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

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.

```
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 2

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.

```
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 3

The key is, we just need to calculate ***what will remain after s1 obtains s2*** .

That is, `(s1, s2) -> (sRemain, matchCnt)` ; for example,
`(abcd, ab) -> (cd, 1)`
`(ababa, ab) -> (a, 2)`
`(a, aaaa) -> (a, 0)`
`(aabaabaab, aba) -> (ab, 2)` as `aabaaba` exactly matches `aba` twice.

And, each time we append `s1` to the `remain string` , to make a sequence: (Using `[]` to mark the `remain string`)
`(abcd, ab): abcd -> [cd]abcd -> [cd]abcd -> [cd]abcd -> ...`
`(ababa, ab): ababa -> [a]ababa -> [a]ababa -> [a]ababa -> ...`
`(a, aaaa): a -> [a]a -> [aa]a -> [aaa]a -> a -> [a]a -> [aa]a -> ...`
`(aabaabaab, aba): aabaabaab -> [ab]aabaabaab -> [ab]aabaabaab -> ...`

Obviously, there will be a loop in the sequence, assume the length of loop is `loop` , and the length before the loop is `k` .
`(abcd, ab): loop=1, k=1`
`(a, aaaa): loop=4, k=0`
`(aabaabaab, aba): loop=1, k=1`

So, we just need to calculate `the count of each loop` , and `the count before entering the loop` , and calculate the total.

Here is the code with detailed comment, 21ms.

```
public class Solution {
    public int getMaxRepetitions(String s1, int n1, String s2, int n2) {
        if (!ableToObtain(s1, s2)) return 0; // check if [s1. ∞] obtains s2
        int cnt=0, k=-1;
        String s=s1;
        StringBuilder remainBuilder; // record `remain string`
        ArrayList<String> stringList=new ArrayList<>(); // record all the `remain
string`
        ArrayList<Integer> countList=new ArrayList<>(); // record matching count
from start to the current remain string
        stringList.add(""); // record empty string
        countList.add(0);
        for (int i=0;i<=n1;i++) {
            remainBuilder=new StringBuilder();
            cnt+=getRemain(s, s2, remainBuilder); // get the next remain string,
returns the count of matching
            String remain=remainBuilder.toString();
            if ((k=stringList.indexOf(remain))!=-1) break; // if there is a loop,
break
            stringList.add(remain); // record the remain string into arraylist
            countList.add(cnt);
            s=remain+s1; // append s1 to make a new string
        }
        // here, k is the beginning of the loop
        if (k==-1) return cnt/n2; // if there is no loop
        int countOfLoop=cnt-countList.get(k), loopLength=stringList.size()-k; //
```

```
        int countOfLoop=cnt-countList.get(k), loopLength=stringList.size()-k; //
get matching count in the loop, and loop length
        cnt=countList.get(k);
        n1-=k;
        cnt+=countOfLoop*(n1/loopLength);
        n1%=loopLength;
        cnt+=countList.get(k+n1)-countList.get(k);
        return cnt/n2;
    }

    // check if [s1. ∞] obtains s2
    private boolean ableToObtain(String s1, String s2) {
        boolean[] cnt=new boolean[26];
        for (char c: s1.toCharArray()) cnt[c-'a']=true;
        for (char c: s2.toCharArray()) {
            if (!cnt[c-'a']) return false;
        }
        return true;
    }

    // get remain string after s1 obtains s2, return the matching count
    private int getRemain(String s1, String s2, StringBuilder remain) {
        int cnt=0, lastMatch=-1, p2=0;
        for (int p1=0;p1<s1.length();p1++) {
            if (s1.charAt(p1)==s2.charAt(p2)) {
                if (++p2==s2.length()) {
                    p2=0;
                    cnt++;
                    lastMatch=p1;
                }
            }
        }
        remain.append(s1.substring(lastMatch+1));
        return cnt;
    }
}
```

written by ckcz123 original link here