3022. Minimize OR of Remaining Elements Using Operations

Hard Greedy Bit Manipulation Array

Problem Description

You are provided with an array of integers called `nums` and another integer `k`. Your objective is to minimize the value of the bitwise OR (`|`) of the array's elements. You are allowed to perform up to `k` operations. In a single operation, you can pick any element `nums[i]`, except the last one, and replace both `nums[i]` and `nums[i + 1]` with `nums[i] & nums[i + 1]`, where `&` represents the bitwise AND operation. The problem asks for the smallest possible bitwise OR value of the array after performing no more than `k` operations.

Intuition

To solve this problem, we need to understand how bitwise operations work. Bitwise AND (`&`) reduces the set bits so, for any two numbers, `x & y` will have fewer or equal set bits compared to `x | y`. So performing the AND operation will not increase the bitwise AND result but may reduce it.

The approach to finding the minimum bitwise OR hinges on handling the individual bits of the integers. The solution iterates over the bits from the most significant bit to the least significant bit (from left to right). For each bit position, it checks whether setting that bit could reduce the overall OR of the array without exceeding `k` operations.

The solution maintains two variables, `ans` and `rans`. `ans` is the current running result for the minimum possible value of bitwise OR, and `rans` is for the result if we decide not to set the current bit under examination.

The `cnt` variable keeps track of how many numbers still have the current bit set after applying the AND operation with `ans | (1 << i)` (which is equivalent to setting the current bit). If `cnt` is greater than `k`, it means we cannot set this bit in the result since we do not have enough operations left to eliminate the set bits in all elements by doing AND operations. Therefore, `rans` must include this bit.

On completion, `rans` will have the minimum bitwise OR value without exceeding `k` operations, as it correctly takes into account whether or not we can afford to set each bit considering the remaining allowable operations.

Solution Approach

In the provided implementation, the idea is to find the minimum possible value of the bitwise OR of the array after up to `k` operations. The solution uses bit manipulation and greedy strategy to achieve this.

The algorithm proceeds as follows:

1. Initialize two variables, `ans` and `rans`, to 0. `ans` serves as a partial answer considering the addition of the current bit, and `rans` is the result if we decide not to allow the current bit in the result.
2. Iterate over the bit positions from the 29th bit (considering 32-bit integers) to the 0th bit:
   - Calculate `test` by setting the current `i`th bit, on top of the partial answer `ans`.
   - Initialize `cnt` to count how many numbers, after the AND operation with `test`, still keep the current bit. Initialize `val` to 0, representing the resulting AND of the numbers while considering the bit of interest.
   - Iterate over each number in `nums` and apply the AND operation with `test` to update `val`. The `val` variable accumulates the result of bitwise AND on the current bit position.
     - If `val` is non-zero after the AND operation, increment `cnt`.
   - If the count `cnt` is greater than `k`, it implies we can't afford to set this bit with the remaining number of operations. In that case, set the current bit in `rans`.
   - If `cnt` is less than or equal to `k`, we can set this bit in `ans` and continue potentially setting more bits in subsequent iterations.
3. By the end of the loop, `rans` will hold the minimum possible value of the bitwise OR after at most `k` operations.

The use of `ans` and `rans` along with the count `cnt` allows us to determine, for each bit, whether setting it would leave us with a value that exceeds the allowed number of operations. By greedily attempting to set each bit starting from the most significant and moving to the least significant, we ensure that we are minimizing the OR value as much as possible.

This algorithm relies on bit manipulation by individually examining each bit position across all numbers in the array. By operating on each bit, we are breaking down the problem into smaller parts, which makes it more tractable since bitwise AND and OR operations are straightforward for individual bits. The greedy aspect comes into play by deciding whether to set a bit in our answer based on current information and what is allowed by our constraint `k`.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have the following input:

- `nums = [14, 7, 3]`
- `k = 1`

The binary representations of the numbers in the array are:

- `14` in binary: `1110`
- `7` in binary: `0111`
- `3` in binary: `0011`

Now, let's apply the algorithm step by step:

1. We initialize `ans = 0` and `rans = 0`. These will keep track of the bitwise OR value as we proceed.
2. We iterate over the bit positions from the 29th to the 0th bit. Since the example integers are small, we'll start from the 3rd bit because the higher bits are 0 for all numbers.
3. For the 3rd bit (`i = 3`), we calculate `test = ans | (1 << 3)` which is `1000` in binary (8 in decimal).
4. We then initialize `cnt = 0` and `val = 0` and iterate over each number in `nums`, &-ing each number with `test` to determine which numbers would retain the 3rd bit when AND-ed with `test`. Since none of the numbers in `nums` has a higher bit than 3, `cnt` remains `0`, and thus we proceed without increasing `rans`, but we set bit 3 in `ans` making `ans = 1000`.
5. For the 2nd bit (`i = 2`), we would calculate `test = ans | (1 << 2)` which gives us `1100` in binary. We iterate over the numbers again and find that only `14` (`1110`) would retain the 2nd bit when AND-ed with `test`. So `cnt = 1` which is `<= k`. Therefore, we do not set the 2nd bit in `rans` and `ans` becomes `1100`.
6. For the 1st bit (`i = 1`), we calculate `test = ans | (1 << 1)` which is `1110` in binary. Similarly, `cnt` would be 1 since only `14` has the 1st bit set when AND-ed with `1110`. Again, `cnt` is `<= k`, we don't set the 1st bit in `rans`, and now `ans` is `1110`.
7. Lastly, we perform the same check for the 0th bit (`i = 0`), `test` becomes `1111`. Now, all numbers `14`, `7`, `3` have the 0th bit set when AND-ed with `test`, giving us a `cnt` of 3, which is greater than our `k = 1`. So we cannot unset this bit with our `k` operations. We add the 0th bit to `rans` by setting it, making `rans = 0001`.

By the end of the process, we've got an `ans` of `1110` (`14` in decimal) and a `rans` of `0001` (1 in decimal), which means our minimum possible bitwise OR value after a maximum of 1 operation is 1. This is because we can perform a single AND operation on the last two elements (`7 & 3` to get `7 & 3` which is `0111 & 0011` equals `0011` (3 in decimal). Then, performing the OR operation across the array `14 | 3` gives us `1110 | 0011` which equals `1111`. However, since we've determined through the algorithm that we can set the 0th bit in `rans`, `rans = 0001` is the smallest we can achieve. Now, if we replace `14` and `7` with their AND, we have `nums = [14 & 7, 3]` which equals `[6, 3]`. The `|` of `4` and `3` is `4 | 3` which is `0100 | 0011` equals `0111` (7 in decimal). However, since we only made one operation to replace two elements with their AND, and `rans = 0001` signifies that our algorithm indicates that the minimum OR is `0001` (1 in decimal), we have achieved a minimum bitwise OR of 1.

Thus, by following the steps, we have determined that we could minimize the bitwise OR to 1 after at most one operation. This walk-through demonstrates how the given algorithm leverages bit manipulation and a greedy approach to reduce the bitwise OR of an array in a limited number of operations.

Solution Implementation

Python

```python
class Solution:
    def minOrAfterOperations(self, nums: List[int], k: int) -> int:
        # Initialize the running minimum OR value and the result
        min_or_value = 0
        result = 0
        # Iterate over the bit positions from 29 down to 0
        for bit_position in range(29, -1, -1):
            # Calculate a test OR value by setting the current bit
            test_or = min_or_value | (1 << bit_position)
            # Initialize counters for the number of numbers meeting the criteria
            count_met = 0
            current_val = 0
            # Check each number against the test OR value
            for num in nums:
                # Calculate the AND of the current number and the test OR value
                if current_val == 0:
                    current_val = test_or & num
                else:
                    current_val &= test_or & num
                # If the result is non-zero, the criterion is met
                if current_val:
                    count_met += 1
            # If the count is greater than k, do not include this bit in the final result
            if count_met > k:
                result |= (1 << bit_position)
            # Otherwise, include this bit in the running minimum OR value
            else:
                min_or_value |= (1 << bit_position)
        # Return the calculated result
        return result
```

Java

```java
class Solution {
    public int minOrAfterOperations(int[] nums, int k) {
        // Initialize the answer and the running answer variables
        int answer = 0;
        int runningAnswer = 0;

        // Start from the 29th bit (assuming 32-bit integers) and work down to the 0th bit
        for (int i = 29; i >= 0; --i) {
            // Calculate a test value by setting the i-th bit of the current answer
            int testValue = answer | (1 << i);
            // Initialize the counter for the number of nums values that have the i-th bit set
            int count = 0;
            // Initialize the value to store the result of the AND operation
            int andResult = 0;

            // Iterate through all numbers in the input array
            for (int num : nums) {
                // If andResult is zero, perform AND operation between test and the current number
                if (andResult == 0) {
                    andResult = test & num;
                } else {
                    // If andResult is not zero, accumulate the AND result with the current AND operation
                    andResult &= test & num;
                }

                // Increment the count if the result of the AND operation is not zero
                if (andResult != 0) {
                    count++;
                }
            }

            // If count exceeds k, only update the running answer with the i-th bit set
            if (count > k) {
                runningAnswer |= (1 << i);
            } else {
                // Otherwise, update the answer with the i-th bit set
                answer |= (1 << i);
            }
        }

        // Return the running answer value which is the correct minimum OR value after k operations
        return runningAnswer;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to find the minimum OR value after performing operations.
    int minOrAfterOperations(vector<int>& nums, int k) {
        int possibleAnswer = 0; // Variable to store the possible OR result
        int result = 0; // Variable to store the final OR result

        // Iterate through each bit position starting from the highest (29th bit) to the lowest
        for (int i = 29; i >= 0; --i) {
            int testValue = possibleAnswer | (1 << i); // Create a test value by setting the i-th bit
            int count = 0; // Counter to store how many numbers have a non-zero bit at the i-th position
            int runningAnd = 0; // Variable to store the running AND of numbers AND with testValue

            // Iterate through all elements in the nums array
            for (int number : nums) {
                if (runningAnd == 0) {
                    // `0 & ...` when runningAnd is 0, will always result in 0. Thus, it's equivalent to `runningAnd = testValue & number`
                    runningAnd = testValue & number;
                } else {
                    // Perform the AND operation with testValue and the current number, then AND with runningAnd
                    runningAnd &= testValue & number;
                }
                if (runningAnd) {
                    // Increase the counter if runningAnd has a non-zero bit at the i-th position
                    count++;
                }
            }

            // If the count is greater than k, add the bit to the result
            if (count > k) {
                result |= (1 << i);
            } else {
                // If not, add the bit to the possibleAnswer to maintain state for the next iteration
                possibleAnswer |= (1 << i);
            }
        }

        return result; // Return the final result
    }
};
```

TypeScript

```typescript
/**
 * Function to find the minimum OR value after performing operations.
 * @param {number[]} nums - The array of integers.
 * @param {number} k - The number of operations allowed.
 * @return {number} - The minimum OR value.
 */
function minOrAfterOperations(nums: number[], k: number): number {
    let possibleAnswer = 0; // Variable to store the possible OR result
    let result = 0; // Variable to store the final OR result

    // Iterate through each bit position starting from the highest (29th bit) to the lowest
    for (let i = 29; i >= 0; --i) {
        let testValue = possibleAnswer | (1 << i); // Create a test value by setting the i-th bit
        let count = 0; // Counter to store how many numbers have a non-zero bit at the i-th position
        let runningAnd = 0; // Variable to store the running AND of numbers AND with testValue

        // Iterate through all elements in the nums array
        for (let number of nums) {
            if (runningAnd === 0) {
                // `0 & ...` when runningAnd is 0, will always result in 0. Thus, it's equivalent to `runningAnd = testValue & number`
                runningAnd = testValue & number;
            } else {
                // Perform the AND operation with testValue and the current number, then AND with runningAnd
                runningAnd &= testValue & number;
            }
            if (runningAnd) {
                // Increase the counter if runningAnd has a non-zero bit at the i-th position
                count++;
            }
        }

        // If the count is greater than k, add the bit to the result
        if (count > k) {
            result |= (1 << i);
        } else {
            // If not, retain the bit in possibleAnswer to maintain state for the next iteration
            possibleAnswer |= (1 << i);
        }
    }

    return result; // Return the final result
}
```

Python3

```python
class Solution:
    def minOrAfterOperations(self, nums: List[int], k: int) -> int:
        # Initialize the running minimum OR value and the result
        min_or_value = 0
        result = 0
        # Iterate over the bit positions from 29 down to 0
        for bit_position in range(29, -1, -1):
            # Calculate a test OR value by setting the current bit
            test_or = min_or_value | (1 << bit_position)
            # Initialize counters for the number of numbers meeting the criteria
            count_met = 0
            current_val = 0
            # Check each number against the test OR value
            for num in nums:
                # Calculate the AND of the current number and the test OR value
                if current_val == 0:
                    current_val = test_or & num
                else:
                    current_val &= test_or & num
                # If the result is non-zero, the criterion is met
                if current_val:
                    count_met += 1
            # If the count is greater than k, do not include this bit in the final result
            if count_met > k:
                result |= (1 << bit_position)
            # Otherwise, include this bit in the running minimum OR value
            else:
                min_or_value |= (1 << bit_position)
        # Return the calculated result
        return result
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be analyzed as follows:

- There is an outer loop that iterates from 29 down to 0. This loop runs exactly 30 times, which corresponds to the 30 bits in the integer representation (assuming a 32-bit integer where the last two bits are used for sign representation in most cases).
- Inside the outer loop, there is an inner loop that iterates through all the `num` elements in the `nums` list. If `n` is the size of the `nums` list, then this loop runs `n` times for each iteration of the outer loop.
- Within the inner loop, basic bit manipulation operations (such as AND operations and checks). These operations are constant time.

Considering the above, we can determine that the time complexity is $O(30 \times n)$ which simplifies to $O(n)$ because the 30 is a constant factor and does not depend on the size of the input.

Space Complexity

Analyzing the space complexity:

- We have integer variables `ans`, `rans`, `test`, `cnt`, and `val` which all use constant space.
- There is no additional data structure that scales with the input size.

Therefore, the space complexity is $O(1)$ because the memory usage does not increase with the size of the input list `nums`.