1562. Find Latest Group of Size M

Medium Array Binary Search Simulation Leetcode Link

Problem Description

In this problem, we are given an array `arr` that is a permutation of numbers from `1` to `n`, meaning it contains all integers in that range exactly once. We also have a binary string of size `n` starting with all zeroes. We follow steps from `1` to `n`, where at each step `i`, we set the bit at position `arr[i]` to `1`.

We are tasked with finding the latest step at which there is a contiguous group of `1`s in the binary string that is exactly of length `m`. A group of `1`s can be considered a substring of the binary string that consists only of `1`s and cannot be extended further on either side without encountering a `0`. If we cannot find such a group at any step, the function should return `-1`.

Intuition

The intuition behind the solution lies in leveraging the properties of the contiguous groups of `1`s and keeping track of their lengths as the binary array is updated. Here's how we approach the problem:

1. Notice that if `m` equals `n`, the answer would be `n` because we can only form a contiguous group of `1`s of length `n` at the last step.
2. Initialize an array `cnt` to keep track of the lengths of contiguous groups of `1`s. The size of this array is `n + 2` to handle edge cases involving the first and last elements without going out of bounds.
3. Iterate through the `arr` array, decrementing the value of each `arr[i]` by one to match the 0-based indexing of the `cnt` array.
4. At each step, we detect the lengths of contiguous groups to the left and right of the current position. We do this by looking at the `cnt` array at the positions immediately before and after the current bit's position.
5. If either the left or right contiguous group of `1`s is of length `m`, we update the answer to the current step, as this step represents the latest occurrence of a group of length `m`.
6. We update the `cnt` array to reflect the changes in the lengths of contiguous groups of `1`s. The new length is the sum of the lengths of the contiguous groups to the left and right, plus one for the current bit being set to `1`.

The algorithm efficiently keeps track of the sizes of contiguous groups of `1`s at each turn and updates the answer when the group of size `m` is affected. This provides us the latest step when such a group exists.

Solution Approach

The implementation of the solution makes use of array manipulation techniques to efficiently track the lengths of contiguous groups of `1`s as the binary string gets updated.

Here's a step-by-step breakdown of the Reference Solution Approach provided in Python:

1. The problem is first simplified by handling the special case where `m` is equal to `n`. If this is the case, we return `n` since the only possible group of length `m` will be formed at the last step.
2. The `cnt` array of length `n + 2` is initialized with zeros to keep track of the lengths of contiguous groups of `1`s. Extra elements are added to the bounds to avoid index out of bounds errors when checking neighbors.
3. We iterate over the `arr` using a loop: `for i, v in enumerate(arr):`. In each iteration, `v` represents the position in the binary string from the permutation where we turn a `0` into a `1`.
4. Since `arr` is 1-indexed and Python arrays are 0-indexed, we adjust by decrementing `v` by one (`v -= 1`).
5. We retrieve the lengths of the groups on the left `l = cnt[v - 1]` and on the right `r = cnt[v + 1]` of the current position.
6. We check if the current setting of the bit creates or destroys a group of ones with the length equal to `m`. If either the left or right contiguous group size equals `m`, we update the answer with the current step `ans = i` since it could be the latest step at which there exists a group of ones of length `m`.
7. The lengths of the contiguous groups that are updated as a result of the current operation are then set at both ends of the new group by updating `cnt[v - l]` and `cnt[v + r]`.
8. The length of the new group is the sum of lengths of the left and right groups plus one for the current position (`l + r + 1`).
9. We continue this process until all elements in `arr` have been processed.
10. Finally, the variable `ans` holds the latest step at which a group of `1`s of length exactly `m` was created, and it is returned. If no such group was created, `ans` will remain as its initialized value `-1`, which is correctly returned.

By using an array to keep track of the lengths of groups and updating this as we perform each operation, we maintain a running record of the state of the binary string throughout the process, enabling us to identify the correct step at which the condition is met.

Example Walkthrough

Let's demonstrate the solution approach with an example. Suppose `arr = [3, 5, 1, 2, 4]` and `m = 1`. The binary string starts as `00000` and `n = 5`.

To aid our understanding, we'll be tracking the binary string, the `cnt` array, and which step we encounter a contiguous group of `1`s of length `m`.

1. Since `m` is not equal to `n`, we proceed with the solution and initialize `cnt` as `[0, 0, 0, 0, 0, 0, 0]`. This extra padding helps avoid out-of-bounds errors.
2. In the first step, we turn the 3rd bit of the binary string to `1`, which is at index `2` in 0-based indexing. The binary string becomes `00100`. Here, the left group `l` is `0`, and the right group `r` is also `0`. Since there are no contiguous groups of 1s, we just set the `cnt` array at indices `[2]` to `1`.

   Binary string: `00100`

   `cnt` array: `[0, 0, 1, 0, 0, 0, 0]`
3. In the second step, we set the 5th bit to `1`: `00101`. The left group on 5th position is `0` and the right group is `0`. We update the `cnt` at `[4]` to `1`. Since a contiguous subsection of length `1` is formed, we set `ans` to the current step `i` which is `2`.

   Binary string: `00101`

   `cnt` array: `[0, 0, 1, 0, 1, 0, 0]`

   `ans = 2`
4. For the third step, we update the 1st bit: `10101`. The left group `l` is `0` and the right group `r` is `0`. Update `cnt` at `[0]` to `1`. Since a contiguous subsection of length `1` is formed, we update `ans` to `3`.

   Binary string: `10101`

   `cnt` array: `[1, 0, 1, 0, 1, 0, 0]`

   `ans = 3`
5. In the fourth step, setting the 2nd bit to `1` updates the binary string to `11101`. The left group `l` is `1`, and the right group `r` is `1`. We update `cnt` at `[2]` and `cnt` at `[0]` to `1 + 1 + 1 = 3` since now they're part of a larger group. After this change, there's no group of length `1`, so `ans` is not updated.

   Binary string: `11101`

   `cnt` array: `[3, 0, 3, 0, 1, 0, 0]`
6. In the final step, we set the 4th bit to `1`: `11111`. The left group `l` is `3`, and the right group `r` is `1`. We update `cnt` at `[0]` and `cnt` at `[4]` to `3 + 1 + 1 = 5`. There's no longer a group of length `1`, so `ans` remains the same.

   Binary string: `11111`

   `cnt` array: `[5, 0, 3, 0, 5, 0, 0]`
7. Since we have processed all elements in `arr`, and we did not encounter a new group of `1`s of length `m` since step 3, `ans` remains as `3`.

Therefore, the latest step at which a contiguous group of `1`s of length exactly `m` was created is step `3`, which is the returned result of this example problem.

Python Solution

```python
class Solution:
    def findLatestStep(self, arr: List[int], length_m: int) -> int:
        # Length of the array 'arr' which represents the positions filled
        total_length = len(arr)

        # Situation where each step fills a unique position, resulting in 'm' as the total length
        if length_m == total_length:
            return total_length

        # Initialize an array to keep track of continuous segment lengths around each position
        # Extra two slots for padding the left and right ends to simplify boundary conditions
        count = [0] * (total_length + 2)

        # Variable to keep track of the latest step where there is at least one segment of length 'm'
        latest_step = -1

        # Iterate over the array 'arr', 'pos' holds the position to be filled during each step
        for step, pos in enumerate(arr):
            # Adjust 'pos' to be zero-indexed
            pos -= 1

            # Retrieve lengths of segments directly left (l) and right (r) of the current position
            left_segment, right_segment = count[pos - 1], count[pos + 1]

            # Check if either segment adjacent to the filled position is exactly of length 'm'
            if left_segment == length_m or right_segment == length_m:
                latest_step = step  # Record the latest valid step

            # Update the border values of the segment including the new position 'pos'
            # The new segment length is the sum of left, right segments plus the new position itself
            count[pos - left_segment] = count[pos + right_segment] = left_segment + right_segment + 1

        # Return the latest step where a continuous segment of length 'm' occurred
        return latest_step

# Example usage:
# result = Solution().findLatestStep([3, 5, 1, 2, 4], 1)
# Assuming the List type is imported from typing, this would return 4, since the latest step
# where a length of 1 is left occurs at step 4 (zero-indexed, would be position 5 in one-indexed)

```

Java Solution

```java
class Solution {
    // Method to find the latest step where there are exactly 'm' continuous blocks that are filled
    public int findLatestStep(int[] arr, int m) {
        int n = arr.length;
        // If 'm' is equal to the length of the array, the answer is the length since all will be filled continuously last
        if (m == n) {
            return n;
        }

        // Initialize the count array to track the number of continuous filled blocks
        // Extra space is allocated for boundaries to avoid IndexOutOfBoundException
        int[] count = new int[n + 2];
        int answer = -1; // Initialize the answer as -1, assuming no such step is found

        // Iterate through the array to simulate the filling process
        for (int i = 0; i < n; ++i) {
            int currentValue = arr[i]; // Get the current position to fill
            // Retrieve length of continuous blocks to the left and right
            int left = count[currentValue - 1];
            int right = count[currentValue + 1];

            // If either side has exactly 'm' filled blocks, update the answer to the current step
            if (left == m || right == m) {
                answer = i;
            }

            // Update the count of the new continuous block (after merging with adjacent filled blocks if any)
            // We update the beginning and end of the continuous block
            count[currentValue - left] = left + right + 1;
            count[currentValue + right] = left + right + 1;
        }
        // Return the step number where we last saw 'm' continuous blocks filled
        return answer;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function that finds the last step number where there is a group with m consecutive '1's after
    // turning the bits one by one from the 'arr' array on a binary string of all '0's.
    int findLatestStep(vector<int>& arr, int m) {
        int n = arr.size();
        // Immediately return the size of the array if m equals the array size since the last step
        // would turn the entire string into '1's which is a group of m consecutive '1's.
        if (m == n)
            return n;

        // Initialize a count array to keep track of lengths of consecutive '1's with
        // two extra elements for handling edge cases (prevent out of bounds access).
        vector<int> lengthAtEdges(n + 2, 0);

        // Variable to store the answer, start with -1 to signify not found.
        int latestStep = -1;

        // Iterate over the input array to simulate flipping the bits.
        for (int step = 0; step < n; ++step) {
            // Get the current position to flip from the array.
            int positionToFlip = arr[step];

            // Find the lengths of consecutive '1's to the left and right of the position.
            int leftConsecutiveLength = lengthAtEdges[positionToFlip - 1];
            int rightConsecutiveLength = lengthAtEdges[positionToFlip + 1];

            // If flipping this bit completes a group of size m, then set 'latestStep' to
            // the current step (1-indexed as per the problem statement).
            // Since 'step' starts from 0, we need to add 1 to align with the problem statement.
            if (leftConsecutiveLength == m || rightConsecutiveLength == m)
                latestStep = step + 1;

            // Update the length at the edges of the group formed by flipping the current bit.
            // The total length of the new group is the sum of the left and right lengths, plus 1
            // for the flipped bit itself.
            lengthAtEdges[positionToFlip - leftConsecutiveLength] =
            lengthAtEdges[positionToFlip + rightConsecutiveLength] =
                leftConsecutiveLength + rightConsecutiveLength + 1;
        }

        // Return the latest step at which there was a group of m consecutive '1's.
        return latestStep;
    }
};

```

Typescript Solution

```typescript
// Size of the array
let arraySize: number;

// Array to track lengths of consecutive '1's at the edges
let lengthAtEdges: number[];

// Initialize the variables necessary for storing the state of the solution.
function initialize(n: number) {
    arraySize = n;
    lengthAtEdges = Array(n + 2).fill(0);
}

// Function that simulates flipping the bits at each step and finds
// the last step number where there is a group with exactly m
// consecutive '1's after flipping the bits.
function findLatestStep(arr: number[], m: number): number {
    initialize(arr.length);

    // Special case: if m equals the array size, return the size of
    // the array since in the last step the entire string will be '1's.
    if (m === arraySize) {
        return arraySize;
    }

    // Variable to store the answer; initialized to -1 to signify not found
    let latestStep: number = -1;

    // Iterate over the array to simulate flipping the bits
    for (let step = 0; step < arraySize; ++step) {
        let positionToFlip: number = arr[step];

        // Find the lengths of consecutive '1's to the left and right
        let leftConsecutiveLength: number = lengthAtEdges[positionToFlip - 1];
        let rightConsecutiveLength: number = lengthAtEdges[positionToFlip + 1];

        // Check if flipping this bit completes a group of size m
        if (leftConsecutiveLength === m || rightConsecutiveLength === m) {
            latestStep = step + 1;
        }

        // Update the length at the edges of the group formed by
        // flipping the current bit including flipped bit itself
        let newGroupLength: number = leftConsecutiveLength + rightConsecutiveLength + 1;
        lengthAtEdges[positionToFlip - leftConsecutiveLength] = newGroupLength;
        lengthAtEdges[positionToFlip + rightConsecutiveLength] = newGroupLength;
    }

    // Return the latest step with a group of m consecutive '1's
    return latestStep;
}

```

Time and Space Complexity

The given Python code aims to solve a problem by tracking lengths of segments of '1's in a binary array, which is initially all '0's. Each element in `arr` represents the position being flipped from '0' to '1'. The code returns the last step (`i`) where there exists a segment with exactly `m` '1's.

Time Complexity

The primary operation in this loop is accessing and modifying elements in the `cnt` list, which takes constant time O(1) for each access or modification.

The for-loop runs for each element in `arr`, which means it iterates `n` times, where `n` is the length of `arr`. Inside this loop, there are constant-time operations being done. Therefore, the overall time complexity of the code is `O(n)`.

Space Complexity

The space complexity is determined by the additional space used by the algorithm aside from the input. In this code, the dominant extra space is used by the `cnt` array, which has a length of `n + 2`. Hence, space complexity is also `O(n)`.

To summarize, the algorithm has a time complexity of `O(n)` and a space complexity of `O(n)`.