

# 莫队算法 2

## 树上莫队

### 算法模型

树上单点修改 + 树上路径查询，且路径难以用树剖维护。

### 算法实现

考虑先将树转化为括号序列，设结点  $u$  对应的左括号编号为  $\text{dfn}_{1u}$  右括号编号为  $\text{dfn}_{2u}$

对当前维护的括号序列区间  $[l,r]$  仅计算  $[l,r]$  中未匹配的括号代表的结点的贡献。

即某个结点第奇数次出现时加上该结点的贡献，第偶数次出现时减去该结点贡献。

于是对每个询问路径  $u \rightarrow v$  不妨设  $\text{dfn}_{1u} \leq \text{dfn}_{1v}$

如果  $u$  为  $v$  的祖先，则路径对应区间为  $[\text{dfn}_{2u}, \text{dfn}_{1v}]$

否则路径对应区间为  $[\text{dfn}_{2u}, \text{dfn}_{1v}] + \text{LCA}(u,v)$

对于修改操作，如果修改的点在当前区间中且出现奇数次，则考虑更新答案，否则直接更新该点权值即可。

### 算法例题

[洛谷p4074](#)

#### 题意

给定一棵树，树上每个点有一种颜色，总共有  $m$  种颜色。同时给定序列  $\{v_i\}, \{w_i\}$

要求支持单点颜色修改操作和路径权值查询操作。设一条路径上颜色为  $i$  的颜色出现了  $c_i$  次，则该路径的权值为

$$\sum_{i=1}^m v_i \sum_{j=1}^{c_i} w_j$$

#### 题解

树上莫队板子题。

```
const int MAXN=1e5+5,MAXC=1e6+5;
int blk_sz;
```

```
struct Edge{
    int to,next;
}edge[MAXN<<1];
int head[MAXN],edge_cnt,dfn1[MAXN],dfn2[MAXN],invn[MAXN<<1],dfs_t;
void AddEdge(int u,int v){
    edge[++edge_cnt]=Edge{v,head[u]};
    head[u]=edge_cnt;
}
void dfs(int u,int fa){
    dfn1[u]=++dfs_t;
    invn[dfs_t]=u;
    for(int i=head[u];i;i=edge[i].next){
        int v=edge[i].to;
        if(v==fa)continue;
        dfs(v,u);
    }
    dfn2[u]=++dfs_t;
    invn[dfs_t]=u;
}
namespace LCA{
    int d[MAXN],sz[MAXN],f[MAXN];
    int h_son[MAXN],mson[MAXN],p[MAXN];
    void dfs_1(int u,int fa,int depth){
        sz[u]=1;f[u]=fa;d[u]=depth;mson[u]=0;
        for(int i=head[u];i;i=edge[i].next){
            int v=edge[i].to;
            if(v==fa)continue;
            dfs_1(v,u,depth+1);
            sz[u]+=sz[v];
            if(sz[v]>mson[u])
                h_son[u]=v,mson[u]=sz[v];
        }
    }
    void dfs_2(int u,int top){
        p[u]=top;
        if(mson[u])dfs_2(h_son[u],top);
        for(int i=head[u];i;i=edge[i].next){
            int v=edge[i].to;
            if(v==f[u]||v==h_son[u])continue;
            dfs_2(v,v);
        }
    }
    void init(int root){dfs_1(root,0,0);dfs_2(root,root);}
    int query(int u,int v){
        while(p[u]!=p[v]){
            if(d[p[u]]<d[p[v]])swap(u,v);
            u=f[p[u]];
        }
    }
}
```

```

        return d[u]<d[v]?u:v;
    }
};
struct query{
    int l,r,t,idx;
    bool operator < (const query &b)const{
        if(l/blk_sz!=b.l/blk_sz)return l<b.l;
        if(r/blk_sz!=b.r/blk_sz)return ((l/blk_sz)&1)?(r<b.r):(r>b.r);
        return ((r/blk_sz)&1)?(t<b.t):(t>b.t);
    }
}q[MAXN];
struct opt{
    int pos,pre,now;
}p[MAXN];
int cc[MAXN],lc[MAXN],val[MAXN],w[MAXN];
int col[MAXN],qn,pn,lef=1,rig=0;
LL ans[MAXN],cur_ans;
bool vis[MAXN];
void update_ans(int idx,int c){
    if(!vis[idx])
        cur_ans+=1LL*val[c]*w[++col[c]];
    else
        cur_ans-=1LL*val[c]*w[col[c]--];
    vis[idx]=!vis[idx];
}
void update_val(opt &p){
    if(vis[p.pos]){
        update_ans(p.pos,cc[p.pos]);
        cc[p.pos]=p.now;
        update_ans(p.pos,cc[p.pos]);
    }
    else
        cc[p.pos]=p.now;
    swap(p.pre,p.now);
}
int main()
{
    int n=read_int(),m=read_int(),q0=read_int();
    blk_sz=pow(n,2.0/3);
    _rep(i,1,m)val[i]=read_int();
    _rep(i,1,n)w[i]=read_int();
    _for(i,1,n){
        int u=read_int(),v=read_int();
        AddEdge(u,v);
        AddEdge(v,u);
    }
    _rep(i,1,n)cc[i]=lc[i]=read_int();
    LCA::init(1);
    dfs(1,0);
    while(q0--){
        int type=read_int();

```

```
if(type==0){
    pn++;
    int pos=read_int(),v=read_int();
    p[pn].pos=pos,p[pn].pre=lc[pos],p[pn].now=(lc[pos]=v);
}
else{
    qn++;
    int u=read_int(),v=read_int();
    if(dfn1[u]>dfn1[v])swap(u,v);
    if(LCA::query(u,v)==u)
        q[qn].l=dfn1[u],q[qn].r=dfn1[v];
    else
        q[qn].l=dfn2[u],q[qn].r=dfn1[v];
    q[qn].t=pn,q[qn].idx=qn;
}
}
sort(q+1,q+qn+1);
int tim=0;
_rep(i,1,qn){
    while(lef>q[i].l)lef--,update_ans(invn[lef],cc[invn[lef]]);
    while(rig<q[i].r)rig++,update_ans(invn[rig],cc[invn[rig]]);
    while(lef<q[i].l)update_ans(invn[lef],cc[invn[lef]]),lef++;
    while(rig>q[i].r)update_ans(invn[rig],cc[invn[rig]]),rig--;
    while(tim<q[i].t)update_val(p[++tim]);
    while(tim>q[i].t)update_val(p[tim--]);
    int u=invn[q[i].l],v=invn[q[i].r],p=LCA::query(u,v);
    if(u==p)
        ans[q[i].idx]=cur_ans;
    else{
        update_ans(p,cc[p]);
        ans[q[i].idx]=cur_ans;
        update_ans(p,cc[p]);
    }
}
_rep(i,1,qn)
enter(ans[i]);
return 0;
}
```

## 回滚莫队

### 算法模型

普通莫队仅适用于当询问得区间拓展/删除都可以  $O(1)$  维护的情况。

其中有一种操作难以  $O(1)$  维护时，可以利用回滚莫队仅维护一种操作，时间复杂度仍为  $O(n\sqrt{m})$  只是常数增大。

## 算法实现

设块大小为  $S$ 。当询问的左右端点都属于同一个块时，可以  $O(S)$  暴力处理。

剩下的询问先按左端点分块，然后按右端点从小到大排序。

对每个块  $[l,r]$  中的询问，先初始化当前莫队区间的右端点为  $r$ 。左端点为  $r+1$ 。

莫队区间的右端点仍然单增，于是每个块的右端点移动仍可以  $O(n)$  维护。

对莫队区间的左端点每次都从  $r+1$  开始重新向左拓展，然后每次处理完询问后再回滚到  $r+1$ 。注意回滚仅回滚左端点不回滚右端点。

于是每个询问的左端点移动可以  $O(S)$  维护。总时间复杂度仍为  $O\left(mS + \frac{n^2}{S}\right)$ 。取  $S = O\left(\frac{n}{\sqrt{m}}\right)$  时间复杂度为  $O(n\sqrt{m})$ 。

另外注意当跑到另一个块时需要消除之前的右端点影响，可以考虑暴力清空(因为只有  $O(S)$  个块)，也可以回滚上一个块的右端点。

## 算法例题

[Loj 2874](#)

### 题意

给定一个序列，每个询问序列区间  $[l,r]$ 。设区间  $[l,r]$  中数  $i$  出现  $c_i$  次，求  $\max(i*c_i)$ 。

### 题解

显然对于区间拓展操作，答案很容易维护，但区间删除操作答案难以维护。

直接套回滚莫队板子即可。

```
const int MAXN=1e5+5;
int blk_sz,a[MAXN],b[MAXN],vis[MAXN];
struct query{
    int l,r,idx;
    bool operator < (const query &b) const{
        if(l/blk_sz!=b.l/blk_sz) return l<b.l;
        return r<b.r;
    }
}opt[MAXN];
LL st[MAXN],cur;
int tp;
void add(int v){
    vis[v]++;
    st[++tp]=cur;
    cur=max(cur,1LL*vis[v]*b[v]);
}
```

```
}  
void del(int v){  
    vis[v]--;  
    cur=st[tp--];  
}  
LL ans[MAXN];  
int main()  
{  
    int n=read_int(),q=read_int();  
    blk_sz=n/sqrt(q)+1;  
    _rep(i,1,n)a[i]=b[i]=read_int();  
    sort(b+1,b+n+1);  
    int m=unique(b+1,b+n+1)-b;  
    _rep(i,1,n)a[i]=lower_bound(b+1,b+m,a[i])-b;  
    _for(i,0,q)opt[i].l=read_int(),opt[i].r=read_int(),opt[i].idx=i;  
    sort(opt,opt+q);  
    _for(i,0,q){  
        if(opt[i].l/blk_sz==opt[i].r/blk_sz){  
            _rep(j,opt[i].l,opt[i].r)  
                add(a[j]);  
            ans[opt[i].idx]=cur;  
            for(int j=opt[i].r;j>=opt[i].l;j--)  
                del(a[j]);  
        }  
    }  
    int lef=1,rig=0,lst=-1;  
    _for(i,0,q){  
        if(opt[i].l/blk_sz!=opt[i].r/blk_sz){  
            if(opt[i].l/blk_sz!=lst){  
                while(lef<=rig)del(a[rig--]);  
                lst=opt[i].l/blk_sz;  
                rig=min(lst*blk_sz+blk_sz-1,n);  
                lef=rig+1;  
            }  
            while(rig<opt[i].r)add(a[++rig]);  
            int tlef=lef;  
            while(tlef>opt[i].l)add(a[--tlef]);  
            ans[opt[i].idx]=cur;  
            while(tlef<lef)del(a[tlef++]);  
        }  
    }  
    _for(i,0,q)enter(ans[i]);  
    return 0;  
}
```

## `bitset` 与莫队

## 算法例题

洛谷p4688

### 题意

给定长度为  $n$  的序列  $a$  和  $m$  个询问。

每个询问给定  $[l_1, r_1], [l_2, r_2], [l_3, r_3]$  不断删去三个区间中的相同元素，直到三个区间不存在相同元素，问三个区间剩下的数的个数和。

例如，给定序列  $1, 1, 1, 2, 3$ ，询问区间  $[1, 3], [2, 4], [3, 5]$ ，进行删去操作后三个区间元素为  $\{1\}, \{2\}, \{2, 3\}$ 。

### 题解

不难发现答案为  $r_1 - l_1 + r_2 - l_2 + r_3 - l_3 + 3 - s$  其中  $s$  为公共元素的个数。

考虑维护每个原询问对应的公共元素的 `bitset` 初始时全为  $1$ ，记为  $s_i$  将原询问拆成三个询问区间丢掉莫队。

`bitset` 维护莫队区间元素，记为 `cur` 每次处理询问只需要将当前询问对应对应  $s_i$  更新为  $s_i \& \text{cur}$  即可。

但是不难发现 `bitset` 不支持重复元素，于是进行如下改造。

定义  $b_i$  为序列  $a$  中不大于  $a_i$  的位置的个数，然后当  $a_i$  在莫队区间第  $c_i$  次出现时，将  $b_i - c_i$  加入 `bitset`

另外需要维护  $m$  个  $s_i$  于是空间复杂度为  $O(\frac{nm}{w})$  难以承受。考虑将询问分成三份，于是空间复杂度变为原来的  $\frac{1}{3}$

原时间复杂度  $O(n\sqrt{m} + \frac{nm}{w})$  另外由于询问分成了三份，于是新时间复杂度为  $O(n\sqrt{3m} + \frac{nm}{w})$

```
const int MAXN=1e5+5,MAXM=3.5e4;
bitset<MAXN> qs[MAXM],cur;
int a[MAXN],b[MAXN],c[MAXN],ans[MAXM];
int blk_sz,col[MAXN];
struct query{
    int l,r,idx;
    bool operator < (const query &b)const{
        if(l/blk_sz!=b.l/blk_sz)return l<b.l;
        return ((l/blk_sz)&1)?(r<b.r):(r>b.r);
    }
}q[MAXM*3];
void add(int v){
    col[v]++;
    cur[c[v]-col[v]]=1;
```

```
}  
void del(int v){  
    cur[c[v]-col[v]]=0;  
    col[v]--;  
}  
int main()  
{  
    int n=read_int(),m=read_int();  
    _rep(i,1,n)a[i]=b[i]=read_int();  
    sort(b+1,b+n+1);  
    int nn=unique(b+1,b+n+1)-b;  
    _rep(i,1,n){  
        a[i]=lower_bound(b+1,b+nn,a[i])-b;  
        c[a[i]]++;  
    }  
    _for(i,1,nn)c[i]+=c[i-1];  
    while(m){  
        int qpos=0,qcnt=0;  
        mem(col,0);  
        while(m&&qpos<MAXM){  
            m--;  
            qs[qpos].set();  
            ans[qpos]=0;  
            _for(i,0,3){  
                int l=read_int(),r=read_int();  
                q[qcnt++]=query{l,r,qpos},ans[qpos]+=r-l+1;  
            }  
            qpos++;  
        }  
        blk_sz=n/sqrt(qcnt)+1;  
        sort(q,q+qcnt);  
        cur.reset();  
        int lef=1,rig=0;  
        _for(i,0,qcnt){  
            while(lef>q[i].l)add(a[--lef]);  
            while(rig<q[i].r)add(a[++rig]);  
            while(lef<q[i].l)del(a[lef++]);  
            while(rig>q[i].r)del(a[rig--]);  
            qs[q[i].idx]&=cur;  
        }  
        _for(i,0,qpos)  
            enter(ans[i]-3*qs[i].count());  
    }  
    return 0;  
}
```

From:  
<https://wiki.cvbbacm.com/> - CVBB ACM Team

Permanent link:  
[https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal\\_string:jxm2001:%E8%8E%AB%E9%98%9F%E7%AE%97%E6%B3%95\\_2&rev=1627818460](https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal_string:jxm2001:%E8%8E%AB%E9%98%9F%E7%AE%97%E6%B3%95_2&rev=1627818460) 

Last update: **2021/08/01 19:47**