828. Unique Letter String (/problems/unique-letter-string/)

May 5, 2018 | 478 views

A character is unique in string `S` if it occurs exactly once in it.

For example, in string `S = "LETTER"`, the only unique characters are `"L"` and `"R"`.

Let's define `UNIQ(S)` as the number of unique characters in string `S`.

For example, `UNIQ("LETTER") = 2`.

Given a string S, calculate the sum of `UNIQ(substring)` over all non-empty substrings of `S`.

If there are two or more equal substrings at different positions in `S`, we consider them different.

Since the answer can be very large, retrun the answer modulo `10 ^ 9 + 7`.

Example 1:

```
Input: "ABC"
Output: 10
Explanation: All possible substrings are: "A","B","C","AB","BC" and "ABC".
Evey substring is composed with only unique letters.
Sum of lengths of all substring is 1 + 1 + 1 + 2 + 2 + 3 = 10
```

Example 2:

```
Input: "ABA"
Output: 8
Explanation: The same as example 1, except uni("ABA") = 1.
```

Note: `0 <= S.length <= 10000`.

Approach #1: Maintain Answer of Suffix [Accepted]

Intuition

We can think of substrings as two for-loops, for the left and right boundary of the substring. To get a handle on this problem, let's try to answer the question: what is the answer over all substrings that start at index `i`? Let's call this $F(i)$. If we can compute this faster than linear (brute force), we have an approach.

Now let U be the unique letters function, eg. $U(\text{"LETTER"}) = 2$.

The key idea is we can write U as a sum of disjoint functions over each character. Let $U_{\text{"A"}}(x)$ be 1 if "A" occurs exactly once in x, otherwise 0, and so on with every letter. Then $U(x) = \sum_{c \in \mathcal{A}} U_c(x)$, where $\mathcal{A} = \{\text{"A"}, \text{"B"}, \dots\}$ is the alphabet.

Algorithm

This means we only need to answer the following question (26 times, one for each character): how many substrings have exactly one "A"? If we knew that `S[10] = S[14] = S[20] = "A"` (and only those indexes have an `"A"`), then when `i = 8`, the answer is `4` (`j = 10, 11, 12, 13`); when `i = 12` the answer is `6` (`j = 14, 15, 16, 17, 18, 19`), and so on.

In total, $F(0) = \sum_{c \in \mathcal{A}} \text{index}[c][1] - \text{index}[c][0]$, where `index[c]` are the indices `i` (in order) where `S[i] == c` (and padded with `S.length` if out of bounds). In the above example, `index["A"] = [10, 14, 20]`.

Now, we want the final answer of $\sum_{i \geq 0} F(i)$. There is a two pointer approach: how does $F(1)$ differ from $F(0)$? If for example `S[0] == "B"`, then most of the sum remains unchanged (specifically, $\sum_{c \in \mathcal{A}, c \neq \text{"B"}} \text{index}[c][1] - \text{index}[c][0]$), and only the $c = \text{"B"}$ part changes, from $\text{index}[\text{"B"}][1] - \text{index}[\text{"B"}][0]$ to $\text{index}[\text{"B"}][2] - \text{index}[\text{"B"}][1]$.

We can manage this in general by keeping track of `peek[c]`, which tells us the correct index `i = peek[c]` such that our current contribution by character `c` of $F(i)$ is `index[c][peek[c] + 1] - index[c][peek[c]]`.

Java | Python

Copy

```python
class Solution(object):
    def uniqueLetterString(self, S):
        N = len(S)
        index = collections.defaultdict(list)
        peek = collections.defaultdict(int)
        for i, c in enumerate(S):
            index[c].append(i)
        for c in index:
            index[c].extend([N, N])

        def get(c):
            return index[c][peek[c] + 1] - index[c][peek[c]]

        ans = 0
        cur = sum(get(c) for c in index)
        for i, c in enumerate(S):
            ans += cur
            oldv = get(c)
            peek[c] += 1
            cur += get(c) - oldv
        return ans % (10**9 + 7)
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is the length of `S`.
- Space Complexity: $O(N)$.

Approach #2: Split by Character [Accepted]

Intuition U为字符只出现一次的个数

As in *Approach #1*, we have $U(x) = \sum_{c \in \mathcal{A}} U_c(x)$, where $\mathcal{A} = \{\text{"A"}, \text{"B"}, \dots\}$ is the alphabet, and we only need to answer the following question (26 times, one for each character): how many substrings have exactly one "A"?

Algorithm 我们只需要计算有多少个子串含有这个字符(1次)

Consider how many substrings have a specific "A". For example, let's say `S` only has three `"A"`'s, at '`S[10] = S[14] = S[20] = "A"`; and we want to know the number of substrings that contain `S[14]`. The answer is that there are 4 choices for the left boundary of the substring `(11, 12, 13, 14)`, and 6 choices for the right boundary `(14, 15, 16, 17, 18, 19)`. So in total, there are 24 substrings that have `S[14]` as their unique `"A"`.

Continuing our example, if we wanted to count the number of substrings that have `S[10]`, this would be `10 * 4` - note that when there is no more `"A"` characters to the left of `S[10]`, we have to count up to the left edge of the string.

We can add up all these possibilities to get our final answer.

对于当前的某个字符位置，计算其左边子串长度和右边子串长度，两者长度相乘，即为该字符只出现一次的子串个数

Java | Python

Copy

```
class Solution(object):
    def uniqueLetterString(self, S):
        index = collections.defaultdict(list)
        for i, c in enumerate(S):
            index[c].append(i)

        ans = 0
        for A in index.values():
            A = [-1] + A + [len(S)]
            for i in xrange(1, len(A) - 1):
                ans += (A[i] - A[i-1]) * (A[i+1] - A[i])
        return ans % (10**9 + 7)
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is the length of `S`.
- Space Complexity: $O(N)$. We could reduce this to $O(\mathcal{A})$ if we do not store all the indices, but compute the answer on the fly.

Analysis written by: @awice (https://leetcode.com/awice). Approach #2 inspired by @lee215 (http://leetcode.com/lee215).

Rate this article:

Comments: 0

Sort By

Login to Comment (/accounts/login/?next=/articles/unique-letter-string/)