18. 4Sum

June 2, 2020 | 12.3K views

Average Rating: 3.54 (26 votes)

Given an array `nums` of n integers and an integer `target`, are there elements a, b, c, and d in `nums` such that $a + b + c + d$ = `target`? Find all unique quadruplets in the array which gives the sum of `target`.

Note:

The solution set must not contain duplicate quadruplets.

Example:

```
Given array nums = [1, 0, -1, 0, -2, 2], and target = 0.

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

Solution

This problem is a follow-up of 3Sum, so take a look at that problem first if you haven't. 4Sum and 3Sum are very similar; the difference is that we are looking for unique quadruplets instead of triplets.

As you see, 3Sum just wraps Two Sum in an outer loop. As it iterates through each value `v`, it finds all pairs whose sum is equal to `target - v` using one of these approaches:

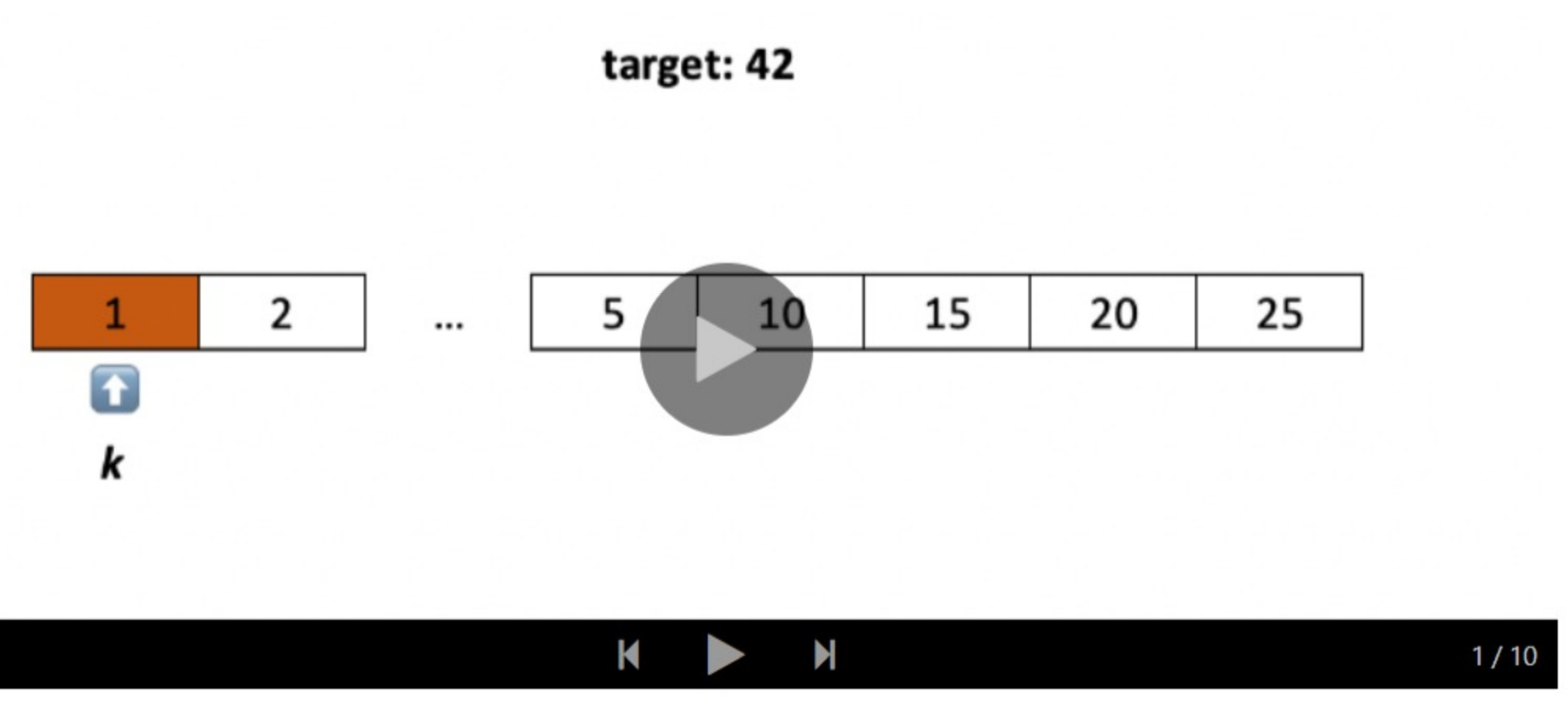
1. Two Sum uses a hash set to check for a matching value.
2. Two Sum II uses the two pointers pattern in a sorted array.

Following a similar logic, we can implement 4Sum by wrapping 3Sum in another loop. But wait - there is a catch. If an interviewer asks you to solve 4Sum, they can follow-up with 5Sum, 6Sum, and so on. What they are really expecting at this point is a kSum solution. Therefore, we will focus on a generalized implementation here.

Approach 1: Two Pointers

The two pointers pattern requires the array to be sorted, so we do that first. Also, it's easier to deal with duplicates if the array is sorted: repeated values are next to each other and easy to skip.

For 3Sum, we enumerate each value in a single loop, and use the two pointers pattern for the rest of the array. For kSum, we will have `k - 2` nested loops to enumerate all combinations of `k - 2` values.



Algorithm

We can implement `k - 2` loops using a recursion. We will pass the starting point and `k` as the parameters. When `k == 2`, we will call `twoSum`, terminating the recursion.

1. For the main function:
   - Sort the input array `nums`.
   - Call `kSum` with `start = 0`, `k = 4`, and `target`, and return the result.
2. For `kSum` function:
   - Check if the sum of `k` smallest values is greater than `target`, or the sum of `k` largest values is smaller than `target`. Since the array is sorted, the smallest value is `nums[start]`, and largest - the last element in `nums`.
     - If so, no need to continue - there are no `k` elements that sum to `target`.
   - If `k` equals `2`, call `twoSum` and return the result.
   - Iterate `i` through the array from `start`:
     - If the current value is the same as the one before, skip it.
     - Recursively call `kSum` with `start = i + 1`, `k = k - 1`, and `target - nums[i]`.
     - For each returned `set` of values:
       - Include the current value `nums[i]` into `set`.
       - Add `set` to the result `res`.
   - Return the result `res`.
3. For `twoSum` function:
   - Set the low pointer `lo` to `start`, and high pointer `hi` to the last index.
   - While low pointer is smaller than high:
     - If the sum of `nums[lo]` and `nums[hi]` is less than `target`, increment `lo`.
       - Also increment `lo` if the value is the same as for `lo - 1`.
     - If the sum is greater than `target`, decrement `hi`.
       - Also decrement `hi` if the value is the same as for `hi + 1`.
     - Otherwise, we found a pair:
       - Add it to the result `res`.
       - Decrement `hi` and increment `lo`.
   - Return the result `res`.

C++ Java Python3 Copy

```python
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        def kSum(nums: List[int], target: int, k: int) -> List[List[int]]:
            res = []
            if len(nums) == 0 or nums[0] * k > target or target > nums[-1] * k:
                return res
            if k == 2:
                return twoSum(nums, target)
            for i in range(len(nums)):
                if i == 0 or nums[i - 1] != nums[i]:
                    for _, set in enumerate(kSum(nums[i + 1:], target - nums[i], k - 1)):
                        res.append([nums[i]] + set)
            return res

        def twoSum(nums: List[int], target: int) -> List[List[int]]:
            res = []
            lo, hi = 0, len(nums) - 1
            while (lo < hi):
                sum = nums[lo] + nums[hi]
                if sum < target or (lo > 0 and nums[lo] == nums[lo - 1]):
                    lo += 1
                elif sum > target or (hi < len(nums) - 1 and nums[hi] == nums[hi + 1]):
                    hi -= 1
                else:
                    res.append([nums[lo], nums[hi]])
                    lo += 1
                    hi -= 1
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(n^{k-1})$, or $\mathcal{O}(n^3)$ for 4Sum. We have $k - 2$ loops, and `twoSum` is $\mathcal{O}(n)$.

  Note that for $k > 2$, sorting the array does not change the overall time complexity.

- Space Complexity: $\mathcal{O}(n)$. We need $\mathcal{O}(k)$ space for the recursion. k can be the same as n in the worst case for the generalized algorithm.

  Note that, for the purpose of complexity analysis, we ignore the memory required for the output.

Approach 2: Hash Set

Since elements must sum up to the exact target value, we can also use the Two Sum: One-pass Hash Table approach.

In 3Sum: Hash Set, we solved the problem without sorting the array. To do that, we needed to sort values within triplets, and track them in a hash set. Doing the same for k values could be impractical.

So, for this approach, we will also sort the array and skip duplicates the same way as in the Two Pointers approach above. Thus, the code will only differ in the `twoSum` implementation.

Algorithm

`twoSum` implementation here is almost the same as in Two Sum: One-pass Hash Table. The only difference is the check to avoid duplicates. Since the array is sorted, we can just compare the found pair with the last one in the result `res`.

C++ Java Python3 Copy

```python
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        def kSum(nums: List[int], target: int, k: int) -> List[List[int]]:
            if len(nums) == 0 or nums[0] * k > target or target > nums[-1] * k:
                return []
            if k == 2:
                return twoSum(nums, target)
            res = []
            for i in range(len(nums)):
                if i == 0 or nums[i - 1] != nums[i]:
                    for _, set in enumerate(kSum(nums[i + 1:], target - nums[i], k - 1)):
                        res.append([nums[i]] + set)
            return res

        def twoSum(nums: List[int], target: int) -> List[List[int]]:
            res = []
            s = set()
            for i in range(len(nums)):
                if len(res) == 0 or res[-1][1] != nums[i]:
                    if target - nums[i] in s:
                        res.append([target - nums[i], nums[i]])
                s.add(nums[i])
            return res

        nums.sort()
        return kSum(nums, target, 4)
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(n^{k-1})$, or $\mathcal{O}(n^3)$ for 4Sum. We have $k - 2$ loops iterating over n elements, and `twoSum` is $\mathcal{O}(n)$.

  Note that for $k > 2$, sorting the array does not change the overall time complexity.

- Space Complexity: $\mathcal{O}(n)$ for the hash set. The space needed for the recursion will not exceed $\mathcal{O}(n)$.

Rate this article: