3Sum

LeetCode Admin Apr 23, 2020

Solution

This problem is a follow-up of Two Sum, and it is a good idea to first take a look at Two Sum and Two Sum II. An interviewer may ask to solve Two Sum first, and then throw 3Sum at you. Pay attention to subtle differences in problem description and try to re-use existing solutions!

Two Sum, Two Sum II and 3Sum share a similarity 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 determine the best conceivable runtime (BCR) for 3Sum:

1. Two Sum uses a hashmap to find complement values, and therefore achieves $\mathcal{O}(N)$ time complexity.
2. Two Sum II uses the two pointers pattern and also has $\mathcal{O}(N)$ time complexity for a sorted array. We can use this approach for any array if we sort it first, which bumps the time complexity to $\mathcal{O}(n \log n)$.

Considering that there is one more dimension in 3Sum, it sounds reasonable to shoot for $\mathcal{O}(n^2)$ time complexity as our BCR.

Approach 1: Two Pointers

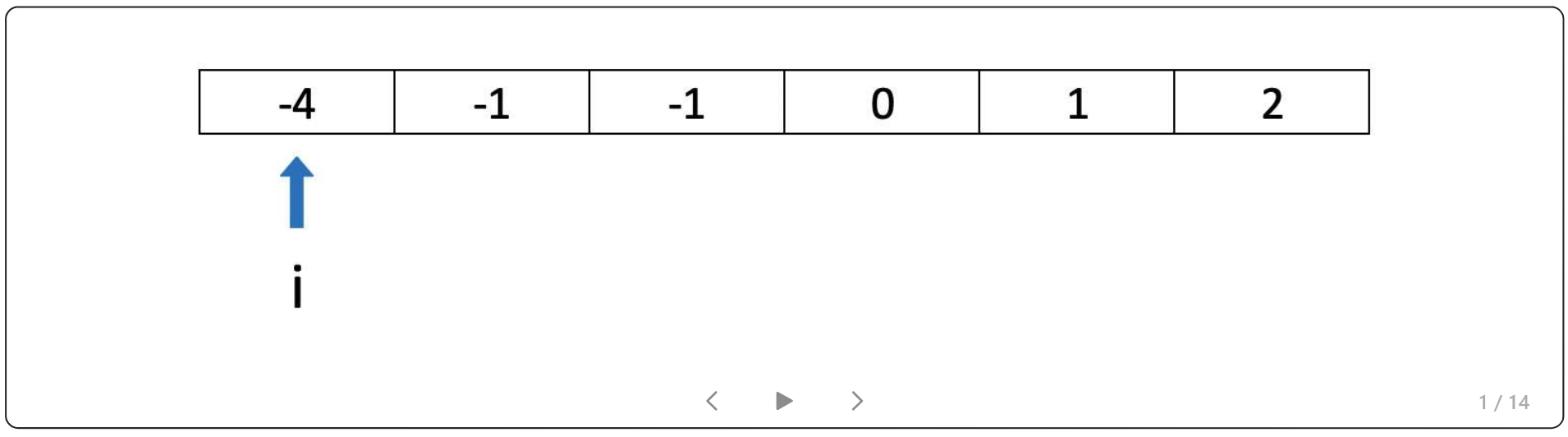
We will follow the same two pointers pattern as in Two Sum II. It requires the array to be sorted, so we'll do that first. As our BCR is $\mathcal{O}(n^2)$, sorting the array would not change the overall time complexity.

To make sure the result contains 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.

> If you are wondering how to solve this problem without sorting the array, go over the "No-Sort" approach below. There are cases when that approach is preferable, and your interviewer may probe your knowledge there.

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 two pointers 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. Head to the Two Sum II solution for the detailed explanation.



Algorithm

The implementation is straightforward - we just need to modify `twoSumII` to produce triplets and skip repeating values.

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 `twoSumII` function:

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

3. Return the result `res`.

C++ | Java | Python3 — Copy

```java
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> res = new ArrayList<>();
        for (int i = 0; i < nums.length && nums[i] <= 0; ++i)
            if (i == 0 || nums[i - 1] != nums[i]) {
                twoSumII(nums, i, res);
            }
        return res;
    }
    void twoSumII(int[] nums, int i, List<List<Integer>> res) {
        int lo = i + 1, hi = nums.length - 1;
        while (lo < hi) {
            int sum = nums[i] + nums[lo] + nums[hi];
            if (sum < 0) {
                ++lo;
            } else if (sum > 0) {
                --hi;
            } else {
                res.add(Arrays.asList(nums[i], nums[lo++], nums[hi--]));
                while (lo < hi && nums[lo] == nums[lo - 1])
                    ++lo;
            }
        }
    }
}
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(n^2)$. `twoSumII` is $\mathcal{O}(n)$, and we call it n times.
  Sorting the array takes $\mathcal{O}(n \log n)$, so overall complexity is $\mathcal{O}(n \log n + n^2)$. This is asymptotically equivalent to $\mathcal{O}(n^2)$.
- Space Complexity: from $\mathcal{O}(\log n)$ to $\mathcal{O}(n)$, depending on the implementation of the sorting algorithm. For the purpose of complexity analysis, we ignore the memory required for the output.

Approach 2: Hashset

Since triplets must sum up to the target value, we can try the hash table approach from the Two Sum solution. This approach won't work, however, if the sum is not necessarily equal to the target, like in 3Sum Smaller and 3Sum Closest.

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 Two Sum: One-pass Hash Table approach, so that the sum of the pivot element (`nums[i]`) and the pair (`-nums[i]`) is equal to zero.

To do that, we process each element `nums[j]` to the right of the pivot, and check whether a complement `-nums[i] - nums[j]` is already in the hashset. If it is, we found a triplet. Then, we add `nums[j]` to the hashset, so it can be used as a complement from that point on.

Like in the approach above, we will also sort the array so we can skip repeated values. We provide a different way to avoid duplicates in the "No-Sort" approach below.

Algorithm

The main function is the same as in the Two Pointers approach above. Here, we use `twoSum` (instead of `twoSumII`), modified to produce triplets and skip repeating values.

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 `twoSum` for the current position `i`.
2. For `twoSum` function:
    - For each index `j > i` in `A`:
        - Compute `complement` value as `-nums[i] - nums[j]`.
        - If `complement` exists in hashset `seen`:
            - We found a triplet - add it to the result `res`.
            - Increment `j` while the next value is the same as before to avoid duplicates in the result.
        - Add `nums[j]` to hashset `seen`
3. Return the result `res`.

C++ | Java | Python3 — Copy

```java
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> res = new ArrayList<>();
        for (int i = 0; i < nums.length && nums[i] <= 0; ++i)
            if (i == 0 || nums[i - 1] != nums[i]) {
                twoSum(nums, i, res);
            }
        return res;
    }
    void twoSum(int[] nums, int i, List<List<Integer>> res) {
        var seen = new HashSet<Integer>();
        for (int j = i + 1; j < nums.length; ++j) {
            int complement = -nums[i] - nums[j];
            if (seen.contains(complement)) {
                res.add(Arrays.asList(nums[i], nums[j], complement));
                while (j + 1 < nums.length && nums[j] == nums[j + 1])
                    ++j;
            }
            seen.add(nums[j]);
        }
    }
}
```

- Time Complexity: $\mathcal{O}(n^2)$. `twoSum` is $\mathcal{O}(n)$, and we call it n times.

  Sorting the array takes $\mathcal{O}(n \log n)$, so overall complexity is $\mathcal{O}(n \log n + n^2)$. This is asymptotically equivalent to $\mathcal{O}(n^2)$.
- Space Complexity: $\mathcal{O}(n)$ for the hashset.

Approach 3: "No-Sort"

What if you cannot modify the input array, and you want to avoid copying it due to memory constraints?

We can adapt the hashset approach above to work for an unsorted array. We can put a combination of three values into a hashset to avoid duplicates. Values in a combination should be ordered (e.g. ascending). Otherwise, we can have results with the same values in the different positions.

Algorithm

The algorithm is similar to the hashset approach above. We just need to add few optimizations so that it works efficiently for repeated values:

1. Use another hashset `dups` to skip duplicates in the outer loop.
    - Without this optimization, the submission will time out for the test case with 3,000 zeroes. This case is handled naturally when the array is sorted.

2. Instead of re-populating a hashset every time in the inner loop, we can use a hashmap and populate it once. Values in the hashmap will indicate whether we have encountered that element in the current iteration. When we process `nums[j]` in the inner loop, we set its hashmap value to `i`. This indicates that we can now use `nums[j]` as a complement for `nums[i]`.
    - This is more like a trick to compensate for container overheads. The effect varies by language, e.g. for C++ it cuts the runtime in half. Without this trick the submission may time out.

C++ | Java | Python3 — Copy

```java
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        Set<List<Integer>> res = new HashSet<>();
        Set<Integer> dups = new HashSet<>();
        Map<Integer, Integer> seen = new HashMap<>();
        for (int i = 0; i < nums.length; ++i)
            if (dups.add(nums[i])) {
                for (int j = i + 1; j < nums.length; ++j) {
                    int complement = -nums[i] - nums[j];
                    if (seen.containsKey(complement) && seen.get(complement) == i) {
                        List<Integer> triplet = Arrays.asList(nums[i], nums[j], complement);
                        Collections.sort(triplet);
                        res.add(triplet);
                    }
                    seen.put(nums[j], i);
                }
            }
        return new ArrayList(res);
    }
}
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(n^2)$. We have outer and inner loops, each going through n elements.

  While the asymptotic complexity is the same, this algorithm is noticeably slower than the previous approach. Lookups in a hashset, though requiring a constant time, are expensive compared to the direct memory access.

- Space Complexity: $\mathcal{O}(n)$ for the hashset/hashmap.

  For the purpose of complexity analysis, we ignore the memory required for the output. However, in this approach we also store output in the hashset for deduplication. In the worst case, there could be $\mathcal{O}(n^2)$ triplets in the output, like for this example: `[-k, -k + 1, ..., -1, 0, 1, ... k - 1, k]`. Adding a new number to this sequence will produce `n / 3` new triplets.

Further Thoughts

This is a well-known problem with many variations and its own Wikipedia page.

For an interview, we recommend focusing on the Two Pointers approach above. It's easier to get it right and adapt for other variations of 3Sum. Interviewers love asking follow-up problems like 3Sum Smaller and 3Sum Closest!

Share

Comments (222) Sort by: Best

Type comment here... (Markdown supported)

Preview Comment

Article Commenting Rules

1. This comment section is for questions and comments regarding this **LeetCode article**. All posts must respect our LeetCode Community Rules.

2. Concerns about errors or bugs in the article, problem description, or test cases should be posted on LeetCode Feedback, so that our team can address them.

votrubac Aug 22, 2021

Addressing several question regarding the "No-Sort" approach. We recommend approach 1 for interviews, and approach 3 is to address a possible follow-up question (e.g. "what if you cannot sort the array"). The point here is that it's possible, though the efficiency would heavily depend on the input. If we have a very large array with many duplicates and a few matching triplets, the "No-Sort" approach would be more memory efficient.

Figuring out and communicating pros and cons of each approach is very important during an interview.

467 Show 11 Replies Reply

yazmatazz May 14, 2020

This should be a hard question

1.1K Show 12 Replies Reply

ANDi_numbers Jul 13, 2020

the max input size of 3000 is too much. I used a similar hashtable solution (N^2) and it didn't pass.

106 Show 8 Replies Reply

wilderfield May 24, 2020

I personally prefer to build up the results as tuples inside of a hashset. This way you don't need to add complicated logic for checking duplicates. How do others feel about this?

75 Show 6 Replies Reply

sm1 Nov 25, 2021

In the first approach, when we find a triplet: "Increment lo while the next value is the same as before to avoid duplicates in the result." Why don't we do this step for the hi pointer?

24 Show 2 Replies Reply

Elir Feb 02, 2022

Read more

17 Show 3 Replies Reply

ztztzt8888 May 04, 2020

I use a single set to keep the lists from the get-go, and it is working just fine.

```
public List<List<Integer>> threeSum(int[] array) {
    Arrays.sort(array);
    Set<List<Integer>> triplets = new HashSet<>();
    for (int left = 0; left < array.length - 2; left++) {
        int mid = left + 1;
        int right = array.length - 1;
        while (mid < right) {
            int sum = array[left] + array[mid] + array[right];
```

39 Show 6 Replies Reply

mqh7 Dec 16, 2020

how come the space complexity isn't O(1) for two pointer method if we don't use any extra space in approach 1?

11 Show 5 Replies Reply

BaryLove Oct 24, 2021

in solution 1, why are we only skipping duplicates from the start and not from the end as well?

8 Show 2 Replies Reply

rexthk Jan 03, 2021

The lowest possible space complexity for approach #1 should be O(1) ? Quicksort requires only constant space.

8 Show 2 Replies Reply

1 2 3 4 5 6 ... 23