2170. Minimum Operations to Make the Array Alternating Medium Greedy Hash Table Counting Array

Problem Description

1. Every element at an even index (except the first) must be equal to the element two places before it. Mathematically, this is expressed as nums[i - 2] = nums[i] for i ranging from 2 to n - 1.

particular characteristic to be called "alternating". For an array to be considered alternating, it must satisfy two conditions:

You are provided with an array nums which contains n positive integers, and it is indexed starting from 0. These types of arrays have a

Leetcode Link

- 2. Every element at an odd index must be different from the element immediately preceding it. This is expressed as nums [i 1] != nums [i] for i ranging from 1 to n-1.
- The goal is to calculate the minimum number of changes needed to be made to the array to achieve the "alternating" property. Each
- operation allows you to choose an index i and change the element nums [i] to any positive integer of your choosing. Intuition

To minimize the number of operations needed to make the array alternating, we should make the most frequent elements in even and odd positions stay the same as much as possible since changing them would require more operations.

The solution involves the following steps: 1. Separate the numbers in the array into two groups based on their indices: one consisting of the numbers at even indices and the

other consisting of those at odd indices. 2. Count the frequency of each number in both even and odd indexed groups. Find the most common elements (we might need the

two most common elements from each group in case the most common elements in both groups are the same).

elements might be the same.

3. Calculate the minimum operations needed by considering the most common elements. If the most common element in the evenindexed group is different from the most common in the odd-indexed group, we'll keep those and change the rest. If they are the

same, we have to look at the second most common elements to decide which will require fewer changes.

and odd indices. The provided code determines the most common elements and their frequencies for even and odd positions separately and then

uses a generator to compute the minimum necessary changes, considering the cases where the most common even and odd

4. The minimum number of operations is the total length of the array minus the sum of the counts of the chosen elements for even

Solution Approach

In the provided solution, several key programming concepts and data structures are utilized: 1. Counter from collections module: The Counter class in Python's collections module is used to count the frequency of each

element in both the even-indexed and odd-indexed subarrays of nums. By calling Counter(nums[i::2]).most_common(2), we get

the top two most common elements and their respective counts for the subarray starting at index i with a step of 2 (meaning

2. List slicing: The slice operation nums [1::2] generates a new list containing every second element of nums starting from index 1.

This is used to separate the original array into the aforementioned even-indexed and odd-indexed subarrays.

3. Tuple unpacking and conditional expression: Tuples are used to represent the elements and their counts returned by most_common(). The conditional expression is used to return a default count of zero when there is only one common element

from the Counter.

Example Walkthrough

1 nums = [1, 2, 2, 3, 1, 5]

Step 2: Count the frequency

every other element starting from i).

found by most_common().

During the walkthrough, we'll use the steps detailed in the solution approach.

Odd-indexed group: [2, 3, 5] (elements at indices 1, 3, 5)

Even-indexed group: 1 appears 2 times, 2 appears 1 time

Count the frequency of elements in each group:

Odd-indexed group: 2, 3, and 5 each appear 1 time

Step 3: Find the most common elements

Step 4: Calculate the minimum operations

Count of 1's in even-indexed group = 2

Calculate the minimum changes needed:

for a, n1 in get(0) for b, n2 in get(1) if a != b). If the element for the even position (a) is the same as for the odd position (b), that particular combination is ignored (if a != b). Otherwise, n1 and n2 are the frequency counts of the chosen elements, and n - (n1 + n2) computes the number of changes needed. 5. Function definition: The get function is defined to encapsulate the functionality of getting the two most common elements from

either the even-indexed or odd-indexed numbers. It handles edge cases where there might be less than two elements returned

4. List comprehensions and generator expressions: The min() function is used with a generator expression that iterates over all

possible combinations of elements and their counts for even and odd positions. The generator expression is min(n - (n1 + n2)

By combining these tools and methods, the solution efficiently calculates the minimum number of operations needed to make the input array alternating. The use of Counter and most_common methods has a time complexity of O(N), where N is the length of the array. Since we iterate over at most four combinations (two from each list split), the total complexity remains O(N).

Let's consider a small example to illustrate the solution approach described above. Suppose we are given the following array nums:

Step 1: Separation into even and odd indexed groups Separate nums into two groups based on indices: Even-indexed group: [1, 2, 1] (elements at indices 0, 2, 4)

Identify the most common elements: Even-indexed most common element: 1, with frequency 2

Since 2 is also present in even-indexed group but is not the most frequent, we consider it for the odd group. No conflict yet.

Odd-indexed most common element: Each element has the same frequency 1, but let's say we choose 2

If we keep element 1 in the even positions, we need to change one element (2 at index 2).

Count of 2's in odd-indexed group = 1 (since we only need to consider odd-indexed positions)

For the odd positions, if we choose to keep 2, we don't need to make any changes.

Summarizing the changes:

Total array length = 6

Python Solution

class Solution:

6

10

11

12

13

14

15

16

22

23

24

25

26

27

28

29

= 3 changes

Minimum number of operations = Total length - (Even count + Odd count) = 6 - (2 + 1)

element of the even group and two elements of the odd group that are not 2.

from collections import Counter # Importing Counter from collections module

If there are no elements, return two pairs of zeros

Count the elements at alternating indices starting from idx

If there is only one element, return it with a pair of zeros

Otherwise return the two most common elements and their counts

n - (count_even + count_odd) # Calculate the minimum number

if value_even != value_odd # Only consider pairs with different values

// Generate frequency arrays for even and odd indices, and compare them

// Calculate and update the minimum operations needed

// Ensure that we consider different values (from even and odd indices)

// Loop through the numbers array, incrementing by 2 from the specified start index

// Iterate over the frequency map to find the most and second most frequent numbers

def minimumOperations(self, nums: List[int]) -> int:

counts = Counter(nums[idx::2]).most_common(2)

def get_most_common_alternate(idx):

return [(0, 0), (0, 0)]

return [counts[0], (0, 0)]

alternating indices that are not the same.

private int[] numbers; // An array of numbers

private int[][] getFrequency(int start) {

int key = entry.getKey();

int value = entry.getValue();

private int arrayLength; // Length of the numbers array

for (int[] evenFreqElement : getFrequency(0)) {

for (int i = start; i < arrayLength; i += 2) {</pre>

int topCount = 0; // Count of the most frequent number

int secondValue = 0; // Second most frequent number

int topValue = 0; // Most frequent number

for (int[] oddFreqElement : getFrequency(1)) {

if (evenFreqElement[0] != oddFreqElement[0]) {

return minOperations; // return the minimum operations calculated

int secondCount = 0; // Count of the second most frequent number

for (Map.Entry<Integer, Integer> entry : frequencyMap.entrySet()) {

if not counts:

return counts

return min(

NameError. Here's how you can import it:

from typing import List

if len(counts) == 1:

In a scenario where the most frequent even and odd elements were the same, we would require looking at the second most common elements, but in our example, this isn't the case.

Internal function to get the two most common elements and their counts in alternating indices.

By applying the solution approach, we understand that to make nums alternating, we need to make 3 changes: replace the second

17 18 # This stores the length of the nums list 19 n = len(nums)20 # The result is the minimum count of operations which is obtained by taking the 21 # total count 'n' minus the sum of counts of the two most common elements at

for value_even, count_even in get_most_common_alternate(0) # Get most common at even indices

Please note that List must be imported from typing if it hasn't been defined earlier in the code. Otherwise, Python will raise a

int minOperations = Integer.MAX_VALUE; // Initialize minimum operations with the maximum integer value

// Generates a 2D array containing the two most frequent elements and their counts at either even or odd indices

frequencyMap.put(numbers[i], frequencyMap.getOrDefault(numbers[i], 0) + 1); // Update frequency map

// If current frequency is greater than the most frequent, update top and second top frequencies and values

Map<Integer, Integer> frequencyMap = new HashMap<>(); // Map to keep track of frequency of elements

minOperations = Math.min(minOperations, arrayLength - (evenFreqElement[1] + oddFreqElement[1]));

for value_odd, count_odd in get_most_common_alternate(1) # Get most common at odd indices

```
5
      // Calculates the minimum number of operations required
      public int minimumOperations(int[] nums) {
          this.numbers = nums; // Assigns passed array to the numbers property
          arrayLength = nums.length; // Assigns array length to arrayLength property
8
```

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

60

6

10

11

12

13

14

15

16

17

19

22

23

24

25

26

27

28

29

31

33

34

35

36

46

48

49

50

52

51 }

*/

59 };

Typescript Solution

* @param nums The input array of numbers

const length = nums.length;

// Populate the frequency arrays

const current = nums[i];

if (i % 2 === 1) {

} else {

for (let i = 0; i < length; i++) {

if (oddIndexMax !== evenIndexMax) {

return length - Math.max(

Time and Space Complexity

for both halves of the list).

Space Complexity

Counters.

for odd indices, this results in a complexity of O(n).

oddFrequencies[current]++;

evenFrequencies[current]++;

const maxValue = 10 ** 5;

* @returns The minimum number of operations required

let oddFrequencies = new Array(maxValue).fill(0);

let evenFrequencies = new Array(maxValue).fill(0);

// Find the max frequency element for odd and even indices

let oddMaxFreq = oddFrequencies[oddIndexMax];

let evenMaxFreq = evenFrequencies[evenIndexMax];

let oddIndexMax = oddFrequencies.indexOf(Math.max(...oddFrequencies));

// Handle the case when the max frequency elements are the same

oddMaxFreq + evenFrequencies[evenSecondMaxIndex],

evenMaxFreq + oddFrequencies[oddSecondMaxIndex]

let evenIndexMax = evenFrequencies.indexOf(Math.max(...evenFrequencies));

// If the max frequency elements are different, the answer is straightforward

return length - oddFrequencies[oddIndexMax] - evenFrequencies[evenIndexMax];

let evenSecondMaxIndex = evenFrequencies.indexOf(Math.max(...evenFrequencies));

// The answer is the max of either changing the even-indexed or the odd-indexed elements

function minimumOperations(nums: number[]): number {

Java Solution

class Solution {

```
if (value > topCount) {
 41
 42
                     secondValue = topValue;
                     secondCount = topCount;
 43
 44
                     topValue = key;
 45
                     topCount = value;
 46
                 } else if (value > secondCount) { // If current is only greater than second, update only second most frequent
                     secondValue = key;
 47
                     secondCount = value;
 48
 49
 50
 51
             // Return an array of arrays, where each array is of format: {element value, frequency count}
             return new int[][] {{topValue, topCount}, {secondValue, secondCount}};
 52
 53
 54
 55
C++ Solution
  1 #include <vector>
  2 #include <unordered map>
    #include <climits>
    #include <utility>
    using namespace std;
    // Define a pair of integers for easier handling of operations.
    typedef pair<int, int> IntPair;
    class Solution {
     public:
         int minimumOperations(vector<int>& nums) {
 12
 13
             int minOperations = INT_MAX; // Initialize with the maximum possible integer value.
             int size = nums.size(); // Get the size of the input array.
 14
             // Get the frequency pairs of numbers appearing at even indices.
 15
             for (const auto& [numEven, frequencyEven] : getFrequency(0, nums)) {
 17
                 // Get the frequency pairs of numbers appearing at odd indices.
                 for (const auto& [numOdd, frequencyOdd] : getFrequency(1, nums)) {
 18
 19
                     // Ensure we are not considering the same number for both even and odd positions.
                     if (numEven != numOdd) {
 20
                         // Calculate the minimum operations required and update minOperations.
 21
 22
                         minOperations = min(minOperations, size - (frequencyEven + frequencyOdd));
 23
 24
 25
 26
             return minOperations; // Return the calculated minimum operations.
 27
 28
 29
         // Helper method to get top two frequent numbers and their counts from even or odd indices.
 30
         vector<IntPair> getFrequency(int startIndex, vector<int>& nums) -
 31
             unordered_map<int, int> frequencyMap; // Map to hold number frequencies.
 32
 33
             // Build frequency map for numbers at indices determined by startIndex (even or odd).
             for (int i = startIndex; i < nums.size(); i += 2) {</pre>
 34
 35
                 ++frequencyMap[nums[i]];
 36
```

// Variables to keep track of the two most frequent numbers and their counts.

// Update second most frequent number before changing the most frequent one.

int firstNum = 0, firstCount = 0;

if (count > firstCount) {

int secondNum = 0, secondCount = 0;

// Find the two most frequent numbers.

secondNum = firstNum;

firstNum = currentNum;

firstCount = count;

secondCount = count;

secondCount = firstCount;

} else if (count > secondCount) {

secondNum = currentNum;

for (const auto& [currentNum, count] : frequencyMap) {

// Return the top two frequent numbers and their counts.

return {{firstNum, firstCount}, {secondNum, secondCount}};

* Finds the minimum number of operations required to make all the even-indexed elements

* equal and all the odd-indexed elements equal, but not necessarily equal to each other

// Initialize frequency arrays for odd-indexed and even-indexed elements

37 // Zero out the max frequency elements to find the second max 38 oddFrequencies[oddIndexMax] = 0; 39 evenFrequencies[evenIndexMax] = 0; let oddSecondMaxIndex = oddFrequencies.indexOf(Math.max(...oddFrequencies));

);

Time Complexity

} else {

2. Counter(nums[i::2]).most_common(2): This operation finds the two most common elements in the Counter object. The Counter.most_common method has 0(n log k) complexity, where k is the number of most common elements requested. Since k is

The time complexity of the provided code can be analyzed by examining the operations within the minimumOperations function.

1. get Function Call for Half of the Array (Based on Even and Odd Indices): This operation involves creating a Counter on half of the

array, which takes O(n/2), where n is the length of nums. Since the get function is called twice, once for even indices and once

constant (2 in this case), the complexity for this step simplifies to O(n) (for half of the list, but effectively O(n) since called twice

3. The list comprehension iterates through all combinations of two most common elements for even and odd indices. In the worst

case, there are 2 elements from the first half and 2 from the second half, creating 4 combinations to consider. This is a constant

- factor and does not depend on n, so it is O(1). Therefore, the overall time complexity of the provided code is O(n).
- The space complexity can be determined by examining the memory use relative to the input size: 1. The Counter object in the get method stores the frequencies of elements for half of the array, assuming a worst-case scenario

where all items are unique this would be 0(n/2), but since it's called twice, the space complexity is 0(n) in total for both

2. The most common elements are stored as a list of tuples, with up to four tuples stored at any time (most_common(2) for two

halves). This is a constant space 0(1). Therefore, the overall space complexity of the code is O(n).