1471. The k Strongest Values in an Array

Medium Array Two Pointers Sorting

Problem Description

In this problem, we must find the "k" strongest values in an array of integers, `arr`. A number is considered stronger than another if its distance from the median of the array is greater. The distance is determined by the absolute value of the difference between a number and the median. If two numbers are equally distant from the median, the larger one is considered stronger. The median here is defined as the middle value of the sorted array — if the array has an odd number of elements, it's the exact middle one; with an even number of elements, it is the lower of the two middle values.

In sum, our task is to:

1. Identify the median of the array,
2. Determine how strong each number is in relation to the median and to other numbers,
3. Sort the numbers based on their strength, and
4. Return the "k" strongest values.

The array does not come in any sorted order, and the 'stronger' relationship defined between any two elements is non-standard, relying on both their distance from the median of the array and their value relative to one another.

Intuition

The initial intuition for this problem might lean toward a sorting solution because we need to organize the array based on certain criteria (i.e., which integers are strongest). However, this isn't a standard sort; we first need to identify the median, which will inform our custom sort criteria.

The first step is to sort the array to get the median value. Once we have that, we can sort the array again with a custom sorting function that prioritizes strength: it will sort by the distance from the median first (the larger distance, the stronger) and, if two numbers are equidistant from the median, by the numeric values themselves (the bigger number, the stronger).

The Python code implements this with two sorts—the first regular sorting to find the median and the second utilizing a lambda function that sorts by the negative absolute difference from median (to ensure reverse order for larger strengths) and then by numeric value also in reverse order for equal strengths. The list is sliced to only the `k` strongest according to our logic, and that's the end result.

Solution Approach

To solve this problem effectively, the `Solution` class implements the `getStrongest` method, utilizing a two-step sorting approach:

1. **First Sort to Find Median:**
   - To find the median, the array is sorted using Python's built-in sorting method. Once sorted, the median can be calculated by accessing the element at the index `(len(arr) - 1) >> 1`. The `>>` operator is a right bitwise shift by one position, equivalent to integer division by 2, thus centering us close to the middle. For an even length array, this will give us the lower of the two middle values as required.
2. **Second Sort for Finding the Strongest Elements:**
   - After determining the median, we need to sort the array again based on the strength criteria using a custom sorting key. This key is a lambda function: `lambda x: (-abs(x - m), -x)`. This function ensures that Python's sort method, which sorts in ascending order by default, will sort elements in descending order by their strength. It does this in the following way:
     - The absolute difference `abs(x - m)` is negated to sort in reverse (stronger numbers have a higher absolute difference, but we negate it to sort them first).
     - For elements with the same absolute difference from the median, the second part of the tuple, `-x`, is used to sort them in descending order (again, larger values are considered stronger).
3. **Selecting the Top `k` Strongest Elements:**
   - The final array sorted by strength is then sliced to only return the first `k` elements, which represent the strongest values based on our criteria.

By using both built-in sorting and custom key functions, this approach efficiently organizes the data to retrieve the strongest values. Additionally, it demonstrates an effective use of Python's sorting capabilities to handle the custom comparison logic inherent in this problem. The final line of code `return arr[:k]` simply returns the top `k` elements from the sorted array, which are the `k` strongest values by the problem's definition.

The overall time complexity of this solution is dominated by the sorting operations, which is `O(n log n)`, making this a fairly efficient solution for the given problem.

Example Walkthrough

Let's go through a small example to illustrate the solution approach—suppose we have an array `arr = [6, 7, 11, 7, 6, 8]` and we want to find the `k = 3` strongest values.

1. **First Sort to Find Median:**
   - Initially, we sort the array to find the median. After sorting, `arr` becomes `[6, 6, 7, 7, 8, 11]`.
   - To find the median, we check the value at index `(6 - 1) >> 1`, which simplifies to `2`. So, the median `m` is the value at index `2`, which is `7`.
2. **Second Sort for Finding the Strongest Elements:**
   - Next, we're going to sort the array based on the strength criteria in relation to the median.
   - We define our lambda function for the custom sorting key: `lambda x: (-abs(x - m), -x)`.
   - This means, for each element `x`, we are considering the tuple `(-abs(x - 7), -x)` for the sorting. Calculating these tuples, we get:
     - For `6`: `(-1, -6)`
     - For `7`: `(-0, -7)`
     - For `8`: `(-1, -8)`
     - For `11`: `(-4, -11)`
   - According to the sorting criteria, the array should be sorted in descending order of these tuples, which results in: `[11, 8, 6, 6, 7, 7]`. Notice that `11` comes first because it has the highest absolute difference from the median.
3. **Selecting the Top `k` Strongest Elements:**
   - Finally, we slice the sorted array to get the first `k` elements. Here, `k` is `3`, so we take the first three elements of our sorted array: `[11, 8, 6]`.
   - Thus, the `k = 3` strongest values in the array `arr` are `[11, 8, 6]`.

This walkthrough demonstrates how the solution strategy breaks down the task into finding the median, sorting by a custom strength comparison, and then selecting the strongest values as required, adhering to the problem's criteria.

Solution Implementation

Python

```python
class Solution:
    def getStrongest(self, arr, k):
        # Sort the array in non-decreasing order to find the median
        arr.sort()

        # Find the median value, using the formula given
        # If(len(arr) - 1) >> 1) is equivalent to (len(arr) - 1) // 2
        # If finds the index of the median after the sort
        median = arr[(len(arr) - 1) // 2]

        # Sort the array again using a custom key
        # The sort is primarily based on the absolute difference from the median,
        # secondary on the values themselves in descending order
        arr.sort(key=lambda x: (-abs(x - median), -x))

        # Return the first k strongest elements
        return arr[:k]
```

Java

```java
class Solution {
    public int[] getStrongest(int[] arr, int k) {
        // Sort the array to find the median
        Arrays.sort(arr);
        // Calculate the median index and value using bitwise right shift for integer division by 2
        int median = arr[(arr.length - 1) >> 1];

        // Create a list to hold the array elements for custom sorting
        List<Integer> numsList = new ArrayList<>();
        for (int value : arr) {
            numsList.add(value);
        }

        // Sort the list by the strongest criteria
        numsList.sort((a, b) -> {
            // Calculate the absolute difference from the median for both elements
            int diffA = Math.abs(a - median);
            int diffB = Math.abs(b - median);

            // If both elements have the same strength, then sort by descending natural order
            if (diffA == diffB) {
                return b - a;
            } else {
                // Otherwise, sort by the strength, which is the absolute difference from the median
                return diffB - diffA;
            }
        });

        // Create an array to store the k strongest elements
        int[] strongest = new int[k];
        // Copy the first k strongest elements from the sorted list to the array
        for (int i = 0; i < k; i++) {
            strongest[i] = numsList.get(i);
        }

        // Return the array containing the k strongest elements
        return strongest;
    }
}
```

C++

```cpp

#include <vector>    // Include header for vector
#include <algorithm> // Include header for sort algorithm
using namespace std;

class Solution {
public:
    vector<int> getStrongest(vector<int>& arr, int k) {
        // Step 1: Sort the array in nondecreasing order.
        sort(arr.begin(), arr.end());

        // Step 2: Find the median element 'medianOfArray'.
        // Note: The formula used is (arr.size() - 1) / 2) which gives the median index
        // for both even and odd length arrays since array index is 0-based.
        int medianOfArray = arr[(arr.size() - 1) / 2];

        // Step 3: Sort the array based on the strength criteria using a custom comparator lambda.
        // The strength of an element x is defined as the absolute difference |x - medianOfArray|.
        // If two elements have the same strength, the larger element is considered stronger.
        sort(arr.begin(), arr.end(), [&](int a, int b) {
            int strengthA = abs(a - medianOfArray);
            int strengthB = abs(b - medianOfArray);
            // First compare the strengths. If equal, compare the actual values.
            return strengthA == strengthB ? a > b : strengthA > strengthB;
        });

        // Step 4: Select the first 'k' elements of the sorted array as the result.
        // This uses a range constructor of vector that takes two iterators.
        vector<int> strongestElements(arr.begin(), arr.begin() + k);

        // Step 5: Return the result vector containing the 'k' strongest elements.
        return strongestElements;
    }
};
```

TypeScript

```typescript

// Import the array and utility modules for sorting and working with arrays
import * as arraySorts from 'array';     // Assuming 'array' is a module you've created or imported
import * as utils from 'utility';        // Assuming 'utility' is a module you've created or imported

// Define getStrongest method which finds the strongest 'k' elements in the array
function getStrongest(arr: number[], k: number): number[] {
    // Step 1: Sort the array in nondecreasing order.
    arr.sort((a, b) => a - b);

    // Step 2: Find the median element 'medianOfArray'.
    // Since array indices start at 0, ((length of array - 1) / 2) gives the median index for both even and odd length arrays.
    const medianIndex: number = Math.floor((arr.length - 1) / 2);
    const medianOfArray: number = arr[medianIndex];

    // Step 3: Sort the array based on the strength criteria using a custom comparator.
    // The strength of an element 'x' is defined as the absolute difference |x - medianOfArray|.
    // If two elements have the same strength, the larger element is considered stronger.
    arr.sort((a, b) => {
        const strengthA: number = Math.abs(a - medianOfArray);
        const strengthB: number = Math.abs(b - medianOfArray);
        // First compare the strengths. If equal, compare the actual values.
        if (strengthA !== strengthB) {
            return strengthB - strengthA;
        }
        return b - a;
    });

    // Step 4: Select the first 'k' elements of the sorted array to be the result.
    return arr.slice(0, k);
}

// Example usage:
// let strongest = getStrongest([1,2,3,4,5], 2);
// console.log(strongest);  // This will output the 2 strongest elements as per the defined criteria


class Solution:
    def getStrongest(self, arr, k):
        # Sort the array in non-decreasing order to find the median
        arr.sort()

        # Find the median value, using the formula given
        # If(len(arr) - 1) >> 1) is equivalent to (len(arr) - 1) // 2
        # It finds the index of the median after the sort
        median = arr[(len(arr) - 1) // 2]

        # Sort the array again using a custom key
        # The sort is primarily based on the absolute difference from the median,
        # secondary on the values themselves in descending order
        arr.sort(key=lambda x: (-abs(x - median), -x))

        # Return the first k strongest elements
        return arr[:k]
```

Time and Space Complexity

The given code includes sorting an array and then sorting it again with a custom key. Here's the detailed complexity analysis:

Time Complexity

1. The first sort function call sorts the array `arr` using the default sorting algorithm in Python, Timsort, which has a time complexity of `O(n log n)`, where `n` is the length of the array.
2. The calculation of the median value `m` involves a simple arithmetic operation, which has time complexity `O(1)`.
3. The second sort uses a lambda function as the key, which also sorts the array `arr`, but this time it's based on a comparison involving absolute value differences and then the values themselves. This sorting step also has a time complexity of `O(n log n)`. However, the constant factors may be larger because the key function is more complex than the default comparison.

Since there are two consecutive sorting steps, but each is `O(n log n)`, the overall time complexity of these sorting steps doesn't change and remains `O(n log n)`.

4. The slicing operation `arr[:k]` has a time complexity of `O(k)` since it creates a new list of `k` elements from the sorted `arr`.

Combining these steps, the overall time complexity is determined by the most expensive operation, which is sorting twice. Therefore, the total time complexity is `O(n log n)`.

Space Complexity

1. The sorting operation of the array, by default, is done in-place in Python, so it doesn't consume additional space proportional to the input size. However, the sorted() function, which is used internally, creates a new list, so the space complexity is `O(n)` to accommodate this.
2. The space complexity for calculating the median is `O(1)` since it's just storing a single value.
3. The second sorting operation does not use additional space besides what's already accounted for by the input array and the temporary space used by the sorting algorithm.
4. The slicing operation creates a new array of size `k`. So the space complexity of this part is `O(k)`.

Since the largest space used is the space for the original array and the space for the output array, the overall space complexity is `O(n)` when considering `k` to be less than or equal to `n`. If we consider `k` separate contextually as it could be, then space complexity could also be denoted as `O(n + k)` but typically `k` is considered subordinate to `n` if we do not have information that `k` is of the same order as `n` or larger.