1674. Minimum Moves to Make Array Complementary

Medium Array Hash Table **Prefix Sum**

You are presented with an integer array nums with an even length n and an integer limit. You're able to perform operations where

Problem Description

you replace any integer in the array with another integer that is within the inclusive range from 1 to limit. The goal is to make the array complementary. An array is complementary if for every index i (0-indexed), the sum nums[i] + nums[n - 1 - i] is the same across the entire array. For instance, the array [1,2,3,4] is complementary because every pair of numbers adds up to 5.

Leetcode Link

Your task is to determine the minimum number of moves required to make the array nums complementary.

To solve this problem efficiently, we need to think in terms of pairs and how changing an element impacts the sums across the array.

use a difference array d to record the number of operations required to achieve each possible sum.

moves.

replaced.

Intuition

1. Since nums has an even length, we only need to consider the first half of the array when paired with their corresponding element from the other end because we want to make nums[i] + nums[n - 1 - i] equal for all i. 2. Each pair (nums[i], nums[n - 1 - i]) can contribute to the sum in three different ways:

We also need to record the minimum operations required to achieve a target sum for each pair at various intervals. The strategy is to

Here's the process of thinking that leads to the solution approach:

 No operation is needed if the sum of this pair already equals the target complementary sum. One operation is needed if we can add a number to one of the elements in the pair to achieve the target sum.

Two operations when neither of the elements in the pair can directly contribute to the target sum and both need to be

3. We use a trick called "difference array" or "prefix sum array" to efficiently update the range of sums with the respective counts of operations. We track the changes in the required operations as we move through different sum ranges.

4. We increment moves for all the sums and later decrement as per the pair values and the allowed limits.

- 5. Finally, we iterate over the sum range from 2 to 2 * limit and accumulate the changes recorded in our difference array. This reveals the total moves required for each sum to be the target sum.
- the array complementary. Understanding the optimized approach is a bit tricky, but it cleverly manages the different cases to minimize the total number of

6. Our goal is to find the minimum of these accumulated moves, which represents the least number of operations needed to make

The solution implements an optimization strategy called difference array, which is a form of prefix sum optimization. Here's the step by step explanation of how the algorithm works by walking through the code:

1. Initialize a difference array d with a length of 2 * limit + 2. This array will store the changes in the number of moves required to

make each sum complementary. The extra indices account for all possible sums which can occur from 2 (minimum sum) to 2 *

limit (maximum sum). 1 d = [0] * (limit * 2 + 2)

n by 2. For each pair (nums[i], nums[n - 1 - i]), determine their minimum (a) and maximum (b) values.

1 for i in range(n >> 1):

1 d[a + 1] -= 1

1 ans, s = n, 0

1 for v in d[2 : limit * 2 + 1]:

if ans > s:

ans = s

Example Walkthrough

Integer array nums: [1, 2, 4, 3]

2 d[b + limit + 1] += 1

Solution Approach

a, b = min(nums[i], nums[n - i - 1]), max(nums[i], nums[n - i - 1])3. Update the difference array to reflect the default case where 2 moves are needed for all sums from 2 to 2 * limit.

2. Loop through the first half of the nums array to check pairs of numbers. Here n >> 1 is a bitwise operation equivalent to dividing

1 d[2] += 22 d[limit * 2 + 1] -= 2 4. Update the difference array for scenarios where only 1 move is required. This happens when adding a number to the minimum of the pair, up to the maximum of the pair with limit.

5. For the exact sum a + b, decrease the number of moves by 1 as this is already part of the complementary sum. For a + b + 1,

6. Initialize ans with n, representing the maximum possible number of moves (every element needing 2 moves), and a sum s which

7. Loop through the potential sums to calculate the prefix sum (which represents the cumulative number of moves) at each value.

1 d[a + b] = 12 d[a + b + 1] += 1

revert this operation since we consider the sum a + b only.

Update ans to hold the minimum number of moves required.

will be used to accumulate the total moves from the difference array.

By leveraging the difference array technique, the algorithm efficiently calculates the minimum number of operations required by reducing the problem to range updates and point queries, which can be processed in a linear time complexity relative to the limit.

The length of the array n is 4, which is even, and the limit is 4, so any replacement operations must yield a number between 1 to 4.

8. Finally, we return ans, which now contains the minimum number of moves required to make nums complementary.

Let's walk through an example using the above solution approach. Suppose we have the following:

3. The difference array is first updated to indicate that, by default, 2 moves are needed for each sum.

1. We initialize our difference array d with a size of 2 * limit + 2 to account for all possible sums.

1 d[2] += 2

Integer limit: 4

2. There are two pairs to consider: (1, 3) and (2, 4).

4. Now, we loop through each pair. For the first pair (1, 3), the minimum is a = 1 and the maximum is b = 3. We then update the

5. For the exact sum a + b, which is 4, we decrease the number of moves by 1 as this is already part of the complementary sum.

1 d[a + 1] = 1 # d[2] becomes 1

1 d[a + b] = 1 # d[4] becomes -1

4 d[a + b] = 1 # d[6] becomes -1

1 ans, s = n, 0 # ans = 4, s = 0

could be accomplished by changing the 4 to a 2.

for i in range(n // 2):

delta[2] += 2

delta[limit * 2 + 1] = 2

 $delta[min_val + 1] -= 1$

delta[max_val + limit + 1] += 1

delta[min_val + max_val] -= 1

current_moves += moves_change

if min_moves > current_moves:

// Iterate over each pair

delta[2] += 2;

min_moves = current_moves

for (int i = 0; i < pairsCount; ++i) {</pre>

delta[limit * 2 + 1] -= 2;

delta[minOfPair + 1] -= 1;

delta[maxOfPair + limit + 1] += 1;

delta[min0fPair + max0fPair] -= 1;

if (currentCost < minMoves) {</pre>

minMoves = currentCost;

delta[min0fPair + max0fPair + 1] += 1;

int currentCost = 0; // Variable to accumulate current cost

return minMoves; // Return the minimum moves required

// Return the minimum moves to make all pair sums within the limit

const lower = Math.min(nums[i], nums[n - i - 1]); // Lower of the pair

const upper = Math.max(nums[i], nums[n - i - 1]); // Upper of the pair

// Decrement by one because at least one of the numbers need to change

// No changes needed if the sum equals the sum of the lower, upper pair

// Increment by two for the case where both numbers need to change

const delta: number[] = new Array(limit * 2 + 2).fill(0); // Array to track the changes in moves

function minMoves(nums: number[], limit: number): number {

// Calculate the change in moves for each pair

for (let i = 0; i < Math.floor(n / 2); ++i) {</pre>

const n = nums.length; // Number of elements in nums

return min_moves # Return the minimum moves required

delta[min_val + max_val + 1] += 1

for moves_change in delta[2: limit * 2 + 1]:

1 for v in d[2:8 + 1]:

3 ans = min(ans, s)

2 s += v

Python Solution

class Solution:

1.

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

33

34

35

36

37

39

41

40

41

42

43

45

8

9

10

11

12

13

14

15

16

17

18

19

25

44 };

return minMoves;

delta[2] += 2;

delta[limit * 2 + 1] -= 2;

delta[upper + limit + 1] += 1;

delta[lower + 1] -= 1;

Typescript Solution

40 }

5 d[a + b + 1] += 1 # d[7] becomes 1

2 d[a + b + 1] += 1 # d[5] becomes 1

2 d[b + limit + 1] += 1 # d[8] becomes 1

1 d = [0] * (4 * 2 + 2) # [0, 0, 0, 0, 0, 0, 0, 0]

2 d[9] -= 2 # Since our limit is 4, `limit * 2 + 1` is 9.

difference array based on the cases where 1 move is required.

For the value after a + b, which is 5, we revert this operation.

1 d[a + 1] = 1 # d[3] becomes 1 2 d[b + limit + 1] += 1 # d[9] stays at -1 because of the earlier decrement

7. After updating the difference array with all pairs, we initialize ans with the maximum possible number of moves and then

6. We repeat steps 4 and 5 for the second pair (2, 4). The minimum is a = 2 and the maximum is b = 4.

8. Finally, we iterate through the possible sums using the accumulated changes and find the minimum number of moves.

accumulate the changes to find the total moves required for each sum.

def minMoves(self, nums: List[int], limit: int) -> int: delta = [0] * (limit * 2 + 2) # Array to track the changes in the number of movesn = len(nums) # Number of elements in the list # Iterate over the first half of the list

Find the minimum and maximum of the pair (i-th and mirrored i-th elements)

Always counting 2 moves since every pair needs at least one move

Accumulate the deltas to include the total move changes up to each sum

min_val, $max_val = min(nums[i], nums[n - i - 1]), max(nums[i], nums[n - i - 1])$

Reducing by 1 move if only one element of the pair is changed to match the other

Reducing by another move if the sum can be achieved without altering either element

min_moves, current_moves = n, 0 # Initialize min_moves to maximum possible and current_moves

If the current number of moves is less than the previous minimum, update min_moves

int[] delta = new int[limit * 2 + 2]; // Delta array to store the cost changes

// Always needs 2 moves if sum is larger than max possible sum 'limit * 2'

int minMoves = pairsCount * 2; // Initialize minMoves to the maximum possible value

// If the sum is greater than maxOfPair + limit, then we'll need an additional move

// We need one less move for making the sum equals minOfPair + maxOfPair (to make a straight sum)

// If we make minOfPair + 1 the new sum, we would need one less move

// Taking the minimum and maximum of the pair for optimization

int minOfPair = Math.min(nums[i], nums[nums.length - i - 1]);

int maxOfPair = Math.max(nums[i], nums[nums.length - i - 1]);

The final accumulated changes in the difference array would be something like [0, 0, 1, 1, 0, 1, -1, 0, 2, -1]. We add these to

our running total s and continuously update ans to find that the minimum number of moves required to make nums complementary is

Therefore, the minimum number of moves required to make the array [1, 2, 4, 3] complementary with a limit of 4 is 1, which

Java Solution 1 class Solution { public int minMoves(int[] nums, int limit) { int pairsCount = nums.length / 2;// Store the number of pairs

```
28
            // Iterate over possible sums
29
30
            for (int sum = 2; sum <= limit * 2; ++sum) {</pre>
31
                currentCost += delta[sum]; // Update current cost
32
                // If current cost is less than minMoves, update minMoves
```

C++ Solution

#include <vector>

```
#include <algorithm>
   class Solution {
   public:
       // Function to determine the minimum number of moves required
       // to make all the sums of pairs in nums be less than or equal to limit
       int minMoves(std::vector<int>& nums, int limit) {
 8
            int n = nums.size(); // Number of elements in nums
 9
            std::vector<int> delta(limit * 2 + 2, 0); // Array to track the changes in moves
10
11
12
           // Calculate the change in moves for each pair
            for (int i = 0; i < n / 2; ++i) {
13
14
                int lower = std::min(nums[i], nums[n - i - 1]); // Lower of the pair
15
                int upper = std::max(nums[i], nums[n - i - 1]); // Upper of the pair
16
17
                // Increment by two for the worst case (both need to change)
18
                delta[2] += 2;
19
                delta[limit * 2 + 1] -= 2;
20
21
                // Decrement by one as at least one of the numbers need to change
22
                delta[lower + 1] -= 1;
23
                delta[upper + limit + 1] += 1;
24
25
                // No operation needed when the sum equals lower+upper
26
                delta[lower + upper] -= 1;
                delta[lower + upper + 1] += 1;
27
28
29
30
            int minMoves = n; // Start with the worst case moves count
31
            int currentMoves = 0; // Initialize a counter for the current moves
32
33
           // Accumulate the delta values and find the minimum moves
            for (int i = 2; i <= limit * 2; ++i) {
34
                currentMoves += delta[i];
35
36
                if (minMoves > currentMoves) {
37
                    minMoves = currentMoves; // Keep track of the minimum moves
38
39
```

delta[lower + upper] -= 1; 20 delta[lower + upper + 1] += 1; 21 22 23 let minMoves = n; // Initialize minimum moves with the worst case, which is n moves let currentMoves = 0; // Initialize the counter for the current moves 24

```
26
        // Accumulate the delta values and find the minimum moves required
 27
        for (let i = 2; i <= limit * 2; ++i) {
            currentMoves += delta[i];
 28
 29
            if (minMoves > currentMoves) {
                minMoves = currentMoves; // Update minimum moves if a new minimum is found
 31
 32
 33
 34
        // Return the minimum moves to make all pair sums within the limit
 35
        return minMoves;
 36 }
 37
Time and Space Complexity
Time Complexity:
The given code performs its core logic in a for-loop that iterates n/2 times, where n is the length of the list nums. For each iteration of
the loop, the code performs a constant number of arithmetic operations which are independent of the size of n. So, the time
complexity of the for-loop is O(n/2) which simplifies to O(n).
```

number of operations, contributing O(limit) to the complexity. Thus, the total time complexity of the code is O(n) + O(limit), which is the linear time complexity dominated by the larger of n and limit.

Space Complexity: The space complexity of the code is governed by the size of the list d, which is initialized as 0 * (limit * 2 + 2). This means the

After updating the d list, the code runs another loop that iterates 2*limit - 1 times. Each iteration of this loop performs a constant

space used by d is proportional to 2*limit + 2, which is O(limit).

Therefore, the space complexity of the given code is O(limit).