**动态规划**

**1.背包模板**

const int N = 110;

const int M = 100010;

const int INF = 0x3fffffff;

struct Node{

int id,val;

Node(){}

Node(int \_id,int \_v){

id = \_id; val = \_v;

}

}q[M];

int dp[M],V; ////V是所有商品的总价值

int cnt[N],v[N];

**//01背包**

void ZeroOnePack(int cost,int value)

{

for(int i = V; i >= cost; --i)

dp[i] = max(dp[i],dp[i-cost]+value);

}

**//完全背包**

void CompletePack(int cost,int value)

{

for(int i = cost; i <= V; ++i)

dp[i] = max(dp[i],dp[i-cost]+value);

}

**//多重背包**

void MultiPack(int num,int cost,int value)

{

for(int j = 0; j < cost; ++j){

int head = 1,tail = 0;

for(int k = j,i = 0; k <= V; k += cost, ++i){

int r = dp[k]-i\*value;

while(head <= tail && r >= q[tail].val) tail--;

q[++tail] = Node(i,r);

while(q[head].id < i-num) head++;

dp[k] = q[head].val+i\*value;

}

}

}

**2.数位dp**

int dp[pos][state][...];

int a[size];

int solve( int n )

{

int cnt = 0;

while( n ){

a[cnt++] = n%10;

n /= 10;

}

}

int dfs( int pos , int state , ... , int limit ) //limit为true，该数位上的数受a[i]的限制，否则取值可以为0到9

{

if( pos == -1 && ..... ) return 1;

else return 0; //符合所有要求的条件，返回1；否则返回0；

if( !limit && dp[pos][state][...] != -1 ) return dp[pos][state][...];

int up = (limit?a[pos]:9); //获取该数位的上界

int ret = 0;

for( int i = 0 ; i <= up ; ++i ){

............

ret += dfs( pos-1 , ... , ... ,limit&&(a[pos]==i) );

}

if( !limit ) dp[pos][state][...] = ret; //记忆化搜索

return ret;

}

**hdu2089:给定一个数的区间，求区间内的数不含有4和62的个数**

typedef long long ll;

int dp[20][2]; //dp[pos][state] pos为数位,state为1表示前面的那个数为6；

int a[20]; //记录需要搜索的数

int dfs( int pos , int state , int limit ) //limit为true，该数位上的数受a[i]的限制，否则取值可以为0到9

{

if( pos == -1 ) return 1;

if( !limit && dp[pos][state] != -1 ) return dp[pos][state];

int up = (limit?a[pos]:9); //获取该数位的上界

int ret = 0;

for( int i = 0 ; i <= up ; ++i ){

if( i == 4 ) continue;

if( state == 1 && i == 2 ) continue;

ret += dfs( pos-1 , i == 6 , limit&&(a[pos]==i) );

}

if( !limit ) dp[pos][state] = ret; //记忆化搜索

return ret;

}

int solve( int n )

{

int cnt = 0;

while( n ){

a[cnt++] = n%10;

n /= 10;

}

return dfs( cnt-1 , 0 , true );

}

int main()

{

int a,b;

while( ~scanf("%d%d",&a,&b) ){

memset( dp , -1 , sizeof(dp) );

if( a == 0 && b == 0 ) break;

int L1 = solve(b);

int L2 = solve(a-1);

printf("%d\n",L1-L2);

}

return 0;

}

**hdu3652：求数中出现13且是13倍数的个数**

typedef long long ll;

int dp[15][20][2][2]; //dp[pos][mod][lead][flag] pos为层次，

//mod为%13的余数,lead记录前导是否为1,flag记录是否出现连续的13

int a[15]; //记录数位的值

//lead为true表示有前导为1，flag为true表示出现了连续的13

int dfs( int pos , int mod , int lead , int flag , bool limit )

{

if( pos == -1 ){

if( !mod && flag ) return 1;

else return 0;

}

if( !limit && dp[pos][mod][lead][flag] != -1 ) return dp[pos][mod][lead][flag];

int up = ( limit ? a[pos] : 9 );

int ans = 0;

for( int i = 0 ; i <= up ; ++i ){

//int s = i\*pow( 10 , pos );

//int t = (s%13+mod)%13;

int t = (mod\*10+i)%13;

ans += dfs( pos-1 , t , i == 1 , flag || (lead&&(i==3)) , limit&&(i==a[pos]) );

}

if( !limit ) dp[pos][mod][lead][flag] = ans;

return ans;

}

int solve( int n )

{

int cnt = 0;

while( n ){

a[cnt++] = n%10;

n /= 10;

}

return dfs( cnt-1 , 0 , 0 , 0 , true );

}

int main()

{

memset(dp,-1,sizeof(dp));

int N;

while( ~scanf("%d",&N) ){

printf("%d\n",solve(N));

}

return 0;

}

**hdu2870 最大子矩阵**

const int N = 1010;

const int INF = 0x3fffffff;

struct Node{

int id,val;

Node(){}

Node(int \_id,int \_v){

id = \_id; val = \_v;

}

}Stack[N];

int H[N],L[N],R[N],M[N][N],n,m;

char data[N][N];

void deal()

{

int tail;

Stack[0] = Node(0,-1);

tail = 0;

for(int i = 0; i < m; ++i){

while(tail != 0 && H[i] <= Stack[tail].val) tail--;

L[i] = Stack[tail].id+1;

Stack[++tail] = Node(i+1,H[i]);

}

Stack[0] = Node(m+1,-1);

tail = 0;

for(int i = m-1; i >= 0; --i){

while(tail != 0 && H[i] <= Stack[tail].val) tail--;

R[i] = Stack[tail].id-1;

Stack[++tail] = Node(i+1,H[i]);

}

}

int solve()

{

memset(H,0,sizeof(H));

int ans = 0;

for(int i = 0; i < n; ++i){

for(int j = 0; j < m; ++j)

if(M[i][j] == 1) H[j]++;

else H[j] = 0;

deal();

for(int j = 0; j < m; ++j){

ans = max(ans,H[j]\*(R[j]-L[j]+1));

}

}

return ans;

}

void Change(char c1,char c2,char c3,char c4)

{

for(int i = 0; i < n; ++i){

for(int j = 0; j < m; ++j){

if(data[i][j] == c1 || data[i][j] == c2 || data[i][j] == c3 || data[i][j] == c4)

M[i][j] = 1;

else

M[i][j] = 0;

}

}

}

int main()

{

while(~scanf("%d%d",&n,&m)){

for(int i = 0; i < n; ++i) scanf(" %s",data[i]);

int ans = 0;

//全部变成‘a’

Change('a','w','y','z');

ans = max(ans,solve());

//全部变成‘b’

Change('b','w','x','z');

ans = max(ans,solve());

//全部变成‘c’

Change('c','x','y','z');

ans = max(ans,solve());

printf("%d\n",ans);

}

return 0;

}

**hdu1024 最大M子段和**

我们用dp[i][j]表示前j个数划分为k段的最大和，且以第j个数结尾。

于是有状态转移方程dp[i][j] = max(dp[i][j-1],dp[i-1][k])+data[j]  (i-1 <= k <= j-1)。

这里我们可以用滚动数组对上述方程的空间进行优化。

用r[j]表示dp[i-1][k]的最大值，dp[j]表示dp[i][j]，于是有dp[j] = max(dp[j-1],r[j-1])+data[j]。而r的值可由上一层的dp值得到。

typedef long long llt;

const int N = 1000010;

const int M = 510;

const llt INF = 0x3ffffffffffffff;

const int mod = 10000;

llt r[N],dp[N],data[N];

int main()

{

int m,n; llt c;

while(~scanf("%d%d",&m,&n)){

for(int i = 1; i <= n; ++i) scanf("%lld",&data[i]);

memset(r,0,sizeof(r));

for(int i = 1; i <= m; ++i){

c = -INF;

for(int j = i; j <= n; ++j){

dp[j] = max(dp[j-1],r[j-1])+data[j];

r[j-1] = c;

c = max(c,dp[j]);

}

}

printf("%lld\n",c);

}

return 0;

}

**字符串模板**

**1.kmp**

在目标串(长度为n)中查找子串(长度为m)

const int N = 1005;

//复杂度0(n+m)

int nt[N];

void GetNext(char\* p)

{

int pLen = strlen(p);

nt[0] = -1;

int k = -1,j = 0;

while (j < pLen - 1){

//p[k]表示前缀，p[j]表示后缀

if (k == -1 || p[j] == p[k]){

++k;

++j;

if (p[j] != p[k])

nt[j] = k; //之前只有这一行

else

//因为不能出现p[j] = p[ nt[j ]]

//所以当出现时需要继续递归，k = nt[k] = nt[nt[k]]

nt[j] = nt[k];

}else{

k = nt[k];

}

}

}

int KMP(char\* s, char\* p)

{

int i = 0,j = 0;

int sLen = strlen(s),pLen = strlen(p);

while (i < sLen && j < pLen){

//如果j = -1,或者当前字符匹配成功(即S[i] == P[j]),都令i++，j++

if (j == -1 || s[i] == p[j]){

i++;

j++;

}else{

//如果j != -1,且当前字符匹配失败(即S[i] != P[j]),则令 i 不变，j = nt[j]

//nt[j]即为j所对应的nt值

j = nt[j];

}

}

if(j == pLen)

return i - j;

else

return -1;

}

**2.Manacher**

求一个字符串中最长的回文子串。

const int N = 110100;

int len[2\*N];

char st[2\*N],data[N];

int Manacher(char\*st,int\*len){

int id = 0,mx = 0,mlen = 0,slen = strlen(st);

//mx是字符x左边的字符中以某个字符为中心的回文串的右端所能到达最远的位置

for (int i = 2; i < slen; ++i) {

if(mx > i) len[i] = min(len[2\*id-i], mx-i);

else len[i] = 1;

while(st[i-len[i]] == st[i+len[i]]) len[i]++;

if (i + len[i] >= mx) {

id = i;

mx = i+len[i];

}

mlen = max(mlen,len[i]);

}

return mlen-1;

}

int main()

{

while(~scanf(" %s",data)){

int slen = strlen(data);

//首尾插入不同字符，防止while循环时越界

//st[0] = '\*',st[2\*slen+2] = '\0';

for(int i = 0 ; i <= slen ; ++i){

st[2\*i+1] = '#';

st[2\*i+2] = data[i];

}

st[0] = '\*';

printf("%d\n",Manacher(st,len));

}

return 0;

}

**3.Trie树**

字典树的性质：

1.根结点不包含字符，其他节点都包含一个字符

2.从根结点到某个结点的字符串对应一个单词

3.每个节点的子节点的字符都不同。

用空间换时间

在字典树中查询一个单词(长度为n)的复杂度是0(n)

可以用字典树来查询，统计大量的字符串

#define STCHAR 'a'

const int maxn = 26;

struct Trie

{

bool flag; //从根到此是否为一个单词

int cnt; //记录前缀数

Trie \*next[maxn]; //有多少个分支

Trie(){

flag = false;

cnt = 0;

for(int i=0;i<maxn;i++)

next[i] = NULL;

}

};

// 声明一个根，不含任何信息

Trie \*root;

//初始化该根

void trie\_init()

{

root = new Trie();

}

// 插入一个字符串

void trie\_insert(char \*word)

{

Trie \*temp = root;

while(\*word != '\0')

{

if(temp->next[\*word-STCHAR] == NULL)// 为空才建立

{

Trie \*cur = new Trie();

temp->next[\*word-STCHAR] = cur;

}

temp = temp->next[\*word-STCHAR];

temp->cnt++;

word++;

}

temp->flag = true;//插入一个完整的单词

}

// 查找一个字符串

int trie\_search(char \*word)

{

Trie \*temp = root;

for(int i = 0; word[i] != '\0'; i++)

{

if(temp == NULL || temp->next[word[i]-STCHAR] == NULL)

return 0;

temp = temp->next[word[i]-STCHAR];

}

return temp->cnt;

}

//遍历整个字典树，寻找next指针都为空的单词个数

int ans;

void trie\_traverse(Trie \*root)

{

int flag = 0;

for( int i = 0 ; i < maxn ; ++i ){

if(root->next[i] != NULL){

flag = 1;

trie\_traverse(root->next[i]);

}

}

if(root->flag == true && flag == 0)

ans++;

}

void trie\_del(Trie \*cur)

{

for(int i = 0;i < maxn;i++)

{

if(cur->next[i] != NULL)

trie\_del(cur->next[i]);

}

delete cur;

}

**4.ac自动机**

ac自动机用来进行多个字符串的匹配，可以查找在一个文本串中存在哪些字典串。

1.建立字典树

2.在字典树上建立失败指针

3.字符串在字典树上进行匹配

**hdu2222**

#define STCHAR 'a'

const int N = 1000100;

const int maxn = 26;

char data[N];

struct Trie

{

int num;

Trie \*next[maxn]; //有多少个分支

Trie \*fail;

Trie(){

num = 0;

fail = NULL;

for(int i=0;i<maxn;i++)

next[i] = NULL;

}

};

// 声明一个根，不含任何信息

Trie \*root;

//初始化该根

void trie\_init()

{

root = new Trie();

}

// 插入一个字符串

void trie\_insert(char \*word)

{

Trie \*temp = root;

int len = strlen(word);

for(int i = 0; i < len; ++i){

int x = word[i]-'a';

if(temp->next[x] == NULL){

Trie \*cur = new Trie();

temp->next[x] = cur;

}

temp = temp->next[x];

}

temp->num++;

}

//建立失败指针

void build\_fail\_pointer()

{

Trie \*p = root;

queue<Trie\*>qu;

qu.push(p);

while(!qu.empty())

{

p = qu.front();

qu.pop();

for(int i = 0; i < maxn; i++)

{

if(p->next[i] != NULL) ///第i个孩子存在

{

if(p == root) ///p是根，根节点的孩子的失败指针都指向自己

{

p->next[i]->fail = root;

}

else

{

Trie \*node = p->fail;

while(node != NULL)

{

if(node->next[i]!=NULL)

{

p->next[i]->fail = node->next[i];

break;

}

node = node->fail;

}

if(node == NULL)

p->next[i]->fail = root;

}

qu.push(p->next[i]);

}

}

}

}

int cnt;

void ac\_automation(char \*ch)

{

Trie \*p = root;

int len = strlen(ch);

for(int i = 0; i < len; i++)

{

int x = ch[i] - STCHAR;

while(!p->next[x] && p != root) p = p->fail;

p = p->next[x];

if(!p) p = root;

Trie \*temp = p;

while(temp != root)

{

if(temp->num >= 0)

{

cnt += temp->num;

temp->num = -1;

}

else break;

temp = temp->fail;

}

}

}

///记得释放接内存，用完及时归还系统，不然会爆的。

void freeNode(Trie \*node)

{

if(node != NULL)

{

for(int i = 0; i < maxn; i++)

freeNode(node->next[i]);

}

free(node);

}

int main()

{

int T,n;

scanf("%d",&T);

while(T--){

scanf("%d",&n);

trie\_init();

for(int i = 0; i < n; ++i){

scanf(" %s",data);

trie\_insert(data);

}

scanf(" %s",&data);

build\_fail\_pointer();

cnt = 0;

ac\_automation(data);

freeNode(root);

printf("%d\n",cnt);

}

return 0;

}

**poj2775**

/\*

题目大意：一个长度为m的字符串，不包含模式串的个数。

http://blog.csdn.net/morgan\_xww/article/details/7834801

对于一个长度为x的字符串，变成x+1的字符串，有4种转移方法，在后面添加A，C，G或T。

所以我们将一个长度为0的字符串变成长度为m的字符串，在转移过程中不包含模式串，一共有多少中转移方法。

用模式串建立ac自动机。

通过自动机上结点关系得到结点间的状态转移，记录成一个邻接矩阵trie[i][j]，表示结点i到节点j的路径数。

添加一个字符相当于在图上走一步。

我们要计算从根结点走m步到合法结点的路径数，trie^m。

不合法结点：ac自动机上的结点是模式串的最后一个字符或是字符失败指针指向的结点是模式串最后一个字符。

状态转移：

if trie[u][i] = -1

trie[u][i] = trie[fail[u]][i];

\*/

typedef long long ll;

const int maxn = 105;

const int col = 4;

const int mod = 100000;

struct Matrax{

int r,c;

ll a[maxn][maxn];

Matrax(){

memset(a,0,sizeof(a));

}

};

Matrax Multi\_Matrax(Matrax x,Matrax s)

{

Matrax ans;

ans.r = x.r; ans.c = s.c;

for(int i = 0; i < ans.r; ++i){

for(int j = 0; j < ans.c; ++j){

for(int k = 0; k < ans.c; ++k){

ans.a[i][j] = (ans.a[i][j]+(x.a[i][k]\*s.a[k][j])%mod);

}

ans.a[i][j] %= mod;

}

}

return ans;

}

Matrax Fast\_Matrax(Matrax temp,ll n)

{

Matrax ans;

ans.r = ans.c = temp.r;

for(int i = 0; i < ans.r; ++i) ans.a[i][i] = 1;

while(n){

if(n&1){

ans = Multi\_Matrax(ans,temp);

}

temp = Multi\_Matrax(temp,temp);

n >>= 1;

}

return ans;

}

class AcAutomation

{

public:

int index; //结点总数,0为根节点

int fail[maxn]; //失败指针

int trie[maxn][col]; //trie tree

int tag[maxn]; //单词结尾标记

int id['Z'+1];

void init(){

index = 1;

memset(trie[0],-1,sizeof(trie[0]));

memset(tag,0,sizeof(tag));

id['A'] = 0;

id['T'] = 1;

id['C'] = 2;

id['G'] = 3;

}

//构建trie

void trie\_insert(char st[]){

int p = 0;

int len = strlen(st);

for(int i = 0; i < len; ++i){

int x = id[st[i]];

if(trie[p][x] == -1){

memset(trie[index],-1,sizeof(trie[index]));

trie[p][x] = index++;

}

p = trie[p][x];

}

tag[p] = 1;

}

//用bfs计算每个节点fail指针

void build\_fail(){

fail[0] = 0;

queue<int>q;

for(int i = 0; i < col; ++i){

if(trie[0][i] != -1){

fail[trie[0][i]] = 0;

q.push(trie[0][i]);

}else{

trie[0][i] = 0; //不能丢

}

}

while(!q.empty()){

int u = q.front();

q.pop();

if(tag[fail[u]])

tag[u] = 1;

for(int i = 0; i < col; ++i){

int &v = trie[u][i];

if(v != -1){

q.push(v);

fail[v] = trie[fail[u]][i];

}else{

//因为这些结点是通过bfs处理的，不用担心trie[fail[u]][i]值无效。

v = trie[fail[u]][i];

}

}

}

}

//根据trie树状态转移构建矩阵

Matrax buildMatrix(){

Matrax ans;

ans.r = ans.c = index;

for(int i = 0; i < index; ++i){

for(int j = 0; j < col; ++j){

if(!tag[i] && !tag[trie[i][j]])

ans.a[i][trie[i][j]]++;

}

}

return ans;

}

};

int main()

{

int n,m;

char st[20];

AcAutomation ac;

ac.init();

scanf("%d%d",&n,&m);

for(int i = 0; i < n; ++i){

scanf(" %s",st);

ac.trie\_insert(st);

}

ac.build\_fail();

Matrax re = ac.buildMatrix();

re = Fast\_Matrax(re,m);

int ans = 0;

for(int i = 0; i < re.r; ++i){

ans = (ans+re.a[0][i])%mod;

}

printf("%d\n",ans);

return 0;

}

**5.后缀自动机**

typedef long long llt;

const int N = 1010;

//后缀自动机

/\*

算法实现:

1. 令last为对应整个字符串的状态（最初last=0，在每次字符添加操作后我们都会改变last的值）。

2.建立一个新的状态cur，令len(cur)=len(last)+1，而link(cur)的值并不确定。

3. 我们最初在last，如果它没有字符c的转移，那就添加字符c的转移，指向cur，

然后走向其后缀链接，再次检查——如果没有字符c的转移，就添加上去。如果在

某个节点已有字符c的转移，就停止，并且令p为这个状态的编号。

4.如果“某节点已有字符c的转移”这一事件从未发生，而我们来到了空状态-1（经由t\_0的后缀指针前来），

我们简单地令link(cur)=0，跳出。

5.假设我们停在了某一状态q，是从某一个状态p经字符c的转移而来。现在有两种情况：len(p)+1=len(q)或不然。

6.如果len(p)+1=len(q)，那么我们简单地令link(cur)=q，跳出。

7.否则，情况就变得更加复杂。必须新建一个q的“拷贝”状态：建立一个新的状态clone，

将q的数据拷贝给它（后缀链接，以及转移），除了len的值：需要令len(clone)=len(p)+1.

8.在拷贝之后，我们将cur的后缀链接指向clone，并将q的后缀链接重定向到clone。

9.最终，我们需要做的最后一件事情就是——从p开始沿着后缀链接走，对每个状态

我们都检查是否有指向q的，字符c的转移，如果有就将其重定向至clone（如果没有，就终止循环）。

10.在任何情况下，无论在何处终止了这次添加操作，我们最后都将更新last的值，将其赋值为cur。

如果我们还需要知道哪些节点是终止节点而哪些不是，我们可以在构建整个字符串的后缀自动机之后

找出所有终止节点。对此我们考虑对应整个字符串的节点（显然，就是我们储存在变量last中的节点），

我们沿着它的后缀链接走，直到到达初始状态，并且将途径的每个节点标记为终止节点。很好理解，

如此我们标记了字符串s所有后缀的对应状态，也就是我们想要找出的终止状态。

\*/

struct state{

int len,link; //link为后缀链接，len表示结点到起始结点的最大长度

map<char,int>next; //st[0].next[c] = 1表示0结点经过状态c转移到节点1

}st[N\*2];

int sz,last;

void sa\_init()

{

sz = last = 0;

st[0].len = 0;

st[0].link = -1;

++sz;

for(int i = 0; i < N; ++i)

st[i].next.clear();

}

void sa\_extend(char c)

{

int cur = sz++; //新添加的状态结点

st[cur].len = st[last].len+1;

int p;

for(p = last; p != -1 && !st[p].next.count(c); p = st[p].link)

st[p].next[c] = cur;

if(p == -1)

st[cur].link = 0;

else{

int q = st[p].next[c];

if(st[p].len+1 == st[q].len)

st[cur].link = q;

else{

int clone = sz++;

st[clone].len = st[p].len+1;

st[clone].next = st[q].next;

st[clone].link = st[q].link;

for(; p != -1 && st[p].next[c] == q; p = st[p].link)

st[p].next[c] = clone;

st[q].link = st[cur].link = clone;

}

}

last = cur;

}

int main()

{

return 0;

}

·给定文本串T,问T中是否存在子串P

从初始状态开始转移，当前状态没有要求的字符转移，那么答案就是“no”

·给定字符串S，问它有多少不同的子串

任意一个子串对应自动机中的一条路径，答案为从初始状态开始，自动机中的路径数

因为后缀自动机是一张又向无环图，可以用dp来计算不同的路径数

dp[v]:表示从状态v出发的路径数(包括长度为0的串)

dp[v] = 1+sum(dp[u]) v-->u;

答案为dp[s]-1;

·给定字符串，求其所有不同子串的总长度

考虑两个状态：不同子串的个数d[v]和他们的总长ans[v].

ans[v] = sum(d[u]+ans[u]) v-->u;

·给定字符串s，问s的所有子串排序后第k大的

字典序第k小的子串--自动机中第k小的路径，考虑从每个状态出发的不同路径数

·给定字符串s，找到和它循环同构的字典序最小字符串

将字符串s+s建立后缀自动机。该自动机将包含和s循环同构的所有字符串

从而问题变成在自动机中找出字典序最小，长度为len(s)的路径.

·给定文本串T：问字符串p在T中出现的次数(出现次数可以相交)

计算每个状态的cnt[v](endpos(v)集合的大小)

答案为对应状态的cnt值

那么如何计算cnt

对于每个状态，若不是拷贝而来，则cnt = 1，否则cnt = 0

然后按长度len降序遍历所有序列，并将当前的cnt[v]加给后缀链接，cnt[link[v]] += cnt[u]。

**数据结构**

**1.树状数组**

#define lowbit(x) (x&-x)

const int maxn = 1000010;

int C[maxn],A[maxn];

void init(int n)

{

memset(C,0,sizeof(C));

for(int i = 1; i <= n; ++i){

for(int j = i-lowbit(i)+1; j <= i; ++j){

C[i] += A[j];

}

}

}

//sum[A1,Ax]

int sum(int x)

{

int ret = 0;

while(x){

ret += C[x];

x -= lowbit(x);

}

return ret;

}

//A[x] += d

void add(int x,int d,int n)

{

while(x <= n){

C[x] += d;

x += lowbit(x);

}

}

int main()

{

return 0;

}

**hdu1394 求逆序对**

#include <iostream>

#include <algorithm>

#include <cstdio>

#include <cstring>

using namespace std;

/\*

hdu1394 求逆序对

依次遍历A[i],只要知道前面有多少个数大于A[i],就知道前面的数相对于A[i]构成多少个逆序对m。

对于前面每个数都加上1,m = i-sum(A[i])。

而对于将序列第一个数放到最后一个,这样增加n-A[i]个逆序对,减少A[i]-1个逆序对。

\*/

#define lowbit(x) (x&-x)

const int maxn = 1000010;

int C[maxn],A[maxn];

void init(int n)

{

memset(C,0,sizeof(C));

// for(int i = 1; i <= n; ++i){

// for(int j = i-lowbit(i)+1; j <= i; ++j){

// C[i] += A[j];

// }

// }

}

//sum[A1,Ax]

int sum(int x)

{

int ret = 0;

while(x){

ret += C[x];

x -= lowbit(x);

}

return ret;

}

//A[x] += d

void add(int x,int d,int n)

{

while(x <= n){

C[x] += d;

x += lowbit(x);

}

}

int main()

{

int n;

while(~scanf("%d",&n)){

init(n);

int temp = 0;

for(int i = 1; i <= n; ++i){

scanf("%d",&A[i]);

A[i]++;

add(A[i],1,n);

temp += i-sum(A[i]);

}

int ans = temp;

for(int i = 1; i <= n; ++i){

temp = temp+(n-A[i])-(A[i]-1);

ans = min(ans,temp);

}

printf("%d\n",ans);

}

return 0;

}

**2.线段树**

const int maxn = 10010;

int A[maxn];

int Sum[maxn<<2],Add[maxn<<2]; //Sum求和,Add为懒惰标记

void init()

{

memset(Add,0,sizeof(Add));

}

//更新结点信息

void PushUp(int rt)

{

Sum[rt] = Sum[rt<<1]+Sum[rt<<1|1];

}

//建树

void Build(int l,int r,int rt)

{

if(l == r){

Sum[rt] = A[l];

return;

}

int m = (l+r)>>1;

Build(l,m,rt<<1);

Build(m+1,r,rt<<1|1);

PushUp(rt);

}

//点修改 A[L] += C;

void Update(int L,int C,int l,int r,int rt)

{

if(l == r){

Sum[rt] += C;

return;

}

int m = (l+r)>>1;

if(L <= m) Update(L,C,l,m,rt<<1);

else Update(L,C,m+1,rt<<1|1);

PushUp(rt);

}

//下推标记

void PushDown(int rt,int ln,int rn)

{

//ln,rn为左子树，右子树的数字数量

if(Add[rt]){

//下推标记

Add[rt<<1] += Add[rt];

Add[rt<<1|1] += Add[rt];

//修改子节点的Sum使之与对应的Add相对应

Sum[rt<<1] += Add[rt]\*rt;

Sum[rt<<1|1] += Add[rt]\*rn;

Add[rt] = 0;

}

}

//区间修改 A[L,R] += C;

void Update(int L,int R,int C,int l,int r,int rt)

{

if(L <= l && r <= R){

Sum[rt] += C\*(r-l+1);

Add[rt] += C;

return;

}

int m = (l+r)>>1;

PushDown(rt,m-l+1,r-m);

if(L <= m) Update(L,R,C,l,m,rt<<1);

if(R > m) Update(L,R,C,m+1,r,rt<<1|1);

PushUp(rt);

}

//查询

int Query(int L,int R,int l,int r,int rt)

{

if(L <= l && r <= R){

return Sum[rt];

}

int m = (l+r)>>1;

//下推标记

PushDown(rt,m-l+1,r-m);

int ans = 0;

if(L <= m) ans += Query(L,R,l,m,rt<<1);

if(R > m) ans += Query(L,R,m+1,r,rt<<1|1);

return ans;

}

int main()

{

return 0;

}

**3.RMQ**

const int maxn = 10010;

int A[maxn],d[maxn][32]; //d[i][j]表示从i开始,长度为2^j的一段元素中的最小值

void RMQ\_init(int n)

{

for(int i = 0; i < n; ++i) d[i][0] = A[i];

for(int j = 1; (1<<j)-1 <= n; ++j){

for(int i = 0; i+(1<<j)-1 < n; ++i){

d[i][j] = min(d[i][j-1],d[i+(1<<(j-1))][j-1]);

}

}

}

int RMQ(int L,int R)

{

int k = 0;

while((1<<(k+1)) <= R-L+1) k++;

return min(d[L][k],d[R-(1<<k)+1][k]);

}

int main()

{

return 0;

}

**4.树链剖分**

**hdu3699 点权修改**

#include <iostream>

#include <algorithm>

#include <cstdio>

#include <cstring>

using namespace std;

//#pragma comment(linker, "/STACK:1024000000,1024000000")

const int maxn = 10010;

const int N = 50010;

//hdu3966

//写得心态奔溃,写树上搜索,数组开小了,写线段树,懒惰标记写错了。

/\*

树链剖分

http://blog.csdn.net/acdreamers/article/details/10591443

\*/

struct Edge{

int node;

Edge\*next;

}edge[N\*2];

Edge\*head[N];

int Ecnt,index;

int sz[N],top[N],son[N],dep[N],fa[N],tid[N],rk[N];

void init()

{

Ecnt = index = 0;

memset(son,-1,sizeof(son));

fill(head,head+N,(Edge\*)0);

}

void mkEdge(int a,int b)

{

edge[Ecnt].node = b;

edge[Ecnt].next = head[a];

head[a] = edge+Ecnt++;

}

//记录所有重边

void dfs1(int u,int father,int d)

{

dep[u] = d;

fa[u] = father;

sz[u] = 1;

for(Edge\*p = head[u]; p; p = p->next){

int v = p->node;

if(v != father){

dfs1(v,u,d+1);

sz[u] += sz[v];

if(son[u] == -1 || sz[v] > sz[son[u]]){

son[u] = v;

}

}

}

}

//连接所有重链

void dfs2(int u,int tp)

{

top[u] = tp;

tid[u] = ++index;

rk[tid[u]] = u;

if(son[u] == -1) return;

dfs2(son[u],tp);

for(Edge\*p = head[u]; p; p = p->next){

int v = p->node;

if(v != son[u] && v != fa[u]){

dfs2(v,v);

}

}

}

int Add[N<<2],A[N<<2],data[N];

void Build(int l,int r,int rt)

{

if(l == r){

A[rt] = data[rk[l]];

return;

}

int m = (l+r)>>1;

Build(l,m,rt<<1);

Build(m+1,r,rt<<1|1);

}

void PushDown(int rt)

{

if(Add[rt]){

Add[rt<<1] += Add[rt];

Add[rt<<1|1] += Add[rt];

A[rt<<1] += Add[rt];

A[rt<<1|1] += Add[rt];

Add[rt] = 0;

}

}

void Update(int L,int R,int C,int l,int r,int rt)

{

if(L <= l && r <= R){

A[rt] += C;

Add[rt] += C;

return;

}

PushDown(rt);

int m = (l+r)>>1;

if(L <= m) Update(L,R,C,l,m,rt<<1);

if(R > m) Update(L,R,C,m+1,r,rt<<1|1);

}

int Query(int L,int l,int r,int rt)

{

if(l == r){

return A[rt];

}

PushDown(rt);

int m = (l+r)>>1;

if(L <= m) return Query(L,l,m,rt<<1);

else return Query(L,m+1,r,rt<<1|1);

}

void solve(int L,int R,int C,int l,int r,int rt)

{

int u = L,v = R,f1,f2;

while(u != v){

if(dep[u] > dep[v]){

swap(u,v);

}

f1 = top[u];

f2 = top[v];

if(f1 == f2){

Update(tid[son[u]],tid[v],C,l,r,rt);

v = u;

}else if(dep[f1] > dep[f2]){

Update(tid[f1],tid[u],C,l,r,rt);

u = fa[f1];

}else{

Update(tid[f2],tid[v],C,l,r,rt);

v = fa[f2];

}

}

Update(tid[u],tid[v],C,l,r,rt);

}

char st[10];

int main()

{

int m,n,q;

while(~scanf("%d%d%d",&n,&m,&q)){

init();

for(int i = 1; i <= n; ++i){

scanf("%d",&data[i]);

}

int a,b,c;

for(int i = 1; i <= m; ++i){

scanf("%d%d",&a,&b);

mkEdge(a,b);

mkEdge(b,a);

}

dfs1(1,0,1);

dfs2(1,1);

memset(Add,0,sizeof(Add));

memset(A,0,sizeof(A));

Build(1,n,1);

for(int i = 1; i <= q; ++i){

scanf(" %s%d",st,&a);

if(st[0] == 'I'){

scanf("%d%d",&b,&c);

solve(a,b,c,1,n,1);

}

if(st[0] == 'D'){

scanf("%d%d",&b,&c);

solve(a,b,-c,1,n,1);

}

if(st[0] == 'Q'){

printf("%d\n",Query(tid[a],1,n,1));

}

}

}

return 0;

}

**poj2763 边权修改**

#define lowbit(x) (x&-x)

//#pragma comment(linker, "/STACK:1024000000,1024000000")

//将一条边对应成一个点,利用点的深度,一条边上较深的点对应成该边

const int maxn = 10010;

const int N = 100010;

struct store{

int u,v,len;

}vec[N];

struct Edge{

int node;

Edge\*next;

}edge[N\*2];

Edge\*head[N];

int Ecnt,index;

int sz[N],top[N],son[N],dep[N],fa[N],tid[N],rk[N];

void init()

{

Ecnt = index = 0;

memset(son,-1,sizeof(son));

fill(head,head+N,(Edge\*)0);

}

void mkEdge(int a,int b)

{

edge[Ecnt].node = b;

edge[Ecnt].next = head[a];

head[a] = edge+Ecnt++;

}

void dfs1(int u,int father,int d)

{

dep[u] = d;

fa[u] = father;

sz[u] = 1;

for(Edge\*p = head[u]; p; p = p->next){

int v = p->node;

if(v != father){

dfs1(v,u,d+1);

sz[u] += sz[v];

if(son[u] == -1 || sz[v] > sz[son[u]]){

son[u] = v;

}

}

}

}

void dfs2(int u,int tp)

{

top[u] = tp;

tid[u] = ++index;

rk[tid[u]] = u;

if(son[u] == -1) return;

dfs2(son[u],tp);

for(Edge\*p = head[u]; p; p = p->next){

int v = p->node;

if(v != son[u] && v != fa[u]){

dfs2(v,v);

}

}

}

int C[N],A[N];

void add(int x,int k,int n)

{

int t = k-A[x];

A[x] = k;

while(x <= n){

C[x] += t;

x += lowbit(x);

}

}

int sum(int x,int n)

{

int ret = 0;

while(x){

ret += C[x];

x -= lowbit(x);

}

return ret;

}

int solve(int s,int e,int n)

{

int u = s,v = e,f1,f2,ans = 0;

while(u != v){

if(dep[u] > dep[v]){

swap(u,v);

}

f1 = top[u];

f2 = top[v];

if(f1 == f2){

ans += sum(tid[v],n)-sum(tid[son[u]]-1,n);

v = u;

}else if(dep[f1] > dep[f2]){

ans += sum(tid[u],n)-sum(tid[f1]-1,n);

u = fa[f1];

}else{

ans += sum(tid[v],n)-sum(tid[f2]-1,n);

v = fa[f2];

}

}

return ans;

}

int main()

{

int n,q,s;

while(~scanf("%d%d%d",&n,&q,&s)){

init();

int a,b,c;

for(int i = 0; i < n-1; ++i){

scanf("%d%d%d",&a,&b,&c);

mkEdge(a,b);

mkEdge(b,a);

vec[i].u = a; vec[i].v = b; vec[i].len = c;

}

dfs1(1,0,1);

dfs2(1,1);

memset(A,0,sizeof(A));

memset(C,0,sizeof(C));

for(int i = 0; i < n-1; ++i){

int u = vec[i].u,v = vec[i].v;

int t = (dep[u] > dep[v]) ? u : v;

add(tid[t],vec[i].len,n);

}

for(int i = 0; i < q; ++i){

scanf("%d",&a);

if(a == 1){

scanf("%d%d",&b,&c);

int u = vec[b-1].u,v = vec[b-1].v;

int t = (dep[u] > dep[v]) ? u : v;

add(tid[t],c,n);

}else{

scanf("%d",&b);

printf("%d\n",solve(s,b,n));

s = b;

}

}

}

return 0;

}

**特殊函数**

1.unique去重函数，返回去重后元素尾地址

2.next\_permutation(vec,vec+n) 对vec元素进行全排列

3.lgamma(n):求ln((n-1)!) tgamma(n):求(n-1)!

4.sscanf从一个字符串中读取数据

原形：

int sscanf(const char\*buffer,const char\*format[]...);

读取整数

int a,b,c;

char st[] = "2012:10:3";

sscanf(st,"%d:%d:%d",&a,&b,&c);

读取字符串可以使用%[]

%[a-z]:匹配a到z中的任意字符,贪婪性(尽可能多的匹配)

%[aB']:匹配a,B,',中的一员,贪婪性

%[^a]: 匹配非a的任意字符,贪婪性

%\*s:表示过滤掉这个字符串

const char\*s = "iios/12DDWDFF@122";

char buf[20];

sscanf(s,"%\*[^/]/%[^@]",buf);

buf为"12DDWDFF"

STL

1.set

set是用红黑树实现的关联式容器，不会出现重复的元素，且会对内部的元素自动排序

查找，删除，插入的复杂度都是log(n)

iterator一定不能使用过时的

begin() //返回set容器的第一个元素

end() //返回set容器的最后一个元素

clear() //删除set容器中的所有元素

empty() //判断set容器是否为空

size() //返回当前set容器中元素个数

count() //查找set中某个值出现次数，因为不是1 or 0，所有用来判断某元素是否出现在set中

erase(iterator) //删除定位器iterator指向的值

erase(first,second) //删除定位器first和second之间的值

erase(key\_value) //删除键值key\_value的值

find() //返回给定值的定位器，没有找到返回end()

insert(key\_value) //将key\_value插入到set中，返回值是pair<set<int>::iterator,bool>,bool标志着插入是否成功，而iterator代表插入值的位置，若key\_value已经在set中，则iterator表示key\_value在set中的位置

insert(first,second) //将定位器first到second之间的元素插入到set中，返回值是void

lower\_bound(key\_value) //返回第一个大于等于key\_value的定位器

upper\_bound(key\_value) //返回第一个大于key\_value的定位器

元素的遍历,使用迭代器

set<int>::iterator iter;

for(iter = st.begin(); iter != st.end(); ++iter);

自定义比较函数

set中传入结构体，可以在结构体中定义比较函数

2.map

map是一种用红黑树实现的关联式容器，实现了键值对的映射,不会出现重复的键

查找，插入，删除的时间复杂度为log(n)

声明

map<string,int>mp;

插入

1.用[]进行单个插入

mp["yuan"] = 1; //若键值"yuan"已存在，则会做赋值修改操作

2.用mp.insert(pair<string,int>);//返回pair<iterator,bool>

取值

用[],若键不存在，则会插入该键

用at,若键不存在，则会报错

容量查询

empty()

size()

count()

迭代器

begin()

end()

删除

erase(iterator) //删除迭代器指向位置的键值对，并返回一个指向下一个元素的迭代器

erase(key\_type)

clear()

查找

find(key\_type) //返回指向该键的迭代器

离散化处理：

//离散化

//对于数据少，范围大，且仅需要数据相对大小的关系

//A[n],B[n]:B是A的副本

sort(A,A+n);

int sz = unique(A,A+n)-A;

int a = lower\_bound(A,A+sz,B[i])-A; //a为B[i]的映射(范围为0~sz-1),lg(n)