Code Template for ACM-ICPC

CZWin32768 @ BIT

September 20, 2018

Contents

1 Tree

1.1 Persistent-Segment-Tree

```
// calc number of different prefix in the string list [s_l, ..., s_i, ..., s_r]
#include<bits/stdc++.h>
using namespace std;

namespace Trie {
    const int SIZE = 100005;
    int node[SIZE][26];
    int tot, bel[SIZE];
    void Insert(string& str) {
        int cur = 0;
        for(int i = 0; i < str.size(); i++) {
            int p = str[i] - 'a';
            if(node[cur][p] == 0) {
                tot++;
                node[cur][p] = tot;
                memset(node[tot], 0, sizeof(node[tot]));
            }
            cur = node[cur][p];
            bel[cur] = 0;
        }
    }
    void init() {
        tot = 0;
        memset(node[0], 0, sizeof(node[0]));
    }
}

namespace PST {
    const int MAXN = 100005;
    const int M = MAXN * 40;
    int tot;
    int n;
    int T[MAXN],lson[M],rson[M],c[M];
    void init(int _n) {
        tot = 0;
        n = _n;
    }
    int build(int l,int r) {
        int root = tot++;
        c[root] = 0;
        if(l != r) {
            int mid = (l+r)>>1;
            lson[root] = build(l,mid);
            rson[root] = build(mid+1,r);
        }
        return root;
    }
    int update(int root,int pos,int val) {
        int newroot = tot++, tmp = newroot;
        c[newroot] = c[root] + val;
        int l = 1, r = n;
        while(l < r) {
            int mid = (l+r)>>1;
            if(pos <= mid) {
```

```
            lson[newroot] = tot++; rson[newroot] = rson[root];
            newroot = lson[newroot]; root = lson[root];
            r = mid;
        }
        else {
            rson[newroot] = tot++; lson[newroot] = lson[root];
            newroot = rson[newroot]; root = rson[root];
            l = mid+1;
        }
        c[newroot] = c[root] + val;
    }
    return tmp;
}
int query(int root,int pos) {
    int ret = 0;
    int l = 1, r = n;
    while(pos < r) {
        int mid = (l+r)>>1;
        if(pos <= mid) {
            r = mid;
            root = lson[root];
        }
        else {
            ret += c[lson[root]];
            root = rson[root];
            l = mid+1;
        }
    }
    return ret + c[root];
}
}
string s[PST::MAXN];

int main() {
    int N;
    while(~scanf("%d",&N)) {
        PST::init(N);
        Trie::init();
        for(int i = 1; i <= N; i++) {
            cin >> s[i];
            Trie::Insert(s[i]);
        }
        PST::T[N+1] = PST::build(1, N);
        for(int i = N; i >= 1; i--) {
            int cur = 0;
            PST::T[i] = PST::T[i+1];
            for(int j = 0; j < s[i].size(); j++) {
                int p = s[i][j] - 'a';
                cur = Trie::node[cur][p];
                if(Trie::bel[cur]) {
                    //Eliminate the influence of appeared prefix
                    PST::T[i] = PST::update(PST::T[i], Trie::bel[cur], -1);
                }
                Trie::bel[cur] = i; //record the last position of prefix
            }

            PST::T[i] = PST::update(PST::T[i],i,s[i].size());
        }
        int Q;
```

```
        scanf("%d",&Q);
        int Z = 0;
        while(Q--) {
            int l, r;
            scanf("%d%d",&l,&r);
            l += Z; l %= N;
            r += Z; r %= N;
            if(l > r) swap(l, r);
            Z = PST::query(PST::T[l+1],r+1);
            printf("%d\n",Z);
        }
    }
}
```

2 Test

2.1 test

```
#include<bits/stdc++.h>
using namespace std;

int main() {
  printf("666");
}
```

3 Number-Representation

3.1 BigDecimal

```
// methods
public static double add (double v1, double v2);
public static double sub (double v1, double v2);
public static double mul (double v1, double v2);
public static double div (double v1, double v2);
public static double div (double v1, double v2, int scale);
public static double round (double v1, double v2);

//example
double v1 = 14, v2 = 9;
BigDecimal b1 = new BigDecimal(Double.toString(v1));
BigDecimal b2 = new BigDecimal(Double.toString(v2));
BigDecimal res = b1.divide(b2, 10, BigDecimal.ROUND_HALF_UP);

/*
ROUND PROPERTIES:

ROUND_CEILING: If the BigDecimal is positive, behave as for ROUND_UP; if negative, behave as for
    ROUND_DOWN.
ROUND_DOWN: Never increment the digit prior to a discarded fraction (i.e., truncate).
ROUND_FLOOR: If the BigDecimal is positive, behave as for ROUND_DOWN; if negative behave as for
    ROUND_UP.
ROUND_HALF_DOWN: Behave as for ROUND_UP if the discarded fraction is > .5; otherwise, behave as
    for ROUND_DOWN.
```

```
ROUND_HALF_EVEN: Behave as for ROUND_HALF_UP if the digit to the left of the discarded fraction is
    odd; behave as for ROUND_HALF_DOWN if it's even.
ROUND_HALF_UP: Behave as for ROUND_UP if the discarded fraction is >= .5; otherwise, behave as for
    ROUND_DOWN.
ROUND_UNNECESSARY: This "pseudo-rounding-mode" is actually an assertion that the requested
    operation has an exact result, hence no rounding is necessary.
ROUND_UP: Always increment the digit prior to a non-zero discarded fraction.
*/
```

3.2 pydecimal

```
import decimal as D
D.getcontext().prec = 10
D.getcontext().rounding = D.ROUND_HALF_DOWN
print(D.Decimal(14) / D.Decimal("9"))
```