995. Minimum Number of K Consecutive Bit Flips

Hard Bit Manipulation Queue Array Prefix Sum Sliding Window

Problem Description

In this problem, we have a binary array called `nums` with elements only consisting of 1s and 0s. We're also given an integer `k`. A *k-bit flip* means selecting a contiguous subarray of length `k` and flipping all the bits in it, changing all 0s to 1s and all 1s to 0s, simultaneously. The goal is to determine the minimum number of k-bit flips required to turn all elements of the array into 1s. If it's not possible to achieve an array with all 1s, we must return -1.

For example, if our array is `[0,1,0]` and `k` is 2, the minimum number of flips would be one: flip the first two elements to get `[1,0,1]` and the array now contains no 0.

Intuition

Instead of flipping subarrays and keeping track of the entire array after each flip, which could be both time-consuming and memory-intensive, we use a clever approach called **Difference Array**. A difference array, in this context, allows us to track flips made on the array without altering the original array.

Here's the intuition behind the solution:

- As we move through the array from left to right, we only consider flipping when encountering a 0 that needs to be a 1.
- If `nums[i]` is 0, we need to flip the subarray starting from this position of length `k`. Therefore, if `i + k` would go beyond the length of the array, it's impossible to flip all the 0s to 1s, so we should return -1.
- Instead of directly flipping, we use a difference array `d` to record the flips. When we decide to flip a subarray starting at `i`, we increment `d[i]`.
- After making a flip, we should also indicate that the effect of this flip ends right after position `i + k - 1`, so we decrement `d[i + k]`. This helps to ensure that future calculations only consider flips that affect the current position.
- To determine if a bit at position i should be flipped, we accumulate sum `s` from the start of the array to the current position. If `s` is even, the current bit retains its original value, and if odd, it indicates that the current bit has been flipped an odd number of times.
- The parity (even or odd nature) of the sum `s` combined with the current bit `nums[i]` tells us if the current position's bit is effectively 0 or 1 after all previous flips. If it's 0, we need to flip; otherwise, we do not.
- The flip count `ans` increments whenever we flip a subarray, and the sum `s` is updated to reflect the new flip.

By using this strategy, only a linear scan of the array is needed, and the record of flips can be maintained in a space-efficient manner, leading to an optimized solution that avoids the complications of managing multiple overlapping subarray flips. The final return value `ans` gives the minimum number of flips required or -1 if it's impossible.

Solution Approach

The implementation of the solution uses a greedy approach and a difference array to efficiently keep track of the flips required at each position in the array.

Here's a breakdown of how the algorithm is implemented:

1. Initialize a difference array `d` that has the same length as `nums` plus one. This array is used to record the increments and decrements corresponding to flips in the subarrays.
2. Initialize two variables: `ans` to keep the count of the minimum flips needed, and `s` to maintain the cumulative sum of flips up to the current position.
3. Iterate over each element `x` in `nums` using its index `i`. The current value `x` combined with the cumulative sum `s` determines if the current position requires a flip (`x % 2 == s % 2`).
4. If a flip is needed (i.e., the current bit is effectively 0):
   - Check if flipping the subarray starting at `i` and ending at `i + k - 1` would exceed the bounds of the array. If it does, return `-1` since it's impossible to flip all 0s to 1s.
   - Otherwise, record the flip at `i` by incrementing `d[i]` by 1.
   - Indicate the end of the effect of this flip by decrementing `d[i + k]` by 1. This ensures that the flip's influence does not extend beyond the desired subarray.
   - Increment `ans` since a flip has been performed, and also increment the sum `s` by 1 to reflect this flip in future iterations.
5. After processing all elements, return `ans`, which now contains the minimum number of flips required.

It's important to note that the difference array `d` is not a direct representation of the array after flips but a way to efficiently calculate the impact of all flips on any given position as we iterate through the array.

The greedy aspect of this solution lies in the fact that we always perform a flip when necessary and possible without considering subsequent elements, ensuring that we do not do more flips than needed. This approach minimizes the number of flips overall, leading to an optimal solution.

Example Walkthrough

Let's use a simple example to illustrate the solution approach with `nums = [0, 1, 0, 1, 0]` and `k = 3`.

1. Initialize the difference array `d` with a length of `nums.length + 1`, which means `d = [0, 0, 0, 0, 0, 0]`.
2. Initialize `ans = 0` to track the number of flips, and initialize `s = 0` to track the cumulative sum of flips.
3. Start iterating over `nums`:
   - At index `i = 0`, `nums[i]` is 0. `s % 2 == 0`, so the current bit is effectively 0. To make it 1, we flip starting at index `i` and ending at `i + k - 1` (which is index 2).
   - Increment `d[i]` by 1, so `d` becomes `[1, 0, 0, 0, -1, 0]`.
   - Decrement `d[i + k]` by 1; no change here because `i + k` is out of range.
   - Increment `ans` and `s`, so `ans = 1` and `s = 1`.
4. Move to `i = 1`, where `nums[i]` is 1. `s % 2 == 1`, and `nums[i] % 2 == 1`, so the bit is effectively 0 and needs no flip.
5. Continue to `i = 2`, `nums[i]` is 0. `s % 2 == 1`, so the current bit is effectively 1. No flip needed.
6. Go to `i = 3`, `nums[i]` is 1. `s % 2 == 1`, therefore the bit is effectively 0. We flip starting at `i` and ending at `i + k - 1` (which is 5 and out of bounds).
   - We cannot perform this flip. Therefore, return `-1` as it's impossible to make the entire array 1's.

If we had a valid `k` that allowed for all the bits to be flipped within the bounds of the array, we would continue this algorithm until the end, and `ans` would give us the minimum number of flips required.

In this example, due to the impossibility of performing the last flip (it goes out of bounds), the answer is `-1` indicating we cannot achieve an array of all 1's.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def minKBitFlips(self, nums: List[int], k: int) -> int:
        length = len(nums)  # Length of the input array.
        flips = [0] * (length + 1)  # Differential array to track flips.
        total_flips = 0  # Total flips made so far.
        flip_counter = 0  # The aggregated flips affecting the current position.

        # Go through each number in the array.
        for index, val in enumerate(nums):
            flip_counter += flips[index]  # Add current differential flips to the counter.

            # Check if we need to flip the current bit to 1.
            # If the sum of our counter and the value is even, that means it is 0 and we need to flip it.
            if (val + flip_counter) % 2 == 0:
                if index + k > length:  # If flip would go beyond array, it's impossible.
                    return -1

                # We need to flip the current bit and the next k-1 bits.
                # Mark the beginning of flip.
                flips[index] += 1
                # Cancel out the flip after the k bits.
                flips[index + k] -= 1

                flip_counter += 1  # Account for the new flip in our counter.
                total_flips += 1  # Increment our total flips.

        return total_flips  # Return the total number of flips needed.

# Example:
# solution = Solution()
# result = solution.minKBitFlips([0,1,0], 1)  # Output: 2
```

Java

```java
class Solution {

    /**
     * Flips the minimum number of bits (0 -> 1 or 1 -> 0), each time flipping a substring of length k,
     * to make the entire array of bits have a value of 1. If it is not possible, return -1.
     *
     * @param nums The array of bits (0s and 1s) to be flipped.
     * @param k The length of each substring to flip at a time.
     * @return The minimum number of flips needed, or -1 if it is not possible.
     */
    public int minKBitFlips(int[] nums, int k) {
        int length = nums.length; // The length of the input array.
        int[] flips = new int[length + 1]; // Difference array to track flips.
        int totalFlips = 0; // The number of flips made.
        int flipCounter = 0; // Counter to track the effect of flips done so far.

        for (int i = 0; i < length; ++i) {
            flipCounter += flips[i]; // update flip effect from previous operations

            // Check if the current bit is the same as the total flips made (even number of flips).
            if ((nums[i] % 2 == flipCounter % 2)) {
                // If we can't flip the next k bits because we're at the end, return -1.
                if (i + k > length) {
                    return -1;
                }
                // Increment the position where the flip started.
                flips[i]++;
                // Decrement the position right after where the flip ends to nullify the flip effect later.
                flips[i + k]--;
                // We've flipped once more, so increment the flip counter and total flips.
                flipCounter++;
                totalFlips++;
            }
        }
        // Return the total number of flips required to make all bits 1.
        return totalFlips;
    }
}
```

C++

```cpp
class Solution {
public:
    int minKBitFlips(vector<int>& nums, int k) {
        int size = nums.size(); // Get the size of the array
        vector<int> flips(size + 1, 0); // Initialize a difference array to record flips
        int result = 0; // Number of flips made
        int flipCount = 0; // Current cumulative flips when iterating the array

        // Iterate through the array to flip bits
        for (int i = 0; i < size; ++i) {
            flipCount += flips[i]; // Update cumulative flips

            // Check if the current bit is unflipped (odd number of flips needed to flip flips[i])
            if (flipCount % 2 == nums[i]) {
                // If k flips starting from i end beyond the array, it's not possible to flip all bits
                if (i + k > size) {
                    return -1;
                }
                // Apply the flip operation, using a difference array technique
                flips[i] += 1;  // Record a flip at the current position
                flips[i + k] -= 1; // Mark the end of the flipped segment

                // Increment flip counts correspondingly
                flipCount++;
                result++;
            }
        }

        // Return the total flips required to have all 1s in the array
        return result;
    }
};
```

TypeScript

```typescript
// Function to determine the minimum number of K consecutive bit flips required
// to make all the numbers in the array nums to be 1. If it's not possible, return -1.
function minKBitFlips(nums: number[], k: number): number {
    const numLength = nums.length;
    const flipDiff = new Array(numLength + 1).fill(0);
    let flipsRequired = 0;
    let cumulativeFlips = 0;

    // Iterate through the array to decide where to flip.
    for (let index = 0; index < numLength; ++index) {
        cumulativeFlips += flipDiff[index]; // Update the cumulative flips up to the current index

        // Check if the current bit is the same as the number of cumulative flips mod 2,
        // which means it's currently 0 and needs to be flipped.
        if (cumulativeFlips % 2 === nums[index] % 2) {
            // If flipping k bits starting from the current index goes out of bounds, return -1.
            if (index + k > numLength) {
                return -1;
            }

            // Perform the flip at the current index and record the flip differential.
            flipDiff[index]++;
            flipDiff[index + k]--;
            cumulativeFlips++; // Include the current flip in the cumulative count.
            flipsRequired++;  // Increment the count of flips required.
        }
    }

    // Return the total number of flips required to make all bits 1.
    return flipsRequired;
}
```

```python
from typing import List

class Solution:
    def minKBitFlips(self, nums: List[int], k: int) -> int:
        length = len(nums)  # Length of the input array.
        flips = [0] * (length + 1)  # Differential array to track flips.
        total_flips = 0  # Total flips made so far.
        flip_counter = 0  # The aggregated flips affecting the current position.

        # Go through each number in the array.
        for index, val in enumerate(nums):
            flip_counter += flips[index]  # Add current differential flips to the counter.

            # Check if we need to flip the current bit to 1.
            # If the sum of our counter and the value is even, that means it is 0 and we need to flip it.
            if (val + flip_counter) % 2 == 0:
                if index + k > length:  # If flip would go beyond array, it's impossible.
                    return -1

                # We need to flip the current bit and the next k-1 bits.
                # Mark the beginning of flip.
                flips[index] += 1
                # Cancel out the flip after the k bits.
                flips[index + k] -= 1

                flip_counter += 1  # Account for the new flip in our counter.
                total_flips += 1  # Increment our total flips.

        return total_flips  # Return the total number of flips needed.

# Example:
# solution = Solution()
# result = solution.minKBitFlips([0,1,0], 1)  # Output: 2
```

Time and Space Complexity

The time complexity of the given code can be considered as `O(n)` where `n` is the length of the input list `nums`. This is because the code iterates through the list exactly once, and all operations inside the loop can be done in constant time `O(1)` without any nested loops.

The space complexity of the code is `O(n)` due to the extra list `d` which has the length of `n+1`. Apart from the variables and the loop index, no additional space that depends on the size of the input is used.