2680. Maximum OR

Medium Greedy Bit Manipulation Array Prefix Sum Leetcode Link

Problem Description

In this problem, you're given an array of integers `nums` and an integer `k`. Your task is to maximize the value of a bitwise OR operation over all the elements of the array, with the option to double any element of the array up to `k` times. Doubling an element is done by multiplying it by `2`. The final result should be the maximum value obtained by performing the bitwise OR operation on all elements after at most `k` doublings. Bitwise OR operation takes two numbers and compares their bits, resulting in a number that has a `1` in each bit position if either of the original numbers has a `1` in that position.

Intuition

The key insight to solve this problem is understanding how the bitwise OR operation works and how doubling a number influences the result. When you apply the bitwise OR operation over an array of numbers, the resulting value has `1`s in all bit positions where there is a `1` in any of the numbers. Therefore, increasing the value of one number by multiplying it by `2` essentially shifts all of the `1` bits to the left by one position. Doing this `k` times would shift the `1` bits to the left `k` times, potentially creating a larger number.

The greedy approach is based on the idea that to maximize the overall result, we should focus on doubling the numbers that contribute the most `1` bits that are not already present in the accumulated OR result. Therefore, instead of randomly distributing the doublings across different numbers, the strategy is to double the same number `k` times to maximally shift its `1` bits.

Preprocessing the input array to calculate suffix OR values, which represent the bitwise OR of all numbers from the current position to the end, helps to understand the impact of each element on the total OR value. Combining this with a running prefix OR value, we can assess the total OR value if we were to double the current element `k` times. By iterating over each array element and performing this calculation, while keeping track of the maximum OR value obtained thus far, we can arrive at the optimal solution.

Solution Approach

The solution approach outlined in the reference is a strategic combination of greediness, preprocessing, and bit manipulation to solve the problem.

Here are the specific steps we take in the implementation:

1. **Calculate Suffix OR:** We create a suffix OR array `suf` that helps with understanding the impact of each element on the total OR value from that element to the end. This is achieved by iterating from the end of the `nums` array towards the initial element and cumulatively applying the bitwise OR operation. Therefore, `suf[i]` represents the OR result of `nums[i] | nums[i + 1] | ... | nums[n - 1]`.

2. **Iterate with a Prefix OR and a Running Maximum:** We keep a running prefix OR value `pre` that represents the cumulative OR value from the start of the array up to the current index. We also keep track of the maximum value encountered `ans`. At each step, we apply `k` doublings to the current element (`nums[i]`) by left-shifting its bits `k` times (equivalent to `nums[i] << k`), and then we calculate the OR with the prefix `pre` and suffix `suf[i + 1]`. The `max` function allows us to update `ans` if the newly calculated OR is greater than the previous maximum.

3. **Update Prefix OR:** After handling each element, we need to update the prefix OR value `pre` with the current element before moving on to the next in order to properly prepare for the next iteration.

4. **Return the Maximum Value:** After processing all elements, `ans` holds the maximum possible OR value, which we then return.

This approach is efficient because it computes the prefix and suffix OR values on the fly and only considers doubling an element at most `k` times. By recognizing the importance of bit positions in OR operations, it strategically selects where to apply the operation to achieve the maximum result.

In terms of algorithms and data structures:

- The **suffix OR array** is a form of dynamic programming that allows for constant-time access to the OR result of all elements from any given position to the end.
- The **bit manipulation** with left shifts (`<<`) enables efficient doubling of the elements.
- The **greedy choice** is based on utilizing all `k` doublings on one element to maximize its contribution to the overall OR result.

The algorithm avoids unnecessary doublings by considering the impact of each bit position and prioritizing the doubling where it adds the most significant new bits to the running OR result.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we are given the following array `nums` and the integer `k`:

```
nums = [3, 6, 8, 2]
k = 3
```

Our goal is to maximize the bitwise OR value by doubling at most `k = 3` elements.

1. **Calculate Suffix OR:**

   Create the suffix OR array `suf`:

   - `suf[3] = nums[3] = 2 (binary: 10)`
   - `suf[2] = nums[2] | suf[3] = 8 | 2 = 10 (binary: 1010)`
   - `suf[1] = nums[1] | suf[2] = 6 | 10 = 14 (binary: 1110)`
   - `suf[0] = nums[0] | suf[1] = 3 | 14 = 15 (binary: 1111)`

   Thus, `suf = [15, 14, 10, 2]`.

2. **Iterate with a Prefix OR and a Running Maximum:**

   Initialize `pre` as `0` and `ans` as `0`. Now iterate through `nums`:

   - For `i = 0`, `nums[0] = 3 (binary: 11)`. After doubling `k` times: `3 << 3 = 24 (binary: 11000)`. The combined OR with the suffix OR is `11000 | 14 = 30 (binary: 11110)`. Update `ans` to `30`.
   - Update `pre` to include `nums[0]`: `pre = pre | 3 = 3 (binary: 11)`.
   - For `i = 1`, `nums[1] = 6 (binary: 110)`. After doubling `k` times: `6 << 3 = 48 (binary: 110000)`. The combined OR with `pre` and suffix OR is `110000 | 11 | 10 = 50 (binary: 110010)`. `ans` remains `30` as `50` does not increase the OR value.
   - Update `pre` to include `nums[1]`: `pre = 3 | 6 = 7 (binary: 111)`.
   - For `i = 2`, `nums[2] = 8 (binary: 1000)`. After doubling `k` times: `8 << 3 = 64 (binary: 1000000)`. Combined OR with `pre` and suffix OR is `1000000 | 111 | 2 = 71 (binary: 1000111)`. Update `ans` to `71`.
   - Update `pre` to include `nums[2]`: `pre = 7 | 8 = 15 (binary: 1111)`.
   - For `i = 3`, `nums[3] = 2 (binary: 10)`. After doubling `k` times: `2 << 3 = 16 (binary: 10000)`. Combined OR with `pre` is `10000 | 1111 = 31 (binary: 11111)`. `ans` remains `71` as `31` does not increase the OR value.

3. **Return the Maximum Value:**

   After processing all elements, the maximum OR value we found is `71`, which is in binary `1000111`. Return `71` as the answer.

This walkthrough demonstrates the systematic approach of combining both bit manipulation with the greedy concept of maximizing bit shifts and using dynamic programming techniques to maintain suffix and prefix OR values.

Python Solution

```python
class Solution:
    def maximumOr(self, nums: List[int], k: int) -> int:
        # Length of the input list.
        length_nums = len(nums)

        # Suffix array to store the suffix OR starting from each position.
        suffix_or = [0] * (length_nums + 1)

        # Compute the suffix OR values in a right-to-left manner.
        for i in range(length_nums - 1, -1, -1):
            suffix_or[i] = suffix_or[i + 1] | nums[i]

        # Initialize answer and prefix OR.
        max_or_result = prefix_or = 0

        # Enumerate over nums to find the maximum OR after applying the shift
        for i, num in enumerate(nums):
            # Calculate OR by combining current prefix OR, the shifted num, and the suffix OR;
            # updating the max if a larger value is found.
            max_or_result = max(max_or_result, prefix_or | (num << k) | suffix_or[i + 1])

            # Update the prefix OR accumulatively.
            prefix_or |= num

        # Return the maximum value found.
        return max_or_result

```

Java Solution

```java
class Solution {
    public long maximumOr(int[] nums, int k) {
        int length = nums.length; // Store the length of the input array
        long[] suffixOr = new long[length + 1]; // Create an array to store suffix OR values
        // Calculate the suffix OR values from the end of the array to the beginning
        for (int i = length - 1; i >= 0; --i) {
            suffixOr[i] = suffixOr[i + 1] | nums[i];
        }
        long maxOr = 0; // Variable to store the maximum OR result
        long prefixOr = 0; // Variable to accumulate the prefix OR values

        // Loop through the given array
        for (int i = 0; i < length; ++i) {
            // Calculate the maximum OR by considering the current number shifted left by k bits,
            // the OR of previous numbers and the OR of the remaining numbers to the right
            maxOr = Math.max(maxOr, prefixOr | (1L * nums[i] << k) | suffixOr[i + 1]);
            // Update the prefixOr with the current number
            prefixOr |= nums[i];
        }
        // Return the maximum OR value found
        return maxOr;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <cstring>

class Solution {
public:
    long long maximumOr(std::vector<int>& nums, int k) {
        int n = nums.size(); // Get the size of the input vector nums.
        long long suffixOr[n + 1]; // Create an array to store the suffix OR results with an extra element for the boundary conditior
        std::memset(suffixOr, 0, sizeof(suffixOr)); // Initialize the suffixOr array with 0.

        // Build the suffix OR array, where each element has the OR of itself and all elements to its right.
        for (int i = n - 1; i >= 0; --i) {
            suffixOr[i] = suffixOr[i + 1] | nums[i];
        }

        long long maxOr = 0; // Initialize the variable to store the maximum OR value.
        long long prefixOr = 0; // Initialize the prefix OR, which will contain the OR of all elements processed so far.

        // Iterate over all elements in the array.
        for (int i = 0; i < n; ++i) {
            // Calculate the maximum OR value by considering the current prefix OR, the transformed current number,
            // and the suffix OR. The transformation shifts the current number left by 'k' bits.
            maxOr = std::max(maxOr, prefixOr | (static_cast<long long>(nums[i]) << k) | suffixOr[i + 1]);
            // Update the prefix OR to include the current number.
            prefixOr |= nums[i];
        }

        // Return the maximum calculated OR value.
        return maxOr;
    }
};

```

Typescript Solution

```typescript
// Function to calculate the maximum OR value after replacing at most one integer
// in the array nums with any integer between 0 and 2^k - 1, inclusive.
function maximumOr(nums: number[], k: number): number {
    // Initialize the length of the input array.
    const length = nums.length;
    // Initialize the suffix array of type bigint to store cumulative OR from the end.
    const suffixOr: bigint[] = Array(length + 1).fill(0n);

    // Populate the suffixOr array with cumulative OR values starting from the end.
    for (let i = length - 1; i >= 0; i--) {
        suffixOr[i] = suffixOr[i + 1] | BigInt(nums[i]);
    }

    // Initialize the variables to store the current answer and the prefix OR.
    let [maxOrValue, prefixOr] = [0, 0n];

    // Iterate through the nums array to find the maximum OR value.
    for (let i = 0; i < length; i++) {
        // Update the maxOrValue with the maximum OR value achievable by:
        // - Taking the current prefix OR
        // - ORing with the current number shifted left by k bits
        // - ORing with the suffix OR of the elements to the right
        maxOrValue = Math.max(
            maxOrValue,
            Number(prefixOr | (BigInt(nums[i]) << BigInt(k)) | suffixOr[i + 1])
        );
        // Update the prefix OR with the current number.
        prefixOr |= BigInt(nums[i]);
    }

    // Return the maximum OR value found.
    return maxOrValue;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n)`. This is because there is a single loop that iterates over the array `nums` of size `n` backwards to build the `suf` array and another loop that goes forward through `nums` to compute `ans`. Both loops have iterations that perform a constant number of operations, therefore the time complexity remains linear in relation to the size of `nums`.

The space complexity of the code is `O(n)`. Additional space is used for the `suf` array which is of size `n + 1`. Since this is proportional to the size of the original array `nums`, the space complexity scales linearly with the input size, hence `O(n)`.