2078. Two Furthest Houses With Different Colors

Easy Greedy Array

Leetcode Link

Problem Description

The problem provides a street lined with `n` evenly spaced houses, each with a color represented numerically. An array `colors` is given, where `colors[i]` depicts the color of the `i-th` house. The task is to determine the maximum distance between any two houses that have different colors.

This distance is measured in terms of the array indices, thus the distance between the `i-th` and `j-th` houses is `abs(i - j)`, which is the absolute value of the difference between their indices.

The key to this problem is identifying the farthest pair of houses with dissimilar colors, which will give us the largest possible index difference.

Intuition

The solution utilizes the fact that to maximize the distance between two differently colored houses, we should consider houses at the ends of the street.

- First, we check if the first and last houses (`colors[0]` and `colors[-1]`) are different. If they are, this is the maximum possible distance (`n - 1`) and we can return this value immediately.
- If the colors of the first and last house are the same, we then need to check for the next closest house to each of them that has a different color to calculate the maximum distance.
- We initialize two pointers, `i` from the start (excluding the first house) and `j` from the end (excluding the last house), and move each pointer towards the center of the array until a house with a different color is found.
- While `colors[i]` is the same as the first house's color, we increment `i`. Similarly, we decrement `j` while `colors[j]` is the same as the first house's color. This process skips over houses with the same color as the first house.
- After finding the differently colored houses, we calculate the distance from them to the opposite ends of the street, specifically `n - 1 - i` and `j`.
- The maximum distance is the larger of these two values, ensuring we have the greatest possible range between two houses of different colors, as required.

This approach ensures we find the optimal distance without checking every possible pair, which would be less efficient for large arrays.

Solution Approach

The implementation of the solution leverages a straightforward approach without the use of complex algorithms or data structures. Instead, it utilizes a simple iterative process with two pointers and basic conditional logic.

- Firstly, the length of the `colors` list, `n`, is obtained.
- The colors of the first (`colors[0]`) and last (`colors[-1]`) houses are compared. If they are different, the solution is immediately found because the maximum distance in a linear array is between the endpoints (indexes `0` and `n - 1`). Thus, the code returns `n - 1`.

```
1 if colors[0] != colors[-1]:
2     return n - 1
```

- If the colors are the same, two pointers `i` and `j` are introduced. The pointer `i` is initialized to start from the second house (`index 1`) and moves towards the end of the street, while `j` is initialized to start from the second to last house (`index n - 2`) and moves towards the beginning of the street.

```
1 i, j = 1, n - 2
```

- The while loops are used to advance `i` and `j`. As long as `colors[i]` matches the color of the first house, `i` is incremented. And as long as `colors[j]` matches the color of the first house, `j` is decremented. These loops stop at the first occurrences of colors different from the first house's color.

```
1 while colors[i] == colors[0]:
2     i += 1
3 while colors[j] == colors[0]:
4     j -= 1
```

- After exiting the loops, `i` is at the position of the first house from the left with a different color than the first house, and `j` is at the position of the first house from the right with a different color than the first house. The distances from these points to the respective ends of the street are calculated:

```
1 return max(n - i - 1, j)
```

  The function returns the greater of these two distances. The subtraction of `1` in `n - i - 1` accounts for the zero-based indexing. By taking the maximum of these two distances, we ensure that we are indeed returning the maximum possible distance between two houses with different colors.

This solution approach does not require sorting or searching algorithms, nor any particular data structure other than the array itself. It efficiently traverses the array at most twice, resulting in a time complexity of O(n). This efficient method bypasses the need to compare every pair of houses, thereby preventing the solution from becoming inefficient, especially for large arrays.

Example Walkthrough

Let's consider a small example where the `colors` array has 7 houses with the following colors represented by the numbers: `1 1 1 4 5 1 1`. The length of this array, `n`, is 7.

1. We start by comparing the `colors[0]`, which is 1 (the color of the first house), with `colors[-1]`, which is also 1 (the color of the seventh house). Since these colors are the same, we cannot conclude at this point and need to examine further.
2. Set up two pointers:
   - Pointer `i` starts from index 1 (the second house).
   - Pointer `j` starts from index 5 (the second to last house, or `n - 2`).
3. Conduct a while loop to move `i` towards the right until a house with a different color is found.
   - Loop continues until `colors[i]` differs from `colors[0]`. The while loop would iterate as follows:
     - `colors[1]` is 1, same as first house, increment `i` (now `i` is 2).
     - `colors[2]` is 1, still the same, increment `i` (now `i` is 3).
     - `colors[3]` is 4, which is different! Stop here.
4. Conduct another while loop to move `j` towards the left until a house with a different color is found (relative to the color of the first house).
   - Loop continues until `colors[j]` differs from `colors[0]`. The while loop would iterate as follows:
     - `colors[5]` is 1, same as the first house, decrement `j` (now `j` is 4).
     - `colors[4]` is 5, which is different! Stop here.
5. After the loops, we have `i` at index 3 and `j` at index 4. Now we calculate the distances from these houses to the ends of the street:
   - Distance from the house at `i` to the end of the street: `n - i - 1` → `7 - 3 - 1` = 3
   - Distance from the house at `j` to the beginning of the street: `j` → 4
6. The maximum of the two distances is 4, which occurs between the house with color 5 (at index 4) and the first house (at index 0).

The final answer is the maximum distance of 4, which is the maximum distance between any two houses with different colors in the given `colors` array.

Python Solution

```
1  from typing import List
2
3  class Solution:
4      def maxDistance(self, colors: List[int]) -> int:
5          # Get the number of elements in the 'colors' list
6          num_colors = len(colors)
7
8          # If the first and last colors are different, the maximum distance is the length of the list minus one
9          if colors[0] != colors[-1]:
10             return num_colors - 1
11
12         # Initialize two pointers, one starting from the beginning and the other from the end of the list
13         # These pointers will be used to find the maximum distance on both ends of the list
14         left_index, right_index = 1, num_colors - 2
15
16         # Move the left pointer towards the right as long as the color is the same as the first color
17         while colors[left_index] == colors[0]:
18             left_index += 1
19
20         # Move the right pointer towards the left as long as the color is the same as the first color
21         while colors[right_index] == colors[0]:
22             right_index -= 1
23
24         # Calculate the distance from the altered left and right pointers to the corresponding ends
25         # Take the maximum of these two distances to get the farthest distance with different colors
26         return max(num_colors - left_index - 1, right_index)
27
```

Java Solution

```
1  class Solution {
2
3      public int maxDistance(int[] colors) {
4          // Length of the colors array.
5          int arrayLength = colors.length;
6
7          // If the first and last color are different, the maximum distance
8          // is the length of the array minus 1.
9          if (colors[0] != colors[arrayLength - 1]) {
10             return arrayLength - 1;
11         }
12
13         // Initialize indices 'left' and 'right' to the start and end of the array respectively.
14         int left = 0;
15         int right = arrayLength - 1;
16
17         // Increment 'left' index until we find a different color from the left end.
18         while (colors[++left] == colors[0]); // Empty loop body; iteration is done in the 'while' expression.
19
20         // Decrement 'right' index until we find a different color from the right end.
21         while (colors[--right] == colors[0]); // Empty loop body; iteration is done in the 'while' expression.
22
23         // Calculate the distance from the left-most different color to the right end,
24         // and from the left end to the right-most different color.
25         int leftDistance = arrayLength - left - 1;
26         int rightDistance = right;
27
28         // The maximum distance is the larger of 'leftDistance' and 'rightDistance'.
29         return Math.max(leftDistance, rightDistance);
30     }
31 }
32
```

C++ Solution

```
1  class Solution {
2  public:
3      int maxDistance(vector<int>& colors) {
4          int sizeOfColors = colors.size(); // The size of the colors vector
5          // Directly return the maximum possible distance if the first and last colors are different
6          if (colors[0] != colors[sizeOfColors - 1]) {
7              return sizeOfColors - 1;
8          }
9
10         int leftIndex = 0; // Initialize a left pointer at the start of the vector
11         int rightIndex = sizeOfColors; // Initialize a right pointer at the end of the vector
12
13         // Move the left pointer towards the right until we find a color different from the first color
14         while (colors[++leftIndex] == colors[0]);
15
16         // Move the right pointer towards the left until we find a color different from the first color
17         while (colors[--rightIndex] == colors[0]);
18
19         // Calculate the maximum distance possible from either ends
20         // by comparing the distance from the right-most different color to the beginning and
21         // the distance from the left-most different color to the end
22         return max(sizeOfColors - leftIndex - 1, rightIndex);
23     }
24 };
25
```

Typescript Solution

```
1  function maxDistance(colors: number[]): number {
2      const sizeOfColors = colors.length; // The size of the colors array
3
4      // Directly return the maximum possible distance if the first and last colors are different
5      if (colors[0] !== colors[sizeOfColors - 1]) {
6          return sizeOfColors - 1;
7      }
8
9      let leftIndex = 0; // Initialize a left pointer at the start of the array
10     let rightIndex = sizeOfColors - 1; // Initialize a right pointer at the end of the array
11
12     // Move the left pointer towards the right until we find a color different from the first color
13     while (colors[++leftIndex] === colors[0]);
14
15     // Move the right pointer towards the left until we find a color different from the last color
16     while (colors[--rightIndex] === colors[0]);
17
18     // Calculate the maximum distance possible from either end by comparing the distance from the right-most different color to the b
19     return Math.max(sizeOfColors - leftIndex - 1, rightIndex);
20 }
21
22 // Example usage:
23 // const result = maxDistance([1, 1, 2, 1, 1]);
24 // console.log(result); // This should output the maximum distance between two distinct colors.
25
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed based on the linear iterations it performs.

1. Initial check for the first and last element: `O(1)` - as it is a direct comparison.
2. Two while loops to find the first index `i` from the start that has a different color and the first index `j` from the end that has a different color. Both of these while loops run in `O(n)` time in the worst case, where `n` is the length of the colors list. However, each loop runs at most once per element and they do not iterate over the same elements multiple times.

The overall time complexity is dominated by the two while loops, which results in `O(n)` (where `n` is the number of elements in `colors`).

Space Complexity

The space complexity of the code is analyzed by looking at the extra space required apart from the input.

1. Constants `n`, `i`, and `j` are used for iterating and storing indices: `O(1)` - constant space complexity as it does not depend on the input size.
2. No additional data structures or recursive calls that use additional space proportional to the input size.

Therefore, the space complexity of the given code is `O(1)` (constant space complexity).