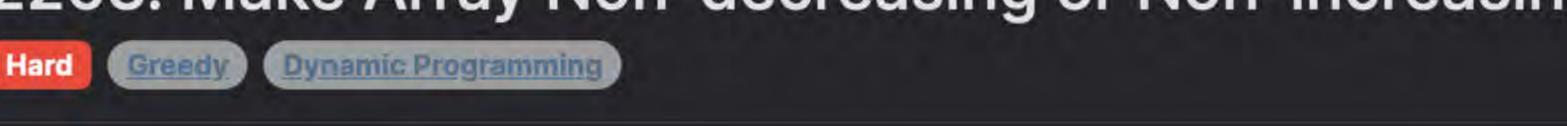
2263. Make Array Non-decreasing or Non-increasing



In this LeetCode problem, we are given an array nums of integers with a 0-based index. The goal is to determine the minimum number

Problem Description

of operations needed to transform nums into a non-decreasing (every element is greater than or equal to the previous one) or nonincreasing (every element is less than or equal to the previous one) sequence. An operation consists of selecting any element in the array and either incrementing it by 1 or decrementing it by 1.

Leetcode Link

Intuition

For both cases, the approach is to use dynamic programming (DP) to efficiently calculate the minimum number of operations. The

convert the first i elements of the array into a sequence that ends with the number j. To do this, the solution iterates over each element x of the input array and updates the DP table to reflect the minimum number of

basic idea of the DP solution is to create a 2D array f where each entry f[i][j] represents the minimum number of operations to

operations considering both the possibility of incrementing and decrementing. Specifically, during the update process for each number x, it considers the possible outcomes for having each value from 0 to 1000 at position i. For each of these values, it finds the minimum operations from the previous step and the current cost of making x equal

to the considered value. The cost is simply the absolute difference between x and the considered value. Finally, the solution iterates over the values of the last element (the last row of the DP table), collecting the minimum number of operations required to achieve both non-decreasing and non-increasing order, respectively. The result is the minimum of these two

values. In summary, the intuition is to explore all possible target values for each element separately for both non-decreasing and nonincreasing cases, combine them smartly using dynamic programming to keep track of the minimum cost at each step, and finally

Solution Approach The solution uses dynamic programming as its core approach, which is a method usually used to solve problems by breaking them down into smaller and simpler sub-problems, solving each sub-problem just once, and saving their solutions - typically using a

The solve function in the code defines the process for transforming the array to either non-decreasing or non-increasing order by

considering one direction at a time. The function sets up a 2D array f to store the minimum operations needed to achieve the desired

order with an array ending in value j at index i. The size of f is (n+1) x 1001, where n is the length of nums and 1001 accommodates

Iterate through each element x in nums, starting at index 1 for the array f since index 0 is used for the base case where no

For each element x, we iterate from 0 to 1000 to consider all possible values j that the current element could take. mi stores the

After updating f[i][j] for all values from 0 to 1000, it retains the minimum value reached for the entire sequence by taking the

Post the solve function, the main part of the convertArray method applies this process twice: once for nums as is, and once for the

reversed nums. Reversing nums effectively solves for a non-increasing sequence, as the reversed-non-decreasing sequence is the

Let's walk through the main steps of the solve function:

take the minimum of the costs from each of these two scenarios.

operations have been applied (f[0] is initialized with zeros).

minimum of the last row of f, essentially, min(f[n]).

all possible values an element can take after the operations (assuming the limit as 1000).

Iterative approach for dynamic programming with memory optimization properties.

considering all possible values that each element can take post operations (0 to 1000).

0 to 1000. For example, to make 3 a 0, we would need f[1][0] = 3 operations.

3. Continue filling f using the same approach for elements 5, 1, and 6.

the determination of the minimum number of operations needed dynamically as the array is processed.

memory-based data structure (array, map, etc.).

minimum operations needed to get to the previous element's state, stored in mi. This dynamic programming step ensures that for each element, f stores the accumulated minimum operations to adjust the sequence to non-decreasing/non-increasing up to that element.

• For every possible value j, it calculates the number of operations required to adjust x to j (abs(x - j)) and adds it to the

minimum number of operations seen so far for transforming the sequence up to the previous element.

original non-increasing sequence. The final answer is the minimum number of operations between the non-decreasing and non-increasing transformations.

 A 2D list f to serve as the DP table. Patterns:

Separating the problem into two scenarios (non-decreasing and non-increasing) and taking the minimum of the two results.

The clever part of this algorithm is its ability to handle all possible end values for each subarray in an efficient manner, allowing for

Let's take a small example to illustrate the solution approach with the given array nums:

Example Walkthrough

1 nums = [3, 2, 5, 1, 6]

Algorithm Used:

Data Structures:

Dynamic Programming.

Converting to Non-Decreasing:

We loop over each element in nums starting from index 1 in f as index 0 represents the base case where the array is empty (no operations needed).

1. Element 3: The initial minimum operations matrix f at index 1 will be set considering the cost of making 3 all possible values from

2. Element 2: Now, for each target value j from 0 to 1000, we calculate the minimum of f[1] [0..1000] and add the cost to convert

Initialize 2D array f with size (n+1) x 1001, where n is the length of nums. In our case, n is 5, so we have f of size 6 x 1001. We're

We want to transform this array into either a non-decreasing or non-increasing sequence using the minimum number of operations.

At the end of this process, f[5][j] contains the minimum operations needed to convert the array into a non-decreasing sequence ending with value j. The answer will be the minimum value in f[5][0..1000].

values.

Summary:

Converting to Non-Increasing:

2 to j, updating f[2][j] accordingly.

take the minimum value from f[5] [0..1000].

from typing import List

def solve(arr):

class Solution:

We repeat the same process, but before starting, we reverse the array: 1 Reversed nums = [6, 1, 5, 2, 3]

2. Element 1: Just as in the non-decreasing case, we iterate over all possible target values j, and for each, add the minimum of

After this, the f[5][j] row reflects the minimum operations required to make the original array non-increasing. The final step is to

1. Element 6: Similar to the non-decreasing process, f[1] [0..1000] is set according to the cost to change 6 into all possible ending

Finally, the solution will return the smaller value between the two minimums obtained for both the non-decreasing and nonincreasing scenarios. This represents the minimum number of operations required to convert the initial array into either a non-

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

for j in range(1001):

return min(solve(nums), solve(nums[::-1]))

return min(min_ops[n])

// Main method to convert the array.

// Populate the dp table

for (int i = 1; $i \le n$; ++i) {

public int convertArray(int[] nums) {

private int computeMinimumOperations(int[] nums) {

int n = nums.length; // Length of the array

for (int j = 0; $j \ll 1000$; ++j) {

print(sol.convertArray([1, 5, 3, 3]))

n = len(arr) # Length of the array

 $min_{ops} = [[0] * 1001 for _ in range(n + 1)]$

if min_value > min_ops[i - 1][j]:

min_value = min_ops[i - 1][j]

Find and return the minimum value from the last row

min_ops[i][j] = min_value + abs(num - j)

// Find the minimum of the original array and the reversed array

int[][] dp = new int[n + 1][1001]; // DP table to store minimal operations

int minPrevious = Integer.MAX_VALUE; // Initialize to max value

// Calculate min operations for getting value j at index i

minPrevious = Math.min(minPrevious, dp[i - 1][j]);

dp[i][j] = minPrevious + Math.abs(j - nums[i - 1]);

// Reversing the nums vector to calculate cost from both ends

// Return the minimum cost of the two possible scenarios

// Helper function to calculate the minimum cost of making all

// Initialize the 2D array with zero using memset

// hold minimum cost to make the first i elements equal to j

// Create a 2D array to store subproblem solutions, where f[i][j] will

minCostPrevious = min(minCostPrevious, f[i - 1][j]);

// with the sum of minCostPrevious and current cost

f[i][j] = minCostPrevious + abs(nums[i - 1] - j);

int minCostPrevious = INT_MAX; // Initialize the minimum cost to a large value

// Update minCostPrevious to the minimum cost found in previous row

// Calculate cost of making nums[i-1] equal to j and update f[i][j]

// Return the minimum cost of making all elements of the array equal until the last element

reverse(nums.begin(), nums.end());

return min(cost1, cost2);

int n = nums.size();

int f[n + 1][1001];

memset(f, 0, sizeof(f));

for (int i = 1; $i \le n$; ++i) {

// Cost calculated after reversing array

// elements equal by incrementing or decrementing

for (int j = 0; $j \le 1000$; ++j) {

return *min_element(f[n], f[n] + 1001);

1 // Global variable to define the maximum value for calculation

let cost1: number = calculateMinimumCost(nums);

let cost2: number = calculateMinimumCost(nums);

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

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

// Cost calculated after reversing array

let n: number = nums.length;

// Function to find the minimum cost to make all elements of the array equal

// Create a 2D array to store subproblem solutions, where dp[i][j] will

// Reversing the nums array to calculate cost from both ends

int calculateMinimumCost(vector<int>& nums) {

int cost2 = calculateMinimumCost(nums);

// Method to compute the minimum operations to satisfy the condition.

// Update minimum from the previous row

f[1] [0..1000] and the cost to convert 1 to j into f[2][j].

3. Proceed with each element (5, 2, 3) in the reversed array filling f accordingly.

Helper function to solve the problem for both original and reversed arrays

Initialize a 2D array to store the minimum number of operations

Update the minimum value for the previous row

needed to make all elements equal to any value between 0 and 1000

for i, num in enumerate(arr, 1): # Enumerate from 1 for 1-based indexing

min_value = float('inf') # Initialize the minimum value to be infinity

Compute the minimum operations for current element to be j

Call the solve function on both the original and reversed array, and return the minimum result

return Math.min(computeMinimumOperations(nums), computeMinimumOperations(reverseArray(nums)));

decreasing or a non-increasing sequence. Python Solution

30 Java Solution

6

8

9

10

11

12

13

14

15

17

18

19

20

21

6

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

41

42

43

45

8

9

10

11

12

13

20

21

22

44 };

Typescript Solution

2 const MAX_VALUE: number = 1000;

nums.reverse();

27 # Usage

28 # sol = Solution()

class Solution {

13

14

16

17

18

19

20

22

23

24

25

26

```
22
 23
 24
             // Find the minimal operations among all possibilities for the last element
 25
             int answer = Integer.MAX_VALUE; // Initialize to max value
 26
             for (int cost : dp[n]) {
                 answer = Math.min(answer, cost); // Update the answer with the minimal cost
 27
 28
 29
 30
             return answer; // Return the final minimal cost
 31
 32
 33
         // Helper method to reverse the given array.
 34
         private int[] reverseArray(int[] nums) {
 35
             int[] copiedNums = nums.clone(); // Clone the array to avoid modifying the original one
 36
 37
             // Reverse the array in place
 38
             for (int start = 0, end = copiedNums.length - 1; start < end; ++start, --end) {</pre>
 39
                 // Swap elements
 40
                 int temp = copiedNums[start];
 41
                 copiedNums[start] = copiedNums[end];
 42
                 copiedNums[end] = temp;
 43
 44
 45
             return copiedNums; // Return the reversed array
 46
 47 }
 48
C++ Solution
  1 class Solution {
  2 public:
         // Function to find the minimum cost to make all elements of the array equal
         int convertArray(vector<int>& nums) {
             int cost1 = calculateMinimumCost(nums);
  5
```

14 // Return the minimum cost of the two possible scenarios return Math.min(cost1, cost2); 15 16 } 17 // Helper function to calculate the minimum cost of making all elements equal by incrementing or decrementing

```
23
         // hold minimum cost to make the first i elements equal to j
         let dp: number[][] = Array.from({length: n + 1}, () => Array(MAX_VALUE + 1).fill(0));
 24
 25
 26
         // Initialize the subproblem solutions for base cases here if necessary
 27
 28
         for (let i: number = 1; i <= n; ++i) {</pre>
             let minCostPrevious: number = Number.MAX_SAFE_INTEGER; // Initialize the minimum cost to a large value
 29
             for (let j: number = 0; j <= MAX_VALUE; ++j) {</pre>
 30
 31
                 // Update minCostPrevious to the minimum cost found in previous row
                 if (i > 1) {
 32
                     minCostPrevious = Math.min(minCostPrevious, dp[i - 1][j]);
 34
 35
 36
                 // Calculate cost of making nums[i-1] equal to j and update dp[i][j]
                 // with the sum of minCostPrevious and current cost
 37
                 dp[i][j] = minCostPrevious + Math.abs(nums[i - 1] - j);
 38
 39
 40
 41
         // Return the minimum cost of making all elements of the array equal until the last element
 42
 43
         return Math.min(...dp[n]);
 44
 45
    // Example usage:
 47 // let nums = [1, 2, 3]
 48 // let cost = convertArray(nums)
    // console.log(cost);
Time and Space Complexity
Time Complexity
The given function convertArray includes a nested function solve which computes the minimum cost to make all elements of the
array equal by only increments or only decrements.

    The outer function convertArray calls solve twice, once for the original array and once for the reversed array.

    Within solve, there is a nested loop. The outer loop runs n times, where n is the length of nums.

  • The inner loop runs for a constant 1001 times since it iterates over a preset range from 0 to 1000.
```

• Inside the inner loop, the computation is done in constant time, namely the minimum comparison mi > f[i - 1][j], the

Space Complexity

Putting it together, the time complexity of the solve function is 0(n * 1001), and since it is called twice, the overall time complexity of convertArray is 0(2 * n * 1001) which simplifies to 0(n) because constants are dropped in Big O notation and 1001 is a constant.

The space complexity is determined by the amount of memory used by the function relative to the input size:

minimum update mi = f[i - 1][j], and the cost calculation f[i][j] = mi + abs(x - j).

 The function solve uses a 2D array f of size (n + 1) * 1001, where n is the length of the input array nums. No other significant data structures are used that scale with the input size.

to solve.

The intuition behind the solution is to consider separately the scenarios where we convert the array to non-decreasing and nonincreasing orders. We will calculate the minimum operations required for both scenarios and then return the smaller result.

When looking for the minimum number of operations, it's important to keep in mind that we can use an operation on the same element multiple times if required.

The space complexity of the solve function is 0((n + 1) * 1001) which simplifies to 0(n), again dropping the constant 1001. Thus, the overall space complexity of convertArray is O(n) as only one instance of the 2D array f is maintained throughout the calls