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:

1. 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`.
2. 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.
3. We then iterate over the string `s` with the index `i` and character `c`, expanding the `window` by adding the character `c`.
4. If the character `c` is "necessary," meaning `need[c] >= window[c]` (it contributes to forming a window that covers `t`), we increment `cnt`.
5. 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.
6. 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.
7. 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"`.

1. We initialize `need` with the frequencies of character of `t`: `need = {'A': 1, 'B': 1, 'C': 1}`.
2. Initialize `window` as an empty counter, `cnt` as 0, `j` and `k` both to 0, and `mi` as infinity.
3. We start iterating over `s` with our `i` pointer starting from the 0th index.
4. When `i = 0`, `c = A`, we found a necessary character `A`. We update `window` to `{'A': 1}` and increment `cnt` to 1.
5. At `i = 3`, `c = B`, `window` becomes `{'A': 1, 'B': 1}` and `cnt` now is 2.
6. 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.
7. 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`.
8. 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.
9. When `j = 3`, `s[j] = B`, which is necessary, we decrement `window['B']` and, since `need['B']` is still greater, decrement `cnt`.
10. 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`.
11. We continue with the expansion of the window by moving `i` to the right.
12. At `i = 9`, `cnt` again matches the length of `t`, indicating another potential window "BECODEBA".
13. 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`.
14. We continue this process of expansion and shrinkage until we reach the end of `s`.
15. 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`.
16. 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

```python
from collections import Counter
from math import inf

class Solution:
    def minWindow(self, source: str, target: str) -> str:
        # Create a counter for the target to keep a record of each character's frequency
        target_counter = Counter(target)
        window_counter = Counter()  # This will keep a count of characters in the current window
        valid_char_count = 0       # Number of characters that meet the target criteria
        left = 0                   # Left pointer to shrink the window
        min_left = -1              # Left boundary index of the minimum window
        min_size = inf             # Initialize min_size to positive infinity

        # Iterate over each character in the source string
        for right, char in enumerate(source):
            # Include current character in the window
            window_counter[char] += 1
            # If the current character is needed and the window contains enough of this character
            if target_counter[char] >= window_counter[char]:
                valid_char_count += 1

            # If the window has all the characters needed
            while valid_char_count == len(target):
                # If this window is smaller than the minimum so far, update minimum size and index
                if right - left + 1 < min_size:
                    min_size = right - left + 1
                    min_left = left

                # If the character at the left pointer is less frequent in the window than in the target,
                # reducing it further would break the window condition
                if target_counter[source[left]] >= window_counter[source[left]]:
                    valid_char_count -= 1

                # Shrink the window from the left
                window_counter[source[left]] -= 1
                left += 1

        # If no window meets the criteria, return an empty string
        return '' if min_left < 0 else source[min_left:min_left + min_size]

```

Java Solution

```java
class Solution {
    public String minWindow(String source, String target) {
        // Array to store the frequency of characters needed from the target string
        int[] charFrequencyInTarget = new int[128];
        // Array to store the frequency of characters in the current window of the source string
        int[] charFrequencyInWindow = new int[128];

        int sourceLength = source.length();
        int targetLength = target.length();

        // Populate the frequency array for the target string
        for (int i = 0; i < targetLength; ++i) {
            charFrequencyInTarget[target.charAt(i)]++;
        }

        int matchCount = 0; // Count of characters that matches from target
        int windowStart = 0; // The start index of the current window
        int minWindowStart = -1; // The start index of the minimum window
        int minLength = Integer.MAX_VALUE; // Length of the smallest window

        // Iterate over the source string
        for (int windowEnd = 0; windowEnd < sourceLength; ++windowEnd) {
            // Include the current character in the window
            charFrequencyInWindow[source.charAt(windowEnd)]++;

            // If the character is needed and we have not more than needed, increase the match count
            if (charFrequencyInTarget[source.charAt(windowEnd)] >= charFrequencyInWindow[source.charAt(windowEnd)]) {
                matchCount++;
            }

            // When we have all characters from the target in our window
            while (matchCount == targetLength) {
                int windowLength = windowEnd - windowStart + 1; // Get the current window's length

                // Update minimum length and starting index if a new minimum is found
                if (windowLength < minLength) {
                    minLength = windowLength;
                    minWindowStart = windowStart;
                }

                // The character at window start is going to be removed since window is moving forward
                char charAtStart = source.charAt(windowStart);

                // If the character is one that is needed and after removing there are not enough of it, decrease match count
                if (charFrequencyInTarget[charAtStart] >= charFrequencyInWindow[charAtStart]) {
                    matchCount--;
                }

                // Remove the character from the window
                charFrequencyInWindow[charAtStart]--;
                windowStart++; // Move the window forward
            }
        }

        // Return the minimum window substring or an empty string if no such window exists
        return minWindowStart < 0 ? "" : source.substring(minWindowStart, minWindowStart + minLength);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    string minWindow(string s, string t) {
        int need[128] = {0}; // Array to store the frequency of characters required from 't'
        int window[128] = {0}; // Array to store the frequency of characters in the current window
        int strLength = s.size(), targetLength = t.size(); // Store the length of the strings 's' and 't'

        // Populate the need array with frequencies of characters in 't'
        for (char& c : t) {
            ++need[c];
        }

        // Initialize variables for the sliding window technique
        int matches = 0; // Number of characters that match 't' in the current window
        int start = 0; // Start index of the minimum window
        int minStart = -1; // Start index of the overall minimum window found
        int minLength = INT_MAX; // Length of the overall minimum window found

        // Iterate over the string 's' using 'i' as the end index of the window
        for (int i = 0; i < strLength; ++i) {
            // Include the character at the current position into the window
            ++window[s[i]];
            // If the current character is needed and the window contains enough
            // instances of it to match 't', then increment the match count
            if (need[s[i]] >= window[s[i]]) {
                ++matches;
            }
            // When all characters from 't' are found in the current window
            while (matches == targetLength) {
                // Update minimum window if the current window's length is smaller
                if (i - start + 1 < minLength) {
                    minLength = i - start + 1;
                    minStart = start;
                }
                // Exclude character at the start position from the window
                if (need[s[start]] >= window[s[start]]) {
                    --matches;
                }
                --window[s[start++]];
            }
        }

        // If no window was found, return an empty string. Otherwise, return the minimum window
        return minStart < 0 ? "" : s.substr(minStart, minLength);
    }
};

```

Typescript Solution

```typescript
function minWindow(source: string, target: string): string {
    // Initialize arrays to keep track of character frequency in `target`
    // and the current window in `source`
    const targetFreq: number[] = new Array(128).fill(0);
    const windowFreq: number[] = new Array(128).fill(0);

    // Populate target character frequencies
    for (const char of target) {
        ++targetFreq[char.charCodeAt(0)];
    }

    let validCharCount = 0; // Keeps track of how many characters match target in current window
    let left = 0; // Left pointer for the sliding window
    let start = -1; // Start index of the min-length window
    let minLength = Infinity; // Length of the smallest window found

    // Iterate over the `source` string to find the minimum window
    for (let right = 0; right < source.length; ++right) {
        // Include the current character in the window
        const rightCharIndex = source.charCodeAt(right);
        ++windowFreq[rightCharIndex];

        // If the character is needed and window has enough of that character, increase valid count
        if (windowFreq[rightCharIndex] <= targetFreq[rightCharIndex]) {
            ++validCharCount;
        }

        // Try and contract the window from the left if it contains all the required characters
        while (validCharCount === target.length) {
            if (right - left + 1 < minLength) {
                minLength = right - left + 1; // Update the smallest window length
                start = left; // Update the start index of the smallest window
            }

            // Decrease the left-most character's frequency and move the left pointer
            const leftCharIndex = source.charCodeAt(left);
            if (targetFreq[leftCharIndex] >= windowFreq[leftCharIndex]) {
                --validCharCount; // If the character was contributing to valid count, decrease the valid count
            }
            --windowFreq[leftCharIndex];
            ++left;
        }
    }

    // If `start` is not updated, no valid window is found
    // Otherwise, return the minimum length window from `source`
    return start < 0 ? '' : source.slice(start, start + minLength);
}

```

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 operations 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)`.