

```
// User function Template for Java

class Solution {

    int search(String pat, String txt) {
        // code here

        int result = 0;

        int charCnt = pat.length(); // Character count of the pattern.
        Map<Character, Integer> charFreq = new HashMap<>(); // Character frequency of
        the pattern.

        // Calculating and storing the count of each character in the given pattern.
        for (int index = 0; index < pat.length(); index++) {
            char letter = pat.charAt(index);
            charFreq.put(letter, charFreq.getOrDefault(letter, 0) + 1);
        }

        for (int start = 0, end = 0; end < txt.length(); end++) {

            char charAtEnd = txt.charAt(end);

            if (charFreq.containsKey(charAtEnd)) {
                charFreq.put(charAtEnd, charFreq.get(charAtEnd) - 1);
                charCnt--;
            }
        }
    }
}
```

```
if ((end - start + 1) == pat.length()) {  
  
    if (charCnt == 0) {  
        result++;  
    }  
  
    char charAtStart = txt.charAt(start);  
  
    if (charFreq.containsKey(charAtStart)) {  
        charFreq.put(charAtStart, charFreq.get(charAtStart) + 1);  
        charCnt++;  
    }  
  
    start++;  
}  
  
return result;  
}  
}  
  
/* Above is the solution for https://www.geeksforgeeks.org/problems/count-occurrences-of-anagrams5839/1 */  
  
/* Please check and fix bugs for the above code by keeping the code as same as possible  
and variable names the same. */
```

I've reviewed the code for counting occurrences of anagrams of a pattern in a text string. The approach is correct using a sliding window technique with a frequency map, but there's a logical issue in the implementation. Here's the corrected version:

```
java

// User function Template for Java

class Solution {

    int search(String pat, String txt) {
        // code here

        int result = 0;

        int charCnt = pat.length(); // Character count of the pattern.
        Map<Character, Integer> charFreq = new HashMap<>(); // Character frequency of the pattern.

        // Calculating and storing the count of each character in the given pattern.
        for (int index = 0; index < pat.length(); index++) {
            char letter = pat.charAt(index);
            charFreq.put(letter, charFreq.getOrDefault(letter, 0) + 1);
        }

        int start = 0;
        for (int end = 0; end < txt.length(); end++) {

            char charAtEnd = txt.charAt(end);

            if (charFreq.containsKey(charAtEnd)) {
                charFreq.put(charAtEnd, charFreq.get(charAtEnd) - 1);
            }
        }
    }
}
```

```
        if (charFreq.get(charAtEnd) >= 0) {
            charCnt--;
        }
    }

    if ((end - start + 1) == pat.length()) {

        if (charCnt == 0) {
            result++;
        }

        char charAtStart = txt.charAt(start);

        if (charFreq.containsKey(charAtStart)) {
            charFreq.put(charAtStart, charFreq.get(charAtStart) + 1);
            if (charFreq.get(charAtStart) > 0) {
                charCnt++;
            }
        }
    }

    start++;
}

}

return result;
}
}
```

Key Fixes Made:

1. **Fixed the character count logic:** In the original code, when we encounter a character from the pattern, we always decrement `charCnt`. However, this is incorrect because if a character appears more times in the window than in the pattern, its frequency can become negative, but we shouldn't count those extra occurrences as valid matches.
2. **Added proper bounds checking:** Now we only decrement `charCnt` when the frequency remains non-negative after decrementing, and only increment `charCnt` when the frequency becomes positive after incrementing.
3. **Maintained the same variable names and structure:** The solution keeps all variable names and the overall approach identical to your original code.

The algorithm now correctly:

- Uses a sliding window of size equal to the pattern length
- Maintains a frequency map of characters in the pattern
- Tracks how many characters still need to be matched (`charCnt`)
- Counts valid anagrams when all characters are matched (`charCnt == 0`)
- Properly updates frequencies and counts when the window slides

This should now correctly solve the "Count Occurrences of Anagrams" problem on GeeksforGeeks.

Just give an example of the fixed logic

Example:

Let's take `txt = "cbaebabacd"` and `pat = "abc"`.

We want to count how many times an anagram of "abc" appears in "cbaebabacd".

Step-by-step with the fixed logic:

1. Initialize:

- o `charFreq = {'a':1, 'b':1, 'c':1}`
- o `charCnt = 3` (number of characters left to match)
- o `result = 0`
- o `start = 0`

2. Window [0,0]: 'c'

- o 'c' is in `charFreq` → decrement freq: `{'a':1, 'b':1, 'c':0}`
- o Since freq became 0 ($>=0$), decrement `charCnt` to 2.
- o Window size = 1 (not equal to pat length 3) → no check.

3. Window [0,1]: 'c','b'

- o 'b' is in `charFreq` → decrement freq: `{'a':1, 'b':0, 'c':0}`
- o Since freq became 0 ($>=0$), decrement `charCnt` to 1.
- o Window size = 2 → no check.

4. Window [0,2]: 'c','b','a'

- o 'a' is in `charFreq` → decrement freq: `{'a':0, 'b':0, 'c':0}`
- o Since freq became 0 ($>=0$), decrement `charCnt` to 0.
- o Window size = 3 → check: `charCnt==0` → found anagram! `result=1`.
- o Now remove `txt[0]='c'` :
 - 'c' is in `charFreq` → increment freq: `{'a':0, 'b':0, 'c':1}`

- Since freq became 1 (>0), increment charCnt to 1.

- start becomes 1.

5. Window [1,3]: 'b','a','e'

- 'e' not in charFreq → ignore.
 - Window size = 3 → check: charCnt=1 (not 0) → no anagram.
 - Remove txt[1]='b':
 - 'b' is in charFreq → increment freq: {'a':0, 'b':1, 'c':1}
 - Since freq became 1 (>0), increment charCnt to 2.

- start becomes 2.

6. Window [2,4]: 'a','e','b'

- 'b' is in charFreq → decrement freq: {'a':0, 'b':0, 'c':1}
 - Since freq became 0 (≥ 0), decrement charCnt to 1.
 - Window size = 3 → check: charCnt=1 → no anagram.
 - Remove txt[2]='a':
 - 'a' is in charFreq → increment freq: {'a':1, 'b':0, 'c':1}
 - Since freq became 1 (>0), increment charCnt to 2.

- start becomes 3.

7. Window [3,5]: 'e','b','a'

- 'a' is in charFreq → decrement freq: {'a':0, 'b':0, 'c':1}
 - Since freq became 0 (≥ 0), decrement charCnt to 1.
 - Window size = 3 → check: charCnt=1 → no anagram.

- Remove `txt[3]='e'` (not in map) → ignore.

- `start` becomes 4.

8. Window [4,6]: 'b','a','c'

- 'c' is in `charFreq` → decrement freq: `{'a':0, 'b':0, 'c':0}`

- Since freq became 0 ($>=0$), decrement `charCnt` to 0.

- Window size = 3 → check: `charCnt==0` → found anagram! `result=2`.

- Remove `txt[4]='b'`:

- 'b' is in `charFreq` → increment freq: `{'a':0, 'b':1, 'c':0}`

- Since freq became 1 (>0), increment `charCnt` to 1.

- `start` becomes 5.

...and so on.

Final Result: 2 anagrams found at indices [0,2] and [4,6] ("cba" and "bac").

The fixed logic ensures that:

- We only count when exactly all characters are matched (no extras).
- We properly handle characters that appear more than needed in the window.

Please provide some similar and relevant leetcode questions.

Only provide problem number, problem name and URL only in a table.

Here are some similar and relevant LeetCode problems:

Problem Number	Problem Name	URL
567	Permutation in String	https://leetcode.com/problems/permutation-in-string/
438	Find All Anagrams in a String	https://leetcode.com/problems/find-anagrams-in-a-string/
76	Minimum Window Substring	https://leetcode.com/problems/minimum-window-substring/
3	Longest Substring Without Repeating Characters	https://leetcode.com/problems/longest-substring-without-repeating-characters/
30	Substring with Concatenation of All Words	https://leetcode.com/problems/substring-with-concatenation-of-all-words/
159	Longest Substring with At Most Two Distinct Characters	https://leetcode.com/problems/longest-substring-with-at-most-two-distinct-characters/
340	Longest Substring with At Most K Distinct Characters	https://leetcode.com/problems/longest-substring-with-at-most-k-distinct-characters/
424	Longest Repeating Character Replacement	https://leetcode.com/problems/longest-repeating-character-replacement/
992	Subarrays with K Different Integers	https://leetcode.com/problems/subarrays-with-k-different-integers/

Problem Number	Problem Name	URL
----------------	--------------	-----