对它做适当的变形，就得到了

$$T = N \times Z_{a_1} (N-1) \times Z_{a_2} + \cdots + 1 \times Z_{a_N}$$

从该式中应该可以发现，当只有一个工厂时，为了最小化总时间，应该从花费时间较少的玩具开始加工。

那么，当有多个工厂时又该如何呢？因为只有一个工厂时可以用贪心法求解，如果我们已经确定了各个玩具应该在哪个工厂加工的话，就能够计算对应的最小总时间了。但是，本题中的N高达50，不可能枚举所有的分配方案。于是我们再回顾一下只有一个工厂时的最小总时间T的表达式。

$$T = N \times Z_{a_1} (N-1) \times Z_{a_2} + \cdots + 1 \times Z_{a_N}$$

这个式子除了可以看成是制作多个玩具的一个工厂外，还可以看成是多个只能制作一个玩具的工厂，只不过它们各自需要花费1倍到N倍的时间。这样来想的话，这道题也不过是普通的指派问题而已，用最小费用流就能解决了。

```
// 输入
int N, M;
int Z[MAX_N][MAX_M];

void solve() {
  // 0~N-1: 玩具
  // N~2N-1: 0号工厂
  // 2N~3N-1: 1号工厂
  // ...
  // MN~(M+1)N-1: M-1号工厂
  int s = N + N * M, t = s + 1;
  V = t + 1;
  for (int i = 0; i < N; i++) {
    add_edge(s, i, 1, 0);
  }
  for (int j = 0; j < M; j++) {
    for (int k = 0; k < N; k++) {
      add_edge(N + j * N + k, t, 1, 0);
      for (int i = 0; i < N; i++) {
        add_edge(i, N + j * N + k, 1, (k + 1) * Z[i][j]);
      }
    }
  }

  printf("%.6f\n", (double) min_cost_flow(s, t, N) / N);
}
```

Intervals （POJ No.3680）

给定 N 个带权的开区间。i 号区间覆盖(a_i, b_i)，权重为 w_i。现在要从中选取一些区间，要求任意点都不被超过 K 个区间覆盖，目标是最大化总的权重。

限制条件

- $1 \leqslant K \leqslant N \leqslant 200$
- $1 \leqslant a_i < b_i \leqslant 100000$
- $1 \leqslant w_i \leqslant 100000$

样例 1

输入

```
N = 3
K = 1
(a, b, w) = {(1, 2, 2), (2, 3, 4), (3, 4, 8)}
```

输出

```
14 (选择所有区间)
```

样例 2

输入

```
N = 3
K = 1
(a, b, w) = {(1, 3, 2), (2, 3, 4), (3, 4, 8)}
```

输出

```
12 (选取2号和3号区间)
```

样例 3

输入

```
N = 3
K = 2
(a, b, w) = {(1, 100000, 100000), (1, 150, 301), (100, 200, 300)}
```

输出

```
100301 (选取1号和2号区间)
```

首先不妨考虑一下K=1的情况。此时问题等价于从这N个区间中选取一个元素互不相交的子集，目标是最大化子集元素的权重和。这个问题又被称为区间图的最大权独立集问题，可以用如下的DP算法求解。

首先，对所有区间的端点排序得到一个x_i的数组。令

$dp[i]$:=只考虑$b_k \leq x_i$的区间所能得到的最大总权重

则有

$$dp[i] = max(dp[i-1], max\{dp[j]+w_k \mid a_k=x_j \text{ 且 } b_k=x_i\}$$

那么，了解了K=1时的解法，让我们参考它再看一下K>1时的解法。任意点都不被超过K个区间覆盖就等价于可以把答案划分为K个互不相交的区间组成的子集。那么，利用K=1的情况中所用的DP，每次求得最优解，再将选中的区间删去，对剩余部分采取同样处理重复K次，这个方法可行么？这显然是不行的，应该很容易就能找到反例。不妨再来仔细看一下刚才的DP递推式。它可以看作是在求解如下所建的图中的最短路问题。

- 给m个端点x_i建立对应的顶点v_i
- 从v_{i-1}向v_i连一条费用为0的边，对区间k，如果$a_k=x_j$且$b_k=x_i$，则从v_j向v_i连一条费用为$-w_k$的边。

而K=1时DP所得到的最大总权重，就是该图中从v_0到v_{m-1}的最短路的费用的相反数。

在该图中沿着权重为$-w_i$的边增广就对应于选中区间w_i。因此，令这类边的容量为1，而其余边的容量为∞，则一个流量为K的v_0–v_{m-1}流就对应原题所要求的K个子集。因此，利用最小费用流，原问题也就迎刃而解了。不过需要注意的是，本题的图中含有负权边。这里我们用上一小节的专栏中所介绍的技术，先令所有负权边初始都满流来处理负权边的情况。

样例2对应的图

```
// 输入
int N, K;
int a[MAX_N], b[MAX_N], w[MAX_N];

void solve() {
  // 预处理端点集合
  vector<int> x;
  for (int i = 0; i < N; i++) {
    x.push_back(a[i]);
    x.push_back(b[i]);
```

```
  }
  sort(x.begin(), x.end());
  x.erase(unique(x.begin(), x.end()), x.end());

  // 建图
  int m = x.size();
  int s = m, t = s + 1;
  V = t + 1;
  int res = 0;
  add_edge(s, 0, K, 0);
  add_edge(m - 1, t, K, 0);
  for (int i = 0; i + 1 < m; i++) {
    add_edge(i, i + 1, INF, 0);
  }
  for (int i = 0; i < N; i++) {
    int u = find(x.begin(), x.end(), a[i]) - x.begin();
    int v = find(x.begin(), x.end(), b[i]) - x.begin();
    // 从u向v连一条容量为1、费用为-w[i]的边
    add_edge(v, u, 1, w[i]);
    add_edge(s, v, 1, 0);
    add_edge(u, t, 1, 0);
    res -= w[i];
  }

  res += min_cost_flow(s, t, K + N);
  printf("%d\n", -res);
}
```

专栏　线性规划问题

在线性不等式或等式组成的约束条件下，最大化或最小化线性目标函数值的问题被称为线性规划问题（LP：Linear Programming）。实际上最短路问题、网络流问题等许多最优化问题都可以写成线性规划问题的形式。而所有的线性规划问题通过适当的变形之后，都可以转成如下标准型。

$$\max\{c^T x : Ax \leqslant b, x \geqslant 0, x \in \mathbb{R}^n\}$$

这里，A 是矩阵，b 和 c 是向量，$x \geqslant 0$ 这样的不等式表示的是向量中所有的维度都满足所给不等式。

在线性规划问题中，有一个被称为“对偶性”的重要概念。首先，让我们考虑如下 LP 问题。

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 3x_2 \\ \text{s.t.} \quad & x_1 + 2x_2 \leqslant 6 \quad (1) \\ & x_1 + x_2 \leqslant 4 \quad (2) \\ & x_1, x_2 \geqslant 0 \end{aligned}$$

当 $x_1=x_2=2$ 时，目标函数的值为 10。通过将 2 个约束条件相加，得到

$$(1)+(2): 2x_1+3x_2 \leqslant 10$$

所以 10 就是最优解。更一般地，将两个约束条件分别乘上大于零的 y_1 和 y_2 后再相加，就可以得到如下不等式。

$$y_1(1)+y_2(2): (y_1+y_2)x_1+(2y_1+y_2)x_2 \leqslant 6y_1+4y_2$$

这里假设 $y_1+y_2 \geqslant 2$ 和 $2y_1+y_2 \geqslant 3$ 成立的话，那么右侧的 $6y_1+4y_2$ 就是最优解的上界。以最小化满足条件的上界为目标，就得到了下面的 LP 问题。

$$\begin{array}{lll} \text{maximize} & 6y_1+4y_2 & \\ \text{s.t.} & y_1+y_2 \geqslant 2 & (1') \\ & 2y_1+y_2 \geqslant 3 & (2') \\ & y_1+y_2 \geqslant 0 & \end{array}$$

对于这样得到的新问题，如果我们再考虑最大化它的目标函数值的下界的问题，就又会得到和最开始一样的问题。我们将这两个问题分别称为原问题和对偶问题。更一般地，标准型的最大化问题

$$(\mathrm{P})\max\{c^T x : Ax \leqslant b, x \geqslant 0, x \in \mathbb{R}^n\}$$

的对偶问题是最小化问题

$$(\mathrm{D})\min\{b^T y : A^T y \geqslant c, y \geqslant 0, y \in \mathbb{R}^n\}$$ [1]

从对偶问题的导出过程可以知道(P 的最优解)≤(D 的最优解)。事实上，当存在最优解时，等号总是成立的，这又被称为强对偶定理。例如，最大流问题是最小割问题的对偶问题，由强对偶定理可知它们的值是相等的。另外，最大匹配问题是最小顶点覆盖的对偶问题，但是把这些问题写成 LP 问题的形式，最优解未必是整数。不过对于二分图，最优解一定是整数，因此二分图的最大匹配等于最小顶点覆盖。

利用单纯形法或内点法，我们能够高效地求解 LP 问题，不过在程序设计竞赛当中，却不常出现需要使用这些算法求解的题目。不过，了解 LP 问题的对偶性的概念，对于理解算法是很有帮助的，有时候还能有助于将问题转化成网络流的形式得以解决。

① 对偶问题里的 y 和原问题里的 x 的维度可能不同，不一定都是 n。——译者注

3.6 与平面和空间打交道的计算几何

➠比赛中常常会出一些几何方面的问题。大多数的几何问题实现起来都比较复杂，并有较多的边界情况和数值误差需要考虑，所以解答起来可能比看起来的还要难。几何问题千变万化，要一一讲解是非常困难的，本节将围绕其中最常用的技巧和思想，介绍求解几何问题的方法。

3.6.1 计算几何基础

Jack Straws （POJ 1127）

桌子上放着 n 根木棍，木棍 i 的两端的坐标分别是(p_{ix}, p_{iy})和(q_{ix}, q_{iy})。给定 m 对木棍(a_i, b_i)，请判断每对木棍是否相连。当两根木棍之间有公共点时，就认为它们是相连的。通过相连的木棍间接的连在一起的两根木棍也认为是相连的。

⚠限制条件

- $2 \leqslant n \leqslant 12$
- $0 \leqslant p_{ix}, p_{iy}, q_{ix}, q_{iy} \leqslant 100$
- $0 \leqslant m \leqslant 10000$, $1 \leqslant a_i, b_i \leqslant n$

样例

输入

```
n = 4
p = {(0, 4), (0, 1), (1, 2), (1, 0)}
q = {(4, 1), (2, 3), (3, 4), (2, 1)}
m = 4
(a, b) = {(1, 2), (1, 4), (2, 3), (2, 4)}
```

输出

```
CONNECTED
NOT CONNECTED
CONNECTED
NOT CONNECTED
```

样例对应的场景

木棍就是二维平面上的线段，只要能够判断线段是否相交，那么建图以后就可以轻松地进行连接性判断。那么，应该如何判断两条线段是否相交呢？首先会想到计算两直线的交点，然后判断交点是否在线段上这一方法。那么两条直线的交点要怎么求得呢？虽然可以把直线表示成方程，通过联立方程组求解。但在几何问题中，运用向量的内积和外积进行计算是非常方便的。对于二维向量$p_1=(x_1, y_1)$和$p_2=(x_2, y_2)$，我们定义内积$p_1 \cdot p_2=x_1x_2+y_1y_2$，外积$p_1 \times p_2=x_1y_2-y_1x_2$。要判断点$q$是否在线段$p_1-p_2$上，只要先利用外积根据是否有$(p_1-q)\times(p_2-q)=0$来判断点$q$是否在直线$p_1-p_2$上，再利用内积根据是否有$(p_1-q)\cdot(p_2-q)\leqslant 0$来判断点$q$是否落在$p_1-p_2$之间。而要求两直线的交点，通过变量$t$将直线$p_1-p_2$上的点表示为$p_1+t(p_2-p_1)$，交点又在直线$q_1-q_2$上，所以有：

$$(q_2-q_1)\times(p_1+t(p_2-p_1)-q_1)=0$$

于是可以利用下式求得t的值。

$$p_1+\frac{(q_2-q_1)\times(q_1-p_1)}{(q_2-q_1)\times(p_2-p_1)}(p_2-p_1)$$

但是，使用这个方法时还要注意边界情况。让我们来看看样例中的木棍2和木棍4。这两条线段是平行的，对应直线没有交点。但平行的线段也有可能有公共点，所以此时需要特别注意。对此有不同的处理方法，这里我们选择通过检查端点是否在另一条线段上来判断。

```
double EPS = 1e-10;

// 考虑误差的加法运算
double add(double a, double b) {
  if (abs(a + b) < EPS * (abs(a) + abs(b))) return 0;
  return a + b;
}

// 二维向量结构体
struct P {
  double x, y;
  P() {}
```

```
  P(double x, double y) : x(x), y(y) {
  }
  P operator + (P p) {
    return P(add(x, p.x), add(y, p.y));
  }
  P operator - (P p) {
    return P(add(x, -p.x), add(y, -p.y));
  }
  P operator * (double d) {
    return P(x * d, y * d);
  }
  double dot(P p) {  // 内积
    return add(x * p.x, y * p.y);
  }
  double det(P p) {  // 外积
    return add(x * p.y, -y * p.x);
  }
};

// 判断点q是否在线段p1-p2上
bool on_seg(P p1, P p2, P q) {
  return (p1 - q).det(p2 - q) == 0 && (p1 - q).dot(p2 - q) <= 0;
}

// 计算直线p1-p2与直线q1-q2的交点
P intersection(P p1, P p2, P q1, P q2) {
  return p1 + (p2 - p1) * ((q2 - q1).det(q1 - p1) / (q2 - q1).det(p2 - p1));
}

// 输入
int n;
P p[MAX_N], q[MAX_N];
int m;
int a[MAX_M], b[MAX_M];

bool g[MAX_N][MAX_N];  // 相连关系图

void solve() {
  for (int i = 0; i < n; i++) {
    g[i][i] = true;
    for (int j = 0; j < i; j++) {
      // 判断木棍i和木棍j是否有公共点
      if ((p[i] - q[i]).det(p[j] - q[j]) == 0) {
        // 平行时
        g[i][j] = g[j][i] = on_seg(p[i], q[i], p[j])
                          || on_seg(p[i], q[i], q[j])
                          || on_seg(p[j], q[j], p[i])
                          || on_seg(p[j], q[j], q[i]);
      } else {
        // 非平行时
        P r = intersection(p[i], q[i], p[j], q[j]);
        g[i][j] = g[j][i] = on_seg(p[i], q[i], r) && on_seg(p[j], q[j], r);
      }
    }
```

```
    }
    // 通过Floyd-Warshall算法判断任意两点间是否相连
    for (int k = 0; k < n; k++) {
      for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
          g[i][j] |= g[i][k] && g[k][j];
        }
      }
    }

    for (int i = 0; i < m; i++) {
      puts(g[a[i] - 1][b[i] - 1] ? "CONNECTED" : "NOT CONNECTED");
    }
  }
```

像这样，几何问题中需要特别注意处理边界情况和避免分类讨论时产生疏漏。具有代表性的边界情况有：

(1) 两直线平行、三点共线或除零的情况。
(2) 将所有点排序后处理时，弄错了对相同x坐标或y坐标的点的处理顺序。
(3) 判断直线和多边形是否相交时，漏掉了直线恰好通过多边形顶点的情况。
(4) 判断两个实心物体是否相交时，忘记了其中一个完全在另一个内部的情况。
(5) 在物体移动的模拟类问题中，将初始相接触但随后分离的情况误判为碰撞。

此外，求解几何问题时往往会用到浮点数，因而要多注意一下误差问题。由于以整数形式输入的情况很常见，所以如果可能的话，尽量保持整数形式处理也是一种办法。

事实上，判断线段相交时，通常不用前面这样利用两直线交点的方法，而是利用基于ccw函数[①]的方法。有兴趣的读者可以查阅一下相关资料。另外，C++中可以把STL的complex类[②]当作二维向量使用，这样就不用自己实现各种运算且容易完成各类操作，十分方便。这次我们介绍了如何求两直线的交点，此外也有需要求直线同圆的交点或是两个圆的公切线之类的情况。虽然计算并不那么复杂，但容易出现各种差错。对这类在问题中经常会用到的函数，不妨预先准备好相应的模块。

① ccw是Counter Clock Wise的缩写。——译者注
② 严格来说std::complex是C++ Standard Library的一部分，但不属于Standard Templete Library的一部分。——译者注

专栏　计算误差

在处理 double 之类的浮点数时，需要注意浮点误差。浮点数将数字表示为(尾数)×2^(指数)的形式[1]，对于一定范围内的整数，可以精确表示，但对于 0.1 这样的简单的小数，却无法精确表示。对于不能精确表示的数，只能通过所能表示的数中最接近的数近似表示。近似表示造成的误差则称为舍入误差。

对于 double 类型，尾数部分大致相当于 10 进制下的 15 位。多数情况下，就其计算结果而言，精度已经足够了。但当我们要比较两个计算后的结果时，就需要特别注意。

例如上面的问题，首先求出交点，然后判断交点是否在线段上。如果我们完全不考虑误差，程序将会得到 Wrong Answer。这是因为由于误差，原本应该相等的结果有可能实际上并不相等。比较包含舍入误差的浮点数时所采用的方法，一般是选取合适的足够小的常数 EPS，按如下规则处理

$$a < 0 \rightarrow a < -\mathrm{EPS}$$
$$a <= 0 \rightarrow a < \mathrm{EPS}$$
$$a == 0 \rightarrow \mathrm{abs}(a) < \mathrm{EPS}$$

在大多数计算结果不是太大的情况下，都可以使用该方法处理。但在几何问题的计算过程中，常会通过内积或外积得到原坐标值平方大小的结果，在比较这些大的结果时需要格外注意。假设所取的 ESP 为 10^{-10}。现在要求对因为误差导致原本相等却实际不等的大约 10^8 大小的两个数作差，并判断差是否等于 0。由于 double 的精度只有约十进制 15 位，所得差的绝对值将大于 EPS，所以会被误判为不等。像这样，求两个非常接近的数的差时将会发生有效位丢失，导致所得结果的有效数字位数大大减少。前面的程序，我们所采取的方法是，在进行浮点数减法时，如果两个数按相对误差比较是相等的就令结果为 0。这样我们在计算的过程中处理了误差，所以在与 0 进行比较时，就可以不考虑误差直接比较了。

众所周知处理误差是一个非常复杂且深奥的问题。不过在程序设计竞赛中，有时候题目描述中会说明计算中的微小误差不会影响结果。即便没有这类说明，大多数情况下，利用上面介绍的方法也足够了。如果无论如何都无法满足精度的话，还可以使用别的处理方法。比如说可以使用分数类避免浮点数运算，在 C++中还可以使用精度更高的浮点数类型 long double[2]，在 Java 中还可以使用 BigDecimal 之类的高精度浮点数类。此外，对确定是 0 的结果进行特别处理可以让程序对误差更健壮。在上面的例子中，两直线相交的交点必在直线上，此时可以省去 on_seg 函数的前半部分。

① 这是最常见的一类浮点数，但还有其他类型的浮点数。——译者注

② C++中long double的精度不会低于double，但有可能只等于double。比如在Visual C++中二者都是64位双精度浮点数；而在GCC中double是64位双精度浮点数，（x86架构下）long double是80位扩展精度浮点数。——译者注

3.6.2 极限情况

White Bird （AOJ 2308）

平面上有 N 个障碍物，i 号障碍物是左下角为(L_i, B_i)，右上角为(R_i, T_i)的与坐标轴平行的长方形。从原点以初速度 V 向任意角度发射出一只白鸟。设重力加速度沿 y 轴负方向，大小为 9.8，射出的鸟将呈抛物线飞出，直到撞到障碍物为止。鸟可以在中途产一枚卵，所产的卵将沿 y 轴负方向竖直落下，直到撞到障碍物为止。请问是否可以让卵击中坐标位于(X, Y)的猪。

限制条件

- $0 \leqslant N \leqslant 50$
- $0 \leqslant V \leqslant 50$
- $0 \leqslant X, Y \leqslant 300$
- $0 \leqslant L_i, B_i, R_i, T_i \leqslant 300$
- 题目保证 L_i, B_i, R_i, T_i 偏移 10^{-6} 也不影响答案

样例 1

输入

```
N = 0
V = 7
(X, Y) = (3, 1)
```

输出

```
Yes
```

样例 2

输入

```
N = 1
V = 7
(X, Y) = (3, 1)
(L, B, R, T) = {(1, 1, 2, 2)}
```

输出

```
No
```

最后的限制条件常常会在几何问题中附带出现，根据这一点就无需考虑只有通过像穿过针孔一样的唯一线路才能让卵击中猪的情况了。首先，让我们考虑一下如何判断以某个角度射出的鸟是否可以产卵击中猪。只要射出的鸟在撞到障碍物之前能够从猪的正上方飞过，并且此时与猪之间没有障碍物的话，在正上方产卵就可以击中猪了。判断白鸟是否撞到障碍物，就是判断抛物线和长方形是否相交（如果将长方形分解为线段，只判断抛物线是否同各条线段相交，就可能无法很好地处理抛物线恰好经过长方形的顶点的情况，需要注意），稍加计算即可完成。

接下来，我们思考一下应该如何枚举所有关键射出角度。假设以某个角度射出时不会撞到障碍物，我们逐渐降低这个角度，直到某处变成

(1) 恰好经过(X, Y)
(2) 恰好经过某个障碍物的左上角或右上角

就不能再降低了。虽然作为解的角度可能有无穷多个，但因为无论哪个都可以不断降低直至变为1或2的情况，所以只要检查这些角度就足够了。

极限情况

```
const double g = 9.8;  // 重力加速度

// 输入
int N, V, X, Y;
int L[MAX_N], B[MAX_N], R[MAX_N], T[MAX_N];

// 计算以vy的速度竖直向上射出t秒后的位置
double calc(double vy, double t) {
  return vy * t - g * t * t / 2;
}

// a相对lb和ub的位置
int cmp(double lb, double ub, double a) {
  return a < lb + EPS ? -1 : a > ub - EPS ? 1 : 0;
}

// 判断当射出路径经过点(qx,qy)时，卵是否能击中猪
bool check(double qx, double qy) {
  // 设初速度在x方向和y方向的分量分别为vx和vy，设通过(qx,qy)的时间为t
  // 求解联立方程式vx^2 + vy^2 = V^2, vx * t = qx, vy * t - 1/2 g t^2 = qy
  double a = g * g / 4, b = g * qy - V * V, c = qx * qx + qy * qy;
```

```
double D = b * b - 4 * a * c;
if (D < 0 && D > -EPS) D = 0;
if (D < 0) return false;
for (int d = -1; d <= 1; d += 2) {  // 验证联立方程式的两个解的循环
  double t2 = (-b + d * sqrt(D)) / (2 * a);
  if (t2 <= 0) continue;
  double t = sqrt(t2);
  double vx = qx / t, vy = (qy + g * t * t / 2) / t;

  // 判断是否通过猪的正上方
  double yt = calc(vy, X / vx);
  if (yt < Y - EPS) continue;

  bool ok = true;
  for (int i = 0; i < N; i++) {
    if (L[i] >= X) continue;
    // 判断在猪正上方的鸟和猪之间是否有障碍物
    if (R[i] == X && Y <= T[i] && B[i] <= yt) ok = false;
    // 判断在飞到猪正上方之前是否会撞到障碍物
    int yL = cmp(B[i], T[i], calc(vy, L[i] / vx));  // 左侧的相对位置
    int yR = cmp(B[i], T[i], calc(vy, R[i] / vx));  // 右侧的相对位置
    int xH = cmp(L[i], R[i], vx * (vy / g));  // 最高点的相对位置
    int yH = cmp(B[i], T[i], calc(vy, vy / g));
    if (xH == 0 && yH >= 0 && yL < 0) ok = false;
    if (yL * yR <= 0) ok = false;
  }
  if (ok) return true;
  }
  return false;
}

void solve() {
  // 截掉猪以右的障碍物
  for (int i = 0; i < N; i++) {
    R[i] = min(R[i], X);
  }
  bool ok = check(X, Y);       // 直接撞上猪的情况
  for (int i = 0; i < N; i++) {
    ok |= check(L[i], T[i]);  // 经过左上角的情况
    ok |= check(R[i], T[i]);  // 经过右上角的情况
  }
  puts(ok ? "Yes" : "No");
}
```

像这样在几何问题中，当可行解可以取连续一段的值时，很多时候只要考虑边界的极限情况就能够顺利解决问题了。

3.6.3 平面扫描

Coneology （POJ 2932）

平面上有 N 个两两没有公共点的圆，i 号圆的圆心在 (x_i, y_i)，半径为 r_i。求所有最外层的，即不包含于其他圆内部的圆。

限制条件

- $1 \leqslant N \leqslant 40000$

样例

输入

```
N = 5
(x, y, r) = {(0, -2, 1), (0, 3, 3), (0, 0, 10), (0, 1.5, 1), (50, 50, 10)}
```

输出

```
2
3 5（最外层的圆有两个，它们是3号和5号。）
```

由于有任意两圆都没有公共点这一条件，要判断一个圆是否在其他圆的内部，只要判断其圆心是否在其他圆内即可。这样判断每个圆是否是最外层的复杂度为 $O(N)$，因此很容易得到 $O(N^2)$ 复杂度的算法。现在，给大家介绍一个利用平面扫描这一技术得到的更为高效的算法。

在几何问题中，我们经常利用平面扫描技术来降低算法的复杂度。所谓平面扫描，是指扫描线在平面上按给定轨迹移动的同时，不断根据扫描线扫过部分更新信息，从而得到整体所要求的结果的方法。扫描的方法，既可以从左向右平移与 y 轴平行的直线，也可以固定射线的端点逆时针转动。

扫描的方法

对于这道题，我们在从左向右平移与 y 轴平行的直线的同时，维护与扫描线相交的最外层的圆的集合。从左向右移动的过程中，只有扫描线移动到圆的左右两端时，圆与扫描线的相交关系才会

发生变化，因此我们先将所有这样的x坐标枚举出来并排好序。

首先，我们来看一下扫描线移动到某个圆的左端时的情形。此时，我们需要判断该圆是否包含在其他圆中。为此，我们只需从当前与扫描线相交的最外层的圆中，找到上下两侧y坐标方向距离最近的两个圆，并检查它们就足够了。这是因为，假设该圆被包含于更远的圆中，却不被包含于最近的圆中，就会如下图所示，得出两圆相交的结论。而这与题目所给条件不符。于是，只要用二叉查找树来维护这些圆，就能够在$O(\log n)$时间内取得待检查的圆了。

产生矛盾

其次，我们看一下扫描线移动到某个圆的右端时的情形。此时的处理很简单，如果该圆已经包含于其他圆中就什么也不做，如果是最外层的圆就将它从二叉树中删去。综上，总的复杂度为$O(n\log n)$。

```
// 输入
int N;
double x[MAX_N], y[MAX_N], r[MAX_N];

// 判断圆i是否在圆j的内部
bool inside(int i, int j) {
  double dx = x[i] - x[j], dy = y[i] - y[j];
  return dx * dx + dy * dy <= r[j] * r[j];
}

void solve() {
  // 枚举关键点
  vector<pair<double, int> > events;  // 圆的左右两端的x坐标
  for (int i = 0; i < N; i++) {
    events.push_back(make_pair(x[i] - r[i], i));       // 圆的左端
    events.push_back(make_pair(x[i] + r[i], i + N));   // 圆的右端
  }
  sort(events.begin(), events.end());

  // 平面扫描
  set<pair<double, int> > outers;  // 与扫描线相交的最外层的圆的集合
```

```
    vector<int> res;                    // 最外层圆的列表
    for (int i = 0; i < events.size(); i++) {
      int id = events[i].second % N;
      if (events[i].second < N) {       // 扫描到左端
        set<pair<double, int> >::iterator it = outers.lower_bound(make_pair(y[id], id));
        if (it != outers.end() && inside(id, it->second)) continue;
        if (it != outers.begin() && inside(id, (--it)->second)) continue;
        res.push_back(id);
        outers.insert(make_pair(y[id], id));
      } else {                          // 扫描到右端
        outers.erase(make_pair(y[id], id));
      }
    }

    sort(res.begin(), res.end());
    printf("%d\n", res.size());
    for (int i = 0; i < res.size(); i++) {
      printf("%d%c", res[i] + 1, i + 1 == res.size() ? '\n' : ' ');
    }
  }
```

3.6.4 凸包

Beauty Contest（POJ 2187）

平面上有N个牧场。i号牧场的位置在格点(x_i, y_i)，所有牧场的位置互不相同。请计算距离最远的两个牧场间的距离，输出最远距离的平方。

限制条件

- $2 \leqslant N \leqslant 50000$
- $-10000 \leqslant x_i, y_i \leqslant 10000$

样例

输入

```
N = 8
(x, y) = {(0, 5), (1, 8), (3, 4), (5, 0), (6, 2), (6, 6), (8, 3), (8, 7)}
```

输出

```
80（点(1,8)与点(5,0)）
```

由于在时间限制内无法完全枚举所有点对并取距离的最大值，需要避免计算一些不必要的点对。如果某个点在另外三个点组成的三角形的内部，那么它就不可能属于最远点对，因而可以删去。这样，最后需要考虑的点，就只剩下不在任意三个点组成的三角形内部的，所给点集中最外围的

点了。这些最外围的点的集合，就是包围原点集的最小凸多边形的顶点组成的集合，称为原点集的凸包。因为顶点的坐标限定为整数，坐标值的范围不超过M的凸多边形的顶点数只有$O(\sqrt{M})$个，所以只要枚举凸包上的所有点对并计算距离就可以求得最远点对了。

样例对应的凸包

求凸包的算法有很多，要求n个点集对应的凸包，只要$O(n\log n)$时间。这里给大家介绍一种比较容易实现的基于平面扫描法的Graham扫描算法。

首先，把点集按x坐标→y坐标的字典序升序排序。那么排序后的第一个和最后一个点必然是凸包上的顶点，它们之间的部分可以分成上下两条链分别求解。求下侧的链时只要从小到大处理排序后的点列，逐步构造凸包。在构造过程中的凸包末尾加上新的顶点后，可能会破坏凸性，此时只要将凹的部分的点从末尾除去就好了。求上侧的链也是一样地从大到小处理即可。排序的复杂度为$O(n\log n)$，剩余部分处理的复杂度为$O(n)$。

构造下侧的链

```
// 字典序比较
bool cmp_x(const P& p, const P& q) {
  if (p.x != q.x) return p.x < q.x;
  return p.y < q.y;
}

// 求凸包
vector<P> convex_hull(P* ps, int n) {
  sort(ps, ps + n, cmp_x);
  int k = 0;              // 凸包的顶点数
```

```
  vector<P> qs(n * 2);  // 构造中的凸包
  // 构造凸包的下侧
  for (int i = 0; i < n; i++) {
    while (k > 1 && (qs[k - 1] - qs[k - 2]).det(ps[i] - qs[k - 1]) <= 0) k--;
    qs[k++] = ps[i];
  }
  // 构造凸包的上侧
  for (int i = n - 2, t = k; i >= 0; i--) {
    while (k > t && (qs[k - 1] - qs[k - 2]).det(ps[i] - qs[k - 1]) <= 0) k--;
    qs[k++] = ps[i];
  }
  qs.resize(k - 1);
  return qs;
}

// 距离的平方
double dist(P p, P q) {
  return (p - q).dot(p - q);
}

// 输入
int N;
P ps[MAX_N];

void solve() {
  vector<P> qs = convex_hull(ps, N);
  double res = 0;
  for (int i = 0; i < qs.size(); i++) {
    for (int j = 0; j < i; j++) {
      res = max(res, dist(qs[i], qs[j]));
    }
  }
  printf("%.0f\n", res);
}
```

事实上，即使坐标范围变大这道题也能求解。为此我们需要再次用到凸包的性质。假设最远点对是p和q，那么p就是点集中$(p-q)$方向最远的点，而q是点集中$(q-p)$方向最远的点。因此，可以按照逆时针逐渐改变方向，同时枚举出所有对于某个方向上最远的点对[①]，那么最远点对一定也包含于其中。在逐渐改变方向的过程中，对踵点对只有在方向等于凸包某条边的法线方向时发生变化，此时点将向凸包上对应的相邻点移动。令方向逆时针旋转一周，那么对踵点对也在凸包上转了一周，这样就可以在凸包顶点数的线性时间内求得最远点对。像这样，在凸包上旋转扫描的方法又叫做旋转卡壳法[②]。

① 这样的点对又叫做对踵点对。——译者注

② Rotating calipers通常被称为旋转卡（qiǎ）壳（ké），其名称来源于算法的过程就像用游标卡尺卡着凸包旋转一周一样。事实上卡壳这个中文单词并无此意，在这里可能是一个误用。当然如果认为这里的卡壳是短语，壳指代凸包的话，正确的读法应该是卡（kǎ）壳。也有将该方法直译为旋转卡（kǎ）尺的。——译者注

对踵点对的变化

```
void solve() {
  vector<P> qs = convex_hull(ps, N);
  int n = qs.size();
  if (n == 2) {  // 特别处理凸包退化的情况
    printf("%.0f\n", dist(qs[0], qs[1]));
    return;
  }
  int i = 0, j = 0;  // 某个方向上的对踵点对
  // 求出x轴方向上的对踵点对
  for (int k = 0; k < n; k++) {
    if (!cmp_x(qs[i], qs[k])) i = k;
    if (cmp_x(qs[j], qs[k])) j = k;
  }
  double res = 0;
  int si = i, sj = j;
  while (i != sj || j != si) {  // 将方向逐步旋转180度
    res = max(res, dist(qs[i], qs[j]));
    // 判断先转到边i-(i+1)的法线方向还是边j-(j+1)的法线方向
    if ((qs[(i + 1) % n] - qs[i]).det(qs[(j + 1) % n] - qs[j]) < 0) {
      i = (i + 1) % n;  // 先转到边i-(i+1)的法线方向
    } else {
      j = (j + 1) % n;  // 先转到边j-(j+1)的法线方向
    }
  }
  printf("%.0f\n", res);
}
```

3.6.5 数值积分

Intersection of Two Prisms （AOJ 1313）

有一个侧棱与 z 轴平行的棱柱 P_1 和一个侧棱与 y 轴平行的棱柱 P_2。它们都向两端无限延伸，底面分别是包含 M 个顶点和 N 个顶点的凸多边形，其中第 i 个顶点的坐标分别是 (X_{1i}, Y_{1i}) 和 (X_{2i}, Z_{2i})。请计算这两个棱柱公共部分的体积。

限制条件

- $3 \leqslant M, N \leqslant 100$
- $-100 \leqslant X_{1i}, Y_{1i}, X_{2i}, Z_{2i} \leqslant 100$

样例

输入

```
M = 4
N = 3
(X1, Y1) = {(7, 2), (3, 3), (0, 2), (3, 1)}
(X2, Z2) = {(4, 2), (0, 1), (8, 1)}
```

输出

```
4.708333333333333
```

首先，会想到先求出公共部分的凸多面体的顶点坐标，然后再计算其体积这一方法。公共部分的凸多面体的顶点都是一个棱柱的侧面与另一个棱柱的侧棱的交点，可以通过$O(nm)$时间的枚举求得，因而这一方法貌似很合理。但因为涉及三维空间的几何运算，实现起来是非常麻烦的。

事实上，只要沿着x轴对棱柱切片，就可以非常简洁地解决这道题。我们按某个x值对侧棱与z轴平行的棱柱P_1切片后，就得到了$[y_1, y_2]\times(-\infty, \infty)$这样的在$z$轴方向无限延伸的长方形的横截面。同样的，我们按某个x值对侧棱与y轴平行的棱柱P_2切片后，就得到了$(-\infty, \infty)\times[z_1, z_2]$这样的在$y$轴方向无限延伸的长方形的横截面。因此，我们按某个$x$值对两个棱柱的公共部分切片后，得到的横截面就是长方形$[y_1, y_2]\times[z_1, z_2]$。而长方形的面积通过$(y_2-y_1)\times(z_2-z_1)$就可以轻松求得，之后只要关于$x$轴对面积求积分就能得到公共部分的体积了。

y-z平面所截的横截面

首先，我们枚举出原棱柱底面顶点的所有x坐标并排序。于是，在相邻两个x坐标之间的区间中，按x值切片得到的长方形的顶点坐标是关于x的线性函数，所以面积就是关于x的二次函数，其积分很容易计算。虽然可以通过求得表达式后再来计算二次函数的积分，不过利用下面的Simpson公式则更为轻松。

$$\int_a^b f(x)\mathrm{d}x \approx \frac{b-a}{6}\left(f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right)$$

Simpson公式就是在数值积分中用二次函数来近似原函数进行积分而得到的公式。如果原函数本身就是次数不超过二的多项式的话，那么通过该公式就可以求得精确的积分值。利用该公式，我们就无需求出关于x的多项式，而只要计算按区间的端点和中点切片得到长方形的面积就足够了。

```
// 输入
int M, N;
int X1[MAX_M], Y1[MAX_M];
int X2[MAX_N], Z2[MAX_N];

// 计算按x值对多边形切片得到的宽度
double width(int* X, int* Y, int n, double x) {
  double lb = INF, ub = -INF;
  for (int i = 0; i < n; i++) {
    double x1 = X[i], y1 = Y[i], x2 = X[(i + 1) % n], y2 = Y[(i + 1) % n];
    // 检查与第i条边是否相交
    if ((x1 - x) * (x2 - x) <= 0 && x1 != x2) {
      // 计算交点的坐标
      double y = y1 + (y2 - y1) * (x - x1) / (x2 - x1);
      lb = min(lb, y);
      ub = max(ub, y);
    }
  }
  return max(0.0, ub - lb);
}

void solve() {
  // 枚举区间的端点
  int min1 = *min_element(X1, X1 + M), max1 = *max_element(X1, X1 + M);
  int min2 = *min_element(X2, X2 + N), max2 = *max_element(X2, X2 + N);
  vector<int> xs;
  for (int i = 0; i < M; i++) xs.push_back(X1[i]);
  for (int i = 0; i < N; i++) xs.push_back(X2[i]);
  sort(xs.begin(), xs.end());

  double res = 0;
  for (int i = 0; i + 1 < xs.size(); i++) {
    double a = xs[i], b = xs[i + 1], c = (a + b) / 2;
    if (min1 <= c && c <= max1 && min2 <= c && c <= max2) {
      // 利用Simpson公式求积分
      double fa = width(X1, Y1, M, a) * width(X2, Z2, N, a);
```

```
        double fb = width(X1, Y1, M, b) * width(X2, Z2, N, b);
        double fc = width(X1, Y1, M, c) * width(X2, Z2, N, c);
        res += (b - a) / 6 * (fa + 4 * fc + fb);
      }
    }
    printf("%.10f\n", res);
  }
```

像本题这样，几何问题中往往会因为选用方法的不同，导致实现的复杂程度产生巨大差异。因此，在想到一个解法之后，应该再进一步思考一下是否还有更简洁的解法。

3.7 一起来挑战 GCJ 的题目（2）

➠让我们运用迄今为止所介绍的技术，一起来挑战一下GCJ的题目吧。

3.7.1 Numbers

Numbers （2008 Round 1A C）

请输出 $(3+\sqrt{5})^n$ 整数部分的最后 3 位。如果结果不超过 2 位，请补足前导 0。

限制条件

Small

- $2 \leqslant n \leqslant 30$

Large

- $2 \leqslant n \leqslant 2000000000$

样例 1

输入

```
N=2
```

输出

027（$(3+\sqrt{5})^2 \fallingdotseq 27.41640786$，因此整数部分的最后3位补足前导0之后是027）

样例 2

输入

```
N=5
```

输出

935（$(3+\sqrt{5})^5 \fallingdotseq 3935.739820$，因此整数部分的最后3位是935）

如果用double来计算 $(3+\sqrt{5})^n$，精度显然是不够的。虽然使用Java中支持高精度小数的BigDecimal

可以通过Small测试数据，但是，要通过Large测试数据，就必须把 $\sqrt{5}$ 的值计算到大约n位精度，因此朴素的方法是无法求出答案的。其实，经过仔细分析可以发现，这个问题不需要求出 $\sqrt{5}$ 的值也可以求解。

首先将 $(3+\sqrt{5})^n$ 展开之后可以发现是 $a_n+b_n\sqrt{5}$ 的形式。同样地，我们有 $(3-\sqrt{5})^n=a_n-b_n\sqrt{5}$ 。因此，$(3+\sqrt{5})^n+(3-\sqrt{5})^n=2a_n$ 是个整数，其中 $0<(3-\sqrt{5})^n<1$ ，这正是解题的关键。由于 $(3+\sqrt{5})^n=2a_n-(3-\sqrt{5})^n$ ，所以 $(3+\sqrt{5})^n$ 的整数部分等于 $2a_n-1$ 。

根据上面的推导，只要高效地求出 a_n 就可以解决这个问题了。由于 $(3+\sqrt{5})^{n+1}=(3+\sqrt{5})(3+\sqrt{5})^n$ $=(3+\sqrt{5})(a_n+b_n\sqrt{5})$ ，我们可以得到 a_n 、b_n 和 a_{n+1} 、b_{n+1} 的递推关系。

$$a_{n+1}=3a_n+5b_n$$
$$b_{n+1}=a_n+3b_n$$
$$a_0=1\ b_0=0$$

我们可以使用矩阵表示这个递推关系，因此可以使用快速幂运算，在$O(\log n)$的时间内求出 a_n 和 b_n 。由于只需要最后3位，因此运算时mod 1000就可以了。

```
const int MOD = 1000;

typedef vector<int> vec;
typedef vector<vec> mat;

// 输入
int n;

void solve() {
  mat A(2, vec(2, 0));
  A[0][0] = 3; A[0][1] = 5;
  A[1][0] = 1; A[1][1] = 3;
  A = pow(A, n);
  printf("%03d\n", (A[0][0] * 2 + MOD - 1) % MOD);
}
```

我们称 $a-b\sqrt{n}$ 与 $a+b\sqrt{n}$ 互为共轭。$(a+b\sqrt{n})(a-b\sqrt{n})=a^2-nb^2$ ，$(a+b\sqrt{n})+(a-b\sqrt{n})=2a$ ，因此一对共轭的数具有相加或者相乘之后可以得到整数的性质。此外，还有相乘后取共轭和先取共轭再相乘结果相同等良好性质。如果记住这些结论，在解题时往往能比较容易想到解法。

这个问题的输出格式比较特殊，如果不足3位则要补足前导0。不过因为样例中包含了这种情况，所以如果忘记了也会注意到这个问题。但是也有许多问题的样例并不包含一些特殊情况，需要特别注意。此外，C和Java中都可以通过`printf("%03d", answer)`的方法自动补上需要的前导0。

3.7.2 No Cheating

No Cheating （2008 Round 3 C）

某个高中的期末考试是在一个很大的教室中进行的。这个教室的座位有 M 行 N 列。为了防止作弊，学校在安排学生座位时，需要使得任何一个学生都无法看到其他人的答案。已知在 (x, y) 位置的学生可以看到 $(x-1, y)$, $(x+1, y)$ $(x-1, y-1)$, $(x+1, y-1)$ 四个位置的学生的答案。另外，在教室中有一些座位已经损坏而无法安排学生就做。现在要求一次考试最多能安排多少学生参加。

限制条件

Small

- $1 \leqslant N, M \leqslant 10$

Large

- $1 \leqslant N, M \leqslant 80$

样例 1

输入

```
M = 2
N = 3
座位情况如下图（'x'表示已经坏掉的座位，'.'表示没有坏的座位）
...
...
```

输出

```
4（按照下面的方法安排座位）
o.o
o.o
```

样例 2

输入

```
M = 2
N = 3

x.x
xxx
```

输出

```
1
```

样例 3

输入

```
M = 2
N = 3

x.x
x.x
```

输出

```
2
```

可以将其看作这样一个图$G=(V, E)$，把没有坏的座位看作顶点。对于顶点v，如果能看到顶点u对应的座位的答案，就在它们之间连一条无向边$e=(v, u)$。这样，问题就变成了求任意两点都不相邻的最大的顶点集合$S \subseteq V$。

这个问题就是图的最大独立集问题。对于最大独立集问题，如果图是二分图，可以很容易求出答案。而事实上，在这个问题里，所有的边都连接着x为偶数的顶点和x为奇数的顶点，因此正是一个二分图。

```
const int dx[4] = {-1, -1, 1, 1}, dy[4] = {-1, 0, -1, 0};

// 输入
int M, N;
char seat[MAX_M][MAX_N + 1];  // 座位

void solve() {
  int num = 0;
  V = M * N;
  for (int y = 0; y < M; y++) {
    for (int x = 0; x < N; x++) {
      if (seat[y][x] == '.') {
        num++;
        for (int k = 0; k < 4; k++) {
          int x2 = x + dx[k], y2 = y + dy[k];
          if (0 <= x2 && x2 < N && 0 <= y2 && y2 < M && seat[y2][x2] == '.') {
            add_edge(x * M + y, x2 * M + y2);
          }
        }
      }
    }
  }
  printf("%d\n", num - bipartite_matching());
}
```

3.7.3 Stock Charts

Stock Charts （2009 Round 2 C）

你持有 n 支股票，你希望将股价在一年中的变动情况，以折线图的形式画出来。对于每支股票，你有这 n 支股票在一年中 k 个时间点的数据，并把相邻的数据点用线段连起来，画出折线图（图 A）。将每支股票都画一幅图很浪费空间，但是如果不同股票的线段在端点相接或者相交又会产生混乱。因此你希望在每一张图中画上若干条没有公共点的线段。求最少需要画几幅图？（图 B）

图A：折线图的例子

图B：把3支股票画在2幅图上的例子

⚠限制条件

- $2 \leqslant k \leqslant 25$
- $0 \leqslant \text{price}_{i,j} \leqslant 1000000$

（$\text{price}_{i,j}$ 表示股票 i 在时间 j 的股价）

Small

- $1 \leqslant n \leqslant 16$

Large

- $1 \leqslant n \leqslant 100$

样例

输入

```
n = 5, k = 2
price = {{1, 1}, {2, 2}, {5, 4}, {4, 4}, {4, 1}}
```

输出

2（第三支和第五支股票在一张图上，其余股票在另一张图上）

我们可以试想一下以股票为顶点，两支股票是否能画在同一幅图里来决定是否连边。例如，在由于线段相交或者相接而不能画在同一幅图里的股票之间连一条边。这样，这个问题就变成了求这个图的最小着色数。不过由于求解一般图的最小着色数是NP困难的，因此没有办法高效地求出答案。我们需要利用这个问题特有的性质来寻找其他解法。在这个问题中，如果两个折线图不相交，那么其中一条折线必然完全在另一条折线的上方。让我们来试着利用这条性质。

当股票i可以画在股票j的上方时，我们从顶点i向顶点j连一条边，便可以得到一个DAG。考虑在这个图上的路径，容易发现在一条路径上的顶点可以全部画在一幅图中。因此原问题又可以进一步转化为如下问题：使用尽可能少的路径覆盖图中的所有顶点。

有向图的例子

路径的例子

路径的条数和路径的起点个数相等。因此我们试着最小化路径起点的个数。由于对于不是起点的顶点一定都存在另外一个顶点作为路径上的前趋顶点，因此只需要最大化这样的顶点的个数就可以了。

我们考虑左右各有n个顶点的二分图。在前面所建的图中，如果顶点i有一条到顶点j的边，则在二分图中从连接左边的顶点i和右边的顶点j。这个建图方法也可以看成是把前面的图中的每个顶点都拆成2个顶点得到的。

从前面的图得到的二分图

接下来考虑这个图的匹配。我们把匹配中各边对应的原图的边连接起来之后，就可以得到若干条路径，并且匹配中包含的边数和不是起点的顶点数相等。因此求出二分图的最大匹配，然后用n减去这个值，就得到了最小路径覆盖的路径数。[1]

① 在转化为二分图时，非常重要的一点是原有向图中不能包含圈。对于一般的有向图，如果尝试使用同样的算法，就会因为产生圈儿导致无法正确计算出结果。

```
const int MAX_N = 100;
const int MAX_K = 25;

int N, K, P[MAX_N][MAX_K];

void solve() {
  V = N * 2;
  for (int i = 0; i < V; i++) G[i].clear();

  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      if (i == j) continue;
      bool f = true;
      for (int k = 0; k < K; k++) {
        if (P[j][k] >= P[i][k]) f = false;
      }
      if (f) add_edge(i, N + j);
    }
  }

  int ans = N - bipartite_matching();
  printf("%d\n", ans);
}
```

3.7.4 Watering Plants

Watering Plants （2009 Round 2 D）

你在温室中种植了 N 株植物。为了给这些植物浇水，你购买了 2 台自动浇水的机器。每株植物 i 占据了圆心为(X_i, Y_i)，半径为 R_i 的圆形区域。并且保证任意两个圆都互相不相交或者相切。每台机器都可以给某个完全包含于半径为 r 的圆形区域内的植物浇水。求最小的 r 使得存在一种方案能给所有植物浇水。

限制条件

- $1 \leqslant X_i \leqslant 1000$
- $1 \leqslant Y_i \leqslant 1000$
- $1 \leqslant R_i \leqslant 100$

Small

- $1 \leqslant N \leqslant 3$

Large

- $1 \leqslant N \leqslant 40$

样例 1

输入

```
N = 3
(X, Y, R) = {(20, 10, 2), (20, 20, 2), (40, 10, 3)}
```

输出

```
r = 7.0
```

样例 2

输入

```
N = 3
(X, Y, R) = {(20, 10, 3), (30, 10, 3), (40, 10, 3)}
```

输出

```
r = 8.0
```

整理一下问题中的重要信息，可以将原题简化为：

- 平面上有N个圆。
- 要在平面上放置2个半径为R的大圆，使得N个圆都完全包含在至少一个大圆中。
- 求R的最小值。

首先我们考虑一个简化后的问题，即只放置1个大圆的情况。在这种情况下应该如何求解呢？稍作思考就会发现要使R最小，则一定满足下面三种情况的其中一种

a. 和三圆相内切。
b. 在径向相对位置和两圆内切。
c. 和某个圆重合。

三种情况对应的图示

b和c的情况比较容易计算，但是a的情况应该如何计算圆心坐标呢？况且即使算出了圆心坐标，那么在2个大圆的情况下又应该如何做呢？把N个圆分成2组的情况有2^N种，无法一一尝试。不过

我们发现，半径最小的大圆只有有限种情况，因此我们试着考虑每次从这些情况中选出2个圆，并判断是否能覆盖所有的圆。由于这样的圆只有$O(N^3)$个，所以总算得到了一个比较可行的算法了。但是我们仍然需要解决前面遗留的一个很麻烦的问题：求和三圆相内切的圆。难道没有更容易的解法了吗？

如果2个半径为R的圆可以覆盖所有的圆的话，半径为$R'>R$的圆也可以。因此我们可以使用二分搜索。不过对于一个R，怎么判断2个半径为R的圆能否覆盖所有的圆呢？和之前一样，我们试着最小化需要考虑的圆的数量。由于现在圆的半径已经确定，所以可能会认为有无数种可能的圆。其实对于覆盖的圆的集合相同的大圆，我们只需要选择其中一个有代表性的就可以了。因此实际上只需要考虑如下两种情况就足够了

a. 和两圆内切。
b. 和某个圆同心。

两种情况对应的图示

a的情况应该如何计算圆心的坐标呢？和圆心在(x, y)半径为r的圆相内切的圆的圆心的轨迹为圆心为(x, y)，半径为$R-r$的圆。这样，只需要计算两圆的交点就可以了，比起之前的方法简单了不少。

和两圆相内切的圆

而且，这种方法需要考虑的圆只有$O(N^2)$个，即使算上二分搜索的复杂度，也完全可以在规定的时间内的出答案。

```
typedef long long ll;

// 输入
int N;
int X[MAX_N], Y[MAX_N], R[MAX_N];

// 求出圆心在(x,y)半径为r的大圆能覆盖的圆的集合
ll cover(double x, double y, double r) {
  ll S = 0;
  for (int i = 0; i < N; i++) {
    if (R[i] <= r) {
      double dx = x - X[i], dy = y - Y[i], dr = r - R[i];
      if (dx * dx + dy * dy <= dr * dr) {
        S |= 1LL << i;
      }
    }
  }
  return S;
}

// 判断两个半径为r的大圆是否能覆盖所有的圆
bool C(double r) {
  vector<ll> cand;  // 一个大圆能覆盖的集合的列表
  cand.push_back(0);

  // 情况a
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < i; j++) {
      if (R[i] < r && R[j] < r) {
        // 计算两圆的交点
        double x1 = X[i], y1 = Y[i], r1 = r - R[i];
        double x2 = X[j], y2 = Y[j], r2 = r - R[j];
        double dx = x2 - x1, dy = y2 - y1;
        double a = dx * dx + dy * dy;
        double b = ((r1 * r1 - r2 * r2) / a + 1) / 2;
        double d = r1 * r1 / a - b * b;
        if (d >= 0) {
          d = sqrt(d);
          double x3 = x1 + dx * b;
          double y3 = y1 + dy * b;
          double x4 = -dy * d;
          double y4 = dx * d;
          // 考虑到可能产生误差，因此对i和j做特别处理
          ll ij = 1LL << i | 1LL << j;
          cand.push_back(cover(x3 - x4, y3 - y4, r) | ij);
          cand.push_back(cover(x3 + x4, y3 + y4, r) | ij);
        }
      }
    }
  }

  // 情况b
  for (int i = 0; i < N; i++) {
    if (R[i] <= r) {
```

```
      cand.push_back(cover(X[i], Y[i], r) | 1LL << i);
    }
  }

  // 每次从可能的圆中取出2个，并判断是否能覆盖所有的圆
  for (int i = 0; i < cand.size(); i++) {
    for (int j = 0; j < i; j++) {
      if ((cand[i] | cand[j]) == (1LL << N) - 1) {
        return true;
      }
    }
  }
  return false;
}

void solve() {
  // 对半径r使用二分搜索
  double lb = 0, ub = 10000;
  for (int i = 0; i < 100; i++) {
    double mid = (lb + ub) / 2;
    if (C(mid)) ub = mid;
    else lb = mid;
  }

  printf("%.6f\n", ub);
}
```

正如在前一节中提到的那样，如果在几何问题中使用了浮点数，就需要注意浮点数产生的精度误差。例如在前面这个问题中，需要计算两个圆i和j相切的圆，然后判断是否每个圆都被完全覆盖了。这时i和j都应该是被完全覆盖的。但是由于精度误差，可能会导致这两个圆被判断为不能被完全覆盖。因此对i和j的判断需要特殊处理。一般来说，在对有误差的浮点数进行比较时，需要选取适当的较小的数EPS，然后进行如下改写

$$a < b \rightarrow a + \mathrm{EPS} < b$$
$$a \leqslant b \rightarrow a < b + \mathrm{EPS}$$
$$a == b \rightarrow \mathrm{abs}(a - b) < \mathrm{EPS}$$

不过需要注意的是，像这种在二分搜索中进行的比较的情况，如果在比较时使用了考虑误差的比较方法，得到的结果反而可能产生差错。虽然在本题中，产生EPS大小的误差也不会有什么问题，但是在某些问题中也有可能会导致较大的误差产生。

3.7.5 Number Sets

Number Sets （2008 Round 1B B）

给定连续的整数区间$[A, B]$和整数 P。最初，区间$[A, B]$的所有数都属于一个只包含自己的集合。对于区间中的每一个数对，如果它们有不小于 P 的公共质因数，就把这两个数所在的集合合并。问对所有的数对都执行完这个操作之后，总共还剩下多少个集合？

限制条件

Small

- $1 \leqslant A \leqslant B \leqslant 100$
- $2 \leqslant P \leqslant B$

Large

- $1 \leqslant A \leqslant B \leqslant 10^{12}$
- $B \leqslant A+1000000$
- $2 \leqslant P \leqslant B$

样例 1

输入

```
A = 10, B = 20, P = 5
```

输出

```
9（{10, 15, 20}在一个集合里，其他的数都恰好各自在一个独立的集合里，共9个集合）
```

样例 2

输入

```
A = 10, B = 20, P = 3
```

输出

```
7（{10, 12, 15, 18, 20}, {11}, {13}, {14}, {16}, {17}, {19}共7个集合）
```

首先因为这是一个集合合并的问题，所以貌似可以使用并查集求解。如果对区间内的任意两个数查询是否包含不小于P的公共质因数，每一次查询由于需要分解质因数所以需要花费 $O(\sqrt{B})$ 的时间。因此，整个算法的时间复杂度就是 $O((B-A)^2\sqrt{B})$ 。虽然可以通过Small数据，但是无法通过Large数据。

我们注意到Large有一个限制条件是$B-A \leqslant 1000000$。如果满足$A \leqslant a \leqslant b \leqslant B$的整数$a$, b有公共质因数p，那么p一定能整除$b-a$。由于$B-A \leqslant 1000000$，因此p也不会超过1000000。

当a和b有公共质因数p时，a和b都是p的倍数。因此，不需要尝试$[A, B]$中所有的数对，只需要遍历所有可能的质因数p，对每一个p合并其所有倍数，就可以更加高效地求得答案。在合并时，从不小于A的最小的p的倍数$((A+p-1)/p \times p)$开始，不断枚举p的倍数直到不超过B的最大的p的倍数$(B/p \times p)$，并把这些数所在的集合合并起来。结合之前对区间筛法和并查集复杂度的分析，可以得出的整个算法的复杂度是$O(B-A)$。

```
typedef long long ll;

int prime[1000000];  // 不超过1000000的第i个的素数
int p;               // 素数的个数

// 输入
ll A, B, P;

void solve() {
  int len = B - A + 1;
  init_union_find(len);  // 初始化并查集

  for (int i = 0; i < p; i++) {
    // 对不小于P的素数进行处理
    if (prime[i] >= P) {
      // 不小于A的最小的prime[i]的倍数
      ll start = (A + prime[i] - 1) / prime[i] * prime[i];
      // 不大于B的最大的prime[i]的倍数
      ll end = B / prime[i] * prime[i];

      for (ll j = start; j <= end; j += prime[i]) {
        // start和j属于同一个集合
        unite(start - A, j - A);
      }

    }
  }
  int res = 0;
  for (ll i = A; i <= B; i++) {
    // find(i) == i时，i就是并查集的根
    // 集合的个数等于根的个数
    if(find(i - A) == i - A) res++;
  }
  printf("%d\n", res);
}
```

3.7.6 Wi-fi Towers

Wi-fi Towers （2009 World Final D）

给定一个无线电塔的网络。对于每座无线电塔，都有一个半径参数，这座无线电塔可以给这个半径范围内的其他无线电塔发送信号。

无线电塔的网络

现在，所有的无线电塔都在以某种古老的“协议A”进行通信不过需要把其中一部分无线电塔升级到一种较先进的新“协议B”。如果某个无线电塔被升级到协议B，则在这个无线电塔的电波范围所能覆盖到的所有无线电塔也都必须要升级到协议B才可以（不过需要注意的是，电波范围覆盖到这座无线电塔的无线电塔不必非升级到协议B）。

考虑到升级的花费和升级后的收益，我们给每座无线电塔打一个分数（正分表示升级之后收益更多，负分表示升级的花费更多）。现在需要选择一些合适的无线电塔进行升级，使得升级的无线电塔的总分最大。假设无线电塔的个数为 n，无线电塔 i 的位置为(x_i, y_i)，电波的半径是 r_i，分数是 s_i。

限制条件

Small dataset

- $1 \leqslant n \leqslant 15$

Large dataset

- $1 \leqslant n \leqslant 500$

样例

输入

```
n = 5
x = {0, 0, 5, 10, 15}
y = {1, -1, 0, 0, 1}
r = {7, 7, 1, 6, 2}
s = {10, 10, -15, 10, -20}
```

输出

```
5
```

这个问题可以归约到s-t最小割问题上。我们加入s, t顶点，然后考虑把这两个顶点分开的割中的最小割。

首先，很自然地想到对于一个割割成的两个顶点集合，让其中一个全部使用“协议B”，另一个全部使用“协议A”。这里不妨假设包含s的集合使用“协议B”，包含t的集合使用“协议A”。

由于是“最小”割，就需要搞清楚最小化的是什么。看起来最小化“损失”是一个比较合理的解释。我们先不考虑无线电塔之间的互相关系，而是先考虑怎么归约到最小割上。在这里，每一个无线电塔对应图中的一个顶点。

分数为负的的无线电塔如果和s相连，也就是升级到了协议B，就产生了损失，因此和s之间连一条容量为分数的绝对值的边。由于如果和t相连，也就是维持协议A不变，就没有收益也没有损失，因此不需要连边（也可以看成是连了一条容量为0的边）。

分数为正的的无线电塔如果从s可达，也就是升级到了协议B，就可以看成是产生了负的损失，但是之后根据最大流最小割定理用最大流求解的话，是不能有容量为负的边的。因此，我们认为分数为正的的无线电塔最开始就是以协议B通讯的，这样使用协议B的损失是0，而如果使用协议A则损失了对应的分数。这样，就可以使得所有边的容量都是正的（注意在求完最小割之后，还需要加上这些分数）。

接下来只需要搞清楚电波之间的关系并加入图中，就可以归约到最小割了。我们要将升级无线电塔i时，其他的无线电塔j也必须升级这一关系加入图中。

需要防止的情况是i和s相连，同时j和t相连。在这种情况下，从i到j的边（如果存在的话）一定属于割集。因此，我们可以从i向j连一条容量是∞的边，这样最小割就不会包含这条边，因为若i和s相连，则j也一定和s相连，从而满足了限制条件。[①]

① 请注意在这个构图中的相反的情况，也就是j和s相连，同时i和t相连是允许的。另外，有向图的割等于从和s相连的顶点集合出发，到和t相连顶点集合的边的容量之和。

综上所述，原问题归约到了最小割上。根据最大流最小割定理，接下来只需要使用合适的最大流算法就可以了。[①]

```
// 输入
int n, x[MAX_N], y[MAX_N], r[MAX_N], s[MAX_N];

int sqr(int x) {
  return x * x;
}

void solve() {
  // 把n作为源点、n+1作为汇点共n+2个顶点的网络
  rep (v, n + 2) G[v].clear();

  int ans = 0;
  for (int i = 0; i < n; i++) {
    if (s[i] < 0) {
      add_edge(n, i, -s[i]);
    }
    else {
      ans += s[i];
      add_edge(i, n + 1, s[i]);
    }

    for (int j = 0; j < n; j++) {
      if (i == j) continue;
      if (sqr(x[i] - x[j]) + sqr(y[i] - y[j]) <= sqr(r[i])) {
        add_edge(j, i, INF);
      }
    }
  }

  ans -= max_flow(n, n + 1);
  printf("%d\n", ans);
}
```

① 这一类问题也被叫做最大权闭合图问题。——译者注

练 习 题

3.1 不光是查找值！“二分搜索”

■ 最大化最小值

POJ 3258: River Hopscotch
POJ 3273: Monthly Expense
POJ 3104: Drying
POJ 3045: Cow Acrobats

■ 最大化平均值

POJ 2976: Dropping tests
POJ 3111: K Best

■ 查找第k大的值

POJ 3579: Median
POJ 3685: Matrix

■ 最小化第k大的值

POJ 2010: Moo University - Financial Aid
POJ 3662: Telephone Lines

■ 其他

POJ 1759: Garland
POJ 3484: Showstopper

3.2 常用技巧精选（一）

■ 尺取法

POJ 2566: Bound Found
POJ 2739: Sum of Consecutive Prime Numbers
POJ 2100: Graveyard Design

■ 反转

POJ 3185: The Water Bowls
POJ 1222: Extended Lights Out

■ 弹性碰撞

POJ 2674: Linear world

■ 折半枚举

POJ 3977: Subset
POJ 2549: Sumsets

■ 坐标离散化

AOJ 0531: Paint Color

3.3 活用各种数据结构

■ Binary Indexed Tree

POJ 1990: MooFest
POJ 3109: Inner Vertices
POJ 2155: Matrix
POJ 2886: Who Gets the Most Candies?

■ 线段树和平方分割

POJ 3264: Balanced Lineup
POJ 3368: Frequent values
POJ 3470: Walls
POJ 1201: Intervals
UVa 11990: "Dynamic" Inversion

3.4 熟练掌握动态规划

■ 状态压缩DP

POJ 2441: Arrange the Bulls
POJ 3254: Corn Fields
POJ 2836: Rectangular Covering
POJ 1795: DNA Laboratory
POJ 3411: Paid Roads

■ 矩阵的幂

POJ 3420: Quad Tiling
POJ 3735: Training little cats

■ 利用数据结构高效求解

POJ 3171: Cleaning Shifts

3.5 借助水流解决问题的网络流

■ 最大流与最小割

POJ 3713: Transferring Sylla
POJ 2987: Firing
POJ 2914: Minimum Cut
POJ 3155: Hard Life

■ 二分图匹配

POJ 1274: The Perfect Stall
POJ 2112: Optimal Milking
POJ 1486: Sorting Slides
POJ 1466: Girls and Boys
POJ 3692: Kindergarten
POJ 2724: Purifying Machine
POJ 2226: Muddy Fields
AOJ 2251: Merry Christmas

■ 最小费用流

POJ 3068: “Shortest” pair of paths
POJ 2195: Going Home
POJ 3422: Kaka's Matrix Travels

（续）

练　习　题	
AOJ 2266: Cache Strategy AOJ 2230: How to Create a Good Game **3.6　与平面和空间打交道的计算几何** **■ 极限情况** POJ 1981: Circle and Points POJ 1418: Viva Confetti AOJ 2201: Immortal Jewels **■ 平面扫描** POJ 3168: Barn Expansion POJ 3293: Rectilinear polygon POJ 2482: Stars in Your Window	**■ 凸包** POJ 1113: Wall POJ 1912: A highway and the seven dwarfs POJ 3608: Bridge Across Islands POJ 2079: Triangle POJ 3246: Game POJ 3689: Equations **■ 数值积分** OJ 2256: Divide the Cake AOJ 2215: Three Silhouettes

第 4 章
登峰造极——高级篇

4.1 更加复杂的数学问题

➡在本节中，我们将对在2.6节中介绍的算法进行拓展，以解决更加复杂的问题。例如线性方程组、线性同余方程等的求解。此外，还会介绍在计数问题中十分有用的容斥原理。

4.1.1 矩阵

1. 线性方程组

求解线性方程组比较容易，在程序设计竞赛中也经常出现这一类问题。接下来要介绍的，是其中一种未知数个数和方程个数相等，并且有唯一解的较为简单的情况。首先，我们试着求解下面的例题。通过手算求解一些方程，可以更好的理解接下来要介绍的算法。

$$x-2y+3z=6 \text{.....(1)}$$
$$4x-5y+6z=12 \text{....(2)}$$
$$7x-8y+10z=21 \text{..(3)}$$

我们希望把这个方程组最终变为这样的形式

$$x+0y+0z=a$$
$$0x+y+0z=b$$
$$0x+0y+z=c$$

因此从(2)中减去对(1)的两边同时乘以4得到的式子，从(3)中减去对(1)的两边同时乘以7得到的式子。这样除了(1)之外的式子里的未知数x就都被消去了。用同样的方法，我们可以得到唯一含有y的式子和唯一含有z的式子。

$$x-2y+3z=6$$
$$3y-6z=-12 \text{.....(2')}$$
$$6y-11z=-21 \text{.....(3')}$$

目标是只使(2')中含有未知数y，消去其他方程中的y。首先把(2')的两边同时除以3。

$$x-2y+3z=6$$
$$y-2z=-4 \text{.....(2'')}$$
$$6y-11z=-21$$

从(1)中减去对(2")的两边同时乘以–2得到的式子，从(3')中减去对(2")的两边同时乘以6得到的式子。

$$x - z = -2 \text{.....(1')}$$
$$y - 2z = -4$$
$$z = 3 \text{.....(3")}$$

从(1')中减去对(3")的两边同时乘以–1得到的式子，从(2")中减去对(3")的两边同时乘以–2得到的式子，就可以求得解(x=1, y=2, z=3)。即使未知数和式子的数量增加了，也可以用完全一样的方法求解。求解的过程中，在消去某个未知数时，打算保留该未知数的式子的对应未知数系数可能是0，在这种情况下，只需要调整方程的顺序，使得对应的系数不为0就可以了（另外，为了减小误差，应该选择要消去的未知数系数的绝对值尽可能大的方程[①]）。

```
const double EPS = 1E-8;
typedef vector<double> vec;
typedef vector<vec> mat;

// 求解Ax=b，其中A是方阵
// 当方程组无解或者有无穷多解时，返回一个长度为0的数组
vec gauss_jordan(const mat& A, const vec& b) {
  int n = A.size();
  mat B(n, vec(n + 1));
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) B[i][j] = A[i][j];
  // 把b存放在A的右边方便一起处理
  for (int i = 0; i < n; i++) B[i][n] = b[i];

  for (int i = 0; i < n; i++) {
    // 把正在处理的未知数系数的绝对值最大的式子换到第i行
    int pivot = i;
    for (int j = i; j < n; j++) {
      if (abs(B[j][i]) > abs(B[pivot][i])) pivot = j;
    }
    swap(B[i], B[pivot]);

    // 无解或者有无穷多解
    if (abs(B[i][i]) < EPS) return vec();

    // 把正在处理的未知数的系数变为1
    for (int j = i + 1; j <= n; j++) B[i][j] /= B[i][i];
    for (int j = 0; j < n; j++) {
      if (i != j) {
        // 从第j个式子中消去第i个未知数
        for (int k = i + 1; k <= n; k++) B[j][k] -= B[j][i] * B[i][k];
      }
    }
  }
```

① 该方法被称为列主元高斯消元法，相应的还有更为数值稳定的全主元高斯消元法。——译者注。

```
    vec x(n);
    // 存放在右边的b就是答案
    for (int i = 0; i < n; i++) x[i] = B[i][n];
    return x;
  }
```

这个算法被称作高斯消元法，复杂度为$O(n^3)$（其中n为方程数）。

对算法进行少许修改之后，就可以处理方程数和未知数的数量不相等或者解不唯一的情况，我们暂不在此介绍。仔细分析算的每一个步骤，也很容易求得行列式的值以及矩阵的秩。具体的方法可以参见讲解线性代数的书。

2. 期望值和方程组

Random Walk

有一个 $N \times M$ 大小的格子。从(0,0)出发，每一步朝着上下左右 4 个格子中可以移动的格子等概率移动。另外有一些格子中有石头，因此无法移至这些格子。求第一次到达(N–1, M–1)格子的期望步数。题目假定至少存在一条从(0,0)出发到(N–1,M–1)的路径。

限制条件

- $2 \leqslant N, M \leqslant 10$

样例 1

输入

N = 10, M = 10（格子如下图所示。'#'和'.'分别表示石头和可以移动到的格子。）

```
..######.#
......#..#
.#.##.##.#
.#........
##.##.####
....#....#
.#######.#
....#.....
.####.####
....#.....
```

输出

```
1678.00000000
```

样例 2

输入

```
N = 10, M = 10

..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
```

输出

```
542.10052168
```

样例 3

输入

```
N = 3, M = 10

.#...#...#
.#.#.#.#.#
...#...#..
```

输出

```
361.00000000
```

设$E(x,y)$表示从(x,y)出发，到终点的期望步数。我们先考虑从(x,y)向上下左右4个方向都可以移动的情况。由于向4个方向的移动都是等概率的，因此可以在$E(x,y)$和$E(x+dx,y+dy)(|dx|+|dy|=1)$之间建立起如下关系。

$$E(x,y)=\frac{1}{4}E(x-1,y)+\frac{1}{4}E(x+1,y)+\frac{1}{4}E(x,y-1)+\frac{1}{4}E(x,y+1)+1$$

如果移动不是等概率的，只需要把1/4改成相应的数值就可以了。

如果存在不能移动的方向，我们也可以列出类似的式子。此外，当$(x,y)=(N-1,M-1)$时，我们有$E(N-1,M-1)=0$。为了使方程有唯一解，我们令有石头的格子和无法到达终点的格子都有$E(x,y)=0$。把得到的方程联立，就可以求解期望步数了。

```
// 输入
char grid[MAX_N][MAX_M + 1];
int N, M;

bool can_goal[MAX_N][MAX_M]; // can_goal[x][y]为true的话，(x,y)可以到达终点
int dx[4] = {-1, 1, 0, 0}, dy[4] = {0, 0, -1, 1};

// 搜索可以达到终点的点
void dfs(int x, int y) {
  can_goal[x][y] = true;
  for (int i = 0; i < 4; i++) {
    int nx = x + dx[i], ny = y + dy[i];
    if (0 <= nx && nx < N && 0 <= ny && ny < M && !can_goal[nx][ny] &&
        grid[nx][ny] != '#') {
      dfs(nx, ny);
    }
  }
}

void solve() {
  mat A(N * M, vec(N * M, 0));
  vec b(N * M, 0);
  for (int x = 0; x < N; x++) {
    for (int y = 0; y < M; y++) {
      can_goal[x][y] = false;
    }
  }
  dfs(N - 1, M - 1);

  // 构建矩阵
  for (int x = 0; x < N; x++) {
    for (int y = 0; y < M; y++) {

      // 到达终点，或者是(x,y)无法到达终点的情况
      if (x == N - 1 && y == M - 1 || !can_goal[x][y]) {
        A[x * M + y][x * M + y] = 1;
        continue;
      }

      // 其余情况
      int move = 0;
      for (int k = 0; k < 4; k++) {
        int nx = x + dx[k], ny = y + dy[k];
        if (0 <= nx && nx < N && 0 <= ny && ny < M && grid[nx][ny] == '.') {
          A[x * M + y][nx * M + ny] = -1;
          move++;
        }
      }
      b[x * M + y] = A[x * M + y][x * M + y] = move;
    }
  }
  vec res = gauss_jordan(A, b);
  printf("%.8f\n", res[0]);
}
```

4.1.2 模运算的世界

1. 逆元

接下来，我们来学习如何求解线性同余方程。让我们考虑如何求解线性方程$ax \equiv b \pmod m$。对于实数运算下的方程$ax=b$，由于a存在倒数，因此很容易求解。如果在mod m的运算下，也有像满足$ay \equiv 1 \pmod m$这样的a的倒数一样的数存在，方程就可以求解了。我们把这样的y叫做a的逆元，记作a^{-1}。如果能求解逆元，那么就有$x=a^{-1} \times xa=a^{-1} \times b$，也就可以求出$x$了。由于方程$ax \equiv 1 \pmod m$等价于存在整数$k$使得$ax=1+mk$，因此稍作变形之后，可以变为求解满足$ax-mk=1$的$x$的问题。这个问题可以使用extgcd求解。同时，也可以知道如果$\gcd(a,m)!=1$，那么逆元是不存在的。

```
int mod_inverse(int a, int m) {
  int x, y;
  extgcd(a, m, x, y);
  return (m + x % m) % m;
}
```

如果a和m不互素，那么$ax \equiv b \pmod m$就等价于

$$(a/\gcd(a,m))x \equiv b/\gcd(a,m) \pmod{m/\gcd(a,m)}$$

从式子中可以看出，当b无法整除$\gcd(a,m)$时，原方程无解。在有解的情况下，我们有

$$x \equiv (a/\gcd(a,m))^{-1} \times (b/\gcd(a,m)) \pmod{m/\gcd(a,m)}$$

因此，$ax \equiv b \pmod m$的解为

$$x \equiv (a/\gcd(a,m))^{-1} \times (b/\gcd(a,m)) + (m/\gcd(a,m)) \times k \pmod m \ (0 \leqslant k < \gcd(a,m))$$

需要注意的是这里和实数的情况有所不同，有可能有多解，也有可能无解。

2. 费马小定理

在p是素数的情况下，对任意整数x都有$x^p \equiv x \pmod p$。这个定理被称作费马小定理。其中如果x无法被p整除，我们有$x^{p-1} \equiv 1 \pmod p$。利用这条性质，在p是素数的情况下，就很容易求出一个数的逆元。把上面的式子变形之后得到$a^{-1} \equiv a^{p-2} \pmod p$，因此可以通过快速幂运算求出逆元。

在不是素数的情况下，我们也有类似的欧拉定理可以使用。假设$m = p_1^{e_1} p_2^{e_2} \ldots p_n^{e_n}$，那么$m$的欧拉函数$\varphi(m)$的定义如下

$$\varphi(m)=m \times \Pi(p_i-1)/p_i \text{（}\Pi\text{为连乘符号）}$$

欧拉函数的值等于不超过m并且和m互素的数的个数。此时对于和m互素的x，有$x^{\varphi(m)} \equiv 1 \pmod m$

成立[1]。在m是素数时，根据定义$\varphi(m)=m-1$，因此欧拉定理也可以看作是费马小定理的推广。

因为n的整数分解可以在$O(\sqrt{n})$时间内完成，所以对于某一个n的欧拉函数也可以在$O(\sqrt{n})$时间内求得。另外，我们可以利用埃氏筛法，每次发现质因子时就把它的倍数的欧拉函数乘上$(p-1)/p$，这样就可以一次性求出$1\sim n$的欧拉函数值的表了。

```
// 求欧拉函数值。O(√n)
int euler_phi(int n) {
  int res = n;
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
      res = res / i * (i - 1);
      for (; n % i == 0; n /= i);
    }
  }
  if (n != 1) res = res / n * (n - 1);
  return res;
}

int euler[MAX_N];

// O(MAX_N)时间筛出欧拉函数值的表
void euler_phi2() {
  for (int i = 0; i < MAX_N; i++) euler[i] = i;
  for (int i = 2; i < MAX_N; i++) {
    if (euler[i] == i) {
      for (int j = i; j < MAX_N; j += i) euler[j] = euler[j] / i * (i - 1);
    }
  }
}
```

3. 线性同余方程组

下面给大家介绍一下如何求解由多条线性同余方程联立得到的线性同余方程组。用数学化的符号表示就是求解$a_i\times x\equiv b_i \pmod{m_i}$ $(1\leqslant i\leqslant n)$这样的方程组。如果方程组有解，那么一定有无穷多解，而且解的全集一定可以写成$x\equiv b \pmod{m}$的形式，因此问题就转化为求解b和m。如果我们能求解方程组$x\equiv b_1 \pmod{m_1}$，$a\times x\equiv b_2 \pmod{m_2}$，那么只要对方程逐个求解，对于任意有限的$n$我们就都可以求出答案了。

因为$x\equiv b_1 \pmod{m_1}$，所以可以把x写成$x=b_1+m_1\times t$的形式。把它代入第二条式子，可以得到

$$a(b_1+m_1\times t)\equiv b_2 \pmod{m_2}$$

移项整理后得到

$$a\times m_1\times t\equiv b_2-a\times b_1 \pmod{m_2}$$

① 当x和m不互素时，则有，存在一个$h(x)$，对任意$y\geqslant h(x)$均有$x^{y+\varphi(m)}\equiv x^y \pmod{m}$成立。

由于这只是一个一次同余方程，因此很容易求解。当$\gcd(m_2, a\times m_1)$无法整除$b_2-a\times b_1$时原方程组无解。

```
// 返回一个(b, m)的数对
pair<int, int> linear_congruence(const vector<int>& A, const vector<int>& B,
                                 const vector<int>& M) {
  // 由于最开始没有任何限制，所以先把解设为表示所有整数的x≡0 (mod 1)
  int x = 0, m = 1;

  for (int i = 0; i < A.size(); i++) {
    int a = A[i] * m, b = B[i] - A[i] * x, d = gcd(M[i], a);
    if (b % d != 0) return make_pair(0, -1); // 无解
    int t = b / d * mod_inverse(a / d, M[i] / d) % (M[i] / d);
    x = x + m * t;
    m *= M[i] / d;
  }
  return make_pair(x % m, m);
}
```

4. 中国剩余定理

我们假设同余方程组里所有的a_i都等于1，并且所有的m_i都互素。这样，答案就一定是$x\equiv b \pmod{\Pi m_i}$。反之，对于一个合数n，我们假设有$n=ab$（其中a和b互素）。那么如果$x \bmod n$的值确定，$x \bmod a$和$x \bmod b$的值就都确定了。也就是说，我们有$(x \bmod n) \Leftrightarrow (x \bmod a, x \bmod b)$这样一组对应关系。

换句话说，以合数n为模数来考虑与以a和b为模数来考虑是等价的。这个定理叫做中国剩余定理。通过对n进行分解，对于模合数的情况只需要考虑模p^k（p为素数）的情况就可以了。其中，如果n不能被任何一个完全平方数整除[①]，那么问题就可以转化为模数为素数的情况，从而变得容易求解。中国剩余定理不是一个算法，而是可以看成在思考算法时的一个提示

例：$f(x) \equiv 0 \pmod n \Leftrightarrow f(x) \equiv 0 \pmod{p^k}$ $(p^k|n)$

5. n!（n的阶乘）

在计数问题中，经常需要用到$n!$。在学完前面介绍的内容之后，有必要了解$n!$在mod p下的一些性质。下面我们假设p是素数，$n!=a\,p^e$（a无法被p整除），并试图求解$a \bmod p$和e。e是$n!$能够迭代整除p的次数，因此可以使用下面的式子进行计算。

$$n/p+n/p^2+n/p^3+\cdots$$

这个结论很显然，因为n/d和不超过n的能被d整除的正整数的个数相等。由于只需要对于$p^t\leqslant n$的t进行计算，因此复杂度是$O(\log_p n)$。

接下来计算$a \bmod p$。首先计算$n!=1\times2\times\cdots\times n$的因数中不能被$p$整除的项的积。假设$n=10, p=3$，则

① 这样的数叫做Square-free number。——译者注

$n!=1\times2\times4\times5\times7\times8\times10\times(3\times6\times9)$

$1\times2\times4\times5\times7\times8\times10\equiv1\times2\times1\times2\times1\times2\times1 \pmod p$

从这个例子中可以看出，不能被p整除的项在mod p下呈周期性。因此，不能被p整除的项的积等于$(p-1)!^{(n/p)}\times(n \bmod p)!$。事实上，根据威尔逊定理，我们有$(p-1)!\equiv-1$。因为除了1和$p-1$之外其余的项都可以和各自的逆元相乘得到1。

接下来，计算可以被p整除的项的积。可以被p整除的项有$p, 2p, 3p, \cdots, (n/p)p$。把这些项分别除以$p$之后得到$1, 2, 3, \cdots, n/p$。因此，问题的范围就由$n$缩小到了$n/p$。如果预处理出$0\leqslant n<p$范围中$n!$ mod p的表，就可以在$O(\log_p n)$时间内算出答案。如果不预处理，那么复杂度是$O(p\log_p n)$。

```
int fact[MAX_P]; // 预处理的n! mod p的表。O(p)。

// 分解n!≡a p^e，返回a mod p。O(log_p n)。
int mod_fact(int n, int p, int& e) {
  e = 0;
  if (n == 0) return 1;

  // 计算p的倍数的部分
  int res = mod_fact(n / p, p, e);
  e += n / p;

  // 由于(p-1)!≡-1，因此(p-1)!^(n/p)只需要知道n/p的奇偶性就可以计算了。
  if (n / p % 2 != 0) return res * (p - fact[n % p]) % p;
  return res * fact[n % p] % p;
}
```

6. C_n^k mod p

了解了$n!$在mod p下的性质之后，C_n^k mod p也就可以计算了。首先，把C_n^k写成$n!$的积的形式。

$$C_n^k=\frac{n!}{k!(n-k)!}$$

设$n!=a_1p_1^{e_1}, k!=a_2p_2^{e_2}, (n-k)!=a_3p_3^{e_3}$。从式子中可以看出，当$e_1>e_2+e_3$时，$C_n^k$可以被$p$整除，$e_1=e_2+e_3$时无法被$p$整除。在无法整除的情况下，$C_n^k\equiv a_1(a_2a_3)^{-1}$。

```
// 求nCk mod p。O(log_p n)。
int mod_comb(int n, int k, int p) {
  if (n < 0 || k < 0 || n < k) return 0;
  int e1, e2, e3;
  int a1 = mod_fact(n, p, e1), a2 = mod_fact(k, p, e2), a3 = mod_fact(n - k, p, e3);
  if (e1 > e2 + e3) return 0;
  return a1 * mod_inverse(a2 * a3 % p, p) % p;
}
```

另外，如果把 C_n^k 画成杨辉三角形的样子，会发现得到的图形具有自相似的性质。也可以根据这个性质求出 C_n^k。设 $n=\Sigma n_i p^i$, $k=\Sigma k_i p^i$（表示成 p 进制），则 $C_n^k \equiv \Pi_{n_i} C_{k_i} \pmod p$。[①]

mod 2的情况（白色三角形表示0，黑色三角形表示1）

4.1.3 计数

1. 容斥原理

容斥原理

给定 $a_1, a_2, \cdots, a_m$，求 1 到 n 的整数中至少能整除 a 中一个元素的数有几个？

限制条件

- $1 \leqslant n \leqslant 10^9$
- $1 \leqslant m \leqslant 15$

样例 1

输入

```
n = 100, m = 2
a = {2, 3}
```

输出

```
67（2的倍数有50个，3的倍数有33个，6的倍数有16个，因此50+33-16=37个）
```

样例 2

输入

```
n = 100, m = 3
a = {2, 3, 7}
```

① 这个定理被称作Lucas定理。——译者注

输出

```
72
```

如果对于1到n中的每个数都判断是否满足条件，那么复杂度是$O(n\times m)$，无法在时间限制之内得出答案。由于m的范围比较小，我们可以试着对这一条件加以利用。

首先我们考虑一下m=1的情况。此时，答案是n/a_1，因此很容易求解。当m=2时，如果直接计算$n/a_1+n/a_2$，就会发现$\mathrm{lcm}(a_1,a_2)$的倍数被计算了2遍，因此减掉重复计算的部分就得到了$n/a_1+n/a_2-n/\mathrm{lcm}(a_1,a_2)$。依次类推，就可以发现对于一般情况的计算方法。

文氏图

下面我们就来试着写出一般情况的计算公式。假设$A_i(1\leqslant i\leqslant m)$是$X$的一些子集，我们要求$|\cup A_i|$。例如在上面的问题里，就可以看成$A_i=\{n|n$可以整除$a_i\}$。此时，有如下等式成立。

$$|\bigcup A_i|=\sum_{1\leqslant i\leqslant m}|A_i|-\sum_{1\leqslant i<j\leqslant m}|A_i\cap A_j|+\sum_{1\leqslant i<j<k\leqslant m}|A_i\cap A_j\cap A_k|-\cdots+(-1)^{m+1}\sum|A_1\cap A_2\cap\cdots\cap A_m|$$

这个式子被称作容斥原理。一般在集合的个数m比较小，并且$|A_i\cap A_j\cap\cdots\cap A_k|$这些项容易计算时适合使用容斥原理。假设计算一个$|A_i\cap A_j\cap\cdots\cap A_k|$需要花费$O(f)$的时间，那么总共就需要花费$O(2^m f)$的时间。不过，这是最坏情况下的花费。如果满足一些特殊性质，那么可能并不需要枚举所有的2^m种情况，或者可以使用DP来计算。

特殊情况下的文氏图

```
typedef long long ll;

int a[MAX_M];
int n, m;
void solve() {
```

```
  ll res = 0;
  for (int i = 1; i < (1 << m); i++) {
    int num = 0;
    for (int j = i; j != 0; j >>= 1) num += j & 1;  // i的二进制表示中1的数量
    ll lcm = 1;
    for (int j = 0; j < m; j++) {
      if (i >> j & 1) {
        lcm = lcm / gcd(lcm, a[j]) * a[j];
        // 如果lcm大于n，则n/lcm=0。因此在溢出之前break。
        if(lcm > n) break;
      }
    }
    if (num % 2 == 0) res -= n / lcm;
    else res += n / lcm;
  }
  printf("%d\n", res);
}
```

2. 莫比乌斯函数

请思考下面的问题。

没有周期性的字符串的计数

求所有由 a~z 组成的（不一定要使用所有字母）长度为 n 的字符串中，没有周期性的字符串的个数。如果一个长度为 n 的字符串可以通过一个长度为 $m<n$ 的字符串重复 n/m 次后得到，我们就称它具有周期性。

限制条件

- $2 \leqslant n \leqslant 10^9$

样例 1

输入

```
n = 2
```

输出

```
650（满足条件的是像aa和bb这样相同字母连续的字符串之外的其他字符串，因此有26*25=650种）
```

样例 2

输入

```
n = 4
```

输出

```
5895（从26⁴种可能中除去abab形式的字符串之后，aaaa形式的字符串也被除去了，因此共有26⁴-26²=456300种）
```

样例 3

输入

```
n = 15315300
```

输出

```
334
```

我们定义一个字符串的最小循环节的长度为这个字符串的周期。例如：abab的循环节是ab，周期是2，abcabcabcabc的循环节是abc，周期是3。为了记述方便，我们令X^m表示把X重复m次之后拼接得到的字符串（例如：$(ab)^2$=abab）。假设d是n的一个约数，下面我们试着统计满足题目条件的周期为d的约数的字符串的个数。显然，把任意长度为d的字符串作为X代入$X^{(n/d)}$中得到的字符串的周期一定是d的约数，所以周期为d的约数的字符串共有26^d个。

由于我们要求的是周期恰好为n的字符串的个数，因此貌似可以使用容斥原理求解。但是当我们试着使用容斥原理求解时就会发现，约数的个数虽然不算多，但是也可能达到100个或者200个，如果直接套用容斥原理的话无法在规定时间内得出答案。

可以发现周期为d的约数的字符串组成的集合和周期为e的约数的字符串组成的集合的交集是周期为gcd(d,e)的约数的字符串组成的集合。根据这个性质，在容斥原理中加上或者减去的2^m个集合对应的项中，只有n的约数个数种不同。所以，只需要对每个集合分别计算加了多少次和减了多少次，就可以在O(约数个数)时间内算出答案。

包含关系

可以知道d是n的约数时，被加减了多次的总和（26^d的系数）只和n/d相关，而与n无关。求这个系数的函数叫做莫比乌斯函数，记作$\mu(n/d)$。莫比乌斯函数满足下面的关系式。这个式子被叫做莫比乌斯反演公式。

$$f(n)=\sum_{d|n}g(d) \Leftrightarrow g(n)=\sum_{d|n}\mu(\frac{n}{d})f(d)$$

在这个问题中，$f(d)$是周期为d的约数的字符串的个数，$g(d)$是周期恰好为d的字符串的个数。由于$f(d)$已经可以高效地求出了，因此就可以用右边的式子算出$g(n)$。莫比乌斯函数可以通过如下方法计算得到。

- 若n可以被除1以外的完全平方数整除，$\mu(n)=0$。
- 否则设n的质因数的个数为k，$\mu(n)=(-1)^k$。

例如因为1的质因数的个数是0，所以$\mu(1)=1$，$30=2\times3\times5$共有3个质因数，所以$\mu(30)=-1$，$12=4\times3$可以被完全平方数整除，所以$\mu(12)=0$。由于整数分解可以在$O(\sqrt{n})$时间内完成，所以$\mu(n)$也就可以在$O(\sqrt{n})$时间内计算出来。另外，如果使用埃氏筛法，可以在$O(n)$时间内求出1~n所有的μ的值。

在本题中，由于n很大，所以无法预处理出一个1~n的表。对于所有的约数d计算$\mu(d)$只需要花费O(约数的个数$\times\sqrt{n}$)的时间，可以在规定时间内出解（$n\leqslant10^9$的约数个数的最大值是800左右）。不过，由于计算出了n的整数分解，对于n的所有约数d的$\mu(d)$的值也就可以很容易地计算出来，因此实际上只需要$O(\sqrt{n})$时间就可以了。假如$n=60=4\times3\times5$的话，就可以按照如下方法计算。

$\mu(1)=\mu(2\times3)=\mu(2\times5)=(3\times5)=1$

$\mu(2)=\mu(3)=\mu(5)=\mu(2\times3\times5)=-1$

其他为0

```
// 把n的约数的莫比乌斯函数值用map的形式返回。O(√n)
map<int, int> moebius(int n) {
  map<int, int> res;
  vector<int> primes;

  // 枚举n的质因数
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
      primes.push_back(i);
      while (n % i == 0) n /= i;
    }
  }
  if (n != 1) primes.push_back(n);

  int m = primes.size();
  for (int i = 0; i < (1 << m); i++) { // 虽然要执行2^m次，但是这不超过n的约数个数
    int mu = 1, d = 1;
    for (int j = 0; j < m; j++) {
      if (i >> j & 1) {
        mu *= -1;
        d *= primes[j];
```

```
      }
    }
    res[d] = mu;
  }
  return res;
}

const int MOD = 10009;

// 输入
int n;

void solve() {
  int res = 0;
  map<int, int> mu = moebius(n);
  for (map<int, int>::iterator it = mu.begin(); it != mu.end(); ++it) {
    res += it->second * mod_pow(26, n / it->first, MOD); // μ(d)*26^(n/d)
    res = (res % MOD + MOD) % MOD;
  }
  printf("%d\n", res);
}
```

4.1.4　具有对称性的计数

Pólya计数定理

在组合问题中，有时要求把旋转和翻转之后相同的状态看成本质相同的状态，并要求计算本质不同的状态个数。能够在这一类问题中发挥作用的就是Pólya定计数理。

让我们先来看一个简单的问题。用k种颜色给一个2×2的格子染色一共有多少种方法？旋转之后相同的染色方案看作同一种。

我们把旋转分成0度、90度、180度、270度四种情况分别计算。

(1) 旋转0度之后不变的染色方案有k^4种。
(2) 旋转90度之后不变的染色方案是所有格子都染相同颜色，共k种。
(3) 旋转180度之后不变的染色方案是对角线的格子染相同颜色，共k^2种。
(4) 旋转270度之后不变的染色方案是所有格子都染相同颜色，共k种。

实际上，取这4个值的平均值$(k^4+k^2+2\times k)/4$就是答案了。

让我们思考一下为什么可以这样计算。下图中(a)形式的染色方案在(1)中被重复计算了4次。(b)形式的染色方案在(1)中计算了2次，在(3)中计算了2次，共被重复计算了4次。(c)形式的染色方案

在(1)中计算了1次，(2)中计算了1次，(3)中计算了1次，(4)中计算了1次，共被重复计算了4次。这三种情况都被重复计算了4次，因此除以4就是最后答案了。

和容斥原理不同，这里先把所有方案重复计算了相同的次数，然后再把结果除以重复的次数。像这样的计数方法，被称作Pólya计数定理。

下面我们试着解决一些更复杂的问题。

石头染色方案计数问题

有 n 块石头排成一圈。现在要用 m 种颜色染这 n 块石头，问一共有多少种不同的染色方案。旋转之后相同的方案视作同一种。输出方案数 mod 1000000007 之后的结果。

限制条件

- $1 \leqslant n \leqslant 10^9$
- $1 \leqslant m \leqslant 10^9$

样例 1

输入

```
n = 2
m = 10
```

输出

```
55（2块石头染不同颜色的方案有10×9/2=45种，染相同颜色的方案有10种，共55种）
```

样例 2

输入

```
n = 4
m = 10
```

输出

```
2530（之前的问题的k=10的情况）
```

样例 3

输入

```
n = 4
m = 2
```

输出

```
6
```

样例 4

输入

```
n = 1000000000
m = 1000000000
```

输出

```
898487047
```

n=4的情况我们已经解决了。下面我们试着用同样的方法求解这个问题。首先考虑旋转的种类。显然，共有旋转0个位置、旋转1个位置、旋转2个位置……旋转$n-1$个位置等n种转法。

下面我们计算旋转k个位置之后和原来相同的染色方案的个数。首先我们按照顺时针顺序从0到$n-1$给石头编号。由于旋转k个位置之后和原来相同，所以第i个石头和第$(i+k) \bmod n$个石头的颜色相同。这么递推下去可以知道第i个石头和第$(i+k\times t) \bmod n$个石头的颜色相同。求解$k\times t=0 \pmod n$的最小的t。很显然，$t=n/\gcd(k,n)$满足条件，并且是最小的。

颜色一定和第i个石头相同的石头，可以像下面这样不断旋转来枚举得到

$$i\to(i+k) \bmod n\to(i+2k) \bmod n\to\cdots\to i$$

颜色一定和第i个石头相同的石头的集合叫做i的轨迹。很显然，i的轨迹和$(i+k) \bmod n$的轨迹是完全相同的。我们可以把n个石头划分为若干条互不相交的轨迹。而旋转之后和原来相同的染色方案就是将每一条轨迹里所有的石头都染同一种颜色。因此，只要知道轨迹的个数，就可以算出有多少种染色方案旋转之后和原来相同了。

接下来，我们求解旋转k个位置情况下的个数。每一个轨迹中石头的个数是$t=n/\gcd(k,n)$。由于在本题中不同轨迹中的石头的个数相同，所以轨迹的个数就是$n/t=\gcd(k,n)$。因此，旋转k个位置之后和原来相同的染色方案数就是$m^{\gcd(k,n)}$。

n=6,k=3时的轨迹

计算最后答案可以不断枚举k的值，并把$m^{\gcd(k,n)}$加起来。但是因为$n \leqslant 10^9$，如果老老实实按照$0 \sim n-1$的顺序枚举，无法在规定时间内出解。不过，$\gcd(k,n)$只有有限个不同的值，所以可以把相同的值合并在一起计算。

设d是n的约数。我们要统计满足$\gcd(k,n)=d$的$k(0 \leqslant k<n)$的个数。很显然，k是d的倍数，所以可以把k写做$k=d \times t(0 \leqslant t<n/d)$。其中$d=\gcd(k,n)=\gcd(d \times t,n)=d \times \gcd(t,n/d)$。因此，$\gcd(t,n/d)=1$，满足条件的$t$的个数就是欧拉函数$\varphi(n/d)$。

把所有的值加起来之后除以n就是答案了。计算的表达式为

$$\frac{1}{n}\sum_{d|n} m^d \varphi(n/d)$$

在计算这个表达式时，需要对n的所有约数d求解欧拉函数$\varphi(d)$。但是，由于d的质因数也是n的质因数，如果事先求出了n的质因数，那么$\varphi(d)$就很容易求出了。n的质因数的个数有$O(\log n)$个，因此计算每个$\varphi(d)$需要$O(\log n)$时间。

```
const int MOD = 1000000007;
typedef long long ll;

// 输入
int n, m;

void solve() {
  map<int, int> primes = prime_factor(n);
  vector<int> divs = divisor(n);
  ll res = 0;
```

```
    for (int i = 0; i < divs.size(); i++) {

      // 求divs[i]的欧拉函数值
      ll euler = divs[i];
      for (map<int, int>::iterator it = primes.begin(); it != primes.end(); ++it) {
        int p = it->first;
        if (divs[i] % p == 0) euler = euler / p * (p - 1);
      }

      res += euler * mod_pow(m, n / divs[i], MOD) % MOD;
      res %= MOD;
    }

    // 最后除以n
    printf("%lld\n", res * mod_pow(n, MOD - 2, MOD) % MOD);
  }
```

实际上，这个问题也可以使用莫比乌斯函数求解。设$f(d)$为周期恰好为d的染色方案的总数，其中即使旋转后和原来相同也当作不同的方案计算。$f(d)$可以使用“没有周期性的字符串的计数”问题中用过的方法求解。如果再考虑上旋转，$f(d)$中的每一种染色方案都恰好被算了d次。因此，答案就是

$$\sum_{d|n} f(d)/d$$

设$d(n)$表示n的约数的个数，那么这个方法中对于一个d求$f(d)$的复杂度是$O(d(n))$，一共要计算n的约数个数那么多次，所以总的复杂度是$O(d(n)^2+\sqrt{n})$。而且，把这个式子变形并化简之后，就有

$$\sum_{d|n}\frac{f(d)}{d}=\sum_{d|n}\frac{1}{d}\sum_{e|d}\mu(\frac{d}{e})m^e=\sum_{e|n}m^e\sum_{d|\frac{n}{e}}\frac{\mu(d)}{ed}=\frac{1}{n}\sum_{d|n}m^{\frac{n}{d}}\sum_{e|d}\mu(\frac{d}{e})e=\frac{1}{n}\sum_{d|n}m^{\frac{n}{d}}\varphi(d)$$

这和通过Pólya计数定理得到的结果是完全相同的。

附有把翻转之后得到的方案看作同一方案等条件的问题，以及立方体的染色问题等都可以利用Pólya定理求解。另外，有同样颜色的石头数量有上限限制、相邻的石头不能染成相同的颜色等附加约束的问题也可以求解。

4.2 找出游戏的必胜策略

➡本节，我们来讨论一下双人对战游戏中的必胜策略，并介绍Nim游戏和Grundy值[①]等重要概念。

4.2.1 游戏与必胜策略

1. 硬币游戏1

硬币游戏 1

Alice 和 Bob 在玩这样一个游戏。给定 k 个数字 $a_1,a_2,\cdots,a_k$。一开始，有 x 枚硬币，Alice 和 Bob 轮流取硬币。每次所取硬币的枚数一定要在 $a_1,a_2,\cdots,a_k$ 当中。Alice 先取，取走最后一枚硬币的一方获胜。当双方都采取最优策略时，谁会获胜？题目假定 $a_1,a_2,\cdots,a_k$ 中一定有 1。

⚠限制条件

- $1\leqslant x\leqslant 10000$
- $1\leqslant k\leqslant 100$
- $1\leqslant a_i\leqslant x$

样例 1

输入

```
x = 9
k = 2
a = {1, 4}
```

输出

```
Alice
```

① Grundy值又叫Nim值，国内则更常称为Sprague-Grundy函数。——译者注

样例 2

输入

```
x = 10
k = 2
a = {1, 4}
```

输出

```
Bob
```

下面来考虑当轮到自己时，还有j枚硬币时的胜负情况。

- 题目规定取光所有的硬币就获胜，这等价于轮到自己时如果没有硬币了就失败。因此，j=0时是必败态。
- 如果对于某个$i(1 \leqslant i \leqslant k)$，$j-a_i$是必败态的话，$j$就是必胜态。（如果当前有$j$枚硬币，只要取走$a_i$枚对手就必败→自己必胜）。
- 如果对于任意的$i(1 \leqslant i \leqslant k)$，$j-a_i$都是必胜态的话，$j$就是必败态。（不论怎么取，对手都必胜→自己必败）。

根据这些规则，我们就能利用动态规划算法按照j从小到大的顺序计算必胜态必败态。只要看x是必胜态还是必败态，就能知道谁会获胜了。

像这样，通过考虑各个状态的胜负条件，判断必胜态和必败态，是有胜败的游戏的基础。

```
// 输入
int X, K, A[MAX_K];

// 动态规划所用的数组
bool win[MAX_X + 1];

void solve() {
  // 轮到自己时没有硬币了，则失败
  win[0] = false;

  for (int j = 1; j <= X; j++) {
    // 如果可以让对手到达必败态，则必胜
    win[j] = false;
    for (int i = 0; i < K; i++) {
      win[j] |= A[i] <= j && !win[j - A[i]];
    }
  }

  if (win[X]) puts("Alice");
  else puts("Bob");
}
```

2. A Funny Game

A Funny Game （POJ 2484）

n 枚硬币排成一个圈。Alice 和 Bob 轮流从中取一枚或两枚硬币。不过，取两枚时，所取的两枚硬币必须是连续的。硬币取走之后留下空位，相隔空位的硬币视为不连续的。Alice 开始先取，取走最后一枚硬币的一方获胜。当双方都采取最优策略时，谁会获胜？

例子

限制条件

- $0 \leqslant n \leqslant 1000000$

样例 1

输入

```
n = 1
```

输出

```
Alice
```

样例 2

输入

```
n = 3
```

输出

```
Bob
```

n高达1000000，考虑到还有将连续部分分裂成几段等的情况，状态数非常地多，搜索和动态规划法都难以胜任。需要更加巧妙地判断胜败关系。

首先，试想一下如下情况。能够把所有的硬币分成像下图这样的两个完全相同的组的状态，是必胜状态？还是必败状态？

被分成两个相同的组的状态的例子

事实上这是必败态。不论自己采取什么选取策略，对手只要在另一组采取相同的策略，就又回到了分成两个相同的组的状态。

必定能够再次回到同样的情况

不断这样循环下去，总会在某次轮到自己时没有硬币了。也就是说，因为对手取走了最后一枚硬币而败北。

接下来，让我们回到正题。Alice在第一步取走了一枚或两枚硬币之后，原本成圈的硬币就变成了长度为$n-1$或是$n-2$的链。这样只要Bob在中间位置，根据链长的奇偶性，取走一枚或两枚硬币，就可以把所有硬币正好分成了两个长度相同的链。

Bob总是能够将硬币分成两个长度相同的链

这正如我们前面所讨论的一样，是必败态。也就是说，Alice必败，而Bob必胜。只不过，当$n \leqslant 2$时，Alice可以在第一步取光，所以胜利的是Alice。在这类游戏当中，作出对称的状态后再完全模仿对手的策略常常是有效的。

```
// 输入
int n;

void solve() {
  if (n <= 2) puts("Alice");
  else puts("Bob");
}
```

3. Euclid's Game

Euclid's Game （POJ 2348）

让我们看一下这个以辗转相除法为基础的游戏。

给定两个整数 a 和 b。Stan 和 Ollie 轮流从较大的数字中减去较小数字的倍数。这里的倍数指的是 1 倍、2 倍等这样的正整数倍，并且相减后的结果不能小于零。Stan 先手，在自己的回合将其中一个数变为零的一方获胜。当双方都采取最优策略时，谁会获胜？

限制条件

- a 和 b 都是正整数（在 32 位有符号整数范围之内）

样例 1

输入

```
a = 34, b = 12
```

输出

```
Stan wins
```

样例 2

输入

```
a = 15, b = 24
```

输出

```
Ollie wins
```

让我们来找找看该问题中必胜态和必败态的规律。首先，如果$a>b$则交换，假设$a<b$。另外，如果b已经是a的倍数了则必胜，所以假设b并非a的倍数。此时，a和b的关系，按自由度的观点，可以分成以下两类。

(1) b-a<a的情况
(2) b-a>a的情况

对于第一种情况，如果从b中减去a的2倍及以上的数则变为负数，所以只能从b中减去a，没有选择的余地。相对的，对于第二种情况，有从b中减去a，减去2a，或更高的倍数等多种选择。

对于第一种情况，要判断必胜还是必败是很简单的。因为没有选择的余地，如果b减去a之后所得到的状态是必败态的话，它就是必胜态，如果得到的是必胜态的话，它就是必败态。

例如，从(4, 7)这个状态出发就完全没有选择的机会，按照

(4, 7) → (4, 3) → (1, 3)

的顺序，轮到(1, 3)的一方将获胜，所以有

(4, 7) → (4, 3) → (1, 3)
必胜 → 必败 → 必胜

可见(4, 7)是必胜态。

接下来，我们来看一下第二种情况是必胜态还是必败态。假设x是使得$b-ax<a$的整数，考虑一下从b中减去$a(x-1)$的情况。例如对于(4, 19)则减去12。

此时，接下来的状态就成了前边讲过的没有选择余地的第一种情况。如果该状态是必败态的话，当前状态就是必胜态。

那么，如果减去$a(x-1)$后的状态是必胜态的话，该如何是好呢？此时，从b中减去ax后的状态是减去$a(x-1)$后的状态唯一可以转移到的状态，根据假设，减去$a(x-1)$后的状态是必胜态，所以该状态是必败态。因此，当前状态是必胜态。

例如对于(4, 17)，由于从17中减去12得到的(4, 5)就是必败态，所以只要减去12就能获胜。另一方面，对于(4, 19)，减去12得到的(4, 7)是必胜态，因此(4, 3)就是必败态，只要减去16就能获胜。

由此可知，第二种情况总是必胜的。所以，从初始状态开始，最先达到有自由度的第二种状态的一方必胜。

```
// 输入
int a, b;


void solve() {
  bool f = true;

  for (;;) {
    if (a > b) swap(a, b);
```

```
    // b是a的倍数时必胜
    if (b % a == 0) break;

    // 如果是解说中的第二种情况必胜
    if (b - a > a) break;

    b -= a;
    f = !f;
  }

  if (f) puts("Stan wins");
  else puts("Ollie wins");
}
```

4.2.2 Nim

1. Nim

Nim

有 n 堆石子，每堆各有 a_i 颗石子。Alice 和 Bob 轮流从非空的石子堆中取走至少一颗石子。Alice 先取，取光所有石子的一方获胜。当双方都采取最优策略时，谁会获胜？

Nim的样子

限制条件

- $1 \leqslant n \leqslant 1000000$
- $1 \leqslant a_i \leqslant 10^9$

样例 1

输入

```
n = 3
a = {1, 2, 4}
```

输出

```
Alice
```

样例 2

输入

```
n = 3
a = {1, 2, 3}
```

输出

```
Bob
```

这个游戏是称为Nim的经典游戏，该游戏的策略也成为了许多游戏的基础。要判断该游戏的胜负只要用异或运算就好了。有以下结论成立。

$$a_1 \text{ XOR } a_2 \text{ XOR } \cdots \text{ XOR } a_n \neq 0 \rightarrow \text{必胜态}$$
$$a_1 \text{ XOR } a_2 \text{ XOR } \cdots \text{ XOR } a_n = 0 \rightarrow \text{必败态}$$

因此，只要计算异或值，如果非零则Alice必胜，为零则Bob必胜。

让我们来简略地证明一下。首先一旦从XOR为零的状态取走至少一颗石子，XOR就一定会变成非零。因此，可以证实必败态只能转移到必胜态。

接下来，我们来证明必胜态总是能转移到某个必败态。观察XOR的二进制表示最高位的1，选取石子数的二进制表示对应位也为1的某堆石子。只要从中取走使得该位变为0，且其余XOR中的1也反转的数量的石子，XOR就可以变成零。

```
int N, A[MAX_N];

void solve() {
  int x = 0;
  for (int i = 0; i < N; i++) x ^= A[i];

  if (x != 0) puts("Alice");
  else puts("Bob");
}
```

2. Georgia and Bob[①]

Georgia and Bob （POJ 1704）

Georgia 和 Bob 在玩如下游戏。

棋盘的例子

① 这里介绍的问题又被称为Staircase Nim。——译者注

如上图所示，排成直线的格子上放有 n 个棋子。棋子 i 在左数第 p_i 个格子上。Georgia 和 Bob 轮流选择一个棋子向左移动。每次可以移动一格及以上任意多格，但是不允许反超其他的棋子，也不允许将两个棋子放在同一个格子内。

无法进行移动操作的一方失败。假设 Georgia 先进行移动，当双方都采取最优策略时，谁会获胜？

限制条件

- $1 \leqslant n \leqslant 1000$
- $1 \leqslant p_i \leqslant 10000$

样例 1

输入

```
n = 3
p = {1, 2, 3}
```

输出

```
Bob will win
```

样例 2

输入

```
n = 8
p = {1, 5, 6, 7, 9, 12, 14, 17}
```

输出

```
Georgia will win
```

如果将棋子两两成对当作整体考虑，我们就可以把这个游戏转为Nim游戏。先按棋子个数的奇偶分情况讨论。首先，考虑棋子个数为偶数的情况。把棋子从前往后两两组成一对，那么，我们就可以将每对棋子看成Nim中的一堆石子。石子堆中石子的个数等于两个棋子之间的间隔。

棋盘所对应Nim的例子

让我们想想看为什么能够这样转换。考虑其中的某一对棋子，将右边的棋子向左移动就相当于从Nim的石子堆中取走石子。

将右边的棋子向左移动的例子

另一方面，将左边的棋子向左移动，石子的数量就增加了。这就与Nim不同了。但是，即便对手增加了石子的数量，只要将所加部分减回去就回到了原来的状态；即便自己增加了石子的数量，只要对手将所加的部分减回去也回到了原来的状态。因此，该游戏的胜负状态和所转移成的Nim的胜负状态是一致的。

将左边的棋子向左移动的例子

当棋子的个数为奇数时，对最左边的棋子按下图进行特殊处理后，同样可以转成Nim。

奇数个的情况

```
const int MAX_N = 1000;

int N, P[MAX_N];

void solve() {
  if (N % 2 == 1) P[N++] = 0;
  sort(P, P + N);

  int x = 0;
  for (int i = 0; i + 1 < N; i += 2) {
    x ^= (P[i + 1] - P[i] - 1);
  }

  if (x == 0) puts("Bob will win");
  else puts("Georgia will win");
}
```

4.2.3 Grundy数

1. 硬币游戏2

硬币游戏 2

Alice 和 Bob 在玩这样一个游戏。给定 k 个数字 $a_1,a_2,\cdots,a_k$。一开始，有 n 堆硬币，每堆各有 x_i 枚硬币。Alice 和 Bob 轮流选出一堆硬币，从中取出一些硬币。每次所取硬币的枚数一定要在 $a_1,a_2,\cdots,a_k$ 当中。Alice 先取，取光硬币的一方获胜。当双方都采取最优策略时，谁会获胜？题目保证 $a_1,a_2,\cdots,a_k$ 中一定有 1。

限制条件

- $1 \leqslant n \leqslant 1000000$
- $1 \leqslant k \leqslant 100$
- $1 \leqslant x_i, a_i \leqslant 10000$

样例1

输入

```
n = 3
k = 3
a = {1, 3, 4}
x = {5, 6, 7}
```

输出

```
Alice
```

样例2

输入

```
n = 3
k = 3
a = {1, 3, 4}
x = {5, 6, 8}
```

输出

```
Bob
```

这和本节最初介绍的硬币问题1类似，只不过那道题中只有一堆硬币，而本题中有n堆。如果依然用动态规划算法的话，状态数将高达$O(x_1 \times x_2 \times \cdots \times x_n)$。

在此，为了高效地求解该问题，给大家介绍一下Grundy值这一重要概念。利用它，不光是这个游戏，其他许多游戏都可以转成前面所介绍的Nim。

让我们再来考虑一下只有一堆硬币的情况。硬币枚数x所对应的Grundy值的计算方法如下。

```
int grundy(int x) {
  集合 S = {};
  for (j = 1, ..., k) {
    if (a_j <= x)   将grundy(x - a_j)加到S中
 }
  return  最小的不属于S的非负整数
}
```

也就是说，当前状态的Grundy值就是除任意一步所能转移到的状态的Grundy值以外的最小非负整数。这样的Grundy值，和Nim中的一个石子堆类似，有如下性质。

- Nim中有x颗石子的石子堆，能够转移成有$0, 1, \cdots, x-1$颗石子的石子堆
- 从Grundy值为x的状态出发，可以转移到Grundy值为$0, 1, \cdots, x-1$的状态

只不过，与Nim不同的是，转移后的Grundy值也有可能增加。不过，对手总能够选取合适的策略再转移回相同Grundy值的状态，所以对胜负没有影响[①]。另外，上面的程序是用单纯的递归函数实现的，改成动态规划或记忆化搜索之后，就能够保证求解的复杂度为$O(xk)$。

了解了一堆硬币的Grundy值的计算方法之后，就可以将它看作Nim中的一个石子堆。Nim中我们用如下方法判断胜负。

- 所有石子堆的石子数x_i的XOR

  x_1 XOR x_2 XOR $\cdots$ XOR x_k
  为零则必败，否则必胜

Grundy值等价于Nim中的石子数，所以对于Grundy值的情况，有

- 所有硬币堆的Grundy值的XOR

  $grundy(x_1)$ XOR $grundy(x_2)$ XOR $\cdots$ XOR $grundy(x_n)$
  为零则必败，否则必胜

不光是这个游戏，在许多游戏中，都可以根据“当前状态的Grundy值等于除任意一步所能转移到的状态的Grundy值以外的最小非负整数”这一性质，来计算Grundy值，再根据XOR来判断胜负。

```
// 输入
int N, K, X[MAX_N], A[MAX_K];

// 利用动态规划计算grundy值的数组
int grundy[MAX_X + 1];

void solve() {
  // 轮到自己时剩0枚则必败
  grundy[0] = 0;

  // 计算grundy值
  int max_x = *max_element(X, X + N);
  for (int j = 1; j <= max_x; j++) {
    set<int> s;
    for (int i = 0; i < K; i++) {
      if (A[i] <= j) s.insert(grundy[j - A[i]]);
    }

    int g = 0;
    while (s.count(g) != 0) g++;
    grundy[j] = g;
  }
```

① 但是，对于状态可能有循环时，需要注意不分胜负、达成平局（游戏不会结束）的情况。因为在这个游戏中，石子数始终是减少的，所以不会发生平局。

```
    // 判断胜负
    int x = 0;
    for (int i = 0; i < N; i++) x ^= grundy[X[i]];

    if (x != 0) puts("Alice");
    else puts("Bob");
  }
```

2. Cutting Game

Cutting Game （POJ 2311）

两个人在玩如下游戏。

准备一张分成 $w \times h$ 的格子的长方形纸张，两人轮流切割纸张。要沿着格子的边界切割，水平或者垂直地将纸张切成两部分。切割了 n 次之后就得到了 $n+1$ 张纸，每次都选择切得的某一张纸再进行切割。首先切出只有一个格子的纸张（1×1 的各自组成的纸张）的一方获胜。当双方都采取最优策略时，先手是必胜？还是必败？

例子

限制条件

- $2 \leqslant w, h \leqslant 200$

样例 1

输入

```
w = 2, h = 2
```

输出

```
LOSE
```

样例 2

输入

```
w = 4, h = 2
```

输出

```
WIN
```

前面的硬币问题2中，有n堆硬币，我们求出每堆硬币的Grundy值，再根据它们XOR后的值判断胜负。另一方面，这个游戏中，初始只有一张纸，纸张的数量随着切割增加。这样会发生分割的游戏，也能够计算Grundy值。

当$w \times h$的纸张分成两张时，假设所分得的纸张的Grundy值分别为g_1和g_2，则这两张纸对应的状态的Grundy值可以表示为g_1 XOR g_2。

在Nim中，不论有几堆石子，初始状态是怎样的，只要XOR的结果相同，那么对胜负是没有影响的。这里也是同样的，只要Grundy值相同，即便发生分割，只要对分割后的各部分取XOR，就可以用这一个Grundy值来代表几个游戏复合而成的状态，Grundy值也可以同样计算。[①]

了解了会发生分割的游戏的处理方法之后，只要像之前的问题一样，枚举所有一步能转移到的状态的Grundy值，就能够计算Grundy值了。

另外，切割纸张时，一旦切割出了长或宽为1的纸张，下一步就一定能够切割出1×1的纸张，所以可以知道此时必败。因此，切割纸张时，总要保证长和宽至少为2（无论如何都不能保证时，就是必败态。此时根据Grundy值的定义，不需要特别处理其Grundy值也是0）。

```
const int MAX_WH = 200;

//   记忆化搜索所用的数组，程序开始执行时全部初始化为-1
int mem[MAX_WH + 1][MAX_WH + 1];


int grundy(int w, int h) {
  if (mem[w][h] != -1) return mem[w][h];

  set<int> s;
  for (int i = 2; w - i >= 2; i++) s.insert(grundy(i, h) ^ grundy(w-i, h));
  for (int i = 2; h - i >= 2; i++) s.insert(grundy(w, i) ^ grundy(w, h-i));

  int res = 0;
  while (s.count(res)) res++;
  return mem[w][h] = res;
}


void solve(int w, int h) {
  if (grundy(w, h) != 0) puts("WIN");
  else puts("LOSE");
}
```

① 或者也可以说，这是因为XOR运算满足结合律。

4.3 成为图论大师之路

➠图是非常有用的数据结构，除了在第2章已经介绍的算法外，还有各种各样的相关算法。在此，我们主要讨论强连通分量分解和最近公共祖先等问题。

4.3.1　强连通分量分解

对于一个有向图顶点的子集S，如果在S内任取两个顶点u和v，都能找到一条从u到v的路径，那么就称S是强连通的。如果在强连通的顶点集合S中加入其他任意顶点集合后，它都不再是强连通的，那么就称S是原图的一个强连通分量（SCC：Strongly Connected Component）。任意有向图都可以分解成若干不相交的强连通分量，这就是强连通分量分解。把分解后的强连通分量缩成一个顶点，就得到了一个DAG（有向无环图）。

虚线包围的部分构成一个强连通分量

强连通分量分解可以通过两次简单的DFS实现。第一次DFS时，选取任意顶点作为起点，遍历所有尚未访问过的顶点，并在回溯前给顶点标号（post order，后序遍历）。对剩余的未访问过的顶点，不断重复上述过程。

完成标号后，越接近图的尾部（搜索树的叶子），顶点的标号越小。第二次DFS时，先将所有边反向，然后以标号最大的顶点为起点进行DFS。这样DFS所遍历的顶点集合就构成了一个强连通分量。之后，只要还有尚未访问的顶点，就从中选取标号最大的顶点不断重复上述过程。

后续遍历的例子。根据搜索顺序的不同，标号结果也可能不同

反向后的图

正如前文所述，我们可以将强连通分量缩点并得到DAG。此时可以发现，标号最大的节点就属于DAG头部（搜索树的根）的强连通分量。因此，将边反向后，就不能沿边访问到这个强连通分量以外的顶点。而对于强连通分量内的其他顶点，其可达性不受边反向的影响，因此在第二次DFS时，我们可以遍历一个强连通分量里的所有顶点。

边反向后，从8、9、10号顶点只能到达其头部方向的顶点11和12

该算法只进行了两次DFS，因而总的复杂度是$O(|V|+|E|)$。

```
int V; // 顶点数
vector<int> G[MAX_V];    // 图的邻接表表示
vector<int> rG[MAX_V];   // 把边反向后的图
vector<int> vs;          // 后序遍历顺序的顶点列表
bool used[MAX_V];        // 访问标记
int cmp[MAX_V];          // 所属强连通分量的拓扑序
```

```
void add_edge(int from, int to) {
  G[from].push_back(to);
  rG[to].push_back(from);
}

void dfs(int v) {
  used[v] = true;
  for (int i = 0; i < G[v].size(); i++) {
    if (!used[G[v][i]]) dfs(G[v][i]);
  }
  vs.push_back(v);
}

void rdfs(int v, int k) {
  used[v] = true;
  cmp[v] = k;
  for (int i = 0; i < rG[v].size(); i++) {
    if (!used[rG[v][i]]) rdfs(rG[v][i], k);
  }
}

int scc() {
  memset(used, 0, sizeof(used));
  vs.clear();
  for (int v = 0; v < V; v++) {
    if (!used[v]) dfs(v);
  }
  memset(used, 0, sizeof(used));
  int k = 0;
  for (int i = vs.size() - 1; i >= 0; i--) {
    if (!used[vs[i]]) rdfs(vs[i], k++);
  }
  return k;
}
```

Popular Cows （POJ No.2186）

每头牛都想成为牛群中的红人。给定 N 头牛的牛群和 M 个有序对(A, B)。(A, B)表示牛 A 认为牛 B 是红人。该关系具有传递性，所以如果牛 A 认为牛 B 是红人，牛 B 认为牛 C 是红人，那么牛 A 也认为牛 C 是红人。不过，给定的有序对中可能包含(A, B)和(B, C)，但不包含(A, C)。求被其他所有牛认为是红人的牛的总数。

限制条件

- $1 \leqslant N \leqslant 10000$
- $1 \leqslant M \leqslant 50000$
- $1 \leqslant A, B \leqslant N$

样例

输入

```
N = 3
M = 3
(A, B) = {(1, 2), (2, 1), (2, 3)}
```

输出

```
1（3号牛）
```

考虑以牛为顶点的有向图，对每个有序对(A, B)连一条从A到B的有向边。那么，被其他所有牛认为是红人的牛对应的顶点，也就是从其他所有顶点都可达的顶点。虽然这可以通过从每个顶点出发搜索求得，但总的复杂度却是$O(NM)$，是不可行的，必须要考虑更为高效的算法。

假设有两头牛A和B都被其他所有牛认为是红人。那么显然，A被B认为是红人，B也被A认为是红人，即存在一个包含A、B两个顶点的圈，或者说，A、B同属于一个强连通分量。反之，如果一头牛被其他所有牛认为是红人，那么其所属的强连通分量内的所有牛都被其他所有牛认为是红人。由此，我们把图进行强连通分量分解后，至多有一个强连通分量满足题目的条件。而按前面介绍的算法进行强连通分量分解时，我们还能够得到各个强连通分量拓扑排序后的顺序，唯一可能成为解的只有拓扑序最后的强连通分量。所以在最后，我们只要检查这个强连通分量是否从所有顶点可达就好了。该算法的复杂度为$O(N+M)$，足以在时限内解决原题。

```
// 输入
int N, M;
int A[MAX_M], B[MAX_M];

void solve() {
  V = N;
  for (int i = 0; i < M; i++) {
    add_edge(A[i] - 1, B[i] - 1);
  }
  int n = scc();

  // 统计备选解的个数
  int u = 0, num = 0;
  for (int v = 0; v < V; v++) {
    if (cmp[v] == n - 1) {
      u = v;
      num++;
    }
  }

  // 检查是否从所有点可达
  memset(used, 0, sizeof(used));
  rdfs(u, 0);  // 重用强连通分量分解的代码
```

```
    for (int v = 0; v < V; v++) {
      if (!used[v]) {
        //  从该点不可达
        num = 0;
        break;
      }
    }

    printf("%d\n", num);
  }
```

4.3.2　2-SAT

给定一个布尔方程，判断是否存在一组布尔变量的真值指派使整个方程为真的问题，被称为布尔方程的可满足性问题（SAT）。SAT问题是NP完全的，但对于满足一定限制条件的SAT问题，还是能够有效求解的。我们将下面这种布尔方程称为合取范式。

$$(a \vee b \vee \cdots) \wedge (c \vee d \vee \cdots) \wedge \cdots$$

其中$a, b, \cdots$称为文字，它是一个布尔变量或其否定。像$(a \vee b \vee ...)$这样用$\vee$连接的部分称为子句。如果合取范式的每个子句中的文字个数都不超过两个，那么对应的SAT问题又称为2-SAT问题。

- 2-SAT布尔公式的例子
  - $(a \vee b) \wedge \neg a$ 令a为假而b为真，则可以满足
  - $(a \vee \neg b) \wedge (b \vee c) \wedge (\neg c \vee \neg a)$ 令a和b为真而c为假，则可以满足
  - $(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$ 无法满足

利用强连通分量分解，可以在布尔公式子句数的线性时间内解决2-SAT问题。首先，利用$\Rightarrow$（蕴涵）将每个子句$(a \vee b)$改写成等价形式$(\neg a \Rightarrow b \wedge \neg b \Rightarrow a)$。这样原布尔公式就变成了把$a \Rightarrow b$形式的布尔公式用$\wedge$连接起来的形式。对每个布尔变量$x$，构造两个顶点分别代表$x$和$\neg x$，以$\Rightarrow$关系为边建立有向图。此时，如果图上的$a$点能够到达$b$点的话，就表示当$a$为真时$b$也一定为真。因此，该图中同一个强连通分量中所含的所有文字的布尔值均相同。

如果存在某个布尔变量x，x和$\neg x$均在同一个强连通分量中，则显然无法令整个布尔公式的值为真。反之，如果不存在这样的布尔变量，那么对于每个布尔变量x，让

$$x\text{ 所在的强连通分量的拓扑序在}\neg x\text{ 所在的强连通分量之后} \Leftrightarrow x\text{ 为真}$$

就是使得该公式的值为真的一组合适的布尔变量赋值。

(a∨¬b)∧(b∨c)∧(¬c∨¬a)所对应的图

```
int main() {
  // 布尔公式为(a∨ ¬b)∧ (b∨ c)∧ (¬c∨ ¬a)时
  // 构造6个顶点，分别对应a、b、c、¬a、¬b、¬c。
  V = 6;

  // a∨ ¬b转成¬a⇒¬b∧b⇒a
  add_edge(3, 4); // 从¬a连一条到¬b的边
  add_edge(1, 0); // 从b连一条到a的边
  // b∨ c转成¬b⇒c∧¬c⇒b
  add_edge(4, 2); // 从¬b连一条到c的边
  add_edge(5, 1); // 从¬c连一条到b的边
  // ¬c∨ ¬a转成c⇒¬a∧a⇒¬c
  add_edge(2, 3); // 从c连一条到¬a的边
  add_edge(0, 5); // 从a连一条到¬c的边

  // 进行强连通分量分解
  scc();

  // 判断是否x和¬x在不同的强连通分量中
  for (int i = 0; i < 3; i++) {
    if (cmp[i] == cmp[3 + i]) {
      printf("NO");
      return 0;
    }
  }

  // 如果可满足，则给出一组解
  printf("YES\n");
  for (int i = 0; i < 3; i++) {
    if (cmp[i] > cmp[3 + i]) {
      printf("true\n");
    } else {
      printf("false\n");
    }
  }

  return 0;
}
```

Priest John's Busiest Day（POJ No.3683）

约翰是街区里唯一的神父。假设有 N 对新人打算在同一天举行结婚仪式。第 i 对新人的结婚仪式的时间为 S_i 到 T_i，在其仪式开始时或是结束时需要进行一个用时 D_i 的特别仪式（也就是从 S_i 到 S_i+D_i 或是从 T_i-D_i 到 T_i），该特别仪式需要神父在场。请判断是否可以通过合理安排每个特别仪式在开始还是结束时进行，从而保证神父能够出席所有的特别仪式。如果可能的话，请输出出席各个特别仪式的时间。当然，神父不可能同时出席多个特别仪式。不过神父前往仪式的途中所花费的时间可以忽略不计，神父可以在出席完一个特别仪后，立刻出席另一个开始时间与其结束时间相等的特别仪式。

限制条件

- $1 \leqslant N \leqslant 1000$

样例

输入

```
N = 2
(S, T, D) = {(08:00, 09:00, 30), (08:15, 09:00, 20)}
```

输出

```
YES
08:00 08:30
08:40 09:00
```

对于每个结婚仪式i，只有在开始或结束时进行特别仪式两种选择。因此可以定义变量x_i

x_i为真 ⇔ 在开始时进行特别仪式

这样，对于结婚仪式i和j，如果$S_i \sim S_i+D_i$和$S_j \sim S_j+D_j$冲突，就有$\neg x_i \vee \neg x_j$为真。对于开始和结束、结束和开始、结束和结束等三种情况，也可以得到类似的条件。于是，要保证所有特别仪式的时间不冲突，只要考虑将这所有的子句用∧连接起来所得到的布尔公式就好了。例如，对于输入样例，可以的到布尔公式

$$(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$$

而当x_1为真而x_2为假时，其值为真。这样，我们就把原问题转为了2-SAT问题。接下来只要进行强连通分量分解并判断是否有使得布尔公式值为真的一组布尔变量赋值就好了。

```
// 输入
int N;
int S[MAX_N], T[MAX_N], D[MAX_N];  // S和T是换算成分钟后的时间
```

```
void solve() {
  // 0~N-1: x_i
  // N~2N-1: ¬x_i
  V = N * 2;
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < i; j++) {
      if (min(S[i] + D[i], S[j] + D[j]) > max(S[i], S[j])) {
        // x_i⇒¬x_j、x_j⇒¬x_i
        add_edge(i, N + j);
        add_edge(j, N + i);
      }
      if (min(S[i] + D[i], T[j]) > max(S[i], T[j] - D[j])) {
        // x_i⇒x_j、¬x_j⇒¬x_i
        add_edge(i, j);
        add_edge(N + j, N + i);
      }
      if (min(T[i], S[j] + D[j]) > max(T[i] - D[i], S[j])) {
        // ¬x_i⇒¬x_j、x_j⇒x_i
        add_edge(N + i, N + j);
        add_edge(j, i);
      }
      if (min(T[i], T[j]) > max(T[i] - D[i], T[j] - D[j])) {
        // ¬x_i⇒x_j、¬x_j⇒x_i
        add_edge(N + i, j);
        add_edge(N + j, i);
      }
    }
  }
  scc();

  // 判断是否可满足
  for (int i = 0; i < N; i++) {
    if (cmp[i] == cmp[N + i]) {
      printf("NO\n");
      return;
    }
  }

  // 如果可满足，则给出一组解
  printf("YES\n");
  for (int i = 0; i < N; i++) {
    if (cmp[i] > cmp[N + i]) {
      // x_i为真，即在结婚仪式开始时举行
      printf("%02d:%02d %02d:%02d\n", S[i] / 60, S[i] % 60, (S[i] + D[i]) / 60,
             (S[i] + D[i]) % 60);
    } else {
      // x_i为假，即在结婚仪式结束时举行
      printf("%02d:%02d %02d:%02d\n", (T[i] - D[i]) / 60, (T[i] - D[i]) % 60,
              T[i] / 60, T[i] % 60);
    }
  }
}
```

4.3.3　LCA

在有根树中，两个节点u和v的公共祖先中距离最近的那个被称为最近公共祖先（LCA，Lowest Common Ancestor)。用于高效计算LCA的算法有许多，在此我们介绍其中的两种。在下文中，我们都假设节点数为n。

LCA的例子（4和7的LCA为2，8和6的LCA为1，5和8的LCA为5）

1. 基于二分搜索的算法

记节点v到根的深度为depth(v)。那么，如果节点w是u和v的公共祖先的话，让u向上走depth(u)-depth(w)步，让v向上走depth(v)-depth(w)步，就都将走到w。因此，首先让u和v中较深的一方向上走|depth(u)-depth(v)|步，再一起一步步向上走，直到走到同一个节点，就可以在O(depth(u)+depth(v))时间内求出LCA。

```
// 输入
vector<int> G[MAX_V];  // 图的邻接表表示
int root; // 根节点的编号

int parent[MAX_V];      // 父亲节点（根节点的父亲记为-1）
int depth[MAX_V];       // 节点的深度

void dfs(int v, int p, int d) {
  parent[v] = p;
  depth[v] = d;
  for (int i = 0; i < G[v].size(); i++) {
    if (G[v][i] != p) dfs(G[v][i], v, d + 1);
  }
}

// 预处理
void init() {
  // 预处理出parent和depth
  dfs(root, -1, 0);
}

// 计算u和v的LCA
```

```
int lca(int u, int v) {
  // 让u和v向上走到同一深度
  while (depth[u] > depth[v]) u = parent[u];
  while (depth[v] > depth[u]) v = parent[v];
  // 让u和v向上走到同一节点
  while (u != v) {
    u = parent[u];
    v = parent[v];
  }
  return u;
}
```

节点的最大深度是$O(n)$，所以该算法的复杂度也是$O(n)$。如果只需计算一次LCA的话，这便足够了。但如果要计算多对节点的LCA的话如何是好呢？刚才的算法，通过不断向上走到同一节点来计算u和v的LCA。这里，到达了同一节点后，不论再怎么向上走，到达的显然还是同一节点。利用这一点，我们能够利用二分搜索求出到达共同祖先所需的最少步数吗？事实上，只要利用如下预处理，就可以实现二分搜索。

首先，对于任意顶点v，利用其父亲节点信息，可以通过parent2[v]=parent[parent[v]]得到其向上走两步所到的顶点。再利用这一信息，又可以通过parent4[v]=parent2[parent2[v]]得到其向上走四步所到的顶点。依此类推，就能够得到其向上走2^k步所到的顶点parent[k][v]。有了k=floor(log n)以内的所有信息后，就可以二分搜索了，每次的复杂度是$O(\log n)$。另外，预处理parent[k][v]的复杂度是$O(n\log n)$。

```
// 输入
vector<int> G[MAX_V];  // 图的邻接表表示
int root;              // 根节点的编号

int parent[MAX_LOG_V][MAX_V];  // 向上走2^k步所到的节点（超过根时记为-1）
int depth[MAX_V];              // 节点的深度

void dfs(int v, int p, int d) {
  parent[0][v] = p;
  depth[v] = d;
  for (int i = 0; i < G[v].size(); i++) {
    if (G[v][i] != p) dfs(G[v][i], v, d + 1);
  }
}

// 预处理
void init(int V) {
  // 预处理出parent[0]和depth
  dfs(root, -1, 0);
  // 预处理出parent
  for (int k = 0; k + 1 < MAX_LOG_V; k++) {
    for (int v = 0; v < V; v++) {
      if (parent[k][v] < 0) parent[k + 1][v] = -1;
      else parent[k + 1][v] = parent[k][parent[k][v]];
    }
```