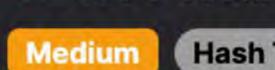
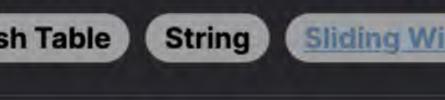
## 1297. Maximum Number of Occurrences of a Substring



Hash Table

**Problem Description** 



Sliding Window Leetcode Link

The problem is to find the substring that appears the most times in a given string s. The catch is that these substrings have to comply with two rules:

2. The length of the substring must be at least minSize and at most maxSize.

1. The quantity of unique characters in each substring must be less than or equal to maxLetters.

A key thing to note is that though substrings can have lengths anywhere from minSize to maxSize, we are interested in finding the maximum number of occurrences, and typically substrings with smaller lengths will have higher chances of repeated occurrence. The solution should return the maximum frequency of any such substring found in s that follows the above rules.

Intuition

that we're also looking for maximum occurrence leads to a strategy where smaller substrings are more likely to be repeated; hence, we prioritize evaluating substrings of minSize. The intuition for the solution is to slide a window of size minSize across the string s and use a set to track the unique characters and

When dissecting the problem, we focus on the constraints - unique characters within the substring and the substring's size. The fact

a counter (hashmap) to count occurrences of each substring that fits the rules. For each window, we check the following: 1. If the set size is larger than maxLetters, we ignore this substring as it doesn't meet the first rule.

- 2. If the set size is within the limits, we add the substring to the counter and check if it's the most frequent one we've seen so far.
- This way we iterate over the string once (0(n) time complexity, where n is the length of the string), and only consider substrings of

length minSize because regardless of maxSize, any repeated instances of a longer substring will always contain a repeated minSize

substring within it. Thus, minSize is the key to finding the maximum occurrence. **Solution Approach** 

## The solution uses a sliding window algorithm and two data structures—a set and a counter from Python's collections module. The set is used to keep track of the unique characters within the current minSize window of the string, while the Counter is a hashmap to

keep track of the frequency of each unique substring that fits the criteria. Here are the steps in the implementation:

3. Iterate over the string with a variable i ranging from 0 to len(s) - minSize + 1. This is your sliding window that will only

consider substrings of size minSize. 4. At each iteration, create a substring t which is a slice of s starting from index i to i + minSize.

2. Initialize ans as an integer to keep track of the maximum frequency found.

1. Initialize cnt as a Counter object to keep track of the frequency of each qualifying substring.

- 5. For the substring t, create a set ss to determine the number of unique characters it contains.
- 6. A crucial optimization here is that if len(ss) is greater than the maxLetters, the substring does not satisfy the required constraints, and no further action is taken for that window.
- count of how many times t has been seen. 8. Update ans which keeps track of the highest frequency seen so far by comparing it with the frequency count of t in cnt. 9. After the loop finishes, ans will hold the highest frequency of occurrence of any valid substring and is returned.

7. If len(ss) is less than or equal to maxLetters, add/subtract one to the cnt[t] for that substring sequence, which increases the

- The algorithm sweeps through the string only once, making it efficient with time complexity of O(n) (for the loop) times O(k) (for the
- set creation, where k is at most minSize), and it only considers substrings of length minSize due to the provided insight that longer substrings will naturally contain these smaller, frequently occurring substrings if they are to be frequent themselves. The space complexity mainly depends on the number of unique substrings of length minSize that can be formed from s, which in worst cases is

O(n). Overall, the algorithm efficiently finds the most frequent substring of size within the given bounds that also contains no more than maxLetters unique characters.

Let's consider a simple example with the following parameters:

## maxLetters: 2 minSize: 2

Example Walkthrough

maxSize: 3

1. We initialize cnt as an empty Counter object and ans as 0.

s: "ababab"

- Following the solution approach:
- 2. We then iterate over s using a window size of minSize. In this example, minSize is 2, so we will be sliding through the string two characters at a time.

which are {'a', 'b'}.

most frequent qualifying substring.

max\_frequency = 0

substring\_counter = Counter()

# Extract a substring of length minSize

substring = s[i: i + minSize]

unique\_chars = set(substring)

4. Since the size of ss is 2 and does not exceed maxLetters, which is also 2, we proceed to increment the count of this substring in

count in cnt for the respective substring.

cnt: cnt["ab"] += 1. 5. Now we slide the window by one and repeat. Our next substring is "ba" (from index 1 to 2), and ss will again be {'b', 'a'}. We

3. In the first iteration, i is 0, and our substring t is "ab" (from index 0 to 1). We create a set ss containing unique characters in "ab",

- increment cnt ["ba"]. 6. This process is repeated for the entire string: we then check "ab" again, and "ba" again. Each time, we are incrementing the
- 7. After we've processed each substring, we observe that "ab" and "ba" each have a count of 3 in cnt. 8. ans is then updated to be the maximum of the values in cnt, which in this case is 3.
- The final answer, held by ans, is 3, indicating that the most frequent substrings of length minSize are "ab" and "ba," each appearing 3 times in the string s. Our example is now complete, demonstrating how the sliding window and counter work together to find the

**Python Solution** from collections import Counter

### 12 13 # Loop through the string to check all possible substrings of length minSize for i in range(len(s) - minSize + 1): 14 15

9

10

11

16

17

18

19

20

class Solution:

```
21
22
               # If the number of unique characters is less than or equal to maxLetters
23
                if len(unique_chars) <= maxLetters:</pre>
24
                    # Increment the frequency count for this substring
25
                    substring_counter[substring] += 1
26
27
                    # Update max_frequency with the maximum value between the current max
                    # and the frequency of the current substring
28
29
                    max_frequency = max(max_frequency, substring_counter[substring])
30
31
           # Return the maximum frequency
           return max_frequency
32
33
Java Solution
   class Solution {
       public int maxFreq(String s, int maxLetters, int minSize, int maxSize) {
           // Initialize the answer variable to store the maximum frequency.
           int maxFrequency = 0;
           // Create a HashMap to store the frequency of substrings.
           Map<String, Integer> substringFrequency = new HashMap<>();
           // Loop through the string to find valid substrings of length minSize.
           for (int start = 0; start <= s.length() - minSize; ++start) {</pre>
 8
                // Extract substring of size `minSize` from the original string.
9
                String substring = s.substring(start, start + minSize);
10
               // HashSet to track unique characters in the current substring.
11
                Set<Character> uniqueChars = new HashSet<>();
12
13
               // Calculate the number of unique characters in the substring.
                for (int j = 0; j < minSize; ++j) {</pre>
14
                    // Add unique characters in the substring to the set.
15
                    uniqueChars.add(substring.charAt(j));
16
17
               // Check if the number of unique characters meets the requirement.
               if (uniqueChars.size() <= maxLetters) {</pre>
19
20
                    // Increment the count of this valid substring in the map.
```

substringFrequency.put(substring, substringFrequency.getOrDefault(substring, 0) + 1);

// Update the maximum frequency with the highest count found so far.

// Return the maximum frequency of any valid substring.

maxFrequency = Math.max(maxFrequency, substringFrequency.get(substring));

def maxFreq(self, s: str, maxLetters: int, minSize: int, maxSize: int) -> int:

# Create a Counter object to keep track of the frequencies of the substrings

# Initialize the variable to store the maximum frequency found

# Create a set of unique characters in the substring

# This function finds the maximum frequency of any substring that meets the criteria

# C++ Solution

return maxFrequency;

21

22

23

24

25

26

27

28

30

29 }

```
1 #include <string>
 2 #include <unordered_map>
 3 #include <unordered_set>
   #include <algorithm>
   class Solution {
   public:
        int maxFreq(string s, int maxLetters, int minSize, int maxSize) {
           // Initialize the answer to be returned
 9
            int maxFrequency = 0;
10
11
12
           // Make use of an unordered_map to count the occurrences of substrings
13
            unordered_map<string, int> substringCounts;
14
15
           // Loop through the string, only up to the point where minSize substrings can still be formed
            for (int i = 0; i <= s.size() - minSize; ++i) {</pre>
16
17
18
                // Extract the substring of length minSize starting at index i
                string t = s.substr(i, minSize);
19
20
21
                // Create a set of unique characters (ss) from the substring
                unordered_set<char> uniqueChars(t.begin(), t.end());
24
                // Check if the number of unique characters does not exceed maxLetters
25
                if (uniqueChars.size() <= maxLetters) {</pre>
26
27
                    // If the conditions are met, increment the count for this substring
                    // and update the maxFrequency with the maximum value
28
                    maxFrequency = std::max(maxFrequency, ++substringCounts[t]);
29
30
31
32
33
           // Return the maximum frequency found for any substring that meets the criteria
            return maxFrequency;
34
35
36 };
37
```

**Time and Space Complexity** 

is bounded by minSize, this does not affect the order of space complexity.

which there are n - minSize + 1.

```
Typescript Solution
 1 // Import necessary elements from 'collections' for Map and Set
 2 import { Map, Set } from 'collections';
   // Function to calculate the maximum frequency of any substring meeting certain criteria
   function maxFreq(s: string, maxLetters: number, minSize: number, maxSize: number): number {
       // Initialize the answer to be returned
       let maxFrequency: number = 0;
 9
       // Make use of a Map to count the occurrences of substrings
       let substringCounts: Map<string, number> = new Map<string, number>();
10
11
12
       // Loop through the string, only up to the point where minSize substrings can still be formed
       for (let i = 0; i <= s.length - minSize; ++i) {
13
14
15
           // Extract the substring of length minSize starting at index i
           let t: string = s.substring(i, i + minSize);
16
17
           // Create a set of unique characters from the substring
18
19
           let uniqueChars: Set<string> = new Set<string>(t.split(''));
20
           // Check if the number of unique characters does not exceed maxLetters
21
22
           if (uniqueChars.size <= maxLetters) {</pre>
23
24
               // If the conditions are met, increment the count for this substring
               let count: number = substringCounts.get(t) || 0;
25
26
               substringCounts.set(t, count + 1);
27
28
               // Update the maxFrequency with the maximum value
29
               maxFrequency = Math.max(maxFrequency, count + 1);
30
31
32
33
       // Return the maximum frequency found for any substring that meets the criteria
       return maxFrequency;
34
35 }
36
```

The time complexity of the code is O(n \* minSize) where n is the length of the string s. This is because the code iterates over the string in slices of length minSize which takes O(minSize) time for each slice, and this is done for all starting points in the string, of

The space complexity of the code is O(n) in the worst case. This is due to the fact that the Counter could potentially store every unique substring of length minSize in the worst scenario where all possible substrings are unique. Since the length of each substring