1054. Distant Barcodes Heap (Priority Queue) Medium Hash Table Sorting Counting ) <u>Array</u> <u>Greedy</u>

# **Problem Description**

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.

In this problem, we are given a list of integers representing barcodes. The goal is to rearrange the barcodes so that no two

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

0

class Solution:

positions.

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

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.

unique barcode to its count of appearances.

- 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. 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: Counting Frequencies: We use the Counter class from the collections module to count the frequency of each barcode in the

cnt = Counter(barcodes) **Sorting Based on Frequency**: We then sort the original list of barcodes using the sort method, applying a custom sorting key.

given list. The Counter object, named cnt, will give us a mapping of each unique barcode value to its count of appearances.

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)) 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

We first calculate the size of the array n and then create the answer array ans of that same size filled with zeros.

• 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

• 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:

high-frequency barcodes, minimizing the risk of identical adjacent barcodes:

The clever part comes with Python's slice assignment:

where no two adjacent barcodes are equal.

cnt = Counter(barcodes)

n = len(barcodes)

ans = [0] \* n

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

by their natural value:

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

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

Solution Implementation

from collections import Counter

total = len(barcodes)

rearranged = [0] \* total

**Python** 

class Solution:

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

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.

Returning the Result: The last line of the function simply returns the ans array, which is now a rearranged list of barcodes

return ans **Example Walkthrough** 

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

sorted\_barcodes = [1, 1, 1, 2, 2, 3] # 1 is most common, then 2, and 3 is least common

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

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

```
Following the steps of the solution approach:
   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\})
   Sorting Based on Frequency: Using the sorting key in sort, we arrange the barcodes first by decreasing frequency and then
```

def rearrangeBarcodes(self, barcodes: List[int]) -> List[int]:

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

ans[::2] = barcodes[: (n + 1) // 2]

ans[1::2] = barcodes[(n + 1) // 2 :]

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

ans[1::2] = [2, 2, 3]So, ans becomes:

This rearranged array adheres to the condition that no two consecutive barcodes are the same. In this example, barcode 1, which

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:

```
occurred most frequently, is well spread out, followed by barcode 2, and barcode 3 is placed at the end to satisfy the condition.
```

def rearrange\_barcodes(self, barcodes: List[int]) -> List[int]:

# Assign barcodes to even indices first (0, 2, 4, ...)

# Then assign barcodes to the odd indices (1, 3, 5, ...)

# We take the first half of the sorted barcodes

rearranged[::2] = barcodes[: (total + 1) // 2]

# We take the second half of the sorted barcodes

rearranged[1::2] = barcodes[(total + 1) // 2 :]

// Fill even indices first, then odd indices

// Return the rearranged barcodes

// Create and initialize count array

std::memset(count, 0, sizeof(count));

// Count the occurrences of each barcode

int count[maxBarcodeValue + 1];

for (int barcode : barcodes) {

++count[barcode];

int n = barcodes.size();

int index = 0;

std::vector<int> result(n);

// then the rest in odd positions

for (int i = 0; i < n; i += 2) {

const count = Array(maxBarcodeValue + 1).fill(0);

// Initialize the answer array that will be rearranged

for (let i = start; i < totalBarcodes; i += 2) {</pre>

rearranged[i] = barcodes[insertionIndex];

// Count the occurrences of each barcode

for (const barcode of barcodes) {

// The length of the barcodes array

const totalBarcodes = barcodes.length;

const rearranged = Array(totalBarcodes);

for (let start = 0; start < 2; ++start) {</pre>

count[barcode]++;

let insertionIndex = 0;

return rearranged;

C++

public:

#include <vector>

#include <cstring>

});

class Solution {

#include <algorithm>

for (int pass = 0, index = 0; pass < 2; ++pass) {</pre>

rearranged[i] = tempBarcodes[index++];

// Rearranges barcodes in such a way that no two equal barcodes are adjacent

// Find the highest value in barcodes to create an array large enough

int maxBarcodeValue = \*std::max\_element(barcodes.begin(), barcodes.end());

// Sort the barcodes based on the frequency of each barcode (descending),

// and if frequencies are equal, sort by the barcode value (ascending)

return count[a] > count[b] || (count[a] == count[b] && a < b);</pre>

// Distribute the most frequent barcodes first, filling even positions,

std::sort(barcodes.begin(), barcodes.end(), [&](int a, int b) {

std::vector<int> rearrangeBarcodes(std::vector<int>& barcodes) {

for (int i = pass; i < length; i += 2) {</pre>

# Count the frequency of each barcode

# Create a placeholder list for the answer

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

```
# Return the final arrangement of barcodes
       return rearranged
Java
import java.util.Arrays; // Required for sorting the array
public class Solution {
    public int[] rearrangeBarcodes(int[] barcodes) {
       // Determine the length of the barcodes array
       int length = barcodes.length;
       // Use wrapper class Integer for custom sorting
       Integer[] tempBarcodes = 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) {</pre>
            tempBarcodes[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(tempBarcodes, (a, b) -> count[a] == count[b] ? 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

```
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
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

// Loop through the sorted barcodes to rearrange them such that no two equal barcodes are adjacent

barcodes.sort((a, b) => (count[a] === count[b] ? a - b : count[b] - count[a]));

```
insertionIndex++;
      // Return the rearranged barcodes
      return rearranged;
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
Time and Space Complexity
Time Complexity
```

// Sort the barcodes array based on the frequency of each number and then by the number itself if frequencies are equal

## The time complexity of the given code is determined by several operations: Counting elements with Counter(barcodes) takes O(n) time, where n is the number of elements in the barcodes list.

**Space Complexity** 

barcodes.

Slicing the list into two parts (for odd and even positions) is done in O(n) time.

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.

Sorting the barcodes list with the custom sort key based on their count and value takes 0(n log n) time.

The space complexity of the code is influenced by: The cnt object (counter of the elements), which takes O(unique) space, where unique is the number of unique elements in

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 0(n + unique), which simplifies to 0(n) if we consider that the number of unique items is

less than or equal to n.