Count The Repetitions

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 1

IDEA:

Given a str2, for each str, we can give a value v to this str such that, after greedily looking through str, our imaginary next step is to find str2[v].
In our problem, str is always (str1,n), with a given str1, so, we can take one more step and say that for each n, there is a unique v associated to n(i.e to (str,n)).

define a division and a modulo between two strings as follow:

str/str2=argmax{i, (str2,i) can be obtained by str}
str%str2=the v mentioned above associated with str.

All possible values of v is less than str2.size(),
so (str1,n)%str2 will begin to **repeat a pattern** after a certain n less than str2.size().
(the pattern is the same because in the cases with the same v, our situations are exactly the same),
so is (str1,n)/str2-(str1,n+1)/str2 for the same reason.
We can therefore precompute a table for all these values with O(str1.length*str2.length).

(str1,n) can be divided in three parts:

sth before pattern(A) + pattern parts(B) + sth after pattern(C)

The pattern does not necessarily begin in the first str1, we shall see if n is great enough so that there can be a pattern.

The last pattern(C) is not necessarily complete, we need to calculate it separately.

We can finish in just looking to the precomputed table and doing some simple maths.

```cpp
class Solution {
public:
    int getMaxRepetitions(string s1, int n1, string s2, int n2) {
        vector<int> rapport(102,-1);
        vector<int> rest(102,-1);
        int b=-1;int posRest=0;int rap=0;
        int last=-1;
        rapport[0]=rest[0]=0;//case when n=0
        for(int i=1;i<=s2.size()+1;i++){
            int j;
            for(j=0;j<s1.size();j++){
                if(s2[posRest]==s1[j]){
                    posRest++;
                    if(posRest==s2.size()){
                        rap++;
                        posRest=0;
                    }
                }
            }
            for(int k=0;k<i;k++){
                if(posRest==rest[k]){b=k;last=i;break;}
            }
            rapport[i]=rap;rest[i]=posRest;
            if(b>=0)break;
        }
        int interval=last-b;
        if(b>=n1)return rapport[n1]/n2;
        return ((n1-b)/interval*(rapport[last]-rapport[b])+rapport[(n1-b)%interva
l+b])/n2;
//corrected thanks to @zhiqing_xiao and @iaming
    }
};
```

written by 70664914 original link here

Solution 2

I didn't come up with any good solution so I tried brute force. Key points:

1. How do we know "string s2 can be obtained from string s1"? Easy, use two pointers iterate through s2 and s1. If chars are equal, move both. Otherwise only move pointer1.
2. We repeat step 1 and go through s1 for n1 times and count how many times can we go through s2.
3. Answer to this problem is times go through s2 divide by n2.

```java
public class Solution {
    public int getMaxRepetitions(String s1, int n1, String s2, int n2) {
        char[] array1 = s1.toCharArray(), array2 = s2.toCharArray();
        int count1 = 0, count2 = 0, i = 0, j = 0;

        while (count1 < n1) {
            if (array1[i] == array2[j]) {
                j++;
                if (j == array2.length) {
                    j = 0;
                    count2++;
                }
            }
            i++;
            if (i == array1.length) {
                i = 0;
                count1++;
            }
        }

        return count2 / n2;
    }
}
```

written by shawngao original link here

Solution 3

IDEA:
Imagine s1 and s2 repeat inifinite times as below (for example s1 = "abcd" and s2 = "ab")
abcdabcdabcd...
ababab...
say <i, j> are pairs of pointers to s1 and s2 in greedy matched characters,
in above example will be
<0, 0>, <1, 1>, <4, 2>, <5, 3>, <8, 4>, <9, 5>...
In a brute force solution we can keep increasing i and j until i exceeds s1 x n1.

Say the lengths of s1 and s2 are m1 and m2. It's easy to prove that:
If there are two pairs <i1, j1>, <i2, j2> satisfying:
(i2 - i1) % m1 == 0 && (j2 - j1) % m2 == 0,
let d1 = i2 - i1 and d2 = j2 - j1,
then for all positive integer k, <i1 + d1 * k, j1 + d2 * k> will be pairs too.

So without brute force matching, my trick is to use above conclusion to push <i, j> quickly to near the end of the expanded string s1 x n1, after the first <i1, j1> and <i2, j2> pair is found. Here's the solution:

```java
public class Solution {
 public int getMaxRepetitions(String s1, int n1, String s2, int n2) {
  int m1 = s1.length();
  int m2 = s2.length();
  if (m1 == 0 || m2 == 0)
   return 0;
  int i, j;
  // extra code to remove unnecessary characters in s1
  StringBuffer sb = new StringBuffer();
  boolean[] used = new boolean[26];
  int[] counts = new int[26]; // count of each character in s1
  for (i = 0; i < m2; i++) {
   j = s2.charAt(i) - 'a';
   used[j] = true;
  }
  for (i = 0; i < m1; i++) {
   j = s1.charAt(i) - 'a';
   if (used[j])
    sb.append(s1.charAt(i));
   counts[j]++;
  }
  for (i = 0; i < 26; i++) {
   if (used[i] && counts[i] == 0) // character in s2 not in s1
    return 0;
  }
  s1 = sb.toString();
  m1 = s1.length();

  // extra code to reduce s1 and s2 if it contains repeating pattern
  for (i = 1; i <= m1 / 2; i++) {
   if (m1 % i != 0)
    continue;
   if (repeatAtK(s1, i)) {
    s1 = s1.substring(0, i);
    n1 *= m1 / i;
    m1 = i;
    break;
   }
  }
  for (i = 1; i <= m2 / 2; i++) {
   if (m2 % i != 0)
    continue;
   if (repeatAtK(s2, i)) {
    s2 = s2.substring(0, i);
    n2 *= m2 / i;
    m2 = i;
    break;
   }
  }

  int[][] ocs = new int[26][m1]; // occurrences of each character in s1
  Arrays.fill(counts, 0);
  for (i = 0; i < m1; i++) {
   j = s1.charAt(i) - 'a';
   ocs[j][counts[j]] = i;
   counts[j]++;
  }

  // simple case
  if (m2 == 1) {
   j = s2.charAt(0) - 'a';
   return counts[j]*n1/n2;
  }

  return getMaxRepetitionsProcessed(counts, ocs, n1, s2.toCharArray(), n2);
 }

 public int getMaxRepetitionsProcessed(int[] counts, int[][] ocs, int n1, char[]
ca2, int n2) {
  int m1 = ocs[0].length;
  int m2 = ca2.length;
  // <i, j> pairs in slot mod m1/m2
  int[][][] r = new int[m1][m2][2];
  // pos[c][0] is the current index of character c in i,
  // pos[c][1] is which occurrence in s1
  int[][] pos = new int[26][2];
  int i, j, k, r1 = 0, r2 = 0;
  boolean found = false;
  for (i = 0; i < 26; i++) {
   pos[i][0] = ocs[i][0];
  }
  for (i = 0; i < m1; i++) {
   for (j = 0; j < m2; j++) {
    r[i][j][0] = -1;
   }
  }
  for (i = 0, j = 0; i < m1 * n1; i++, j++) {
   k = ca2[j % m2] - 'a';
   // move pos[k] to a position equal or after i by iterating k's occurrences
   // move pos[k] to a position equal or after i by iterating k's occurrences
   while (pos[k][0] < i) {
    pos[k][1]++;
    if (pos[k][1] < counts[k]) {
     pos[k][0] += ocs[k][pos[k][1]] - ocs[k][pos[k][1] - 1];
    } else {
     pos[k][1] = 0;
     pos[k][0] += ocs[k][0] + m1 - ocs[k][counts[k] - 1];
    }
   }
   i = pos[k][0];
   if (i >= m1 * n1) {
    return j / m2 / n2;
   }
   r1 = i % m1;
   r2 = j % m2;
   if (!found && r[r1][r2][0] < 0) {
    r[r1][r2][0] = i;
    r[r1][r2][1] = j;
   } else if (!found) { // push by mod trick here
    int d1 = i - r[r1][r2][0];
    int d2 = j - r[r1][r2][1];
    k = (m1 * n1 - i) / d1;
    i += k * d1;
    j += k * d2;
    for (r1 = 0; r1 < 26; r1++) { // update all pos[c] the same way as i
     pos[r1][0] += k * d1;
    }
    found = true;
   }
  }
  return j / m2 / n2;
 }

 public boolean repeatAtK(String s, int k) { // check if s is repeated every k ch
aracters
  int m = s.length();
  int x = m / k;
  for (int i = 0; i < k; i++) {
   for (int j = 0; j < x; j++) {
    if (s.charAt(i) != s.charAt(j * k + i))
     return false;
   }
  }
  return true;
 }
}
```

Regarding the time complexity, the for loop should be less than 2 * m1 * m2 times according to pigeonhole theorem, and the while loop inside should be less thant m1 (actually can be amortized to d1/d2), so the overall time complexity should be less than O(m1 * m1 * m2). I believe in reality it should be much less.

Ugly coding because I'm not good at it, just want to share this idea to see if it makes sense. Any improvement is most welcome.

written by chenxu614 original link here