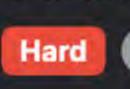
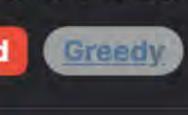
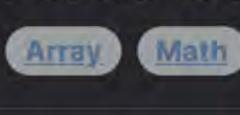
1330. Reverse Subarray To Maximize Array Value







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  $0 \ll i < nums.length - 1$ .

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.

maximize this value. Intuition

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

### 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 (s) of the array, which is the sum of the absolute differences between all consecutive elements. This

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.

value is the baseline that we will try to improve with the reversal operation.

- 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 the maximum and minimum 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.

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.

Solution Approach

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.

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. 1 for x, y in pairwise(nums):

2. The solution then iterates over these pairs again, considering the potential impact of a reversal operation that includes the first

```
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.
```

simulate the effect of a reversal.

b = abs(x - y)

mx = max(mx, a - b)

mi = min(mi, a + b)

8 ans = max(ans, s + max(mx - mi, 0))

2 ans = max(ans, s + abs(nums[0] - y) - abs(x - y))

ans = max(ans, s + abs(nums[-1] - x) - abs(x - y))

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

1 for k1, k2 in pairwise((1, -1, -1, 1, 1)): mx, mi = -inf, inffor x, y in pairwise(nums): a = k1 \* x + k2 \* y

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

could come from such operations. This part of the solution uses a trick that combines pairs with different signs in order to

value. It explores flipping the sign of the components of the pairs and then measures the potential increase in the total value that

It uses variables mx and mi to keep track of the maximum and minimum possible values when considering both options of element placement after a reversal. The max(mx - mi, 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, mi, and ans are effectively used to measure and track

possible improvements to the total value without ever having to perform an actual subarray reversal.

3 The new value would be |3-4|+|4-2|+|2-5|=5, which is not an improvement over s.

10 The new value would be |4-2|+|2-3|+|3-5|=5, which again is not an improvement.

answer =  $max(answer, total_difference + abs(nums[0] - y) - abs(x - y))$ 

max\_difference = -inf # set to negative infinity to find the maximum

min\_difference = inf # set to positive infinity to find the minimum

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

# Iterate over all pairwise elements to find the new possible max difference

answer = max(answer, total\_difference + max(max\_difference - min\_difference, 0))

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

# Check for all possible reversal segments using pairwise comparison

for coef1, coef2 in pairwise((1, -1, -1, 1, 1)):

for x, y in pairwise(nums):

a = coef1 \* x + coef2 \* y

public int maxValueAfterReverse(int[] nums) {

for (int i = 0; i < n - 1; ++i) {

for (int i = 0; i < n - 1; ++i) {

for (int i = 0; i < n - 1; ++i) {

const int infinity = 1 << 30;</pre>

for (int k = 0; k < 4; ++k) {

maxValue = totalVariation;

totalVariation += abs(nums[i] - nums[i + 1]);

// Try to find reverses that could benefit from the extremities

int maxDiff = -infinity, minSum = infinity;

for (int i = 0; i < n - 1; ++i) {

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

for (let i = 0; i < length - 1; ++i) {

// Calculate the initial sum of absolute differences

const length = nums.length;

let currentSum = 0;

int dirStart = directions[k], dirEnd = directions[k + 1];

int currentVariation = abs(nums[i] - nums[i + 1]);

// Iterate over the array and look for optimal reversing points

int weightedEdgeValue = dirStart \* nums[i] + dirEnd \* nums[i + 1];

maxDiff = max(maxDiff, weightedEdgeValue - currentVariation);

minSum = min(minSum, weightedEdgeValue + currentVariation);

maxValue = max(maxValue, totalVariation + max(0, maxDiff - minSum));

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

// Trying to reverse from the start to each position and checking for possible improvement

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]));

// Compare the differences between the max difference and min sum and update the max value accordingly

// Placeholder for infinity to handle extreme values

int directions  $[5] = \{1, -1, -1, 1, 1\}$ ; // Direction multipliers to simplify the max/min calculations

// Initialize the starting sum of absolute differences

int n = nums.length;

int totalSum = 0;

 $abs_difference = abs(x - y)$ 

answer =  $max(answer, total_difference + abs(nums[-1] - x) - abs(x - y))$ 

The new value would be |2 - 3| + |3 - 4| + |4 - 5| = 3, which is less than s.

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

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

2 If we reverse this section, we would get the array [3, 4, 2, 5].

9 If we reverse from the end, we would get the array [4, 2, 3, 5].

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

6 If we reverse from the start, we would get [2, 3, 4, 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.

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

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

5 mx = max(-inf, 4 - 1, 3 - 1, 2 - 3) = 3

6 mi = min(inf, 4 + 1, 3 + 1, 2 + 3) = 4

for nums would be to leave the array unchanged.

11

1 For the pair (4, 3):

5 For the pair (3, 2):

In our case:

subarray reversals without actually reversing them. 1 Let's iterate over the pairs and consider the flips in sign for each pair, which simulate the effects of reversals. Examining each element and computing potential value changes (ignoring the sign flips for brevity):

The potential increase to s is mx - mi, which is 3 - 4 = -1. Since it's not positive, it doesn't improve the current value.

Therefore, in this case, performing a reversal operation does not lead to an increased total value. The result is that the best strategy

In this example, the maximum value that can be obtained after reversing any subarray is the same as the initial value s = 5.

10 If we continue this approach for each possible subarray reversal, we realize that no reversal in the middle improves the value for

3. For the final part, we consider subarrays in the middle of the array. We try to manipulate the calculations based on possible

```
Python Solution
  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):
```

```
Java Solution
```

1 class Solution {

return answer

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

6

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

31 32

33

34

35

36

37

38

39

40

41

4

5

6

42 };

```
totalSum += Math.abs(nums[i] - nums[i + 1]);
  8
             // Initialize the answer with the initial total sum
  9
             int maxSum = totalSum;
 10
 11
 12
             // Check if reversing subarray starting from the beginning or ending at the end gives better sum
 13
             for (int i = 0; i < n - 1; ++i) {
 14
                 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]));
 15
 16
 17
 18
             // Prepare directions for the operations to be applied
 19
             int[] directions = \{1, -1, -1, 1, 1\};
 20
             // Use infinity to initialize max and min difference
             final int infinity = Integer.MAX_VALUE;
 21
 22
             // Check for all four combinations of directions
 23
             for (int k = 0; k < 4; ++k) {
 24
                 int direction1 = directions[k], direction2 = directions[k + 1];
 25
                 int maxDiff = -infinity, minDiff = infinity;
 26
                 // Traverse and find the maximum and minimum values accordingly
 27
                 for (int i = 0; i < n - 1; ++i) {
 28
                     int a = direction1 * nums[i] + direction2 * nums[i + 1];
 29
                     int absoluteDifference = Math.abs(nums[i] - nums[i + 1]);
 30
                     maxDiff = Math.max(maxDiff, a - absoluteDifference);
                     minDiff = Math.min(minDiff, a + absoluteDifference);
 31
 32
 33
                 // Update if the difference between maxDiff and minDiff improves the sum
 34
                 maxSum = Math.max(maxSum, totalSum + Math.max(0, maxDiff - minDiff));
 35
             return maxSum; // Return the maximized sum after operations
 36
 37
 38 }
 39
C++ Solution
  1 class Solution {
    public:
         int maxValueAfterReverse(vector<int>& nums) {
             int totalVariation = 0; // Total variation in the original array based on sum of absolute differences
                                     // Max value after reversing a subarray
             int maxValue = 0;
             int n = nums.size();
  8
             // Calculating the total initial variation of the array
```

### 43 Typescript Solution

```
currentSum += Math.abs(nums[i] - nums[i + 1]);
  8
  9
 10
        // Initialize answer with the initial sum
 11
         let maxSum = currentSum;
 12
        // Try reversing nums[0...i] and nums[i+1...n-1] for each i and calculate the maximum sum
 13
         for (let i = 0; i < length - 1; ++i) {
 14
             const difference = Math.abs(nums[i] - nums[i + 1]);
 15
 16
             maxSum = Math.max(maxSum, currentSum + Math.abs(nums[0] - nums[i + 1]) - difference);
 17
             maxSum = Math.max(maxSum, currentSum + Math.abs(nums[length - 1] - nums[i]) - difference);
 18
 19
 20
        // Constants for direction in the next loop, simulating a 2D direction array
 21
         const directions = [1, -1, -1, 1, 1];
        const infinity = 1 << 30; // A large number representing infinity</pre>
 22
 23
 24
        // Try to maximize the sum by considering each subarray and its combination with directions
 25
         for (let k = 0; k < 4; ++k) {
 26
             let maxAdjustedValue = -infinity;
             let minAdjustedValue = infinity;
 27
 28
             for (let i = 0; i < length - 1; ++i) {
 29
 30
                 const adjustedDifference = directions[k] * nums[i] + directions[k + 1] * nums[i + 1];
                 const absoluteDifference = Math.abs(nums[i] - nums[i + 1]);
 31
 32
 33
                maxAdjustedValue = Math.max(maxAdjustedValue, adjustedDifference - absoluteDifference);
                minAdjustedValue = Math.min(minAdjustedValue, adjustedDifference + absoluteDifference);
 34
 35
 36
 37
            maxSum = Math.max(maxSum, currentSum + Math.max(0, maxAdjustedValue - minAdjustedValue));
 38
 39
 40
        // Return the maximum sum after reversal operation
        return maxSum;
Time and Space Complexity
```

# **Time Complexity**

each pair of consecutive elements.

41 42 } 43

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

- 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),

• 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

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 0(1) extra space.

we can sum these to still get O(N).

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 0(1) space rather than 0(N).

Thus, the overall space complexity of the function is 0(1).