1243. Array Transformation

Simulation

Problem Description

array of the previous day through a set of specific transformations. The rules for transforming any given array element are: 1. If an element is smaller than both its immediate neighbors (left and right), increase its value by 1.

The problem presents a scenario where you start with an initial integer array arr, and each day a new array is derived from the

2. If an element is larger than both its immediate neighbors, decrease its value by 1.

- 3. The first and last elements of the array remain unchanged regardless of their neighbors.
- This process is to be repeated day after day until eventually, the array reaches a state where no further changes occur in other
- words, the array becomes stable and does not change from one day to the next. The objective is to determine what this final array looks like.

Intuition

To solve this problem, we can simulate the array transformation process day by day until the array becomes stable. Here's how we can arrive at the solution approach:

1. Create a flag (f) to keep track of whether any change has occurred on a given day. 2. Iterate over the array, starting from the second element and ending at the second-to-last element (since the first and last elements do not change).

3. For each element, compare it with its left and right neighbors to decide whether to increment or decrement it according to the rules. If the current element is smaller than both neighbors, increment it.

- If the current element is larger than both neighbors, decrement it. 4. To avoid impacting the comparison of subsequent elements, store the original array in a temporary array (t) before making any changes.
- 5. Continue the day-by-day simulation until a day passes where no changes are made to the array, indicating the array has become stable. 6. When no changes occur (the flag f remains false after a full iteration), return the stable array.
- Solution Approach
- The solution approach follows a straightforward brute-force method to simulate the day-to-day transformation of the array until it reaches a state of equilibrium where no further changes occur. Here's how the implementation of the solution is carried out:

changes are made to the array in the current iteration (day). If no changes are made, f remains False and the loop ends.

arr[i] -= 1

f = True

f = True

At the start of each iteration, the current array (arr) is cloned into a temporary array (t). This is important because we want to evaluate the elements against their original neighbors, and modifying arr in-place would disturb those comparisons.

The solution uses a while loop flagged by a boolean f which is initially set to True. This flag f is used to check whether any

The implementation then iterates over all the elements of the array starting from index 1 and ending at len(arr) - 2 to

- ensure the first and last elements remain unchanged, in compliance with rule 3 of the problem description. During each iteration, the algorithm checks each element against its neighbors:
- ∘ If t[i] (the element in the temporary array) is greater than both t[i 1] and t[i + 1], the element in the original array arr[i] is decremented by 1, and f is set to True indicating a change.
- If t[i] is less than both t[i 1] and t[i + 1], arr[i] is incremented by 1, and again, f is set to True. if t[i] < t[i - 1] and t[i] < t[i + 1]: arr[i] += 1

```
If f is False after the inner loop, it means no changes were made in the latest day, so the array has reached its final stable
   state and the loop exits.
   The final stable array (arr) is then returned.
The solution does not use any complex algorithms or data structures; it is a simple iterative approach that exploits array indexing
```

Example Walkthrough

Let's say we have an initial integer array arr = [6, 4, 8, 2, 3]. We want to apply the solution approach to this array to find out

what the final stable array will look like after applying the transformations day after day as per the rules stated.

and conditional logic to simulate the situation described in the problem. It ensures the integrity of comparisons through the use of

Day 1:

• The array arr at the end of Day 1 is [6, 5, 7, 3, 3].

• We create a temporary array t which is a copy of arr: t = [6, 4, 8, 2, 3].

t[2] = 7, no change as it is not less than or greater than both neighbors.

Initialize a flag to track if there were any transformations.

Keep transforming the array until there are no changes.

Set the flag to False expecting no changes.

for i in range(1, len(temp_arr) - 1):

arr[i] -= 1

changed = True

changed = True

public List<Integer> transformArray(int[] arr) {

Return the transformed array.

boolean isTransforming = true;

isTransforming = false;

int[] tempArr = arr.clone();

--arr[i];

while (isTransforming) {

We iterate starting from the second element to the second to last:

if t[i] > t[i - 1] and t[i] > t[i + 1]:

 \circ t[1] = 4, it's less than both its neighbors t[0] = 6 and t[2] = 8, so arr[1] becomes 4 + 1 = 5. \circ t[2] = 8, it's greater than both t[1] = 4 (before increment) and t[3] = 2, so arr[2] becomes 8 - 1 = 7. \circ t[3] = 2, it's less than t[2] = 8 (before decrement) and t[4] = 3, so arr[3] becomes 2 + 1 = 3.

Day 2:

class Solution:

def transformArray(self, arr):

changed = True

while changed:

• We clone the array again: t = [6, 5, 7, 3, 3]. • Iterating through t: □ t[1] = 5, no change as it is not less than or greater than both neighbors.

No elements in arr changed during Day 2, so now we know that the array has become stable, and the process can end here. The

final array is [6, 5, 7, 3, 3]. This array will remain unchanged in subsequent days, as it meets none of the conditions for

incrementing or decrementing any of its elements (except for the first and last elements, which do not change anyway).

• t[3] = 3, no change as it is not less than or greater than both neighbors. • The array arr at the end of Day 2 is [6, 5, 7, 3, 3].

Therefore, our final stable array, after applying the given transformation rules, is [6, 5, 7, 3, 3].

If the current element is larger than its neighbours, decrement it.

if temp arr[i] > temp_arr[i - 1] and temp_arr[i] > temp_arr[i + 1]:

Set the flag to True to indicate a change has been made.

Set the flag to True to indicate a change has been made.

// Create a temporary copy of the array to hold the initial state before transformation

// Iterate through each element of the array, excluding the first and last elements

// If the current element is greater than both neighbors, decrement it

// Since a transformation occurred, flag it to continue the loop

if (tempArr[i] > tempArr[i - 1] && tempArr[i] > tempArr[i + 1]) {

// Flag to keep track of whether the array is still being transformed

// Initially assume no transformation will occur this cycle

// Continue looping until no more transformations occur

for (int i = 1; i < tempArr.length - 1; ++i) {

changed = true; // Mark change

// Function to transform the array according to given conditions

changed = false: // Reset flag for each iteration

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

changed = true; // Mark change

changed = true; // Mark change

let changed = true; // Flag to keep track of any changes in the array

// Process each element of the array except the first and the last

// If current element is greater than both neighbors, decrease it by 1

// Else if current element is less than both neighbors, increase it by 1

if (clonedArray[i] > clonedArray[i - 1] && clonedArray[i] > clonedArray[i + 1]) {

else if (clonedArray[i] < clonedArray[i - 1] && clonedArray[i] < clonedArray[i + 1]) {</pre>

const clonedArray = [...arr]; // Clone the current state of the array by spreading in a new array

// Return the transformed array

function transformArray(arr: number[]): number[] {

// Loop until no more changes are made

arr[i]--:

arr[i]++;

return arr;

while (changed) {

TypeScript

isTransforming = true;

a temporary array and signals the completion of the simulation using a loop control variable.

Solution Implementation **Python**

changed = False # Create a copy of the array to hold the initial state. temp_arr = arr.copy() # Iterate over the elements of the array except the first and last.

If the current element is smaller than its neighbours, increment it. if temp arr[i] < temp_arr[i - 1] and temp_arr[i] < temp_arr[i + 1]:</pre> arr[i] += 1

return arr

class Solution {

```
Java
```

```
// If the current element is less than both neighbors, increment it
                if (tempArr[i] < tempArr[i - 1] && tempArr[i] < tempArr[i + 1]) {</pre>
                    ++arr[i];
                    // Since a transformation occurred, flag it to continue the loop
                    isTransforming = true;
        // Convert the transformed array to a list of integers
        List<Integer> resultList = new ArrayList<>();
        for (int item : arr) {
            resultList.add(item);
        // Return the final transformed list
        return resultList;
C++
#include <vector>
using namespace std;
class Solution {
public:
    // Function to transform the array according to given conditions
    vector<int> transformArray(vector<int>& arr) {
        bool changed = true; // Flag to keep track of any changes in the array
        // Loop until no more changes are made
        while (changed) {
            changed = false: // Reset flag for each iteration
            vector<int> clonedArray = arr; // Clone the current state of the array
            // Process each element of the array except the first and the last
            for (int i = 1; i < arr.size() - 1; ++i) {
                // If current element is greater than both neighbors, decrease it by 1
                if (clonedArray[i] > clonedArray[i - 1] && clonedArray[i] > clonedArray[i + 1]) {
                    --arr[i];
                    changed = true; // Mark change
                // If current element is less than both neighbors, increase it by 1
                if (clonedArray[i] < clonedArray[i - 1] && clonedArray[i] < clonedArray[i + 1]) {</pre>
                    ++arr[i]:
```

```
// Return the transformed array
   return arr;
class Solution:
   def transformArray(self, arr):
       # Initialize a flag to track if there were any transformations.
       changed = True
       # Keep transforming the array until there are no changes.
       while changed:
           # Set the flag to False expecting no changes.
           changed = False
           # Create a copy of the array to hold the initial state.
            temp_arr = arr.copy()
           # Iterate over the elements of the array except the first and last.
            for i in range(1, len(temp_arr) - 1):
               # If the current element is larger than its neighbours, decrement it.
               if temp arr[i] > temp_arr[i - 1] and temp_arr[i] > temp_arr[i + 1]:
                   arr[i] -= 1
                   # Set the flag to True to indicate a change has been made.
                    changed = True
               # If the current element is smaller than its neighbours, increment it.
               if temp arr[i] < temp_arr[i - 1] and temp_arr[i] < temp_arr[i + 1]:</pre>
                   arr[i] += 1
                   # Set the flag to True to indicate a change has been made.
                   changed = True
       # Return the transformed array.
       return arr
```

• The outer while loop continues executing until no more changes are made to the array. In the worst case scenario, it could run for a number of iterations proportional to the size of the array, n. This is because each iteration could potentially only decrease or increase each element by 1, and for the array to become stable, it might require multiple single-unit adjustments at different positions.

Time Complexity

Time and Space Complexity

• The inner for loop goes through the elements of the array, starting from 1 to len(arr) - 2. This for loop has a time complexity of 0(n - 2), which simplifies to O(n).

The time complexity of the given code is primarily dependent on two nested loops: the outer while loop and the inner for loop.

loop, a full pass through most of the array is completed using the for loop. **Space Complexity**

Therefore, in the worst case, the time complexity of the entire algorithm becomes 0(n^2) since for each iteration of the while

- The space complexity of the code consists of:
- A temporary array t that is a copy of the input array arr. This copy is made in each iteration of the while loop. The temporary array t has the same size as the input array, which gives us a space complexity of O(n).
- No other significant extra space is used, as the operations are performed in place with just a few extra variables (f, i) whose space usage is negligible. So, the total space complexity of the algorithm is O(n).