LeetCode 76 — Minimum Window Substring

1. Problem Title & Link

- **Title:** LeetCode 76 — Minimum Window Substring
- **Link:** https://leetcode.com/problems/minimum-window-substring/

2. Problem Statement (Short Summary)

Given two strings s and t, return the **smallest substring of s** that contains **all the characters in t** (including their frequency).

If no such window exists, return "".

Example:

s = "ADOBECODEBANC", t = "ABC"

Output: "BANC"

This is a **classic sliding window + hashmap frequency** problem.

3. Examples (Input → Output)

Example 1

Input: s = "ADOBECODEBANC", t = "ABC"

Output: "BANC"

Example 2

Input: s = "a", t = "a"

Output: "a"

Example 3

Input: s = "a", t = "aa"

Output: ""

4. Constraints

- 1 <= s.length, t.length <= 10^5
- s and t consist of English letters
- Frequency counts matter:
  - If t = "AABC", window needs 2 'A', 1 'B', 1 'C'

5. Core Concept (Pattern / Topic)

Sliding Window + Hashmap Frequency Matching

This problem uses:

- Frequency map for target string t
- Expand right to include characters

- Contract left to shrink window
- Maintain **valid window** condition (matched count)

This is THE standard template for:

- LC 76
- LC 567
- LC 438
- LC 3

6. Thought Process (Step-by-Step Explanation)

Goal:

Find smallest substring of s that contains all chars of t.

Steps:

1. Build frequency map t_count for characters in t
2. Maintain:
    - window_count (current window character counts)
    - have = number of characters meeting required freq
    - need = number of unique chars in t
3. Expand window with right pointer
4. When window becomes valid (have == need):
    - Try to shrink from left to find smaller window
5. Keep updating minimum window length
6. Continue till right reaches end

This two-pointer expansion + shrink strategy ensures O(n) time.

7. Visual / Intuition Diagram

Example:

s = "ADOBECODEBANC"

t = "ABC"

Sliding window moves like:

A D O B E C O D E B A N C

↑

left, right

Expand right until window is valid:

ADOBEC → contains A, B, C

Then shrink:

DOBEC → still valid

OBEC → still valid

BEC → still valid

EC → NOT valid → stop and expand again

Best window found = "BANC".

8. Pseudocode

```
t_count = freq map of t
window_count = {}
have = 0
need = unique chars in t

result = (inf, "")  # length, substring
left = 0

for right in range(len(s)):
    add s[right] to window_count
    if counts match for that char:
        have += 1

    while have == need:
        update result if smaller window found
        remove s[left] from window_count
        if removing breaks requirement:
            have -= 1
        left += 1

return result.substring
```

9. Code Implementation

✅ **Python**

```
from collections import Counter

class Solution:
    def minWindow(self, s: str, t: str) -> str:
        if not t or not s:
            return ""

        t_count = Counter(t)
        window = {}
```

```
        have = 0
        need = len(t_count)

        res = [float('inf'), 0, 0]  # length, left, right
        left = 0

        for right in range(len(s)):
            c = s[right]
            window[c] = window.get(c, 0) + 1

            if c in t_count and window[c] == t_count[c]:
                have += 1

            while have == need:
                # update result
                if (right - left + 1) < res[0]:
                    res = [right - left + 1, left, right]

                # pop from the left
                window[s[left]] -= 1
                if s[left] in t_count and window[s[left]] < t_count[s[left]]:
                    have -= 1
                left += 1

        _, L, R = res
        return s[L:R+1] if res[0] != float('inf') else ""
```

✅ **Java**

```
class Solution {
    public String minWindow(String s, String t) {
        if (t.length() == 0 || s.length() == 0) return "";

        int[] tCount = new int[128];
        int[] wCount = new int[128];

        int need = 0;
        for (char c : t.toCharArray()) {
            if (tCount[c] == 0) need++;
            tCount[c]++;
        }

        int have = 0;
```

```
        int minLen = Integer.MAX_VALUE, start = 0;
        int left = 0;

        for (int right = 0; right < s.length(); right++) {
            char c = s.charAt(right);
            wCount[c]++;

            if (wCount[c] == tCount[c]) have++;

            while (have == need) {
                if (right - left + 1 < minLen) {
                    minLen = right - left + 1;
                    start = left;
                }

                char lc = s.charAt(left);
                wCount[lc]--;
                if (wCount[lc] < tCount[lc]) have--;
                left++;
            }
        }

        return minLen == Integer.MAX_VALUE ? "" : s.substring(start, start +
minLen);
    }
}
```

10. Time & Space Complexity

Metric	Complexity
Time	O(n) — each char visited at most twice
Space	O(1) — fixed ASCII map / O(k) for hashmap

11. Common Mistakes / Edge Cases

- Forgetting to check character frequency (not just presence)
- Shrinking the window before it is valid
- Not updating result correctly
- Not adjusting have when removing left character
- Handling uppercase/lowercase issues

Edge cases:

- t longer than s → return ""
- s or t empty

12. Detailed Dry Run (Step-by-Step Table)

Input:

s = "ADOBECODEBANC"

t = "ABC"

t_count = {A:1, B:1, C:1}

have = 0, need = 3

r	s[r]	window	have	left	window valid?	result
0	A	A:1	1	0	no	INF
1	D	A:1 D:1	1	0	no	INF
2	O	...	1	0	no	INF
3	B	... B:1	2	0	no	INF
4	E	...	2	0	no	INF
5	C	... C:1	3	0	YES	window = "ADOBEC"
Shrink:						

- Move left until invalid → smallest becomes "BEC"
  ↓
  Record result: "BEC" (len=3)

Continue scanning…

Eventually smaller window found: "BANC"

Final answer: "BANC"

13. Common Use Cases

- Search highlighting (Google Docs)
- DNA pattern matching
- Scan logs for required event patterns
- Substring extraction problems

14. Common Traps

- Sliding too early before window is valid
- Forgetting frequency matching
- Window validity condition mismanaged
- Not shrinking properly

- Using substring operations inefficiently in Java

15. Builds To (Related Problems)

- **LC 3** — Longest substring without repeating chars
- **LC 567** — Permutation in String
- **LC 438** — Find all anagrams in string
- **LC 159** — Longest substring with at most 2 chars

16. Alternate Approaches + Comparison

Approach	Time	Space	Comment
Brute force	$O(n^2)$	O(1)	Too slow
Sliding window	O(n)	O(1)	Best solution
Trie/DP	—	—	Not suitable

17. Why This Solution Works (Short Intuition)

Expanding the right pointer ensures we include required characters.
Shrinking the left pointer ensures we find the **smallest valid window**.
Tracking frequencies ensures correctness.

18. Variations / Follow-Up Questions

- What if characters are case-insensitive?
- What if input is Unicode?
- What if we want the **count** of all valid windows?
- How to return **all minimal windows**?