2030. Smallest K Length Subsequence With Occurrences of a Letter

Leetcode Link

Problem Description

Given a string `s`, we need to find the lexicographically smallest subsequence of length `k` that contains the character `letter` exactly `repetition` number of times.

** Constraints:**

- 1 <= k <= s.length <= 1000
- 1 <= repetition <= k <= 1000
- 1 <= letter.length == 1
- s and letter consist of only lowercase English letters.

Example

Let's walk through an example:

Input: s = "leetcode", k = 4, letter = 'e', repetition = 2

Output: "eecd"

Explanation: The lexicographically smallest subsequence that meets the requirement is "eecd" with 2 'e' characters.

Approach

The main idea of the solution is to use a stack data structure to maintain the desired subsequence characters. We can iterate through the input string, and for each character, we try to keep the stack in lexicographically increasing order if the remaining characters and constraints allow us to do so.

There are three cases we need to cover:

1. If the character is equal to `letter`, push it onto the stack and decrement the `required` count.
2. If the character is not equal to `letter` and we can still push more characters onto the stack to meet the length of `k`, push the character.
3. If the character is equal to `letter` but our stack is already full (`stack.size() == k`), don't push it to the stack.

Finally, we convert the stack into a string and return it as the answer.

ASCII Illustration

Suppose `s = "leetcode", k = 4, letter = 'e', repetition = 2`

Initial state:

- `stack = []`
- `required = 2`
- `nLetters = 3` (number of 'letter' in the input string)

Processing each character of the input string:

s[0]: 'l'

- `stack = ['l']`

s[1]: 'e'

- `stack = ['e']` (pop 'l' since we need to add 'e')

s[2]: 'e'

- `stack = ['e', 'e']`

s[3]: 't'

- Ignore (since we've already added the required number of 'e')

s[4]: 'c'

- `stack = ['e', 'e', 'c']`

s[5]: 'o'

- Ignore (since adding 'o' would make the sequence lexographically larger)

s[6]: 'd'

- `stack = ['e', 'e', 'c', 'd']` (our final subsequence)

s[7]: 'e'

- Ignore (already added the required number of 'e')

Final answer: "eecd".

Solution in Python

```python
class Solution:
    def smallestSubsequence(self, s: str, k: int, letter: str, repetition: int) -> str:
        stack = []
        required = repetition
        nLetters = s.count(letter)

        for i in range(len(s)):
            c = s[i]
            while stack and stack[-1] > c and len(stack) + len(s) - i - 1 >= k and (stack[-1] != letter or nLetters > required):
                popped = stack.pop()
                if popped == letter:
                    required += 1
            if len(stack) < k:
                if c == letter:
                    stack.append(c)
                    required -= 1
                elif k > len(stack) + required:
                    stack.append(c)
            if c == letter:
                nLetters -= 1

        return "".join(stack)
```

Solution in Java

```java
import java.util.*;

class Solution {
    public String smallestSubsequence(String s, int k, char letter, int repetition) {
        int ans = 0;
        List<Character> stack = new ArrayList<>();
        int required = repetition;
        int nLetters = (int) s.chars().filter(c -> c == letter).count();

        for (int i = 0; i < s.length(); ++i) {
            char c = s.charAt(i);
            while (!stack.isEmpty() && stack.get(stack.size() - 1) > c && stack.size() + s.length() - i - 1 >= k && (stack.get(stack.
                char popped = stack.remove(stack.size() - 1);
                if (popped == letter)
                    ++required;
            }
            if (stack.size() < k)
                if (c == letter) {
                    stack.add(c);
                    --required;
                } else if (k > stack.size() + required) {
                    stack.add(c);
                }
            if (c == letter)
                --nLetters;
        }

        for (char c : stack)
            ans += c;

        return ans;
    }
}
```

Solution in JavaScript

```
/**
 * @param {string} s
 * @param {number} k
 * @param {character} letter
 * @param {number} repetition
 * @return {string}
 */
var smallestSubsequence = function(s, k, letter, repetition) {
    const stack = [];
    let required = repetition;
    let nLetters = s.split(letter).length - 1;

    for (let i = 0; i < s.length; ++i) {
        const c = s[i];
        while (stack.length > 0 && stack[stack.length - 1] > c && stack.length + s.length - i - 1 >= k && (stack[stack.length - 1
            const popped = stack.pop();
            if (popped === letter)
                ++required;
        }
        if (stack.length < k)
            if (c === letter) {
                stack.push(c);
                --required;
            } else if (k > stack.length + required) {
                stack.push(c);
            }
        if (c === letter)
            --nLetters;
    }

    for (const c of stack)
        ans += c;

    return ans;
};
```

Solution in C++

```
#pragma
#include <vector>
#include <string>
#include <algorithm>

class Solution {
 public:
  string smallestSubsequence(std::string s, int k, char letter, int repetition) {
    std::string ans;
    std::vector<char> stack;
    int required = repetition;
    int nLetters = count(begin(s), end(s), letter);

    for (int i = 0; i < s.length(); ++i) {
      const char c = s[i];
      while (!stack.empty() && stack.back() > c &&
             stack.size() + s.length() - i - 1 >= k &&
             (stack.back() != letter || nLetters > required)) {
        const char popped = stack.back();
        stack.pop_back();
        if (popped == letter)
          ++required;
      }
      if (stack.size() < k)
        if (c == letter) {
          stack.push_back(c);
          --required;
        } else if (k > stack.size() + required) {
          stack.push_back(c);
        }
      if (c == letter)
        --nLetters;
    }

    for (const char c : stack)
      ans += c;

    return ans;
  }
};
```

Solution in C#

```
//
// csharp
using System;
using System.Collections.Generic;

public class Solution {
    public string SmallestSubsequence(string s, int k, char letter, int repetition) {
        string ans = "";
        List<char> stack = new List<char>();
        int required = repetition;
        int nLetters = s.Length - s.Replace(letter.ToString(), "").Length;

        for (int i = 0; i < s.Length; ++i) {
            char c = s[i];
            while (stack.Count > 0 && stack[^1] > c && stack.Count + s.Length - i - 1 >= k && (stack[^1] != letter || nLetters > requ
                char popped = stack[stack.Count - 1];
                stack.RemoveAt(stack.Count - 1);
                if (popped == letter)
                    ++required;
            }
            if (stack.Count < k)
                if (c == letter) {
                    stack.Add(c);
                    --required;
                } else if (k > stack.Count + required) {
                    stack.Add(c);
                }
            if (c == letter)
                --nLetters;
        }

        foreach (char c in stack)
            ans += c;

        return ans;
    }
}
```

Explanation of the Solutions

In this problem, we need to find the smallest subsequence of length `k` containing the character `letter` exactly `repetition` times. We use a stack to push the characters of the input string while maintaining the lexicographically smallest subsequence.

Let's discuss the intuition and code implementation of each solution language.

Python Solution

The python solution defines a class `Solution` with a method `smallestSubsequence` that takes three parameters: `s`, `k`, `letter`, and `repetition`.

1. Initialize an empty stack (`stack = []`), a variable `required` equal to `repetition`, and a variable `nLetters` equal to the count of occurrences of the `letter` in the input string `s`.
2. Iterate through the input string using the loop `for i in range(len(s))`. At each iteration, store the current character in variable `c`, and handle three cases mentioned before.
3. If the character is higher in the lexicographical order, then pop it from the stack to maintain the lexicographically smallest subsequence.
4. If the character is equal to the letter and the stack size is less than `k`, push `c` to the stack and decrease the required count.
5. If the character is different from the letter and the stack size plus required is less than `k`, push `c` to the stack.
6. If the character is equal to the letter, decrease the `nLetters` count.
7. Join the stack to form a string and return it as the answer.

Java Solution

The Java solution is similar to the Python solution but uses a List to hold the subsequence characters instead of a list as in the Python solution. Also, instead of using the count method of the String class, we use a lambda expression and a stream filter to count the occurrences of the letter.

JavaScript Solution

In the JavaScript solution, we use the method splIt to count the occurrences of the letter in the input string s. The rest of the code follows the same steps as in the Python solution, but we use methods push and pop to add and remove elements from the stack, respectively.

C++ Solution

The C++ solution uses a vector of char to hold the subsequence characters. It uses the count method of the algorithm library to count the occurrences of the letter in the input string s. The code follows the same steps as in the Python solution, and the stack is a vector of char.

C# Solution

The C# solution uses a List to hold the subsequence characters and a lambda expression to count the occurrences of the letter in the input string s. It follows the same steps as in the Python solution, but using the methods Add and RemoveAt to add and remove elements from the stack, respectively.