汕头大学 ACM 模板

目录

- 汕头大学 ACM 模板
  - 目录
  - 动态规划
    - 背包问题
      - 01背包
      - 完全背包
      - 多重背包
      - 可行性多重背包
    - LIS
    - 技巧
  - 数学
    - 数值分析
      - 大数运算
      - 平方根
      - 矩阵
      - 快速傅里叶变换
    - 数论
      - 一些性质
      - GCD & LCM
      - Extened GCD
      - 素数筛法
      - 欧拉函数
      - 快速幂
      - 素性检测
      - 质因数分解
      - 反素数
  - 图论
    - 一些性质
    - 拓扑排序
    - 最短路
      - Bellman-Ford (SPFA)
      - Dijkstra
      - Floyd
      - K短路
    - 生成树
      - Kruskal
      - Prim
      - 次小生成树
    - 网络流

        - Edmonds-Karp
        - Scaling
        - Dinic
        - 上下界可行流
    - 连通分量
        - 强连通分量（Kosaraju）
        - 强连通分量（Tarjan）
        - 双连通分量（Tarjan）
    - LCA
        - Tarjan 离线算法
        - 倍增法
    - 匹配与覆盖
    - Floyd判圈算法
    - 欧拉通路/回路
- 字符串算法
    - 字符串哈希算法
    - Karp-Rabin
    - Z-algorithm
    - KMP
    - AC自动机
    - 后缀数组
- 区间问题
    - Sparse Table（稀疏表）
    - 树状数组
    - 线段树
    - 主席树
    - 莫队算法
        - 普通莫队
        - 带修改莫队
- 其他
    - 枚举技巧
        - 枚举排列
        - 枚举子集
    - 位运算魔法
    - 并查集
    - 表达式树
    - 整体二分
    - 一些杂七杂八的东西
        - .vimrc
        - debug template
    - 对拍相关
        - 随机数生成
        - 脚本
        - 树数据生成

动态规划

背包问题

状态：从前 n 个背包中组成重量为 m 的……

01背包

$O(VN)$

滚动数组，从右往左。

完全背包

$O(VN)$

滚动数组，从左往右。

多重背包

$O(V\sum\log M_i)$

每种物品分解成多个01背包中的物品，每个物品个数分别为1、2、4……个。

可行性多重背包

$O(VN)$

状态：用了前i种物品填满容量为j的背包后，最多还剩下几个第i种物品可用。

$$dp[i][j]=\begin{cases}M[i], & \text{if } dp[i-1][j]\ge 0\\ \max\{dp[i][j-C[i]]-1\}(C[i]\le j\le V), & \text{if } dp[i-1][j]<0\end{cases}$$

```
// HDU 2844
int n, m;
while (cin >> n >> m && n) {
    for (int i = 1; i <= n; ++i) cin >> a[i];
    for (int i = 1; i <= n; ++i) cin >> c[i];
    memset(dp, -1, sizeof(dp));
    dp[0] = 0;
    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j <= m; ++j) {
            if (dp[j] >= 0) dp[j] = c[i];
            else if (j >= a[i]) dp[j] = max(dp[j], dp[j - a[i]] - 1);
        }
    }
    int ans = 0;
    for (int i = 1; i <= m; ++i)
        if (dp[i] >= 0) ++ans;
    cout << ans << '\n';
}
```

LIS

$dp[i]$表示长度为$i+1$的上升子序列中末尾元素的最小值，len表示当前最长上升子序列的长度；

遍历数组，对于每一个元素，把它替换成尽量长的最长子序列的最后一个元素。

$O(n\log n)$

```
int n;
cin >> n;
dp[len++] = INF;
for (int i = 0; i < n; ++i) {
    cin >> arr[i];
    int pos = lower_bound(dp, dp + len, arr[i]) - dp;
    dp[pos] = arr[i];
    if (pos == len) ++len;
}
cout << len << '\n';
```

技巧

1. 某些 dp 的状态可看作分层图，此时转移可用邻接矩阵，并使用矩阵快速幂进行状态转移。（在无权图中，设A为图的邻接矩阵，A^n为矩阵的 n 次幂，则A^n中的元素(i,j)表示顶点 i 到顶点 j 的长度为 n 的路径数）

数学

数值分析

大数运算

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

public class Main{

    public static void main(String args[]){
        Scanner cin = new Scanner(System.in);
        //使用Sacnner类创建cin对象
        BigInteger a, b;//创建大数对象
        while(cin.hasNext()){
            a = cin.nextBigInteger();
            b = cin.nextBigInteger();
            System.out.println("a+b=" + a.add(b));
            System.out.println("a-b=" + a.subtract(b));
            System.out.println("a*b=" + a.multiply(b));
            System.out.println("a/b=" + a.divide(b));
            System.out.println("a%b=" + a.remainder(b));

            if(a.compareTo(b) == 0) //比较两数的大小
                System.out.println("a==b");
            else if(a.compareTo(b) > 0)
                System.out.println("a>b");
            else
```

```
                System.out.println("a<b");

            System.out.println(a.abs());//取绝对值

            int e = 10;
            System.out.println(a.pow(e));//求a^e

            System.out.println(a.toString()); //将大数a转字符串输出

            int p = 8;
            System.out.println(a.toString(p)); //将大数a转换成p进制后 按字符串输出
        }
    }
}
```

平方根

占坑（

矩阵

```
struct matrix {
    typedef LL ele_type;
    int r, c;
    vector<vector<ele_type>> ma;

    matrix(int r, int c): r(r), c(c), ma(r, vector<ele_type>(c)) {}
    matrix(matrix &&rhs) noexcept : r(rhs.r), c(rhs.c) {
        ma = move(rhs.ma);
    }
    matrix(const matrix &rhs) = default;

    matrix& operator= (matrix &&rhs) noexcept {
        if (this != &rhs) {
            r = rhs.r;
            c = rhs.c;
            ma = move(rhs.ma);
        }
        return *this;
    }

    matrix operator* (const matrix &rhs) const {
        matrix res(r, rhs.c);
        for (int i = 0; i < r; ++i) {
            for (int j = 0; j < c; ++j) {
                for (int k = 0; k < rhs.c; ++k) {
                    res.ma[i][k] += ma[i][j] * rhs.ma[j][k];
                    res.ma[i][k] %= MOD;
//                    if (res.ma[i][k] >= MOD2) res.ma[i][k] -= MOD2; // MOD2 == MOD * MOD
                }
            }
        }
        return res;
    }
```

```
    void setUnit() {
        for (int i = 0; i < r; ++i) ma[i][i] = 1;
    }
};
```

快速傅里叶变换

$O(n \log n)$

```
complex<double> a[N << 1];
complex<double> b[N << 1];
void init_pos(complex<double> *a, int n) {  // 逆向加法
    for (int i = 0, j = 0; i != n; i++) {
        if (i > j) swap(a[i], a[j]);
        for (int l = n >> 1; (j ^= l) < l; l >>= 1);
    }
}
void FFT(int I, complex<double> *a, int n) {  // 傅里叶变换(I==1) or 插值
    init_pos(a, n);
    for (int i = 2, mid = 1; i <= n; i <<= 1, mid <<= 1) {
        complex<double> wn(cos(2.0 * PI / i), sin(I * 2.0 * PI / i));
        for (int j = 0; j < n; j += i) {
            complex<double> w(1, 0);
            for (int k = j; k < j + mid; k++, w = w * wn) {
                complex<double> l = a[k], r = w * a[k + mid];
                a[k] = l + r;
                a[k + mid] = l - r;
            }
        }
    }
    if (I == 1) return;
    for (int i = 0; i < n; i++) {
        a[i] /= n;
    }
}
```

数论

一些性质

1. (a - b) % k == 0 <=> a % k == b % k
2. a % b < a / 2 (a >= b)
3. 完全平方数有奇数个约数。1到n中有 $\lfloor \sqrt{n} \rfloor$ 个完全平方数。
4. 有 n 位的数的平方根有 $\lceil \frac{n}{2} \rceil$ 位。

GCD & LCM

$O(\log \max(a, b))$

```
inline int gcd(int a, int b) {
    int tmp;
    while (b != 0) {
        tmp = b;
        b = a % b;
        a = tmp;
    }
    return a;
}

inline int lcm(int a, int b) {
    return a / gcd(a, b) * b;
}
```

Extened GCD

$O(\log \max(a, b))$

```
// solve ax + by = gcd(a, b)
int extgcd(int a, int b, int &x, int &y) { // return gcd(a, b)
    int d = a;
    if (b != 0) {
        d = extgcd(b, a % b, y, x);
        y -= (a / b) * x;
    }
    else {
        x = 1; y = 0;
    }
    return d;
}

// get inverse number of a modulo m
int mod_inv(int a, int m) {
    int x, k;
    extgcd(a, m, x, k);
    return (x % m + m) % m;
}
```

素数筛法

$O(n \log \log n)$

```
// Eratosthenes筛法
int C[N]; // 若是素数则为0，否则为该数的最大素因子
void sieve(int size) {
    for (int i = 2; i * i < size; i++) {
        if (!C[i]) {
            for (int j = i * 2; j < size; j += i)
                C[j] = i;
        }
    }
}
```

$O(n)$

```
// Euler筛法
int C[N]; // 该数的最小质因数
int prime[N], pn; // 素数表
void sieve(int size) {
    for (int i = 2; i < size; ++i) { //
        if (!C[i]) C[i] = prime[pn++] = i;
        for (int j = 0; i * prime[j] < size; ++j) {
            C[i * prime[j]] = prime[j];
            if (i % prime[j] == 0) break; // smallest prime factor of i is prime[j], so we need break
        }
    }
}
```

欧拉函数

欧拉函数 $\phi(n)$ 定义为：$1 \le k \le n$，并且 k 与 n 互质的 k 的个数。

```
// use linear sieve to compute phi(x) for all x up to n
int phi[N];
int C[N];
int p[N], pn;
void getPhi(int n) {
    phi[1] = 1;
    for (int i = 2; i < n; ++i) {
        if (!C[i]) {
            p[pn++] = i;
            phi[i] = i - 1; // ① i is prime
        }
        for (int j = 0; i * p[j] < n; ++j) {
            C[i * p[j]] = p[j];
            if (i % p[j] == 0) {
                phi[i * p[j]] = phi[i] * p[j]; // ② phi(ip) = phi(i) * p, if p divides i
                break; // smallest prime factor of i is p[j], so we need break
            }
            else {
                phi[i * p[j]] = phi[i] * phi[p[j]]; // ③ p does not divides i, i.e. they are co-prime, so phi(ip) = phi(i) * phi(p) due to phi is a multiplicative function
            }
        }
    }
}
```

快速幂

mul：$O(1)$

quickPow：$O(\log n)$

```
LL mul(LL a, LL b, LL mod) { // 带模乘法
    if (mod <= 2000000000) return a * b % mod;
    LL d = llround((long double)a * b / mod);
    LL ans = a * b - d * mod;
    if (ans < 0) ans += mod;
```

```
    return ans;
}

// 求a在模MOD下的逆元 | 费马小定理 | inv(a, MOD - 2, MOD)
LL quickPow(LL a, LL n, LL mod) { // 带模快速幂
    LL ans = 1;
    while (n) {
        if (n & 1) ans = mul(ans, a, mod);
        a = mul(a, a, mod);
        n >>= 1;
    }
    return ans;
}
```

素性检测

$O(\sqrt{n})$

```
// 试除法
bool isprime(int n) {
    for (int i = 2; i * i <= n; ++i) {
        if (n % i == 0) return false;
    }
    return true;
}
```

$O(\log n)$

```
// Miller-Rabin 素性检测
bool isprime(LL n, int S = 7) {
    if (n == 2) return true;
    if (n < 2 || n % 2 == 0) return false;

    LL u = n - 1, t = 0;
    while (u % 2 == 0) {
        u >>= 1;
        ++t;
    }

    const int SPRP[7] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
    // cover all numbers < 2^64

    for (int k = 0; k < S; ++k) {
        LL a = SPRP[k] % n;
        if (a == 0 || a == 1 || a == n - 1) continue;

        LL x = qpow(a, u, n); // 带模快速幂
        if (x == 1 || x == n - 1) continue;

        for (int i = 0; i < t - 1; ++i) {
            x = mul(x, x, n); // 带模乘法
            if (x == 1) return false;
            if (x == n - 1) break;
        }
```

```
        if (x == n - 1) continue;

        return false;
    }
    return true;
}
```

质因数分解

$O(\sqrt{n})$

```
// 直接小除到大
vector<int> p; // 存放分解的质因数
void factorize(int n) {
    for (int i = 2; i * i <= n; ++i) {
        while (n % i == 0) {
            p.push_back(i);
            n /= i;
        }
        if (n == 1) return;
    }
    if (n > 1) p.push_back(n);
}
```

$O(\log n)$

```
// 预处理出各数的最小质因数，然后按最小质因数除
int C[N]; // 最小质因数，可用欧拉筛获得
vector<int> p;
void factorize(int n) {
    for (int t = n; t > 1; ) {
        int x = C[t]; // t的最小质因数
        while (C[t] == x) {
            p.push_back(x);
            t /= C[t];
        }
    }
}
```

Pollard's ρ：$O(n^{\frac{1}{4}})$（大约）

factorize：$O(\text{玄学})$

```
// Pollard's ρ 算法 POJ 1811
inline LL g(LL x, int c, LL mod) { // （伪）随机数生成
    return (mul(x, x, mod) + c) % mod;
}
LL pollard_rho(LL n, int c) {
    LL x = 2, y = 2, i = 1, k = 2;
    while (true) {
        x = g(x, c, n);
        // if (x == y) return n;

        LL d = __gcd(llabs(x - y), n);
```

```
        if (d > 1 && d <= n) return d;
        if (++i == k) {
            y = x;
            k <<= 1;
        }
    }
}

vector<LL> factor; // 刚执行完后是无序的
void factorize(LL n) { // 能够承受10000个数左右的分解（? ）
    if (n == 1) return;

    if (isprime(n)) {
        factor.push_back(n);
        return;
    }

    LL d = n;
    for (int c = 1; d == n; ++c) {
        d = pollard_rho(n, c);
    }

    factorize(d);
    factorize(n / d);
}
```

反素数

对于任何正整数x，其约数的个数记作$g(x)$。例如$g(1) = 1$、$g(6) = 4$。

如果某个正整数x满足：$g(x) > g(i)(0 < i < x)$，则称x为反质数。

性质一：一个反素数的质因子必然是从2开始连续的质数

性质二：分解过后的形式（如 $p = 2^{t_1} * 3^{t_2} * 5^{t_3} * \cdots$ ），必然有 $t1 \geq t2 \geq t3 \geq \cdots$

```
int anti_prime[] =
{1396755360,1102701600,735134400,698377680,551350800,367567200,294053760,245044800,183783600,147026880,122522400,110270160,73513440,61261200,43243200,36756720,32432400,21621600,17297280,14414400,10810800,8648640,7207200,6486480,4324320,3603600,2882880,2162160,1441440,1081080,720720,665280,554400,498960,332640,277200,221760,166320,110880,83160,55440,50400,45360,27720,25200,20160,15120,10080,7560,5040,2520,1680,1260,840,720,360,240,180,120,60,48,36,24,12,6,4,2,1,0};
```

图论

一些性质

1. 先从任意一个节点u开始深搜找到一个最远点v，再从v深搜找到最远点w，则v->w就是树的直径。
2. 握手定理：任何一个无向图都有偶数个度数为奇数的顶点。

拓扑排序

$O(V+E)$

```cpp
// dfs版拓扑排序，可判断是否为有向无环图，若不是则返回 false
vector<int> topo; // 拓扑序的倒序
bool dfs(int u) {
    vis[u] = -1; // -1为正在处理的节点，0为未处理的节点，1为已经处理完的节点
    for (int v = 1; v <= n; ++v) { // 对于每一个u的后继v
        if (g[u][v]) {
            if (vis[v] < 0) return false;
            if (!vis[v] && !dfs(v)) return false;
            // 若v没被处理并且v中发现环，则停止深搜，返回false
            // 若v是已被处理完的节点，则忽略
        }
    }
    vis[u] = 1;
    topo.push_back(u);
    return true;
}
bool toposort() {
    memset(vis, 0, sizeof(vis));
    topo.clear();
    for (int u = 1; u <= n; ++u) {
        if (!vis[u]) {
            if (!dfs(u)) return false;
        }
    }
    return true;
}
```

```cpp
// bfs版拓扑排序，会遍历所有*不在*环中的节点
void toposort() { // 无向图版
    queue<int> q;
    for (int i = 1; i <= n; ++i) {
        if (degree[i] == 1) q.push(i);
    }
    while (!q.empty()) {
        int u = q.front(); q.pop();
        // operation here
        for (int i = 0; i < g[u].size(); ++i) {
            int v = g[u][i];
            if (--degree[v] == 1) {
                // 具体操作
                q.push(v);
            }
        }
    }
}
```

最短路

Bellman-Ford (SPFA)

$O(VE)$（对于大部分图 SPFA 还是很快的）

```cpp
// 单纯判环 UVA 11090
double d[N];
bool vis[N];
bool SPFA(int u) {
    if (vis[u]) return true;
    vis[u] = true;
    for (int i = g[u]; ~i; i = es[i].next) {
        double tmp = d[u] + es[i].v;
        if (tmp < d[es[i].b]) {
            d[es[i].b] = tmp;
            if (SPFA(es[i].b)) return true;
        }
    }
    vis[u] = false;
    return false;
}

bool detect() {
    for (int i = 1; i <= n; ++i) {
        vis[i] = false;
        d[i] = 0;
    }
    for (int i = 1; i <= n; ++i) if (SPFA(i)) return true;
    return false;
}
```

```cpp
// 求最短路并判环
double d[N]; // 最短路长度
int L[N]; // 源点到各个节点的最短路的边数
bool inq[N];

bool SPFA(int s) { // true表示无环，false表示有环
    memset(d, 0x3f, sizeof(d));
    memset(inq, 0, sizeof(inq));
    memset(L, 0, sizeof(L));
    queue<int> q;
    d[s] = 0;
    q.push(s);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        inq[u] = false;
        for (int i = g[u]; ~i; i = es[i].next) {
            int v = es[i].b; double val = es[i].v;
            double tmp = d[u] + val;
            if (tmp < d[v]) {
                L[v] = L[u] + 1;
                if (L[v] >= n) return false; // 最短路长度大于等于顶点数即有环
                d[v] = tmp;
                if (!inq[v]) {
                    q.push(v);
                    inq[v] = true;
                }
            }
```

```
        }
    }
    return true;
}
```

Dijkstra

$O(E \log E)$

```
int d[N];
bool vis[N];
void dijkstra(int s) {
    memset(d, 0x3f, sizeof(d));
    memset(vis, 0, sizeof(vis));
    priority_queue<pair<int, int>> q;
    d[s] = 0;
    q.push({0, s});
    while (!q.empty()) {
        int u = q.top().second; q.pop();
        if (vis[u]) continue;
        vis[u] = true;
        for (auto &e : g[u]) {
            int v = e.first, w = e.second;
            int tmp = d[u] + w;
            if (tmp < d[v]) {
                d[v] = tmp;
                q.push({-tmp, v});
            }
        }
    }
}
```

Floyd

$O(V^3)$

```
int d[N][N];
int g[N][N];
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) d[i][j] = 0;
        else if (g[i][j]) d[i][j] = g[i][j];
        else d[i][j] = INF;
    }
}
for (int k = 1; k <= n; ++k) {
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j) {
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

K短路

O(玄学)

```cpp
// POJ 2449
#include <bits/stdc++.h>

using namespace std;

const int INF = 0x3f3f3f3f;
const double EPS = 1e-8;
//const double PI = acos(-1.0);
//const int MOD = 1000000007;

typedef long long LL;
typedef unsigned long long uLL;
const int N = 1000+10;

int S, T, K;
vector<pair<int, int> > g[N];
vector<pair<int, int> > rev[N]; // 将所有边反向的逆图
int h[N];
int cnt[N]; // 每个节点的访问次数
bool vis[N];

void dij() { // 求出最优的h，即每个节点到终点的最短路长度
    memset(h, 0x3f, sizeof(h));
    priority_queue<pair<int, int> > q;
    h[T] = 0;
    q.push({0, T});
    while (!q.empty()) {
        int u = q.top().second; q.pop();
        if (vis[u]) continue;
        vis[u] = true;
        for (int i = 0; i < rev[u].size(); ++i) {
            int tmp = h[u] + rev[u][i].second;
            if (tmp < h[rev[u][i].first]) {
                h[rev[u][i].first] = tmp;
                q.push({-tmp, rev[u][i].first});
            }
        }
    }
}

int astar() {
    priority_queue<pair<int, int> > q;
    q.push({-h[S], S});
    while (!q.empty()) {
        int u = q.top().second, val = -q.top().first; q.pop();
        if (++cnt[u] == K && u == T) return val; // 访问K次即K短路
        if (cnt[u] > K) continue;
        for (int i = 0; i < g[u].size(); ++i) {
            int v = g[u][i].first, t = g[u][i].second;
            if (cnt[v] < K) {
                q.push({-val + h[u] - t - h[v], v});
```

```
            }
        }
    }
    return -1;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n, m;
    scanf("%d%d", &n, &m);
    while (m--) {
        int a, b, t;
        scanf("%d%d%d", &a, &b, &t);
        g[a].push_back({b, t});
        rev[b].push_back({a, t});
    }
    scanf("%d%d%d", &S, &T, &K);
    if (S == T) ++K;
    dij();
    printf("%d\n", astar());
    return 0;
}
```

生成树

Kruskal

对边排序然后贪心

$O(E \log E)$

Prim

和Dijkstra相似

$O(E \log E)$

次小生成树

$O(E \log E + V^2)$ or $O(E \log E + V \log V)$

```
// POJ 1679
#include <bits/stdc++.h>

using namespace std;

const int INF = 0x3f3f3f3f;
const double EPS = 1e-8;
// const double PI = acos(-1.0);
// const int MOD = 1000000007;

typedef long long LL;
```

```cpp
typedef unsigned long long uLL;
const int N = 100 + 10;
const int M = 2000000 + 10;

int n, m;
struct edge {
    int a, b, w;
    bool operator<(const edge &r) const { return w < r.w; }
};
vector<edge> es;

int fa[N];
int find(int u) { return (fa[u] < 0 ? u : (fa[u] = find(fa[u]))); }

vector<pair<int, int> > spt[N]; // 最小生成树
vector<int> candi; // 次小生成树的候选边（即不在最小生成树中的边）
void setUnion(int a, int b) {
    int r1 = find(a), r2 = find(b);
    if (r1 == r2) return;
    if (fa[r1] < fa[r2]) {
        fa[r2] = r1;
        return;
    }
    if (fa[r1] == fa[r2]) --fa[r2];
    fa[r1] = r2;
}

int kruskal() {
    memset(fa, -1, sizeof(fa));
    candi.clear();
    sort(es.begin(), es.end());
    int ans = 0;
    for (int i = 0; i < es.size(); ++i) {
        edge &e = es[i];
        if (find(e.a) != find(e.b)) {
            setUnion(e.a, e.b);
            spt[e.a].push_back({e.b, e.w});
            spt[e.b].push_back({e.a, e.w});
            ans += e.w;
        } else {
            candi.push_back(i);
        }
    }
    return ans;
}

int F[N][N]; // 最小生成树中任意两点路径中最大边权

bool vis[N];

void dfs(int u, int fa, int ans, int a) {
    for (int i = 0; i < spt[u].size(); ++i) {
        int v = spt[u][i].first;
```

```
        if (v != fa) {
            dfs(v, u, F[a][v] = max(ans, spt[u][i].second), a);
        }
    }
}

void pre() { // 求出F，可利用LCA与区间查询将复杂度将为O(nlogn)
    for (int i = 1; i <= n; ++i) dfs(i, -1, -1, i);
}

void addedge(int u, int v, int w) { es.push_back({u, v, w}); }

int solve(int ori) {
    int ans = INF;
    for (int i = 0; i < candi.size(); ++i) { // 枚举每条候选边
        edge &e = es[candi[i]];
        ans = min(ori - F[e.a][e.b] + e.w, ans); // 将它与F[e.a][e.b]的边替换
    }
    return ans;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int T;
    scanf("%d", &T);
    while (T--) {
        scanf("%d%d", &n, &m);
        es.clear();
        for (int i = 1; i <= n; ++i) spt[i].clear();
        for (int i = 0; i < m; ++i) {
            int u, v, w;
            scanf("%d%d%d", &u, &v, &w);
            addedge(u, v, w);
        }
        int ans = kruskal();
        pre();
        int tmp = solve(ans);
        if (tmp == ans) // 若次小生成树权值与最小生成树相同
            printf("Not Unique!\n");
        else
            printf("%d\n", ans);
    }
    return 0;
}
```

$O(E \log E)$

```
// codeforces contest 1108 F
// improved kruskal
// return the number of alternative edges of MST
int kruskal() {
    int ret = 0;
```

```cpp
    memset(fa, -1, sizeof(fa));
    sort(es.begin(), es.end());
    map<int, set<pii>> mp;
    int cnt = 0;
    for (int i = 0; i < es.size(); ++i) {
        auto &e = es[i];
        if (i + 1 == es.size() || es[i + 1].w != e.w) {
            for (int j = i; j >= 0 && es[j].w == e.w; --j) {
                if (find(es[j].u) != find(es[j].v)) ++cnt;
            }
            for (int j = i; j >= 0 && es[j].w == e.w; --j) {
                if (find(es[j].u) != find(es[j].v)) {
                    unite(find(es[j].u), find(es[j].v));
                    --cnt;
                }
            }
            ret += cnt;
            cnt = 0;
        }
    }
    return ret;
}
```

网络流

Edmonds-Karp

$O(VE^2)$

```cpp
// UVA 820
struct edge {
    int a, b, cap;
};
vector<edge> es;
vector<int> g[N];
int p[N]; // 每次增广的路径记录
int flows[N]; // 记录每次增广流大小顺便用于bfs (

inline void addedge(int a, int b, int cap) {
    es.push_back({a, b, cap});
    es.push_back({b, a, 0});
    int i = es.size();
    g[a].push_back(i - 2);
    g[b].push_back(i - 1);
}

int bfs(int s, int d) {
    memset(flows, 0, sizeof(flows));
    queue<int> q;
    q.push(s);
    flows[s] = INF;
    while (!q.empty()) {
        int u = q.front(); q.pop();
```

```cpp
        for (int ee = 0; ee < g[u].size(); ++ee) {
            const int &e = g[u][ee];
            const edge &eg = es[e];
            if (!flows[eg.b] && eg.cap > 0) {
                flows[eg.b] = min(flows[u], eg.cap);
                p[eg.b] = e;  // 记录此次增广的路径，用于更新
                if (eg.b == d) return flows[d];
                q.push(eg.b);
            }
        }
    }
    return flows[d];
}

int ekalgo(int s, int d) {
    int maxflow = 0;
    while (true) {
        if (!bfs(s, d)) break;
        for (int u = d; u != s; u = es[p[u]].a) {
            es[p[u]].cap -= flows[d];
            es[p[u]^1].cap += flows[d];
        }
        maxflow += flows[d];
    }
    return maxflow;
}
```

Scaling

$O(E^2 \log c)$ (c is the initial threshold)

```cpp
// UVA 820
struct edge {
    int a, b, cap;
};
vector<edge> es;
vector<int> g[N];
int p[N]; // 用于dfs的标识数组

inline void addedge(int a, int b, int cap) {
    es.push_back({a, b, cap});
    es.push_back({b, a, 0});
    int i = es.size();
    g[a].push_back(i - 2);
    g[b].push_back(i - 1);
}

int dfs(int u, int f) { // 每次只增广大小大于等于阈值的流，若无则返回0
    if (u == d) {
        return f;
    }
    p[u] = 1;
    for (auto ee : g[u]) {
        edge &e = es[ee];
```

```
        if (!p[e.b] && e.cap >= thre) {
            int flow = dfs(e.b, min(f, e.cap));
            if (flow >= thre) {
                e.cap -= flow;
                es[ee^1].cap += flow;
                return flow;
            }
        }
    }
    return 0;
}

int scal() {
    int maxflow = 0;
    thre = 1000;  // 阈值
    while (thre) {
        while (true) {
            memset(p, 0, sizeof(p));
            int flow = dfs(s, INF);
            if (!flow) break;
            maxflow += flow;
        }
        thre >>= 1;
    }
    return maxflow;
}
```

Dinic

$O(V^2E)$

```
int d[N];
bool bfs() {
    memset(d, -1, sizeof(d));
    queue<int> q;
    q.push(0);
    d[0] = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v = 1; v <= n; ++v) {
            if (d[v] < 0 && g[u][v] > 0) {
                d[v] = d[u] + 1;
                q.push(v);
            }
        }
    }
    return d[n] >= 0;
}

bool vis[N];
int dfs(int u, int f) {
    if (u == n) return f;
    int ans = 0;
    vis[u] = true;
```

```
    for (int v = 1; v <= n; ++v) {

        if (!vis[v] && d[v] == d[u] + 1 && g[u][v] > 0) {
            int flow = dfs(v, min(f, g[u][v]));
            if (flow > 0) {
                g[u][v] -= flow;
                g[v][u] += flow;
                ans += flow;
                f -= flow;
                if (f <= 0) break;
            }
        }
    }
    return ans;
}

int dinic() {
    int ans = 0;
    while (bfs()) {
        int flow;
        while (memset(vis, 0, sizeof(vis)), flow = dfs(0, INF)) ans += flow; // O(EV)
    }
    return ans;
}
```

上下界可行流

建一个特殊的图然后跑网络流（具体参见“图论总结 by amber.pdf”）

连通分量

强连通分量（Kosaraju）

$O(V+E)$

```
// POJ 2186: 分解强连通分量，如果图中任何一个点都能到最后一个连通分量，则输出最后一个连通分量所含节点数
int n, m;
vector<int> g[N];
vector<int> rev[N];
bool vis[N];
int s[N], stop; // 第一次dfs得到的dfs序列
int ord[N]; // 节点所在强连通分量的拓扑序

void dfs1(int u) {
    vis[u] = true;
    for (int i = 0; i < g[u].size(); ++i) {
        if (!vis[g[u][i]]) dfs1(g[u][i]);
    }
    s[stop++] = u;
}

void dfs2(int u, int k) {
    ord[u] = k;
```

```
    for (int i = 0; i < rev[u].size(); ++i) {
        if (!ord[rev[u][i]]) dfs2(rev[u][i], k);
    }
}

int cnt;
void check(int u) {
    vis[u] = true;
    ++cnt;
    for (int i = 0; i < rev[u].size(); ++i) {
        if (!vis[rev[u][i]]) check(rev[u][i]);
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    scanf("%d%d", &n, &m);
    while (m--) {
        int a, b;
        scanf("%d%d", &a, &b);
        g[a].push_back(b);
        rev[b].push_back(a);
    }
    for (int i = 1; i <= n; ++i)
        if (!vis[i]) dfs1(i);
    int k = 0;
    while (stop > 0) { // Kosaraju here
        int u = s[--stop];
        if (ord[u]) continue;
        dfs2(u, ++k); // 按栈中的顺序处理结点得到的是拓扑序
    }
    int ans = 0;
    int u = 0;
    for (int i = 1; i <= n; ++i) {
        if (ord[i] == k) { // 拓扑序中最后一个连通分量，即无出边的分量
            u = i;
            ++ans;
        }
    }
    memset(vis, 0, sizeof(vis));
    check(u); // 查看是否任意节点都可达该分量
    if (cnt != n) printf("0\n");
    else printf("%d\n", ans);
    return 0;
}
```

强连通分量（Tarjan）

$O(V+E)$

```
// POJ 2186: 分解强连通分量，如果图中任何一个点都能到最后一个连通分量，则输出最后一个连通分量所含节点数
#include <bits/stdc++.h>
```

```cpp
using namespace std;

const int INF = 0x3f3f3f3f;
const double EPS = 1e-8;
//const double PI = acos(-1.0);
//const int MOD = 1000000007;

typedef long long LL;
typedef unsigned long long uLL;
const int N = 10000+10;

int n, m;
vector<int> g[N];
int cnt; // dfs访问过程中的时间标记
int s[N], stop; // 存放还未被决定连通分量的节点，在栈中的节点一定有路径到当前访问的节点
int ord[N]; // 节点属于哪个连通分量，同时连通分量是拓扑有序的
bool ins[N]; // 是否在栈中
int dfn[N]; // 时间戳
int low[N]; // 各个节点能到达的时间戳最早的节点

int cpn;

void dfs(int u) { // 本质上就是一个模板 dfs 而已
    s[stop++] = u;
    ins[u] = true;
    dfn[u] = low[u] = ++cnt;
    for (int i = 0; i < g[u].size(); ++i) {
        if (!dfn[g[u][i]]) { // 如果该节点没有被访问
            dfs(g[u][i]);
            low[u] = min(low[u], low[g[u][i]]);
        }
        else if (ins[g[u][i]]) // 如果该节点在栈中
            low[u] = min(low[u], dfn[g[u][i]]);
    }
    if (dfn[u] == low[u]) { // 新连通分量，u是该连通分量的“根”
        ++cpn;
        int t; // 存放栈顶元素
        do {
            t = s[--stop];
            ins[t] = false;
            ord[t] = cpn;
        } while (t != u); // 栈顶到u的顶点都属于该连通分量
    }
}

int out[N];

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    scanf("%d%d", &n, &m);
```

```
    while (m--) {
        int a, b;
        scanf("%d%d", &a, &b);
        g[a].push_back(b);
    }
    for (int i = 1; i <= n; ++i) // Tarjan here
        if (!dfn[i]) dfs(i);
    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j < g[i].size(); ++j) {
            if (ord[i] != ord[g[i][j]]) ++out[ord[i]];
        }
    }
    int cnt = 0;
    for (int i = 1; i <= cpn; ++i)
        if (!out[i]) {
            ++cnt;
        }
    if (cnt > 1) printf("0\n");
    else {
        int ans = 0;
        for (int i = 1; i <= n; ++i)
            if (!out[ord[i]]) ++ans;
        printf("%d\n", ans);
    }
    return 0;
}
```

双连通分量（Tarjan）

性质：**点双连通图的所有节点都在一个环上，所有节点都在一个环上的图是双连通的**

$O(V+E)$

```
// POJ 2942
// 点双连通图就是没有任何一个单独的点可以成为割点的图
#include <bits/stdc++.h>

using namespace std;

const int INF = 0x3f3f3f3f;
const double EPS = 1e-8;
//const double PI = acos(-1.0);
//const int MOD = 1000000007;

typedef long long LL;
typedef unsigned long long uLL;
const int N = 1000+10;

int n, m;
vector<int> g[N];
bool hate[N][N];
int cnt; // dfs访问过程中的时间戳
int s[N], stop;
bool ins[N];
```

```cpp
int dfn[N]; // 每个节点的时间戳
int low[N]; // 每个节点不经过父节点能到达的最小节点的时间戳
int ord[N];
int cpn;

int tmp[N];
int color[N];
bool expelled[N];

inline void init() {
    for (int i = 1; i <= n; ++i) g[i].clear();
    memset(hate, 0, sizeof(hate));
    cnt = stop = cpn = 0;
    memset(ins, 0, sizeof(ins));
    memset(dfn, 0, sizeof(dfn));
    memset(ord, 0, sizeof(ord));
    memset(expelled, 1, sizeof(expelled));
}

bool bipartite(int u) { // 二分图判定
    for (int i = 0; i < g[u].size(); ++i) {
        int v = g[u][i];
        if (ord[v] == cpn) {
            if (!color[v] && !bipartite((color[v] = -color[u], v))) return false;
            if (color[v] && color[v] == color[u]) return false;
        }
    }
    return true;
}

void dfs(int u, int fa) { // 本质上就是一个模板 dfs 而已
    s[stop++] = u;
    ins[u] = true; // 点双连通分量所以存点
    dfn[u] = low[u] = ++cnt;
    for (int i = 0; i < g[u].size(); ++i) {
        int v = g[u][i];
        if (!dfn[v]) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);

            // 另: low[v] > dnf[u] 就说明V-U是桥
            if (low[v] >= dfn[u]) { // u是该双连通分量的“根”, 也是图中的一个割点
                ++cpn; // 新的双连通分量
                int cnt = 0;
                int t; // 存放栈顶元素
                do {
                    t = s[--stop];
                    ins[t] = false;
                    tmp[cnt++] = t; // 该数组中的元素为同一个连通分量中的点, 用于后面的二分图判断
                    ord[t] = cpn;
                } while (t != v);
                tmp[cnt++] = u; // u也属于该连通分量
                ord[u] = cpn; // 同一节点可能属于多个点双连通分量
```

```
                memset(color, 0, sizeof(color));
                color[u] = 1;
                if (cnt >= 3 && !bipartite(u)) { // 不是二分图即有奇圈
                    for (int i = 0; i < cnt; ++i) expelled[tmp[i]] = false;
                }
            }
        }
        else if (ins[v] && v != fa)
            low[u] = min(low[u], dfn[v]);
    }
}

int solve() {
    int ans = 0;
    for (int i = 1; i <= n; ++i) if (expelled[i]) ++ans;
    return ans;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    while (scanf("%d%d", &n, &m) && n) {
        init();
        for (int i = 0; i < m; ++i) {
            int a, b;
            scanf("%d%d", &a, &b);
            hate[a][b] = hate[b][a] = true;
        }
        for (int i = 1; i <= n; ++i) {
            for (int j = i + 1; j <= n; ++j) {
                if (!hate[i][j]) {
                    g[i].push_back(j);
                    g[j].push_back(i);
                }
            }
        }

        for (int i = 1; i <= n; ++i) // Tarjan here
            if (!dfn[i]) dfs(i, 0);

        printf("%d\n", solve());
    }
    return 0;
}
```

LCA

Tarjan 离线算法

$O((V+E)+(V+Q)\log V)$ (Q is the number of queries)

```cpp
// POJ 1470
// call: tarjan(root);
int fa[N];
bool vis[N];
vector<int> g[N];
vector<int> qs[N];
int ans[N];
int root;

int find(int u) {
    if (fa[u] < 0)
        return u;
    return fa[u] = find(fa[u]);
}

inline void setUnion(int a, int b) {
    int r1 = find(a), r2 = find(b);
    if (r1 == r2) return;
    fa[r1] = r2;
}

void tarjan(int u) {
    vis[u] = true;
    for (int i = 0; i < qs[u].size(); ++i) { // 处理该节点对应的查询
        int q = qs[u][i];
        if (vis[q]) ++ans[find(q)]; // 如果对应的节点已被访问，则其所在集合的代表就是lca
    }
    for (int i = 0; i < g[u].size(); ++i) { // dfs
        int ch = g[u][i];
        tarjan(ch);
        setUnion(ch, u); // 处理完一个儿子后就将其与父亲合并
    }
}

inline void init() {
    memset(fa, -1, sizeof(fa));
    memset(vis, 0, sizeof(vis));
    memset(ans, 0, sizeof(ans));
}
```

倍增法

init： $O(V \log V)$

getLCA： $O(\log V)$

```cpp
// 预处理后任意两点的LCA
int dep[N];
vector<int> g[N];
int ans[N];
int root;
int anc[20][N];
int n;
```

```cpp
void dfs(int u, int fa, int w) {
    anc[0][u] = fa;
    maxe[0][u] = w;
    for (int k = 1; k < LOGN; ++k) {
        if ((1<<k) > dep[u]) break;
        anc[k][u] = anc[k-1][anc[k-1][u]];
        maxe[k][u] = max(maxe[k-1][anc[k-1][u]], maxe[k-1][u]);
    }
    for (auto &e : spt[u]) {
        int v = e.first;
        if (v == fa) continue;
        dep[v] = dep[u] + 1;
        dfs(v, u, e.second);
    }
}
void init() { // 预处理
    dep[root] = 0;
    dfs(root, root, 0);
}
inline int getk(int u, int k) { // 得到节点u的第k个祖先
    if (k > dep[u]) return root;
    for (int i = 0; (1<<i) <= k; ++i) {
        if ((1<<i) & k) u = anc[i][u];
    }
    return u;
}
int getLCA(int u, int v) {
    if (dep[u] > dep[v]) swap(u, v);
    v = getk(v, dep[v] - dep[u]); // 先让v走到和u相同深度的节点

    // if (u == v) return u;
    // for (int k = LOG_V; k >= 0; --k) {
    //     if (anc[k][u] != anc[k][v]) {
    //         u = anc[k][u];
    //         v = anc[k][v];
    //     }
    // }
    // return anc[0][u];
    int l = 0, r = dep[u];
    while (l < r) {    // 二分搜索找到u、v最低的公共祖先
        int mid = l + (r - l) / 2;
        if (getk(u, mid) == getk(v, mid)) {
            r = mid;
        }
        else {
            l = mid + 1;
        }
    }
    return getk(u, l);
}
```

匹配与覆盖

占坑（

Floyd判圈算法

$O(n)$

```
// 求出环中有多少个节点
int n, a, b;
while (cin >> n && n) {
    cin >> a >> b;
    auto nxt = [&](LL x) {
        return (a * x % n * x + b) % n;
    };
    LL fast = 0, slow = 0;
    int cnt = 0;
    do {
        fast = nxt(nxt(fast));
        slow = nxt(slow);
    } while (fast != slow);
    do {
        slow = nxt(slow);
        ++cnt;
    } while (slow != fast);
    cout << n - cnt << '\n';
}
```

欧拉通路/回路

定义：图上每条边都经过且只经过一次。

存在性：

- 无向图：所有点连通且
  - 每个点的度都是偶数**（回路）**
  - 恰好两个点的度是奇数，其他都是偶数**（通路）**（奇数点即通路的起点与终点）
- 有向图：所有点连通且
  - 每个点的入度都等于出度**（回路）**
  - 一个点的入度比出度大 1（终点），另一个点的出度比入度大 1（起点），其他点入度等于出度**（通路）**

字符串算法

字符串哈希算法

$O(n)$

```
unsigned BKDRhash (char *str) {
    const unsigned seed = 131;  // 31 131 1313 13131 131313 etc..
    unsigned hash = 0;
    while (*str) {
        hash = hash * seed + (*str++);
```

```
    }
    return hash % hashSize;
}
unsigned DJBhash(const char str[]) {
    unsigned h = 5381;
    for (int i = 0; str[i]; ++i) {
        h += (h << 5) + (str[i]);
    }
    return h % hashSize;
}
```

Karp-Rabin

preprocess： $O(n)$

getval： $O(1)$

```
const int A = 911382323;
const int B = 972663749;
int h[N]; // h[k] contains the hash value of the prefix s[0...k]
int p[N]; // p[k] == Ak mod B

void preprocess(const char s[]) {
    int len = strlen(s);
    h[0] = s[0];
    p[0] = 1;
    for (int i = 1; i < len; ++i) {
        h[i] = ((LL)h[i-1] * A + s[i]) % B;
        p[i] = ((LL)p[i-1] * A) % B;
    }
}
int getval(int a, int b) { // 子串[a,b]的哈希值
    if (a == 0) return h[b];
    int tmp = (h[b] - ((LL)h[a - 1] * p[b - a + 1]) % B) % B;
    return (tmp < 0 ? tmp + B : tmp);
}
```

Z-algorithm

Z数组的使用：把模式串 P 加在被匹配串 S 前，中间隔一个无干扰字符，如 P#S ，再对这个字符串计算 Z-array，则 Z-array 中值为 P 的长度的位置即是模式串出现的位置。

$O(n)$

```
int z[N]; // z[i]为以位置i为起点的是s的前缀的最长子串的长度
void Z(const char s[]) { // compute z-array
    int len = strlen(s);
    int x = 0, y = 0;
    for (int i = 1; i < len; ++i) {
        z[i] = max(0, min(z[i-x], y - i + 1)); // 若i在[x, y]之外，则z[i]为0，否则为min(z[i-x], y - i + 1)
        while (i + z[i] < len && s[z[i]] == s[i + z[i]]) {
            x = i; y = i + z[i]; ++z[i];
        }
    }
}
```

KMP

KMP可用来求串的最大循环节。

$O(n)$

```
template<typename T>
vector<int> kmp(const T &a) { // fail[i]是为a[0...i]的后缀的最长前缀
    int n = a.size();
    vector<int> fail(n);
    fail[0] = -1;
    for (int i = 1; i < n; ++i) {
        fail[i] = fail[i - 1];
        while (~fail[i] && a[fail[i] + 1] != a[i]) {
            fail[i] = fail[fail[i]];
        }
        if (a[fail[i] + 1] == a[i]) {
            ++fail[i];
        }
    }
    return fail;
}
vector<int> KMP(const string &s, const string &p) {
    auto &&fail = kmp(p);
    vector<int> res;
    for (int i = 0, j = 0; i < s.size(); ++i) {
        while (s[i] != p[j] && j > 0) j = fail[j - 1] + (fail[j - 1] != j - 1);
        if (s[i] == p[j]) {
            if (++j == p.size()) {
                res.push_back(i - p.size() + 2);
                j = fail[j - 1] + (fail[j - 1] != j - 1);
            }
        }
    }
    return res;
}
```

AC自动机

AC自动机就是把pattern从一个字符串变成了一个trie树（即有多个pattern串，即字典）的KMP

建字典：$O(cS)$ (c is the size of the charset and S is the total length of all words in the trie)

匹配：$O(n+z)$ (n is the length of the original string and z is the total time that the patterns appear)

```
// HDU 2222 2896
// 失配后回溯到是字典中的某个词的前缀的后缀的位置
struct ACautomaton {
    int trie[N][26];
    int fail[N];
    int cnt[N];
    int tol = 0;
    int newnode() {
        for (int i = 0; i < 26; ++i) trie[tol][i] = 0;
        fail[tol] = 0;
        cnt[tol] = 0;
        return tol++;
    }
    void init() {
        tol = 0;
        newnode();
    }
    void add(const char s[]) { // 在字典中新加入一个词
        int cur = 0;
        for (int i = 0; s[i]; ++i) {
            int c = s[i] - 'a';
            if (trie[cur][c] == 0) trie[cur][c] = newnode();
            cur = trie[cur][c];
        }
        ++cnt[cur];
    }
    void build() { // bfs
        queue<int> q;
        q.push(0);
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (int i = 0; i < 26; ++i) {
                int &v = trie[u][i];
                if (v) {
                    if (u != 0) fail[v] = trie[fail[u]][i]; // 非根节点失配才回溯
                    // 如果trie[fail[u]][i]存在，即该位置就是fail节点v的失配指针
                    // 如果不存在，则它保存着节点fail[u]失配时的失配指针
                    q.push(v);
                }
                else { // 不存在
                    v = trie[fail[u]][i]; // 保存u的失配指针
                }
            }
        }
    }
    int count(const char s[]) { // 返回字典的词在s中总共出现了多少次（有重复）
        int ans = 0;
        int u = 0;
        for (int i = 0; s[i]; ++i) {
            int c = s[i] - 'a';
```

```
            u = trie[u][c];
            int p = u;
            while (p && cnt[p]) { // 后缀
                ans += cnt[p];
                cnt[p] = 0;
                p = fail[p];
            }
        }
        return ans;
    }
} ac;
```

后缀数组

LCP：Longest Common Prefix

很大。后缀数组也一样，它的含义是S的各个后缀的排列SA，它满足：

$$Suffix(SA[i]) < Suffix(SA[i+1]) \quad (1 \le i < n)$$

注意这里不用等号，任何两个后缀的长度不同，因此不可能相等。为了方便，我们另外定义一个名次数组Rank=SA^{-1}，即若SA[i]=j，则Rank[j]=i，因此Rank[i]保存的是后缀Suffix(i)在所有后缀中从小到大的“名次”。

倍增法构造：$O(n \log n)$

计算LCP：$O(n)$

LCP定理：设$i < j$，则$LCP(i,j) = \min\{LCP(k-1,k) \mid i+1 \le k \le j\}$

```
// ACM-ICPC 2018 焦作赛区网络预赛 H：子串出现次数
struct SuffixArray {
    int n;
    int arr[N];
    int sa[N]; // 1 ~ n
    int rank[N]; // 0 ~ n - 1
    int lcp[N]; // LCP between sa[i] and sa[i-1], 1 ~ n
    int t1[N], t2[N], cnt[N];

    void init (const string &s) {
        n = 0;
        for (auto i : s) arr[n++] = i;
        arr[n++] = 0;
        build_sa(127);
//        build();
    }

    // m为最大字符集数加1
    void build_sa(int m) {
        int *x = t1, *y = t2;
        for (int i = 0; i < m; ++i) cnt[i] = 0;
        for (int i = 0; i < n; ++i) ++cnt[x[i] = arr[i]];
```

```
        for (int i = 1; i < m; ++i) cnt[i] += cnt[i-1];
        for (int i = n-1; i >= 0; --i) sa[--cnt[x[i]]] = i;
        for (int k = 1; k <= n; k <<= 1) {
            int p = 0;

            // 按照次关键字进行排序
            for (int i = n - k; i < n; i++) y[p++] = i;
            for (int i = 0; i < n; i++) if (sa[i] >= k) y[p++] = sa[i] - k;

            // 按照主关键字进行排序
            for (int i = 0; i < m; i++) cnt[i] = 0;
            for (int i = 0; i < n; i++) ++cnt[x[y[i]]];
            for (int i = 1; i < m; i++) cnt[i] += cnt[i - 1];
            for (int i = n - 1; i >= 0; i--) sa[--cnt[x[y[i]]]] = y[i];
            swap(x, y);

            // 计算rank
            p = 1;
            x[sa[0]] = 0;
            for (int i = 1; i < n; i++)
                x[sa[i]] = (y[sa[i - 1]] == y[sa[i]] && y[sa[i - 1] + k] == y[sa[i] + k]
                            ? p - 1 : p++);
            if (p >= n) break;
            m = p;
        }
        for (int i = 0; i < n; ++i) rank[sa[i]] = i;
    }

    void build_lcp() {
        int k = 0;
        for(int i = 0; i < n; i++) {
            if(k) k--;
            if(rank[i] - 1 < 0) continue; // rank[i]可能是0
            int j = sa[rank[i]-1];
            while(arr[i+k] == arr[j+k]) k++;
            lcp[rank[i]] = k;
        }
    }

    // 所有height值之和为重复子串的个数
    // 同一子串会在后缀数组中连续出现
} sa;
```

区间问题

Sparse Table（稀疏表）

构建：$O(n\log n)$

查询：$O(1)$

```cpp
int minima[17][N];
int a[N];
int n;
void init() {
    for (int i = 0; i < n; ++i) {
        minima[0][i] = a[i];
    }
    for (int i = 1; (1<<i) <= n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (j + (1<<i) > n) break;
            minima[i][j] = min(minima[i-1][j], minima[i-1][j+(1<<(i-1))]);
        }
    }
}
inline int getmin(int a, int b) { // return minimum in [a, b]
    int len = b - a + 1;
    int k = 0;
    while ((1<<(k+1)) <= len) ++k;
    return min(minima[k][a], minima[k][b-(1<<k)+1]);
}
```

树状数组

建树： $O(n)$

查询、修改： $O(\log n)$

```cpp
// 求区间和
int binary_indexed_tree[N]; // [0] is redundant
// tree[i]'s length is the largest power of two that divides i and that ends at position i.
int n;
int arr[N]; // original array

int sum(int k) { // get sum of arr[1, k]
    int s = 0;
    while (k >= 1) {
        s += binary_indexed_tree[k];
        k -= k & -k; // lowest 1-bit, the largest power of 2 that divides k
    }
    return s;
}
inline void add(int k, int v) {
    while (k <= n) {
        binary_indexed_tree[k] += v;
        k += k & -k;
    }
}
inline void init() {
    for (int i = 1; i <= n; ++i) add(i, arr[i]);
}

int inversion() { // 求逆序数（权值树状数组）
    int ans = 0;
```

```
    for (int i = 0; i < n; ++i) {
        ans += i - sum(arr[i]);
        add(arr[i], 1);
    }
    return ans;
}
```

线段树

建树： $O(n)$

查询、修改： $O(\log n)$

```
int segTree[N<<2];
int n;

/* --- from bottom to top --- */
// minimum query
int getMin(int a, int b) {
    a += n; b += n;
    int ans = INF;
    while (a <= b) {
        if (a & 1) ans = min(ans, segTree[a++]);
        if (!(b & 1)) ans = min(ans, segTree[b--]);
        a >>= 1; b >>= 1;
    }
    return ans;
}
void modifyMin(int k, int v) { // modified arr[k] value
    k += n;
    segTree[k] = v;
    for (k >>= 1; k >= 1; k >>= 1) {
        segTree[k] = min(segTree[2 * k], segTree[2 * k + 1]);
    }
}

/* -- from top to bottom --- */
// range sum update and query
inline void pushdown(int k, int x, int y) {
    int inc = segTree[k].inc;
    if (inc) {
        segTree[k].v += (y - x + 1) * inc;
        segTree[k].inc = 0;
        segTree[k << 1].inc += inc;
        segTree[k << 1 | 1].inc += inc;
    }
}
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;

    if (a <= x && b >= y) return segTree[k].v + segTree[k].inc * (y - x + 1);
    pushdown(k, x, y);
    int half = (x + y) >> 1;
```

```
    return sum(a, b, k << 1, x, half) + sum(a, b, k << 1 | 1, half + 1, y);
}
void add(int a, int b, int k, int x, int y, int v) {
    if (b < x || a > y) return;

    if (a <= x && b >= y) {
        segTree[k].inc += v; // 将该区间内的元素都加v
        return;
    }
    pushdown(k, x, y);
    int half = (x + y) >> 1;
    int len = b - a + 1; // 相交区间的长度 (min(y, b) - max(x, a) + 1)
    if (a < x && b >= x) len = b - x + 1;
    if (b > y && a <= y) len = y - a + 1;
    segTree[k].v += len * v;
    add(a, b, k << 1, x, half, v); add(a, b, k << 1 | 1, half + 1, y, v);
}

// rmq
int tree[N<<2];
int lazy[N<<2];
int a[N];
int n, m;

inline int lch(int k) { return k << 1; }
inline int rch(int k) { return k << 1 | 1; }

void build(int k, int x, int y) {
    if (x == y) {
        tree[k] = a[x];
        return;
    }
    int h = (x + y) / 2;
    build(lch(k), x, h);
    build(rch(k), h + 1, y);
    tree[k] = min(tree[lch(k)], tree[rch(k)]);
}

void pushdown(int k) {
    tree[lch(k)] += lazy[k];
    tree[rch(k)] += lazy[k];
    lazy[lch(k)] += lazy[k];
    lazy[rch(k)] += lazy[k];
    lazy[k] = 0;
}

void add(int a, int b, int k, int x, int y, int v) {
    if (a <= x && b >= y) {
        tree[k] += v;
        lazy[k] += v;
        return;
    }
    pushdown(k);
```

```
    int h = (x + y) / 2;
    if (a <= h) add(a, b, lch(k), x, h, v);
    if (b > h) add(a, b, rch(k), h + 1, y, v);
    tree[k] = min(tree[lch(k)], tree[rch(k)]);
}

int rmq(int a, int b, int k, int x, int y) {
    if (a <= x && b >= y) {
        return tree[k];
    }
    pushdown(k);
    int h = (x + y) / 2;
    int ret = INF;
    if (a <= h) ret = min(ret, rmq(a, b, lch(k), x, h));
    if (b > h) ret = min(ret, rmq(a, b, rch(k), h + 1, y));
    return ret;
}
```

主席树

维护着区间$[1,i] (i \in [1,n])$的信息（一段与区间有关的信息（如权值线段树什么的），主席树本质上就是n棵权值线段树，不过利用了重复信息压缩空间而已）

适用主席树的题目：满足与树状数组相似的区间减法

时间：与原生线段树相同

空间：每次修改增加$O(\log n)$的空间

```
// HDU 2665: 求区间第k大的值
#include <bits/stdc++.h>

using namespace std;

const int INF = 0x3f3f3f3f;
const double EPS = 1e-8;
// const double PI = acos(-1.0);
// const int MOD = 1000000007;

typedef long long LL;
typedef unsigned long long uLL;
const int N = 100000 + 10;
const int M = 100000 + 10;

struct node {
    int l, r; // pointers to l & r subtree
    int cnt;
    node() : l(-1), r(-1), cnt(0) {}
} pool[N * 20]; // 节点静态内存池

int n, m;
int w[N]; // 原序列的值
pair<int, int> ans[N]; // 离散后的值对原值的映射
int cnt; // 内存池的指针
```

```cpp
int roots[N]; // 可持久化线段树

int query(int l, int r, int x, int y, int k) {  // x, y 是权值
    if (x == y) return x;
    int dif = (pool[pool[r].l].cnt - pool[pool[l].l].cnt); // 左节点元素个数差
    int h = (x + y) >> 1;
    if (k <= dif) return query(pool[l].l, pool[r].l, x, h, k); // 说明第k小的值在左子树
    return query(pool[l].r, pool[r].r, h + 1, y, k - dif); // 否则在右子树，此时k要减去dif
}
void add(int s, int t, int x, int y, int v) {
    // s代表上一版本的节点，t代表当前版本的节点
    pool[t] = pool[s]; // 复制上一版本的节点
    ++pool[t].cnt; // 修改
    if (x == y) return;
    int h = (x + y) >> 1;
    if (v <= h)
        add(pool[s].l, pool[t].l = ++cnt, x, h, v); // 要修改的值属于左子树，更新该版本的节点的左子树
    else
        add(pool[s].r, pool[t].r = ++cnt, h + 1, y, v);
}

void build(int u, int x, int y) {
    pool[u] = node();
    if (x == y) return;
    int h = (x + y) >> 1;
    build(pool[u].l = ++cnt, x, h);
    build(pool[u].r = ++cnt, h + 1, y);
}
void init() {
    build(roots[0] = cnt = 0, 1, n);
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int t;
    cin >> t;
    while (t--) {
        cin >> n >> m;
        init();
        for (int i = 1; i <= n; ++i) {
            cin >> w[i];
            ans[i].first = w[i];
            ans[i].second = i;
        }
        sort(ans + 1, ans + 1 + n); // 排序好后下标即为离散后的值，first即为原值
        for (int i = 1; i <= n; ++i) w[ans[i].second] = i; // 原序列值改为离散后的值
        for (int i = 1; i <= n; ++i) {
            add(roots[i - 1], roots[i] = ++cnt, 1, n, w[i]);
        }
        for (int i = 0; i < m; ++i) {
            int s, t, k;
```

```
            cin >> s >> t >> k;
            cout << ans[query(roots[s - 1], roots[t], 1, n, k)].first << '\n';
        }
    }
    return 0;
}
```

莫队算法

莫队算法学习笔记 | Sengxian's Blog

普通莫队

离线算法，将询问分为$\sqrt{n}$个块后，按照$(\lfloor \frac{l}{\sqrt{n}} \rfloor, r)$进行排序，然后依序处理。

$O(n\sqrt{n})$

```
// 树上莫队 SPOJ COT2
#include <bits/stdc++.h>

using namespace std;

const int INF = 0x3f3f3f3f;
const double EPS = 1e-8;
//const double PI = acos(-1.0);
//const int MOD = 1000000007;

typedef long long LL;
typedef unsigned long long uLL;
const int N = 10000*4+10;
const int M = 100000+10;

int n, m;
vector<int> g[N];
int stk[N], sz; // 分块所用辅助栈
int block_size, block_cnt;
int w[N]; // 节点离散化后的权值
int bel[N]; // 每个节点所属的块
int dep[N]; // 节点深度
int pa[N]; // 节点老爸
struct qry{
    int u, v, id;
    bool operator< (const qry &b) const {
        return bel[u] < bel[b.u] || (bel[u] == bel[b.u] && bel[v] < bel[b.v]);
    }
} query[M];
int ans[M];
bool vis[N]; // 节点是否在查询集合中
int nowAns = 0; // 每一轮的查询结果
int wcnt[N]; // 每个权值的计数

int anc[N][20]; // LCA
```

```
void LCAinit() {
    for (int u = 1; u <= n; ++u) anc[u][0] = pa[u];
    bool ok = false;
    for (int k = 1; !ok; ++k) { // 初始化anc表
        ok = true;
        for (int u = 1; u <= n; ++u) {
            if ((1<<k) > dep[u]) continue;
            anc[u][k] = anc[anc[u][k-1]][k-1];
            ok = false;
        }
    }
}

inline int getk(int u, int k) { // 得到节点u的第k个祖先
    if (k > dep[u]) return 1;
    for (int i = 0; (1<<i) <= k; ++i) {
        if ((1<<i) & k) u = anc[u][i];
    }
    return u;
}
int getLCA(int u, int v) {
    if (dep[u] > dep[v]) swap(u, v);
    v = getk(v, dep[v] - dep[u]); // 先让v走到和u相同深度的节点

    int l = 0, r = dep[u];
    while (l < r) {    // 二分搜索找到u、v最低的公共祖先
        int mid = l + (r - l) / 2;
        if (getk(u, mid) == getk(v, mid)) {
            r = mid;
        }
        else {
            l = mid + 1;
        }
    }
    return getk(u, l);
}

void dfs(int u, int fa) { // 分块
    pa[u] = fa;
    dep[u] = dep[fa] + 1;
    int bottom = sz;
    for (int i = 0; i < (int)g[u].size(); ++i) {
        int v = g[u][i];
        if (v == fa) continue;
        dfs(v, u);
        if (sz - bottom >= block_size) {
            ++block_cnt;
            while (sz != bottom) {
                bel[stk[--sz]] = block_cnt;
            }
        }
    }
    stk[sz++] = u;
```

```
}

void discrete(vector<pair<int, int>> &vs) { // 离散化
    sort(vs.begin(), vs.end());
    w[vs[0].second] = 1;
    for (int i = 1; i < vs.size(); ++i) {
        if (vs[i].first == vs[i-1].first) w[vs[i].second] = w[vs[i-1].second];
        else w[vs[i].second] = i + 1;
    }
}

void flip(int u) { // “异或”操作
    if (vis[u]) {
        if (--wcnt[w[u]] == 0) --nowAns;
        vis[u] = false;
    }
    else {
        if (++wcnt[w[u]] == 1) ++nowAns;
        vis[u] = true;
    }
}

void moveto(int u, int v) { // 节点 u 向节点 v 转移
    if (dep[u] < dep[v]) swap(u, v);
    while (dep[u] > dep[v]) {
        flip(u);
        u = pa[u];
    }
    while (u != v) {
        flip(u); u = pa[u];
        flip(v); v = pa[v];
    }
}

void solve() {
    LCAinit();

    int u = 1, v = 1;
    int LCA = 1;
    flip(1);
    for (int i = 0; i < m; ++i) {
        const qry &q = query[i];
        flip(LCA);
        moveto(u, q.u); moveto(v, q.v); // 转移区间
        flip(LCA = getLCA(q.u, q.v));
        ans[q.id] = nowAns;
        u = q.u; v = q.v;
    }

    for (int i = 0; i < m; ++i) printf("%d\n", ans[i]);
}

int main() {
```

```cpp
    ios::sync_with_stdio(false);
    cin.tie(0);

    vector<pair<int, int>> vs;
    scanf("%d%d", &n, &m);
    block_size = int(ceil(sqrt(n)));
    for (int i = 1; i <= n; ++i) {
        int w;
        scanf("%d", &w);
        vs.emplace_back(w, i);
    }
    discrete(vs);

    for (int i = 1; i < n; ++i) {
        int u, v;
        scanf("%d%d", &u, &v);
        g[u].push_back(v);
        g[v].push_back(u);
    }
    for (int i = 0; i < m; ++i) {
        scanf("%d%d", &query[i].u, &query[i].v);
        query[i].id = i;
    }

    dep[0] = -1;
    dfs(1, 0); // 分块
    while (sz) bel[stk[--sz]] = block_cnt; // 分块操作的最后一块
    sort(query, query + m);

    solve();
    return 0;
}
```

带修改莫队

将在线变为离线，按照$(\lfloor \frac{l}{n^{\frac{2}{3}}} \rfloor, \lfloor \frac{r}{n^{\frac{2}{3}}} \rfloor, t)$三元组从小到大排序，$t$表示这个询问之前经过了多少次修改。

$O(n^{\frac{5}{3}})$

```cpp
// BZOJ 2120
void udTime(int x, int op) {
    int pos = ms[x].a, val = ms[x].b, pre = ms[x].p;
    if (op > 0) {
        if (pos <= R && pos >= L) {
            if (--cnt[a[pos]] == 0) --now;
            if (++cnt[val] == 1) ++now;
        }
        a[pos] = val;
    } else {
        if (pos <= R && pos >= L) {
            if (--cnt[a[pos]] == 0) --now;
            if (++cnt[pre] == 1) ++now;
        }
```

```
        a[pos] = pre;
    }
}

void update(int x, int op) {
    if (x == 0) return;
    cnt[a[x]] += op;
    if (cnt[a[x]] == 0 && op < 0) --now;
    if (cnt[a[x]] == 1 && op > 0) ++now;
}

void moveto(int l, int r, int t) {
    while (T < t) udTime(++T, 1);
    while (T > t) udTime(T--, -1);
    while (L < l) update(L++, -1);
    while (L > l) update(--L, 1);
    while (R < r) update(++R, 1);
    while (R > r) update(R--, -1);
}

void solve() {
    S = pow(n, 2.0 / 3);
    sort(qs, qs + qn);
    now = 0;
    L = R = 0;
    T = mn;
    for (int i = 0; i < qn; ++i) {
        const qry &q = qs[i];
        moveto(q.l, q.r, q.t);
        ans[q.i] = now;
    }
}
```

其他

枚举技巧

枚举排列

```
// 比一般的生成法稍快，但生成的排列不按字典序
int num[N] = {1,2,3,4,5,6,7,8,9};
void go(int i) {
    // 函数开始时会得到一个新排列

    for (; i < N; ++i) {
        for (int j = i + 1; j < N; ++j) {
            swap(num[i], num[j]);
            go(i + 1);
            swap(num[i], num[j]);
        }
    }
}
```

枚举子集

```
inline void genSub(int s) { // s可为任意的集合
    for (int i = s; i; i = (i - 1) & s) {
        // i 即为一个子集
    }
}
inline void genRsub(int n, int r) {
    // n 为原始集合大小，枚举大小为 r 的子集
    for (int s = (1 << r) - 1; s < (1 << n); ) {
        // s 即为一个子集

        int x = s & -s, y = s + x;
        int t = s & ~y;
        s = ((t / x) >> 1) | y;
        // s = (t >> (__builtin_ctz(t) + 1)) | y; // optimization
    }
}
```

位运算魔法

```
inline bool isPowerOf2(unsigned n) {
    return (n & (n-1)) == 0; // n & (n-1) sets the last 1 bit of n to 0
}
inline int getLargestPowerOf2(unsigned v) {
    v |= v >> 1;
    v |= v >> 2;
    v |= v >> 4;
    v |= v >> 8;
    v |= v >> 16;
    return v ^ (v>>1);
}
inline int getRoundUpPowerOf2(unsigned v) {
    v--;
    v |= v >> 1;
    v |= v >> 2;
    v |= v >> 4;
    v |= v >> 8;
```

```
    v |= v >> 16;
    v++;
    return v;
}
```

并查集

$O(\log n)$

```
// 按秩求并的并查集
int find(int u) {
    if (fa[u] < 0) // 只有一个节点时高度为-1，因为有可能节点编号从0开始
        return u;
    return fa[u] = find(fa[u]);
}

inline void setUnion(int a, int b) {
    int r1 = find(a), r2 = find(b);
    if (r1 == r2) return;
    if (fa[r1] < fa[r2]) {
        fa[r2] = r1;
        return;
    }
    if (fa[r1] == fa[r2]) {
        --fa[r2];
    }
    fa[r1] = r2;
}
```

表达式树

```
// ACM-ICPC 2018 沈阳赛区网络预赛 B
struct node {
    int op;
    int l, r;
} tree[N];
int cnt;

int build(string &s, int x, int y) {
    bool ok = true;
    for (int i = x; i < y; ++i)
        if (!isdigit(s[i])) {
            ok = false;
            break;
        }
    if (ok) {
        auto &u = tree[++cnt];
        u.l = u.r = 0;
        u.op = stoi(s.substr(x, y - x));
        return cnt;
    }
    int p1 = -1, p2 = -1, p3 = -1, p = 0;
```

```cpp
    for (int i = x; i < y; ++i) {
        switch (s[i]) {
            case '(':
                p++;
                break;
            case ')':
                p--;
                break;
            case '+':
            case '-':
                if (!p) p1 = i;
                break;
            case '*':
                if (!p) p2 = i;
                break;
            case 'd':
                if (!p && p3 < 0) p3 = i;
                break;
        }
    }
    if (p1 < 0) p1 = p2; // 没有括号外的加减号
    if (p1 < 0) p1 = p3; // 也没有乘号
    if (p1 < 0) return build(s, x + 1, y - 1); // 也没有d，则整个表达式被括号包着
    int tmp = ++cnt;
    auto &u = tree[tmp];
    u.l = build(s, x, p1);
    u.r = build(s, p1 + 1, y);
    u.op = s[p1];
    return tmp;
}
```

整体二分

```cpp
// BZOJ 3110: 求区间第K大，支持插入（修改），查询。
// 对于每个查询，二分之前的修改（限定修改的值的范围，若不在范围内则不执行该修改）
// 由于需要对每个查询进行二分，所以需要依二分结果将操作分成两类
void solve(int ql, int qr, int L, int R) {
    if (ql > qr) return;
    if (L == R) {
        for (int i = ql; i <= qr; ++i) {
            if (qs[i].op == 2) ans[qs[i].id] = L;
        }
        return;
    }
    int t1 = 0, t2 = 0, mid = L + (R - L) / 2;
    for (int i = ql; i <= qr; ++i) {
        if (qs[i].op == 1) {
            if (qs[i].c <= mid)
                q1[t1++] = qs[i];
            else
                q2[t2++] = qs[i],
                add(qs[i].a, qs[i].b, 1, -n, n, 1);  // interval add
        } else {
```

```
            LL tmp = sum(qs[i].a, qs[i].b, 1, -n, n);  // interval sum
            if (tmp < qs[i].c)
                qs[i].c -= tmp, q1[t1++] = qs[i];
            else
                q2[t2++] = qs[i];
        }
    }

    // sort queries and clear tree
    int qmid = ql + t1;
    for (int i = ql; i < qmid; ++i) {
        qs[i] = q1[i - ql];
        if (qs[i].op == 1 && qs[i].c > mid) add(qs[i].a, qs[i].b, 1, -n, n, -1);
    }
    for (int i = qmid; i <= qr; ++i) {
        qs[i] = q2[i - qmid];
        if (qs[i].op == 1 && qs[i].c > mid) add(qs[i].a, qs[i].b, 1, -n, n, -1);
    }
    solve(ql, qmid - 1, L, mid);
    solve(qmid, qr, mid + 1, R);
}
```

一些杂七杂八的东西

1. 某个元素在冒泡排序中被交换的次数等于其在逆序对中出现的次数。
2. $\lceil \frac{a}{b} \rceil = \frac{a-1}{b} + 1$

.vimrc

```
syntax on
set number " line number
set cursorline " show line cursor
set shiftwidth=4
set softtabstop=4
set tabstop=4
set expandtab " change tab to blank
set autoindent
set smartindent
set cindent
set showmatch " show match braces
set matchtime=3 " time for vim to wait for showmatch
set ruler

nnoremap <F4> :w <CR> :!g++ % -o %< --std=c++14 -Wall -Wshadow -g -fsanitize=address -
fsanitize=undefined && for i in ./in*; do echo $i; ./%< < $i; done <CR>
```

debug template

```
#ifdef LOCAL
template <typename T>
auto is_printable_impl(int)
    -> decltype(cout << declval<T &>(), std::true_type{});
template <typename T>
```

```
std::false_type is_printable_impl(...);
template <typename T>
using is_printable = decltype(is_printable_impl<T>(0));

template <typename Tuple, size_t N>
struct TuplePrinter;
struct debug {
    template <typename T>
    typename enable_if<!is_printable<T>::value, debug &>::type operator<<(
        const T &x) {
        int i = 0;
        for (auto it = begin(x); it != end(x); ++it)
            *this << "[" << i++ << ": " << *it << "] ";
        return *this;
    }
    template <typename T>
    typename enable_if<is_printable<T>::value, debug &>::type operator<<(
        const T &x) {
        cout << x;
        return *this;
    }
    template <typename T1, typename T2>  // pair-printer
    debug &operator<<(const pair<T1, T2> &p) {
        *this << "{" << p.first << ", " << p.second << "}";
        return *this;
    }
    template <typename... Args>
    debug &operator<<(const tuple<Args...> &t) {
        cout << "(";
        TuplePrinter<decltype(t), sizeof...(Args) - 1>::print(t);
        cout << ")";
        return *this;
    }
    ~debug() { cout << endl; }
#else
struct debug {
    template <typename T>
    debug &operator<<(const T &foo) {
        return *this;
    }
#endif
};
template <typename Tuple, size_t N>
struct TuplePrinter {
    static void print(const Tuple &t) {
        TuplePrinter<Tuple, N - 1>::print(t);
        debug() << get<N>(t);
    }
};
template <typename Tuple>
struct TuplePrinter<Tuple, 0> {
    static void print(const Tuple &t) { debug() << get<0>(t); }
};
```

```
#define name(x) "[" #x ": " << x << "] "
template <typename T>
void print2d(const vector<vector<T>> &v) {
#ifdef LOCAL
    for (int i = 0; i < v.size(); ++i) {
        cout << i << ": ";
        debug() << name(v[i]);
    }
#endif
}
```

对拍相关

随机数生成

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
int r(int s, int e) { return uniform_int_distribution<int>(s, e)(rng); }
```

脚本

```
// 对拍
@echo off
:loop
    data > input.txt
    main < input.txt > output.txt
    test < input.txt > answer.txt
    fc output.txt answer.txt > nul
if not errorlevel 1 goto loop

// checker
@echo off
:loop
    data > input.txt
    main < input.txt > output.txt
    test < output.txt
if not errorlevel 1 goto loop

while true; do
    ./gen > in
    diff -w <(./sol < in) <(./test < in) || break
done
```

树数据生成

```
int main() {
    int n = r(2, 20);
    printf("%d\n", n);
    for(int i = 2; i <= n; ++i) {
        printf("%d %d\n", r(1, i - 1), i); // 前者是后者的父节点
    }
    return 0;
}
```