Previous Next

76. Minimum Window Substring

Aug. 27, 2018 | 249.3K views

Average Rating: 4.51 (144 votes)

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

Example:

```
Input: S = "ADOBECODEBANC", T = "ABC"
Output: "BANC"
```

Note:

- If there is no such window in S that covers all characters in T, return the empty string `""`.
- If there is such window, you are guaranteed that there will always be only one unique minimum window in S.

Solution

Approach 1: Sliding Window

Intuition

The question asks us to return the minimum window from the string S which has all the characters of the string T. Let us call a window `desirable` if it has all the characters from T.

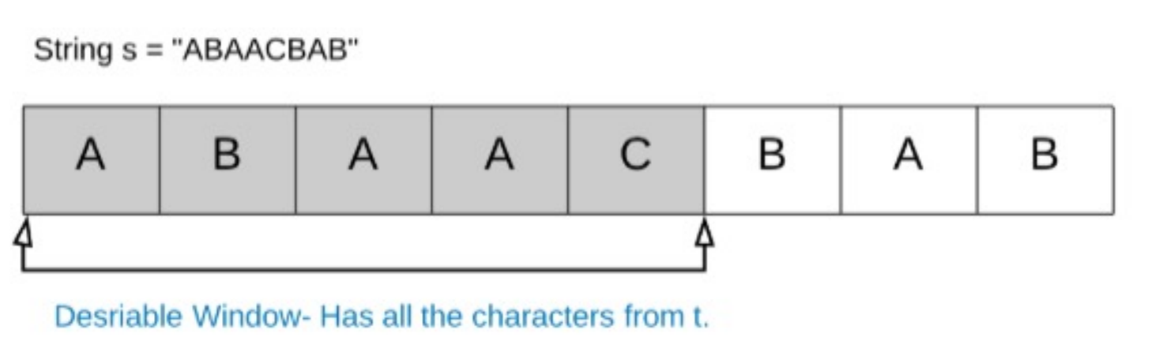
We can use a simple sliding window approach to solve this problem.

In any sliding window based problem we have two pointers. One $right$ pointer whose job is to expand the current window and then we have the $left$ pointer whose job is to contract a given window. At any point in time only one of these pointers move and the other one remains fixed.

The solution is pretty intuitive. We keep expanding the window by moving the right pointer. When the window has all the desired characters, we contract (if possible) and save the smallest window till now.

The answer is the smallest desirable window.

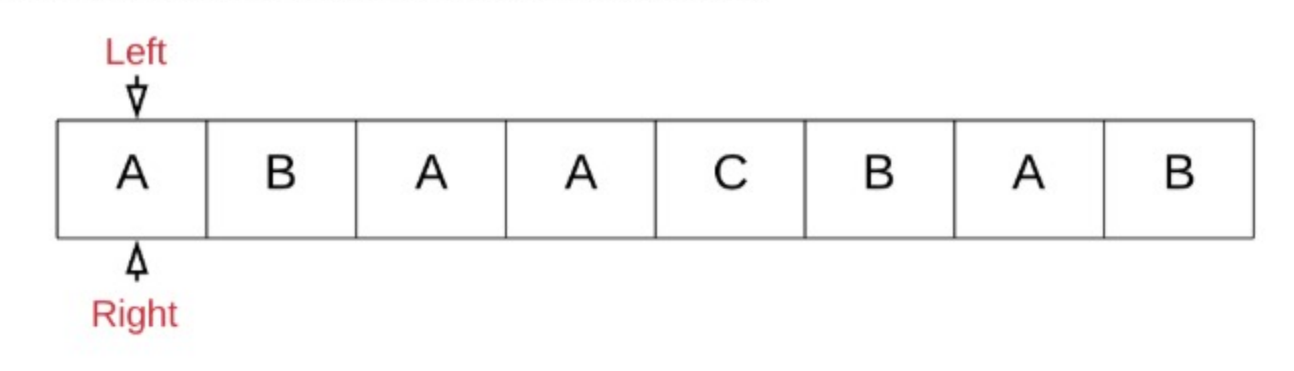
For eg. `S = "ABAACBAB" T = "ABC"`. Then our answer window is `"ACB"` and shown below is one of the possible desirable windows.



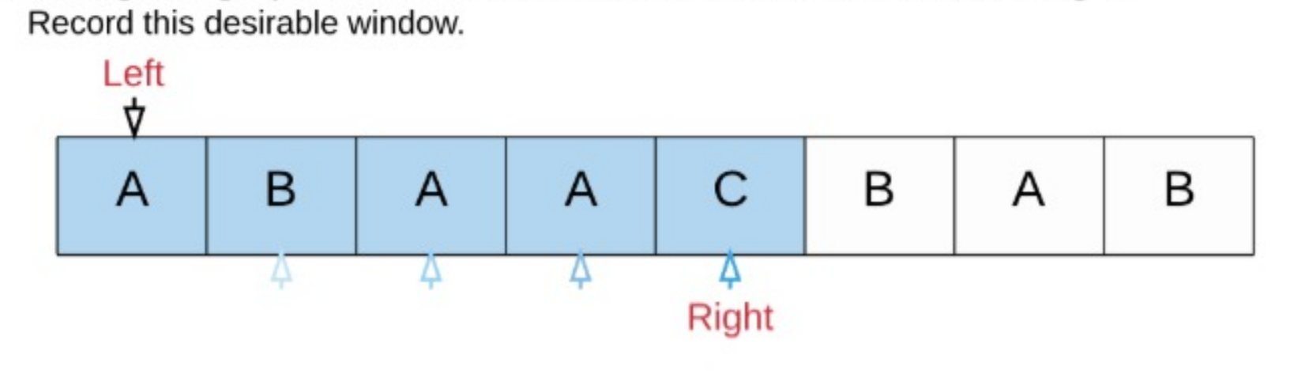
Algorithm

1. We start with two pointers, $left$ and $right$ initially pointing to the first element of the string S.
2. We use the $right$ pointer to expand the window until we get a desirable window i.e. a window that contains all of the characters of T.
3. Once we have a window with all the characters, we can move the left pointer ahead one by one. If the window is still a desirable one we keep on updating the minimum window size.
4. If the window is not desirable any more, we repeat $step\ 2$ onwards.

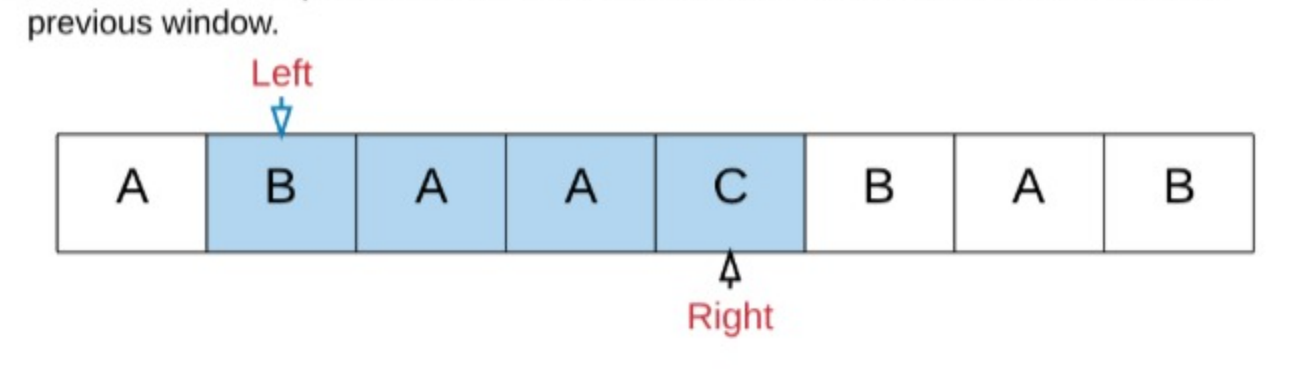
1. Initial State: Left and Right pointers are at index 0.



2. Moving the right pointer until the window has all the elements from string T. Record this desirable window.

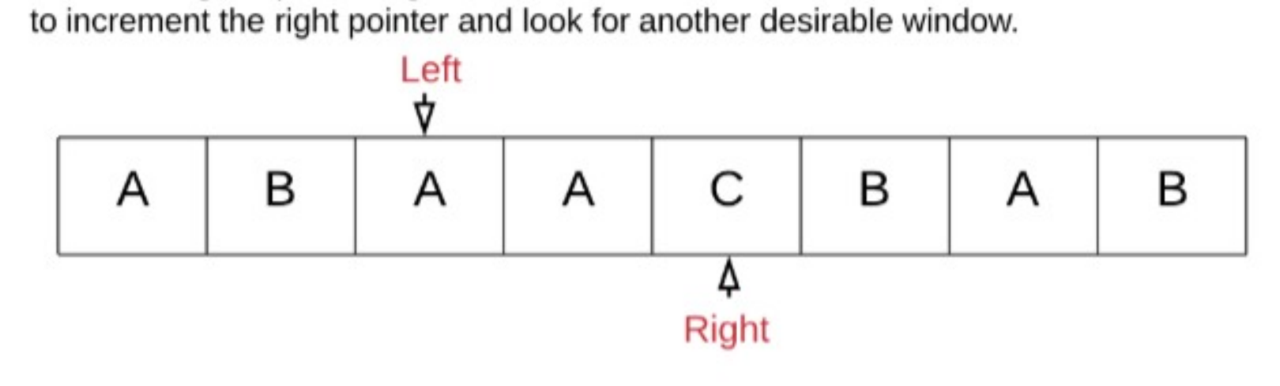


3. Now move the left pointer. Notice the window is still desirable and smaller than the previous window.

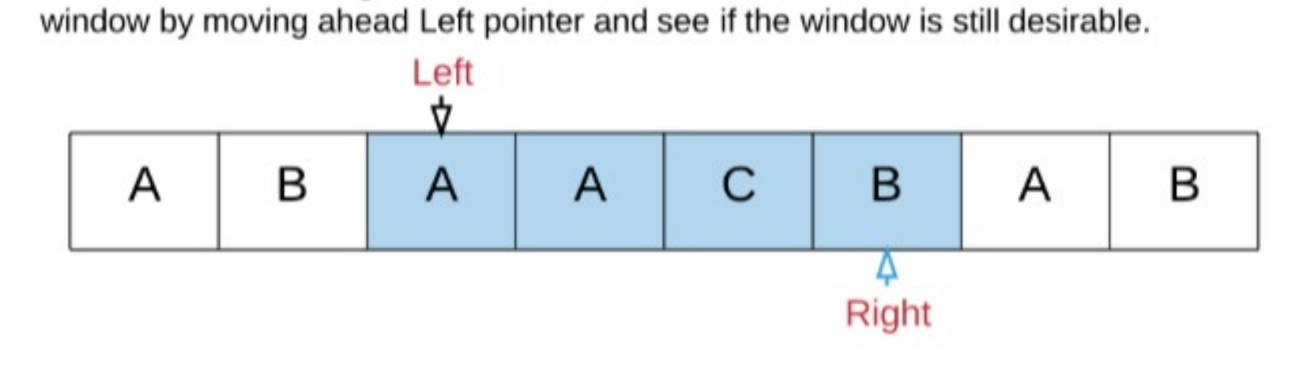


The above steps are repeated until we have looked at all the windows. The smallest window is returned.

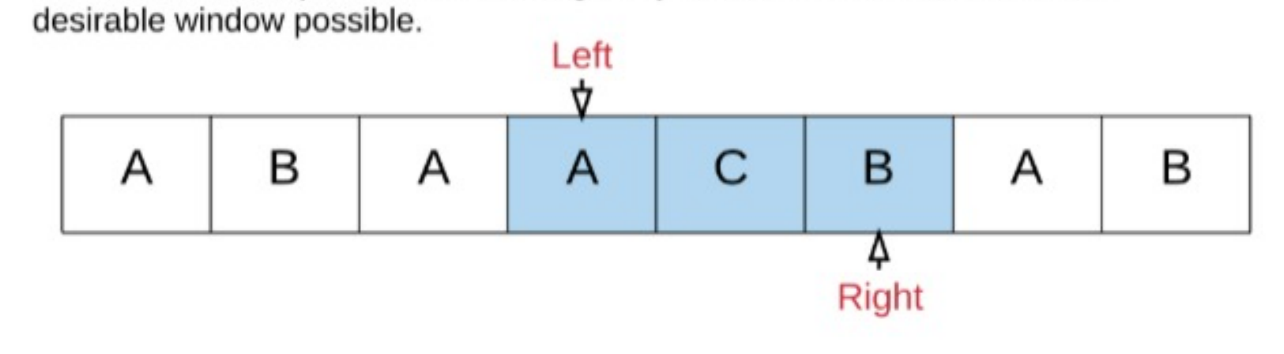
4. After moving left pointer again, the window is no more desirable. Hence we need to increment the right pointer and look for another desirable window.



5. As shown below we again have a desirable window. Now we can contract the window by moving ahead Left pointer and see if the window is still desirable.



6. The window shown below is the smallest desirable window for this example. We continue these steps on the remaining array to see if there is even smaller desirable window possible.



Java Python Copy

```python
def minWindow(self, s, t):
    """
    :type s: str
    :type t: str
    :rtype: str
    """

    if not t or not s:
        return ""

    # Dictionary which keeps a count of all the unique characters in t.
    dict_t = Counter(t)

    # Number of unique characters in t, which need to be present in the desired window.
    required = len(dict_t)

    # left and right pointer
    l, r = 0, 0

    # formed is used to keep track of how many unique characters in t are present in the current window in its desired frequency.
    # e.g. if t is "AABC" then the window must have two A's, one B and one C. Thus formed would be = 3 when all these conditions are met.
    formed = 0

    # Dictionary which keeps a count of all the unique characters in the current window.
    window_counts = {}
```

Complexity Analysis

- Time Complexity: $O(|S| + |T|)$ where |S| and |T| represent the lengths of strings S and T. In the worst case we might end up visiting every element of string S twice, once by left pointer and once by right pointer. $|T|$ represents the length of string T.
- Space Complexity: $O(|S| + |T|)$. $|S|$ when the window size is equal to the entire string S. $|T|$ when T has all unique characters.

Approach 2: Optimized Sliding Window

Intuition

A small improvement to the above approach can reduce the time complexity of the algorithm to $O(2 * |filtered\_S| + |S| + |T|)$, where $filtered\_S$ is the string formed from S by removing all the elements not present in T.

This complexity reduction is evident when $|filtered\_S| <<< |S|$.

This kind of scenario might happen when length of string T is way too small than the length of string S and string S consists of numerous characters which are not present in T.

Algorithm

We create a list called $filtered\_S$ which has all the characters from string S along with their indices in S, but these characters should be present in T.

```
S = "ABCDDDDDDEEAFFBC" T = "ABC"
filtered_S = [(0, 'A'), (1, 'B'), (2, 'C'), (11, 'A'), (14, 'B'), (15, 'C')]
Here (0, 'A') means in string S character A is at index 0.
```

We can now follow our sliding window approach on the smaller string $filtered\_S$.

Java Python Copy

```python
def minWindow(self, s, t):
    """
    :type s: str
    :type t: str
    :rtype: str
    """
    if not t or not s:
        return ""

    dict_t = Counter(t)

    required = len(dict_t)

    # Filter all the characters from s into a new list along with their index.
    # The filtering criteria is that the character should be present in t.
    filtered_s = []
    for i, char in enumerate(s):
        if char in dict_t:
            filtered_s.append((i, char))

    l, r = 0, 0
    formed = 0
    window_counts = {}

    ans = float("inf"), None, None

    # Look for the characters only in the filtered list instead of entire s. This helps to reduce our
```

Complexity Analysis

- Time Complexity : $O(|S| + |T|)$ where |S| and |T| represent the lengths of strings S and T. The complexity is same as the previous approach. But in certain cases where $|filtered\_S|$ <<< $|S|$, the complexity would reduce because the number of iterations would be $2 * |filtered\_S| + |S| + |T|$.
- Space Complexity : $O(|S| + |T|)$.

Rate this article: ★ ★ ★ ★ ★

Previous Next

Comments: 55

Sort By

Type comment here... (Markdown is supported)

Preview Post

v1s1on ★ 499 March 11, 2019 12:34 PM

Short C++ solution:

```
string minWindow(string s, string t) {
        vector<int> hist(128, 0);
        for (char c : t) hist[c]++;
```

Read More

58 Share Reply

SHOW 14 REPLIES

Aria_fighting ★ 46 February 3, 2019 11:00 AM

why we need to use .intValue() here? What does it use for? I removed it from the code and found it could not pass the last test case. Could someone give me an explanation?

39 Share Reply

SHOW 10 REPLIES

logical_paradox ★ 254 October 20, 2018 10:56 AM

So you're saying that `O(|filtered_s| + |S| + |T|) < O(|S| + |T|)`? That makes no sense.

Why would `|S|` still remain in the equation when `|filtered_S| <<< |S|`?

20 Share Reply

SHOW 2 REPLIES

minzhu1987 ★ 19 June 23, 2019 9:00 PM

Why the chars in T are not unique? Why s="aa" and t = "aa" result in the answering being "aa" not "a". That doesn't make any sense!

19 Share Reply

a-b-c ★ 677 October 31, 2018 1:08 PM

it was really helpful, keep writing more solutions for other problems :)

12 Share Reply

dance-henry ★ 22 June 11, 2019 12:35 AM

Is the Time Complexity of first approach truly O(|S|+|T|)? It looped thru each character in string S, then for each character in string S, it will looped thru the current window to try to get the shortest string. Isn't it O(|S|*|T|)?

9 Share Reply

SHOW 2 REPLIES

kagaya056 ★ 6 December 28, 2019 12:09 PM

Should have mentioned that there can be duplicated char in string t.

6 Share Reply

ramineedi ★ 32 January 26, 2019 7:26 AM

run time can further be reduced by checking if ans[0] == len(t) and returning from there, everytime we find a desirable window. This way, if we find a minimum substring towards the left half of s, you don't have to keep continuing the search.

6 Share Reply

enjoymrsun ★ 92 September 30, 2018 3:37 AM

are you sure the 2nd approach time complex is lower?

6 Share Reply

SHOW 1 REPLY

Susieeee ★ 6 September 15, 2018 10:08 PM

Thanks for the solution. But I do not understand why use.intValue()
If I do not use it, it only fails one test case.

3 Share Reply

SHOW 1 REPLY

‹ 1 2 3 4 5 6 ›