FizzBuzzed

3sum Solution - Top Interview Questions and Tricks #1 - 3sum and the 2 pointer technique

20 Apr 2018 12 mins read

‘3sum’ is one of the most popular technical interview questions. There are many variations, including 3sum closest, and ‘k-sum’ (where the interviewer chooses an arbitrary number ‘k’ to replace 3). Some evidence of it’s popularity:

- This quora question and this answer to it
- More than 1.5 million submissions and 1500 upvotes on leetcode.

The Problem - 3 sum (from leetcode)

Given an array `nums` of n integers, are there elements a, b, c in `nums` such that $a + b + c = 0$? Find all unique triplets in the array which give the sum of zero.

Note: The solution set must not contain duplicate triplets.

Example: Given array nums = [-1, 0, 1, 2, -1, -4],

A solution set is: [[-1, 0, 1], [-1, -1, 2]]

The Solution

3sum is a simple problem on the face of it, but there are a couple things that make it tricky:

1. Ensuring no duplicate outputs.
2. Creating the $O(n^2)$ solution that the interviewers expect (instead of the $O(n^3)$ straightforward solution).

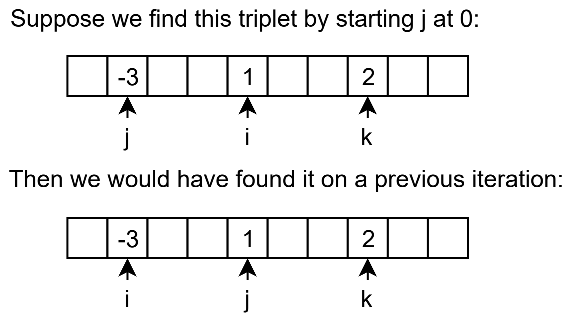
A basic, $O(n^3)$, solution to 3sum (ignoring the, 'don't output duplicates' constraint), is to simply iterate through each possible combination of 3 numbers from the array and then test if they add to 0. To do this, we can just use three for loops.

In C++:

```
vector<vector<int>> threeSum(const vector<int>& nums) {
  vector<vector<int>> output;
  for (int i = 0; i < nums.size(); ++i) {
    for (int j = i + 1; j < nums.size(); ++j) {
      for (int k = j + 1; k < nums.size(); ++k) {
        if (k != j + 1 && nums[k] == nums[k - 1]) continue;
        if (nums[i] + nums[j] + nums[k] == 0) {
          output.push_back({nums[i], nums[j], nums[k]});
        }
      }
    }
  }
  return output;
}
```

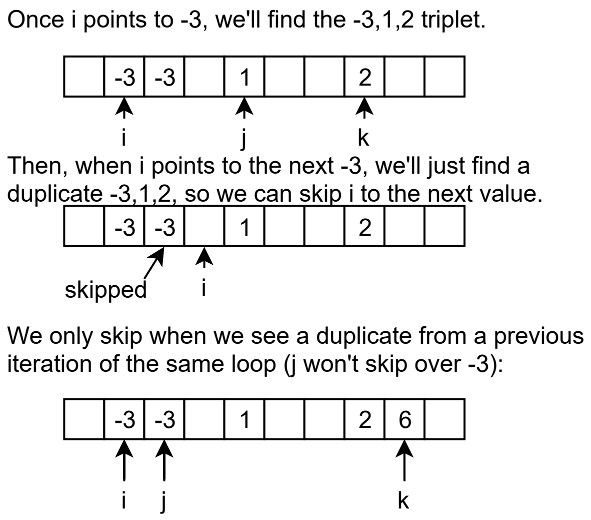
Note: You might think the 2nd and 3rd for loops (indexed by j and k) should start at 0 instead of i+1 and j+1 respectively (skipping over indices that are already used). It's okay to start where we do because:

Suppose we found a triplet, a_0, a_1, a_2 with j pointing to a_0 and i pointing to a_1 with $j < i$. Then, we would have found the same triplet with i pointing to a_0 and j pointing to a_1 on a previous iteration. (See picture for illustration).


Suppose we find this triplet by starting j at 0:
-3
1
2
j
i
k
Then we would have found it on a previous iteration:
-3
1
2
i
j
k

Let's modify our solution to take into account the 'no duplicates' constraint. Right now, to remove duplicates, a very common solution would be to unique-ify the values using a set. We can do that, but it's a bit of a hassle for 3sum. We can remove duplicates in a simpler way.

The simple way to remove duplicates is to stop considering the same value for the same index more than once. (Don't let i, j or k refer to the same value in the array twice). Let x refer to some value in the array. Once we have set $i = x$ and found all y, z such that $x + y + z = 0$, we never have to consider setting $i = x$ again. One way to do this is to sort the array first, then simply skip over duplicates inside each loop.


Once i points to -3, we'll find the -3,1,2 triplet.
-3
-3
1
2
i
j
k
Then, when i points to the next -3, we'll just find a duplicate -3,1,2, so we can skip i to the next value.
-3
-3
1
2
skipped
i
We only skip when we see a duplicate from a previous iteration of the same loop (j won't skip over -3):
-3
-3
1
2
6
i
j
k

A brute force solution with working code that removes duplicates is as follows:

```
vector<vector<int>> threeSum(vector<int>& nums) {
  vector<vector<int>> output;
  sort(nums.begin(), nums.end());
  for (int i = 0; i < nums.size(); ++i) {
    if (i != 0 && nums[i] == nums[i - 1]) continue;
    for (int j = i + 1; j < nums.size(); ++j) {
      if (j != i + 1 && nums[j] == nums[j - 1]) continue;
      for (int k = j + 1; k < nums.size(); ++k) {
        if (k != j + 1 && nums[k] == nums[k - 1]) continue;
        if (nums[i] + nums[j] + nums[k] == 0) {
          output.push_back({nums[i], nums[j], nums[k]});
        }
      }
    }
  }
  return output;
}
```

Now, this solution is correct, but interviewers won't accept just the $O(n^3)$ solution. We want to reduce the complexity of our answer to $O(n^2)$. How can we do it?

There are two common ways:

1. Use a hash map
2. Use the 'two pointer' trick

The hash map solution

We're searching for triplets such that $a + b + c = 0$. Let's suppose we choose a value, T from the array, and then set $a = T$. Then, rearranging the equation ($T + b + c = 0$ becomes $b + c = -T$) and the problem becomes finding b and c from the rest of the values in the array such that $b + c = -T$.

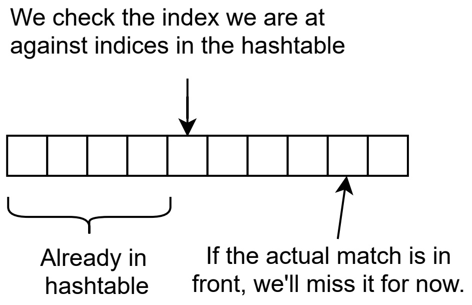
This sub problem is the same as another, related leetcode problem (2sum).

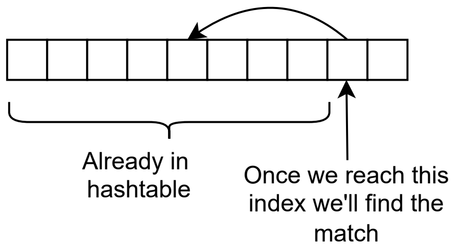
How can we solve our problem, 2sum, in O(n)?

Let's again imagine we fix a single value, B. Then, we are search for a value, V, such $V + B = -T$. Does $V = -T - B$ exist in the array? How can we quickly determine this? Use a hash map. If we convert the array to a hash map, we will be able to quickly search for any value.

The only problem is that we might accidentally use the same value twice (e.g. if the array is just [-2, 4, 10] we should have no outputs, but if we simply create a hash table out of the whole array, we might accidentally use -2 twice and output [-2, 4, -2]).

To avoid making this mistake, we can build the hash map as we go through the array:


We check the index we are at against indices in the hashtable
Already in hashtable
If the actual match is in front, we'll miss it for now.



Using this idea, the code for solving 3sum with a hashmap to get a $O(n^2)$ solution is:

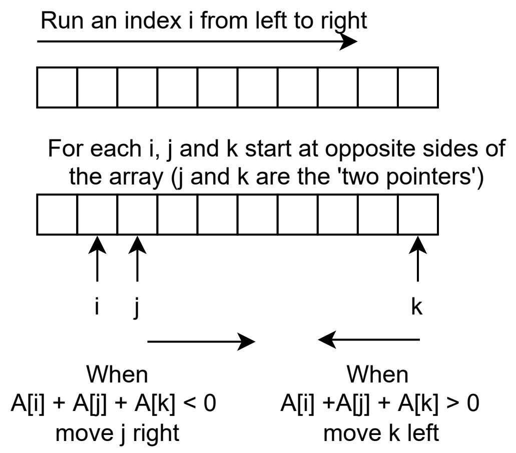
```
vector<vector<int>> threeSum(vector<int>& nums) {
  vector<vector<int>> output;
  sort(nums.begin(), nums.end());
  for (int i = 0; i < nums.size(); ++i) {
    if (i != 0 && nums[i] == nums[i - 1]) continue;
    unordered_set<int> seen;
    for (int j = i + 1; j < nums.size(); ++j) {
      if (seen.count(-nums[i] - nums[j])) {
        output.push_back({nums[i], nums[j], -nums[i] - nums[j]});
        // Skip duplicates
        while (nums[j+1] == nums[j] && j + 1 < nums.size()) ++j;
      }
      seen.insert(nums[j]);
    }
  }
  return output;
}
```

Note: Interestingly, this gets time limit exceeded on leetcode, but passes if we replace unordered_set with just set, which runs faster on the input sizes given by leetcode, but makes the solution asymptotically $O(n^2 log(n))$.

The two pointer trick

The 'two pointer trick' gives a really nice solution to 3sum that doesn't require any extra data structures. It runs really quickly and some interviewers 'expect' this solution (which might be somewhat unfair, but now that you're seeing it, it's to your advantage).

For the two pointer solution, the array must first be sorted, then we can use the sorted structure to cut down the number of comparisons we do. The idea is shown in this picture:


Run an index i from left to right
For each i, j and k start at opposite sides of the array (j and k are the 'two pointers')
i
j
k
When A[i] + A[j] + A[k] < 0 move j right
When A[i] +A[j] + A[k] > 0 move k left

Once we have that idea, the code is as follows:

```
vector<vector<int>> threeSum(vector<int>& nums) {
  vector<vector<int>> output;
  sort(nums.begin(), nums.end());
  for (int i = 0; i < nums.size(); ++i) {
    // Never let i refer to the same value twice to avoid duplicates.
    if (i != 0 && nums[i] == nums[i - 1]) continue;
```

```
      int j = i + 1;
      int k = nums.size() - 1;
      while (j < k) {
        if (nums[i] + nums[j] + nums[k] == 0) {
          output.push_back({nums[i], nums[j], nums[k]});
          ++j;
          // Never let j refer to the same value twice (in an output) to avoid duplica
          while (j < k && nums[j] == nums[j-1]) ++j;
        } else if (nums[i] + nums[j] + nums[k] < 0) {
          ++j;
        } else {
          --k;
        }
      }
    }
    return output;
```

There's one thing that might not be clear for this solution. Does it really get all the pairs? Couldn't we miss some?

Let's suppose we move k left many times, then move j forward once, couldn't j match with any of the values that k skipped over?

Let's call the value that j was at to being with j_0 and the value that j is at now j_1. Similarly, let's call k's initial value k_0 and k's final value k_n.

Let's call the value being searched for T. We know that, for each of the values $k_i (0 < i < n)$ (the values that k skipped over) $j_0 + k_i > 0$ (otherwise the algorithm wouldn't have skipped over those k). We also know that $j_1 > j_0$. Combining these inequalities we have $j_1 + k_i > 0$ (i.e. definitely not = to 0).

Using this same logic the other direction, we can see that j won't skip possible values as well.

Conclusion

3sum, 2sum and other variants are extremely popular interview questions. When you are asked 2sum, you can usually just use the hash table solution. When you are asked 3sum, interviewers usually look for the 2 pointer solution. Handling duplicates is tricky at first, but easy when you understand it. Make sure you practice 3sum, 2sum, and variants like 3sum closest (all found on leetcode).

If you want more advice on the most frequently asked interview questions, please sign up below:

Previous
‹ **Why you failed your tech in...**

Next
Interview with a Distribute... ›