ALL TRACKS > ALGORITHMS > STRING ALGORITHMS > > PROBLEM

4 LIVE EVENTS

# Cats Substrings

Attempted by: **57** / Accuracy: **60%** / ★★★★☆

Tag(s): Medium-Hard

PROBLEM **EDITORIAL** MY SUBMISSIONS ANALYTICS

This problem has several possible solutions. I'll describe one with suffix array.

Let's change problem statement a bit. We are asked about number of common substrings. Let's consider all pairs of common substrings, starting at some fixed position in some string from first set and at some fixed position in some string from second set. **Number of such pairs is equal to LCP (longest common prefix) of two corresponding suffixes** - if LCP is equal to *X*, you can pick strings of size *1, 2, 3, ..., X-1, X*, and they'll be pairwise equal, but strings of size *X+1* will not be equal, otherwise you know that LCP is at least *X+1*.

So we can say that we are asked to find sum of values of LCP over all possible pairs of positions from first and second set.

Let's **concatenate all strings from first set** using some deliminator: st1+"#"+st[2]+"#"+st[3]+"#"+... Same for strings from second set, but with a different deliminator, for example: st1+"$"+st[2]+"$"+st[3]+"$"+... For sets from sample test **we'll get ab#bab and bab$ba. It can be shown that our problem is equal to finding sum of values of LCP** for all possible pairs of positions in these two resulting strings - because of deliminators, we'll only use characters from some particular string from original set, but not from several strings at once.

It gives us **O(N^3)** solution, which can be imroved to **O(N^2)** by using ideas of dynamic programming.

We have to come up with **N*log(N)** *(or at least Nlog^2(N), if you are going to implement suffix array in naive way)* solution.

We'll use idea of **suffix array** here. If you are not familiar with it - learn it first (https://en.wikipedia.org/wiki/Suffix_array).

It is known fact that LCP of two positions of suffix array is equal to **smallest value among all LCP of adjacent positions (let's denote it as ALCP)** in part of array, bounded by two given positions. This means that if we have some value *X* fixed - two positions of suffix array have LCP at least *X* if and only if there are no ALCP values smaller than *X* in segment between these two positions; small values of ALCP are splitting array in parts.

For some fixed value of *X* we know the configuration of these parts; if we'll increase or decrease value of *X*, configuration will change if and only if there is some ALCP with such value - now this position is splitting array (or is not splitting it anymore - depending on direction in which we are changing *X*).

If we'll pick *X=INF*, then clearly every position **is in its own part** - because no pair of positions in suffix array has such a large ALCP. Now we can decrease value of X and **proceed events of form "two parts are merging into a single one"**. Key observation is - whenever we are merging two parts of array into a single part at *X=P*, we know that LCP between any position in original left part and any position in original right part **is same and equal to P** (because smallest value between them is ALCP equal to *P*).

Now let's build suffix array for **concatenation of first and second big string** (for sample we can get something like *ab#bab@bab$ba*), and keep information about number of positions from first and second string in every part of array. When merging two parts of array, we can calculate number of positions from first and second string in resulting part of array, and also how many pairs of positions from this group will have LCP equal to given value. And that's basically all information we need to find an answer.

If we'll detone size of input as N - this solution can be implemented in *O(Nlog(N))* or *O(Nlog^2(N))*, both versions should give you AC easily.

## Tester Solution

```
#include<bits/stdc++.h>
#define bs 1000000007

const int N = 100005;

using namespace std;

int n;
string st1[N];
string st2[N];
long long ans;
int m;
string Z;
int cp;
vector<int> v;
vector<pair<int, int> > events;
int l[N*5],r[N*5];
int c1[N*5],c2[N*5];
int last;
long long TTL;
long long pw[1<<21];
long long s[1<<21];

void remove_block(int l,int r)
{
        TTL-=1ll*c1[l]*c2[l];
}

void add_block(int l,int r)
{
        TTL+=1ll*c1[l]*c2[l];
}

long long eval(int ps,int span)
{
        return (s[ps+span]-s[ps])*pw[(1<<20)-ps];
}

int lcp(int a,int b)
{
        if (a<b)
                swap(a,b);
        int l,r;
        l=0;
        r=Z.size()-b;
        while (l<r)
        {
                int mid=l+r+1;
                mid/=2;
                if (eval(a,mid)==eval(b,mid))
                        l=mid;
                else
                        r=mid-1;
        }
        return l;
}

bool cmp(int a,int b)
{
        int l=lcp(a,b);
        if (a+l==Z.size())
                return true;
        if (b+l==Z.size())
                return false;
        return (Z[a+l]<Z[b+l]);
}

int sz1,sz2;

int main(){
        //freopen("beavers.in","r",stdin);
        //freopen("beavers.out","w",stdout);
        //freopen("F:/in.txt","r",stdin);
        //freopen("F:/output.txt","w",stdout);
        //ios_base::sync_with_stdio(0);
        //cin.tie(0);

        pw[0]=1;
        for (int i=1;i<=(1<<20);i++)
                pw[i]=pw[i-1]*173;

        cin>>n;
        for (int i=1;i<=n;i++)
                cin>>st1[i];
        cin>>m;
        for (int i=1;i<=m;i++)
                cin>>st2[i];

        for (int i=1;i<=n;i++)
        {
                Z+=st1[i];
                sz1+=st1[i].size();
                Z+="*";
        }

        cp=Z.size();

        for (int i=1;i<=m;i++)
        {
                Z+=st2[i];
                sz2+=st2[i].size();
                Z+="#";
        }

        for (int i=1;i<=Z.size();i++)
                s[i]=s[i-1]+Z[i-1]*pw[i];

        for (int i=0;i<Z.size();i++)
        {
                v.push_back(i);
        }


        sort(v.begin(),v.end(),cmp);

        for (int i=0;i<v.size();i++)
        {
                l[i]=i;
                r[i]=i;
                if (Z[v[i]]>='a'&&Z[v[i]]<='z')
                {
                        if (v[i]<cp)
                                c1[i]=1;
                        else
                                c2[i]=1;
                }
        }

        for (int i=0;i+1<v.size();i++)
        {
                int val=lcp(v[i],v[i+1]);
                events.push_back(make_pair(val,i));
        }

        sort(events.begin(),events.end());
        reverse(events.begin(),events.end());

        last=1e9;
        for (int i=0;i<events.size();i++)
        {
                int val=events[i].first;
                ans+=TTL*1ll*(last-val);
                last=val;
                int ps=events[i].second;
                remove_block(l[ps],ps);
                remove_block(ps+1,r[ps+1]);
                l[r[ps+1]]=l[ps];
                r[l[ps]]=r[ps+1];
                c1[l[ps]]+=c1[ps+1];
                c2[l[ps]]+=c2[ps+1];
                add_block(l[ps],r[l[ps]]);
        }

        ans+=1ll*last*TTL;
        cout<<ans<<endl;

        return 0;
}
```

---

---

## ABOUT US

Blog

Engineering Blog

Updates & Releases

Team

Careers

In the Press

## HACKEREARTH

API

Chrome Extension

CodeTable

HackerEarth Academy

Developer Profile

Resume

Get Badges

Campus Ambassadors

Get Me Hired

Privacy

Terms of Service

## DEVELOPERS

AMA

Code Monk

Judge Environment

Solution Guide

Problem Setter Guide

Practice Problems

HackerEarth Challenges

College Challenges

College Ranking

Organise Hackathon

Hackathon Handbook

Competitive Programming

Open Source

## EMPLOYERS

Developer Sourcing

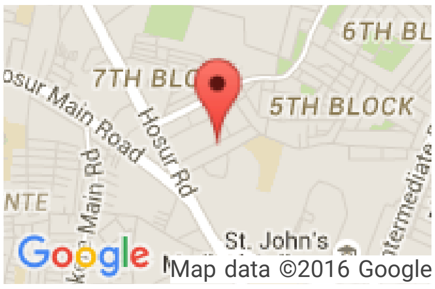
Lateral Hiring

Campus Hiring

Hackathons

FAQs

## REACH US



Customers

IIIrd Floor, Salarpuria Business Center,
4th B Cross Road, 5th A Block,
Koramangala Industrial Layout,
Bangalore, Karnataka 560095, India.

contact@hackerearth.com
+91-80-4155-4695
+1-650-461-4192