

# 算法竞赛个人模板

Cu\_OH\_2

2023 年 8 月 25 日

目录

1通用1

1.1基础框架1

1.2实用代码1

1.3注意事项1

2动态规划1

2.1单调队列优化多重背包1

2.2二进制分组优化多重背包1

2.3动态DP2

3字符串2

3.1KMP算法2

3.2扩展KMP算法2

3.3字典树3

3.4AC自动机3

3.5后缀自动机4

3.6回文自动机4

3.7Manacher算法5

3.8最小表示法5

3.9字符串哈希5

4数学5

4.1快速幂5

4.2矩阵快速幂6

4.3排列奇偶性6

4.4组合数递推6

4.5线性基6

4.6高精度7

4.7连续乘法逆元7

4.8数论分块8

4.9欧拉函数8

4.10线性素数筛8

4.11欧几里得算法 + 扩展欧几里得算法8

4.12中国剩余定理8

4.13哥德巴赫猜想9

5数据结构9

5.1哈希表9

5.2ST表9

5.3并查集9

5.4树状数组10

5.5二维树状数组10

5.6线段树10

5.7历史最值线段树11

5.8动态开点线段树12

5.9可持久化线段树12

5.10李超线段树13

6树论13

6.1LCA13

6.2树的直径14

6.3树哈希14

6.4树链剖分15

6.5树上启发式合并15

6.6点分治16

7图论16

7.12-SAT16

7.2Bellman-Ford 算法17

7.3Dijkstra 算法17

7.4Dinic 算法18

7.5Floyd 算法19

7.6Kosaraju 算法19

7.7Tarjan 算法19

7.8K 短路20

7.9SSP 算法20

7.10原始对偶算法21

7.11Prim 算法22

7.12Kruskal 算法22

7.13Kruskal 重构树22

8计算几何23

8.1平面坐标旋转23

9杂项算法23

9.1普通莫队算法23

9.2带修改莫队算法24

9.3整体二分24

9.4离散化25

9.5快速排序25

9.6枚举集合25

9.7CDQ 分治 + CDQ 分治 = 多维偏序26

9.8CDQ 分治 + 数据结构 = 多维偏序26

10博弈论27

10.1Fibonacci 博弈27

10.2Wythoff 博弈27

10.3Green Hackenbush 博弈27

# 1 通用

## 1.1 基础框架

```
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

void solve()
{
    return;
}

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int T = 1;
    //cin >> T;
    while (T--) solve();
    return 0;
}
```

## 1.2 实用代码

```
//debug 常用宏
#define debug(x) cout << #x << " = " << x << endl

//本地文件读写
freopen("A.in", "r", stdin);
freopen("A.out", "w", stdout);

//builtin 系列位运算
__builtin_ffs(x); //最低位1是第几位 (从1开始, 不存在则0)
__builtin_clz(x)/__builtin_clzll(x); //前导高0的个数
__builtin_ctz(x)/__builtin_ctzll(x); //末尾低0的个数
__builtin_popcount(x); //1的个数
__builtin_parity(x); //1的个数的奇偶性

//最高位 1 的位置 (从0开始, 注意x不能为0)
__lg(x);

//long double 用浮点函数后面加l
sqrtl(x)/fabsl(x)/cosl(x);

// 随机数生成器 (C++11, 返回unsigned_int)
mt19937 mt(time(0));
mt();
shuffle(v.begin(), v.end(), mt);
```

## 1.3 注意事项

```
//1.爆long long了吗?
//2.数组首尾边界初始化了吗?
//3.测试组间数据清空重置了吗?
//4.交互题用endl了吗?
//5.clear()重置数据了吗?
//6.size()参与减法溢出了吗?
//7.for(j)循环写成++i了吗?
```

# 2 动态规划

## 2.1 单调队列优化多重背包

```
/*
* 时间复杂度: O(nm)
* 说明: dp[j]只有可能从dp[j-k*w[i]]转移来
*/
const int N = 100005;
const int M = 40005;

ll n, m; //种数、容积
ll v[N], w[N], k[N]; //价值、体积、数量
ll dp[M]; //使用i容积的最大价值
```

```
struct Node
{
    ll key, id;
};

void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= n; ++i) cin >> v[i] >> w[i] >> k[i];
    for (int i = 1; i <= n; ++i)
    {
        vector<deque<Node>> dq(w[i]);
        auto key = [&](int j) { return dp[j] - j / w[i] * v[i]; }; //
        //dp[j]在比较基准下的指标
        auto join = [&](int j) //dp[j]入队
        {
            auto& q = dq[j % w[i]];
            while (q.size() && key(j) >= q.back().key) q.pop_back();
            q.push_back({ key(j), j });
            return;
        };
        for (int j = m; j >= max(0ll, m - k[i] * w[i]); --j) join(j);
        for (int j = m; j >= w[i]; --j)
        {
            auto& q = dq[j % w[i]];
            while (q.size() && q.front().id >= j) q.pop_front();
            if (j - k[i] * w[i] >= 0) join(j - k[i] * w[i]);
            dp[j] = max(dp[j], q.front().key + j / w[i] * v[i]);
        }
    }
    ll ans = 0;
    for (int i = 0; i <= m; ++i) ans = max(ans, dp[i]);
    cout << ans << '\n';
    return;
}
```

## 2.2 二进制分组优化多重背包

```
/*
* 时间复杂度: O(nmlogk)
* 说明: 二进制分组优化多重背包, 可bitset优化
*/
const int N = 100005;
const int M = 40005;

struct Item
{
    ll v, w; //价值、体积
};

ll n, m; //种数、容积
ll dp[M]; //使用i容积的最大价值

void solve()
{
    cin >> n >> m;
    vector<Item> items;
    ll x, y, z;
    for (int i = 1; i <= n; ++i)
    {
        ll b = 1;
        cin >> x >> y >> z;
        while (z > b)
        {
            z -= b;
            items.push_back({ x * b, y * b });
            b <<= 1;
        }
        items.push_back({ x * z, y * z });
    }
    for (auto e : items)
    {
        for (int i = m; i >= e.w; --i)
        {
            dp[i] = max(dp[i], dp[i - e.w] + e.v);
        }
    }
    ll ans = 0;
    for (int i = 0; i <= m; ++i) ans = max(ans, dp[i]);
    cout << ans << '\n';
    return;
}
```

## 2.3 动态 DP

```

/*****
* 时间复杂度: O(qlogn)
* 说明: 将dp转移方程表示为矩阵乘法, 用线段树维护矩阵, 实现带修改dp.
*****/
//CF1814E
const int N = 200005;
const ll INFL = 0x3f3f3f3f3f3f3f3f;

struct SegTree
{
    struct Node
    {
        int lef, rig;
        array<array<ll, 2>, 2> mat;
    };
    vector<Node> tree;

    inline int ls(int src) { return src * 2; }
    inline int rs(int src) { return src * 2 + 1; }
    inline Node& ln(int src) { return tree[ls(src)]; }
    inline Node& rn(int src) { return tree[rs(src)]; }

    inline void update(int src)
    {
        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                auto v1 = ln(src).mat[i][1] + rn(src).mat[1][j];
                auto v2 = ln(src).mat[i][0] + rn(src).mat[1][j];
                auto v3 = ln(src).mat[i][1] + rn(src).mat[0][j];
                tree[src].mat[i][j] = min({ v1, v2, v3 });
            }
        }
        return;
    }

    inline void calc(int src, ll val)
    {
        tree[src].mat[1][1] = val;
        tree[src].mat[0][0] = 0;
        tree[src].mat[0][1] = tree[src].mat[1][0] = INFL;
        return;
    }

    SegTree(int x) { tree.resize(x * 4 + 1); }

    void build(int src, int lef, int rig, ll arr[])
    {
        tree[src].lef = lef;
        tree[src].rig = rig;
        if (lef == rig)
        {
            calc(src, arr[lef]);
            return;
        }
        int mid = lef + rig >> 1;
        build(ls(src), lef, mid, arr);
        build(rs(src), mid + 1, rig, arr);
        update(src);
        return;
    }

    void modify(int src, int pos, ll val)
    {
        if (tree[src].lef == tree[src].rig)
        {
            calc(src, val);
            return;
        }
        int mid = tree[src].lef + tree[src].rig >> 1;
        if (pos <= mid) modify(ls(src), pos, val);
        else modify(rs(src), pos, val);
        update(src);
        return;
    }

    ll query() { return tree[1].mat[1][1] * 2; }
};

int n, q, k;
ll a[N], x;

void solve()
{
    cin >> n;
    for (int i = 1; i <= n - 1; ++i) cin >> a[i];

```

```

SegTree sgt(n - 1);
sgt.build(1, 1, n - 1, a);
cin >> q;
for (int i = 1; i <= q; ++i)
{
    cin >> k >> x;
    sgt.modify(1, k, x);
    cout << sgt.query() << '\n';
}
return;
}

```

## 3 字符串

### 3.1 KMP 算法

```

/*****
* 时间复杂度: O(n)
* 说明:
* 1. nxt[i] 表示 t[i] (下标从 0 开始) 失配时下一次匹配的位置
* 2. nxt[n] 在匹配中无必要作用, 但构成前缀数组
* 3. 前缀数组 pi[i] = nxt[i+1] + 1, 代表前缀 t[0,i] 的最长前后缀长度
*****/
struct KMP
{
    string t;
    vector<int> nxt;

    KMP() {}
    KMP(const string& str) { init(str); }

    void init(const string& str)
    {
        t = str;
        nxt.resize(t.size() + 1);
        nxt[0] = -1;
        for (int i = 1; i <= t.size(); ++i)
        {
            int now = nxt[i - 1];
            while (now != -1 && t[i - 1] != t[now]) now = nxt[now];
            nxt[i] = now + 1;
        }
        return;
    }

    int first(const string& s)
    {
        int ps = 0, pt = 0;
        while (ps < s.size())
        {
            while (pt != -1 && s[ps] != t[pt]) pt = nxt[pt];
            ps++, pt++;
            if (pt == t.size()) return ps - t.size();
        }
        return -1;
    }

    vector<int> every(const string& s)
    {
        vector<int> v;
        int ps = 0, pt = 0;
        while (ps < s.size())
        {
            while (pt != -1 && s[ps] != t[pt]) pt = nxt[pt];
            ps++, pt++;
            if (pt == t.size())
            {
                v.push_back(ps - t.size());
                pt = nxt[pt];
            }
        }
        return v;
    }
};

```

### 3.2 扩展 KMP 算法

```

/*****
* 时间复杂度: O(n)
* 说明: Z 函数代表后缀与母串的最长公共前缀
*****/

```

```

struct ExKMP
{
    string t;
    vector<int> z;

    ExKMP(const string& str)
    {
        t = str;
        z.resize(t.size());
        z[0] = t.size();
        int l = 0, r = -1;
        for (int i = 1; i < t.size(); ++i)
        {
            if (i <= r && z[i - 1] < r - i + 1) z[i] = z[i - 1];
            else
            {
                z[i] = max(0, r - i + 1);
                while (i + z[i] < t.size() && t[z[i]] == t[i + z[i]]) z[i]++;
            }
            if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
        }
    }

    vector<int> ext(const string& s)
    {
        vector<int> res(s.size());
        int l = 0, r = -1;
        for (int i = 0; i < s.size(); ++i)
        {
            if (i <= r && z[i - 1] < r - i + 1) res[i] = z[i - 1];
            else
            {
                res[i] = max(0, r - i + 1);
                while (i + res[i] < s.size() && res[i] < t.size() && t[res[i]] == s[i + res[i]]) res[i]++;
            }
            if (i + res[i] - 1 > r) l = i, r = i + res[i] - 1;
        }
        return res;
    }
};

```

### 3.3 字典树

```

/*****
* 时间复杂度: O(sigma(n))
* 说明:
* 1.字典树也即前缀树, 每个结点代表一个前缀
* 2.字母表变化只需要修改映射函数F()
* 3.若需要遍历trie树可以用out数组记录出边降低复杂度
*****/
struct Trie
{
    const int ALPSZ = 26;
    vector<vector<int>>> trie;
    vector<int> tag;
    //vector<vector<int>>> out;

    inline int F(char c) { return c - 'a'; }

    Trie() { init(); }

    void init()
    {
        create();
        return;
    }
    int create()
    {
        trie.push_back(vector<int>(ALPSZ));
        tag.push_back(0);
        //out.push_back(vector<int>());
        return trie.size() - 1;
    }
    void insert(const string& t)
    {
        int now = 0;
        for (auto e : t)
        {
            if (!trie[now][F(e)])
            {
                int newNode = create();
                //out[now].push_back(F(e));
                trie[now][F(e)] = newNode;
            }
        }
    }
};

```

```

        now = trie[now][F(e)];
        tag[now]++;
    }
    return;
}
int count(const string& pre)
{
    int now = 0;
    for (auto e : pre)
    {
        now = trie[now][F(e)];
        if (now == 0) return 0;
    }
    return tag[now];
};

```

### 3.4 AC 自动机

```

/*****
* 时间复杂度: O(alpsz*sigma(len(t))+len(s))
* 说明:
* 1.本模板以小写英文字母为字母表举例, 修改字母表可以通过修改F()函数完成。
* 2.Trie图优化: 建立fail指针时, fail指针指向的结点有可能依然失配, 需要多次跳转才能到达匹配结点。可以将所有结点的空指针补全, 置为该结点的跳转终点。此时根据BFS序, 在计算tr[x][i]的fail指针时, fail[x]一定已遍历过, 且tr[fail[x]][i]一定存在, 要么为fail[x]接收i的后继状态, 要么为tr[x][i]的跳转终点。无论哪种情况, fail[tr[x][i]]都可以直接置为tr[fail[x]][i]。
* 3.last优化: 多模式匹配过程中, 对于文本串的每个前缀s', 沿fail指针路径寻找s'后缀的模式串, 途中可能经过无贡献的模式串真前缀结点; last优化使得跳转时跳过真前缀结点直接到达上方第一个模式串结点。last数组可以完全替代fail数组。
* 4.树上差分优化: 统计每种模式串出现次数时, 每匹配到一个模式串都要向上跳转一次, 这个过程相当于区间加一, 可以用更新差分数组代替, 最后再计算前缀和即可。
* 5.注意: 统计出现的模式串种类数时会将标记清空
*****/
struct ACAM
{
    vector<vector<int>>> trie; //trie树指针
    vector<int> tag; //标记数组
    vector<int> fail; //失配函数
    vector<int> last; //跳转路径上一个模式串结点
    vector<int> cnt; //计数器
    const int ALPSZ = 26; //字母表大小
    int ord; //结点个数

    inline int F(char c) { return c - 'a'; }

    ACAM() { init(); }

    void init()
    {
        ord = -1;
        newNode();
    }
    int newNode()
    {
        trie.push_back(vector<int>(ALPSZ));
        tag.push_back(0);
        return ++ord;
    }
    void addPat(const string& t)
    {
        int now = 0;
        for (auto e : t)
        {
            if (!trie[now][F(e)]) trie[now][F(e)] = newNode();
            now = trie[now][F(e)];
        }
        tag[now]++;
        return;
    }
    void buildAM()
    {
        fail.resize(ord + 1);
        last.resize(ord + 1);
        cnt.resize(ord + 1);
        queue<int> q;
        for (int i = 0; i < 26; ++i)
        {
            //第一层结点的fail指针都指向0, 不需要处理
            if (trie[0][i]) q.push(trie[0][i]);
        }
        while (q.size())
    }
};

```

```

{
    int now = q.front();
    q.pop();
    for (int i = 0; i < 26; ++i)
    {
        int son = trie[now][i];
        if (son)
        {
            fail[son] = trie[fail[now]][i];
            if (tag[fail[son]]) last[son] = fail[son];
            else last[son] = last[fail[son]];
            q.push(trie[now][i]);
        }
        else trie[now][i] = trie[fail[now]][i];
    }
}
return;
}
int count(const string& s) //统计出现的模式串种数
{
    int now = 0, ans = 0;
    for (auto e : s)
    {
        now = trie[now][F(e)];
        int p = now;
        while (p)
        {
            ans += tag[p];
            tag[p] = 0;
            p = last[p];
        }
    }
    return ans;
}
};

```

### 3.5 后缀自动机

```

/*****
* 时间复杂度: O(n*ALPSZ)
* 说明: 字符集较大可以将next换成map<char,int>
*****/
struct SAM
{
    struct State
    {
        int maxlen; //结点代表的最长子串长度
        int link; //后缀链接, 连向不在该点中的最长后缀
        vector<int> next;
        State(): maxlen(0), link(-1) { next.resize(26); }
    };
    vector<State> node;
    vector<ll> cnt; //子串出现次数 (endpos集合大小)
    int now; //接收上一个字符到达的结点
    int size; //当前结点个数

    inline int F(char c) { return c - 'a'; }

    SAM(int x)
    {
        node.resize(x * 2 + 5);
        cnt.resize(x * 2 + 5);
        now = 0; //从根节点开始转移
        size = 1; //建立一个代表空串的根节点
    }

    void extend(char c)
    {
        int nid = size++;
        cnt[nid] = 1;
        node[nid].maxlen = node[now].maxlen + 1;
        int p = now;
        while (p != -1 && node[p].next[F(c)] == 0)
        {
            node[p].next[F(c)] = nid;
            p = node[p].link;
        }
        if (p == -1) node[nid].link = 0; //连向根结点
        else
        {
            int ori = node[p].next[F(c)];
            if (node[p].maxlen + 1 == node[ori].maxlen) node[nid].link = ori;
            else
            {
                //将ori结点的一部分拆出来分成新结点split
            }
        }
    }
}

```

```

    int split = size++;
    node[split].maxlen = node[p].maxlen + 1;
    node[split].link = node[ori].link;
    node[split].next = node[ori].next;
    while (p != -1 && node[p].next[F(c)] == ori)
    {
        node[p].next[F(c)] = split;
        p = node[p].link;
    }
    node[ori].link = node[nid].link = split;
}
}
now = nid;
return;
}

void build(const string& s)
{
    for (auto e : s) extend(e);
    return;
}

void DFS(int x, vector<vector<int>>& son)
{
    for (auto e : son[x])
    {
        DFS(e, son);
        cnt[x] += cnt[e]; //link树上父节点endpos为所有子节点endpos之和
    }
    return;
}

void count() //计算endpos大小
{
    //建立link树
    vector<vector<int>> son(size);
    for (int i = 1; i < size; ++i) son[node[i].link].push_back(i);

    //在link树上dfs
    DFS(0, son);
    return;
}

ll substr() //本质不同子串个数
{
    ll res = 0;
    for (int i = 1; i < size; ++i)
    {
        res += node[i].maxlen - node[node[i].link].maxlen;
    }
    return res;
}
};

```

### 3.6 回文自动机

```

/*****
* 时间复杂度: O(n)
* 说明:
* 1. 每个结点代表一个本质不同回文串。link链: 多字串->单字符->偶根->奇根。
* 2. 每个本质不同回文子串次数: 最后由母串向子串传递。
    每个前缀的后缀回文子串个数: 新建时由最长回文后缀向新串传递。
*****/
struct PAM
{
    struct State
    {
        int len; //长度
        int link; //最长回文后缀结点
        vector<int> next; //两边加上某字符时对应的结点
        State() { next.resize(26); }
        State(int x, int y): len(x), link(y) { next.resize(26); }
    };
    vector<State> node;
    vector<ll> cnt; //本质不同回文串出现次数
    int now; //接收上一个字符到达的结点
    int size; //当前结点个数

    inline int F(char c) { return c - 'a'; }

    PAM(int x)
    {
        node.resize(x + 3);
        node[0] = State(-1, 0); //奇根, link无意义
    }
}

```

```

node[1] = State(0, 0); //偶根, link指向奇根
cnt.resize(x + 3);
now = 0; //第一个字符由奇根转移
size = 2;
}

void build(const string& s)
{
    auto find = [&](int x, int p) //寻找x后缀中左方为s[p]的最长回文子串
    {
        while (p - node[x].len - 1 < 0 || s[p] != s[p - node[x].len - 1]) x = node[x].link;
        return x;
    };
    for (int i = 0; i < s.size(); ++i)
    {
        now = find(now, i);
        if (!node[now].next[F(s[i])]) //对应结点不存在则需要新建
        {
            int nid = size++;
            node[nid].len = node[now].len + 2; //新建状态结点
            node[nid].link = 1; //若now=0, 对应结点为单字符, 指向偶根
            if (now) node[nid].link = node[find(node[now].link, i)].next[F(s[i])]; //否则指向再前一个结点的扩展
            node[now].next[F(s[i])] = nid;
        }
        now = node[now].next[F(s[i])];
        cnt[now]++;
    }
    for (int i = size - 1; i >= 2; --i) cnt[node[i].link] += cnt[i]; //数量由母串向子串传递
    return;
}
};

```

## 3.7 Manacher 算法

```

/*****
* 时间复杂度: O(n)
* 说明: 用n+1个分隔符将字符串分隔可以将奇偶回文子串过程统一处理
*****/
struct Manacher
{
    vector<int> odd, even; //以[i]或[i,i+1]为中心的最长回文串半径
    void work(const string& s)
    {
        odd.resize(s.size());
        even.resize(s.size() - 1);
        int lef = 0, rig = -1, r;
        for (int i = 0; i < s.size(); ++i)
        {
            if (i > rig) r = 1;
            else r = min(odd[lef + rig - i], rig - i) + 1; //利用对称位置答案
            while (i - r >= 0 && i + r < s.size() && s[i - r] == s[i + r]) r++; //暴力扩展
            odd[i] = --r; //记录答案
            if (i + r > rig) lef = i - r, rig = i + r; //扩展lef,rig范围
        }
        lef = 0, rig = -1;
        for (int i = 0; i + 1 < s.size(); ++i)
        {
            if (i + 1 > rig) r = 1;
            else r = min(even[lef + rig - i - 1], rig - i) + 1;
            while (i + 1 - r >= 0 && i + r < s.size() && s[i + 1 - r] == s[i + r]) r++;
            even[i] = --r;
            if (i + r > rig) lef = i + 1 - r, rig = i + r;
        }
    }
    return;
}
};

```

## 3.8 最小表示法

```

/*****
* 时间复杂度: O(n)
* 说明: 求循环rotate得到的n种表示中字典序最小的一种
*****/
const int N = 300005;

```

```

int n, a[N];

void solve()
{
    cin >> n;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    auto norm = [](int x) { return (x - 1) % n + 1; };
    int p1 = 1, p2 = 2, len = 1;
    while (p1 <= n && p2 <= n & len <= n)
    {
        if (a[norm(p1 + len - 1)] == a[norm(p2 + len - 1)]) len++;
        else if (a[norm(p1 + len - 1)] < a[norm(p2 + len - 1)]) p2 += len, len = 1;
        else p1 += len, len = 1;
        if (p1 == p2) p1++;
    }
    int ans = min(p1, p2);
    return;
}

```

## 3.9 字符串哈希

```

/*****
* 时间复杂度: O(n)
* 说明: 字符串传入前必须处理为下标从1开始的模式!
*****/
const int M1 = 998244389;
const int M2 = 998244391;
const int B1 = 31;
const int B2 = 29;
const int N = 1000005;

struct Base
{
    array<ll, N> pow{};
    Base(int base, int mod)
    {
        pow[0] = 1;
        for (int i = 1; i <= N - 1; ++i)
        {
            pow[i] = pow[i - 1] * base % mod;
        }
    }
    const ll operator[](int idx) const { return pow[idx]; }
} p1(B1, M1), p2(B2, M2);

struct Hash
{
    vector<ll> hash1, hash2;
    void build(const string& s)
    {
        int n = s.size() - 1;
        hash1.resize(n + 1);
        hash2.resize(n + 1);
        for (int i = 1; i <= n; ++i)
        {
            hash1[i] = (hash1[i - 1] * B1 % M1 + s[i] - 'a' + 1) % M1;
            hash2[i] = (hash2[i - 1] * B2 % M2 + s[i] - 'a' + 1) % M2;
        }
        return;
    }
    ll merge(ll x, ll y) { return x << 31 | y; }
    ll calc(int lef, int rig)
    {
        ll res1 = (hash1[rig] - hash1[lef - 1] * p1[rig - lef + 1] % M1 + M1) % M1;
        ll res2 = (hash2[rig] - hash2[lef - 1] * p2[rig - lef + 1] % M2 + M2) % M2;
        return merge(res1, res2);
    }
};

```

## 4 数学

### 4.1 快速幂

```

/*****
* 时间复杂度: O(sqrt(n))
* 说明:
* 1. 特殊情况下需要对res和a的初值进行取模, 注意p不可取模
* 2. 利用费马小定理求乘法逆元时注意仅当mod为质数时有效

```

```

* 3. 若p较大且mod为质数可以将p对mod-1取模
*****/
ll qpow(ll a, ll p, ll mod)
{
    ll res = 1;
    while (p)
    {
        if (p & 1) res = res * a % mod;
        a = a * a % mod;
        p >>= 1;
    }
    return res;
}

ll inv(ll a, ll mod)
{
    return qpow(a, mod - 2, mod);
}

```

## 4.2 矩阵快速幂

```

/*****
* 时间复杂度:  $O(n^3 \log p)$ 
* 说明: 已知递推式可以表示为矩阵乘法, 快速求数列第i项
*****/
const int MOD = 1e9 + 7;

struct Square
{
    int n;
    vector<vector<ll>> a;
    Square(int n): n(n) { a.resize(n, vector<ll>(n)); }
    void unit()
    {
        for (int i = 0; i < n; ++i)
        {
            a[i][i] = 1;
        }
        return;
    }
};

Square mult(const Square& lhs, const Square& rhs)
{
    assert(lhs.n == rhs.n);
    int n = lhs.n;
    Square res(n);
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            for (int k = 0; k < n; ++k)
            {
                res.a[i][j] += lhs.a[i][k] * rhs.a[k][j] % MOD;
                res.a[i][j] %= MOD;
            }
        }
    }
    return res;
}

Square qpow(Square a, ll p)
{
    int n = a.n;
    Square res(n);
    res.unit();
    while (p)
    {
        if (p & 1) res = mult(res, a);
        a = mult(a, a);
        p >>= 1;
    }
    return res;
}

```

## 4.3 排列奇偶性

```

/*****
* 时间复杂度:  $O(n)$ 
* 说明:
* 1. 顺序排列为偶排列
* 2. 交换任意两个数, 排列奇偶性改变
* 3. 排列奇偶性等于逆序对数奇偶性

```

```

* 4. 求环的个数可以 $O(n)$ 求得排列奇偶性
*****/
void solve()
{
    cin >> n;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    bool inv = n & 1;
    vector<bool> vis(n + 1);
    for (int i = 1; i <= n; ++i)
    {
        if (vis[i]) continue;
        int cur = i;
        while (!vis[cur])
        {
            vis[cur] = 1;
            cur = a[cur];
        }
        inv ^= 1;
    }
    return;
}

```

## 4.4 组合数递推

```

/*****
* 时间复杂度:  $O(nm)$ 
* 说明: 递推预处理组合数
*****/
struct Comb
{
    vector<vector<ll>> c;
    Comb(int x, int y, int mod)
    {
        c.resize(x + 1, vector<ll>(y + 1));
        for (int i = 0; i <= x; ++i) c[i][0] = 1;
        for (int i = 1; i <= x; ++i)
        {
            for (int j = 1; j <= i; ++j) c[i][j] = (c[i - 1][j - 1] + c[i - 1][j]) % mod;
        }
    }
    ll val(int x, int y)
    {
        if (x < 0 || y < 0) return 0;
        else return c[x][y];
    }
};

```

## 4.5 线性基

```

/*****
* 时间复杂度: 插入 $O(b)$ /求最大异或和 $O(b)$ 
* 说明:
* 1. 可以求子序列最大异或和
* 2. v中非零元素表示一组线性基
* 3. 线性基大小表征线性空间维数
* 4. 线性基中没有异或和为0的子集
* 5. 线性基中各数二进制最高位不同
*****/
const int N = 55;
const int B = 50;

template<int bit>
struct LinearBasis
{
    vector<ll> v;
    LinearBasis() { v.resize(bit); }
    void insert(ll x)
    {
        for (int i = bit - 1; i >= 0; --i)
        {
            if (x >> i & 1ll)
            {
                if (v[i]) x ^= v[i];
                else
                {
                    v[i] = x;
                    break;
                }
            }
        }
    }
    return;
}

```



```

11 qmax()
{
    11 res = 0;
    for (int i = bit - 1; i >= 0; --i)
    {
        if ((res ^ v[i]) > res) res ^= v[i];
    }
    return res;
}
void merge(const LinearBasis<bit>& b)
{
    for (auto e : b.v) insert(e);
    return;
}
};

```

## 4.6 高精度

```

/*****
* 时间复杂度: O(n)/O(n^2)
* 说明: 待完善, 注意复杂度
*****/
const int N = 5005;

struct Large
{
    array<ll, N> ar{};
    int len = 0;
    Large() {}
    Large(ll x)
    {
        int p = 0;
        while (x)
        {
            ar[p++] = x % 10;
            x /= 10;
        }
        updateLen();
    }
    Large(const string& s)
    {
        for (int i = 0; i < s.size(); ++i)
        {
            ar[i] = s[s.size() - 1 - i] - '0';
        }
        updateLen();
    }

    void updateLen()
    {
        len = ar.size();
        for (int i = ar.size() - 1; i >= 0; --i)
        {
            if (ar[i]) break;
            len = i;
        }
        return;
    }

    Large& operator=(const Large& rhs)
    {
        for (int i = 0; i < ar.size(); ++i) ar[i] = rhs.ar[i];
        updateLen();
        return *this;
    }

    Large operator+(const Large& rhs) const
    {
        Large res;
        for (int i = 0; i < ar.size(); ++i) res.ar[i] = ar[i] + rhs.ar[i];
        for (int i = 0; i < ar.size() - 1; ++i)
        {
            res.ar[i + 1] += res.ar[i] / 10;
            res.ar[i] %= 10;
        }
        res.updateLen();
        return res;
    }

    Large& operator+=(const Large& rhs)
    {
        for (int i = 0; i < ar.size(); ++i) ar[i] += rhs.ar[i];
        for (int i = 0; i < ar.size() - 1; ++i)
        {
            ar[i + 1] += ar[i] / 10;

```

```

        ar[i] %= 10;
    }
    updateLen();
    return *this;
}

Large operator-(const Large& rhs) const
{
    Large res;
    for (int i = 0; i < ar.size(); ++i) res.ar[i] = ar[i] - rhs.ar[i];
    for (int i = 0; i < ar.size() - 1; ++i)
    {
        if (res.ar[i] < 0)
        {
            res.ar[i] += 10;
            res.ar[i + 1]--;
        }
    }
    res.updateLen();
    return res;
}

Large operator*(const ll rhs) const
{
    Large res;
    for (int i = 0; i < ar.size(); ++i) res.ar[i] = ar[i] * rhs;
    for (int i = 0; i < ar.size() - 1; ++i)
    {
        if (res.ar[i] > 9)
        {
            res.ar[i + 1] += res.ar[i] / 10;
            res.ar[i] %= 10;
        }
    }
    res.updateLen();
    return res;
}

Large& operator*=(const ll rhs)
{
    for (int i = 0; i < ar.size(); ++i) ar[i] *= rhs;
    for (int i = 0; i < ar.size() - 1; ++i)
    {
        if (ar[i] > 9)
        {
            ar[i + 1] += ar[i] / 10;
            ar[i] %= 10;
        }
    }
    updateLen();
    return *this;
}

Large operator*(const Large& rhs) const
{
    Large res;
    Large dup = *this;
    for (int i = 0; i < rhs.len; ++i)
    {
        res += dup * rhs.ar[i];
        dup *= 10;
    }
    return res;
}

Large& operator*=(const Large& rhs)
{
    *this = *this * rhs;
    return *this;
}
};

ostream& operator<<(ostream& out, const Large& large)
{
    if (large.len == 0)
    {
        cout << '0';
        return out;
    }
    for (int i = large.len - 1; i >= 0; --i) cout << large.ar[i];
    return out;
}

```

## 4.7 连续乘法逆元

```

/*****
* 时间复杂度: O(n)
* 说明:
* 1. 线性时间复杂度求[1,n]的逆元
* 2. 注意mod必须与1-n所有数互质, 否则不存在逆元
*****/
struct ConInv
{
    vector<ll> inv;
    ConInv(int sz, ll mod)
    {
        inv.resize(sz);
        inv[1] = 1;
        for (int i = 2; i <= sz; ++i)
        {
            inv[i] = (mod - mod / i) * inv[mod % i] % mod;
        }
    }
};

```

## 4.8 数论分块

```

/*****
* 时间复杂度: O(sqrt(n))
* 说明: k%i=k-k/i*i => sigma(k%i)=k*n-sigma(k/i*i)
*****/
ll n, k;

int main()
{
    //求sigma[i=1,n](k%i)
    ll ans = 0;
    cin >> n >> k;
    for (ll lef = 1, rig; lef <= n; lef = rig + 1) //分块
    {
        if (k >= lef)
        {
            rig = min(n, k / (k / lef));
        }
        else //该区间大于k (余数都为k)
        {
            rig = n;
        }
        ans += k * (rig - lef + 1) - (k / lef) * (lef + rig) * (rig - lef + 1) / 2;
    }
    cout << ans << '\n';
    return 0;
}

```

## 4.9 欧拉函数

```

/*****
* 时间复杂度: O(sqrt(n))
* 说明:
* 1. 欧拉函数的性质:
* I. phi(x)=x*Π((p[i]-1)/p[i]), p[i]为x的第i个质因数;
* II. 若x为质数:
* i%x==0 => phi(i*x)=x*phi(i)
* i%x!=0 => phi(i*x)=(x-1)*phi(i)
* 2. 若求[1,r]内的欧拉函数, 可以先筛出sqrt(r)以内的质数, 用这些质数
* 贡献范围内的数, 再特判所有数sqrt(r)以上的质因子即可, 类似素数筛。
*****/
//求n的欧拉函数, 类似于质因数分解
int phi(int n)
{
    int res = n;
    for (int i = 2; i * i <= n; i++)
    {
        if (n % i == 0) res = res / i * (i - 1);
        while (n % i == 0) n /= i;
    }
    if (n > 1) res = res / n * (n - 1);
    return res;
}

```

## 4.10 线性素数筛

```

/*****
* 时间复杂度: O(n)
* 说明:
* 1. 筛出x以内所有质数
* 2. sieve[i]表征i是否为合数
* 3. prime中为升序排列的质数
*****/
struct PrimeSieve
{
    vector<int> sieve;
    vector<ll> prime;

    void build(int x)
    {
        sieve.resize(x+1);
        sieve[1] = 1;
        for (int i = 2; i <= x; ++i)
        {
            if (sieve[i] == 0) prime.push_back(i);
            for (auto e : prime)
            {
                if (e > x / i) break;
                sieve[i * e] = 1;
                if (i % e == 0) break;
            }
        }
        return;
    }
};

```

## 4.11 欧几里得算法 + 扩展欧几里得算法

```

/*****
* 时间复杂度: O(logn)
* 说明:
* 1. 欧几里得算法: 求最大公因数
* 2. 扩展欧几里得算法: 求解ax+by=gcd(a,b)
* 3. 由扩展欧几里得算法求出一组解x1,y1后, 可得解集:
*   x=x1+b/gcd(a,b)*k
*   y=y1-a/gcd(a,b)*k;
*   解出的x1不保证是最小正整数, 需要手动调整。
* 4. ax+by=1有解=>1是gcd(a,b)倍数=>gcd(a,b)=1
* 5. 扩展欧几里得还可以解同余方程求乘法逆元
*****/
ll gcd(ll a, ll b)
{
    return b == 0 ? a : gcd(b, a % b);
}

ll exgcd(ll a, ll b, ll& x, ll& y)
{
    if (b == 0) { x = 1, y = 0; return a; }
    ll d = exgcd(b, a % b, x, y);
    ll newx = y, newy = x - a / b * y;
    x = newx, y = newy;
    return d;
}

ll inv(ll a, ll mod)
{
    ll x, y;
    exgcd(a, mod, x, y);
    return x;
}

ll a, b, x, y, g;

void solve()
{
    cin >> a >> b;
    g = exgcd(a, b, x, y);
    auto M = [](ll x, ll m) { return (x % m + m) % m; };
    cout << M(x, b / g) << '\n';
    return;
}

```

## 4.12 中国剩余定理

```

/*****
* 时间复杂度: O(nlogn)
* 说明:
* 1. 中国剩余定理, 解线性同余方程组, 总有唯一解

```

```

* 2.爆longlong时可能需要快速乘(模数过大也可能爆精度)
*****/
struct CRT
{
    vector<pair<ll, ll>> f;
    inline ll norm(ll x, ll mod) { return (x % mod + mod) % mod; }
    ll qmul(ll a, ll b, ll mod)
    {
        //a = norm(a, mod);
        //b = norm(b, mod);
        ll res = a * b - (ll)((1d)a / mod * b + 1e-8) * mod;
        return norm(res, mod);
    }
    ll exgcd(ll a, ll b, ll& x, ll& y)
    {
        if (b == 0)
        {
            x = 1, y = 0;
            return a;
        }
        ll d = exgcd(b, a % b, x, y);
        ll newx = y, newy = x - a / b * y;
        x = newx, y = newy;
        return d;
    }
    ll inv(ll a, ll mod)
    {
        ll x, y;
        exgcd(a, mod, x, y);
        return norm(x, mod);
    }
    void insert(int r, int m)
    {
        f.push_back({ r, m });
        return;
    }
    ll solve()
    {
        ll mul = 1, ans = 0;
        for (auto e : f) mul *= e.second;
        for (auto e : f)
        {
            ll m = mul / e.second;
            ll c = m * inv(m, e.second);
            ans += c * e.first;
        }
        return norm(ans, mul);
    }
};

```

### 4.13 哥德巴赫猜想

```

// 1. >=6 的整数可以写成三个质数之和
// 2. >=4 的偶数可以写成两个质数之和
// 3. >=7 的奇数可以写成三个奇质数之和

```

## 5 数据结构

### 5.1 哈希表

```

/*****
* 时间复杂度: O(1)
* 说明:
* 1. 自定义随机化哈希函数, 降低碰撞概率
* 2. unordered_map采用开链法, gp_hash_table采用查探法
*****/
#include<bits/stdc++.h>
#include<unordered_map>

#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/hash_policy.hpp>

using namespace std;
using ll = long long;

using namespace __gnu_pbds;

struct CustomHash
{
    static uint64_t splitmix64(uint64_t x)
    {

```

```

        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const
    {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::
            now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};

unordered_map<ll, ll, CustomHash> mp;
gp_hash_table<ll, ll, CustomHash> ht;

```

### 5.2 ST 表

```

/*****
* 时间复杂度: 建表O(nlogn)/查询O(1)
* 说明: 可重复贡献问题[f(r,r)=r]的静态区间查询, 一般是最值/gcd
*****/
struct ST
{
    int sz;
    vector<vector<ll>> st;

    ST(int x) { init(x); }
    void init(int x)
    {
        sz = x;
        st.resize(sz + 1, vector<ll>(32));
    }
    void build(ll arr[])
    {
        for (int i = 1; i <= sz; ++i) st[i][0] = arr[i];
        int lg = log2(sz);
        for (int i = 1; i <= lg; ++i)
        {
            for (int j = 1; j <= sz; ++j)
            {
                st[j][i] = st[j][i - 1];
                if (j + (1 << (i - 1)) <= sz)
                {
                    st[j][i] = max(st[j][i], st[j + (1 << (i - 1))][i - 1]);
                }
            }
        }
    }
    ll query(int lef, int rig)
    {
        int len = int(log2(rig - lef + 1));
        return max(st[lef][len], st[rig - (1 << len) + 1][len]);
    }
};

```

### 5.3 并查集

```

/*****
* 时间复杂度: 查找近似O(1)/合并近似O(1)
* 说明: 路径压缩/启发式合并保证时间复杂度
*****/
struct DSU
{
    vector<int> f;
    vector<int> v; //集合大小
    DSU() {}
    DSU(int x) { init(x); }
    void init(int x)
    {
        f.resize(x + 1);
        v.resize(x + 1);
        for (int i = 1; i <= x; ++i) f[i] = i;
        for (int i = 1; i <= x; ++i) v[i] = 1;
        return;
    }
    int find(int id) { return f[id] == id ? id : f[id] = find(f[id]); }
    void merge(int x, int y)
    {
        int fx = find(x), fy = find(y);

```

```

    if (fx == fy) return;
    if (v[fx] > v[fy]) swap(fx, fy);
    f[fx] = fy;
    v[fy] += v[fx];
    return;
}
};

```

## 5.4 树状数组

```

/*****
* 时间复杂度：建立O(n)/修改O(logn)/查询O(logn)
* 说明：
* 1. 动态维护满足区间减法的性质，一般是求和
* 2. 单点修改，区间查询
* 3. 时间戳优化可以替代暴力清空
* 4. 将加法换成取最大值就可以维护不可逆前缀最值
*****/
struct Fenwick
{
    int sz;
    vector<ll> tree;
    //vector<int> tag;
    //int now;

    inline int lowbit(int x) { return x & -x; }

    Fenwick() {}
    Fenwick(int x) { init(x); }
    void init(int x)
    {
        sz = x;
        tree.resize(sz + 1);
        //tag.resize(sz + 1);
        //now = 0;
    }
    /*
    void clear()
    {
        now++;
        return;
    }
    */
    void add(int dst, ll v)
    {
        while (dst <= sz)
        {
            //if (tag[dst] != now) tree[dst] = 0;
            tree[dst] += v;
            //tag[dst] = now;
            dst += lowbit(dst);
        }
        return;
    }
    ll pre(int dst)
    {
        ll res = 0;
        while (dst)
        {
            /*
            if (tag[dst] == now) res += tree[dst];
            dst -= lowbit(dst);
            */
            res += tree[dst];
            dst -= lowbit(dst);
        }
        return res;
    }
    inline ll rsum(int lef, int rig) { return pre(rig) - pre(lef - 1); }
};
void build(ll arr[])
{
    for (int i = 1; i <= sz; ++i)
    {
        tree[i] += arr[i];
        int j = i + lowbit(i);
        if (j <= sz) tree[j] += tree[i];
    }
    return;
}
};

```

## 5.5 二维树状数组

```

/*****
* 时间复杂度：修改O(log^2n)/查询O(log^2n)
* 说明：普通树状数组的二维版，维护矩阵
*****/
struct Fenwick2
{
    int sz;
    vector<vector<ll>> tree;

    inline int lowbit(int x) { return x & -x; }

    Fenwick2() {}
    Fenwick2(int x) { init(x); }

    void init(int x)
    {
        sz = x;
        tree.resize(sz + 1, vector<ll>(sz + 1));
        return;
    }

    void add(int x, int y, ll val)
    {
        for (int i = x; i <= sz; i += lowbit(i))
        {
            for (int j = y; j <= sz; j += lowbit(j))
            {
                tree[i][j] += val;
            }
        }
        return;
    }

    ll pre(int x, int y)
    {
        ll res = 0;
        for (int i = x; i >= 1; i -= lowbit(i))
        {
            for (int j = y; j >= 1; j -= lowbit(j))
            {
                res += tree[i][j];
            }
        }
        return res;
    }

    ll sum(int x1, int y1, int x2, int y2)
    {
        return pre(x2, y2) - pre(x1 - 1, y2) - pre(x2, y1 - 1) + pre(
            x1 - 1, y1 - 1);
    }
};

```

## 5.6 线段树

```

/*****
* 时间复杂度：建立O(n)/询问O(logn)/修改O(logn)
* 说明：
* 1. 维护区间性质，要求性质能由子区间性质得到。
* 2. 区间修改，区间查询。若仅单点修改则不需要标记。
* 3. 使线段树维护不同性质只需要改变上方函数和默认值。
* 4. modify也要push。
*****/
struct SegTree
{
    struct Node
    {
        int lef, rig;
        ll val, tag;
    };
    vector<Node> tree;

    const ll VDEF = 0;
    const ll TDEF = 0;

    inline void update(int src) //由子节点及其标记更新父节点
    {
        ll lw = tree[src << 1].rig - tree[src << 1].lef + 1;
        ll rw = tree[src << 1 | 1].rig - tree[src << 1 | 1].lef + 1;
        ll lv = tree[src << 1].val + tree[src << 1].tag * lw;
        ll rv = tree[src << 1 | 1].val + tree[src << 1 | 1].tag * rw;
        tree[src].val = merge(lv, rv);
        return;
    }

    inline void push(int src) //下传标记并消耗
    {

```

```

    if (tree[src].lef < tree[src].rig)
    {
        tree[src << 1].tag += tree[src].tag;
        tree[src << 1 | 1].tag += tree[src].tag;
    }
    ll wid = tree[src].rig - tree[src].lef + 1;
    tree[src].val += tree[src].tag * wid;
    tree[src].tag = TDEF;
    return;
}
inline void mark(int src, ll val) //更新标记
{
    tree[src].tag += val;
    return;
}
inline ll merge(ll x, ll y) //合并两个查询结果
{
    return x + y;
}

SegTree() {}
SegTree(int x) { init(x); }
void init(int x) { tree.resize(x * 4 + 1); }

void build(int src, int lef, int rig, ll arr[])
{
    tree[src] = { lef, rig, VDEF, TDEF };
    if (lef == rig) tree[src].val = arr[lef];
    else
    {
        int mid = lef + (rig - lef) / 2;
        build(src << 1, lef, mid, arr);
        build(src << 1 | 1, mid + 1, rig, arr);
        update(src);
    }
    return;
}
void build(int src, int lef, int rig)
{
    tree[src] = { lef, rig, VDEF, TDEF };
    if (lef == rig) return;
    int mid = lef + (rig - lef) / 2;
    build(src << 1, lef, mid);
    build(src << 1 | 1, mid + 1, rig);
    update(src);
    return;
}
void assign(int src, int pos, ll val)
{
    push(src);
    if (tree[src].lef == tree[src].rig)
    {
        tree[src].val = val;
        return;
    }
    if (pos <= tree[src << 1].rig) assign(src << 1, pos, val);
    else assign(src << 1 | 1, pos, val);
    update(src);
    return;
}
void modify(int src, int lef, int rig, ll val)
{
    if (lef <= tree[src].lef && tree[src].rig <= rig)
    {
        mark(src, val);
        return;
    }
    push(src);
    if (lef <= tree[src << 1].rig) modify(src << 1, lef, rig, val);
    if (rig >= tree[src << 1 | 1].lef) modify(src << 1 | 1, lef, rig, val);
    update(src);
    return;
}
ll query(int src, int lef, int rig)
{
    push(src);
    if (lef <= tree[src].lef && tree[src].rig <= rig) return tree[src].val;
    ll res = VDEF;
    if (lef <= tree[src << 1].rig) res = merge(res, query(src << 1, lef, rig));
    if (rig >= tree[src << 1 | 1].lef) res = merge(res, query(src << 1 | 1, lef, rig));
    return res;
}
};

```

## 5.7 历史最值线段树

```

/*****
* 时间复杂度：询问O(logn)/修改O(logn)
* 说明：
* 1.维护区间历史最值，支持区间加减
* 2.上方标记一定新于下方标记，因此下传可以整体施加
*****/
struct SegTree
{
    struct Node
    {
        int lef, rig;
        ll mval; //历史最值
        ll tag, mtag; //当前修改标签、tag生命周期内最值
    };
    vector<Node> tree;
    inline ll merge(ll x, ll y) { return min(x, y); } //最大还是最小
    inline void affect(ll& x, ll y) { x = merge(x, y); } //取最值
    inline void update(int src) //由于节点及其标记更新父节点
    {
        ll lv = tree[src << 1].mval + merge(tree[src << 1].mtag, 0);
        ll rv = tree[src << 1 | 1].mval + merge(tree[src << 1 | 1].mtag, 0);
        tree[src].mval = merge(lv, rv);
        return;
    }
    inline void push(int src) //下传标记并消耗
    {
        if (tree[src].lef < tree[src].rig)
        {
            affect(tree[src << 1].mtag, tree[src << 1].tag + tree[src << 1].mtag);
            affect(tree[src << 1 | 1].mtag, tree[src << 1 | 1].tag + tree[src << 1 | 1].mtag);
            tree[src << 1].tag += tree[src].tag;
            tree[src << 1 | 1].tag += tree[src].tag;
        }
        tree[src].mval += merge(tree[src].mtag, 0);
        tree[src].mtag = tree[src].tag = 0;
        return;
    }
    inline void mark(int src, ll val) //更新标记
    {
        tree[src].tag += val;
        affect(tree[src].mtag, tree[src].tag);
        return;
    }
}

SegTree() {}
SegTree(int x) { init(x); }
void init(int x) { tree.resize(x * 4 + 1); }

void build(int src, int lef, int rig)
{
    tree[src] = { lef, rig, 0, 0, 0 };
    if (lef == rig) return;
    int mid = lef + (rig - lef) / 2;
    build(src << 1, lef, mid);
    build(src << 1 | 1, mid + 1, rig);
    update(src);
    return;
}
void modify(int src, int lef, int rig, ll val)
{
    if (lef <= tree[src].lef && tree[src].rig <= rig)
    {
        mark(src, val);
        return;
    }
    push(src);
    if (lef <= tree[src << 1].rig) modify(src << 1, lef, rig, val);
    if (rig >= tree[src << 1 | 1].lef) modify(src << 1 | 1, lef, rig, val);
    update(src);
    return;
}
ll query(int src, int lef, int rig)
{
    push(src);
    if (lef <= tree[src].lef && tree[src].rig <= rig) return tree[src].mval;
    ll res = 0;
    if (lef <= tree[src << 1].rig) res = merge(res, query(src << 1, lef, rig));
    if (rig >= tree[src << 1 | 1].lef) res = merge(res, query(src << 1 | 1, lef, rig));
    return res;
}

```

```
};
```

## 5.8 动态开点线段树

```

/*****
* 时间复杂度：询问 $O(\log n)$ /修改 $O(\log n)$ 
* 说明：注意空间大小
*****/
struct SegTree
{
    struct Node
    {
        int ls = 0, rs = 0;
        ll val = 0, tag = 0;
    };
    vector<Node> tree;
    int ord;
    SegTree(int x)
    {
        tree.resize(x * 64 + 1);
        ord = 1;
    }
    void push(int src, int lef, int rig)
    {
        if (lef < rig)
        {
            if (!tree[src].ls) tree[src].ls = ++ord;
            if (!tree[src].rs) tree[src].rs = ++ord;
            tree[tree[src].ls].tag += tree[src].tag;
            tree[tree[src].rs].tag += tree[src].tag;
        }
        tree[src].val += tree[src].tag * (rig - lef + 1);
        tree[src].tag = 0;
        return;
    }
    void modify(int src, int lef, int rig, int l, int r, ll val)
    {
        if (lef >= l && rig <= r)
        {
            tree[src].tag += val;
            return;
        }
        int mid = lef + (rig - lef) / 2;
        if (l <= mid)
        {
            if (!tree[src].ls) tree[src].ls = ++ord;
            modify(tree[src].ls, lef, mid, l, r, val);
        }
        if (r >= mid + 1)
        {
            if (!tree[src].rs) tree[src].rs = ++ord;
            modify(tree[src].rs, mid + 1, rig, l, r, val);
        }
        tree[src].val += (min(rig, r) - max(lef, l) + 1) * val;
        return;
    }
    ll query(int src, int lef, int rig, int l, int r)
    {
        push(src, lef, rig);
        if (lef >= l && rig <= r) return tree[src].val;
        ll res = 0;
        int mid = lef + (rig - lef) / 2;
        if (l <= mid)
        {
            if (!tree[src].ls) tree[src].ls = ++ord;
            res += query(tree[src].ls, lef, mid, l, r);
        }
        if (r >= mid + 1)
        {
            if (!tree[src].rs) tree[src].rs = ++ord;
            res += query(tree[src].rs, mid + 1, rig, l, r);
        }
        return res;
    }
};

```

## 5.9 可持久化线段树

```

/*****
* 时间复杂度：所有操作 $O(\log(\text{seglen}))$ 
* 说明：
* 1.建空根：可以不靠离散化维护大区间，但要谨慎考虑空间复杂度。

```

```

* 2.主席树维护区间值域上性质：用可持久化权值线段树维护值域，将序列元素逐
* 个插入，由前缀和性质，区间值域上性质蕴含在新树和旧树的差之中。
* 3.标记永久化：路过结点时标记不下放，也不通过子结点更新，而是直接改变其
* 值；向下搜索时记录累积标记值并在最后作用（因此assign()在维护最大值
* 无效）。
* 4.区间第k大也可以整体二分/划分树。
* 5.若维护区间超过int，记得把32换成64。
*****/
struct PerSegTree
{
    struct Node
    {
        int ls, rs;
        ll val, tag;
        Node(): ls(0), rs(0), val(0), tag(0) {}
    };
    vector<Node> tree;
    vector<int> root;
    int size;
    ll L, R;

    int _build(ll l, ll r, ll a[])
    {
        int now = size++;
        if (l == r) tree[now].val = a[l];
        else
        {
            ll m = l + (r - l) / 2;
            tree[now].ls = _build(l, m, a);
            tree[now].rs = _build(m + 1, r, a);
            tree[now].val = tree[tree[now].ls].val + tree[tree[now].rs].val;
        }
        return now;
    }
    void init(ll l, ll r, int cnt, ll a[]) //建初始树
    {
        size = 0;
        L = l, R = r;
        tree.resize(cnt * 32 + 5);
        root.push_back(_build(L, R, a));
        return;
    }
    void init(ll l, ll r, int cnt) //建一个空根
    {
        size = 1;
        L = l, R = r;
        tree.resize(cnt * 32 + 5);
        root.push_back(0);
        return;
    }
    void assign(int ver, ll pos, ll val) { root.push_back(_assign(
        root[ver], L, R, pos, val, 0)); }
    int _assign(int src, ll l, ll r, ll pos, ll val, ll tag)
    {
        int now = size++;
        tree[now] = tree[src];
        tag += tree[now].tag;
        if (l == r) tree[now].val = val - tag;
        else
        {
            ll m = l + (r - l) / 2;
            if (pos <= m) tree[now].ls = _assign(tree[now].ls, l, m,
                pos, val, tag);
            else tree[now].rs = _assign(tree[now].rs, m + 1, r, pos,
                val, tag);
        }
        return now;
    }
    void modify(int ver, ll lef, ll rig, ll val) { root.push_back(
        _modify(root[ver], L, R, lef, rig, val)); }
    int _modify(int src, ll l, ll r, ll lef, ll rig, ll val)
    {
        int now = size++;
        tree[now] = tree[src];
        if (lef <= l && r <= rig) tree[now].tag += val;
        else if (l <= rig && r >= lef)
        {
            tree[now].val += val * (min(rig, r) - max(lef, l) + 1);
            ll m = l + (r - l) / 2;
            if (lef <= m) tree[now].ls = _modify(tree[now].ls, l, m,
                lef, rig, val);
            if (rig > m) tree[now].rs = _modify(tree[now].rs, m + 1, r,
                lef, rig, val);
        }
        return now;
    }
    ll query(int ver, ll lef, ll rig) { return _query(root[ver], L, R,
        lef, rig, 0); }
    ll _query(int src, ll l, ll r, ll lef, ll rig, ll tag)

```

```

{
    tag += tree[src].tag;
    if (lef <= l && r <= rig) return tree[src].val + (r - l + 1) * tag;
    else if (l <= rig && r >= lef)
    {
        int m = l + (r - l) / 2;
        ll res = 0;
        if (lef <= m) res += _query(tree[src].ls, l, m, lef, rig, tag);
        if (rig > m) res += _query(tree[src].rs, m + 1, r, lef, rig, tag);
        return res;
    }
    else return 0;
}
ll kth(ll lef, ll rig, int k) { return _kth(root[lef - 1], root[rig], L, R, k); }
ll _kth(int osrc, int nsrc, ll l, ll r, int k)
{
    if (l == r) return l;
    int nsum = tree[tree[nsrc].ls].val + tree[tree[nsrc].ls].tag;
    int osum = tree[tree[osrc].ls].val + tree[tree[osrc].ls].tag;
    int dif = nsum - osum;
    int m = l + (r - l) / 2;
    if (dif >= k) return _kth(tree[osrc].ls, tree[nsrc].ls, l, m, k);
    else return _kth(tree[osrc].rs, tree[nsrc].rs, m + 1, r, k - dif);
}
};

```

## 5.10 李超线段树

```

/*****
* 时间复杂度：建立 $O(n)$ /修改 $O(\log^2 n)$ /查询 $O(\log n)$ 
* 说明：
* 1. 谨慎使用，注意浮点数精度和结点初始化问题
* 2. 标记永久化，整条链每一层的值都可能是答案
*****/
const int N = 100005;
const double EPS = 1e-9;

struct Seg
{
    double k, b;
    int lef, rig;
    void init(int x0, int y0, int x1, int y1)
    {
        lef = x0, rig = x1;
        if (x0 == x1)
        {
            k = 0, b = max(y0, y1);
        }
        else
        {
            k = double(y1 - y0) / (x1 - x0);
            b = y0 - x0 * k;
        }
    }
    double at(int x) { return k * x + b; }
} seg[N];

struct LCSTree
{
    struct Node
    {
        int lef, rig, id;
    };
    vector<Node> tree;

    LCSTree(int x) { tree.resize(x * 4 + 1); }

    void build(int src, int lef, int rig)
    {
        tree[src] = { lef, rig, 0 };
        if (lef == rig) return;
        int mid = (lef + rig) / 2;
        build(src << 1, lef, mid);
        build(src << 1 | 1, mid + 1, rig);
        return;
    }

    void add(int src, int id)
    {
        if (seg[id].lef <= tree[src].lef && seg[id].rig >= tree[src].rig)

```

```

{
    update(src, id);
    return;
}
if (seg[id].lef <= tree[src << 1].rig) add(src << 1, id);
if (seg[id].rig >= tree[src << 1 | 1].lef) add(src << 1 | 1, id);
return;
}

bool compare(int id1, int id2, int x)
{
    if (id1 == 0) return 1;
    if (id2 == 0) return 0;
    double r1 = seg[id1].at(x);
    double r2 = seg[id2].at(x);
    if (fabs(r1 - r2) < EPS) return id2 < id1;
    else return r2 > r1 + EPS;
}

void update(int src, int id)
{
    int mid = (tree[src].lef + tree[src].rig) / 2;
    if (compare(tree[src].id, id, mid)) swap(tree[src].id, id);
    if (tree[src].lef == tree[src].rig) return;
    if (compare(tree[src].id, id, tree[src].lef)) update(src << 1, id);
    if (compare(tree[src].id, id, tree[src].rig)) update(src << 1 | 1, id);
    return;
}

int query(int src, int x)
{
    if (tree[src].lef == tree[src].rig) return tree[src].id;
    if (x <= tree[src << 1].rig)
    {
        int r = query(src << 1, x);
        if (compare(r, tree[src].id, x)) return tree[src].id;
        else return r;
    }
    else
    {
        int r = query(src << 1 | 1, x);
        if (compare(r, tree[src].id, x)) return tree[src].id;
        else return r;
    }
}
};

```

## 6 树论

### 6.1 LCA

```

/*****
* 时间复杂度： $O(\log M)$ 
* 说明：适用于有根树
*****/
const int N = 500005;

vector<int> node[N];

struct LCA
{
    vector<int> d; // 到根距离
    vector<vector<int>> st;

    void dfs(int x)
    {
        for (auto e : node[x])
        {
            if (e == st[x][0]) continue;
            d[e] = d[x] + 1;
            st[e][0] = x;
            dfs(e);
        }
        return;
    }

    void build(int sz)
    {
        int lg = int(log2(sz));
        for (int i = 1; i <= lg; ++i)
        {
            for (int j = 1; j <= sz; ++j)

```

```

        {
            if (d[j] >= (1 << i))
            {
                st[j][i] = st[st[j][i - 1]][i - 1];
            }
        }
    }
    return;
}

LCA() {}
LCA(int x, int root) { init(x, root); }

void init(int x, int root)
{
    d.resize(x + 1);
    st.resize(x + 1, vector<int>(32));
    dfs(root);
    build(x);
    return;
}

int query(int a, int b)
{
    if (d[a] < d[b]) swap(a, b);
    int dif = d[a] - d[b];
    for (int i = 0; dif >> i; ++i)
    {
        if (dif >> i & 1) a = st[a][i];
    }
    if (a == b) return a;
    else
    {
        for (int i = 31; i >= 0; --i)
        {
            while (st[a][i] != st[b][i])
            {
                a = st[a][i];
                b = st[b][i];
            }
        }
        return st[a][0];
    }
}
};

```

## 6.2 树的直径

```

/*****
* 时间复杂度: O(N)
* 说明:
* 1. 距离任一点最远的点一定是直径的一端
* 2. 任一点距所有叶的最远距离对应的叶一定是直径端点
*****/
const int N = 200005;

struct Edge { int to; ll v; };
vector<Edge> node[N];

pair<int, ll> farthest(int id, ll d, int pa)
{
    pair<int, ll> ret = { id, d };
    for (auto e : node[id])
    {
        pair<int, ll> res;
        if (e.to != pa) res = farthest(e.to, d + e.v, id);
        if (res.second > ret.second) ret = res;
    }
    return ret;
}

int n, m;

void solve()
{
    cin >> n >> m;
    int u, v;
    ll w;
    for (int i = 1; i <= m; ++i)
    {
        cin >> u >> v >> w;
        node[u].push_back({ v, w });
        node[v].push_back({ u, w });
    }
    int s = farthest(1, 0, -1).first;
    auto res = farthest(s, 0, -1);
}

```

```

int t = res.first;
ll d = res.second;
return;
}

```

## 6.3 树哈希

```

/*****
* 时间复杂度: O(nlogn)
* 说明: 判断有根树同构。无根树可通过找重心转换为有根树。
*****/
struct TreeHash
{
    int n, root;
    vector<vector<int>> node;
    vector<int> hav;
    map<vector<int>, int> mp;
    int ord = 0;

    void getTree(vector<int>& p)
    {
        n = p.size() - 1;
        node.clear();
        node.resize(n + 1);
        hav.clear();
        hav.resize(n + 1);
        root = -1;
        for (int i = 1; i <= n; ++i)
        {
            if (p[i])
            {
                node[p[i]].push_back(i);
                node[i].push_back(p[i]);
            }
            else root = i;
        }
        return;
    }

    void getD(int id, int pa, vector<int>& sz, vector<int>& d)
    {
        sz[id] = 1;
        int res = 0;
        for (auto e : node[id])
        {
            if (e != pa)
            {
                getD(e, id, sz, d);
                sz[id] += sz[e];
                res = max(res, sz[e]);
            }
        }
        if (id == root) d[id] = res;
        else d[id] = max(res, n - sz[id]);
        return;
    }

    vector<int> center()
    {
        vector<int> res;
        vector<int> sz(n + 1), d(n + 1);
        int mnn = n;
        getD(root, -1, sz, d);
        for (int i = 1; i <= n; ++i) mnn = min(mnn, d[i]);
        for (int i = 1; i <= n; ++i) if (d[i] == mnn) res.push_back(i);
        return res;
    }

    vector<int> hash(vector<int>& p)
    {
        vector<int> res;
        getTree(p);
        auto v = center();
        for (auto e : v) dfs(e, -1, res.push_back(hav[e]));
        sort(res.begin(), res.end());
        return res;
    }

    int hash(vector<int>& p, int root)
    {
        getTree(p);
        dfs(root, -1);
        return hav[root];
    }
}

```



```

void dfs(int id, int pa)
{
    vector<int> v;
    for (auto e : node[id])
    {
        if (e != pa)
        {
            dfs(e, id);
            v.push_back(hav[e]);
        }
    }
    sort(v.begin(), v.end());
    if (mp.count(v) == 0) mp[v] = ++ord;
    hav[id] = mp[v];
    return;
}
};

```

## 6.4 树链剖分

```

/*****
* 时间复杂度: O(nlogn)
* 说明: 维护树上两点间路径相关性质, 也可求LCA。
*****/
const int N = 100005;

vector<int> node[N];

struct HLD
{
    vector<int> pa, dep, sz, hson;
    vector<int> top, dfn, rnk;
    int ord = 0;

    HLD(int x, int root)
    {
        pa.resize(x + 1);
        dep.resize(x + 1);
        sz.resize(x + 1);
        hson.resize(x + 1);
        top.resize(x + 1);
        dfn.resize(x + 1);
        rnk.resize(x + 1);
        build(root);
        decomp(root);
    }

    void build(int x)
    {
        sz[x] = 1;
        int mxsz = 0;
        for (auto e : node[x])
        {
            if (e != pa[x])
            {
                pa[e] = x;
                dep[e] = dep[x] + 1;
                build(e);
                sz[x] += sz[e];
                if (sz[e] > mxsz)
                {
                    mxsz = sz[e];
                    hson[x] = e;
                }
            }
        }
        return;
    }

    void decomp(int x)
    {
        top[x] = x;
        dfn[x] = ++ord;
        rnk[ord] = x;
        if (hson[pa[x]] == x) top[x] = top[pa[x]];
        for (auto e : node[x]) if (e == hson[x]) decomp(e);
        for (auto e : node[x]) if (e != pa[x] && e != hson[x]) decomp(e);
        return;
    }

    int lcm(int u, int v)
    {
        while (top[u] != top[v])
        {
            if (dep[u] < dep[v]) v = pa[top[v]];

```

```

        else u = pa[top[u]];
    }
    if (dep[u] < dep[v]) return u;
    else return v;
};

```

## 6.5 树上启发式合并

```

/*****
* 时间复杂度: O(nlogn) (*状态更新复杂度)
* 说明:
* 1. 维护一个用于得出答案的状态, 离线预处理每个子树的答案
* 2. 用dfn序代替递归的贡献计算和清除可以优化常数
*****/
const int N = 100005;

vector<int> node[N];

int n;
ll a[N];

struct DsuOnTree
{
    struct State
    {
        vector<int> cnt;
        map<int, ll> mp;
        State() { init(); }
        void init() { cnt.resize(1e5 + 1); }
        void add(ll val)
        {
            if (cnt[val]) mp[cnt[val]] -= val;
            if (mp[cnt[val]] == 0) mp.erase(cnt[val]);
            cnt[val]++;
            mp[cnt[val]] += val;
            return;
        }
        void del(ll val)
        {
            mp[cnt[val]] -= val;
            if (mp[cnt[val]] == 0) mp.erase(cnt[val]);
            cnt[val]--;
            if (cnt[val]) mp[cnt[val]] += val;
            return;
        }
        ll ans() { return mp.rbegin()->second; }
    } state;
    vector<int> big; //每个结点的重子
    vector<int> sz; //每个子树的大小
    vector<ll> ans; //每个子树的答案
    const int root = 1;

    DsuOnTree()
    {
        big.resize(n + 1);
        sz.resize(n + 1);
        ans.resize(n + 1);
    }

    void dfs0(int x, int p)
    {
        sz[x] = 1;
        for (auto e : node[x])
        {
            if (e == p) continue;
            dfs0(e, x);
            sz[x] += sz[e];
            if (sz[big[x]] < sz[e]) big[x] = e;
        }
        return;
    }

    void del(int x, int p) //删除子树贡献
    {
        state.del(a[x]);
        for (auto e : node[x])
        {
            if (e == p) continue;
            del(e, x);
        }
        return;
    }

    void add(int x, int p) //计算子树贡献
    {
        state.add(a[x]);
        for (auto e : node[x])
        {

```

```

        if (e == p) continue;
        add(e, x);
    }
    return;
}
void dfs(int x, int p, bool keep)
{
    for (auto e : node[x]) //计算轻子子树答案
    {
        if (e == big[x] || e == p) continue;
        dfs(e, x, 0);
    }
    if (big[x]) dfs(big[x], x, 1); //计算重子子树答案和贡献
    for (auto e : node[x]) //计算轻子子树贡献
    {
        if (e == big[x] || e == p) continue;
        add(e, x);
    }
    state.add(a[x]); //计算自己贡献
    ans[x] = state.ans(); //计算答案
    if (keep == 0) del(x, p); //删除子树贡献
    return;
}
void work()
{
    dfs0(root, 0);
    dfs(root, 0, 0);
    return;
}
}
};

void solve()
{
    cin >> n;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    int u, v;
    for (int i = 1; i <= n - 1; ++i)
    {
        cin >> u >> v;
        node[u].push_back(v);
        node[v].push_back(u);
    }
    DsuOnTree dot;
    dot.work();
    for (int i = 1; i <= n; ++i) cout << dot.ans[i] << ' ';
    cout << endl;
    return;
}
}

```

## 6.6 点分治

```

/*****
* 时间复杂度：处理结点数O(nlogn)
* 说明：用于树上路径计数问题：以重心为根分治子树，再计算经过重心的路径
*****/
const int N = 100005;
const int D[3][2] = { -1, 0, 1, -1, 0, 1 };

int n, sz[N], maxd[N];
string s;
vector<int> node[N];
bool vis[N];
multiset<pair<int, int>> st;

void getRoot(int x, int fa, int sum, int& root)
{
    sz[x] = 1, maxd[x] = 0;
    for (auto e : node[x])
    {
        if (vis[e] || e == fa) continue;
        getRoot(e, x, sum, root);
        sz[x] += sz[e];
        maxd[x] = max(maxd[x], sz[e]);
    }
    maxd[x] = max(maxd[x], sum - sz[x]);
    if (maxd[x] < maxd[root]) root = x;
    return;
}

void dfs(int x, int fa, pair<int, int> p)
{
    p.first += D[s[x] - 'a'][0];
    p.second += D[s[x] - 'a'][1];
    st.insert(p);
    for (auto e : node[x])
    {

```

```

        if (vis[e] || e == fa) continue;
        dfs(e, x, p);
    }
    return;
}

ll work(int x)
{
    ll res = 0;
    multiset<pair<int, int>> ns;
    for (auto e : node[x])
    {
        if (vis[e]) continue;
        dfs(e, x, make_pair(0, 0));
        for (auto p : st)
        {
            pair<int, int> inv;
            inv.first = -(p.first + D[s[x] - 'a'][0]);
            inv.second = -(p.second + D[s[x] - 'a'][1]);
            if (inv == make_pair(0, 0)) res++;
            res += ns.count(inv);
        }
        for (auto p : st) ns.insert(p);
        st.clear();
    }
    return res;
}

ll divide(int x)
{
    ll res = 0;
    vis[x] = 1;
    res += work(x);
    for (auto e : node[x])
    {
        if (vis[e]) continue;
        int root = 0;
        getRoot(e, x, sz[e], root);
        res += divide(root);
    }
    return res;
}

void solve()
{
    cin >> n >> s;
    s = ' ' + s;
    for (int i = 1; i <= n - 1; ++i)
    {
        int u, v;
        cin >> u >> v;
        node[u].push_back(v);
        node[v].push_back(u);
    }
    maxd[0] = n + 1;
    int root = 0;
    getRoot(1, 0, n, root);
    cout << divide(root) << '\n';
    return;
}
}

```

## 7 图论

### 7.1 2-SAT

```

/*****
* 时间复杂度：O(N+M)
* 说明：按照推导关系建有向图，判断是否有两个矛盾点在同一强连通分量中
*****/
const int N = 2000005;

ll n, m, x, y;
bool a, b;
vector<int> node[N];

struct Tarjan
{
    int sz, cnt, ord;
    stack<int> stk;
    vector<vector<int>> g; //新图
    vector<int> dfn, low, id, val;
    Tarjan(int x)
    {
        sz = x; //点数
        cnt = 0; //强连通分量个数

```

```

ord = 0; //时间戳
dfn.resize(sz + 1); //dfs序
low.resize(sz + 1); //能到达的最小dfn
id.resize(sz + 1); //对应的强连通分量编号
val.resize(sz + 1); //新图点权
}
void dfs(int x)
{
    stk.push(x);
    dfn[x] = low[x] = ++ord;
    for (auto e : node[x])
    {
        if (dfn[e] == 0)
        {
            dfs(e);
            low[x] = min(low[x], low[e]);
        }
        else if (id[e] == 0)
        {
            low[x] = min(low[x], low[e]);
        }
    }
    if (dfn[x] == low[x]) //x为强连通分量的根
    {
        cnt++;
        while (dfn[stk.top()] != low[stk.top()])
        {
            id[stk.top()] = cnt;
            stk.pop();
        }
        id[stk.top()] = cnt;
        stk.pop();
    }
    return;
}
void shrink()
{
    for (int i = 1; i <= sz; ++i)
    {
        if (id[i] == 0) dfs(i);
    }
    return;
}
void rebuild()
{
    for (int i = 1; i <= sz; ++i)
    {
        for (auto e : node[i])
        {
            if (id[i] != id[e]) g[id[i]].push_back(id[e]);
        }
    }
    return;
}
};

struct TwoSat
{
    int sz;
    vector<int> res;
    inline int negate(int x)
    {
        if (x > n) return x - n;
        else return x + n;
    }
    TwoSat(int x)
    {
        sz = x;
        res.resize(sz + 1);
    }
    bool work()
    {
        Tarjan tj(sz * 2);
        tj.shrink();
        for (int i = 1; i <= n; ++i)
        {
            if (tj.id[i] == tj.id[negate(i)]) return 0;
        }
        for (int i = 1; i <= n; ++i)
        {
            res[i] = tj.id[i] > tj.id[negate(i)];
        }
        return 1;
    }
};

void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= m; ++i)

```

```

{
    cin >> x >> a >> y >> b;
    node[x + (!a) * n].push_back(y + b * n);
    node[y + (!b) * n].push_back(x + a * n);
}
TwoSat ts(n);
if (!ts.work()) cout << "IMPOSSIBLE\n";
else
{
    cout << "POSSIBLE\n";
    for (int i = 1; i <= n; ++i) cout << ts.res[i] << ' ';
}
return;
}

```

## 7.2 Bellman-Ford 算法

```

/*****
* 时间复杂度: O(NM)
* 说明:
* 1. 适用于带负权边的单源最短路问题
* 2. 可判断负环, negCir()要在work()后调用
*****/
const int N = 1505;
const ll INFL = 0x3f3f3f3f3f3f3f3f;

struct Edge {ll to, v;};

vector<Edge> node[N];

struct BellmanFord
{
    int sz;
    vector<ll> dis;

    BellmanFord(int x)
    {
        sz = x;
        dis.resize(sz + 1, INFL);
    }

    void work(int s)
    {
        dis[s] = 0;
        for (int i = 1; i <= sz - 1; ++i)
        {
            for (int j = 1; j <= sz; ++j)
            {
                for (auto e : node[j])
                {
                    dis[e.to] = min(dis[e.to], dis[j] + e.v);
                }
            }
        }
        return;
    }

    bool negCir()
    {
        for (int i = 1; i <= sz; ++i)
        {
            for (auto e : node[i])
            {
                if (dis[e.to] > dis[i] + e.v) return 1;
            }
        }
        return 0;
    }
};

```

## 7.3 Dijkstra 算法

```

/*****
* 时间复杂度: 朴素O(N^2)/堆优化O(MlogM)
* 说明:
* 1. 只适用于非负边权
* 2. 稀疏图用堆优化, 稠密图用朴素
* 3. 注意处理图不连通的情况 (dis==INFL)
*****/
const int N = 100005;
const ll INFL = 0x3f3f3f3f3f3f3f3f;

struct Edge {int to, v;};

```

```

vector<Edge> node[N];

struct Dijkstra
{
    struct NodeInfo
    {
        int id;
        ll d;
        bool operator < (const NodeInfo& p1) const
        {
            return d > p1.d;
        }
    };

    int sz;
    vector<int> vis;
    vector<ll> dis;

    Dijkstra(int x)
    {
        sz = x;
        vis.resize(sz + 1);
        dis.resize(sz + 1, INFLL);
    }

    void workO(int s)
    {
        priority_queue<NodeInfo> pq;
        dis[s] = 0;
        pq.push({ s, 0 });
        while (pq.size())
        {
            int now = pq.top().id;
            pq.pop();
            if (vis[now] == 0)
            {
                vis[now] = 1; //被取出一定是最短路
                for (auto e : node[now])
                {
                    if (vis[e.to] == 0 && dis[e.to] > dis[now] + e.v)
                    {
                        dis[e.to] = dis[now] + e.v;
                        pq.push({ e.to, dis[e.to] });
                    }
                }
            }
        }
        return;
    }

    void workS(int s)
    {
        auto take = [&](int x)
        {
            vis[x] = 1;
            for (auto e : node[x])
            {
                dis[e.to] = min(dis[e.to], dis[x] + e.v);
            }
            return;
        };
        dis[s] = 0;
        take(s);
        for (int i = 1; i <= sz - 1; ++i)
        {
            ll mnn = INFLL;
            int id = 0;
            for (int j = 1; j <= sz; ++j)
            {
                if (vis[j] == 0 && dis[j] < mnn)
                {
                    mnn = dis[j];
                    id = j;
                }
            }
            if (mnn == INFLL) return;
            take(id);
        }
        return;
    }
};

```

## 7.4 Dinic 算法

```

* 时间复杂度: 最差 $O(N^2 \cdot M)$  / 二分图匹配 $O(\sqrt{N} \cdot M)$ 
* 说明:
* 1. 求有向网络最大流/最小割
* 2. 也可以求二分图最大匹配
* 3. cap表示残量, cap为0的边满流
*****
const ll INFLL = 0x3f3f3f3f3f3f3f3f;
const int N = 3005;

struct Edge
{
    int to; //终点
    int rev; //反向边对其起点的编号
    ll cap; //残量
    Edge() {}
    Edge(int to, int rev, ll cap) : to(to), rev(rev), cap(cap) {}
};

vector<Edge> node[N];

void AddEdge(int from, int to, ll cap)
{
    int x = node[to].size();
    int y = node[from].size();
    node[from].push_back(Edge(to, x, cap));
    node[to].push_back(Edge(from, y, 0));
    return;
}

struct Dinic
{
    int sz;
    vector<int> dep; //每个点所属层深度
    vector<int> done; //每个点下一个要处理的邻接边
    queue<int> q;

    Dinic(int x)
    {
        sz = x;
        dep.resize(sz + 1);
        done.resize(sz + 1);
    }

    bool bfs(int s, int t) //建立分层图
    {
        for (int i = 1; i <= sz; ++i) dep[i] = 0;
        q.push(s);
        dep[s] = 1;
        done[s] = 0;
        bool f = 0;
        while (q.size())
        {
            int now = q.front();
            q.pop();
            if (now == t) f = 1; //到达终点说明存在增广路
            for (auto e : node[now])
            {
                if (e.cap && dep[e.to] == 0) //还有残量且未访问过
                {
                    q.push(e.to);
                    done[e.to] = 0; //有增广路, 需要重新处理
                    dep[e.to] = dep[now] + 1;
                }
            }
        }
        return f;
    }

    ll dfs(int x, int t, ll flow) //统计增广路总流量
    {
        if (x == t || flow == 0) return flow; //找到汇点或断流
        ll rem = flow; //结点x当前剩余流量
        for (int i = done[x]; i < node[x].size() && rem; ++i)
        {
            done[x] = i; //前i-1条边已经搞定, 不会再有增广路
            auto& e = node[x][i];
            if (e.cap && dep[e.to] == dep[x] + 1) //还有残量且为下一层
            {
                ll inflow = dfs(e.to, t, min(rem, e.cap)); //计算流向e.to的最大流量
                if (inflow == 0) dep[e.to] = 0; //e.to无法流入, 本次增广不再考虑
                e.cap -= inflow; //更新残量
                node[e.to][e.rev].cap += inflow; //更新反向边
                rem -= inflow; //消耗流量
            }
        }
        return flow - rem;
    }
};

```

\*\*\*\*\*

```

11 work(int s, int t)
{
    11 aug = 0, ans = 0;
    while (bfs(s, t))
    {
        while (aug = dfs(s, t, INFL))
        {
            ans += aug;
        }
    }
    return ans;
}
};

```

## 7.5 Floyd 算法

```

/*****
* 时间复杂度:  $O(N^3)$ 
* 说明:
* 1. 求任意两点间最短路
* 2. 最短路径计数
*****/
const int N = 105;
const 11 INFL = 0x3f3f3f3f3f3f3f3f;

int n, m;
11 dis[N][N], cnt[N][N];

void floyd()
{
    for (int i = 1; i <= n; ++i)
    {
        for (int j = 1; j <= n; ++j)
        {
            for (int k = 1; k <= n; ++k)
            {
                if (dis[j][k] > dis[j][i] + dis[i][k])
                {
                    dis[j][k] = dis[j][i] + dis[i][k];
                    cnt[j][k] = cnt[j][i] * cnt[i][k];
                }
                else if (dis[j][k] == dis[j][i] + dis[i][k])
                {
                    cnt[j][k] += cnt[j][i] * cnt[i][k];
                }
            }
        }
    }
    return;
}

void solve()
{
    cin >> n >> m;
    for (int i = 1; i <= n; ++i)
    {
        for (int j = 1; j <= n; ++j)
        {
            dis[i][j] = INFL;
        }
    }
    11 u, v, w;
    for (int i = 1; i <= m; ++i)
    {
        cin >> u >> v >> w;
        dis[u][v] = dis[v][u] = w;
        cnt[u][v] = cnt[v][u] = 1;
    }
    floyd();
    return;
}

```

## 7.6 Kosaraju 算法

```

/*****
* 时间复杂度:  $O(N+M)$ 
* 说明: 有向图强连通分量
*****/
const int N = 10005;

vector<int> node[N];

struct Kosaraju

```

```

{
    int sz, index = 0;
    vector<int> vis, ord;
    vector<vector<int>> rev;
    vector<int> id; //强连通分量编号
    Kosaraju(int x)
    {
        sz = x;
        vis.resize(sz + 1);
        id.resize(sz + 1);
        rev.resize(sz + 1);
        ord.resize(1);
        for (int i = 1; i <= sz; ++i)
        {
            for (auto e : node[i])
            {
                rev[e].push_back(i);
            }
        }
        for (int i = 1; i <= sz; ++i) if (vis[i] == 0) dfs1(i);
        for (int i = sz; i >= 1; --i) if (id[ord[i]] == 0) index++,
            dfs2(ord[i]);
    }

    void dfs1(int x)
    {
        vis[x] = 1;
        for (auto e : node[x])
        {
            if (vis[e] == 0) dfs1(e);
        }
        ord.push_back(x);
        return;
    }

    void dfs2(int x)
    {
        id[x] = index;
        for (auto e : rev[x])
        {
            if (id[e] == 0) dfs2(e);
        }
        return;
    }
};

```

## 7.7 Tarjan 算法

```

/*****
* 时间复杂度:  $O(N+M)$ 
* 说明: 求强连通分量, 也可求缩点后新图
*****/
const int N = 10005;

int n, m;
int a[N]; //旧图点权
vector<int> node[N];

struct Tarjan
{
    int sz, cnt, ord;
    stack<int> stk;
    vector<vector<int>> g; //新图
    vector<int> dfn, low, id, val;
    Tarjan(int x)
    {
        sz = x; //点数
        cnt = 0; //强连通分量个数
        ord = 0; //时间戳
        dfn.resize(sz + 1); //dfs序
        low.resize(sz + 1); //能到达的最小dfn
        id.resize(sz + 1); //对应的强连通分量编号
        val.resize(sz + 1); //新图点权
    }

    void dfs(int x)
    {
        stk.push(x);
        dfn[x] = low[x] = ++ord;
        for (auto e : node[x])
        {
            if (dfn[e] == 0) //未访问
            {
                dfs(e);
                low[x] = min(low[x], low[e]);
            }
            else if (id[e] == 0) //在栈中

```

```

    {
        low[x] = min(low[x], low[e]);
    }
}
if (dfn[x] == low[x]) //x为强连通分量的根
{
    cnt++;
    while (dfn[stk.top()] != low[stk.top()]) //强连通分量中只有根dfn=low
    {
        val[cnt] += a[stk.top()];
        id[stk.top()] = cnt;
        stk.pop();
    }
    val[cnt] += a[stk.top()];
    id[stk.top()] = cnt;
    stk.pop();
}
return;
}
void shrink()
{
    for (int i = 1; i <= sz; ++i)
    {
        if (id[i] == 0) dfs(i);
    }
    return;
}
void rebuild()
{
    for (int i = 1; i <= sz; ++i)
    {
        for (auto e : node[i])
        {
            if (id[i] != id[e]) g[id[i]].push_back(id[e]);
        }
    }
    return;
}
}
};

```

## 7.8 K 短路

```

/*****
* 时间复杂度: O(NklogN)
* 说明: 利用A*算法。以估价函数值优先搜索, 第k次访问某结点即k短路。
*****/
const int N = 1005;
const ll INFL = 0x3f3f3f3f3f3f3f3f;

struct E
{
    ll to, v;
};

struct V
{
    ll id, d;
    bool operator<(const V& v) const { return d > v.d; }
};

int n, m, k;
vector<E> node[N];

struct Dijkstra
{
    int sz;
    vector<ll> d;
    vector<int> vis;
    priority_queue<V> pq;
    vector<vector<E>> rev;

    void rebuild()
    {
        for (int i = 1; i <= sz; ++i)
        {
            for (auto e : node[i])
            {
                rev[e.to].push_back({ i, e.v });
            }
        }
        return;
    }
    Dijkstra(int x, int s)
    {
        sz = x;
    }
}

```

```

d.resize(sz + 1, INFL);
vis.resize(sz + 1);
rev.resize(sz + 1);
rebuild();
d[1] = 0;
pq.push({ 1, 0 });
while (pq.size())
{
    auto now = pq.top();
    pq.pop();
    if (vis[now.id]) continue;
    vis[now.id] = 1;
    for (auto e : rev[now.id])
    {
        if (vis[e.to] == 0 && d[e.to] > d[now.id] + e.v)
        {
            d[e.to] = d[now.id] + e.v;
            pq.push({ e.to, d[e.to] });
        }
    }
}
};

void solve()
{
    cin >> n >> m >> k;
    int u, v, w;
    for (int i = 1; i <= m; ++i)
    {
        cin >> u >> v >> w;
        node[u].push_back({ v, w });
    }
    Dijkstra dj(n, n);
    priority_queue<V> pq;
    vector<int> vis(n + 1);
    pq.push({ n, dj.d[n] });
    vector<ll> ans(k, -1);
    while (pq.size())
    {
        auto now = pq.top();
        pq.pop();
        if (now.id == 1 && vis[now.id] < k) ans[vis[now.id]] = now.d;
        vis[now.id]++;
        for (auto e : node[now.id])
        {
            if (vis[e.to] >= k) continue;
            pq.push({ e.to, now.d - dj.d[now.id] + e.v + dj.d[e.to] });
        }
    }
    for (int i = 0; i < k; ++i) cout << ans[i] << '\n';
    return;
}

```

## 7.9 SSP 算法

```

/*****
* 时间复杂度: O(NMF) (伪多项式, 与最大流有关)
* 说明:
* 1. 求最小费用最大流
* 2. 无法处理负环, 需要提前排除
*****/
const int N = 5005;
const ll INFL = 0x3f3f3f3f3f3f3f3f;

struct Edge
{
    int to; // 终点
    int rev; // 反向边对其起点的编号
    ll cap; // 残量
    ll cost; // 单位流量费用
    Edge() {}
    Edge(int to, int rev, ll cap, ll cost) : to(to), rev(rev), cap(cap), cost(cost) {}
};

vector<Edge> node[N];

void addEdge(int from, int to, ll cap, ll cost)
{
    int x = node[to].size();
    int y = node[from].size();
    node[from].push_back(Edge(to, x, cap, cost));
    node[to].push_back(Edge(from, y, 0, -cost));
    return;
}

```

```

struct SSP
{
    int sz;
    vector<ll> dis; //源点到i的最小单位流量费用
    vector<int> vis;
    vector<int> done; //每个点下一个要处理的邻接边
    queue<int> q;
    ll minc, maxf;

    SSP(int x)
    {
        sz = x;
        dis.resize(sz + 1);
        vis.resize(sz + 1);
        done.resize(sz + 1);
        minc = maxf = 0;
    }

    bool spfa(int s, int t) //寻找单位流量费用最小的增广路
    {
        vis.assign(sz + 1, 0);
        done.assign(sz + 1, 0);
        dis.assign(sz + 1, INFL);
        dis[s] = 0;
        q.push(s);
        vis[s] = 1;
        while (q.size())
        {
            int now = q.front();
            q.pop();
            vis[now] = 0;
            for (auto e : node[now])
            {
                if (e.cap && dis[e.to] > dis[now] + e.cost) //还有残量且可松弛
                {
                    dis[e.to] = dis[now] + e.cost;
                    if (vis[e.to] == 0) q.push(e.to), vis[e.to] = 1;
                }
            }
        }
        return dis[t] != INFL;
    }

    ll dfs(int x, int p, int t, ll flow) //沿增广路计算流量和费用
    {
        if (x == t || flow == 0) return flow; //找到汇点或断流
        vis[x] = 1; //防止零权环死循环
        ll rem = flow; //结点x当前剩余流量
        for (int i = done[x]; i < node[x].size() && rem; ++i)
        {
            done[x] = i; //前i-1条边已经搞定, 不会再有增广路
            auto& e = node[x][i];
            if (e.to != p && vis[e.to] == 0 && e.cap && dis[e.to] == dis[x] + e.cost)
            {
                ll inflow = dfs(e.to, x, t, min(rem, e.cap)); //计算流向e.to的最大流量
                e.cap -= inflow; //更新残量
                node[e.to][e.rev].cap += inflow; //更新反向边
                rem -= inflow; //消耗流量
            }
        }
        vis[x] = 0; //出递归栈后可重新访问
        return flow - rem;
    }

    void work(int s, int t)
    {
        ll aug = 0;
        while (spfa(s, t))
        {
            while (aug = dfs(s, 0, t, INFL))
            {
                maxf += aug;
                minc += dis[t] * aug;
            }
        }
        return;
    }
};

```

## 7.10 原始对偶算法

\*\*\*\*\*

```

* 时间复杂度: O(MlogMF) (伪多项式, 与最大流有关)
* 说明:
* 1. 求最小费用最大流
* 2. 无法处理负环, 需要提前排除
*****/
const int N = 5005;
const ll INFL = 0x3f3f3f3f3f3f3f3f;

struct Edge
{
    int to; //终点
    int rev; //反向边对其起点的编号
    ll cap; //残量
    ll cost; //单位流量费用
    Edge() {}
    Edge(int to, int rev, ll cap, ll cost) : to(to), rev(rev), cap(cap), cost(cost) {}
};

vector<Edge> node[N];

void addEdge(int from, int to, ll cap, ll cost)
{
    int x = node[to].size();
    int y = node[from].size();
    node[from].push_back(Edge(to, x, cap, cost));
    node[to].push_back(Edge(from, y, 0, -cost));
    return;
}

struct PrimalDual
{
    struct NodeInfo
    {
        int id;
        ll d;
        bool operator < (const NodeInfo& p1) const
        {
            return d > p1.d;
        }
    };

    int sz;
    vector<ll> h; //势能
    vector<int> vis;
    vector<int> done; //每个点下一个要处理的邻接边
    vector<ll> dis;
    queue<int> q;
    priority_queue<NodeInfo> pq;
    ll minc, maxf;

    PrimalDual(int x)
    {
        sz = x;
        h.resize(sz + 1, INFL);
        vis.resize(sz + 1);
        done.resize(sz + 1);
        dis.resize(sz + 1);
        minc = maxf = 0;
    }

    void spfa(int s) //求初始势能
    {
        h[s] = 0;
        q.push(s);
        vis[s] = 1;
        while (q.size())
        {
            auto now = q.front();
            q.pop();
            vis[now] = 0;
            for (auto e : node[now])
            {
                if (e.cap && h[e.to] > h[now] + e.cost)
                {
                    h[e.to] = h[now] + e.cost;
                    if (vis[e.to] == 0) q.push(e.to), vis[e.to] = 1;
                }
            }
        }
        return;
    }

    bool dijkstra(int s, int t)
    {
        dis.assign(sz + 1, INFL);
        vis.assign(sz + 1, 0);
        done.assign(sz + 1, 0);
        dis[s] = 0;
        pq.push({ s, 0 });
    }
};

```

```

while (pq.size())
{
    int now = pq.top().id;
    pq.pop();
    if (vis[now] == 0)
    {
        vis[now] = 1; //被取出一定是最短路
        for (auto e : node[now])
        {
            ll cost = e.cost + h[now] - h[e.to];
            if (vis[e.to] == 0 && e.cap && dis[e.to] > dis[now] + cost)
            {
                dis[e.to] = dis[now] + cost;
                pq.push({ e.to, dis[e.to] });
            }
        }
    }
    vis.assign(sz + 1, 0); //还原vis
    return dis[t] != INFL;
}

ll dfs(int x, int t, ll flow) //沿增广路计算流量和费用
{
    if (x == t || flow == 0) return flow; //找到汇点或断流
    vis[x] = 1; //防止零权环死循环
    ll rem = flow; //结点x当前剩余流量
    for (int i = done[x]; i < node[x].size() && rem; ++i)
    {
        done[x] = i; //前i-1条边已经搞定，不会再有增广路
        auto& e = node[x][i];
        if (vis[e.to] == 0 && e.cap && e.cost == h[e.to] - h[x])
            //势能差等于费用表明是最短路
        {
            ll inflow = dfs(e.to, t, min(rem, e.cap)); //计算流向e.to的最大流量
            e.cap -= inflow; //更新残量
            node[e.to][e.rev].cap += inflow; //更新反向边
            rem -= inflow; //消耗流量
        }
    }
    vis[x] = 0; //出递归栈后可重新访问
    return flow - rem;
}

void work(int s, int t)
{
    spfa(s);
    ll aug = 0;
    while (dijkstra(s, t))
    {
        for (int i = 1; i <= sz; ++i) h[i] += dis[i]; //更新势能
        while (aug = dfs(s, t, INFL))
        {
            maxf += aug;
            minc += aug * h[t];
        }
    }
    return;
}
};

```

## 7.11 Prim 算法

```

/*****
* 时间复杂度:  $O(N^2)$ 
* 说明:
* 1. 选点法最小生成树, 适用于稠密图
* 2. 注意考虑图不连通的情况
*****/
const int N = 5005;
const int M = 200005;

const ll INFL = 0x3f3f3f3f3f3f3f3f;

struct Edge {ll to, v;};

vector<Edge> node[N];
int n, m;

struct Prim
{
    int sz;
    vector<int> vis;
    vector<ll> dis;

```

```

Prim(int x)
{
    sz = x;
    vis.resize(sz + 1);
    dis.resize(sz + 1, INFL);
}

ll work()
{
    int now = 1;
    ll ans = 0;
    for (int i = 1; i <= sz - 1; ++i)
    {
        vis[now] = 1;
        for (auto e : node[now])
        {
            dis[e.to] = min(dis[e.to], e.v);
        }
        ll mnn = INFL;
        for (int j = 1; j <= sz; ++j)
        {
            if (vis[j] == 0 && dis[j] < mnn)
            {
                mnn = dis[j];
                now = j;
            }
        }
        if (mnn == INFL) return 0; //不连通
        ans += mnn;
    }
    return ans;
}
};

```

## 7.12 Kruskal 算法

```

/*****
* 时间复杂度:  $O(M\log M)$ 
* 说明:
* 1. 选边法最小生成树, 适用于稀疏图
* 2. 注意考虑图不连通的情况
*****/
const int N = 5005;
const int M = 200005;

struct Edge
{
    ll x, y, v;
    bool operator <(const Edge& e)
    {
        return v < e.v;
    }
};

Edge e[M];
int n, m;

ll kruskal()
{
    DSU dsu(n);
    ll ans = 0;
    sort(e + 1, e + 1 + m);
    for (int i = 1; i <= m; ++i)
    {
        if (dsu.find(e[i].x) != dsu.find(e[i].y))
        {
            ans += e[i].v;
            dsu.merge(e[i].x, e[i].y);
        }
    }
    return ans;
}

```

## 7.13 Kruskal 重构树

```

/*****
* 时间复杂度: 建立 $O(N)$ /查询 $O(\log N)$ 
* 说明:
* 1. 用于解决最小瓶颈路问题
* 2. 考虑了初始图不连通的问题
* 3. 注意n=1特殊情况 (不用建树)
*****/

```



```

const int N = 100005;

struct DSU
{
    vector<int> f;
    void init(int x)
    {
        f.resize(x + 1);
        for (int i = 1; i <= x; ++i) f[i] = i;
        return;
    }
    int find(int id) { return f[id] == id ? id : f[id] = find(f[id]); }
    void attach(int x, int y) //将fx连向fy, 不按秩合并
    {
        int fx = find(x), fy = find(y);
        f[fx] = fy;
        return;
    }
};

struct LCA
{
    vector<int> d;
    vector<vector<int>> st;
    void dfs(int x, vector<vector<int>>& son)
    {
        for (auto e : son[x])
        {
            d[e] = d[x] + 1;
            st[e][0] = x;
            dfs(e, son);
        }
        return;
    }
    void build(int x)
    {
        int lg = int(log2(x));
        for (int i = 1; i <= lg; ++i)
        {
            for (int j = 1; j <= x; ++j)
            {
                if (d[j] >= (1 << i))
                {
                    st[j][i] = st[st[j][i - 1]][i - 1];
                }
            }
        }
        return;
    }
    void init(int x)
    {
        d.resize(x + 1);
        st.resize(x + 1, vector<int>(32));
        return;
    }
    int query(int x, int y)
    {
        if (d[x] < d[y]) swap(x, y);
        int dif = d[x] - d[y];
        for (int i = 0; dif >> i; ++i)
        {
            if (dif >> i & 1) x = st[x][i];
        }
        if (x == y) return x;
        for (int i = 31; i >= 0; --i)
        {
            while (st[x][i] != st[y][i])
            {
                x = st[x][i];
                y = st[y][i];
            }
        }
        return st[x][0];
    }
};

struct Edge
{
    int x, y, v;
    bool operator<(const Edge& rhs) const { return v < rhs.v; }
} edg[N];

struct KrsRebTree
{
    int size; //当前结点数, 最多为n*2-1
    vector<vector<int>> son; //子结点
    vector<int> val; //点权
    LCA lca;
    DSU dsu;
};

```

```

void build(int n, int m)
{
    son.resize(n * 2);
    val.resize(n * 2);
    dsu.init(n * 2 - 1);
    size = n;
    sort(edg + 1, edg + 1 + m);
    for (int i = 1; i <= m && size < n * 2 - 1; ++i)
    {
        int fx = dsu.find(edg[i].x);
        int fy = dsu.find(edg[i].y);
        if (fx == fy) continue;
        size++;
        dsu.attach(fx, size);
        dsu.attach(fy, size);
        son[size].push_back(fx);
        son[size].push_back(fy);
        val[size] = edg[i].v;
    }
    lca.init(size);
    for (int i = n + 1; i <= size; ++i)
    {
        if (dsu.find(i) == i) lca.dfs(i, son); //对所有树的根dfs
    }
    lca.build(size);
    return;
}

int query(int x, int y)
{
    if (dsu.find(x) == dsu.find(y)) return val[lca.query(x, y)];
    else return -1;
}
};

```

## 8 计算几何

### 8.1 平面坐标旋转

```

/*****
* 时间复杂度: O(1)
* 说明: 二维平面上一点绕另一点逆时针旋转
*****/
const double PI = acos(-1);

inline double deg_to_rad(int x) { return x * PI / 180; }

struct Point
{
    double x, y;

    void rotate(double rad)
    {
        double newx = x * cos(rad) - y * sin(rad);
        double newy = x * sin(rad) + y * cos(rad);
        x = newx;
        y = newy;
        return;
    }

    void rotate(Point p, double rad)
    {
        Point rela = { x - p.x, y - p.y };
        rela.rotate(rad);
        x = rela.x + p.x;
        y = rela.y + p.y;
        return;
    }
};

```

## 9 杂项算法

### 9.1 普通莫队算法

```

/*****
* 时间复杂度: O((n+m)sqrt(n))
* 说明: 线性序列普通莫队
*****/
const int N = 50005;
const int M = 50005;

```

```

ll n, m, k, a[N], BLOCK;
ll ans[M];

struct Q
{
    ll l, r, id;
    bool operator<(const Q& rhs) const
    {
        // 奇偶化排序优化常数
        int lb = l / BLOCK, rb = rhs.l / BLOCK;
        if (lb == rb)
        {
            if (r == rhs.r) return 0;
            else return (r < rhs.r) ^ (lb & 1);
        }
        else return lb < rb;
    }
} q[M];

void solve()
{
    cin >> n >> m >> k;
    BLOCK = n / sqrt(m); // 块大小
    for (int i = 1; i <= n; ++i) cin >> a[i];

    // 离线处理询问
    for (int i = 1; i <= m; ++i) q[i].id = i, cin >> q[i].l >> q[i].r;
    sort(q + 1, q + 1 + m);

    // 计算首个询问答案
    vector<int> cnt(k + 1);
    for (int i = q[1].l; i <= q[1].r; ++i) cnt[a[i]]++;
    ll res = 0;
    for (int i = 1; i <= k; ++i) res += cnt[i] * cnt[i];
    ans[q[1].id] = res;

    // 开始转移
    ll l = q[1].l, r = q[1].r;
    auto del = [&](int p)
    {
        res -= cnt[a[p]] * cnt[a[p]];
        cnt[a[p]]--;
        res += cnt[a[p]] * cnt[a[p]];
        return;
    };
    auto add = [&](int p)
    {
        res -= cnt[a[p]] * cnt[a[p]];
        cnt[a[p]]++;
        res += cnt[a[p]] * cnt[a[p]];
        return;
    };
    for (int i = 2; i <= m; ++i)
    {
        while (r < q[i].r) add(++r);
        while (r > q[i].r) del(r--);
        while (l < q[i].l) del(l--);
        while (l > q[i].l) add(--l);
        ans[q[i].id] = res;
    }
    for (int i = 1; i <= m; ++i) cout << ans[i] << '\n';
    return;
}

```

## 9.2 带修改莫队算法

```

/*****
* 时间复杂度: n,m,t同级时O(n^(5/3))
* 说明:
*****/
const int N = 150005;
const int M = 150005;

ll BLOCK;

struct Q
{
    ll l, r, id, t;
    bool operator<(const Q& rhs) const
    {
        // 左右端点都分块
        if (l / BLOCK == rhs.l / BLOCK)
        {
            if (r / BLOCK == rhs.r / BLOCK) return t < rhs.t;

```

```

            else return r / BLOCK < rhs.r / BLOCK;
        }
        else return l / BLOCK < rhs.l / BLOCK;
    }
} q[M];

struct C
{
    ll p, o, v;
} c[M];

ll n, m, a[N], ans[N];

void solve()
{
    cin >> n >> m;
    BLOCK = pow(n, 2.0 / 3);
    for (int i = 1; i <= n; ++i) cin >> a[i];
    ll mx = *max_element(a + 1, a + 1 + n);

    // 离线处理询问
    char op;
    ll t = 0, ord = 0, u, v;
    for (int i = 1; i <= m; ++i)
    {
        cin >> op >> u >> v;
        if (op == 'R') c[++t] = { u, a[u], v }, a[u] = v;
        else ord++, q[ord] = { u, v, ord, t };
    }
    sort(q + 1, q + 1 + ord);

    // 计算首个询问答案
    vector<ll> cnt(mx + 1);
    ll res = 0, l = q[1].l, r = q[1].r, nowt = t;
    auto del = [&](int p)
    {
        cnt[a[p]]--;
        if (cnt[a[p]] == 0) res--;
        return;
    };
    auto add = [&](int p)
    {
        cnt[a[p]]++;
        if (cnt[a[p]] == 1) res++;
        return;
    };
    auto chg = [&](int p, ll v)
    {
        if (p >= 1 && p <= r) del(p);
        a[p] = v;
        if (p >= 1 && p <= r) add(p);
        return;
    };
    while (nowt > q[1].t) a[c[nowt].p] = c[nowt].o, nowt--;
    for (int i = 1; i <= r; ++i) add(i);
    ans[q[1].id] = res;

    // 开始转移
    for (int i = 2; i <= ord; ++i)
    {
        for (int j = q[i - 1].t + 1; j <= q[i].t; ++j) chg(c[j].p, c[j].v);
        for (int j = q[i - 1].t; j > q[i].t; --j) chg(c[j].p, c[j].o);
        while (r < q[i].r) add(++r);
        while (r > q[i].r) del(r--);
        while (l < q[i].l) del(l--);
        while (l > q[i].l) add(--l);
        ans[q[i].id] = res;
    }
    for (int i = 1; i <= ord; ++i) cout << ans[i] << '\n';
    return;
}

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int T = 1;
    // cin >> T;
    while (T--) solve();
    return 0;
}

```

## 9.3 整体二分

```

/*****
* 时间复杂度: 框架O(qlogm)
* 说明:
* 1. 对多个需要二分解决的询问同时二分
* 2. 二分对象为答案值域, 但也将询问序列分到两个值域区间中
* 3. 对于区间[l, r)的check不能到达O(q)/O(m), 应只考虑[l, r)中的值或询问
* 4. 注意分到右半区间的询问目标值要削减
* 5. 注意值域区间和询问区间的开闭
* 6. 注意必要时对元素值去重
*****/
const int N = 300005;

struct Fenwick { /*带时间戳树状数组*/ } fen;
struct Discret { /*离散化*/ } D;

struct Q
{
    int l, r, k, id;
} q[N];

int n, m;
pair<int, int> a[N];
int ans[N];

void bis(int lef, int rig, int q1, int qr)
{
    if (lef == rig - 1)
    {
        for (int i = q1; i < qr; ++i) ans[q[i].id] = lef;
        return;
    }
    int mid = lef + rig >> 1;
    for (int i = lef; i < mid; ++i)
    {
        fen.add(a[i].second, 1);
    }
    queue<Q> q1, q2;
    for (int i = q1; i < qr; ++i)
    {
        int cnt = fen.rsum(q[i].l, q[i].r);
        if (cnt < q[i].k) q2.push({ q[i].l, q[i].r, q[i].k - cnt, q[i].id });
        else q1.push(q[i]);
    }
    int qm = q1 + q1.size();
    for (int i = q1; i < qm; ++i)
    {
        if (q1.size()) q[i] = q1.front(), q1.pop();
        else q[i] = q2.front(), q2.pop();
    }
    fen.clear();
    bis(lef, mid, q1, qm);
    bis(mid, rig, qm, qr);
    return;
}

void solve()
{
    cin >> n >> m;
    fen.init(n);
    for (int i = 1; i <= n; ++i)
    {
        cin >> a[i].first;
        a[i].second = i;
        D.insert(a[i].first);
    }
    D.work();
    for (int i = 1; i <= n; ++i) a[i].first = D[a[i].first];
    sort(a + 1, a + 1 + n);
    for (int i = 1; i <= m; ++i)
    {
        cin >> q[i].l >> q[i].r >> q[i].k;
        q[i].id = i;
    }
    bis(1, n + 1, 1, m + 1);
    for (int i = 1; i <= m; ++i) cout << D.v[ans[i] - 1] << '\n';
    return;
}

```

## 9.4 离散化

```

/*****
* 时间复杂度: O(logn)
* 说明: 注意起始序号
*****/
struct Discret

```

```

{
    vector<ll> v;
    void insert(ll val)
    {
        v.push_back(val);
        return;
    }
    void work()
    {
        sort(v.begin(), v.end());
        v.erase(unique(v.begin(), v.end()), v.end());
        return;
    }
    void clear()
    {
        v.clear();
        return;
    }
    ll operator[](ll val)
    {
        return lower_bound(v.begin(), v.end(), val) - v.begin();
    }
};

```

## 9.5 快速排序

```

/*****
* 时间复杂度: O(nlogn)
* 说明: 两倍常数, 但跳过所有与基准相等的值
*****/
const int N = 100005;

int n;
ll a[N];

int median(int x, int y, int z)
{
    if (a[x] > a[y] && a[z] > a[y]) return a[x] > a[z] ? z : x;
    else if (a[x] < a[y] && a[z] < a[y]) return a[x] < a[z] ? z : x;
    else return y;
}

void QuickSort(int lef, int rig) //[lef, rig]
{
    if (rig <= lef) return;
    int mid = lef + (rig - lef) / 2;
    int pivot = median(lef, mid, rig);
    swap(a[pivot], a[lef]);
    int lp = lef; //第一个等于基准的值
    for (int i = lef + 1; i <= rig; ++i)
    {
        if (a[i] < a[lef]) swap(a[i], a[++lp]);
    }
    swap(a[lef], a[lp]);
    int rp = lp; //最后一个等于基准的值
    for (int i = lp + 1; i <= rig; ++i)
    {
        if (a[i] == a[lp]) swap(a[i], a[++rp]);
    }
    QuickSort(lef, lp - 1);
    QuickSort(rp + 1, rig);
    return;
}

```

## 9.6 枚举集合

```

/*****
* 时间复杂度: O(枚举对象个数)
* 说明: 枚举子集、超集、固定大小集合
*****/
struct EnumSet
{
    vector<int> subset(int x) // 枚举x的子集
    {
        vector<int> res;
        for (int i = x; i >= 1; i = (i - 1) & x) res.push_back(i);
        res.push_back(0);
        return res;
    }

    vector<int> kset(int b, int k) // 枚举b位大小为k的集合
    {
        vector<int> res;
    }
};

```

```

int now = (1 << k) - 1;
while (now < (1 << b))
{
    res.push_back(now);
    int lowbit = now & -now;
    int x = now + lowbit;
    int y = ((now & ~x) / lowbit) >> 1;
    now = x | y;
}
return res;
}

vector<int> superset(int x, int b) // 枚举x的b位超集
{
    vector<int> res;
    for (int i = x; i < (1 << b); i = (i + 1) | x) res.push_back(i);
    return res;
}
};

```

## 9.7 CDQ 分治 + CDQ 分治 = 多维偏序

```

/*****
* 时间复杂度:  $O(n \log^d(d-1)n)$ 
* 说明:
* 1.cdq注意事项详见[CDQ分治+数据结构=多维偏序]
* 2.n维偏序需要n-1层cdq
* 3.第i层cdq将整个区间按第i+1维归并排序, 同时将第i维降为二进制, 然后调用第i+1层cdq; 第n-1层cdq递归将左右分别按第n维排序, 再用双指针按照第n维大小归并, 同时计算左部前n-2维全0元素对右部前n-2维全1元素的贡献
*****/
const int N = 100005;

struct Elem
{
    ll a, b, c;
    ll cnt, id;
    bool xtag;
    bool operator!=(const Elem& e) const
    {
        return a != e.a || b != e.b || c != e.c;
    }
};
e[N], ee[N], eee[N];

int n, k, ans[N], res[N];

bool bya(const Elem& e1, const Elem& e2)
{
    if (e1.a == e2.a && e1.b == e2.b) return e1.c < e2.c;
    else if (e1.a == e2.a) return e1.b < e2.b;
    else return e1.a < e2.a;
}

void cdq2(int lef, int rig)
{
    if (lef == rig - 1) return;
    int mid = lef + rig >> 1;
    cdq2(lef, mid);
    cdq2(mid, rig);
    int p1 = lef, p2 = mid, now = lef;
    int sum = 0;
    while (now < rig)
    {
        //左半部分xtag为0的可以贡献右半部分xtag为1的
        if (p2 == rig || p1 < mid && ee[p1].c <= ee[p2].c)
        {
            eee[now] = ee[p1++];
            sum += eee[now].cnt * (eee[now].xtag == 0);
        }
        else
        {
            eee[now] = ee[p2++];
            res[eee[now].id] += sum * (eee[now].xtag == 1);
        }
        now++;
    }
    for (int i = lef; i < rig; ++i) ee[i] = eee[i];
    return;
}

void cdq1(int lef, int rig)
{
    if (lef == rig - 1) return;
    int mid = lef + rig >> 1;
    cdq1(lef, mid);

```

```

cdq1(mid, rig);
int p1 = lef, p2 = mid, now = lef;
while (now < rig)
{
    if (p2 == rig || p1 < mid && e[p1].b <= e[p2].b)
    {
        ee[now] = e[p1++];
        ee[now].xtag = 0;
    }
    else
    {
        ee[now] = e[p2++];
        ee[now].xtag = 1;
    }
    now++;
}
for (int i = lef; i < rig; ++i) e[i] = ee[i];
cdq2(lef, rig);
return;
}

void solve()
{
    cin >> n >> k;
    vector<Elem> ori(n);
    for (int i = 0; i < n; ++i)
    {
        cin >> ori[i].a >> ori[i].b >> ori[i].c;
        ori[i].cnt = 1;
    }
    sort(ori.begin(), ori.end(), bya);
    int cnt = 0;
    for (auto& x : ori)
    {
        if (cnt == 0 || e[cnt] != x) cnt++, e[cnt] = x, e[cnt].id = cnt;
        else e[cnt].cnt++;
    }
    cdq1(1, cnt + 1);
    for (int i = 1; i <= cnt; ++i)
    {
        res[e[i].id] += e[i].cnt - 1;
        ans[res[e[i].id]] += e[i].cnt;
    }
    for (int i = 0; i < n; ++i) cout << ans[i] << '\n';
    return;
}

```

## 9.8 CDQ 分治 + 数据结构 = 多维偏序

```

/*****
* 时间复杂度:  $O(n \log^d(d-1)n)$ 
* 说明:
* 1.每降一维需要乘 $O(\log n)$ 时间
* 2.适用于高维偏序等小元素对大元素有贡献的问题
* 3.元素需要提前去重
* 4.注意小于等于和小于做法不同, 如分治顺序与排序复原/mid的移动
* 5.贡献有顺序要求如dp时, 先左再合并再右
* 6.有时需要离散化才能利用数据结构
*****/
const int N = 100005;

struct Fenwick { /*带时间戳最大值树状数组*/ } fen;
struct Discret { /*离散化*/ } D;

struct Elem
{
    ll a, b, c;
    ll w, dp;
    bool operator!=(const Elem& e) const { return a != e.a || b != e.b || c != e.c; }
};
e[N];

int n;

bool bya(const Elem& e1, const Elem& e2)
{
    if (e1.a == e2.a && e1.b == e2.b) return e1.c < e2.c;
    else if (e1.a == e2.a) return e1.b < e2.b;
    else return e1.a < e2.a;
}

bool byb(const Elem& e1, const Elem& e2)
{
    if (e1.b == e2.b) return e1.c < e2.c;
    else return e1.b < e2.b;
}

```

```

}

void cdq(int lef, int rig)
{
    if (e[lef].a == e[rig - 1].a) return;
    int mid = lef + (rig - lef) / 2;

    // 需要保证e[mid-1].a和e[mid].a不同
    if (e[lef].a == e[mid].a)
    {
        while (e[lef].a == e[mid].a) mid++;
    }
    else
    {
        while (e[mid - 1].a == e[mid].a) mid--;
    }

    // 解决左半
    cdq(lef, mid);

    // 解决合并
    sort(e + lef, e + mid, byb);
    sort(e + mid, e + rig, byb);
    int p1 = lef, p2 = mid;
    while (p2 < rig)
    {
        while (p1 < mid && e[p1].b < e[p2].b)
        {
            fen.add(D[e[p1].c], e[p1].dp);
            p1++;
        }
        e[p2].dp = max(e[p2].dp, e[p2].w + fen.pres(D[e[p2].c] - 1));
        p2++;
    }
    fen.clear();

    // 解决右半
    sort(e + mid, e + rig, bya); // 复原排序
    cdq(mid, rig);
    return;
}

void solve()
{
    cin >> n;
    vector<Elem> ori(n);
    for (int i = 0; i < n; ++i)
    {
        cin >> ori[i].a >> ori[i].b >> ori[i].c >> ori[i].w;
        ori[i].dp = ori[i].w;
        D.insert(ori[i].c);
    }
    D.work();
    fen.init(D.v.size());
    sort(ori.begin(), ori.end(), bya);
    int cnt = 0;
    for (auto& x : ori)
    {
        if (cnt == 0 || e[cnt] != x) e[++cnt] = x;
        else e[cnt].dp = e[cnt].w = max(e[cnt].w, x.w);
    }
    cdq(1, cnt + 1);
    ll ans = 0;
    for (int i = 1; i <= cnt; ++i) ans = max(ans, e[i].dp);
    cout << ans << '\n';
    return;
}

```

## 10 博弈论

### 10.1 Fibonacci 博弈

```

/*****
* 时间复杂度: O(1)
* 说明:
* 1. 有一堆石子, 两人轮流取。先手第一次不能直接取完。每次至少取一个,
* 但最多取上一个人的两倍。取走最后一个石子的人获胜。
* 2. 结论: 是斐波那契数则先手必败, 否则先手必胜。
*****/
bool Fibonacci(ll x) // 返回先手是否必胜
{
    ll a = 1, b = 1;
    while (max(a, b) <= x)
    {
        if (a < b) a += b;

```

```

        else b += a;
        if (max(a, b) == x) return 0;
    }
    return 1;
}

```

### 10.2 Wythoff 博弈

```

/*****
* 时间复杂度: O(1)
* 说明:
* 1. 有两堆石子, 两人轮流取。每次可以在一堆中取任意个石子或在两堆中取同
* 样多的任意个石子, 取走最后一个石子的人获胜。
* 2. 若x和y极大则需要注意精度问题。
*****/
bool Wythoff(ll x, ll y) // 返回先手是否必胜
{
    const double K = ((1.0 + sqrt(5.0)) / 2.0);
    ll res = abs(x - y) * K;
    return res != min(x, y);
}

```

### 10.3 Green Hackenbush 博弈

```

/*****
* 时间复杂度: O(n)
* 说明:
* 1. 有一棵有根树, 两人轮流选择一个子树删除, 删除根结点的人失败。
* 2. 有一颗有根树, 两人轮流删除一条边以及不与根相连的部分, 无边可删
* 的人失败。
* 3. 结论: 以边为对象: 叶结点父边sg值为1, 中间结点父边sg值为所有邻接
* 边sg值异或+1; 以点为对象: 叶结点sg值为0, 其他结点sg值为所有邻接
* 点sg值+1的异或和。
*****/
void dfs(int x, int fa)
{
    sg[x] = 0;
    for (auto e : node[x])
    {
        if (e == fa) continue;
        dfs(e, x);
        sg[x] ^= sg[e] + 1;
    }
    return;
}

```