1170. Compare Strings by Frequency of the Smallest Character

Medium Array Hash Table String Binary Search Sorting

Leetcode Link

#### The problem presents a function f(s) that calculates the frequency of the lexicographically smallest character within a non-empty

**Problem Description** 

string s. For instance, if s = "dcce", then f(s) equals 2 because the smallest character ('c') appears twice in the string.

Given two arrays, one containing strings words and the other containing query strings queries, the task is to determine how many

words in words have a higher f value than the f value of each query string in queries. Specifically, for each query queries [i], you

need to count the number of strings in words where f(queries[i]) is less than f(W) for each string W in words.

The expected output is an array answer, where answer[i] corresponds to the count for queries[i]. Essentially, the output tells us for each query, the count of words in words with a smaller lexicographically smallest character frequency.

Intuition

The intuition behind the solution involves first understanding what the function f does in terms of string processing. It counts the

#### Approaching the solution, we consider these steps:

1. Precompute the f values for all strings in words since we will need to compare these with each query's f value. This precomputation helps to avoid recalculating these values for each query, which would be inefficient.

2. Sort the precomputed f values of the words array. With a sorted list of f values, we can efficiently determine where a particular

O Few seeds account adjusted its fiveling. Then to

to the right of this point are larger.

with a larger f value than the query.

number of times the smallest lexicographical character appears in a string.

solution, where n is the length of the longest array between queries and words.

queries] creates the result list by performing this calculation for each query.

3. For each query, calculate its f value. Then, to find out how many f values in words are larger than that of the query, we use binary search to find the insertion point in the sorted list of f values of words. This gives us the position where the f value of the query would fit if it were to be inserted into the list, effectively giving us the count of f values that are greater because all values

value would be placed: all elements larger than this value would come after it in the sorted array.

- 4. The bisect\_right function from Python's bisect module is used to perform the binary search. It returns the insertion point (index) in the sorted nums list, which allows us to subtract this index from the total number of words to get the count of words
- 5. The final result for each query is calculated by taking the total number of words n and subtracting the insertion index. This count is added to the result list, which is returned at the end.
  The process leverages binary search for efficiency, ensuring that an overall time complexity better than O(n^2) is achieved for the
- string from the string module, and the bisect\_right function from the bisect module. Here's how these are used step-by-step:

  1. **Define the function f(s)**: The function f(s) is where the calculation of the smallest character's frequency is implemented. It uses the Counter class to count the occurrences of each character in string s. Counter(s) returns a dictionary-like object where

The implementation of the solution to this problem uses Python's Counter class from the collections module, the ascii\_lowercase

characters are keys, and their counts are values.

2. Find the smallest character's frequency: Once we have the counts, we iterate over ascii\_lowercase (a string containing all lowercase letters in alphabetical order) to find the smallest character present in s. As soon as we find a character c that is in s

(i.e., cnt[c] is not zero), we return its count cnt[c]. Using ascii\_lowercase ensures we check characters in lexicographical

### 3. **Precompute f for words**: We create a list called nums that holds the f value for each word in words. This is done with a list comprehension [f(w) for w in words], calculating f(w) for each w in words.

**Example Walkthrough** 

4 - 2 = 2 words with a larger f value.

larger f value.

higher f value.

**Python Solution** 

class Solution:

11

17

18

19

20

21

22

23

24

25

26

27

28

29

8

9

10

11

12

13

14

15

55

C++ Solution

1 #include <vector>

2 #include <string>

using std::vector;

using std::upper\_bound;

6 using std::string;

7 using std::sort;

10 class Solution {

public:

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35 36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

#include <algorithm> // Needed for sorting and upper\_bound

auto f = [](const string &s) {

count[c - 'a']++;

if (frequency) {

for (int frequency : count) {

for (int i = 0; i < numWords; ++i) {

for (const string &query : queries) {

wordFrequencies[i] = f(words[i]);

sort(wordFrequencies, wordFrequencies + numWords);

// Iterate the queries to find out the respective counts

// Prepare the answer vector to hold the result

// Add the count to the answer vector

answer.push\_back(greaterFrequencyCount);

return frequency;

for (char c : s) {

return 0;

vector<int> answer;

**}**;

// This method will take a vector of query strings and a vector of word strings

// Count how many times each character appears in the string

int numWords = words.size(); // Cache the number of words for efficiency

int queryFrequency = f(query); // Get frequency of current query

// Calculate the frequency of each word's smallest character

// and will return a vector with the count of words that have a higher frequency

vector<int> numSmallerByFrequency(vector<string>& queries, vector<string>& words)

// of the smallest character than the frequency of the smallest character in each query string.

// Find and return count of the first non-zero frequency (smallest character)

// Sort the frequencies in non-decreasing order to perform efficient lookups later

// Find how many words have a greater frequency than the current query frequency

// The number of words with a greater frequency is the total number of words minus

// the number of words with a frequency less than or equal to the query frequency

int greaterFrequencyCount = numWords - (upper\_bound(wordFrequencies, wordFrequencies + numWords, queryFrequency) - word

// 'f' is a lambda function that calculates the frequency of the smallest character in a string

int count[26] = {0}; // Initialize an array to store the count of each character from 'a' to 'z'

int wordFrequencies[numWords]; // This array will hold the frequencies of the smallest char in each word

Java Solution

1 from collections import Counter

results = []

for query in queries:

from typing import List

from bisect import bisect\_right

def calculate\_frequency(s: str) -> int:

# List to hold the result for each query

int wordsLength = words.length;

// Sort the frequencies

Arrays.sort(frequencies);

int[] frequencies = new int[wordsLength];

for (int i = 0; i < wordsLength; ++i) {</pre>

// Calculate frequency of smallest char for each word

frequencies[i] = smallestCharFrequency(words[i]);

char\_count = Counter(s)

# Counts the occurrence of each character

method for this purpose.

order.

Solution Approach

5. **Find each query's count**: For each query q in queries, we calculate f(q) and then find the index at which f(q) would be inserted while maintaining the order in nums. This is where bisect\_right comes in. It returns the insertion point to the right of existing values of f(q). This index tells us how many words in words have a larger f value than f(q).

6. Prepare the result: Knowing the index from bisect\_right, the number of words with a larger f value is found by subtracting this

index from the total number of words, which is len(words). The comprehension [n - bisect\_right(nums, f(q)) for q in

4. Sort nums: nums is sorted so that binary search can be used to efficiently find positions within it. We use Python's built-in sorted

7. Return the result: The resultant list is returned, which contains the count for each query indicating how many words in words have a higher f value than that of the query.
In conclusion, the solution makes use of a combination of string processing, precomputation, binary search, and efficient sorting to

arrive at the result. It leverages the properties of sorted arrays and binary search (bisect\_right) to perform queries in logarithmic

Let's go through an example to illustrate the solution approach described above. Suppose we have a list of words, words = ["a",

"aa", "aaa", "aaaa"] and a list of queries, queries = ["a", "aa", "aaa"]. Our goal is to count how many words have a higher

frequency of the lexicographically smallest character for each query. Let's walk through each step of the approach.

2. Precompute f values for words: We precompute the f values for the words. This gives us nums = [1, 2, 3, 4].

3. Sort the nums list: We sort the precomputed f values of words. Our nums list is already sorted: nums = [1, 2, 3, 4].

time, after an initial sort cost, allowing for an efficient and scalable approach to solve the given problem.

1. **Define and calculate f for each word**: We first define the function f(s) to calculate the frequency of the lexicographically smallest character in a given string s. For the words list, since all words consist of the character 'a', the f values are simply the lengths of the strings. Thus, f("a") = 1, f("aa") = 2, and so on.

4. **Find each query's count**: Now, we need to find out for each query how many f values in words are larger than the query's f value. For queries[i] = "a", f("a") is 1. Using binary search, we determine the position where 1 would fit in the sorted nums list (which is index 1, right after the first occurrence of 1). Since there are a total of 4 words, and 1 would be inserted at index 1, there are 4 - 1 = 3 words with a larger f value.

Repeating this for queries [1] = "aa" with f("aa") = 2, binary search tells us that 2 would be inserted at index 2 in nums, leaving

Lastly, for queries [2] = "aaa", f("aaa") = 3, binary search gives us an insertion index of 3, so there are 4 - 3 = 1 word with a

6. Return the result: The result for our query is [3, 2, 1], indicating for each query in queries, how many words in words have a

The example confirms our approach, showing how we can effectively calculate the counts required using precomputation, sorting, and binary search.

def numSmallerByFrequency(self, queries: List[str], words: List[str]) -> List[int]:

# Helper method to calculate the frequency of the smallest character

# Calculate the frequency for each word and sort the results

query\_frequency = calculate\_frequency(query)

results.append(num\_of\_higher\_frequency\_words)

return results # Returns the final list of results

# Append the result to the results list

word\_frequencies = sorted(calculate\_frequency(word) for word in words)

# Calculate the number of words with higher frequency than each query

# Use binary search to find the number of words with higher frequency

5. Prepare the result: The counts we determined in the previous step give us our final result array: [3, 2, 1].

for c in 'abcdefghijklmnopqrstuvwxyz':

if c in char\_count:

return char\_count[c]

return 0 # In case the string is empty, though the problem assumes non-empty strings

16

# Iterates over lowercase ascii characters to find the smallest character and return its count

1 class Solution {
2
3 // Function to count how many strings in 'words' have a frequency smaller than the frequency of each string in 'queries'
4 public int[] numSmallerByFrequency(String[] queries, String[] words) {

num\_of\_higher\_frequency\_words = len(word\_frequencies) - bisect\_right(word\_frequencies, query\_frequency)

16 // For each query, count how many words have a smaller frequency 17 int queriesLength = queries.length; int[] answer = new int[queriesLength]; 18 19 for (int i = 0; i < queriesLength; ++i) {</pre> 20 int queryFrequency = smallestCharFrequency(queries[i]); 21 // Binary search to find the number of words with a larger frequency 22 int left = 0; 23 int right = wordsLength; 24 while (left < right) {</pre> 25 int mid = (left + right) >> 1; if (frequencies[mid] > queryFrequency) { 26 27 right = mid; 28 } else { 29 left = mid + 1;30 31 32 // The result for this query is the number of frequencies greater than the query frequency 33 answer[i] = wordsLength - left; 34 35 36 return answer; 37 38 39 // Helper function to calculate the frequency of the smallest character in a string 40 private int smallestCharFrequency(String s) { int[] charCounts = new int[26]; // There are 26 lowercase English letters 41 42 // Count the occurrences of each character in the string for (int i = 0; i < s.length(); ++i) {</pre> 43 44 charCounts[s.charAt(i) - 'a']++; 45 46 // Find the smallest non-zero frequency 47 for (int count : charCounts) { **if** (count > 0) { 48 49 return count; 50 51 52 return 0; // Return 0 if the string was empty (though this should not happen given the problem constraints) 53 54

# 60 61 // Return the filled answer vector 62 return answer; 63 } 64 }; 65

Typescript Solution

```
// Function to count the smallest character frequency in a given string.
   function frequencyOfSmallestChar(s: string): number {
       const frequencyCounter = new Array(26).fill(0); // Initialize an array for alphabets.
       // Count frequency of each character in the string.
       for (const char of s) {
           frequencyCounter[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
6
       // Find and return the frequency of the smallest character.
       return frequencyCounter.find(count => count > 0);
9
10 }
11
   // The main function that applies the frequencyOfSmallestChar function on queries compared to words.
   function numSmallerByFrequency(queries: string[], words: string[]): number[] {
       // Map the 'words' array into an array of smallest character frequencies
14
       // and sort it in ascending order.
15
       const sortedWordFrequencies = words.map(frequencyOfSmallestChar).sort((a, b) => a - b);
16
       const answerArray: number[] = []; // Initialize an array to store the results.
17
18
       // Process each query string.
19
       for (const query of queries) {
20
21
           // Find frequency of the smallest character in the current query string.
22
           const queryFrequency = frequencyOfSmallestChar(query);
23
           let leftIndex = 0,
24
               rightIndex = sortedWordFrequencies.length;
25
26
           // Perform a binary search to find the count of elements in 'sortedWordFrequencies'
27
           // that are greater than 'queryFrequency'.
28
           while (leftIndex < rightIndex) {</pre>
29
               const midIndex = Math.floor((leftIndex + rightIndex) / 2);
30
               if (sortedWordFrequencies[midIndex] > queryFrequency) {
31
                   rightIndex = midIndex;
32
               } else {
33
                   leftIndex = midIndex + 1;
34
35
36
37
           // Push the count (length of the array minus the final leftIndex) to the result array.
38
           answerArray.push(sortedWordFrequencies.length - leftIndex);
39
       return answerArray; // Return the final result array.
40
41 }
42
Time and Space Complexity
Time Complexity
The time complexity of the provided code involves several components:
 1. Calculating the frequency of the smallest character in each word using the f function. This operation involves creating a Counter
```

## 2. Sorting the frequencies of words. The sorted function has a time complexity of O(n log n), where n is the length of the words list.

for each word, which has a time complexity of O(m), where m is the average string length. This is done for all words, so for n words, this part would have a time complexity of O(n \* m).

The space complexity is the additional space used by the algorithm:

are used one at a time, this is not multiplied by n.

(bisect\_right) to find the position in the sorted list of word frequencies. The frequency calculation is O(m), and binary search in a sorted list of size n is  $O(\log n)$ . Therefore, for q queries, this part has a time complexity of  $O(q * (m + \log n))$ .

3. For each query in queries, the function calculates the frequency of the smallest character and uses binary search

- Putting it all together, the total time complexity is  $0(n * m) + 0(n \log n) + 0(q * (m + \log n))$ .

  Space Complexity
  - Storing the frequencies of all words, which is O(n) space.
     The sorted frequency list, which again takes O(n) space.

3. The counter created for each word, which in the worst case takes 0(m) space; however, since the counters are not stored and

4. The final result list which will contain q elements, resulting in O(q) space.

So, the total space complexity is 0(n + q), since n and q space requirements dominate the 0(m) space needed for the counter for each word or query string.