2859. Sum of Values at Indices With K Set Bits

Easy Bit Manipulation Array Leetcode Link

Problem Description

You have an array called `nums`, and each element in the array is associated with an index from 0 to the length of the array minus 1. Your goal is to calculate the sum of elements whose indexes have a specific number of bits set to 1 when viewed in binary. This specific number of set bits is given by the integer `k`.

For instance, if `k` is 3 and you're looking at the index 7 (which is `0111` in binary), since 7 has three set bits, you would include the value of `nums[7]` in your sum. You repeat this process for all indices in the `nums` array and return the total sum.

Intuition

To solve this problem efficiently, you can iterate over each index of the `nums` array while simultaneously checking how many set bits are in the binary representation of that index. If the count of set bits is equal to `k`, then you add the value at that index to your running total sum.

For the implementation:

1. You iterate through `nums` array using `enumerate` to get both index (`i`) and value (`x`).
2. For each index `i`, you check how many set bits it has by using the built-in Python method `bit_count()`.
3. If `i.bit_count()` is equal to `k`, you include the value `x` in your sum.
4. You aggregate these values to determine the final answer, which is the sum of all the numbers at indices with `k` set bits.

This solution is direct and leverages Python's built-in functions to handle binary operations efficiently, avoiding the need for manual bit-counting or more complex bitwise manipulation.

Solution Approach

The solution uses a simple linear scan algorithm which is quite effective for this problem. No complex data structures or patterns are necessary.

Here is a step-by-step breakdown of the implementation using the provided solution code as a reference:

1. We use a `for` loop in combination with `enumerate` to iterate over the `nums` array. This gives us access to both the index and the value at that index.
2. For each index `i`, we check the number of set bits in its binary representation. This is done using the `bit_count()` method.
3. We include the value `x` if the number of set bits (1s in the binary form) of the index `i` is equal to `k`. The condition `i.bit_count() == k` evaluates to `True` or `False` based on whether the number of set bits matches `k`.
4. We use the built-in `sum` function to add up all the values `x` for which the corresponding index `i` satisfies the condition mentioned above.

In terms of algorithms and data structures, this solution can be categorized as a brute-force approach since it evaluates the condition for every element in the array without using a more nuanced algorithm or data structure to optimize the process. However, since the problem has to do with checking each index individually, the brute-force method is appropriate and efficient in this context, especially since Python's `bit_count()` method is optimized internally.

The `sum` expression employs a generator expression, which is more memory-efficient than building an intermediary list of elements to sum, as it calculates the sum on the fly.

In short, the solution is elegant, taking advantage of Python's language features to achieve the goal with minimal code.

Example Walkthrough

Let's take an example where `nums = [0, 1, 2, 4, 8, 16, 3, 5]` and `k = 2`. We want to find the sum of elements in `nums` where the indices have exactly two bits set to 1 when viewed in binary.

1. We start with the first element, `nums[0]` which is `0`. The index `0` has 0 bits set to 1 when viewed in binary (`0b0`), so it does not match `k`.
2. Move to the second element, `nums[1]` which is `1`. The index `1` has 1 bit set to 1 in binary (`0b1`), so it also does not match `k`.
3. Continue to `nums[2]` which is `2`. The index `2` in binary is `0b10`, which has `1` bit set, not matching `k`.
4. Next is `nums[3]` at index `3`, with a value of `4`. Binary `3` is `0b11`, which has `2` bits set. This matches `k`, so we include `4` in our sum.
5. Move to `nums[4]`, value `8` and index `4`(`0b100`). Index `4` has only `1` bit set, so we do not add `8` to our sum.
6. For `nums[5]`, value `16` at index `5`(`0b101`), the binary representation has `2` set bits, matching `k`. We include `16`.
7. At `nums[6]`, index `6` (`0b110`) has `2` bits set. Since `k` is `2`, we add value `3` to our sum.
8. Lastly, `nums[7]` is `5`, with index `7` (`0b111`), which has `3` set bits, so it's not included.

Now, we sum up the included numbers: `4 + 16 + 3 = 23`.

So the final sum of the values in the `nums` array at indices with exactly `k` set bits is `23`.

Using the provided solution approach:

1. We iterate over `nums` with the index and value.
2. For each index, we use `bit_count()` to count set bits.
3. We check if the count matches `k`
4. We conclude with the sum of values that match the condition.

In this example, we can see how the algorithm steps through each element and uses the `bit_count()` function to filter and sum the correct elements efficiently.

Python Solution

```python
# Import the typing module to use the List type for type hinting
from typing import List

class Solution:
    def sum_indices_with_k_set_bits(self, nums: List[int], k: int) -> int:
        # Initialize the variable to store the sum of indices
        sum_of_indices = 0

        # Loop through each index and value in the list of numbers
        for index, value in enumerate(nums):
            # If the number of set bits (1s) in the binary representation of the index
            # is equal to k, add the current index to the sum_of_indices
            if bin(index).count("1") == k:
                sum_of_indices += value

        # Return the final sum of indices with exactly k set bits
        return sum_of_indices

```

Java Solution

```java
public class Solution {

    /**
     * This function calculates the sum of the elements in the nums array at indices that have exactly k bits set to 1 in their binar
     * @param nums An array of integers.
     * @param k The number of set bits (bits set to 1) desired in the index's binary representation.
     * @return The sum of the elements at indices with exactly k set bits.
     */
    public int sumIndicesWithKSetBits(int[] nums, int k) {
        int sum = 0; // Initialize the sum to 0.

        // Iterate through the nums array.
        for (int index = 0; index < nums.length; ++index) {
            // If the current index has exactly k set bits, add the corresponding element to the sum.
            if (bitCount(index) == k) {
                sum += nums[index];
            }
        }

        return sum; // Return the final sum.
    }

    /**
     * This helper function counts the number of set bits (bits set to 1) in the binary representation of a number.
     * @param number The number whose set bits are to be counted.
     * @return The count of set bits in the number.
     */
    private int bitCount(int number) {
        int count = 0; // Initialize the count of set bits to 0.

        // Continue until all bits are processed.
        while (number != 0) {
            // Clear the least significant bit set (Brian Kernighan's Algorithm).
            number &= (number - 1);
            count++; // Increment the count for each bit cleared.
        }

        return count; // Return the count of set bits.
    }
}

```

C++ Solution

```cpp
#include <vector> // Include the necessary header for the vector type

class Solution {
public:
    // Function to sum the elements in 'nums' at indices with 'k' bits set
    int sumIndicesWithKSetBits(vector<int>& nums, int k) {
        int sum = 0; // Initialize the sum result to 0

        // Iterate over all indices of the 'nums' vector
        for (int index = 0; index < nums.size(); ++index) {
            // Use the __builtin_popcount function to count the number of set bits in 'index'
            // Check if the number of set bits is equal to 'k'
            if (__builtin_popcount(index) == k) {
                sum += nums[index]; // If so, add the value at this index to 'sum'
            }
        }

        return sum; // Return the final sum of desired elements
    }
};

```

Typescript Solution

```typescript
/**
 * This function calculates the sum of the elements in the nums array at indices that have exactly k bits set to 1 in their binary re
 * @param nums - An array of numbers.
 * @param k - The number of set bits (bits set to 1) desired in the index's binary representation.
 * @returns The sum of the elements at indices with exactly k set bits.
 */
function sumIndicesWithKSetBits(nums: number[], k: number): number {
    let sum = 0; // Initialize the sum to 0.

    // Iterate through the nums array.
    for (let index = 0; index < nums.length; ++index) {
        // If the current index has exactly k set bits, add the corresponding element to the sum.
        if (bitCount(index) === k) {
            sum += nums[index];
        }
    }

    return sum; // Return the final sum.
}

/**
 * This helper function counts the number of set bits (bits set to 1) in the binary representation of a number.
 * @param number - The number whose set bits are to be counted.
 * @returns The count of set bits in the number.
 */
function bitCount(number: number): number {
    let count = 0; // Initialize the count of set bits to 0.

    // Continue until all bits are processed.
    while (number) {
        // Clear the least significant bit set (Brian Kernighan's Algorithm).
        number &= number - 1;
        count++; // Increment the count for each bit cleared.
    }

    return count; // Return the count of set bits.
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided method `sumIndicesWithKSetBits` primarily depends on the number of elements in `nums` and the computation of `bit_count` for each index.

- Iterating over all indices of the `nums` list: If there are `n` elements in `nums`, this requires `O(n)` time.
- For each index `i`, the `bit_count()` operation is performed, which takes `O(log(i))` time since it counts the number of set bits in the binary representation of the index.

Since `i` ranges from `0` to `n-1`, in the worst-case scenario, `bit_count()` will be called with `i = n-1`, which takes `O(log(n-1))` time. Therefore, the combined time complexity for the computation over all indices would be `O(n log(n))`.

Overall, the **Time Complexity** is `O(n log(n))`.

Space Complexity

The space complexity is fairly straightforward to analyze:

- The sum function generates a temporary generator expression and does not store values, thus it uses constant space.
- No additional data structures are used that scale with the input size.

Therefore, the **Space Complexity** is `O(1)` — constant space complexity, since extra space does not depend on the input size.