**·RMQ**

**一维：**

/\*求区间最值

对于长度为n的数列A，

回答若干询问RMQ(A,i,j)(i,j<=n)，

返回数列A中下标在i,j里的最小(大）值

ST算法:online O(nlogn)的预处理，O(1)地回答每个询问。\*/

int h[MAXNUM];

int dpt[MAXNUM][20];

//int dps[MAXNUM][20];

void initRMQ(int n)

{

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

dpt[i][0]=h[i];

//dps[i][0]=h[i];

}

for(int j=1;j<20;j++){

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

dpt[i][j]=max(dpt[i][j-1],dpt[i+(1<<(j-1))][j-1]);

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

}

}

}

int query(int a,int b)

{

int k=log2(b-a+1.0);

Int ans=max(dpt[a][k],dpt[b-(1<<k)+1][k]);

//int ans=max(dpt[a][k],dpt[b-(1<<k)+1][k])-min(dps[a][k],dps[b-(1<<k)+1][k]);

return ans;

}

**二维：**

dp[row][col][i][j] 表示[row,row+2^i-1]x[col,col+2^j-1] 二维区间内的最小值

int val[maxn][maxn];

int dp[maxn][maxn][9][9];

void init()

{

for(int row = 1; row <= N; row++)

for(int col = 1; col <=M; col++)

dp[row][col][0][0] = val[row][col];

int mx = log(double(N)) / log(2.0);

int my = log(double(M)) / log(2.0);

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

{

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

{

if(i == 0 && j ==0) continue;

for(int row = 1; row+(1<<i)-1 <= N; row++)

{

for(int col = 1; col+(1<<j)-1 <= M; col++)

{

if(i == 0)//y轴二分

dp[row][col][i][j]=max(dp[row][col][i][j-1],dp[row][col+(1<<(j-1))][i][j-1]);

else//x轴二分

dp[row][col][i][j]=max(dp[row][col][i-1][j],dp[row+(1<<(i-1))][col][i-1][j]);

}

}

}

}

}

int query(int x1,int x2,int y1,int y2)

{

int kx = log(double(x2-x1+1)) / log(2.0);

int ky = log(double(y2-y1+1)) / log(2.0);

int m1 = dp[x1][y1][kx][ky];

int m2 = dp[x2-(1<<kx)+1][y1][kx][ky];

int m3 = dp[x1][y2-(1<<ky)+1][kx][ky];

int m4 = dp[x2-(1<<kx)+1][y2-(1<<ky)+1][kx][ky];

return max( max(m1,m2) , max(m3,m4));

}

**·并查集**

vector<int> father,rankk;

void init(int n)

{

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

father.push\_back(i);

rankk.push\_back(0);

}

}

void find(int x)

{

return father[x]=father[x]==x?x:find(father[x]);

/\*权值修改

if(father[x]==x) return x;

else{

int temp=father[x];

father[x]=find(father[x]);

num[x]+=num[temp]+1;

}

return father[x];

\*/

}

void merge(int x,int y)

{

int a=find(x);

int b=find(y);

if(rankk[a]<rankk[b]){

fatehr[a]=b;

}else{

father[b]=a;

if(rankk[b]==rankk[a]){

rank[a]++;

}

}

}

**·树状数组**

int num[MAXNUM];

int tree[MAXNUM];

int lowbit(int x)

{

return x&(-x);

}

void bulidtree(int n)

{

memset(tree,0,sizeof(tree));

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

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

tree[j]+=num[i];

//tree[j]=max(tree[j],num[i]);

}

}

}

void update(int i,int value);

{

while(i<=n){

tree[i]+=value;

//tree[i]=max(tree[i],value);

i+=lowbit(i);

}

}

int query(int k)

{

int sum=0;

while(k){

sum+=tree[k];

k-=lowbit(k);

}

/\*int ans=0;

while(k){

ans=max(ans,tre[k]);

k-=lowbit(k);

}

return ans;

\*/

return sum;

}

**·字典树**

int tot,root;

int child[MAXN][2/\*字符数\*/];

bool flag[MAXN];

void init(){

memset(child[0],0,sizeof(child[1]));

flag[0]=false;

root=tot=0;

}

void insert()

{

int \*cur=&root;

for(char\*p=str;\*p;p++){

cur=&child[\*cur][\*p-'0'];

if(\*cur==0){

\*cur=++tot;

memset(child[tot],0,sizeof(child[tot]));

}

flag[\*cur]=false;

}

if(!child[\*cur][0]&&!child[\*cur][1])

flag[\*cur]=true;

}

int query()

{

int ans=0;

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

if(flag[i]==true) ans++;

}

return ans;

}

**·块状链表**

1.将一个字符插入到某个位置的前面

2.询问当前位置的字符

3.删除

o(根号n)

struct data{

int s;

char a[m\*2+5];

data \*nextt;

};

data\* root;

char str[MAXM];

int len;

void init()

{

if(len==0) return;

int tot=0;

root=new(data);

root->nextt=NULL;

root->s=0;

data\* k=root;

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

k->s++;

k->a[++tot]=str[i];

if(k->s==m){

data \*t=new(data);

k->nextt=t;

t->s=0;

t->nextt=NULL;

tot=0;

k=t;

}

}

}

void \_insert(int posi,char w)

{

int temp=posi;

if(posi>len) posi=len;

if(root==NULL){

root=new(data);

root->s=1;

root->nextt=NULL;

root->a[1]=w;

}else{

data \*k=root;

while(posi>k->s&&k->nextt!=NULL)

{

posi-=k->s;

k=k->nextt;

}

k->s++;

if(temp<=len){

memcpy(k->a+posi+1,k->a+posi,sizeof(char)\*(k->s-posi+1));

k->a[posi]=w;

}else{

k->a[posi+1]=w;

}

if(k->s==2\*m){

data \*t=new(data);

t->nextt=k->nextt;

k->nextt=t;

memcpy(t->a+1,k->a+m+1,sizeof(char)\*m);

t->s=k->s=m;

}

}

}

char query(int posi)

{

data\* k=root;

while(posi>k->s&&k->nextt!=NULL){

posi-=k->s;

k=k->nextt;

}

return k->a[posi];

}

void del(int posi)

{

data \*k=root;

while(posi>k->s&&k->nextt!=NULL){

posi-=k->s;

k=k->nextt;

}

memcpy(k->a+posi,k->a+posi+1,sizeof(int)\*(k->s-posi));

k->s--;

}

void end\_(data\* k)

{

if(k->nextt!=NULL){

end\_(k->nextt);

}

delete(k);

}

**·线段树（区间更新）**

/\*

1.add c to each of Aa, Aa+1, ... , Ab. -10000 ≤ c ≤ 10000.

2.query the sum of Aa, Aa+1, ... , Ab.

\*/

long long tree[4\*MAXNUM];

long long num[MAXNUM];

long long mark[4\*MAXNUM];

long long ans=0;

void buildup(int index)

{

tree[index]=tree[index\*2]+tree[index\*2+1];

}

void pushdown(int index,long long l,long long r)

{

long long len1,len2;

if(mark[index]!=0){

if((r-l+1)%2){

len1=(r-l)/2+1;

len2=(r-l)/2;

}else{

len1=(r-l+1)/2;

len2=len1;

}

mark[index\*2]+=mark[index];

mark[index\*2+1]+=mark[index];

tree[index\*2]+=mark[index]\*len1;

tree[index\*2+1]+=mark[index]\*len2;

mark[index]=0;

}

}

void build(int index,long long l,long long r)

{

if(r==l){

tree[index]=num[l];

return;

}

long long m=(l+r)/2;

build(2\*index,l,m);

build(2\*index+1,m+1,r);

buildup(index);

}

//求区间[begin,end]的num[i]之和 [l,r]为总的区间

void query(int index,long long begin,long long end,long long l,long long r,long long &ans)

{

if(begin>r||end<l)return;

if(begin<=l&&end>=r) {ans+=tree[index];return;}

pushdown(index,l,r);

long long m=(l+r)/2;

query(index\*2,begin,end,l,m,ans);

query(index\*2+1,begin,end,m+1,r,ans);

buildup(index);

}

void change(int index,long long l,long long r,long long begin,long long end,long long num)

{

if(begin>r||end<l)return;

if(begin<=l&&end>=r){

tree[index]+=(r-l+1)\*num;

mark[index]+=num;

return;

}

pushdown(index,l,r);

long long m=(l+r)/2;

change(index\*2,l,m,begin,end,num);

change(index\*2+1,m+1,r,begin,end,num);

buildup(index);

}

**·单调栈**

int num[MAXNUM];

struct high{

int index;

long long height;

}h[MAXNUM];

void mono\_s(int n)

{

memset(num,0,sizeof(num));

long long ans=0;

h[0].height=-1;

int s=0;

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

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

h[i].height=0;

h[i].index=0;

}

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

while(num[i]<h[s].height){

ans=max(ans,(i-h[s-1].index-1)\*h[s].height);

s--;

}

s++;

h[s].height=num[i];

h[s].index=i;

}

for(int i=s;i>0;i--){

ans=max(ans,(n-h[i-1].index)\*h[i].height);

/\*这里 长度 为 n-h[i-1].index

h[i-1].index的num若比h[i].height小，那就在栈里；

若不在栈里，则一定比它大。 \*/

}

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

}

**附：  
·线段树+离散化**

/\*

HDU 2528 Mayor's posters

本题大意：给定一些海报，可能相互重叠，告诉你每个海报

的宽度（高度都一样的）和先后叠放顺序，问没有被完全盖住的有多少张？

海报最多10000张，但是墙有10000000块瓷砖长，海报不会落在瓷砖中间。

如果直接建树，就算不TLE,也会MLE。即单位区间长度太多。

其实10000张海报，有20000个点，最多有19999个区间。对各个区间编号，就是离散化。然后建数。

其实浮点数也是一样离散化的。

writer:kuangbin

\*/

#include<stdio.h>

#include<algorithm>

#include<math.h>

usingnamespace std;

constint MAXN=10010;

struct Cpost

{

int l,r;

}posters[MAXN];

int x[MAXN\*2];

int hash[10000005];

struct Node

{

int l,r;

bool bCovered;//标记是否被完全覆盖

}segTree[MAXN\*8];//这里必须开到线段数的四倍，？？

void Build(int i,int l,int r)//建立线段树

{

segTree[i].l=l;

segTree[i].r=r;

segTree[i].bCovered=false;

if(l==r)return;

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

Build(i<<1,l,mid);

Build(i<<1|1,mid+1,r);

}

bool Post(int i,int l,int r)//贴上一个好报，同时判断是否被完全覆盖

{

if(segTree[i].bCovered) returnfalse;

if(segTree[i].l==l&&segTree[i].r==r)

{

segTree[i].bCovered=true;

returntrue;

}

bool bResult;

int mid=(segTree[i].l+segTree[i].r)>>1;

if(r<=mid) bResult=Post(i<<1,l,r);

elseif(l>mid)

bResult=Post(i<<1|1,l,r);

else

{

bool b1=Post(i<<1,l,mid);

bool b2=Post(i<<1|1,mid+1,r);

bResult=b1||b2;//不能直接或上去，因为如果前面的真，后面的会不做的

}

//这个很重要，要反馈回原结点，如果左右儿子都被完全覆盖了，自然也完全覆盖了

if(segTree[i<<1].bCovered && segTree[i<<1|1].bCovered)

segTree[i].bCovered=true;

return bResult;

}

int main()

{

int T;

int i,j,k;

int n;

scanf("%d",&T);

while(T--)

{

scanf("%d",&n);

int nCount=0;

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

{

scanf("%d%d",&posters[i].l,&posters[i].r);

x[nCount++]=posters[i].l;

x[nCount++]=posters[i].r;

}

sort(x,x+nCount);//先排序

nCount=unique(x,x+nCount)-x;//合并掉相同的项

for(i=0;i<nCount;i++)

hash[x[i]]=i;

Build(1,0,nCount-1);

int res=0;

for(i=n-1;i>=0;i--)//要从上面开始看。

if(Post(1,hash[posters[i].l],hash[posters[i].r]))

res++;

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

}

return0;

}

·**二维线段树**

每次操作可以是编辑某个矩形区域，这个区域的0改为1，1改为0，每次查询只查询某一个点的值（区间更新 单点查询）

#define xlson kx<<1, xl, mid

#define xrson kx<<1|1, mid+1, xr

#define ylson ky<<1, yl, mid

#define yrson ky<<1|1, mid+1, yr

#define MAXN 1005

#define mem(a) memset(a, 0, sizeof(a))

Bool tree[MAXN<<2][MAXN<<2];

int X, N, T;

int num, X1, X2, Y1, Y2;

char ch;

void editY(int kx,int ky,int yl,int yr)

{

if(Y1<=yl && yr<=Y2)

{

tree[kx][ky] = !tree[kx][ky];

return ;

}

int mid = (yl+yr)>>1;

if(Y1 <= mid) editY(kx,ylson);

if(Y2 > mid) editY(kx,yrson);

}

void editX(int kx,int xl,int xr)

{

if(X1<=xl && xr<=X2)

{

editY(kx,1,1,N);

return ;

}

int mid = (xl+xr)>>1;

if(X1 <= mid) editX(xlson);

if(X2 > mid) editX(xrson);

}

void queryY(int kx,int ky,int yl,int yr)

{

if(tree[kx][ky]) num ++;

if(yl==yr) return ;

int mid = (yl+yr)>>1;

if(Y1 <= mid) queryY(kx,ylson);

else queryY(kx,yrson);

}

void queryX(int kx,int xl,int xr)

{

queryY(kx,1,1,N);

if(xl==xr) return ;

int mid = (xl+xr)>>1;

if(X1<= mid)queryX(xlson);

else queryX(xrson);

}

int main()

{

while(~scanf("%d", &X))while(X--)

{

mem(tree);

scanf("%d %d%\*c", &N,&T);

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

{

scanf("%c %d %d%\*c",&ch,&X1,&Y1);

if(ch == 'C')

{

scanf("%d %d%\*c", &X2, &Y2);

editX(1,1,N);

}

else

{

num = 0;

queryX(1,1,N);

if(num & 1)printf("1\n");

else printf("0\n");

}

}

if(X) printf("\n");

}

return 0;

}

**·STL**

高效的随即存取，而不在乎插入和删除的效率，使用vector   
   大量的插入和删除，而不关心随即存取，则应使用list   
   随即存取，而且关心两端数据的插入和删除，则应使用deque。

1. Deque

元素删除 尾部删除用pop\_back();头部删除用pop\_front();

元素插入：尾部插入用push\_back()，头部插入用push\_front()，其它位置插入用insert(&pos, elem)

1. Queue

默认>;若要改为<：priority\_queue<int, vector<int>, greater<int> > p;(<)

queue 的基本操作有：  
访问队首元素，如例：q.front()，即最早被压入队列的元素。  
访问队尾元素，如例：q.back()，即最后被压入队列的元素。  
判断队列空，如例：q.empty()，当队列空时，返回true。  
访问队列中的元素个数，如例：q.size()