string

```
/*
char*
    任意进制转换:
        //将 10 进制 n 转换为 r 进制并赋给 s
        itoa(int n,char* s,int r)
string:
    迭代器:
        //创建名为 it 的迭代器
        string::iterator it
    反转:
        //原地反转
        reverse(s.begin(), s.end());
        //反转并赋给 s1
        s1.assign(s.rbegin(), s.rend());
    大小写转换:
        transform(s.begin(), s.end(), s.begin(), ::toupper);
        transform(s.begin(), s.end(), s.begin(), ::tolower);
    类型转换:
        string ->int :
            string s("123");
            int i = atoi(s.c_str());
        int -> string:
            int a;
            stringstream(s) >> a;
    子串:
        //返回 pos 开始的 n 个字符组成的字符串
        string substr(int pos = 0,int n = npos)
    插入:
        //在 p0 位置插入字符串 s
        s.insert(int p0,string s)
    更改:
        s.assign(str);
        //如果 str 是”iamangel”就是把”ama”赋给字符串
        s.assign(str,1,3);
        //把字符串 str 从索引值 2 开始到结尾赋给 s
        s.assign(str,2,string::npos);
        s.assign(“gaint”);
        //把’n’‘I’‘c’‘o’‘\0’赋给字符串
        s.assign(“nico”,5);
        //把五个 x 赋给字符串
        s.assign(5,’x’);
    删除:
        //从索引 13 开始往后全删除
        s.erase(13);
        //从索引 7 开始往后删 5 个
        s.erase(7,5);
        //删除 it 指向的字符，返回删除后迭代器的位置
        iterator erase(iterator it);
        //删除 [first, last) 之间的所有字符，返回删除后迭代器的位置
        iterator erase(iterator first, iterator last);
    查找:
        //从 pos 开始查找字符 c 在当前字符串的位置
        int find(char c, int pos = 0)
        //从 pos 开始查找字符串 s 在当前串中的位置
```

```
        int find(const char *s, int pos = 0)
        //从 pos 开始查找字符串 s 中前 n 个字符在当前串中的位置
        int find(const char *s, int pos, int n)
        //从 pos 开始查找字符串 s 在当前串中的位置
        int find(const string &s, int pos = 0)
    删除所有特定字符:
        str.erase(std::remove(str.begin(), str.end(), 'a'), str.end());
    删除所有重复字符:
        //要求对象有序 O(n+n), 如果先排序 O(nlogn+n+n)
        str.erase(unique(str.begin(),str.end(),str.end())
 */
```

KMP

```
/*
 * 初始化 nex 数组
 * nex[i]：表示前 i 个字符的最长相同前后缀长度
 */
void getNext(char s[],int n){
    int i=0,j=-1;
    nex[0]=-1;
    while(i<n){
        if(j==-1 || s[i]==s[j]){
            nex[++i]=++j;
        }else{
            j=nex[j];
        }
    }
}
/*
 * KMP 匹配 (位置/个数 (可重叠/不可重叠))
 * 循环节:
 *     前 i 个字符的最小循环节长度: i-nex[i]
 *     循环节个数: i/(i-nex[i])
 */
int kmp(char s[],int n,char p[],int m){
    int i=0,j=0;
    // int cnt=0;
    getNext(p,m);
    while(i<n && j<m){
        if(j==-1 || s[i]==p[j]){
            i++;
            j++;
        }else{
            j=nex[j];
        }
        if(j==m){
            //匹配位置
            return i-j+1;
            //匹配个数
            //cnt++;
            //不可重叠
            //j=0;
            //可重叠
            //j=nex[j];
```

```
        }
    }
    //return cnt;
}
```

Manacher

```
/*
 * 求字符串 s 的最大回文子串
 * ma[]: 新字符串 (ma,mp 都注意要开两倍空间!)
 * mp[i]: 表示以 i 为中心的回文子串的半径 (包括特殊字符)
 * mx: 能延伸到最右端的位置
 * id: 能延伸到最右端的回文串中心位置
 */
int manacher(char s[],int len){
    //构造新字符串 两个字符之间插入一个其他字符
    //第 0 个字符忽略 (即加入另一种字符)
    int l=0;
    ma[l++]='$';
    ma[l++]='#';
    for(int i=0;i<len;i++){
        ma[l++]=s[i];
        ma[l++]='#';
    }
    ma[l]='\0';
    int mx=0,id=0;
    for(int i=1;i<l;i++){
        //递推部分 比较 mx 和 i 的位置
        //右边:2*id-i 表示 i 以 id 的对称点 因为不能超出 mx 所以要和 mx-i 取 min
        //左边: mp[i]=1
        mp[i]=mx>i?min(mp[2*id-i],mx-i):1;
        //更新部分
        //往两边更新
        while(ma[i+mp[i]]==ma[i-mp[i]]){
            mp[i]++;
        }
        //更新全局 mx 和 id
        if(i+mp[i]>mx){
            mx=i+mp[i];
            id=i;
        }
    }
    return *max_element(mp,mp+l)-1;
}
```

AC 自动机

```
/*
 * AC 自动机
 * tr[u][i]: 表示节点 u 的子节点 ('a'-'z'=>0-25) 编号
 * cnt: 总节点数/节点编号
 * fail[u]: 节点 u 的 fail 指针指向节点编号
 * ==> fail 指针: 指向当前失配的字符串的最长后缀字符串
 * ==> 比如匹配到 abcd, d 失配, 那么 c 的 fail 指针指向的就是后缀为 abc(或者 bc 或者 c) 的最长字符串
 * val[u]: 以节点 u 所表示字符串作为结尾的单词数
```

```
 * sum[u]: 以节点 u 所表示字符串作为前缀的单词数
 */
//插入单词，构建字典树
void insert(char *s){
    int len=strlen(s);
    int now=0;
    for(int i=0;i<len;i++){
        int id=s[i]-'a';
        if(!tr[now][id]){
            tr[now][id]=++cnt;
        }
        now=tr[now][id];
        sum[now]++;
    }
    val[now]++;
}
//使用 bfs 构造 fail 指针
void build(){
    queue<int> q;
    //第一层节点均指向根
    for(int i=0;i<26;i++){
        if(tr[0][i]){
            fail[tr[0][i]]=0;
            q.push(tr[0][i]);
        }
    }
    while(!q.empty()){
        int u=q.front();
        q.pop();
        //处理出 fail 指针或者该节点走一步 (fail 跳转不算) 能到达的节点
        for(int i=0;i<26;i++){
            if(tr[u][i]){
                //已经路径压缩，子节点 fail 直接指向父节点的 fail 的对应子节点
                //(无论是否存在，若不存在会自动指向上一级 fail)
                fail[tr[u][i]]=tr[fail[u]][i];
                q.push(tr[u][i]);
            }else{
                //预处理该节点走一步 (fail 跳转不算) 能到达的节点
                //相当于路径压缩，这样下面的节点就不用 while(fail[fail[...]]) 这样跳
                tr[u][i]=tr[fail[u]][i];
            }
        }
    }
}
int query(char *s){
    int len=strlen(s);
    int now=0;
    int ans=0;
    for(int i=0;i<len;i++){
        int id=s[i]-'a';
        now=tr[now][id];
        //从当前节点一直往上跳转直到根或者 val[t] 为-1
        for(int t=now;t&&~val[t];t=fail[t]){
            ans+=val[t];
            //避免重复匹配
            val[t]=-1;
```

```
        }
    }
    return ans;
}
/*
 * 01 异或字典树
 */
void insert(ll x){
    int rt=0;
    for(int i=32;i>=0;i--){
        //从高位，分解为二进制数位
        int id=(x>>i)&1;
        if(!tr[rt][id]){
            tr[rt][id]=++cnt;
        }
        rt=tr[rt][id];
    }
    //末尾标记原数字
    val[rt]=x;
}
ll query(ll x){
    int rt=0;
    for(int i=32;i>=0;i--){
        int id=(x>>i)&1;
        //不管这一位 (id) 是 1 还是 0，优先找与这一位不同的节点，这样异或值就会尽量大
        if(tr[rt][id^1]){
            rt=tr[rt][id^1];
        }
        else{
            rt=tr[rt][id];
        }
    }
    return val[rt];
}
/*
 * AC 自动机上 dp
 * bzoj1030: 求长度为 m 且不含有所给 n 个单词的字符串的个数
 * dp[i][j]: 长度为 i(从 trie 树根节点走 i 步) 在节点 j 的满足条件字符串个数
 * ans=sum(dp[m][i]) i(0-cnt)
 * val[i]: 标记 i 节点是否可以访问 (即对应字符串是否含有所给单词)
 *         在构建 fail 指针的时候，也要更新 val[u]|=val[fail[u]]
 */
int solve(){
    dp[0][0]=1;
    //枚举步数
    for(int i=1;i<=m;i++){
        //枚举所有节点
        for(int j=0;j<=cnt;j++){
            //标记的单词 (或是 fail 指针有标记) 都不能经过
            if(val[j]){
                continue;
            }
            for(int k=0;k<26;k++){
                dp[i][tr[j][k]]=(dp[i][tr[j][k]]+dp[i-1][j])%MOD;
            }
        }
```

```
    }
    int sum=1;
    for(int i=1;i<=m;i++){
        sum=(sum*26)%MOD;
    }
    int ans=0;
    for(int i=0;i<=cnt;i++){
        if(!val[i]){
            ans+=dp[m][i];
        }
    }
    return ((sum-ans)%MOD+MOD)%MOD;
}
/*
 * AC 自动机上数位 dp
 * bzoj3530: 给定整数 n 和其他 m 个整数，求 0-N 中，不含有这 m 个整数子串的数的个数
 * 先将模式串插入 trie 树中，跑数位 dp 模板 (不用分解数位了)，dp 数组多加一维标记节点位置
 * 判断下一个节点有效性 (即是否含有模式串，注意题目是含有任意一个还是含有全部 (状态压缩))
 * zero: 是否有前导零限制，即 zero 为 true 时，该位不能选 0
 * flag: 表示是否含有 trie 树中的串
 * dp[i][j][k]: 表示前 i 位数在 u 节点上前面是否含有无效串的数的个数
 */
int dfs(int len,int u,int flag,int limit,int zero){
    //递归边界，即枚举完一个可能的数，如果没有任何限制就返回 1，即一个满足要求的数
        if(len<0){
        return (!flag && !zero) ? 1 : 0;
    }
    //在没有上限限制和没有前导零限制的情况下直接记忆化搜索
        if(dp[len][u][flag]!=-1 && !limit && !zero){
        return dp[len][u][flag];
    }
        int up = limit ? n[len]-'0' : 9;
    int ans = 0;
        for(int i=0;i<=up;i++){
        //有前导零限制且当前枚举位为 0，相当于这一状态下面递归的所有状态都是无效的
        //所以直接舍弃掉这一位从下一位开始并从 trie 树根节点开始
        //比如 1234，枚举最高位的时候 0 是不可以的，有前导零限制
        //所以 0000,0001,0002 等这些数都是无效的
        //直接从高二位开始算，也就相当于是 234，同理这一位枚举到也不行
        //枚举到 1 的时候 100，101 这些就是有效状态
            if(i==0 && zero){
            ans=(ans+dfs(len-1,0,flag,limit && i==up, 1))%MOD;
        }else{
            //当前节点含有 trie 树的串或者是下一个节点含有
            ans=(ans+dfs(len-1,tr[u][i],flag|val[tr[u][i]],limit && i==up,zero && i==0))%MOD;
        }
    }
    if(!limit && !zero){
        dp[len][u][flag] = ans;
    }
        return ans;
}
int solve(){
    len=strlen(n);
    //反转，从高位枚举
    reverse(n,n+len);
```

```
    memset(dp,-1,sizeof(dp));
    return dfs(len-1,0,0,1,1);
}
/*
 * 矩阵快速幂加速 dp(递推)
 * 有时候推出 ac 自动机加 dp, 结果 n 的范围是 1e9, 就需要用到矩阵快速幂来优化 dp
 * bzoj1009: 求有多少个 n 位数不含有所给 m 位数的子串
 * 和上面 ac 自动机上 dp 是一样的, 枚举长度, 枚举节点, 枚举子节点 (0-9), 状态转移
 * 由于 n 比较大, 而 m 比较小, 我们可以把问题转化为求出从 trie 树根节点走 n 步的有效路径数
 * 所以通过构建 fail 树后, 可以计算出 trie 树初始邻接矩阵 (1 步能走到的), 注意无效状态
 * 然后使用矩阵快速幂加速, 计算邻接矩阵的 n 次方, 即走 n 步的路径数矩阵
 * 枚举 i, 计算 sum(m[0][i]) 即为答案 (从根节点开始到其他各个节点刚好走 n 步的路径数)
 */
int solve(){
    Mat tmp,a;
    memset(tmp.m,0,sizeof(tmp.m));
    //dp 初始状态
    tmp.m[0][0]=1;
    for(int i=0;i<=tot;i++){
        for(int j=0;j<10;j++){
            if(!val[tr[i][j]]){
                //从 i 节点到 j 节点存在可行路径 (包括计算 fail 指针时指向祖先节点的关系)
                a.m[i][tr[i][j]]+=1;
            }
        }
    }
    //矩阵快速幂求长度为 n 的路径条数
    a=tmp*(a^n);
    int ans=0;
    for(int i=0;i<=tot;i++){
        //所有从根节点到任意节点 n 步的路径数
        ans+=a.m[0][i];
        ans%=k;
    }
    return ans;
}
```

后缀数组

倍增

```
//"banana" 后缀为 [banana$ anana$ nana$ ana$ na$ a$ $]
//sa[i]: 排名第 i(从 0 开始) 小的后缀的首字符下标
//比如 [6 5 3 1 0 4 2]==>[$ a$ ana$ anana$ banana$ na$ nana$]
//rk[i]: 下标 i 开始的后缀 (不含 $) 的排名 (按字典序从小到大, 相当于 sa 的逆)
//[4 3 6 2 5 1]
//h[i]: 排名为 i 的后缀和排名为 i-1 的后缀的最长公共前缀
//[1(ana$-a$) 3(anana$-ana$) 0 0 2]
//辅助数组: t[N],t2[N],c[N];
void build_sa(int n, int m){
    //n 为字符串的长度, 字符集的值为 0~m-1
    //相当于在后面加一个 $
    //有时候是数字数组而不是字符数组, 最好加上 s[n]=0
    n++;
    int *x=t, *y=t2;
    //基数排序
```

```
for(int i=0;i<m;i++){
    c[i]=0;
}
for(int i=0;i<n;i++){
    c[x[i]=s[i]]++;
}
for(int i=1;i<m;i++){
    c[i]+=c[i-1];
}
//或者 ~i 表示 i!=-1
for(int i=n-1;i>=0;i--){
    sa[--c[x[i]]]=i;
}
for(int k=1; k<=n; k<<=1){
    int p=0;
    for(int i=n-k;i<n;i++){
        y[p++]=i;
    }
    for(int i=0;i<n;i++){
        if(sa[i]>=k){
            y[p++]=sa[i]-k;
        }
    }
    //类似上面，只是把 i 换成 y[i]
    for(int i=0;i<m;i++){
        c[i]=0;
    }
    for(int i=0;i<n;i++){
        c[x[y[i]]]++;
    }
    for(int i=1;i<m;i++){
        c[i]+=c[i-1];
    }
    for(int i=n-1;i>=0;i--){
        sa[--c[x[y[i]]]]=y[i];
    }
    swap(x, y);
    p=1;
    x[sa[0]]=0;
    for(int i=1;i<n;i++){
        x[sa[i]]=y[sa[i-1]]==y[sa[i]] && y[sa[i-1]+k]==y[sa[i]+k]? p-1 : p++;
    }
    if (p>=n){
        break;
    }
    m = p;
}
//去掉 $
n--;
for(int i = 0; i <= n; i++){
    rk[sa[i]] = i;
}
//计算 h
int k=0;
for(int i = 0; i < n; i++){
    if(k){
```

```
            k--;
        }
        int j = sa[rk[i] - 1];
        while(s[i + k] == s[j + k]){
            k++;
        }
        h[rk[i]] = k;
    }
}
```

DC3

```
/*
 * 使用 DC3 构建后缀数组 O(n) by Kuangbin 模板
 * 所有数组要开三倍
 * wa[N*3],wb[N*3],wv[N*3],wss[N*3]
 */
#define F(x) ((x)/3+((x)%3==1?0:tb))
#define G(x) ((x)<tb?(x)*3+1:((x)-tb)*3+2)
int c0(int *r,int a,int b){
    return r[a] == r[b] && r[a+1] == r[b+1] && r[a+2] == r[b+2];
}
int c12(int k,int *r,int a,int b){
    if(k == 2){
        return r[a] < r[b] || ( r[a] == r[b] && c12(1,r,a+1,b+1) );
    }else{
        return r[a] < r[b] || ( r[a] == r[b] && wv[a+1] < wv[b+1]);
    }
}
void sort(int *r,int *a,int *b,int n,int m){
    int i;
    for(i = 0;i < n;i++){
        wv[i] = r[a[i]];
    }
    for(i = 0;i < m;i++){
        wss[i] = 0;
    }
    for(i = 0;i < n;i++){
        wss[wv[i]]++;
    }
    for(i = 1;i < m;i++){
        wss[i] += wss[i-1];
    }
    for(i = n-1;i >= 0;i--){
        b[--wss[wv[i]]] = a[i];
    }
}
void dc3(int *r,int *sa,int n,int m){
    int i, j, *rn = r + n;
    int *san = sa + n, ta = 0, tb = (n+1)/3, tbc = 0, p;
    r[n] = r[n+1] = 0;
    for(i = 0;i < n;i++){
        if(i %3 != 0){
            wa[tbc++] = i;
        }
    }
```

```
    sort(r + 2, wa, wb, tbc, m);
    sort(r + 1, wb, wa, tbc, m);
    sort(r, wa, wb, tbc, m);
    for(p = 1, rn[F(wb[0])] = 0, i = 1;i < tbc;i++){
        rn[F(wb[i])] = c0(r, wb[i-1], wb[i]) ? p-1 : p++;
    }
    if(p < tbc){
        dc3(rn,san,tbc,p);
    }else{
        for(i = 0;i < tbc;i++){
            san[rn[i]] = i;
        }
    }
    for(i = 0;i < tbc;i++){
        if(san[i] < tb){
            wb[ta++] = san[i] * 3;
        }
    }
    if(n % 3 == 1){
        wb[ta++] = n - 1;
    }
    sort(r, wb, wa, ta, m);
    for(i = 0;i < tbc;i++){
        wv[wb[i] = G(san[i])] = i;
    }
    for(i = 0, j = 0, p = 0;i < ta && j < tbc;p++){
        sa[p] = c12(wb[j] % 3, r, wa[i], wb[j]) ? wa[i++] : wb[j++];
    }
    for(;i < ta;p++){
        sa[p] = wa[i++];
    }
    for(;j < tbc;p++){
        sa[p] = wb[j++];
    }
}
//str 和 sa 也要三倍
void da(int str[],int n,int m){
    for(int i = n;i < n*3;i++){
        str[i] = 0;
    }
    dc3(str, sa, n+1, m);
    int i,j,k = 0;
    for(int i = 0; i <= n; i++){
        rk[sa[i]] = i;
    }
    //计算 h
    for(int i = 0; i < n; i++){
        if(k){
            k--;
        }
        int j = sa[rk[i] - 1];
        while(a[i + k] == a[j + k]){
            k++;
        }
        h[rk[i]] = k;
    }
```

```
}
```

后缀数组经典应用

```
/*
 * 预处理 O(nlogn)
 * dp[i][j]: 从 a[i] 开始 2^j 个数的最小值
 */
void RMQ_init(int n){
    for(int i=0;i<=n;i++){
        dp[i][0]=h[i];
    }
    for(int j=1;(1<<j)<=n;j++){
            for(int i=0;i+(1<<(j-1)<n;i++){
                //两段重叠部分小区间
                dp[i][j]=min(dp[i][j-1],dp[i+(1<<(j-1))][j-1]);
            }
    }
}
/*
 * 查询 [l,r] 最小值, 最大值同理
 */
int RMQ(int l,int r){
    int k=0;
    //保证刚好 [l,l+2^k] 和 [r-2^k,r] 重叠
    while((1<<(k+1))<=r-l+1){
        k++;
    }
    return min(dp[l][k],dp[r-(1<<k)+1][k]);
}

//====================== 后缀数组应用总结 ========================
//1. 任意两个后缀的最长公共前缀长度
/*
 * 转化为求两个后缀对应排名区间的最小 h 值, 即 RMQ 问题
 * 要先初始化 RMQ(n+1), 求回文串时 RMQ(2*n+2)
 * 分清查询的是两个后缀首字符还是两个后缀排名
 * l,r 为两个后缀的排名
 */
int solve0(int l,int r){
    if(l==r){
        return n-sa[l];
    }
    if(l>r){
        swap(l,r);
    }
    //这里可灵活处理, 有时候不需要 +1
    return RMQ(l+1,r);
}
/*
 * i,j 为两个后缀的首字符下标
 */
int solve1(int i,int j){
    if(i==j){
        return n-i;
    }
```

```
    int ri=rk[i];
    int rj=rk[j];
    if(ri>rj){
        swap(ri,rj);
    }
    //注意 h 数组是类似差分的表示，区间 n 其实只有 n-1 的 h 数组
    return RMQ(ri+1,rj);
}
//2. 可重叠最长重复子串长度
/*
 * 重复子串＝最长公共前缀＝区间内 min(h)
 * 最长重复子串＝所有区间 max(min(h))=max(h)
 */
int solve2(){
    int ans=0;
    //注意从 1 开始，排名为 0 的是无效后缀'$'
    for(int i=1;i<=n;i++){
        ans=max(ans,h[i]);
    }
    return ans;
}
//3. 不可重叠最长重复子串
/*
 * 先二分答案（子串长度）将问题转化为判定性问题
 * 即判断字符串里是否存在长度为 mid 的重复子串
 * 将后缀按排名顺序进行分组，每组后缀间 h 值 >=mid
 * 否则单独一个组
 * 同一组中判断 max(sa[i]) 和 min(sa[i])
 * 即最左后缀和最右后缀的差值是否 >=mid(即不重叠)
 */
int solve3(){
    int ans=0;
    int l=0,r=n/2;
    //分组和判断同时进行
    //mx mn 分别维护组内最右和最左后端
    int mx=-INF,mn=INF;
    while(l<=r){
        int mid=(l+r)/2;
        bool flag=false;
        //可以直接从 2 开始，因为 1 和 0 的 h 值在这里无意义
        for(int i=2;i<=n;i++){
            if(h[i]<mid){
                //新的分组
                mx=-INF;
                mn=INF;
            }else{
                //h[i] 表示的就是 sa[i] 和 sa[i-1] 这两个后缀的 LCP
                //所以这两个位置都有可能
                mx=max(mx,max(sa[i],sa[i-1]));
                mn=min(mn,min(sa[i],sa[i-1]));
                //poj1743 求的是差分数组所以这里是 >
                if(mx-mn>=mid){
                    flag=true;
                    break;
                }
            }
```

```
        }
        if(flag){
            ans=mid;
            l=mid+1;
        }else{
            r=mid-1;
        }
    }
    return ans;
}
//4. 重叠出现至少 k 次的最长重复子串
/*
 * 同样是二分答案，后缀分组再进行判断
 * 不过这里判断的是组内后缀个数是否 >=k
 */
int solve4(int k){
    int ans=0;
    int l=0,r=n/2;
    while(l<=r){
        int mid=(l+r)/2;
        int cnt=1;
        bool flag=false;
        for(int i=2;i<=n;i++){
            if(h[i]<mid){
                cnt=1;
            }else{
                cnt++;
                if(cnt>=k){
                    flag=true;
                    break;
                }
            }
        }
        if(flag){
            ans=mid;
            l=mid+1;
        }else{
            r=mid-1;
        }
    }
    return ans;
}
//5.(长度大于等于 k) 的不同子串个数
/*
 * 按排名枚举后缀，对于排名为 i 的后缀来说，贡献为 n-sa[i]+h[i](k 为 1 的情况)
 * 即后缀长度 (从后缀第一个字符到后缀第 i 个字符算一个子串) 减去重复部分
 */
int solve5(){
    int ans=0;
    for(int i=1;i<=n;i++){
        if(h[i]){
            h[i]=max(0,h[i]-(k-1));
        }
        ans+=max(0,(n-sa[i]-(k-1)-h[i]));
        //k 为 1 的情况
        //ans+=(n-sa[i]-h[i]);
```

```
    }
    return ans;
}
//6. 最长回文子串
/*
 * 将字符串反转接在原串后面 (中间加入特殊分隔符), 将问题转化为求 LCP
 * 枚举每一个位置分别求出奇数回文串和偶数回文串长度
 */
int solve6(){
    //要先把字符串反转拼接再求 sa
    int ans=0;
    for(int i=0;i<n;i++){
        //奇数回文串
        ans=max(ans,2*solve1(i,2*n-i)-1);
        //偶数回文串
        ans=max(ans,2*solve1(i,2*n-i+1));
    }
    return ans;
}
//7. 出现次数 [a,b]/k 次 (即 [k,k]) 的子串个数
/*
 * 设 cal(k) 为出现次数大于等于 k 的子串个数
 * 问题转化为求 cal(a)-cal(b+1)
 */
int cal(int k){
    //特判 k
    if(k==1){
        //即所有不同子串个数
        return solve5();
    }
    int ans=0;
    //不断枚举每一段 k-1 的 h 区间 (其实就是 k 个后缀)
    //0 没有 h 值, 1 的 h 值恒为 0
    int l=2,r=k;
    int pre=0;
    while(r<=n){
        //这里直接使用 RMQ
        int now=RMQ(l,r);
        if(now>=pre){
            ans+=now-pre;
        }
        pre=now;
        l++;
        r++;
    }
    return ans;
}
int solve7(int a,int b){
    return cal(a)-cal(b+1);
}
//8. 字符串 S 由字符串 T 重复 Q 次得到, 求最大 Q
/*
 * 使用 kmp 更方便, 如果非要用后缀数组请用 dc3 模板
 * 枚举 T 串长度 k, 先判断 len(S)%len(T)==0?
 * 再判断 lcp(suffix(0),suffix(k))==n-k?
 * suffix(i) 指首字符位置为 i 的后缀
```

```
 * 因为 suffix(0) 固定，无需预处理出整个 RMQ
 * 只需 O(n) 求出所有 lcp(suffix(0),x) 即可
 */
int solve8(){
    lcp[rk[0]]=n;
    for(int i=rk[0]-1;i>=0;i--){
        lcp[i]=min(lcp[i+1],h[i+1]);
    }
    for(int i=rk[0]+1;i<=n;i++){
        lcp[i]=min(lcp[i-1],h[i-1]);
    }
    for(int k=1;k<=n;k++){
        if(n%k==0 && lcp[rk[k]]==n-k){
            return n/k;
        }
    }
}
//9. 重复次数最多的连续重复子串
/*
 * 暂时无法理解，贴个题解板子
 */
void solve9(){
    int ans=0,pos=0,lenn;
    //保证有重复，所以是 len/2
    for(int i=1;i<=len/2;i++){
        for(int j=0;j<len-i;j+=i){
            if(str[j]!=str[j+i]){
                continue;
            }
            //通过下标查询
            int k=solve1(j,j+i);
            int tol=k/i+1;
            int r=i-k%i;
            int p=j;
            int cnt=0;
            for(int m=j-1;m>j-i&&str[m]==str[m+i]&&m>=0;m--){
                cnt++;
                if(cnt==r){
                    tol++;
                    p=m;
                }
                else if(rk[p]>rk[m]){
                    p=m;
                }
            }
            if(ans<tol){
                ans=tol;
                pos=p;
                lenn=tol*i;
            }else if(ans==tol && rk[pos]>rk[p]){
                pos=p;
                lenn=tol*i;
            }
        }
    }
    //这里，如果字符总长度小于 2，那么就在原串中找出一个最小的字符就好
```

```
    if(ans<2){
        char ch='z';
        for(int i=0;i<len;i++){
            if(str[i]<ch){
                ch=str[i];
            }
        }
        printf("%c\n",ch);
        return;
    }
    for(int i=pos;i<pos+lenn;i++){
        printf("%c",str[i]);
    }
    printf("\n");
}
//10. 最长公共子串
/*
 * 求 A 和 B 的最长公共子串即转化为求 A 和 B 后缀的最长公共前缀
 * 将 A 和 B 拼接求出 h 数组，当 suffix(sa[i]) 和 suffix(sa[i-1]) 不在同个字符串时
 * h[i] 才有效，求出最大值即可
 */
int solve10(){
    int ans=0;
    for(int i=1;i<=n;i++){
        //al 为第一个字符串长度
        if(sa[i]<al && sa[i-1]<al || sa[i]>=al && sa[i-1]>=al){
            continue;
        }
        ans=max(ans,h[i]);
    }
    return ans;
}
//11. 求长度大于等于 k 的公共子串个数
/*
 * 同样是转化为 A 和 B 所有后缀的最长公共前缀大于等于 k 的贡献累加起来
 * 用常用的处理方法，将两个字符串拼接，中间用分隔符隔开，求出 h 数组
 * 然后对 h 数组分组（把 h 值大于等于 k 的分在一组）
 * 然后在这道题里要使用一个单调栈来维护前面的后缀对后面的后缀的贡献
 * 单调栈维护一个 h 值和一个该 h 值在前面充当 lcp 的
 * sta[i][0]: 维护 h 值 sta[i][1]: 维护这个 h 值代表前面多少个后缀的 lcp 贡献（最小 h 值）
 * 维护的类似一种前缀和的东西，具体看代码详细注释
 */
int solve11(int k){
    int top=0;
    int tot=0, ans=0;
    //对 h 分组
    for(int i=1;i<=n;i++){
        if(h[i]<k){
            //单独一个后缀的分组，无需考虑
            top=0;
            tot=0;
        }else{
            //cnt 表示 h[i] 作为 lcp 的后缀个数
            int cnt=0;
            //前一个后缀属于 A，计算贡献
            //即能产生多少个长度大于等于 k 的公共子串
```

```
            if(sa[i-1]<al){
                //前面 (i-1) 必须是属于 A 的后缀，才能计算这个贡献
                //所以后面把 B 的后缀入栈是没毛病的，这里维护的实际上是一个类似前缀和的东西
                //即栈顶元素已经是累加了栈底所有元素的贡献
                //但是这个单调栈是动态变化的，所以不能单纯保留栈顶元素
                cnt++;
                //比如 h[i]=4，例如 abcd，那么大于等于 k 的子串就可以是
                //ab abc abcd 这三个，所以是 h[i]-k+1
                tot+=h[i]-k+1;
            }
            //维护单调栈 (栈顶到栈底递减)
            //这里不用考虑 h[i] 是 A 的还是 B 的
            while(top && h[i]<=sta[top-1][0]){
                top--;
                //更新贡献
                //这里实际上应该是 tot(贡献) 减去 sta[top][0]*sta[top][1]
                //再加上 h[i]*sta[top][1]
                //相当于就是栈顶被替换了，所以栈顶这个 h 值之前充当了多少后缀的 lcp
                //都要改成这个新的栈顶 h[i] 的贡献了
                tot+=(h[i]-sta[top][0])*sta[top][1];
                //既然 h[i] 比 sta[top][0] 小，那么 sta[top][0] 能作为 lcp(h 最小值) 的 h[i]
                //肯定也可以，所以累加上即可
                cnt+=sta[top][1];
            }
            //此时栈里的所有 h 值都小于 h[i]
            //入栈，维持单调性
            sta[top][0]=h[i];
            sta[top][1]=cnt;
            top++;
            //属于 B 的后缀，累加 A 的贡献
            if(sa[i]>al){
                ans+=tot;
            }
        }
    }
    //同理对 B 串的后缀维护单调栈，累计 B 的贡献
    top=tot=0;
    for(int i=1;i<=n;i++){
        if(h[i]<k){
            top=0;
            tot=0;
        }else{
            int cnt=0;
            if(sa[i-1]>al){
                cnt++;
                tot+=h[i]-k+1;
            }
            while(top && h[i]<=sta[top-1][0]){
                top--;
                tot+=(h[i]-sta[top][0])*sta[top][1];
                cnt+=sta[top][1];
            }
            sta[top][0]=h[i];
            sta[top][1]=cnt;
            top++;
            if(sa[i]<al){
```

```
                ans+=tot;
            }
        }
    }
    return ans;
}
//12. 给 n 个字符串，出现在不少于 k 个字符串的最长子串
/*
 * 同样是拼接字符串对 h 数组分组，二分答案进行判断
 */
bool check12(int mid){
    bool flag=false;
    for(int i=1;i<len;i++){
        //不满足要求的组直接不用管
        if(h[i]<mid){
            continue;
        }
        //组内不同字符串个数
        int cnt=0;
        //标记某个字符串是否出现过
        memset(vis,false,sizeof(vis));
        while(h[i]>=mid && i<len){
            if(!vis[idx[sa[i-1]]]){
                vis[idx[sa[i-1]]]=true;
                cnt++;
            }
            //找到 1 组
            i++;
        }
        //注意最后一个的判断
        if(!vis[idx[sa[i-1]]]){
            vis[idx[sa[i-1]]]=1;
            cnt++;
        }
        //出现在 k 个不同字符串中
        if(cnt>k){
            if(!flag){
                flag=true;
                //清空之前的答案
                //ans 存储出现在至少 k 个不同字符串的后缀 sa 值
                ans.clear();
                ans.push_back(sa[i-1]);
            }
            else{
                //只需加入最后一个满足条件后缀的 sa
                ans.push_back(sa[i-1]);
            }
        }
    }
    //是否有满足要求的答案
    return flag;
}
void solve12(){
    ans.clear();
    int l=1,r=len;
    while(l<=r){
```

```
        int mid=(l+r)>>1;
        if(check(mid)){
            l=mid+1;
        }
        else{
            r=mid-1;
        }
    }
    //输出多个最长公共子串 (至少在 k 个字符串中出现过)
    for(int i=0;i<ans.size();i++){
        for(int j=ans[i];j<ans[i]+r;j++)
            printf("%c",s[j]-1+'a');
        printf("\n");
    }
}
//13. 给 n 个字符串，在每个字符串中至少出现两次且不重叠的最长子串
/*
 * 把每个字符串拼接起来，中间用分隔符分开，求 h 数组
 * 二分子串长度，对 h 数组分组，维护每一个字符串在当前组的最小和最大 sa 值
 * 然后判断不重叠即可 (也同时判断了出现 2 次)
 */
bool check13(int mid){
    //后缀起始位置的最大最小值
    for(int i=0;i<n;i++){
        mx[i]=0;
        mn[i]=INF;
    }
    for(int i=1;i<=len;i++){
        if(h[i]<mid){
            //重置分组
            for(int i=0;i<n;i++){
                mx[i]=0;
                mn[i]=INF;
            }
        }else{
            //一个 h 值表示两个后缀
            mx[idx[sa[i]]]=max(mx[idx[sa[i]]],sa[i]);
            mn[idx[sa[i]]]=min(mn[idx[sa[i]]],sa[i]);
            mx[idx[sa[i-1]]]=max(mx[idx[sa[i-1]]],sa[i-1]);
            mn[idx[sa[i-1]]]=min(mn[idx[sa[i-1]]],sa[i-1]);
            bool flag=true;
            //需要判断每个字符串 (都要出现两次以上，且不重叠)
            for(int j=0;j<n;j++){
                if(mx[j]-mn[j]<mid){
                    flag=false;
                    break;
                }
            }
            if(flag){
                return true;
            }
        }
    }
    return false;
}
int solve13(){
```

```
    //len 为拼接后的字符串长度
    int l=0,r=len+1;
    int ans=0;
    while(l<=r){
        int mid=(l+r)>>1;
        if(check13(mid)){
            ans=mid;
            l=mid+1;
        }else{
            r=mid-1;
        }
    }
    return ans;
}
//14. 给 n 个字符串，正序或逆序出现在每个字符串的最长子串
/*
 * 把每个字符串正序和逆序拼接起来，中间用分隔符分开，求 h 数组
 * 二分子串长度，对 h 数组分组，判断是否存在一组 h 值里面所属 n 个不同字符串
 * 所以在拼接字符串的时候要把每个字符串（正序和逆序）的每个字符标记是第几个字符串
 * for(int j=0;j<siz;j++){
        idx[len]=i;
        //加 2*n+1 保证不会跟分隔符相同
        s[len++]=ss[i][j]-'A'+2*n+1;
    }
    idx[len]=i;
    s[len++]=sep++;
 */
bool check14(int mid){
    //这里用 set 来进行组内所属字符串的去重
    set<int> se;
    for(int i=1;i<len;i++){
        if(h[i]>=mid){
            se.insert(idx[sa[i]]);
            se.insert(idx[sa[i-1]]);
        }else{
            if(se.size()==n){
                return true;
            }
            se.clear();
        }
    }
    //注意最后一个组
    return se.size()==n;
}
int solve14(){
    if(n==1){
        return strlen(ss[0]);
    }
    int l=1,r=100;
    //变量记得初始化!!!
    int ans=0;
    while(l<=r){
        int mid=(l+r)>>1;
        if(check14(mid)){
            ans=mid;
            l=mid+1;
```

```
        }else{
            r=mid-1;
        }
    }
    return ans;
}
//15. 求两个字符串的公共子串，且只能在母串中出现一次，求满足的最短子串
/*
 * 两字符串拼接，按求最长公共子串的方法，不过要判断是否重复出现，重复出现的话，
 * 其对应后缀一定是相邻的，所以要判断相邻的是否有可能重复出现，要枚举从 h[i] 到 1
 */
int solve(int n){
    int ans=0x3f3f3f3f;
    for(int i=1;i<=n;i++){
        if(h[i]==0 || sa[i]<len1 && sa[i-1]<len1 || sa[i]>=len1 && sa[i-1]>=len1){
            continue;
        }
        for(int j=h[i];j>=1;j--){
            if(j>h[i-1] && j>h[i+1]){
                ans=min(ans,j);
            }
        }
    }
    if(ans==0x3f3f3f3f){
        ans=-1;
    }
    return ans;
}
//几个注意点
//字符串数组一般可以用一个 int 数组来表示，特别是需要拼接字符串的题目
//拼接字符串先设定一个比较大的分隔符，比如 300，然后每次分割一次都要 +1
//注意最后一位补 0 s[len]=0
/*
    len=0;
    int sep=30;
    for(int i=0;i<n;i++){
        scanf("%s",str[i]);
        siz[i]=strlen(str[i]);
        for(int j=0;j<siz[i];j++){
             s[len++]=str[i][j]-'a'+1;
             idx[len-1]=i;
        }
        s[len++]=sep++;
    }
    s[len]=0;
    build_sa(len,250);
 */
```

字符串哈希

```
typedef unsigned long long ull;
const ull seed1=146527,seed2=19260817;
const ull MOD1=1000000009,MOD2=998244353;
/*
 * 字符串哈希，记得初始化 (init())
```

```
 */
void init(){
    p[0]=1;
    for(int i=1;i<N;i++){
        p[i]=p[i-1]*seed1%MOD1;
    }
}
void Hash(char s[],int n,ull *h1){
    h[0]=0;
    for(int i=1;i<=n;i++){
        h[i]=(h[i-1]*seed1%MOD1+s[i-1])%MOD1;
    }
}
ull substr(int l,int len,ull *h1){
    return (h1[l+len]-h1[l]*p[len]%MOD1+MOD1)%MOD1;
}
```

最大/小表示法

```
/*
 * 求循环字符串 s 的最小/最大表示
 * i,j: 当前比较两个字符串的起始位置
 * k: 这两个字符串已比较的长度
 */
int getMin(char s[]){
    int n=strlen(s);
    int i=0,j=1,k=0;
    while(i<n && j<n && k<n){
        int t=s[(i+k)%n]-s[(j+k)%n];
        if(!t){
            k++;
        }else{
            if(t>0){
                //如果是求最大表示则为 j+=k+1
                i+=k+1;
            }else{
                 //同理则为 i+=k+1
                j+=k+1;
            }
            if(i==j){
                j++;
            }
            k=0;
        }
    }
    return min(i,j);
}
```

普通并查集

```
/*
 * 基础并查集 (注意初始化 p[i]=i)
 * p[i]: i 的根
 */
int find(int x){
    //递归写法
    return x==p[x]?p[x]:p[x]==find(p[x]);
}
//=========================//
//有时候需要写成这样
int find(int x){
    if(x!=p[x]){
        int fa=p[x];
        p[x]=find(p[x]);
        //这里就可以对临时保存的 fa 进行操作，特别在带权并查集中
        //也可以直接 p[x]=find(p[x])，然后对修改后的 p[x] 进行操作
        v[x]+=v[fa];
    }
    return p[x];
}
int find(int x){
    //有时需要维护多个信息 (带权并查集)，需要非递归写法
    int fa=x;
    //找到 x 的根
    while(fa!=p[fa]){
        fa=p[fa];
    }
    //从 x 重新一步一步往上，沿途更新
    while(x!=p[x]){
        x=p[x];
        p[x]=fa;
    }
    return p[x];
}
int unit(int a,int b){
    //最基础的合并
    int fa=find(a);
    int fb=find(b);
    if(fa!=fb){
        p[fa]=fb;
    }
}
```

带权并查集

```
/*
 * 带权并查集
 * 所谓带权并查集就是在并查集的基础上除了维护根节点再维护一些其他的值
 * 例如集合大小，元素移动次数，元素到根节点的距离等
 * 状态的转移在 find 和 unit 都要考虑，而且要注意顺序
 * eg. poj1182——食物链 设 r[i] 表示 i 和父节点 (根) 的关系
 * r[i]=0 表示和父节点同类，1 表示被父节点吃，2 表示吃父节点
 */
```

```
int find(int x){
    if(x!=p[x]){
        //暂存当前根节点，用于状态转移更新信息
        int t=p[x];
        //递归压缩路径，此时 p[x] 信息已更新
        p[x]=find(p[x]);
        //元素移动次数/元素到根距离
        cnt[x]+=cnt[t];
        //假设 r[x]=1，即 x 被 t 吃，如果 r[t]=2，即 t 吃 p[x]
        //那说明 x 和 p[x] 同类，r[x]=(1+2)%3=0，其他同理
        r[x]=(r[x]+r[t])%3;
    }
    return p[x];
}
void unit(int a,int b){
    int fa=find(a);
    int fb=find(b);
    if(fa!=fb){
        p[fa]=fb;
        //hdu2818 将一堆放到另一堆上 维护某一元素下方元素个数，即到根的距离
        cnt[fa]=siz[fb];
        //集合大小
        siz[fb]+=siz[fa];
    }
}
bool unit(int a,int b,int q){
    q--;
    //q 是操作，0 表示同类，1 表示 a 吃 b
    int fa=find(a);
    int fb=find(b);
    if(fa!=fb){
        p[fb]=fa;
        //可以画图去看 r[fb] 即求 fb 到 fa 的关系
        //必须通过 fb->b(-r[b]) b->a(q) a->fa(r[a]) 三个向量相加即可
        r[fb]=(-r[b]+r[a]+q+3)%3;
        return false;
    }else{
        //已在同一集合
        //同样通过画图，此时 a,b 有相同根 f
        //r[a]+q 即 b->a(q) a->f(r[a])==r[b]，若不符合，即矛盾
        return (r[a]+q)%3!=r[b];
    }
}
```

Dijkstra 堆优化

```
/*
 * 堆优化的 Dijkstra(最短距离、最短路条数)
 * dis[i]: i 到最短路集合的距离
 * pcnt[i]: 到 i 的路径条数
 */
void Dijkstra(){
    for(int i=0;i<N;i++){
        dis[i]=INF;
        vis[i]=false;
```

```
        pcnt[i]=0;
    }
    priority_queue<node> q;
    dis[s]=0;
    pcnt[s]=1;
    q.push(node{s,0});
    while(!q.empty()){
        node tmp=q.top();
        q.pop();
        int u=tmp.v;
        if(vis[u]){
            continue;
        }
        vis[u]=true;
        for(int i=head[u];i!=-1;i=edge[i].next){
            int v=edge[i].v;
            int w=edge[i].w;
            if(!vis[u] && dis[v]>dis[u]+w){
                dis[v]=dis[u]+w;
                //多加了一条 u-v 的边而已，所以到 v 的最短路条数和到 u 的相同
                pcnt[v]=pcnt[u];
                q.push(node{v,dis[v]});
            }else if(!vis[u] && dis[v]==dis[u]+w){
                //相等的情况就多出了 pcnt[u] 条路了
                pcnt[v]+=pcnt[u];
            }
        }
    }
}
```

次小生成树

```
/*
 * used[i][j]: i-j 这条边是否在 MST 中
 * path[i][j]: <i...j> 的路径上的最长边
 * pre[i]: MST 中 i 的前驱节点，即上一步能够更新 low[i] 的点，即将 i 加入 MST 的点
 */
int Prim(){
    for(int i=0;i<n;i++){
        vis[i]=false;
        low[i]=cost[0][i];
        pre[i]=0;
        for(int j=0;j<n;j++){
            used[i][j]=false;
            path[i][j]=false;
        }
    }
    vis[0]=true;
    pre[0]=-1;
    low[0]=0;
    int ans=0;
    for(int i=1;i<n;i++){
        int k=-1,Min=INF;
        for(int j=0;j<n;j++){
            if(!vis[j] && low[j]<Min){
```

```
                Min=low[j]; k=j;
            }
        }
        if(k==-1){
            break;
        }
        ans+=Min;
        vis[k]=true;
        used[k][pre[k]]=used[pre[k]][k]=true;
        for(int j=0;j<n;j++){
            //因为 j 访问过，也就是在 MST 中，即此时可以看成是 j--pre[k]--k
            //j 是 MST 点集的一点，而 k 就刚好被选中要加入这个点集
            //pre[k] 是 k 的前驱，也就是点集的边缘，也就是把 k 带进 MST 的点
            //所以 j 到 k 的最大边就是由两部分更新而来，path[j][pre[k]] 和 low[k]
            if(vis[j] && j!=k){
                path[j][k]=path[k][j]=max(path[j][pre[k]],low[k]);
            }else if(!vis[j] && low[j]>cost[k][j]){
                low[j]=cost[k][j];
                pre[j]=k;
            }
        }
    }
    return ans;
}
//先求出 MST，再枚举不在 MST 中的边求出 SST
int solve(){
    int ans=Prim();
    for(int i=0;i<n;i++){
        for(int j=i+1;j<n;j++){
            if(!used[i][j]){
                ans=min(ans,ans-path[i][j]+cost[i][j]);
            }
        }
    }
    return ans;
}
```

拓扑排序

```
/*
 * 拓扑排序 (bfs)
 * to[i]: 拓扑排序第 i 个节点
 * flag: 是否存在环
 */
void topo(){
    queue<int> q;
    int k=0;
    for(int i=1;i<=n;i++){
        if(ind[i]==0){
            q.push(i);
            topo[k++]=u;
        }
    }
    int cnt=0;
    while(!q.empty()){
```

```
        int u=q.front();
        q.pop();
        cnt++;
        for(int i=head[u];i!=-1;i=edge[i].next){
            int v=edge[i].v;
            ind[v]--;
            if(!ind[v]){
                q.push(v);
                topo[k++]=v;
            }
        }
    }
    if(cnt!=n){
        //有环
        flag=true;
    }
}
```

最小支配集/最小点覆盖/最大独立集

```
/*
 * 贪心求解树上最小支配集/最小点覆盖/最大独立集
 * p[i]: i 的父节点
 * dft[i]: dfs 序第 i 个节点
 * s[]: 求解的集合
 * vis[i]: i 是否被集合覆盖
 */
void dfs(int u,int f){
    //dfs 序
    dft[k++]=u;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        if(v==f){
            continue;
        }
        p[v]=u;
        dfs(v,u);
    }
}
bool s[N],vis[N];
//最小支配集 (覆盖所有点)
int greedy1(){
    int ans=0;
    //逆序进行贪心
    for(int i=n-1;i>=0;i--){
        int t=dft[i];
        //当前点未被覆盖
        if(!vis[t]){
            //父节点不属于支配集
            if(!s[p[t]]){
                //将父节点加入支配集
                s[p[t]]=true;
                ans++;
            }
            //父节点加入支配集，即该节点和父节点的父节点都被覆盖
```

```
            vis[t]=true;
            vis[p[t]]=true;
            vis[p[p[t]]]=true;
        }
    }
    return ans;
}
//最小点覆盖集 (覆盖所有边)
int greedy2(){
    int ans=0;
    //该贪心策略要排除根节点 dft[0]
    for(int i=n-1;i>=1;i--){
        int t=dft[i];
        //当前节点和父节点都不属于覆盖集
        if(!vis[t] && !vis[p[t]]){
            //父节点加入覆盖集
            s[p[t]]=true;
            ans++;
            //当前点和父节点的父节点被覆盖
            vis[t]=true;
            vis[p[t]]=true;
        }
    }
    return ans;
}
//最大点独立集
int greedy3(){
    int ans=0;
    for(int i=n-1;i>=0;i--){
        int t=dft[i];
        //当前节点没有被覆盖
        if(!vis[t]){
            //加入独立集
            s[t]=true;
            ans++;
            //当前节点和父节点被覆盖
            vis[t]=true;
            vis[p[t]]=true;
        }
    }
    return ans;
}
```

树的直径/重心

```
/*
 * dfs 求树的直径
 * 从任意一点 dfs 一遍，找到最远的点 node，再从 node 点 dfs 一遍，找到直径的另一点
 * dfs(1,0) 前记得使 vis[1]=true
 */
void dfs(int u){
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        int w=edge[i].w;
        if(!vis[v]){
```

```
            vis[v]=true;
            d[v]=d[u]+w;
            if(d[v]>ans){
                ans=d[v];
                node=v;
            }
            dfs(v,d[v]);
        }
    }
}
/*
 * dfs 求树的重心
 * dfs 一遍求出 ans 即为重心，siz 初始化为 INF，即重心的最大子树大小
 * 树重心的一些性质：
 * 0. 树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么他们的距离和一样
 * 1. 把两个树通过一条边相连得到一个新的树，那么新的树的重心在连接原来两个树的重心的路径上
 * 2. 把一个树添加或删除一个叶子，那么它的重心最多只移动一条边的距离
 */
void dfs(int u,int f){
    son[u]=1;
    int res=0;
    for (int i=head[u];i!=-1;i=edges[i].next){
        int v=edges[i].v;
        if (v==f){
            continue;
        }
        dfs(v,u);
        son[u]+=son[v];
        res=max(res,son[v]);
    }
    res=max(res,n-son[u]);
    if (res<siz){
        ans=u;
        siz=res;
    }
}
```

强连通分量

```
/*
 * 求有向图的强连通分量
 * 强连通分量：分量内任意两个点都能互相到达
 * low[u]:u 能回溯到的最早祖先
 * dfn[u]:dfs 序
 * scc[u]:u 所属 scc 编号
 * inSta[u]:u 是否在栈中
 * s: 暂存节点
 */
void dfs(int u){
    //初始化 dfs 序
    low[u]=dfn[u]=++idx;
    s.push(u);
    inSta[u]=true;
    int siz=g[u].size();
    for(int i=0;i<siz;i++){
        int v=g[u][i];
```

```
        if(dfn[v]==-1){
            //未访问
            dfs(v);
            //此时已经递归到底，可以用 low[v] 来更新 low[u]
            low[u]=min(low[u],low[v]);
        }else if(inSta[v]){
            //访问过，在栈中 (low[v] 还没完全更新)
            low[u]=min(low[u],dfn[v]);
        }
    }
    if(low[u]==dfn[u]){
        //能回到的最早节点就是本身，说明得到一个强连通分量
        cnt++;
        while(!s.empty()){
            int t=s.top();
            scc[t]=cnt;
            s.pop();
            inSta[t]=false;
            if(t==u){
                break;
            }
        }
    }
}
void tarjan(int n){
    memset(dfn,-1,sizeof(dfn));
    memset(scc,-1,sizeof(scc));
    while(!s.empty()){
        s.pop();
    }
    idx=cnt=0;
    for(int i=1;i<=n;i++){
        //未访问
        if(dfn[i]==-1){
            dfs(i);
        }
    }
}
```

边-双连通分量

```
/*
 * tarjan 算法求桥/边-双连通分量
 * 双连通分量即删除桥后的各个连通分量，可以说 BCC=SCC+ 桥的判定
 * 小结论：一个有桥的连通图要变为边双连通图至少需要加几条边？
 *     双连通分量缩点建树，求叶子节点数量，答案为 (leaf+1)/2
 */
void dfs(int u,int f){
    low[u]=dfn[u]=++idx;
    s.push(u);
    inStack[u]=true;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        //无向图
        if(v==f){
```

```
            continue;
        }
        if(dfn[v]==-1){
            dfs(v,u);
            low[u]=min(low[u],low[v]);
            //桥的判定
            if(low[v]>dfn[u]){
                bridegs.push_back(make_pair(u,v));
            }
        }else if(inStack[v]){
            low[u]=min(low[u],dfn[v]);
        }
    }
    //删掉桥剩下的就是边-双联通分类
    if(low[u]==dfn[u]){
        num++;
        while(!s.empty()){
            int t=s.top();
            s.pop();
            inStack[t]=false;
            bcc[t]=num;
            if(t==u){
                break;
            }
       }
    }
}
void solve(int n){
    num=0;
    memset(dfn, -1, sizeof(dfn));
    memset(low, -1, sizeof(low));
    for(int i=1;i<=n;i++){
        if(dfn[i]==-1){
            dfs(i,0);
        }
    }
}
```

点-双连通分量

```
/*
 * tarjan 算法求割点/点-双连通分量
 * 边/点双连通分量：不存在桥/割点的极大子图
 * 一个割点属于多个点双连通分量，一个桥不属于任何边双连通分量
 * bcc[i]: 表示编号为 i 的连通分量的节点 (vector 数组)
 * rt: 根，而不是父节点
 */
void dfs(int u,int rt){
    low[u]=dfn[u]=++idx;
    //记录儿子节点个数，用于根的特判
    int son=0;
    s.push(u);
    inStack[u]=true;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
```

```
        if(dfn[v]==-1){
            dfs(v,rt);
            low[u]=min(low[u],low[v]);
            //割点的判定
            if(low[v]>=dfn[u]){
                num++;
                while(!s.empty()){
                    int t=s.top();
                    s.pop();
                    inStack[t]=false;
                    bcc[num].push_back(t);
                    if(t==v){
                        break;
                    }
                }
                //点双连通分量点可重复
                bcc[num].push_back(u);
                //根的特判
                if(u==rt){
                    son++;
                }else{
                    cut[u]=true;
                }
            }
        }else if(inStack[v]){
            low[u]=min(low[u],dfn[v]);
        }
    }
    //根节点特判，只要多于两个儿子即为割点
    if(son>=2 && u==rt){
        cut[u]=true;
    }
}
void solve(){
    memset(dfn,-1,sizeof(dfn));
    memset(low,-1,sizeof(low));
    for(int i=1;i<=n;i++){
        if(dfn[i]==-1){
            //注意调用方式
                dfs(i,i);
        }
    }
}
```

LCA

ST 表在线

```
/*
 * ver[i]: 第 i 个访问节点的编号
 * fir[i]: 编号为 i 的节点第一次出现的时间
 * dep[i]: 第 i 个访问节点的深度
 * ver,dep 注意开两倍空间
 */
void dfs(int u,int f,int d){
```

```
    tot++;
    ver[tot]=u;
    fir[u]=tot;
    dep[tot]=d;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        if(v==f){
            continue;
        }
        dfs(v,u,d+1);
        //回溯再记录一次
        tot++;
        ver[tot]=u;
        dep[tot]=d;
    }
}
/*
 * RMQ 初始化
 * dp[i][j]: 从 dep[i] 起 2^j 个数的编号最小值 (ver 和 dep 的编号即访问时间)
 * dp 数组注意开两倍空间
 */
void RMQ_init(int n){
    for(int i=0;i<n;i++){
        dp[i][0]=i;
    }
    for(int j=1;(1<<j)<=n;j++){
        for(int i=0;i+(1<<j)-1<n;i++){
            //和普通的 rmq 不同，这里比较的是深度，但 dp 保存的是位置
            int a=dp[i][j-1];
            int b=dp[i+(1<<(j-1))][j-1];
            dp[i][j]=(dep[a]<dep[b]?a:b);
        }
    }
}
int RMQ(int l,int r){
    int k = 31 - __builtin_clz(r - l + 1);
    int a=dp[l][k];
    int b=dp[r-(1<<k)+1][k];
    return (dep[a]<dep[b]?a:b);
}
// 找出访问 u 和 v 之间 (fir[u]...fir[v]) 深度最小的点的编号 idx，由 ver 映射成节点编号
int LCA(int u,int v){
    int x=fir[u];
    int y=fir[v];
    if(x>y){
        swap(x,y);
    }
    int idx=RMQ(x,y);
    return ver[idx];
}
```

Tarjan 离线

```
/*
 * vis[u]: 访问标记
 * p[u]:u 的根/父节点
```

```
 * que[u]: 跟 u 有联系的查询链表
 */
void LCA(int u,int f){
    vis[u]=true;
    int siz=g[u].size();
    for(int i=0;i<siz;i++){
        int v=g[u][i].v;
        if(v==f){
            continue;
        }
        LCA(v,u);
        //回溯之后更新父节点
        p[v]=u;
    }
    int qsiz=que[u].size();
    for(int i=0;i<qsiz;i++){
        // 遍历跟 u 有查询关系的点
        int v=que[u][i].q;
        int id=que[u][i].idx;
        if(vis[v]){
            //如果已经访问过则 lca 为并查集中 v 的根
            lca[id]=Find(v);
        }
    }
}
```

倍增在线

```
/*
 * d[i]: 节点 i 的深度
 * fa[u][i]: 表示从 u 往上跳 2^i 格可以到达的点
 */
void dfs(int u){
    //倍增
    for(int i=1;pw[i]<=d[u];i++){
        fa[u][i]=fa[fa[u][i-1]][i-1];
    }
    for(int i=0;i<g[u].size();i++){
        int v=g[u][i];
        if(v!=fa[u][0]){
            //fa[v][0] 即 v 往上跳 1 即 v 的父节点
            fa[v][0]=u;
            //深度
            d[v]=d[u]+1;
            dfs(v);
        }
    }
}
int lca(int x,int y){
    if(d[x]<d[y]){
        swap(x,y);
    }
    int tmp=d[x]-d[y];
    for(int i=0;pw[i]<=tmp;i++){
        //往上跳
        if(tmp&pw[i]){
```

```
            x=fa[x][i];
        }
    }
    //跳到刚好，即 y=lca(x,y)
    if(x==y){
        return x;
    }
    for(int i=16;i>=0;i--){
        //x 和 y 一起跳
        if(fa[x][i]!=fa[y][i]){
            x=fa[x][i];
            y=fa[y][i];
        }
    }
    return fa[x][0];
}
```

判断二分图

```
/*
 * 染色法判断二分图 (奇圈): 任何无回路的图都是二分图
 * col[i]: 点 i 的颜色 (1,-1 代表不同颜色, 0 代表未染色)
 */
bool dfs(int u,int c){
    //将 u 涂成 c 颜色
    col[u]=c;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        //相邻顶点不能同色
        if(col[v]==c){
            return false;
        }
        //如果未涂色则尝试递归涂色, 如果失败返回 false, 则该判断也 false
        if(col[v]==0 && !dfs(v,-c)){
            return false;
        }
    }
    return true;
}
bool solve(){
    for(int i=0;i<n;i++){
        if(col[i]==0 && !dfs(i,1)){
            return false;
        }
    }
    return true;
}
```

最大匹配

```
/*
 * 可以加个源点汇点跑最大流, 但有时候比较毒瘤可能会 T...
 * 匈牙利算法
 * g[][]: 图的邻接矩阵表示
 * vis[i]: 本轮是否已访问
 * mt[i]: i 所匹配的点
 */
bool dfs(int u, int n){
    for(int i=1;i<=n;++i){
        if(!vis[i] && g[u][i]){
            vis[i] = true;
            if(mt[i]==-1 || dfs(mt[i], n)){
                mt[i] = u;
                return true;
            }
        }
    }
    return false;
}
int solve(){
    int ans=0;
    memset(mt,-1,sizeof(mt));
    for(int i=1;i<=n;++i){
```

```
        memset(vis,0,sizeof(vis));
        ans += dfs(i, n);
    }
    return ans;
}
```

完美匹配

```
/*
 * 跑一遍 Hungary 算法，判断是否每个点都有匹配
 * mat[i]: 左点集 i 对应的匹配点
 */
```

最优匹配 (KM 算法)

```
/* * KM 算法计算二分图带权最优匹配
 * n 个男生和 n 个女生，两两之间有好感度 (边权)，匹配使得总好感度最大
 * love[i][j]: 女生 i 和男生 j 的好感度
 * exg[i]/exb[i]: 女生期望值 (初始值为最高好感度)，男生期望值 (初始值为 0)
 * visg[i]/visb[i]: 记录每一轮匹配中是否匹配过
 * match[i]: 男生 i 匹配到的女生，初始值为-1
 * slack[i]: 男生 i 还需要多少期望值才能匹配到女生
 */
bool dfs(int u){
    visg[u]=true;
    for(int i=0;i<n;i++){
        if(visb[i]){
            continue;
        }
        //计算期望度和好感度差值
        int gap=exg[u]+exb[i]-love[u][i];
        //这里改成 gap!=0 可以求出最小完备匹配
        if(gap==0){
            //符合要求
            visb[i]=true;
            //男生没有匹配，或者男生对应女生可以找另一个男生
            if(match[i]==-1 || dfs(match[i])){
                match[i]=u;
                return true;
            }
        }else{
            //无法找到匹配的，必须修改期望度 (对当前女生 u，也就是 KM 算法里枚举的 i)
            slack[i]=min(slack[i],gap);
        }
    }
    return false;
}
int km(){
    memset(match,-1,sizeof(match));
    memset(exb,0,sizeof(exb));
    for(int i=0;i<n;i++){
        exg[i]=love[i][0];
        for(int j=1;j<n;j++){
            exg[i]=max(exg[i],love[i][j]);
        }
```

```
    }
    for(int i=0;i<n;i++){
        memset(slack,INF,sizeof(slack));
        while(true){
            memset(visg,false,sizeof(visg));
            memset(visb,false,sizeof(visb));
            //找到
            if(dfs(i)){
                break;
            }
            //找不到
            int d=INF;
            //枚举男生
            for(int j=0;j<n;j++){
                if(!visb[j]){
                    //降到某个 slack[j]
                    d=min(d,slack[j]);
                }
            }
            //这一轮问过的女生降低期望值，男生增加期望值
            for(int j=0;j<n;j++){
                if(visg[j]){
                    exg[j]-=d;
                }
                if(visb[j]){
                    exb[j]+=d;
                }else{
                    //没访问过的男生，因为女生期望值降低，所以距离女生期望值又近一步
                    slack[j]-=d;
                }
            }
        }
    }
    int ans=0;
    for(int i=0;i<n;i++){
        ans+=love[match[i]][i];
    }
    return ans;
}
```

最小支配集/最小点覆盖/最大独立集/最大团/最小路径覆盖

```
/*
    1. 二分图一些性质
        最小点覆盖数 == 最大匹配数   ||  最小边覆盖数 == 最大独立数 == 顶点数-最大匹配数
        最大独立集 == 补图的最大团
        (最大独立集：一个最大的点的集合，该集合内的任意两点没有边相连)
        (最大团：一个最大的点的集合，该集合内的任意两点都有边相连)
    2. 有向图的最小路径覆盖问题 ==> 二分图解决
        路径覆盖：通过几条路径 (多条边) 覆盖有向图所有顶点，一个点也可成为路径覆盖，长度为 0
        2.1 最小不相交路径覆盖：每一条路径经过的顶点各不相同。例如 1->3>4, 2, 5
            将有向图的每个点拆成入点和出点，对每一条有向边 (u,v)，都在二分图中将 u 的入点连向 v 的出点
            结论：最小路径覆盖数 =|V(G)|-二分图最大匹配数
        2.2 打印出每条路径：
            在 dinic 算法的 dfs 增广中，记录流向
```

```
            if(dis>0){
                ......
                to[u]=v;
                return dis;
            }
            枚举所有入点 (1-n)，当 to[i] 不为零时，沿着 i=to[i] 路径输出即可
            for(int i=1;i<=n;i++){
                if(to[i]){
                    int t=i;
                    do{
                        if(t>n){
                            t-=n;
                        }
                        printf("%d ",t);
                        int x=to[t];
                        to[t]=0;
                        t=x;
                    }while(t);
                printf("\n");
            }
        2.3 最小可相交路径覆盖：每一条路径经过的顶点可以相同。例如 1->3->4，2->3>5
            先用 floyd 求出传递闭包，然后对当前的图求最大匹配 (可以直接用邻接矩阵跑匈牙利算法)
            for(int k=1;k<=n;k++){
                for(int i=1;i<=n;i++){
                    for(int j=1;j<=n;j++){
                        if(a[i][k]&&a[k][j]){
                            a[i][j]=true;
                        }
                    }
                }
            }
 */
```

多重匹配

```
/*
    二分图多重匹配即二分图中一个点可以和多条边相关联，Li 表示点 i 最多可以和多少条边相关联。
    二分图多重匹配可分为最大匹配与最优匹配两种，分别可以用最大流与最大费用最大流解决。
    1. 二分图多重最大匹配：
        建立源点 S 和汇点 T，S 向 X 连边，Y 向 T 连边，容量为 Li，原二分图中各边保留，容量为 1
        (若该边可以使用多次则容量大于 1)，答案即该网络的最大流
    2. 二分图多重最优匹配：
        建立源点 S 和汇点 T，S 向 X 连边，Y 向 T 连边，容量为 Li，费用为 0，原来二分图中各边保留，容量为 1
        (若该边可以使用多次则容量大于 1)，费用为该边的权值，答案即该网络的最大费用最大流
*/
```

最大流 (Dinic)

```
/*
 * s,t: 源点，汇点
 * dep[i]: i 点的深度
 * cur[i]: i 的邻接链表当前访问到的边
 * bfs 构建分层图，dfs 增广，时间复杂度：O(n^2 m)
 */
bool bfs(){
```

```
    memset(dep,0,sizeof(dep));
    dep[s]=1;
    queue<int> q;
    q.push(s);
    while(!q.empty()){
        int u=q.front();
        q.pop();
        for(int i=head[u];i!=-1;i=edge[i].next){
            int v=edge[i].v;
            //可用流量大于 0 &&& 未标记深度
            if(edge[i].w>0 && dep[v]==0){
                dep[v]=dep[u]+1;
                q.push(v);
            }
        }
    }
    return dep[t]!=0;
}
int dfs(int u,int flow){
    if(u==t){
        return flow;
    }
    //当前弧优化
    for(int &i=cur[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        int w=edge[i].w;
        if(dep[v]==dep[u]+1 && w>0){
            //dfs 向下增广
            int dis=dfs(v,min(w,flow));
            if(dis>0){
                //正向边和反向边更新可用流量
                edge[i].w-=dis;
                edge[i^1].w+=dis;
                return dis;
            }
        }
    }
    return 0;
}
int dinic(){
    int ans=0;
    while(bfs()){
        //不断构建分层图再增广，加上当前弧优化
        for(int i=1;i<N;i++){
            cur[i]=head[i];
        }
        while(int d=dfs(s,INF)){
            ans+=d;
        }
    }
    return ans;
}
```

最小费用最大流

```
/*
 * 最小费用最大流 (mcmf)
 * inq[i]: i 节点是否在队列中
 * d[i]: bellmanford 算法的距离
 * p[i]: 记录流向 i 的上一条边编号
 * a[i]: i 的可改进量 (通过量)
 */
bool BellmanFord(int &flow,int &cost){
    for(int i=0;i<N;i++){
        d[i]=INF;
        inq[i]=false;
    }
    d[s]=0;
    p[s]=0;
    a[s]=INF;
    queue<int> q;
    q.push(s);
    inq[s]=true;
    while(!q.empty()){
        int u=q.front();
        q.pop();
        inq[u]=false;
        for(int i=head[u];i!=-1;i=edge[i].next){
            int v=edge[i].v;
            int w=edge[i].w;
            int c=edge[i].c;
            if(w>0 && d[v]>d[u]+c){
                d[v]=d[u]+c;
                p[v]=i;
                a[v]=min(a[u],w);
                if(!inq[v]){
                    q.push(v);
                    inq[v]=true;
                }
            }
        }
    }
    if(d[t]==INF){
        return false;
    }
    flow+=a[t];
    cost+=d[t]*a[t];
    for(int u=t;u!=s;u=edge[p[u]].u){
        edge[p[u]].w-=a[t];
        edge[p[u]^1].w+=a[t];
    }
    return true;
}
void mcmf(int &flow,int &cost){
    flow=0,cost=0;
    while(BellmanFord(flow,cost));
}
```

上下界网络流

```
/*
 * 0. 假设流量上下界为 [l,r]
 * 1. 无源汇可行流 (循环流)
 *     添加超级源汇 ss,tt, 对于每一条弧 (u,v), 拆成 (u,v,(r-l)),(u,tt,l),(ss,v,l)
 *     跑一遍 ss 到 tt 的最大流, 若附加边 (u,tt),(ss,v) 都满流, 说明存在可行流
 * 2. 有源汇可行流
 *     建立弧 (t,s,(INF)), 转化为无源汇可行流, 按无源汇可行流的方法建图跑一遍最大流即可,
 *     而且此时 (t,s) 的流量就是原图的总流量
 * 3. 有源汇最大流
 *     按照有源汇可行流的方法建图, 如果存在可行流, 不用清空流量, 再跑一遍从 s 到 t 的最大流
 * 4. 有源汇最小流
 *     按照有源汇可行流建图, 但不要建立弧 (t,s), 跑一遍 ss 到 tt 的最大流, 再建立弧 (t,s,(INF)),
 *     再跑一遍 ss 到 tt 的最大流
 * */
```

常见模型

```
/*
 * 0. 最小割 = 最大流
 * 1. 最大权闭合子图 = 正权点和-最小割
 *    闭合子图就是给定一个有向图, 从中选择一些点组成一个点集 V。对其中任意一个点, 其后续节点都仍然在 V 中
 *    建立源点 s 和汇点 t, 将源点 s 与所有权值为正的点相连, 容量为权值
 *    将所有权值为负的点与汇点 t 相连, 容量为权值的绝对值
 *    权值为 0 的点不做处理, 同时将原来的边容量设置为无穷大
 * 2. 最小路径覆盖 (见二分图)
 * 3. 有组合/套餐的最大收益问题转化为最小割问题 (luoguP1361)
 * 4. 最小割点, 拆点 (包括 s 和 t, 最后跑 s+n 到 t 的最小割) 转化为最小割 (边)
 * 5. 牛吃饭问题, 一头牛有喜欢的饮料列表和食物列表, 求最大能满足的牛数
 *    牛拆点, 一边连饮料, 一边连食物, 跑最大流即可
 * 6. 火星探险/深海机器人问题, 在一个二维矩阵上走, 某些点第一次走到能获得收益, 可重复走, 求最大收益
 *    如果有多个起点和终点分别连上源汇点, 中间拆点, x 和 x'之间连两条边, <x,x',1,-1> 和 <x,x',INF,0>
 * 7. 跑多次网络流的情况, 如果无需重新建图, 也要记住将之前跑的流量重置, 可以在 Edge 结构体里设置一个 cap
 *    加边时 cap 和 w 相同, 重置时枚举所有边使 w=cap
 * 8. 输出方案考虑枚举边判断流量, 以及在 dfs 时记录前驱
 * 9. 注意 N 的大小, 注意顶点数 (特别是拆点后), 注意 s 和 t, 注意网络流中是有向边, 反边不等于无向边
 */
```

线段树

单点更新区间查询

```
//数组记得开 4 倍空间
//子节点信息更新父节点信息
void PushUp(int i){
    sum[i]=sum[lson]+sum[rson];
    Max[i]=max(Max[lson],Max[rson]);
}
//建树
void Build(int i,int l,int r){
    le[i]=l;
    ri[i]=r;
    if(l==r){
        //是 a[l] 不是 a[i]
        sum[i]=Max[i]=a[l];
        return;
    }
    int mid=(l+r)>>1;
    Build(lson,l,mid);
    Build(rson,mid+1,r);
    PushUp(i);
}
//单点更新 (累加/覆盖)
void Update(int i,int p,int v){
    if(le[i]==ri[i]){
        sum[i]+=v;
        Max[i]+=v;
        //sum[i]=Max[i]=v;
        return;
    }
    int mid=(le[i]+ri[i])>>1;
    if(p<=mid){
        Update(lson,p,v);
    }
    if(p>mid){
        Update(rson,p,v);
    }
    PushUp(i);
}
//区间查询 (和/最值)
int ans;
void Query(int i,int l,int r){
    if(l<=le[i] && ri[i]<=r){
        ans+=sum[i];
        //ans=max(ans,Max[i]);
        return;
    }
    if(le[i]==ri[i]){
        return;
    }
    int mid=(le[i]+ri[i])>>1;
    if(l<=mid){
        Query(lson,l,r);
    }
```

```
    if(r>mid){
        Query(rson,l,r);
    }
}
```

区间更新

```
//Build,PushUp,Update 同单点更新，主要多了 tag 数组以及将标记下放的 PushDown 函数
void PushDown(int i){
    if(tag[i]){
        //先累加父节点标记
        tag[lson]+=tag[i];
        tag[rson]+=tag[i];
        //和的累加使用的是父节点标记，因为自身的标记有可能是之前的累加
        sum[lson]+=(ri[lson]-le[lson]+1)*tag[i];
        sum[rson]+=(ri[rson]-le[rson]+1)*tag[i];
        //父节点标记记得清空
        tag[i]=0;
    }
}
void Update(int i,int l,int r,ll v){
    if(l<=le[i] && r>=ri[i]){
        //整个区间更新，无需更新每一个叶子
        tag[i]+=v;
        sum[i]+=(ri[i]-le[i]+1)*v;
        return;
    }
    //不同于单点更新，这里需要先下放标记，更新子节点信息，再递归下去查询
    PushDown(i);
    if(le[i]==ri[i]){
        return;
    }
    int mid=(le[i]+ri[i])>>1;
    if(l<=mid){
        Update(lson,l,r,v);
    }
    if(r>mid){
        Update(rson,l,r,v);
    }
    PushUp(i);
}
//hdu4027——区间开根号更新，区间开根号没有什么更好的方法，只能暴力到叶子
//但是可以加一个优化，但叶子的值为 1 时，标记 vis[i]=true，然后 PushUp 到父节点
//这样当遇到一个区间全为 1 时，就直接返回
void PushUp_sqrt(int i){
    tr[i].sum=tr[lson].sum+tr[rson].sum;
    //整个区间都是 1
    tr[i].vis=tr[lson].vis&&tr[rson].vis;
}
void Update(int i,int l,int r){
    if(tr[i].vis){
        return;
    }
    if(tr[i].l==tr[i].r){
        //暴力到叶子
        tr[i].sum=floor(sqrt(tr[i].sum));
```

```
        if(tr[i].sum==1){
            //标记为 1，无需再更新
            tr[i].vis=1;
        }
        return;
    }
    //......
}
```

多标记

```
//注意标记下传的顺序
//lson rson 同理，这里省略一个
void PushDown(int i){
    //set 优先级最高（假设存在多种标记，set 肯定是先来的，如果是后来的就会把 mul 和 add 覆盖）
    if(Set[i]){
        Set[lson]=Set[rson]=Set[i];   Add[lson]=Add[rson]=0;   Mul[lson]=Mul[rson]=1;
        sum1[lson]=((ri[lson]-le[lson]+1)*Set[i])%MOD;
        sum2[lson]=((((ri[lson]-le[lson]+1)*Set[i])%MOD)*Set[i])%MOD;
        sum3[lson]=(((((ri[lson]-le[lson]+1)*Set[i])%MOD)*Set[i]%MOD)*Set[i])%MOD;
        Set[i]=0;
    }
    if(Mul[i]!=1){
        Mul[lson]=(Mul[lson]*Mul[i])%MOD;   Mul[rson]=(Mul[rson]*Mul[i])%MOD;
        Add[lson]=(Add[lson]*Mul[i])%MOD;   Add[rson]=(Add[rson]*Mul[i])%MOD;
        sum3[lson]=(((((sum3[lson]*Mul[i])%MOD)*Mul[i])%MOD)*Mul[i])%MOD;
        sum2[lson]=(((sum2[lson]*Mul[i])%MOD)*Mul[i])%MOD;
        sum1[lson]=(sum1[lson]*Mul[i])%MOD;
        Mul[i]=1;
    }
    if(Add[i]){
        Add[lson]=(Add[lson]+Add[i])%MOD;   Add[rson]=(Add[rson]+Add[i])%MOD;
        sum3[lson]=(sum3[lson]+(3*sum2[lson]%MOD*Add[i]%MOD+(3*Add[i]%MOD*Add[i]%MOD
        *sum1[lson]%MOD+(Add[i]*Add[i]%MOD*Add[i]%MOD)*(ri[lson]-le[lson]+1))%MOD))%MOD;
        sum2[lson]=(sum2[lson]+(2*sum1[lson]%MOD*Add[i]%MOD
        +(Add[i]*Add[i])%MOD*(ri[lson]-le[lson]+1)))%MOD;
        sum1[lson]=(sum1[lson]+(ri[lson]-le[lson]+1)*Add[i]%MOD)%MOD;
        Add[i]=0;
    }
}
void Update(int i,int l,int r,int o,ll v){
    if(l<=le[i] && r>=ri[i]){
        if(o==1){
            //Add 操作，更新 Add 标记
            Add[i]=(Add[i]+v)%MOD;
            //(a1^3+a2^3+...+an^3)==>((a1+v)^3+(a2+v)^3+...+(an+v)^3)==... 化简
            sum3[i]=(sum3[i]+(3*sum2[i]%MOD*v%MOD+3*v%MOD*v%MOD*sum1[i]%MOD+v*v%MOD*v%MOD
            *(ri[i]-le[i]+1)%MOD))%MOD;
            //(a1^2+a2^2+...+an^2)==>((a1+v)^2+(a2+v)^2+...+(an+v)^2)==... 化简
            sum2[i]=(sum2[i]+(2*sum1[i]*v%MOD+v*v%MOD*(ri[i]-le[i]+1)%MOD))%MOD;
            //(a1+a2+...+an)==>(a1+v+a2+v+...+an+v)=(a1+a2+...+an)+n*v
            sum1[i]=(sum1[i]+(ri[i]-le[i]+1)*v%MOD)%MOD;
        }else if(o==2){
            //Mul 操作，更新 Mul，Add 标记
            Mul[i]=(Mul[i]*v)%MOD;
            Add[i]=(Add[i]*v)%MOD;
```

```
            sum3[i]=(((((sum3[i]*v)%MOD)*v)%MOD)*v)%MOD;
            sum2[i]=(((sum2[i]*v)%MOD)*v)%MOD;
            sum1[i]=(sum1[i]*v)%MOD;
        }else if(o==3){
            //Set 操作，更新 Set 标记，覆盖 Mul，Add 标记
            Set[i]=v;
            Add[i]=0;
            Mul[i]=1;
            sum1[i]=((ri[i]-le[i]+1)*v)%MOD;
            sum2[i]=((((ri[i]-le[i]+1)*v)%MOD)*v)%MOD;
            sum3[i]=(((((ri[i]-le[i]+1)*v)%MOD)*v%MOD)*v)%MOD;
        }
        return;
    }
    //......
}
```

权值线段树

```
/*
 * 权值线段树，维护的是值域的关系，普通线段树维护的是定义域的关系，相当于一颗反的线段树
 * 一般需要离散化 (ai>=1e7)，从而转化为离线数据结构
 */
//建树
void build(int i,int l,int r){
    le[i]=l;
    ri[i]=r;
    if(l==r){
        return;
    }
    int mid=(l+r)/2;
    build(lson,l,mid);
    build(rson,mid+1,r);
}
//插入或删除一个数
void update(int i,int p,int v){
    tr[i]+=v;
    if(le[i]==ri[i]){
        return;
    }
    int mid=(le[i]+ri[i])/2;
    if(p<=mid){
        update(lson,p,v);
    }else{
        update(rson,p,v);
    }
}
//查询一个数出现次数
int find(int i,int x){
    if(le[i]==ri[i]){
        return tr[i];
    }
    int mid=(le[i]+ri[i])/2;
    if(x<=mid){
        return find(lson,x);
```

```
    }else{
        return find(rson,x);
    }
}
//查询区间（值域区间）内数出现的次数
//比如查询 [1,3] 即 1 2 3 这三个数共出现了多少次
int ans;
void find(int i,int l,int r){
    if(l<=le[i] && r>=ri[i]){
        ans+=tr[i];
        return;
    }
    int mid=(le[i]+ri[i])/2;
    if(l<=mid){
        find(lson,l,r);
    }
    if(r>mid){
        find(rson,l,r);
    }
}
//查询全局第 k 大
int kth(int i,int k){
    if(le[i]==ri[i]){
        return le[i];
    }
    if(k<=tr[lson]){
        return kth(lson,k);
    }else{
        return kth(rson,k-tr[lson]);
    }
}
//查询数字 x 的排名
int rnk(int i,int x){
    if(le[i]==ri[i]){
        return 1;
    }
    int mid=(le[i]+ri[i])/2;
    if(x<=mid){
        return rnk(lson,x);
    }else{
        return tr[lson]+rnk(rson,x);
    }
}
//查询 x 的前驱（小于 x 且最大的数）
int Pre(int x){
    //如果离散化了，最外再套一个原数组 b[kth(...)]
    return kth(1,rnk(1,x)-1);
}
//查询 x 的后继（大于 x 最小的数）
int Nex(int x){
    return kth(1,rnk(1,x+1));
}
```

扫描线

矩形周长并

```cpp
#include <cstdio>
#include <algorithm>
using namespace std;
#define lson i<<1
#define rson i<<1|1
const int N=1e4+50;
struct Seg{
    //线段的左右端点 (x)，上下边 (1 下 -1 上)，高 (y)
    int l,r,d,h;
    Seg(){}
    Seg(int l,int r,int h,int d):l(l),r(r),h(h),d(d){}
    //从下到上处理线段
    bool operator <(const Seg& rhs)const{
        return h<rhs.h;
    }
}a[2][N];
int le[4*N],ri[4*N];
//边的覆盖，线段长度
int cnt[4*N],sum[4*N];
int all[2][N<<1];
int n;
int xa,ya,xb,yb;
void PushUp(int p,int i){
    if(cnt[i]){
        //i 有覆盖，那更新覆盖长度
        sum[i]=all[p][ri[i]+1]-all[p][le[i]];
    }else{
        sum[i]=sum[lson]+sum[rson];
    }
}
void Build(int i,int l,int r){
    le[i]=l;
    ri[i]=r;
    cnt[i]=sum[i]=0;
    if(l==r){
        return;
    }
    int mid=(l+r)>>1;
    Build(lson,l,mid);
    Build(rson,mid+1,r);
}
void Update(int p,int i,int l,int r,int v){
    if(l<=le[i] && r>=ri[i]){
        cnt[i]+=v;
        //记得 pushup，因为扫描线没有 query 操作，所以也无需懒惰标记，区间更新后直接推到根
        PushUp(p,i);
        return;
    }
    int mid=(le[i]+ri[i])>>1;
    if(l<=mid){
        Update(p,lson,l,r,v);
    }
```

```
    if(r>mid){
        Update(p,rson,l,r,v);
    }
    PushUp(p,i);
}
int main(void){
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        //矩形左下角和右上角坐标
        scanf("%d%d%d%d",&xa,&ya,&xb,&yb);
        //a[0] 保存横边 a[1] 保存竖边
        //all[0] 保存离散化的横坐标 all[1] 保存纵坐标
        a[0][i]=Seg(xa,xb,ya,1);
        a[0][i+n]=Seg(xa,xb,yb,-1);
        all[0][i]=xa;
        all[0][i+n]=xb;
        a[1][i]=Seg(ya,yb,xa,1);
        a[1][i+n]=Seg(ya,yb,xb,-1);
        all[1][i]=ya;
        all[1][i+n]=yb;
    }
    //别忘记扩大两倍
    n<<=1;
    //对线段排序
    sort(a[0]+1,a[0]+1+n);
    sort(a[1]+1,a[1]+1+n);
    //对坐标排序，为了离散化
    sort(all[0]+1,all[0]+1+n);
    sort(all[1]+1,all[1]+1+n);
    //去重
    int m[2];
    m[0]=unique(all[0]+1,all[0]+1+n)-all[0]-1;
    m[1]=unique(all[1]+1,all[1]+1+n)-all[1]-1;
    int ans=0;
    //求周长并需要计算横边和竖边
    for(int i=0;i<2;i++){
        Build(1,1,m[i]);
        //上一条边长度
        int last=0;
        for(int j=1;j<=n;j++){
            //离散化
            int l=lower_bound(all[i]+1,all[i]+1+m[i],a[i][j].l)-all[i];
            int r=lower_bound(all[i]+1,all[i]+1+m[i],a[i][j].r)-all[i];
            if(l>r){
                swap(l,r);
            }
            //区间更新
            Update(i,1,l,r-1,a[i][j].d);
            //根节点覆盖的总和即为当前高度的总边长
            ans+=abs(sum[1]-last);
            //减去前一次已经计算过的周长
            last=sum[1];
        }
    }
    printf("%d\n",ans);
    return 0;
```

```
}
```

矩形面积并

```
/*
 *  线段排序之后从下往上扫描，遇到下边（即 d=1），更新线段树对应区间（注意线段的 [1...3] 对应线段树 [1..2]）
 *  更新后计算当前线段树覆盖的总长度（即根节点的 sum），乘以下一条线段（上面一条）即得到一部分的面积
*/
#include <bits/stdc++.h>
using namespace std;
#define lson i<<1
#define rson i<<1|1
#define mid (l+r)/2
int n;
const int N=1005;
double xa,ya,xb,yb;
struct node{
    double l,r,h;
    int d;
    bool operator <(const node& rhs)const{
        return h<rhs.h;
    }
}a[N];
double all[N];
//覆盖边数量（即 d 值的累加）
int cnt[N];
//离散化区间所对应的真正长度
double sum[N];
void pushup(int i,int l,int r){
    if(cnt[i]){
        //注意实际的线段和线段树的区别
        //实际是 [1...2...3] 三个点两段
        //建树时最好是 [l,r-1]，也就是以段来建树
        //计算长度再恢复到 [1...3]
        sum[i]=all[r+1]-all[l];
    }else{
        sum[i]=sum[lson]+sum[rson];
    }
}
void build(int i,int l,int r){
    cnt[i]=sum[i]=0;
    if(l==r){
        return;
    }
    build(lson,l,mid);
    build(rson,mid+1,r);
}
void update(int i,int l,int r,int L,int R,int v){
    if(L<=l && R>=r){
        cnt[i]+=v;
        //一定要 pushup
        pushup(i,l,r);
        return;
    }
    if(l==r){
        return;
```

```
    }
    if(L<=mid){
        update(lson,l,mid,L,R,v);
    }
    if(R>mid){
        update(rson,mid+1,r,L,R,v);
    }
    pushup(i,l,r);
}
int main(void){
    int cas=1;
    while(~scanf("%d",&n)){
        if(n==0){
            break;
        }
        for(int i=1;i<=n;i++){
            scanf("%lf%lf%lf%lf",&xa,&ya,&xb,&yb);
            a[i]=node{xa,xb,ya,-1};
            a[i+n]=node{xa,xb,yb,1};
            all[i]=xa;
            all[i+n]=xb;
        }
        n*=2;
        sort(a+1,a+1+n);
        sort(all+1,all+1+n);
        int m=unique(all+1,all+1+n)-all-1;
        build(1,1,m);
        double ans=0;
        for(int i=1;i<n;i++){
            int l=lower_bound(all+1,all+1+m,a[i].l)-all;
            int r=lower_bound(all+1,all+1+m,a[i].r)-all;
            if(l<r){
                update(1,1,m,l,r-1,a[i].d);
            }
            //每次扫描到一条边映射到线段树上，然后计算部分面积
            //等于上面一条（即下一条）线段与该线段差值乘以线段树（根）覆盖区间实际长度
            ans+=sum[1]*(a[i+1].h-a[i].h);
        }
        printf("Test case #%d\n",cas++);
        printf("Total explored area: %.2lf\n\n",ans);
    }
    return 0;
}
```

矩形面积交

```
/*
 *  线段排序之后从下往上扫描，遇到下边（即 d=1），更新线段树对应区间（注意线段的 [1...3] 对应线段树 [1..2]）
 *  更新后计算当前线段树覆盖两次的总长度（即根节点的 two），乘以下一条线段（上面一条）即得到一部分的面积
*/
#include <bits/stdc++.h>
using namespace std;
#define lson i<<1
#define rson i<<1|1
const int N=1050;
int t,n;
```

```
double xa,ya,xb,yb;
struct Seg{
    double l,r,h;
    int d;
    Seg(){}
    Seg(double l,double r,double h,int d):l(l),r(r),h(h),d(d){}
    bool operator <(const Seg& rhs)const{
        return h<rhs.h;
    }
}a[N<<1];
double all[N<<1];
int le[4*N],ri[4*N];
//覆盖边数量 (即 d 值的累加)
int cnt[4*N];
//区间覆盖 1 次和 2 次的真正长度
double one[4*N],two[4*N];
//扫描线中的 PushUp 是用来通过 cnt 来更新 sum(或者 one,two)
void PushUp(int i){
    if(cnt[i]>=2){
        //覆盖两次
        two[i]=one[i]=all[ri[i]+1]-all[le[i]];
    }else if(cnt[i]==1){
        //覆盖一次
        one[i]=all[ri[i]+1]-all[le[i]];
        if(le[i]==ri[i]){
            two[i]=0;
        }else{
            //该区间覆盖一次，子节点又覆盖一次
            two[i]=one[lson]+one[rson];
        }
    }else{
        if(le[i]==ri[i]){
            two[i]=one[i]=0;
        }else{
            two[i]=two[lson]+two[rson];
            one[i]=one[lson]+one[rson];
        }
    }
}
void Build(int i,int l,int r){
    le[i]=l;
    ri[i]=r;
    cnt[i]=one[i]=two[i]=0;
    if(l==r){
        return;
    }
    int mid=(l+r)>>1;
    Build(lson,l,mid);
    Build(rson,mid+1,r);
}
void Update(int i,int l,int r,int v){
    if(l<=le[i] && r>=ri[i]){
        cnt[i]+=v;
        PushUp(i);
        return;
    }
```

```
    int mid=(le[i]+ri[i])>>1;
    if(l<=mid){
        Update(lson,l,r,v);
    }
    if(r>mid){
        Update(rson,l,r,v);
    }
    PushUp(i);
}
int main(void){
    scanf("%d",&t);
    while(t--){
        scanf("%d",&n);
        for(int i=1;i<=n;i++){
            scanf("%lf%lf%lf%lf",&xa,&ya,&xb,&yb);
            a[i]=Seg(xa,xb,ya,1);
            a[i+n]=Seg(xa,xb,yb,-1);
            all[i]=xa;
            all[i+n]=xb;
        }
        n<<=1;
        sort(a+1,a+1+n);
        sort(all+1,all+1+n);
        int m=unique(all+1,all+1+n)-all-1;
        Build(1,1,m);
        double ans=0;
        for(int i=1;i<n;i++){
            int l=lower_bound(all+1,all+1+m,a[i].l)-all;
            int r=lower_bound(all+1,all+1+m,a[i].r)-all;
            if(l<r){
                Update(1,l,r-1,a[i].d);
            }
            ans+=two[1]*(a[i+1].h-a[i].h);
        }
        printf("%.2lf\n",ans);
    }
    return 0;
}
```

立方体体积交

```
/*
 * 三维扫描线就是枚举 z 坐标然后做二维的扫描线
 */
#include <bits/stdc++.h>
using namespace std;
#define lson i<<1
#define rson i<<1|1
typedef long long ll;
const int N=2550;
struct Seg{
    int l,r,h,d;
    bool operator <(const Seg& rhs)const{
        return h<rhs.h;
    }
}a[N];
```

```
//立方体
struct Rect{
    int xa,ya,za,xb,yb,zb;
}r[N];
int le[4*N],ri[4*N];
int cnt[4*N],one[4*N],two[4*N],three[4*N];
int X[N],Z[N];
void PushUp(int i){
        int l=le[i],r=ri[i];
        if (cnt[i]>=3){
                three[i] = X[r+1]-X[l];
                two[i]=one[i]=0;
        }else if(cnt[i]==2){
                three[i]=three[lson]+three[rson]+two[lson]+two[rson]+one[lson]+one[rson];
                two[i]=X[r+1]-X[l]-three[i];
                one[i]=0;
        }else if(cnt[i]==1) {
                three[i]=three[lson]+three[rson]+two[lson]+two[rson];
                two[i]=one[lson]+one[rson];
                one[i]=X[r+1]-X[l]-three[i]-two[i];
        }else{
                three[i]=three[lson]+three[rson];
                two[i]=two[lson]+two[rson];
                one[i]=one[lson]+one[rson];
        }
}
void Build(int i,int l,int r){
    le[i]=l;
    ri[i]=r;
    cnt[i]=one[i]=two[i]=three[i]=0;
    if(l==r){
        return;
    }
    int mid=(l+r)>>1;
    Build(lson,l,mid);
    Build(rson,mid+1,r);
}
void Update(int i,int l,int r,int v){
    if(l<=le[i] && r>=ri[i]){
        cnt[i]+=v;
        PushUp(i);
        return;
    }
    int mid=(le[i]+ri[i])>>1;
    if(l<=mid){
        Update(lson,l,r,v);
    }
    if(r>mid){
        Update(rson,l,r,v);
    }
    PushUp(i);
}
int t,n;
int xa,xb,ya,yb,za,zb;
int main(void){
    scanf("%d",&t);
```

```
    for(int cas=1;cas<=t;cas++){
        scanf("%d",&n);
        //x z 坐标的个数
        int cnt_x=0;
        int cnt_z=0;
        //点的个数
        int pcnt=0;
        for(int i=1;i<=n;i++){
            scanf("%d%d%d%d%d%d",&xa,&ya,&za,&xb,&yb,&zb);
            r[pcnt++]=Rect{xa,ya,za,xb,yb,zb};
            X[cnt_x++]=xa;
            X[cnt_x++]=xb;
            Z[cnt_z++]=za;
            Z[cnt_z++]=zb;
        }
        //离散化 z 用于枚举每一层
        //离散化 x 用于每一层的扫描线
        sort(X,X+cnt_x);
        sort(Z,Z+cnt_z);
        cnt_x=unique(X,X+cnt_x)-X;
        cnt_z=unique(Z,Z+cnt_z)-Z;
        ll V=0;
        //枚举离散化后的 z
        for(int i=0;i<cnt_z-1;i++){
            //这一层边的数量
            int k=0;
            ll S=0;
            //枚举立方体
            for(int j=0;j<pcnt;j++){
                //找处于 Z[i] 位置的立方体 映射在 Z[i] 平面上进行二维的扫描线
                if(r[j].za<=Z[i] && r[j].zb>Z[i]){
                    a[k++]=Seg{r[j].xa,r[j].xb,r[j].ya,1};
                    a[k++]=Seg{r[j].xa,r[j].xb,r[j].yb,-1};
                }
            }
            sort(a,a+k);
            //这里如果 build 1 到 cnt_x 会 t
            Build(1,1,N);
            for(int j=0;j<k-1;j++){
                int ql=lower_bound(X,X+cnt_x,a[j].l)-X;
                int qr=lower_bound(X,X+cnt_x,a[j].r)-X;
                Update(1,ql,qr-1,a[j].d);
                S+=(ll)three[1]*(ll)(a[j+1].h-a[j].h);
            }
            V+=S*(ll)(Z[i+1]-Z[i]);
        }
        printf("Case %d: %lld\n",cas,V);
    }
}
```

区间合并

```
/**
 * 线段树维护最大连续区间
 */
```

```cpp
#include <bits/stdc++.h>
using namespace std;
#define lson i<<1
#define rson i<<1|1
const int N=5e4+50;
int n,m;
int le[4*N],ri[4*N];
//左最大连续区间 右最大连续区间 总最大连续区间
int lm[4*N],rm[4*N],mx[4*N];
void PushUp(int i){
    lm[i]=lm[lson];
    rm[i]=rm[rson];
    mx[i]=max(max(mx[lson],mx[rson]),rm[lson]+lm[rson]);
    //左儿子区间满，父节点左区间要加上右儿子左区间
    if(mx[lson]==ri[lson]-le[lson]+1){
        lm[i]+=lm[rson];
    }
    //右儿子区间满，父节点右区间要加上左儿子右区间
    if(mx[rson]==ri[rson]-le[rson]+1){
        rm[i]+=rm[lson];
    }
}
void Build(int i,int l,int r){
    le[i]=l;
    ri[i]=r;
    lm[i]=rm[i]=mx[i]=r-l+1;
    if(l==r){
        return;
    }
    int mid=(l+r)>>1;
    Build(lson,l,mid);
    Build(rson,mid+1,r);
}
void Update(int i,int l,int r,int v){
    if(le[i]==ri[i]){
        //修复 1 破坏 0
        lm[i]=rm[i]=mx[i]=v;
        return;
    }
    int mid=(le[i]+ri[i])>>1;
    if(l<=mid){
        Update(lson,l,r,v);
    }
    if(r>mid){
        Update(rson,l,r,v);
    }
    PushUp(i);
}
int ans;
void Query(int i,int t){
    //叶子 最大连续区间为空 最大连续区间为整段
    if(le[i]==ri[i] || mx[i]==0 || mx[i]==ri[i]-le[i]+1){
        ans+=mx[i];
        return;
    }
    int mid=(le[i]+ri[i])>>1;
```

```
    //看左子树
    if(t<=mid){
        if(t>=ri[lson]-rm[lson]+1){
            //t 在左子树的右区间内
            Query(lson,t);
            Query(rson,mid+1);
        }else{
            //t 不在左子树的右区间内
            Query(lson,t);
        }
    }else{
        if(t<=le[rson]+lm[rson]-1){
            //t 在右子树的左区间内
            Query(rson,t);
            Query(lson,mid);
        }else{
            //t 不在左子树的右区间内
            Query(rson,t);
        }
    }
}
int main(void){
    while(~scanf("%d%d",&n,&m)){
        char s[2];
        int x;
        Build(1,1,n);
        stack<int> sta;
        while(m--){
            scanf("%s",s);
            if(s[0]=='D'){
                scanf("%d",&x);
                sta.push(x);
                Update(1,x,x,0);
            }else if(s[0]=='Q'){
                scanf("%d",&x);
                ans=0;
                Query(1,x);
                printf("%d\n",ans);
            }else if(s[0]=='R'){
                x=sta.top();
                sta.pop();
                Update(1,x,x,1);
            }
        }
    }
}
```

其他常见模型

```
/*
 * 0. poj2528: 离散化 (注意线段相接的判断)+ 染色 (区间更新，查询时暴力到叶子)
 * 1. zoj1610: 染色 (区间更新，查询暴力到叶子，维护前一个颜色，判断颜色是否中断)
 * 2. hdu3974: dfs 序建线段树 (dfs 一遍记录每个点‘进去’和‘出来’的时间戳，看成一个区间建树)
 *             区间维护的是子树的状态，因为对于每一个节点来说，进和出的中间就是访问子树
 * 3. hdu4614: 常规线段树 + 二分
```

```
 */
```

RMQ

```
/*
 * 预处理 O(nlogn)
 * dp[i][j]: 从 a[i] 开始 2^j 个数的最小值
 * 数组下标从 1 开始，注意-1 +1 <= 等细节
 */
void init(){
    for(int i=1;i<=n;i++){
        dp[i][0]=a[i];
    }
    for(int j=1;(1<<j)<=n;j++){
        for(int i=1;i+(1<<(j-1))-1<=n;i++){
            dp[i][j]=max(dp[i][j-1],dp[i+(1<<(j-1))][j-1]);
        }
    }
}
/*
 * 查询 [l,r] 最大值
 */
int rmq(int l,int r){
    int k=0;
    //保证刚好 [l,l+2^k] 和 [r-2^k,r] 重叠
    while((1<<(k+1))<=r-l+1){
        k++;
    }
    return max(dp[l][k],dp[r-(1<<k)+1][k]);
}
```

树状数组

单点更新区间求和

```
/*
 * 树状数组 (普通版): 单点更新，区间求和 (sum(r)-sum(l-1))
 * c[i]: 树状数组，c[i]=a[i-lowbit(i)+1]+a[i-lowbit(i)+2]+...+a[i]
 */
int lowbit(int x){
    return x&(-x);
}
int add(int i,int x){
    while(i<=n){
        c[i]+=x;
        i+=lowbit(i);
    }
}
int sum(int i){
    int ans=0;
    while(i>0){
        //这里也可以维护最值 ans=max(ans,c[i]);
        ans+=c[i];
        i-=lowbit(i);
    }
```

```
    return ans;
}
```

求逆序数

```
/*
 * 离散化 + 树状数组求逆序数
 */
int solve(){
    for(int i=1;i<=n;i++){
        int k=lower_bound(b+1,b+n+1,a[i])-b;
        add(k,1);
        ans+=i-sum(k);
    }
    return ans;
}
```

区间更新单点查询

```
/*
 * 树状数组 (加强版 1): 区间更新, 单点查询 (sum(i))
 * 设 d[i]=a[i]-a[i-1] 所以 a[i]=d[1]+d[2]+...+d[i]
 * 树状数组 c 维护 d 的前缀和, 也相当于变相维护了 a[i], 所以单点查询是查询 sum
 */
void update(int l,int r){
    add(l,1);
    add(r+1,-1);
}
```

区间更新区间求和

```
/*
 * 树状数组 (加强版 2): 区间更新, 区间求和
 * 维护两个树状数组 c1 保存 d[i] 的前缀和 c2 保存 d[i]*i 的前缀和
 * 求和 ans=\sum[((k+1)*c1[i]-c2[i])]
 */
void add(int i,int x){
    int k=i;
    while(i<=n){
        c1[i]+=x;
        c2[i]+=k*x;
        i+=lowbit(i);
    }
}
void add_range(int l,int r,int x){
    add(l,x);
    add(r+1,-x);
}
int sum(int i){
    int ans=0;
    int k=i;
    while(i>0){
        ans+=((k+1)*c1[i]-c2[i]);
        i-=lowbit(i);
    }
    return ans;
```

```
}
int ask_range(int l,int r){
    return sum(r)-sum(l-1);
}
```

二维树状数组

单点更新区间求和

```
/*
 * 二维数组数组 (I)，单点更新，区间求和 (sum(x2,y2)-sum(x1-1,y2)-sum(x2,y1-1)+sum(x1-1,y1-1))
 */
void add(int x,int y,int z){
    for(int i=x;i<=n;i+=lowbit(i)){
        for(int j=y;j<=m;j+=lowbit(j)){
            c[i][j]+=z;
        }
    }
}
int sum(int x,int y){
    int ans=0;
    for(int i=x;i>=1;i-=lowbit(i)){
        for(int j=y;j>=1;j-=lowbit(j)){
            ans+=c[i][j];
        }
    }
    return ans;
}
```

区间更新单点查询

```
/*
 * 二维树状数组 (II): 区间更新，单点查询 (sum(x,y))
 * 和一维的一样也是用树状数组维护一个差分数组
 */
void add_range(int x1,int y1,int x2,int y2,int w){
    add(x1,y1,w);
    add(x2+1,y2+1,w);
    add(x2+1,y1,-w);
    add(x1,y2+1,-w);
}
```

莫队

普通莫队

```
/*
 * 莫队是一种基于分块思想的离线算法，用于解决区间问题，适用范围如下：
 * 0. 只有询问没有修改。
 * 1. 允许离线。
 * 2. 在已知 [l,r] 答案的情况下可以 O(1) 得到 [l,r-1],[l,r+1],[l-1,r],[l+1,r] 的答案。
 *
 * 有 n 个不同颜色的袜子，多个询问在区间 [l,r] 里随机抽两个袜子，同一颜色的概率是多少
 * 对区间长度为 n 的询问来说，答案为 (sum(ai^2)-n)/n*(n-1)，其中 ai 为颜色 i 在区间内的个数
 * 分母 n*(n-1) 表示抽的所有可能情况 (第一只有 n 种选择，第二只有 n-1 种)
 * 分子同理为 sum(ai*(ai-1))=sum(ai^2)-sum(ai)=sum(ai^2)-n
```

```
 */
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=5e4+50;
int pw(int x){
    return x*x;
}
int n,m;
int be[N];
ll a[N],sum[N];
struct Q{
    int id,l,r;
    ll zi,mu;
}q[N];
bool cmp1(Q a,Q b){
    //左端点同块按右端点排序，否则按左端点排序
    return be[a.l]==be[b.l]?a.l<b.l:a.r<b.r;
}
bool cmp2(Q a,Q b){
    return a.id<b.id;
}
int ans;
//查询的答案就是 (sum(ai^2)-n)/(n(n-1))
void fun(int i,int x){
    //更新分子 sum(ai^2) 部分
    ans-=pw(sum[a[i]]);
    sum[a[i]]+=x;
    ans+=pw(sum[a[i]]);
}
int main(void){
    scanf("%d%d",&n,&m);
    int unit=sqrt(n);
    for(int i=1;i<=n;i++){
        scanf("%lld",&a[i]);
        //分块
        be[i]=i/unit+1;
    }
    for(int i=0;i<m;i++){
        scanf("%d%d",&q[i].l,&q[i].r);
        q[i].id=i;
    }
    //对查询的区间预处理排序
    sort(q,q+m,cmp1);
    //上一次查询的区间
    int l=1,r=0;
    for(int i=0;i<m;i++){
        //左端点移动到当前查询区间左端点，同时更新答案
        while(l<q[i].l){
            fun(l,-1);
            l++;
        }
        //以下同理，但要注意边界问题 l l-1 r+1 r
        while(l>q[i].l){
            fun(l-1,1);
            l--;
```

```
        }
        while(r<q[i].r){
            fun(r+1,1);
            r++;
        }
        while(r>q[i].r){
            fun(r,-1);
            r--;
        }
        //统计答案
        ll len=(q[i].r-q[i].l+1)*1LL;
        q[i].zi=ans-len;
        q[i].mu=len*(len-1);
        ll g=__gcd(q[i].zi,q[i].mu);
        q[i].zi/=g;
        q[i].mu/=g;
    }
    sort(q,q+m,cmp2);
    for(int i=0;i<m;i++){
        printf("%lld/%lld\n",q[i].zi,q[i].mu);
    }
    return 0;
}
```

带修莫队

```
/*
 * 有 n 个不同颜色，两种操作，修改某个位置的颜色，查询某个区间 [l,r] 里不同颜色个数
 * 比普通的莫队多了一个修改的操作，所以 Q 结构体多了一个时间戳 tim
 * 排序的时候也要多加一个判断，在 l，r 都同块的情况下对 tim 排序
 * 除了询问要保存时间，修改也要在对应时间保存起来，记录原来的颜色和修改后的颜色
 * 查询答案时，先对时间 T 进行调整，再套用普通莫队
 */
#include <bits/stdc++.h>
using namespace std;
const int N=1e5+50;
int n,m,a[N],be[N];
struct Q{
    //多了一个 tim 类似于保存版本号
    int id,l,r,tim,ans;
}q[N];
bool cmp1(Q a,Q b){
    return be[a.l]==be[b.l]?(be[a.r]==be[b.r]?a.tim<b.tim:a.r<b.r):a.l<b.l;
}
bool cmp2(Q a,Q b){
    return a.id<b.id;
}
struct Change{
    int pos,New,Old;
}c[N];
char qe[2];
int x,y;
int Time;
int now[N];
int ans,color[N*100],l,r;
void revise(int x,int d){
```

```
    //这里的 x 对应 going 的 d, 而 d 则是代表加或减
    //x 这种颜色的个数
    color[x]+=d;
    if(d>0){
        //刚好第一次出现这种颜色, 更新答案
        ans+=color[x]==1;
    }
    if(d<0){
        //这种颜色刚好被删完, 更新答案
        ans-=color[x]==0;
    }
}
//带时间修改, 将 x 的颜色改为 d
void going(int x,int d){
    //x 如果在当前的区间内
    if(l<=x && x<=r){
        //更新答案
        revise(d,1);
        revise(a[x],-1);
        //修改颜色
        a[x]=d;
    }
}
int main(void){
    scanf("%d%d",&n,&m);
    int unit=sqrt(n);
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i]);
        be[i]=i/unit+1;
        now[i]=a[i];
    }
    int qcnt=0;
    for(int i=1;i<=m;i++){
        scanf("%s%d%d",qe,&x,&y);
        if(qe[0]=='Q'){
            q[++qcnt]=Q{qcnt,x,y,Time,0};
        }else if(qe[0]=='R'){
            //记录修改状态
            c[++Time]=Change{x,y,now[x]};
            now[x]=y;
        }
    }
    sort(q+1,q+1+qcnt,cmp1);
    l=1,r=0;
    int T=0;
    for(int i=1;i<=qcnt;i++){
        //维护时间 T
        //q[i].tim 表示在这个时刻发生的查询, 所以维护当前时刻 T 直到为 tim
        while(T<q[i].tim){
            //所以在这个过程中要执行修改
            going(c[T+1].pos,c[T+1].New);
            T++;
        }
        while(T>q[i].tim){
            going(c[T].pos,c[T].Old);
            T--;
```

```
        }
        //时间维护完，维护空间顺序，同普通莫队
        while(l<q[i].l){
            revise(a[l],-1);
            l++;
        }
        while(l>q[i].l){
            revise(a[l-1],1);
            l--;
        }
        while(r<q[i].r){
            revise(a[r+1],1);
            r++;
        }
        while(r>q[i].r){
            revise(a[r],-1);
            r--;
        }
        q[i].ans=ans;
    }
    sort(q+1,q+1+qcnt,cmp2);
    for(int i=1;i<=qcnt;i++){
        printf("%d\n",q[i].ans);
    }
    return 0;
}
```

树上莫队

```
/*
 * 给一棵树，树上每个节点有一个糖果，一共有 m 种糖果美味度分别为 vi，每个游客第 i 次吃同种糖果的新奇度 wi
 * 多组询问：l 到 r 路径上的 sum(vi*wi) (注意 wi 是会随着同种糖果的多次访问而变化)
 */
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=1e5+50;
int n,m,q,l,r,t,cnt1,cnt2,tot,clk;
vector<int> g[N];
int be[N<<1];
int pw[25];
//用于倍增求 lca f[u][i] 表示从 u 往上跳 2^i 格可以到达的点
int fa[N][17];
int c[N],d[N];
int v[N],w[N],now[N];
//表示第几次访问这个糖果，从而间接访问了 wi
int u[N];
//标记是否访问，每次访问就 ^1，这样子树的节点一进一出最后就不会影响结果的贡献
//(因为 l 和 r 的路径是不含有 l 或 r 的子树的，但是 dfs 序有)
bool vis[N];
struct Q{
    int id,l,r,tim;
}a[N];
struct Change{
    int pos,New,Old;
}ch[N];
```

```
ll ans[N],sum;
//dfs 序
int id[N<<1],in[N],out[N];
void dfs(int u){
    //双向映射
    in[u]=++clk;
    id[clk]=u;
    //倍增
    for(int i=1;pw[i]<=d[u];i++){
        fa[u][i]=fa[fa[u][i-1]][i-1];
    }
    for(int i=0;i<g[u].size();i++){
        int v=g[u][i];
        if(v!=fa[u][0]){
            //fa[v][0] 即 v 往上跳 1 即 v 的父节点
            fa[v][0]=u;
            //深度
            d[v]=d[u]+1;
            dfs(v);
        }
    }
    out[u]=++clk;
    id[clk]=u;
}
int lca(int x,int y){
    if(d[x]<d[y]){
        swap(x,y);
    }
    int tmp=d[x]-d[y];
    for(int i=0;pw[i]<=tmp;i++){
        //往上跳
        if(tmp&pw[i]){
            x=fa[x][i];
        }
    }
    //跳到刚好，即 y=lca(x,y)
    if(x==y){
        return x;
    }
    for(int i=16;i>=0;i--){
        //x 和 y 一起跳
        if(fa[x][i]!=fa[y][i]){
            x=fa[x][i];
            y=fa[y][i];
        }
    }
    return fa[x][0];
}
bool cmp(Q x,Q y){
    return be[x.l]==be[y.l]?(be[x.r]==be[y.r]?x.tim<y.tim:x.r<y.r):x.l<y.l;
}
//修改节点，即修改答案贡献
void revise(int x){
    if(vis[x]){
        //之前访问过，那再修改的话，就要把前面计算的贡献减去
        sum-=1ll*v[c[x]]*w[u[c[x]]--];
```

```
    }else{
        //反之加上贡献
        sum+=1ll*v[c[x]]*w[++u[c[x]]];
    }
    //标记访问一次，状态反转
    vis[x]^=1;
}
//在这个时间点，把节点 x 修改为颜色 y
void going(int x,int y){
    if(vis[x]){
        //之前访问过的，必须修改贡献
        revise(x);
        //先减贡献，修改节点值，再加贡献
        c[x]=y;
        revise(x);
    }else{
        //之前没访问过的直接修改
        c[x]=y;
    }
}
int main(){
    scanf("%d%d%d",&n,&m,&q);
    pw[0]=1;
    for(int i=1;i<=17;++i){
        pw[i]=pw[i-1]*2;
    }
    for(int i=1;i<=m;++i){
        scanf("%d",&v[i]);
    }
    for(int i=1;i<=n;++i){
        scanf("%d",&w[i]);
    }
    //建树
    for(int i=1;i<n;++i){
        int l,r;scanf("%d%d",&l,&r);
        g[l].push_back(r);
        g[r].push_back(l);
    }
    for(int i=1;i<=n;++i){
        scanf("%d",&c[i]);
        //now 记录当前状态用于修改和恢复
        now[i]=c[i];
    }
    //分块单位
    int unit=pow(n,2.0/3);
    //构造 dfs 序，将树结构转化为序列
    dfs(1);
    for(int i=1;i<=clk;++i){
        //分块
        be[i]=i/unit+1;
    }
    while(q--){
        scanf("%d%d%d",&t,&l,&r);
        if(t){
            //询问
            if(in[l]>in[r]){
```

```
                swap(l,r);
            }
            a[++cnt1]=Q{cnt1,(lca(l,r)==l)?in[l]:out[l],in[r],cnt2};
        }else{
            //修改，把 l 的值修改成 r，原来的值是 now[l]
            ch[++cnt2]=Change{l,r,now[l]};
            now[l]=r;
        }
    }
    sort(a+1,a+1+cnt1,cmp);
    //初始化空区间 l=1 r=0
    l=1,r=0,t=0;
    for(int i=1;i<=cnt1;++i){
        //同普通带修莫队，先处理时序
        while(t<a[i].tim){
            going(ch[t+1].pos,ch[t+1].New);
            t++;
        }
        while(t>a[i].tim){
            going(ch[t].pos,ch[t].Old);
            t--;
        }
        //再处理空间，注意要用 id 映射，同普通带修莫队
        while(l<a[i].l){
            revise(id[l]);
            l++;
        }
        while(l>a[i].l){
            revise(id[l-1]);
            l--;
        }
        while(r<a[i].r){
            revise(id[r+1]);
            r++;
        }
        while(r>a[i].r){
            revise(id[r]);
            r--;
        }
        //树上莫队特殊处理
        int x=id[l],y=id[r],tmp=lca(x,y);
        if(x!=tmp&&y!=tmp){
            //xy 分别在 lca 两边，则 lca 这个点只算了一次，需要再更新一次贡献
            //(可以写个 dfs 序看下，lca 的 dfs 出序刚好是区间右端点 +1)
            revise(tmp);
            ans[a[i].id]=sum;
            revise(tmp);
        }else{
            //x 或 y 是 lca(x,y)，直接记录答案
            ans[a[i].id]=sum;
        }
    }
    for(int i=1;i<=cnt1;i++){
        printf("%lld\n",ans[i]);
    }
}
```

可持久化字典树

树上路径异或最大值

```
/*
 * 可持久化字典树查询树上任意两个节点路径间某个数与所给值的最大异或值
 * rt[i]: i 节点对应的 trie 树的根
 * ch[][],sz[],tot: 字典树，注意空间开大，sz[i] 表示根节点到该节点形成的前缀出现的累计的次数
 * d[],fa[][]: 倍增求 lca
 */
//创建字典树节点
int newnode(){
    memset(ch[tot],0,sizeof(ch[tot]));
    sz[tot]=0;
    return tot++;
}
void insert(int p,int u,int val){
    //每个节点对应创建一颗字典树,dfs 序对应的根的顺序
    p=rt[p];
    u=rt[u];
    for(int i=31;i>=0;i--){
        int id=(val>>i)&1;
        if(!ch[u][id]){
            //当前 trie 树的节点，剩下的指针指向上一棵树的对应节点位置
            ch[u][id]=newnode();
            ch[u][id^1]=ch[p][id^1];
            //复制上一棵树信息
            sz[ch[u][id]]=sz[ch[p][id]];
        }
        u=ch[u][id];
        p=ch[p][id];
        //u 就是当前的 trie 树特有的节点，所以 sz++
        sz[u]++;
    }
}
//dfs 序建可持久化 trie 树
void dfs(int u){
    //u 的祖先节点信息都已知，所以同时处理出 fa[u]
    for(int i=1;(1<<i)<=d[u];i++){
        fa[u][i]=fa[fa[u][i-1]][i-1];
    }
    rt[u]=newnode();
    insert(fa[u][0],u,a[u]);
    int siz=g[u].size();
    for(int i=0;i<siz;i++){
        int v=g[u][i];
        if(v==fa[u][0]){
            continue;
        }
        fa[v][0]=u;
        d[v]=d[u]+1;
        dfs(v);
    }
}
int lca(int x,int y){
    //lca 省略
```

```
}
int query(int x,int y,int val){
    int lc=lca(x,y);
    //单独考虑 lca 的异或值，因为后面减去两倍 lca 的贡献，等于除去了 lca
    int res=a[lc]^val;
    x=rt[x];
    y=rt[y];
    lc=rt[lc];
    int ans=0;
    for(int i=31;i>=0;i--){
        int id=(val>>i)&1;
        //类似于求两点间的距离 dis[u]+dis[v]-2*dis[lca(u,v)]
        //dis 表示到根的距离，而这里 sz 表示的是点到根的所有字典树的这一位上这个值的前缀和
        if(sz[ch[x][id^1]]+sz[ch[y][id^1]]-2*sz[ch[lc][id^1]]>0){
            ans+=(1<<i);
            id=id^1;
        }
        x=ch[x][id];
        y=ch[y][id];
        lc=ch[lc][id];
    }
    return max(ans,res);
}
```

子树异或最大值

```
/*
 * 可持久化字典树查询树上某个节点子树的某个数与所给值的最大异或值
 * 先 dfs(1,0) 求 dfs 序
 * 再对于 dfs 序的每个节点权值 a[mp[i]]，建字典树
 * rt[i]=newnode();
 * insert(i-1,i,a[mp[i]]);
 */
void dfs(int u,int f){
    in[u]=++cnt;
    mp[cnt]=u;
    int siz=g[u].size();
    for(int i=0;i<siz;i++){
        int v=g[u][i];
        if(v==f){
            continue;
        }
        dfs(v,u);
    }
    //注意这里出树是 cnt 不是 ++cnt，这样就不会重复建树
    //这样 in[u] out[u] 表示的就是 u 的子树的范围
    out[u]=cnt;
}
int ch[N*100][2],sz[N*100],tot,rt[N];
void init(){
    //注意初始化
    cnt=0;
    tot=1;
    memset(ch,0,sizeof(ch));
    for(int i=1;i<=n;i++){
        g[i].clear();
```

```
    }
}
int newnode(){
    memset(ch[tot],0,sizeof(ch[tot]));
    sz[tot]=0;
    return tot++;
}
void insert(int p,int u,int val){
    p=rt[p];
    u=rt[u];
    for(int i=30;i>=0;i--){
        int id=(val>>i)&1;
        if(!ch[u][id]){
            ch[u][id]=newnode();
            ch[u][id^1]=ch[p][id^1];
            sz[ch[u][id]]=sz[ch[p][id]];
        }
        u=ch[u][id];
        p=ch[p][id];
        sz[u]++;
    }
}
int query(int u,int x){
    //前缀和思想
    int l=rt[in[u]-1];
    int r=rt[out[u]];
    int ans=0;
    for(int i=30;i>=0;i--){
        int id=(x>>i)&1;
        //贪心求异或最大
        if(sz[ch[r][id^1]]-sz[ch[l][id^1]]>0){
            ans+=(1<<i);
            id^=1;
        }
        l=ch[l][id];
        r=ch[r][id];

    }
    return ans;
}
```

双指针

一维

```
/*
 * 封装成一个滑窗结构体，维护区间内不同字母个数（根据题目需要）
 */
struct window{
    int siz;
    int cnt[26];
    window(){
        siz=0;
        memset(cnt,0,sizeof(cnt));
    }
    void add(int x){
        if(!cnt[x]){
            siz++;
        }
        cnt[x]++;
    }
    void remove(int x){
        cnt[x]--;
        if(!cnt[x]){
            siz--;
        }
    }
};
void solve(char s[],int n,int k){
        window w;
    int n=strlen(s);
    int l=0,r=0;
    ll ans=0;
    while(l<n){
        //右指针速度大于左指针
        while(w.siz<k && r<n){
            w.add(s[r++]-'a');
        }
        if(w.siz<k){
            break;
        }
        //同时更新答案
        ans+=(n-r+1);
        w.remove(s[l++]-'a');
    }
}
```

二维

```
/*
 * 二维尺取法（枚举一维）：求满足 01 矩阵中 1 个数大于等于 k 的最小矩形大小
 * pre[i][j]：从 (1,1) 到 (i,j) 的 1 个数（二维前缀和）
 */
void solve(){
    int ans=INF;
    //枚举列，确定矩形左右边界
    for(int i=1;i<=m;i++){
```

```
        for(int j=1;j<=m;j++){
            //尺取
            int l=1,r=1;
            while(r<=n){
                while(pre[r][j]-pre[r][i-1]-pre[l-1][j]+pre[l-1][i-1]<k && r<=n){
                    r++;
                }
                if(pre[r][j]-pre[r][i-1]-pre[l-1][j]+pre[l-1][i-1]<k){
                    break;
                }
                ans=min(ans,(j-i+1)*(r-l+1));
                l++;
            }
        }
    }
}
```

单调队列/单调栈

最大 m 子段和

```
/*
 * 一般来说单调栈就是半个单调队列
 * 子段长度小于等于 m 的最大和
 * pre[]: 前缀和数组
 * q: 单调队列 (双端), 维护下标, 对应的前缀和从小到大
 */
int solve(int n,int m){
    deque<int> q;
    q.push_back(1);
    int ans=0;
    int i=2;
    while(i<=n){
        //维护单调性
        while(!q.empty() && pre[q.back()]>pre[i]){
            q.pop_back();
        }
        //后来的下标总是放在队尾
        q.push_back(i);
        //把老的下标 (超过 i-m 范围) 出队
        while(!q.empty() && q.front()<i-m){
            q.pop_front();
        }
        ans=max(ans,pre[i]-pre[q.front()]);
        i++;
    }
    return ans;
}
```

m 区间最小值

```
/*
 * m 区间的最小值
 * 有时候要根据题目需要手写单调队列
 * 这题也可以直接用 deque 维护下标即可
 * 维护 pair(idx,val), val 从小到大
```

```
 */
struct Queue{
    pair<int,int> a[N];
    int l,r;
    Queue(){
        memset(a,0,sizeof(a));
        l=0;
        r=0;
    }
    void push(int i,int x){
        if(isEmpty()){
            a[r++]=make_pair(i,x);
            return;
        }
        //维护单调递增性
        while(l<r && a[r-1].second>x){
            r--;
        }
        //去除过老的元素
        while(l<r && a[l].first+m<=i){
            l++;
        }
        a[r++]=make_pair(i,x);
    }

    //直接取出堆头 (最小值)
    int getMin(){
        return a[l].second;
    }

    bool isEmpty(){
        return l==r;
    }
}q;
```

作为最大/最小值能延伸的区间

```
#include <bits/stdc++.h>
using namespace std;
const int N=1e6+50;
int h[N],n;
int le[N],ri[N];
int main(void){
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        scanf("%lld",&h[i]);
    }
    //求出每个数作为最大值/最小值能延伸到的区间
    stack<int> ss;
    for(int i=1;i<=n;i++){
        while(ss.size()>0 && h[i]<=h[ss.top()]){
            ss.pop();
        }
        if(ss.size()>0){
            le[i]=ss.top()+1;
        }else{
```

```
            le[i]=1;
        }
        ss.push(i);
    }
    while(!ss.empty()){
        ss.pop();
    }
    for(int i=n;i>=1;i--){
        while(ss.size()>0 && h[i]<=h[ss.top()]){
            ss.pop();
        }
        if(ss.size()>0){
            ri[i]=ss.top()-1;
        }else{
            ri[i]=n;
        }
        ss.push(i);
    }
    return 0;
}
```

矩阵快速幂

```
/*
 * 矩阵快速幂
 * Mat: 矩阵类型, 包含一个二维数组
 */
Mat operator * (Mat a, Mat b) {
    Mat ans;
    memset(ans.mat, 0, sizeof(ans.mat));
    for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
            for(int k=0; k<n; k++) {
                ans.mat[i][j] += (a.mat[i][k] % mod) * (b.mat[k][j] % mod) % mod;
                ans.mat[i][j] %= mod;
            }
        }
    }
    return c;
}
Mat operator ^ (Mat a, ll n)  {
    Mat ans;
    for(int i=0; i<6; i++){
        for(int j=0; j<6; j++){
             ans.mat[i][j]=(i==j);
        }
    }
    while(n) {
        if(n%2){
            ans=ans*a;
        }
        a=a*a;
        n/=2;
    }
    return ans;
```

```
}
```

BM 递推

```
/*
 * 万一真的要用到照抄就是了。注意别抄错
 */
#include<bits/stdc++.h>
using namespace std;
#define rep(i,a,n) for (int i=a;i<n;i++)
#define per(i,a,n) for (int i=n-1;i>=a;i--)
#define pb push_back
#define mp make_pair
#define all(x) (x).begin(),(x).end()
#define fi first
#define se second
#define SZ(x) ((int)(x).size())
typedef vector<int> VI;
typedef long long ll;
typedef pair<int,int> PII;
const ll mod=1000000007;
ll powmod(ll a,ll b){
    ll res=1;a%=mod; assert(b>=0);
    for(;b;b>>=1){
        if(b&1)res=res*a%mod;a=a*a%mod;
    }
    return res;
}
// head
ll n;
namespace linear_seq {
    const int N=10010;
    ll res[N],base[N],_c[N],_md[N];

    vector<int> Md;
    void mul(ll *a,ll *b,int k) {
        rep(i,0,k+k) _c[i]=0;
        rep(i,0,k) if (a[i]) rep(j,0,k) _c[i+j]=(_c[i+j]+a[i]*b[j])%mod;
        for (int i=k+k-1;i>=k;i--) if (_c[i])
            rep(j,0,SZ(Md)) _c[i-k+Md[j]]=(_c[i-k+Md[j]]-_c[i]*_md[Md[j]])%mod;
        rep(i,0,k) a[i]=_c[i];
    }
    int solve(ll n,VI a,VI b) { // a 系数 b 初值 b[n+1]=a[0]*b[n]+...
        ll ans=0,pnt=0;
        int k=SZ(a);
        assert(SZ(a)==SZ(b));
        rep(i,0,k) _md[k-1-i]=-a[i];_md[k]=1;
        Md.clear();
        rep(i,0,k) if (_md[i]!=0) Md.push_back(i);
        rep(i,0,k) res[i]=base[i]=0;
        res[0]=1;
        while ((1ll<<pnt)<=n) pnt++;
        for (int p=pnt;p>=0;p--) {
            mul(res,res,k);
            if ((n>>p)&1) {
```

```
                for (int i=k-1;i>=0;i--) res[i+1]=res[i];res[0]=0;
                rep(j,0,SZ(Md)) res[Md[j]]=(res[Md[j]]-res[k]*_md[Md[j]])%mod;
            }
        }
        rep(i,0,k) ans=(ans+res[i]*b[i])%mod;
        if (ans<0) ans+=mod;
        return ans;
    }
    VI BM(VI s) {
        VI C(1,1),B(1,1);
        int L=0,m=1,b=1;
        rep(n,0,SZ(s)) {
            ll d=0;
            rep(i,0,L+1) d=(d+(ll)C[i]*s[n-i])%mod;
            if (d==0) ++m;
            else if (2*L<=n) {
                VI T=C;
                ll c=mod-d*powmod(b,mod-2)%mod;
                while (SZ(C)<SZ(B)+m) C.pb(0);
                rep(i,0,SZ(B)) C[i+m]=(C[i+m]+c*B[i])%mod;
                L=n+1-L; B=T; b=d; m=1;
            } else {
                ll c=mod-d*powmod(b,mod-2)%mod;
                while (SZ(C)<SZ(B)+m) C.pb(0);
                rep(i,0,SZ(B)) C[i+m]=(C[i+m]+c*B[i])%mod;
                ++m;
            }
        }
        return C;
    }
    int gao(VI a,ll n) {
        VI c=BM(a);
        c.erase(c.begin());
        rep(i,0,SZ(c)) c[i]=(mod-c[i])%mod;
        return solve(n,c,VI(a.begin(),a.begin()+SZ(c)));
    }
};

int main() {
    /*push_back 进去前 8~10 项左右、最后调用 gao 得第 n 项 */
    //2018 焦作网络赛 L 题
    vector<int>v;
    v.push_back(1);
    v.push_back(4);
    v.push_back(17);
    v.push_back(49);
    v.push_back(129);
    v.push_back(321);
    v.push_back(769);
    v.push_back(1793);
    v.push_back(4097);
    v.push_back(9217);
    int nCase;
    scanf("%d", &nCase);
    while(nCase--){
        scanf("%lld", &n);
```

```
        printf("%lld\n",1LL * linear_seq::gao(v,n-1) % mod);
    }
}
```

大数平方数判断

```
import java.math.BigInteger;
import java.util.Scanner;

public class IsSquare {
    public static boolean check(BigInteger now){
        if(now.compareTo(BigInteger.ZERO)==0||now.compareTo(BigInteger.ONE)==0){
            return true;
        }
        if(now.mod(BigInteger.valueOf(3)).compareTo(BigInteger.valueOf(2))==0){
            return false;
        }
        String s = now.toString();
        if(s.length()%2==0){
            s = s.substring(0,s.length()/2+1);
        }else{
            s = s.substring(0,(1+s.length())/2);
        }
        BigInteger res = BigInteger.ZERO;
        BigInteger m = new BigInteger(s);
        BigInteger two = new BigInteger("2");
        if(s == "1"){
            res = BigInteger.ONE;
        }else{
            while(now.compareTo(m.multiply(m)) < 0){
                m = (m.add(now.divide(m))).divide(two);
            }
            res = m;
        }
        if (res.multiply(res).compareTo(now) == 0){
            return true;
        }
        return false;
    }
}
```

快读

-

*c++

```
namespace IO{
    char buf[1<<15],*S,*T;
    inline char gc(){
        if (S==T){
            T=(S=buf)+fread(buf,1,1<<15,stdin);
            if (S==T)return EOF;
        }
        return *S++;
    }
```

```
    inline int read(){
        int x;bool f;char c;
        for(f=0;(c=gc())<'0'||c>'9';f=c=='-');
        for(x=c^'0';(c=gc())>='0'&&c<='9';x=(x<<3)+(x<<1)+(c^'0'));
        return f?-x:x;
    }
    inline ll readll(){
        ll x;bool f;char c;
        for(f=0;(c=gc())<'0'||c>'9';f=c=='-');
        for(x=c^'0';(c=gc())>='0'&&c<='9';x=(x<<3)+(x<<1)+(c^'0'));
        return f?-x:x;
    }
}
using IO::read;
using IO::readll;
```

java

```
import java.io.*;
import java.util.StringTokenizer;
public class Main{
    static class InputReader{
        public BufferedReader reader;
        public StringTokenizer tokenizer;
        public InputReader(InputStream stream) {
            reader = new BufferedReader(new InputStreamReader(stream), 32768);
            tokenizer = null;
        }
        public String next() {
            while (tokenizer == null || !tokenizer.hasMoreTokens()) {
                try {
                    tokenizer = new StringTokenizer(reader.readLine());
                } catch (IOException e) {e.printStackTrace();}
            }
            return tokenizer.nextToken();
        }
        public int nextInt(){
            return Integer.parseInt(next());
        }
    }
    public static void main(String[] args) {
        InputReader cin=new InputReader(new BufferedInputStream(System.in));
    }
}
```

离散化

```
/*
 * 离散化
 */
void solve(){
    int x[]={0,2,10000,1,3,500,19999999,212222222,13};
    int n=8;
    sort(x+1,x+1+n);
    for(int i=1;i<=n;i++){
        //根据具体题目调整
```

```
        int k=lowerbound(x+1,x+1+n,x[i])-x;
    }
}
```

mt19937 随机数

```
#include <bits/stdc++.h>
using namespace std;
int main(void){
    mt19937 rnd(time(0));
    for(int i=0;i<5;i++){
        printf("%lld\n",rnd());
    }
    return 0;
}
```

bitset

```
/*
构造函数
    bitset<4> bitset1;           //无参构造，长度为 4，默认每一位为 0
    bitset<8> bitset2(12);       //长度为 8，二进制保存，前面用 0 补充
    string s = "100101";
    bitset<10> bitset3(s);       //长度为 10，前面用 0 补充
    char s2[] = "10101";
    bitset<13> bitset4(s2);      //长度为 13，前面用 0 补充
    cout << bitset1 << endl;     //0000
    cout << bitset2 << endl;     //00001100
    cout << bitset3 << endl;     //0000100101
    cout << bitset4 << endl;     //0000000010101
    bitset<2> bitset1(12);     //12 的二进制为 1100，但 bitset1 的 size=2，取 ** 后面 ** 部分，即 00
    string s = "100101";
    bitset<4> bitset2(s);     //s 的 size=6，而 bitset 的 size=4，取 ** 前面 ** 部分，即 1001
    char s2[] = "11101";
    bitset<4> bitset3(s2);     //与 bitset2 同理，取前面部分，即 1110
    cout << bitset1 << endl;     //00
    cout << bitset2 << endl;     //1001
    cout << bitset3 << endl;     //1110
操作
    bitset<4> foo (string("1001"));
    bitset<4> bar (string("0011"));
    cout << (foo^=bar) << endl;       // 1010 (foo 对 bar 按位异或后赋值给 foo)
    cout << (foo&=bar) << endl;       // 0010 (按位与后赋值给 foo)
    cout << (foo|=bar) << endl;       // 0011 (按位或后赋值给 foo)
    cout << (foo<<=2) << endl;        // 1100 (左移 2 位，低位补 0，有自身赋值)
    cout << (foo>>=1) << endl;        // 0110 (右移 1 位，高位补 0，有自身赋值)
    cout << (~bar) << endl;           // 1100 (按位取反)
    cout << (bar<<1) << endl;         // 0110 (左移，不赋值)
    cout << (bar>>1) << endl;         // 0001 (右移，不赋值)
    cout << (foo==bar) << endl;       // false (0110==0011 为 false)
    cout << (foo!=bar) << endl;       // true  (0110!=0011 为 true)
    cout << (foo&bar) << endl;        // 0010 (按位与，不赋值)
    cout << (foo|bar) << endl;        // 0111 (按位或，不赋值)
    cout << (foo^bar) << endl;        // 0101 (按位异或，不赋值)
    cout << foo[0] << endl;             // 1
```

```
    cout << foo[2] << endl;             // 0
其他函数
    bitset<8> foo ("10011011");
    cout << foo.count() << endl;     //5     (count 函数用来求 bitset 中 1 的位数
    cout << foo.size() << endl;      //8      (size 函数用来求 bitset 的大小，一共有 8 位
    cout << foo.test(0) << endl;     //true (test 函数用来查下标处的元素是 0 还是 1，并返回
    cout << foo.any() << endl;     //true    (any 函数检查 bitset 中是否有 1
    cout << foo.none() << endl;     //false (none 函数检查 bitset 中是否没有 1
    cout << foo.all() << endl;     //false    (all 函数检查 bitset 中是全部为 1
    bitset<8> foo ("10011011");
    cout << foo.flip(2) << endl;     //10011111 (flip 函数传参数时，用于将参数位取反
    cout << foo.flip() << endl;      //01100000 (flip 函数不指定参数时，将 bitset 每一位全部取反
    cout << foo.set() << endl;       //11111111 (set 函数不指定参数时，将 bitset 的每一位全部置为 1
    cout << foo.set(3,0) << endl;   //11110111 (set 函数指定两位参数时，将一参位置设置为二参
    cout << foo.set(3) << endl;      //11111111 (set 函数只有一个参数时，将参数下标处置为 1
    cout << foo.reset(4) << endl;   //11101111 (reset 函数传一个参数时将参数下标处置为 0
    cout << foo.reset() << endl;     //00000000 (reset 函数不传参数时将 bitset 的每一位全部置为 0
类型转换
    bitset<8> foo ("10011011");
    string s = foo.to_string();                //将 bitset 转换成 string 类型
    unsigned long a = foo.to_ulong();          //将 bitset 转换成 unsigned long 类型
    unsigned long long b = foo.to_ullong(); //将 bitset 转换成 unsigned long long 类型
    cout << s << endl;     //10011011
    cout << a << endl;     //155
    cout << b << endl;     //155
*/
```

hdu6468——求 1-n 字典序第 m 个数

```
#include <bits/stdc++.h>
using namespace std;
/*
 * 求 1-n 字典序排列的第 m 个数
 */
int solve(int n,int m){
    //考虑一颗完全 10 叉树，树的所有节点就是 1-n，要求的就是前序遍历的第 m 个节点
    //m 是可以走的步数
    int i=1;
    m--;
    while(m!=0){
        //计算 i 到 i+1 的字典序中间相隔的个数
        int s=i,e=i+1;
        int num=0;
        //防止越界
        while(s<=n){
            //计算每一层相差的个数
            //n+1：比如 20-29 其实是 10 个，而 e 就不用 +1，因为 e 在这里表示 30(40/50...)
            num+=min(n+1,e)-s;
            s*=10;
            e*=10;
        }
        if(m<num){
            //向下
            i*=10;
            //走一步
            m--;
        }else{
            //向右
            i++;
            //对前序遍历来说，走了 num 步
            m-=num;
        }
    }
    return i;
}
int main(){
    int n,m;
    scanf("%d%d",&n,&m);
    printf("%d\n",solve(n,m));
    return 0;

}
```

cf1144E——求字符串中位数 (26 进制模拟)

```
#include <bits/stdc++.h>
using namespace std;
const int N=2e5+50;
char s[N],t[N];
int n,a[N],b[N],c[N];
int main(void){
        scanf("%d",&n);
```

```
        scanf("%s",s);
        scanf("%s",t);
        for(int i=0;i<n;i++){
                //后移一位，可能有进位
                a[i+1]=s[i]-'a';
                b[i+1]=t[i]-'a';
        }
        //加法
        for(int i=n;i>=1;i--){
                c[i]+=(a[i]+b[i]);
                if(c[i]>=26){
                        c[i]-=26;
                        c[i-1]++;
                }
        }
        //除法
        for(int i=0;i<=n;i++){
                int t=c[i]%2;
                if(t){
                        c[i+1]+=26;
                }
                c[i]=c[i]/2;
        }
        for(int i=1;i<=n;i++){
                printf("%c",(char)('a'+c[i]));
        }
        printf("\n");
        return 0;
}
```

牛客 548B——除法模拟

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
int t;
ll a,b,l,r,c;
ll Pow(ll a,ll n){
    ll ans=1;
    while(n){
        if(n%2){
            ans=ans*a%b;
        }
        a=a*a%b;
        n/=2;
    }
    return ans;
}
int main(){
    scanf("%d",&t);
    while(t--){
        scanf("%lld%lld%lld%lld",&a,&b,&l,&r);
        int i=l;
        c=a*Pow(10,l-1);
        while(i++<=r){
```

```
            c%=b;
            c*=10;
            printf("%lld",c/b);
        }
        printf("\n");
    }
}
```

hdu4507——数位 dp 求平方和

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const ll MOD=1e9+7;
struct node{
    ll cnt;
    ll sum;
    ll powsum;
}dp[30][10][10];
int a[30];
ll pw[30];
void init(){
    pw[0]=1;
    for(int i=1;i<30;i++){
        pw[i]=(pw[i-1]*10)%MOD;
    }
    for(int i=0;i<30;i++){
        for(int j=0;j<10;j++){
            for(int k=0;k<10;k++){
                dp[i][j][k]=node{-1,0,0};
            }
        }
    }
}
node dfs(int pos,int summod,int valmod,bool limit){
    if(pos<0){
        node tmp;
        tmp.cnt=(summod!=0 && valmod!=0);
        //只有计数才能在这里返回，其他两个值要在回溯的时候计算
        //换句话说，递归边界只是为了判断是否存在这个合法的数
        tmp.sum=tmp.powsum=0;
        return tmp;
    }
    if(!limit && dp[pos][summod][valmod].cnt!=-1){
        return dp[pos][summod][valmod];
    }
    int up=limit?a[pos]:9;
    node ans;
    ans.cnt=ans.sum=ans.powsum=0;
    for(int i=0;i<=up;i++){
        if(i==7){
            continue;
        }
        node tmp=dfs(pos-1,(summod+i)%7,(valmod*10+i)%7,limit&&(i==up));
```

```
        //普通数位 dp 计数
        ans.cnt=(ans.cnt+tmp.cnt)%MOD;
        //sum 计算
        //比如递归到最后一位是 3 4 两个符合的数位，当前位是第 1 位，数位为 2
        //回溯计算就是 3+4=7 -->3+4+23+24=54
        //也就是加上 pw[1]=10 乘以 i=2 乘以递归边界计数 tmp.cnt=2
        ans.sum=(ans.sum+tmp.sum+((pw[pos]*i)%MOD*tmp.cnt)%MOD)%MOD;
        //powsum 计算
        //同上 3^2+4^2=25 --> 3^2+4^2+23^2+24^2
        //也就是加上 (20+3)^2+(20+4)^2=20^2+20^2+3^2+4^2+2*10*(3+4)
        ans.powsum=((ans.powsum+tmp.powsum)%MOD+((((tmp.cnt*pw[pos])
        %MOD*pw[pos])%MOD*i*i)%MOD)+(((tmp.sum*pw[pos])%MOD*2*i)%MOD))%MOD;
    }
    if(!limit){
        dp[pos][summod][valmod]=ans;
    }
    return ans;
}
ll solve(ll x){
    int k=0;
    while(x){
        a[k++]=x%10;
        x/=10;
    }
    return dfs(k-1,0,0,true).powsum;
}
int t;
ll l,r;
int main(void){
    init();
    scanf("%d",&t);
    while(t--){
        scanf("%lld%lld",&l,&r);
        ll ansr=solve(r)%MOD;
        ll ansl=solve(l-1)%MOD;
        printf("%lld\n",(ansr-ansl+MOD)%MOD);
    }
    return 0;
}
```

树链剖分

```
/**
 * 给出一棵带点权的树，三种操作，修改某个点权，查询 u v 路径上的最大点权和点权和
 */
#include <bits/stdc++.h>
using namespace std;
#define lson i<<1
#define rson i<<1|1
#define mid (l+r)/2
const int N=1e6+50;
const int INF=0x3f3f3f3f;
struct Edge{
    int v,next;
}edge[N*2];
int cnt,head[N];
void init(){
    cnt=0;
    memset(head,-1,sizeof(head));
}
void add(int u,int v){
    edge[cnt]=Edge{v,head[u]};
    head[u]=cnt++;
    edge[cnt]=Edge{u,head[v]};
    head[v]=cnt++;
}
int n,q,u,v,clk;
char qu[10];
//点权，父节点，子树大小，节点深度，重儿子，dfs 序双向映射，重链顶点
int a[N],fa[N],siz[N],dep[N],son[N],rk[N],id[N],top[N];
void dfs1(int u,int f,int d){
    fa[u]=f;
    dep[u]=d;
    siz[u]=1;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        if(v==f){
            continue;
        }
        dfs1(v,u,d+1);
        siz[u]+=siz[v];
        if(siz[v]>siz[son[u]]){
            son[u]=v;
        }
    }
}
void dfs2(int u,int t){
    top[u]=t;
    id[u]=++clk;
    rk[clk]=u;
    //只连接重链
    if(!son[u]){
        return;
    }
    dfs2(son[u],t);
    for(int i=head[u];i!=-1;i=edge[i].next){
```

```
        int v=edge[i].v;
        if(v!=son[u] && v!=fa[u]){
            //轻儿子本身一条链
            dfs2(v,v);
        }
    }
}
int mx[N],sum[N];
void pushup(int i){
    mx[i]=max(mx[lson],mx[rson]);
    sum[i]=sum[lson]+sum[rson];
}
void build(int i,int l,int r){
    //mx 要清空为负 inf
    mx[i]=-INF;
    sum[i]=0;
    if(l==r){
        return;
    }
    build(lson,l,mid);
    build(rson,mid+1,r);
    pushup(i);
}
void update(int i,int l,int r,int p,int v){
    if(l==r && l==p){
        sum[i]=mx[i]=v;
        return;
    }
    if(p<=mid){
        update(lson,l,mid,p,v);
    }else{
        update(rson,mid+1,r,p,v);
    }
    pushup(i);
}
int ans;
int query(int i,int l,int r,int ql,int qr,int o){
    if(l>=ql && r<=qr){
        if(o==1){
            return mx[i];
        }else{
            return sum[i];
        }
    }
    int ans=0;
    if(o==1){
        ans=-INF;
        if(ql<=mid){
            ans=max(ans,query(lson,l,mid,ql,qr,o));
        }
        if(qr>mid){
            ans=max(ans,query(rson,mid+1,r,ql,qr,o));
        }
    }else{
        ans=0;
        if(ql<=mid){
```

```
            ans+=query(lson,l,mid,ql,qr,o);
        }
        if(qr>mid){
            ans+=query(rson,mid+1,r,ql,qr,o);
        }
    }
    return ans;
}
int solve(int u,int v,int o){
    int res;
    if(o==1){
        res=-INF;
    }else{
        res=0;
    }
    //重链 top 节点或者本身
    //最好不要直接 swap, 有时候不能要交换的不是比较的值
    int fu=top[u],fv=top[v];
    while(fu!=fv){
        if(dep[fu]>=dep[fv]){
            if(o==1){
                //计算重链贡献, 如果是轻链则为本身节点
                res=max(res,query(1,1,n,id[fu],id[u],1));
            }else{
                res+=query(1,1,n,id[fu],id[u],2);
            }
            //走上下一条链, 别写错成 u=fa[u] 这条重链已经计算过, 要直接跳到 fa[fu]
            u=fa[fu];
            fu=top[u];
        }else{
            if(o==1){
                res=max(res,query(1,1,n,id[fv],id[v],1));
            }else{
                res+=query(1,1,n,id[fv],id[v],2);
            }
            v=fa[fv];
            fv=top[v];
        }
    }
    if(id[u]>id[v]){
        if(o==1){
            res=max(res,query(1,1,n,id[v],id[u],1));
        }else{
            res+=query(1,1,n,id[v],id[u],2);
        }
    }else{
        if(o==1){
            res=max(res,query(1,1,n,id[u],id[v],1));
        }else{
            res+=query(1,1,n,id[u],id[v],2);
        }
    }
    return res;
}
int main(void){
    scanf("%d",&n);
```

```
    init();
    for(int i=0;i<n-1;i++){
        scanf("%d%d",&u,&v);
        add(u,v);
    }
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i]);
    }
    dfs1(1,0,0);
    dfs2(1,1);
    build(1,1,n);
    for(int i=1;i<=n;i++){
        update(1,1,n,id[i],a[i]);
    }
    scanf("%d",&q);
    while(q--){
        scanf("%s %d%d",qu,&u,&v);
        if(qu[0]=='C'){
            update(1,1,n,id[u],v);
        }else{
            int res=0;
            if(qu[1]=='M'){
                res=solve(u,v,1);
            }else{
                res=solve(u,v,2);
            }
            printf("%d\n",res);
        }
    }
    return 0;
}
```

平衡树

Splay

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N=1e6+50;
//树结构
int tr[N][2],fa[N];
//节点值
int val[N];
//节点值重复次数
int cnt[N];
//子树大小（包括重复元素，非节点数）
int siz[N];
//根节点
int rt;
//节点个数
int tot;
//更新信息
void update(int x){
    siz[x]=siz[tr[x][0]]+siz[tr[x][1]]+cnt[x];
}
//查找 x 值位置
int find(int x){
    int now=rt;
    while(x!=val[now]){
        if(x<val[now]){
            if(tr[now][0]==0){
                break;
            }
            now=tr[now][0];
        }else{
            if(tr[now][1]==0){
                break;
            }
            now=tr[now][1];
        }
    }
    return now;
}
//同时更新父节点和子节点连接
void connect(int x,int f,int son){
    fa[x]=f;
    tr[f][son]=x;
}
//创建节点
void newnode(int v,int f){
    tot++;
    val[tot]=v;
    siz[tot]=cnt[tot]=1;
    tr[tot][0]=tr[tot][1]=0;
    connect(tot,f,v>val[f]);
}
//获取是父节点的左儿子还是右儿子
int get(int x){
```

```
    return tr[fa[x]][1]==x;
}
//单旋转,x 上旋
void rotate(int x){
    int y=fa[x];
    int yy=fa[y];
    int xs=get(x);
    int ys=get(y);
    int b=tr[x][xs^1];
    //右旋,x 的右儿子变成 y 的左二子; 左旋,x 的左儿子变成 y 的右儿子
    connect(b,y,xs);
    //右旋, 父节点 y 变成右儿子, 左旋, 父节点 y 变成左二子
    connect(y,x,(xs^1));
    //x 顶替 y 的位置, 连接 yy
    connect(x,yy,ys);
    //x 和 y 发生变动, 更新信息
    //注意这里要先更新 y 的信息, 再更新 x 的信息!!
    update(y);
    update(x);
}
//伸展, 通过多次旋转将 x 旋转到 to 的儿子
void splay(int x,int to){
    while(fa[x]!=to){
        int f=fa[x];
        int ff=fa[f];
        if(ff!=to){
            //分两种情况,x,y,yy 成一条链的, 直接单旋 y
            //yy 左儿子是 y,y 右儿子是 x, 或者 yy 右儿子是 y,y 左儿子是 z, 单旋 x
            rotate(get(x)==get(f)?f:x);
        }
        rotate(x);
    }
    //旋转到根则更新根
    if(to==0){
        rt=x;
    }
}
//插入元素
void insert(int x){
    //特判插入第一个节点
    if(rt==0){
        newnode(x,0);
        rt=tot;
        return;
    }
    //找到 x 位置
    int ip=find(x);
    if(val[ip]==x){
        //找到, 重复次数直接 +1
        cnt[ip]++;
        //更新信息
        update(ip);
        //旋转到根
        splay(ip,0);
    }else{
        //找不到, 创建节点
```

```
        newnode(x,ip);
        //更新信息，旋转到根
        update(ip);
        splay(tot,0);
    }
}
//删除元素
void del(int x){
    int ip=find(x);
    //旋转到根
    splay(ip,0);
    //找不到
    if(val[ip]!=x){
        return;
    }
    //此时已旋转到根
    if(cnt[ip]>1){
        //存在不止一个值
        cnt[ip]--;
        update(ip);
    }else if(tr[ip][0]==0 && tr[ip][1]==0){
        //剩一个节点，删除即为空树
        rt=0;
        tot=0;
        //清空子树
        tr[rt][0]=tr[rt][1]=0;
    }else if(tr[ip][0]!=0 && tr[ip][1]==0){
        //只有左子树，删除后左子树即为根
        rt=tr[ip][0];
        fa[tr[ip][0]]=0;
    }else if(tr[ip][0]==0 && tr[ip][1]!=0){
        //只有右子树
        rt=tr[ip][1];
        fa[tr[ip][1]]=0;
    }else{
        //有左右子树
        //找到左子树的最右节点
        int p=tr[ip][0];
        while(tr[p][1]!=0){
            p=tr[p][1];
        }
        //旋转到 ip 的子树，不能直接旋转到根，因为后面还要用到 ip 的信息
        splay(p,ip);
        //成为新的根
        rt=p;
        fa[p]=0;
        fa[tr[ip][1]]=p;
        tr[p][1]=tr[ip][1];
        update(p);
    }
}
//获取 v 的排名（第几小）
int rnk(int x){
    //找到元素位置
    int ip=find(x);
    //旋转到根
```

```
    splay(ip,0);
    //排名即左儿子大小 +1
    return siz[tr[ip][0]]+1;
}
//获取第 k 小元素，排名的逆操作
int kth(int k){
    int now=rt;
    while(true){
        if(k<=siz[tr[now][0]]){
            //左儿子
            now=tr[now][0];
        }else if(k>cnt[now]+siz[tr[now][0]]){
            //右儿子
            k-=cnt[now]+siz[tr[now][0]];
            now=tr[now][1];
        }else{
            //当前节点
            break;
        }
    }
    return val[now];
}
int pre(int x){
    //先将 x 插入,splay 到根，直接找左子树的最右节点，再删除 x
    insert(x);
    int now=tr[rt][0];
    int ans;
    while(now){
        if(!tr[now][1]){
             ans=val[now];
             break;
        }
        now=tr[now][1];
    }
    del(x);
    return ans;
}
int nxt(int x){
    //同理，找到右子树的最左节点
    insert(x);
    int now=tr[rt][1];
    int ans;
    while(now){
        if(!tr[now][0]){
             ans=val[now];
             break;
        }
        now=tr[now][0];
    }
    del(x);
    return ans;
}
int m,o,x;
int main(void){
    scanf("%d",&m);
    while(m--){
```

```
        scanf("%d%d",&o,&x);
        if(o==1){
            insert(x);
        }else if(o==2){
            del(x);
        }else if(o==3){
            printf("%d\n",rnk(x));
        }else if(o==4){
            printf("%d\n",kth(x));
        }else if(o==5){
            printf("%d\n",pre(x));
        }else if(o==6){
            printf("%d\n",nxt(x));
        }
        //如果是简单的求前驱后继，用 set 即可
        /**
         * auto it=s.upper_bound(x)
         * if(it!=s.end()){
         *      *it 为后继
         * }
         * if(it!=s.begin()){
         *      it++;
         *      *it 为前驱
         * }
         */
    }
    return 0;
}
```

Splay 维护区间

```
#include <bits/stdc++.h>
using namespace std;
const int N=1e6+50;
//因为是维护序列问题，所以 splay 里存的是下标而不是权值
int tr[N][2],fa[N];
int val[N];
int siz[N];
int rt;
int tot;
//懒惰标记数组（该子树是否需要旋转）
bool lazy[N];
void pushup(int x) {
    //这里节点维护的不是权值，所以直接 +1，而不是加重复次数 cnt[i]
    siz[x]=siz[tr[x][0]]+siz[tr[x][1]]+1;
}
void pushdown(int x){
    //左右子树互换并下传标记
    if (lazy[x]){
        swap(tr[x][0], tr[x][1]);
        lazy[tr[x][0]]^=1;
        lazy[tr[x][1]]^=1;
        lazy[x]^=1;
    }
}
int get(int x){
```

```
    return tr[fa[x]][1]==x;
}
void connect(int x,int f,int son){
    fa[x]=f;
    tr[f][son]=x;
}
void rotate(int x) {
    int y=fa[x];
    int yy=fa[y];
    int ys=get(y);
    int xs=get(x);
    //比普通 splay 多了 pushdown 的操作
    pushdown(x);
    pushdown(y);
    int b=tr[x][xs^1];
    connect(b,y,xs);
    connect(y,x,(xs^1));
    connect(x,yy,ys);
    pushup(y);
    pushup(x);
}
//x 旋转到 to 的儿子
void splay(int x,int to){
    while(fa[x]!=to){
        int f=fa[x];
        int ff=fa[f];
        if(ff!=to){
            rotate(get(x)==get(f)?f:x);
        }
        rotate(x);
    }
    if(to==0){
        rt=x;
    }
}
int kth(int k) {
    int now=rt;
    while(true){
        pushdown(now);
        int t=siz[tr[now][0]]+1;
        if(k<t){
            now=tr[now][0];
        }else if(k>t){
            k-=t;
            now=tr[now][1];
        }else{
            break;
        }
    }
    //返回节点位置
    return now;
}
//解决区间最重要的就是提取区间
//提取区间 [l,r] 的方法就是将节点 l-1 旋转到根节点，将节点 r+1 旋转到根节点的右儿子
//此时根节点的右儿子的左子树就是提取的区间
//然后为了能够提取区间 [1,n]，我们设立两个哨兵节点 1 和 n+1，把下标都 +1
```

```
//所以要提取区间 [l,r] 也就是要旋转 l-1 和 r+1,+1 之后就是 l 和 r+2
void reverse(int l, int r) {
    int x=kth(l);
    int y=kth(r+2);
    splay(x,0);
    splay(y,x);
    pushdown(rt);
    //根节点的右儿子的左儿子，标记翻转一次
    lazy[tr[y][0]]^=1;
}
void Build(int l,int r,int f) {
    if(l>r){
        return;
    }
    int mid=(l+r)/2;
    Build(l,mid-1,mid);
    Build(mid+1,r,mid);
    fa[mid]=f;
    //本题，下标和权值一样
    val[mid]=mid-1;
    pushup(mid);
    tr[f][mid>f]=mid;
}
int n,m,l,r;
int main(void) {
    scanf("%d%d",&n,&m);
    //增加两个哨兵节点
    Build(1,n+2,0);
    //中间的点即为根
    rt=(1+n+2)/2;
    for(int i=0;i<m;i++){
        scanf("%d%d",&l,&r);
        reverse(l,r);
    }
    for(int i=2;i<=n+1;i++){
        //维护的是下标，最后映射成权值
        printf("%d ", val[kth(i)]);
    }
    return 0;
}
```

区间合并

```
/**
 * hdu1540 区间合并，查询某个点向左向右最大延伸区间
 */
#include <bits/stdc++.h>
using namespace std;
#define lson i<<1
#define rson i<<1|1
#define mid (l+r)/2
const int N=50050;
int lm[4*N],rm[4*N],mx[4*N];
void pushup(int i,int l,int r){
    lm[i]=lm[lson];
    rm[i]=rm[rson];
    if(lm[lson]==mid-l+1){
        lm[i]+=lm[rson];
    }
    if(rm[rson]==r-mid){
        rm[i]+=rm[lson];
    }
    mx[i]=max(max(mx[lson],mx[rson]),rm[lson]+lm[rson]);
}
//单点更新省略 void update(int i,int l,int r,int p,int c){}
int query(int i,int l,int r,int p){
    if(l==r && l==p){
        return mx[i];
    }
    if(p<=mid){
        return query(lson,l,mid,p);
    }else{
        return query(rson,mid+1,r,p);
    }
}
//查询 1 到 x 的右边区间
int queryR(int i,int l,int r,int ql,int qr){
    if(ql<=l && qr>=r){
        return rm[i];
    }
    int la=0,ra=0;
    if(ql<=mid){
        la=queryR(lson,l,mid,ql,min(mid,qr));
    }
    if(qr>mid){
        ra=queryR(rson,mid+1,r,mid+1,qr);
        if(ra==qr-mid){
            return la+ra;
        }else{
            return ra;
        }
    }else{
        return la;
    }
}
//查询 x 到 n 的左边区间
int queryL(int i,int l,int r,int ql,int qr){
```

```
    if(ql<=l && qr>=r){
        return lm[i];
    }
    int la=0,ra=0;
    if(qr>mid){
        ra=queryL(rson,mid+1,r,max(ql,mid+1),qr);
    }
    if(ql<=mid){
        la=queryL(lson,l,mid,ql,mid);
        if(la==mid-ql+1){
            return ra+la;
        }else{
            return la;
        }
    }else{
        return ra;
    }
}
int n,m,x;
char q[10];
stack<int> sta;
int main(void){
    while(~scanf("%d%d",&n,&m)){
        build(1,1,n);
        while(m--){
            scanf("%s",q);
            if(q[0]=='D'){
                scanf("%d",&x);
                sta.push(x);
                update(1,1,n,x,0);
            }else if(q[0]=='Q'){
                scanf("%d",&x);
                int ri=queryR(1,1,n,1,x);
                int le=queryL(1,1,n,x,n);
                int mm=query(1,1,n,x);
                printf("%d\n",ri+le-mm);
            }else if(q[0]=='R'){
                int las=sta.top();
                sta.pop();
                update(1,1,n,las,1);
            }
        }
    }
    return 0;
}
```

主席树

普通主席树

```
/**
 * 权值线段树可以求全局第 k 大，而主席树就是可持久化权值线段树，
 * 因此可以求区间第 k 大，以及由此延伸出的各种应用
 * 1. 静态区间第 k 大
 * 2. 区间内在某个范围数的个数
 * 3. 区间内不同数的个数
 */
#include <bits/stdc++.h>
using namespace std;
#define mid (l+r)/2
const int N=1e6+50;
//记录多少个根，每棵树根的编号
int cnt,tr[N];
//左右子树根的编号（不再是 i<<1 和 i<<1|1）
int lr[40*N],rr[40*N];
//普通权值线段树 sum[i] 保存的就是值在 [le[i],ri[i]] 之间的数的个数
//同理，主席树保存到就是值在 [le[i],ri[i]] 之间数的个数，但是由于静态主席树每棵树的结构都是一样的
int sum[40*N];
//建树，返回根节点编号
int build(int l,int r){
    int rt=++cnt;
    sum[rt]=0;
    if(l==r){
        return rt;
    }
    lr[rt]=build(l,mid);
    rr[rt]=build(mid+1,r);
    return rt;
}
//插入一个节点（一颗新树）
//pre: 上一个线段树的根
//l,r: 线段树区间范围
//x: 插入的值
int update(int pre,int l,int r,int x){
    int rt=++cnt;
    lr[rt]=lr[pre];
    rr[rt]=rr[pre];
    //比上一时刻的树多了一个数 x，所以 sum+1
    sum[rt]=sum[pre]+1;
    //递归下去，只修改一条链
    if(l<r){
        if(x<=mid){
            lr[rt]=update(lr[pre], l, mid, x);
        }else{
            rr[rt]=update(rr[pre], mid+1, r, x);
        }
    }
    return rt;
}
//查询区间 [u,v] 第 k 大/小，注意这里 l,r 是线段的范围
int query(int u,int v,int l,int r,int k){
    //递归边界
```

```
    if(l>=r){
        return l;
    }
    //相减就是左子树大小 (左子树对应区间数的个数)
    int x=sum[lr[v]]-sum[lr[u]];
    //同全局第 k 大/小, 注意两颗线段树要同时走左子树或右子树, 保证结构相同才能前缀和
    if(k<=x){
        return query(lr[u], lr[v], l, mid, k);
    }else{
        return query(rr[u], rr[v], mid+1, r, k-x);
    }
}
//查询主席树区间 [u,v] 在线段树区间 [l,r] 值在区间 [x,y] 的个数
int query(int u,int v,int l,int r,int x,int y){
    if(x>y){
        return 0;
    }
    //整个区间的数都满足条件
    if(x<=l && r<=y){
        return sum[v]-sum[u];
    }
    int ans=0;
    if(x<=mid){
        ans+=query(lr[u],lr[v],l,mid,x,y);
    }
    if(y>mid){
        ans+=query(rr[u],rr[v],mid+1,r,x,y);
    }
    return ans;
}
```

带修主席树 (树状数组套主席树)

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define mid (l+r)/2
const int N=60005;
int a[N], b[N];
int tr[N], lr[N<<5], rr[N<<5], c[N<<5];
int ctr[N];
int m,cnt,T,n,q;
char s[10];
struct que{
    int l,r,k;
    bool Q;
}Q[10005];
int build(int l,int r){
    int rt=++cnt;
    c[rt]=0;
    if(l==r){
        return rt;
    }
    lr[rt]=build(l,mid);
    rr[rt]=build(mid+1,r);
    return rt;
}
int update(int pre,int l,int r,int p,int v){
    int rt=++cnt;
    lr[rt]=lr[pre];
    rr[rt]=rr[pre];
    c[rt]=c[pre]+v;
    if(l<r){
        if(p<=mid){
            lr[rt]=update(lr[pre],l,mid,p,v);
        }else{
            rr[rt]=update(rr[pre],mid+1,r,p,v);
        }
    }
    return rt;
}
int lowbit(int x){
    return x&(-x);
}
void add(int i, int p, int v){
    while(i<=n){
        //比如修改第一个值 实际上就只有序号为 1 2 4 的根修改了
        ctr[i]=update(ctr[i],1,m,p,v);
        i+=lowbit(i);
    }
}
//查询
int use[N];
int sum(int i){
    int ans=0;
    while(i>0){
        //i 是根序号 use[i] 是根标号
```

```
        //因为是跟前缀和有关，所以往左子树搜
        ans+=c[lr[use[i]]];
        i-=lowbit(i);
    }
    return ans;
}
//根的标号和根的序号不同，bit 中使用的是根的序号也就是加入的顺序
//而主席树查询用的是根的标号，也就是节点标号
//u,v: 根的标号  lrr,rrr: 根的序号
int query(int u,int v,int lrr,int rrr,int l,int r,int k){
    if(l>=r){
        return l;
    }
    //左子树大小，比静态主席树多了 sum(rrr)-sum(lrr)
    //因为加了 bit，此时的 c[lr[v]]-c[lr[u]] 已经不再是左子树大小
    //需要加上 bit 对这段区间的查询值 sum(rrr)-sum(lrr)
    int tmp=sum(rrr)-sum(lrr)+c[lr[v]]-c[lr[u]];
    if(tmp>=k){
        //记录 bit 查询跳转时候使用的节点，即查询路径
        for(int i=lrr;i>0;i-=lowbit(i)){
            use[i]=lr[use[i]];
        }
        for(int i=rrr;i>0;i-=lowbit(i)){
            use[i]=lr[use[i]];
        }
        //递归查询
        return query(lr[u],lr[v],lrr,rrr,l,mid,k);
    }else{
        for(int i=lrr;i;i-=lowbit(i)){
            use[i]=rr[use[i]];
        }
        for(int i=rrr;i;i-=lowbit(i)){
            use[i]=rr[use[i]];
        }
        return query(rr[u],rr[v],lrr,rrr,mid+1,r,k-tmp);
    }
}
int main(){

    scanf("%d",&T);
    while(T--){
        scanf("%d%d",&n,&q);
        cnt=0;
        m=0;
        for(int i=1;i<=n;i++){
            scanf("%d",&a[i]);
            b[m++]=a[i];
        }
        for(int i=0;i<q;i++){
            scanf("%s",s);
            if(s[0]=='Q'){
                scanf("%d%d%d", &Q[i].l, &Q[i].r, &Q[i].k);
                Q[i].Q=true;
            }
            else{
                scanf("%d%d", &Q[i].l, &Q[i].r);
```

```
                Q[i].Q=false;
                b[m++]=Q[i].r;
            }
        }
        //离散化建树
        sort(b,b+m);
        m=unique(b+1,b+1+m)-b;
        tr[0]=build(1,m);
        for(int i=1;i<=n;i++){
            int k=lower_bound(b,b+m,a[i])-b;
            tr[i]=update(tr[i-1],1,m,k,1);
        }
        //初始化树状数组
        for(int i=1;i<=n;i++){
            ctr[i]=tr[0];
        }
        for(int i=0;i<q;i++){
            if(Q[i].Q){
                for(int j=Q[i].l-1;j;j-=lowbit(j))
                    use[j]=ctr[j];
                for(int j=Q[i].r;j;j-=lowbit(j))
                    use[j]=ctr[j];
                //普通主席树查询操作，离散化映射
                printf("%d\n", b[query(tr[Q[i].l-1],tr[Q[i].r],Q[i].l-1,Q[i].r,1,m,Q[i].k)]);
            }else{
                //将 a[que[i].l] 修改为 que[i].r 离散化后即将 kl 修改为 kr
                int kl=lower_bound(b, b+m, a[Q[i].l])-b;
                int kr=lower_bound(b, b+m, Q[i].r)-b;
                //树状数组单点修改
                //普通树状数组的数组下标 i 改成了主席树的根，同样是修改的位置
                add(Q[i].l,kl,-1);
                add(Q[i].l,kr,1);
                //原数组也修改
                a[Q[i].l]=Q[i].r;
            }
        }
    }
    return 0;
}
```

数列分块

```cpp
/**
 * 数列分块 1：区间更新，单点查询
 */
#include <bits/stdc++.h>
using namespace std;
const int N=5e4+50;
int a[N],bel[N],add[N];
int n,o,l,r,c;
int block;
void update(int l,int r,int c){
    //在一个区间内，暴力修改
    if(bel[l]==bel[r]){
        for(int i=l;i<=r;i++){
            a[i]+=c;
        }
    }else{
        //修改两边
        for(int i=l;i<=bel[l]*block;i++){
            a[i]+=c;
        }
        for(int i=(bel[r]-1)*block+1;i<=r;i++){
            a[i]+=c;
        }
        //修改整个区间
        for(int i=bel[l]+1;i<bel[r];i++){
            add[i]+=c;
        }
    }
}
int main(void){
    scanf("%d",&n);
    block=sqrt(n);
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i]);
        bel[i]=(i-1)/block+1;
    }
    for(int i=1;i<=n;i++){
        scanf("%d%d%d%d",&o,&l,&r,&c);
        if(o==0){
            update(l,r,c);
        }else{
            printf("%d\n",a[r]+add[bel[r]]);
        }
    }
    return 0;
}
```

```cpp
/**
 * 数列分块 2：区间更新，查询区间小于 x 的个数
 */
#include <bits/stdc++.h>
using namespace std;
const int N=1e5+50;
int a[N],b[N],bel[N],add[N];
int block;
int n;
int o,l,r,c;
void bsort(int x){
    int l=(x-1)*block+1;
    int r=min(l+block-1,n);
    for(int i=l;i<=r;i++){
        b[i]=a[i];
    }
    sort(b+l,b+r+1);
}
void update(int l,int r,int c){
    if(bel[l]==bel[r]){
        for(int i=l;i<=r;i++){
            a[i]+=c;
        }
        bsort(bel[l]);
    }else{
        for(int i=l;i<=bel[l]*block;i++){
            a[i]+=c;
        }
        bsort(bel[l]);
        for(int i=(bel[r]-1)*block+1;i<=r;i++){
            a[i]+=c;
        }
        bsort(bel[r]);
        for(int i=bel[l]+1;i<bel[r];i++){
            add[i]+=c;
        }
    }
}
//块内具有单调性
//二分查找第 x 个块小于 c 的个数
int cal(int x,int c){
    int l=(x-1)*block+1;
    int r=min(l+block-1,n);
    int ans=0;
    while(l<=r){
        int mid=(l+r)/2;
        if(b[mid]+add[x]<c){
            ans=mid;
            l=mid+1;
        }else{
            r=mid-1;
        }
    }
    return ans?(ans-(x-1)*block):0;
}
//查询 [l,r] 中小于 c 的数的个数
```

```
int query(int l,int r,int c){
    int ans=0;
    if(bel[l]==bel[r]){
        for(int i=l;i<=r;i++){
            if(a[i]+add[bel[l]]<c){
                ans++;
            }
        }
    }else{
        for(int i=l;i<=bel[l]*block;i++){
            if(a[i]+add[bel[l]]<c){
                ans++;
            }
        }
        for(int i=(bel[r]-1)*block+1;i<=r;i++){
            if(a[i]+add[bel[r]]<c){
                ans++;
            }
        }
        for(int i=bel[l]+1;i<bel[r];i++){
            ans+=cal(i,c);
        }
    }
    return ans;
}
int main(void){
    scanf("%d",&n);
    block=sqrt(n);
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i]);
        bel[i]=(i-1)/block+1;
    }
    int cnt=(n-1)/block+1;
    //每一块进行排序预处理
    for(int i=1;i<=cnt;i++){
        bsort(i);
    }
    for(int i=1;i<=n;i++){
        scanf("%d%d%d%d",&o,&l,&r,&c);
        if(o==0){
            update(l,r,c);
        }else{
            printf("%d\n",query(l,r,c*c));
        }
    }
    return 0;
}
```

```cpp
/**
 * 数列分块 3：区间更新，查询区间里 x 的前驱
 */
#include <bits/stdc++.h>
using namespace std;
const int N=1e5+50;
int a[N],b[N],add[N],bel[N];
int n,o,l,r,c;
int block;
void bsort(int x){
    int l=(x-1)*block+1;
    int r=min(l+block-1,n);
    for(int i=l;i<=r;i++){
        b[i]=a[i];
    }
    sort(b+l,b+r+1);
}
void update(int l,int r,int c){
    if(bel[l]==bel[r]){
        for(int i=l;i<=r;i++){
            a[i]+=c;
        }
        bsort(bel[l]);
    }else{
        for(int i=l;i<=bel[l]*block;i++){
            a[i]+=c;
        }
        bsort(bel[l]);
        for(int i=(bel[r]-1)*block+1;i<=r;i++){
            a[i]+=c;
        }
        bsort(bel[r]);
        for(int i=bel[l]+1;i<bel[r];i++){
            add[i]+=c;
        }
    }
}
//在 x 块找到最接近 c 的最大数
int cal(int x,int c){
    int l=(x-1)*block+1;
    int r=min(l+block-1,n);
    int ans=0;
    while(l<=r){
        int mid=(l+r)/2;
        if(b[mid]+add[x]<c){
            ans=mid;
            l=mid+1;
        }else{
            r=mid-1;
        }
    }
    return b[ans]+add[x];
}
int query(int l,int r,int c){
    int ans=-1;
    if(bel[l]==bel[r]){
```

```
        for(int i=l;i<=r;i++){
            int t=a[i]+add[bel[l]];
            if(t<c && t>ans){
                ans=t;
            }
        }
    }else{
        for(int i=l;i<=bel[l]*block;i++){
            int t=a[i]+add[bel[l]];
            if(t<c && t>ans){
                ans=t;
            }
        }
        for(int i=(bel[r]-1)*block+1;i<=r;i++){
            int t=a[i]+add[bel[r]];
            if(t<c && t>ans){
                ans=t;
            }
        }
        for(int i=bel[l]+1;i<bel[r];i++){
            int t=cal(i,c);
            if(t<c && t>ans){
                ans=t;
            }
        }
    }
    return ans;
}
int main(void){
    //b[0] 设置为-1, 这样找 a[0] 的前驱就不会找到 0 而是找到-1(实际上是找不到)
    b[0]=-1;
    scanf("%d",&n);
    block=sqrt(n);
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i]);
        bel[i]=(i-1)/block+1;
    }
    int cnt=(n-1)/block+1;
    for(int i=1;i<=cnt;i++){
        bsort(i);
    }
    for(int i=1;i<=n;i++){
        scanf("%d%d%d%d",&o,&l,&r,&c);
        if(o==0){
            update(l,r,c);
        }else{
            printf("%d\n",query(l,r,c));
        }
    }
    return 0;
}
```

```cpp
/**
 * 数列分块 4：区间更新，区间求和
 */
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=5e4+50;
ll a[N];
int bel[N];
//add 是单点累加标记 w 是块累加标记 add 时也要更新 w
ll add[N],w[N];
int n,o,l,r;
ll c;
ll mod;
int block;
void update(int l,int r,ll c){
    if(bel[l]==bel[r]){
        for(int i=l;i<=r;i++){
            a[i]+=c;
            w[bel[l]]+=c;
        }
    }else{
        for(int i=l;i<=bel[l]*block;i++){
            a[i]+=c;
            w[bel[l]]+=c;
        }
        for(int i=(bel[r]-1)*block+1;i<=r;i++){
            a[i]+=c;
            w[bel[r]]+=c;
        }
        for(int i=bel[l]+1;i<bel[r];i++){
            add[i]+=c;
            w[i]+=c*block;
        }
    }
}
ll query(int l,int r){
    ll ans=0;
    if(bel[l]==bel[r]){
        for(int i=l;i<=r;i++){
            ans+=(a[i]+add[bel[l]])%mod;
        }
    }else{
        for(int i=l;i<=bel[l]*block;i++){
            ans+=(a[i]+add[bel[l]])%mod;
        }
        for(int i=(bel[r]-1)*block+1;i<=r;i++){
            ans+=(a[i]+add[bel[r]])%mod;
        }
        for(int i=bel[l]+1;i<bel[r];i++){
            ans+=w[i]%mod;
        }
    }
    return ans%mod;
}
int main(void){
```

```
    scanf("%d",&n);
    block=sqrt(n);
    for(int i=1;i<=n;i++){
        scanf("%lld",&a[i]);
        bel[i]=(i-1)/block+1;
        //初始化每一块的和
        w[bel[i]]+=a[i];
    }
    for(int i=1;i<=n;i++){
        scanf("%d%d%d%lld",&o,&l,&r,&c);
        if(o==0){
            update(l,r,c);
        }else{
            mod=c+1;
            int ans=query(l,r);
            printf("%lld\n",ans%mod);
        }
    }
    return 0;
}
```

```cpp
/**
 * 数列分块 5：区间开方，区间求和
 */
#include <bits/stdc++.h>
using namespace std;
const int N=5e4+50;
int a[N],bel[N],mx[N],w[N];
int n,block,o,l,r,c;
//区间开方更新
void update(int l,int r){
    if(bel[l]==bel[r]){
        if(mx[bel[l]]!=0 && mx[bel[l]]!=1){
            for(int i=l;i<=r;i++){
                int t=a[i]-(int)sqrt(a[i]);
                w[bel[l]]-=t;
                a[i]-=t;
            }
            //更新区间最大值，因为开方的部分区间会影响到整个区间的最大值
            //要记得从 0 开始，不能从原来的 mx[bel[l]] 开始
            mx[bel[l]]=0;
            for(int i=(bel[l]-1)*block+1;i<=bel[l]*block;i++){
                mx[bel[l]]=max(mx[bel[l]],a[i]);
            }
        }
    }else{
        if(mx[bel[l]]!=0 && mx[bel[l]]!=1){
            for(int i=l;i<=bel[l]*block;i++){
                int t=a[i]-(int)sqrt(a[i]);
                w[bel[l]]-=t;
                a[i]-=t;
            }
            mx[bel[l]]=0;
            for(int i=(bel[l]-1)*block+1;i<=bel[l]*block;i++){
                mx[bel[l]]=max(mx[bel[l]],a[i]);
            }
        }
        if(mx[bel[r]]!=0 && mx[bel[r]]!=1){
            for(int i=(bel[r]-1)*block+1;i<=r;i++){
                int t=a[i]-(int)sqrt(a[i]);
                w[bel[r]]-=t;
                a[i]-=t;
            }
            mx[bel[r]]=0;
            for(int i=(bel[r]-1)*block+1;i<=bel[r]*block;i++){
                mx[bel[r]]=max(mx[bel[r]],a[i]);
            }
        }
        for(int i=bel[l]+1;i<bel[r];i++){
            if(mx[i]!=0 && mx[i]!=1){
                mx[i]=0;
                for(int j=(i-1)*block+1;j<=i*block;j++){
                    int t=a[j]-(int)sqrt(a[j]);
                    w[i]-=t;
                    a[j]-=t;
                    mx[i]=max(mx[i],a[j]);
                }
```

```
            }
        }
    }
}
int query(int l,int r){
    int ans=0;
    if(bel[l]==bel[r]){
        if(mx[bel[l]]!=0){
            for(int i=l;i<=r;i++){
                ans+=a[i];
            }
        }
    }else{
        for(int i=l;i<=bel[l]*block;i++){
            if(mx[bel[l]]!=0){
                ans+=a[i];
            }
        }
        for(int i=(bel[r]-1)*block+1;i<=r;i++){
            if(mx[bel[r]]!=0){
                ans+=a[i];
            }
        }
        for(int i=bel[l]+1;i<bel[r];i++){
            ans+=w[i];
        }
    }
    return ans;
}
int main(void){
    scanf("%d",&n);
    block=sqrt(n);
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i]);
        bel[i]=(i-1)/block+1;
        w[bel[i]]+=a[i];
        mx[bel[i]]=max(mx[bel[i]],a[i]);
    }
    for(int i=1;i<=n;i++){
        scanf("%d%d%d%d",&o,&l,&r,&c);
        if(o==0){
            update(l,r);
        }else{
            printf("%d\n",query(l,r));
        }
    }
    return 0;
}
```

```cpp
/**
 * 数列分块 6：动态插值，单点查询
 */
#include <bits/stdc++.h>
using namespace std;
const int N=1e5+50;
int a[N],bel[N];
int n,block,o,l,r,c;
vector<int> vec[400];
//a[i] 前面插入值 x
void insert(int x,int v){
    //找到要插入的块
    int now=0;
    int whi=0;
    for(int i=1;;i++){
        int siz=vec[i].size();
        now+=siz;
        if(now>=x){
            whi=i;
            x-=(now-siz);
            break;
        }
    }
    vec[whi].insert(vec[whi].begin()+x-1,v);
}
//查询 a[x]
int pos(int x){
    int now=0;
    for(int i=1;;i++){
        int siz=vec[i].size();
        now+=siz;
        if(now>=x){
            int iii=x-(now-siz);
            return vec[i][iii-1];
        }
    }
}
int main(void){
    scanf("%d",&n);
    block=sqrt(n);
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i]);
        bel[i]=(i-1)/block+1;
        vec[bel[i]].push_back(a[i]);
    }
    for(int i=1;i<=n;i++){
        scanf("%d%d%d%d",&o,&l,&r,&c);
        if(o==0){
            insert(l,r);
        }else{
            printf("%d\n",pos(r));
        }
    }
    return 0;
}
```

```cpp
/**
 * 数列分块 7：区间乘法，加法，单点查询
 */
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=1e5+50;
const int mod=1e4+7;
ll a[N],bel[N];
//标记
ll add[N],mul[N];
int n,block;
int o,l,r,c;
void Add(int l,int r,int w){
    if(bel[l]==bel[r]){
        //同一块的，先加上标记
        for(int i=(bel[l]-1)*block+1;i<=bel[l]*block;i++){
            a[i]=(a[i]*mul[bel[l]]+add[bel[l]])%mod;
        }
        add[bel[l]]=0;
        mul[bel[l]]=1;
        //再暴力更新
        for(int i=l;i<=r;i++){
            a[i]=(a[i]+w)%mod;
        }
    }else{
        //左边部分块，标记
        for(int i=(bel[l]-1)*block+1;i<=bel[l]*block;i++){
            a[i]=(a[i]*mul[bel[l]]+add[bel[l]])%mod;
        }
        add[bel[l]]=0;
        mul[bel[l]]=1;
        //暴力
        for(int i=l;i<=bel[l]*block;i++){
            a[i]=(a[i]+w)%mod;
        }
        //右边部分块
        for(int i=(bel[r]-1)*block+1;i<=bel[r]*block;i++){
            a[i]=(a[i]*mul[bel[r]]+add[bel[r]])%mod;
        }
        add[bel[r]]=0;
        mul[bel[r]]=1;
        for(int i=(bel[r]-1)*block+1;i<=r;i++){
            a[i]=(a[i]+w)%mod;
        }
        //中间整块
        for(int i=bel[l]+1;i<bel[r];i++){
            add[i]=(add[i]+w)%mod;
        }
    }
}
void Mul(int l,int r,int w){
    if(bel[l]==bel[r]){
        //同一块的，先加上标记
        for(int i=(bel[l]-1)*block+1;i<=bel[l]*block;i++){
            a[i]=(a[i]*mul[bel[l]]+add[bel[l]])%mod;
```

```
        }
        add[bel[l]]=0;
        mul[bel[l]]=1;
        //再暴力更新
        for(int i=l;i<=r;i++){
            a[i]=(a[i]*w)%mod;
        }
    }else{
        //左边部分块，标记
        for(int i=(bel[l]-1)*block+1;i<=bel[l]*block;i++){
            a[i]=(a[i]*mul[bel[l]]+add[bel[l]])%mod;
        }
        add[bel[l]]=0;
        mul[bel[l]]=1;
        //暴力
        for(int i=l;i<=bel[l]*block;i++){
            a[i]=(a[i]*w)%mod;
        }
        //右边部分块
        for(int i=(bel[r]-1)*block+1;i<=bel[r]*block;i++){
            a[i]=(a[i]*mul[bel[r]]+add[bel[r]])%mod;
        }
        add[bel[r]]=0;
        mul[bel[r]]=1;
        for(int i=(bel[r]-1)*block+1;i<=r;i++){
            a[i]=(a[i]*w)%mod;
        }
        //中间整块
        for(int i=bel[l]+1;i<bel[r];i++){
            add[i]=(add[i]*w)%mod;
            mul[i]=(mul[i]*w)%mod;
        }
    }
}
int main(void){
    scanf("%d",&n);
    block=sqrt(n);
    for(int i=1;i<=n;i++){
        scanf("%lld",&a[i]);
        bel[i]=(i-1)/block+1;
        mul[i]=1;
    }
    for(int i=1;i<=n;i++){
        scanf("%d%d%d%d",&o,&l,&r,&c);
        if(o==0){
            Add(l,r,c);
        }else if(o==1){
            Mul(l,r,c);
        }else{
            printf("%d\n",(a[r]*mul[bel[r]]+add[bel[r]])%mod);
        }
    }
    return 0;
}
```

快速乘

```
//快速乘法取模
ll qmul_mod(ll a,ll b,ll mod){
    ll ans=0;
    while(b){
        if((b%=mod)&1){
            ans+=a%=mod;
        }
        b>>=1;
        a<<=1;
    }
    return ans%mod;
}
//O(1) 快速乘取模
 ll ksc(ll x,ll y,ll mod){
    return (x*y-(ll)((long double)x/mod*y)*mod+mod)%mod;
}
```

预处理 +O(1) 快速幂

```
/**
 * 给出正整数 x 和 n 个正整数 ai, 求 x^ai % p
 * x^ai=(x^(ai % s))*(x^ai/s)^s
 * 先预处理出 x 的 0-S-1 次幂, 即 x^(ai % s)
 * 再预处理 x 的 s 次的 0-S-1 次幂 即 x^s^(ai/s)
 */
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=5e6+50;
const ll mod=998244352;
int n;
ll x,a[N];
ll p[N],sp[N];
int main(void){
    scanf("%lld%d",&x,&n);
    ll s=sqrt(mod+0.5)+1;
    p[0]=1;
    for(int i=1;i<=s;i++){
        p[i]=p[i-1]*x%mod;
    }
    sp[0]=1;
    for(int i=1;i<=s;i++){
        sp[i]=sp[i-1]*p[s]%mod;
    }
    for(int i=0;i<n;i++){
        scanf("%lld",&a[i]);
    }
    for(int i=0;i<n;i++){
        printf("%lld ",sp[a[i]/s]*p[a[i]%s]%mod);
    }
    printf("\n");
    return 0;
}
```

莫队

```
//莫队核心就是如何从 [l,r] 的答案推出 [l-1,r],[l+1,r],[l,r-1],[l,r+1] 的答案
//复杂度 O(n*sqrt(n)* 区间移动操作)
//分块
bel[i]=(i-1)/block+1;
//询问的排序
struct Q{
    int id,l,r,ans;
}que[N];
bool cmp1(Q a,Q b){
    return bel[a.l]==bel[b.l]?a.r<b.r:a.l<b.l;
}
//奇偶排序优化
bool cmp1(Q a,Q b){
    return (bel[a.l])^(bel[b.l])?a.l<b.l:(((bel[a.l])&1)?a.r<b.r:a.r>b.r);
}
//询问区间移动
int l=1,r=0;
ans=0;
for(int i=0;i<q;i++){
    while(l<que[i].l){
        del(l++);
    }
    while(r>que[i].r){
        del(r--);
    }
    while(l>que[i].l){
        add(--l);
    }
    while(r<que[i].r){
        add(++r);
    }
    que[i].ans=ans;
}
//add 和 del 函数
//1. 区间不同数个数
void add(int i){
    if(++cnt[a[i]]==1){
        ans++;
    }
}
void del(int i){
    if(--cnt[a[i]]==0){
        ans--;
    }
}
//2. 异或值为 K 的子区间个数
void add(int i){
    ans+=1ll*cnt[pre[i]^k];
    cnt[pre[i]]++;

}
void del(int i){
    cnt[pre[i]]--;
    ans-=1ll*cnt[pre[i]^k];
```

```
}
//3. 维护区间公式
//区间内选的两个数相同的概率，需要维护每个数出现次数平方和
void add(int i){
    zi--;
    zi+=(2ll*cnt[a[i]]+1);
    cnt[a[i]]++;
}
void del(int i){
    zi++;
    zi-=(2ll*cnt[a[i]]-1);
    cnt[a[i]]--;
}
//4. 询问区间存在性答案
//是否存在 a+b=k b-a=k a*b=k
//两个 bitset 一个维护 x 一个维护 N-x 的存在性
void add(int i){
    if(++cnt[a[i]]==1){
        forw[a[i]]=1;
        rev[N-a[i]]=1;
    }
}
void del(int i){
    if(--cnt[a[i]]==0){
        forw[a[i]]=0;
        rev[N-a[i]]=0;
    }
}
//统计答案
int x=que[i].x;
if(que[i].opt==1){
    //有任意 a 和 a+x 同时存在
    if((forw & (forw << x)).any()){
        que[i].ans=1;
    }
}else if(que[i].opt==2){
    //求 a+b=x 即 N-b-a=N-x
    //即有任意 a 和 N-b-(N-x) 同时存在，即 forw & rev >> N-x
    if((forw &(rev >> N-x)).any()){
        que[i].ans=1;
    }
}else{
    //暴力枚举因子
    for(int j=1;j<=(int)sqrt(x);j++){
        if(x%j==0){
            if(forw[j] && forw[x/j]){
                que[i].ans=1;
                break;
            }
        }
    }
}
//5. 询问区间区间逆序数 (莫队 + 树状数组)
void add(int i,int x){}
void query(int i){}
for(int i=0;i<m;i++){
```

```
        while(l<que[i].l){
                add(a[l],-1);
        ans-=query(a[l]-1);
        l++;
    }
        while(r>que[i].r){
                add(a[r],-1);
        ans-=r-l-query(a[r]);
        r--;
    }
        while(l>que[i].l){
                l--;
        add(a[l],1);
        ans+=query(a[l]-1);
    }
        while(r<que[i].r){
                r++;
        add(a[r],1);
        now+=r-l+1-query(a[r]);
    }
        que[i].ans=ans;
}
//带修莫队，多了时序的表示
struct Q{
    //tim 表示发生在第几次修改之后的询问
    int id,l,r,tim,ans;
}que[N];
bool cmp1(Q a,Q b){
    return bel[a.l]==bel[b.l]?(bel[a.r]==bel[b.r]?a.tim<b.tim:a.r<b.r):a.l<b.l;
}
//修改，当前时刻将 a[x] 修改为 d
void going(int x,int d){
    //修改的位置影响当前询问区间
    if(x>=l && x<=r){
        del(a[x]);
        add(d);
    }
    a[x]=d;
}
//先维护时间顺序
int T=0;
for(int i=0;i<qcnt;i++){
    //维护时间
    while(T<que[i].tim){
        going(c[T].pos,c[T].New);
        T++;
    }
    while(T>que[i].tim){
        //T 当前的修改还未执行（初始化 T=0，但 T 为 0 时的修改并未执行，所以无需抵消 T 的修改）
        T--;
        going(c[T].pos,c[T].Old);
    }
    //...
}
//树上带修莫队
/*
```

```
 * 给一棵树，树上每个节点有一个糖果，一共有 m 种糖果美味度分别为 vi，每个游客第 i 次吃同种糖果的新奇度 wi
 * 多组询问：l 到 r 路径上的 sum(vi*wi) (注意 wi 是会随着同种糖果的多次访问而变化)
 */
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=1e5+50;
int n,m,q,l,r,t,cnt1,cnt2,tot,clk;
vector<int> g[N];
int be[N<<1];
int pw[25];
//用于倍增求 lca f[u][i] 表示从 u 往上跳 2^i 格可以到达的点
int fa[N][17];
int c[N],d[N];
int v[N],w[N],now[N];
//表示第几次访问这个糖果，从而间接访问了 wi
int u[N];
//标记是否访问，每次访问就 ^1，这样子树的节点一进一出最后就不会影响结果的贡献
//(因为 l 和 r 的路径是不含有 l 或 r 的子树的，但是 dfs 序有)
bool vis[N];
struct Q{
    int id,l,r,tim;
}a[N];
struct Change{
    int pos,New,Old;
}ch[N];
ll ans[N],sum;
//dfs 序
int id[N<<1],in[N],out[N];
void dfs(int u){
    //双向映射
    in[u]=++clk;
    id[clk]=u;
    //倍增
    for(int i=1;pw[i]<=d[u];i++){
        fa[u][i]=fa[fa[u][i-1]][i-1];
    }
    for(int i=0;i<g[u].size();i++){
        int v=g[u][i];
        if(v!=fa[u][0]){
            //fa[v][0] 即 v 往上跳 1 即 v 的父节点
            fa[v][0]=u;
            //深度
            d[v]=d[u]+1;
            dfs(v);
        }
    }
    out[u]=++clk;
    id[clk]=u;
}
int lca(int x,int y){
    if(d[x]<d[y]){
        swap(x,y);
    }
    int tmp=d[x]-d[y];
    for(int i=0;pw[i]<=tmp;i++){
```

```
        //往上跳
        if(tmp&pw[i]){
            x=fa[x][i];
        }
    }
    //跳到刚好，即 y=lca(x,y)
    if(x==y){
        return x;
    }
    for(int i=16;i>=0;i--){
        //x 和 y 一起跳
        if(fa[x][i]!=fa[y][i]){
            x=fa[x][i];
            y=fa[y][i];
        }
    }
    return fa[x][0];
}
bool cmp(Q x,Q y){
    return be[x.l]==be[y.l]?(be[x.r]==be[y.r]?x.tim<y.tim:x.r<y.r):x.l<y.l;
}
//修改节点，即修改答案贡献
void revise(int x){
    if(vis[x]){
        //之前访问过，那再修改的话，就要把前面计算的贡献减去
        sum-=1ll*v[c[x]]*w[u[c[x]]--];
    }else{
        //反之加上贡献
        sum+=1ll*v[c[x]]*w[++u[c[x]]];
    }
    //标记访问一次，状态反转
    vis[x]^=1;
}
//在这个时间点，把节点 x 修改为颜色 y
void going(int x,int y){
    //这个点访问过，这个时刻的修改会影响到当前询问区间
    if(vis[x]){
        //之前访问过的，必须修改贡献
        revise(x);
        //先减贡献，修改节点值，再加贡献
        c[x]=y;
        revise(x);
    }else{
        //之前没访问过的直接修改
        c[x]=y;
    }
}
int main(){
    scanf("%d%d%d",&n,&m,&q);
    pw[0]=1;
    for(int i=1;i<=17;++i){
        pw[i]=pw[i-1]*2;
    }
    for(int i=1;i<=m;++i){
        scanf("%d",&v[i]);
    }
```

```
for(int i=1;i<=n;++i){
    scanf("%d",&w[i]);
}
//建树
for(int i=1;i<n;++i){
    int l,r;scanf("%d%d",&l,&r);
    g[l].push_back(r);
    g[r].push_back(l);
}
for(int i=1;i<=n;++i){
    scanf("%d",&c[i]);
    //now 记录当前状态用于修改和恢复
    now[i]=c[i];
}
//分块单位
int unit=pow(n,2.0/3);
//构造 dfs 序，将树结构转化为序列
dfs(1);
for(int i=1;i<=clk;++i){
    //分块
    be[i]=i/unit+1;
}
while(q--){
    scanf("%d%d%d",&t,&l,&r);
    if(t){
        //询问
        if(in[l]>in[r]){
            swap(l,r);
        }
        //如果 l 就是 lca(l,r)，起点就是 in[l]，否则起点就是 out[l]
        a[++cnt1]=Q{cnt1,(lca(l,r)==l)?in[l]:out[l],in[r],cnt2};
    }else{
        //修改，把 l 的值修改成 r，原来的值是 now[l]
        ch[++cnt2]=Change{l,r,now[l]};
        now[l]=r;
    }
}
sort(a+1,a+1+cnt1,cmp);
//初始化空区间 l=1 r=0
l=1,r=0,t=0;
for(int i=1;i<=cnt1;++i){
    //同普通带修莫队，先处理时序
    while(t<a[i].tim){
        t++;
        going(ch[t].pos,ch[t].New);
    }
    while(t>a[i].tim){
        going(ch[t].pos,ch[t].Old);
        t--;
    }
    //再处理空间，注意要用 id 映射，同普通带修莫队
    while(l<a[i].l){
        revise(id[l]);
        l++;
    }
    while(r>a[i].r){
```

```
            revise(id[r]);
            r--;
        }
        while(l>a[i].l){
            l--;
            revise(id[l]);
        }
        while(r<a[i].r){
            r++;
            revise(id[r]);
        }
        //树上莫队特殊处理
        int x=id[l],y=id[r],tmp=lca(x,y);
        if(x!=tmp&&y!=tmp){
            //xy 分别在 lca 两边，则 lca 这个点只算了一次
            //(可以写个 dfs 看下，lca 的 dfs 出序刚好是区间右端点 +1)，需要再更新一次贡献
            revise(tmp);
            revise(tmp);
        }
        //x 或 y 是 lca(x,y)，直接记录答案
        ans[a[i].id]=sum;
    }
    for(int i=1;i<=cnt1;i++){
        printf("%lld\n",ans[i]);
    }
}
```

dfs 序建线段树

```
#include <bits/stdc++.h>
using namespace std;
#define lson i<<1
#define rson i<<1|1
#define mid (l+r)/2
const int N=2e5+50;
const int M=4e5+50;
int n,q,x,o,a[N];
char s[10];
struct Edge{
    int v,next;
}edge[M];
int cnt,head[N];
void init(){
    cnt=0;
    memset(head,-1,sizeof(head));
}
void add(int u,int v){
    edge[cnt]=Edge{v,head[u]};
    head[u]=cnt++;
    edge[cnt]=Edge{u,head[v]};
    head[v]=cnt++;
}
int clk,in[N],out[N],rk[N];
void dfs(int u,int f){
    in[u]=++clk;
    rk[clk]=u;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].v;
        if(v==f){
            continue;
        }
        dfs(v,u);
    }
    out[u]=clk;
}
int sum[N*4],lazy[N*4];
void pushup(int i){
    sum[i]=sum[lson]+sum[rson];
}
void build(int i,int l,int r){
    if(l==r){
        sum[i]=a[rk[l]];
        return;
    }
    build(lson,l,mid);
    build(rson,mid+1,r);
    pushup(i);
}
void pushdown(int i,int l,int r){
    if(lazy[i]){
        lazy[lson]^=lazy[i];
        lazy[rson]^=lazy[i];
        sum[lson]=mid-l+1-sum[lson];
```

```
        sum[rson]=r-mid-sum[rson];
        lazy[i]=0;
    }
}
void update(int i,int l,int r,int ql,int qr){}
int query(int i,int l,int r,int ql,int qr){}
bool isF[N];
int main(void){
    scanf("%d",&n);
    init();
    for(int i=2;i<=n;i++){
        scanf("%d",&x);
        add(x,i);
        isF[x]=true;
    }
    dfs(1,0);
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i]);
    }
    build(1,1,n);
    scanf("%d",&q);
    while(q--){
        scanf("%s %d",s,&o);
        if(s[0]=='p'){
            //区间更新，如果单点更新 in[o] 即可
            update(1,1,n,in[o],out[o]);
        }else if(s[0]=='g'){
            //子树查询转化为区间查询
            int ans=query(1,1,n,in[o],out[o]);
            printf("%d\n",ans);
        }
    }
    return 0;
}
```

dfs 序

```
/*
给定一颗树，和每个节点的权值．有 7 个经典的关于 dfs 序的问题：
1．对某个节点 X 权值加上一个数 W，查询某个子树 X 里所有点权的和．
由于 X 的子树在 DFS 序中是连续的一段，只需要维护一个 dfs 序列，用树状数组实现：单点修改和区间查询．

2．对节点 X 到 Y 的最短路上所有点权都加一个数 W，查询某个点的权值．
这个操作等价于
a．对 X 到根节点路径上所有点权加 W
b．对 Y 到根节点路径上所有点权加 W
c．对 LCA(x，y) 到根节点路径上所有点权值减 W
d．对 LCA(x,y) 的父节点 fa(LCA(x，y)) 到根节点路径上所有权值减 W
当单点更新 X 时,X 实现了对 X 到根的路径上所有点贡献了 W.
于是只需要更新四个点 (单点更新)，查询一个点的子树内所有点权的和 (区间求和) 即可．
用树状数组维护差分前缀和
add(st[u]，w，cnt);
add(st[v]，w，cnt);
add(lca，-w，cnt);
add(parent[lca]，-w，cnt);

3．对节点 X 到 Y 的最短路上所有点权都加一个数 W，查询某个点子树的权值之和．
同问题 2 中的修改方法，转化为修改某点到根节点的权值加/减 W
当修改某个节点 A，查询另一节点 B 时
只有 A 在 B 的子树内，Y 的值会增加
W*(dep[A]-dep[B]+1)=>W*(dep[A]+1)-W*dep[B]
那么我们处理两个数组就可以实现：
处理出数组 sum1，每次更新 W*(dep[A]+1)，和数组 sum2，每次更新 W.
每次查询结果为 sum1(R[B])-sum1(L[B]-1)-(sum2(R[B])-sum2(L[B]-1))*dep[B].


4．对某个点 X 权值加上一个数 W，查询 X 到 Y 路径上所有点权之和．
求 X 到 Y 路径上所有的点权之和，和前面 X 到 Y 路径上所有点权加一个数相似
这个问题转化为
X 到根节点的和 +Y 到根节点的和-LCA(x，y) 到根节点的和-fa(LCA(x,y)) 到根节点的和
更新某个点 x 的权值时，只会对它的子树产生影响，对 x 的子树的每个点到根的距离都加了 W.
那么我们用差分前缀和，更新一个子树的权值．给 L[x] 加上 W，给 R[x]+1 减去 W
那么 sum(1~L[k]) 就是 k 到根的路径点权和．

5．对节点 X 的子树所有节点加上一个值 W，查询 X 到 Y 的路径上所有点的权值和
同问题 4 把路径上求和转化为四个点到根节点的和
X 到根节点的和 +Y 到根节点的和-LCA(x,y) 到根节点的和-parent(LCA(x,y)) 到根节点的
修改一点 A，查询某点 B 到根节点时，只有 B 在 A 的子树内，A 对 B 才有贡献．
贡献为 W*(dep[B]-dep[A]+1)=>W*(1-dep[A])+W*dep[B]
和第三题一样，用两个 sum1,sum2 维护 W*(dep[A]+1)，和 W.
最后答案就是 sum2*dep[B]-sum1.

6．对子树 X 里所有节点加上一个值 W，查询某个点的值．
对 DFS 序来说，子树内所有节点加 W，就是一段区间加 W，即区间修改，单点查询．

7．对子树 X 里所有节点加上一个值 W，查询某个子树的权值和．
子树所有节点加 W，就是某段区间加 W，查询某个子树的权值和，就是查询某段区间的和
*/
```

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef pair<int,int> pii;
const int N=1e5+50;
//三维 LIS(偏序), 第一维排序, 第二维 cdq 分治, 第三维树状数组
struct node{
    int x,y,z,id;
    bool operator <(const node &r)const{
        return x!=r.x?x<r.x:y!=r.y?y<r.y:z<r.z;
    }
}q[N],tmp[N];
//维护一个二维的权值 bit(i 表示 z 坐标 x 表示 dp 状态) first 表示长度 second 表示大小
//普通 bit 表示前缀和, 这里的 bit 表示前 i 个三维点的 LIS(长度和方案数)
pii dp[N],c[N];
pii zero(0,0);
int lowbit(int x){
    return x&(-x);
}
//实现一个 pii 的 dp 状态的相加
void update(pii &a,pii &b){
    if(a.first<b.first){
        a=b;  //长度大于当前的, 更新长度
    }else if(a.first==b.first){
        a.second+=b.second;  //长度相等的, 更新方案数
    }
}
void add(int i,pii x){
    while(i<=n){
        update(c[i],x); //bit 里维护的是一个 pair, 所以这里不是简单的 c[i]+=x
        i+=lowbit(i);
    }
}
pii sum(int i){
    pii ans=zero;
    while(i>0){
        update(ans,c[i]);
        i-=lowbit(i);
    }
    return ans;
}
void clr(int i){
    while(i<=n){
        c[i]=zero;
        i+=lowbit(i);
    }
}
void cdq(int l,int r){
    if(l>=r){
        return;
    }
    int mid=(l+r)/2;
    cdq(l,mid);  //单独处理左区间, 即处理出前半部分的 dp 状态
    //利用左区间信息更新右区间
    //tmp 临时记录这个区间内的操作
    int k=0;
    for(int i=l;i<=r;i++){
```

```
        tmp[k]=q[i];
        tmp[k++].x=0;  //第一维 x 已经降掉
    }
    sort(tmp,tmp+k);  //对 y 和 z 排序
    for(int i=0;i<k;i++){
        node& t=tmp[i];
        //这里 tmp 是排好序的，所以先出来的 y 和 z 肯定小
        if(t.id<=mid){
            //属于左区间的操作，已经在 cdq(l,mid) 中处理完毕
            //加入权值 bit，表示 t.z 累加上 dp[t.id] 个
            add(t.z,dp[t.id]);
        }else{
            //属于右区间的操作 (未操作)，查询 t.z 前面的 dp 状态前缀和
            //并累加到当前 id 的 dp 状态
            pii a=sum(t.z);
            a.first++;
            update(dp[t.id],a);
        }
    }
    //处理右区间前把属于左区间的操作的 bit 清空
    for(int i=0;i<k;i++){
        if(tmp[i].id<=mid){
            clr(tmp[i].z);
        }
    }
    cdq(mid+1,r);  //单独处理右区间
}
int main(void){
    int T;
    scanf("%d",&T);
    while(T--){
        scanf("%d",&n);
        for(int i=0;i<n;i++){
            scanf("%d%d%d",&x,&y,&z);
            q[i]=node{x,y,z,i};
            Z[i]=z;
        }
        sort(q,q+n);
        sort(Z,Z+n);  //离散化 z
        kn=unique(Z,Z+n)-Z;
        for(int i=0;i<n;i++){
            q[i].z=lower_bound(Z,Z+kn,q[i].z)-Z+1;
            q[i].id=i;  //注意是排序后的 id
        }
        for(int i=0;i<n;i++){
            dp[i].first=dp[i].second=1;
        }
        cdq(0,n-1);
        pii ans=zero;
        for(int i=0;i<n;i++){
            update(ans,dp[i]);  //直接 update 累加，同时更新 LIS 和方案数
        }
        printf("%d %d\n",ans.first,ans.second);
    }
    return 0;
}
```