Queues and Strings

locked

by pruthvishalcodi1

Editorial by PruthvishE

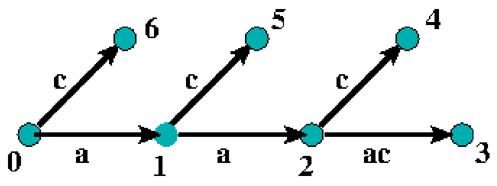
This was easily the hardest problem. The setters solution was based on Ukkonen's Algorithm. The time complexity is O(Q).

The sum of length of all edges of the suffix tree is equal to the number of distinct substrings in the string. So, If we had to merely append characters to the end of string, it can be easily done with Ukkonen itself, because we could maintain sum of length of all edges. We also need to maintain the number of leaves, because the length of each leaf edge increases by one after each step. Length of remaining edges remain same, unless the edge has been split in previous step.

Ukkonen's algorithm can be modified to handle deletions as well, and its not too hard to figure out. Here are the details.

The first step is to realize that each leaf corresponds to a unique suffix of the string, and we can do bookkeeping to maintain the leaves for each suffix. Only those suffixes whose length is equal to remainder(refer to stackoverflow article) or less have no leaves corresponding to them. The algorithm will automatically ensure that we remove only those suffixes for which a leaf exists.

Due to bookkeeping, we can get hold of the leaf corresponding to the current string. We only need to remove this leaf. We also might need to remove some internal edges that were being used only by the suffix we are removing. To see this, consider the suffix tree for string "aaac".



Suppose we remove the first 'a' from string now. We would just need to remove corresponding leaf(i.e. node 3). Now suppose we were to remove the second 'a' as well. Then we would need to remove vertex 4, as well as vertex 2, because the 1-2 edge is being used only by this suffix. In general, when removing a leaf, keep removing its parent as well if

The parent has out degree 0. And it is not the active node. To see the reason for point 2 above, consider the string "aaacaac". The active node after adding last 'c' will be node 2. Now lets say, you remove the first 'a' from the beginning. After that your suffix tree looks like:

Statistics
Difficulty: Medium

Time Complexity:	O(Q)
Required Knowledge:	Suffix-trees
Publish Date:	Jul 06 2019

Now you also want to delete the second 'a'. However, you cannot remove node 2, because the node 2 may have outdegree 0, but it is being used by the active suffix. After removal of node 4, our suffix tree looks as follows.

This example also illustrates that after removing the front character, the active point itself can become a leaf. This can be identified as active edge child of active node becoming null. In this case, you will need to establish the current active point as a leaf, and move to suffix link of active node. This will ensure a condition we had in the beginning, that "The algorithm will automatically ensure that we remove only those suffixes for which a leaf exists". This is all the high level details you need to know. Read the solutions below for details. Some of the top solutions are very neat as well and can be read.

```
OR
```

We can solve this problem by Suffix Array in O(nlogn) time.

Read all the qureys and ignore all the '-' to get the whole string, reverse it and build SA Query '+' stands for Add a Suffix longer than all the existing ones Query '-' stands for delete the last char in all the suffix We create a seg-tree to maintain the suffix, rules are:

suffixes are ordered the same as they are ordered in SA No suffix is the prefix of other suffixes suppose a suffix CUR, and its previous suffix PRE, we maintain Len(CUR) - LCP(CUR, PRE) for CUR, we call it RemL(CUR) suppose a suffix CUR, and its next suffix NXT, we maintain Len(CUR) - LCP(CUR, NXT) for CUR, we call it RemR(CUR) Let's talk about the two operations

When we add a suffix CUR, check its previous suffix and the next suffix if PRE or NXT is the prefix of CUR. If it does, delete it When deleting char from all the suffix, we obtain that all the RemL and RemR are reduced by 1.Check if any of them reduce to 0 and delete that suffix at any time Ans are the sum of all the RemL. Just maitain it in the seg tree. use seg-tree to get PRE, NXT, LCP and do each insert or reduce operation in O(logn) time. Then the problem is solved. A good implementation is needed to avoid TLE.

Set by PruthvishE

Problem Setter's code :

```
#include <vector>
#include <list>
#include <map>
#include <set>
#include <deque>
#include <queue>
#include <bitset>
#include <sstream>
#include <algorithm>
#include <functional>
#include <numeric>
#include <iostream>
#include <cstdio>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <cstring>
#include <cassert>

using namespace std;

#define forn(i, n) for(int i = 0; i < int(n); ++i)
#define for1(i, n) for(int i = 1; i <= int(n); ++i)
#define ford(i, n) for(int i = int(n) - 1; i >= 0; --i)
#define fore(i, l, r) for(int i = int(l); i < int(r); ++i)
#define forit(it, s) for(typeof((s).begin()) it = (s).begin(); it != (s).en
d(); ++it)
```

```
#define sz(v) int((v).size())
#define all(v) (v).begin(), (v).end()
#define pb push_back
#define X first
#define Y second
#define mp make_pair
template<typename T> inline T abs(T a){ return ((a < 0) ? -a : a); }
template<typename T> inline T sqr(T a){ return a * a; }

typedef long long li;
typedef long double ld;
typedef pair<int, int> pt;

const int INF = (int)1E9 + 7;
const ld EPS = 1E-9;
const ld PI = 3.1415926535897932384626433832795;

const int NMAX = 2000000;


//node of the tree in ukkonen algo
struct node{
    int l, r, par, link, dep, idx;
    map<char, int> next;

    node() {
        l = r = par = dep = 0;
        link = -1, idx = -1;
    }

    node(int l, int r, int par, int dep) : l(l), r(r), par(par), dep(dep){
        link = -1, idx = -1;
    }
};

//position on the tree, vertex and length of the path of the edge
struct position{
    int V, L, dep;
    position() {
        V = L = dep = 0;
    }
    position(int V, int L, int dep) : V(V), L(L), dep(dep){
    }
};


char s[NMAX];
int head, tail;

node t[2 * NMAX + 1];
int szt;

//returns length of the edge
int leng(int v){
    return t[v].r - t[v].l;
}

//function that adds edge to parent, creates new vertex
int add_edge_to_parent(int l, int r, int parent){
    int nidx = szt++;
    t[nidx] = node(l, r, parent, t[parent].dep + (r - l));

    if(r == NMAX){
        t[nidx].idx = NMAX - t[nidx].dep;
        t[parent].idx = max(t[parent].idx, t[nidx].idx);
    }

    return (t[parent].next[s[l]] = nidx);
}

//function that adds edge to parent, but doesn't create new vertex
//it uses vertex nidx
int add_edge_to_parent_idx(int l, int r, int parent, int nidx){
    t[nidx].l = l;
    t[nidx].r = r;
```

```
        t[nidx].par = parent;
        t[nidx].dep = t[parent].dep + (r - l);

        if(r == NMAX){
            t[nidx].idx = NMAX - t[nidx].dep;
            t[parent].idx = max(t[parent].idx, t[nidx].idx);
        }

        return (t[parent].next[s[l]] = nidx);
    }

    //split edge on two parts
    int split_edge(position pos){
        int v = pos.V, up = pos.L, down = leng(v) - up;

        if(up == 0) return v;
        if(down == 0) return t[v].par;

        int mid = add_edge_to_parent(t[v].l, t[v].l + down, t[v].par);
        t[v].l += down, t[v].par = mid;
        t[mid].next[s[t[v].l]] = v;
        t[mid].idx = t[v].idx;

        return mid;
    }

    //function that move current position in the tree by one character
    //acts like fast go down, but it doesn't know if the character can be read
    position read_char(position pos, char c){
        int v = pos.V, up = pos.L;
        if(up > 0)
            return s[t[v].r - up] == c ? position(v, up - 1, pos.dep + 1) : pos
ition(-1, -1, -1);
        else{
            int nextv = t[v].next.count(c) ? t[v].next[c] : -1;
            return nextv != -1 ? position(nextv, leng(nextv) - 1, pos.dep + 1)
: position(-1, -1, -1);
        }
    }

    //function that reads some substring from some vertex
    //this function knows that this substring can be found from the position
    position fast_go_down(int v, int l, int r){
        if(l == r) return position(v, 0, t[v].dep);
        while(true){
            v = t[v].next[s[l]];
            if(leng(v) >= r - l)
                return position(v, leng(v) - (r - l), t[v].dep - leng(v) + (r -
l));
            l += leng(v);
        }
        throw;
    }

    //function that returns suffix link from the vertex
    int link(int v){
        if(t[v].link == -1)
            t[v].link = split_edge(fast_go_down(link(t[v].par), t[v].l + int(t[
v].par == 0), t[v].r));
        return t[v].link;
    }

    //global deque of suffixes, and current sum of all edges in suffix tree
    //all suffixes stored by decreasing of their length
    deque<int> suffixes;
    li cntSubstrings;

    //add char to position in the tree
    //step of ukkonen algo
    position add_char_to_tree(position pos, int i){
        while(true){
            position npos = read_char(pos, s[i]);
            if(npos.V != -1)
                return npos;
            int mid = split_edge(pos);

            int leaf = add_edge_to_parent(i, NMAX, mid);
            suffixes.pb(leaf);
            cntSubstrings += leng(leaf);

            int nmid = link(mid);
            pos = position(nmid, 0, t[nmid].dep);

            if(mid == 0)
                return pos;
        }
        throw;
    }

    node root;
    position pos;

    //initialize of the tree
    void make_tree(){
        szt = 0; suffixes = deque<int>(); cntSubstrings = 0;
        root = node(-1, -1, -1, 0), root.link = 0, t[szt++] = root;
        pos = position(0, 0, 0);
    }

    //function that adds char to the tree (add operation)
    void add_char(char c){
        s[tail++] = c;
        pos = add_char_to_tree(pos, tail - 1);
    }

    //function that deletes inner vertex of degree 2 from the tree
    void delete_vertex(int v){
        assert(v > 0);
        assert(sz(t[v].next) == 1);

        int posdown = leng(pos.V) - pos.L;

        int par = t[v].par, npar = t[v].next.begin()->Y;
        int nleaf = t[npar].idx;

        #ifdef ssu1
        cout << "delete " << v << endl;
        cout << "leaf = " << nleaf << endl;
        #endif

        int nleng = leng(v) + leng(npar);

        if(!t[npar].next.empty()){
            add_edge_to_parent_idx(nleaf + t[t[v].par].dep, nleaf + t[t[v].par
    ].dep + nleng, par, npar);
        }else{
            add_edge_to_parent_idx(nleaf + t[t[v].par].dep, NMAX, par, npar);
        }

        if(pos.V == v){
            pos.V = npar, pos.L = leng(npar) - posdown;
        }else if(pos.V == npar){
        //    cerr << "down = " << posdown + leng(v) << endl;
            pos.L = leng(npar) - (posdown + leng(v));
        }
    }

    //returns suffix link position to position pos
    //as link but uses positions terms
    position position_link(position pos){
    //    if(pos.dep == 0) return pos;

        assert(pos.dep > 0);
        int v = pos.V, l = t[v].l, r = t[v].r - pos.L;

    //    cerr << v << " " << l << " " << r << " " << t[v].par << endl;

        position ans = fast_go_down(link(t[v].par), l + int(t[v].par == 0), r);
```

```
//    cerr << ans.V << " " << leng(ans.V) - ans.L << " " << ans.dep << end
l;

    assert(ans.dep + 1 == pos.dep);
    return ans;
}

//function that deletes largest suffix from the tree
//so it makes delete operation
void delete_char(){
    assert(head < tail);
    assert(sz(suffixes) > 0);

    int deepest = suffixes.front();

    assert(t[deepest].r == NMAX);

    if(pos.V == deepest){
        #ifdef ssu1
        cout << 'y' << endl;
        #endif

        cntSubstrings -= leng(deepest);
        pos = position_link(pos);
        int v = pos.V, up = pos.L, down = leng(v) - up;

        int cur = sz(suffixes);
        t[deepest].l = head + cur + t[t[deepest].par].dep;
        t[deepest].dep = t[t[deepest].par].dep + leng(deepest);
        t[deepest].idx = head + cur;

        if(v == deepest)
            pos.L = leng(v) - down;

        suffixes.pb(deepest);
        cntSubstrings += leng(deepest);
    }else{
        #ifdef ssu1
        cout << 'x' << endl;
        #endif

        int par = t[deepest].par;
        t[par].next.erase(s[t[deepest].l]);
        cntSubstrings -= leng(deepest);

        if(par > 0 && sz(t[par].next) == 1)
            delete_vertex(par);
    }

    head++;
    suffixes.pop_front();
}

//function that returns the number of distinct substrings
li count_distinct_substrings(){
    li ans = cntSubstrings;
//    int szs = tail - head;

    ans -= li(NMAX) * sz(suffixes);
    ans += li(tail) * sz(suffixes);

    return ans;
}

//just debug function, you can debug tree with it
void print(int v, int L){
    printf("%d (leaf = %d) (dep = %d = %d): ", v, t[v].idx, t[v].dep, L);

    assert(t[v].dep == L);

    forit(it, t[v].next){
        int u = it->Y;

        string curs;
        int l = t[u].l, r = min(tail, t[u].r);
        fore(i, l, r)
            curs += s[i];

        printf("%s (%d %d) %d; ", curs.c_str(), t[u].l, t[u].r, u);
    }
    printf("\n");

    forit(it, t[v].next){
        int u = it->Y;
        print(u, L + leng(u));
    }
}

const int mod = 1000000000 + 7;

int main() {
    #ifdef ssu1
    freopen("input.txt", "rt", stdin);
    //freopen("output.txt", "wt", stdout);
    #endif

    make_tree();

    int n;
    cin >> n;

//    print(0, 0);
    li ans = 0;
    forn(i, n){
        char type;
        scanf(" %c ", &type);

        if(type == '+'){
            char c;
            scanf(" %c ", &c);

            add_char(c);
        }else{
            assert(type == '-');

            delete_char();
        }

        #ifdef ssu1
        cout << "suffixes: " << sz(suffixes) << endl;
        forn(i, sz(suffixes))
            cout << suffixes[i] << " ";
        cout << endl;
        cout << "pos:" << pos.V << " " << leng(pos.V) - pos.L << " " << pos
.dep << endl;
        print(0, 0); printf("\n");

        cout << "substr = " << count_distinct_substrings() << endl;
        cout << endl;
        #endif

        ans = (ans + count_distinct_substrings()) % mod;
//        printf("%I64d\n", ans);
//        if(i > 5) break;
    }
    cout << ans << endl;
//    cerr << clock() << endl;
    return 0;
}
```