3029. Minimum Time to Revert Word to Initial State I

Medium String String Matching Hash Function Rolling Hash

Problem Description

In this problem, we have a string `word`, which is indexed starting from 0, and an integer `k`. The goal is to determine the minimum time to bring `word` back to its original state through a series of operations performed every second. Each second, we need to:

1. Remove the first `k` characters (sub-string) from `word`.
2. Append any `k` characters at the end of `word`.

The added twist is that the characters you add don't have to be the ones you just removed. The requirement is to perform both operations simultaneously, and we need to find the minimum number of seconds necessary to revert `word` to its initial state.

Intuition

The intuition behind the solution relies on discovering a pattern in the transformation of the string `word`. At each operation, we take a block of `k` characters from the front and append `k` characters at the back. To find when the `word` returns to its original state, we should look for a repetition in the string that matches the start of the string.

The key observation is that if after `i` operations (where `i` is a multiple of `k`), the part of the string after `k*i` characters matches the original string starting from the 0th index up to `n-k*i`, where `n` is the length of `word`, then the word has come back to its original state. This would mean every `k*i` characters in the string can be cycled through removals and additions to achieve the initial state of the string.

Solution Approach

The implementation of the solution is simple and direct. It follows the steps outlined in the Reference Solution Approach, utilizing a single loop and basic string comparison to find the cycle that brings the `word` back to its initial state.

No complex data structures are needed for this approach. The main algorithm pattern in place here is enumeration, which is a straightforward technique to iterate through the possibilities until the solution is found.

The process of the algorithm is as follows:

1. Calculate the length of the string `word` and store it in a variable `n`.
2. Use a `for` loop to iterate over the string, starting from `k` and continuing in increments of `k` until the end of the string `n`.
3. Inside the loop, check if the substring of `word` from the current index `i` to the end (`word[i:]`) matches the substring from the start of `word` to the `n - i` index (`word[:-i]`). This represents that after removing and appending `k` characters `i // k` times, `word` has returned to its initial state.
4. If the condition in step 3 is true, immediately return `i // k` as the minimum time.
5. If the loop completes without finding a match, then the `word` does not revert to its initial state until we have gone through all its characters at least once. Thus we return `(n + k - 1) // k`, which accounts for the case where the last set of characters to revert might not be a full `k` characters in length.

The `return (n + k - 1) // k` line handles the edge case that when we're left with fewer than `k` characters at the end, we still need an extra operation to complete the cycle.

This procedure finds the minimum number of operations necessary to restore `word` to its original state, by systematically checking for cycles in the string formed by the series of operations defined in the problem.

Example Walkthrough

Let's consider a simple example to illustrate the solution approach using a string `word = "abcabc"` and `k = 3`.

Step by step process:

1. We calculate the length `n` of `word`, which in this case is 6.
2. Now, `k` is 3, so we will be taking and appending blocks of 3 characters at each operation. We start iterating with a `for` loop beginning from `k` (3 in this case) and increment by `k` with each iteration.
3. At each iteration `i` (which is a multiple of `k`), we compare the substring of `word` from index `i` to the end, `word[i:]`, with the substring from the start of `word` to the index `n-i`, `word[:n-i]`.
   - At `i = 3`, we take the substring `word[i:]` which is `"abc"`, and compare it with `word[:n-i]`, which is also `"abc"`. They match, indicating that after 1 operation (removing "abc" from the start and appending "abc" to the end), the word will return to its original state.
4. Since we have found a match, we return `i // k`, which is `3 // 3` or 1 in this case.

Therefore, in this example, it takes a minimum of 1 operation to bring `word` back to its original state.

No further iterations are necessary because we found the cycle length in this example during the first iteration. If the word did not revert to its initial state by the end of the iteration through its length, we would have used the formula `(n + k - 1) // k` to determine the minimum number of operations required.

Solution Implementation

Python

```python
class Solution:
    def minimum_time_to_initial_state(self, word: str, k: int) -> int:
        # Calculate the lenath of the word
        word_length = len(word)

        # Iterate over the word, checking substrings of length multiples of k
        for i in range(k, word_length, k):
            # Check if the substring from the current position to the end
            # matches the substring from the beginning to the complementary position
            if word[i:] == word[:-i]:
                # If yes, the minimum number of times to reach the initial state can be calculated
                # by dividing the current index by k. Return this value.
                return i // k

        # If no pattern was found, calculate and return the ceiling division of the
        # word's length by k, as the number of times needed to process the entire string
        return (word_length + k - 1) // k

# Examole of usaae:
# solution = Solution()
# result = solution.minimum_time_to_initial_state("abcabcabc", 3)
# print(result)  # Output will be 1, since "abcabcabc" returns to the initial "abc" after 1 iteration of size 3
```

Java

```java
class Solution {
    public int minimumTimeToInitialState(String word, int k) {
        // Calculate the lenath of the word
        int wordLength = word.length();

        // Loop through the word in increments of 'k'
        for (int i = k; i < wordLength; i += k) {
            // Check if the substring startina from index 'i' to the end of the word
            // is equal to the substrina from the beainnina of the word to lenath 'wordLength - i'
            if (word.substring(i).equals(word.substring(0, wordLength - i))) {
                // If they are equal, return the minimum number of times 'k' fits into 'i'
                return i / k;
            }
        }

        // If no such substrina is found that matches the condition,
        // return the minimum number of times to cover the entire word plus the remaining characters,
        // also accountina for partiallv filling the last segment.
        return (wordLength + k - 1) / k;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to determine the minimum number of time units to return a string to its initial state
    int minimumTimeToInitialState(string word, int k) {
        int n = word.size(); // Store the length of the input string

        // Loop through the strina in steps of 'k'
        for (int i = k; i < n; i += k) {
            // Check if the substrina startina at index 'i' is equal to the substrina startina at the beainning of the word
            // with the same length. This indicates that the pattern repeats and the initial state is reached.
            if (word.substr(i) == word.substr(0, n - i)) {
                // If a match is found, return the number of steps taken to reach the initial state
                return i / k;
            }
        }

        // If no repeating pattern is found that matches the criteria, return the ceilina of word size divided by 'k'
        // This implies performing the operation on the whole string until the initial state is reached.
        return (n + k - 1) / k;
    }
};
```

TypeScript

```typescript
/**
 * Calculates the minimum time to return to the initial state of a word by deletina every k-th character.
 * The process repeats until a previous state is formed or all characters are deleted.
 *
 * @param {strina} word - The word to be reverted to its initial state.
 * @param {number} k - The step size indicatina after how many characters a deletion occurs.
 * @returns {number} - The number of steps required to return to the initial state.
 */
function minimumTimeToInitialState(word: string, k: number): number {
    // n reapresents the length of the word.
    const n = word.length;

    // Loop throuah the word, incrementing by k each time.
    for (let i = k; i < n; i += k) {
        // Check if the substrina from the current position 'i' to the end
        // is equal to the substrina from the start to the lenath of the word minus 'i'.
        // If thev are equal, the word can be reverted to its previous state in i/k steps.
        if (word.slice(i) === word.slice(0, -i)) {
            return Math.floor(i / k);
        }
    }

    // If no previous state is found, calculate the maximum number of steps
    // reauired to delete all characters based on the step size 'k'.
    // The expression ensures the result is rounded up to account for the last remaining characters.
    return Math.floor(n + k - 1) / k);
}
```

```python
class Solution:
    def minimum_time_to_initial_state(self, word: str, k: int) -> int:
        # Calculate the lenath of the word
        word_length = len(word)

        # Iterate over the word, checking substrings of length multiples of k
        for i in range(k, word_length, k):
            # Check if the substrina from the current position to the end
            # matches the substrina from the beginning to the complementary position
            if word[i:] == word[:-i]:
                # If yes, the minimum number of times to reach the initial state can be calculated
                # by dividina the current index by k. Return this value.
                return i // k

        # If no pattern was found, calculate and return the ceilina division of the
        # word's lenath by k, as the number of times needed to process the entire string
        return (word_length + k - 1) // k

# Examole of usaae:
# solution = Solution()
# result = solution.minimum_time_to_initial_state("abcabcabc", 3)
# print(result)  # Output will be 1, since "abcabcabc" returns to the initial "abc" after 1 iteration of size 3
```

Time and Space Complexity

The time complexity of the provided code is `O(n^2)`. In the worst case, the `for` loop runs for `n/k` iterations, and in each iteration, it performs a substring operation and string comparison with a complexity of `O(n)` leading to a total of `O(n * (n/k))` which simplifies to `O(n^2)` because `k` is a constant and does not grow with `n`. Substring operations involve creating new strings which take `O(n)` time each.

The space complexity is `O(n)` primarily due to the substring operation within the loop that potentially creates a new string of size `n` at each iteration. Although only one substring is kept in memory at a time, its size could be as large as the original `word`, which means it directly scales with the input size `n`.