最小生成树(MST)

kruskal算法，先给边排序，然后从小到大依据并查集合并

```
struct edge
{
    int u, v, cost;
    bool operator<(const edge& m1)
    {
        return cost < m1.cost;
    }
}e[];

int V, E;              //V表示点，E表示边
int Find(int x)
{
    if(pre[x] != x)
        pre[x] = Find(pre[x]);
    return pre[x];
}


int kruskal()
{
    sort(e, e + E);             //如果是从e[0]记录边的话
    for(int i = 1; i <= V; i++) pre[i] = i;     //如果点是从1开始
    int ans = 0, x, y, cnt = 0;
    for(int i = 0; i < E && cnt < V - 1; i++)
    {
        x = Find(e[i].u);
        y = Find(e[i].v);
        if(x != y)
        {
            cnt++;
            pre[x] = y;
            ans += e[i].cost;
        }
    }
    if(cnt == V - 1) return ans;
    return -1;
}
```

线段树

build

```
LL arr[maxn], tree[maxn<<2], lazy[maxn<<2];

void build(int node, int l, int r)
{
    if(l == r)
    {
```

```
        tree[node] = arr[l];
        return;
    }
    int mid =  (l + r) / 2;
    build(node * 2, l, mid);
    build(node * 2 + 1, mid + 1, r);
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
```

简单的单点查询

```
int query(int node, int l, int r, int x)
{
    if(x == l)
        return tree[node];
    int ans = 0, mid = (l + r) / 2;
    if(x <= mid) ans += query(node * 2, l, mid, x);
    if(x > mid) ans+= query(node * 2 + 1, mid + 1, r, x);
    return ans;
}
```

简单的区间查询

```
int query(int node, int l, int r, int x, int y)
{
    if(x <= l && y >= r)
        return tree[node];
    int ans = 0, mid = (l + r) / 2;
    if(x <= mid) ans += query(node * 2, l, mid, x, y);
    if(y > mid) ans+= query(node * 2 + 1, mid + 1, r, x, y);
    return ans;
}
```

简单的更新

```
void update(int node, int l, int r, int x, int c)
{
    if(l == r)
    {
        tree[node] += c;
        return;
    }
    int mid = (l + r) / 2;
    if(x <= mid) update(2*node, l, mid, x, c);
    else update(2 * node + 1, mid + 1, r, x, c);
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
//若是区间更新的话直接更新到叶节点(不需要pushdown)
void update(int node, int l, int r, int x, int y, int c)
{
    if(l == r)
    {
        tree[node] += c;
        return;
    }
```

```
    int mid = (l + r) / 2;
    if(x <= mid)  update(2*node, l, mid, x, y, c);
    if(y > mid) update(2 * node + 1, mid + 1, r, x, y, c);
    tree[node] = tree[node*2] + tree[node*2+1];
}
```

标记节点

```
//相加型 (若是覆盖型所有的+=变成=)  具体如何传递看lazy记录的是什么
void push_down(int node, int length)
{
    if(lazy[node])
    {
        tree[2 * node] += lazy[node] * ((length + 1) / 2);    //向子节进行更新
        tree[2 * node + 1] += lazy[node] * (length / 2);
        lazy[2 * node] += lazy[node];                          //标记子节点
        lazy[2 * node + 1] += lazy[node];
        lazy[node] = 0;                                        //父节点标记清空
    }
}

LL query(int node, int l, int r, int x, int y)
{
    if(x <= l && y >= r)
        return tree[node];

    push_down(node, r  - l + 1);           //多的地方

    int mid = (l + r) / 2;
    LL ans = 0LL;
    if(x <= mid) ans += query(node * 2, l, mid, x, y);
    if(y > mid) ans += query(node * 2 + 1, mid + 1, r, x, y);
    return ans;
}

void update(int node, int l, int r, int x, int y, LL c)
{
    if(x <= l && y >= r)
    {
        tree[node] += (r - l + 1) * c;
        lazy[node] += c;
        return;
    }

    push_down(node, r - l + 1);

    int mid = (l + r) / 2;
    if(x <= mid) update(2*node, l, mid, x, y, c);
    if(y > mid) update(2 * node + 1, mid + 1, r, x, y, c);
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
```

树剖

对一棵树分成几条链，把树形变为线性，减少处理难度

注：①图用链式前向星存储，在 dfs1 和 dfs2 中会用到

②线段树用了lazy标记

③点从1开始计数，默认根的深度为0，根的父亲为0，若是一个结点没有儿子，son 值为0， id存储的是某个结点对应的 dfs 序，dfn 存储 dfs 序中的结点顺序

④有多组样例时，build要注意初始化，还有lazy标记，num(链式前向星边个数) 和 tol (dfs序标记) 要设置为 0

⑤不要忘记build

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
const int maxn = 1e5 + 10;
const int INF = 0x3f3f3f3f;
 ll mod = 998244353;

int w[maxn], n, m, r;

struct edge                         //链式前向星
{
    int to, next;
}e[maxn*2];

int head[maxn], num;          //head为0表示搜索到了尽头

void add_edge(int u, int v)
{
    e[++num].to = v;
    e[num].next = head[u];
    head[u] = num;
}

ll tree[maxn<<2], lazy[maxn<<2], dfn[maxn];                  //线段树模板

void build(int node, int l, int r)
{
    if(l == r)
    {
        tree[node] = w[dfn[l]] % mod;
        return;
    }
    int mid =  (l + r) / 2;
    build(node * 2, l, mid);
    build(node * 2 + 1, mid + 1, r);
    tree[node] = (tree[node * 2] + tree[node * 2 + 1]) % mod;
}

void push_down(int node, int length)
{
    if(lazy[node])
```

```
    {
        tree[2 * node] += lazy[node] * ((length + 1) / 2);    //向子节进行更新
        tree[2 * node + 1] += lazy[node] * (length / 2);
        lazy[2 * node] += lazy[node];                          //标记子节点
        lazy[2 * node + 1] += lazy[node];
        lazy[node] = 0;                                        //父节点标记清空

        tree[2 * node] %= mod;   tree[2 * node + 1] %= mod;
    }
}

ll query(int node, int l, int r, int x, int y)
{
    if(x <= l && y >= r)
        return tree[node];

    push_down(node, r  - l + 1);            //多的地方

    int mid = (l + r) / 2;
    ll ans = 0LL;
    if(x <= mid) ans += query(node * 2, l, mid, x, y);
    if(y > mid) ans += query(node * 2 + 1, mid + 1, r, x, y);
    return ans % mod;
}

void update(int node, int l, int r, int x, int y, ll c)
{
    if(x <= l && y >= r)
    {
        tree[node] += (r - l + 1) * c;
        tree[node] %= mod;
        lazy[node] += c;
        return;
    }

    push_down(node, r - l + 1);

    int mid = (l + r) / 2;
    if(x <= mid) update(2*node, l, mid, x, y, c);
    if(y > mid) update(2 * node + 1, mid + 1, r, x, y, c);
    tree[node] = (tree[node * 2] + tree[node * 2 + 1]) % mod;
}

int siz[maxn], son[maxn], top[maxn], dep[maxn], faz[maxn], id[maxn], tol;

void dfs1(int x)
{
    siz[x] = 1;
    son[x] = 0;
    for(int i = head[x]; i; i = e[i].next)
    {
        int v = e[i].to;
        if(v == faz[x])  continue;
        dep[v] = dep[x] + 1;              //标记深度
        faz[v] = x;                       //标记父亲
        dfs1(v);
        siz[x] += siz[v];                 //记录子树个数
        if(siz[v] > siz[son[x]])  son[x] = v;          //标记重儿子
```

```
    }
}

void dfs2(int x, int rt)
{
    id[x] = ++tol;                      //记录在dfs序中的编号
    dfn[tol] = x;
    top[x] = rt;                         //记录所在链深度最小的结点
    if(son[x])  dfs2(son[x], rt);
    for(int i = head[x]; i; i = e[i].next)
    {
        int v = e[i].to;
        if(v == faz[x] || v == son[x])  continue;
        dfs2(v, v);
    }
}

void updRange(int u, int v, int val)
{
    while(top[u] != top[v])               //当两个点不在一条链上
    {
        if(dep[top[u]] < dep[top[v]])  swap(u, v);  //让u为所在链顶端深度更大的那个点
        update(1, 1, n, id[top[u]], id[u], val);
        u = faz[top[u]];
    }
    if(dep[u] > dep[v]) swap(u, v);     //让u为深度浅的那个点
    update(1, 1, n, id[u], id[v], val);
}

ll qRange(int u, int v)
{
    ll ans = 0;
    while(top[u] != top[v])               //当两个点不在一条链上
    {
        if(dep[top[u]] < dep[top[v]])  swap(u, v);  //让u为所在链顶端深度更大的那个点
        ans +=  query(1, 1, n, id[top[u]], id[u]);
        ans %= mod;
        u = faz[top[u]];
    }
    if(dep[u] > dep[v]) swap(u, v);     //让u为深度浅的那个点
    ans += query(1, 1, n, id[u], id[v]);
    return ans % mod;
}

void updSon(int x, int val)
{
    update(1, 1, n, id[x], id[x] + siz[x] - 1, val);
}

ll qSon(int x)
{
    return query(1, 1, n, id[x], id[x] + siz[x] - 1) % mod;
}

int main()
{
    scanf("%d %d %d %lld", &n, &m, &r, &mod);
    for(int i = 1; i <= n; i++) scanf("%d", &w[i]);
```

```
    for(int i = 1; i < n; i++)
    {
        int x, y;
        scanf("%d %d", &x, &y);
        add_edge(x, y);  add_edge(y, x);
    }
    dfs1(r);
    dfs2(r, r);
    build(1, 1, n);

    int mark, x, y, z;
    for(int i = 1; i <= m; i++)
    {
        scanf("%d", &mark);
        if(mark == 1)
        {
            scanf("%d %d %d", &x, &y, &z);
            updRange(x, y, z);
        }
        else if(mark == 2)
        {
            scanf("%d %d", &x, &y);
            printf("%lld\n", qRange(x, y));
        }
        else if(mark == 3)
        {
            scanf("%d %d", &x, &z);
            updSon(x, z);
        }
        else
        {
            scanf("%d", &x);
            printf("%lld\n", qSon(x));
        }
    }
}
```

题目大意是说对于一棵树，初始所有点的权值 $F(x)$ 为0，有三种操作：①对于结点 x, 给定一个值 w，然后对于树上所有结点 y, 权值加上 $w-dist(x,y)$（包括本身）；② 对于结点 x，$F(x)=min(x,0)$；③询问 $F(x)$。

对于操作①的转化很巧妙：$w-dist(x,y)=w-dep(x)-dep(y)+2dep(lca(x,y))$，其中每次①操作的 $w-dep(x)$ 都可以记录下来，$dep(y)$ 也是一个定值，而对于 $dep(lca(x,y))$, 可以用树剖来记录，若记根节点 rt 深度为 1，只需要让结点 x 到根结点路径上的结点加上 2，$dep(lca(x,y))$就是结点 y 到根结点的权值和。

然后对于操作②，若是某个结点的权值大于 0， 我们可以用数组去记录减去的数是多少。

刚学的树剖模板，千万**不要忘记初始化**

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
const int maxn = 5e4 + 10;
```

```cpp
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

int n, m;

struct edge                         //链式前向星
{
    int to, next;
}e[maxn*2];

int head[maxn], num;          //head为0表示搜索到了尽头

void add_edge(int u, int v)
{
    e[++num].to = v;
    e[num].next = head[u];
    head[u] = num;
}

ll tree[maxn<<2], lazy[maxn<<2], dfn[maxn];                  //线段树模板

void build(int node, int l, int r)
{
    if(l == r)
    {
        tree[node] = 0;
        lazy[node] = 0;
        return;
    }
    int mid =  (l + r) / 2;
    build(node * 2, l, mid);
    build(node * 2 + 1, mid + 1, r);
    tree[node] = 0;
    lazy[node] = 0;
}

void push_down(int node, int length)
{
    if(lazy[node])
    {
        tree[2 * node] += lazy[node] * ((length + 1) / 2);    //向子节进行更新
        tree[2 * node + 1] += lazy[node] * (length / 2);
        lazy[2 * node] += lazy[node];                          //标记子节点
        lazy[2 * node + 1] += lazy[node];
        lazy[node] = 0;                                        //父节点标记清空
    }
}

ll query(int node, int l, int r, int x, int y)
{
    if(x <= l && y >= r)
        return tree[node];

    push_down(node, r  - l + 1);           //多的地方

    int mid = (l + r) / 2;
    ll ans = 0LL;
    if(x <= mid) ans += query(node * 2, l, mid, x, y);
```

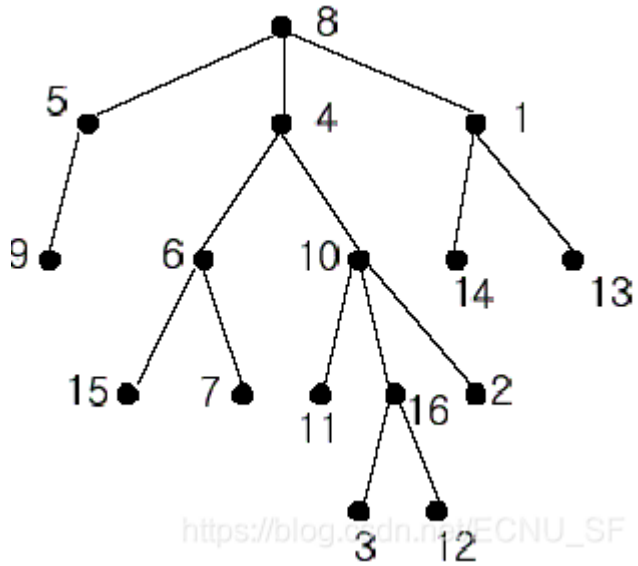
```
    if(y > mid) ans += query(node * 2 + 1, mid + 1, r, x, y);
    return ans;
}

void update(int node, int l, int r, int x, int y, int c)
{
    if(x <= l && y >= r)
    {
        tree[node] += (r - l + 1) * c;
        lazy[node] += c;
        return;
    }

    push_down(node, r - l + 1);

    int mid = (l + r) / 2;
    if(x <= mid) update(2*node, l, mid, x, y, c);
    if(y > mid) update(2 * node + 1, mid + 1, r, x, y, c);
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

int siz[maxn], son[maxn], top[maxn], dep[maxn], faz[maxn], id[maxn], tol;

void dfs1(int x)
{
    siz[x] = 1;
    son[x] = 0;
    for(int i = head[x]; i; i = e[i].next)
    {
        int v = e[i].to;
        if(v == faz[x])  continue;
        dep[v] = dep[x] + 1;              //标记深度
        faz[v] = x;                       //标记父亲
        dfs1(v);
        siz[x] += siz[v];                 //记录子树个数
        if(siz[v] > siz[son[x]])  son[x] = v;          //标记重儿子
    }
}

void dfs2(int x, int rt)
{
    id[x] = ++tol;                      //记录在dfs序中的编号
    dfn[tol] = x;
    top[x] = rt;                         //记录所在链深度最小的结点
    if(son[x])  dfs2(son[x], rt);
    for(int i = head[x]; i; i = e[i].next)
    {
        int v = e[i].to;
        if(v == faz[x] || v == son[x])  continue;
        dfs2(v, v);
    }
}

void updRange(int u, int v, int val)
{
    while(top[u] != top[v])               //当两个点不在一条链上
    {
        if(dep[top[u]] < dep[top[v]])  swap(u, v);  //让u为所在链顶端深度更大的那个点
```

```
        update(1, 1, n, id[top[u]], id[u], val);
        u = faz[top[u]];
    }
    if(dep[u] > dep[v]) swap(u, v);      //让u为深度浅的那个点
    update(1, 1, n, id[u], id[v], val);
}

ll qRange(int u, int v)
{
    ll ans = 0;
    while(top[u] != top[v])                //当两个点不在一条链上
    {
        if(dep[top[u]] < dep[top[v]])  swap(u, v);  //让u为所在链顶端深度更大的那个点
        ans +=  query(1, 1, n, id[top[u]], id[u]);
        u = faz[top[u]];
    }
    if(dep[u] > dep[v]) swap(u, v);      //让u为深度浅的那个点
    ans += query(1, 1, n, id[u], id[v]);
    return ans;
}

ll pre[maxn];

int main()
{
    int t, mark, x, w;
    scanf("%d", &t);
    while(t--)
    {
        scanf("%d %d", &n, &m);
        fill(head + 1, head + 1 + n, 0);
        fill(pre + 1, pre + 1 + n, 0);
        num = 0; tol = 0;
        for(int i = 1; i < n; i++)
        {
            int x, y;
            scanf("%d %d", &x, &y);
            add_edge(x, y);  add_edge(y, x);
        }

        dep[1] = 1;
        dfs1(1);
        dfs2(1, 1);
        build(1, 1, n);
        int cnt = 0;
        ll cur = 0, tmp;
        for(int i = 1; i <= m; i++)
        {
            scanf("%d", &mark);
            if(mark == 1)
            {
                scanf("%d %d", &x, &w);
                cur += w - dep[x];
                cnt++;
                updRange(1, x, 2);
            }
            else if(mark == 2)
            {
```

```
                scanf("%d", &x);
                tmp = cur - cnt * dep[x] + qRange(1, x) + pre[x];
                if(tmp > 0)
                    pre[x] -= tmp;
            }
            else
            {
                scanf("%d", &x);
                tmp = cur - cnt * dep[x] + qRange(1, x) + pre[x];
                printf("%lld\n", tmp);
            }
        }
    }
}
```

先理解LCA模板

在有根树中，两个结点 u, v 的公共祖先中距离最近的那个被称为最近公共祖先(LCA, Lowest Common Ancestor)。比如在下图中，根是结点8，LCA(3, 11) = 10, LCA(15, 12) = 4。



在有根树中由于根的存在，结点与根结点的最短路径长度为这个结点的深度(depth)。那么，如果 $LCA(u,v)=w$，让 u 向上走 $depth(u)-depth(w)$ 步，让 v 向上走 $depth(v)-depth(w)$ 步,就都可以走到 w。因此，**首先让 u,v 中较深的一方走 $|depth(u)-depth(v)|$ 步，再一起一步一步向上走，直到走到同一个结点**，就可以在线性时间里求出LCA。

而有没有更优的算法呢？ST表可以做到！

我们可以用父结点的信息推出子节点的信息，如 $parent2[v]=parent[parent[v]]$，$parent4[v]=parent2[parent2[v]]$，若我们记录 $f[i][k]$ 为 i 结点往上的第 2^k 个结点，我们可以得到以下的转移关系：

$f[i][0]=parent[i],\quad f[i][j]=f[i][f[i][j-1]]$(如果 $f[i][j-1]$存在)

所以我们在一开始时，可以**预处理得到每个结点的深度与父结点**即 $f[i][0]$，而类似于区间 dp，我们也可以通过两重循环，其中第一重为**区间长度**，因为长度长的区间必须通过短区间合并而来，第二重为**各个结点**：

```
void dfs(int x, int fa, int d)      //通过dfs预处理所有结点的父结点与深度
{
    f[x][0] = fa;                   //x的父结点即为f[x][0]
    depth[x] = d;                   //得到长度
```

```
    for(unsigned int i = 0; i < G[x].size(); i++)
        if(G[x][i] != fa)
            dfs(G[x][i], x, d + 1);
}

dfs(root, 0, 0);
for(int j = 1; j < MAX_LOG_V; j++)    //先循环步数
{
    for(int i = 1; i <= n; i++)   //再循环结点
    if(f[i][j-1])             //如果f[i][j-1]存在
        f[i][j] = f[f[i][j-1]][j-1];
}
```

而得到了 st表，该如何求解LCA呢？首先第一步还是让 u,v走到同一深度，由于 st 表的第二维是指数级别的增长，因此相比于线性时间的一步步地上升，我们可以**用对数时间迅速地上升至同一高度。**

在上升到同一高度之后，原来的方法是两个结点同时一步步地上升直到到达同一个结点，而我们现在可以**采取二分的方式。**比如两个结点距离 LCA 的距离为 15， 若是同时上升 16，则会超过 LCA，于是不上升；同时上升 8，距离变成 7，再上升4，距离变成3，再上升2，距离变成1，最后一起上升 1，到达 LCA。所以这里步数是从大到小进行遍历，这个处理技巧是非常关键的：

```
int lca(int u, int v)
{
    if(depth[v] > depth[u]) swap(u, v);        //让u的深度更大

    for(int i = 0; i <= MAX_LOG_V; i++)                    //让u,v到达统一深度
        if((depth[u] - depth[v]) >> i & 1)
            u = f[u][i];

    if(u == v) return u;
    for(int k = MAX_LOG_V; k >= 0; k--)          //利用二分(st表)来计算LCA
    {
        if(f[u][k] != f[v][k])          //若是不会超出LCA就上升
        {
            u = f[u][k];
            v = f[v][k];
        }
    }
    return f[u][0];
}
```

这样通过预处理 $O(nlogn)$， 每次查询 LCA 的时间就可以变成 $O(logn)$ 了，所以这里面**倍增的思想**很关键。

附上POJ的模板题以及AC代码 link

```
//#include <bits/stdc++.h>
#include <stdio.h>
#include <vector>
#include <string.h>

using namespace std;
typedef long long ll;
```

```cpp
const int maxn = 1e4 + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

vector<int> G[maxn];
int fa[maxn], depth[maxn], f[maxn][15], n, root;

void dfs(int x, int fa, int d)     //通过dfs预处理所有结点的父结点与深度
{
    f[x][0] = fa;              //x的父结点即为f[x][0]
    depth[x] = d;              //得到长度
    for(unsigned int i = 0; i < G[x].size(); i++)
        if(G[x][i] != fa)
            dfs(G[x][i], x, d + 1);
}

int lca(int u, int v)
{
    if(depth[v] > depth[u]) swap(u, v);       //让u的深度更大

    for(int i = 0; i < 15; i++)                    //让u,v到达统一深度
        if((depth[u] - depth[v]) >> i & 1)
            u = f[u][i];

    if(u == v) return u;
    for(int k = 14; k >= 0; k--)         //利用二分(st表)来计算LCA
    {
        if(f[u][k] != f[v][k])
        {
            u = f[u][k];
            v = f[v][k];
        }
    }
    return f[u][0];
}

int main()
{
    int t;
    scanf("%d", &t);
    while(t--)
    {
        int x, y;
        scanf("%d", &n);
        memset(fa, 0, sizeof(fa));
        memset(f, 0, sizeof(f));
        for(int i = 1; i <= n; i++)
            G[i].clear();

        for(int i = 1; i < n; i++)
        {
            scanf("%d %d", &x, &y);
            G[x].push_back(y);
            G[y].push_back(x);
            fa[y] = x;
        }
        for(int i = 1; i <= n; i++)      //找到root
            if(!fa[i])
```

```
            {
                root = i;
                break;
            }

        dfs(root, 0, 0);
        for(int j = 1; j < 15; j++)     //先循环步数
        {
            for(int i = 1; i <= n; i++)    //再循环结点
                if(f[i][j-1])
                    f[i][j] = f[f[i][j-1]][j-1];
        }

        scanf("%d %d", &x, &y);
        printf("%d\n", lca(x, y));
    }
}
```

ST表的用处

在上面，用到了一种很厉害的数据结构 ST表，其实这也是动态规划的一种，**用统计的思想来解决问题。**在上面的有根树之中，若要记录每个结点 i 往上 k 步是哪一个结点，则需要一个二维数组，我们可以进行对比：

	普通二维数组	ST表
空间复杂度	$O(V^2)$	$O(Vlog(dep))$
时间复杂度	$O(V^2)$	$O(Vlog(dep))$
查询时间	$O(1)$	$O(log(dep))$

因此 **ST表虽然没有记录全部的信息，但是却可以在对数时间里得到我们想要的信息**，有点像线段树，树状数组。ST表一个很大的用处是求解区间的最大/小值。

比如求最大值，对于数组 $a[i]$, $dp[i][j]$ 表示从下标 i 开始长度为 2^j 的区间的最大值，可以得出 $dp[i][0] = a[0]$, 而 j 每增加1，就说明区间长度增倍，则有以下转移式：

$dp[i][j] = max(dp[i][j-1], dp[i+2^{j-1}][j-1]$

这样子就可以在 $O(nlogn)$ 的时间里维护出 ST表。那该如何求出某个区间 $[L, R]$ 的最大值呢？比如区间 $[3, 8]$，长度虽然不是2的幂次，但可以通过分别查找区间 $[3, 6]$ 和 区间 $[5, 8]$，即$dp[3][2]$ 与 $dp[5][2]$ 得到。所以两个长度大于等于 $(R - L + 1)/2$ 的区间 (一个左边界为 L， 一个右边界为 R) 可以覆盖我们要求的区间。

```
void init_st(int n)
{
    for(int i = 1; i <= n; i++)  //区间跨度为1
        f[i][0] = a[i];
    int m = (int)(log((double)n) / log(2.0));   //算出最大跨度2^m

    for(int j = 1; j <= m; j++)                 //第一维为区间长度
        for(int i = 1; i + (1<<j) - 1 <= n; i++)
            f[i][j] = max(f[i][j-1], f[i+(1<<(j-1))][j-1]);
}

int Find(int l, int r)
```

```
{
    int k = (int)(log(1.0 * (r-l+1)) / log(2.0));      //求出两个区间的长度
    return max(f[l][k], f[r-(1<<k)+1][k]);
}
```

下面来看一道 ST表稍微难一点的运用：link，大意是给你一个长为 n 的序列和一个常数 k 有 m 次询问，每次查询一个区间 $[l,r]$ 内所有数最少分成多少个连续段，使得每段的和都 $\leq k$，如果这一次查询无解，输出 "Chtholly"，其中 $(n,m \leq 1e6)$

由于询问次数很多，只能把每次询问时间压到对数或者常数级别，而我们可以用 st 表来记录这些信息。若记 $dp[i][0]$ **表示区间从当前开始，下一个区间最晚的位置**，即 $[i,dp[i][0]-1]$ 求和小于等于 k, $[i,dp[i][0]]$ 大于 k, 那么相应地 $dp[i][j]$ 就是下 2^j 个区间最晚开始的位置，我们可以得到以下转移方程：

$$dp[i][j]=dp[dp[i][j-1]][j-1]$$

而这里需要考虑到两种情况：

①若是有一个数大于 k， 那么这个值是无法在区间里的，在处理时就非常麻烦，所以在读入数据时，我们可以标记哪些位置的数值大于 k, 然后**记录前缀和**，这样就可以快速判断一个区间里是不是有大于 k 的数了，然后我们 st 表只处理小于等于 k 的数。

②若是区间可以包含到最后一个数都不超过 k,也需要特殊处理，我就让 $dp[i][j]=0$ 表示可以扩展到最后一个元素。

而求 $dp[i][0]$, 可以用尺取法维护一个区间，其他细节可以参照代码来看：

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
const int maxn = 1e6 + 3;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

int a[maxn], b[maxn], f[maxn][21], n, m, k, tol;

int main()
{
    scanf("%d %d %d", &n, &m, &k);
    for(int i = 1; i <= n; i++)
    {
        scanf("%d", &a[++tol]);
        if(a[tol] > k)        //只保留小于等于k的数
        {
            b[i] = 1;      //超过k的下标用b数组标记
            tol--;
        }
    }
    for(int i = 2; i <= n; i++)     //求前缀和，现在b[i]表示[1,i]超过k的数
        b[i] += b[i-1];

    int l = 1, r = 1;
    ll tmp = 0;
    while(l <= tol)
    {
        while(r <= n && tmp + a[r] <= k)      //[l, r)区间和小于等于k，找到r的最大值
        {
            tmp += a[r];
            r++;
```

```
        }

        if(r == n + 1)  f[l][0] = 0;          //如果剩下的所有元素都被包含了，用0表示
        else f[l][0] = r;
        tmp -= a[l];
        l++;
    }

    for(int j = 1; j <= 20; j++)
        for(int i = 1; i + (1<<j) <= n; i++)
        {
            if(f[i][j-1] == 0 || f[f[i][j-1]][j-1] == 0)  //有这个条件可以直接break
                break;
            f[i][j] = f[f[i][j-1]][j-1];
        }

    for(int cas = 1; cas <= m; cas++)
    {
        int ans = 0;
        scanf("%d %d", &l, &r);
        if(b[r] - b[l-1] > 0)       //如果区间包含有大于k的元素
        {
            printf("Chtholly\n");
            continue;
        }
        r -= b[r];  l -= b[l-1];          //得到现在的下标
        for(int j = 20; j >= 0; j--)
        {
            if(f[l][j] <= r && f[l][j] > 0)
            {
                ans += 1<<j;
                l = f[l][j];
            }
        }
        printf("%d\n", ans + 1);

    }

}
```

ST表比较灵活的使用

还是从题目开始入手吧，其实我一开始是做到了接下来的这道题，然后才了解了 ST表和树上倍增……所以是通过题来花时间了解数据结构，然后最后一步当然是看看这题是如何运用的啦。

link有一个树状的城市网络（即 n 个城市由 $n-1$ 条道路连接的连通图），首都为 1 号城市，每个城市售卖价值为 a_i 的珠宝。你是一个珠宝商，现在安排有 q 次行程，每次行程为从 u 号城市前往 v 号城市（走最短路径），保证 v 在 u 前往首都的最短路径上。 在每次行程开始时，你手上有价值为 c 的珠宝（每次行程可能不同），并且每经过一个城市时（包括 u 和 v ），假如那个城市中售卖的珠宝比你现在手上的每一种珠宝都要优秀（价值更高，即严格大于），那么你就会选择购入。现在你想要对每一次行程，求出会进行多少次购买事件。其中，$2 \leq n \leq 10^5, 1 \leq q \leq 10^5$。

理解一下题意，每个结点都有一个权重，然后每次出发时都给你一个值，并保证你是朝根方向往上走，若是到达一个结点，它的权重比所有经过结点权重以及手上的初值要大，就计数 +1。也就是说，如果你在某个结点处买了珠宝，那么这个结点的权重就是你拥有的最大值，那么从哪里走到这个结

点不会影响后续的情况，所以我们不妨记录一下这个信息。

让 $dp[i][j]$ **表示若在结点 i 买了珠宝，往根向上走，再买到第 2^j 个珠宝的结点**，我们可以得到：

$$dp[i][j] = dp[dp[i][j-1]][j-1]$$

这里同时有些特殊情况要处理：

① $dp[i][j]$ 可能并不存在，比如对于根结点 1, $dp[1][j]$ 就是不存在的，不妨我们记 $dp[i][j] = 0$，表示在 i 结点出发，买到第 2^j 件珠宝之前，就已经到达了根结点；

② $dp[i][0]$ 是表示下一个买珠宝的城市，这个在动态规划前需要先求出来。如果它的父节点 fa 权重大于 i, 那么 $dp[i][0] = fa$, 若小于它，该怎么求呢？比如举例，父结点之后买的 6 件珠宝权重都小于等于结点 i, 直到第 7 个结点，我们可以从大步长开始跨越，那么我们分别跨越 4，2，1 个步长到达这个结点(要是难以理解，可以看具体代码的实现)

而现在题目要求的是带着珠宝 c 从某个结点 u 出发，到达u, 而可以让 u **多出一个儿子** u'，u' 权重为 c，这样我们就是要求 $dp[u'][v]$, **求的时候我们也是从大步长开始跨越，可以通过在树中的深度来判断有没有超过目标结点。**

为什么要从大步长开始呢？这其实是 st表中一个关键的处理，比如步长相差6，从小步长就会先跨越 1, 再跨越 2, 但是4 就不行了，这样会凑不齐，而我们从大步长跨越，就会先跨越 4, 再跨越 2，这样可以凑出 6 出来。

下面就是具体的代码实现咯，总结一下， **st表关键之处是给出适合题目的定义**，题目的解可以用 $dp[i][j]$ 用对数时间求出来，然后预处理时要求出 $dp[i][0]$， **要想清楚什么地方步长要从大到小开始遍历。**

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
const int maxn = 2e5 + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

int a[maxn], depth[maxn], to[maxn], f[maxn][16], n, q;
vector<int> G[maxn];

void dfs(int x, int fa, int step)
{
    int cur = fa;                           //先预处理出dp[x][0]
    for(int j = 15; j >= 0; j--)
        if(f[cur][j] && a[f[cur][j]] <= a[x])
            cur = f[cur][j];
    if(a[cur] > a[x])           //这种情况只会在父节点大于子节点情况发生
        f[x][0] = cur;
    else
        f[x][0] = f[cur][0];

    for(int j = 1; j <= 15; j++)                //进行动态规划
        f[x][j] = f[f[x][j-1]][j-1];

    depth[x] = step;
    for(auto y: G[x])
        if(y != fa)
            dfs(y, x, step + 1);
}

int main()
```

```
{
    scanf("%d %d", &n, &q);
    for(int i = 1; i <= n; i++)
        scanf("%d", &a[i]);
    int x, y;
    for(int i = 1; i < n; i++)
    {
        scanf("%d %d", &x, &y);
        G[x].push_back(y);
        G[y].push_back(x);
    }
    for(int i = n + 1; i <= n + q; i++)          //给每一个出发结点多一个儿子
    {
        scanf("%d %d %d", &x, &to[i-n], &a[i]);
        G[i].push_back(x);
        G[x].push_back(i);
    }

    dfs(1, 0, 0);

    for(int cas = 1; cas <= q; cas++)
    {
        int ans = 0, x = cas + n, y = to[cas];
        for(int i = 15; i >= 0; i--)
            if(f[x][i] > 0 && depth[f[x][i]] >= depth[y])     //用深度来判断有没有
超过目标结点
            {
                ans += 1<<i;
                x = f[x][i];
            }
        printf("%d\n", ans);
    }
}
```

树上差分

点差分(两个点之间都加上某一个权值)

统计以每个节点为根的树的节点的权值和，就是当前节点的最终权值

```
for(int i = 1; i <= k; i++)
{
        int x, y, z;
       scanf("%d %d", &x, &y);
        power[x]++;  power[y]++;            //x, y++
        z = lca(x, y);
        power[z]--;  power[f[z][0]]--;        //lca[x, y]--, fa[lca[x, y]]--;
}

void dfs2(int x, int fa)                     //统计子树的值
{
    for(unsigned int i = 0; i < G[x].size(); i++)
    {
        if(G[x][i] != fa)
```

```
        {
            dfs2(G[x][i], x);
            power[x] += power[G[x][i]];
        }
    }
}
```

边差分(两个点之间的路径都加上某一个权值)

统计以每个节点为根的树的节点的权值和，就是当前节点到父亲节点的边的最终权值(所以除了根节点，都代表一条边)

```
P2680 你一个n个点的树,对于n-1条边各有边权,
给出一些点对(x,y),同时定义dis(x,y)表示x,y两点间的树上距离,
现允许你将一条边的权值变为0,请你最小化最大的dis值
```

进行二分，判断能否将最大值降到某一个值。对于某个值 x 的判断，先找出所有路径长度大于 x 的边，若数量为k, 最大值为 mx(路径长度可以用lca做)，然后求出他们的共同经过的边(这个可以用书上差分做，每条路径所在边权值+1，最后判断权值是不是等于k)，判断这些边中是否有边w 可以做到 mx - w <= x。

```
struct node
{
    int to, w;
    node(int x, int y): to{x}, w{y} {}
};

//lca 模板，求深度的同时求一下与根节点的距离
vector<node> G[maxn];
int f[maxn][19], MAX_LOG_V, depth[maxn], D[maxn];

void dfs(int x, int fa, int d)

int lca(int u, int v)

int n, m, plan[maxn][2], dist[maxn], mx, C[maxn], cnt, tmp_mx, l, r, mid;

int dfs2(int x, int fa)
{
    for(unsigned int i = 0; i < G[x].size(); i++)
    {
        if(G[x][i].to == fa)  continue;             //如果子树中存在这样的边或者当前节点到孩子的边可以做到
        if(dfs2(G[x][i].to, x) || (tmp_mx - G[x][i].w <= mid && C[G[x][i].to] == cnt))  return 1;
        C[x] += C[G[x][i].to];
    }
    return 0;
}

int check(int x)
{
    cnt = 0;
    tmp_mx = 0;
    fill(C + 1, C + 1 + n, 0);
    for(int i = 1; i <= m; i++)
```

```
    {
        if(dist[i] <= x) continue;
        C[plan[i][0]]++;  C[plan[i][1]]++;          //进行边差分，lca += 2
        C[lca(plan[i][0], plan[i][1])] -= 2;
        cnt++;
        tmp_mx = max(tmp_mx, dist[i]);
    }
    return dfs2(1, 0);
}

int main()
{
    scanf("%d %d", &n, &m);        //n 个点， m 条路径
    for(int i = 1; i < n; i++)
    {
        int x, y, z;
        scanf("%d %d %d", &x, &y, &z);
        G[x].push_back(node(y, z));
        G[y].push_back(node(x, z));
    }

    dfs(1, 0, 0);
    MAX_LOG_V = (int)(log((double)n) / log(2.0));
    for(int j = 1; j < MAX_LOG_V; j++)    //先循环步数
    {
        for(int i = 1; i <= n; i++)   //再循环结点
        if(f[i][j-1])             //如果f[i][j-1]存在
            f[i][j] = f[f[i][j-1]][j-1];
    }

    for(int i = 1; i <= m; i++)
    {
        int x, y;
        scanf("%d %d", &x, &y);
        plan[i][0] = x, plan[i][1] = y;
        dist[i] = D[x] + D[y] - 2 * D[lca(x, y)];     //求两条路径的距离
        mx = max(mx, dist[i]);                        //求出最长的
    }

    l = 0, r = mx;          //二分
    while(r - l > 1)
    {
        mid = (l + r) / 2;
        if(check(mid))
            r = mid;
        else l = mid;
    }
    printf("%d\n", r);
}
```

动态开点

大致题意是说给定一个允许有重复整数元素的集合，第一种操作是增加一个整数，第二种操作是删除一个整数，第三种操作是给定一个整数，判断是否能从集合内再找两个整数组成一个三角形。

第一种操作和第二种操作直接用 STL 的 multiset 就可以做到，但是第三种操作就不好维护了。

若是判断的整数 x是三角形的最大边，那么只需要在小于等于 x 的集合元素中挑选两个最大的相加判断是否大于 x；若是判断的整数是中间边，只需要挑选小于等于 x 的最大元素和大于等于 x 的最小元素就可以；若是最小边的话，在大于等于 x 的元素里挑选，一定是挑选两条差值最小的边。所以**最关键的还是动态的记录相邻边的差值。**

赛后看了别人的解法，也是类似的，第一种第二种操作用 map 来记录个数即可，若记第三种操作挑选的元素是 $a, b\ (b \geq a)$，**那么能组成三角形等价于** $b - a < x < b + a$，若是我们记 b 的前驱结点为 b' (小于等于 b 的最大整数，可以相等)，那么我们有 $b - b' \leq b - a < x < b + a \leq b + b'$，那么若存在 a 可以，b' 一定可以。

我们记 $k = lower\_bound(x/2 + 1)$，那么 $b_{min} \geq k$, 若是 $k + k' \leq x$，那么 $b_{min} = k.next$，否则 $b_{min} = k$ (这里可以仔细想一想)，而我们只需要判断在 大于等于 b_{min} 的元素中，有没有和前驱元素差值小于 x 的，这个就由权值线段树来维护。

由于区间可以到 $1e9$，所以采取了动态开点的方式(也是第一次学了这种操作)，大部分与普通线段树差不多，理解理解代码就好啦。若是添加一个元素集合里没有，那么对后面，自身元素有影响，若是集合里只有一个，那么对自身有影响；若是删除一个元素后集合里只有一个，那么对自身有影响，若是集合里就没有了，那么对自身和后面元素有影响。

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
typedef unsigned long long ull;
typedef pair<double, double> P;
const int maxn = 2e5 + 10;
const int MAX = 1e9;
const int INF = 0x3f3f3f3f;
const double eps = 1e-11;
const ll mod = 998244353;

struct node
{
    int val, l, r;
}tree[maxn<<2];                          //这棵线段树记录某个点和其前驱结点(小于等于它本身的最大结点)的差值

int cnt, q, root;
map<int, int> mp;

void update(int &id, int l, int r, int pos, int val)      //单点修改，将位置pos的值更新为val
{
    if(!id)                                 //如果该结点还没有扩展开
        id = ++cnt, tree[id].val = val;
    if(l == r)
    {
        tree[id].val = val;
        return;
    }

    int ans = 2e9, mid = (l + r) / 2;
    if(pos <= mid)  update(tree[id].l, l, mid, pos, val);
```

```
    else  update(tree[id].r, mid + 1, r, pos, val);

    if(tree[id].l)  ans = min(ans, tree[tree[id].l].val);       //if判断该结点是否被扩展开
    if(tree[id].r)  ans = min(ans, tree[tree[id].r].val);
    tree[id].val = ans;
}

int query_min(int id, int l, int r, int x, int y)                  //查询[x, y]区间的最小值
{
    if(!id)  return 2e9;                                        //如果该区间没有结点记录，返回INF
    if(x <= l && y >= r)  return tree[id].val;

    int ans = 2e9, mid = (l + r) / 2;
    if(x <= mid)  ans = min(ans, query_min(tree[id].l, l, mid, x, y));
    if(y > mid)  ans = min(ans, query_min(tree[id].r, mid + 1, r, x, y));
    return ans;
}

void add(int x)
{
    mp[x]++;
    if(mp[x] == 1)                          //如果该结点的值是第一次加入
    {
        auto it = mp.lower_bound(x);
        ++it;
        if(it != mp.end() && it->second == 1)          //如果与后驱结点的差值可更新
            update(root, 1, MAX, it->first, it->first - x);
        it--;
        if(it == mp.begin()) update(root, 1, MAX, x, 2e9);          //得到该结点与前驱结点的差值
        else update(root, 1, MAX, x, x - (--it)->first);
    }
    else if(mp[x] == 2)
        update(root, 1, MAX, x, 0);
}

void del(int x)
{
    if(--mp[x] > 1) return;

    auto it = mp.lower_bound(x);
    int l = -MAX;
    if(it != mp.begin())
    {
        l = (--it)->first;
        it++;
    }
    if(mp[x] == 0)
    {
        update(root, 1, MAX, x, 2e9);                   //更新本身
        it++;
        if(it != mp.end() && it->second == 1)                //更新后驱结点
            update(root, 1, MAX, it->first, it->first - l);
        mp.erase(x);                    //这一步很关键，等于0就要从map中删除
    }
```

```
    else
        update(root, 1, MAX, x, x - l);
}

bool check(int x)
{
    auto it = mp.lower_bound(x / 2 + 1), ip = it;
    if(it == mp.end())  return false;

    if(it -> second > 1)  return true;
    else if(it == mp.begin() || (--ip) -> first + it->first <= x)
        it++;
    if(it == mp.end())  return false;
    return query_min(1, 1, MAX, it -> first, MAX) < x;
}

int main()
{
    scanf("%d", &q);
    for(int i = 1; i <= q; i++)
    {
        int x, y;
        scanf("%d %d", &x, &y);
        if(x == 1)  add(y);
        else if(x == 2)  del(y);
        else
        {
            if(check(y)) puts("Yes");
            else puts("No");
        }
    }
}
```

Boruvka算法&异或字典树

题意是说给定了一棵树，每条边都有一个权值，我们可以进行删边或者增边操作，每次需要保证操作后所有点是连通的，并且保证若是存在环，环上所有值异或和为0。求最小权值和。

由异或的性质可知，从某个顶点 i 到顶点 j 若是有连边，这个连边的值是确定的；若是我们确定了某一个点的值，其他点的值也可以确定下来，路径异或就可以转成顶点异或。比如我们假定 1 号结点的值为 0，这样其他结点的值也可以被确定，**两点间连边的值就等于两点的异或值。**

那么问题就转化成 $n(n \leq 1e5)$ 个结点，每个结点都有权值，两点间的连边值为结点的异或，求最小生成树的代价。kruskal 算法的复杂度为 $O(n^2logn)$, 肯定不行，这时候就需要用到一种神奇的 Boruvka 算法来解决。

Boruvka算法的核心思想是：**从所有当前的连通块向其他连通块扩展出最小边，直到只剩一个连通块。**每次循环找到当前连通块伸出去扩展的最小边，然后连接这些最小边，由于每次连通块至少减半，所以最多进行 $logn$ 次，那么时间复杂度就由每次循环的时间决定了。

而我们做这题其实不需要具体实现 Boruvka 算法，而是需要它的思想。当某个连通块和另一个连通块进行连接时，若异或不为0，记二进制为 1 的最高位为 i， 那么这两个连通块，肯定一个块里第 i 位的点权全为 1，一个全为 0。

这样的话我们就可以用神奇的异或字典树处理，每一个数字都可以用二进制表示塞进这棵树中，每次合并的集合都是最高位的1不同的两个集合进行合并，于是可以从上往下做，从最高位把集合分开，

然后查询两个集合的最小连边。处理时有一个小技巧，可以将所有权值先排序，这样个集合内的元素就在一个连续的区间里。

我觉得洛谷这几个题解写的更好 link, 其实是CF888G的原题，复杂度为 $O(nlog^2n)$。

这个异或字典树太神了

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
typedef pair<int, int> P;
const int maxn = 1e5 + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 1e9 + 7;

int trie[maxn*30][2], L[maxn*30], R[maxn*30], a[maxn];
int root, n, tol;

void Insert(int &now, int i, int dep)          //给a[i]个数插入Trie树，现在是第dep位
{
    if(!now) now = ++tol;
    L[now] = min(L[now], i);  R[now] = max(R[now], i);      //统计这一段有哪些数

    if(dep <= 0) return;                      //到了第0位，不需要往下了
    int bit = a[i]>>(dep-1) & 1;
    Insert(trie[now][bit], i, dep - 1);
}

ll query(int now, int val, int dep)       //从now结点开始，此时是第dep位
{
    if(dep <= 0)  return 0;
    int bit = val>>(dep-1) & 1;
    if(trie[now][bit]) return query(trie[now][bit], val, dep - 1);
    return query(trie[now][bit^1], val, dep - 1) + (1<<(dep-1));
}

ll dfs(int now, int dep)                              //当前到第dep位，将其左右兄弟的连通块合并
{
    if(dep <= 0) return 0;
    if(trie[now][0] && trie[now][1])                          //如果左右两边都有连通块
    {
        ll mx = 2e15;                                                 //mx为合并左右两个连通块的花费
        //if(R[trie[now][0]] - L[trie[now][0]] <= R[trie[now][1]] - L[trie[now][1]])      //启发式搜索
            for(int i = L[trie[now][0]]; i <= R[trie[now][0]]; i++)
                mx = min(mx, query(trie[now][1], a[i], dep - 1));
        // else
            // for(int i = L[trie[now][1]]; i <= R[trie[now][1]]; i++)
            //     mx = min(mx, query(trie[now][0], a[i], dep - 1));
        return dfs(trie[now][0], dep - 1) + dfs(trie[now][1], dep - 1) + mx + (1<<(dep-1));
    }
    if(trie[now][0])  return dfs(trie[now][0], dep - 1);
    return dfs(trie[now][1], dep - 1);
```

```
}

void xor_mst()
{
    sort(a + 1, a + 1 + n);
    memset(L, INF, sizeof(L));
    for(int i = 1; i <= n; i++) Insert(root, i, 30);
    printf("%lld\n",dfs(root, 30));
}

vector<P> G[maxn];
int cnt;

void Dfs(int x, int fa, int val)
{
    a[++cnt] = val;
    for(auto &y: G[x])
    {
        if(y.first == fa) continue;
        Dfs(y.first, x, val ^ y.second);
    }
}

int main()
{
    scanf("%d", &n);
    for(int i = 1; i <= n - 1; i++)
    {
        int x, y, z;
        scanf("%d %d %d", &x, &y, &z);
        G[x].push_back(P(y, z));  G[y].push_back(P(x, z));
    }
    Dfs(0, -1, 0);

    xor_mst();
}
```

边权转点权

题目大意是一棵树给定边权，都是非负数，树上的每条路径可以使得经过的边的边权减 1，问最少需要几次路径？ 其中还有 q 次修改，每次修改一条边权，每次修改完之后输出更新后的答案。

经过分析可以发现，若记 $w(u,v)$ 为 u,v 间的边权，那么就存在 $w(u,v)$ 条路径通过 u,v。可以举几个例子看看，比如 u 连有三个点v_1,v_2,v_3：

①若边权分别为 10，5，2， 可以用 7 条 uv_1 的路径可以与所有 uv_2,uv_3 的路径相连接，那么u至少是 3 条路径的端点；

②若边权分别是5，4，3，那么 3条 uv_1 的路径可以与 3条 uv_2 路径连接， 2条 uv_1 的路径可以与2条 uv_3 路径连接，1条 uv_2 的路径可以与 1条 uv_3 路径连接，u 端点数可以为 0。

②若边权分别是5，4，2，由于是奇数，那么不管怎么连接，u 端点数至少为 1。

通过上面的举例我们可以得到，**若是一个点的边权和为 d， 最大边权为 mx，大于其他边权和 d - mx, 那么这个点的路径端点数为 mx - (d - mx) = 2*mx - d；若最大边权小于等于其他边权，若 d 为偶数，则可以互相连接，端点数为 0， 否则奇数的话端点数为1。**

于是我们需要维护每个点的边权和，这个用一个数组就可以，还需要维护最大边权，这个可以用

multiset 来做到；计算完所有点的端点数之后，除以 2 为我们所需要答案。每次更新我们只需要相应的更改数组和 multiset, 复杂度为 $O(logn)$。

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
const int maxn = 1e5 + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

int x[maxn], y[maxn], val[maxn], n, q, p, w;
ll d[maxn];
multiset<int> s[maxn];

int cal(int x)                //计算每个点的路径端点数
{
    ll mx = *s[x].rbegin();
    if(mx * 2 <= d[x])  return d[x] & 1;
    return 2 * mx - d[x];
}

int main()
{
    scanf("%d %d", &n, &q);
    for(int i = 1; i < n; i++)
    {
        scanf("%d %d %d", &x[i], &y[i], &val[i]);
        d[x[i]] += val[i];  d[y[i]] += val[i];
        s[x[i]].insert(val[i]);  s[y[i]].insert(val[i]);
    }

    ll ans = 0;
    for(int i = 1; i <= n; i++)  ans += cal(i);
    printf("%lld\n", ans / 2);
    while(q--)
    {
        scanf("%d %d", &p, &w);
        ans -= cal(x[p]) + cal(y[p]);
        s[x[p]].erase(s[x[p]].find(val[p]));
 s[y[p]].erase(s[y[p]].find(val[p]));
        s[x[p]].insert(w);  s[y[p]].insert(w);
        d[x[p]] += w - val[p];  d[y[p]] += w - val[p];
        val[p] = w;
        ans += cal(x[p]) + cal(y[p]);
        printf("%lld\n", ans / 2);
    }
}
```

给定一张无向图，有些边已经被标记成 $0,1$，我们需要给未标记的边赋值 0，1，使得无向图中任意一个环中所有边的异或和为0。问有几种方案?

分析

先从简单的入手，给定一个连通图，把里面的边赋值 0，1，使得任意一个环，里面所有边的边权异或值为0，求方案数。

我们可以不直接考虑边如何赋值，可以考虑点。**在一个环中，每个点都会贡献两条边，若是我们定义边权为两个顶点的异或和，那么不管点权是多少，在环中一个边的权值可以拆成两个点权值的异或，一个顶点都会被异或两次，那么边权异或和肯定为0。**

那么任意一种点权取值就对应了一种边权取值，但我们要注意若是两种方案的点权取值完全相反，即 一种方案的 0 在另一种方案中全是 1，1 全是 0，那么对应的边权取值是一样的。所以一个有 k 个顶点的连通图，方案数为 2^{k-1} 种。

那么对于一个无向图来说，只要拆成各个连通图考虑然后相乘就可以了。

但有的边权已经确定了怎么办？若一条边的边权已经确定，那么其中一个顶点值确定，另一个顶点值也相应确定了，即不能任意取值了。我们可以推得**一个边权确定的连通图(顶点都由确定边权的边连接)，只要其中一个顶点确定，其他顶点也会被确定**，若这个图有 k 个顶点，即答案需要除掉 2^{k-1}。这里需要注意的是，可能有一个环的边都已经确定但是异或和不为0，这时候直接方案数就为0。

所以我们边权转点权之后，需要两遍 dfs，分别求出相应的连通图。

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
const int maxn = 1e5 + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

ll qpow(ll x, ll n)
{
    ll t = 1;
    for (; n; n >>= 1, x = x * x % mod)
        if (n & 1)
            t = t * x % mod;
    return t;
}

struct edge
{
    int to, next, w;
}e[maxn*2];

int head[maxn], tol;

void add_edge(int x, int y, int w)
{
    e[++tol].to = y;
    e[tol].w = w;
    e[tol].next = head[x];
    head[x] = tol;
}

int n, m, vis[maxn], cur, mark[maxn], ans;
```

```
bool dfs1(int x)
{
    vis[x] = 1;
    cur++;
    for(int i = head[x]; i; i = e[i].next)
    {
        int to = e[i].to, w = e[i].w;
        if(w == -1)  continue;
        if(vis[to])
        {
            if((mark[x] ^ mark[to]) != w) return false;
        }
        else
        {
            mark[to] = mark[x] ^ w;
            if(!dfs1(to))  return false;
        }
    }
    return true;
}

void dfs2(int x)
{
    vis[x] = 1;
    cur++;
    for(int i = head[x]; i; i = e[i].next)
    {
        int to = e[i].to;
        if(vis[to])  continue;
        dfs2(to);
    }
}

int main()
{
    scanf("%d %d", &n, &m);
    for(int i = 1; i <= m; i++)
    {
        int x, y, z;
        scanf("%d %d %d", &x, &y, &z);
        add_edge(x, y, z);
        add_edge(y, x, z);
    }

    for(int i = 1; i <= n; i++)        //求确定边权的连通图
    {
        if(!dfs1(i))
        {
            printf("%d\n", 0);
            return 0;
        }
        ans -= cur - 1;
        cur = 0;
    }

    fill(vis + 1, vis + 1 + n, 0);
    for(int i = 1; i <= n; i++)         //求连通图
```

```
    {
        dfs2(i);
        ans += cur - 1;
        cur = 0;
    }

    printf("%lld\n", qpow(2, ans));
}
```

换根

给定一棵树，每条边有流量限制，一个结点的流量定义为将该点看为源点，最多能流出多少水(可以从不是自身的叶子结点流出)，问这棵树最大的结点流量是多少？

分析

求出一个结点的流量，我们可以对这个结点用一遍树型dp，对于结点 u 和 其儿子 v， 若 v 是叶子结点，那么 $dp[u]+=w(u,v)$, 否则 $dp[u]+=min(dp[v],w(u,v))$。对一个结点我们可以用 $O(n)$ 时间算出其流量，但是若是用此法算出所有的结点，是$O(n^2)$ 的复杂度，会超时。

这时候就要考虑**换根的思想：先算出固定某一点为根的答案然后考虑把它的儿子换成根会发生什么样的变化，如果这个变化是比较好算的，那么我们就可考虑每个点x为根的答案都根据以它父亲为根的结果去推。**

若我们算出父亲的流量 $f[u]$, 对于 儿子的流量 $f[v]$, 若 v 是叶子结点，那么 $f[u]$ 中肯定有 $w(u,v)$ 流向 v， $f[u]-w(u,v)$ 流向其他，那么从 v 就最多可以流出去 $min(f[u]-w,w)$; 若 v 不是， 那么 $f[u]$ 有 $min(w,dp[v])$ 流向 v， $f[u]-min(w,dp[v])$ 流向其他，那么从 v 就最多可以流出去 $dp[to]+min(w,f[u]-min(w,dp[v]))$。

那么我们需要两遍 dfs， 第一遍算出一个结点的流量，第二遍进行换根。

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
const int maxn = 2e5 + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

struct edge
{
    int to, next, w;
}e[maxn*2];

int head[maxn], num, t, n, deg[maxn], d[maxn], f[maxn];

void add_edge(int x, int y, int w)
{
    e[++num].to = y;
    e[num].w = w;
    e[num].next = head[x];
    head[x] = num;
```

```
}

void dfs1(int x, int fa)              //求出结点1的流量
{
    d[x] = 0;
    for(int i = head[x]; i; i = e[i].next)
    {
        int to = e[i].to, w = e[i].w;
        if(to == fa)  continue;
        dfs1(to, x);
        if(deg[to] == 1)  d[x] += w;
        else d[x] += min(d[to], w);
    }
}

void dfs2(int x, int fa)                    //换根
{
    for(int i = head[x]; i; i = e[i].next)
    {
        int to = e[i].to, w = e[i].w;
        if(to == fa)  continue;
        if(deg[to] == 1)  f[to] = min(f[x]- w, w);
        else f[to] = d[to] + min(f[x] - min(w, d[to]), w);
        dfs2(to, x);
    }
}

int main()
{
    scanf("%d", &t);
    while(t--)
    {
        scanf("%d", &n);
        fill(head + 1, head + 1 + n, 0);
        fill(deg + 1, deg + 1 + n, 0);
        num = 0;

        for(int i = 1; i < n; i++)
        {
            int x, y, z;
            scanf("%d %d %d", &x, &y, &z);
            add_edge(x, y, z);  add_edge(y, x, z);
            deg[x]++;  deg[y]++;
        }

        dfs1(1, 0);
        f[1] = d[1];
        dfs2(1, 0);
        int ans = 0;
        for(int i = 1; i <= n; i++)  ans = max(ans, f[i]);
        printf("%d\n", ans);
    }
}
```

CDQ分治

```
#include <bits/stdc++.h>

using namespace std;
typedef unsigned long long ll;
const int maxn = 1e5 + 10;
const double INF = 1e15;
const double eps = 1e-8;

int n, ans[maxn];
ll k1, k2;

struct node
{
    ll x, y, z;
    int id;
}arr[maxn];

bool cmpx(const node& m1, const node& m2)
{
    return m1.x < m2.x;
}

bool cmpy(const node& m1, const node& m2)
{
    return m1.y < m2.y;
}

struct treearray
{
    int tree[maxn], n;

    int lowerbit(int x)
    {
        return x&(-x);
    }

    int query(int i)                //查询[1 -i]最大值
    {
        int ans = 0;
        for(; i; i -= lowerbit(i))
            ans = max(ans, tree[i]);
        return ans;
    }

    void modify(int i,int k)              //将i处值修改为k， 维护最大值
    {
        for(;i <= n; i += lowerbit(i))
            tree[i] = max(k, tree[i]);
    }

    void Clear(int x)
    {
        for (int i = x; i <= n; i += lowerbit(i))
            tree[i] = 0;
    }
```

```
}t;

ll CoronavirusBeats()
{
    ll k3 = k1, k4 = k2;
    k1 = k4;
    k3 ^= k3 << 23;
    k2 = k3 ^ k4 ^ (k3 >> 17) ^ (k4 >> 26);
    return k2 + k4;
}

vector<ll> v;
void init()
{
    for (int i = 1; i <= n; i++)
    {
        arr[i].x = CoronavirusBeats();
        arr[i].y = CoronavirusBeats();
        arr[i].z = CoronavirusBeats();
        arr[i].id = i;
        v.push_back(arr[i].z);
    }
    sort(v.begin(), v.end());                                   //将z离散化，从1开始
    v.erase(unique(v.begin(), v.end()), v.end());
    for(int i = 1; i <= n; i++)
        arr[i].z = lower_bound(v.begin(), v.end(), arr[i].z) - v.begin() + 1;
    t.n = n;

}

void solve(int L, int R, int mid)
{
    int p1 = L, p2 = mid + 1;
    sort(arr + L, arr + 1 + mid, cmpy);
    sort(arr + 1 + mid, arr + R + 1, cmpy);

    while(p2 <= R)
    {
        while(p1 <= mid && arr[p1].y <= arr[p2].y)
        {
            t.modify(arr[p1].z, ans[arr[p1].id]);
            p1++;
        }
        ans[arr[p2].id] = max(ans[arr[p2].id], t.query(arr[p2].z) + 1);
        p2++;
    }

    for(int i = L; i <= mid; i++)  t.Clear(arr[i].z);
    sort(arr + L, arr + R + 1, cmpx);
}

void CDQ(int L, int R)
{
    if(L >= R) return;
    int mid = (L + R) / 2;
    CDQ(L, mid);
    solve(L, R, mid);
    CDQ(mid + 1, R);
```

```
}

int main()
{
    cin>>n>>k1>>k2;
    init();
    sort(arr + 1, arr + 1 + n, cmpx);

    for (int i = 1; i <= n; i++) ans[i] = 1;
    CDQ(1, n);

    int ret = 0;
    for(int i = 1; i <= n; i++)
        ret = max(ret, ans[i]);
    printf("%d\n", ret);
    for(int i = 1; i<= n; i++)
        printf("%d ", ans[i] - 1);
}
```