Unique Substrings in Wraparound String

Consider the string `s` to be the infinite wraparound string of "abcdefghijklmnopqrstuvwxyz", so `s` will look like this: "...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....".

Now we have another string `p`. Your job is to find out how many unique non-empty substrings of `p` are present in `s`. In particular, your input is the string `p` and you need to output the number of different non-empty substrings of `p` in the string `s`.

Note: `p` consists of only lowercase English letters and the size of p might be over 10000.

Example 1:

```
Input: "a"
Output: 1

Explanation: Only the substring "a" of string "a" is in the string s.
```

Example 2:

```
Input: "cac"
Output: 2
Explanation: There are two substrings "a", "c" of string "cac" in the string s.
```

Example 3:

```
Input: "zab"
Output: 6
Explanation: There are six substrings "z", "a", "b", "za", "ab", "zab" of string "z
ab" in the string s.
```

Solution 1

After failed with pure math solution and time out with DFS solution, I finally realized that this is a DP problem...
The idea is, if we know the max number of unique substrings in `p` ends with `'a', 'b', ..., 'z'`, then the summary of them is the answer. Why is that?

1. The max number of unique substring ends with a letter equals to the length of max contiguous substring ends with that letter. Example `"abcd"`, the max number of unique substring ends with `'d'` is 4, apparently they are `"abcd", "bcd", "cd" and "d"`.
2. If there are overlapping, we only need to consider the longest one because it covers all the possible substrings. Example: `"abcdbcd"`, the max number of unique substring ends with `'d'` is 4 and all substrings formed by the 2nd `"bcd"` part are covered in the 4 substrings already.
3. No matter how long is a contiguous substring in `p`, it is in `s` since `s` has infinite length.
4. Now we know the max number of unique substrings in `p` ends with `'a', 'b', ..., 'z'` and those substrings are all in `s`. Summary is the answer, according to the question.

Hope I made myself clear...

```java
public class Solution {
    public int findSubstringInWraproundString(String p) {
        // count[i] is the maximum unique substring end with ith letter.
        // 0 - 'a', 1 - 'b', ..., 25 - 'z'.
        int[] count = new int[26];

        // store longest contiguous substring ends at current position.
        int maxLengthCur = 0;

        for (int i = 0; i < p.length(); i++) {
            if (i > 0 && (p.charAt(i) - p.charAt(i - 1) == 1 || (p.charAt(i - 1)
- p.charAt(i) == 25))) {
                maxLengthCur++;
            }
            else {
                maxLengthCur = 1;
            }

            int index = p.charAt(i) - 'a';
            count[index] = Math.max(count[index], maxLengthCur);
        }

        // Sum to get result
        int sum = 0;
        for (int i = 0; i < 26; i++) {
            sum += count[i];
        }
        return sum;
    }
}
```

written by shawngao original link here

Solution 2

```
int findSubstringInWraproundString(string p) {
        vector<int> letters(26, 0);
        int res = 0, len = 0;
        for (int i = 0; i < p.size(); i++) {
            int cur = p[i] - 'a';
            if (i > 0 && p[i - 1] != (cur + 26 - 1) % 26 + 'a') len = 0;
            if (++len > letters[cur]) {
                res += len - letters[cur];
                letters[cur] = len;
            }
        }
        return res;
    }
```

written by zyoppy008 original link here

Solution 3

Hi there! I am sharing two different solutions with explanations.

Well, the first solution idea is Optimized brute force. The 'naivest' way is for each substring of p in alphabetic order (with rotation %26) brute force all substrings and add to some set. Then at last just return size of the set. That method works but it has two problems, they are:

- Memory limit
- Time limit

Because it works for O(n^3) time and O(n^2) space.
How can we optimize memory? Because we are considering only strings in alphabetic order, it is sufficient to remember the first character index, last character index and the length for each found substring. This way we replace set of strings to 3D boolean array.
Well, how can we optimize time complexity? We can increase performance of counting unique substrings for already found substring, which size is greater than 26. The evidence mentioned above helps us again, because we just need to consider combinations of first 26 character and the last 26 characters for different length, by incrementing length by 26. Such a way we get algorithm that runs for O(26*n) time and O((26^2)*n) space complexities.

```
public class Solution {
    public int findSubstringInWraproundString(String p) {
        if(p == null || p.isEmpty()) return 0;
        boolean [][][] set = new boolean[26][26][p.length()+1];
        int i = 0;
        int count = 0;
        boolean [] visited = new boolean[26];
        StringBuilder build = new StringBuilder();
        char [] s = p.toCharArray();
        int n = s.length;
        while(i<n){
            char prev = s[i];
            build.append(prev);
            i++;
            while(i<p.length() && s[i]-'a' == (prev-'a'+1)%26){
                prev = s[i];
                build.append(prev);
                i++;
            }

            String next = build.toString();
            int l = next.charAt(0)-'a';
            int r = next.charAt(next.length()-1)-'a';
            if(!set[l][r][next.length()]){
                count++;
                set[l][r][next.length()] = true;
                count+= countUniqueSubstr(next, set);
            }
            build.setLength(0);
        }
```

```
        return count;
    }


    public int countUniqueSubstr(String str, boolean [][][] set){
        int count = 0;
        if(str.length()>26){
            int n = str.length();
            for(int i = 0;i<26;i++){
                int l = str.charAt(i)-'a';
                for(int j = n-1;j>=n-26;j--){
                    int r = str.charAt(j)-'a';
                    int limit = j-i+1;
                    int start = r-l+1;
                    if(r<l){
                        start = 27-l+r;
                    }
                    for(int size = start;size<=limit;size+=26){
                        if(!set[l][r][size]){
                            count++;
                            set[l][r][size] = true;
                        }
                    }
                }
            }

        } else {
            for(int size = 1;size<str.length();size++){
                for(int i = 0;i<=str.length()-size;i++){
                    String s = str.substring(i, i+size);
                    int l = s.charAt(0)-'a';
                    int r = s.charAt(s.length()-1)-'a';
                    if(!set[l][r][s.length()]){
                        count++;
                        set[l][r][s.length()] = true;
                    }
                }
            }
        }
        return count;
    }
}
```

The second solution is simple, just keep track of maximum length of alphabetic substrings ending at certain character then sum them up.

```java
public class Solution {
    public int findSubstringInWraproundString(String p) {
        if(p == null || p.isEmpty()) return 0;
        int dp[] = new int[26];
        int i = 0;
        int n = p.length();
        char [] s = p.toCharArray();
        int len = 1;
        while(i<n){
            char prev = s[i];
            i++;
            dp[prev - 'a'] = Math.max(dp[prev-'a'], len);
            while(i<p.length() && s[i]-'a' == (prev-'a'+1)%26){
                prev = s[i];
                len++;
                i++;
                dp[prev - 'a'] = Math.max(dp[prev-'a'], len);
            }
            dp[prev - 'a'] = Math.max(dp[prev-'a'], len);
            len = 1;
        }
        int count = 0;
        for(int j = 0;j<26;j++) count+=dp[j];
        return count;
    }
}
```

P.S: Sorry for poor and dirty code. I hope it will be understandable anyway

written by ZhassanB original link here

From Leetcoder.