Hello World ! ---线段树

/\*

线段树求逆序数 求小逆序数 神奇

题意：给定一个序列，对该序列的n种排列（排列如下）的每种排列(0 ~ n-1)的逆序数求最小值：

a1, a2, ..., an-1, an

a2, a3, ..., an, a1

a3, a4, ..., an, a1, a2

...

an, a1, a2, ..., an-1

思路：先求出初始序列的逆序数，可以归并，这里用的是线段数求。

设当前逆序数为sum，则每次把第一个数x移到最后，则新序列的逆序数 = sum - x + (n - 1 - x)

sum = sum - a[i] + (n - 1 - a[i]); //太神奇了，这个转移方程.

前面部分用线段树求初始逆序简单说一下，就是先建一棵树，每个节点[l,r]保存一个sum值，表示到目前为止[l,r]出现的个数。如当前序列为1,3, 则节点[0,3].sum = 2, [0, 4].sum = 2。然后每次扫到一个新的数x都先询问旧序列中[x+1, n-1]中出现的个数...

\*\*\* 逐个插入值（即输入的一个值）； 在每插入一个值后就更新包含该区间的所有的数的个数（加一）\*\*\*

\*/

void Update(int root)

{

if(root == 0) return ;

f[root].sum++;

Update(root >> 1);

}

int Query(int root, int l, int r)

{

if(l <= f[root].l && f[root].r <= r) {

return f[root].sum;

}

int mid = MID(f[root].l, f[root].r);

if(r <= mid) return Query(L(root), l, r);

else if(mid < l) return Query(R(root), l, r);

else return Query(L(root), l, mid) + Query(R(root), mid+1, r);

}

void build(int root, int l, int r)

{

f[root].l = l, f[root].r = r, f[root].sum = 0;

if(l == r) {

pos[l] = root;

return ;

}

int mid = MID(l, r);

build(L(root), l, mid);

build(R(root), mid+1, r);

}

int main()

{

int x;

while(scanf("%d", &n) != EOF) {

build(1, 0, n-1);

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

scanf("%d", &x);

a[i] = x;

high[x] = Query(1, x, n-1);

Update(pos[x]);

}

int sum = 0;

for(int i = 0; i < n; i++) sum += high[i];

int ans = sum;

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

sum = sum - a[i] + (n - 1 - a[i]); //太神奇了，这个转移方程.

ans = min(ans, sum);

}

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

}

}

/\*

简单线段树

描述：板子h\*w上贴1\*w0的广告，从左上角贴起，询问广告所在的row。每个广告height都为1

思路：将h分段，结构中增加w域，保存区间[a,b]所以行剩余可贴的最大量。

\*/

int Query\_and\_Update(int root, int w)

{

if(f[root].l == f[root].r) {

f[root].rest -= w;

return f[root].l;

}

int result ;

if(f[L(root)].rest >= w) result = Query\_and\_Update(L(root), w);

else result = Query\_and\_Update(R(root), w);

f[root].rest = max(f[L(root)].rest, f[R(root)].rest);

return result;

}

void build(int u, int l, int r)

{

f[u].l = l , f[u].r = r , f[u].rest = W;

if(l == r) return ;

int mid = MID(l, r);

build(L(u), l, mid);

build(R(u), mid+1, r);

}

int main()

{

int n,w;

while(scanf("%d%d%d", &H, &W, &n) != EOF) {

build(1, 1, min(n, H));

while(n--) {

scanf("%d", &w);

if(f[1].rest < w) printf("-1\n");

else printf("%d\n", Query\_and\_Update(1, w));

}

}

return 0;

}

/\*

线段树 好题 好巧妙

题意：人们按顺序来排队或插队，按顺序给出每个人要插入的位置p，即当前者要在队列里第p个人后面；还有每个人的编号。输出最后的队列的编号序列。

思路：记得以前做过一遍了，可是现在再自己做一遍还是不会。太弱了...

就不分析了。建一棵空树，每个节点保存该区间的空位。从后往前插入，p[i]的意义就变成“第i个人站的位置前面要有p[i]个空位”

\*/

void Update(int u, int x)

{

f[u].rest--;

if(f[u].l == f[u].r) {

f[u].num = y;

return ;

}

if(f[L(u)].rest > x) Update(L(u), x);

else Update(R(u), x - f[L(u)].rest);

}

void build(int u, int l, int r)

{

f[u].l = l, f[u].r = r;

f[u].rest = r - l + 1;

if(l == r) {

pos[l] = u;

return ;

}

int mid = MID(l, r);

build(L(u), l, mid);

build(R(u), mid+1, r);

}

int main()

{

int x;

while(scanf("%d", &n) != EOF) {

build(1, 1, n);

for(int i = 0; i < n; i++) scanf("%d%d", &a[i], &b[i]);

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

x = a[i], y = b[i];

Update(1, x);

}

for(int i = 1; i <= n; i++) printf("%d%c", f[pos[i]].num, i == n ? '\n' : ' ');

}

return 0;

}

/\*

线段树 约瑟夫环问题 好题 反素数

题意：N个孩子顺时针坐成一个圆圈且从1到N编号，每个孩子手中有一张标有非零整数的卡片。第K个孩子先出圈，如果他手中卡片上的数字A大于零，下一个出圈的是他左手边第A个孩子。否则，下一个出圈的是他右手边第(-A)个孩子。第p个出圈的孩子会得到F(p)个糖果，F(p)为p的因子数。求得到糖果数最多的是哪个孩子及得到多少糖果。

思路：至今仍不知道下面这个公式是怎么推出来的...

if(a[index] > 0) pos = (pos - 1 + a[index] - 1) % n + 1;

else pos = ((pos - 1 + a[index]) % n + n) % n + 1;

线段树方面跟上题一样。

http://blog.csdn.net/ahfywff/article/details/7222193

\*/

int su[37]={1,2,4,6,12,24,36,48,60,120,180,240,360,720,840,1260,1680,2520,5040,7560,10080,15120,20160,25200,27720,45360,50400,

55440,83160,110880,166320,221760,277200,332640,498960,500001};

int yinzi[37]={1,2,3,4,6,8,9,10,12,16,18,20,24,30,32,36,40,48,60,64,72,80,84,90,96,100,108,120,128,144,160,168,180,192,200,1314521};

struct Node {

int l, r, sum;

} f[MAXN \* 3];

int a[MAXN];

char name[MAXN][11];

int ans , n;

int fansushu(int n)

{

for(int i = 0; ; i++) if(su[i] > n) {

ans = yinzi[i-1];

return su[i-1];

}

}

void build(int u, int l, int r)

{

f[u].l = l, f[u].r = r, f[u].sum = r - l + 1;

if(l == r) return ;

build(L(u), l, MID(l,r));

build(R(u), MID(l,r)+1, r);

}

int Query(int root, int num)

{

f[root].sum--;

if(f[root].l == f[root].r) {

return f[root].l;

}

if(f[L(root)].sum >= num) Query(L(root), num);

else Query(R(root), num - f[L(root)].sum);

}

int main()

{

int K;

while(scanf("%d%d", &n, &K) != EOF) {

for(int i = 1; i <= n; i++) scanf("%s%d", name[i], a+i);

int p = fansushu(n);

build(1, 1, n);

int pos = K; /\* 当前环中第pos个孩子 \*/

int index ; /\* idx为当前出去的孩子的原始位置 \*/

while(p--) {

index = Query(1, pos);

n--; //不先抹消k的位置吗？ ---在Query里面做了

if(n == 0) break;

if(a[index] > 0) pos = (pos - 1 + a[index] - 1) % n + 1;

else pos = ((pos - 1 + a[index]) % n + n) % n + 1;

}

printf("%s %d\n", name[index], ans);

}

return 0;

}

/\*

线段树 离散化

题意:在墙上贴海报,海报可以互相覆盖,问最后可以看见几张海报

// \* 贴思路 \* //

思路 : 这题数据范围很大,直接搞超时+超内存,需要离散化:

离散化简单的来说就是只取我们需要的值来用,比如说区间[1000,2000],[1990,2012] 我们用不到[-∞,999][1001,1989][1991,1999][2001,2011][2013,+∞]这些值,所以我只需要1000,1990,2000,2012就够了,将其分别映射到0,1,2,3,在于复杂度就大大的降下来了

所以离散化要保存所有需要用到的值,排序后,分别映射到1~n,这样复杂度就会小很多很多

而这题的难点在于每个数字其实表示的是一个单位长度(并非一个点),这样普通的离散化会造成许多错误(包括我以前的代码,poj这题数据奇弱)

给出下面两个简单的例子应该能体现普通离散化的缺陷:

例子一:1-10 1-4 5-10

例子二:1-10 1-4 6-10

普通离散化后都变成了[1,4][1,2][3,4]

线段2覆盖了[1,2],线段3覆盖了[3,4],那么线段1是否被完全覆盖掉了呢?

例子一是完全被覆盖掉了,而例子二没有被覆盖

为了解决这种缺陷,我们可以在排序后的数组上加些处理,比如说[1,2,6,10]

如果相邻数字间距大于1的话,在其中加上任意一个数字,比如加成[1,2,3,6,7,10],然后再做线段树就好了. 关于这一点我因为没有考虑好而RE了一次

\*/

void build(int u, int l, int r)

{

f[u].l = l, f[u].r = r, f[u].label = -1;

if(l == r) {

pos[l] = u;

return ;

}

int mid = MID(l, r);

build(L(u), l, mid);

build(R(u), mid+1, r);

}

void put\_down(int u)

{

f[L(u)].label = f[R(u)].label = f[u].label;

f[u].label = -1;

}

void Update(int u, int l, int r, int color)

{

if(l == f[u].l && f[u].r == r) {

f[u].label = color;

return ;

}

if(f[u].label != -1) put\_down(u);

int mid = MID(f[u].l, f[u].r);

if(r <= mid) Update(L(u), l, r, color);

else if(mid < l) Update(R(u), l, r, color);

else {

Update(L(u), l, mid, color);

Update(R(u), mid+1, r, color);

}

}

void put\_all\_down(int u)

{

if(f[u].l == f[u].r) {

return ;

}

if(f[u].label != -1) put\_down(u);

put\_all\_down(L(u));

put\_all\_down(R(u));

}

int main()

{

int cases, n;

scanf("%d", &cases);

while(cases--) {

scanf("%d", &n);

int top = 0;

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

scanf("%d%d", &post[i][0], &post[i][1]);

p[top++] = post[i][0];

p[top++] = post[i][1];

}

sort(p, p+top);

map[p[0]] = 0;

int Index = 1;

for(int i = 1; i < top; i++)

if(p[i] == p[i-1]) continue;

else if(p[i] == p[i-1]+1) map[p[i]] = Index++;

else map[p[i]-1] = Index++, map[p[i]] = Index++;

build(1, 0, Index-1);

for(int i = 0; i < n; i++) Update(1, map[post[i][0]], map[post[i][1]], i);

put\_all\_down(1);

memset(boo, false, sizeof(bool) \* (n+1));

int ans = 0;

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

if(f[pos[i]].label != -1 && !boo[f[pos[i]].label]) {

boo[f[pos[i]].label] = true;

ans++;

}

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

}

return 0;

}

/\*

线段树 扩点 扩线 区间覆盖 好题

题意：给出n条平行于Y轴的线段(y', y'', x)，然后3条一组，问有多少组可见线段组。“可见”的定义为，两条线段能由一条水平先连接但不交于其它的线段。“可见线段组”的定义为该组内的3条线段两两可见。

心得：之前看poj\_3225，死活看不懂，跳过了。做这道题，它们的离散化方法是一样的：线段 = 一个闭区间 + 两个端点。而线段树是只存点的，为了解决这个问题我们可以把这个闭区间也化成一个点，这样就可以存入线段树里了。而线段 \* 2这一方法正好可以实现这一点。如下图，

区间[0,2]离散化后就变成这么些点:

真实的线段: [0], (0,1), [1], (1,2), [2] ...

化为线段树里对应的点: 0 1 2 3 4 ...

当然也可以用poj\_2528这样的方法来做，不过就是要处理的东西多些，麻烦些。poj\_2528的解法是应对区间过大时防止MLE/TLE的解法，跟这道题不一样，这道题的重点是同化区间和点，poj\_2528的重点是缩短区间的长度。

\*\*\* 偶数点代表点，奇数代表线段，遇到有线段类的题目（用线段树做）经常要考虑乘以2，表示浮点的线段 \*\*\*

思路：

下面摘抄自大牛们的blog:

"能明显的感觉到是区间覆盖问题了。但是有一个细节问题，就是中间的水平线不一定经过整点，所以即使这个区间的所有点都被覆盖，也不能说其就不能穿过一条线，于是，可以将所有线段的长度扩大至2倍，这样就解决了这个问题。"

"从左到右，一次对每条线段，先进行查询，看左边能看见多少条线段，然后进行覆盖，因为很明显，如果一条线段能看见另一条线段，那么这个关系必然是相互的，所以对每条线段，只需要往左看就行了"

"注意：如 样例 中 ，0 2 2,3 4 2这两条线段，可以看到 2 3之间是没有被覆盖的，但是在线段树中我们看不到这条线段，因为 变成 浮点数了，不能处理，那么我们可以将 坐标 x2，这样就变成 4 6，中间就多出一个点 5 了，就可以判断了。。

偶数点代表点，奇数代表线段，遇到有线段类的题目（用线段树做）经常要考虑乘以2，表示浮点的线段。。。poj 3225这题类似"

"注意:由于线段包括端点,如果这条直线y=c刚好穿过某条线段的端点,则情况会变得有些麻烦.可以采用这种方法来做:将上下纵坐标乘2,横坐标不变,改变原来[y,y+1)的节点存储方式,变为[y,y]式,这样,就可以简单地处理端点问题,并且它对于所有情况都有很好的效果.自己画个图就明白了."

\*/

void init()

{

memset(path, false, sizeof(path));

for(int i = 1; i <= n; i++) vis[i].clear();

}

void build(int u, int l, int r)

{

f[u].l = l, f[u].r = r, f[u].label = 0;

if(l == r) return ;

int mid = MID(l, r);

build(L(u), l, mid);

build(R(u), mid+1, r);

}

inline bool cmp(const Line & a, const Line & b)

{

return a.x < b.x;

}

void Query(int u, int l, int r, int now)

{

if(f[u].label == 0) return;

else if(f[u].label != -1) {

if(path[now][f[u].label]) return ;

path[now][f[u].label] = true;

vis[now].push\_back(f[u].label);

return ;

}

int mid = MID(f[u].l, f[u].r);

if(r <= mid) Query(L(u), l, r, now);

else if(mid < l) Query(R(u), l, r, now);

else {

Query(L(u), l, mid, now);

Query(R(u), mid+1, r, now);

}

}

void push\_down(int u)

{

f[L(u)].label = f[R(u)].label = f[u].label;

f[u].label = -1;

}

void Update(int u, int l, int r, int now)

{

if(l == f[u].l && f[u].r == r) {

f[u].label = now;

return;

}

if(f[u].label > 0) push\_down(u);

f[u].label = -1;

int mid = MID(f[u].l, f[u].r);

if(r <= mid) Update(L(u), l, r, now);

else if(mid < l) Update(R(u), l, r, now);

else {

Update(L(u), l, mid, now);

Update(R(u), mid+1, r, now);

}

}

int calc()

{

int ans = 0;

for(int x = 1; x <= n; x++)

for(int Size1 = vis[x].size(), i = 0; i < Size1; i++) {

int y = vis[x][i];

for(int Size2 = vis[y].size(), j = 0; j < Size2; j++) {

int z = vis[y][j];

if(path[x][z]) ans++;

}

}

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

// printf("# %d : ", i);

// for(int j = 0; j < vis[i].size(); j++) printf("%d ", vis[i][j]);

// printf("\n");

// }

return ans;

}

int main()

{

int cases;

scanf("%d", &cases);

while(cases--) {

scanf("%d", &n);

init();

build(1, 0, MAX\_HIGH);

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

scanf("%d%d%d", &line[i].y1, &line[i].y2, &line[i].x);

line[i].y1 <<= 1;

line[i].y2 <<= 1;

}

sort(line+1, line+n+1, cmp);

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

Query(1, line[i].y1, line[i].y2, i);

Update(1, line[i].y1, line[i].y2, i);

}

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

}

return 0;

}

/\*

区间线段树 区间覆盖 括号序列 lazy

题意：给一串括号序列，问[l,r]的括号序列是否合法。合法的定义：1,Empty 2,S合法，则(S)合法 3,A和B合法，则AB合法。有两种操作，一种一般区间修改(reverse, set)，还有Query.

思路：之前一遇到括号序列就跪，因为我到现在都搞不清楚what is regular brackets sequence...

所以在大牛blog上看到下面这句话的时候心情很激动~~from : http://hi.baidu.com/zyz913614263/item/ed68843a6839ecd2392ffa2b

\*\*\* "把左括号当做-1，右括号当做1，对于一个区间，只要从左到右的和的最大值为0，并且区间的和为0，则这个区间就是完全匹配的" \*\*\*

知道上面这个定理后，接下来都是自己写的。这道题细节还是挺多的。

(1)在Update和Query下子节点时，都要先put\_down。

(2)我的lazy标记只有一个，后来看别人的代码时发现他们用了两个，一个is\_reverse, 一个is\_set。其实可以合成一个的。

(3)细节多WA在有lazy来时和put\_down的时候的处理：保存lazy时要更新节点属性put\_self。刚开始时我把更新lazy值和更新节点属性分开了，后来发现不对，前后的lazy值会影响节点属性的更新，所以干脆都扔到change()里，方便多了。

\*\*\* "把左括号当做-1，右括号当做1，对于一个区间，只要从左到右的和的最大值为0，并且区间的和为0，则这个区间就是完全匹配的" \*\*\*

\*/

//Bracket Sequence UESTC 1546

void put\_self(int u, char op) //更新节点u

{

if(op == '(' || op == ')') {

int tmp = (op == '(' ? -1 : 1);

f[u].end = tmp \* (f[u].r - f[u].l + 1);

f[u].min = min(tmp, f[u].end);

f[u].max = max(tmp, f[u].end);

} else {

f[u].end = - f[u].end;

int tmp = f[u].max;

f[u].max = -f[u].min;

f[u].min = -tmp;

}

}

void change(int u, char arrive) //遇到修改[l,r]操作时边lazy，边更新自己的属性(max,min,end)

{

if(arrive == '(' || arrive == ')') {

put\_self(u, arrive);

f[u].lazy = arrive;

} else if(arrive == 'r') {

put\_self(u, 'r');

if(f[u].lazy == 'r') f[u].lazy = '\0';

else if(f[u].lazy == '\0') f[u].lazy = 'r';

else f[u].lazy = (f[u].lazy == ')' ? '(' : ')');

}

}

void put\_up(int u)

{

Node & ll = f[L(u)];

Node & rr = f[R(u)];

f[u].end = ll.end + rr.end;

f[u].max = max(ll.max, ll.end+rr.max);

f[u].min = min(ll.min, ll.end+rr.min);

}

void put\_down(int u)

{

int mid = MID(f[u].l, f[u].r);

Update(L(u), f[u].l, mid, f[u].lazy);

Update(R(u), mid+1, f[u].r, f[u].lazy);

f[u].lazy = '\0';

}

void build(int u, int l, int r)

{

f[u].l = l, f[u].r = r, f[u].lazy = '\0';

if(l == r) {

f[u].min = f[u].max = f[u].end = (str[l] == '(' ? -1 : 1); //小错误大烦恼，忘了初始化min...debug了好久

return ;

}

int mid = MID(l, r);

build(L(u), l, mid);

build(R(u), mid+1, r);

put\_up(u);

}

Pair Query(int u, int l, int r)

{

if(l == f[u].l && f[u].r == r) {

return Pair (f[u].end, f[u].max);

}

if(f[u].lazy != '\0') put\_down(u); //所以说，小错误有大关系...漏了判断就得debug很久

int mid = MID(f[u].l, f[u].r);

if(r <= mid) return Query(L(u), l, r);

else if(mid < l) return Query(R(u), l, r);

else {

Pair p1 = Query(L(u), l, mid) , p2 = Query(R(u), mid+1, r);

return Pair(p1.first + p2.first, max(p1.second, p1.first + p2.second));

}

}

void Update(int u, int l, int r, char op)

{

if(l == f[u].l && f[u].r == r) {

change(u, op); //把lazy的更新，max/end的更新放到change里了

return ;

}

if(f[u].lazy != '\0')put\_down(u);

int mid = MID(f[u].l, f[u].r);

if(r <= mid) Update(L(u), l, r, op);

else if(mid < l) Update(R(u), l, r, op);

else Update(L(u), l, mid, op), Update(R(u), mid+1, r, op);

put\_up(u);

}

int main()

{

int cases, Cas = 0, query, l, r;

char op[10];

scanf("%d", &cases);

while(cases--) {

printf("Case %d:\n", ++Cas);

scanf("%d%s%d", &n, str, &query);

build(1, 0, n-1);

while(query--) {

scanf("%s%d%d\n", op, &l, &r);

if(op[0] == 'q') {

Pair p = Query(1, l,r);

puts(p.first == 0 && p.second < 1 ? "YES" : "NO");

// printf("!! %d %d\n", p.first, p.second);

} else if(op[0] == 's') {

scanf("%s", str);

Update(1, l, r, str[0]);

} else Update(1, l, r, 'r');

}

puts("");

}

return 0;

}

/\*

区间线段树 lazy 延迟标记的好题

题意：求区间最长\*连续\*上升序列。两种操作：一般的询问(l,r)，要求返回该区间的LCIS(longest continuous increasing subsequence)；add操作(l,r, add\_value)

思路：很久很久以前看过解题思路了，不过要开始写的时候又想犯懒直接看题解，最后还是忍住了自己写...

主要考虑合并，父节点区间的lcis可能是两个子节点的最大lcis，也可能是两个子节点拼接时可产生一个新的lcis，即左孩子的右端比右孩子的左端小的时候。

所以我们要维护这么几个值：Seg\_tree\_Node {

l,r : 区间

l\_value, r\_value : 区间边界值

maxl : 区间最长cis的长度

l\_cis : 以左边界为起点的cis的长度

r\_cis : 以右边界为终点的cis的长度

最后有一个add，是用来搞lazy的~~

}

这里的add跟一般lazy操作的add意义没什么区别。在这里lazy能够成立的条件为，对于整个区间[l,r]，都增加add值不会改变[l,r]里任意区间的LCIS。

但是[l,r]都增加add会影响非[l,r]子集的其它区间的lcis。而这时候我们的l\_value/r\_value的作用就体现出来了，合并操作是考虑两个子节点的最大lcis，和左孩子的右端+右孩子的左端的lcis ，也就是说合并只用到了l\_value/r\_value的比较，那我们就可以维护这两个值，使得它们一直保持更新，一有add过来就更新，那么我们就可以在保持lazy延迟的基础上，又实现了合并的正确性了~

也就是说，记录区间边界值l\_value/r\_value是为了方便我们的合并操作.

其实我觉得难点就在于如何维护区间边界l\_value/r\_value保持最新，因为这点WA了好几次，代码里的注释也都是关于这一点的，也是我做这道题的收获的地方吧。在put\_down时更新子节点的l\_value/r\_value，在put\_up时更新自己的l\_value/r\_value，在lazy延迟标记的时候也要更新自己的l\_value/r\_value.

另外顺便记下put\_down和put\_up的位置，这道题我的put\_up只有Update()的最下方出现，因为对于[l,r]，只有在update其部分区间的时候需要put\_up；put\_down只出现在Query()的开头，因为Query下去需要下降lazy...

\*/

void put\_up(int u)

{

Node & a = f[u];

Node & ll = f[L(u)];

Node & rr = f[R(u)];

a.maxl = max(ll.maxl, rr.maxl);

a.l\_value = ll.l\_value + a.add, a.r\_value = rr.r\_value + a.add; //因为要保持l\_value/r\_value为最新，所以...

a.l\_cis = ll.l\_cis, a.r\_cis = rr.r\_cis;

if(ll.r\_value < rr.l\_value) {

a.maxl = max(a.maxl, ll.r\_cis + rr.l\_cis);

}

if(ll.l\_cis == ll.r - ll.l + 1 && ll.r\_value < rr.l\_value) {

a.l\_cis = ll.l\_cis + rr.l\_cis;

}

if(rr.r\_cis == rr.r - rr.l + 1 && ll.r\_value < rr.l\_value) {

a.r\_cis = ll.r\_cis + rr.r\_cis;

}

}

void put\_down(int u)

{

Node & a = f[u];

Node & ll = f[L(u)];

Node & rr = f[R(u)];

ll.add += a.add, rr.add += a.add;

ll.l\_value += a.add, ll.r\_value += a.add; //又感觉在push\_down里也要改l\_value/r\_value，这样才是最新的 ---才不是！！

rr.l\_value += a.add, rr.r\_value += a.add; //想起来了，刚才这样感觉是说要改ll/rr的l\_value值，但是之前写成改a的l\_value了...

//哈哈，改完这里就AC了~~

//ATTENTION

a.add = 0;

}

void build(int u, int l, int r)

{

Node & a = f[u];

a.l = l, a.r = r, a.add = 0;

if(l == r) {

scanf("%d", &a.r\_value);

a.l\_value = a.r\_value;

a.l\_cis = a.r\_cis = a.maxl = 1;

return ;

}

int mid = MID(l, r);

build(L(u), l, mid);

build(R(u), mid+1, r);

put\_up(u);

}

int Query(int u, int l, int r) //询问的时候不用put\_up，因为整个区间都增加add不会影响节点的属性

{

if(l == f[u].l && f[u].r == r) return f[u].maxl;

put\_down(u); //应该放前面，因为递归回去的时候会有else{..}那边比较l\_value/r\_value的值，如果不先push\_down的话 //没关系，放哪儿都一样。

int mid = MID(f[u].l, f[u].r);

if(r <= mid) return Query(L(u), l, r);

else if(mid < l) return Query(R(u), l, r);

else {

int result = max(Query(L(u), l, mid), Query(R(u), mid+1, r));

Node & a = f[L(u)];

Node & b = f[R(u)];

if(a.r\_value < b.l\_value) {

result = max(result, a.r + 1 - max(l, a.r - a.r\_cis + 1) + min(r, b.l + b.l\_cis - 1) + 1 - b.l);

}

return result;

}

}

void Update(int u, int l, int r, int add)

{

if(l == f[u].l && f[u].r == r) {

f[u].add += add;

f[u].l\_value += add, f[u].r\_value += add; //我觉得应该在这里改变l\_value/r\_value,而不是在push\_down()里，因为query的时候要比较l\_value/r\_value，所以l\_value/r\_value一定要保持最新

return ;

}

int mid = MID(f[u].l, f[u].r);

if(r <= mid) Update(L(u), l, r, add);

else if(mid < l) Update(R(u), l, r, add);

else Update(L(u), l, mid, add), Update(R(u), mid+1, r, add);

put\_up(u); //ATTENDTION

}

int main()

{

int cases, Cas = 0, query, l, r, n, add;

char op[2];

scanf("%d", &cases);

while(cases--) {

printf("Case #%d:\n", ++Cas);

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

build(1, 1, n);

while(query--) {

scanf("%s %d%d", op, &l, &r);

if(op[0] == 'q') printf("%d\n", Query(1, l, r));

else scanf("%d", &add), Update(1, l, r, add);

}

}

return 0;

}

/\*

去年+今年提交次数 = 42次 这个数字足以说明我A过了的时候的心情，而该题的rank也排在40几

题意略

总思路跟一般线段树没什么大得差别，主要是优化时间，因为操作数很大，不能每次insert的时候都释放，所以要加lazy延迟操作。

但跟一般延迟又有点不一样，一般延迟都是只记下来，Query的时候释放，但因为这题需要在某hero要升级时就要level++(why? add = level \* e)，

所以应该是当子树可以升级时就要释放。

所以优化时间的重点在判断以x为根的树有没有可以升级的？

对于每个hero，可以升级的标准是exp >= need[level+1] ,need[i]指升级到i等级需要的经验，而insert时exp += e\*level，e指题目输入的e

结合上面两式，得当 e > (need[level+1]-exp)/level + ((need[level+1]-exp)%level != 0);时该hero可以升级。

还有，释放并不能像一般线段树那样整棵树都释放，会超时，要只释放能升级的树。如x可以升级，则释放x到下一层（想象一下线段树的分层），

如果x的左儿子lc可以升级而x的右儿子rc不可以，则继续释放左儿子lc，右儿子就停在那儿不往下了。

有很多小细节，这里也记一下。

1.当节点x没有升级而做延迟时，不仅f[x].lazy要改变，f[x].exp,f[x].need也要改变，为什么？因为我们需要f[x].need来判断是否满足升级的

条件，f[x].exp时刻代表着以x为根的整棵树的最大经验值。这里面我讲讲f[x].exp的更新：f[x].exp += f[x].level\*e;

因为经验值最大的hero，其level肯定也是最大，所以f[x].level也即该hero的level，所以我们直接更新f[x].exp += f[x].level\*e是正确的

2.上面说的都是在insert()里面，update()里面我也犯了一个错误，就是没更新f[x].level，以为更新f[x].level没用，这想法错误的原因不

难得出。

3.build\_tree()一定要在输入每个等级要的经验值后才做，因为build\_tree()里面用到第二等级的值.

\*/

**int** Query**(int** x**,int** l**,int** r**)**

**{**

**if(**l **==** f**[**x**].**l **&&** f**[**x**].**r **==** r**)return** f**[**x**].**exp**;** //信不信我把这句放在上面也对？好，我试试

**if(**f**[**x**].**lazy**)**update**(**x**);**

**int** mid **=** MID**(**f**[**x**].**l**,**f**[**x**].**r**);**

**if(**r **<=** mid**)return** Query**(**L**(**x**),**l**,**r**);**

**else if** **(**mid **<** l**)return** Query**(**R**(**x**),**l**,**r**);**

**else return** MAX**(**Query**(**L**(**x**),**l**,**mid**),**Query**(**R**(**x**),**mid**+**1**,**r**));**

**}**

**void** update**(int** x**)**

**{**

**int** mid **=** MID**(**f**[**x**].**l**,**f**[**x**].**r**),**lc **=** L**(**x**),**rc **=** R**(**x**);**

**if(**f**[**x**].**lazy**){**

insert**(**lc**,**f**[**x**].**l**,**mid**,**f**[**x**].**lazy**);**

insert**(**rc**,**mid**+**1**,**f**[**x**].**r**,**f**[**x**].**lazy**);**

f**[**x**].**lazy **=** 0**;**

**}**

f**[**x**].**need **=** MIN**(**f**[**lc**].**need**,**f**[**rc**].**need**);**

f**[**x**].**exp **=** MAX**(**f**[**lc**].**exp**,**f**[**rc**].**exp**);**

f**[**x**].**level **=** MAX**(**f**[**lc**].**level**,**f**[**rc**].**level**);**

**}**

**void** insert**(int** x**,int** l**,int** r**,int** e**)**

**{**

**if(**f**[**x**].**l **==** f**[**x**].**r**){**

f**[**x**].**exp **+=** e**\***f**[**x**].**level**;**

**while(**f**[**x**].**exp **>=** need**[**f**[**x**].**level**+**1**])**f**[**x**].**level**++;**

f**[**x**].**need **= (**need**[**f**[**x**].**level**+**1**]-**f**[**x**].**exp**)/**f**[**x**].**level **+ ((**need**[**f**[**x**].**level**+**1**]-**f**[**x**].**exp**)%**f**[**x**].**level **!=** 0**);**

**return** **;**

**}**

**if(**l **==** f**[**x**].**l **&&** f**[**x**].**r **==** r**){**

f**[**x**].**need **-=** e**;**

f**[**x**].**lazy **+=** e**;**

f**[**x**].**exp **+=** e **\*** f**[**x**].**level**;**

**if(**f**[**x**].**need **<=** 0**)** update**(**x**);**

**return** **;**

**}**

**int** mid **=** MID**(**f**[**x**].**l**,**f**[**x**].**r**),**lc **=** L**(**x**),**rc **=** R**(**x**);**

**if(**f**[**x**].**lazy**){**

insert**(**lc**,**f**[**x**].**l**,**mid**,**f**[**x**].**lazy**);**

insert**(**rc**,**mid**+**1**,**f**[**x**].**r**,**f**[**x**].**lazy**);**

f**[**x**].**lazy **=** 0**;**

**}**

**if(**r **<=** mid**)**insert**(**lc**,**l**,**r**,**e**);**

**else if(**mid **<** l**)**insert**(**rc**,**l**,**r**,**e**);**

**else** **{**

insert**(**lc**,**l**,**mid**,**e**);**

insert**(**rc**,**mid**+**1**,**r**,**e**);**

**}**

update**(**x**);**

**}**

**void** build\_tree**(int** x**,int** l**,int** r**)**

**{**

f**[**x**].**l **=** l**;**f**[**x**].**r **=** r**;**

f**[**x**].**lazy **=** 0**;**

f**[**x**].**need **=** need**[**2**];**

f**[**x**].**level **=** 1**;**

f**[**x**].**exp **=** 0**;**

**if(**l **==** r**)return** **;**

**int** mid **=** MID**(**l**,**r**);**

build\_tree**(**L**(**x**),** l**,** mid**);**

build\_tree**(**R**(**x**),** mid**+**1**,** r**);**

**}**

**int** **main()**

**{**

// freopen("in.txt","r",stdin);

**int** cases**,**Cas**,**querys**,**i**,**beg**,**end**,**e**;**

**char** op**;**

scanf**(**"%d"**,&**cases**);**

Cas **=** 1**;**

**while(**cases**--)**

**{**

printf**(**"Case %d:\n"**,**Cas**++);**

scanf**(**"%d%d%d"**,&**n**,&**maxLevel**,&**querys**);**

need**[**maxLevel**+**1**] =** inf**;**

**for(**need**[**1**] =** 0**,**i **=** 2**;**i **<=** maxLevel**;**i**++)**scanf**(**"%d"**,**need**+**i**);**

build\_tree**(**1**,**1**,**n**);**

**while(**querys**--)**

**{**

**while(**op **=** getchar**(),**op **==** ' ' **||** op **==** '\n'**);**

**if(**op **==** 'W'**){**

scanf**(**"%d%d%d"**,&**beg**,&**end**,&**e**);**

insert**(**1**,**beg**,**end**,**e**);**

**}**

**else** **{**

scanf**(**"%d%d"**,&**beg**,&**end**);**

printf**(**"%d\n"**,**Query**(**1**,**beg**,**end**));**

**}**

**}**

puts**(**""**);**

**}**

**return** 0**;**

**}**

/\*

区间线段树 区间合并

题意：Hotel有N房间，两种操作：一种checkin(x)，某个团的x个人要住连续的x间房间(k,k+1,..,k+x-1)，若有合适的房间则输出最小房间编号k，否则0，表示谢绝这个团的生意。另一种checkout(x,y)，原住x,x+1,..,x+y-1的人要离开，则这些房间就空了出来，这里面可能有一些原本就是空的。

思路：经典区间线段树。维护这么几个值:

Seg\_tree\_Node {

int l, r;

int l\_rest, r\_rest; 以l为起点的连续空余房间数，以r为终点的连续空余房间数

int max, max\_pos; 区间最多的连续空余房间数，以及对应的最小编号。

int add; for lazy..\_zZ

}

Query的顺序为：<1>有合适的房间？f[1].max >= need ?

<1.5> 如果f[u].max == need 则直接返回f[u].max\_pos

<2>u的左节点l\_child, l\_child.max >= need ? 是就进左节点

<3>u的左右节点交界处？

<4>上面都不行的话就进右节点吧

养成了风格，当有lazy来时如果要保持当前节点的属性为最新，则需要更新当前节点属性，我用put\_self()。

下降lazy用put\_down()，有时在Update()时下降，有时在Query()时也要下降。

Update的最后总会有个put\_up()

\*/

void put\_up(int u)

{

Node & a = f[u];

Node & ll = f[L(u)];

Node & rr = f[R(u)];

if(ll.max >= rr.max) {

a.max = ll.max;

a.max\_pos = ll.max\_pos;

} else {

a.max = rr.max;

a.max\_pos = rr.max\_pos;

}

a.l\_rest = ll.l\_rest; //小错误大烦恼...因为一个字母拼错了，debug了好久!

a.r\_rest = rr.r\_rest;

if(ll.r\_rest != 0 && rr.l\_rest != 0) {

if(a.max < ll.r\_rest + rr.l\_rest || (a.max == ll.r\_rest + rr.l\_rest && a.max\_pos > ll.r - ll.r\_rest + 1)) {

a.max = ll.r\_rest + rr.l\_rest;

a.max\_pos = ll.r - ll.r\_rest + 1;

}

if(ll.l\_rest == ll.r - ll.l + 1) a.l\_rest = ll.l\_rest + rr.l\_rest;

if(rr.r\_rest == rr.r - rr.l + 1) a.r\_rest = rr.r\_rest + ll.r\_rest;

}

}

void build(int u, int l, int r)

{

f[u].l = l, f[u].r = r, f[u].add = 0;

if(l == r) {

f[u].max = f[u].l\_rest = f[u].r\_rest = 1;

f[u].max\_pos = l;

return ;

}

int mid = MID(l, r);

build(L(u), l, mid);

build(R(u), mid+1, r);

put\_up(u);

}

void put\_down(int u)

{

int mid = MID(f[u].l, f[u].r);

Update(L(u), f[u].l, mid, f[u].add);

Update(R(u), mid+1, f[u].r, f[u].add);

f[u].add = 0;

}

void put\_self(int u)

{

Node & a = f[u];

a.max\_pos = a.l;

a.max = a.l\_rest = a.r\_rest = (a.r - a.l + 1) \* (int)(a.add == 1);

}

void Update(int u, int l, int r, int add)

{

if(l == f[u].l && f[u].r == r) {

f[u].add = add ;

put\_self(u);

return ;

}

if(f[u].add) put\_down(u);

int mid = MID(f[u].l , f[u].r);

if(r <= mid) Update(L(u), l, r, add);

else if(mid < l) Update(R(u), l, r, add);

else Update(L(u), l, mid, add), Update(R(u), mid+1, r, add);

put\_up(u);

}

int Query(int u, int need)

{

Node & ll = f[L(u)];

Node & rr = f[R(u)];

if(f[u].max == need) return f[u].max\_pos; //MARK 呆会儿看看max的更新 //感觉这句不影响吧..

if(f[u].add) put\_down(u);

if(ll.max >= need) return Query(L(u), need);

else if(ll.r\_rest + rr.l\_rest >= need) return ll.r - ll.r\_rest + 1;

else return Query(R(u), need);

}

int main()

{

int query, n, op, need, l, r;

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

build(1, 1, n);

while(query--) {

scanf("%d", &op);

if(op == 1) {

scanf("%d", &need);

if(f[1].max < need) printf("0\n");

else {

int result = Query(1, need);

if(result) Update(1, result, result+need-1, 2);

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

}

} else {

scanf("%d%d", &l, &r);

Update(1, l, r+l-1, 1);

}

}

return 0;

**}**