

# 76. Minimum Window Substring

Hard Hash Table String Sliding Window

Leetcode Link

## Problem Description

The problem requires us to find the smallest segment (substring) from string `s` that contains all the characters from string `t`, including duplicates. This segment or substring must be minimized in length and must contain each character from `t` at least as many times as it appears in `t`. If we are unable to find such a segment, we should return an empty string. A key point is that the solution is guaranteed to be unique where applicable.

## Intuition

To solve this problem, we employ a sliding window technique along with character counting. The main idea revolves around expanding and shrinking a window on string `s` to include the characters from `t` in the most efficient way possible.

Firstly, we need to know how many of each character from `t` need to be in our window. This is where `need`, our character count for `t`, comes into play.

We also keep a counter for the characters in our current window (`window`) and the number of characters from `t` in the window (`cnt`). Two pointers (`i` and `j`) mark the start and end of the current window. As we traverse `s`, we expand our window by including the current character and check if this character is "necessary" (meaning it's still required to meet the `t` character count criterion). If it is, we increment `cnt`.

Whenever `cnt` reaches the length of `t`, it means our window potentially includes all characters from `t`. At this point, we should attempt to shrink the window from the left by incrementing `j`, all the while making sure the window still contains all characters from `t`. If during this shrinkage we find a window smaller than our previous smallest (`mi`), we update our minimum window size (`mi`) and starting position (`k`).

The process of expanding and shrinking continues until the end of string `s`. If by the end we haven't found any such window, we return an empty string. Otherwise, we return the smallest window from `s` that contains all characters from `t`, starting at `k` with length `mi`.

## Solution Approach

The solution utilizes a sliding window approach, which dynamically changes its size, along with two hash tables or counters to keep track of the characters in the sliding window (`window`) and the characters needed (`need`). The python `Counter` from the collections module has been used for this purpose as it conveniently allows counting the frequency of each character in strings `s` and `t`.

Here's the breakdown of the algorithm:

- First, we initialize the `need` counter with the character frequencies of the string `t` since we want to find a substring of `s` that contains at least these many characters of `t`.
- We also initialize a `window` counter with no characters, an integer `cnt` set to 0 to count the "necessary characters," indices `j` and `k` (`j` for the start of the sliding window and `k` for the start of the minimum window), and a variable `mi` representing the minimum length infinity (`inf` provided by python) of the window.
- We then iterate over the string `s` with the index `i` and character `c`, expanding the `window` by adding the character `c`.
- If the character `c` is "necessary," meaning `need[c] >= window[c]` (it contributes to forming a window that covers `t`), we increment `cnt`.
- A while loop starts whenever `cnt` matches the length of `t`, indicating that our current window has all the characters needed in string `t`. Inside this loop:
  - We check if the current window size (`i - j + 1`) is smaller than the previously recorded minimum (`mi`). If it is, we update `mi` with the new window size and `k` with the new start position of the window (`j`).
  - Shrink the window from the left by moving `j` right. If the character `s[j]` at the left boundary was "necessary," decrement `cnt`. Given we are potentially removing a "necessary character," we update the `window` count for that character.
  - Continue to move `j` to the right as long as `cnt` equals the length of `t` and the condition `need[s[j]] >= window[s[j]]` holds true.
- After the loop ends, if `cnt` does not equal the length of `t`, it means our current window is missing some characters from `t`, and we expand the window by moving `i` to the right, otherwise, we continue to shrink the window.
- Once we've traversed all of `s`, we check if we found a minimum window (by checking if `k >= 0`). If we have, we return the substring `s[k:k + mi]`, otherwise, we return an empty string as per the problem's requirement.

This approach ensures that all characters of `t` are included in the minimum window in `s` as efficiently as possible, by appropriately expanding and shrinking the size of the window.

## Example Walkthrough

Let's take a small example to illustrate the solution approach. Suppose `s = "ADOBECODEBANC"` and `t = "ABC"`.

- We initialize `need` with the frequencies of character of `t`: `need = {'A': 1, 'B': 1, 'C': 1}`.
- Initialize `window` as an empty counter, `cnt` as 0, `j` and `k` both to 0, and `mi` as infinity.
- We start iterating over `s` with our `i` pointer starting from the 0th index.
- When `i = 0, c = A`, we found a necessary character `A`. We update `window` to `['A': 1]` and increment `cnt` to 1.
- At `i = 3, c = B`, `window` becomes `['A': 1, 'B': 1]` and `cnt` now is 2.
- At `i = 5, c = C`, `window` updates to `['A': 1, 'B': 1, 'C': 1]` and `cnt` increases to 3, which is the length of `t`. Now we have a potential window "ADOBEC" from index 0 to 5.
- Since `cnt` equals the length of `t`, we enter the while loop and notice the current window size is smaller than `mi` (infinity), so we update `mi = 6`, and `k = 0`.
- Next, we move `j` right to shrink the window while checking if it still contains `t` fully. The window now can be shrunk because it contains extra characters not needed.
- When `j = 3, s[j] = B`, which is necessary, we decrement `window['B']` and, since `need['B']` is still greater, decrement `cnt`.
- We exit the while loop because the window is not valid (no longer contains all of `t`), `cnt` is no longer equal to the length of `t`.
- We continue with the expansion of the window by moving `i` to the right.
- At `i = 9, cnt` again matches the length of `t`, indicating another potential window "BECODEBA".
- Entering the while loop, we successfully shrink this window to "CODEBA" and then to "ODEBA" and finally to "DEBA" before it becomes invalid as `cnt` drops below the length of `t`.
- We continue this process of expansion and shrinkage until we reach the end of `s`.
- At the end of traversal, we have a `mi` of 4 and a `k` of 9, which corresponds to the window "BANC" which is the smallest window containing all characters of `t`.
- We then return this window `s[9:9+4]` which equals "BANC".

By following this approach using the sliding window technique and character counting, we efficiently find the smallest segment in `s` that contains all the characters of `t`.

## Python Solution

```
1 from collections import Counter
2 from math import inf
3
4 class Solution:
5     def minWindow(self, source: str, target: str) -> str:
6         # Create a counter for the target to keep a record of each character's frequency
7         target_counter = Counter(target)
8         window_counter = Counter() # This will keep a count of characters in the current window
9         valid_char_count = 0 # Number of characters that meet the target criteria
10        left = 0 # Left pointer to shrink the window
11        min_left = -1 # Left boundary index of the minimum window
12        min_size = inf # Initialize min_size to positive infinity
13
14        # Iterate over each character in the source string
15        for right, char in enumerate(source):
16            # Include current character in the window
17            window_counter[char] += 1
18            # If the current character is needed and the window contains enough of this character
19            if target_counter[char] >= window_counter[char]:
20                valid_char_count += 1
21
22            # If the window has all the characters needed
23            while valid_char_count == len(target):
24                # If this window is smaller than the minimum so far, update minimum size and index
25                if right - left + 1 < min_size:
26                    min_size = right - left + 1
27                    min_left = left
28
29                # If the character at the left pointer is less frequent in the window than in the target,
30                # reducing it further would break the window condition
31                if target_counter[source[left]] >= window_counter[source[left]]:
32                    valid_char_count -= 1
33
34                # Shrink the window from the left
35                window_counter[source[left]] -= 1
36                left += 1
37
38        # If no window meets the criteria, return an empty string
39        return '' if min_left < 0 else source[min_left:min_left + min_size]
40
```

## Java Solution

```
1 class Solution {
2     public String minWindow(String source, String target) {
3         // Array to store the frequency of characters needed from the target string
4         int[] charFrequencyInTarget = new int[128];
5         // Array to store the frequency of characters in the current window of the source string
6         int[] charFrequencyInWindow = new int[128];
7
8         int sourceLength = source.length();
9         int targetLength = target.length();
10
11        // Populate the frequency array for the target string
12        for (int i = 0; i < targetLength; ++i) {
13            charFrequencyInTarget[target.charAt(i)]++;
14        }
15
16        int matchCount = 0; // Count of characters that matches from target
17        int windowStart = 0; // The start index of the current window
18        int minWindowStart = -1; // The start index of the minimum window
19        int minLength = Integer.MAX_VALUE; // Length of the smallest window
20
21        // Iterate over the source string
22        for (int windowEnd = 0; windowEnd < sourceLength; ++windowEnd) {
23            // Include the current character in the window
24            charFrequencyInWindow[source.charAt(windowEnd)]++;
25
26            // If the character is needed and we have not more than needed, increase the match count
27            if (charFrequencyInTarget[source.charAt(windowEnd)] >= charFrequencyInWindow[source.charAt(windowEnd)]) {
28                matchCount++;
29            }
30
31            // When we have all characters from the target in our window
32            while (matchCount == targetLength) {
33                int windowLength = windowEnd - windowStart + 1; // Get the current window's length
34
35                // Update minimum length and starting index if a new minimum is found
36                if (windowLength < minLength) {
37                    minLength = windowLength;
38                    minWindowStart = windowStart;
39                }
40
41                // The character at window start is going to be removed since window is moving forward
42                char charAtStart = source.charAt(windowStart);
43
44                // If the character is one that is needed and after removing there are not enough of it, decrease match count
45                if (charFrequencyInTarget[charAtStart] >= charFrequencyInWindow[charAtStart]) {
46                    matchCount--;
47                }
48
49                // Remove the character from the window
50                charFrequencyInWindow[charAtStart]--;
51                windowStart++; // Move the window forward
52            }
53        }
54
55        // Return the minimum window substring or an empty string if no such window exists
56        return minWindowStart < 0 ? "" : source.substring(minWindowStart, minWindowStart + minLength);
57    }
58 }
59
```

## C++ Solution

```
1 class Solution {
2 public:
3     string minWindow(string s, string t) {
4         int need[128] = {0}; // Array to store the frequency of characters required from 't'
5         int window[128] = {0}; // Array to store the frequency of characters in the current window
6         int strLength = s.size(), targetLength = t.size(); // Store the length of the strings 's' and 't'
7
8         // Populate the need array with frequencies of characters in 't'
9         for (char& c : t) {
10             ++need[c];
11         }
12
13        // Initialize variables for the sliding window technique
14        int matches = 0; // Number of characters that match 't' in the current window
15        int start = 0; // Start index of the minimum window
16        int minStart = -1; // Start index of the overall minimum window found
17        int minLength = INT_MAX; // Length of the smallest window found
18
19        // Iterate over the string 's' using 'i' as the end index of the window
20        for (int i = 0; i < strLength; ++i) {
21            // Include the character at the current position into the window
22            ++window[s[i]];
23
24            // If the current character is needed and the window contains enough
25            // instances of it to match 't', then increment the match count
26            if (need[s[i]] >= window[s[i]]) {
27                ++matches;
28            }
29
30            // When all characters from 't' are found in the current window
31            while (matches == targetLength) {
32                // Update minimum window if the current window's length is smaller
33                if (i - start + 1 < minLength) {
34                    minLength = i - start + 1;
35                    minStart = start;
36                }
37
38                // Exclude character at the start position from the window
39                if (need[s[start]] >= window[s[start]]) {
40                    --matches;
41                }
42                --window[s[start++]];
43            }
44
45            // If no window was found, return an empty string. Otherwise, return the minimum window
46            return minStart < 0 ? "" : s.substr(minStart, minLength);
47        }
48    };
49 }
```

## Typescript Solution

```
1 function minWindow(source: string, target: string): string {
2     // Initialize arrays to keep track of character frequency in 'target'
3     // and the current window in 'source'
4     const targetFreq: number[] = new Array(128).fill(0);
5     const windowFreq: number[] = new Array(128).fill(0);
6
7     // Populate target character frequencies
8     for (const char of target) {
9         ++targetFreq[char.charCodeAt(0)];
10    }
11
12    let validCharCount = 0; // Keeps track of how many characters match target in current window
13    let left = 0; // Left pointer for the sliding window
14    let start = -1; // Start index of the min-length window
15    let minLength = Infinity; // Length of the smallest window found
16
17    // Iterate over the 'source' string to find the minimum window
18    for (let right = 0; right < source.length; ++right) {
19        // Include the current character in the window
20        const rightCharIndex = source.charCodeAt(right);
21        ++windowFreq[rightCharIndex];
22
23        // If the character is needed and window has enough of that character, increase valid count
24        if (windowFreq[rightCharIndex] <= targetFreq[rightCharIndex]) {
25            ++validCharCount;
26        }
27
28        // Try and contract the window from the left if it contains all the required characters
29        while (validCharCount === target.length) {
30            if (right - left + 1 < minLength) {
31                minLength = right - left + 1; // Update the smallest window length
32                start = left; // Update the start index of the smallest window
33            }
34
35            // Decrease the left-most character's frequency and move the left pointer
36            const leftCharIndex = source.charCodeAt(left);
37            if (targetFreq[leftCharIndex] >= windowFreq[leftCharIndex]) {
38                --validCharCount; // If the character was contributing to valid count, decrease the valid count
39            }
40            --windowFreq[leftCharIndex];
41            ++left;
42        }
43    }
44
45    // If 'start' is not updated, no valid window is found
46    // Otherwise, return the minimum length window from 'source'
47    return start < 0 ? '' : source.slice(start, start + minLength);
48 }
49
```

## Time and Space Complexity

The time complexity of the provided code is  $O(m + n)$ . This is because the code iterates through all characters of string `s` once which is of length `m`, and for each character, it performs a constant number of operations. Additionally, it iterates through the characters of string `t` once which is of length `n` to fill the `need` Counter object. The while loop and the inner traversals are also constant time because it will at most iterate for every character in string `s`. Altogether, this gives us two linear traversals resulting in  $O(2m + n)$ , which simplifies to  $O(m + n)$ .

The space complexity of the code is  $O(C)$ , where `C` is the fixed size of the character set. In the code, we have two Counter objects: `need` and `window`. Since the Counter objects will only have as many entries as there are unique characters in strings `s` and `t`, and given that the size of the character set is fixed at 128, the space used by these counters does not depend on the size of the input strings themselves but on the size of the character set, making it  $O(C)$ .