ANUSAR - Editorial

9

4

PROBLEM LINK:

Practice
Contest

Author: Anudeep Nekkanti
Tester: Gerald Agapov
Editorialist: Praveen Dhinwa

DIFFICULTY:

MEDIUM HARD

PREREQUISITES:

Suffix Array, Suffix Tree, dfs, Segment tree, Fenwick Tree or BIT.

PROBLEM:

Given a string S and Q queries. In each query you are given an integer F. You need to find out number of substrings of S which occur at least F times in S.

QUICK EXPLANATION

Suffix Array solution:
Construct suffix array SA of the string S. Create an array LCP such that LCP[i] = lcp(SA[i], SA[i + 1]). Full form of lcp is longest common prefix.

Now the LCP array can be seen as a histogram with each bar having height LCP[i]. Then for some range [i,j] if minimum of [LCP[i], LCP[i+1].. LCP[j]] is m, then it means that there is a substring of length m, which is present at j-i+2 indices in string S.

Using this property we can find the solution in following way. Let D[i] represent the number of substrings which repeat exactly i times. If we can compute the array D, we can easily answer all the queries. For a given frequency F, required answer will be D[F] + D[F+1] + ... D[length of S].

For a bar of height H (LCP[i] = H), let us assume that we know the largest interval (L[i], R[i]) such that minimum of LCP array over the interval is equal to H, then it means that we have got a substring of length H, which repeats exactly R[i]-L[i]+2 times. (See examples given in the explanation).
We will update the D array accordingly.

Suffix Tree Solution
Construct suffix tree of the string S, then you just need to do a dfs over it and keep updating the count of appearance of substrings.

EXPLANATION

First we will explain more about significance of array D and how it helps to solve the problem. Then the next section will elaborate on different ways of computing L[i] and R[i] for a given histogram H.

Let us take an example string and create its suffix array. Consider the string S = "ABABBAABB".

Suffix position in S	Suffixes of S	LCP array of S
6	AABB	1
1	ABABBAABB	2
7	ABB	3
3	ABBAABB	0
9	B	1
5	BAABB	2
2	BABBAABB	1
4	BBAABB	2
8	BB	Not Defined

Now A appears 4 times in S. A is the prefix of first 4 suffixes in the suffix array. Minimum of LCP[0], LCP[1], LCP[2] (0 based indexing) is 1 which represents that A has appeared 4 times.

As pointed out in the quick explanation section, we can express LCP array as histogram with following values [1, 2, 3, 0, 1, 2, 1, 2]. You can also very easily verify the fact that if for some range[i, j], minimum of LCP array is m, then their exists a substring of length m occurring at j - i + 2 positions in the string S.

L[i] for given LCP will be [0, 1, 2, 0, 4, 5, 4, 7].
R[i] for given LCP will be [2, 2, 2, 7, 7, 5, 7, 7].

Follow this question
By Email:
You are not subscribed to this question.

subscribe me

(you can adjust your notification settings on your profile)

By RSS:
Answers

Answers and Comments

Tags:

editorial	×3,002
segment-tree	×350
medium-hard	×220
stack	×33
suffix-array	×31
cook46	×8
disjoint-set	×7
suffix-trees	×3

Asked: **19 May '14, 00:24**

Seen: **2,644 times**

Last updated: **08 Jun, 01:22**

Related questions

ACM14KP3 - Editorial
LCH15JGH - Editorial
ACM14KG4 - Editorial
UNIQUE - Editorial
FBCHEF - Editorial
STREETTA - Editorial
ACM14KP4 - Editorial
SSTORY - Editorial
TRIPS-Editorial
ACM14AM5-Editorial

Now let us find out how to update D using LCP.
We will iterate over each possible value in the LCP array (in whichever order you wish), Let the value on which we are currently iterating be V.

As we know that for the range [L[i], R[i]], minimum value of LCP array is equal to V.
Note that there are exactly V substrings (consider any prefix of the suffix corresponding to LCP[i], ie longest common prefix of i and i + 1 in the suffix array, it has size V and hence there will be V prefixes of it). Hence there will V * (D[R[i] - L[i] + 2) substrings repeating (R[i] - L[i] + 2) times. So we will update D[R[i] - L[i] + 2] by adding val * D[R[i] - L[i] + 2] to it.

eg. Let us suppose we have following suffixes in the sorted order.
AA
AAB
AAC
LCP = {2, 2}.
L = {0, 0}.
R = {1, 1}.

When we are iterating over V = 2, Assume we are at position i = 0. V = 2 and R[i] - L[i] + 2 = 3. We will add 2 * 3 = 6 into D[3]. In other words it refers to adding substrings A and AA 3 times.

The above mentioned approach has some small *pitfalls* which need to be taken care of. We are doing some over-counting in the approach. We need to fix that up. There are two kind of issues with the above approach which are explained in detail here.

Issue 1
Let us consider that a string S = ABABAA has following suffixes
A
AA
ABAA
ABABAA
BAA
BABAA

LCP = [1, 1, 3, 0, 2] L = [0, 0, 2, 0, 4] R = [2, 2, 2, 4, 4]

When we are iterating over LCP array, Let us assume that current V = 1, If we are at i = 0 (ie at suffix A) we will count that A has occurred exactly 4 times (because R[0] - L[0] + 2 = 4, LCP[0] = 1). But when are considering V = 2, If we are at position i = 2 (ie we are at suffix ABAA), we have R[2] = 2, L[2] = 2. We will increment the count of D[(R[2] - L[2] + 2)] ie D[2] by 2, (we are updating count of occurrences of substrings A and AB by 2 more). But this is not correct, because in this approach we have counted A 6 times, which is wrong (You can easily see that A appears exactly 4 times).

So There is over counting in our current method. We need to somehow get rid of this. Note that at suffix ABAA, the suffix just preceding it is AA.

For V = 2 and suffix ABAA, when we count for A we have already counted the fact that substring A has appeared four times when the V = 1.

So we can not say that all prefixes of length LCP[2] are counted exactly once. Notice that LCP[L[i] - 1] is 1 and hence we are going to recount just the first prefix of suffix ABAA (ie A). So instead of directly saying that there are exactly LCP[i] substrings which appear exactly R[i] - L[i] + 2 times, we will say that there are exactly min(LCP[i], LCP[i] - LCP[L[i] - 1], LCP[i] - LCP[R[i] + 1]) unique substrings appearing exactly R[i] - L[i] + 2 times. If you have not understood this fact, **please** take more examples and convince yourself.

Issue 2
Now let us consider that our suffix array have following suffixes.
A
AB
ACD

Our LCP array will be [1, 1].

As said earlier, we will iterating over LCP and our current V = 1.Assume that we are currently at the suffix A (ie i = 0), we have R[i] = 1. We will update D[3] by 1 * 3.
When we go to the i = 1, we will also do the same. This amounts to over counting.
For a particular V in the LCP array, When at some point I have considered the range between L[i] and R[i] corresponding to some i (st LCP[i] = V), then I should not re-count the values corresponding to index j (LCP[j] = V) such that j lies in the range L[i] to R[i] because this range has already been considered and it will only amount to over counting.

We can implement this by a two pointer method, We can maintain a pointer 'right' which denote the rightmost index which has been considered. Note that right is always non-decreasing, Hence it will guarantee that our algorithm takes O(N) time.

Pseudo Code:
Let P be a list of lists, P[i] denotes the positions of occurrence of i in LCP array. eg. If LCP = [1, 2, 0, 1, 2, 1]
Then P will contain three list. P[0] = {2}, P[1] = {0, 3, 5}, P[2] = {1, 4}.

```
for V = 0 to N:
    sz = P[val].size()
    // right is maintained for implementing two pointer method.
    right = 0;
    for j = 0 to sz:
        index = P[val][j]
        // to take care of the issue 2, dont overcount, always check whether your current
element under
        // consideration is not inside the range which has already been considered.
        if (index >= right):
            lo = L[index], hi = R[index]
            // len denotes the current number of prefixes of the suffix corresponding to size V.
            len = V;
            // to take care of the issue 1.
            if (hi is defined, ie 0 <= hi < LCP.size()):
                len = min(len, val - LCP[hi])
            if (lo is defined, ie 0 <= lo < LCP.size()):
                len = min(len, val - LCP[hi]);
            // add the current prefixes
            D[hi-lo+2] += len * (hi - lo + 2);

            // update the right pointer
            right = index + 1;
```

Now only difficulty in the entire algorithm described is to how to compute L[i] and R[i] for a given array LCP.

Ways of Computing L[i], R[i] for all elements of Array A

Now you have to solve the following problem. You are given an array A. For each element A[i] you have to find L[i] and R[i] where L[i] is the largest j <= i such that L[k] >= L[i] for all k lying between i and j. Similarly define R[i]. (In other words, For each index i, [L[i], R[i]] denotes the largest range such that all elements have minimum value equal to A[i].)

Now if we can solve the problem of finding L[i], we can easily solve the problem of finding R[i] for each i (We can just reverse the array A and then finding R[i] is exactly the same as finding L[i]). Hence from now on, we will only look forward to solve problem of finding L[i].

If you have not solved the problem HISTOGRA on spoj, then try solving that. There are a lot of ways of solving this problem, you can read University of Ulm Local Contest judges solutions for problem H and solution mentioned on geeksforgeeks site. I will very briefly go through all of the methods mentioned there. Note that all these methods are explained in the details in the above given links. You are highly recommended to read those.

Segment Tree Based Solution

As we know the largest value in the array A can go up to 10^5, Hence our segment tree is made on new array B of size [10^5 + 1] where B[j] shows that largest index i where j has occurred in the array A.
We scan the array A from left to right. For finding out L[i], we can query the segment tree built on array B to find out the maximum value in the range [0, B[A[i]]].
For updating the array B, we can simply add B[A[i]] = i and update the segment tree accordingly. Essentially we have to maintain the information in the segment tree about the indices of the elements and maximum of each node and we will go from left to right and will do update and query operations the segment tree accordingly.

For reference solution, you can see editorialists solution to get an idea of its implementation. There are two segment trees, minSegmentTree and maxSegmentTree representing finding L[i] and R[i] respectively. Rest details are just similar to things explained before.

Stack Based Solution
Assume that we maintain a stack. Initially the stack is empty. We go from left to right in the array A, we will pop out the elements in stack which are >= A[i], then the element at the top is the element which we were searching for, we can easily find its index (For that we can store pair of [value, index] in the stack). L[i] will be the value of the of index of the topmost element in the stack. We also have to add the current element into the stack.

Observe the following important properties:
1. Elements in the stack are always in the increasing order. This fact is very important for correctness of the algorithm.
2. Each element is inserted at most once. Only the inserted elements in the stack are popped out, Hence this ensures that complexity of the entire algorithm is O(N).

For sample implementation of this idea, you can refer setter solution.

Disjoint Set Union Based solution
This is also an interesting solution, You should look into this link for getting more ideas about the solution. It is left as a home work for the readers. You can look at setter solution to get an example of working solution.

Suffix Tree Approach

You can solve the task easily by using suffix tree structure, You should do dfs over the suffix tree nodes. To get exact implementation details, you can view tester's solution.

AUTHOR'S, TESTER'S AND EDITORIALIST's SOLUTIONS:

Author's solution based on stack + suffix array
Author's solution based on DSU + suffix array
Tester's solution based on suffix tree
Editorialists's solution based on segment tree + suffix array

segment-tree disjoint-set editorial suffix-array cook46 suffix-trees medium-hard stack

This question is marked "community wiki". edited 20 May '14, 10:57 asked 19 May '14, 00:24
dpraveen ♦♦
1.4k●52●76●89
accept rate: 11%

@All, If something is unclear or grammatically incorrect, please point that out. If you feel that any portion is not described well, then please don't hesitate to point that out. Thank you.

dpraveen ♦♦ (19 May '14, 08:38)

There should be a minor correction in the suffix array explanation:

> " ... we can express LCP array as histogram with following values [1, 2, 3, 0, 1, 5, 1, 2]. "

The histogram must have values [1, 2 , 3 , 0, 1, **2**, 1, 2].

wittyceaser (19 May '14, 23:40)

> "... Hence there will V * (D[R[i] - L[i] + 2]) substrings repeating (R[i] - L[i] + 2) times. So we will update D[R[i] - L[i] + 2] by adding val * D[R[i] - L[i] + 2] to it."

How is the term (R[i] - L[i] + 2) obtained? Specifically, the "... **+ 2**" part.

wittyceaser (20 May '14, 00:13)

Let us consider that a string S has following suffixes
ABAA
ABCB
ABCD So LCP = [2, 3]. L = [0, 1] R = [1, 1] If you are at suffix ABAA(ie. i = 0), Your LCP[i] = 2, (R[i] - L[i] + 2 = 3), 3 tells you the suffix AB has appeared at 3 different positions ie at suffixes (ABAA, ABCB, ABCD, all the three suffixes)

dpraveen ♦♦ (20 May '14, 11:05)

4 Answers:

oldest newest **most voted**

2

The setter's solutions seems to be wrong, as tested on one recent problem :P

link | award points

answered **08 Jun, 01:22**
amitsaharana
553●2●6●11
accept rate: 0%

1

There is also rather easy solution using suffix automaton. Namely, for each state *v* you know how many substrings correspond to *v* and how many times each of them occurs in the string(number of paths from the start to *v* and from *v* to the end, respectively). Then you just multiply something and count some partial sums :D

Also, you must take care of multiple final states in the automaton, but it's all details.

link | award points

answered **19 May '14, 00:37**
mmaxio
61●1●2●7
accept rate: 0%

@mmaxio: Suffix array solution is too complicated to explain :(But as many people wont be comfortable with suffix tree, hence I decide to explain the suffix array solution itself. Tester solved it using dfs over suffix tree.

dpraveen ♦♦ (19 May '14, 08:09)

0

Can anybody explain suffix tree concept that how can we solve this problem using Suffix Tree + DFS.. ??

link | award points

edited **19 May '14, 17:32**

answered **19 May '14, 17:31**
paras_meena
1
accept rate: 0%

I will explain the solution in slightly more detail today. Thank you for having patience.

dpraveen ♦♦ (19 May '14, 18:18)

Essentially idea is almost similar to suffix array, using dfs we will find the number of substrings which will start at that node and add their count to our currently built answer. If you know the suffix tree, it wont be tough to formalize this. You can view mmaxio's above comment for suffix automata.

dpraveen ♦♦ (20 May '14, 11:00)

0

I have a question regarding tester's solution. Which algorithm is used to build the suffix tree ? or is it based on this link
http://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english

link | award points

answered **19 May '14, 20:24**
deepanshum007
137●1●1●3
accept rate: 0%

As far as I know, it is. It is very similar to the link you have mentioned.

dpraveen ♦♦ (20 May '14, 11:02)

Your answer

B *I* ?

[hide preview] community wiki

Post Your Answer