1521. Find a Value of a Mysterious Function Closest to Target

Hard Bit Manipulation Segment Tree Array Binary Search Leetcode Link

Problem Description

Winston has a function `func` which he can only apply to an array of integers, `arr`. He has a target value, and he wants to find two indices, `l` and `r`, where `func` applied to the elements between `l` and `r` (inclusive) in `arr` will give a result that, when we take the absolute difference with the target, gives the smallest possible value. In other words, he wants to minimize `|func(arr, l, r) - target|`.

The mysterious function `func` simply takes the bitwise AND of all the elements from `l` to `r`. The bitwise AND operation takes two bits and returns 1 if both bits are 1, otherwise it returns 0. When applied to a range of integers, it processes their binary representations bit by bit.

Intuition

To solve this problem, we take advantage of the properties of the bitwise AND operation. Specifically, when you perform a bitwise AND of a sequence of numbers, the result will never increase; it can only stay the same or decrease. This means as you extend the range of `l` to `r`, the result of `func` applied to that range can only decrease.

Given this, we start with a single element and then iteratively add more elements to our range, keeping track of all possible results of `func` that we've seen so far with the current element and all previous elements. We store these in a set to avoid duplicates and to quickly find the minimum absolute difference from the target.

Each time we add a new number to the range, we update our set. We use bitwise AND with each value in our set and the new number. This will give us all the possible results with the new number at the right end of the range. We also include the new number itself in the set.

Since the number of different possible results is limited (because the number of bits in the binary representation of the numbers is limited), this set will not grow too large. For each new number, we iterate through the set to find the result closest to our target by calculating the absolute difference from the target for each value in the set and keep track of the minimum.

The minimum of these absolute differences encountered while processing the entire array gives us the answer.

Solution Approach

The solution to this problem makes use of a hash table, embodied in Python as a set, to keep track of all unique results of the `func` as we iterate through the array.

Here is a step-by-step breakdown of the algorithm:

1. Initialize an answer variable, `ans`, with the absolute difference between the first element of `arr` and `target`.
2. Initialize a set `s`, which initially contains just the first element of `arr`; this set is used to keep all possible results of the bitwise AND operation.
3. Iterate through each element `x` in `arr`:
   - Create a new set based on `s` where each element `y` from `s` is replaced with `x & y` (this represents extending the range with the new element at the right end).
   - Add the current element `x` to the new set to include the case where the range is just the single element `x`.
   - Update `ans` with the minimum value between the previous `ans` and the smallest absolute difference between any member of the current set and the `target`.
4. At the end of the loop, `ans` will hold the minimum possible value of `|func(arr, l, r) - target|` as it contains the smallest absolute difference found during the iteration.

Algorithm and Data Structures:

- **Hash Table/Set**: Used to store all possible results of the bitwise AND operation without duplication.
- **Bitwise AND**: Used within the main logic to find all possible results when considering a new element in `arr`.
- **Iteration**: Scanning through each element in `arr` and updating the set and the minimum absolute difference accordingly.
- **Monotonicity**: The property that the bitwise AND of a set of non-negative integers is monotonic when extending the range to the right is utilized. That is, adding more numbers to the right in the range cannot increase the result of the AND operation.

This approach is efficient because as we move through the array, we do not need to compute the result of `func` for every possible range; instead, we only need to consider a limited number of possibilities at each step, thanks to the monotonicity of the bitwise AND operation and the fact that there is a limited number of bits in binary representations of the integers.

Space and Time Complexity:

- The space complexity is `O(N)`, where `N` is the size of the set. In the worst case, it is proportional to the number of bits in the binary representation of the numbers, which is far less than the length of `arr`.
- The time complexity is `O(NlogM)`, where `N` is the length of `arr` and `M` is the maximum number possible in the array (which affects the number of bits we need to consider when doing bitwise operations).

Example Walkthrough

Let's go through an example to illustrate the solution approach. Suppose we have the following input:

- `arr = [3, 1, 5, 7]`
- `target = 2`

We need to find two indices `l` and `r` such that the bitwise AND of the numbers between indices `l` and `r` (inclusive) in `arr` minimizes the absolute difference with the target value `2`.

1. We initialize `ans` with the absolute difference between first element of `arr` and the target: `abs(3 - 2) = 1`.
2. We create a set `s` that starts with the first element of `arr`: `{3}`.
3. We start iterating through each element `x` in `arr` starting from the second element:
   - First, we process `x = 1`:
     - Create a temporary set: `{3 & 1}` which evaluates to `{1}`.
     - Add `x` (which is `1`) to the set, resulting in `{1}`. (No change in this case as `1 & 3` also gives `1`)
     - Now, `s` becomes `{1}`.
     - Update `ans` with the minimum between the previous `ans` (`1`) and `abs(1 - 2)` which gives `1`. So `ans` remains `1`.
   - Next, we process `x = 5`:
     - New set: `{1 & 5}` which evaluates to `{1}`.
     - Add `x` (which is `5`) to the set, resulting in `{1, 5}`.
     - Update `ans` with the minimum absolute difference in the set `{abs(1 - 2), abs(5 - 2)}` which are `1` and `3`, respectively. Thus, the `ans` remains `1`.
   - Finally, we process `x = 7`:
     - New set: `{1 & 7, 5 & 7}` which simplifies to `{1, 5}` (since `1 & 7 = 1` and `5 & 7 = 5`).
     - Add `x` (which is `7`) to the set, resulting in `{1, 5, 7}`.
     - Update `ans` with the minimum absolute difference in the set `{abs(1 - 2), abs(5 - 2), abs(7 - 2)}` which are `1`, `3`, and `5`, respectively. The `ans` remains `1`.
4. After processing the entire array, we find that the minimum possible value of `|func(arr, l, r) - target|` is `1`, which we stored in `ans`.

Therefore, the two indices `l` and `r` in `arr` that lead to this `ans` make up the solution to our example. In this case, because the target is `2`, and the closest value obtainable from `arr` using the `func` is `1`, the result is produced by the range `l` to `r` that contains any of the single elements equal to `1` or the range that produces `1` via bitwise AND, which in this case could be from index `0` to `1` (`3 AND 1` yields `1`).

Python Solution

```python
class Solution:
    def closest_to_target(self, arr: List[int], target: int) -> int:
        # Initialize the answer with the absolute difference between
        # the first element and the target
        closest_difference = abs(arr[0] - target)

        # Initialize a set with the first element from the array
        # This set will hold the results of 'AND' operation
        seen = {arr[0]}

        # Iterate over elements in the array
        for num in arr:
            # Update set with results of 'AND' operation of the current number
            # with all previously seen results and include the current number itself
            seen = {num & prev_result for prev_result in seen} | {num}

            # Calculate the closest_difference by finding the minimum absolute difference
            # between any result in 'seen' and the target
            closest_difference = min(closest_difference, min(abs(result - target) for result in seen))

        # Return the closest difference found
        return closest_difference

```

Java Solution

```java
class Solution {

    // Method to find the smallest difference between any array element bitwise AND and the target
    public int closestToTarget(int[] arr, int target) {
        // Initialize the answer with the absolute difference between the first array element and target
        int closestDifference = Math.abs(arr[0] - target);

        // Initialize a set to store the previous results of bitwise ANDs
        Set<Integer> previousResults = new HashSet<>();
        previousResults.add(arr[0]); // add the first element to set of previous results

        // Iterate through the array to compute bitwise ANDs
        for (int element : arr) {
            // A new HashSet to store current results
            Set<Integer> currentResults = new HashSet<>();

            // Compute bitwise AND of the current element with each element in previousResults set
            for (int previousResult : previousResults) {
                currentResults.add(element & previousResult);
            }

            // Also add the current array element by itself
            currentResults.add(element);

            // Iterate over current results to find the closest difference to target
            for (int currentResult : currentResults) {
                // Update the smallest difference if a smaller one is found
                closestDifference = Math.min(closestDifference, Math.abs(currentResult - target));
            }

            // Update the previousResults set with currentResults for the next iteration
            previousResults = currentResults;
        }

        // Return the smallest difference found
        return closestDifference;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_set>
#include <cmath>
#include <algorithm>

class Solution {
public:
    int closestToTarget(std::vector<int>& arr, int target) {
        // Initialize the answer with the difference between
        // the first element and the target.
        int closestDifference = std::abs(arr[0] - target);

        // Use a set to store the results of AND operations of elements (prefix results).
        std::unordered_set<int> prefixResults;
        prefixResults.insert(arr[0]);

        // Iterate over the array.
        for (int num : arr) {
            // Use a new set to store current results of AND operations.
            std::unordered_set<int> currentResults;
            currentResults.insert(num);

            // Perform AND operation with the previous result and insert it into current results.
            for (int prefixResult : prefixResults) {
                currentResults.insert(num & prefixResult);
            }

            // Iterate over current results to find the closest to the target.
            for (int currentResult : currentResults) {
                closestDifference = std::min(closestDifference, std::abs(currentResult - target));
            }

            // Move current results to the prefix results for the next iteration.
            prefixResults = std::move(currentResults);
        }

        // Return the smallest difference found.
        return closestDifference;
    }
};

```

Typescript Solution

```typescript
function closestToTarget(arr: number[], target: number): number {
    // Initialize the answer with the absolute difference between
    // the first element of the array and the target value.
    let answer = Math.abs(arr[0] - target);

    // Initialize a set to keep track of the previously computed AND values.
    let previousSet = new Set<number>();
    previousSet.add(arr[0]); // Add the first element to the previous set.

    // Iterate over each element in the array.
    for (const number of arr) {
        // Create a new set for the current number to store AND values.
        const currentSet = new Set<number>();
        currentSet.add(number); // Add the current number itself to the set.

        // Calculate the AND of the current number with each value from the previous set.
        for (const previousValue of previousSet) {
            const andValue = number & previousValue; // Compute the AND value.
            currentSet.add(andValue); // Add the new AND value to the current set.
        }

        // Iterate through the current set to find the value closest to the target.
        for (const currentValue of currentSet) {
            // Update the answer with the minimum absolute difference found so far.
            answer = Math.min(answer, Math.abs(currentValue - target));
        }

        // Update the previous set to be the current set for the next iteration.
        previousSet = currentSet;
    }

    // Return the answer after processing all elements.
    return answer;
}

```

Time and Space Complexity

The code provided calculates the closest number to the target number in the list by bitwise AND operation. Let's analyze the time complexity and space complexity of the code.

Time Complexity:

The time complexity of this algorithm can be analyzed by looking at the two nested loops. The outer loop runs `n` times, where `n` is the length of `arr`. The inner loop, due to the nature of bitwise AND operations, runs at most `log(M)` times where `M` is the maximum value in the array. This is because with each successive AND operation, the set of results can only stay the same size or get smaller, and a number `M` has at most `log(M)` bits to be reduced in successive AND operations. Therefore, the time complexity is `O(n * log(M))`.

Space Complexity:

Regarding space complexity, the set `s` can have at most `log(M)` elements due to the fact that each AND operation either reduces the number of bits or keeps them unchanged. Since we are only storing integers with at maximum `log(M)` different bit-pattern reductions, the space complexity is `O(log(M))`.