Previous Next

466. Count The Repetitions

Aug. 23, 2017 | 13.3K views

Define `S = [s,n]` as the string S which consists of n connected strings s. For example, `["abc", 3]` ="abcabcabc".

On the other hand, we define that string s1 can be obtained from string s2 if we can remove some characters from s2 such that it becomes s1. For example, "abc" can be obtained from "abdbec" based on our definition, but it can not be obtained from "acbbe".

You are given two non-empty strings s1 and s2 (each at most 100 characters long) and two integers $0 \le n1 \le 10^6$ and $1 \le n2 \le 10^6$. Now consider the strings S1 and S2, where `S1=[s1,n1]` and `S2=[s2,n2]`. Find the maximum integer M such that `[S2,M]` can be obtained from `S1`.

Example:

```
Input:
s1="acb", n1=4
s2="ab", n2=2

Return:
2
```

Solution

Approach #1 Brute force [Time Limit Exceeded]

Intuition

According to the question, we need to find m such that $[S2, m]$ is the largest subsequence that can be found in $S1$. $S2$ is essentially $[s2, n2]$ and $S1$ is $[s1, n1]$ and so, we can find the number of times $s2$ repeats in $[s1, n1]$, say $\mathbf{repeat\_count}$. And the number of times $S2$ repeats in $S1$ is therefore $(\mathbf{repeat\_count/n2})$. Simple.

Algorithm

- Initialize $\mathrm{index}=0$ and $\mathbf{repeat\_count}=0$. index represents the current index in $s2$ to be checked against $s1$ and $\mathbf{repeat\_count}$ represents the number of times $s2$ repeats in $S1$.
- Iterate over the variable i from 0 to $n1-1$:
  - Iterate over the variable j from 0 to $\mathrm{size(s1)}-1$:
    - If $\mathrm{s1[j]}$ is equal to $\mathrm{s2[index]}$, increment index.
    - If $index$ is equal to $size(s2)$, this implies that $s2$ has completed one repartition and hence set $\mathrm{index}=0$ and increment the $\mathbf{repeat\_count}$.
- Return $(\mathbf{repeat\_count}\ /\ \mathbf{n2})$ since, $S2$ is $[\mathrm{s2,n2}]$.

C++ Copy

```cpp
int getMaxRepetitions(string s1, int n1, string s2, int n2)
{
    int index = 0, repeat_count = 0;
    int s1_size = s1.size(), s2_size = s2.size();
    for (int i = 0; i < n1; i++) {
        for (int j = 0; j < s1_size; j++) {
            if (s1[j] == s2[index])
                ++index;
            if (index == s2_size) {
                index = 0;
                ++repeat_count;
            }
        }
    }
    return repeat_count / n2;
}
```

Complexity Analysis

- Time complexity: $O(n1 * size(s1))$.
  - We iterate over the entire length of string $s1$ for $n1$ times.
- Space complexity: $O(1)$ extra space for index and $\mathbf{repeat\_count}$.

Approach #2 A better brute force [Accepted]

Intuition

In Approach #1, we simply checked for repetition over the entire $[s1, n1]$. However, $n1$ could be quiet large and thus, is inefficient to iterate over complete $S1$. We can take advantage of the fact that $s1$ is repeating and hence, we could find a pattern of repetition of $s2$ in $S1$. Once, we get the repetition pattern, we can easy calculate how many times the pattern repeats in $n2$ in $O(1)$.

But what's the pattern!

In approach #1, we kept index which tells the index to search in $s2$. We try to see in the below illustration if this index repeats itself after some fixed iterations of $s1$ or not and if so, then how can we leverage it.

After finding the repitition pattern, we can calculate the sum of repeating pattern, part before repitition and part left after repitition as the result in $O(1)$.

But will this repitition always take place?

Yes! By **Pigeonhole principle**, which states that if n items are put into m containers, with $n > m$, then at least one container must contain more than one item. So, according to this, we are sure to find 2 same $index$ after scanning at max $\mathrm{size(s2)}$ blocks of $s1$.

Algorithm

- Intialize $count = 0$ nd $index = 0$, which are same as in Approach #1.
- Initialize 2 arrays, say indexr and countr of size $(\mathrm{size(s2)}+1)$, initialized with 0. The size $(\mathrm{size(s2)}+1)$ is based on the Pigeonhole principle as discussed above. The 2 arrays specifies the index and count at the start of each $s1$ block.
- Iterate over i from 0 to $n1-1$:
  - Iterate over j from 0 to $\mathrm{size(s1)}-1$:
    - If $\mathrm{s1[j]} == \mathrm{s2[index]}$, increment index.
    - If index is equal to $\mathrm{size(s2)}$, set $\mathrm{index}=0$ and increment count.
  - Set $\mathrm{countr[i]}=\mathrm{count}$ and $\mathrm{indexr[i]}=\mathrm{index}$
  - Iterate over k from 0 to $i-1$:
    - If we find the repitition, i.e. current $\mathrm{index}=\mathrm{indexr[k]}$, we calculate the count for block before the repitition starts, the repeating block and the block left after repitition pattern, which can be calculated as:

$$\mathbf{prev\_count} = \mathbf{countr}[k]$$
$$\mathbf{pattern\_count} = (\mathbf{countr}[i] - \mathbf{countr}[k]) * \frac{\mathbf{n1} - \mathbf{1} - k}{i - k}$$
$$\mathbf{remain\_count} = \mathbf{countr}\,[k + (\mathbf{n1} - \mathbf{1} - k)\,\%\,(i - k)] - \mathbf{countr}[k]$$

    - Sum the 3 counts and return the sum divided by $n2$, since $\mathrm{S2}=[\mathrm{s2,n2}]$
    - If no repetition is found, return $\mathrm{countr[n1\text{-}1]/n2}$.

C++ Copy

```cpp
2  {
3      if (n1 == 0)
4          return 0;
5      int indexr[s2.size() + 1] = { 0 }; // index at start of each s1 block
6      int countr[s2.size() + 1] = { 0 }; // count of repititions till the present s1 block
7      int index = 0, count = 0;
8      for (int i = 0; i < n1; i++) {
9          for (int j = 0; j < s1.size(); j++) {
10             if (s1[j] == s2[index])
11                 ++index;
12             if (index == s2.size()) {
13                 index = 0;
14                 ++count;
15             }
16         }
17         countr[i] = count;
18         indexr[i] = index;
19         for (int k = 0; k < i; k++) {
20             if (indexr[k] == index) {
21                 int prev_count = countr[k];
22                 int pattern_count = (countr[i] - countr[k]) * (n1 - 1 - k) / (i - k);
23                 int remain_count = countr[k + (n1 - 1 - k) % (i - k)] - countr[k];
24                 return (prev_count + pattern_count + remain_count) / n2;
25             }
26         }
27     }
28     return countr[n1 - 1] / n2;
29 }
```

Complexity analysis

- Time complexity: $\mathrm{O(size(s1)*size(s2))}$.
  - According to the Pigeonhole principle, we need to iterate over $s1$ only $(\mathrm{size(s2)}+1)$ times at max.
- Space complexity: $O(\mathrm{size(s2)})$ extra space for indexr and countr string.

Rate this article:

Previous Next

Comments: 8 Sort By

Type comment here... (Markdown is supported)

Preview Post

yuxuzi 158 May 4, 2018 4:18 AM

Input:
"aaa"
20
"aaaaa"
1

Read More

18 Share Reply

learnMachining 7 June 17, 2018 3:40 PM

Nice solution and good explanation!
However, Line 22 should be:

```
int pattern_count = (countr[i] - countr[k]) * ((n1 - 1 - k) / (i - k));
```

Read More

7 Share Reply

Albert_G 10 September 24, 2017 2:38 PM

Does this line have problem? (countr[i] - countr[k]) * (n1 - 1 - k) / (i - k)
Shouldn't it be (countr[i] - countr[k]) * ((n1 - 1 - k) / (i - k)).
For example if (countr[i] - countr[k]) = 3, (n1 - 1 - k) = 19, (i - k) = 5.
Than use the first line, it is (3 * 19) / 5 = 11;
But actually it should be 9, which is 3 * (19 / 5).

Read More

3 Share Reply

Teppi 3 November 16, 2017 2:09 PM

Yes, I think you're right. For Python, that line should be changed to (countr[i] - countr[k]) * ((n1 - 1 - k) / (i - k)) to pass the 41st test case.

2 Share Reply

SHOW 2 REPLIES

CatherineWong 2 October 29, 2018 8:47 AM

the figure of "Count the repetition" is missing

1 Share Reply

mdholakia 0 October 4, 2018 1:03 AM

Aren't, the example strings used in Approach#2 solutions incorrect?
s1: "abaacdbac" and s2: "adcbd" explanation incorrect:
However, per the problem definition, s2 must be obtainable from s1 by dropping letters without rearranging - "For example, 'abc' can be obtained from 'abdbec' based on our definition, but it can not be obtained from 'acbbe'.

Read More

0 Share Reply

SHOW 1 REPLY

liyiou 23 May 21, 2020 8:20 PM

I originally had a question about the remain_count,

```
int remainCount = countr[k + (n1 - 1 - k) % (i - k)] - countr[k];
```

Read More

0 Share Reply

meganlee 1093 June 17, 2018 3:10 AM

Another way to write **Brute-force** Solution 1: it's TLE but it's correct

```
public int getMaxRepetitions(String s1, int n1, String s2, int n2) {
    int count = 0;
    // i points to cur char in s1, j points to cur char in s2
```

Read More

0 Share Reply