2465. Number of Distinct Averages

Easy Array Hash Table Two Pointers Sorting Leetcode Link

Problem Description

The given problem involves an array of integers `nums`, which has an even number of elements. We are instructed to perform a series of operations until the array becomes empty. In each operation, we must remove the smallest and the largest numbers from the array then calculate their average. This process is repeated until no numbers are left in the array. Our goal is to determine how many *distinct averages* we can get from these operations. It is important to note that in case of multiple instances of the minimum or maximum values, any occurrence of them can be removed.

Intuition

Considering that we need to find the minimum and maximum values of the array to calculate averages, a straightforward approach would be to sort the array first. With the array sorted, the minimum value will always be at the beginning of the array, and the maximum value will be at the end.

By sorting the array, we simplify the problem as follows:

- The minimum number of the array will be `nums[0]`, the second smallest `nums[1]`, and so on.
- The maximum number will be `nums[-1]`, the second-largest `nums[-2]`, and similar for the other elements.

After each operation of removing the smallest and largest elements, the subsequent smallest and largest elements become the adjacent values (the next elements in the sorted array). Therefore, we avoid the need for repeated searching for min and max in an unsorted array.

To find the distinct averages:

- We iterate over half of the list (`len(nums) >> 1`), since every operation removes two elements.
- For each iteration, we calculate the average of `nums[i]` and `nums[-i - 1]`, which is effectively the average of the ith smallest and ith largest value in the array.
- We use a `set` to collect these averages, which automatically ensures that only distinct values are kept.
- The length of this set is the number of distinct averages we have calculated, which is what we want to return.

This algorithm is efficient because it sorts the array once, and then simply iterates through half of the array, resulting in a complexity of O(n log n) due to the sorting step.

Solution Approach

The solution uses Python's built-in sorting mechanism to organize the elements in `nums` from the smallest to the largest. By sorting the array, the algorithm simplifies the process of finding the smallest and largest elements in each step.

Here's a step-by-step explanation of the solution:

- First, `nums.sort()` is called to sort the array in place. After the sort, `nums[0]` will contain the smallest value, and `nums[-1]` will contain the largest value, and so on for the ith smallest and ith largest elements.

- The generator expression `(nums[i] + nums[-i - 1] for i in range(len(nums) >> 1))` then iterates through the first half of the elements in the sorted list. The expression `len(nums) >> 1` is an efficient way to divide the length of `nums` by 2, using a bit-shift to the right.

- In each iteration, `nums[i] + nums[-i - 1]` calculates the sum of the ith smallest and ith largest element, which is equivalent to finding the sum of the elements at the symmetric positions from the start and the end of the sorted array.

- This sum is divided by 2 to calculate the average, but since the division by 2 is redundant when only interested in uniqueness (it does not affect whether the values are unique), it is not performed explicitly.

- All of these sums (representing the averages) are then collected into a `set`. As sets only store distinct values, any duplicate averages calculated during the iteration will only appear once.

- Finally, `len(set(...))` returns the number of distinct elements in the set, which corresponds to the number of distinct averages that were calculated.

In terms of data structures and patterns used:

- An array/list data structure is the primary structure utilized.
- Sorting is the main algorithmic pattern applied.
- A set is used to deduplicate values and count distinct elements.

This solution is particularly elegant because it leverages the sorted order of the array and the property of set collections to avoid unnecessary computations and simplify logic.

Example Walkthrough

Let's use a small example to illustrate the solution approach described above.

Suppose we have the following array:

```
nums = [1, 3, 2, 6, 4, 5]
```

The problem requires us to continually remove the smallest and largest numbers, calculate their average, and determine the number of *distinct averages* we can get from these operations. Let's walk through this step by step:

Step 1: Sort the array. Sorting the array in increasing order will give us:

```
nums.sort()  # After sorting: [1, 2, 3, 4, 5, 6]
```

Step 2: Initialize an empty set to store unique averages.

```
averages = set()
```

Step 3: Remove the smallest and largest numbers and calculate their averages.

```
# Since there are six elements, we iterate over half of that, which is three elements.
length_half = len(nums) >> 1  # Equivalent to dividing the length of nums by 2

for i in range(length_half):
    minimum = nums[i]  # ith smallest after sorting
    maximum = nums[-i - 1]  # ith largest after sorting
    # We calculate the sum since the division by 2 won't affect uniqueness
    avg = minimum + maximum
    averages.add(avg)  # Add the sum (representing the average) to the set of unique averages
```

Iteration example:

- First iteration for `i=0`: minimum is `nums[0]` which is `1`; maximum is `nums[-1]` which is `6`. Their sum is `1 + 6 = 7`. Add `7` to the set of averages.
- Second iteration for `i=1`: minimum is `nums[1]` which is `2`; maximum is `nums[-2]` which is `5`. Their sum is `2 + 5 = 7`. As `7` is already present in the set, it's not added again.
- Third iteration for `i=2`: minimum is `nums[2]` which is `3`; maximum is `nums[-3]` which is `4`. Their sum is `3 + 4 = 7`. Again, already present.

Step 4: The set of averages now has distinct sums. After the iterations, our set of averages will be:

```
averages = {7}
```

Step 5: Get the number of unique averages. Finally, we get the unique count by measuring the length of the set `averages`:

```
unique_averages_count = len(averages)  # This will be 1
```

Based on this example, even though we calculated the average (its sum representation) three times, our set contains only one element. Therefore, the number of distinct averages in the `nums` array we started with is `1`.

This example illustrates that the solution approach is efficient and avoids unnecessary complexity by using sorting, taking advantage of the properties of the set, and simplifying the problem into one that can be solved in linear time after sorting.

Python Solution

```python
from typing import List

class Solution:
    def distinct_averages(self, nums: List[int]) -> int:
        # Sort the input list of numbers
        nums.sort()

        # Initialize an empty set to store unique averages
        unique_averages = set()

        # Calculate the average of each pair of numbers, one from the start
        # and one from the end of the list, moving towards the middle
        for i in range(len(nums) // 2):  # Integer division to get half-way index
            # Calculate the sum of the pair and add it to the set.
            # We don't actually divide by 2 since it's the average of two nums and
            # we are interested in distinct averages. The div by 2 won't affect uniqueness.
            average = nums[i] + nums[-i - 1]

            # Add the calculated sum (which represents an average) to the set
            unique_averages.add(average)

        # Return the number of unique averages
        return len(unique_averages)

```

Java Solution

```java
class Solution {
    public int distinctAverages(int[] nums) {
        // Sort the array to facilitate pairing of elements
        Arrays.sort(nums);

        // Create an array to count distinct averages.
        // Since the problem constraints are not given, assuming 201 is the maximum value based on the given code.
        int[] count = new int[201];

        // Get the length of the nums array
        int n = nums.length;

        // Initialize the variable to store the number of distinct averages
        int distinctCount = 0;

        // Loop through the first half of the sorted array
        for (int i = 0; i < n / 2; ++i) {
            // Calculate the average of the ith and its complement element (nums[i] + nums[n - i - 1])
            // and increase the count for this average.
            // We do not actually compute the average to avoid floating point arithmetic
            // since the problem seems to be working with integer addition only.
            int sum = nums[i] + nums[n - i - 1];
            if (++count[sum] == 1) {
                // If the count of a particular sum is 1, it means it is distinct, increase the distinctCount
                ++distinctCount;
            }
        }

        // Return the total count of distinct averages found
        return distinctCount;
    }
}

```

C++ Solution

```cpp
#include <algorithm> // For std::sort
#include <vector>
using namespace std;

class Solution {
public:
    int distinctAverages(vector<int>& nums) {
        // Sort the input vector in non-decreasing order.
        sort(nums.begin(), nums.end());

        // Initialize a count array of size 201 to store
        // the frequency of the sum of pairs.
        int countArray[201] = {};

        // Obtain the size of the input vector.
        int numElements = nums.size();

        // Initialize a variable to store the count of distinct averages.
        int distinctCount = 0;

        // Loop through the first half of the vector as we are creating pairs
        // that consist of one element from the first half and one from the second.
        for (int i = 0; i < numElements / 2; ++i) {
            // Calculate the sum of the current pair: the i-th element and its corresponding
            // element in the second half of the array (mirror position regarding the center).
            int pairSum = nums[i] + nums[numElements - i - 1];

            // If this sum appears for the first time, increase the distinct count.
            if (++countArray[pairSum] == 1) {
                ++distinctCount;
            }
        }

        // Return the count of distinct averages.
        return distinctCount;
    }
};

```

Typescript Solution

```typescript
// This function calculates the number of distinct averages that can be formed
// by the sum of pairs taken from the start and end of a sorted array.
function distinctAverages(nums: number[]): number {
    // Sort the array in non-decreasing order
    nums.sort((a, b) => a - b);

    // Initialize a frequency array to keep track of the distinct sums
    const frequency: number[] = Array(201).fill(0);

    // Variable to hold the number of distinct averages
    let distinctCount = 0;

    // Determine the length of 'nums' array
    const length = nums.length;

    // Iterate over the first half of the sorted array
    for (let i = 0; i < length >> 1; ++i) {
        // Calculate the sum of the current element and its corresponding element from the end
        const sum = nums[i] + nums[length - i - 1];

        // Increase the frequency count for the calculated sum
        frequency[sum]++;

        // If this is the first time the sum appears, increment the distinctCount
        if (frequency[sum] === 1) {
            distinctCount++;
        }
    }

    // Return the total number of distinct averages
    return distinctCount;
}

```

Time and Space Complexity

The time complexity of the code above is `O(n log n)` and the space complexity is `O(n)`.

Time Complexity

1. `nums.sort()`: Sorting the list of `n` numbers has a time complexity of `O(n log n)`.

2. The list comprehension `set(nums[i] + nums[-i - 1] for i in range(len(nums) >> 1))` iterates over the sorted list but only up to the halfway point, which is `n // 2` iterations. Although the iteration is linear in time with respect to the number of elements it processes, the dominant term for time complexity comes from the sorting step.

Therefore, combining both steps we get `O(n log n)`, which is the overall time complexity.

Space Complexity

1. The sorted in-place method `nums.sort()` does not use additional space other than a few variables for the sorting algorithm itself (`O(1)` space).

2. The list comprehension inside the `set` function creates a new list with potentially `n / 2` elements (in the worst-case scenario, where all elements are distinct before combining them), and then a `set` is created from this list. The space required for this set is proportional to the number of distinct sums, which is also up to `n / 2`. Hence, this gives us a space complexity of `O(n)`.

Combining both considerations, the overall space complexity is `O(n)`. This accounts for the space needed to store the unique sums in the worst-case scenario where all sums are distinct.