Suffix Tries with Strings

locked

by pruthvishalcodi1

Editorial by PruthvishE

This was the hardest problem of the contest set by Gennady. It's an awefully difficult problem so don't be disappointed if you don't understand this editorial fully in first go.

We'd call any other string whose suffix tree is equivalent to S's suffix tree as good string and through out deal in 1-based strings. Now let's start making some observations :

Number of leaves in suffix trie depend upon length of the longest suffix of S which is also a prefix of some other suffix of S. Further more if length of this suffix (denoted as X from now on) is L, number of leaves in suffix trie is N - L. : Let's start adding suffixes to our trie one by one starting from the longest one. Uptil some point, for each new suffix we get a new leaf in our trie. Now consider that first suffix which when added to trie creates no new leaf. This suffix couldnt have created any node either as any suffix which creates new node creates a leaf. So it means this whole suffix was present as path from root already. Hence it was prefix of some suffix which was already added. Any suffix that'd be added after this won't create any new leaves either. Further more, if length of the first suffix that dint create a leaf was L, we had already added N-L suffixes creating N-L leaves. So if L is the length of the longest suffix which is also the prefix of some other suffix, there are exactly N-L leaves in the tree.

Number of distinct letters in any good string is constant. Degree of root of the tree is equal to the number of distinct letters in the string. As for isomorphic trees, degrees are also preserved, degree of root is also preserved and hence all good strings have same number of different letters.

All distinct letters in any good string would've appeared atleast once in its first N-L letters. If possible assume this is not the case : there is a good string for which there is some letter that occurs for the first time after N-L letters.

S----- (N - L) ----------> X---- L —> P---- L —>

S----- (N - L) ----------> X---- L —> P---- L —>

These are the two cases. In first case clearly all letters of X have appeared in first N-L characters. In second case, take the first occurence of character that appears first time in X. This same character also must appear in portion of P which contains X and hence appears to the left as well but this is contradiction as we chose the first occurrence of that letter.

For all 1 <= i, j <= N-L, if ith and jth letters were same in S, they would be same in all good strings and if they were different in S, they'd be different in all good strings. [a]Assume they were same in S. Then look at paths of length N-i+1 and N-j+1 from leaf to root in tree. Such paths exist and are unique - they belong to ith suffix and jth suffix respectively. If ith

Statistics

Difficulty: Medium
Required Knowledge: Tries, Hashing, KMP / Z algorithm
Publish Date: Jul 06 2019

and jth letter were same in S, these two paths share atleast 1 edge. As all trees are isomorphic, they all have this property. Also it is easy to see that if a tree has this property, then ith and jth letters have to be same. Hence the claim.

First N-L letters of any good string are uniquely determined modulo some permutation. From previous three claims, all good strings contain exactly D distinct letters which have appeared in first N-L characters. Also we know that which characters are equal amongst first N-L characters and which are not. This way we can partition first N-L character into D groups - where each group has to get a distinct letter. We can assign letters to these groups in 26 * 25 * ... (26-D+1) ways. Modulo this permutation, all letters have been fixed.

From now on, we will assume that these N-L letters are same as first N-L letters of S and would later multiply our answer by 26 * 25 ... (26-D+1).

There are at most N-L ways of filling remaining L characters. We know that last L character (also called as suffix X in discussion) have to be prefix of some other longer suffix. There are N-L longer suffixes. As soon as we fix a longer suffix (denoted by P in subsequent discussion), all the remaining L characters are uniquely determined (letter by letter, starting from the first one). Not all of these N-L different candidates of P however give us a solution so there are at most N-L different trees.

** For all i <= N-L, longest common prefix (denoted as LCP from now on) of ith suffix and N-Lth suffix (denoted as LCP(i, N-L)) is an invariant. Actually from explanation [a] of a previous claim, it is easy to see that LCP(i, j) is preserved for all 1 <= i, j <= N-L. Now as we've already arranged first N-L characters so that pairwise equality of letters is maintained - we can move both the indices forward until one of them becomes N-L. More formally, if LCP(i, N-L) is preserved for all i <= N-L, we can be assured that LCP(i, j) is preserved for all i,j <= N-L.

Approach 1:

So now we have an O(N2) solution based on observations made already. Fix some i <= N-L such that ith suffix is our P. There are O(N) such different P. For a given P, find out remaining characters in O(N) time using simulation. After that run Z algorithm to find out LCP(i, N-L) for all i in our generated string in O(N) time. If all of these are same as corresponding values of S, we've found a new way. A subtle point here is to note that different candidates of P might give us same X. To avoid counting them more than once, we put all different X in a set and finally take its size. One could also put hashes of these X and put them in set instead of putting whole X itself.

Finally ans is = |# of ways of picking P that give distinct X| * 26 * 25 ... * (26-D + 1). But as this takes O(N^2) time, it is not sufficient to get Ac. We need to do make more observations now.

Denote Q = max over i <= N-L { LCP(i, N-L) } and let Kth suffix be one such suffix which achieves this values of Q. First Q-1 letters of X are uniquely determined. As from a previous claim**, LCP(K, N-L) is to be preserved across all good strings, and it is Q in S - it has to be Q in all good strings. That means we can fill up Q-1 characters after N-L letters based on next Q-1 characters of K+1th suffix.

So now we can assume that we know exactly N-L+ Q-1 characters of any good string uniquely. We've to try to fill in remaining characters.

No matter how we choose remaining characters, for all i <= N-L, LCP(i, N-L) in our string >= LCP (i, N-L) in S Choose any i <= N-L. There are two cases: Case 1 : LCP(i, N-L) < Q in S. In this case it is easy to see that LCP(i, N-L) in our string == LCP(i, N-L) in S.

Case 2 : LCP(i, N-L) == Q in S. In this case we've already ensured that LCP(i,N-L) in our string >= Q.

From this it is also clear that we have still to ensure that for all i <= N-L, LCP(i, N-L) in our string == LCP(i, N-L) in S. For this we must remember following rule: if for some i <= N-L,

LCP(i, N-L) = Q, then Qth character of suffix X should be different from Q+1th character of suffix i. So now we've set of forbidden letters which can't come in Qth position of X.

We can examine all candidates of P in O(1) time each. For any candidate of P, we've to check that its first Q-1 letters are same as those of first Q-1 letters of X that we've already fixed. This we can do either using hashing or using string algorithms like Z or KMP. Then we must check that Qth letter of P is not a forbidden letter. This is again O(1) check.

However one problem with this approach is that multiple candidates of P might give same suffix X. All such candidates of P should be counted only once.

We can use hashing to find out number of distinct X we can get in total in time O(N log N) Our plan is to find out hash of all distinct X that we can obtain and put them all in a set so as we can find out only unique strings. So now the only problem is to find out hashes quickly.

For all i <= N- 2L, we can find out hash of next L characters in constant time after precomputing sequence hash on S. For all i > N-2L, our X would contain some full periods of S[i...N-L] and one partial (probably empty) period. Length of this period denoted by len is equal to N-L-i+1. Number of full periods is L / len and partial length is L % len.

If we deal with only polynomial hashes, we can find out hash of one period in constant time (it is nothing but hash of S[i...N-L]). If hash of string A is H, then hash of AA (two copies of A is) : H * PRIMElen(A) + H.

Continuing this way we can find out hash of X in time O(L/len). Total time taken is : summation over len { L / len } which is O(L logL). As L <= N, Total time is O(N log N).

I'm repeating with emphasis, this is a fairly invovled problem and you should read editorial again and again if you don't get it and try out lot of examples by hand. You can also look at Tester/Setter solution for reference. You could also take a look at setter's O(N3) solution for a clearer understanding.

 Set by PruthvishE

Problem Setter's code :

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include<assert.h>
#define REP(i,a,b) for(i=a;i<b;i++)
#define rep(i,n) REP(i,0,n)

#define ll long long
#define ull unsigned ll

#define M 42424242
#define D 1007

char in[220000], str[220000];
int z[220000];

ull pw[220000], in_hs[220000];
ull good_hs[220000]; int good_sz;

void ullSort(ull d[],int s){int i,j;ull k,t;if(s<=1)return;k=d[0]/2+d[s-1]/
2+(d[0]%2+d[s-1]%2)/2;i=-1;j=s;for(;;){while(d[++i]<k);while(d[--j]>k);if(i
>=j)break;t=d[i];d[i]=d[j];d[j]=t;}ullSort(d,i);ullSort(d+j+1,s-j-1);}

void Z_algorithm(char str[], int len, int z[]){
  int i, k, a=0, b=0;

  REP(i,1,len){
    if(i <= b && z[i-a] < b-i+1){ z[i]=z[i-a]; continue; }
```

```
    if(i > b) a = b = i; else a = i;
    while(b < len && str[b-a]==str[b]) b++;
    z[i] = b-a; b--;
  }
}

int main(){
  int i,j,k,l,m,n;
  int T;
  ll res, mul;
  int used[300], ng_char[300];
  int suf_sz, z_mx;
  int uni_len, mx_len, repeat, rest;
  ull hs1, hs2, tmp;

  assert( scanf("%d",&T)==1 );
  assert( T<=10 );

  pw[0] = 1;
  REP(i,1,220000) pw[i] = pw[i-1] * D;

  while(T--){
    assert( scanf("%s",in) == 1 );
    n = strlen(in);
    assert( n <= 100000 );
    rep(i,n) assert( 'a'<=in[i] && in[i]<='z' );

    REP(i,'a','z'+1) used[i] = ng_char[i] = 0;
    rep(i,n) used[in[i]]=1;

    k = 0; REP(i,'a','z'+1) k += used[i];
    mul = 1; rep(i,k) mul = (mul*(26-i))%M;

    rep(i,n) str[i] = in[n-1-i];
    Z_algorithm(str, n, z);
    suf_sz = 0;
    REP(i,1,n) if(suf_sz < z[i]) suf_sz = z[i];

    k = 0;
    REP(i,n-suf_sz-1,n) str[k++] = in[i];
    str[k++] = '|';
    rep(i,n) str[k++] = in[i];
    Z_algorithm(str, k, z);
    z_mx = 1;
    REP(i,suf_sz+2,n+1) if(z_mx <  z[i]) z_mx = z[i];
    REP(i,suf_sz+2,n+1) if(z_mx == z[i]) ng_char[str[i+z[i]]] = 1;

    k = 0;
    REP(i,n-suf_sz,n) str[k++] = in[i];
    str[k++] = '|';
    rep(i,n) str[k++] = in[i];
    Z_algorithm(str, k, z);

    in_hs[0]=0;
    rep(i,n) in_hs[i+1] = in_hs[i]*D + in[i];

    good_sz = 0;
    rep(i,n-suf_sz) if(z[suf_sz+1+i] >= z_mx-1){
      mx_len = n - suf_sz - i;
      uni_len = z_mx - 1;
      repeat = uni_len / mx_len;
      rest   = uni_len % mx_len;

      if(ng_char[in[i+rest]]) continue;

      if(repeat){
        hs1 = in_hs[i+uni_len] - in_hs[i]*pw[uni_len];
        hs2 = 0;
        tmp = in_hs[i+mx_len] - in_hs[i]*pw[mx_len];
        rep(k,repeat) hs2 = hs2 * pw[mx_len] + tmp;
        hs2 = hs2 * pw[rest] + in_hs[i+rest] - in_hs[i]*pw[rest];
        if(hs1 != hs2) continue;
      }

      mx_len = n - suf_sz - i;
```

```
      uni_len = suf_sz;
      repeat = uni_len / mx_len;
      rest   = uni_len % mx_len;

      hs2 = 0;
      if(repeat){
        tmp = in_hs[i+mx_len] - in_hs[i]*pw[mx_len];
        rep(k,repeat) hs2 = hs2 * pw[mx_len] + tmp;
      }
      hs2 = hs2 * pw[rest] + in_hs[i+rest] - in_hs[i]*pw[rest];
      good_hs[good_sz++] = hs2;
/*      printf("--- %llu (%d)\n",hs2,uni_len);*/
    }

    ullSort(good_hs, good_sz);
    res = 1;
    REP(i,1,good_sz) if(good_hs[i]!=good_hs[i-1]) res++;
    res = (res * mul)%M;
    printf("%d\n",(int)res);
  }

  return 0;
}
```