1531. String Compression II

Leetcode Link

Problem Explanation

The problem is about data compression using the Run Length Encoding (RLE) method and further minimising the length of the RLE string by deleting at most `k` characters from the original string. In the RLE method, consecutive similar characters are replaced with a single occurrence of that character followed by the count for the consecutive times it has occurred. For instance, "aaabbb" changes to "a3b3".

The task is to find the minimum length of the string after applying RLE, given that we can delete at most `k` characters from the original string. It's also mentioned that we do not need to add '1' after single characters and the string contains only lowercase English letters.

Let us walk through an example:

s = "aaabcccd", k = 2

If we compress s without deleting anything, the string becomes "a3bc3d", of length 6. If we delete 2 characters of 'a', then we will have s = "abcccd" which on compression gives "abc3d" (length 5). The optimal way is to delete 'b' and 'd', then the compressed version of s will be "a3c3" (length 4).

Solution Approach

The solution uses a recursive approach with memoization. It maintains a `2-D dp` array to store the optimal compression length for every substring of `s` with at most `k deletions`. The function `compression(s, i, k)` calculates the optimal compression length for the `i-th` substring of 's' with at most `k deletions`. If `k is less than 0` or `i equals the length of s` or the remaining length of 's' is less than or equal to 'k', it returns the values accordingly.

We try to make all characters in the substring from `i to j` the same and for that, we keep the character that has a maximum frequency in this range and try to remove other characters.

C++ Solution

```cpp

cpp
class Solution {
    static constexpr int kMax = 101;
    vector<vector<int>> dp;

    int compression(const string& s, int i, int k) {
        if (k < 0)
            return kMax;
        if (i == s.length() || s.length() - i <= k)
            return 0;
        if (dp[i][k] != kMax)
            return dp[i][k];

        int maxFreq = 0;
        vector<int> count(128);

        for (int j = i; j < s.length(); ++j) {
            maxFreq = max(maxFreq, ++count[s[j]]);
            dp[i][k] = min(
                dp[i][k],
                getLength(maxFreq) +
                compression(s, j + 1, k - (j - i + 1 - maxFreq)));
        }

        return dp[i][k];
    }

    int getLength(int maxFreq) {
        if (maxFreq == 1)
            return 1;
        if (maxFreq < 10)
            return 2;
        if (maxFreq < 100)
            return 3;
        return 4;
    }

public:
    int getLengthOfOptimalCompression(string s, int k) {
        dp.resize(s.length(), vector<int>(k + 1, kMax));
        return compression(s, 0, k);
    }
};
```

Python Solution

Python solution also uses dynamic programming. It creates a dp array to keep track of minimum length of the RLE string by checking all possible chars from 'a' to 'z'. In the outer loop, it iterates on the original string backwards and in the inner loop, it checks for the number of same characters in the range. Then it updates the `dp[i+1][del]` based on the current min length.

```python

python
class Solution:
    def getLengthOfOptimalCompression(self, s: str, k: int) -> int:
        C = 26
        n = len(s)
        counter = [0] * C
        dp = [[0] * (k + 1) for _ in range(n + 1)]
        for i in range(n - 1, -1, -1):
            dp[i] = dp[i + 1][:]
            counter[ord(s[i]) - ord('a')] += 1
            del_ = 0
            for a in range(C):

                dp[i][del_] = min(dp[i][del_],
                                 (dp[i + counter[a]][del_] if del_ >= counter[a] else 1 +
                                  dp[i + 1][(del_ if a != ord(s[i]) - ord('a') else del_ - 1) if del_ > 0 else 0]) +
                                 len(str(counter[a])) if counter[a] > 1 else 0)
                if a != ord(s[i]) - ord('a'):
                    del_ += counter[a]
                    counter[a] = 0
          return dp[0][k]
```

Java Solution

Java solution uses a recursive approach. It also uses a dp array of size [26][n+1][k+1] to keep track of minimum length for each char from 'a' to 'z' and max ‘k’ deletions.

```java

java
class Solution {
  public int getLengthOfOptimalCompression(String s, int k) {
    char[] chars = s.toCharArray();
    int[][][] dp = new int[26][chars.length + 2][chars.length + 1];
    return helper(chars, new char[chars.length + 2], 0, 0, 0, k, dp);
  }

  private int helper(char[] chars, char[] path, int i, int j, int last, int k, int[][][] dp) {
    if (k < 0) {
      return Integer.MAX_VALUE / 2;
    }
    if (i == chars.length) {
      return 0;
    }
    char c= chars[i];
    if (c == last) {
      path[j] = c;
      int len = j == 0 ? 0 : (j <= 2 ? 1 : (j <= 10 ? 2 : (j <= 100 ? 3 : 4)));
      return len + helper(chars, path, i + 1, j + 1, last, k, dp);
    }
    if (dp[c - 'a'][j][k] != 0) {
      return dp[c - 'a'][j][k];
    }
    path[0] = c;
    int keep = helper(chars, path, i + 1, 1, c, k, dp);
    int skip = helper(chars, path, i + 1, j, last, k - 1, dp);
    dp[c - 'a'][j][k] = Math.min(keep, skip);
    return dp[c - 'a'][j][k];
  }
}
```

In these solutions, dynamic programming is used to reduce the time complexity and to avoid recomputation. The important factor to consider here is the choice of data structure and how it is used to store the computed results of smaller problems. Depending on the selected approach, the solution can vary but the basic idea remains the same.