≡ **Articles** ›

3. Longest Substring Without Repeating Characters ⬈ (/problems/longest-substring-without-repeating-characters/)

3. Longest Substring Without Repeating Characters ▾

(/?return=/articles/longest-substring-without-repeating-characters/) (/ratings/107/36/?return=/articles/longest-substring-without-repeating-characters/)

Average Rating: 4.88 (293 votes)

April 10, 2016 | 217.8K views

Given a string, find the length of the **longest substring** without repeating characters.

Examples:

Given `"abcabcbb"` , the answer is `"abc"` , which the length is 3.

Given `"bbbbb"` , the answer is `"b"` , with the length of 1.

Given `"pwwkew"` , the answer is `"wke"` , with the length of 3. Note that the answer must be a **substring**, `"pwke"` is a *subsequence* and not a substring.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

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` .

Java

```java
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        int ans = 0;
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j <= n; j++)
                if (allUnique(s, i, j)) ans = Math.max(ans, j - i);
        return ans;
    }

    public boolean allUnique(String s, int start, int end) {
        Set<Character> set = new HashSet<>();
        for (int i = start; i < end; i++) {
            Character ch = s.charAt(i);
            if (set.contains(ch)) return false;
            set.add(ch);
        }
        return true;
    }
}
```

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 [Accepted]

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.

Java

```java
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        Set<Character> set = new HashSet<>();
        int ans = 0, i = 0, j = 0;
        while (i < n && j < n) {
            // try to extend the range [i, j]
            if (!set.contains(s.charAt(j))){
                set.add(s.charAt(j++));
                ans = Math.max(ans, j - i);
            }
            else {
                set.remove(s.charAt(i++));
            }
        }
        return ans;
    }
}
```

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 [Accepted]

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.

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.

Java (Using HashMap)

```java
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length(), ans = 0;
        Map<Character, Integer> map = new HashMap<>(); // current index of character
        // try to extend the range [i, j]
        for (int j = 0, i = 0; j < n; j++) {
            if (map.containsKey(s.charAt(j))) {
                i = Math.max(map.get(s.charAt(j)), i);
            }
            ans = Math.max(ans, j - i + 1);
            map.put(s.charAt(j), j + 1);
        }
        return ans;
    }
}
```

Java (Assuming ASCII 128)

The previous implements all have no assumption on the charset of the string `s`.

If we know that the charset is rather small, we can replace the `Map` with an integer array as direct access table.

Commonly used tables are:

- `int[26]` for Letters 'a' - 'z' or 'A' - 'Z'

- `int[128]` for ASCII

- `int[256]` for Extended ASCII

```java
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length(), ans = 0;
        int[] index = new int[128]; // current index of character
        // try to extend the range [i, j]
        for (int j = 0, i = 0; j < n; j++) {
            i = Math.max(index[s.charAt(j)], i);
            ans = Math.max(ans, j - i + 1);
            index[s.charAt(j)] = j + 1;
        }
        return ans;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. Index j will iterate n times.

- Space complexity (HashMap) : $O(min(m, n))$. Same as the previous approach.

- Space complexity (Table): $O(m)$. m is the size of the charset.

Rate this article:

(/ratings/107/36/?return=/articles/longest-substring-without-repeating-characters/) (/ratings/107/

Join the conversation

Signed in as **tan7**.

Post a Reply

J   **josephgt** commented yesterday

(https://discuss.leetcode.com/user/josephgt)

My python solution (not great but above average speed):

class Solution:
def lengthOfLongestSubstring(self, s):
"""

:type s: str
:rtype: int
"""

window = {}
window_start = 0
max_len = 0

```python
    for i in range(len(s)):
        if s[i] in window and window[s[i]]>=window_start:
            max_len = max(max_len, i-window_start)
            window_start = window[s[i]]+1
        elif i==len(s)-1:
            max_len = max(max_len, i+1-window_start)
        window[s[i]] = i

    return max_len
```

nate_browne commented 3 days ago

An accepted (but slow) Python approach:

(https://discuss.leetcode.com/user/nate_browne)

```python
class Solution(object):
    def lengthOfLongestSubstring(self, s):
        longest = 0
        current = ''
        for ind in range(len(s)):
            current = ''
            for char in s[ind:]:
                if char not in current:
                    current += char
                else:
                    break
                if len(current) > longest:
                    longest = len(current)
        return longest
```

qingtengmuniao commented 4 days ago

```cpp
int lengthOfLongestSubstring(string s) {
```
(https://discuss.leetcode.com/user/qingtengmuniao)
```cpp
    bool has[256] = {false};
    int left = 0, right = 0;
    int res = 0;
    while (right < s.size()) {
        while (has[s[right]]) {
            has[s[left]] = false;
            ++left;
        }
        has[s[right]] = true;
        res = max(res, right-left+1);
```

```
        ++right;
    }

    return res;
}
```

qingtengmuniao commented 4 days ago

```cpp
int lengthOfLongestSubstring(string s) {
```
(https://discuss.leetcode.com/user/qingtengmuniao)
```cpp
    bool has[256] = {false};
    int left = 0, right = 0;
    int res = 0;
    while (right < s.size()) {
        while (has[s[right]]) {
            has[s[left]] = false;
            ++left;
        }
        has[s[right]] = true;
        res = max(res, right-left+1);
```

```
        ++right;
    }

    return res;
}
```

im.shaoqi.jia commented 5 days ago

I
(https://discuss.leetcode.com/user/im-shaoqi-jia)

@llguru (https://discuss.leetcode.com/uid/478601) Thanks a lot ! Very clearly !

llguru commented 5 days ago

L
(https://discuss.leetcode.com/user/llguru)

@im.shaoqi.jia (https://discuss.leetcode.com/uid/478034) It's not always larger. Consider case s = 'abba' and j = 3 (last step in for).

In this case i = 2 (moved on the previous step), but 'a' maps into 0. I.e. map.get(s.charAt(j)) = 0, but i = 2

im.shaoqi.jia commented 6 days ago

I
(https://discuss.leetcode.com/user/im-shaoqi-jia)

what's the reason for "i = Math.max(map.get(s.charAt(j)), i);". The "map.get(s.charAt(j))" is always larger than "i", isn't it ??? Or I'm wrong about that? Please do me a favor !!! Thanks in advance!

d597180714 commented 6 days ago

D
(https://discuss.leetcode.com/user/d597180714)

```java
class Solution {
public int lengthOfLongestSubstring(String s) {
if(s==null||s.equals("")) return 0;
int i=0,j = 0,max=1;int head = 0;
while(i<s.length()){
char t = s.charAt(i);
j = i-1;
while(j >= head){
char b = s.charAt(j);
if(t!=b){
max = (i - j + 1) > max?(i-j+1):max;
}else{
head = j+1;
break;
}
j--;
}
i++;
}
return max;
}
}
```

S

(https://discuss.leetcode.com/user/sweet-
fan)

sweet.fan commented last week

C 语言：

```
int HasChar(char* s, char a) {
    int pos = 0;
    char *temp =s;
```

```
while(*s) {
    pos++;
    if (a == *s) {
        //printf("'%s' pos %d has char='%c'\n",temp, pos, a);
        return pos;
    }
    s++;
}
return 0;
```

```
}
int lengthOfLongestSubstring(char* s) {
    char *substr = NULL;
    char *begin, *begin_max;
    char *temp = NULL;
    int len_max = 0;
    int len = 0;
    int pos = 0;
    int str_len;
```

```
str_len = strlen(s) + 1;
substr = malloc(str_len);
memset(substr, 0, str_len);
temp = substr;
begin = substr;
//begin_max = begin;

while(*s) {
    pos = HasChar(begin,*s);
    if (!pos) {
        *temp++ = *s++;
        len++;
    } else {
        //printf("len=%d\n", len);
        if (len > len_max) {
            len_max = len;
            //begin_max = begin;
        }
        begin = begin + pos;
        len = len - pos;
    }
}
//printf("%s len=%d\n",begin, len);
if (len > len_max) {
    len_max = len;
    //begin_max = begin;
}
//begin_max[len_max] = 0;
//printf("max substr=%s max_len=%d\n",begin_max, len_max);
free(substr);

return len_max;
```

```
}
```

MKorostoff commented last week

PHP implementation:
(https://discuss.leetcode.com/user/mkorostoff)

```php
<?php
function hasRepeatChars($str) {
        $arr = str_split($str);
        if (count($arr) == count(array_unique($arr))) {
                return false;
        }
        return true;
}

function findLongestSub($str) {
        $strlen = strlen($str);
        $start = 0;
        $end = 0;
        $winner = $str[$start];

        while ($start < $strlen && $end < $strlen) {
                $sub = substr($str, $start, $end);

                if (!hasRepeatChars($sub)) {
                        if (strlen($sub) > strlen($winner)) {
                                $winner = $sub;
                        }
                        $end++;
                }
                else {
                        $start++;
                }
        }
        return $winner;
}
print findLongestSub("ABCCCCDIJLDAANFBOENOFEJFLDSKFMVLKNKDOEKKIFCNWENHFGKLNJ");
```

View original thread (https://discuss.leetcode.com/topic/60)

Load more comments...