Educational Codeforces Round 92

比赛链接

D. Segment Intersections

题意

给定两种线段 $[l_a,r_a],[l_b,r_b]$ □每种线段有 n 条，记为 $a_1,a_2\cdots a_n$ 和 $b_1,b_2\cdots b_n$□

每次操作可以任选一条线段 $[l,r]$□将其变换成 $[l-1,r]$ 或 $[l,r+1]$□

要求输出最小操作步数，使得 $\sum_{i=1}^n f(a_i,b_i)\ge k$□其中 $f(a,b)$ 表示线段 a,b 相交部分长度。

题解 1

暴力枚举 i□只考虑前 i 对线段求出最小答案，时间复杂度 $O(n)$□

```
int main()
{
    int t=read_int();
    while(t--){
        LL n=read_int(),k=read_int(),l1=read_int(),r1=read_int(),l2=read_int(),r2=read_int(),ans=1e10;
        LL tot=max(r1,r2)-min(l1,l2),d=max(l1,l2)-min(r1,r2);
        _rep(i,1,n){
            if(tot*i>=k)
            ans=min(ans,d*i+k);
            else
            ans=min(ans,(tot+d)*i+(k-tot*i)*2);
        }
        ans=max(ans,0LL);
        enter(ans);
    }
    return 0;
}
```

题解 2

分类讨论，时间复杂度 $O(1)$□

```
int main()
{
    int t=read_int();
    while(t--){
        LL
n=read_int(),k=read_int(),l1=read_int(),r1=read_int(),l2=read_int(),r2=read
_int();
        if(l1>l2)
        swap(l1,l2),swap(r1,r2);
        if(r1>=r2){
            k-=n*(r2-l2);
            r1-=(r2-l2);
            r2=l2;
            swap(r1,r2);
        }
        else if(r1>=l2){
            k-=n*(r1-l2);
            swap(r1,l2);
            r2-=l2-r1;
            l2=r1;
        }
        if(k<=0){
            puts("0");
            continue;
        }
        int cost=2*(l2-r1),d=l2-r1,len1=r1-l1,len2=r2-l2;
        if(len1+len2+d==0){
            enter(k*2);
            continue;
        }
        if(k>=n*(len1+len2+d)){
            k-=n*(len1+len2+d);
            enter(n*(len1+len2+cost)+2*k);
            continue;
        }
        int ks=k/(len1+len2+d);
        k%=(len1+len2+d);
        if(ks>0)
        enter(ks*(len1+len2+cost)+min(2*k,d+k));
        else
        enter(d+k);
    }
    return 0;
}
```

E. Calendar Ambiguity

题意

求满足同余方程 $xd+y\equiv yd+x\pmod w$，其中 $1\le x,y\le n$。

题解

移项，得 $d(x-y)\equiv 0\pmod w$，记 $w'=\frac w{(w,d)}$，于是问题等价于 $w'\mid (x-y)$。

考虑枚举 $(x-y)=iw'$，于是满足条件的解个数为 $$\sum_{i=1}^{\lfloor\frac n{w'}\rfloor}n-iw'=n\lfloor\frac n{w'}\rfloor-\frac {\lfloor\frac n{w'}\rfloor(1+\lfloor\frac n{w'}\rfloor)}2\lfloor\frac n{w'}\rfloor w$$

时间复杂度 $O(\log(\min(w,d)))$。

```
int main()
{
    int t=read_int();
    while(t--){
        int m=read_int(),d=read_int(),w=read_int(),n,k;
        w/=__gcd(w,d-1);
        n=min(m,d),k=n/w;
        enter(1LL*n*k-1LL*(1+k)*k*w/2);
    }
    return 0;
}
```

F. Bicolored Segments

题意

给定 n 条线段 $[l_i,r_i]$，每种线段有黑白两种颜色。要求选出尽量多的线段，使得异色线段不相交。

题解 1

考虑将异色相交的线段间连一条线，则原题转化为二分图，答案等价于最大独立集。

而二分图的最大独立集等价于总节点点数减去二分图最大匹配，于是问题转化为求而二分图最大匹配。

读入所有线段，把每条线段拆成 l_i 加入和 r_i 删除两个事件。

把所有事件按 x 轴位置排序，如果处于 x 轴同一位置，则加入事件优先于删除事件。按顺序扫描 x 轴并处理事件。

扫描过程中维护两种颜色的集合，每种颜色的集合维护扫描到当前位置时已经加入且未被删除的线段，且集合中线段按右端点从小到大排序。

扫描到加入事件，则将该事件对应线段加入对应颜色的集合。如果扫描到删除事件且该事件对应线段已经配对则跳过。

否则考虑从异色集合中找到一条线段和该线段配对，显然配对的线段右端点越小越优。

考虑该贪心策略的正确性，假设现在考虑的线段是 u_1，被配对的线段是 v_1。

情况一，假设更优策略中 u_1 未配对，则令 u_1 与 v_1 配对，最多导致原来与 v_1 配对的点无法配对，答案不变，假设不成立。

情况二，假设更优策略中 v_1 未配对，则令 u_1 与 v_1 配对，最多导致原来与 u_1 配对的点无法配对，答案不变，假设不成立。

情况三，假设更优策略中 u_1 与 v_2 配对，u_2 与 v_1 配对。

于是根据假设有 u_1 删除前 v_2 已经加入，v_1 删除前 u_2 已加入。(这里删除指遇到删除事件)

根据当前决策方式有 u_1 删除前 u_2 未删除，v_1 删除前 v_2 未删除。

于是 u_2 与 v_2 可以配对，且如果该两点配对，则答案不变，假设不成立。

于是无论如何该贪心算法总是成立。

```
const int MAXN=2e5+5;
int lef[MAXN],rig[MAXN],c[MAXN];
struct Event{
    int x,v,id;
    bool operator < (const Event &b)const{
        return x<b.x||(x==b.x&&v>b.v);
    }
}a[MAXN<<1];
set<pair<int,int> > s[2];
int main()
{
    int n=read_int();
    _for(i,0,n){
        lef[i]=read_int(),rig[i]=read_int(),c[i]=read_int()-1;
        a[i]=Event{lef[i],1,i};
        a[i+n]=Event{rig[i],-1,i};
    }
    sort(a,a+2*n);
    int ans=n;
    _for(i,0,2*n){
        int pos=a[i].id;
        if(a[i].v>0)
        s[c[pos]].insert(make_pair(rig[pos],pos));
        else{
            if(s[c[pos]].find(make_pair(rig[pos],pos))!=s[c[pos]].end()){
```

```
                s[c[pos]].erase(make_pair(rig[pos],pos));
                if(s[c[pos]^1].begin()!=s[c[pos]^1].end()){
                    s[c[pos]^1].erase(s[c[pos]^1].begin());
                    ans--;
                }
            }
        }
    }
    enter(ans);
    return 0;
}
```

题解 2

首先离散化坐标。

考虑 dp设 $dp(c,pos)$ 表示 pos 位置颜色为 c 时前 pos 个位置中最多可以选择多少条线段。

$w(c,l,r)$ 表示区间 $[l,r]$ 中包含多少颜色为 c 的线段。对颜色为 c 的线段 $[l,r]$可以得到状态转移方程

$$dp(c,r)=\max_{0\le i\le l-1}\left(dp(c\oplus 1,i)+w(c,i+1,r)\right)$$

考虑线段树动态维护 $dp(c\oplus 1,i-1)+w(c,i,pos)$

具体得，将每种颜色的线段按 r 为第一关键字l 为第二关键字排序，按顺序计算每个线段答案 cur 并更新贡献。

每个颜色为 c 的线段 $[l,r]$ 的贡献为 $w(c,i,r)\gets 1(0 \le i\le l),dp(c,r)=\max(dp(c,r),cur)$

于是颜色为 c 的线段树维护每个区间 $i\in [l,r]$ 中 $dp(c,i-1)+w(c\oplus 1,i,pos)$ 的最大值即可。

时间复杂度 $O(n\log n)$

```
const int MAXN=2e5+5;
struct Node{
    int lef,rig;
    bool operator < (const Node &b)const{
        return rig<b.rig||(rig==b.rig&&lef<b.lef);
    }
}node1[MAXN],node2[MAXN];
struct Tree{
    int s[MAXN<<3],lef[MAXN<<3],rig[MAXN<<3],lazy[MAXN<<3];
    void build(int k,int L,int R){
        lef[k]=L,rig[k]=R;
        if(L==R)return;
        int M=L+R>>1;
        build(k<<1,L,M);
        build(k<<1|1,M+1,R);
    }
    void push_up(int k){
```

```
        s[k]=max(s[k<<1],s[k<<1|1]);
    }
    void push_down(int k){
        if(lazy[k]){
            s[k<<1]+=lazy[k],lazy[k<<1]+=lazy[k];
            s[k<<1|1]+=lazy[k],lazy[k<<1|1]+=lazy[k];
            lazy[k]=0;
        }
    }
    void add(int k,int L,int R){
        if(L<=lef[k]&&rig[k]<=R){
            s[k]++,lazy[k]++;
            return;
        }
        push_down(k);
        int mid=lef[k]+rig[k]>>1;
        if(mid>=L)
        add(k<<1,L,R);
        if(mid<R)
        add(k<<1|1,L,R);
        push_up(k);
    }
    void update(int k,int pos,int v){
        if(lef[k]==rig[k]){
            s[k]=max(s[k],v);
            return;
        }
        push_down(k);
        int mid=lef[k]+rig[k]>>1;
        if(mid>=pos)
        update(k<<1,pos,v);
        else
        update(k<<1|1,pos,v);
        push_up(k);
    }
    int query(int k,int L,int R){
        if(L<=lef[k]&&rig[k]<=R)
        return s[k];
        push_down(k);
        int mid=lef[k]+rig[k]>>1,ans=0;
        if(mid>=L)
        ans=max(ans,query(k<<1,L,R));
        if(mid<R)
        ans=max(ans,query(k<<1|1,L,R));
        return ans;
    }
}tree1,tree2;
int x[MAXN<<1];
int main()
{
```

```
    int n=read_int(),n1=0,n2=0,l,r,c;
    _for(i,0,n){
        l=read_int(),r=read_int(),c=read_int();
        if(c==1)node1[n1++]=Node{l,r};
        else node2[n2++]=Node{l,r};
        x[i]=l,x[i+n]=r;
    }
    sort(x,x+2*n);
    int m=unique(x,x+2*n)-x;
    _for(i,0,n1)
    node1[i].lef=lower_bound(x,x+m,node1[i].lef)-
x+1,node1[i].rig=lower_bound(x,x+m,node1[i].rig)-x+1;
    _for(i,0,n2)
    node2[i].lef=lower_bound(x,x+m,node2[i].lef)-
x+1,node2[i].rig=lower_bound(x,x+m,node2[i].rig)-x+1;
    sort(node1,node1+n1);sort(node2,node2+n2);
    tree1.build(1,0,m);tree2.build(1,0,m);
    int pos1=0,pos2=0,ans=0,cur;
    while(pos1<n1&&pos2<n2){
        if(node1[pos1].rig<node2[pos2].rig){
            cur=tree2.query(1,0,node1[pos1].lef-1)+1;
            ans=max(ans,cur);
            tree2.add(1,0,node1[pos1].lef-1);
            tree1.update(1,node1[pos1].rig,cur);
            pos1++;
        }
        else{
            cur=tree1.query(1,0,node2[pos2].lef-1)+1;
            ans=max(ans,cur);
            tree1.add(1,0,node2[pos2].lef-1);
            tree2.update(1,node2[pos2].rig,cur);
            pos2++;
        }
    }
    while(pos1<n1){
        cur=tree2.query(1,0,node1[pos1].lef-1)+1;
        ans=max(ans,cur);
        tree2.add(1,0,node1[pos1].lef-1);
        tree1.update(1,node1[pos1].rig,cur);
        pos1++;
    }
    while(pos2<n2){
        cur=tree1.query(1,0,node2[pos2].lef-1)+1;
        ans=max(ans,cur);
        tree1.add(1,0,node2[pos2].lef-1);
        tree2.update(1,node2[pos2].rig,cur);
        pos2++;
    }
    enter(ans);
    return 0;
```

```
}
```

G. Directing Edges

题意

给定 n 个点 m 条边的无向图，其中 k 个点为特殊点，每个点有一个点权，每条边有一个边权。

要求给无向图的边定向，如果将某条边定向为单向边，则不需要支付费用。如果将某条边定向为双向边，则需要支付等于该边边权的费用。

定义图中的某个点为满足当且仅当所有特殊点可以到达沿有向边到达该点，且如果某个边成为满点，则可以获得等于该点点权的利润。

要求输出 n 个数，表示如果第 i 个点为满点能得到的最大收益，其中收益 $=$ 利润 $-$ 费用。

题解

有结论：对边双连通分量，存在某种策略可以将无向边全部定向为单向边并可以得到强连通分量。

于是求出所有边双连通分量并缩点，得到一棵无根树，树上的边为原图的桥，接下来考虑树形 dp。

先考虑如何求出第 1 个节点为满点时的最大收益。(这里及下文的节点均指缩点后的节点)

令第 1 个节点为根节点，设 $\text{dp}(u)$ 为在第 1 个节点为满点且为根节点的前提下 u 是满点时 u 所在子树利益的最大值。

首先 $\text{dp}(u)$ 的初值为 u 缩点中所有点的点权和。接下来考虑状态转移，假设节点 u 为满点，考虑它的子节点 v。

如果所有特殊点都在子节点 v 的子树中，则 v 一定是满点且连一条 $v\to u$ 的单向边，于是 $\text{dp}(u)\gets \text{dp}(v)$。

如果所有特殊点都不在子节点 v 的子树中，则贪心连一条 $u\to v$ 的单向边一定最优，于是 $\text{dp}(u)\gets \text{dp}(v)$。

除了上面两种特殊情况，则由于子节点 v 的子树中存在特殊节点，且 u 为满点，所以必须能从 v 走到 u。

于是要么 v 连一条单向边到 u，这样 v 不是满点，对 u 无贡献。要么 v,u 间连一条双向边，于是贡献为 $dp(v)-w(u,v)$。

所以该情况下有状态转移方程 $dp(u)=\max(0,dp(v)-w(u,v))$。

根据上述方法，可以 $O(n)$ 求解第 1 个节点为满点时的最大收益，接下来考虑换根 dp。

不断转移根，假设当前根为 u，v 为 u 的子节点。

于是先减去 v 对 u 的贡献，这样树被分成了以 u 为根节点的树和以 v 为根节点的树这两棵树。

然后考虑将 u 节点转化为 v 的子节点并计算贡献。

换根采用 dfs 过程且需要回溯。总时间复杂度 $O(n)$

```
const int MAXN=3e5+5;
struct Edge{
    int from,to,id,next;
}edge[MAXN<<1];
int head[MAXN],edge_cnt;
void Insert(int u,int v,int idx){
    edge[++edge_cnt]=Edge{u,v,idx,head[u]};
    head[u]=edge_cnt;
}
int low[MAXN],dfs_id[MAXN],dfs_t,bcc_id[MAXN],bcc_sp[MAXN],bcc_cnt;
bool is_bridge[MAXN];
LL bcc_v[MAXN],dp[MAXN],ans[MAXN];
vector<int> bcc[MAXN];
vector<pair<int,int> >g[MAXN];
int k,c[MAXN],node_sp[MAXN],edge_w[MAXN];
void dfs(int u,int fa){
    low[u]=dfs_id[u]=++dfs_t;
    for(int i=head[u];i;i=edge[i].next){
        int v=edge[i].to;
        if(v==fa)continue;
        if(!dfs_id[v]){
            dfs(v,u);
            low[u]=min(low[u],low[v]);
            if(low[v]>dfs_id[u])
            is_bridge[edge[i].id]=true;
        }
        else
        low[u]=min(low[u],dfs_id[v]);
    }
}
void dfs_2(int u,int fa){
    bcc_id[u]=bcc_cnt;
    bcc_v[bcc_cnt]+=c[u];
    bcc_sp[bcc_cnt]+=node_sp[u];
    bcc[bcc_cnt].push_back(u);
    for(int i=head[u];i;i=edge[i].next){
        if(is_bridge[edge[i].id])continue;
        int v=edge[i].to;
        if(v==fa||bcc_id[v])continue;
        dfs_2(v,u);
    }
}
void get_bcc(int n){
    dfs(1,1);
```

```
    _rep(i,1,n){
        if(!bcc_id[i]){
            bcc_cnt++;
            dfs_2(i,i);
        }
    }
    _rep(i,1,edge_cnt){
        if(is_bridge[edge[i].id]){
            int u=bcc_id[edge[i].from],v=bcc_id[edge[i].to];
            g[u].push_back(make_pair(v,edge_w[edge[i].id]));
        }
    }
}
void update_edge(int u,int v,int d,int w){
    LL t=dp[v];
    if(bcc_sp[v]>0&&bcc_sp[v]<k)
    t=max(0LL,dp[v]-w);
    dp[u]+=d*t;
    bcc_sp[u]+=d*bcc_sp[v];
}
void dp_1(int u,int fa){
    dp[u]=bcc_v[u];
    _for(i,0,g[u].size()){
        int v=g[u][i].first,w=g[u][i].second;
        if(v==fa)continue;
        dp_1(v,u);
        update_edge(u,v,1,w);
    }
}
void dp_2(int u,int fa){
    ans[u]=dp[u];
    _for(i,0,g[u].size()){
        int v=g[u][i].first,w=g[u][i].second;
        if(v==fa)continue;
        update_edge(u,v,-1,w);
        update_edge(v,u,1,w);
        dp_2(v,u);
        update_edge(v,u,-1,w);
        update_edge(u,v,1,w);
    }
}
int main()
{
    int n,m,u,v;
    n=read_int(),m=read_int(),k=read_int();
    _rep(i,1,k)
    node_sp[read_int()]=1;
    _rep(i,1,n)
    c[i]=read_int();
    _rep(i,1,m)
```

```
    edge_w[i]=read_int();
    _rep(i,1,m){
        u=read_int(),v=read_int();
        Insert(u,v,i);Insert(v,u,i);
    }
    get_bcc(n);
    dp_1(1,1);
    dp_2(1,1);
    _rep(i,1,n)
    space(ans[bcc_id[i]]);
    return 0;
}
```