1551. Minimum Operations to Make Array Equal

Medium Math

Problem Description

We are given an array of integers where each element is uniquely determined by its index such that each element is equal to twice its index plus one. The formula for each element at index `i` is `arr[i] = (2 * i) + 1`. We want to make every element of the array equal by repeatedly performing a specific operation: selecting two indices `x` and `y` and then applying the changes `arr[x] -= 1` and `arr[y] += 1`.

Our task is to figure out the minimum number of such operations needed to achieve an array with all elements being equal. This is guaranteed to be possible.

For example, if `n` is 3, our array `arr` would be `[1, 3, 5]`. One way to make all elements equal would be by performing the operation `arr[2] -= 1` and `arr[0] += 1` to get `[2, 3, 4]`, and then `arr[2] -= 1` and `arr[0] += 1` again to get `[3, 3, 3]`. This takes two operations.

Intuition

To find the minimum number of operations, we need to understand the pattern formed by the array. The array is symmetrical around its middle value. This middle value, or target, is what we will reduce all other numbers to. In an array where `n` is odd, the middle value is part of the array, but for an even `n`, it's the average of the two central values.

Next, notice that for any index `i` below the middle of the array, there's a corresponding index from the end of the array where `arr[n-i-1] = arr[i] + 2 * i`. This means that we'll need `i` operations to make `arr[i]` equal to `arr[n-i-1]`, and similarly, all other symmetric pairs will require adjustments relative to their distance from the center.

For the minimum number of operations, we only need to consider half of the array because for every change in one half, the corresponding other half is adjusted by the same number of operations but in the opposite direction. By summing the needed operations for half of the array, we can find the total minimum operations required for the entire array.

The solution code simplifies this calculation using a comprehension list and a sum. It iterates through the first half of the array `for i in range(n >> 1)` (`n >> 1` is a bit shift operation equivalent to integer division by 2, effectively halving `n`) and calculates the difference between the actual value at that index (`i << 1 | 1`) (which calculates `2 * i + 1` using bit operations) and the targeted middle value `n`. Subtracting these differences accumulates the total number of operations required to transform the array's first half to the middle value. The sum of these operations is the answer because the array is symmetrical.

Solution Approach

The solution uses a simple mathematical approach to figure out the minimal number of operations required.

1. The target value for each `arr[i]` would be `n`, as `n = ((n - 1) * 2 + 1 + 1) / 2` which simplifies to `n` when `n` is odd; and `n = (n * 2 / 2)` which is also `n` when `n` is even. To clarify, when `n` is even, the target would be the average of two middle elements, which would still be `n`.
2. The solution iterates only through the first half of the array (`n >> 1`). This bit manipulation effectively halves `n`, ensuring the loop runs from `0` to `n/2 - 1` for even `n` and from `0` to `(n-1)/2` for odd `n`, accessing the first half of the array elements.
3. For each element in the first half of the array, we calculate the number of operations required to convert it to the target using the formula `n - (i << 1 | 1)`. This calculation is performed using bit manipulation:
   - `i << 1` shifts the value of index `i` one bit to the left, effectively multiplying it by 2.
   - The bitwise OR operation `| 1` then adds 1 to the result, giving us the formula `(2 * i + 1)`.
   - Subtracting this from `n` gives us the number of operations needed to reach the middle value from the current index `i`.
4. We sum these values using Python's `sum()`, which gives the total minimum number of operations required for the first half. Since the array is perfectly symmetrical, and the operation is mirrored on the second half of the array, this sum also represents the total operations for the entire array.
5. No additional data structures are needed as the problem deals with operations on array indices rather than the elements themselves, and the array is implied by the formula given for `arr[i]`.

In summary, the algorithm makes use of a symmetrical pattern in the unique array and understands that by bringing `n/2` elements to the middle value, the symmetrical property ensures the other half also becomes equal with the same number of operations. This allows us to solve the problem efficiently in `O(n/2)` time complexity, which simplifies to `O(n)`, and without any additional space, achieving an `O(1)` space complexity.

Example Walkthrough

Let's say `n = 5`, which means our array `arr` would be based on the formula `arr[i] = (2 * i) + 1` and thus the array will be `[1, 3, 5, 7, 9]`.

We want to make all elements equal by can doing the following:

- Determine the target value for each element to be `n`, which in this case is `5`.
- Iterate only through the first half of the array. We are interested in the index range `0` to `n/2 - 1`, which is `0` to `2`.
- Calculate the number of operations to convert each of the first half elements to `5`.

Let's compute:

- For `i=0`: Using the bit manipulation shown `n - (i << 1 | 1)` becomes `5 - (0 << 1 | 1)` ⇒ `5 - 1` ⇒ `4`. So, we need 4 operations to bring `1` up to `5`.
- For `i=1`: It becomes `5 - (1 << 1 | 1)` ⇒ `5 - 3` ⇒ `2`. We need 2 operations to bring `3` up to `5`.
- For `i=2`, which is the middle value, no operation is needed because it's already `5`, our target.

Now, we sum these operations: `4 + 2 + 0 = 6`.

So, to make all elements of the array `[1, 3, 5, 7, 9]` equal with each element being `5`, we need a minimum of 6 operations.

Given this process, the symmetrical nature of the array means that as we add to the first half of the elements to reach the target value, the second half will be reduced by the same count of operations, consequently they meet at the target value.

Thus, for the entire array, the minimal number of operations is the same 6 that we calculated for the first half, due to the symmetry. The operations on the second half are implied and don't have to be explicitly performed or calculated in this approach.

Solution Implementation

Python

```python
class Solution:
    def minOperations(self, n: int) -> int:
        # Initialize the number of operations to 0
        operations_count = 0

        # Loop over the first half of the sequence
        for i in range(n // 2):  # using // for floor division
            # The goal is to make all numbers equal to n
            # Each pair across the center will sum up to (n-1) + (n+1) = 2n
            # Thus, we only need to consider one half of the array and calculate
            # how much each number needs to change to become the middle value, n

            # Calculate the current value at position 'i' in the array
            current_value = (i << 2) + 1  # using * 2 instead of << 1 for clarity

            # Calculate the difference from n, which represents the number of operations needed
            operations_needed = n - current_value

            # Add the number of operations needed to the total count
            operations_count += operations_needed

        # Return the total number of operations
        return operations_count
```

Java

```java
class Solution {
    public int minOperations(int n) {
        // Initialize the variable to store the minimum number of operations required.
        int minOperations = 0;

        // Loop over the first half of the elements.
        for (int i = 0; i < n / 2; ++i) {
            // Calculate the target value which each element should reach
            // which is always equal to the middle value in the array, n.
            // Then, subtract the current value (2 * i + 1) from the target value.
            // This gives the number of operations for the current element to reach the target.
            minOperations += n - ((i * 2) + 1);
        }

        // Return the total number of operations for all elements.
        return minOperations;
    }
}
```

C++

```cpp
class Solution {
public:
    int minOperations(int n) {
        // Variable to store the minimum number of operations needed.
        int minOps = 0;

        // Loop through half of the elements since we only need to make operations
        // on one of the two symmetrical halves of the array.
        for (int i = 0; i < n / 2; ++i) {
            // Each operation counts how far the current element is from the center value.
            // Since the array is virtual and elements are 2 * i + 1, we subtract this
            // from n to find the required operations to make it equal to the center value.
            // Conceptually, this centralizes the elements towards the middle value of the array.
            minOps += n - (2 * i + 1);
        }

        // Return the calculated minimum number of operations.
        return minOps;
    }
};
```

TypeScript

```typescript
/**
 * This function calculates the minimum number of operations needed to make all
 * elements equal in an array where the array contains n elements that
 * increase by 2 starting from 1, 3, 5, ... (2n-1).
 *
 * An operation is defined as incrementing or decrementing an element of the array.
 *
 * The strategy is to make all the elements equal to the middle value of the array.
 *
 * @param {number} n The number of elements in the array.
 * @return {number} The minimum number of operations needed.
 */
function minOperations(n: number): number {
    // Initialize the counter for the minimum operations to 0.
    let operationsCount = 0;

    // Loop over the first half of the array since it is symmetric around the middle.
    for (let i = 0; i < (n >> 1); ++i) {
        // Calculate the operations needed to make the i-th element (from 0th) equal to the middle value
        // (i << 1) represents 2*i, the '| 1' sets the last bit to 1 (making it odd),
        // n - (2*i+1) is how much we need to add to the i-th element to reach the middle value of the array.
        operationsCount += n - ((i << 1) | 1);
    }

    // Return the calculated number of operations.
    return operationsCount;
}
```

```python
class Solution:
    def minOperations(self, n: int) -> int:
        # Initialize the number of operations to 0
        operations_count = 0

        # Loop over the first half of the sequence
        for i in range(n // 2):  # using // for floor division
            # The goal is to make all numbers equal to n
            # Each pair across the center will sum up to (n-1) + (n+1) = 2n
            # Thus, we only need to consider one half of the array and calculate
            # how much each number needs to change to become the middle value, n

            # Calculate the current value at position 'i' in the array
            current_value = (i * 2) + 1  # using * 2 instead of << 1 for clarity

            # Calculate the difference from n, which represents the number of operations needed
            operations_needed = n - current_value

            # Add the number of operations needed to the total count
            operations_count += operations_needed

        # Return the total number of operations
        return operations_count
```

Time and Space Complexity

The function `minOperations` calculates the minimum number of operations required to make all elements of an array of length `n` (where array elements are `2*i+1` for `i=0,1,...,n-1`) equal by incrementing or decrementing elements by 1.

Time Complexity

The time complexity of the function is largely determined by the comprehension loop within the `sum` function. The loop runs from `0` to `n>>1`, which is effectively `n/2`. Each iteration performs a constant amount of work by calculating the difference `(n - (i << 1 | 1))`. Since the loop iterates `n/2` times and each operation is `O(1)`, the total time complexity of the function is `O(n/2)`, which simplifies to `O(n)`.

Space Complexity

The space complexity of the function is `O(1)`. The reason for this is that the function only uses a fixed amount of space, regardless of the input size `n`. It generates no additional data structures that grow with `n`. The sum is computed on the fly, and the loop does not store any intermediate results (aside from temporary variables used in computation, which do not depend on `n` in terms of space).