1630. Arithmetic Subarrays Medium Array Sorting **Leetcode Link**

Problem Description

arithmetic sequence is defined as a sequence where the difference between consecutive elements is constant. This difference is called the common difference and is calculated as s[i+1] - s[i]. Given:

The problem requires us to determine whether segments of an array can be rearranged to form an arithmetic sequence. An

array.

We need to:

An array nums with n integers.

Two arrays l and r each with m integers, representing m range queries. Each query defines a range [l[i], r[i]] on the nums

- Rearrange each subarray defined by the range queries to see if they can form an arithmetic sequence.
- Return a boolean array, where each element corresponds to a range query, indicating whether the subarray can be rearranged to form an arithmetic progression (true) or not (false).

Intuition

sequence must have equal intervals (common differences) between the terms when placed in ascending or descending order.

To check if a list of numbers can be rearranged into an arithmetic sequence, a fundamental property can be utilized: an arithmetic

4. Check two conditions:

Here's how we can approach it:

2. Find the minimum (a1) and maximum (an) elements in the subarray. 3. Calculate the common difference d that should exist if the subarray can be formed into an arithmetic sequence. It is found by dividing the range (an - a1) by the number of intervals (n - 1), where n is the length of the subarray.

Whether the computed common difference leaves no remainder, which means it's an exact interval for an arithmetic

sequence. Whether all expected terms of the arithmetic sequence exist in the subarray. This is done by checking for the presence of each term a1 + (i - 1) * d for i ranging from 1 to n, within the set of numbers in the subarray.

1. For each query, identify the subarray from the original array nums using the given range [l[i], r[i]].

- 5. If both conditions are true, the subarray can be rearranged into an arithmetic sequence. Otherwise, it cannot. The solution code takes these steps and applies them to all queries, returning a list of boolean values as the answer.
- **Solution Approach**

The solution implemented in the given Python function uses a helper function named check to assess each query range for the

possibility to form an arithmetic sequence. The solution leverages mathematical reasoning to efficiently determine whether a range of numbers in an array could be rearranged into an arithmetic sequence.

Here's a breakdown of the approach:

responsible for determining whether the subarray can be rearranged into an arithmetic sequence.

testing.

(nums[l:l+n]) and immediately converted into a set s.

perform the division and check for a remainder (mod).

theoretical arithmetic series (a1 + (i - 1) * d) exists in the set s.

check for this and other queries continues similarly using the described approach.

Let's consider small input arrays to illustrate the solution approach described above:

3. Finding Minimum and Maximum: Python's min and max functions are used to find the smallest and largest elements (a1 and an, respectively) in the subarray. These are critical for calculating the expected arithmetic sequence's common difference.

4. Calculating Common Difference: The common difference d is calculated by dividing the difference between the maximum and

minimum elements (an -a1) by one less than the length of the subarray (n - 1). The divmod function is used to simultaneously

2. Subarray Extraction and Set Conversion: For each query, the relevant subarray is carved out using Python's slicing feature

1. Helper Function: A nested function called check is defined within the checkArithmeticSubarrays method. The check function is

- 5. Arithmetic Sequence Verification: Two checks are performed to verify that an arithmetic sequence can be formed: Zero Remainder Check: The remainder mod from the division must be zero to ensure that the common difference is an integer which is essential for a valid arithmetic sequence.
- If both checks pass, the subarray represented by the current query can indeed be rearranged into an arithmetic sequence. This dual check approach makes efficient use of Python's set operations, which offer average constant time complexity for membership

Sequence Formation Check: A comprehension loop runs for each position i from 1 to n, checking whether each term of the

The algorithm has linear complexity with respect to the number of elements in each query [0(n)], but since it needs to be executed for m queries, the overall complexity is 0(m * n). The usage of sets and minimal iteration ensures that the solution is optimized within these boundaries.

6. Result Compilation: Finally, a list comprehension is utilized to apply the check function to each range [left, right] derived by

zipping the lists 1 and r together. The function returns a list of boolean results that correspond to each range query.

• r = [2, 3, 5]For the first query range [0, 2], we look at subarray [4, 6, 5]. The minimum is 4, the maximum is 6, and the common difference d should be 1 with no remainder. All numbers between 4 and 6 are present and we can confirm it can form an arithmetic sequence. The

Following the approach:

• l = [1, 2, 0]

• r = [3, 4, 2]

Example Walkthrough

• nums = [3, 1, 4, 1, 5]

Here is an illustrative example:

• l = [0, 0, 2]

• nums = [4, 6, 5, 9, 3, 7]

```
common difference to the minimum: 1 + 2 = 3 and 3 + 2 = 5. Since we do not find 3, we can say the original subarray cannot
form an arithmetic sequence. The answer is false.
```

Python Solution

class Solution:

15

16

17

18

19

20

21

22

23

28

9

1 from typing import List

without further checks. The answer is false.

def checkArithmeticSubarrays(self, nums: List[int], l: List[int], r: List[int]) -> List[bool]:

Check if every number in the supposed arithmetic series is in the set.

Go over each query of the arrays and check if it forms an arithmetic array.

return [is_arithmetic(nums, left, right) for left, right in zip(l, r)]

27 # print(result) # Output: [True] since the subarray from index 0 to index 3 is arithmetic

// Check if each subarray is an arithmetic array and return a list of boolean values

public List<Boolean> checkArithmeticSubarrays(int[] nums, int[] l, int[] r) {

// Check each subarray and add the result to the answer list

return all((smallest + i * common_diff) in subarray_set for i in range(subarray_length))

but conceptually, it can be instructive to think of the sequences in sorted order.

Helper function to check if the subarray is an arithmetic array.

def is_arithmetic(nums: List[int], left: int, right: int) -> bool:

subarray_length = right - left + 1

26 # result = sol.checkArithmeticSubarrays(nums=[3,6,9,12], l=[0], r=[3])

List<Boolean> result = new ArrayList<>();

// Iterate over all the given subarrays

for (int i = 0; i < l.length; ++i) {</pre>

if remainder != 0:

return False

3. Query 3: The range [0, 2] gives us [3, 1, 4]. We sort to get [1, 3, 4]. Here, the minimum is 1, the maximum is 4, and d should be (4 - 1) / (3 - 1) = 3 / 2, which again is not an integer. So, the subarray cannot form an arithmetic sequence. The answer is false. The result of running our algorithm on these queries would yield [false, false, false] as none of the subarrays can be rearranged

into an arithmetic sequence. Note that in an actual implementation, we may use sets directly and bypass the explicit sorting step,

1. Query 1: The first range [1, 3] corresponds to the subarray [1, 4, 1]. We sort it to [1, 1, 4] (although for the check we use a

set, but to understand let's consider a sorted list). The minimum is 1 and the maximum is 4. The common difference d should be

(4 - 1) / (3 - 1) = 3 / 2. Since 3 / 2 is not an integer, we instantly know this subarray cannot form an arithmetic sequence

2. Query 2: For the range [2, 4], we extract the subarray [4, 1, 5]. Sorting, we get [1, 4, 5]. The minimum is 1, the maximum is

5, and the common difference d should be (5 - 1) / (3 - 1) = 4 / 2 = 2. We check if each interval is present by adding the

subarray_set = set(nums[left : left + subarray_length]) smallest, largest = min(subarray_set), max(subarray_set) 9 # Calculate common difference and check if it is an integer. 11 common_diff, remainder = divmod(largest - smallest, subarray_length - 1) 12 13 # If the remainder is not zero, it's impossible to have uniform steps. 14

```
Java Solution
```

class Solution {

24 # Example usage:

25 # sol = Solution()

```
result.add(isArithmeticArray(nums, l[i], r[i]));
10
11
12
            return result;
13
14
15
       // Helper function to check if a subarray is an arithmetic array
       private boolean isArithmeticArray(int[] nums, int left, int right) {
16
            Set<Integer> set = new HashSet<>();
17
            int subArraySize = right - left + 1;
18
19
            int minValue = Integer.MAX_VALUE;
20
            int maxValue = Integer.MIN_VALUE;
21
22
           // Populate the set with values from the subarray and find min and max
            for (int i = left; i <= right; ++i) {</pre>
24
                set.add(nums[i]);
25
                minValue = Math.min(minValue, nums[i]);
26
                maxValue = Math.max(maxValue, nums[i]);
27
28
29
           // Check if the difference between max and min is perfectly divisible by the size - 1
30
           // This condition helps to determine if we can have an equal difference 'd'
31
            if ((maxValue - minValue) % (subArraySize - 1) != 0) {
32
                return false;
33
34
35
           // Calculate common difference 'd'
36
            int commonDifference = (maxValue - minValue) / (subArraySize - 1);
37
38
            // Check if every element that should be present in an arithmetic array is in the set
39
            for (int i = 1; i < subArraySize; ++i) {</pre>
                if (!set.contains(minValue + (i - 1) * commonDifference)) {
40
41
                    return false;
42
43
            return true; // The subarray is arithmetic
45
46 }
47
```

31 32 33 34

C++ Solution

#include <vector>

2 #include <unordered_set>

#include <algorithm>

using namespace std;

// Method to check if subarrays are arithmetic sequences

for (int i = left; i <= right; ++i) {</pre>

elements.insert(nums[i]);

int minElement = INT_MAX, maxElement = INT_MIN;

minElement = min(minElement, nums[i]);

maxElement = max(maxElement, nums[i]);

// Find the min and max elements within the subarray

// An arithmetic sequence should have a common difference 'd',

int size = right - left + 1;

vector<bool> checkArithmeticSubarrays(vector<int>& nums, vector<int>& l, vector<int>& r) {

// Internal lambda function to check if a single subarray is an arithmetic sequence

vector<bool> results; // This will store the boolean results for each query

auto isArithmetic = [](const vector<int>& nums, int left, int right) -> bool {

unordered_set<int> elements; // To store unique elements for checking

class Solution {

public:

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

```
27
                 // and satisfy (maxElement - minElement) % (size - 1) == 0
                 if ((maxElement - minElement) % (size - 1)) {
 28
                     return false;
 29
                 // Calculate the common difference between consecutive elements
                 int commonDifference = (maxElement - minElement) / (size - 1);
 35
                 // Verify each element of the theoretical arithmetic sequence
 36
                 for (int i = 1; i < size; ++i) {
                     if (!elements.count(minElement + (i - 1) * commonDifference)) {
 37
 38
                         return false;
 39
 40
 41
 42
                 return true;
 43
             };
 44
 45
             // Iterate over each range query (l and r vectors) to check each subarray
             for (size_t i = 0; i < l.size(); ++i) {
 46
                 results.push_back(isArithmetic(nums, l[i], r[i])); // Store the result for each subarray
 47
 48
 49
 50
             return results; // Return the results for all queries
 51
 52 };
 53
Typescript Solution
     function checkArithmeticSubarrays(nums: number[], leftIndices: number[], rightIndices: number[]): boolean[] {
        // Helper function that checks if the subarray is an arithmetic sequence
        const isArithmeticSequence = (nums: number[], leftIndex: number, rightIndex: number): boolean => {
  3
            const uniqueNumbers = new Set<number>(); // Will store unique numbers in the current subarray
            const subarrayLength = rightIndex - leftIndex + 1;
            let minElement = Number.MAX_SAFE_INTEGER; // Initialize to max possible value
            let maxElement = Number.MIN_SAFE_INTEGER; // Initialize to min possible value
  8
            // Find the minimum and maximum value in the subarray, while also adding to the set
  9
            for (let i = leftIndex; i <= rightIndex; ++i) {</pre>
 10
                uniqueNumbers.add(nums[i]);
 11
                minElement = Math.min(minElement, nums[i]);
 12
 13
                maxElement = Math.max(maxElement, nums[i]);
 14
 15
 16
            // Check for an edge case where elements are not distinct or cannot form an arithmetic sequence
            if ((maxElement - minElement) % (subarrayLength - 1) !== 0) {
 17
 18
                return false;
 19
 20
 21
            // Calculate the common difference 'd' for the potential arithmetic sequence
 22
            const commonDifference = (maxElement - minElement) / (subarrayLength - 1);
 23
            // Check if each expected element of the arithmetic sequence is present in the set
 24
 25
            for (let i = 1; i < subarrayLength; ++i) {</pre>
                if (!uniqueNumbers.has(minElement + (i - 1) * commonDifference)) {
 26
 27
                    return false;
 28
 29
 30
```

results.push(isArithmeticSequence(nums, leftIndices[i], rightIndices[i])); 40 41 42 43 return results; // Return the array of boolean results 44 } 45

Time Complexity

Space Complexity

return true;

31

32

33

34

35

36

37

38

39

};

For a single query (on one 1 to r pair), the steps include: 1. Slicing the subarray, which takes O(k), where k is the length of the subarray (from 1 to r).

Time and Space Complexity

Since we perform the above steps for each subarray defined by pairs of 1 and r, if m is the number of queries (the length of lists 1 and r), the overall time complexity of the checkArithmeticSubarrays method is 0(m * k) where each k can vary for different queries

3. Computing the minimum and maximum values of the subarray, which take 0(k) each.

than O(k) for checking if all elements follow the arithmetic sequence.

// If all elements are present, it's a valid arithmetic sequence

// Iterate over each query defined by leftIndices and rightIndices

for (let i = 0; i < leftIndices.length; ++i) {</pre>

const results: boolean[] = []; // Array to store results of the check for each query

// Push the result of check for arithmetic sequence in the subarray to results

- but is at most n (the total number of elements in nums). In the worst case where each query spans the whole array, the time complexity becomes 0(m * n).
- The space complexity of the provided code includes the space required for the output list and the temporary set used in the check function.

The time complexity of the check function involves iterating over a subarray of the input nums list and performing operations such as

2. Creating a set from the subarray, which takes O(k) in both time for converting a list to a set and potentially less but not better

4. The final arithmetic sequence check, which iterates over a range of size k and could take up to 0(k) in the worst case.

finding the minimum and maximum within this subarray and checking the arithmetic property on each element of the set.

number of such queries, corresponding to the number of elements in 1 and r. 2. The space required for the set s inside the check function which also has a complexity of O(k), where k is the size of the subarray. This set is created for each query and does not grow beyond the size of the largest subarray.

1. The output list that accumulates the boolean results for each query in zip(l, r) has a space complexity of 0(m), where m is the

Since sets and the output list do not accumulate across queries but rather are allocated per query (the set is recreated for each query, and the output list is just accumulated once per query), the overall space complexity remains 0(n) due to the set potentially growing to store pointers to up to n distinct values in the case of a single subarray spanning the entire array. This assumes that the space taken by the set is the dominant term and that k can reach n for at least one query.