# 2573. Find the String with LCP



## Problem Description

The task is to construct the alphabetically smallest string `word` for a given `lcp` matrix which represents the longest common prefixes of the substrings of `word`. The `lcp` matrix is an `n x n` grid where `lcp[i][j]` equals the length of the longest common prefix between `word[i,n-1]` and `word[j,n-1]`. The challenge is to determine the string `word` if it exists or return an empty string otherwise. It tests one's ability to reason about strong relationships between different elements of the matrix.

## Intuition

The key insight to solve this problem involves understanding the relationship between the matrix entries and the characters of the strings at different indices. Since the alphabetically smallest string is required, a natural approach is to attempt to fill the string with the lexicographically earliest character ('a') where possible, and then move on to the next character only if necessary.

We start by initializing a string `w` with an empty character for each position and iterate over the lowercase alphabet. For each character `c`, we look for indices in `s` that have not yet been set and that we can set with `c` without violating the given `lcp` conditions. For every such index, we also set the character `c` at all positions `j` where `lcp[i][j]` is non-zero, since they must share that prefix.

To ensure that the final string does not violate any `lcp` constraints, an additional validation step verifies that the `lcp` matrix entries are consistent with what we would expect given the assigned characters. If an inconsistency is found—a missing character or an incorrect prefix length—the solution concludes that no valid string can be created, thus returning an empty string.

By cautiously filling out the available spots with the smallest possible letter and checking for the `lcp` matrix consistency along the process, we should be able to either arrive at the desired smallest string or determine that such a string doesn't exist.

## Solution Approach

The solution uses a straightforward yet clever approach by iterating over the list of ASCII lowercase letters, which guarantees that we are always selecting the alphabetically smallest possible character for every index.

Here's a breakdown of how the code implements the solution step by step:

1. Initialize the string `w` as a list of empty strings of length `n`, which matches the size of the `lcp` matrix. Each element of `s` represents a character in the answer string.
2. Start iterating over each character `c` in the ASCII lowercase alphabet. The idea is to try and fill `s` with the smallest possible characters first, maintaining lexicographic ordering.
3. Using a while loop, increment the index `i` until finding an index where `s[i]` is still an empty string or until the end of `s` is reached. This finds the first unfilled position in `s` where we can potentially place a new character.
4. If index `i` reaches the end of the string, all characters have been assigned, and the loop breaks.
5. For each index `j` from `i` to `n-1`, if `lcp[i][j]` is not zero, which means there is a common prefix of non-zero length, assign the current character `c` to `s[j]`. This step assigns the character `c` to all positions that share a prefix with `i`.
6. After trying to fill `s` with all characters of the alphabet, check for any unfilled positions left in `s`. If there are any, return an empty string since it means we weren't able to construct a string fitting the `lcp` matrix.
7. Perform validation on the built string from back to front to ensure all `lcp` conditions are met. This step verifies that:
   - For any two indices `i` and `j` that have the same character, the corresponding `lcp[i][j]` must be `1` if either `i` or `j` is `n-1`. Otherwise, `lcp[i][j]` must equal the value of `lcp[i+1][j+1] + 1`.
   - For any two indices `i` and `j` with different characters, `lcp[i][j]` must be `0`. If any of these conditions are not true, it means that the `lcp` matrix is not consistent with the string we have built, and thus the function returns an empty string.
8. If the string passes all the validation checks, join the list `s` to form the final answer string, and return it.

This method utilizes a greedy approach to select the lexicographically smallest letters, combined with a validation step to confirm that the selected characters do not violate the `lcp` conditions. The intelligent ordering of operations ensures an efficient and correct solution.

### Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have an `lcp` matrix:

```
1 lcp = [
2   [3, 1, 0],
3   [1, 2, 0],
4   [0, 0, 1]
5 ]
```

This `lcp` matrix is of size 3×3, so our `word` will have three characters. We'll try to build the smallest possible `word`.

1. **Initialize the string** `s`: `s` starts as `['', '', '']`.
2. **Iterate over ASCII lowercase**: We start from 'a', the smallest character.
3. **Find the first empty string position**: The first index `i` with an empty string is 0.
4. **End check**: Our string `s` is not full.
5. **Assign 'a' to positions with non-zero `lcp`**: Looking at `lcp[0]`, there is a '1' at index 1. We can set `s[0]` and `s[1]` to 'a'.

   After this, our `s` looks like `['a', 'a', '']`.
6. **Move to next index**: Since `s[2]` is still empty, we need to fill it with the next character. Since 'a' can't be placed due to `lcp[0][2]` and `lcp[1][2]` being 0, we try with the next character, which is 'b'.

   Now `s` looks like `['a', 'a', 'b']`.
7. **Check for unfilled positions**: There are no unfilled positions left.
8. **Validate the built string**: Validate `s` using the `lcp` conditions. For `s[0]` and `s[1]`, the `lcp` value is 1, which is correct since they are the same and this matches the precondition set out in the problem description. For `s[0]` and `s[2]`, as well as `s[1]` and `s[2]`, the `lcp` is 0, which is consistent with them having different characters.
9. **String passes validation**: No inconsistencies found.
10. **Return the result**: Join `s` to form the final string. The result for this example is "aab".

Thus, by carefully placing the smallest possible letters and validating against the `lcp` matrix, we correctly identified that the alphabetically smallest string representation that could produce the given `lcp` matrix is "aab".

## Python Solution

```python
from string import ascii_lowercase

class Solution:
    def find_the_string(self, lcp_matrix):
        """
        Reconstructs the original string from the longest common prefix (LCP) matrix.

        :param lcp_matrix: A 2D list representing the LCP matrix between all pairs of prefixes.
        :return: The original string if it can be successfully reconstructed, otherwise an empty string.
        """
        # Number of prefixes (and thus characters in the string)
        num_prefixes = len(lcp_matrix)
        # Initialize the list of characters for the resulting string
        reconstructed_string = ["" for _ in range(num_prefixes)]

        current_char_index = 0  # Index to keep track of where to assign characters in the string

        # Iterate over the lowercase alphabet to assign characters to the string
        for current_char in ascii_lowercase:
            # Find the next index in the string that hasn't been assigned a character
            while current_char_index < num_prefixes and reconstructed_string[current_char_index]:
                current_char_index += 1
            # If all indices have been assigned, stop the process
            if current_char_index == num_prefixes:
                break
            # Assign the current character to all relevant indices in the string
            for j in range(current_char_index, num_prefixes):
                if lcp_matrix[current_char_index][j]:
                    reconstructed_string[j] = current_char

        # Check for any unassigned characters, which indicates a failure to reconstruct the string
        if "" in reconstructed_string:
            return ""

        # Validate the reconstructed string with the given LCP matrix
        for i in reversed(range(num_prefixes)):  # Loop from the end of the string backwards
            for j in reversed(range(num_prefixes)):  # Inner loop for comparison
                # If characters match, the LCP should be 1 more than the next
                if reconstructed_string[i] == reconstructed_string[j]:
                    if i == num_prefixes - 1 or j == num_prefixes - 1:
                        # At the end of the string, LCP should be 1 for matching characters
                        if lcp_matrix[i][j] != 1:
                            return ""
                    elif lcp_matrix[i][j] != lcp_matrix[i + 1][j + 1] + 1:
                        # Elsewhere, LCP should be the next LCP + 1
                        return ""
                # If characters don't match, the LCP should be 0
                elif lcp_matrix[i][j]:
                    return ""

        # Join the reconstructed string list to form the final string and return it
        return "".join(reconstructed_string)

# Example usage:
# solution = Solution()
# result = solution.find_the_string([[0, 3, 0], [3, 0, 0], [0, 0, 0]])
# print(result)  # This would print the reconstructed string or an empty string if it couldn't be reconstructed.
```

## Java Solution

```java
class Solution {
    /**
     * Method to find the string based on the longest common prefix (lcp) information provided.
     */
    public String findTheString(int[][] lcp) {
        // 'n' is the length of the string to be constructed.
        int n = lcp.length;
        // 'a' is the character array that will form the resultant string.
        char[] chars = new char[n];

        // Iterate over each character starting from 'a' to 'z' to construct the string.
        for (char c = 'a'; c <= 'z'; ++c) {
            int i = 0;
            // Skip filled characters in the 'chars' array.
            while (i < n && chars[i] != '\0') {
                ++i;
            }
            // If all characters are filled, break the loop as the string is complete.
            if (i == n) {
                break;
            }
            // Assign the current character to each position in the array that has a
            // non-zero value in the lcp matrix for that index.
            for (int j = i; j < n; ++j) {
                if (lcp[i][j] > 0) {
                    chars[j] = c;
                }
            }
        }

        // Check if there are any unfilled positions in 'chars'. If found, return an empty string.
        for (int i = 0; i < n; ++i) {
            if (chars[i] == '\0') {
                return "";
            }
        }

        // Validate whether the constructed string is correct based on the lcp information.
        for (int i = n - 1; i >= 0; --i) {
            for (int j = n - 1; j >= 0; --j) {
                if (chars[i] == chars[j]) {
                    if (i == n - 1 || j == n - 1) {
                        if (lcp[i][j] != 1) {
                            return "";
                        }
                    } else if (lcp[i][j] != lcp[i + 1][j + 1] + 1) {
                        return "";
                    }
                } else if (lcp[i][j] > 0) {
                    return "";
                }
            }
        }

        // Return the constructed string.
        return new String(chars);
    }
}
```

## C++ Solution

```cpp
class Solution {
public:
    // This method finds a string based on its longest common prefix (lcp) array.
    string findTheString(vector<vector<int>>& lcp) {
        int index = 0;
        int lengthOfString = lcp[0].size(); // Get the length of the string to be reconstructed.
        string reconstructedString(lengthOfString, '\0'); // Initialize the string with null characters.

        // Iterate over each character starting from 'a' to 'z' to reconstruct the string.
        for (char currentChar = 'a'; currentChar <= 'z'; ++currentChar) {
            // Find the next position in the string that hasn't been set yet.
            while (index < lengthOfString && reconstructedString[index]) {
                ++index;
            }
            // If the entire string has been filled, exit the loop.
            if (index == lengthOfString) {
                break;
            }
            // Assign the current character to all positions that have a non-zero lcp with the current `index`.
            for (int i = index; i < lengthOfString; ++i) {
                if (lcp[index][i] > 0) {
                    reconstructedString[i] = currentChar;
                }
            }
        }

        // If there are still unset positions in the string, it means it's impossible to construct.
        if (reconstructedString.find('\0') != string::npos) {
            return "";
        }

        // Check the validity of the constructed string against the lcp array.
        for (int i = lengthOfString - 1; i >= 0; --i) {
            for (int j = lengthOfString - 1; j >= 0; --j) {
                // Characters match and we are at the end of the string or the lcp values are consistent.
                if (reconstructedString[i] == reconstructedString[j]) {
                    if (i == lengthOfString - 1 || j == lengthOfString - 1) {
                        if (lcp[i][j] != 1) {
                            return "";
                        }
                    } else if (lcp[i][j] != lcp[i + 1][j + 1] + 1) {
                        return "";
                    }
                } else if (lcp[i][j]) {
                    // Characters don't match but the lcp value is not zero.
                    // Mismatched characters should not have a non-zero LCP.
                    return "";
                }
            }
        }

        // If all checks pass, return the constructed string.
        return reconstructedString;
    }
};
```

## Typescript Solution

```typescript
// Type definition for LCP Array which is a 2D array with numbers.
type LcpArray = number[][];

// This method finds a string based on its longest common prefix (lcp) array.
function findTheString(lcpArray: LcpArray): string {
    const n = lcpArray.length;
    const lengthOfString = lcpArray.length; // Get the length of the string to be reconstructed.
    let reconstructedString = new Array(lengthOfString).fill('\0'); // Initialize the string with null characters.

    // Loop through all lowercase letters.
    for (let currentChar = 'a'.charCodeAt(0); currentChar <= 'z'.charCodeAt(0); ++currentChar) {
        // Find the next position in the string that hasn't been set yet.
        while (index < lengthOfString && reconstructedString[index] !== '\0') {
            ++index;
        }
        // If the entire string has been filled, exit the loop.
        if (index === lengthOfString) {
            break;
        }
        // Assign the current character to all positions that have a non-zero lcp with the current `index`.
        for (let i = index; i < lengthOfString; ++i) {
            if (lcpArray[index][i] > 0) {
                reconstructedString[i] = String.fromCharCode(currentChar);
            }
        }
    }

    // Convert the array of characters back into a string.
    reconstructedString = reconstructedString.join('');

    // If there are still unset positions in the string, it means it's impossible to construct.
    if (reconstructedString.indexOf('\0') !== -1) {
        return "";
    }

    // Check the validity of the constructed string against the lcp array.
    for (let i = lengthOfString - 1; i >= 0; --i) {
        for (let j = lengthOfString - 1; j >= 0; --j) {
            // Characters match and we are at the end of the string or the lcp values are consistent.
            if (reconstructedString[i] === reconstructedString[j]) {
                if (i === lengthOfString - 1 || j === lengthOfString - 1) {
                    if (lcpArray[i][j] !== 1) {
                        return "";
                    }
                } else if (lcpArray[i][j] !== lcpArray[i + 1][j + 1] + 1) {
                    return "";
                }
            } else if (lcpArray[i][j]) {
                // Mismatched characters should not have a non-zero LCP.
                return "";
            }
        }
    }

    // If all checks pass, return the constructed string.
    return reconstructedString;
}
```

## Time and Space Complexity

### Time Complexity

The given code has the following steps which contribute to its time complexity:

1. Initializing a list `s` of length `n` - $O(n)$.
2. A nested loop structure where it iterates over the `ascii_lowercase` characters and then over the range from `i` to `n` to populate the string array `s`. Since ASCII lowercase has a fixed number of characters (26), and we assume that each character will be the start of a new string only once, we can approximate the time complexity for this loop as $O(n)$. Even though there are nested loops, the outer loop only increments `i` and does not start from the beginning each time.
3. Checking if there is any empty string `"" in s`. This is done by scanning the list once, resulting in a complexity of $O(n)$.
4. Finally, there's a nested loop that compares elements in the `lcp` array to verify the longest common prefix properties. This loop has a worst-case time complexity of $O(n^2)$ as it iterates over each element twice in the worst case.

Therefore, the total time complexity can be approximately expressed as $O(n) + O(n) + O(n^2)$ which simplifies to $O(n^2)$ since the quadratic term dominates for large $n$.

### Space Complexity

The space complexity is determined by:

1. The list `s` of length `n` - $O(n)$.
2. No additional significant space is utilized within the loops.

Thus, the space complexity of the code is $O(n)$.