1502. Can Make Arithmetic Progression From Sequence

Easy Array Sorting

Leetcode Link

Problem Description

The problem gives us an array of numbers named `arr` and asks us to determine whether we can rearrange the array's elements to form an arithmetic progression. An arithmetic progression is a sequence of numbers where the difference between any two consecutive numbers is always the same. For example, in the sequence [1, 3, 5, 7], the difference between consecutive elements is 2, which is consistent throughout the sequence, so it is an arithmetic progression. The goal is to return `true` if the given array can be rearranged to form such a sequence, or `false` otherwise.

Intuition

The intuition behind the solution is to first sort the array. Sorting the array is a key step because if an arithmetic progression exists, it must be the case that when sorted, the difference between each pair of consecutive elements is consistent. Once the array is sorted, we calculate the common difference, which should be the difference between the first two elements (since the sorted array should now represent an arithmetic progression if one is possible).

After calculating this common difference, we iterate through the array to check if every consecutive pair of elements has this same difference. If at any point we find a pair of elements that do not have the common difference, we can conclude that it is not possible to form an arithmetic progression and should return `false`. If we successfully iterate through the entire array without finding discrepancies, then the array can indeed form an arithmetic progression and we return `true`.

The `pairwise` function utilized in the solution is a Python utility that allows us to iterate through the array in element pairs which simplifies the process of checking the difference between consecutive elements. With `all`, we check if all pairs have the same difference `d`, and hence it satisfies the condition of an arithmetic progression.

Solution Approach

The solution approach leverages a sorting algorithm and a simple iteration pattern to verify the arithmetic progression. Here's a step-by-step breakdown of the implementation applied in the provided Python code:

1. **Sorting the Array**: The first line in the function sorts the array in-place using `arr.sort()`, which is a built-in Python method that sorts the list ascendingly by default. The sorted array is necessary to easily compare the difference between consecutive elements.

2. **Finding the Common Difference**: The variable `d` is computed as `arr[1] - arr[0]` i.e., the difference between the first two elements of the sorted array. This difference `d` is what we expect between every pair of consecutive elements in an arithmetic progression.

3. **Verifying the Arithmetic Progression**: The last step is to verify if each consecutive pair of elements in the sorted array has the same difference `d`. This is done using the expression `all(b - a == d for a, b in pairwise(arr))`.

4. The `pairwise` function is from Python's itertools module (potentially, the `grouper` pattern could also be used), which is used here to iterate over the array elements in pairs. Each pair of elements `(a, b)` consists of consecutive elements from the sorted array.

5. The `all` function ensures that every element of the provided iterator evaluates to `True`. It processes the generator expression, which for each pair of elements checks if the difference `b - a` is equal to `d`.

6. If any pair does not satisfy this, `all` will immediately return `False`, indicating that the progression cannot be formed. Otherwise, it will return `True`, confirming that the array is indeed an arithmetic progression.

The elegance of this approach lies in its simplicity and the efficient use of Python's standard libraries to achieve the desired outcome with very few lines of code. Since sorting is the most computationally expensive part of the algorithm, the overall time complexity is `O(n log n)` where `n` is the number of elements in the array due to the sorting operation. The verification process has a time complexity of `O(n)` since it iterates through the sorted elements once. Therefore, the total time complexity remains `O(n log n)`.

Example Walkthrough

Let's take an example to illustrate the solution approach. Consider the array `arr = [9, 5, 1, 3, 7]`. Our goal is to check whether we can rearrange this array into an arithmetic progression.

Following the solution approach:

1. **Sorting the Array**: We first sort the array, which results in `arr = [1, 3, 5, 7, 9]`.

2. **Finding the Common Difference**: We calculate the common difference `d` as the difference between the first two elements, which is `d = arr[1] - arr[0] = 3 - 1 = 2`.

3. **Verifying the Arithmetic Progression**: We then check if the difference between every consecutive pair of elements is equal to `d`.
   - The difference between the second and the third elements (3 and 5) is `5 - 3 = 2`, which is equal to `d`.
   - The difference between the third and the fourth elements (5 and 7) is `7 - 5 = 2`, which is again equal to `d`.
   - The difference between the fourth and the fifth elements (7 and 9) is `9 - 7 = 2`, which is also equal to `d`.

Since all pairs have the same difference, which is equal to `d`, the function `all(b - a == d for a, b in pairwise(arr))` would return `True`.

Therefore, based on the solution approach, we can conclude that it is indeed possible to rearrange the array `[9, 5, 1, 3, 7]` to form an arithmetic progression `[1, 3, 5, 7, 9]`.

The consistent common difference and the successful run of the `all` function on pairwise compared elements support the conclusion that `arr` can be rearranged to form an arithmetic progression, and thus, the final answer is `true`.

Python Solution

```python
from typing import List

class Solution:
    def canMakeArithmeticProgression(self, arr: List[int]) -> bool:
        # Sort the array in non-decreasing order
        arr.sort()

        # Calculate the common difference 'd' between the first two elements
        common_difference = arr[1] - arr[0]

        # Iterate over the sorted array to check if each pair of successive
        # elements has the same difference 'd'
        for i in range(1, len(arr) - 1):
            # If the difference between the current and next element does not equal
            # the common difference 'd', the sequence is not an arithmetic progression
            if arr[i + 1] - arr[i] != common_difference:
                return False

        # If all differences are equal, return True (it is an arithmetic progression)
        return True

```

Java Solution

```java
class Solution {

    // Function to check if it is possible to form an arithmetic progression
    public boolean canMakeArithmeticProgression(int[] arr) {
        // Sort the array in non-decreasing order
        Arrays.sort(arr);

        // Find the common difference 'difference' by subtracting the second element by the first element
        int difference = arr[1] - arr[0];

        // Iterate through the sorted array starting from the third element
        for (int i = 2; i < arr.length; ++i) {
            // Check if the current difference is equal to the common difference 'difference'
            // If not, return false since it can't form an arithmetic progression
            if (arr[i] - arr[i - 1] != difference) {
                return false;
            }
        }

        // If the loop completes without returning false, it means the array can form an arithmetic progression
        return true;
    }
}

```

C++ Solution

```cpp
#include <vector>   // Include the vector header
#include <algorithm> // Include the algorithm header for sort
using namespace std; // Using the standard namespace

class Solution {
public:
    bool canMakeArithmeticProgression(vector<int>& arr) {
        // Sort the array in non-decreasing order
        sort(arr.begin(), arr.end());

        // Calculate the common difference 'd' of the first two elements
        int commonDifference = arr[1] - arr[0];

        // Iterate through the array starting from the third element
        for (size_t i = 2; i < arr.size(); i++) {
            // If the difference between the current and the previous element
            // is not equal to the common difference 'd', return false
            if (arr[i] - arr[i - 1] != commonDifference) {
                return false;
            }
        }

        // If all consecutive elements have the same difference, return true
        return true;
    }
};

```

Typescript Solution

```typescript
// Function to check if the provided array can form an arithmetic progression
function canMakeArithmeticProgression(arr: number[]): boolean {
    // Sort the input array in ascending order
    arr.sort((a, b) => a - b);
    const lengthOfArray = arr.length; // Store the length of the array

    // Loop through the array starting from the third element
    for (let i = 2; i < lengthOfArray; i++) {
        // Calculate the differences between the consecutive elements
        const firstDifference = arr[i - 1] - arr[i - 2];
        const secondDifference = arr[i] - arr[i - 1];

        // If the consecutive differences are not equal, return false
        if (firstDifference !== secondDifference) {
            return false;
        }
    }

    // If all consecutive differences are equal, return true
    return true;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `canMakeArithmeticProgression` function is determined primarily by the sorting operation. In Python, the default sort function uses an algorithm called Timsort, which has a time complexity of `O(n log n)`, where `n` is the length of the array.

The subsequent operation consists of iterating through the sorted array to check if every pair of successive elements has the same difference, `d`. This check uses a generator expression with the `all()` function combined with pairwise iteration, which is `O(n-1)` or simply `O(n)` since it's walking through the array only once.

Thus, the dominating factor here is the sorting, and the overall time complexity of the function is `O(n log n)`.

Space Complexity

For space complexity, the sort operation can be done in-place, but certain implementations may require additional space. Python's Timsort requires `O(n)` space in the worst case.

The pairwise iteration does create pairs for every two adjacent elements in the array, but since this is done by the generator expression, it doesn't create an additional list in memory, it simply iterates through the existing sorted array yielding one element at a time.

Hence, the space complexity of the function, which is mostly governed by the sorting operation's space requirement, is `O(n)`.