2379. Minimum Recolors to Get K Consecutive Black Blocks

Easy String Sliding Window

Problem Description

In this problem, we have a string `blocks` composed of characters `'W'` and `'B'`, which represent white and black blocks, respectively. This string is indexed starting at 0, meaning the first element is at index 0. Our objective is to find the minimum number of operations required to ensure that there is at least one segment in the string consisting of `k` consecutive black blocks. An operation consists of changing a white block into a black block.

Imagine we have a block arrangement like "WBWBBW" and we want at least 3 consecutive black blocks. We need to determine the fewest changes needed to reach that goal; in this case, it's 1 operation by changing the second block from white to black resulting in "WBBWBB".

The problem specifies two key elements central to our task:

- We need to count the number of operations, which correlates to the number of white blocks (`'W'`) that we need to recolor into black blocks (`'B'`).
- We are seeking to create a continuous segment of `k` consecutive black blocks.

Intuition

To arrive at the solution, we need to consider a sliding window approach. The reason a sliding window works well here is that we're interested in a continuous sequence of blocks — specifically, subsequences of length `k`. As we slide the window across the string, we'll look at the number of white blocks within each window, because those are our candidates for recoloring.

Initial thought might involve checking every possible window of `k` blocks in the string, and for each window, counting the number of white blocks and keeping track of the minimum count seen. But this is inefficient because we would be doing repetitive counting. A more optimized approach involves only doing the counting once and then updating the count as the window slides.

The solution code implements this efficient approach:

- First, it counts the number of white blocks within the first `k` blocks. This is our initial answer — the worst-case number of operations, if the first `k` blocks contain the most white blocks.
- Then, the algorithm enters a loop where it slides the window one block at a time to the right. As the window slides, it adjusts the count by adding one if a white block enters the window on the right, and subtracting one if a white block leaves the window on the left.
- After each adjustment, it updates the answer with the new count if it's lower than the previous answer.

This allows us to go through the entire `blocks` string only once (after the initial count), adjusting the white block count and minimum number as we slide the window. The result is the minimum number of operations needed to ensure a sequence of `k` black blocks.

Solution Approach

The implementation of this solution uses a simple but effective pattern known as the sliding window technique. Here's a step-by-step walkthrough of the algorithm, as implemented in the provided solution code:

1. Initialize an integer variable `ans` (short for answer) to hold the minimum number of recoloring operations needed and another variable `cnt` (short for count) to keep track of the number of white blocks within the current window. Calculate the number of white blocks in the first window of size `k` using the `count` method on the substring `blocks[:k]`. Set `ans` to this initial count.
2. Use a loop to iterate through the string `blocks`, starting from the `k`th index up to the end of the string (not inclusive). This loop shifts the fixed-size window one block to the right with each iteration.
3. As the window slides, update `cnt`:
   - Increment `cnt` by 1 if the new block coming into the right side of the window is a white block (`blocks[i] == 'W'`).
   - Decrement `cnt` by 1 if the block that is sliding out of the left side of the window is a white block (`blocks[i - k] == 'W'`).
4. After updating `cnt`, compare it with the current value of `ans`. If `cnt` is smaller (which means the window now contains fewer white blocks than any window seen before), update `ans` to `cnt`.
5. After the loop has iterated over the whole string, `ans` holds the minimum number of operations required. Return `ans`.

No complex data structures are needed for this approach; only integer counters are used. The algorithm runs in linear time, O(n), where n is the length of the `blocks` string. It only requires a constant amount of additional space, resulting in O(1) space complexity.

By maintaining the count `cnt` as the window slides over the string, the solution avoids the need to recount white blocks in each new window. This is much more efficient than a naive approach where you might recount white blocks for every new window of size `k`.

Example Walkthrough

Let's illustrate the solution approach with a smaller example. Consider the string `blocks = "WWBWB"` and let's say we want at least `k = 3` consecutive black blocks.

Following the steps outlined in the solution approach:

1. **Initialize variables and count white blocks in the first window**: The variable `ans` is set to the number of white blocks in the first window of size 3, which is `blocks[:3] = "WWB"`. This has 2 white blocks, so `ans = 2`. We also initialize the count variable `cnt = 2`.
2. **Iterate through `blocks` with a sliding window**: We start our loop from index `3`, which is the fourth block in `blocks`, and go up to the end of the string.
3. **Update count and answer**:
   - At index `3`, the block is `'W'`, so no new black block is added to the window. Since we're not adding a white block on the right, `cnt` remains `2`.
   - However, we should also slide the window, which means we remove the leftmost block from our initial count. The leftmost block within our initial window is white (`blocks[0] = 'W'`), which means we need to decrement `cnt` by 1. Now, `cnt = 1`.

   Therefore, after the first iteration, the window looks at `"WBW"`, with `cnt = 1`.
4. **Update the answer**: Our new `cnt` is less than the previous `ans`, so we set `ans` to `cnt`. Now, `ans = 1`.
5. **Final return value**: In the next and final iteration, the window would look at `"BWB"`, and since there's 1 white block entering the window on the right (`blocks[4] == 'W'`), `cnt` is incremented back to `2`. However, the block that leaves on the left (`blocks[1]`) is a black block, so `cnt` remains `2`. There's no need to update `ans` since `cnt` is not smaller than `ans`.

Following this process, we've completed our sweep of the block string, and `ans` holds the value `1`, which represents the minimum number of operations required to ensure there is a segment of `3` consecutive black blocks in the string.

Thus, the output would be `1` because we need to change one white block to achieve a segment of `3` consecutive black blocks (`"WWBWB"` would become `"WWBBB"`).

Solution Implementation

Python

```python
class Solution:
    def minimumRecolors(self, blocks: str, k: int) -> int:
        # Initial count of 'W' in the first window of size k
        white_count = blocks[:k].count('W')
        # Initialize minimum white blocks to be recolored with the count from the first window
        min_recolors = white_count

        # Slide the window of size k through the blocks string
        for index in range(k, len(blocks)):
            # If the newly included block in the window is white, increment the count
            if blocks[index] == 'W':
                white_count += 1
            # If the block that is exiting the window is white, decrement the count
            if blocks[index - k] == 'W':
                white_count -= 1
            # Update the minimum if the current count is less than the previous minimum
            min_recolors = min(min_recolors, white_count)

        # Return the minimum number of white blocks that need to be recolored
        return min_recolors
```

Java

```java
class Solution {
    // Method to find the minimum number of recolors needed to get at least k consecutive black blocks
    public int minimumRecolors(String blocks, int k) {
        // Initialize the count of white blocks within the first window of size k
        int whiteCount = 0;
        for (int i = 0; i < k; ++i) {
            if (blocks.charAt(i) == 'W') {
                whiteCount++;
            }
        }

        // Initialize the answer with the count of white blocks within the first window
        int minRecolors = whiteCount;

        // Slide the window of size k across the string and update the minimum recolors requird
        for (int i = k; i < blocks.length(); ++i) {
            // If the entering character is white, increment white count
            if (blocks.charAt(i) == 'W') {
                whiteCount++;
            }
            // If the exiting character is white, decrement white count
            if (blocks.charAt(i - k) == 'W') {
                whiteCount--;
            }
            // Update the minimum recolors if the current count is less than the previous minimum
            minRecolors = Math.min(minRecolors, whiteCount);
        }

        // Return the minimum number of recolors required
        return minRecolors;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to find the minimum number of recolors needed
    // to get at least 'k' consecutive black blocks
    int minimumRecolors(string blocks, int k) {
        // Count the number of white blocks in the first 'k' block segment
        int whiteBlockCount = count(blocks.begin(), blocks.begin() + k, 'W');

        // The minimum recolors needed is initially set to the number
        // of white blocks in the first window of size 'k'
        int minRecolors = whiteBlockCount;

        // Iterate over the blocks starting from the 'k'th block
        for (int i = k; i < blocks.size(); ++i) {
            // Increase whiteBlockCount if the current block is white
            whiteBlockCount += blocks[i] == 'W';

            // Decrease whiteBlockCount if the leftmost block of
            // the previous window was white
            whiteBlockCount -= blocks[i - k] == 'W';

            // Update minRecolors to the smallest number of white blocks
            // seen in any window of size 'k'
            minRecolors = min(minRecolors, whiteBlockCount);
        }

        // Return the minimum number of recolors needed
        return minRecolors;
    }
};
```

TypeScript

```typescript
// Function to find the minimum number of recolors to get at least k consecutive black blocks.
// @param blocks - String representing the arrangement of black (B) and white (W) blocks.
// @param k - Number representing the desired consecutive black blocks length.
// @returns The minimum number of recolors required (changing 'W' to 'B').
function minimumRecolors(blocks: string, k: number): number {

    // Initialize a counter for white blocks in the first window of size k.
    let whiteCount = 0;
    for (let i = 0; i < k; ++i) {
        whiteCount += blocks[i] === 'W' ? 1 : 0;
    }

    // The answer starts off as the number of white blocks in the first window.
    let minRecolors = whiteCount;

    // Slide the window of size k across the blocks string while updating the count.
    for (let i = k; i < blocks.length; ++i) {
        // If the new block is white, increase the count.
        whiteCount += blocks[i] === 'W' ? 1 : 0;
        // If the block exiting the window is white, decrease the count.
        whiteCount -= blocks[i - k] === 'W' ? 1 : 0;
        // Update the answer with the minimum count seen so far.
        minRecolors = Math.min(minRecolors, whiteCount);
    }

    // Return the minimum number of recolors needed.
    return minRecolors;
}
```


```python
class Solution:
    def minimumRecolors(self, blocks: str, k: int) -> int:
        # Initial count of 'W' in the first window of size k
        white_count = blocks[:k].count('W')
        # Initialize minimum white blocks to be recolored with the count from the first window
        min_recolors = white_count

        # Slide the window of size k through the blocks string
        for index in range(k, len(blocks)):
            # If the newly included block in the window is white, increment the count
            if blocks[index] == 'W':
                white_count += 1
            # If the block that is exiting the window is white, decrement the count
            if blocks[index - k] == 'W':
                white_count -= 1
            # Update the minimum if the current count is less than the previous minimum
            min_recolors = min(min_recolors, white_count)

        # Return the minimum number of white blocks that need to be recolored
        return min_recolors
```


Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed based on the operations performed within the loop. The loop runs from `k` to `len(blocks)`, which indicates that the loop runs `len(blocks) - k` times. Within each iteration, the code performs constant-time operations such as comparison and increment/decrement operations. Therefore, the overall time complexity is `O(len(blocks) - k)`, which simplifies to `O(n)` where `n` is the length of the `blocks` string.

Space Complexity

The space complexity is determined by the amount of additional space used by the algorithm relative to the input size. The given code uses a fixed number of variables (`ans`, `cnt`, `i`) that do not depend on the size of the input. Therefore, the space complexity remains constant, regardless of the input size. Consequently, the space complexity of the code is `O(1)`, indicating constant space complexity.