1625. Lexicographically Smallest String After Applying Operations

Medium Breadth-First Search String

Leetcode Link

Problem Description

In this problem, we are given a string `s` representing a sequence of digits from `0` to `9`. The length of `s` is guaranteed to be even. We are also provided with two integers `a` and `b`. Our task is to transform `s` into the lexicographically smallest possible string by repeatedly applying two types of operations in any order:

1. **Add Operation**: We can add the integer `a` to each digit at an odd index (where indexing starts from 0). If adding `a` causes the digit to become greater than `9`, it wraps around to `0` and continues from there (essentially we take the sum modulo 10).
2. **Rotate Operation**: We can rotate the string to the right by `b` positions. Rotating by `b` positions means taking the last `b` characters of `s` and moving them to the front.

A lexicographically smaller string is defined as one that would come first in a dictionary. For example, `'2034'` is lexicographically smaller than `'2035'` because it has a smaller digit (`'4'` instead of `'5'`) in the last position where they differ.

The aim, therefore, is to find the sequence of operations that, when applied to `s`, will result in the lexicographically smallest string.

Intuition

The challenge of this problem lies in determining the correct order and amount of operations to apply since both rotate and add operations can be done an unlimited number of times. The brute force approach of trying every possible combination of operations is not feasible due to the potentially massive number of combinations.

Since `s` is of an even length, there are two distinct cases for the rotate operation:

1. If `b` is even, rotating the string does not change the positions of the odd and even indexed characters relative to each other.
2. If `b` is odd, each rotation swaps the positions of odd and even indexed characters.

Given these properties, we can infer a few strategies that help reduce the problem space:

- Rotating the string by its length or or a multiple of it returns the string to its original configuration. Thus, we need to rotate the string at most `n - 1` times to explore all unique configurations.
- Since rotating by an even `b` doesn't change the relative positions of characters, we only need to focus on applying the add operation to odd indices when `b` is even.
- However, when `b` is odd, because rotating changes the parity of the indices, we will need to apply the add operation to all characters, both at odd and even positions.

The solution provided employs a breadth-first search (BFS) type of approach. Here are the steps:

1. Apply the rotation operation once at a time, each time attempting to compute a lexicographically smaller result.
2. After each rotate, apply the add operation repeatedly (up to 10 times because after 10 additions we will have cycled through all possible digit values) to the odd indexed characters, checking each time if a smaller string is achieved.
3. If the rotation step is such that `i` is odd, extend the same add operation to even indexed characters.

By repeatedly applying this sequence of operations, we ensure that we explore all possible combinations of rotations and additions without redundancy. We keep track of the smallest string seen so far and update this minimum whenever we find a smaller one.

This process is guaranteed to terminate because there are a finite number of possible strings (since `s` is of finite length), and every step is designed to make progress towards a lexicographically smaller string.

Solution Approach

The solution provided adopts a method similar to breadth-first search (BFS). In graph theory, BFS is an algorithm for traversing or searching tree or graph data structures. It starts at an arbitrary node and explores all of the neighbor nodes at the present depth prior to moving on to nodes at the next depth level. In our case, the nodes represent different versions of the string we obtain after each rotation and addition operation.

Here's how the solution works:

- We initialize `ans` with the original string `s`. This will hold the smallest string found during the search.
- The variable `n` stores the length of the string.
- A Python list `s` is created from the original string to allow for mutable operations.
- There is an outer loop that permits us to try every possible rotation on the string. We can cycle through all the rotations by rotating one step at a time, up to `n` times.
- The state of `s` is adjusted with `s = s[-b:] + s[:-b]`. This line effectively rotates the string to the right by `b` positions.
- An inner loop (ranging from 0 to 9) applies the addition operation to all odd indices of `s`. This is done `10` times because after 10 additions, the character would have been incremented by `1` ten times, which is equivalent to adding `1` considering the modulo operation.
- If the shift `i` is odd (`b & 1`), we account for the switch between the odd and even indices due to the rotations. Yet another nested loop runs for `10` iterations to apply the addition operation to what were originally even indices before rotation.
- After each application of the addition operation, we join the characters into a string `''.join(s)` and compare it with `ans`. If `s` is lexicographically smaller than `ans`, we update `ans`.
- Finally, after all possible transformations are tried, we return `ans`.

BFS can be a quite wide exploration, but in this algorithm, we are keeping only one copy of the string at each level of depth, rather than exploring multiple strings in parallel. This is an efficient way to handle the problem since we are always iterating from the current smallest string and, by domain constraints, we know there's no need to revisit a string we have already considered.

The Python list structure allows us to perform rotates and additions very efficiently, which is key in handling possibly numerous iterations. The algorithm effectively traverses a space of $ n * 10 * 10 $ possible strings (where $ n $ is the number of rotations and 10 comes from the number of potential additions to both odd and even indices), ensuring that we find the lexicographically smallest string possible.

Example Walkthrough

Let's use a simple example to demonstrate the solution approach. Suppose we are given a string `s = "4625"`, and integers `a = 1`, and `b = 2`.

The string `s` is of length 4, which is even, and we see that the rotation parameter `b` is also even. Hence, rotating the string will not change odd and even positions relative to each other. Let's follow the steps outlined in the solution approach:

- Start with `ans = "4625"`, the original string `s`.
- The length of the string `n = 4`.

We begin with the outer loop, attempting all possible rotations:

1. **First Rotation**: `s = "2546"` (rotated right by `b=2`)
   - Add `a=1` to each digit at odd indices (indexing starts at 0):
     - Apply addition operation: `s[1] = (5 + 1) % 10 = 6` and `s[3] = (6 + 1) % 10 = 8`. Now `s = "2848"`.
     - Update `ans` since `"2848"` is lexicographically smaller than `"4625"`.
   - As `b` is even, we don't need to perform addition on even indices this round.
2. **Second Rotation**: `s = "4828"` (rotated again by `b=2` from original, so it is `"2546" -> "4828"`)
   - Add `a=1` to each digit at the odd indices:
     - Apply addition operation: `s[1] = (8 + 1) % 10 = 9` and `s[3] = (8 + 1) % 10 = 1`. Now `s = "4291"`.
     - Update `ans` since `"4212"` is lexicographically smaller than `"2848"`.
   - As `b` is even, we skip addition to even indices again.
3. **Third Rotation**: `s = "4291"` (rotated right again by `b=2` from original, so it is `"4625" -> "1249"`)
   - Add `a=1` to each digit at the odd indices:
     - Apply addition operation: `s[1] = (2 + 1) % 10 = 3` and `s[3] = (9 + 1) % 10 = 2`. Now `s = "1522"`.
     - `ans` remains `"4212"` because `"1522"` is not lexicographically smaller.
4. **Fourth Rotation**: `s = "2149"` (rotated right again by `b=2` from original, so it is `"4625" -> "2149"`)
   - Add `a=1` to each digit at the odd indices:
     - Apply addition operation: `s[1] = (1 + 1) % 10 = 4` and `s[3] = (9 + 1) % 10 = 2`. Now `s = "2422"`.
     - `ans` remains `"4212"` because `"2422"` is not lexicographically smaller.

After the loop completes, we have tried all rotations and addition operations. Our `ans` holds the lexicographically smallest string: `"4212"`.

Thus, following the BFS-like strategy, we have successfully explored all the possible rotations and have applied addition operations to find the smallest string through efficient updating. The final answer is `"4212"`.

Python Solution

```python
class Solution:
    def find_lex_smallest_string(self, s: str, a: int, b: int) -> str:
        # Initialize answer with the initial string.
        ans = s
        # Get the length of the input string.
        n = len(s)
        # Convert the string to a list for easier manipulation.
        s = list(s)
        # Rotate the string n times, which ensures that
        # all rotations are visited.
        for _ in range(n):
            # Rotate the string based on the value of b.
            s = s[-b:] + s[:-b]
            # Add 'a' to every digit at odd indices.
            for _ in range(10):  # Repeat this operation up to 10 times, one for each digit.
                for i in range(1, n, 2):
                    s[i] = str((int(s[i]) + a) % 10)
                # If 'b' is odd, addition is also applicable at even indices.
                # Since rotation places previously odd indexed numbers at even indices.
                if b % 2 == 1:
                    for i in range(0, n, 2):  # Repeat for even indices.
                        s[i] = str((int(s[i]) + a) % 10)
                # Join the list to form a string and compare with the answer.
                # Update answer if the current string is lexicographically smaller.
                if ans > ''.join(s):
                    ans = ''.join(s)
            # If 'b' is even, only update based on odd indices' additions.
            else:
                current = ''.join(s)
                if ans > current:
                    ans = current
        # Return the lexicographically smallest string obtained.
        return ans
```

Java Solution

```java
class Solution {
    public String findLexSmallestString(String s, int a, int b) {
        // Get the length of the string
        int length = s.length();
        // The variable 'answer' will hold the lexicographically smallest string found
        String answer = s;

        // Rotating the string n times for 'length' times
        for (int i = 0; i < length; ++i) {
            // Perform rotation by 'b' characters to the left
            s = s.substring(b) + s.substring(0, b);
            // Convert string to a character array for manipulation
            char[] charArray = s.toCharArray();

            // Try adding 'a' to every digit (10 times since there are only 10 possibilities for a digit)
            for (int j = 0; j < 10; ++j) {
                // Only add to odd indexed characters of the array starting from 1
                for (int k = 1; k < length; k += 2) {
                    charArray[k] = (char) ('0' + (charArray[k] - '0' + a) % 10 + '0');
                }

                // If 'b' is odd, we need to handle even indexed characters as well because rotation affects them
                if (b % 2 == 1) {
                    for (int k = 0; k < length; k += 2) {
                        // This time add 'a' to even indexed characters
                        for (int k = 0; k < length; k += 2) {
                            charArray[k] = (char) ('0' + (charArray[k] - '0' + a) % 10 + '0');
                        }
                        // Create new string from the character array
                        s = String.valueOf(charArray);
                        // Update the answer if a new lexicographically smaller string is found
                        answer = answer.compareTo(s) < 0 ? answer : s;
                    }
                } else {
                    // If 'b' is even, create a new string and potentially update the answer
                    s = String.valueOf(charArray);
                    if (answer.compareTo(s) > 0) {
                        answer = s;
                    }
                }
            }
        }
        // Return the lexicographically smallest string found
        return answer;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    string findLexSmallestString(string s, int a, int b) {
        int stringLength = s.size();          // Length of the string.
        string smallestLexicalString = s;     // The lexically smallest string found so far.

        // Perform rotation and replacement operations to find the smallest lexicographical string.
        for (int i = 0; i < stringLength; ++i) {
            // Rotate the string by 'b' positions.
            s = s.substr(stringLength - b) + s.substr(0, stringLength - b);

            // Try incrementing the digits at odd indices and finding the smallest string.
            for (int j = 0; j < 10; ++j) {
                // Increment each digit at odd index 'a' by 'a', and handle wrap around using modulo 10.
                for (int k = 1; k < stringLength; k += 2) {
                    s[k] = (s[k] - '0' + a) % 10 + '0';
                }

                // Check if 'b' is odd, if yes, also increment the digits at even indices.
                if (b % 2 == 1) {
                    for (int k = 0; k < stringLength; k += 2) {
                        // Increment each digit at even index 'k' by 'a', and handle wrap around using modulo 10.
                        for (int k = 0; k < stringLength; k += 2) {
                            s[k] = (s[k] - '0' + a) % 10 + '0';
                        }
                        // Update smallestLexicalString if current string 's' is smaller.
                        smallestLexicalString = min(smallestLexicalString, s);
                    }
                } else {
                    // Update smallestLexicalString if current string 's' is smaller.
                    smallestLexicalString = min(smallestLexicalString, s);
                }
            }
        }

        // Return the lexically smallest string after trying all possibilities.
        return smallestLexicalString;
    }
};
```

Typescript Solution

```typescript
// Define the string length variable globally.
let stringLength: number;

// Function to find the lexicographically smallest string after applying rotation and addition operations.
function findLexSmallestString(s: string, a: number, b: number): string {
    stringLength = s.length; // Initialize the string length variable.
    let smallestLexicalString: string = s; // Initialize the smallest lexicographical string found with the original string.

    // Perform rotation and increment operations to find the smallest lexicographical string.
    for (let i = 0; i < stringLength; ++i) {
        // Rotate the string by 'b' positions.
        s = s.substring(stringLength - b) + s.substring(0, stringLength - b);

        // Try incrementing the digits at odd indices and find the smallest string.
        for (let j = 0; j < 10; ++j) {
            // Increment each digit at odd indices by 'a', using modulo 10 for wrap around.
            for (let k = 1; k < stringLength; k += 2) {
                let digit = (parseInt(s[k]) + a) % 10;
                s = s.substring(0, k) + digit + s.substring(k + 1);
            }

            // If 'b' is odd, also apply incrementing to digits at even indices.
            if (b % 2 === 1) {
                for (let k = 0; k < stringLength; k += 2) {
                    // Increment each digit at even indices by 'a', using modulo 10 for wrap around.
                    for (let k = 0; k < stringLength; k += 2) {
                        let digit = (parseInt(s[k]) + a) % 10;
                        s = s.substring(0, k) + digit + s.substring(k + 1);
                    }
                    // Update smallestLexicalString if current string 's' is smaller.
                    smallestLexicalString = minLexicographicalString(smallestLexicalString, s);
                }
            } else {
                // Update smallestLexicalString if current string 's' is smaller.
                smallestLexicalString = minLexicographicalString(smallestLexicalString, s);
            }
        }
    }

    // Return the lexically smallest string after trying all possibilities.
    return smallestLexicalString;
}

// Helper function to find the smaller of two lexicographically ordered strings.
function minLexicographicalString(str1: string, str2: string): string {
    return str1.localeCompare(str2) < 0 ? str1 : str2;
}
```

Time and Space Complexity

Time Complexity

The given code performs a series of rotations and digit increments to find the lexicographically smallest string that can be achieved under given operations. The time complexity can be broken down into multiple parts:

1. The outermost loop runs at most `n` times where `n` is the length of the string, since that's the number of possible rotations we can check.
2. For every rotation, we have an inner loop running 10 times to increment the digits at odd indices.
3. When `b` is odd, an additional loop over 10 iterations is executed for each of the outer loop iterations to increase the even indices.
4. Within the second and third bullet point loops, there is an iteration over the characters of the string in steps, which is $ O(n) $.

Considering these nested loops and their iteration counts, the time complexity can be calculated as follows:

- If `b` is even, the complexity is $ O(n * 10 * n) = O(n^2) $ because the changes are applied to odd indices regardless of the number of rotation.
- If `b` is odd, we must consider additional inner iterations for increasing even indices, which adds another factor of 10, leading to $ O(n * 10 * 10 * n) = O(100 * n^2) = O(n^2) $, as constant factors are dropped in time complexity considerations.

As the `b` being even or odd determines whether the innermost loop is executed, the overall time complexity is $ O(n^2) $.

Space Complexity

The space complexity of the code is primarily due to the storage of the string `s` as a list and the temporary storage of the modified string `t`.

- The list converted string takes $ O(n) $ space.
- The temporary string `t` also takes $ O(n) $ space.

There is no additional space used that grows with the input, as the loops use a constant amount of space. Therefore, the space complexity is $ O(n) $.