2808. Minimum Seconds to Equalize a Circular Array

Medium Greedy Array Hash Table Leetcode Link

Problem Description

In this problem, we are given a 0-indexed array `nums` of `n` integers. The task is to make all elements in the array equal by performing a specific operation repeatedly. During each second, for every element at index `i`, you can update `nums[i]` to be equal to its current value, the value of the previous element `(nums[(i - 1 + n) % n])`, or the value of the next element `(nums[(i + 1) % n])`. The modulo operation ensures that you are wrapping around the array when reaching the ends, which means it actually forms a loop or ring. The goal is to find the **minimum number of seconds** needed to make all the elements in the array equal.

Intuition

The key to solving this problem lies in understanding that we can always make the entire array equal to any one of its current values. This is because in each operation, you are allowed to choose the previous or next element's value, which can eventually spread any number's occurrence across the array.

Since we want to minimize the number of seconds, the best strategy would be to choose the value that will take the least amount of time to spread through the array. Intuitively, if we have clusters of the same number occurring together, we would prefer to choose one such cluster's value and spread it to the rest of the array.

The solution involves the following steps:

- We first group indices of identical elements into lists, using a dictionary where the keys are the array's values and the values are lists of indices where these numbers occur.
- Then, for each group of identical elements:
  - We calculate the distance between the first and last occurrence of the value, taking into account the wrap-around using `(idx[0] + n - idx[-1])`.
  - We also calculate the maximum distance between any two consecutive occurrences, as this will represent the maximum time required to change the value between these two points using `max(t, j - i)` for every pair of consecutive indices in the group.
  - The time needed to make the entire array equal to this value is half the maximum distance found, since the value can spread from both ends of a series.
- The minimum time across all such values is the final answer.

By applying this approach, we ensure that we pick the value that will take the least amount of time to replicate across the entire array.

Solution Approach

The solution provided uses Python's `defaultdict` to categorize indices of identical elements into lists, and the `inf` constant from the `math` module as a representation of infinity, which is employed to find the minimum value as we compare different distances.

Here is the step-by-step breakdown of the implementation:

- A `defaultdict` of lists is instantiated. It will map each unique value in `nums` to a list of indices where it occurs. This is achieved by enumerating over `nums` and appending the index `i` to the list of `d[x]`, where `x` is the value at index `i`.
- A variable `ans` is initialized to infinity (`inf`). This will hold the minimum number of seconds required to make all elements of the array equal.
- The algorithm then iterates over the values of the dictionary, which are the lists of indices for each distinct number in the array.
  - For each list of indices (`idx`), it calculates `t`, the distance considering wrap-around between the first and last occurrence: `idx[0] + n - idx[-1]`.
  - It then proceeds to find the maximum distance between any two consecutive occurrences of the same number within the list. This is done using Python's `pairwise` function (Python 3.10+). If `pairwise` is not available, a simple zip like `zip(idx, idx[1:])` can be used to achieve similar results.
  - For every pair `(i, j)` of consecutive indices in `idx`, `t` is updated to the maximum of its current value and the distance between the consecutive indices `j - i`.
  - Finally, because the value can spread from both ends towards the middle, only half this time is needed to make all elements between `i` and `j` equal, hence `t // 2`.
- The algorithm updates `ans` with the minimum between its current value and `t // 2`. Since `ans` is initialized to infinity and we are finding the minimum over all iterations, `ans` will hold the minimum time needed to make all elements equal after examining all distinct numbers.
- The function returns the value stored in `ans`.

This approach effectively breaks down a seemingly complex problem into a series of calculations based on the distribution of values across the array, using dictionary and list structures to organize data and a simple loop to compute the minimum time.

Example Walkthrough

Let's illustrate the solution approach using a simple example:

Given the array `nums = [1, 2, 3, 2, 1]`, which is 0-indexed:

Step 1: We create a dictionary of list indices for each unique value in `nums`. In our case:

- The value `1` occurs at indices `[0, 4]`.
- The value `2` occurs at indices `[1, 3]`.
- The value `3` occurs at index `[2]`.

Step 2: Initialize `ans` to infinity.

Step 3: Evaluate each group of identical elements:

- For value `1`, the list of indices is `[0, 4]`. Since the array forms a loop:
  - The distance considering wrap-around is `(0 + 5 - 4) % 5 = 1`.
  - There are no consecutive occurrences to calculate the maximum distance, so the maximum distance remains `1`.
  - The time needed is `1 // 2 = 0` (since we can start from both ends, it needs no time to convert values in between).
- For value `2`, the indices are `[1, 3]`. No wrap-around is needed.
  - The maximum distance between consecutive occurrences is `(3 - 1) = 2`.
  - The time needed would be `2 // 2 = 1`.
- For value `3`, there is only one occurrence, no distance to calculate between indices.

Step 4: Find the minimum `ans`:

- For value `1`, `ans` becomes the minimum of `infinity` and `0`, which is `0`.
- For value `2`, `ans` is the minimum of `0` and `1`, which remains `0`.
- For value `3`, no change as there is only one occurrence.

Step 5: Return the value in `ans`, which is `0`.

This would mean that no time is needed to make all the elements equal to `1` in this example, since we theoretically start spreading the value from both ends of the occurrences.

Python Solution

```python
from collections import defaultdict

class Solution:
    def minimum_seconds(self, nums: list) -> int:
        # Dictionary to store the indices of each unique number in nums
        index_mapping = defaultdict(list)

        # Populate index_mapping with positions for every number in nums
        for index, value in enumerate(nums):
            index_mapping[value].append(index)

        # Initialize the minimum seconds to infinity
        minimum_seconds = float('inf')
        n = len(nums)

        # Iterate over the indices for each unique number
        for indices in index_mapping.values():
            # Time spent is the distance between first and last occurrence
            time_spent = indices[0] + n - indices[-1]

            # Iterate over pairs of indices to find max distance in between
            for i in range(len(indices) - 1):
                # Find distance between consecutive occurrences
                pair_time = indices[i + 1] - indices[i]
                # Update time_spent with the maximum gap found so far
                time_spent = max(time_spent, pair_time)

            # Calculate the minimum seconds required (halve the max distance)
            # and compare with minimum found so far
            minimum_seconds = min(minimum_seconds, time_spent // 2)

        # Return the minimum seconds required to process all unique numbers
        return minimum_seconds

```

Java Solution

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

class Solution {
    public int minimumSeconds(List<Integer> nums) {
        // Create a map to hold lists of indices for each unique number in 'nums'
        Map<Integer, List<Integer>> indicesMap = new HashMap<>();
        int n = nums.size(); // Total number of elements in the list

        // Populate the map with lists of indices for each number
        for (int i = 0; i < n; ++i) {
            indicesMap.computeIfAbsent(nums.get(i), k -> new ArrayList<>()).add(i);
        }

        int minSeconds = Integer.MAX_VALUE; // Initialize the minimum seconds to the highest possible value

        // Iterate over the map values, which are lists of indices
        for (List<Integer> indices : indicesMap.values()) {
            int m = indices.size(); // Total number of indices in the current list
            int timeDiff = indices.get(0) + n - indices.get(m - 1);
            // Calculate the initial time difference
            // as the distance from the first to the last occurrence

            // Update the time difference to be the maximum gap between any two consecutive occurrences
            for (int i = 1; i < m; ++i) {
                timeDiff = Math.max(timeDiff, indices.get(i) - indices.get(i - 1));
            }

            // Update the minimum time by comparing with the current calculated time
            minSeconds = Math.min(minSeconds, timeDiff / 2);
        }

        return minSeconds; // Return the minimum number of seconds
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <algorithm>
using namespace std;

class Solution {
public:
    int minimumSeconds(vector<int>& nums) {
        // Create a mapping from each unique number to its indices in the array
        unordered_map<int, vector<int>> indicesMap;
        int n = nums.size(); // get the size of the input vector
        for (int i = 0; i < n; ++i) {
            indicesMap[nums[i]].push_back(i); // map numbers to their indices
        }

        // Initialize the minimum number of seconds to a large value
        int minSeconds = INT_MAX; // use INT_MAX as shorthand for 1 << 30

        // Iterate over the number-index mapping
        for (auto& kv : indicesMap) { // Use 'kv' to represent key-value pairs
            vector<int>& idx = kv.second; // Get the vector of indices
            int m = idx.size(); // Size of the index list
            // Compute initial distance considering the array as circular
            int maxDistance = idx[0] + n - idx[m - 1];
            // Loop over the indices to find the largest distance between any two consecutive indices
            for (int i = 1; i < m; ++i) {
                maxDistance = max(maxDistance, idx[i] - idx[i - 1]);
            }
            // Update the minimum number of seconds with the lower value
            minSeconds = min(minSeconds, maxDistance / 2);
        }

        // Return the minimum number of seconds after completing the loop
        return minSeconds;
    }
};

```

Typescript Solution

```typescript
/**
 * Computes the minimum seconds needed to cover all numbers by a segment of continuous numbers in the list nums.
 * @param nums array of numbers representing different values where we search for the minimum segment.
 * @returns the minimum number of seconds required.
 */
function minimumSeconds(nums: number[]): number {
    // Initializes a map to hold arrays of indices for each unique number
    const indexMap: Map<number, number[]> = new Map();
    const length = nums.length;

    // Populates the indexMap with the indices of occurrences of each number
    for (let i = 0; i < length; ++i) {
        if (!indexMap.has(nums[i])) {
            indexMap.set(nums[i], []);
        }
        indexMap.get(nums[i])!.push(i);
    }

    // Variable to keep track of the minimum seconds needed
    let minSeconds = 1 << 30; // Large initial value

    // Iterates through each set of indices in the map
    for (const [number, indices] of indexMap) {
        const indicesLength = indices.length;

        // Calculates the initial time as the time to cover from the first to the last occurrences
        let currentTime = indices[0] + length - indices[indicesLength - 1];

        // Updates the currentTime based on the maximum gap between consecutive indices
        for (let i = 1; i < indicesLength; ++i) {
            currentTime = Math.max(currentTime, indices[i] - indices[i - 1]);
        }

        // Updates the minimum time
        minSeconds = Math.min(minSeconds, currentTime >> 1);
    }

    // Returns the minimum seconds calculated
    return minSeconds;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by several factors:

- The loop that creates the dictionary `d`, which has a time complexity of `O(n)` since it goes through all the elements of `nums` once.
- The loop that goes through `d.values()` which can potentially iterate through all elements again in the worst case. If all the elements in `nums` are unique, this would again take `O(n)` time.
- Inside the second loop, there's a nested call to `pairwise(idx)`. The `pairwise` function itself has `O(k)` complexity, where `k` is the length of the list `idx` passed to it. In the worst case, where `nums` has many repeated elements, this nested loop could have `O(n)` complexity if all elements are the same.

Given that `pairwise` is called inside the loop for every key in the dictionary `d`, the overall complexity of these nested loops depends on the distribution of the numbers in the `nums` list. The worst-case scenario happens when all elements are the same, leading to a complexity of `O(n)` for the iterations throughout `d.values()`, compounded with the complexity of `pairwise`, leading to a worst-case time complexity of `O(n^2)`.

Therefore, the overall worst-case time complexity of the code is `O(n^2)`.

Space Complexity

The space complexity can be analyzed as follows:

- The dictionary `d` that stores the indices of each element can potentially store `n` keys with a list of indices as values. In the worst case where all numbers are the same, the list of indices would also contain `n` values. Therefore, the worst-case space complexity for `d` is `O(n)`.
- The space used by variables `ans`, `t`, `idx`, `i`, and `j` is constant, hence `O(1)`.

Taking the above points into consideration, the total space complexity is `O(n)` for the dictionary storage.

In conclusion, the code has a time complexity of `O(n^2)` and a space complexity of `O(n)`.