2021 CCF 大学生计算机系统与程序设计竞赛（全国赛）

CCF CCSP 2021

时间：2021 年 12 月 15 日 09:00 ~ 21:00

题目名称	最近数合并	抽卡	量子计算	缓存管理器	可容错键值存储
题目类型	传统型	传统型	传统型	传统型	传统型
输入	标准输入	标准输入	标准输入	标准输入	标准输入
输出	标准输出	标准输出	标准输出	标准输出	标准输出
每个测试点时限	1.0 秒	1.0 秒	1.0 秒	2.0～5.0 秒	4.0 秒
内存限制	512 MiB	512 MiB	512 MiB	512 MiB	512 MiB
子任务数目	20	9	8	6	5
测试点是否等分	是	否	否	否	否

最近数合并（merge）

【题目背景】

在西西艾弗岛上，生活着快乐的小 C、小 S 和小 P。最近，三位小朋友在学校里学到了“两个数之间的距离”这一重要的概念。为了巩固知识，小 P 玩起了“合并最近数”的游戏。

【题目描述】

$A = \{a_1, a_2, \cdots, a_n\}$ 是一个包含 n 个自然数（非负整数）的集合（根据集合的定义，这 n 个数两两不同），且其中的最大值小于一个给定的正整数 k。

小 P 需要对集合 A 中的数进行若干轮合并操作，到 A 中仅剩一个数为止。每轮合并操作的流程如下：

1. 选取数对 (x, y)：
- 从 A 中选取数值大小最接近的两个数 x 和 y，即 $|x-y|$ 在所有数对中最小；
- 如果有多个数对满足条件，则进一步选择其中 $(x+y)$ 最小的；
- 考虑到 A 中的数两两不同，按上述要求（即以 $|x-y|$ 为第一关键字、$(x+y)$ 为第二关键字）可以选出唯一的数对 (x, y)。
2. 将 x 和 y 合并为 z：
- $z = (x+y) \bmod k$，即将 x 和 y 求和后再对 k 取模。
- 具体来说，首先将 x 和 y 从集合 A 中删去；如果集合 A 不包含 z，则再将 z 添加回集合 A。这保证了 A 中的自然数始终两两不同且小于 k。

易知，小 P 的每轮合并操作会使集合 A 的规模（即包含自然数的个数）减少 1 或 2。你需要帮小 P 搞明白，在全部合并操作结束后，进行的合并操作的总轮数和 A 中剩下的那个数。

【输入格式】

从标准输入读入数据。

输入的第一行包含用空格分隔的两个正整数 n 和 k。

输入的第二行包含 n 个用空格分隔的自然数 $a_1, a_2, \cdots, a_n$。

【输出格式】

输出到标准输出。

输出共两行。

第一行输出一个整数 T，表示总共进行了 T 轮合并操作。

第二行输出 T 轮合并操作后 A 中剩下的那个数。

【样例 1 输入】

```
9 10
0 1 2 4 5 6 7 8 9
```

【样例 1 输出】

```
7
5
```

【样例 1 解释】

第一轮：$(0,1)\to 1$，合并后 $A=\{1,2,4,5,6,7,8,9\}$；
第二轮：$(1,2)\to 3$，合并后 $A=\{3,4,5,6,7,8,9\}$；
第三轮：$(3,4)\to 7$，合并后 $A=\{5,6,7,8,9\}$；
第四轮：$(5,6)\to 1$，合并后 $A=\{1,7,8,9\}$；
第五轮：$(7,8)\to 5$，合并后 $A=\{1,5,9\}$；
第六轮：$(1,5)\to 6$，合并后 $A=\{6,9\}$；
第七轮：$(6,9)\to 5$，合并后 $A=\{5\}$。

【样例 2 输入】

```
6 11
7 3 6 5 10 0
```

【样例 2 输出】

```
4
9
```

【子任务】

40% 的测试数据满足 $n\le 200$、$k\le 1000$；
70% 的测试数据满足 $n\le 2000$、$k\le 10^5$；
全部的测试数据满足 $n\le 10^5$、$k\le 10^8$，输入的 a_i（$1\le i\le n$）两两不同且 $0\le a_i<k$。

抽卡（gacha）

【题目背景】

和其他许多同龄的小朋友一样，小 P 也喜欢玩手游抽卡。为了防止沉迷游戏影响学习，他只能在每个休息日的晚上玩一小时游戏。为了在这短暂的时间内赢得尽量多的分数，小 P 想让你帮他计算不同情况下，他可能获得的期望游戏分数。

【题目描述】

游戏中共有三类卡牌：***SSR***、***SR*** 和 ***R***。在每一局游戏中，小 P 恰有 n 次抽卡机会，其中第 i 次有 p_i 的概率抽到 ***SSR***，q_i 的概率为 ***SR***，剩下 $1-p_i-q_i$ 的概率为 ***R***。

SSR 卡牌共有三种，当小 P 抽到 ***SSR*** 类牌时，他会等概率地获得其中的一种。也就是说，他第 i 次抽卡抽到特定的一种 ***SSR*** 的概率为 $\frac{p_i}{3}$。一旦小 P 抽齐三种 ***SSR***，就会触发一次结算：如果此时他手中有 S 张 ***SSR***（可能有重复）和 K 张 ***SR***，则他将获得 S^2+K 的分数。

结算后小 P 手中的卡牌会被清空，他将继续重复“抽卡—结算—清空”的过程直至 n 次抽卡机会用完，过程中多次结算的分数会被累加。需要注意，如果第 n 次抽卡后未触发结算，则此时小 P 手中剩余的卡牌不会产生任何分数。小 P 想知道，n 次抽卡后总分数的期望是多少。

为了吸引玩家参与抽卡，游戏运营方还会举办 m 次活动。在每次活动中，运营方都会修改某一次抽卡的概率（一对 p_i、q_i）；**该修改仅在当前活动中生效，不会影响到其他活动**。为了节省新活动研究攻略的时间，小 P 还需要你帮他算出每次活动中抽卡的分数期望。

【输入格式】

从标准输入读入数据。

第一行两个整数 n、m，表示小 P 有 n 次抽卡机会、活动个数为 m，保证 $n \le 3\times10^5$、$m \le 3\times10^5$。

接下来 n 行，每行有两个整数 p_i、q_i，其中第 i 行为第 i 次抽卡对应的两个概率（概率用不带百分号的百分数表示，即用满足 $0 \le c \le 100$ 的整数 c，表示概率 $c\%$）。

接下来 m 行，每行有三个整数 t、p'、q' 描述一个活动，表示在该活动中将第 t 次抽卡中的概率 p_t、q_t 修改为 p'、q'。

【输出格式】

输出到标准输出。

输出 $m+1$ 行，每行一个整数。

第一行为在不参与活动的情况下，小 P 总分数的期望；接下来 m 行，为对应每次活动中，小 P 总分数的期望。

很显然，每个期望值都是有理数；但考虑到精度问题，你**不能**直接输出这个值。记所求的期望值为 E，你需要输出 $E \times 300^n$ 对 10^9+7 取模的值（容易证明 $E \times 300^n$ 总是一个非负整数）。

【样例 1 输入】

```
3 0
20 40
10 20
30 40
```

【样例 1 输出】

```
324000
```

【样例 1 解释】

想要触发结算，必定小 P 三次均抽出 ***SSR***，且互不相同，概率为 $0.2\times0.1\times0.3\times\frac{3!}{3^3}=\frac{1}{750}$，此时结算分数为 $3^2+0=9$，故总分数期望为 $\frac{3}{250}$。最后输出 $\frac{3}{250}\times300^3=324000$。

【样例 2 输入】

```
4 1
20 40
20 30
30 40
20 30
2 10 20
```

【样例 2 输出】

```
686160000
441360000
```

【样例 2 解释】

不参与活动的结果为 $\frac{953}{11250} \times 300^4 = 686160000$。

【样例 3 输入】

```
3 2
20 40
10 20
20 30
1 10 20
3 30 40
```

【样例 3 输出】

```
216000
108000
324000
```

【样例 3 解释】

注意最后一次活动的概率是按照 $(20,40)(10,20)(30,40)$ 计算的（即与样例一的概率相同），各个活动互不影响。

【样例 4 输入】

```
5 0
10 30
10 30
10 30
10 30
```

```
10 30
```

【样例 4 输出】

```
981399671
```

【样例 4 解释】

981399671 是 47981400000 对 10^9+7 取模的结果。

【子任务】

所有数据保证 $1 \le n \le 3 \times 10^5$，$0 \le m \le 3 \times 10^5$，$0 \le p \le 30$，$p < q \le 70$，$p + q \le 70$，$1 \le t \le n$。

<table>
<tr><th>子任务</th><th>子任务分值</th><th>n</th><th>m</th><th>特殊性质</th></tr>
<tr><td>1</td><td>10</td><td rowspan="2">≤ 9</td><td>≤ 0</td><td rowspan="5">无</td></tr>
<tr><td>2</td><td>5</td><td>≤ 2</td></tr>
<tr><td>3</td><td>18</td><td>≤ 200</td><td rowspan="2">≤ 0</td></tr>
<tr><td>4</td><td>10</td><td>$\le 3,000$</td></tr>
<tr><td>5</td><td>20</td><td rowspan="3">$\le 10^4$</td><td>$\le 10^2$</td></tr>
<tr><td>6</td><td>15</td><td rowspan="2">$\le 10^4$</td><td>$t \le 100$</td></tr>
<tr><td>7</td><td>10</td><td rowspan="3">无</td></tr>
<tr><td>8</td><td>7</td><td>$\le 10^5$</td><td>$\le 10^5$</td></tr>
<tr><td>9</td><td>5</td><td>$\le 3 \times 10^5$</td><td>$\le 3 \times 10^5$</td></tr>
</table>

量子计算（quantum）

【题目背景】

小 P 最近在网上看到了中国的“九章”量子计算机实现了量子优越性（quantum supremacy）的新闻，就向老师请教什么是量子计算和量子计算机。听了老师深入浅出的讲解，小 P 明白了量子计算的原理其实并不复杂，本质上可以等价为他很早就学过的矩阵乘法，只是运算量很大，而量子计算机可以非常快地完成这些运算。好学的小 P 想研究一些量子电路的性质，因此请你帮他实现一个简单的量子电路模拟器。

【题目描述】

比特是经典计算机中的一个基本概念，量子计算机中也存在“量子比特”（quantum bit，简写为 qubit）。经典计算机中一个比特的取值可以是 0 或 1；量子计算机与之类似，一个量子比特可能处于的两个量子态是 $|0\rangle$ 和 $|1\rangle$，其中 $|\rangle$ 是物理学家惯用的 Dirac 符号。经典计算机中的比特取值只可能是 0 和 1 中的一个，而量子比特的状态 $|\Psi\rangle$ 除了处于 $|0\rangle$ 和 $|1\rangle$ 以外，还会处于 $|0\rangle$ 和 $|1\rangle$ 的线性叠加态，记为：

$$|\Psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle = \begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix},\ \alpha_0, \alpha_1 \in \mathbb{C}.$$

该量子比特测量值为 0 和 1 的概率分别为 $|\alpha_0|^2$ 和 $|\alpha_1|^2$，且满足归一化条件:

$$|\alpha_0|^2 + |\alpha_1|^2 = 1.$$

量子状态的操作是通过量子门（quantum gate）来完成的。作用在一个量子比特上的量子门简称“单比特门”，可以被表示为一个 2×2 的酉矩阵[1]。一些常见的单比特门为（下文中将会直接使用这些记号）：

$$H = \begin{bmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{bmatrix} \quad X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \quad Z = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$$

在一个量子比特 $|\Psi\rangle$ 上作用一个单比特门 U，将生成新的量子状态 (α_0', α_1')：

$$\begin{pmatrix}\alpha_0'\\ \alpha_1'\end{pmatrix} = U|\Psi\rangle = \begin{bmatrix}U_{0,0} & U_{0,1}\\ U_{1,0} & U_{1,1}\end{bmatrix}\begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix} = \begin{bmatrix}U_{0,0}\alpha_0 + U_{0,1}\alpha_1\\ U_{1,0}\alpha_0 + U_{1,1}\alpha_1\end{bmatrix}$$

例如，上面提到的 X 被称为“非门”，它会翻转其作用的量子比特 $|\Psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$，生成新的量子状态 $|\Psi'\rangle = \alpha_1|0\rangle + \alpha_0|1\rangle$。

[1] 酉矩阵是指满足 $UU^* = U^*U = I$ 的复矩阵，其中 U^* 表示 U 的共轭转置。此性质对于本题没有实际影响。

类似地，一个包含两个量子比特的量子状态 $|\Psi\rangle$ 有四种可能的状态 $|00\rangle$、$|01\rangle$、$|10\rangle$、$|11\rangle$，其中 $|pq\rangle$ 表示第 0 个量子比特处于状态 $|q\rangle$ 且第 1 个量子比特处于状态 $|p\rangle$ $(p, q \in \{0, 1\})$。该双量子比特的状态也可以处于上述四种状态的线性叠加态：

$$|\Psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle = \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix}$$

与单比特系统相似，$|\alpha_{ij}|^2$ 是该量子状态的四种测量结果的概率分布，且满足:

$$|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$$

类似地，一个作用在两个量子比特上的双比特门可以由一个 4×4 的酉矩阵表示。一种重要的双比特门称为“受控量子门”（controlled gate），它的表示形如：

$$CU = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & U_{0,0} & U_{0,1} \\ & & U_{1,0} & U_{1,1} \end{bmatrix}$$

其中 $U = \begin{bmatrix} U_{0,0} & U_{0,1} \\ U_{1,0} & U_{1,1} \end{bmatrix}$ 是一个 2×2 的酉矩阵。

受控量子门中的两个量子比特分别被称为“控制比特”和“目标比特”。作用一个受控量子门 CU 等价于仅在下标的控制比特所在位为 1 时，在目标比特所在位分别为 0、1 的一个数据对上作用对应的单比特门 U。例如，在一个双量子比特系统上作用一个控制比特为第 1 个比特、目标比特为第 0 个比特的受控非门 CX，相当于将单比特门 X 作用在数据对 $(\alpha_{10}, \alpha_{11})$ 上，从而交换二者的值，并保持 α_{00} 和 α_{01} 的取值不变：

$$\begin{pmatrix} \alpha'_{00} \\ \alpha'_{01} \\ \alpha'_{10} \\ \alpha'_{11} \end{pmatrix} = CX \cdot \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix} \cdot \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix} = \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{11} \\ \alpha_{10} \end{pmatrix}$$

也可简写为：

$$\begin{pmatrix} \alpha'_{10} \\ \alpha'_{11} \end{pmatrix} = X \cdot \begin{pmatrix} \alpha_{10} \\ \alpha_{11} \end{pmatrix} = \begin{bmatrix} & 1 \\ 1 & \end{bmatrix} \cdot \begin{pmatrix} \alpha_{10} \\ \alpha_{11} \end{pmatrix} = \begin{pmatrix} \alpha_{11} \\ \alpha_{10} \end{pmatrix}$$

在一个双量子比特的系统上也可以作用单比特门。它等价于将矩阵 U 分别乘到两个数据对中，每个数据对的两个下标仅有一位不同（即该比特门的作用位）。作用在第 0 个量子比特上的门相当于将矩阵分别与 $(\alpha_{00}, \alpha_{01})$ 和 $(\alpha_{10}, \alpha_{11})$ 相乘，作用在第 1 个量子比特上的门相当于将矩阵分别与 $(\alpha_{00}, \alpha_{10})$ 和 $(\alpha_{01}, \alpha_{11})$ 相乘。例如，将非门作用在第 1 个量子比特上，会分别导致 α_{00} 与 α_{10} 的取值交换和 α_{01} 与 α_{11} 的取值交换。

本题需要处理的是由 n 个量子比特构成的系统，其状态可由 2^n 个基本状态（或称为“本征态”）$|0\dots00\rangle, |0\dots01\rangle, \dots, |1\dots11\rangle$ 的线性组合表示，记为：

$$|\Psi\rangle = \alpha_{0\dots00}|0\dots00\rangle + \alpha_{0\dots01}|0\dots01\rangle + \dots + \alpha_{1\dots11}|1..11\rangle$$

为简单起见，下文中将量子系统的状态统一记为向量 $|\Psi\rangle = (\alpha_{0\dots00}, \alpha_{0\dots01}, \dots, \alpha_{1\dots11})$，也可理解为一个包含 2^n 个复数的有序数组。该数组的下标也可等价地使用十进制表示，即可将状态等价地记为 $|\Psi\rangle = (\alpha_0, \alpha_1, \dots, \alpha_{2^n-1})$。需要注意，$\alpha_i$ 下标 i 的二进制展开中，最低位位于最右端，如 $\alpha_{0\dots011}$ 的下标的第 0 位和第 1 位为 1，其余位为 0，对应十进制表示的 α_3；而在 $\{\alpha_i\}$ 数组的展开 $(\alpha_{0\dots00}, \alpha_{0\dots01}, \dots, \alpha_{1\dots11})$ 中，下标最小的数据位于最左端。

本题需要处理的量子门包括上文提及的单比特门和双比特受控量子门。每个门可由其作用的（一个或两个）量子比特和一个大小为 2×2 的酉矩阵 U 表示。

将一个单比特门 U 作用在一个有 n 个量子比特的系统的第 t 个量子比特上，等价于进行 2^{n-1} 个矩阵乘法。每个矩阵乘法更新下标仅第 t 位不同的一对位置上的数据：

$$\begin{pmatrix} \alpha'_{b_{n-1},\dots,b_{t+1},0,b_{t-1},\dots,b_0} \\ \alpha'_{b_{n-1},\dots,b_{t+1},1,b_{t-1},\dots,b_0} \end{pmatrix} = \begin{bmatrix} U_{0,0} & U_{0,1} \\ U_{1,0} & U_{1,1} \end{bmatrix} \begin{pmatrix} \alpha_{b_{n-1},\dots,b_{t+1},0,b_{t-1},\dots,b_0} \\ \alpha_{b_{n-1},\dots,b_{t+1},1,b_{t-1},\dots,b_0} \end{pmatrix}$$

类似地，将一个控制比特为第 c 个比特，目标比特为第 t 个比特（$c \neq t$）的受控量子门 CU 作用到一个有 n 个量子比特的系统，等价于进行 2^{n-2} 个矩阵乘法。每个矩阵乘法更新两个特定位置上的数据，这两个特定位置的下标满足控制比特所在的位都是 1，且仅有目标比特所在的位不同：

$$\begin{pmatrix} \alpha'_{b_{n-1},\dots,b_{c+1},1,b_{c-1},\dots,b_{t+1},0,b_{t-1},\dots,b_0} \\ \alpha'_{b_{n-1},\dots,b_{c+1},1,b_{c-1},\dots,b_{t+1},1,b_{t-1},\dots,b_0} \end{pmatrix} = \begin{bmatrix} U_{0,0} & U_{0,1} \\ U_{1,0} & U_{1,1} \end{bmatrix} \begin{pmatrix} \alpha_{b_{n-1},\dots,b_{c+1},1,b_{c-1},\dots,b_{t+1},0,b_{t-1},\dots,b_0} \\ \alpha_{b_{n-1},\dots,b_{c+1},1,b_{c-1},\dots,b_{t+1},1,b_{t-1},\dots,b_0} \end{pmatrix}$$

例如，$|\psi_0\rangle = (1, 0, 0, 0, 0, 0, 0, 0)$ 是一个三量子比特的初始状态，在此状态上依次进行下述演化：

1. 在第 0 个量子比特上作用单比特门 H，即将 H 的表示矩阵分别与状态向量中，下标只有第 0 位不同的数据对 $(\alpha_{000}, \alpha_{001})$、$(\alpha_{010}, \alpha_{011})$、$(\alpha_{100}, \alpha_{101})$、$(\alpha_{110}, \alpha_{111})$ 相乘，得到状态：
$$|\psi_1\rangle = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0, 0, 0, 0, 0, 0\right)$$
2. 在第 1 个量子比特上作用单比特门 H，得到状态：
$$|\psi_2\rangle = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 0, 0, 0, 0\right)$$
3. 在第 1 个量子比特上作用单比特门 Z，得到状态：
$$|\psi_3\rangle = \left(\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, 0, 0, 0, 0\right)$$

4. 在第 2 个量子比特上作用单比特门 H，得到状态：

$$|\psi_4\rangle = \left(\frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}\right)$$

5. 作用一个控制比特为第 0 个量子比特、目标比特为第 1 个量子比特、矩阵为 Z 的受控量子门，即分别将矩阵与下标的第 0 位为 1 且仅第 1 位不同的数据对 $(\alpha_{001}, \alpha_{011})$ 和 $(\alpha_{101}, \alpha_{111})$ 相乘，得到状态：

$$|\psi_5\rangle = \left(\frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}\right)$$

6. 在第 0 个量子比特上作用单比特门 Z，得到状态：

$$|\psi_6\rangle = \left(\frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}\right)$$

7. 作用一个控制比特为第 1 个量子比特、目标比特为第 2 个量子比特、矩阵为 Z 的受控量子门，即分别将矩阵与数据对 $(\alpha_{010}, \alpha_{110})$ 和 $(\alpha_{011}, \alpha_{111})$ 相乘，得到最终状态：

$$|\psi_7\rangle = \left(\frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}, -\frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}\right)$$

任务一（子任务 1，30 分）：根据上述定义，从初态

$$\alpha_i = \begin{cases} 1 & i = 0 \\ 0 & i \neq 0 \end{cases} \quad 0 \le i \le 2^n - 1$$

开始实现 n 个量子比特的量子状态上单比特门和双比特受控量子门的模拟。

你或许已经发现，对于作用在类似 $(1, 0, 0, 0 \dots)$ 的初始状态上的一系列量子门，若其作用的量子比特编号都小于 x，则仅需要更新状态的前 2^x 个位置上的数据，并保持剩余位置上的数据取值为零不变。

更进一步，在保持每个量子比特上的量子门都按原来顺序执行的前提下，可以对量子门的执行顺序进行调换，以达到更少的运算次数。图 1 (1) 绘制了上述示例中的量子电路，其每一行代表一个量子比特，横线上的方框代表要在这个比特上执行的量子门，黑色圆圈对应于受控量子门的控制比特。分别考虑每个量子比特的运算顺序，发现该图的依赖有且仅有 $(1 \to 5 \to 6)$、$(2 \to 3 \to 5 \to 7)$、$(4 \to 7)$，其中 $(a \to b)$ 表示量子门 a 必须在量子门 b 之前执行。因此，可以将示例中量子门的执行顺序调换为如图 1 (2) 所示的 $(1, 2, 3, 5, 4, 6, 7)$ ，以将需要更新的值的数量从 $2 + 4 + 4 + 8 + 8 + 8 + 8 = 42$ 降低至 $2 + 4 + 4 + 4 + 8 + 8 + 8 = 38$。

除此之外，还可以对量子比特进行重新编号，以使量子门作用的最大量子比特编号的增长尽可能慢。如图 1 (3) 所示，将量子比特 $\{0, 1, 2\}$ 重编号为 $\{1, 0, 2\}$ 后，需要更新的值的数量可进一步减少至 $2 + 2 + 4 + 4 + 4 + 8 + 8 = 32$。

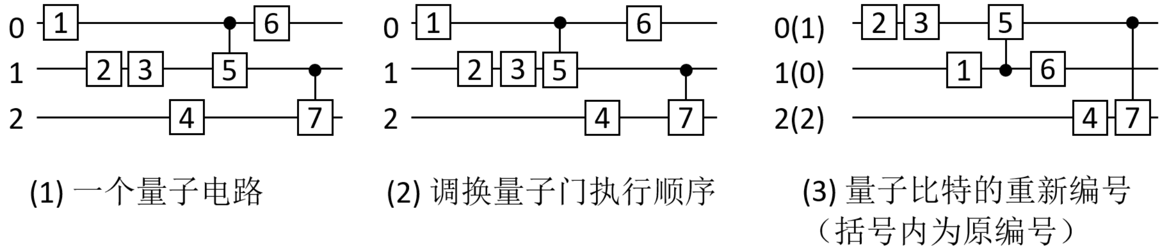


图 1: 量子电路的等价变换

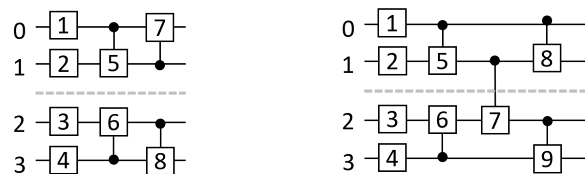
一种基于贪心算法的重编号策略为：进行 n 轮循环，第 i 轮循环挑选出一个量子比特 q_i，使得在满足电路的依赖条件的前提下，可直接作用在量子比特集合 $\{q_1, q_2, \ldots, q_i\}$ 上的门（下文简称为“可执行门”）的数量最大。例如，对图 1 (1) 所示电路进行分析的过程为：

- 第 1 轮循环：在分析开始时，作用在第 0 个量子比特上的可执行门只有量子门 1。量子门 $\{2, 3, 4, 5, 7\}$ 违背了“仅作用在量子比特 0 上”的限制，而尽管量子门 6 仅作用在第 0 个量子比特上，但根据电路的依赖条件，它必须在不可执行的量子门 5 之后执行，因此也不可执行。同理，第 1 个量子比特对应的可执行门集合为 $\{2, 3\}$，第 2 个量子比特对应的可执行门集合为 $\{4\}$。因此，本轮选取 $q_1 = 1$。
- 第 2 轮循环：在 $q_1 = 1$ 的前提下，若再选取 $q_2 = 0$，则可执行门为 $\{1, 2, 3, 5, 6\}$；若再选取 $q_2 = 2$，则可执行门为 $\{2, 3, 4\}$，因此选取 $q_2 = 0$。
- 第三轮循环：选取 $q_3 = 2$，此时可执行所有门。

此贪心算法无法保证求出的解是最优的，但在大部分情况下可以获得比较好的解。

任务二（27 分）：实现上述的优化（减少更新数值的数量）。该任务分为三个子任务，子任务 2（5 分）在仅实现部分位置更新后即可通过，子任务 3（10 分）还需要实现量子门的顺序调换，子任务 4（12 分）需要在子任务 3 的基础上进一步实现量子比特的重新编号。

虽然上述优化可以在一定程度上减小运算量，但是最坏时间复杂度依旧是 $O(2^n \times g)$，其中 n 为量子比特数量，g 为量子门的数量。对于某些特殊电路，可以专门设计模拟算法以加快运算。



(1) 切分为两个独立的部分　　(2) 切分为两个基本独立的部分

图 2: 基于量子电路切分的优化

例如，图 2 (1) 中的量子电路可以被直接切分由为虚线分隔的两部分，因此可以

从初态 $(1,0,0,0)$ 开始分别对上下两电路进行模拟。假设量子门 $\{1,2,3,4\}$ 的表示矩阵为 H，量子门 $\{5,6,8\}$ 的表示矩阵为 Z，量子门 7 的表示矩阵为 X，则上半部分的两个量子比特的模拟结果为 $|\psi\rangle = (\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, \frac{1}{2})$，下半部分的两个量子比特的模拟结果为 $|\phi\rangle = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$。得到两个子量子状态后，可进一步求出这四个量子比特组成的完整量子状态：对于位置 $\overline{b_3b_2b_1b_0}, b_{0\ldots3} \in \{0,1\}$，有 $|\alpha\rangle[\overline{b_3b_2b_1b_0}] = |\psi\rangle[\overline{b_1b_0}] \times |\phi\rangle[\overline{b_3b_2}]$。例如状态的第 $14 = \overline{1110}$ 个位置（即低位为 $\overline{10} = 2$，高位为 $\overline{11} = 3$）的取值为 $|\alpha\rangle[14] = |\psi\rangle[2] \times |\phi\rangle[3] = -\frac{1}{4}$。

从量子计算视角看，可将量子比特直接切分的电路并不能充分发挥其计算能力。一种更复杂的电路如图 2 (2) 所示，其中的量子电路仍旧被切分为虚线分隔的两部分，但两部分之间有较少的受控量子门相连。为处理这类电路，我们可以用和模拟图 2 (1) 中的电路相似的方法分别单独模拟上下两个电路，并特殊处理跨过两部分的量子门。

对图 2 (2) 中的电路，假设量子门 $\{1,2,3,4\}$ 的表示矩阵为 H，量子门 $\{5,6,7,9\}$ 的表示矩阵为 Z，量子门 8 的表示矩阵为 X。则可以按如下步骤模拟该电路：

1. 从初态 $|\psi_0\rangle = \{1,0,0,0\}$ 开始，在第 0、1 个量子比特组成的子状态中模拟量子门 1、2、5，得到状态 $|\psi_1\rangle = \{\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, -\frac{1}{2}\}$；从初态 $|\phi_0\rangle = \{1,0,0,0\}$ 开始，在第 2、3 个量子比特组成的子状态中模拟量子门 3、4、6，得到状态 $|\phi_1\rangle = \{\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, -\frac{1}{2}\}$。
2. 对于量子门 7 这类跨上下两部分量子比特的受控量子门，需要根据上一步得到的量子状态 $\{|\psi_1, \phi_1\rangle\}$ 中控制比特的情况，分别计算控制比特处于状态 $|0\rangle$ 的量子状态通过此受控量子门后的量子状态 $\{|\psi_2^0\rangle, |\phi_2^0\rangle\}$ 和控制比特处于状态 $|1\rangle$ 的量子状态通过此受控量子门后的量子状态 $\{|\psi_2^1\rangle, |\phi_2^1\rangle\}$：
   - 处理控制比特所在的子状态 $|\psi_1\rangle$。将其分裂为下标第 c（控制比特）位取 0 的状态和取 1 的状态，并将剩余位补零即可：

     $$|\psi_2^0\rangle[i] = \begin{cases} |\psi_1\rangle[i] & \text{i 的第 c 位为 0} \\ 0 & \text{i 的第 c 位为 1} \end{cases} \quad \text{且} \quad |\psi_2^1\rangle[i] = \begin{cases} 0 & \text{i 的第 c 位为 0} \\ |\psi_1\rangle[i] & \text{i 的第 c 位为 1} \end{cases}$$

     其中 $i \in \{0, 1, \ldots, 2^{\frac{n}{2}}\}$。具体到本例子，即 $|\psi_2^0\rangle = \{\frac{1}{2}, \frac{1}{2}, 0, 0\}$ 和 $|\psi_2^1\rangle = \{0, 0, \frac{1}{2}, -\frac{1}{2}\}$ 分别表示第 1 个量子比特取 0 和取 1 时第 0、1 个量子比特组成的子状态。
   - 处理目标比特所在的子状态 $|\phi_1\rangle$。根据受控量子门的定义可知，受控量子门相当于在下标的第 c 位取 1 时执行该量子门，在下标的第 c 位取 0 时跳过该量子门。因此，处理方式为将状态 $|\phi_2^0\rangle$ 置为状态 $|\phi_1\rangle$，将状态 $|\phi_2^1\rangle$ 置为在状态 $|\phi_1\rangle$ 的第 t（目标比特）个量子比特上作用与该量子门对应的酉矩阵 U 相同的单比特门后得到的状态。具体到本例子中，下半部分的第 2、3 个量子比特更新后的状态分别为跳过该量子门的状态 $|\phi_2^0\rangle = \{\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, -\frac{1}{2}\}$ 和作用该量子门的状态 $|\phi_2^1\rangle = \{\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\}$。
3. 对状态 $|\psi_2^0\rangle$、$|\psi_2^1\rangle$ 分别作用量子门 8，得到 $|\psi_3^0\rangle = \{\frac{1}{2}, 0, 0, \frac{1}{2}\}$ 和 $|\psi_3^1\rangle = \{0, -\frac{1}{2}, \frac{1}{2}\ 0\}$。对状态 $|\phi_2^0\rangle$、$|\phi_2^1\rangle$ 分别作用量子门 9，得到 $|\phi_3^0\rangle = \{\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}\}$ 和 $|\phi_3^1\rangle = \{\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}\}$。

4. 若电路中存在 2 个跨上下两部分量子比特的受控量子门，可按第 2 步所述方法对 $\{|\psi_2^0\rangle, |\phi_2^0\rangle\}$ 和 $\{|\psi_2^1\rangle, |\phi_2^1\rangle\}$ 分别进行分离操作，得到 $\{|\psi^{00}\rangle, |\phi^{00}\rangle\}$、$\{|\psi^{01}\rangle, |\phi^{01}\rangle\}$、$\{|\psi^{10}\rangle, |\phi^{10}\rangle\}$ 和 $\{|\psi^{11}\rangle, |\phi^{11}\rangle\}$。如果有更多跨两部分的量子门，也可以进行类似的处理：设共有 g_c 个跨上下两部分的量子门，则应计算出 2^{g_c} 个状态对 $\{|\psi^i\rangle, |\phi^i\rangle\}, i \in \{0, 1, \ldots, 2^{g_c}-1\}$。
5. 状态的合并：设 low、high 分别表示状态的第 p 个位置对应的低 $\frac{n}{2}$ 位和高 $\frac{n}{2}$ 位，则该位置的实际取值 $\alpha_p = \sum_{i=0}^{2^{g_c}-1} |\psi^i\rangle[\text{low}] \times |\phi^i\rangle[\text{high}]$。例如，第 $11 = \overline{1011}$ 个数据（即低位为 $\overline{11} = 3$，高位为 $\overline{10} = 2$）的取值为 $|\psi^0\rangle[3] \times |\phi^0\rangle[2] + |\psi^1\rangle[3] \times |\phi^1\rangle[2] = \frac{1}{4}$。

任务三（子任务 5，20 分）：实现上述基于电路切分的优化，以模拟可分为连接较少的两部分的量子电路。切分后的两个子电路分别包含量子比特 $\{q, 0 \le q \le \frac{n}{2}-1\}$ 和量子比特 $\{q, \frac{n}{2} \le q \le n-1\}$（$n$ 为偶数）。

在进行“基于切分的优化”时，上下两部分状态的模拟均可使用任务二中描述的优化，以进一步提升性能。

任务四（23 分）：实现上述混合优化。与任务二类似，该任务也分为三个子任务。子任务 6（3 分）仅需要在上下两部分电路中实现部分位置更新，子任务 7（8 分）需要分别对上下两部分电路进行量子门的顺序调换，子任务 8（12 分）需要分别在上下两部分电路内部进行量子比特的重新编号（仍保证上下两部分电路分别包含量子比特 $\{q, 0 \le q \le \frac{n}{2}-1\}$ 和量子比特 $\{q, \frac{n}{2} \le q \le n-1\}$）。

【输入格式】

从标准输入读入数据。

第一行四个整数 n、g、q、s，表示需要模拟有 n 个量子比特的电路，电路中共有 g 个量子门，需要查询模拟完成后 q 个位置的状态取值，此测例属于子任务 s。

之后 g 行，每行表示一个量子门。量子门输入格式分为两种：

- `s` $t\ r_{00}\ i_{00}\ r_{01}\ i_{01}\ r_{10}\ i_{10}\ r_{11}\ i_{11}$：表示作用在量子比特 t 上的单比特门；
- `c` $a\ t\ r_{00}\ i_{00}\ r_{01}\ i_{01}\ r_{10}\ i_{10}\ r_{11}\ i_{11}$：表示控制比特为 a、目标比特为 t （$a \ne t$）的双比特受控量子门。

每一行的数字之间用空格分隔。a、t 都是 0 到 $n-1$ 的整数，后面均为浮点数。两种情况下，对应门的表示矩阵均为

$$U = \begin{bmatrix} r_{00} + i_{00}\mathbf{j} & r_{01} + i_{01}\mathbf{j} \\ r_{10} + i_{10}\mathbf{j} & r_{11} + i_{11}\mathbf{j} \end{bmatrix}$$

其中 $x + y\mathbf{j}$ 表示实部为 x、虚部为 y 的复数。

输入顺序即为 g 个量子门的默认执行顺序，选手亦可按题面中介绍的规则对电路中的量子门进行等价变换，以减少完成模拟需要更新的位置的数量。

接下来 q 行，每行一个整数 p，表示询问模拟完成后状态向量上位置 p 的值，其中 $p \in \{0, 1, \ldots, 2^n - 1\}$。

【输出格式】

输出到标准输出。

输出共有 q 行，每行两个浮点数 x、y，表示从初态 $|\psi_0\rangle = \{1, 0, 0, \ldots, 0\}$ 开始，作用输入中给定的全部量子门后，位置向量上位置 p 的状态的取值 α_p 的实部和虚部。如果你输出的每个浮点数与参考结果相比，均满足绝对误差不大于 10^{-10} 或相对误差不大于 10^{-6}，则该测试点满分，否则不得分。**建议使用双精度浮点数并至少输出 10 位有效数字**。各种语言中输出 10 位有效数字的做法如下：

- C/C++: `printf("%.10lg", ans);`（需要引入 `stdio.h` 或 `cstdio` 文件）
- Java: `System.out.printf("%.10g", ans);`
- Python: `"{:.10g}".format(ans)`

【样例 1 输入】

```
3 7 8 1
s 0 0.707106 0 0.707106 0 0.707106 0 -0.707106 0
s 1 0.707106 0 0.707106 0 0.707106 0 -0.707106 0
s 1 1 0 0 0 0 0 -1 0
s 2 0.707106 0 0.707106 0 0.707106 0 -0.707106 0
c 0 1 1 0 0 0 0 0 -1 0
s 0 1 0 0 0 0 0 -1 0
c 1 2 1 0 0 0 0 0 -1 0
0
1
2
3
4
5
6
7
```

【样例 1 输出】

```
0.3535522188 0
-0.3535522188 0
-0.3535522188 0
-0.3535522188 0
```

```
0.3535522188 0
-0.3535522188 0
0.3535522188 0
0.3535522188 0
```

【样例 1 解释】

此电路为题面中图 1 (1) 所示的量子电路。

【样例 2 输入】

```
4 9 16 5
s 0 0.707106 0 0.707106 0 0.707106 0 -0.707106 0
s 1 0.707106 0 0.707106 0 0.707106 0 -0.707106 0
s 2 0.707106 0 0.707106 0 0.707106 0 -0.707106 0
s 3 0.707106 0 0.707106 0 0.707106 0 -0.707106 0
c 0 1 1 0 0 0 0 0 -1 0
c 3 2 1 0 0 0 0 0 -1 0
c 1 2 1 0 0 0 0 0 -1 0
c 0 1 0 0 1 0 1 0 0 0
c 2 3 1 0 0 0 0 0 -1 0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

【样例 2 输出】

```
0.2499988952 0
-0.2499988952 0
0.2499988952 0
0.2499988952 0
0.2499988952 0
0.2499988952 0
-0.2499988952 0
0.2499988952 0
0.2499988952 0
-0.2499988952 0
0.2499988952 0
0.2499988952 0
0.2499988952 0
0.2499988952 0
-0.2499988952 0
0.2499988952 0
```

【样例 2 解释】

此电路为题面中图 2 (2) 所示的量子电路。

【样例 3】

见题目目录下的 *3.in* 与 *3.ans*。

【样例 3 解释】

一个属于子任务 1 的量子电路。

【样例 4】

见题目目录下的 *4.in* 与 *4.ans*。

【样例 4 解释】

一个属于子任务 5 的量子电路。

【子任务】

所有数据保证 $0 < q \le 10^3$。

<table>
<tr><th>子任务</th><th>分值</th><th>n</th><th>g</th><th>特殊性质</th></tr>
<tr><td>1</td><td>30</td><td rowspan="4">≤ 20</td><td>$\le 10^2$</td><td>无</td></tr>
<tr><td>2</td><td>5</td><td rowspan="3">$\le 10^4$</td><td>$v \le 10^8$</td></tr>
<tr><td>3</td><td>10</td><td>门重排后 $v \le 10^8$</td></tr>
<tr><td>4</td><td>12</td><td>门重排且比特重编号后 $v \le 10^8$</td></tr>
<tr><td>5</td><td>20</td><td>≤ 32</td><td>≤ 500</td><td>n 为偶数，$g_c \le 3$</td></tr>
<tr><td>6</td><td>3</td><td rowspan="3">≤ 34</td><td rowspan="3">$\le 10^4$</td><td>n 为偶数，$g_c \le 3$，$v \le 10^8$</td></tr>
<tr><td>7</td><td>8</td><td>n 为偶数，$g_c \le 3$，门重排后 $v \le 10^8$</td></tr>
<tr><td>8</td><td>12</td><td>n 为偶数，$g_c \le 3$，门重排且比特重编号后 $v \le 10^8$</td></tr>
</table>

其中 g_c 表示子任务 5、6、7、8 中横跨上下两部分电路的门的数量，v 表示完成模拟需要更新的位置的数量。

缓存管理器（cache）

【题目背景】

小 C、小 S 和小 P 有许多有趣的故事书，放了整整一大架子。好奇的小 P 和他的朋友们经常想听各种不同的故事，但是每次从书架上找实在是太麻烦了！他们听说计算机中有一种叫“缓存”的思想，可以帮助他更快地找到想要的书。请你帮小 P 和朋友们实现这样的缓存系统。

【题目描述】

在计算机系统中，高速缓存（Cache）层通常位于慢速的数据存储层和高速的数据读取层之间。其工作方式是将访问频率高的数据暂存在高速缓存层中，数据读取层如果访问高速缓存层中的数据可以立即返回，减少数据访问的延迟，协调两层间的数据传输速度差异。例如在 CPU 和内存（DRAM）之间会有 L1 ~ L3 多级缓存，它们的速度均远高于内存。当 CPU 多次需要同一份数据时，这些缓存能暂存这份数据，并在后续 CPU 请求时直接返回，避免 CPU 因访问较慢的内存而等待。类似这样的系统还有磁盘缓存，数据库缓存等。一个良好的缓存系统能够有效地提高整个系统的性能和处理能力。

缓存的工作原则是访问局部性原理，它包括：

- 时间局部性：某时刻被访问的数据，在最近一段时间很可能会被再次访问。
- 空间局部性：访问过某数据后，下一步很可能就需要访问它附近的数据。

缓存系统的关键是缓存管理策略的设计，它决定了缓存系统是否能较好的利用局部性进而有更多的缓存命中。

小 P 和朋友们为了更快地找到某本故事书，计划为书架设计一个缓存系统。为了简单起见，我们把它抽象成基于客户端/服务器 (C/S) 架构的网络文件系统缓存，客户端发出的所有文件查询请求都会先经过服务器的缓存系统，希望你帮忙设计出该场景下最佳的缓存管理策略。

【任务假设】

缓存系统由缓冲区和缓存管理器组成。客户端发来请求时，如果缓冲区中已经暂存了该次请求的文件，缓存系统会直接返回该文件，否则需要在文件系统上查找该文件。本次请求是否命中的信息会发送给缓存管理器，使之根据缓存管理策略管理缓冲区。

- 该缓存系统需要接收从 n $(1 \le n \le 10)$ 个不同客户端发送的，总共 m $(1 \le m \le 6 \times 10^5)$ 个文件查询请求（每个客户端仅访问属于自己的文件，它们访问的文件集合保证不相交）。
- 为简化设计，缓存系统是单线程运行的，所有请求会按照它们的查询时间，**顺序**发送给缓存系统。
- 每个请求有 f_{ID} 和 f_{size} 两个属性：

  - f_{ID}：文件的唯一标识符，为 9 位数字，前 2 位为客户端 ID (`00`~`99`)，后 7 位为随机生成的数字。**注意: 本题中 f_{ID} 相近与文件位置无关，因此无法根据 f_{ID} 利用空间局部性**
  - f_{size}：文件的大小，($1 \le f_{\text{size}} \le 1000$)。
  - 保证在不同的请求中，如果 f_{ID} 相同，那么 f_{size} 也相同。
- 缓冲区初始为空，缓冲区中能够存放文件的总大小至多为 k ($10^3 \le k \le 10^6$)。
- 缓冲区是否包含所需文件决定此次请求是否命中：
  - 命中：总命中数加 1，此次请求的额外花费 $= 0$。
  - 未命中：总命中数不变，此次请求的额外花费 $=$ 查找文件所需的常数时间花费 C 与文件传输的花费 f_{size} 之和。

【解题框架】

本题的评测使用**基于标准输入输出的交互方式**，你需要实现缓存管理器来与评测程序交互。

在每一个测试点，评测程序会读取文件请求队列，并在内部维护缓冲区，它会将每次请求的参数和请求是否命中的信息通过**标准输入**传递给缓存管理器，并从管理器的标准输出获得相应的操作。

本地测试方式

在本题附件中提供了使用随机替换策略（Random Replacement）的缓存管理器程序 `rrcache` 作为示例。

在不同系统上，不同语言对于输入数据 `1.in` 的测试方式如下（C/C++ 需要预先编译）：

Windows:

- C/C++: `python run.py "judge.exe -f 1.in" "rrcache.exe"`
- Python3: `python run.py "judge.exe -f 1.in" "python rrcache.py"`
- Java: `python run.py "judge.exe -f 1.in" "java rrcache.java"`

Linux:

- C/C++: `bash ./run.sh "./rrcache" 1.in`
- Python: `bash ./run.sh "python3 rrcache.py" 1.in`
- Java: `bash ./run.sh "java rrcache.java" 1.in`

如果你在 Linux 下无法运行，请先执行 `chmod +x ./run.sh ./judge`。

【输入与输出格式】

输入的第一行包含由空格分隔的 4 个正整数 n, m, k, C。

之后输入的 m 行为客户端请求

- 第 i 行输入为由空格分隔的整数 $f_{\text{ID}}, f_{\text{size}}, h$，其中前两项是请求 i 的属性，h 指示本次请求是否命中（`1` 表示命中，`0` 表示未命中）。
- 缓存管理器在接收每行输入后，**必须立刻输出**需要淘汰的文件集合（否则将会阻塞而无法获得下一行输入）。具体格式为：输出需淘汰文件集合的大小 x。如果 $x > 0$，接着在下一行输出这 x 个文件的 f_{ID}。两个 f_{ID} 之间用空格隔开，行尾**需要有回车**。
- 当评测程序接收到需淘汰文件集合后，会将集合中的文件移出缓冲区。移出文件后，若该次请求未命中且缓冲区剩余空间 $\geq f_{\text{size}}$，那么本次请求的文件会**自动加入**到缓冲区。

在每次输出后，请务必刷新输出缓冲区。各种语言中的做法如下：

- C/C++: `fflush(stdout);`（需要引入 `cstdio` 或 `stdio.h` 头文件）
- Java: `System.out.flush();`
- Python: `sys.stdout.flush()`

注意：附件中的 `*.in` 是评测程序（judge）的输入文件示例，它和上述格式的唯一区别是它的输入部分不包含 h。评测程序读入输入文件后会在内部维护一个缓冲区以及相关状态，并按上述格式打印到你所编写的缓存管理器程序的标准输入。

【样例 1 输入】

```
2 12 2000 50
010256601 308 0
010256601 308 1
000366065 540 0
000336814 798 0
010256601 308 1
010256601 308 1
000526529 496 0
000366065 540 0
000326705 615 0
000526529 496 0
010846557 650 0
010087439 615 0
```

【样例 1 输出】

```
0
0
```

```
3  0
4  0
5  0
6  0
7  1
8  000366065
9  1
10 000526529
11 1
12 010256601
13 1
14 000336814
15 1
16 000526529
17 1
18 000366065
```

【样例 1 解释】

本样例为评测程序与一个缓存管理器程序的交互，它包含 12 个文件请求。

- 因为缓冲区初始为空，请求 1 的文件 `010256601` 未命中，缓存管理器程序输出 0 表示淘汰 0 个文件，文件 `010256601` 自动加入到缓冲区，缓冲区剩余空间为 $2000-308=1692$，本次请求花费为 $308+50=358$。
- 请求 2 再次请求文件 `010256601` 命中，缓存管理器程序输出 0 表示淘汰 0 个文件，本次请求花费为 0。
- 请求 3 和请求 4 均未命中，请求文件都自动加入到缓冲区，缓冲区剩余空间为 $1692-540-798=354$，花费为 $540+798+50\times 2=1438$。
- 请求 5 和请求 6 命中，花费为 0。
- 请求 7 未命中，缓存管理器程序淘汰文件 `000366065`，缓冲区剩余空间为 $354+540=894$，文件 `000526529` 自动加入到缓冲区，缓冲区剩余空间为 $894-496=398$，本次请求花费为 $496+50=546$。
- 请求 8~12 均未命中，花费为 $540+615+496+650+615+50\times 5=3166$。

最终总命中率为 $3/12=25\%$，总花费为 $358+1438+546+3166=5508$。

需要注意，上述缓存管理器的输出**只是示例，不一定是最优结果**。

【样例 2 输入】

```
2 16 3000 15
010763438 857 0
010303496 533 0
010510738 533 0
000141415 650 0
010062839 678 0
000561212 85 0
000032305 1000 0
000032305 1000 1
000141415 650 1
000141415 650 1
000561212 85 1
000141415 650 1
010695377 533 0
010510738 533 1
010303496 533 0
010204168 194 0
```

【样例 2 输出】

```
0
0
0
0
1
010763438
0
1
010062839
0
0
0
0
0
1
```

```
010303496
0
1
010695377
0
```

【样例 2 解释】

本样例中请求 1~7、13、15、16 未命中，其余请求均命中。总命中率为 $6/16 = 37.5\%$，总花费为 $857 + 533 + 533 + 650 + 678 + 85 + 1000 + 533 + 533 + 194 + 15 \times 10 = 5746$。

【子任务】

本题的输入数据修改自公开的缓存系统轨迹（trace）数据集，测试点有下列三类情况：

- 原始轨迹：客户端请求队列为来自于原始轨迹，每个客户端以相同速度均匀发送请求。
- 缓存污染：每个客户端的请求轨迹在随机位置插入冷数据请求序列，该序列中访问的文件之后不会再被访问，保证所有插入的序列长度的总和小于总请求数量的 10%。
- 时间相关：每个客户端只在特定的活跃时间区间均匀发送请求，在输入数据中表现为在总请求队列中不同区间客户端请求分布是不同的。

<table>
<tr><th>测试点</th><th>n</th><th>测试点描述</th><th>每个测试点分值</th><th>测试点时限 (C/C++)(s)</th></tr>
<tr><td>1,2</td><td>1</td><td rowspan="2">原始轨迹</td><td>5</td><td>2.0</td></tr>
<tr><td>3,4</td><td rowspan="2">2</td><td>6</td><td rowspan="2">3.0</td></tr>
<tr><td>5,6</td><td rowspan="2">缓存污染</td><td>8</td></tr>
<tr><td>7</td><td rowspan="2">5</td><td rowspan="2">10</td><td rowspan="2">3.5</td></tr>
<tr><td>8</td><td>时间相关</td></tr>
<tr><td>9,10,11</td><td>10</td><td>时间相关 + 缓存污染</td><td>14</td><td>5.0</td></tr>
</table>

请注意各个测试点的**最长耗时限制不同**。

【评分方式】

缓存管理器的性能由两方面进行评价：总命中率 h 和总花费 c。设第 i 个测试点的分值为 s_i，在本测试点中使用随机替换策略的缓存管理器（C/C++ 版本，seed = 1024）的命中率和花费为 h_{rr} 和 c_{rr}，$h_{\max}$ 为所有选手本测试点的 h 的最大值，$c_{\min}$ 为所有选

手本测试点的 c 的最小值，该测试点的得分为：

$$s_i \times \left(0.5 \times \begin{cases} \dfrac{h - h_{rr}}{h_{\max} - h_{rr}} & h > h_{rr} \\ 0 & \text{otherwise} \end{cases} + 0.5 \times \begin{cases} \dfrac{c_{rr} - c}{c_{rr} - c_{\min}} & c < c_{rr} \\ 0 & \text{otherwise} \end{cases}\right)$$

【参考文献】

在下发的文件中，提供了如下的论文和资料供参考：

1. Cache Replacement Policies
2. The LRU-K page replacement algorithm for database disk buffering
3. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance
4. Evaluating content management techniques for web proxy caches
5. LHD: Improving Cache Hit Rate by Maximizing Hit Density
6. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)
7. ARC: A Self-Tuning, Low Overhead Replacement Cache
8. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches
9. LRFU (least recently/frequently used) replacement policy: A spectrum of block replacement policies
10. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network

可容错键值存储（kv）

【题目背景】

小 C、小 S 和小 P 慢慢长大了，想把他们的故事书送给有需要的人。爱动手的小 P 打算自己设计一个网站，来追踪每本书的情况。因此，他找到你帮他编写一个简单的数据库，用于支撑网站的运行。就像上一题说的，小 C、小 S 和小 P 的故事书有很多很多，所以数据库的操作速度需要尽量快；他们当然也不想丢失任何记录，所以数据库需要能在崩溃后不丢失任何数据。

【题目描述】

在本题中，你需要实现一个可容错的键值存储数据库（fault-tolerant key-value store）。键值存储数据库实现了类似哈希表的功能，通过键 Key 可以查询得到值 Value，还需要支持插入（或覆盖）、删除和清空操作。可容错指的是，即当数据库服务端进程退出并且重新启动以后，可以从持久存储（磁盘文件）中恢复状态并继续提供服务。

输入输出格式

你需要实现的是键值存储数据库的服务端，服务端从**标准输入**中读取请求。每个请求长度相同，都是 77 个字节，按顺序格式如下：

1. 8 字节的十六进制序列号（Seq）
2. 1 个空格（`' '`，ASCII 码为 `0x20`）
3. 1 字节的操作（Op），可能的取值：`'S'`、`'G'`、`'D'`、`'C'`，含义见下
4. 1 个空格（`' '`，ASCII 码为 `0x20`）
5. 32 字节的键（Key），均为可打印 ASCII 字符
6. 1 个空格（`' '`，ASCII 码为 `0x20`）
7. 32 字节的值（Value），均为可打印 ASCII 字符
8. 1 个回车（`'\n'`，ASCII 码为 `0x0a`）

对于每个请求，服务端都应当向 **标准输出** 写入响应，**并（定期）刷新缓冲区**，具体做法请参考第四题的题面。每个响应的长度一样，都为 44 字节，按顺序格式如下：

1. 8 字节的十六进制序列号（Seq）
2. 1 个空格（`' '`，ASCII 码为 `0x20`）
3. 1 字节的返回值（Res），可能的取值：`'0'`、`'1'`
4. 1 个空格（`' '`，ASCII 码为 `0x20`）
5. 32 字节的值（Value）
6. 1 个回车（`'\n'`，ASCII 码为 `0x0a`）

为了便于优化，在启动你的服务端程序时，会传入一个命令行参数表示当前测例是否可能强制退出服务端（并恢复运行）。`1` 表示会强制退出服务，`0` 表示不会出现此情况。你可以从 C/C++ 语言的 `main` 函数的 `char *argv[]` 参数、Python 语言的 `sys.argv` 变量、Java 语言的 `main` 函数的 `String[] args` 参数中获取这个值。

具体要求

需要支持的操作和处理方法如下表：

请求 Op	操作	条件	响应 Res	响应 Value
`'S'`	插入或覆盖	Key 存在	`'1'`	被覆盖的 Value
		Key 不存在	`'0'`	N/A
`'G'`	查询	Key 存在	`'1'`	查询到的 Value
		Key 不存在	`'0'`	N/A
`'D'`	删除键值对	Key 存在	`'1'`	被删除的 Value
		Key 不存在	`'0'`	N/A
`'C'`	清空数据库	N/A	`'1'`	

客户端保证 Key 和 Value 都由空格符以外的可打印字符组成，请求的 Seq 会从 0 开始从小到大递增，并按照 Seq 大小顺序发送给服务端。服务端需要使用相同的 Seq 来标识对于某个请求的响应，并且保证 Value 由空格符号以外的可打印字符组成。上表中响应 Value 取值 N/A 表示服务端可以传输任意合法 Value，客户端不会检查其内容。

客户端会维护一个大小为 16 的滑动窗口，以尽量快的速度发送请求，同时保证发送的最后请求的 Seq 减去目前尚未得到响应的最早请求的 Seq 的差值小于窗口大小。服务端可以乱序返回对窗口中的请求的响应，但要保证返回的结果**与顺序处理是相同的**。更严格地说，服务端对请求的处理应该满足**线性一致性**（linearizability），并且线性化后的顺序与顺序处理相同。

客户端在请求中途可能会将服务端进程强制退出，再重新启动服务端（保证服务端被强制退出时，没有任何请求在等待服务器处理）。因此，服务器每次响应请求前后必须将自己的状态持久化。当客户端重启服务端时，需要从文件系统中恢复强制退出之前的状态，并继续处理后续的请求。

为了实现可容错，服务端通常需要在目录下保存若干文件来持久化当前的状态。本题中要求，服务端创建的所有文件/目录都需要有 `kv-` 的前缀，比如 `kv-data.bin`、`kv-persist.json` 等等。服务端所有创建的文件大小总和**不能超过 32MB**。每个数据点评测前，都会清空目录下的所有文件。

本地测试

下发的文件中提供了一个客户端的实现 `client`。此程序会从文件中读取请求序列，并按照顺序发送请求给服务端。客户端的输入输出格式见下面的描述。首先，你需要完成服务器的实现，而后如下运行客户端：

- C/C++: `./client 1.in 1.ans ./kv`
- Python: `./client 1.in 1.ans python3 kv.py`
- Java: `./client 1.in 1.ans java kv.java`

向客户端传递的第一个参数是输入文件，第二个参数是答案文件，从第三个参数开始是运行你的服务器需要执行的命令。客户端会按照输入文件向服务端发送请求，并把

结果和答案文件比较，最后输出结果和经过的时间。客户端在运行时，会按照上面所述向你的服务端传入一个命令行参数，表示本次测试中是否会包含强制退出与恢复。

如果你在 Linux 下无法运行，请执行 `chmod +x ./client`。

由于评测系统限制，MLE（Memory Limit Exceeded，超出内存限制）错误在评测系统上会显示为 WA（Wrong Answer，答案错误），你可以在评测信息的详情中看到程序最大的内存占用，如果内存占用过大，可能就是内存使用过多导致内存分配失败。

【输入格式】

下发的客户端使用的输入文件由 $N+1$ 行组成，第一行是一个数字，`1` 表示测例有强制退出服务端的操作，`0` 则表示没有。从第二行开始的 N 行，每行由空格分隔为若干段，第一段仅有一个字母，表示需要进行的操作，之后的若干段表示的是操作的参数。字母 `C` 表示 CLR 清空操作，清空数据库；字母 `S` 表示 SET 插入操作，参数分别是 Key 和 Value；字母 `G` 表示 GET 查询操作，参数是 Key；字母 `D` 表示 DEL 删除操作，参数是 Key；字母 `K` 表示强制退出服务端后，再重启服务端。

【输出格式】

客户端使用的答案文件由若干行组成，每一行对应一个输入中**除了强制退出以外**的操作的响应。每一行由空格分隔的两个数字和一个可选的字符串组成，分别表示 Res、是否存在 Value 和对应的 Value。

【样例 1 输入】

```
0
C
S abcd dcba
G abcd
G abc
D abcd
G abcd
```

【样例 1 输出】

```
1 0
0 0
1 1 dcba
0 0
```

```
1 1 dcba
0 0
```

【样例 1 解释】

客户端在读入这一组样例的时候，会发送如下的请求给服务端：

```
00000000 C $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
00000001 S abcd$$$$$$$$$$$$$$$$$$$$$$$$$$$$ dcba$$$$$$$$$$$$$$$$$$$$$$$$$$$$
00000002 G abcd$$$$$$$$$$$$$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
00000003 G abc$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
00000004 D abcd$$$$$$$$$$$$$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
00000005 G abcd$$$$$$$$$$$$$$$$$$$$$$$$$$$$ $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

服务端应该发送形如以下的响应给客户端。其中响应可以不按照序列号从小到大的顺序，同时上表中 N/A 对应的响应 Value 的内容可以不同，即下面的第 1、2、4、6 行：

```
00000000 1 $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
00000001 0 $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
00000002 1 dcba$$$$$$$$$$$$$$$$$$$$$$$$$$$$
00000003 0 $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
00000004 1 dcba$$$$$$$$$$$$$$$$$$$$$$$$$$$$
00000005 0 $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

【样例 2 输入】

```
1
C
S abcd dcba
G abcd
K
```

```
G abc
G abcd
S abcd efgh
G abcd
D abcd
G abcd
```

【样例 2 输出】

```
1 0
0 0
1 1 dcba
0 0
1 1 dcba
1 1 dcba
1 1 efgh
1 1 efgh
0 0
```

【子任务】

<table>
<tr><th>测试点</th><th>每个测试点分值</th><th>N</th><th>Get 操作比例</th><th>强制退出服务端</th></tr>
<tr><td>1, 2, 3, 4</td><td>2.5</td><td>$\le 10^3$</td><td>80 %</td><td rowspan="2">无</td></tr>
<tr><td>5, 6, 7, 8</td><td>5.0</td><td>$\le 10^5$</td><td>50 %</td></tr>
<tr><td>9, 10, 11, 12</td><td>3.75</td><td>$\le 10^4$</td><td rowspan="2">80 %</td><td rowspan="3">有</td></tr>
<tr><td>13, 14, 15, 16</td><td>6.25</td><td rowspan="2">$\le 10^5$</td></tr>
<tr><td>17, 18, 19, 20</td><td>7.5</td><td>50 %</td></tr>
</table>

表中 Get 操作比例指的是 Get 操作占所有操作的比例。子任务中各个测试点均分该子任务的分值。

【评分方式】

设第 i 个测试点的分值为 s_i，当答案正确时，记 t 为运行时间，$t_{\min}$ 为所有选手中答案正确情况下本测试点运行时间 t 的最小值，那么该测试点的得分为：

$$s_i \times (0.1 + 0.9 \times \frac{t_{\min}}{t})$$

当答案错误时，该测试点零分。

【提示】

本题使用并行（多线程或多进程）实现可以有效提高性能，但必须注意数据结构的并发安全性，以及需要满足题目的返回顺序要求。在 C++ 语言中，你可以用 `pthread` 库（包含 `pthread.h` 头文件）或者 `std::thread` 类（包含 `thread` 头文件）实现多线程；在 Python 语言中，可以用标准库中 `multiprocessing` 模块实现**多进程**（注意 Python 的多线程可能无法带来任何收益）；在 Java 语言中，可以用自带的 `Thread` 类实现多线程。

评测环境提供了 4 个 CPU 核心，请不要创建过多的进程/线程，以免带来额外开销。

【参考资料】

在下发的文件中，提供了如下的论文和资料供参考：

1. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing
2. Cuckoo hashing
3. Enhancing Lookup Performance of Key-Value Stores using Cuckoo Hashing
4. The Log-Structured Merge-Tree (LSM-Tree)
5. The Ubiquitous B-Tree