1330. Reverse Subarray To Maximize Array Value

Problem Description

In this problem, we are given an integer array called `nums`. The **value** of this array is defined as the sum of the absolute differences between consecutive elements, expressed as `|nums[i] - nums[i + 1]|` for all `0 <= i < nums.length - 1`.

We are allowed to perform one operation on this array: select any subarray and reverse it. The goal is to find the maximum possible value for the array after possibly performing this reversal operation exactly once.

The task is to figure out if reversing a certain subarray can lead to an increase in the total value of the array as defined and, if so, to maximize this value.

Intuition

To solve this problem, we need to consider what happens to the value of the array when we reverse a subarray. Reversing a subarray could potentially increase the total value if the differences between the numbers at the borders of the subarray and the adjacent numbers outside the subarray are increased.

The solution approach involves several parts:

1. Calculating the initial value (`sum`) of the array, which is the sum of the absolute differences between all consecutive elements. This value is the baseline that we will try to improve with the reversal operation.
2. Checking if reversing the start or the end of the array with any other element `x` or `y` in the array can lead to an increase in value. This means we compare the gains from altering the initial or final elements with each inside pair `x, y`.
3. Iterating over the array to find the best possible subarray to reverse that would result in the highest increase in value. We do this by simulating the effects of reversing a subarray on contribution to the value from each pair of numbers. This involves keeping track of two maximum values possible from the operations considering both scenarios when `|nums[0] - y|` or `|nums[-1] - x|` contributes to the overall value increase.

By iteratively updating the maximum answer (`ans`), the code captures the best possible value that can be achieved by a single reversal of any subarray. The math behind this involves careful consideration of how the absolute value differences change with potential reversals and how this impact can be maximized.

Solution Approach

The solution uses a greedy approach to maximize the value of the array by considering the effect of reversing subarrays on the total value. Here's an explanation of the code implementation:

1. Firstly, the solution calculates the initial total value `s` of the array as the sum of the absolute differences between all consecutive elements.

```
s = sum(abs(x - y) for x, y in pairwise(nums))
```

   This is done using a generator expression within the sum function, which iterates pairs of consecutive elements using Python's `pairwise` utility. The `abs` function is used to calculate the absolute difference.

2. The solution then iterates over these pairs again, considering the potential impact of a reversal operation that includes the first or the last element of the array. The goal here is to find if such a reversal can increase the total value by creating a larger difference at the array's ends.

```
for x, y in pairwise(nums):
    ans = max(ans, s + abs(nums[0] - y) - abs(x - y))
    ans = max(ans, s + abs(nums[-1] - x) - abs(x - y))
```

   For each adjacent pair, it computes two potential new values for `ans` considering the reversal of elements at the start or end of the array, and updates `ans` to the maximum of these values.

3. In the final part of the solution, the algorithm considers reversing subarrays in the middle of the array and how it affects the total value. It explores flipping the sign of the components of the pairs and then measures the potential increase in the total value that could come from such operations. This part of the solution uses a trick that combines pairs with different signs in order to simulate the effect of a reversal.

```
for k1, k2 in pairwise((1, -1, -1, 1, 1)):
    mx = -inf
    for x, y in pairwise(nums):
        a = k1 * x + k2 * y
        b = abs(x - y)
        mx = max(mx, a - b)
        mn = min(mn, a - b)
    ans = max(ans, s + max(x - mn, 0))
```

   It uses variables `mx` and `mn` to keep track of the maximum and minimum possible values when considering both options of element placement after a reversal. The `max(mx - mn, 0)` ensures that only positive changes to the total value are considered, and the overall maximum value is updated accordingly.

Overall, the implemented solution methodically tests all possible single reversal operations that could lead to an increase in the total value, ensuring that the maximum possible result is found. Variables `mx`, `mn`, and `ans` are effectively used to measure and track possible improvements to the total value without ever having to perform an actual subarray reversal.

Example Walkthrough

Let's work through the solution approach with a small example and see how it applies. Suppose we are given the following integer array `nums`:

```
nums = [4, 3, 2, 5]
```

1. Firstly, we calculate the initial value `s` of the array, which is the sum of the absolute differences between consecutive elements. In our case:

```
s = |4 - 3| + |3 - 2| + |2 - 5| = 1 + 1 + 3 = 5
```

2. Now, we consider the effect of a reversal operation that includes the first or the last element. We iterate over the pairs and calculate the new potential value after such a reversal.

```
For the pair (4, 3):
If we reverse this section, we would get the array [3, 4, 2, 5].
The new value would be |3 - 4| + |4 - 2| + |2 - 5| = 5, which is not an improvement over s.

For the pair (3, 2):
If we reverse from the start, we would get [3, 3, 4, 5].
The new value would be |3 - 3| + |3 - 4| + |4 - 5| = 3, which is less than s.

If we reverse from the end, we would get the array [4, 2, 3, 5].
The new value would be |4 - 2| + |2 - 3| + |3 - 5| = 4, which again is not an improvement.

For the pair (2, 5):
Reversing with the start doesn't change anything since it is already at the end.
```

   Here, we do not find any improvement, so we continue the search.

3. For the final part, we consider subarrays in the middle of the array. We try to manipulate the calculations based on possible subarray reversals without actually reversing them.

```
Let's iterate over the pairs and consider the flips in sign for each pair, which simulate the effects of reversals.
3
Examining each element and computing potential value changes (ignoring the sign flips for brevity):
4
mx = max(-inf, 4 - 1, 3 - 1, 2 - 3) = 3
mn = min(inf, 4 - 1, 3 - 1, 2 - 3) = 4
6
The potential increase to s is mx - mn, which is 3 - 4 = -1. Since it's not positive, it doesn't improve the current value.
8
If we continue this approach for each possible subarray reversal, we realize that no reversal in the middle improves the value fr
```

In this example, the maximum value that can be obtained after reversing any subarray is the same as the initial value `s = 5`. Therefore, in this case, performing a reversal operation does not lead to an increased total value. The result is that the best strategy for `nums` would be to leave the array unchanged.

Python Solution

```python
from itertools import pairwise  # pairwise utility from itertools
from math import inf  # constant for representing infinity

class Solution:
    def maxValueAfterReverse(self, nums: List[int]) -> int:
        # Initial total of absolute differences between all adjacent pairs
        total_difference = sum(abs(x - y) for x, y in pairwise(nums))

        # Initialize the answer with the initial total difference
        answer = total_difference

        # Try reversing from the start and compare the impact on total difference
        for x, y in pairwise(nums):
            answer = max(answer, total_difference + abs(nums[0] - y) - abs(x - y))
            answer = max(answer, total_difference + abs(nums[-1] - x) - abs(x - y))

        # Check for all possible reversal sequences using pairwise comparison
        for coeff1, coeff2 in pairwise((1, -1, -1, 1, 1)):
            max_difference = -inf  # set to negative infinity to find the maximum
            min_difference = inf  # set to positive infinity to find the minimum

            # Iterate over all pairwise elements to find the new possible max difference
            for x, y in pairwise(nums):
                a = coeff1 * x + coeff2 * y
                abs_difference = abs(x - y)
                max_difference = max(max_difference, a - abs_difference)
                min_difference = min(min_difference, a + abs_difference)

            # Calculate the max total difference if this segment was reversed
            answer = max(answer, total_difference + max(max_difference - min_difference, 0))

        # Return the maximum value of total difference after any possible reverse operation
        return answer
```

Java Solution

```java
class Solution {
    public int maxValueAfterReverse(int[] nums) {
        int numsLength = nums.length;
        // Initialize the starting sum of absolute differences
        int totalSum = 0;
        for (int i = 0; i < n - 1; ++i) {
            totalSum += Math.abs(nums[i] - nums[i + 1]);
        }

        // Initialize the answer with the initial total sum
        int maxSum = totalSum;

        // Check if reversing subarray starting from the beginning or ending at the end gives better sum
        for (int i = 0; i < n - 1; ++i) {
            maxSum = Math.max(maxSum, totalSum + Math.abs(nums[0] - nums[i + 1]) - Math.abs(nums[i] - nums[i + 1]));
            maxSum = Math.max(maxSum, totalSum + Math.abs(nums[n - 1] - nums[i]) - Math.abs(nums[i] - nums[i + 1]));
        }

        // Prepare directions for the operations to be applied
        int[] directions = {1, -1, -1, 1, 1};
        // Use infinity to initialize max and min difference
        final int infinity = Integer.MAX_VALUE;
        // Check for all four combinations of directions
        for (int k = 0; k < 4; ++k) {
            int direction1 = directions[k], direction2 = directions[k + 1];
            int maxDiff = -infinity, minDiff = infinity;
            // Traverse to find the maximum and minimum values accordingly
            for (int i = 0; i < n - 1; ++i) {
                int a = direction1 * nums[i] + direction2 * nums[i + 1];
                int absoluteDifference = Math.abs(nums[i] - nums[i + 1]);
                maxDiff = Math.max(maxDiff, a - absoluteDifference);
                minDiff = Math.min(minDiff, a + absoluteDifference);
            }
            // Update if the difference between maxDiff and minDiff improves the sum
            maxSum = Math.max(maxSum, totalSum + Math.max(0, maxDiff - minDiff));
        }
        return maxSum; // Return the maximized sum after operations
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int maxValueAfterReverse(vector<int>& nums) {
        int totalVariation = 0; // Total variation in the original array based on sum of absolute differences
        int maxValue = 0;       // Max value after reversing a subarray
        int n = nums.size();

        // Calculating the total initial variation of the array
        for (int i = 0; i < n - 1; ++i) {
            totalVariation += abs(nums[i] - nums[i + 1]);
        }
        maxValue = totalVariation;

        // Trying to reverse from the start to each position and checking for possible improvement
        for (int i = 0; i < n - 1; ++i) {
            maxValue = max(maxValue, totalVariation + abs(nums[0] - nums[i + 1]) - abs(nums[i] - nums[i + 1]));
            maxValue = max(maxValue, totalVariation + abs(nums[n - 1] - nums[i]) - abs(nums[i] - nums[i + 1]));
        }

        int directions[5] = {1, -1, -1, 1, 1}; // Direction multipliers to simplify the max/min calculations
        const int infinity = 1 << 30;          // Placeholder for infinity to handle extreme values

        // Try to find reverses that could benefit from the extremities
        for (int k = 0; k < 4; ++k) {
            int dirStart = directions[k], dirEnd = directions[k + 1];
            int maxDiff = -infinity, minSum = infinity;

            // Iterate over the array and look for optimal reversing points
            for (int i = 0; i < n - 1; ++i) {
                int weightedEdgeValue = dirStart * nums[i] + dirEnd * nums[i + 1];
                int currentVariation = abs(nums[i] - nums[i + 1]);
                maxDiff = max(maxDiff, weightedEdgeValue - currentVariation);
                minSum = min(minSum, weightedEdgeValue + currentVariation);
            }

            // Compare the differences between the max difference and min sum and update the max value accordingly
            maxValue = max(maxValue, totalVariation + max(0, maxDiff - minSum));
        }

        return maxValue; // Return the maximum possible value after reverse operation
    }
};
```

Typescript Solution

```typescript
function maxValueAfterReverse(nums: number[]): number {
    const length = nums.length;
    let currentSum = 0;

    // Calculate the initial sum of absolute differences
    for (let i = 0; i < length - 1; ++i) {
        currentSum += Math.abs(nums[i] - nums[i + 1]);
    }

    // Initialize answer with the initial sum
    let maxSum = currentSum;

    // Try reversing num[0,...i] and num[i+1,...n-1] for each i and calculate the maximum sum
    for (let i = 0; i < length - 1; ++i) {
        const difference = Math.abs(nums[i] - nums[i + 1]);
        maxSum = Math.max(maxSum, currentSum + Math.abs(nums[0] - nums[i + 1]) - difference);
        maxSum = Math.max(maxSum, currentSum + Math.abs(nums[length - 1] - nums[i]) - difference);
    }

    // Constants for direction in the next loop, simulating a 2D direction array
    const directions = [1, -1, -1, 1, 1];
    const infinity = 1 << 30; // A large number representing infinity

    // Try to maximize the sum by considering each subarray and its combination with directions
    for (let k = 0; k < 4; ++k) {
        let maxAdjustedValue = -infinity;
        let minAdjustedValue = infinity;

        for (let i = 0; i < length - 1; ++i) {
            const adjustedDifference = directions[k] * nums[i] + directions[k + 1] * nums[i + 1];
            const absoluteDifference = Math.abs(nums[i] - nums[i + 1]);

            maxAdjustedValue = Math.max(maxAdjustedValue, adjustedDifference - absoluteDifference);
            minAdjustedValue = Math.min(minAdjustedValue, adjustedDifference + absoluteDifference);
        }

        maxSum = Math.max(maxSum, currentSum + Math.max(0, maxAdjustedValue - minAdjustedValue));
    }

    // Return the maximum sum after reversal operation
    return maxSum;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the function `maxValueAfterReverse` consists of several parts:

- The first summation over `pairwise(nums)` takes $O(N)$ time, where N is the length of the list `nums`. This is because we iterate over each pair of consecutive elements.
- The second and third `for` loops each run $O(N)$ times (as they iterate over each pair of `pairwise(nums)` once) and perform a constant amount of work within the loop.
- The fourth `for` loop runs over the pairwise tuples `(1, -1, -1, 1, 1)` is a constant-size tuple, hence the outer loop runs a constant number of times (5 times in this case). The inner loop again runs over each pair of `pairwise(nums)` once, taking $O(N)$ time. Within this inner loop, we perform a few arithmetic operations and comparisons, both of which take constant time.

Since we have a constant number of $O(N)$ operations (the two single $O(N)$ loops and the nested $O(N)$ loop which is executed 5 times), we can sum these to still get $O(N)$.

So the final time complexity of the entire function is $O(N)$.

Space Complexity

The function uses a fixed number of single-element variables (`ans`, `s`, `mx`, `mi`), which only occupy $O(1)$ extra space.

The `pairwise` generator itself does not create a list of pairs but instead produces them one at a time when iterated over, so it only uses $O(1)$ space rather than $O(N)$.

Thus, the overall space complexity of the function is $O(1)$.