1520. Maximum Number of Non-Overlapping Substrings

Problem Explanation

The problem requires to maximize the number of non-overlapping substrings that follow a rule: If a substring includes a character that character should appear only in this substring, and we only maintain the unique substring with minimal length. The input is a string of lowercase letters. The output should be an array of strings, which are the substrings.

Let's go through an example, with the string `adefaddaccc`, the following are all the possible substrings that meet the conditions:

```
"adefaddaccc"
"adefadda",
"ef",
"e",
"f",
"ccc",
```

If we choose the first string, we cannot choose anything else and we'd get only 1. If we choose "adefadda", we are left with "ccc" which is the only one that doesn't overlap, thus obtaining 2 substrings. Notice also, that it's not optimal to choose "ef" since it can be split into two. Therefore, the optimal way is to choose ["e","f","ccc"] which gives us 3 substrings. No other solution of the same number of substrings exists.

Approach & Algorithm

The approach that is used in the solution is greedy. First, we record the leftmost index and rightmost index of each letter. Then, for each character that appears as the leftmost in the string, if it forms a valid result(as explained above) then it must be the best result ending there.

The algorithm works as follows: Initialize the leftmost and rightmost index for each character. For each character, if it's the leftmost occurrence Check if it forms a valid solution: If it's invalid, then ignore this solution If it's valid: If this solution overlaps with the previous solution, then replace the previous solution If it does not overlap, then add this solution.

Python Solution

```python
class Solution:
    def maxNumOfSubstrings(self, s: str) -> List[str]:
        n = len(s)
        left = [n] * 26
        right = [0] * 26

        # Record the leftmost and rightmost index for each character.
        for i in range(n):
            index = ord(s[i]) - ord('a')
            left[index] = min(left[index], i)
            right[index] = i

        res = []
        r = -1

        # For each character (if it's the leftmost occurrence),
        # check if it forms a valid solution.
        for i in range(n):
            if i != left[ord(s[i]) - ord('a')]:
                continue
            new_r = right[ord(s[i]) - ord('a')]
            j = i + 1
            while (j < new_r + 1) :
                if left[ord(s[j]) - ord('a')] < i:
                    print
                    new_r = n
                    break
                new_r = max(new_r, right[ord(s[j]) - ord('a')])
                j = j + 1
            if new_r < n and (i > r or new_r < right[ord(s[r]) - ord('a')]):
                if i > r:
                    res.append(s[i:new_r + 1])
                else:
                    res[-1] = s[i:new_r + 1]
                r = new_r

        return res
```

Java Solution

```java
class Solution {
    public List<String> maxNumOfSubstrings(String s) {
        int n = s.length();
        int[] left = new int[26], right = new int[26];
        Arrays.fill(left, n);
        // Record the leftmost and rightmost index for each character.
        for (int i = 0; i < n; ++i) {
            left[s.charAt(i) - 'a'] = Math.min(left[s.charAt(i) - 'a'], i);
            right[s.charAt(i) - 'a'] = i;
        }
        List<String> res = new ArrayList<>();
        int l = -1, r = -1;
        // For each character (if it's the leftmost occurrence),
        // check if it forms a valid solution.
        for (int i = 0; i < n; ++i) {
            if (i != left[s.charAt(i) - 'a']) continue;
            int newR = right[s.charAt(i) - 'a'];
            for (int j = i + 1; j <= newR; ++j) {
                if (left[s.charAt(j) - 'a'] < i) {
                    newR = n;
                    break;
                }
                newR = Math.max(newR, right[s.charAt(j) - 'a']);
            }
            if (newR < n && (i > r || newR < right[s.charAt(r) - 'a'])) {
                if (i > r) res.add(s.substring(i, newR + 1));
                else res.set(res.size() - 1, s.substring(i, newR + 1));
                l = i;
                r = newR;
            }
        }
        return res;
    }
}
```

JavaScript Solution

```javascript
var maxNumOfSubstrings = function(s) {
    const n = s.length;
    const left = Array(26).fill(n);
    const right = Array(26).fill(-1);
    for (let i = 0; i < n; ++i) {
        left[s.charCodeAt(i) - 97] = Math.min(left[s.charCodeAt(i) - 97], i);
        right[s.charCodeAt(i) - 97] = i;
    }
    let l = -1, r = -1;
    const res = [];
    for (let i = 0; i < n; ++i) {
        if (i == left[s.charCodeAt(i) - 97]) {
            var new_r = right[s.charCodeAt(i) - 97];
            for (let j = i; j <= new_r; ++j) {
                if (left[s.charCodeAt(j) - 97] < i) {
                    new_r = n;
                    break;
                }
                new_r = Math.max(new_r, right[s.charCodeAt(j) - 97]);
            }
            if (new_r < n && (l == -1 || r < new_r)) {
                if (i > r && l != -1) res.pop();
                res.push(s.substring(i, new_r + 1));
                r = new_r;
            }
        }
    }
    return res;
}
```

The above solutions present the general approach which is same for all three languages, but, the syntax is different for all. The solution first records the leftmost and rightmost index for each character and proceeds with a 'greedy' process where if it finds that the character is a leftmost occurrence, it tries to create a valid substring. If it doesn't form a valid substring, this substring is ignored. Otherwise, it is checked if this solution overlaps with the previous solution and accordingly either replaces or adds to the previous outcome.

These solutions are written in a very efficient manner, taking into account the various possibilities and conditions that can occur in the given problem. Key data structures like 'list' or 'arrays' are used for storing the necessary information for execution.

Each language solution follows good coding practices like needful comments, proper naming conventions and readable format which makes them easy to understand for anyone familiar with the respective languages.

Regardless of the language used, the key understanding lies in the logic of how substrings are formed depending on their validity and on whether or not they overlap with previous substrings. That they are solved in different common programming languages shows the universal applicability of the problem-solving logic. The mix of high-level reasoning and a deep understanding of language-specific techniques show a balanced approach to problem-solving in coding, covering a wide variety of difficulties one may face when called to code outside their comfort zone.