Previous Next

3. Longest Substring Without Repeating Characters

April 10, 2016 | 1.1M views

★★★★★
Average Rating: 4.74 (637 votes)

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

Example 1:

```
Input: "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

```
Input: "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: "pwwkew"
Output: 3
Explanation: 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

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 Copy

```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

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 Copy

```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

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 Copy

```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 Copy

```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: ★★★★★

Previous Next

Comments: 618 Sort By

Type comment here... (Markdown is supported)

Preview Post

kxguoniu ★200 October 18, 2018 1:14 PM Report

python3 99.97%

```
class Solution:
    def lengthOfLongestSubstring(self, s):
        dicts = {}
```

Read More

190 Share Reply

SHOW 14 REPLIES

rohan1239 ★147 July 30, 2018 8:34 PM

A bit more intuitive version of solution-3

```
public int lengthOfLongestSubstring(String s) {

        Map<Character, Integer> map= new HashMap<>();
```

Read More

112 Share Reply

SHOW 7 REPLIES

wushi58 ★179 October 17, 2018 1:59 AM Report

Python3 Code, easy to understand

```
def lengthOfLongestSubstring(self, s):
    """
    :type s: str
```

Read More

176 Share Reply

SHOW 13 REPLIES

tryck ★155 April 8, 2019 1:08 AM Report

JavaScript:

```
var lengthOfLongestSubstring = function(s) {
    let max_len = 0;
    let curr = 0;
```

Read More

89 Share Reply

SHOW 7 REPLIES

gangar ★250 November 15, 2018 5:54 AM

Python faster than 99.69%:

```
def lengthOfLongestSubstring(self, s):
    dct = {}
    max_so_far = curr_max = start = 0
```

Read More

138 Share Reply

SHOW 6 REPLIES

wanderwandering ★47 February 15, 2019 3:05 AM Report

Python3
-> 64 ms, faster than 100.00%
-> 12.8 MB, less than 100.00%

```
def lengthOfLongestSubstring(self, s: 'str') -> 'int':
```

Read More

47 Share Reply

SHOW 3 REPLIES

gzrjzcx ★44 January 29, 2019 6:02 AM

c 8ms

```
int lengthOfLongestSubstring(char* s) {
    int count=1;
    int max_count=1;
```

Read More

39 Share Reply

SHOW 2 REPLIES

settiricardo ★74 February 25, 2019 6:15 AM

Simple python solution

```
        d = ""
        f = ""
        for i in range(len(s)):
```

Read More

56 Share Reply

SHOW 6 REPLIES

leo39032506 ★49 January 24, 2019 9:44 AM Report

Concise code of python3

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

Read More

42 Share Reply

SHOW 5 REPLIES

dcnielsen90 ★35 May 4, 2019 6:34 AM Report

faster than 99.87% of Python3 online submissions:

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        ls = ''
```

Read More

32 Share Reply

SHOW 6 REPLIES

‹ 1 2 3 4 5 6 … 61 62 ›