2172. Maximum AND Sum of Array

Hard | Bit Manipulation | Array | Dynamic Programming | Bitmask | Leetcode Link

Problem Description

The given problem involves an array of integers `nums` and an integer `numSlots`. The size of the array `nums` is denoted by `n`, and the condition is that twice the number of `numSlots` should be greater than or equal to `n`. This implies we have `numSlots` where each slot can hold at most two numbers from `nums`.

The objective is to place all the integers in `nums` into the `numSlots` in a way that maximizes the "AND sum". The AND sum is calculated by performing a bitwise AND operation between each number and the number of the slot it is placed in and then summing these values together. Since each slot can hold two numbers, the maximum AND sum will depend on how the numbers are distributed across the slots.

The problem asks us to return the maximum possible AND sum given the numbers in `nums` and the available `numSlots`.

Intuition

The solution involves using dynamic programming to calculate the maximum AND sum. Our dynamic programming state is defined by `f`, which is an array representing all possible placements of the numbers in `nums` into the `numSlots`. More specifically, the index `i` in `f[i]` represents a bitmask, where each bit indicates whether a slot has been filled or not (and if so, by how many numbers). The size of the array `f` is `1 << m`, where `m` is twice the number of `numSlots`, because each slot may be filled with 0, 1, or 2 numbers.

Here's the approach to arriving at the solution:

1. Initialize the `f` array with zeros, which will store the maximum AND sum for each state (each possible way to fill the slots).
2. Iterate over all possible states `i` of the `f` array, which are represented as bitmasks. We use the number of bits set in `i` to determine how many numbers have been placed already (`cnt`).
3. Skip the iterations where `cnt` exceeds `n`, as this would represent an invalid placement where more numbers are placed than available.
4. For each state `i` and for each possible slot `j`, if that slot (`j`) is occupied in this state (`i >> j & 1` is true), then we calculate a new AND sum by:
   - Removing the number from slot `j` (`i ^ (1 << j)`) to find the previous state.
   - Adding the AND operation of the removed number (`nums[cnt - 1]`) and its respective slot number (`j // 2 + 1`).
5. For each state, we maximize `f[i]` with the newly calculated AND sum if it's greater than the current `f[i]`.
6. Continue this process until all states have been evaluated.
7. The maximum AND sum will be the maximum value in the `f` array after evaluating all states.

The intuition behind this approach is that we explore each possible way to place numbers into slots, track what the AND sum would be for each configuration, and use dynamic programming to efficiently compute the maximum possible AND sum we could get from such placements.

Solution Approach

The implementation of the solution uses a dynamic programming approach to solve the problem. Here's how it works step by step referring to the solution provided:

1. **Initialization**: Create an array `f` that will store the maximum AND sum for every bitmask state. As there are `numSlots` which can each contain up to two numbers, the array size is `1 << (2 * numSlots)`. The bitmask has `2 * numSlots` bits because each slot can be in one of three states - empty, with one number, or with two numbers.

```
f = [0] * (1 << m)
```

2. **Iterating Over States**: Iterate over all possible states of placing `n` numbers in `numSlots`. Each state is represented as a bitmask `i` of size `m`.

```
for i in range(1 << m):
```

3. **Bit Counting**: Calculate how many bits (numbers) have been placed already using the `bit_count()` method on the bitmask. If this count `cnt` exceeds the length of `nums`, the current bitmask should not be considered as it represents placing more numbers than are available.

```
cnt = i.bit_count()
if cnt > n:
    continue
```

4. **Traversing The Bits in the Bitmask**: For every possible slot `j` (0 to m-1), check if it is used in state `i` and calculate a new sum considering the number placed in that slot.

```
for j in range(m):
    if i >> j & 1:
```

5. **Calculating the AND Sum**: Use the previous state `i ^ (1 << j)` to look up the previous maximum AND sum and add the AND operation between the last-placed number `nums[cnt - 1]` and the slot index `j // 2 + 1`. The slot index is `j // 2 + 1` because `j` represents each possible space in the slots, and the same slot number can appear twice.

```
f[i] = max(f[i], f[i ^ (1 << j)] + (nums[cnt - 1] & (j // 2 + 1)))
```

6. **Maximization Step**: The dynamic programming essence is here, where we continuously maximize the entry `f[i]` with the newly computed AND sum if it's greater. This will ensure that `f[i]` stores the maximum AND sum we can achieve for state `i`.

7. **Getting the Result**: The final answer is the maximum value in the `f` array, which represents the maximum AND sum over all bitmasks representing different ways to fill the slots.

```
return max(f)
```

The algorithm's time complexity is mainly determined by the two nested loops. The outer loop runs `1 << m` times, where `m = 2 * numSlots`, and the inner loop runs `n` times, making it `O(n * 2^m)` in the worst case. Although this seems exponential, the constraint `2 * numSlots >= n` allows it to work because the actual number of iterations is limited.

The approach capitalizes on dynamic programming to store the intermediate results associated with each possible placement state. Through optimal substructure and overlapping subproblems, it ensures that the final array entry `f` contains the maximum possible AND sum given `numSlots` slots.

Example Walkthrough

To illustrate the solution approach, let's take a small example:

Suppose `nums = [1, 2, 3]` and `numSlots = 2`. Since we have 3 numbers, we need `numSlots` such that `2 * numSlots >= 3` -- which holds true in our case (2 slots * 2 numbers per slot = 4 positions available).

1. The bitmask `m` will be twice the number of slots, so `m = 2 * numSlots = 4`. The possible states will be `(0000, 0001, 0010, 0011, ... , 1111)`, representing the empty slots and how they get filled respectively.
2. Initialize `f = [0] * (1 << m)` to store the maximum AND sums.
3. Start iterating over all possible states of the bitmask from `0000` to `1111`.
4. For each state `i`, calculate how many 1s are in its binary representation to know how many numbers have been placed. For example, the state `0011` means 2 numbers have been placed already.
5. If `cnt` (count of numbers placed) exceeds the number of available numbers (`3`), this state is skipped because it's not valid.
6. If not skipped, for each slot `j` from 0 to `m - 1`, check if number `j` is already placed. If yes, we look at the state `i ^ (1 << j)` (the state before placing number `j`) to see the maximum AND sum from there.
7. Add to this sum the AND operation (bitwise) of the last number taken and its respective slot number. For instance:
   - If `cnt = 2`, it means that `nums[cnt - 1] = nums[1] = 2`.
   - If we are checking slot `j = 1`, which corresponds to slot number `1 // 2 + 1 = 1`. Therefore, we perform `2 & 1`.
8. Set `f[i]` to the maximum of its previous value and the new calculated AND sum.
9. After finishing the iteration for all states, `max(f)` will give us the maximum AND sum.

Here, let's calculate it step by step:

- For `0010`, `cnt = 1`. We place `nums[0] = 1` in slot `1` (`j = 1` means slot number `1`). The AND sum is `1 & 1 = 1`. So `f[0010] = 1`.
- For `0100`, `cnt = 1`. We place `nums[0] = 1` in slot `2` (`j = 2` means slot number `2`). The AND sum is `1 & 1 = 1`. So `f[0100] = 1`.
- For `0110`, `cnt = 2`. We have `nums[1] = 2` in slot `1`, with `f[0100]` being the previous state. The AND sum `2 & 1 = 0`, but `f[0100] = 1` from previous placement. So `f[0110] = max(f[0110], f[0100] + 0) = 1`.

Repeat for each state to fill in `f`. The final maximum AND sum can be found as `max(f)`.

In this example, the maximum AND sum will be achieved by placing numbers in such a way that maximizes each bitwise AND with respective slot numbers. After running the full dynamic programming process, we'll get the maximum possible AND sum which would be returned as the answer.

Python Solution

```
class Solution:
    def maximumANDSum(self, nums: List[int], num_slots: int) -> int:
        num_elements = len(nums)  # Number of elements in the nums array
        slot_states = num_slots << 2  # Total number of slots * 2 (for tracking two elements per slot)
        dp = [0] * (1 << slot_states)  # Dynamic programming table store for all possible slot combinations

        # Iterating over all possible combinations of slots
        for state in range(1 << slot_states):
            cnt = bin(state).count('1')  # Count how many slots are already occupied in this state
            if cnt > num_elements:  # Skip states with more occupied slots than available elements
                continue

            # For each slot in the current state
            for slot_bit_index in range(slot_states):
                # Check if the slot is occupied in the current state
                if state & (1 << slot_bit_index):
                    # Calculate new state after freeing the slot
                    new_state = state ^ (1 << slot_bit_index)
                    # Calculate the corresponding AND sum
                    # Add the AND operation of the number and the slot number (1-indexed)
                    dp[state] = max(dp[state], dp[new_state] + (nums[cnt - 1] & (slot_bit_index // 2 + 1)))
                    # Update the dp table if a better AND sum is found
                    dp[state] = max(dp[state], dp[new_state])  # Ensure we keep the max

        # Returning the maximum AND sum from the last state, which includes all elements in num
        return max(dp)
```

Java Solution

```
class Solution {

    // Method to find the maximum AND sum with nums array and given number of slots
    public int maximumANDSum(int[] nums, int numSlots) {
        int numElements = nums.length; // Number of elements in the nums array
        int maxStates = numSlots << 2; // Each slot can hold at most two numbers (states represented with bit manipulation)
        int[] dp = new int[1 << maxStates]; // Dynamic programming table to store maximum AND sum for each possible state
        // Iterate through all possible states (combinations of filled slots)
        for (int state = 0; state < (1 << maxStates); ++state) {
            int countSetBits = Integer.bitCount(state); // Count the number of slots filled in the current state
            // If the count exceeds the number of elements that can be placed
            if (countSetBits > numElements) {
                continue;
            }

            // Iterate through all possible slots to place the current element
            for (int slot = 0; slot < maxStates; ++slot) {
                // Check if the current slot is occupied in the state
                if ((state >> slot & 1) == 1) {
                    // Calculate the new state by removing the current element from the slot
                    int newState = state ^ (1 << slot);
                    // Calculate the AND sum for the current state by adding the AND of the element with half the slot index plus one
                    dp[state] = Math.max(dp[state], dp[previousState] + (nums[count - 1] & ((slot >> 1) + 1)));
                }
            }
            // Update the maximum AND sum found so far
            maxAndSum = Math.max(maxAndSum, dp[state]);
        }
        // Return the maximum AND sum
        return maxAndSum;
    }
}
```

C++ Solution

```
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class Solution {
public:
    // Function to calculate the maximum AND sum for "nums" with "numSlots"
    int maximumANDSum(vector<int>& nums, int numSlots) {
        int numElements = nums.size();
        int totalPositions = numSlots * 2;

        // Each slot can hold two items, so we shift left to get total positions
        int totalStates = 1 << totalPositions;

        // State array to store the maximum AND sum for each combination
        int dp[totalStates];

        // Initialize the state array
        memset(dp, 0, sizeof(dp));

        // Iterate through all possible combinations of slots
        for (int mask = 0; mask < totalStates; ++mask) {
            // Count the number of slots occupied positions)
            int countSetBits = __builtin_popcount(i);

            // If the count exceeds the number of elements, continue to the next iteration
            if (countSetBits > numElements) {
                continue;
            }

            // Iterate through all possible positions
            for (int pos = 0; pos < totalPositions; ++pos) {
                // Check if the current position is occupied in the combination mask
                if ((i >> pos) & 1) {
                    // Update the dp state with the maximum value between the current state and
                    // the state with the position 'pos' removed, plus the AND sum for the current number
                    // and the current slot (which is given by 'position / 2 + 1')
                    dp[i] = max(dp[i], dp[i ^ (1 << pos)] + (nums[countSetBits - 1] & (pos / 2 + 1)));
                }
            }
        }

        // Find and return the maximum AND sum from the state array
        return *max_element(dp, dp + (1 << totalPositions));
    }
};
```

Typescript Solution

```
function maximumANDSum(nums: number[], numSlots: number): number {
    const numCount = nums.length; // Total number of elements in 'nums' array
    const slotMask = numSlots << 1; // Total number of slot masks, as each slot can take up 2 values
    const dp: number[] = new Array(1 << slotMask).fill(0); // Dynamic programming (dp) array to store intermediate results

    // Iterate over all possible combinations of slot allocations
    for (let mask = 0; mask < 1 << slotMask; ++mask) {
        // Count the number of occupied slots in this combination
        const occupiedSlotsCount = mask.toString(2).split('').filter(bit => bit === '1').length;

        // If the number of occupied slots is higher than the number of elements, skip this combination
        if (occupiedSlotsCount > numCount) {
            continue;
        }

        // Iterate over each position to check and update the dp state
        for (let pos = 0; pos < slotMask; ++pos) {
            // Check if the current position is occupied in the combination mask
            if ((mask >> pos) & 1) {
                // Calculate the slot index by right-shifting 'pos' by one and adding one (slot numbers are 1-indexed)
                const slotIndex = (pos >> 1) + 1;
                // Calculate the new mask value by turning the current position's bit off
                const newMask = mask ^ (1 << pos);
                // Calculates the AND sum and updates the dp state if this state is better
                dp[mask] = Math.max(dp[mask], dp[newMask] + (nums[occupiedSlotsCount - 1] & slotIndex));
            }
        }
    }

    // Returns the maximum AND sum of all possible combinations
    return Math.max(...dp);
};
```

Time and Space Complexity

Time Complexity

The given Python code implements a solution to find the maximum AND sum with a given list of numbers and a fixed number of slots in which these numbers can be placed. It uses dynamic programming with a bitmask to represent different states.

To analyze the time complexity:

- For each state of the bitmask `i`, which ranges from 0 to `(1 << m) - 1` (where `m = numSlots * 2`), we iterate through each bitmask `j` which goes from 0 to `m - 1`.
- Within this loop, we perform a constant-time operation, namely the AND operation and comparison between integers.
- The `bit_count()` method is also constant time on average thanks to modern CPU operations (though it could be considered O(log(m)) in some cases, depending on the implementation)
- The total number of states that we iterate over is `1 << m`, and we perform up to `m` operations for each state.

Hence, the time complexity is `O(m * 2^m)` where `m = numSlots << 1`.

Space Complexity

Looking at the space complexity:

- We utilize a list `f` of size `1 << m`, which is the primary space consumption in the algorithm.
- No additional data structures grow with respect to `n` or `m`, apart from constant space for variables.

Thus, the space complexity of the algorithm is `O(2^m)` where `m = numSlots << 1`.