# Solution: Minimum Window Substring

Let's solve the Minimum Window Substring problem using the Sliding Window pattern.

We'll cover the following



## Statement

We are given two strings, `s` and `t`, find the minimum window substring of `t` in `s`.

The minimum window substring of `t` in `s` is defined as follows:

1. It is the shortest substring of `s` that includes all of the characters present in `t`.
2. The frequency of each character in this substring that belongs to `t` should be equal to or greater than its frequency in `t`.
3. The order of the characters does not matter here.

**Constraints:**

- Strings `s` and `t` consist of uppercase and lowercase English characters.
- $1 \leq$ `s.length`, `t.length` $\leq 10^3$

## Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

### Naive approach

The naive approach would be to find all possible substrings of `s` and then identify the shortest substring that contains all characters of `t` with corresponding frequencies equal to or greater than those in `t`.

To find all possible substrings, we will iterate over `s` one character at a time. For each character, we will form all possible substrings starting from that character.

We will keep track of the frequencies of the characters in the current substring. If the frequencies of the characters of `t` in the substring are equal to or greater than their overall frequencies in `t`, save the substring given that the length of this substring is less than the one already saved. After traversing `s`, return the minimum window substring.

The time complexity of this approach will be $O(n^2)$, where $n$ is the length of `s`. The space complexity of this approach will be $O(n)$, the space used in memory to track the frequencies of the characters of the current substring.

### Optimized approach using sliding window

To eliminate the cost of iterating over each substring separately, we use the sliding window pattern. We are searching for the shortest substring of `s` that contains all the characters of `t`. Once we have found the initial window in `s` that contains all the characters of `t`, we can slide the window in order the find the shortest one. Let's see how this approach can efficiently solve this problem.

The first step is to verify whether or not `t` is a valid string. If it isn't, we return an empty string as our output. Then, we initialize two pointers to apply the sliding window technique to our solution. Before we discuss how they're being used in our solution, we need to take a look at the other components at work.

There are two separate hash maps that we initialize, `reqCount` and `window`. We populate the `reqCount` hash map with the characters in `t` and their corresponding frequencies. This is done by traversing each character of `t`. If it doesn't already exist in the hash map, we add it with count 1, but if it does, we increment its count. The `window` hash map is initialized to contain the same characters present in the `reqCount` hash map with the corresponding counts set to 0. The `window` hash map will be used to keep track of the frequency of the characters of `t` in the current_ window.

We also set up two more variables called `current` and `required`, which tell us whether we need to increase or decrease the size of our sliding window. The `current` variable will initially hold the value 0 but will be incremented by 1 when we find a character whose frequency in the `window` hash map matches its frequency in the `reqCount` hash map. The `required` variable will store the size of the `reqCount` hash map. The moment these two become equal, we have found all the characters that we were looking for in the current window. So, we can start trying to reduce our window size to find the shortest possible substring.

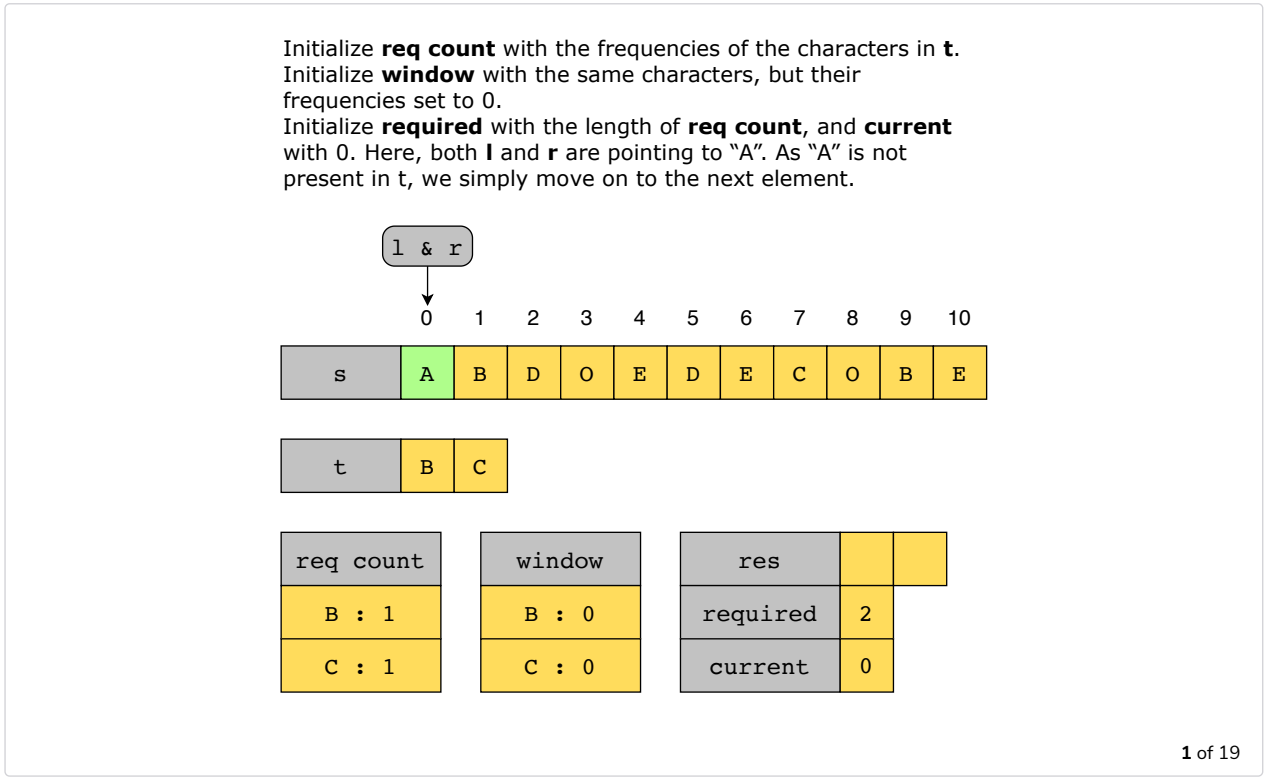
Next, let's look at how we create this window and adjust it. We initialize a variable called `left`, which acts as the left pointer, but on the other side, we don't need to initialize a right pointer explicitly. It can simply be the iterator of our loop, `right`, which traverses the array from left to right. In each iteration of this loop, we perform the following steps:

- If the new character is present in the `window` hash map, we increment its frequency by 1.
- If the new character occurs in `t`, we check if its frequency in the `window` hash map is equal to its frequency in the `reqCount` hash map. Here, we are actually checking if the current character has appeared the same number of times in the current window as it appears in `t`. If so, we increment `current` by 1.
- Finally, if `current` and `required` become equal this means that we have found a substring that meets our requirements. So, we start reducing the size of the current window to find a shorter substring that still meets these requirements. As long as `current` and `required` remain equal, we move the `left` pointer forward, decrementing the frequency of the character being dropped out of the window. By doing this, we remove all the unnecessary characters present at the start of the substring. We keep comparing the size of the current window with the length of the shortest substring we have found so far. If the size of the current window is less than the length of the minimum substring, we update the minimum substring.
- Once `current` and `required` become unequal, it means we have checked all the possible substrings that meet our requirement. Now, we slide the right edge of the window one step forward and continue iterating over `s`.

When `s` has been traversed, we return the minimum window substring.

The slides below illustrate how we would like the algorithm to run:

> **Note:** $l$ and $r$ represent the `left` and `right` pointers respectively.




Java

```java
import java.util.*;

class MinimumWindowSubstring {
    public static String minWindow(String s, String t) {
        // empty string scenario
        if (t.equals("")) {
            return "";
        }

        // creating the two hash maps
        Map<Character, Integer> reqCount = new HashMap<>();
        Map<Character, Integer> window = new HashMap<>();

        // populating reqCount hash map
        for (int i = 0; i < t.length(); i++) {
            char c = t.charAt(i);
            reqCount.put(c, 1 + reqCount.getOrDefault(c, 0));
        }

        // populating window hash map
        for (int i = 0; i < t.length(); i++) {
            char c = t.charAt(i);
            window.put(c, 0);
        }

        // setting up the conditional variables
        int current = 0;
        int required = reqCount.size();
```

> **Note**: The `reqCount.put(c, 1 + reqCount.getOrDefault(c, 0))` statement in our code ensures that a new character may be added to the hash map without any error. Here, `rCount[c] = 1 + rCount[c]` could result in an error if `c` isn't present already. It's the same case with the `window.put(c, 1 + window.getOrDefault(c, 0))` statement.

### Solution summary

To recap, the solution can be divided into the following parts:

- We validate the inputs. If `t` is an empty string, we return an empty string.
- Next, we initialize two hash maps: `reqCount`, to save the frequency of characters in `t`, and `window`, to keep track of the frequency of characters of `t` in the current window. We also initialize a variable, `required`, to hold the number of unique characters in `t`. Lastly, we initialize `current` which keeps track of the characters that occur in `t` whose frequency in the current window is equal to or greater than their corresponding frequency in `t`.
- Then, we iterate over `s` and in each iteration we perform the following steps:
    - If the current character occurs in `t`, we update its frequency in the `window` hash map.
    - If `current` is equal to `required`, we decrease the size of the window from the start. As long as `current` and `required` are equal, we decrease the window size one character at a time, while also updating the minimum window substring. Once `current` falls below `required`, we slide the right edge of the window forward and move on to the next iteration.
- Finally, when `s` has been traversed completely, we return the minimum window substring.

### Time complexity

In the average-case scenario, each hash map operation will cost $O(1)$. So, the time complexity for the solution shown above is $O(n + m)$, where $n$ and $m$ are the lengths of the strings `s` and `t`, respectively. This is because we're accessing each element of `s` just once. For all practical purposes, this is the time complexity of this solution.

In the worst case, each hash map operation will cost $O(m)$. Hence, the overall time complexity would be $O(m + (n \times m))$.

### Space complexity

The space complexity for this solution is $O(m)$, the space used by the hash maps to store the frequencies of characters of `t`.