2897. Apply Operations on Array to Maximize Sum of Squares

Hard Greedy Bit Manipulation Array Hash Table Leetcode Link

Problem Description

In the given problem, we have an array of integers referred to as `nums`, and a positive integer `k`. The task is to perform operations on the elements of this array to ultimately select `k` elements from it and maximize the sum of their squares.

The operation that can be performed involves choosing any two distinct indices `i` and `j` in the array and replacing `nums[i]` with the result of the bitwise AND (`&`) of `nums[i]` and `nums[j]`, while `nums[j]` is replaced with the result of the bitwise OR (`|`) of the same elements. The important thing to notice here is that the AND operation tends to reduce the numbers by turning 1s to 0s when the corresponding bits are different, while the OR operation does the opposite. However, once a bit is turned to 0 by the AND operation, it cannot be reverted back to 1.

After performing these operations any number of times, we are supposed to take `k` numbers from the resulting array and calculate the sum of their squares. The goal is to find out how to perform these operations such that this sum is maximized. Finally, the sum of squares should be returned modulo `10^9 + 7`, due to the possibility of the sum becoming very large.

Intuition

Understanding how bitwise operations work is crucial to solving this problem. When performing the AND operation between two numbers, the bits that are different (where one is 1 and the other is 0) will result in 0. Conversely, the OR operation will produce a 1 bit in places where at least one of the two bits is 1.

The strategy to maximize the sum of squares is to have the numbers with the highest set bits included in the final selection of `k` elements, because these will give larger squares due to the exponential growth of value with each additional higher-order bit. This means that if a bit is set at a higher position, it contributes more significantly to the value of a number than a bit set at a lower position.

To systematically achieve this, we can track the frequency of set bits at each bit position in all numbers of the array. We then construct the `k` largest numbers by taking the most significant set bit (if available) from the bit frequency array, and then proceed to the next most significant bit, fulfilling the count for `k` numbers. With each selection, we decrement the count of set bits because we have utilized that set bit. Finally, we square these numbers and keep a cumulative sum, which is returned modulo `10^9 + 7`.

By doing this, we ensure we have the largest possible numbers that can be formed after the allowed operations, hence maximizing the sum of their squares.

Solution Approach

The solution implements a greedy algorithm, combined with bit manipulations, to construct the largest possible numbers from the available set bits.

First, we initialize a counter array `cnt` of size 31, which corresponds to bits from `0` to `30`, taking into account that the maximum number within 32-bit signed integer range would not exceed 2^31. This array is used to count how many times each bit is set among all the numbers in the input `nums`.

We iterate over every number `x` in the input array and for each bit `i` from `0` to `30` (since we are using 0-indexed bit positions), we check if the `i`-th bit of `x` is set by shifting `x` to the right by `i` positions and using bitwise AND with `1`. If the result is `1`, it means the bit is set, and we increment the count of `i`-th bit in `cnt`.

After preparing the `cnt` array, we are ready to select `k` largest numbers. We do this `k` times:

- Initialize `x` to `0` before each of the `k` selections. This `x` will be the construction of one of the `k` largest possible numbers.
- Iterate through the bit positions from the most significant to the least significant (`30` to `0`), and check if we have any remaining set bits stored in `cnt` at the current position `i`. If we do, we set this bit in `x` (`x |= 1 << i`) meaning we contribute this bit to make our number larger. We then decrement the count of set bits at this position since we've used one.
- Once we have constructed the number `x`, we add its square to `ans` keeping it modulated with `mod = 10**9 + 7` to handle large numbers and prevent integer overflow issues.

By the end of the `k` iterations, we have accumulated the maximum sum of squares possible under the given constraints and operations.

In summary, the data structures used in this algorithm are:

- An array `cnt` to keep track of the count of set bit frequencies.
- Variables `x` and `ans` to store the current number being constructed and the final answer, respectively.

This algorithm efficiently utilizes bit manipulations and a greedy approach to solve the problem in a way that scales well with larger inputs.

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Suppose we have an array `nums = [3, 8, 2]` and `k = 2`. The binary representations of `nums` are `[11, 1000, 10]`, where we can see that `8` has the highest bit set. Our goal is to maximize the sum of squares of any two numbers after performing the described operations.

Following the solution approach:

1. We initialize an array `cnt` of size `31` to track the frequency of set bits at each position. After iterating over `nums`, `cnt` would look like this (showing only bit positions 0 to 3 for brevity, with position 3 being the most significant bit):

```
Position:  0  1  2  3
Count:     1  1  0  1
```

This indicates that we have one `1` at positions `0`, `1`, and `3` across all numbers.

2. Now we construct the largest number possible. With `k = 2`, we do this twice:
    - For the first number, we start with our `x = 0` and scan from bit 30 to 0. Reaching position 3, we see `cnt[3] = 1`, so we can set this bit on our `x` yielding `x = 1000`. We decrement `cnt[3]` by 1 since we've used this set bit.
    - For the second number, we do the same but, since we've used the set bit at position 3, it's no longer available. Thus, we start with `x = 0`, reach position 1, set it since `cnt[1] = 1`, and now `x = 10`.
3. We have two numbers `8` (binary `1000`) and `2` (binary `10`), and we square them and add them together keeping modulo `10^9 + 7`.

```
8^2 + 2^2 = 64 + 4 = 68
```

The maximum sum of squares we can get is `68`. However, considering the operation described, we would realize that no operation can increase the value of `8` since it's the maximum, and performing an AND operation with `2` or `3` would result in a number smaller than or equal to `2`. Hence, our starting array `nums = [3, 8, 2]` is already optimized for selecting `k = 2` elements to maximize the sum of their squares.

By applying this approach to the actual problem, we'd be able to find the maximum sum of squares of `k` numbers from a larger array, with more complex combinations of set bits, while ensuring we're using a greedy method to obtain the largest numbers possible after performing the allowed operations.

Python Solution

```python
class Solution:
    def maxSum(self, nums: List[int], k: int) -> int:
        # Define the modulus variable for large number handling as per the problem statement
        mod = 10**9 + 7

        # Initialize a list to count the number of set bits at each bit position
        bit_count = [0] * 31  # Assuming 32-bit integers, ignoring the sign bit

        # Count the number of set bits for each bit position in all numbers
        for num in nums:
            for i in range(31):  # Traverse through all the bits
                if num >> i & 1:  # Check if the ith bit is set
                    bit_count[i] += 1  # Increment count for this bit position

        # Initialize variable to store the sum of squares of chosen numbers
        max_sum = 0

        # Loop to choose 'k' numbers
        for _ in range(k):
            chosen_num = 0  # This will store the current chosen number with maximum set bits

            # Create the maximum number by setting the highest bits where possible
            for i in range(31):
                if bit_count[i]:  # If there are bits to set at position i
                    chosen_num |= 1 << i  # Set the bit at position i
                    bit_count[i] -= 1  # Decrement count as this bit is now used up

            # Add the square of the chosen number to max_sum and take modulo
            max_sum = (max_sum + chosen_num * chosen_num) % mod

        # Return the computed max_sum
        return max_sum

```

Java Solution

```java
class Solution {
    public int maxSum(List<Integer> nums, int numIterations) {
        final int MODULO = (int) 1e9 + 7; // Define the modulo to prevent integer overflow.
        int[] bitCounts = new int[31]; // Array to keep track of the count of 1 bits at each position.

        // Count the number of 1 bits at each position for all numbers in nums.
        for (int num : nums) {
            for (int i = 0; i < 31; ++i) {
                if ((num >> i & 1) == 1) {
                    ++bitCounts[i];
                }
            }
        }

        long answer = 0; // Initialize the answer which will accumulate the value.

        // Perform calculations for 'numIterations' iterations.
        while (numIterations-- > 0) {
            int x = 0; // Initialize x which will hold the current max value.

            // Create x by setting bit i if there is any number with a 1 bit at position i.
            for (int i = 0; i < 31; ++i) {
                if (bitCounts[i] > 0) {
                    x |= 1 << i; // Set bit i.
                    --bitCounts[i]; // Decrement the count for bit i.
                }
            }

            // Add square of the created number x to the answer.
            answer = (answer + (long) x * x) % MODULO;
        }

        // Cast the final answer to int and return it.
        return (int) answer;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the maximum sum of squares of k integers
    // constructed by bitwise OR of different subsets.
    int maxSum(vector<int>& nums, int k) {
        // Initialize a counter array to store the number of 1's at each bit position.
        int bitCount[31] = {};

        // Count the number of 1's at each bit position for all numbers.
        for (int num : nums) {
            for (int bit = 0; bit < 31; ++bit) {
                if (num & (1 << bit)) {
                    ++bitCount[bit];
                }
            }
        }

        // Initialize the answer as a long long to avoid potential overflow.
        long long answer = 0;
        // Constant mod to take answer modulo 1e9+7 for large sums.
        const int mod = 1e9 + 7;

        // Loop k times to construct k integers.
        while (k--) {
            // Variable to store the currently constructed integer using bitwise OR.
            int currentInt = 0;
            // For each bit position from 0 to 30, we check if we still have numbers left
            // with this bit set to 1. If so, we include this bit in our current integer.
            for (int bit = 0; bit < 31; ++bit) {
                if (bitCount[bit]) {
                    // Set the bit position in the current integer.
                    currentInt |= (1 << bit);
                    // Decrement the count of available 1's at this bit position.
                    --bitCount[bit];
                }
            }
            // Add the square of the current integer to 'answer' and take modulo.
            answer = (answer + static_cast<long long>(currentInt) * currentInt) % mod;
        }

        // Cast the result back to int as per function return type.
        return static_cast<int>(answer);
    }
};

```

Typescript Solution

```typescript
function maxSum(nums: number[], k: number): number {
    // Initialize a count array with 31 zeros, one for each bit position (assuming 32-bit integers)
    const bitCounts: number[] = Array(31).fill(0);

    // Count the number of 1s at each bit position for all numbers in the array
    for (const num of nums) {
        for (let i = 0; i < 31; ++i) {
            if ((num >> i) & 1) {
                bitCounts[i]++;
            }
        }
    }

    // Initialize the answer as a BigInt to support potential large numbers resulting from squaring
    let answer = 0n;

    // Define modulo constant to prevent overflow
    const mod = BigInt(1e9 + 7);

    // Loop to perform calculation k times
    while (k-- > 0) {
        let constructedNumber = 0;

        // Construct the largest number possible using the available bit counts
        for (let i = 0; i < 31; ++i) {
            if (bitCounts[i] > 0) {
                constructedNumber |= 1 << i; // Set the ith bit
                bitCounts[i]--; // Decrement the bit count at ith position
            }
        }

        // Add the square of the constructed number to the answer, and apply modulo operation
        answer = (answer + BigInt(constructedNumber) * BigInt(constructedNumber)) % mod;
    }

    // Return the answer as a number after casting from BigInt
    return Number(answer);
}

```

Time and Space Complexity

Time Complexity

The given code consists of two main parts which contribute to the time complexity:

1. The first `for` loop iterates through all the elements in `nums` and for each element, it loops a constant 31 times to count the number of 1s at each bit position. Suppose there are `n` elements in the input `nums`. The time complexity for this part is `O(n * 31)`. Since 31 is a constant, we simplify this to `O(n)`.

2. The second `for` loop runs `k` times, where `k` is an input parameter that does not depend on the size of `nums`. Inside this `k`-loop, we have another loop that runs 31 times for the bit counting similar to the first part. The time complexity for this part is `O(k * 31)`. Again, since 31 is a constant, this is reduced to `O(k)`.

Since `k` is independent of `n`, and both loops do not nest but run in sequence, we sum their complexities getting `O(n) + O(k)`. This indicates that as `n` or `k` grows, the runtime will increase linearly with whichever is larger. However, if we consider the value of the elements in `nums`, specifically `M`, the maximum value, we recognize that the bit-operations like `x >> i & 1` depend on the maximum number of bits used to represent the numbers in the array. This introduces the term `log M`, and therefore the overall time complexity considering the bit-manipulations would be `O(n * log M)`.

Space Complexity

For space complexity:

1. We only need extra space for the `cnt` array which stores the bit counts for each of the 31 possible bit positions (assuming 32-bit integers). The size of this array is constant hence contributing `O(31)`, which simplifies to `O(1)` space complexity.

2. Besides the `cnt` array, the code uses a fixed number of integer variables. These are also constant space and do not scale with input size `n` or the value `k`.

Given that these values do not depend on the size of the input array or the elements within, the overall space complexity of the code is `O(1)`.

In conjunction with the reference answer, the time complexity is represented as `O(n * log M)` and the space complexity as `O(log M)`, but as analyzed, the correct space complexity should be `O(1)` since the only non-constant space used is the `cnt` array which has a fixed size regardless of the input.