1409. Queries on a Permutation With Key

Medium | Binary Indexed Tree | Array | Simulation | Leetcode Link

Problem Description

In this problem, we are given two pieces of information:

1. An array of queries containing positive integers between 1 and m.
2. We initially have a permutation `P` consisting of all integers from 1 to m in ascending order.

The objective is to process each query by following these steps:

- Identify the position of the current query element (let's call it `queries[i]`) in the permutation `P`. This position is to be considered using 0-based indexing.
- Record the position of `queries[i]` as this is our result for the current query.
- Move the element `queries[i]` from its current position to the beginning of the permutation `P`.

We are asked to return an array that contains the result for each `queries[i]` after processing all the queries.

Intuition

To solve this problem, our approach focuses on simulating the process as described in the problem statement. Here is the thinking process for arriving at the solution:

1. **Initial Setup**: We start by generating the initial permutation `P` which is simply a list of integers from 1 to m.
2. **Processing Queries**: For each value `v` in `queries`, we need to perform the following operations:
   - **Find Position**: Locate the index `j` of `v` within list `P` which represents the initial position of `v` in the permutation.
   - **Record Result**: The index `j` is the answer for this query, so it needs to be added to an answer array `ans`.
   - **Update Permutation**: We then remove `v` from its current position in `P` and insert it at the beginning of `P`.
3. **Maintain Permutation State**: Each iteration modifies the permutation `P` according to the specified rules. Hence, the state of `P` is always maintained after processing each query.

The code provided implements this intuitive process using a loop to iterate through each query and updating the permutation `P` by utilizing Python's list methods `index()`, `pop()`, and `insert()`. Eventually, after processing all queries, the result is the `ans` list containing the positions of each query value before it was moved to the front of the list.

Solution Approach

The implementation of the solution can be walked through as follows using Python's list data structure and some of its built-in methods:

1. **List Creation**: We begin by creating the list `P` that represents the permutation. This is done using `range(1, m + 1)` which generates an iterable from 1 to `m` inclusive. The `list` function is then called to convert this iterable into a list.

   ```
   p = list(range(1, m + 1))
   ```

2. **Iterating Through Queries**: We iterate through each element `v` in the list of queries with a `for` loop.

   ```
   for v in queries:
   ```

3. **Finding Query Position**: The `index()` method of lists is used to find the position `j` of the value `v` within the permutation `P`. As the list is 0-indexed, this will give us the index starting from 0.

   ```
   j = p.index(v)
   ```

4. **Recording the Position**: We append the found position `j` to our list `ans` which will eventually be returned as our result array.

   ```
   ans.append(j)
   ```

5. **Updating the Permutation**: To move the query value to the beginning of `P`, we first remove `v` from its current position using `pop(j)` where `j` is the index found earlier.

   ```
   p.pop(j)
   ```

   Then, we insert the value `v` at the beginning of `P` by using the `insert()` function with 0 as the index to insert at.

   ```
   p.insert(0, v)
   ```

6. **Returning the Result**: After the loop has finished processing all the elements in `queries`, our answer list `ans` is returned. This list contains the original positions of each query value before we moved them.

This solution is efficient because accessing an element's index, removing an element, and inserting an element at the beginning of a list all have a time complexity of O(n), making the overall complexity of the algorithm O(n * q) where `n` is the number of elements in the permutation and `q` is the number of queries, since each query is processed independently.

The space complexity of the solution is O(m) where `m` is the size of the permutation, since it's the space required to store the permutation.

```python
class Solution:
    def processQueries(self, queries: List[int], m: int) -> List[int]:
        p = list(range(1, m + 1))
        ans = []
        for v in queries:
            j = p.index(v)
            ans.append(j)
            p.pop(j)
            p.insert(0, v)
        return ans
```

Example Walkthrough

Let's say we're given the following inputs:

- `queries`: [3, 1, 2, 1]
- `m`: 5

The initial permutation `P` from 1 to `m` (5 in this case) is `[1, 2, 3, 4, 5]`.

Now we process each query one by one:

1. The first query is `3`.
   - We find the position of `3` in `P`, which is `2`.
   - We record this position.
   - We move `3` to the front of `P` resulting in the new permutation `[3, 1, 2, 4, 5]`.
2. The next query is `1`.
   - Position of `1` is now `1` (it moved because of the previous query).
   - We record this position.
   - `1` moves to the front, resulting in `[1, 3, 2, 4, 5]`.
3. The third query is `2`.
   - Position of `2` is `2`.
   - We record the position.
   - `2` moves to the front, permutation is `[2, 1, 3, 4, 5]`.
4. The last query is `1`.
   - Position of `1` is `1` because it was moved to the front earlier.
   - We record the position.
   - `1` is already at the front, so the permutation remains `[2, 1, 3, 4, 5]`.

Each position we recorded forms our result: `[2, 1, 2, 1]`.

Thus, the final answer after processing all queries is the list of recorded positions `[2, 1, 2, 1]`.

Python Solution

```python
class Solution:
    def processQueries(self, queries: List[int], m: int) -> List[int]:
        # Initialize the P sequence with integers from 1 to m
        p_sequence = list(range(1, m + 1))
        # Initialize the answer list to store the results
        results = []
        # Process each query from the queries list
        for value in queries:
            # Find the index of the current value in the P sequence
            index = p_sequence.index(value)
            # Append the index to the results list
            results.append(index)
            # Remove the current value from its index in P
            p_sequence.pop(index)
            # Insert the current value at the beginning of the P sequence
            p_sequence.insert(0, value)
        # Return the results list containing the indices
        return results

# The code assumes the existence of `List` type hint imported from `typing` module
# If not present in the original code file, it should be added at the top as:
# from typing import List
```

Java Solution

```java
class Solution {

    // Function to process the queries and return the indices of each query in the permutation
    public int[] processQueries(int[] queries, int m) {
        // Initialize P as a LinkedList to easily support element removal and insertion at the front
        List<Integer> permutation = new LinkedList<>();

        // Fill permutation with elements 1 to m
        for (int num = 1; num <= m; num++) {
            permutation.add(num);
        }

        // Array to store the answer (indices of each queried element)
        int[] indices = new int[queries.length];

        // Initialize index for placing answers
        int ansIndex = 0;

        // Process each query in the array
        for (int query : queries) {
            // Find the index of the queried number in the permutation
            int queryIndex = permutation.indexOf(query);

            // Store the index in the result answer array
            indices[ansIndex++] = queryIndex;

            // Remove the queried number from its current position
            permutation.remove(queryIndex);

            // Add the queried number to the front of the permutation
            permutation.add(0, query);
        }

        // Return the final array of indices representing the answer
        return indices;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <numeric> // For std::iota

class Solution {
public:
    // This function processes the queries on the permutation array and returns the result.
    vector<int> processQueries(vector<int>& queries, int m) {
        // Initialize the permutation array 'P' with elements from 1 to 'm'.
        vector<int> permutation(m);
        std::iota(permutation.begin(), permutation.end(), 1);

        // Initialize the answer vector to store the results of the queries.
        vector<int> answer;

        // Loop over each value in the queries.
        for (int value : queries) {
            // Initialize an index 'foundIndex' to store the position of the value in 'permutation'.
            int foundIndex = 0;

            // Search for the value in the permutation array.
            for (int i = 0; i < m; ++i) {
                if (permutation[i] == value) {
                    foundIndex = i; // Store index where value is found.
                    break; // Exit the loop since we found the value.
                }
            }

            // Add the found index to the answer vector.
            answer.push_back(foundIndex);

            // Erase the value from its current position.
            permutation.erase(permutation.begin() + foundIndex);

            // Insert the value at the beginning of the permutation array.
            permutation.insert(permutation.begin(), value);
        }

        // Return the results of the queries.
        return answer;
    }
};
```

Typescript Solution

```typescript
function processQueries(queries: number[], m: number): number[] {
    // Initialize the permutation array 'permutation' with elements from 1 to 'm'.
    let permutation: number[] = Array.from({ length: m }, (_, index) => index + 1);

    // Initialize the answer array to store the results of the queries.
    let answer: number[] = [];

    // Loop over each value in the queries array.
    queries.forEach(value => {
        // Find the index of the 'value' in 'permutation'.
        let foundIndex = permutation.indexOf(value);

        // Add the found index to the 'answer' array.
        answer.push(foundIndex);

        // Remove the value from its current position.
        permutation.splice(foundIndex, 1);

        // Insert the value at the beginning of the permutation array.
        permutation.unshift(value);
    });

    // Return the results of the queries.
    return answer;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code primarily depends on two factors:

1. The cost of searching for an index of a value in the list `p` which is done in `O(m)` time where `m` is the length of `p`.
2. The cost of popping and inserting elements to/from the list which can take up to `O(m)` time.

Since for every value in `queries` we perform both indexing and pop-insert operations, we multiply this cost by the number of queries `n`. Therefore, the time complexity is `O(n*m)`, where `n` is the length of `queries`.

Space Complexity

The space complexity of the algorithm is to consider the additional space used by the algorithm excluding the input and output.

Here, aside from the space used by the input `queries` and the output list `ans`, the code maintains a list `p` of size `m`, but since this does not grow with the size of the input `queries`, the additional space remains constant. Thus, the space complexity is `O(1)`, which means it is constant space complexity as it doesn't depend on the size of the input `queries`.