串行调度

时间限制： 3.0 秒

空间限制： 128 MB

相关文件： 题目目录

题目描述

在数据库中，我们通常把能访问并可能更新各种数据项的一个程序执行单元称作“事务”。事务运用以下两个指令访问数据：

1. `READ(x)`：从数据库把数据项 x 传送到执行 `READ` 指令的事务的局部缓冲区；
2. `WRITE(x)`：从执行 `WRITE` 的事务把数据项 x 传回数据库。

当事务执行 `READ(x)` 指令时，将从数据库中得到数据项 x 的值，备份在该事务的局部缓冲区内。接下来经过对获得数据的计算处理，可以再使用 `WRTIE(x)` 指令去更新数据项 x 在数据库中的值。

我们做出以下几点约定：

1. 所有可以由指令访问的数据项有 n 个，依次记做 $x[1],\cdots,x[n]$ ；
2. 系统不会同时执行两个指令，一定是在执行完一个后再执行另一个，并且指令不会执行失败；
3. `WRITE(x)` 指令不会延迟更新，即该指令执行结束意味着数据库中数据项 x 的值已更新。

这里举一个简单的例子，假设 n=3，三个数据项 $x[1],x[2],x[3]$ 依次代表 Alice、 Bob 和 Dundun 的账户余额。那么事务 Alice 转账 50 元给 Bob 可以表示为如下形式：

事务 1：Alice 转账 50 元给 Bob

```
Tmp = READ(x[1]);
Tmp -= 50;
WRITE(x[1], Tmp);
Tmp = READ(x[2]);
Tmp += 50;
WRITE(x[2], Tmp);
```

这里 `Tmp` 是事务 1 缓冲区的一个临时变量，辅助完成对更新后余额的计算。出于某种考虑，现在我们只关心一个事务对哪些数据项执行了读写指令，而具体的数值我们可以忽略不管，所以事务 1 又可以简化为下面的形式：

事务 1（简）：Alice 转账给 Bob

```
READ(x[1]);
WRITE(x[1]);
READ(x[2]);
WRITE(x[2]);
```

类似地，这里再给出另一个简单的例子：

事务 2：Dundun 给 Bob 存入自己的余额那么多的钱（这笔钱并不从 Dundun 账户中出，所以无须 `WRITE(x[3])`）

```
READ(x[3]);
READ(x[2]);
WRITE(x[2]);
```

对于任意一个事务，其中的指令均是按顺序执行的。但是，数据库系统通常会并发地执行多个事务。通俗地说，就是执行完事务 1 的一个指令后，接下来可能会去执行事务 2 的下一个指令，然后继续执行事务 2 或者回来执行事务 1 都是完全有可能的。

这样并发管理事务的好处这里就不提了，我们还是来看看坏处吧。对于给定的 m 个事务，记做 $T[1],\cdots,T[m]$，显而易见的是这 m 个事务会有很多种不同的执行顺序，我们把这个执行顺序称为调度。对于上面的事务 1 和 2，下面是两种不同的调度。

调度 1：串行

```
T[1]: READ(x[1]);
T[1]: WRITE(x[1]);
T[1]: READ(x[2]);
T[1]: WRITE(x[2]);
T[2]: READ(x[3]);
T[2]: READ(x[2]);
T[2]: WRITE(x[2]);
```

调度 2：交错执行

```
T[2]: READ(x[3]);
T[1]: READ(x[1]);
T[1]: WRITE(x[1]);
T[2]: READ(x[2]);
T[1]: READ(x[2]);
T[2]: WRITE(x[2]);
T[1]: WRITE(x[2]);
```

仔细分析的话不难发现，这两个调度的执行结果可能会不同，即按照这两种不同的顺序执行完所有事务，数据项最终的值可能会不同。观察调度 2 的最后两行，$T[1]$ 的结果会把 $T[2]$ 的

覆盖掉，导致 Bob 账户只多了 50 块钱，Dundun 的馈赠就这样因为一个糟糕的调度不翼而飞了。

为了保证数据完整性，数据库系统维护事务要保证隔离性（isolation）：尽管多个事务并发执行，但对于任何一对事务 $T[i]$ 和 $T[j]$，在 $T[i]$ 看来，$T[j]$ 或者在 $T[i]$ 开始前已经完成执行，或者在 $T[i]$ 完成之后开始执行。这样，每个事务都感觉不到系统中有其他事务在并发地执行。因为调度的不确定性，在不加额外控制手段（比如加锁）的情况下，有些事务并不能同时在系统中并发执行。

在一个调度中，如果某个事务的所有指令是连续执行的（即其中没有穿插其他事务的指令），我们就称该事务被**串行执行。**像调度 1 这样所有事务都被串行执行的，我们称之为**串行调度**。串行调度显然是最理想的，所有事务不会互相干扰。对于一个给定的事务集，虽然有些调度并不是串行调度，但不同事务也不会产生冲突。接下来我们在调度上定义一个等价关系，来明确哪些调度和串行调度一样可靠。

考虑一个调度上**连续**的两条属于**不同事务**的指令，如果它们**针对不同的数据项**或者**同为** `READ` **指令**，则交换这两条指令可以得到一个等价的调度。不断交换两条满足上述要求的指令，则可以得到一系列等价的调度（即具有传递性）。如果一个调度等价于任意一个串行化调度，则称该调度是可串行化的，执行该调度时不会破坏数据完整性。

现在 Dundun 有一个在 n 个数据项、m 个事务上的**可串行化的**调度方案，希望你能帮他完成下面两个任务：

1. 找到一个与给定调度等价的串行调度，以证明 Dundun 的调度方案确实是可串行化的。如果有多个等价的串行调度，请输出字典序最小的那种，即优先执行编号较小的事务。

1. 解决 q 个如下的问题：对于给定的两个事务 $T[i]$ 和 $T[j]$，判断是否存在某个等价的串行调度，满足事务 $T[i]$ 可以在事务 $T[j]$ 之前完成。

输入格式

从标准输入读入数据。

第一行四个正整数 n、m、p 和 q，分别表示数据项的个数、事务个数、指令的个数及任务 2 的问题个数，数据项和事务编号均从 1 开始。

接下来 p 行，每行描述一个指令，包含三个整数：

- 第一个数为 op，表示指令的类型，其中 0 表示读指令，1 表示写指令，保证不会有其他的取值；
- 第二个数为 k，表示该指令的数据项为 $x[k]$，保证 $1 \leqslant k \leqslant n$；
- 第三个数为 w，表示该指令属于事务 $T[w]$，保证 $1 \leqslant w \leqslant m$，且每个事务至少包含

一条指令。

这 p 条指令的顺序即为 Dundun 给出的这个可串行化的调度方案。

接下来 q 行，每行用空格分隔的两个整数 i 和 j，表示任务 2 的一个问题，保证 $1 \leqslant i,j \leqslant m$ 。

上述各行的整数之间用一个空格隔开。

输出格式

输出到标准输出。

第一行输出任务一的答案，输出一个 1 到 m 的排列，表示字典序最小的串行调度方案中 m 个事务的执行顺序，每两个整数用空格分隔。

接下来 q 行，每行对应任务 2 的一个问题，如果存在输出 `YES`，否则输出 `NO`。

样例输入

```
3 2 7 2
0 1 1
1 1 1
0 3 2
0 2 2
1 2 2
0 2 1
1 2 1
1 2
2 1
```

样例输出

```
2 1
NO
YES
```

数据生成方式

这里描述了本题目所有测试点的生成方式。你不需要自己生成数据，这部分只是为了方便你分析数据的性质。

下文中所有称“随机生成”的数，都是调用 python 语言的 `random.randint` 生成符合范围的随机整数，可以近似认为每个可能的取值等概率。

生成数据的步骤如下：

1. 使用 `random.shuffle`（近似于所有排列等概率） 生成一个 1 到 m 的排列，作为初始串行调度方案中各事务的排列顺序。
2. 对于每个事务，生成恰好 pm 条指令，每条指令的 op 和 k 随机生成。
3. 不断地重复下列步骤，直到所有指令都被执行：
    - 在还没被执行的指令中找到所有可以通过等价调度变换到最靠前执行的集合 S；
    - 从 S 中随机抽取一条执行。
4. 随机生成 q 对不同的整数 i 和 j，作为任务 2 的问题。

子任务

测试点	***n*=**	***m*=**	***p*=**	***q*=**
1	3	3	6	60
2	5	5	15	102
3	10	10	40	0
4	15	15	75	300
5	20	20	120	0
6	150	150	1,500	3,000
7	200	200	2,000	0
8	300	300	3,300	6,000
9	500	500	6,000	0
10	2,000	2,000	3×104	4×104
11	3,000	3,000	48,000	0
12	5,000	5,000	85,000	105
13	10^4	10^4	1.8×10^5	0
14				2×10^5
15	12,000	12,000	216,000	2.4×10^5
16	15,000	15,000	285,000	0
17				3×10^5
18	18,000	18,000	342,000	3.6×10^5
19	2×10^4	2×10^4	4×10^5	0
20				4×10^5