<https://leetcode.com/problems/total-appeal-of-a-string/description/>

**The appeal of a string is the number of distinct characters found in the string.**

For example, the appeal of "abbca" is 3 because it has 3 distinct characters: 'a', 'b', and 'c'.

Given a string s, **return the total appeal of all of its substrings.**

**A substring is a contiguous sequence of characters within a string.**

**Example 1:**

**Input:** s = "abbca"

**Output:** 28

**Explanation:** The following are the substrings of "abbca":

- Substrings of length 1: "a", "b", "b", "c", "a" have an appeal of 1, 1, 1, 1, and 1 respectively. The sum is 5.

- Substrings of length 2: "ab", "bb", "bc", "ca" have an appeal of 2, 1, 2, and 2 respectively. The sum is 7.

- Substrings of length 3: "abb", "bbc", "bca" have an appeal of 2, 2, and 3 respectively. The sum is 7.

- Substrings of length 4: "abbc", "bbca" have an appeal of 3 and 3 respectively. The sum is 6.

- Substrings of length 5: "abbca" has an appeal of 3. The sum is 3.The total sum is 5 + 7 + 7 + 6 + 3 = 28.

**Example 2:**

**Input:** s = "code"

**Output:** 20

**Explanation:** The following are the substrings of "code":

- Substrings of length 1: "c", "o", "d", "e" have an appeal of 1, 1, 1, and 1 respectively. The sum is 4.

- Substrings of length 2: "co", "od", "de" have an appeal of 2, 2, and 2 respectively. The sum is 6.

- Substrings of length 3: "cod", "ode" have an appeal of 3 and 3 respectively. The sum is 6.

- Substrings of length 4: "code" has an appeal of 4. The sum is 4.The total sum is 4 + 6 + 6 + 4 = 20.

**Constraints:**

1 <= s.length <= 10^5

s consists of lowercase English letters.

**What's the difference between Leetcode 828 and Leetcode 2262 ?**

**Refer to chatGPT**

|  |  |  |
| --- | --- | --- |
| Aspect | **Leetcode 828** | **Leetcode 2262** |

|  |  |  |
| --- | --- | --- |
| **Focus** | **Individual** **characters** **being** **unique** in substrings. | **All distinct characters** in substrings. |

|  |  |  |
| --- | --- | --- |
| **Formula** | (i−prev)×(next−i) | (i−last)×(length−i) |

|  |  |  |
| --- | --- | --- |
| **Result** | Total unique character count across all substrings. | Total "appeal" (distinct character count) of substrings. |

|  |  |  |
| --- | --- | --- |
| **Complexity** | O(n) with efficient precomputation of prev and next  . | O(n) with tracking last for each character. |

**Attempt 1: 2024-06-18**

**Solution 1: DFS (10 min, TLE 53/76)**

**Similar to** [L828.Count Unique Characters of All Substrings of a Given String (Ref.L2262)](note://WEBf394faa89bf4c60aa6f2c29c102ef307) **DFS TLE solution, just need to change implementation of countUniqueChars method**

class Solution {

    long count = 0;

    public long appealSum(String s) {

        helper(s, 0, "");

        return count;

    }

    private void helper(String str, int start, String current) {

        if (start == str.length()) {

            return;

        }

        for (int end = start + 1; end <= str.length(); end++) {

            current = str.substring(start, end);

            count += (long)countUniqueChars(current);

        }

        helper(str, start + 1, current);

    }

    public int countUniqueChars(String str) {

        Map<Character, Integer> charCountMap = new HashMap<>();

        for (char c : str.toCharArray()) {

            charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1);

        }

        return charCountMap.size();

    }

}

**Solution 2: DP (180 min, only able to copy the answer)**

**Refer to**

<https://leetcode.com/problems/total-appeal-of-a-string/solutions/1996390/java-c-python-easy-and-concise-with-explanation/>

**Intuition**

Quite similar to [828. Count Unique Characters of All Substrings of a Given String](https://leetcode.com/problems/count-unique-characters-of-all-substrings-of-a-given-string/discuss/128952/).

You can take 828 as an another chanllendge to practice more.

**Explanation**

assume we have string xxxaxxxxb..., with s[i] = a and s[j] = b.

s[i] is the last character a before that b.

We want to count, how many substring ending at s[j] contains character a.

They are xxxaxxxxb, xxaxxxxb, xaxxxxb, axxxxb ....,

i + 1 substring ending with character a at s[i]

so we do res += i + 1.

We repeatly do this for every s[i] and every one of 26 characters.

**Complexity**

Time O(n)

Space O(26)

public long appealSum(String s) {

int last[] = new int[26];

long res = 0;

for (int i = 0; i < s.length(); ++i) {

last[s.charAt(i) - 'a'] = i + 1;

for (int j : last)

res += j;

}

return res;

}

**Then improved from O(26n) to O(n)**

**Don't do a O(26) for loop to accumulate the last, instead count the total value of last and update it in O(1)**

public long appealSum(String s) {

int last[] = new int[26];

long res = 0, total = 0;

for (int i = 0; i < s.length(); ++i) {

total += i + 1 - last[s.charAt(i) - 'a'];

last[s.charAt(i) - 'a'] = i + 1;

res += total;

}

return res;

}

**or we can do as below way:**

**Count contirbution for each character**

This solution is more like what we do for 828. Count Unique Characters of All Substrings of a Given String.

You can take 828 as an another chanllendge to practice more.

In a substring, multiple same character only get one point.

We can consider that the first occurrence get the point.

Now for each character, we count its countribution for all substring.

For each character s[i],

the substring must start before s[i] to contain s[i]

and need to end after the last occurrence of s[i],

otherwise the last occurrence of character s[i] will get the socre.

In total, there are i - last[s[i]] possible start position,

and n - i possible end position,

so s[i] can contribute (i - last[s[i]]) \* (n - i) points.

From this formula, we can also the difference between problem 2262 and 828.

Complexity: Time O(n), Space O(26)

public long appealSum(String s) {

int last[] = new int[26];

Arrays.fill(last, -1);

long res = 0, n = s.length();

for (int i = 0; i < s.length(); ++i) {

res += (i - last[s.charAt(i) - 'a']) \* (n - i);

last[s.charAt(i) - 'a'] = i;

}

return res;

}

**More readable DP solution**

**Refer to**

<https://algo.monster/liteproblems/2262>

**Problem Description**

The goal is to calculate the total appeal of all possible substrings of a given string. The appeal of a string is determined by the number of distinct characters it contains. For instance, the string "abbca" contains three distinct characters, 'a', 'b', and 'c', therefore its appeal is 3.

To find the total appeal, we must consider each substring of the input string. A substring is defined as any contiguous sequence of characters within the string. For example, the string "abc" has the following substrings: "a", "b", "c", "ab", "bc", "abc". Each substring has its own appeal, and the total appeal is the sum of the appeals of all these substrings.

The challenge lies in efficiently computing this sum without having to explicitly check each substring, as there could be a large number of them.

**Intuition**

The presented solution takes a dynamic approach to calculate the total appeal. It optimizes the process by not generating each substring, but instead, keeping track of how each new character added to the end of the substrings affects their overall appeal.

**The crux of the approach relies on two key observations:**

**When a new character is added that hasn't been seen before in any of the previous substrings, it increases the appeal of all the substrings ending at the previous character by one.**

**If the new character has been seen before, only the substrings that started after the last occurrence of this character have their appeal increased by one.**

To keep track of these effects, an array pos is used to record the last position where each character occurred in the string (-1 if the character hasn't occurred yet).

As we iterate through the string character by character, we adjust the running total appeal t by considering the difference between the current index i and the last occurrence pos[c]. This efficiently calculates the incremental appeal contributed by the current character to all substrings ending with it.

By summing up these incremental contributions t as we go along, we obtain the total appeal of all substrings by the end of the traversal. This method avoids the overhead of explicit substring enumeration and appeal calculation, leading to a much faster solution.

**Solution Approach**

The solution implements an efficient algorithm to calculate the total appeal of all substrings of a given string. It uses a simple linear-time algorithm that leverages a concept similar to dynamic programming, along with an integer array to keep track of the last occurrence of characters.

The key steps in the algorithm include:

**Initialize an array pos with length 26 (since there are 26 lowercase letters in the English alphabet) and fill it with -1. This array is used to store the last position where each character was seen in the string.**

Initialize two integers: ans to accumulate the total appeal and t to keep track of the current sum of appeals as the algorithm iterates through the string.

Iterate over each character c in the string s, using an index i to keep track of the position.

Convert the character c into an array index (0-25) by subtracting the ASCII code of 'a' from the ASCII code of c.

**Update t by adding the difference between i and the last occurrence of c (which is pos[c]). This captures the increase in appeal for all substrings ending at the current character, as explained in the intuition section.**

Add the updated t to ans, incrementally building up the total appeal.

**Finally, update the last occurrence of c in pos to the current index i.**

Continue the iteration until the end of the string and return ans as the total appeal.

The code uses the fact that updating the total appeal with each new character and calculating the incremental appeal based on the last occurrence is sufficient to count the appeal for all possible substrings.

This approach results in O(n) time complexity, where n is the length of the input string. It avoids the need for nested loops, which would result in a higher time complexity. Additionally, the space complexity is O(1) since the auxiliary space used (the pos array) does not grow with the size of the input string.

**Example Walkthrough**

Let's walk through an example to illustrate the solution approach using the string "abaca".

**Initialization:** We start by initializing the array pos with size 26, to represent each letter in the English alphabet, and fill it with -1, indicating that none of the characters have been seen yet. We also initialize ans (accumulated total appeal) and t (current sum of appeals) to 0.

pos is initially [-1, -1, -1, ..., -1], ans = 0, and t = 0.

**Iteration:**

**Character 'a' at index 0:** The index for 'a' is 0 - 'a' = 0. Since 'a' was not seen before (pos[0] == -1), t is updated by 0 - (-1) = 1. Then ans is updated by adding t to it (now ans = 1). The position of 'a' is updated in pos as pos[0] = 0.

**Character 'b' at index 1:** Similar to 'a', 'b' has not been seen before. The index for 'b' is 1 - 'a' = 1. t is updated by 1 - (-1) = 2. Now ans += t (now ans = 3). Update pos[1] with the current index (pos[1] = 1).

**Character 'a' at index 2:** 'a' has been seen before at index 0. The index for 'a' is 0. t is updated by 2 - 0 = 2. Add t to ans (now ans = 3 + 2 = 5). Update pos[0] with the current index (pos[0] = 2).

**Character 'c' at index 3:** 'c' has not been seen before. The index for 'c' is 2 - 'a' = 2. t is updated by 3 - (-1) = 4. ans += t (now ans = 5 + 4 = 9). Update pos[2] with the current index (pos[2] = 3).

**Character 'a' at index 4:** 'a' has been seen before at index 2. The index for 'a' is 0. t is updated by 4 - 2 = 2. ans += t (now ans = 9 + 2 = 11). Update pos[0] with the current index (pos[0] = 4).

**Result:** After iterating through the entire string, ans holds the total appeal of all possible substrings, which is 11 for the example string "abaca".

This walkthrough demonstrates how the solution efficiently calculates each character's contribution to the total appeal as we process the string, resulting in an optimal calculation without examining each substring individually.

**Solution Implementation**

class Solution {

// This method calculates the sum of the appeal of all substrings of a given string s.

// The appeal of a string is defined as the number of distinct characters found in the string.

public long appealSum(String s) {

long totalAppeal = 0; // This variable will store the sum of the appeal of all substrings.

long currentAppeal = 0; // This variable stores the appeal of the substring ending at the current character.

int[] lastPosition = new int[26]; // An array to store the last position of each character.

Arrays.fill(lastPosition, -1); // Initialize last positions to -1 for all characters.

// Iterate over each character in the string to compute the appeal of all possible substrings.

for (int i = 0; i < s.length(); ++i) {

int charIndex = s.charAt(i) - 'a'; // Convert the char to an index 0-25 corresponding to 'a'-'z'.

// Update the current appeal by adding the contribution of the current character.

// The contribution is the difference between the current position and its last seen position.

currentAppeal += i - lastPosition[charIndex];

// Add the current appeal to the total appeal.

totalAppeal += currentAppeal;

// Update the last seen position for the current character.

lastPosition[charIndex] = i;

}

// Return the total appeal sum of all substrings.

return totalAppeal;

}

}

**Time and Space Complexity**

The given code calculates the sum of the appeals of all substrings of a string.

**Time Complexity**

We iterate through each character in the input string only once. The loop runs for n iterations if n is the length of the input string s.

Inside the loop, we perform constant-time operations: indexing an array (pos[c]), arithmetic operations (t += i - pos[c] and ans += t), and updating an array element (pos[c] = i). Since these operations do not depend on the size of n and are done in constant time, the time complexity of the loop is O(n).

Therefore, the overall time complexity of the code is O(n).

**Space Complexity**

We are using an extra array pos of size 26 to keep track of the last positions of each character that appears in the string. The size of this array depends on the size of the character set |\Sigma|, which in this case is the English alphabet and hence |\Sigma| = 26.

Therefore, the space complexity of the code is O(|\Sigma|), which is O(26) for this problem.

Since 26 is a constant and does not change with the input size, we could also consider the space complexity as O(1), constant space.

**Refer to**

[L828.Count Unique Characters of All Substrings of a Given String (Ref.L2262)](note://WEBf394faa89bf4c60aa6f2c29c102ef307)