Solution Review: Problem Challenge 1

We'll cover the following

Quadruple Sum to Target (medium)

Given an array of unsorted numbers and a target number, find all **unique quadruplets** in it, whose **sum is equal to the target number**.

Example 1:

```
Input: [4, 1, 2, -1, 1, -3], target=1
Output: [-3, -1, 1, 4], [-3, 1, 1, 2]
Explanation: Both the quadruplets add up to the target.
```

Example 2:

```
Input: [2, 0, -1, 1, -2, 2], target=2
Output: [-2, 0, 2, 2], [-1, 0, 1, 2]
Explanation: Both the quadruplets add up to the target.
```

Solution

This problem follows the **Two Pointers** pattern and shares similarities with Triplet Sum to Zero.

We can follow a similar approach to iterate through the array, taking one number at a time. At every step during the iteration, we will search for the quadruplets similar to Triplet Sum to Zero whose sum is equal to the given target.

Code

Here is what our algorithm will look like:

Java Python3 C++ JS

```
def search_quadruplets(arr, target):
  arr.sort()
  quadruplets = []
  for i in range(0, len(arr)-3):
    # skip same element to avoid duplicate quadruplets
    if i > 0 and arr[i] == arr[i - 1]:
      continue
    for j in range(i + 1, len(arr)-2):
      # skip same element to avoid duplicate quadruplets
      if j > i + 1 and arr[j] == arr[j - 1]:
        continue
      search_pairs(arr, target, i, j, quadruplets)
  return quadruplets


def search_pairs(arr, target_sum, first, second, quadruplets):
  left = second + 1
  right = len(arr) - 1
  while (left < right):
    quad_sum = arr[first] + arr[second] + arr[left] + arr[right]
    if quad_sum == target_sum:  # found the quadruplet
```

```python
    if quad_sum == target_sum:  # found the quadruplet
      quadruplets.append(
        [arr[first], arr[second], arr[left], arr[right]])
      left += 1
      right -= 1
      while (left < right and arr[left] == arr[left - 1]):
        left += 1  # skip same element to avoid duplicate quadruplets
      while (left < right and arr[right] == arr[right + 1]):
        right -= 1  # skip same element to avoid duplicate quadruplets
    elif quad_sum < target_sum:
      left += 1  # we need a pair with a bigger sum
    else:
      right -= 1  # we need a pair with a smaller sum


def main():
  print(search_quadruplets([4, 1, 2, -1, 1, -3], 1))
  print(search_quadruplets([2, 0, -1, 1, -2, 2], 2))


main()
```

Run

Save Reset

Time complexity

Sorting the array will take $O(N * logN)$. Overall `searchQuadruplets()` will take $O(N * logN + N^3)$, which is asymptotically equivalent to $O(N^3)$.

Space complexity

The space complexity of the above algorithm will be $O(N)$ which is required for sorting.

← Back
Problem Challenge 1

Next →
Problem Challenge 2

✓ Completed

Report an Issue Ask a Question