted to the right. We also know that r*ight*1 < right0. If we combine this inequalities we have *left*i + *right1* < *left*i + *right0* < -x meaning that our assumption was incorrect and there is no such *left*i and *right*1 that will give -x in sum.

Now we can finally define the algorithm:

1. Sort the array

- *left*, pointing to element right after *x*
- *right*, pointing to last element

4. While *left* pointer is less then *right* compare their *sum* to -x:

- *sum* < -*x*, shift the *left* pointer to the right
- *sum* > -*x*, shift the *right* pointer to the left
- *sum* = -*x*, store triplet *a*, *b*, *c* it in the *result* variable, and shift the *left* pointer to the right

One thing that we need to remember when shifting pointers and iterating over array is *no duplicates* requirement. But we can use the method from the brute force approach: skip all values equal to the value that has already been processed.

The complexity of this solution is O(n2) because we need to iterate over all the numbers (which is O(n)) and for every selected number find all pairs so that the sum would be 0 (which is O(n) using two pointer technique).

The code for this algorithm:

C++

```
vector<vector<int>> threeSum(vector<int>& nums) {
    vector<vector<int>> result;

    sort(begin(nums), end(nums));
```

```
        if (firstIdx > 0 && nums[firstIdx] == nums[firstIdx - 1]) {
            continue;
        }

        size_t secondIdx = firstIdx + 1;
        size_t thirdIdx = nums.size() - 1;

         int requiredSum = 0 - nums[firstIdx];

        while (secondIdx < thirdIdx) {
            int currentSum = nums[secondIdx] + nums[thirdIdx];

            if (currentSum < requiredSum) {
                ++secondIdx;
            } else if (currentSum > requiredSum) {
                --thirdIdx;
            } else {
                result.push_back(vector<int>{nums[firstIdx], nums[secondIdx],
nums[thirdIdx]});
                ++secondIdx;
                while (secondIdx < thirdIdx && nums[secondIdx] == num
{
                    ++secondIdx;
                }
```

```
        }

        return result;
    }
```

Python

```
def threeSum(self, nums):
    """
    :type nums: List[int]
    :rtype: List[List[int]]
    """
    result = []

    nums.sort()

    for first_idx in range(len(nums)):
        if first_idx > 0 and nums[first_idx] == nums[first_idx - 1]:
            continue

        second_idx = first_idx + 1
        third_idx = len(nums) - 1

        required_sum  = 0 - nums[first_idx];
```

```
            current_sum = nums[second_idx] + nums[third_idx]

            if current_sum < required_sum:
                second_idx += 1
            elif current_sum > required_sum:
                third_idx -= 1
            else:
                result.append([nums[first_idx], nums[second_idx], nums[third_idx]])
                second_idx += 1
                while second_idx < third_idx and nums[second_idx] == nums[second_idx
- 1]:
                    second_idx += 1

    return result
```

Java

```
public static void main(String[] args) {
    Solution solution = new Solution();

    int[] nums = {-1, 0, 1, 2, -1, -4};

    for (List<Integer> row: solution.threeSum(nums)){
        for (int val: row){
```

```
        System.out.println();
    }
}

public List<List<Integer>> threeSum(int[] nums) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();

    Arrays.sort(nums);

    for (int firstIdx = 0; firstIdx < nums.length - 2; ++firstIdx) {
        if (firstIdx > 0 && nums[firstIdx] == nums[firstIdx - 1]) {
            continue;
        }

        int secondIdx = firstIdx + 1;
        int thirdIdx = nums.length - 1;

        int requiredSum = 0 - nums[firstIdx];

        while (secondIdx < thirdIdx) {
            int currentSum = nums[secondIdx] + nums[thirdIdx];

            if (currentSum < requiredSum) {
                ++secondIdx;
```

Verbetcetera

```
            } else {
                result.add(List.of(nums[firstIdx], nums[secondIdx], nums[thirdIdx]));
                ++secondIdx;
                while (secondIdx < thirdIdx && nums[secondIdx] == nums[secondIdx - 1])
{
                    ++secondIdx;
                }
            }
        }
    }

    return result;
}
```

Extras to help you prepare for interview

System design interview is an important part of the interview process. Find the system design interview answer framework and a collection of freque interview questions in System Design Interview Preparation post.

05.10.2020 ENGINEERING

Verbetcetera