Longest Substring Without Repeating Characters

LeetCode Admin Apr 11, 2016

Video Solution

13:23

Solution Article

Overview

The primary challenge in this problem is to find an efficient way to get all possible longest substrings that contain no duplicate characters. To achieve this, we need to take advantage of Hash Table, which checks if a character occurs before quickly.

In the following three approaches, we utilize a hash table to guarantee substrings have no repeating characters and optimize the algorithm to query possible substrings step by step: the

first approach is intuitive but may cause a TLE, and the second one uses a slide window method to narrow down the search range, and the third make further use of HashMap to reduce the search range faster.

Approach 1: Brute Force

Intuition

Check all the substring one by one to see if it has no duplicate character.

Algorithm

Suppose we have a function `boolean allUnique(String substring)` which will return true if the characters in the substring are all unique, otherwise false. We can iterate through all the possible substrings of the given string `s` and call the function `allUnique`. If it turns out to be true, then we update our answer of the maximum length of substring without duplicate characters.

Now let's fill the missing parts:

1. To enumerate all substrings of a given string, we enumerate the start and end indices of them. Suppose the start and end indices are i and j, respectively. Then we have $0 \leq i < j \leq n$ (here end index j is exclusive by convention). Thus, using two nested loops with i from 0 to $n-1$ and j from $i+1$ to n, we can enumerate all the substrings of `s`.

2. To check if one string has duplicate characters, we can use a set. We iterate through all the characters in the string and put them into the `set` one by one. Before putting one character, we check if the set already contains it. If so, we return `false`. After the loop, we return `true`.

Implementation

C++ | Java | Python3 — Copy

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int n = s.length();

        int res = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                if (checkRepetition(s, i, j)) {
                    res = max(res, j - i + 1);
                }
            }
        }

        return res;
    }

    bool checkRepetition(string& s, int start, int end) {
        unordered_set<char> chars;

        for (int i = start; i <= end; i++) {
            char c = s[i];
            if(chars.count(c)){
                return false;
            }
            chars.insert(c);
        }
```

Complexity Analysis

- Time complexity : $O(n^3)$.

  To verify if characters within index range $[i, j)$ are all unique, we need to scan all of them. Thus, it costs $O(j-i)$ time.

  For a given `i`, the sum of time costed by each $j \in [i+1, n]$ is

  $\sum_{i+1}^{n} O(j-i)$

  Thus, the sum of all the time consumption is:

  $$O\left(\sum_{i=0}^{n-1}\left(\sum_{j=i+1}^{n}(j-i)\right)\right) = O\left(\sum_{i=0}^{n-1}\frac{(1+n-i)(n-i)}{2}\right) = O(n^3)$$

- Space complexity : $O(min(n, m))$. We need $O(k)$ space for checking a substring has no duplicate characters, where k is the size of the `Set`. The size of the Set is upper bounded by the size of the string n and the size of the charset/alphabet m.

Approach 2: Sliding Window

Intuition

Given a substring with a fixed end index in the string, maintain a HashMap to record the frequency of each character in the current substring. If any character occurs more than once, drop the leftmost characters until there are no duplicate characters.

Algorithm

The naive approach is very straightforward. But it is too slow. So how can we optimize it?

In the naive approaches, we repeatedly check a substring to see if it has duplicate character. But it is unnecessary. If a substring s_{ij} from index i to $j-1$ is already checked to have no duplicate characters. We only need to check if $s[j]$ is already in the substring s_{ij}.

To check if a character is already in the substring, we can scan the substring, which leads to an $O(n^2)$ algorithm. But we can do better.

By using HashSet as a sliding window, checking if a character in the current can be done in $O(1)$.

A sliding window is an abstract concept commonly used in array/string problems. A window is a range of elements in the array/string which usually defined by the start and end indices, i.e. $[i, j)$ (left-closed, right-open). A sliding window is a window "slides" its two boundaries to the certain direction. For example, if we slide $[i, j)$ to the right by 1 element, then it becomes $[i+1, j+1)$ (left-closed, right-open).

Back to our problem. We use HashSet to store the characters in current window $[i, j)$ ($j = i$ initially). Then we slide the index j to the right. If it is not in the HashSet, we slide j further. Doing so until s[j] is already in the HashSet. At this point, we found the maximum size of substrings without duplicate characters start with index i. If we do this for all i, we get our answer.

Implementation

C++ | Java | Python3

Copy

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> chars;

        int left = 0;
        int right = 0;

        int res = 0;
        while (right < s.length()) {
            char r = s[right];
            chars[r]++;

            while (chars[r] > 1) {
                char l = s[left];
                chars[l]--;
                left++;
            }

            res = max(res, right - left + 1);

            right++;
        }

        return res;
    }
};
```

Complexity Analysis

- Time complexity : $O(2n) = O(n)$. In the worst case each character will be visited twice by i and j.
- Space complexity : $O(min(m, n))$. Same as the previous approach. We need $O(k)$ space for the sliding window, where k is the size of the `Set`. The size of the Set is upper bounded by the size of the string n and the size of the charset/alphabet m.

Approach 3: Sliding Window Optimized

Intuition

The above solution requires at most 2n steps. In fact, it could be optimized to require only n steps. Instead of using a set to tell if a character exists or not, we could define a mapping of the characters to its index. Then we can skip the characters immediately when we found a repeated character.

Algorithm

The reason is that if $s[j]$ have a duplicate in the range $[i, j)$ with index j', we don't need to increase i little by little. We can skip all the elements in the range $[i, j']$ and let i to be $j' + 1$ directly.

Here is a visualization of the above code.

02:03

Implementation

C++ | Java | Python3

Copy

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int n = int(s.length()), res = 0;
        unordered_map<char, int> mp;

        for (int j = 0, i = 0; j < n; j++){
            if(mp[s[j]] > 0) {
                i = max(mp[s[j]], i);
            }
            res = max(res, j - i + 1);
            mp[s[j]] = j + 1;
        }
        return res;
    }
};
```

Complexity Analysis

- Time complexity : $O(n)$. Index j will iterate n times.
- Space complexity : $O(min(m,n))$. Same as the previous approach.

Tips

All previous implementations have no assumption on the charset of the string `s` .

If we know that the charset is rather small, we can mimic what a HashSet/HashMap does with a boolean/integer array as direct access table. Though the time complexity of query or insertion is still $O(1)$, the constant factor is smaller in an array than in a HashMap/HashSet. Thus, we can achieve a shorter runtime by the replacement here.

Commonly used tables are:

- `int[26]` for Letters 'a' - 'z' or 'A' - 'Z'
- `int[128]` for ASCII
- `int[256]` for Extended ASCII

C++ | Java | Python3

Copy

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        // we will store a senitel value of -1 to simulate 'null'/'None' in C++
        vector<int> chars(128, -1);

        int left = 0;
        int right = 0;

        int res = 0;
        while (right < s.length()) {
            char r = s[right];

            int index = chars[r];
            if (index != -1 and index >= left and index < right) {
                left = index + 1;
            }
            res = max(res, right - left + 1);

            chars[r] = right;
            right++;
        }
        return res;
    }
};
```

For this implementation, the space complexity is fixed to $O(m)$ while the time complexity keeps unchanged. m is the size of the charset.

Share

Comments (998)

Sort by: **Best**

Type comment here... (Markdown supported)

Preview Comment

Article Commenting Rules

1. This comment section is for questions and comments regarding this **LeetCode article**. All posts must respect our LeetCode Community Rules.

2. Concerns about errors or bugs in the article, problem description, or test cases should be posted on LeetCode Feedback, so that our team can address them.

rohan1239 Jul 30, 2018

Read more

475 Show 37 Replies Reply

leo39032506 Jan 23, 2019

Read more

194 Show 16 Replies Reply

Ravindhark Feb 06, 2019

Read more

18 Show 3 Replies Reply

tryck Apr 07, 2019

Read more

```
let curr = 0;
let hash = {};
if(s.length < 2) {
    return s.length;
}
for(let i = 0; i < s.length; i++) {
```

166 Show 11 Replies Reply

kxguoniu Oct 18, 2018

Read more

296 Show 23 Replies Reply

gzrjzcx Jan 28, 2019

Read more

92 Show 8 Replies Reply

wushi58 Oct 16, 2018

Read more

211 Show 17 Replies Reply

wanderwan... Feb 14, 2019

Read more

71 Show 4 Replies Reply

settiricardo Feb 24, 2019

Read more

83 Show 7 Replies Reply

gangar Nov 14, 2018

Read more

148 Show 9 Replies Reply

1 2 3 4 5 6 ··· 100