# 2080. Range Frequency Queries



## Problem Description

This problem involves creating a data structure that can efficiently find the frequency, or the number of occurrences, of a particular value within a given subarray of an integer array. A subarray is a sequence of contiguous elements from the array, and the frequency of a value is how often it appears within that sequence.

To solve this, one must implement a class `RangeFreqQuery` with two functionalities:

1. A constructor that takes an integer array `arr` and prepares it for subsequent queries.
2. A `query` method that, given a left and right index (`left` and `right`) and a value (`value`), returns how many times `value` appears between `arr[left]` and `arr[right]`, inclusive.

Since the array is not modified after the data structure is created, the problem statements suggest preprocessing the array to allow for fast frequency queries.

## Intuition

The solution strategy is based on preprocessing the given array to make querying efficient. We make use of the fact that the array remains static after the data structure is constructed. Here's how the solution works:

1. **Preprocessing**: Instead of checking the frequency of a value each time the `query` is called, we build a hashmap (or dictionary in Python) upon the class initialization. This hashmap maps each unique value in the array to a list of indices where this value occurs. Because each value is appended in the order of appearance, each list of indices is sorted.
2. **Answering Queries**: To find the frequency of a value in a particular range, we use binary search to quickly find the position of the `left` and `right` boundaries within the preprocessed list of indices for that value. Specifically, we find:
   - The first index in the list that is greater than or equal to `left`. If `left` is not in the list, we find the smallest index that is greater than `left`.
   - The last index in the list that is less than or equal to `right`. If `right` is not in the list, we find the largest index that is less than or equal to `right`.

   The Python module `bisect` provides the `bisect_right` function, which can be used to find these indices.
3. **Calculating Frequency**: The frequency is then the difference between the positions found in the list for `right` and `left`. Since `bisect_right` returns a position one past the last occurrence when searching for `right`, subtracting these two positions directly gives the correct frequency.

So, the intuition lies in transforming the problem from repetitive frequency counting into a single preprocessing step followed by fast binary search queries.

## Solution Approach

The implementation of the `RangeFreqQuery` class involves two core components:

### Initialization `__init__(self, arr: List[int])`

When an object of `RangeFreqQuery` is created, the constructor takes an input array `arr` and preprocesses it. The constructor uses a hashmap (`self.mp`) to store the frequencies. It iterates over the array using `enumerate` to access both the index `i` and the value `v` at that index. Then, it appends the index `i` to the list corresponding to the value `v` in the hashmap. This creates a mapping of each unique value to a sorted list of indices where it appears in `arr`.

Here's a breakdown of the initialization steps:

```
self.mp = defaultdict(list)
for i, v in enumerate(arr):
    self.mp[v].append(i)
```

### Querying `query(self, left: int, right: int, value: int) -> int`

The `query` method is where we find out the frequency of `value` between the `left` and `right` index in the array. We perform two binary searches using `bisect_right` from the `bisect` module:

1. Search for the insertion point for `left - 1` in the sorted list of indices for `value`. This gives us `l`, the number of occurrences of `value` before the `left` index.
2. Search for the insertion point for `right` in the same sorted list. This gives us `r`, the number of occurrences of `value` up to and including the `right` index.

The frequency is then the difference `r - l`, which is the count of `value` in the subarray starting at `left` and ending at `right`.

The following is the code for the `query` method:

```
if value not in self.mp:
    return 0
arr = self.mp[value]
l, r = bisect_right(arr, left - 1), bisect_right(arr, right)
return r - l
```

Notice that we return `0` when the `value` is not found in `self.mp`, as it indicates the `value` does not appear in the array. Otherwise, we calculate the frequency using the binary search results and return it.

In summary, the solution involves a combination of hashmap for preprocessing and binary search for efficient queries. By using these algorithms and data structures, the `RangeFreqQuery` class allows us to quickly compute the frequency of a value over different subarrays following a single preprocessing pass.

### Example Walkthrough

Let's say we have the following array `arr = [1, 1, 2, 3, 2, 2, 4, 1, 2]`, and we want to find out how often particular values appear within specified subarrays of `arr`.

We initialize the `RangeFreqQuery` object with `arr`.

- During the initialization, the object creates a hashmap that will look like this:

```
1: [ 0, 1, 7 ]
2: [ 2, 3, 4, 8 ]
3: [ 2, 3, 4, 5, 6 ]
4: [ 6 ]
```

Each unique value from `arr` is now mapped to a sorted list of indices where it appears.

Now, let's suppose we want to find out how often the value `2` appears between indices 2 and 6 (`left = 2, right = 6`). We perform the following steps:

1. Call the `query` method with `left = 2, right = 6, and value = 2`.
2. Look for the sorted list of indices for the value `2` in our hashmap, which is `[2, 4, 5, 8]`.
3. Using the `bisect_right` function from the `bisect` module, we find the insertion point for `left - 1`, which is `1` in this case.
   - `bisect_right([2, 4, 5, 8], 1)` will return `0` because `1` would fit at the start of the list.
   - This means there are `0` occurrences of `1` before the index `2`.
4. Next, we do a similar search for `right`, which is `6`.
   - `bisect_right([2, 4, 5, 8], 6)` will return `3` since `6` would fit between `5` and `8`.
   - There are `3` occurrences of `2` up to and including index `6`.
5. The frequency of value `2` between indices `2` and `6` is therefore `3 - 0 = 3`.

Putting it all together, if we wanted to illustrate this using the `RangeFreqQuery` class and the querying process, we would write the following Python code:

```
from collections import defaultdict
from bisect import bisect_left, bisect_right
from typing import List

class RangeFreqQuery:
    def __init__(self, arr: List[int]):
        self.mp = defaultdict(list)
        for i, v in enumerate(arr):
            self.mp[v].append(i)

    def query(self, left: int, right: int, value):
        if value not in self.mp:
            return 0
        arr = self.mp[value]
        l, r = bisect_right(arr, left - 1), bisect_right(arr, right)
        return r - l

# Initialize with the given array.
rangeFreqQuery = RangeFreqQuery([1, 1, 2, 3, 2, 2, 4, 1, 2])

# Example query
print(rangeFreqQuery.query(2, 6, 2))  # Output: 3
```

In this example, the print statement would output `3`, as the value `2` occurs three times in the subarray of `arr` from indices `2` to `6` (inclusive). This walk-through demonstrates the efficiency and effectiveness of the solution approach.

## Python Solution

```
from collections import defaultdict
from bisect import bisect_left, bisect_right
from typing import List

class RangeFreqQuery:
    def __init__(self, arr: List[int]):
        # Dictionary to store the indices of each value in the array.
        self.index_map = defaultdict(list)
        # Iterate through the list and map each value to its indices.
        for index, value in enumerate(arr):
            self.index_map[value].append(index)

    def query(self, left: int, right: int, value: int) -> int:
        # Check if the value is not in the index map, returning 0 frequency.
        if value not in self.index_map:
            return 0

        # Find the closest index at the left and right boundary using binary search.
        indices = self.index_map[value]
        # bisect_left to find start position inclusively, while bisect_right for the end position exclusively.
        start_position_inclusive = bisect_left(indices, left)
        end_position_exclusive = bisect_right(indices, right)

        # Calculate the frequency by subtracting and and start position.
        frequency = end_position_exclusive - start_position_inclusive
        return frequency

# Example of how to instantiate and use the RangeFreqQuery class:
# obj = RangeFreqQuery(arr)
# freq = obj.query(left, right, value)
```

## Java Solution

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

class RangeFreqQuery {
    private Map<Integer, List<Integer>> indexMap = new HashMap<>();

    // Constructor to initialize the data structure with the given array.
    // It maps each value to a list of its indices in the array.
    public RangeFreqQuery(int[] arr) {
        for (int i = 0; i < arr.length; ++i) {
            indexMap.computeIfAbsent(arr[i], k -> new ArrayList<>()).add(i);
        }
    }

    // Queries the frequency of a value in the specified range.
    // It uses binary search to find the appropriate indices.
    public int query(int left, int right, int value) {
        // If the value is not present in the array, return 0 frequency
        if (!indexMap.containsKey(value)) {
            return 0;
        }

        List<Integer> indices = indexMap.get(value);
        // Find the starting index where the value could be inserted to maintain sorted order.
        int startIndex = binarySearch(indices, left);
        // Find the ending index where the value could be inserted to maintain sorted order.
        int endIndex = binarySearch(indices, right);

        // The frequency is the difference between the start and end insertion points.
        return endIndex - startIndex;
    }

    // Helper method to perform a binary search on a list of indices.
    // It returns the index where the target value could be inserted.
    private int binarySearch(List<Integer> list, int targetValue) {
        int left = 0;
        int right = list.size();

        while (left < right) {
            int mid = (left + right) >> 1; // Equivalent to (left + right) / 2
            if (list.get(mid) < targetValue) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return left;
    }
}

// The class could be used as follows:
// RangeFreqQuery obj = new RangeFreqQuery(arr);
// int frequency = obj.query(left, right, value);
```

## C++ Solution

```
#include <vector>
#include <unordered_map>
#include <algorithm> // includes std::upper_bound

class RangeFreqQuery {
public:
    // HashMap that stores each distinct number and its indices in the input array
    std::unordered_map<int, std::vector<int>> numToIndicesMap;

    // Constructor that fills the HashMap for the given array
    RangeFreqQuery(std::vector<int>& arr) {
        for (int i = 0; i < arr.size(); ++i) {
            numToIndicesMap[arr[i]].push_back(i); // Appends the current index to the corresponding number's vector
        }
    }

    // Method to return the frequency of 'value' within the range [left, right]
    int query(int left, int right, int value) {
        // If the value is not present in the map, return 0 as its frequency is 0
        if (numToIndicesMap.find(value) == numToIndicesMap.end()) return 0;

        // References the vector of indices for 'value'
        auto& indices = numToIndicesMap[value];

        // Find the position in the vector for the first index greater than 'left - 1' (upper_bound is exclusive)
        auto leftIt = std::upper_bound(indices.begin(), indices.end(), left - 1);

        // Find the position in the vector for the first index greater than 'right'
        auto rightIt = std::upper_bound(indices.begin(), indices.end(), right);

        // The frequency is the number of elements in the range [leftIt, rightIt)
        return rightIt - leftIt;
    }
};

// The RangeFreqQuery class can be used as follows:
// RangeFreqQuery obj = new RangeFreqQuery(arr);
// int frequency = obj->query(left, right, value);
```

## Typescript Solution

```
// // Import necessary features from array and algorithm modules (hypothetical imports)
// // In TypeScript/JavaScript, there's no standard library providing an upper_bound function like C++'s <algorithm>,
// // so we'll need to create it or use a third-party library that implements algorithmic functions.

let numToIndicesMap: { [key: number]: number[] } = {}; // equivalent to unordered_map in C++

// Function to "construct" the numToIndicesMap with indices
arr.forEach((value, index) => {
    if (!numToIndicesMap[value]) numToIndicesMap[value] = [];
    numToIndicesMap[value].push(index);
});

// Helper function to mimic the 'std::upper_bound'
function upperBound(arr: number[], target: number): number {
    let start = 0, end = arr.length;
    while (start < end) {
        let mid = Math.floor((start + end) / 2);
        if (arr[mid] <= target) {
            start = mid + 1;
        } else {
            end = mid;
        }
    }
    return start; // end and start both will be pointing to the first element greater than target
}

// Function to return the frequency of 'value' within the range [left, right]
function query(left: number, right: number, value: number): number {
    if (!(value in numToIndicesMap)) {
        return 0; // If 'value' is not found, its frequency is 0
    }

    let indices = numToIndicesMap[value]; // Get the array of indices for 'value'

    // Find the position for the first index greater than 'left - 1' (simulating upper_bound exclusive nature)
    let leftIndex = upperBound(indices, left - 1);

    // Find the position for the first index greater than 'right'
    let rightIndex = upperBound(indices, right);

    // The frequency is the count of elements in the range [leftIndex, rightIndex)
    return rightIndex - leftIndex;
}
```

## Time and Space Complexity

### Time Complexity

**`__init__` constructor:**

- The time complexity for initializing the `RangeFreqQuery` class is $O(n)$, where $n$ is the length of the input array. This is because the constructor iterates through all $n$ elements to populate the `self.mp` dictionary, which maps each unique value to a list of its indices in the array.

**`query` method:**

- The `query` method utilizes binary search twice through the `bisect_right` function to find the range of positions of a particular value between indices `left` and `right`. Therefore, the time complexity for each query is $O(\log k)$, where $k$ is the number of occurrences of the value being queried ($k$ can be at most $n$).

### Space Complexity

- The `self.mp` dictionary stores lists of indices for each unique value in the array. In the worst case, the space complexity is $O(n)$, if all values are unique and each value appears only once. This would mean that the `self.mp` dictionary would have $n$ keys with a single index in each list.

In summary, the time complexity is $O(n)$ for the constructor and $O(\log k)$ for each query, and the space complexity is up to $O(n)$.