1054. Distant Barcodes

Problem Description

In this problem, we are given a list of integers representing barcodes. The goal is to rearrange the barcodes so that no two consecutive barcodes are the same. Fortunately, the problem statement assures us that there is always a valid rearrangement that meets this condition. We need to find and return one such valid arrangement.

Intuition

To solve this problem, we should first think about the constraints: no two adjacent barcodes can be equal. A natural approach to prevent adjacent occurrences of the same barcode would be to arrange the barcodes in such a way that the most frequent ones are as spread out as possible. This way, we reduce the chance that they will end up next to each other.

We can do this by:

1. Counting the frequency of each barcode using `Counter` from the `collections` module, which gives us a dictionary-like object mapping each unique barcode to its count of appearances.
2. Sorting the barcodes based on their frequency. However, simply sorting them in non-ascending order of their frequency is not enough, as we need to make sure that once they are arranged, the condition is met. It's a good idea to also sort them by their value to have a deterministic result if frequencies are equal.
3. Once sorted, the array is split into two: the first half will occupy even positions, and the second half will fill the odd positions in the result array. We do this because even-indexed places followed by odd-indexed ones are naturally never adjacent, which is ideal for our requirement to space out the frequent barcodes.
4. Create a new array for the answer, filling it first with the elements at even indices and then at odd indices.

By doing this, we ensure that the most frequent elements are spread across half the array length, minimizing any chance they are adjacent to each other. The end result is a carefully structured sequence that satisfies the given condition of the problem.

Solution Approach

The solution approach follows a series of logical steps that strategically utilize Python language features and its standard library functionalities. Here's a breakdown of how the solution is implemented:

1. **Counting Frequencies**: We use the `Counter` class from the `collections` module to count the frequency of each barcode in the given list. The `Counter` object, named `cnt`, will give us a mapping of each unique barcode value to its count of appearances.

```
cnt = Counter(barcodes)
```

2. **Sorting Based on Frequency**: We then sort the original list of barcodes using the `sort` method, applying a custom sorting key. The sorting key is a lambda function that sorts primarily by frequency (`-cnt[x]`, where negative is used for descending order) and secondarily by the barcode value (`x`) in natural ascending order in case of frequency ties.

```
barcodes.sort(key=lambda x: (-cnt[x], x))
```

3. **Structuring The Answer**: With the barcodes now sorted, half of the array (rounded up in the case of odd-length arrays) can be placed at even indices and the remaining half at odd indices in a new array named `ans`. This is to ensure that we space out high-frequency barcodes, minimizing the risk of identical adjacent barcodes:

   - We first calculate the size of the array `n` and then create the answer array `ans` of that same size filled with zeros.
   - The clever part comes with Python's slice assignment:
     - `ans[::2]` is a way to refer to every second element of `ans`, starting from the first. We assign the first half of the `barcodes` to these positions.
     - `ans[1::2]` refers to every second element starting from the second, to which we assign the latter half of the `barcodes`.

   Here's how the code accomplishes that:

```
n = len(barcodes)
ans = [0] * n
ans[::2] = barcodes[: (n + 1) // 2]
ans[1::2] = barcodes[(n + 1) // 2 :]
```

4. **Returning the Result**: The last line of the function simply returns the `ans` array, which is now a rearranged list of barcodes where no two adjacent barcodes are equal.

By using this approach, the algorithm effectively distributes barcodes of high frequency throughout the arrangement to satisfy the conditions of not having identical adjacent barcodes.

```
class Solution:
    def rearrangeBarcodes(self, barcodes: List[int]) -> List[int]:
        cnt = Counter(barcodes)
        barcodes.sort(key=lambda x: (-cnt[x], x))
        n = len(barcodes)
        ans = [0] * n
        ans[::2] = barcodes[: (n + 1) // 2]
        ans[1::2] = barcodes[(n + 1) // 2 :]
        return ans
```

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we're given the following list of barcodes:

```
barcodes = [1, 1, 1, 2, 2, 3]
```

Following the steps of the solution approach:

1. **Counting Frequencies**: We use `Counter` to determine the number of occurrences of each barcode. In this example:

```
cnt = Counter([1, 1, 1, 2, 2, 3])  # Results in Counter({1: 3, 2: 2, 3: 1})
```

2. **Sorting Based on Frequency**: Using the sorting key in `sort`, we arrange the barcodes first by decreasing frequency and then by their natural value:

```
sorted_barcodes = [1, 1, 1, 2, 2, 3]  # 1 is most common, then 2, and 3 is least common
```

3. **Structuring The Answer**: The sorted list is now to be split across even and odd indices in a new array. The length of the barcodes list `n` is 6. The first half of the array (which is the first 3 elements, as `(6 + 1) // 2` is 3) is placed at even indices:

```
ans = [0, 0, 0, 0, 0, 0]
ans[::2] = [1, 1, 1]
```

And the latter half of the array (which is the last 3 elements) is placed at odd indices:

```
ans[1::2] = [2, 2, 3]
```

So, `ans` becomes:

```
ans = [1, 2, 1, 2, 1, 3]
```

4. **Returning the Result**: The final `ans` array is returned as the rearranged list of barcodes:

```
return [1, 2, 1, 2, 1, 3]
```

This rearranged array adheres to the condition that no two consecutive barcodes are the same. In this example, barcode 1, which occurred most frequently, is well spread out, followed by barcode 2, and barcode 3 is placed at the end to satisfy the condition.

Python Solution

```
from collections import Counter

class Solution:
    def rearrange_barcodes(self, barcodes: List[int]) -> List[int]:
        # Count the frequency of each barcode
        barcode_counts = Counter(barcodes)

        # Sort barcodes by decreasing frequency and then by value
        # This is to prepare for the rearrangement ensuring no adjacent barcodes are same
        barcodes.sort(key=lambda x: (-barcode_counts[x], x))

        # Get the total number of barcodes
        total = len(barcodes)

        # Create a placeholder list for the answer
        rearranged = [0] * total

        # Assign barcodes to even indices first (0, 2, 4, ...)
        # We take the first half of the sorted barcodes
        rearranged[::2] = barcodes[: (total + 1) // 2]

        # Then assign barcodes to the odd indices (1, 3, 5, ...)
        # We take the second half of the sorted barcodes
        rearranged[1::2] = barcodes[(total + 1) // 2 :]

        # Return the final arrangement of barcodes
        return rearranged
```

Java Solution

```
import java.util.Arrays; // Required for sorting the array

public class Solution {
    public int[] rearrangeBarcodes(int[] barcodes) {
        // Determine the length of the barcode array
        int length = barcodes.length;
        // Use wrapper class Integer for custom sorting
        Integer[] boxedBarcodes = new Integer[length];
        // Variable to keep track of the maximum barcode value
        int maxBarcode = 0;

        // Copy barcodes to the temporary Integer array and find max value
        for (int i = 0; i < length; ++i) {
            boxedBarcodes[i] = barcodes[i];
            maxBarcode = Math.max(maxBarcode, barcodes[i]);
        }

        // Create and populate a count array for barcode frequencies
        int[] count = new int[maxBarcode + 1];
        for (int barcode : barcodes) {
            ++count[barcode];
        }

        // Custom sort the array based on frequency (and then by value if frequencies are equal)
        Arrays.sort(boxedBarcodes, (a, b) -> count[b] != count[a] ? a - b : count[b] - count[a]);

        // Create an array to hold the final rearranged barcodes
        int[] rearranged = new int[length];

        // Use two passes to distribute the barcodes ensuring no two adjacent barcodes are same
        // Fill even indices first, then odd indices
        for (int pass = 0, index = 0; pass < 2; ++pass) {
            for (int i = pass; i < length; i += 2) {
                rearranged[i] = boxedBarcodes[index++];
            }
        }

        // Return the rearranged barcodes
        return rearranged;
    }
}
```

C++ Solution

```
#include <vector>
#include <algorithm>
#include <string>

class Solution {
public:
    // Rearranges barcodes in such a way that no two equal barcodes are adjacent
    std::vector<int> rearrangeBarcodes(std::vector<int>& barcodes) {
        // Find the highest value in barcodes to create an array large enough
        int maxBarcodeValue = *std::max_element(barcodes.begin(), barcodes.end());

        // Create and initialize count array
        int count[maxBarcodeValue + 1];
        std::memset(count, 0, sizeof(count));

        // Count the occurrences of each barcode
        for (int barcode : barcodes) {
            ++count[barcode];
        }

        // Sort the barcodes based on the frequency of each barcode (descending),
        // and if frequencies are equal, sort by the barcode value (ascending)
        std::sort(barcodes.begin(), barcodes.end(), [&](int a, int b) {
            return count[a] > count[b] || (count[a] == count[b] && a < b);
        });

        int n = barcodes.size();
        std::vector<int> result(n);

        // Distribute the most frequent barcodes first, filling even positions,
        // then the rest in odd positions
        int index = 0;
        for (int i = 0; i < n; i += 2) {
            result[i] = barcodes[index++];
        }
        for (int i = 1; i < n; i += 2) {
            result[i] = barcodes[index++];
        }

        // Return the result vector with no two equal barcodes adjacent
        return result;
    }
};
```

Typescript Solution

```
function rearrangeBarcodes(barcodes: number[]): number[] {
    // Find the maximum number value in the barcodes array
    const maxBarcodeValue = Math.max(...barcodes);

    // Initialize a count array with a length of the maximum number + 1, filled with zeroes
    const count = Array(maxBarcodeValue + 1).fill(0);

    // Count the occurrences of each barcode
    for (const barcode of barcodes) {
        count[barcode]++;
    }

    // Sort the barcodes array based on the frequency of each number and then by the number itself if frequencies are equal
    barcodes.sort((a, b) => (count[a] === count[b] ? a - b : count[b] - count[a]));

    // The length of the barcodes array
    const totalBarcodes = barcodes.length;

    // Initialize the answer array that will be rearranged
    const rearranged = Array(totalBarcodes);

    // Loop through the sorted barcodes to rearrange them such that no two equal barcodes are adjacent
    let insertionIndex = 0;
    for (let start = 0; start < 2; ++start) {
        for (let i = start; i < totalBarcodes; i += 2) {
            rearranged[i] = barcodes[insertionIndex];
            insertionIndex++;
        }
    }

    // Return the rearranged barcodes
    return rearranged;
}
```

Time and Space Complexity

Time Complexity

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

1. Counting elements with `Counter(barcodes)` takes $O(n)$ time, where n is the number of elements in the `barcodes` list.
2. Sorting the `barcodes` list with the custom sort key based on their count and value takes $O(n \log n)$ time.
3. Slicing the list into two parts (for odd and even positions) is done in $O(n)$ time.
4. Assigning the sliced lists to `ans` using list slicing operations `ans[::2]` and `ans[1::2]` is done in $O(n)$ time.

So, combining these operations, the overall time complexity of the code is $O(n \log n)$ due to the sort operation.

Space Complexity

The space complexity of the code is influenced by:

1. The `cnt` object (counter of the elements), which takes $O(unique)$ space, where `unique` is the number of unique elements in `barcodes`.
2. The `ans` list, which is a new list of the same length as the input list, taking $O(n)$ space.

Therefore, the total space complexity is $O(n + unique)$, which simplifies to $O(n)$ if we consider that the number of unique items is less than or equal to n.