2370. Longest Ideal Subsequence

Medium Hash Table String Dynamic Programming Leetcode Link

Problem Description

The LeetCode problem presented asks us to find the length of the longest ideal subsequence from a given string `s`. An ideal subsequence is one that satisfies two conditions. First, it must be a subsequence of `s`, which means it can be formed by removing some or no characters from `s` without changing the order of the remaining characters. Second, for every pair of adjacent letters in this subsequence, the absolute difference in their alphabetical order must be less than or equal to a given integer `k`.

To explain further, a `subsequence` is different from a `substring` since a `substring` is contiguous and appears in the same order within the original string, whereas a `subsequence` is not required to be contiguous but must appear in the same relative order.

The problem statement also clarifies that the alphabetical order difference isn't cyclic, meaning that the difference between `a` and `z` is considered to be 25, not 1.

Intuition

To solve this problem, we can use dynamic programming, which involves breaking the problem into smaller subproblems and solving each subproblem just once, storing the solution in a table for easy access when solving subsequent subproblems.

The intuition behind the approach is to consider each character in the string `s` one by one and determine the length of the longest ideal subsequence ending with that character. For each character, we need to find the previous character in the subsequence that can be connected to it (based on the condition about the alphabetical difference) for which the subsequence length is the longest possible.

We create a dictionary `d` that keeps track of the last occurrence index for each character that can be a candidate for forming a subsequence. We also maintain an array `dp` where `dp[i]` represents the length of the longest ideal subsequence ending at `s[i]`.

For each character `s[i]` in the string, we iterate over every lowercase letter and check if it can form a sequence with `s[i]`, satisfying the absolute difference condition. If it's a valid character, we update `dp[i]` to the maximum of its current value or 1 plus the length of the subsequence ending with the valid character found in our dictionary `d`. By doing this iteratively, we build upon the solutions of the subproblems until we reach the end of the string. Finally, the answer is the maximum value in the `dp` array, which represents the length of the longest ideal subsequence we can obtain from `s`.

Solution Approach

The implementation of the solution provided follows the dynamic programming approach to tackle the problem of finding the longest ideal subsequence.

Firstly, we initialize an array `dp` with all elements as `1`, because the minimum length of an ideal subsequence that can end at any character is at least `1` – the character itself. We also create a dictionary `d` to keep track of the last index where each character occurred while checking the sequence.

The core of the solution code is a nested loop. The outer loop iterates through the string `s`, and the inner loop iterates through all lowercase ASCII letters to check whether they can form an ideal subsequence with the current character `s[i]`. Here's a breakdown of each element of the code:

- `n = len(s)`: We determine the length of the given string `s`.
- `dp = [1] * n`: Creates a list `dp` with `n` elements, all initialized to `1`.
- `d = {s[0]: 0}`: Initializes the dictionary `d` with the first character of the string `s` as the key and `0` as its index.
- `for i in range(1, n)`: Outer loop to traverse the string `s` starting from index `1` since index `0` is used for initialization.
- `a = ord(s[i])`: Converts the current character `s[i]` to its ASCII value to facilitate comparison based on alphabetical order.
- Inside the outer loop is an inner loop: `for b in ascii_lowercase`:
  - `abs(a - ord(b)) > k`: Checks if the absolute difference between the ASCII values of the characters is greater than `k`, which means they cannot form an ideal sequence.
  - `if b in d`: If character `b` is within the difference `k`, and it's in dictionary `d`, then it's a candidate for creating or extending an ideal subsequence.
  - `dp[i] = max(dp[i], dp[d[b]] + 1)`: Here, the dynamic programming comes into play. We update the `dp[i]` with the maximum of its current value or the length of the ideal subsequence ending with `b` plus one (to include the current character `s[i]`).
- `d[s[i]] = i`: We update or add the current character and its index to dictionary `d` to keep track of the last position it was seen.
- `return max(dp)`: After filling the `dp` table, the length of the longest ideal subsequence will be the maximum value in the `dp` list.

By keeping track of the ideal subsequences ending with different characters and building upon each other, this algorithm efficiently finds the longest one possible.

Example Walkthrough

Let's consider a string `s = "abcd"` and `k = 2`, which signifies the maximum allowed absolute difference in the alphabetical order.

To illustrate the solution approach, we'll walk through the example step-by-step:

1. **Initialization**: We have `n = 4` since the length of `s` is 4. We create a `dp` list of length `n` initialized with all `1`s because the minimum subsequence length ending at each character is 1 (the character itself). We also have a dictionary, `d`, which will keep track of the last seen index for each character.

2. **Starting Point**: With `s[0] = 'a'`, we initialize `d` with `{'a': 0}`. Since 'a' is the first character, `dp[0] = 1`.

3. **Iteration with `i = 1` for `s[i] = 'b'`**:
   - We check if 'a' can form an ideal subsequence with 'b'. Since `abs(ord('a') - ord('b')) = 1` which is less than or equal to `k`, 'a' can form an ideal subsequence with 'b'.
   - Thus, we update `dp[1]` with the maximum of `dp[1]` and `dp['a' index] + 1`. Since `dp[0] = 1`, this means `dp[1]` becomes `2`.
   - Update `d` to include `'b': 1`.
4. **Iteration with `i = 2` for `s[i] = 'c'`**:
   - We check 'a' and 'b' to form an ideal subsequence with 'c'. Both are within `k` distance.
   - Since both 'a' and 'b' can precede 'c', we choose the one that provides the longest subsequence. `dp[2]` is updated to `max(dp[2], dp['b' index] + 1)`, which is `3`.
   - Update `d` to include `'c': 2`.
5. **Iteration with `i = 3` for `s[i] = 'd'`**:
   - Repeat the above step for 'd', checking 'a', 'b', and 'c'. Again, all are within `k` distance.
   - We again choose the best predecessor, which is 'c' in this case. We update `dp[3]` to `max(dp[3], dp['c' index] + 1)`, which is `4`.
   - Update `d` to include `'d': 3`.
6. **Finish**: We have processed all characters in `s`. The `dp` array is `[1, 2, 3, 4]`, and thus the length of the longest ideal subsequence is `max(dp)` which is `4`.

In this example, the longest ideal subsequence we can form from "abcd" with `k = 2` is "abcd" itself, with the length of `4`. However, for strings with repeating characters or larger alphabetic differences, we would see more variation in the `dp` updates based on which preceding characters provide the longer ideal subsequence according to the `k` condition.

Python Solution

```python
from string import ascii_lowercase

class Solution:
    def longestIdealString(self, s: str, k: int) -> int:
        # The length of the input string
        string_length = len(s)

        # Initialize the dp array where dp[i] represents the length of the longest ideal substring ending at index i
        dp = [1] * string_length

        # Dictionary to keep track of the last index of each character we have encountered so far.
        last_index_dict = {s[0]: 0}

        # Iterate through the string starting from the first character
        for i in range(1, string_length):
            # Get the ascii value of the current character
            current_char_ascii = ord(s[i])

            # Loop through all lowercase letters to find characters within the ideal distance from the current character
            for b in ascii_lowercase:
                if abs(current_char_ascii - ord(b)) > k:
                    # If the ascii difference is greater than k, skip this character
                    continue
                # If the character is within the ideal distance and we've seen this character before
                if b in last_index_dict:
                    # Update the dp value by considering the length of the ideal substring ending at the last index of character b pl
                    dp[i] = max(dp[i], dp[last_index_dict[b]] + 1)

            # Update the last index for the current character
            last_index_dict[s[i]] = i

        # Return the maximum value in the dp array which represents the length of the longest ideal substring
        return max(dp)

```

Java Solution

```java
class Solution {
    public int longestIdealString(String s, int k) {
        int lengthOfString = s.length(); // The length of the string s.
        int longestLength = 1; // Initialize the answer to 1 character as the minimum length of ideal string.
        int[] dynamicProgramming = new int[lengthOfString]; // Array to store the length of the longest ideal substring ending at eac
        Arrays.fill(dynamicProgramming, 1); // At minimum, each character itself is an ideal string.

        // HashMap to keep track of the last index of each character that is part of the longest ideal string so far.
        Map<Character, Integer> lastSeenCharacterMap = new HashMap<>(26);

        // Place the first character in the map to start the process.
        lastSeenCharacterMap.put(s.charAt(0), 0);

        // Iterate through each character in the string starting from the second character.
        for (int i = 1; i < lengthOfString; ++i) {
            char currentChar = s.charAt(i); // The current character for which we are finding the ideal string.

            // Check closeness of every character in the alphabet to the current character within the limit k.
            for (char prevChar = 'a'; prevChar <= 'z'; ++prevChar) {
                // If the absolute difference in ASCII value is within the limit k, we proceed.
                if (Math.abs(currentChar - prevChar) > k) {
                    continue;
                }
                // If the previous character has been seen and is part of an ideal string,
                // we update the DP table by taking the max of the current dp value or
                // the dp value of the previous character's last index plus one.
                if (lastSeenCharacterMap.containsKey(prevChar)) {
                    dynamicProgramming[i] = Math.max(dynamicProgramming[i], dynamicProgramming[lastSeenCharacterMap.get(prevChar)] +
                }
            }

            // Update the last seen index of the current character to be the current index.
            lastSeenCharacterMap.put(currentChar, i);
            // Update longestLength to be the maximum of itself and the current length of the longest ideal string ending at i.
            longestLength = Math.max(longestLength, dynamicProgramming[i]);
        }

        // Return the length of the longest ideal string that can be formed from the input string.
        return longestLength;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int longestIdealString(string s, int k) {
        int stringLength = s.size(); // The length of the input string
        int longestLength = 1; // Initialize the longest length with 1, as the minimum ideal string length is 1
        vector<int> dp(stringLength, 1); // Dynamic programming table with a base case of 1 for each character
        unordered_map<char, int> lastOccurrence; // Stores the last occurrence index of each character encountered

        // Initialize the last occurrence for the first character in the string
        lastOccurrence[s[0]] = 0;

        // Iterate over the string starting from the second character
        for (int i = 1; i < stringLength; ++i) {
            char currentChar = s[i]; // Current character being processed

            // Try extending the ideal string including all characters within 'k' distance of current character
            for (char otherChar = 'a'; otherChar <= 'z'; ++otherChar) {
                // If the other character is more than 'k' distance away, skip it
                if (abs(currentChar - otherChar) > k) continue;

                // Check if we have seen the other character before and extend the ideal string length if possible
                if (lastOccurrence.count(otherChar))
                    dp[i] = max(dp[i], dp[lastOccurrence[otherChar]] + 1);
            }

            // Update the last occurrence index for the current character
            lastOccurrence[currentChar] = i;

            // Update the longest length found so far
            longestLength = max(longestLength, dp[i]);
        }

        // Return the length of the longest ideal string found
        return longestLength;
    }
};

```

Typescript Solution

```typescript
function longestIdealString(s: string, k: number): number {
    // Create a dynamic programming array initialized to 0.
    // This array represents the length of the longest ideal subsequence ending with each letter.
    const longestSubseqLengths = new Array(26).fill(0);

    // Iterate over each character in the input string.
    for (const char of s) {
        // Get the index (0-25) corresponding to the current character.
        const index = char.charCodeAt(0) - 'a'.charCodeAt(0);
        // Temporary variable to hold the maximum subsequence length found so far.
        let maxLength = 0;

        // Iterate over all possible characters to update the current character's maximum length.
        for (let i = 0; i < 26; i++) {
            // If the current character is within 'k' distance of character at index 'i'
            // in the alphabet, update the max length if a longer subsequence is found.
            if (Math.abs(index - i) <= k) {
                maxLength = Math.max(maxLength, longestSubseqLengths[i] + 1);
            }
        }

        // Update the longest subsequence length ending with the current character.
        longestSubseqLengths[index] = Math.max(longestSubseqLengths[index], maxLength);
    }

    // Return the maximum value in the dynamic programming array,
    // which is the length of the longest ideal subsequence found.
    return longestSubseqLengths.reduce((maxValue, currentLength) => Math.max(maxValue, currentLength), 0);
}

```

Time and Space Complexity

The time complexity of the given code is `O(n * 26 * k)` where `n` is the length of the input string `s` and `k` is the maximum difference allowed for the ideal string. For each character in the string `s`, the code iterates over all 26 letters in the `ascii_lowercase` (regardless of `k` because it checks all possibilities within the range) to find characters within the allowed difference `k`. Therefore, we look at all possible `26` characters for each of the `n` positions in the worst case. Additionally, for each character `b` in `ascii_lowercase`, we look up in dictionary `d` which has an average-case lookup time of `O(1)`. Hence, we multiply 26 by this constant lookup time, which doesn't change the O-notation.

The space complexity of the code is `O(n)` because the `dp` array is of size `n`, where `n` is the size of the input string `s`. The dictionary `d` also stores up to `n` index values for all unique characters that have been seen. In this case, the number of unique keys in the dictionary is bounded by the size of the `ascii_lowercase` set, which is a constant (`26`), so it does not change the overall space complexity which is dominated by the `dp` array.