内心深处的图书馆 题解

本题难点与坑点

以下是本题主要考查以下难点与坑点。

该列表仅用于显示笔者安排部分分的思路，不影响评测结果。

快速计算区间

内容：本题 n 值较大，不允许一个个计算。

如何向选手出示难点：明确写明数据范围。

正解对策：分块。

部分分对策：莫队。

若仅完成部分分对策，至少扣除 52 分。

若上述对策均未完成，可能导致程序运行超时，至少扣除 80 分。

快速维护贡献

内容：通过 $\log$ 级别的数据结构，快速查询每种头脑清醒度下的概率。

如何向选手出示难点：明确写明数据范围。

正解对策：树状数组或线段树。

若未完成正解对策，可能导致程序运行超时，至少扣除 72 分。

较大的值域

内容：头脑清醒度和难度值值域较大。

如何向选手出示难点：明确写明数据范围。

正解对策：树状数组：离散化；线段树：离散化或动态开点线段树。

若未完成正解对策，可能导致程序空间超限，至少扣除 56 分。

概率为 0

内容：在部分头脑清醒度下，不可能读懂全部翻开的书页。但 0 没有逆元，无法正常退回贡献。

如何向选手出示难点：提供大样例。

正解对策：封装特制数结构体，记录乘 0 的次数。

若未完成正解对策，可能导致程序持续输出 0，至少扣除 52 分或 40 分（第5,7,9号测试点不使用莫队可规避）。

具有此性质的测试点：5,7,9,15~17,19~25，样例2,样例4,样例5。

相同难度的书页

内容：使用线段树维护贡献时，部分书页难度相同，若未考虑到，进行区间修改时会导致左端点大于右端点。

如何向选手出示难点：提供大样例。

正解对策：合并这两页。

若未完成正解对策，可能导致程序运行时错误，至少扣除 40 分。

具有此性质的测试点：9,16~17,19~25，样例3,样例4,样例5。

Case 1~3

多个独立事件同时成立的概率，是这些独立事件各自单独成立的事件的积。

而针对每一本书，收到头脑清醒度之后，将所有难度值小于等于这个数的书页的概率加起来，就是能看懂这本书的概率。

这个思路明确后，我们就可以开始写出暴力代码了。

本部分分的代码为 `library_sol1.cpp` 和 `library_sol4.cpp`，其中后者预处理了前缀和，效率应该更高。~~甚至能卡过莫队的部分分~~。

更高效地处理头脑清醒度和难度值之间的关系

可以很显然地发现，头脑清醒度和难度值的具体大小是不重要的，相对大小才是重要的。这让我们很容易地想到离散化。

而我们发现，假设有这么一本书，有难度值分别为 1，3，6 的 3 页，那么，头脑清醒度为 3 到 5 之间阅读，是没有区别的。

所以，我们想到了使用线段树之类的带 log 的数据结构，维护区间乘、单点查询，这样，每当有一本新书，就花费 $O(c\log V)$ 的时间复杂度将这本书送入线段树。查询时单点查询即可。

Case 4~9

莫队。

需要注意，这个测试点使用莫队其实有一定的概率卡不过去，请注意优化常数。

本部分分的代码为 `library_sol2.cpp`。

除 0

虽然数据范围是保证了每一页被翻到的概率不为零，但是可没有保证在某个头脑清醒度下看懂一本书的概率不为零。

但书本是会更新的。无论是莫队的部分分做法，还是分块的正解，都免不了将某本书从线段树上取下，再换上新的一本书。

在取下这本书的过程，会除以原来这本书在某个头脑清醒度下贡献的概率，当这个概率为 0 时就会出问题了。

解决这个问题的方案是定义特制数结构体，记录乘 0 的次数，然后除 0 时减掉这个次数即可。输出答案时，先判断乘 0 多少次，再选择输出 0 或原数据。

```
struct Num  //特制数结构体
{
    long long data; //原数据
    int cnt0;       //乘零次数
    Num()   //初始化
    {
        data=1;
        cnt0=0;
    }
    Num(long long v)
    {
        if(v==0)
        {
            data=1;
            cnt0=1;
        }
        else
        {
            data=v;
            cnt0=0;
        }
    }
    Num(long long v,int c)
    {
        data=v;
        cnt0=c;
    }
    operator long long()    //转long long
    {
        if(cnt0>0)
        {
            return 0;
        }
        else
```

```
        {
            return data;
        }
    }
};
Num operator *(Num a,Num b) //自定义运算符
{
    return (Num){a.data*b.data%MOD,a.cnt0+b.cnt0};
}
Num operator /(Num a,Num b)
{
    return (Num){a.data*qpow(b.data,MOD-2)%MOD,a.cnt0-b.cnt0};
}
```

相同的难度

对于一本书而言，按难度排序之后，如果头脑清醒度在相邻页码的难度之间（含左端点，不含右端点），那么看懂这本书的概率不变。因此，我们可以使用线段树区间修改。

但是，如果相邻页码难度相同，这个区间会出现 $l > r$ 的情况。在线段树中，这会导致递归不正确。~~笔者就是因为没考虑到类似情况，在广州市赛 2024 D2T3 中挂掉 90 分的~~。

因此，这需要特别处理，合并难度相同的书页。

正解：分块+动态开点线段树

这里将讲解易于理解的做法，动态开点线段树。它的缺点在于，常数极大，空间不稳定（至少要用 600MB）。

这里只放关键代码。

定义常量：

```
const int MAXN=5e4,MAXM=5e4,MAXC=4e5+1;
const int MIND=0,MAXD=1e9;
const long long MOD=998244353;
```

我们定义 `Book` 结构体。它代表了一本书的信息。

我们想想这个结构体，除了输入数据，要包含什么。

很显然，有时候我们要单独查询一本书的情况，因此我们将输入数据转换成前缀和，在查询时根据前缀和二分。

定义结构体：

```
struct Book
{
    int c;
    vector<pair<int,long long>>pages;   //输入数据
    vector<pair<int,long long>>pfx;     //前缀和，意义为，达到第一个数的难度，看懂这本书的概率为第二个数
```

定义输入方法：

```
    void read_book()
    {
        vector<pair<int,long long>>repeat_pages;    //可能重复难度的书页
        read(c);        //输入页数
        for(int i=0;i<c;i++)
        {
            int d;
            read(d);    //输入各页码难度
            repeat_pages.push_back(make_pair(d,0));
        }
        for(int i=0;i<c;i++)
        {
            read(repeat_pages[i].second);   //输入各页码概率
        }
        sort(repeat_pages.begin(),repeat_pages.end());  //排序以准备去重
        pages.clear();
        pages.reserve(c);   //不加这句可能会未知原因地 Segment Fault，或许申请空间太多次会出错
        for(auto page:repeat_pages) //去重
        {
            if(pages.empty())
            {
                pages.push_back(page);
            }
            else if(pages[int(pages.size())-1].first==page.first)
            {
                pages[int(pages.size())-1].second=(pages[int(pages.size())-1].second+page.second)%MOD;
            }
            else
            {
                pages.push_back(page);
            }
        }
```

输入完后，计算前缀和，并处理询问：

```
        long long nowsum=0; //计算前缀和
        pfx.clear();
        pfx.push_back(make_pair(0,0ll));    //这里加个0，这样查询时就不用特判，降低代码复杂度
        for(int i=0;i<(int)pages.size();i++)
```

```
        {
            nowsum=(nowsum+pages[i].second)%MOD;
            if(i==(int(pages.size())-1)||(pages[i].first!=pages[i+1].first))
            {
                pfx.push_back(make_pair(pages[i].first,nowsum));
            }
        }
    }
    long long get_probability(int x)    //当头脑清醒度为 x 时，有多大的概率看懂这本书？
    {
        vector<pair<int,long long>>::iterator
it=upper_bound(pfx.begin(),pfx.end(),make_pair(x,(long long)0x3fffffffffff));
        it--;
        return it->second;
    }
}a[MAXN+5];
```

特制数部分已在前面给出，这里不再赘述。

接下来，我们可以对于每 $\sqrt{n}$ 本书分块，每块储存块内的所有书合并的概率。

然后，每个分块建一棵线段树。

```
struct Tree
{
    int lc,rc;      //左右儿子
    Num lazy,val;   //当这个节点没有左右儿子时，val 即为这个区间下所有头脑清醒度对应的概
率；若有左右儿子，则 val 无意义。
}tree[30000000];    //这个大小很难定，定太高爆空间，定太低会 RE
int pcnt;   //动态开点的点数
int rt[250];    //每一块对应哪棵线段树
```

接下来，定义一些基本的函数：

```
void handle(int o,Num v)    //本节点修改操作统一封装为handle
{
    tree[o].lazy=(tree[o].lazy*v);
    tree[o].val=(tree[o].val*v);
}
int new_node(int o,int tl,int tr)//新建一个节点。这个函数用于减少之后代码中的判断。
{
    if(o)   //已有节点
    {
        return o;
    }
    o=++pcnt;
    tree[o].lc=tree[o].rc=0;
    tree[o].lazy=Num();
    tree[o].val=Num();
    return o;
}
void pushdown(int o,int tl,int tr)  //下传标记函数
```

```
{
    if(tl==tr)
    {
        return;
    }
    int mid=(tl+tr)>>1;
    tree[o].lc=new_node(tree[o].lc,tl,mid);
    tree[o].rc=new_node(tree[o].rc,mid+1,tr);
    if(tree[o].lazy!=1)
    {
        handle(tree[o].lc,tree[o].lazy);
        handle(tree[o].rc,tree[o].lazy);
        tree[o].lazy=Num();
    }
}
```

定义修改和查询函数：

```
void change(int o,int l,int r,Num v,int tl,int tr)
{
    if(tl==l&&tr==r)
    {
        handle(o,v);
        return;
    }
    pushdown(o,tl,tr);
    int mid=(tl+tr)>>1;
    if(mid>=r)
    {
        change(tree[o].lc,l,r,v,tl,mid);
    }
    else if(mid<l)
    {
        change(tree[o].rc,l,r,v,mid+1,tr);
    }
    else
    {
        change(tree[o].lc,l,mid,v,tl,mid);
        change(tree[o].rc,mid+1,r,v,mid+1,tr);
    }
}
Num query(int o,int x,int tl,int tr)
{
    if(tl==tr||(tree[o].lc==0&&tree[o].rc==0))//已到最深的子节点
    {
        return tree[o].val;
    }
    pushdown(o,tl,tr);
    int mid=(tl+tr)>>1;
    if(mid>=x)
    {
        return query(tree[o].lc,x,tl,mid);
```

```
    }
    else
    {
        return query(tree[o].rc,x,mid+1,tr);
    }
}
```

建好线段树我们得用它，所以定义将一本书加入线段树、从线段树中删除的函数：

```
int BLOCK;  //块长
void add_book(int id)   //将一本书加入线段树中
{
    int bid=id2bid[id]; //这本书所在的分块
    int last=MIND;
    long long nowsum=0;
    for(pair<int,long>page:a[id].pages)
    {
        change(bid,last,page.first-1,Num(nowsum),MIND,MAXD);
        nowsum=(nowsum+page.second)%MOD;
        last=page.first;
    }
    //最后不用考虑 last，因为 nowsum=1 时，任何数乘 1 都为 1，不必修改
}
void del_book(int id)   //让一本书离开线段树，几乎是上面的镜像
{
    int bid=id2bid[id];
    int last=MIND;
    long long nowsum=0;
    for(pair<int,long>page:a[id].pages)
    {
        change(bid,last,page.first-1,Num(1)/Num(nowsum),MIND,MAXD);
        nowsum=(nowsum+page.second)%MOD;
        last=page.first;
    }
}
```

之后是主函数部分。核心都写好了，主函数也不难了。

```
int main()
{
    freopen("library.in","r",stdin);
    #ifndef debug
    freopen("library.out","w",stdout);
    #endif
    //scanf("%d",&n);
    read(n);
    BLOCK=sqrt(n);
    for(int i=1;i<=BLOCK;i++)
    {
        id2bid[i]=1;
    }
```

```
    for(int i=BLOCK+1;i<=n;i++) //计算id2bid的算法，无需使用除法
    {
        id2bid[i]=id2bid[i-BLOCK]+1;
    }
    for(int i=1;i<=n;i++)
    {
        a[i].read_book();
    }
    for(int i=1;i<=id2bid[n];i++)
    {
        rt[i]=new_node(rt[i],MIND,MAXD);
    }
    for(int i=1;i<=n;i++)
    {
        add_book(i);
    }
    read(m);
    for(int qid=1;qid<=m;qid++)
    {
        int op;
        scanf("%d",&op);
        if(op==2)
        {
            int x;
            read(x);
            del_book(x);
            a[x].read_book();
            add_book(x);
        }
        else
        {
            int l,r,v;
            read(l);read(r);read(v);
            Num res=Num();
            int lbid=id2bid[l];
            int rbid=id2bid[r];
            if(lbid==rbid)
            {
                if((lbid-1)*BLOCK+1==l&&min(rbid*BLOCK,n)==r)   //l 到 r 恰好在一个整块
中
                {
                    res=query(rt[lbid],v,MIND,MAXD);
                }
                else
                {
                    for(int i=l;i<=r;i++)   //散块
                    {
                        res=res*Num(a[i].get_probability(v));
                    }
                }
            }
            else
            {
```

```
                if((lbid-1)*BLOCK+1==l) //l 恰好在整块中
                {
                    res=res*query(rt[lbid],v,MIND,MAXD);
                }
                else
                {
                    for(int i=l;i<=lbid*BLOCK;i++)
                    {
                        res=res*Num(a[i].get_probability(v));
                    }
                }
                if(min(rbid*BLOCK,n)==r)    //r 恰好在整块中
                {
                    res=res*query(rt[rbid],v,MIND,MAXD);
                }
                else
                {
                    for(int i=(rbid-1)*BLOCK+1;i<=r;i++)
                    {
                        res=res*Num(a[i].get_probability(v));
                    }
                }
                for(int i=lbid+1;i<=rbid-1;i++) //计算中间的整块
                {
                    res=res*query(rt[i],v,MIND,MAXD);
                }
            }
            write((long long)res);
            putchar('\n');
        }
    }
    return 0;
}
```

完整代码为 `library_sol5.cpp`。

离散化

动态开点线段树倒是不用考虑值域问题，但为了给接下来的优化打基础，还是离散化一下为好。

只需要离散化书本的难度值即可，询问时的头脑清醒度可以用二分，找到离散化数据列表中最后一个小于等于头脑清醒度的难度值。

一个技巧，`lower_bound` 是“第一个小于等于”，`upper_bound` 是“第一个大于”，那么 `upper_bound-1` 就是“最后一个小于等于”。

树状数组

我们知道，前缀和和差分互为逆运算。

因此，我们可以把上面应该放在线段树上的内容，转换为前缀积之后放在常数更小的树状数组上。

但这样依然容易爆空间。因为动态开点线段树遵循按需取用的原则，但树状数组的空间复杂度与自身长度成正比。

好在，一个线段树节点，要存储左节点、右节点、`lazy` 和维护的值（两个 `Num` 结构体），而树状数组只用存储一个 `Num` 结构体。

解决方案有两种，一种是拉长分块长度以减少树状数组数量，另一种如下文所介绍。

各分块独立的离散化

我们发现，开那么大的树状数组，大部分是浪费空间。因为题目中保证的数据范围 C 是所有书本（无论在哪个块）的页数总和。

所以，我们可以在离散化时区分各分块，让各个分块之间的离散化完全独立。

计算完离散化后，再利用 `vector` 的 `resize` 方法，按需构建树状数组，大幅减少空间和时间。

```
//离散化
int dcnt[250];  //按照分块进行离散化
int data[250][MAXC+5];
//在 oiClass 上，如果没有访问这块区域，不会计算空间。
//但正赛按照申请的空间计算，换句话说这里会用掉 381MB。
//当然解决方案也不难，用 vector 即可。
void add_data(int id,int x)
{
    data[id][++dcnt[id]]=x;
}
void init_data()
{
    for(int i=1;i<=249;i++)
    {
        if(dcnt[i]==0)  //每一块至少有一个数（因为主程序那边把 0 也加了进来）
        {
            return;
        }
        sort(data[i]+1,data[i]+1+dcnt[i]);
        dcnt[i]=unique(data[i]+1,data[i]+1+dcnt[i])-data[i]-1;
    }
}
int get_dataid(int id,int x)
{
    return lower_bound(data[id]+1,data[id]+1+dcnt[id],x)-data[id];
}
```

完整代码为 `library_sol3.cpp`。

后记

本题改编自笔者参加过的一场比赛。那场比赛的这个原题，我场上没能做出来~~，因此我决定把这道题放在这里让大家感受感受~~。

可以发现我的代码写得很长，这是笔者的代码风格特点：

1. 基本不在同一行写多句代码；
2. 大括号独占一行；
3. 即使循环或条件代码块只有一行，也会写大括号；

这份代码很多地方都做了封装并重复使用专用代码的处理，笔者觉得这样便于调试。

本题虽然严格来讲不是大模拟，但是需要在意的细节不少。考虑到这点，笔者决定把这道题放在 T4。

题解写得挺长的，估计和这个有的一拼了。

谢谢你看到这里。

版权信息